A Comprehensive Guide to Machine Learning

Soroush Nasiriany, Garrett Thomas, William Wang, Alex Yang
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

June 24, 2019

About

CS 189 is the Machine Learning course at UC Berkeley. In this guide we have created a comprehensive course guide in order to share our knowledge with students and the general public, and hopefully draw the interest of students from other universities to Berkeley's Machine Learning curriculum.

This guide was started by CS 189 TAs Soroush Nasiriany and Garrett Thomas in Fall 2017, with the assistance of William Wang and Alex Yang.

We owe gratitude to Professors Anant Sahai, Stella Yu, and Jennifer Listgarten, as this book is heavily inspired from their lectures. In addition, we are indebted to Professor Jonathan Shewchuk for his machine learning notes, from which we drew inspiration.

The latest version of this document can be found either at `http://www.eecs189.org/` or `http://snasiriany.me/cs189/`. Please report any mistakes to the staff, and contact the authors if you wish to redistribute this document.

Notation

Notation	Meaning
$\mathbb{R}$	set of real numbers
$\mathbb{R}^n$	set (vector space) of n-tuples of real numbers, endowed with the usual inner product
$\mathbb{R}^{m \times n}$	set (vector space) of m-by-n matrices
δ_{ij}	Kronecker delta, i.e. $\delta_{ij} = 1$ if $i = j$, 0 otherwise
$\nabla f(\mathbf{x})$	gradient of the function f at $\mathbf{x}$
$\nabla^2 f(\mathbf{x})$	Hessian of the function f at $\mathbf{x}$
$p(X)$	distribution of random variable X
$p(x)$	probability density/mass function evaluated at x
$\mathbb{E}[X]$	expected value of random variable X
$\text{Var}(X)$	variance of random variable X
$\text{Cov}(X, Y)$	covariance of random variables X and Y

Other notes:

- Vectors and matrices are in bold (e.g. $\mathbf{x}, \mathbf{A}$). This is true for vectors in $\mathbb{R}^n$ as well as for vectors in general vector spaces. We generally use Greek letters for scalars and capital Roman letters for matrices and random variables.
- We assume that vectors are column vectors, i.e. that a vector in $\mathbb{R}^n$ can be interpreted as an n-by-1 matrix. As such, taking the transpose of a vector is well-defined (and produces a row vector, which is a 1-by-n matrix).

Contents

1 Regression I **5**
1.1 Ordinary Least Squares . . . 5
1.2 Ridge Regression . . . 8
1.3 Feature Engineering . . . 11
1.4 Hyperparameters and Validation . . . 12

2 Regression II **17**
2.1 MLE and MAP for Regression (Part I) . . . 17
2.2 Bias-Variance Tradeoff . . . 23
2.3 Multivariate Gaussians . . . 30
2.4 MLE and MAP for Regression (Part II) . . . 37
2.5 Kernels and Ridge Regression . . . 44
2.6 Sparse Least Squares . . . 50
2.7 Total Least Squares . . . 57

3 Dimensionality Reduction **63**
3.1 Principal Component Analysis . . . 63
3.2 Canonical Correlation Analysis . . . 70

4 Beyond Least Squares: Optimization and Neural Networks **79**
4.1 Nonlinear Least Squares . . . 79
4.2 Optimization . . . 81
4.3 Gradient Descent . . . 82
4.4 Line Search . . . 88
4.5 Convex Optimization . . . 89
4.6 Newton's Method . . . 93
4.7 Gauss-Newton Algorithm . . . 96
4.8 Neural Networks . . . 97
4.9 Training Neural Networks . . . 103

5 Classification **107**

5.1 Generative vs. Discriminative Classification . . . 107

5.2 Least Squares Support Vector Machine . . . 109

5.3 Logistic Regression . . . 113

5.4 Gaussian Discriminant Analysis . . . 121

5.5 Support Vector Machines . . . 127

5.6 Duality . . . 134

5.7 Nearest Neighbor Classification . . . 145

6 Clustering **151**

6.1 K-means Clustering . . . 152

6.2 Mixture of Gaussians . . . 155

6.3 Expectation Maximization (EM) Algorithm . . . 156

7 Decision Tree Learning **163**

7.1 Decision Trees . . . 163

7.2 Random Forests . . . 168

7.3 Boosting . . . 169

8 Deep Learning **175**

8.1 Convolutional Neural Networks . . . 175

8.2 CNN Architectures . . . 182

8.3 Visualizing and Understanding CNNs . . . 185

Chapter 1

Regression I

Our goal in machine learning is to extract a *relationship* from data. In **regression** tasks, this relationship takes the form of a function $y = f(\mathbf{x})$, where $y \in \mathbb{R}$ is some quantity that can be predicted from an input $\mathbf{x} \in \mathbb{R}^d$, which should for the time being be thought of as some collection of numerical measurements. The true relationship f is unknown to us, and our aim is to recover it as well as we can from data. Our end product is a function $\hat{y} = h(\mathbf{x})$, called the **hypothesis**, that should approximate f. We assume that we have access to a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where each pair $(\mathbf{x}_i, y_i)$ is an example (possibly noisy or otherwise approximate) of the input-output mapping to be learned. Since learning arbitrary functions is intractable, we restrict ourselves to some **hypothesis class** $\mathcal{H}$ of allowable functions. More specifically, we typically employ a **parametric model**, meaning that there is some finite-dimensional vector $\mathbf{w} \in \mathbb{R}^d$, the elements of which are known as **parameters** or **weights**, that controls the behavior of the function. That is,

$$h_{\mathbf{w}}(\mathbf{x}) = g(\mathbf{x}, \mathbf{w})$$

for some other function g. The hypothesis class is then the set of all functions induced by the possible choices of the parameters $\mathbf{w}$:

$$\mathcal{H} = \{h_{\mathbf{w}} \mid \mathbf{w} \in \mathbb{R}^d\}$$

After designating a **cost function** L, which measures how poorly the predictions $\hat{y}$ of the hypothesis match the true output y, we can proceed to search for the parameters that best fit the data by minimizing this function:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} L(\mathbf{w})$$

1.1 Ordinary Least Squares

Ordinary least squares (OLS) is one of the simplest regression problems, but it is well-understood and practically useful. It is a **linear regression** problem, which means that we take $h_{\mathbf{w}}$ to be of the form $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}$. We want

$$y_i \approx \hat{y}_i = h_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{x}_i^\top \mathbf{w}$$

for each $i = 1, \dots, n$. This set of equations can be written in matrix form as

$$\underbrace{\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}}_{\mathbf{y}} \approx \underbrace{\begin{bmatrix} \mathbf{x}_1{}^\top \\ \vdots \\ \mathbf{x}_n{}^\top \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}}_{\mathbf{w}}$$

In words, the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ has the input datapoint $\mathbf{x}_i$ as its ith row. This matrix is sometimes called the **design matrix**. Usually $n \geq d$, meaning that there are more datapoints than measurements.

There will in general be no exact solution to the equation $\mathbf{y} = \mathbf{Xw}$ (even if the data were perfect, consider how many equations and variables there are), but we can find an approximate solution by minimizing the sum (or equivalently, the mean) of the squared errors:

$$L(\mathbf{w}) = \sum_{i=1}^{n} (\mathbf{x}_i{}^\top \mathbf{w} - y_i)^2 = \min_{\mathbf{w}} \|\mathbf{Xw} - \mathbf{y}\|_2^2$$

Now that we have formulated an optimization problem, we want to go about solving it. We will see that the particular structure of OLS allows us to compute a closed-form expression for a globally optimal solution, which we denote $\mathbf{w}^*_{\text{OLS}}$.

Approach 1: Vector calculus

Calculus is the primary mathematical workhorse for studying the optimization of differentiable functions. Recall the following important result: if $L : \mathbb{R}^d \to \mathbb{R}$ is continuously differentiable, then any local optimum $\mathbf{w}^*$ satisfies $\nabla L(\mathbf{w}^*) = \mathbf{0}$. In the OLS case,

$$\begin{aligned} L(\mathbf{w}) &= \|\mathbf{Xw} - \mathbf{y}\|_2^2 \\ &= (\mathbf{Xw} - \mathbf{y})^\top (\mathbf{Xw} - \mathbf{y}) \\ &= (\mathbf{Xw})^\top \mathbf{Xw} - (\mathbf{Xw})^\top \mathbf{y} - \mathbf{y}^\top \mathbf{Xw} + \mathbf{y}^\top \mathbf{y} \\ &= \mathbf{w}^\top \mathbf{X}^\top \mathbf{Xw} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y} \end{aligned}$$

Using the following results from matrix calculus

$$\begin{aligned} \nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) &= \mathbf{a} \\ \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A} \mathbf{x}) &= (\mathbf{A} + \mathbf{A}^\top)\mathbf{x} \end{aligned}$$

the gradient of L is easily seen to be

$$\begin{aligned} \nabla L(\mathbf{w}) &= \nabla_{\mathbf{w}}(\mathbf{w}^\top \mathbf{X}^\top \mathbf{Xw} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}) \\ &= \nabla_{\mathbf{w}}(\mathbf{w}^\top \mathbf{X}^\top \mathbf{Xw}) - 2\nabla_{\mathbf{w}}(\mathbf{w}^\top \mathbf{X}^\top \mathbf{y}) + \underbrace{\nabla_{\mathbf{w}}(\mathbf{y}^\top \mathbf{y})}_{\mathbf{0}} \\ &= 2\mathbf{X}^\top \mathbf{Xw} - 2\mathbf{X}^\top \mathbf{y} \end{aligned}$$

where in the last line we have used the symmetry of $\mathbf{X}^\top \mathbf{X}$ to simplify $\mathbf{X}^\top \mathbf{X} + (\mathbf{X}^\top \mathbf{X})^\top = 2\mathbf{X}^\top \mathbf{X}$. Setting the gradient to $\mathbf{0}$, we conclude that any optimum $\mathbf{w}^*_{\text{OLS}}$ satisfies

$$\mathbf{X}^\top \mathbf{X} \mathbf{w}^*_{\text{OLS}} = \mathbf{X}^\top \mathbf{y}$$

If $\mathbf{X}$ is full rank, then $\mathbf{X}^\top\mathbf{X}$ is as well (assuming $n \geq d$), so we can solve for a unique solution

$$\mathbf{w}^*_{\text{OLS}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

Note: Although we write $(\mathbf{X}^\top\mathbf{X})^{-1}$, in practice one would not actually compute the inverse; it is more numerically stable to solve the linear system of equations above (e.g. with Gaussian elimination).

In this derivation we have used the condition $\nabla L(\mathbf{w}^*) = \mathbf{0}$, which is a *necessary* but not *sufficient* condition for optimality. We found a critical point, but in general such a point could be a local minimum, a local maximum, or a saddle point. Fortunately, in this case the objective function is **convex**, which implies that any critical point is indeed a global minimum. To show that L is convex, it suffices to compute the **Hessian** of L, which in this case is

$$\nabla^2 L(\mathbf{w}) = 2\mathbf{X}^\top\mathbf{X}$$

and show that this is positive semi-definite:

$$\forall \mathbf{w},\ \mathbf{w}^\top(2\mathbf{X}^\top\mathbf{X})\mathbf{w} = 2(\mathbf{X}\mathbf{w})^\top\mathbf{X}\mathbf{w} = 2\|\mathbf{X}\mathbf{w}\|_2^2 \geq 0$$

Approach 2: Orthogonal projection

There is also a linear algebraic way to arrive at the same solution: orthogonal projections.

Recall that if V is an inner product space and S a subspace of V, then any $\mathbf{v} \in V$ can be decomposed uniquely in the form

$$\mathbf{v} = \mathbf{v}_S + \mathbf{v}_\perp$$

where $\mathbf{v}_S \in S$ and $\mathbf{v}_\perp \in S^\perp$. Here $S^\perp$ is the orthogonal complement of S, i.e. the set of vectors that are perpendicular to every vector in S.

The **orthogonal projection** onto S, denoted P_S, is the linear operator that maps $\mathbf{v}$ to $\mathbf{v}_S$ in the decomposition above. An important property of the orthogonal projection is that

$$\|\mathbf{v} - P_S\mathbf{v}\| \leq \|\mathbf{v} - \mathbf{s}\|$$

for all $\mathbf{s} \in S$, with equality if and only if $\mathbf{s} = P_s\mathbf{v}$. That is,

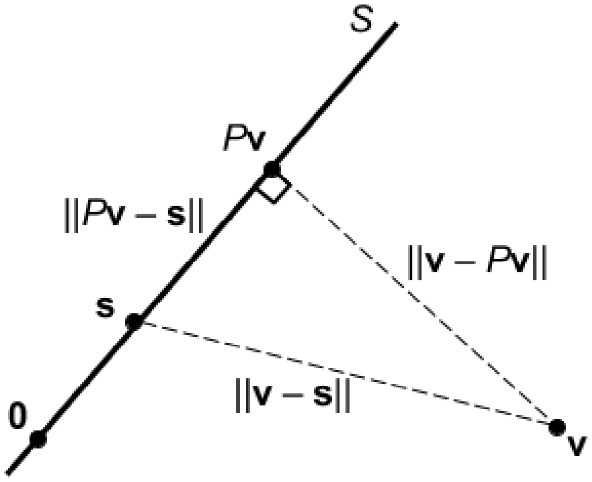
$$P_S\mathbf{v} = \operatorname*{arg\,min}_{\mathbf{s}\in S} \|\mathbf{v} - \mathbf{s}\|$$

Proof. By the Pythagorean theorem,

$$\|\mathbf{v} - \mathbf{s}\|^2 = \|\underbrace{\mathbf{v} - P_S\mathbf{v}}_{\in S^\perp} + \underbrace{P_S\mathbf{v} - \mathbf{s}}_{\in S}\|^2 = \|\mathbf{v} - P_S\mathbf{v}\|^2 + \|P_S\mathbf{v} - \mathbf{s}\|^2 \geq \|\mathbf{v} - P_S\mathbf{v}\|^2$$

with equality holding if and only if $\|P_S\mathbf{v} - \mathbf{s}\|^2 = 0$, i.e. $\mathbf{s} = P_S\mathbf{v}$. Taking square roots on both sides gives $\|\mathbf{v} - \mathbf{s}\| \geq \|\mathbf{v} - P_S\mathbf{v}\|$ as claimed (since norms are nonnegative). □

Here is a visual representation of the argument above:



In the OLS case,

$$\mathbf{w}^*_{\text{OLS}} = \arg\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

But observe that the set of vectors that can be written $\mathbf{X}\mathbf{w}$ for some $\mathbf{w} \in \mathbb{R}^d$ is precisely the range of $\mathbf{X}$, which we know to be a subspace of $\mathbb{R}^n$, so

$$\min_{\mathbf{z}\in\text{range}(\mathbf{X})} \|\mathbf{z} - \mathbf{y}\|_2^2 = \min_{\mathbf{w}\in\mathbb{R}^d} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

By pattern matching with the earlier optimality statement about P_S, we observe that $P_{\text{range}(\mathbf{X})}\mathbf{y} = \mathbf{X}\mathbf{w}^*_{\text{OLS}}$, where $\mathbf{w}^*_{\text{OLS}}$ is any optimum for the right-hand side. The projected point $\mathbf{X}\mathbf{w}^*_{\text{OLS}}$ is always unique, but if $\mathbf{X}$ is full rank (again assuming $n \geq d$), then the optimum $\mathbf{w}^*_{\text{OLS}}$ is also unique (as expected). This is because $\mathbf{X}$ being full rank means that the columns of $\mathbf{X}$ are linearly independent, in which case there is a one-to-one correspondence between $\mathbf{w}$ and $\mathbf{X}\mathbf{w}$.

To solve for $\mathbf{w}^*_{\text{OLS}}$, we need the following fact[1]:

$$\text{null}(\mathbf{X}^\top) = \text{range}(\mathbf{X})^\perp$$

Since we are projecting onto range($\mathbf{X}$), the orthogonality condition for optimality is that $\mathbf{y} - P\mathbf{y} \perp \text{range}(\mathbf{X})$, i.e. $\mathbf{y} - \mathbf{X}\mathbf{w}^*_{\text{OLS}} \in \text{null}(\mathbf{X}^\top)$. This leads to the equation

$$\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w}^*_{\text{OLS}}) = \mathbf{0}$$

which is equivalent to

$$\mathbf{X}^\top\mathbf{X}\mathbf{w}^*_{\text{OLS}} = \mathbf{X}^\top\mathbf{y}$$

as before.

1.2 Ridge Regression

While Ordinary Least Squares can be used for solving linear least squares problems, it falls short due to numerical instability and generalization issues. Numerical instability arises when the features of the data are close to collinear (leading to linearly dependent feature columns), causing the input

[1] This result is often stated as part of the Fundamental Theorem of Linear Algebra.

matrix $\mathbf{X}$ to lose its rank or have singular values that very close to 0. Why are small singular values bad? Let us illustrate this via the singular value decomposition (SVD) of $\mathbf{X}$:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$$

where $\mathbf{U} \in \mathbb{R}^{n\times n}, \mathbf{\Sigma} \in \mathbb{R}^{n\times d}, \mathbf{V} \in \mathbb{R}^{d\times d}$. In the context of OLS, we must have that $\mathbf{X}^\top\mathbf{X}$ is invertible, or equivalently, $\text{rank}(\mathbf{X}^\top\mathbf{X}) = \text{rank}(\mathbf{X}^\top) = \text{rank}(\mathbf{X}) = d$. Assuming that $\mathbf{X}$ and $\mathbf{X}^\top$ are full column rank d, we can express the SVD of $\mathbf{X}$ as

$$\mathbf{X} = \mathbf{U}\begin{bmatrix}\mathbf{\Sigma}_d \\ \mathbf{0}\end{bmatrix}\mathbf{V}^\top$$

where $\mathbf{\Sigma}_d \in \mathbb{R}^{d\times d}$ is a diagonal matrix with strictly positive entries. Now let's try to expand the $(\mathbf{X}^\top\mathbf{X})^{-1}$ term in OLS using the SVD of $\mathbf{X}$:

$$\begin{aligned}(\mathbf{X}^\top\mathbf{X})^{-1} &= (\mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_d & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\mathbf{U}\begin{bmatrix}\mathbf{\Sigma}_d \\ \mathbf{0}\end{bmatrix}\mathbf{V}^\top)^{-1} \\ &= (\mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_d & \mathbf{0}\end{bmatrix}\mathbf{I}\begin{bmatrix}\mathbf{\Sigma}_d \\ \mathbf{0}\end{bmatrix}\mathbf{V}^\top)^{-1} \\ &= (\mathbf{V}\mathbf{\Sigma}_d^2\mathbf{V}^\top)^{-1} = (\mathbf{V}^\top)^{-1}(\mathbf{\Sigma}_d^2)^{-1}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Sigma}_d^{-2}\mathbf{V}^\top\end{aligned}$$

This means that $(\mathbf{X}^\top\mathbf{X})^{-1}$ will have singular values that are the squared inverse of the singular values of $\mathbf{X}$, potentially leading to extremely large singular values when the singular value of $\mathbf{X}$ are close to 0. Such excessively large singular values can be very problematic for numerical stability purposes. In addition, abnormally high values to the optimal $\mathbf{w}$ solution would prevent OLS from generalizing to unseen data.

There is a very simple solution to these issues: penalize the entries of $\mathbf{w}$ from becoming too large. We can do this by adding a penalty term constraining the norm of $\mathbf{w}$. For a fixed, small scalar $\lambda > 0$, we now have:

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$$

Note that the λ in our objective function is a **hyperparameter** that measures the sensitivity to the values in $\mathbf{w}$. Just like the degree in polynomial features, λ is a value that we must choose arbitrarily through validation. Let's expand the terms of the objective function:

$$\begin{aligned}L(\mathbf{w}) &= \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2 \\ &= \mathbf{w}^\top\mathbf{X}^\top\mathbf{X}\mathbf{w} - 2\mathbf{w}^\top\mathbf{X}^\top\mathbf{y} + \mathbf{y}^\top\mathbf{y} + \lambda\mathbf{w}^\top\mathbf{w}\end{aligned}$$

Finally take the gradient of the objective and find the value of $\mathbf{w}$ that achieves $\mathbf{0}$ for the gradient:

$$\begin{aligned}\nabla_{\mathbf{w}}L(\mathbf{w}) &= \mathbf{0} \\ 2\mathbf{X}^\top\mathbf{X}\mathbf{w} - 2\mathbf{X}^\top\mathbf{y} + 2\lambda\mathbf{w} &= \mathbf{0} \\ (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})\mathbf{w} &= \mathbf{X}^\top\mathbf{y} \\ \mathbf{w}^*_{\text{RIDGE}} &= (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}\end{aligned}$$

This value is guaranteed to achieve the (unique) global minimum, because the objective function is **strongly convex**. To show that f is strongly convex, it suffices to compute the Hessian of f, which in this case is

$$\nabla^2 L(\mathbf{w}) = 2\mathbf{X}^\top\mathbf{X} + 2\lambda\mathbf{I}$$

and show that this is **positive definite (PD)**:

$$\forall \mathbf{w} \neq \mathbf{0},\ \mathbf{w}^\top(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})\mathbf{w} = (\mathbf{X}\mathbf{w})^\top\mathbf{X}\mathbf{w} + \lambda\mathbf{w}^\top\mathbf{w} = \|\mathbf{X}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_2^2 > 0$$

Since the Hessian is positive definite, we can equivalently say that the eigenvalues of the Hessian are strictly positive and that the objective function is strongly convex. A useful property of strongly convex functions is that they have a unique optimum point, so the solution to ridge regression is unique. We cannot make such guarantees about ordinary least squares, because the corresponding Hessian could have eigenvalues that are 0. Let us explore the case in OLS when the Hessian has a 0 eigenvalue. In this context, the term $\mathbf{X}^\top\mathbf{X}$ is not invertible, but this does *not* imply that no solution exists! In OLS, there always exists a solution, and when the Hessian is PD that solution is unique; when the Hessian is PSD, there are infinitely many solutions. (There always exists a solution to the expression $\mathbf{X}^\top\mathbf{X}\mathbf{w} = \mathbf{X}^\top\mathbf{y}$, because the range of $\mathbf{X}^\top\mathbf{X}$ and the range space of $\mathbf{X}^\top$ are equivalent; since $\mathbf{X}^\top\mathbf{y}$ lies in the range of $\mathbf{X}^\top$, it must equivalently lie in the range of $\mathbf{X}^\top\mathbf{X}$ and therefore there always exists a $\mathbf{w}$ that satisfies the equation $\mathbf{X}^\top\mathbf{X}\mathbf{w} = \mathbf{X}^\top\mathbf{y}$.)

The technique we just described is known as **ridge regression**. Note that now the expression $\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}$ is invertible, regardless of rank of $\mathbf{X}$. Let's find $(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}$ through SVD:

$$\begin{aligned}
(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1} &= \left(\mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_r & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{U}^\top\mathbf{U}\begin{bmatrix}\mathbf{\Sigma}_r & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top + \lambda\mathbf{I}\right)^{-1}\\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_r^2 & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top + \lambda\mathbf{I}\right)^{-1}\\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_r^2 & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix}\mathbf{V}^\top + \mathbf{V}(\lambda\mathbf{I})\mathbf{V}^\top\right)^{-1}\\
&= \left(\mathbf{V}\left(\begin{bmatrix}\mathbf{\Sigma}_r^2 & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{bmatrix} + \lambda\mathbf{I}\right)\mathbf{V}^\top\right)^{-1}\\
&= \left(\mathbf{V}\begin{bmatrix}\mathbf{\Sigma}_r^2 + \lambda\mathbf{I} & \mathbf{0}\\ \mathbf{0} & \lambda\mathbf{I}\end{bmatrix}\mathbf{V}^\top\right)^{-1}\\
&= (\mathbf{V}^\top)^{-1}\begin{bmatrix}\mathbf{\Sigma}_r^2 + \lambda\mathbf{I} & \mathbf{0}\\ \mathbf{0} & \lambda\mathbf{I}\end{bmatrix}^{-1}\mathbf{V}^{-1}\\
&= \mathbf{V}\begin{bmatrix}(\mathbf{\Sigma}_r^2 + \lambda\mathbf{I})^{-1} & \mathbf{0}\\ \mathbf{0} & \frac{1}{\lambda}\mathbf{I}\end{bmatrix}\mathbf{V}^\top
\end{aligned}$$

Now with our slight tweak, the matrix $\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}$ has become full rank and thus invertible. The singular values have become $\frac{1}{\sigma^2+\lambda}$ and $\frac{1}{\lambda}$, meaning that the singular values are guaranteed to be at most $\frac{1}{\lambda}$, solving our numerical instability issues. Furthermore, we have partially solved the overfitting issue. By penalizing the norm of $\mathbf{x}$, we encourage the weights corresponding to relevant features that capture the main structure of the true model, and penalize the weights corresponding to complex features that only serve to fine tune the model and fit noise in the data.

1.3 Feature Engineering

We've seen that the least-squares optimization problem

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

represents the "best-fit" *linear* model, by projecting $\mathbf{y}$ onto the subspace spanned by the columns of $\mathbf{X}$. However, the true input-output relationship $y = f(\mathbf{x})$ may be nonlinear, so it is useful to consider nonlinear models as well. It turns out that we can still do this under the framework of linear least-squares, by augmenting the data with new **features**. In particular, we devise some function $\boldsymbol{\phi} : \mathbb{R}^\ell \to \mathbb{R}^d$, called a **feature map**, that maps each raw data point $\mathbf{x} \in \mathbb{R}^\ell$ into a vector of features $\boldsymbol{\phi}(\mathbf{x})$. The hypothesis function then writes

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=1}^{d} w_j \phi_j(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$$

Note that the resulting model is still linear with respect to the features, but it is nonlinear with respect to the original data if $\boldsymbol{\phi}$ is nonlinear. The component functions ϕ_j are sometimes called **basis functions** because our hypothesis is a linear combination of them. In the simplest case, we could just use the components of $\mathbf{x}$ as features (i.e. $\phi_j(\mathbf{x}) = x_j$), but in general it is helpful to disambiguate the features of an example from the example's entries.

We can then use least-squares to estimate the weights $\mathbf{w}$, just as before. To do this, we replace the original data matrix $\mathbf{X} \in \mathbb{R}^{n\times\ell}$ by $\boldsymbol{\Phi} \in \mathbb{R}^{n\times d}$, which has $\boldsymbol{\phi}(\mathbf{x}_i)^\top$ as its ith row:

$$\min_{\mathbf{w}} \|\boldsymbol{\Phi}\mathbf{w} - \mathbf{y}\|_2^2$$

Example: Fitting Ellipses

Let's use least-squares to estimate the parameters of an ellipse from data.

Assume that we have n data points $\mathcal{D} = \{(x_{1,i}, x_{2,i})\}_{i=1}^n$, which may be noisy (i.e. could be off the actual orbit). Our goal is to determine the relationship between x_1 and x_2.

We assume that the ellipse from which the points were generated has the form

$$w_1 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + w_4 x_1 + w_5 x_2 = 1$$

where the coefficients $w_1, \ldots, w_5$ are the parameters we wish to estimate.

We formulate the problem with least-squares:

$$\min_{\mathbf{w}} \|\boldsymbol{\Phi}\mathbf{w} - \mathbf{1}\|_2^2$$

where

$$\boldsymbol{\Phi} = \begin{bmatrix} x_{1,1}^2 & x_{2,1}^2 & x_{1,1}x_{2,1} & x_{1,1} & x_{2,1} \\ x_{1,2}^2 & x_{2,2}^2 & x_{1,2}x_{2,2} & x_{1,2} & x_{2,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{1,n}^2 & x_{2,n}^2 & x_{1,n}x_{2,n} & x_{1,n} & x_{2,n} \end{bmatrix}$$

In this case, the feature map $\boldsymbol{\phi}$ is given by

$$\boldsymbol{\phi}(\mathbf{x}) = (x_1^2, x_2^2, x_1x_2, x_1, x_2)$$

Note that there is no "target" vector $\mathbf{y}$ here, so this is not a traditional regression problem, but it still fits into the framework of least-squares.

Polynomial Features

The example above demonstrates an important class of features known as **polynomial features**. Remember that a polynomial is linear combination of monomial basis terms. Monomials can be classified in two ways, by their degree and dimension:

Dimension \ Degree	0	1	2	3	…
1 (univariate)	1	x	x^2	x^3	$\cdots$
2 (bivariate)	1	x_1, x_2	x_1^2, x_2^2, x_1x_2	$x_1^3, x_2^3, x_1^2x_2, x_1x_2^2$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

A big reason we care polynomial features is that any smooth function can be approximated arbitrarily closely by some polynomial.[2] For this reason, polynomials are said to be **universal approximators**.

One downside of polynomials is that as their degree increases, their number of terms increases rapidly. Specifically, one can use a "stars and bars" style combinatorial argument[3] to show that a polynomial of degree d in ℓ variables has

$$\binom{\ell+d}{\ell} = \frac{(\ell+d)!}{\ell!d!}$$

terms. To get an idea for how quickly this quantity grows, consider a few examples:

ℓ \ d	1	3	5	10	25
1	2	4	6	11	26
3	4	20	56	286	3276
5	6	56	252	3003	142506
10	11	286	3003	184756	183579396
25	26	3276	142506	183579396	126410606437752

Later we will learn about the **kernel trick**, a clever mathematical method that allows us to circumvent this rapidly growing cost in certain cases.

1.4 Hyperparameters and Validation

As above, consider a hypothesis of the form

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=1}^{d} w_j \phi_j(\mathbf{x}) = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})$$

[2] **Taylor's theorem** gives more precise statements about the approximation error.

[3] We count the number of distinct monomials of degree at most d in ℓ variables $x_1, \dots, x_\ell$, or equivalently, the number of distinct monomials of degree exactly d in $\ell+1$ variables $x_0 = 1, x_1 \dots, x_\ell$. Every monomial has the form $x_0^{k_0} \dots x_\ell^{k_\ell}$ where $k_0 + \cdots + k_\ell = d$. This corresponds to an arrangement of d stars and ℓ bars, where the number of stars between consecutive bars (or the ends of the expression) gives the degree of that ordered variable. For example,

$$*|***|** \quad \leftrightarrow \quad x_0^1 x_1^3 x_2^2$$

The number of unique ways to arrange these stars and bars is the number of ways to choose the positions of the ℓ bars out of the total $\ell+d$ slots, i.e. $\ell+d$ choose ℓ. (You could also pick the positions of the d stars out of the total $\ell+d$ slots; the expression is symmetric in ℓ and d.)

Observe that the model order d is not one of the decision variables being optimized when we fit to the data. For this reason d is called a **hyperparameter**. We might say more specifically that it is a **model hyperparameter**, since it determines the structure of the model.

For another example, recall **ridge regression**, in which we add an ℓ^2 penalty on the parameters $\mathbf{w}$:

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$$

The regularization weight λ is also a hyperparameter, as it is fixed during the minimization above. However λ, unlike the previously discussed hyperparameter d, is not a part of the model. Rather, it is an aspect of the optimization procedure used to fit the model, so we say it is an **optimization hyperparameter**. Hyperparameters tend to fall into one of these two categories.

Since hyperparameters are not determined by the data-fitting optimization procedure, how should we choose their values? A suitable answer to this question requires some discussion of the different types of error at play.

Types of Error

We have seen that it is common to minimize some measure of how poorly our hypothesis fits the data we have, but what we actually care about is how well the hypothesis predicts *future* data. Let us try to formally distinguish the various types of error. Assume that the data are distributed according to some (unknown) distribution $\mathcal{D}$, and that we have a **loss function** $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, which is to measure the error between the true output y and our estimate $\hat{y} = h(\mathbf{x})$. The **risk** (or **true error**) of a particular hypothesis $h \in \mathcal{H}$ is the expected loss over the whole data distribution:

$$R(h) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\ell(h(\mathbf{x}), y)]$$

Ideally, we would find the hypothesis that minimizes the risk, i.e.

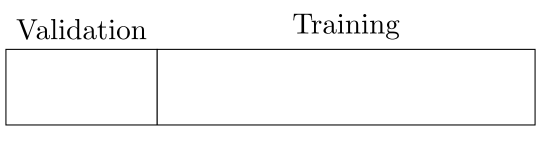
$$h^* = \operatorname*{arg\,min}_{h\in\mathcal{H}} R(h)$$

However, computing this expectation is impossible because we do not have access to the true data distribution. Rather, we have access to samples $(\mathbf{x}_i, y_i) \overset{\text{iid}}{\sim} \mathcal{D}$. These enable us to approximate the real problem we care about by minimizing the **empirical risk** (or **training error**)

$$\hat{R}_{\text{TRAIN}}(h) = \frac{1}{n}\sum_{i=1}^{n} \ell(h(\mathbf{x}_i), y_i)$$

But since we have a finite number of samples, the hypothesis that performs the best on the training data is not necessarily the best on the whole data distribution. In particular, if we both train and evaluate the hypothesis using the same data points, the training error will be a very biased estimate of the true error, since the hypothesis has been chosen specifically to perform well on those points. This phenomenon is sometimes referred to as "data incest".

A common solution is to set aside some portion (say 30%) of the data, to be called the **validation set**, which is disjoint from the training set and *not* allowed to be used when fitting the model:



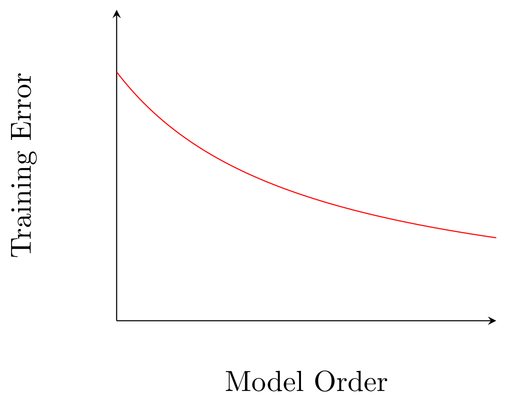
We can use this validation set to estimate the true error by the **validation error**

$$\hat{R}_{\text{VAL}}(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(h(\mathbf{x}_i^{\text{val}}), y_i^{\text{val}})$$

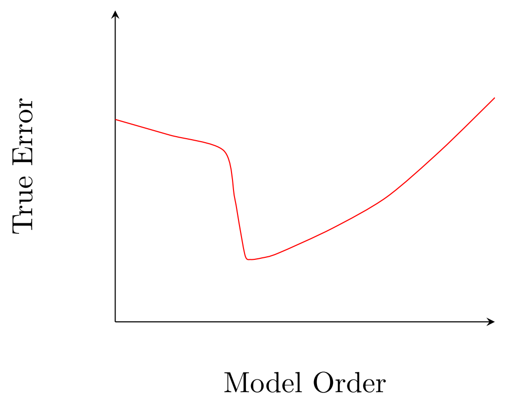
With this estimate, we have a simple method for choosing hyperparameter values: try a bunch of configurations of the hyperparameters and choose the one that yields the lowest validation error.

The effect of hyperparameters on error

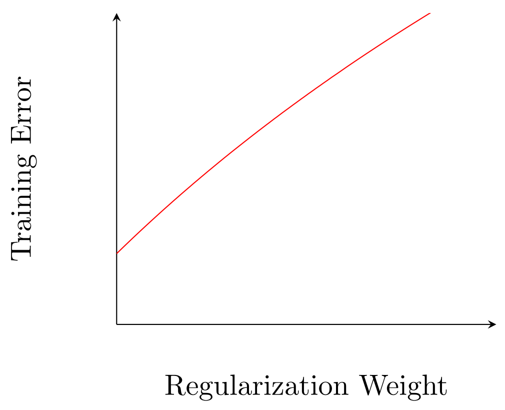
Note that as we add more features to a linear model, training error can only decrease. This is because the optimizer can set $w_i = 0$ if feature i cannot be used to reduce training error.



Adding more features tends to reduce true error as long as the additional features are useful predictors of the output. However, if we keep adding features, these begin to fit noise in the training data instead of the true signal, causing true error to actually increase. This phenomenon is known as **overfitting**.



The validation error tracks the true error reasonably well as long as the validation set is sufficiently large. The regularization hyperparameter λ has a somewhat different effect on training error. Observe that if $\lambda = 0$, we recover the exact OLS problem, which is directly minimizing the training error. As λ increases, the optimizer places less emphasis on the training error and more emphasis on reducing the magnitude of the parameters. This leads to a degradation in training error as λ grows:

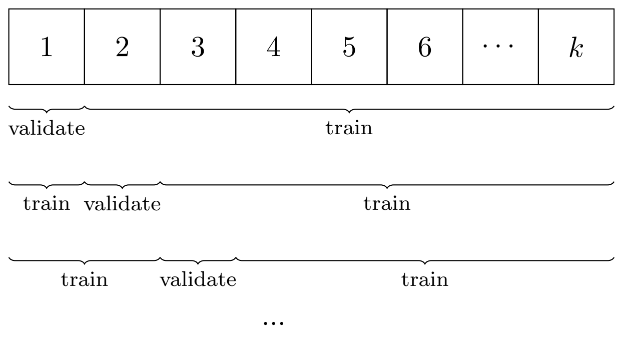


Cross-validation

Setting aside a validation set works well, but comes at a cost, since we cannot use the validation data for training. Since having more data generally improves the quality of the trained model, we may prefer not to let that data go to waste, especially if we have little data to begin with and/or collecting more data is expensive. Cross-validation is an alternative to having a dedicated validation set.

k-fold cross-validation works as follows:

1. Shuffle the data and partition it into k equally-sized (or as equal as possible) blocks.
2. For $i = 1, \dots, k$,
    - Train the model on all the data except block i.
    - Evaluate the model (i.e. compute the validation error) using block i.



3. Average the k validation errors; this is our final estimate of the true error.

Observe that, although every datapoint is used for evaluation at some time or another, the model is always evaluated on a different set of points than it was trained on, thereby cleverly avoiding the "data incest" problem mentioned earlier.

Note also that this process (except for the shuffling and partitioning) must be repeated for every hyperparameter configuration we wish to test. This is the principle drawback of k-fold cross-validation as compared to using a held-out validation set – there is roughly k times as much computation required. This is not a big deal for the relatively small linear models that we've seen so far, but it can be prohibitively expensive when the model takes a long time to train, as is the case in the Big Data regime or when using neural networks.

Chapter 2

Regression II

2.1 MLE and MAP for Regression (Part I)

So far, we've explored two approaches of the regression framework, Ordinary Least Squares and Ridge Regression:

$$\hat{\mathbf{w}}_{\text{OLS}} = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$$
$$\hat{\mathbf{w}}_{\text{RIDGE}} = \arg\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$$

One question that arises is why we specifically use the ℓ^2 norm to measure the error of our predictions, and to penalize the model parameters. We will justify this design choice by exploring the statistical interpretations of regression — namely, we will employ Gaussians, MLE and MAP to validate what we've done so far through a different lens.

Probabilistic Model

In the context of **supervised learning**, we assume that there exists a **true underlying model** mapping inputs to outputs:

$$f : \mathbf{x} \to f(\mathbf{x})$$

The true model is unknown to us, and our goal is to find a **hypothesis model** that best represents the true model. The only information that we have about the true model is via a dataset

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$$

where $\mathbf{x}_i \in \mathbb{R}^d$ is the input and $y_i \in \mathbb{R}$ is the observation, a noisy version of the true output $f(\mathbf{x}_i)$:

$$Y_i = f(\mathbf{x}_i) + Z_i$$

We assume that $\mathbf{x}_i$ is a fixed value (which implies that $f(\mathbf{x}_i)$ is fixed as well), while Z_i is a random variable (which implies that Y_i is a random variable as well). We always assume that Z_i has zero mean, because otherwise there would be systematic bias in our observations. The Z_i's could be Gaussian, uniform, Laplacian, etc... In most contexts, we us assume that they are **independent identically distributed (i.i.d)** Gaussians: $Z_i \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$. We can therefore say that Y_i is a random variable whose probability distribution is given by

$$Y_i \overset{\text{iid}}{\sim} \mathcal{N}(f(\mathbf{x}_i), \sigma^2)$$

Now that we have defined the model and data, we wish to find a hypothesis model $h_{\boldsymbol{\theta}}$ (parameterized by $\boldsymbol{\theta}$) that best captures the relationships in the data, while possibly taking into account prior beliefs that we have about the true model. We can represent this as a probability problem, where the goal is to find the optimal model that maximizes our probability.

Maximum Likelihood Estimation

In **Maximum Likelihood Estimation** (MLE), the goal is to find the hypothesis model that maximizes the probability of the data. If we parameterize the set of hypothesis models with $\boldsymbol{\theta}$, we can express the problem as

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = p(\text{data} = \mathcal{D} \mid \text{ true model} = h_{\boldsymbol{\theta}})$$

The quantity $\mathcal{L}(\boldsymbol{\theta})$ that we are maximizing is also known as the **likelihood**, hence the term MLE. Substituting our representation of $\mathcal{D}$ we have

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = p(y_1, \ldots, y_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \boldsymbol{\theta})$$

Note that we implicitly condition on the $\mathbf{x}_i$'s, because we treat them as *fixed* values of the data. The only randomness in our data comes from the y_i's (since they are noisy versions of the true values $f(\mathbf{x}_i)$). We can further simplify the problem by working with the **log likelihood** $\ell(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = \log \mathcal{L}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})$

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = \arg\max_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})$$

With logs we are still working with the same problem, because logarithms are monotonic functions. In other words we have that:

$$P(A) < P(B) \iff \log P(A) < \log P(B)$$

Let's decompose the log likelihood:

$$\ell(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = \log p(y_1, \ldots, y_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \boldsymbol{\theta}) = \log \prod_{i=1}^{n} p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta}) = \sum_{i=1}^{n} \log[p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta})]$$

We decoupled the probabilities from each datapoints because their corresponding noise components are independent. Note that the logs allow us to work with sums rather products, simplifying the problem — one reason why the log likelihood is such a powerful tool. Each individual term $p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta})$ comes from a Gaussian

$$Y_i \mid \boldsymbol{\theta} \sim \mathcal{N}(h_{\boldsymbol{\theta}}(\mathbf{x}_i), \sigma^2)$$

Continuing with logs:

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) \tag{2.1}$$

$$= \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log[p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta})] \tag{2.2}$$

$$= \arg\max_{\boldsymbol{\theta}} -\Big(\sum_{i=1}^{n} \frac{(y_i - h_{\boldsymbol{\theta}}(\mathbf{x}_i))^2}{2\sigma^2}\Big) - n \log \sqrt{2\pi}\sigma \tag{2.3}$$

$$= \arg\min_{\boldsymbol{\theta}} \Big(\sum_{i=1}^{n} \frac{(y_i - h_{\boldsymbol{\theta}}(\mathbf{x}_i))^2}{2\sigma^2}\Big) + n \log \sqrt{2\pi}\sigma \tag{2.4}$$

$$= \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} (y_i - h_{\boldsymbol{\theta}}(\mathbf{x}_i))^2 \tag{2.5}$$

Note that in step (4) we turned the problem from a maximization problem to a minimization problem by negating the objective. In step (5) we eliminated the second term and the denominator in the first term, because they do not depend on the variables we are trying to optimize over.

Now let's look at the case of regression — our hypothesis has the form $h_{\boldsymbol{\theta}}(\mathbf{x}_i) = \mathbf{x}_i{}^\top\boldsymbol{\theta}$, where $\boldsymbol{\theta} \in \mathbb{R}^d$, where d is the number of dimensions of our featurized datapoints. For this specific setting, the problem becomes:

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^d} \sum_{i=1}^{n} (y_i - \mathbf{x}_i{}^\top\boldsymbol{\theta})^2$$

This is just the Ordinary Least Squares (OLS) problem! We just proved that OLS and MLE for regression lead to the same answer! We conclude that MLE is a probabilistic justification for why using squared error (which is the basis of OLS) is a good metric for evaluating a regression model.

Maximum a Posteriori

In **Maximum a Posteriori** (MAP) Estimation, the goal is to find the model, for which the data maximizes the probability of the model:

$$\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\text{true model} = h_{\boldsymbol{\theta}} \mid \text{data} = \mathcal{D})$$

The probability distribution that we are maximizing is known as the **posterior**. Maximizing this term directly is often infeasible, so we we use **Bayes' Rule** to re-express the objective.

$$\begin{aligned}
\hat{\boldsymbol{\theta}}_{\text{MAP}} &= \arg\max_{\boldsymbol{\theta}} p(\text{true model} = h_{\boldsymbol{\theta}} \mid \text{data} = \mathcal{D}) \\
&= \arg\max_{\boldsymbol{\theta}} \frac{p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\boldsymbol{\theta}}) \cdot p(\text{true model} = h_{\boldsymbol{\theta}})}{p(\text{data} = \mathcal{D})} \\
&= \arg\max_{\boldsymbol{\theta}} p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\boldsymbol{\theta}}) \cdot p(\text{true model} = h_{\boldsymbol{\theta}}) \\
&= \arg\max_{\boldsymbol{\theta}} \log p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\boldsymbol{\theta}}) + \log p(\text{true model} = h_{\boldsymbol{\theta}}) \\
&= \arg\min_{\boldsymbol{\theta}} -\log p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\boldsymbol{\theta}}) - \log p(\text{true model} = h_{\boldsymbol{\theta}})
\end{aligned}$$

We treat $p(\text{data} = \mathcal{D})$ as a constant value because it does not depend on the variables we are optimizing over. Notice that MAP is just like MLE, except we add a term $p(\text{true model} = h_{\boldsymbol{\theta}})$ to our objective. This term is the **prior** over our true model. Adding the prior has the effect of favoring certain models over others *a priori*, regardless of the dataset. Note the MLE is a special case of MAP, when the prior does not treat any model more favorably over other models. Concretely, we have that

$$\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\min_{\boldsymbol{\theta}} -\Big(\sum_{i=1}^{n} \log[p(y_i \mid \mathbf{x}_i, \boldsymbol{\theta})]\Big) - \log[p(\boldsymbol{\theta})]$$

Again, just as in MLE, notice that we implicitly condition on the $\mathbf{x}_i$'s because we treat them as constants. Also, let us assume as before that the noise terms are i.i.d. Gaussians: $N_i \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$. For the prior term $P(\boldsymbol{\Theta})$, we assume that the components θ_j are i.i.d. Gaussians:

$$\theta_j \overset{\text{iid}}{\sim} \mathcal{N}(\theta_{j_0}, \sigma_h^2)$$

Using this specific information, we now have:

$$\begin{aligned}
\hat{\boldsymbol{\theta}}_{\text{MAP}} &= \arg\min_{\boldsymbol{\theta}} \left(\frac{\sum_{i=1}^n (y_i - h_{\boldsymbol{\theta}}(\mathbf{x}_i))^2}{2\sigma^2} \right) + \left(\frac{\sum_{j=1}^d (\theta_j - \theta_{j_0})^2}{2\sigma_h^2} \right) \\
&= \arg\min_{\boldsymbol{\theta}} \left(\sum_{i=1}^n (y_i - h_{\boldsymbol{\theta}}(\mathbf{x}_i))^2 \right) + \frac{\sigma^2}{\sigma_h^2} \left(\sum_{j=1}^d (\theta_j - \theta_{j_0})^2 \right)
\end{aligned}$$

Let's look again at the case for linear regression to illustrate the effect of the prior term when $\theta_{j_0} = 0$. In this context, we refer to the linear hypothesis function $h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x}$.

$$\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\min_{\boldsymbol{\theta} \in \mathbb{R}^d} \sum_{i=1}^n (y_i - \mathbf{x}_i{}^\top \boldsymbol{\theta})^2 + \frac{\sigma^2}{\sigma_h^2} \sum_{j=1}^d \theta_j^2$$

This is just the Ridge Regression problem! We just proved that Ridge Regression and MAP for regression lead to the same answer! We can simply set $\lambda = \frac{\sigma^2}{\sigma_h^2}$. We conclude that MAP is a probabilistic justification for adding the penalized ridge term in Ridge Regression.

MLE vs. MAP

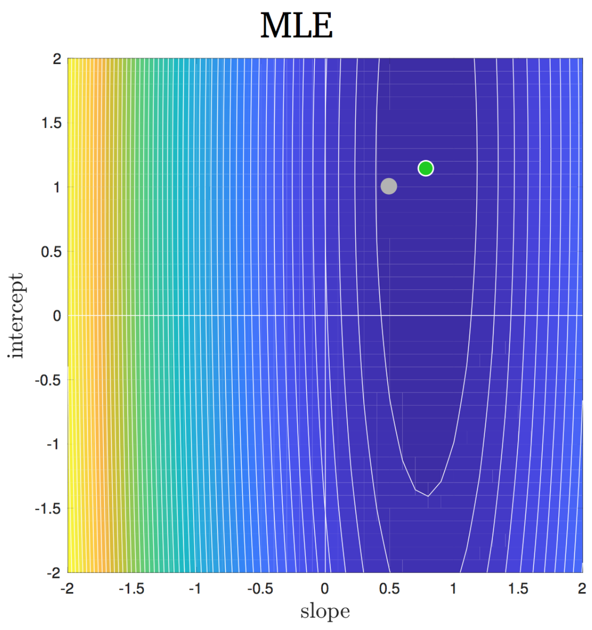
Based on our analysis of Ordinary Least Squares Regression and Ridge Regression, we should expect to see MAP perform better than MLE. But is that always the case? Let us visit a simple 2D problem where

$$f(x) = \text{slope} \cdot x + \text{intercept}$$

Suppose we already know the true underlying model parameters:

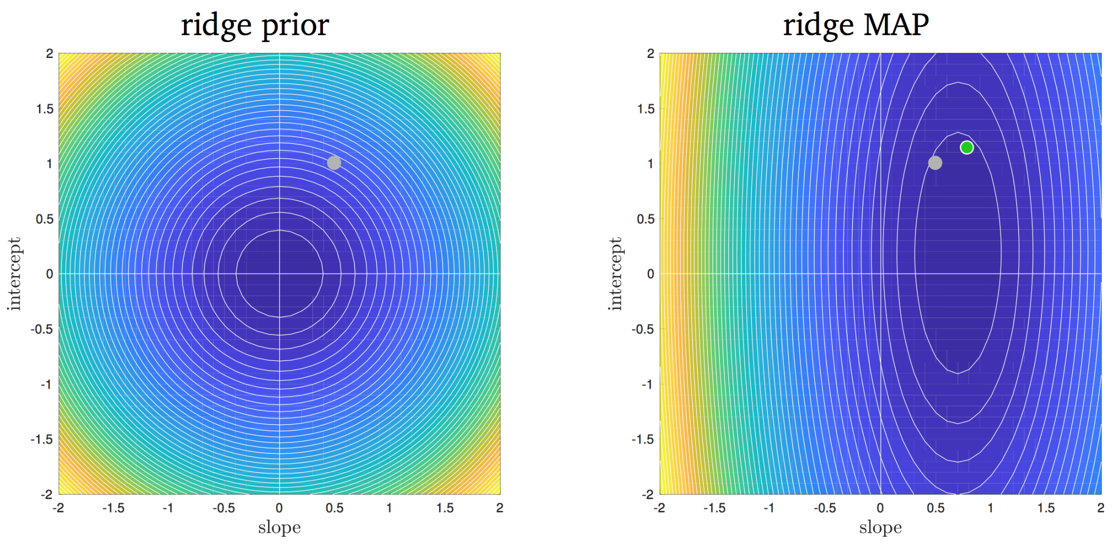
$$(\text{slope}^*, \text{intercept}^*) = (0.5, 1.0)$$

we would like to know what parameters MLE and MAP will select, after providing them with some dataset $\mathcal{D}$. Let's start with MLE:



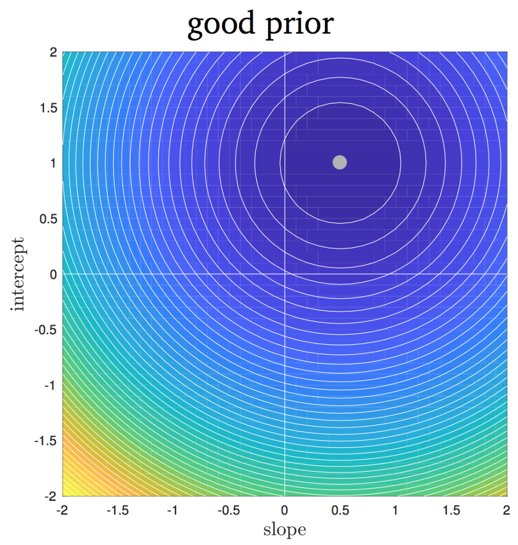
The diagram above shows the the contours of the likelihood distribution in model space. The gray dot represents the true underlying model. MLE chooses the point that maximizes the likelihood, which is indicated by the green dot. As we can see, MLE chooses a reasonable hypothesis, but this hypothesis lies in a region on high variance, which indicates a high level of uncertainty in the predicted model. A slightly different dataset could significantly alter the predicted model.

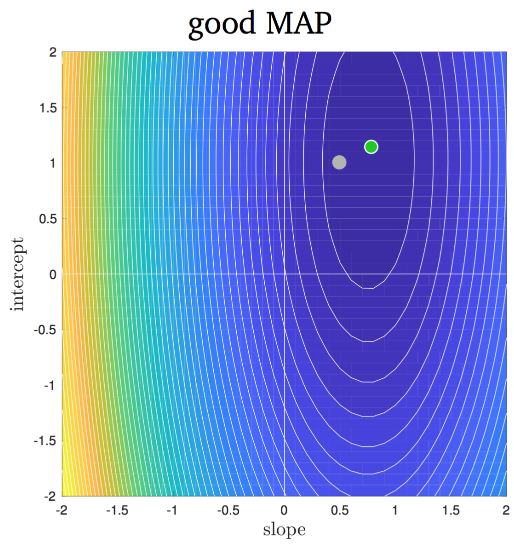
Now, let's take a look at the hypothesis model from MAP. One question that arises is where the prior should be centered and what its variance should be. This depends on our belief of what the true underlying model is. If we have reason to believe that the model weights should all be small, then the prior should be centered at zero with a small variance. Let's look at MAP for a prior that is centered at zero:



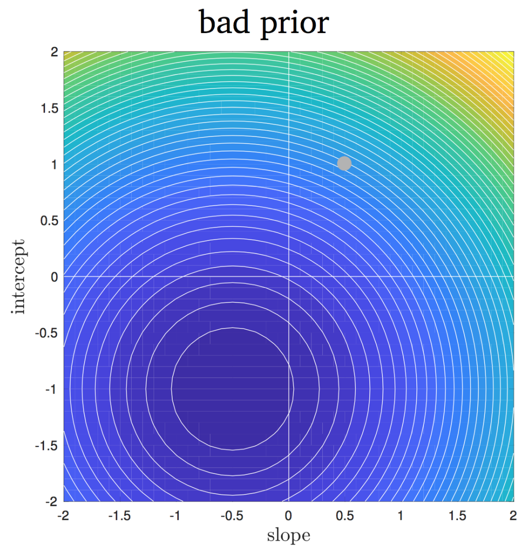
For reference, we have marked the MLE estimation from before as the green point and the true model as the gray point. The prior distribution is indicated by the diagram on the left, and the posterior distribution is indicated by the diagram on the right. MAP chooses the point that maximizes the posterior probability, which is approximately $(0.70, 0.25)$. Using a prior centered at zero leads us to skew our prediction of the model weights toward the origin, leading to a less accurate hypothesis than MLE. However, the posterior has significantly less variance, meaning that the point that MAP chooses is less likely to overfit to the noise in the dataset.

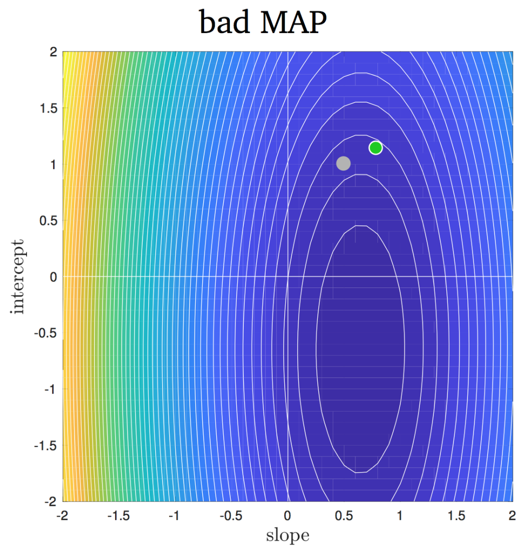
Let's say in our case that we have reason to believe that both model weights should be centered around the 0.5 to 1 range.





Our prediction is now close to that of MLE, with the added benefit that there is significantly less variance. However, if we believe the model weights should be centered around the -0.5 to -1 range, we would make a much poorer prediction than MLE.





As always, in order to compare our beliefs to see which prior works best in practice, we should use cross validation!

2.2 Bias-Variance Tradeoff

Recall from our previous discussion on supervised learning, that for a fixed input $\mathbf{x}$ the corresponding measurement Y is a noisy measurement of the true underlying response $f(\mathbf{x})$:

$$Y = f(\mathbf{x}) + Z$$

Where Z is a zero-mean random variable, and is typically represented as a Gaussian distribution. Our goal in regression is to recover the underlying model $f(.)$ as closely as possible. We previously mentioned MLE and MAP as two techniques that try to find of reasonable approximation to $f(.)$ by solving a probabilistic objective. We briefly compared the effectiveness of MLE and MAP, and noted that the effectiveness of MAP is in large part dependent on the prior over the parameters we optimize over. One question that naturally arises is: how exactly can we measure the effectiveness of a hypothesis model? In this section, we would like to form a theoretical metric that can exactly measure the effectiveness of a hypothesis function h. Keep in mind that this is only a theoretical metric that cannot be measured in real life, but it can be approximated via empirical experiments — more on this later.

Before we introduce the metric, let's make a few subtle statements about the data and hypothesis. As you may recall from our previous discussion on MLE and MAP, we had a dataset

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$$

In that context, we treated the $\mathbf{x}_i$'s in our dataset $\mathcal{D}$ as *fixed* values. In this case however, we treat the $\mathbf{x}_i$'s as values sampled from **random variables $\mathbf{X}_i$**. That is, $\mathcal{D}$ is a random variable, consisting of random variables $\mathbf{X}_i$ and Y_i. For some arbitrary test input $\mathbf{x}$, $h(\mathbf{x}; \mathcal{D})$ depends on the random variable $\mathcal{D}$ that was used to train h. Since $\mathcal{D}$ is random, we will have a slightly different hypothesis model $h(\mathbf{x}; \mathcal{D})$ every time we use a new dataset. Note that $\mathbf{x}$ and $\mathcal{D}$ are completely independent from one another — $\mathbf{x}$ is a test point, while $\mathcal{D}$ consists of the training data.

Metric

Our objective is to, for a fixed test point $\mathbf{x}$, evaluate how closely the hypothesis can estimate the *noisy* observation Y corresponding to $\mathbf{x}$. Note that we have denoted $\mathbf{x}$ here as a lowercase letter because we are treating it as a fixed constant, while we have denoted the Y and $\mathcal{D}$ as uppercase letters because we are treating them as random variables. Y and $\mathcal{D}$ as **independent random variables**, because our $\mathbf{x}$ and Y have no relation to the set of $\mathbf{X}_i$'s and Y_i's in $\mathcal{D}$. Again, we can view $\mathcal{D}$ as the training data, and $(\mathbf{x}, Y)$ as a test point — the test point $\mathbf{x}$ is probably not even in the training set $\mathcal{D}$! Mathematically, we express our metric as the expected squared error between the hypothesis and the observation $Y = f(\mathbf{x}) + Z$:

$$\varepsilon(\mathbf{x}; h) = \mathbb{E}[(h(\mathbf{x}; \mathcal{D}) - Y)^2]$$

The expectation here is over two random variables, $\mathcal{D}$ and Y:

$$\mathbb{E}_{\mathcal{D},Y}[(h(\mathbf{x}; \mathcal{D}) - Y)^2] = \mathbb{E}_{\mathcal{D}}[\mathbb{E}_Y[(h(\mathbf{x}; \mathcal{D}) - Y)^2 | \mathcal{D}]]$$

Note that the error is w.r.t the observation Y and not the true underlying model $f(\mathbf{x})$, because we do not know the true model and only have access to the noisy observations from the true model.

Bias-Variance Decomposition

The error metric is difficult to interpret and work with, so let's try to decompose it into parts that are easier to understand. Before we start, let's find the expectation and variance of Y:

$$\mathbb{E}[Y] = \mathbb{E}[f(\mathbf{x}) + Z] = f(\mathbf{x}) + \mathbb{E}[Z] = f(\mathbf{x})$$

$$\text{Var}(Y) = \text{Var}(f(\mathbf{x}) + Z) = \text{Var}(Z)$$

Also, in general for any random variable X, we have that

$$\text{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2 \implies \mathbb{E}[X^2] = \text{Var}(X) + \mathbb{E}[X]^2$$

Let's use these facts to decompose the error:

$$\begin{aligned}
\varepsilon(\mathbf{x}; h) &= \mathbb{E}[(h(\mathbf{x}; \mathcal{D}) - Y)^2] = \mathbb{E}[h(\mathbf{x}; \mathcal{D})^2] + \mathbb{E}[Y^2] - 2\mathbb{E}[h(\mathbf{x}; \mathcal{D}) \cdot Y] \\
&= \Big(\text{Var}(h(\mathbf{x}; \mathcal{D})) + \mathbb{E}[h(\mathbf{x}; \mathcal{D})]^2\Big) + \Big(\text{Var}(Y) + \mathbb{E}[Y]^2\Big) - 2\mathbb{E}[h(\mathbf{x}; \mathcal{D})] \cdot \mathbb{E}[Y] \\
&= \Big(\mathbb{E}[h(\mathbf{x}; \mathcal{D})]^2 - 2\mathbb{E}[h(\mathbf{x}; \mathcal{D})] \cdot \mathbb{E}[Y] + \mathbb{E}[Y]^2\Big) + \text{Var}(h(\mathbf{x}; \mathcal{D})) + \text{Var}(Y) \\
&= \Big(\mathbb{E}[h(\mathbf{x}; \mathcal{D})] - \mathbb{E}[Y]\Big)^2 + \text{Var}(h(\mathbf{x}; \mathcal{D})) + \text{Var}(Y) \\
&= \underbrace{\Big(\mathbb{E}[h(\mathbf{x}; \mathcal{D})] - f(\mathbf{x})\Big)^2}_{bias^2 \text{ of method}} + \underbrace{\text{Var}(h(\mathbf{x}; \mathcal{D}))}_{\text{variance of method}} + \underbrace{\text{Var}(Z)}_{\text{irreducible error}}
\end{aligned}$$

Recall that for any two independent random variables $\mathcal{D}$ and Y, $g_1(\mathcal{D})$ and $g_2(Y)$ are also independent, for any functions g_1, g_2. This implies that $h(\mathbf{x}; \mathcal{D})$ and Y are independent, allowing us to express $\mathbb{E}[h(\mathbf{x}; \mathcal{D}) \cdot Y] = \mathbb{E}[h(\mathbf{x}; \mathcal{D})] \cdot \mathbb{E}[Y]$ in the second line of the derivation. The final decomposition, also known as the **bias-variance decomposition**, consists of three terms:

- **Bias^2 of method**: Measures how well the *average* hypothesis (over all possible training sets) can come close to the true underlying value $f(\mathbf{x})$, for a fixed value of $\mathbf{x}$. A low bias means that on average the regressor $h(\mathbf{x})$ accurately estimates $f(\mathbf{x})$.
- **Variance of method**: Measures the variance of the hypothesis (over all possible training sets), for a fixed value of $\mathbf{x}$. A low variance means that the prediction does not change much as the training set varies. An un-biased method (bias = 0) could have a large variance.
- **Irreducible error**: This is the error in our model that we cannot control or eliminate, because it is due to errors inherent in our noisy observation Y.

The decomposition allows us to measure the error in terms of bias, variance, and irreducible error. Irreducible error has no relation with the hypothesis model, so we can fully ignore it in theory when minimizing the error. As we have discussed before, models that are very complex have very little bias because on *average* they can fit the true underlying model value $f(\mathbf{x})$ very well, but have very high variance and are very far off from $f(\mathbf{x})$ on an individual basis.

Note that the error above is only for a fixed input $\mathbf{x}$, but in regression our goal is to minimize the average error over all possible values of $\mathbf{X}$. If we know the distribution for $\mathbf{X}$, we can find the effectiveness of a hypothesis model as a whole by taking an expectation of the error over all possible values of $\mathbf{x}$: $\mathbb{E}_{\mathbf{X}}[\varepsilon(\mathbf{x}; h)]$.

Alternative Decomposition

The previous derivation is short, but may seem somewhat arbitrary. Let's explore an alternative derivation. At its core, it uses the technique that $\mathbb{E}[(Z-Y)^2] = \mathbb{E}[((Z-\mathbb{E}[Z]) + (\mathbb{E}[Z]-Y))^2]$ which decomposes to easily give us the variance of Z and other terms.

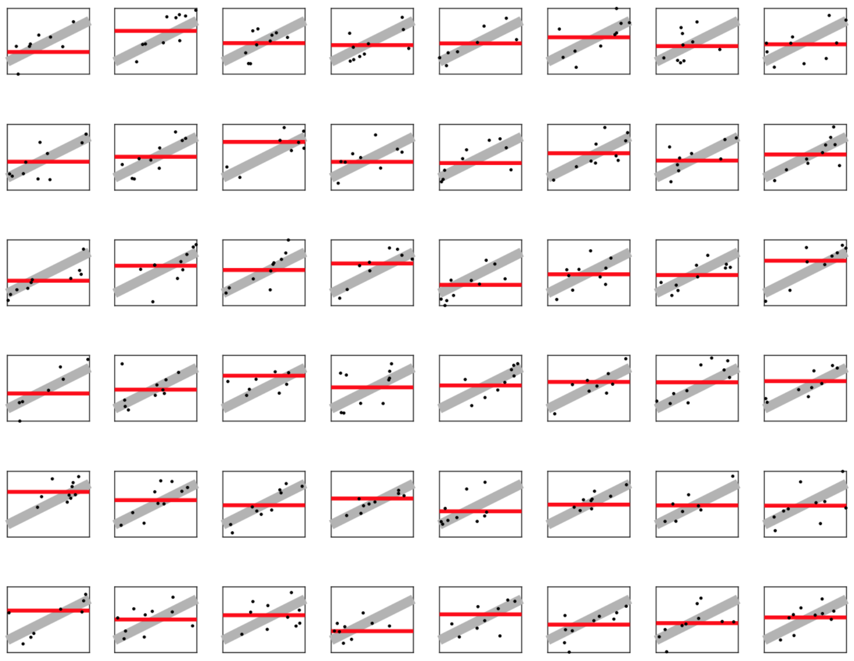
$$
\begin{aligned}
\varepsilon(\mathbf{x};h) &= \mathbb{E}[(h(\mathbf{x};\mathcal{D}) - Y)^2] \\
&= \mathbb{E}\left[\big(h(\mathbf{x};\mathcal{D}) - \mathbb{E}[h(\mathbf{x};\mathcal{D})] + \mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y\big)^2\right] \\
&= \mathbb{E}\left[\big(h(\mathbf{x};\mathcal{D}) - \mathbb{E}[h(\mathbf{x};\mathcal{D})]\big)^2\right] + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y\big)^2\right] + 2\mathbb{E}\left[\big(h(\mathbf{x};\mathcal{D}) - \mathbb{E}[h(\mathbf{x};\mathcal{D})]\big)\cdot\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y\big)\right] \\
&= \mathbb{E}\left[\big(h(\mathbf{x};\mathcal{D}) - \mathbb{E}[h(\mathbf{x};\mathcal{D})]\big)^2\right] + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y\big)^2\right] + 2\mathbb{E}[\cancel{h(\mathbf{x};\mathcal{D}) - \mathbb{E}[h(\mathbf{x};\mathcal{D})]}]\cdot\mathbb{E}[\mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y] \\
&= \mathbb{E}\left[\big(h(\mathbf{x};\mathcal{D}) - \mathbb{E}[h(\mathbf{x};\mathcal{D})]\big)^2\right] + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y\big)^2\right] \\
&= \mathrm{Var}((h(\mathbf{x};\mathcal{D})) + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - Y\big)^2\right] \\
&= \mathrm{Var}((h(\mathbf{x};\mathcal{D})) + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - \mathbb{E}[Y] + \mathbb{E}[Y] - Y\big)^2\right] \\
&= \mathrm{Var}((h(\mathbf{x};\mathcal{D})) + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - \mathbb{E}[Y]\big)^2\right] + \mathbb{E}[(Y-\mathbb{E}[Y])^2] + 2\left(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - \mathbb{E}[Y]\right)\cdot\mathbb{E}[\cancel{\mathbb{E}[Y] - Y}] \\
&= \mathrm{Var}((h(\mathbf{x};\mathcal{D})) + \mathbb{E}\left[\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - \mathbb{E}[Y]\big)^2\right] + \mathbb{E}[(Y-\mathbb{E}[Y])^2] \\
&= \mathrm{Var}((h(\mathbf{x};\mathcal{D})) + \big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - \mathbb{E}[Y]\big)^2 + \mathrm{Var}(Y) \\
&= \mathrm{Var}((h(\mathbf{x};\mathcal{D})) + \big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - f(\mathbf{x})\big)^2 + \mathrm{Var}(Z) \\
&= \underbrace{\big(\mathbb{E}[h(\mathbf{x};\mathcal{D})] - f(\mathbf{x})\big)^2}_{bias^2 \text{ of method}} + \underbrace{\mathrm{Var}(h(\mathbf{x};\mathcal{D}))}_{\text{variance of method}} + \underbrace{\mathrm{Var}(Z)}_{\text{irreducible error}}
\end{aligned}
$$

Experiments

Let's confirm the theory behind the bias-variance decomposition with an empirical experiment that measures the bias and variance for polynomial regression with 0 degree, 1st degree, and 2nd degree polynomials. In our experiment, we will repeatedly fit our hypothesis model to a random training set. We then find the expectation and variance of the fitted models generated from these training sets.

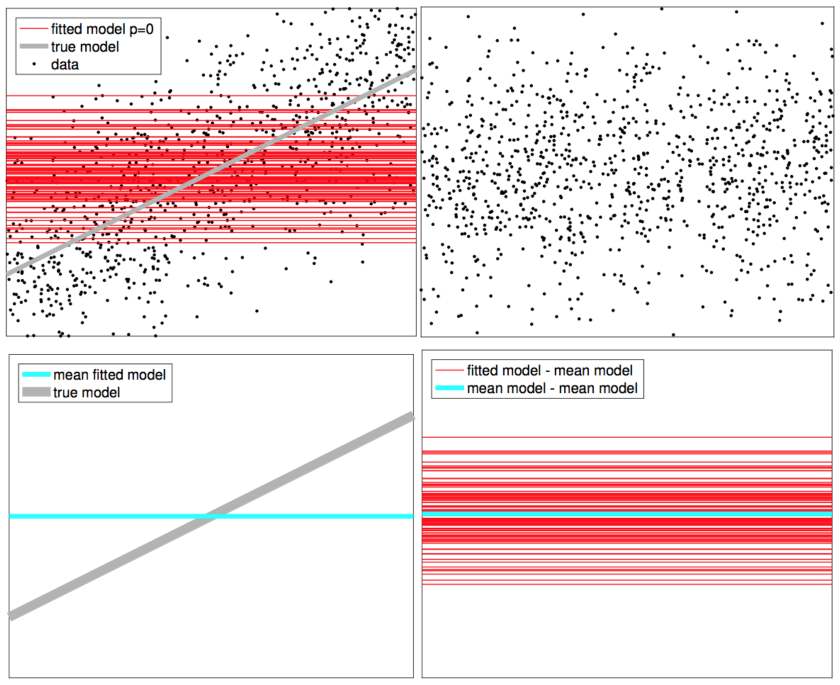
Let's first look at a 0 degree (constant) regression model. We repeatedly fit an optimal constant line to a training set of 10 points. The true model is denoted by gray and the hypothesis is denoted by red. Notice that at each time the red line is slightly different due to the different training set used.

Fitting A Model over Multiple Datasets: $p = 0$



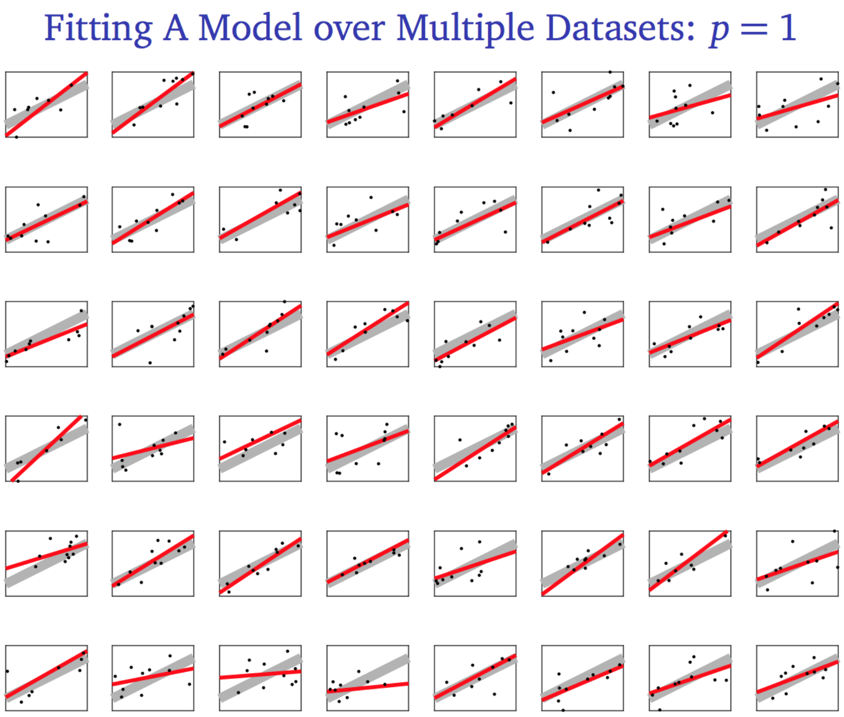
Let's combine all of these hypotheses together into one picture to see the bias and variance of our model.

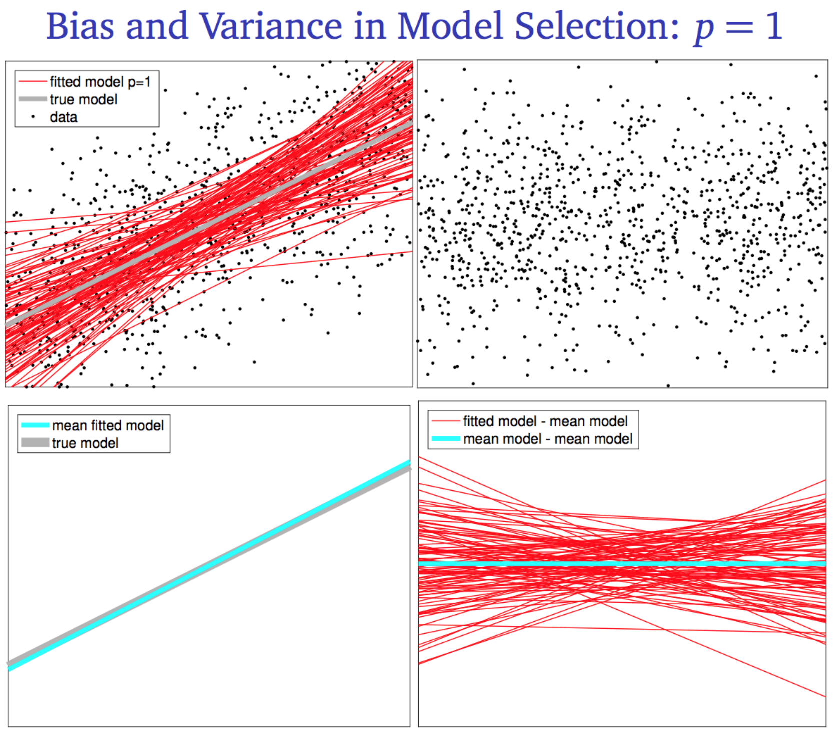
Bias and Variance in Model Selection: $p = 0$



On the top left diagram we see all of our hypotheses and all training sets used. The bottom left diagram shows the average hypothesis in cyan. As we can see, this model has low bias for **x**'s in the center of the graph, but very high bias for $\mathbf{x}$'s that are away from the center of the graph. The diagram in the bottom right shows that the variance of the hypotheses is quite high, for all values of $\mathbf{x}$.

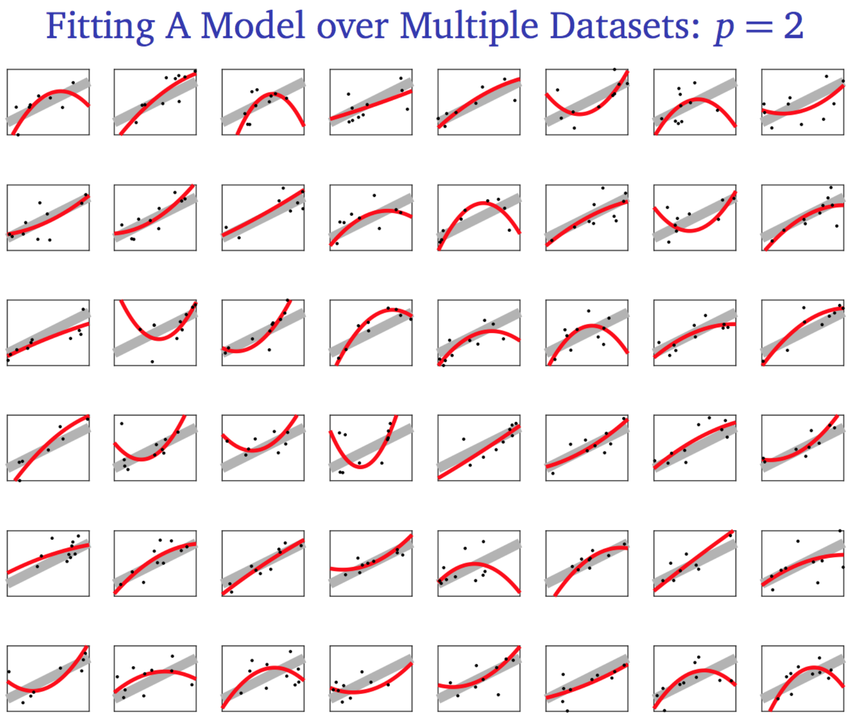
Now let's look at a 1st degree (linear) regression model.

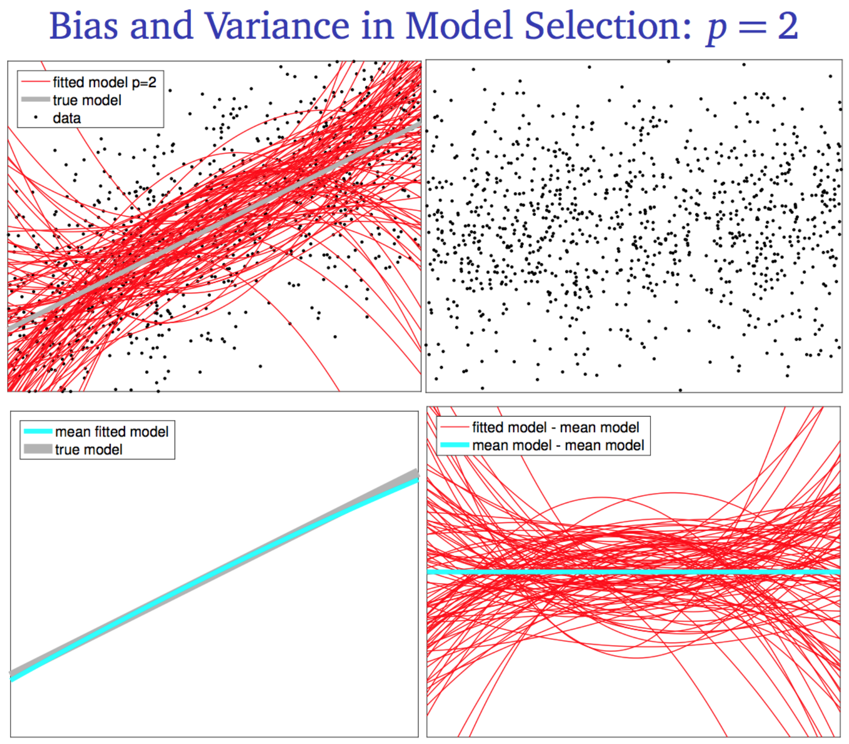




The bias is now very low bias for all $\mathbf{x}$'s. The variance is low for $\mathbf{x}$'s in the middle of the graph, but higher for **x**'s that are away from the center of the graph.

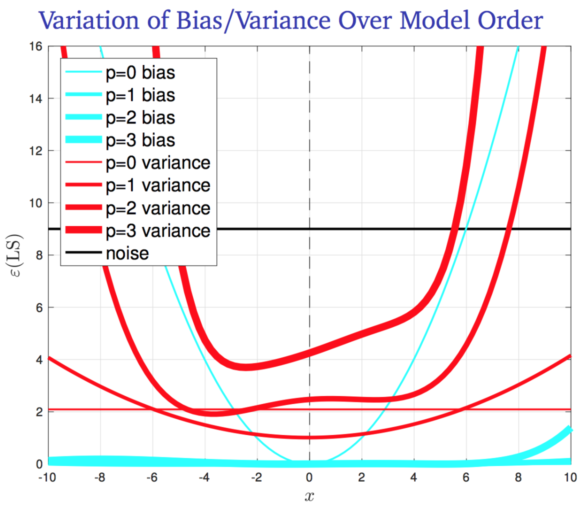
Finally, let's look at a 2nd degree (quadratic) regression model.

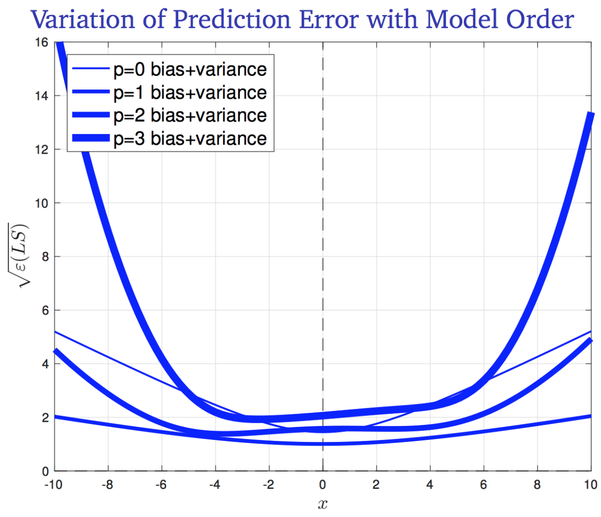


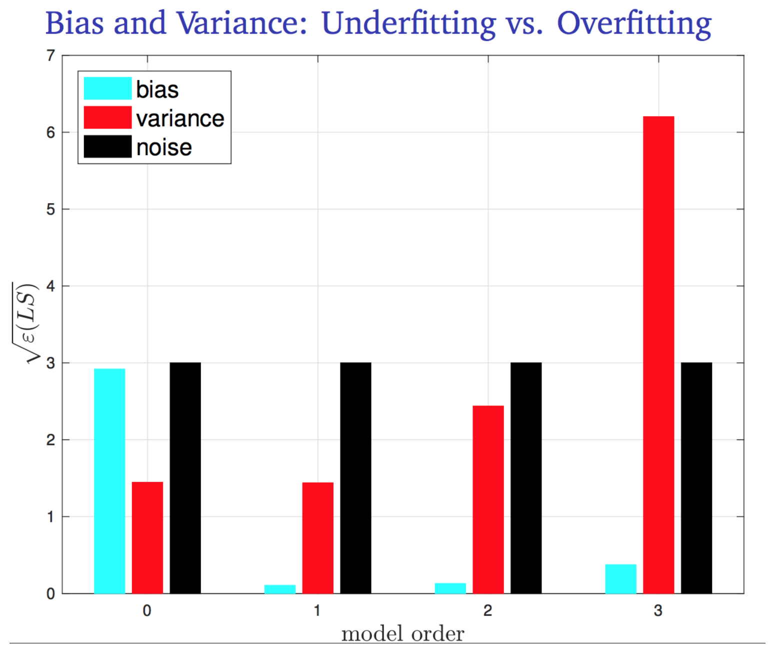


The bias is still very low for all **x**'s. However, the variance is much higher for all values of **x**.

Let's summarize our results. We find the bias and the variance empirically and graph them for all values of **x**, as shown in the first two graphs. Finally, we take an expectation over the bias and variance over all values of $\mathbf{x}$, as shown in the third graph.







The bias-variance decomposition confirms our understanding that the true model is linear. While a quadratic model achieves the same theoretical bias as a linear model, it overfits to the data, as indicated by its high variance. On the other hand a constant model underfits the data, as indicated by its high bias. In the process of training our model, we can tell that a constant model is a poor choice, because its high bias is reflected in poor training error. However we cannot tell that a quadratic model is poor, because its high variance is not reflected in the training error. This is the reason why we use validation data and cross-validation as a means to measure the performance of our hypothesis model on unseen data.

Takeaways

Let us conclude by stating some implications of the Bias-Variance Decomposition:

1. Underfitting is equivalent to high bias; most overfitting correlates to high variance.
2. Training error reflects bias but not variance. Test error reflects both. In practice, if the training error is much smaller than the test error, then there is overfitting.
3. Variance $\to 0$ as $n \to \infty$.
4. If f is in the set of hypothesis functions, bias will decrease with more data. If f is not in the set of hypothesis functions, then there is an underfitting problem and more data won't help.
5. Adding good features will decrease the bias, but adding a bad feature rarely increase the bias. (just set the coefficient of the feature to 0)
6. Adding a feature usually increase the variance, so a feature should only be added if it decreases bias more than it increases variance.
7. Irreducible error can not be reduced.
8. Noise in the test set only affects $\text{Var}(Z)$, but noise in the training set also affects bias and variance.
9. For real-world data, f is rarely known, and the noise model might be wrong, so we can't calculate bias and variance. But we can test algorithms over synthetic data.

2.3 Multivariate Gaussians

So far in our discussion of MLE and MAP in regression, we considered a set of Gaussian random variables $Z_1, Z_2, \dots, Z_k$, which can represent anything from the noise in data to the parameters of a model. One critical assumption we made is that these variables are independent and identically distributed. However, what about the case when these variables are dependent and/or non-identical? For example, in time series data we have the relationship

$$Z_{i+1} = rZ_i + U_i$$

where $U_i \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$ and $-1 \leq r \leq 1$ (so that it doesn't blow up)
Here's another example: consider the "sliding window" (like echo of audio)

$$Z_i = \mathbf{\Sigma} r_j U_{i-j}$$

where $U_i \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$
In general, if we can represent the random vector $\mathbf{Z} = (Z_1, Z_2, \dots, Z_k)$ as

$$\mathbf{Z} = \mathbf{RU}$$

where $\mathbf{Z} \in \mathbb{R}^n$, $\mathbf{R} \in \mathbb{R}^{n \times n}$, $\mathbf{U} \in \mathbb{R}^n$, and $U_i \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$, we refer to $\mathbf{Z}$ as a **Jointly Gaussian Random Vector**. Our goal now is to derive its probability density formula.

Definition

There are three equivalent definitions of a jointly Gaussian (JG) random vector:

1. A random vector $\mathbf{Z} = (Z_1, Z_2, \dots, Z_k)$ is JG if there exists a base random vector $\mathbf{U} = (U_1, U_2, \dots, U_l)$ whose components are independent standard normal random variables, a transition matrix $\mathbf{R} \in \mathbb{R}^{k \times l}$, and a mean vector $\boldsymbol{\mu} \in \mathbb{R}^k$, such that $\mathbf{Z} = \mathbf{RU} + \boldsymbol{\mu}$.

2. A random vector $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_k)^\top$ is JG if $\sum_{i=1}^{k} a_i Z_i$ is normally distributed for every $a = (a_1, a_2, \ldots, a_k)^\top \in \mathbb{R}^k$.

3. (Non-degenerate case only) A random vector $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_k)^\top$ is JG if

$$f_{\mathbf{Z}}(\mathbf{z}) = \frac{1}{\sqrt{|\det(\boldsymbol{\Sigma})|}} \frac{1}{(\sqrt{2\pi})^k} e^{-\frac{1}{2}(\mathbf{Z}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{Z}-\boldsymbol{\mu})}$$

   Where $\boldsymbol{\Sigma} = \mathbb{E}[(\mathbf{Z}-\boldsymbol{\mu})(\mathbf{Z}-\boldsymbol{\mu})^\top] = \mathbb{E}[(\mathbf{RU})(\mathbf{RU})^\top] = \mathbf{R}\mathbb{E}[\mathbf{U}\mathbf{U}^\top]\mathbf{R}^\top = \mathbf{R}I\mathbf{R}^\top = \mathbf{R}\mathbf{R}^\top$
   $\boldsymbol{\Sigma}$ is also called the **covariance matrix** of $\mathbf{Z}$.

Note that all of these conditions are equivalent. In this note we will start by showing a proof that (1) $\implies$ (3). We will leave it as an exercise to prove the rest of the implications needed to show that the three conditions are in fact equivalent.

Proving (1) $\implies$ (3)

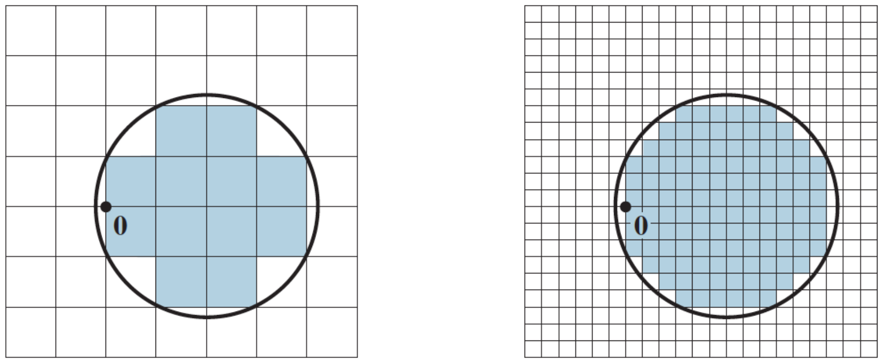
In the context of the noise problem we defined earlier, we are starting with condition (1), ie. $\mathbf{Z} = \mathbf{RU}$ (in this case $k = l = n$), and we would like to derive the probability density of $\mathbf{Z}$. Note that here we removed the $\boldsymbol{\mu}$ from consideration because in machine learning we always assume that the noise has a mean of 0. We leave it as an exercise for the reader to prove the case for an arbitrary $\boldsymbol{\mu}$.

We will first start by relating the probability density function of $\mathbf{U}$ to that of $\mathbf{Z}$. Denote $f_{\mathbf{U}}(\mathbf{u})$ as the probability density for $\mathbf{U} = \mathbf{u}$, and similarly denote $f_{\mathbf{Z}}(\mathbf{z})$ as the probability density for $\mathbf{Z} = \mathbf{z}$.

One may initially believe that $f_{\mathbf{U}}(\mathbf{u}) = f_{\mathbf{Z}}(\mathbf{Ru})$, but this is NOT true. Remember that since there is a change of variables from $\mathbf{U}$ to $\mathbf{Z}$, we must make sure to incorporate the change of variables constant, which in this case is the absolute value of the determinant of $\mathbf{R}$. Incorporating this constant, we will have the correct formula:

$$f_{\mathbf{U}}(\mathbf{u}) = |\det(\mathbf{R})| f_{\mathbf{Z}}(\mathbf{Ru})$$

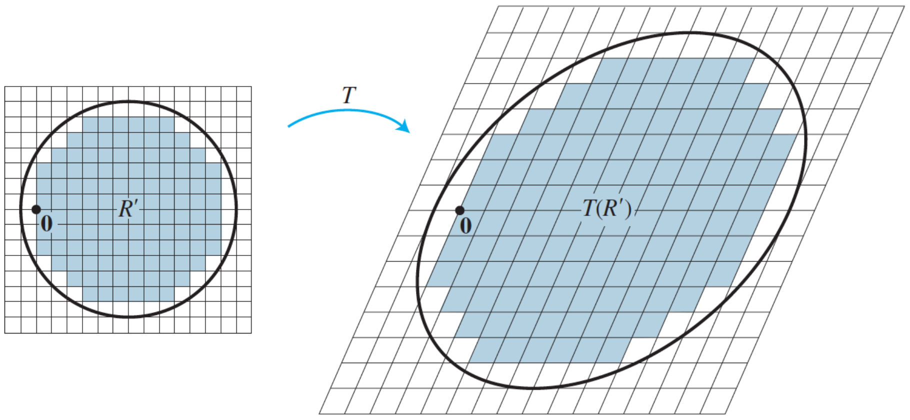
Let's see why this is true, with a simple 2D geometric explanation. Define $\mathbf{U}$ space to be the 2D space with axes U_1 and U_2. Now take any arbitrary region $\mathbf{R'}$ in $\mathbf{U}$ space (note that this $\mathbf{R'}$ is different from the matrix $\mathbf{R}$ that relates $\mathbf{U}$ to $\mathbf{Z}$). As shown in the diagram below, we have some off-centered circular region $\mathbf{R'}$ and we would like to approximate the probability that $\mathbf{U}$ takes a value in this region. We can do so by taking a Riemann sum of the density function $f_{\mathbf{U}}(.)$ over smaller and smaller squares that make up the region $\mathbf{R'}$:



Mathematically, we have that

$$P(\mathbf{U} \subseteq \mathbf{R}') = \iint_{\mathbf{R}'} f_{\mathbf{U}}(u_1, u_2)\, du_1\, du_2 \approx \sum\sum_{\mathbf{R}'} f_{\mathbf{U}}(u_1, u_2)\, \Delta u_1\, \Delta u_2$$

Now, let's apply the linear transformation $\mathbf{Z} = \mathbf{R}\mathbf{U}$, mapping the region $\mathbf{R}'$ in $\mathbf{U}$ space, to the region $T(\mathbf{R}')$ in $\mathbf{Z}$ space.



The graph on the right is now $\mathbf{Z}$ space, the 2D space with axes Z_1 and Z_2. Assuming that the matrix $\mathbf{R}$ is invertible, there is a one-to-one correspondence between points in $\mathbf{U}$ space to points in $\mathbf{Z}$ space. As we can note in the diagram above, each unit square in $\mathbf{U}$ space maps to a parallelogram in $\mathbf{Z}$ space (in higher dimensions, we would use the terms **hypercube** and **parallelepiped**). Recall the relationship between each unit hypercube and the parallelepiped it maps to:

$$\text{Area(parallelepiped)} = |\det(\mathbf{R})| \cdot \text{Area(hypercube)}$$

In this 2D example, if we denote the area of each unit square as $\Delta u_1 \Delta u_2$, and the area of each unit parallelepiped as ΔA, we say that

$$\Delta A = |\det(\mathbf{R})| \cdot \Delta u_1 \Delta u_2$$

Now let's take a Riemann sum to find the probability that $\mathbf{Z}$ takes a value in $T(\mathbf{R}')$:

$$\begin{aligned} P(\mathbf{Z} \subseteq T(\mathbf{R}')) &= \iint_{T(\mathbf{R}')} f_{\mathbf{Z}}(z_1, z_2)\, dz_1\, dz_2 \\ &\approx \sum\sum_{T(\mathbf{R}')} f_{\mathbf{Z}}(\mathbf{z})\, \Delta A \\ &= \sum\sum_{\mathbf{R}'} f_{\mathbf{Z}}(\mathbf{R}\mathbf{u})\, |\det(\mathbf{R})| \Delta u_1 \Delta u_2 \end{aligned}$$

Note the change of variables in the last step: we sum over the squares in $\mathbf{U}$ space, instead of parallelograms in $\mathbf{R}$ space.

So far, we have shown that (for any dimension n)

$$P(\mathbf{U} \subseteq \mathbf{R}') = \int \dots \iint_{\mathbf{R}'} f_{\mathbf{U}}(\mathbf{u})\, du_1 du_2 \dots du_n$$

and

$$P(\mathbf{Z} \subseteq T(\mathbf{R}')) = \int \dots \iint_{\mathbf{R}'} f_{\mathbf{Z}}(\mathbf{Ru}) \, |\det(\mathbf{R})| du_1 du_2 \dots du_n$$

Notice that these two probabilities are equivalent! The probability that $\mathbf{U}$ takes value in $\mathbf{R}'$ must equal the probability that the transformed random vector $\mathbf{Z}$ takes a value in the transformed region $T(\mathbf{R}')$.

Therefore, we can say that

$$\begin{aligned} P(\mathbf{U} \subseteq \mathbf{R}') &= \int \dots \iint_{\mathbf{R}'} f_{\mathbf{U}}(\mathbf{u}) \, du_1 du_2 \dots du_n \\ &= \int \dots \iint_{\mathbf{R}'} f_{\mathbf{Z}}(\mathbf{Ru}) \, |\det(\mathbf{R})| du_1 du_2 \dots du_n \\ &= P(\mathbf{Z} \subseteq T(\mathbf{R}')) \end{aligned}$$

We conclude that

$$f_{\mathbf{U}}(\mathbf{u}) = f_{\mathbf{Z}}(\mathbf{Ru}) \, |\det(\mathbf{R})|$$

An almost identical argument will allow us to state that

$$f_{\mathbf{Z}}(\mathbf{z}) = f_{\mathbf{U}}(\mathbf{R}^{-1}\mathbf{z}) \, |\det\Big(\mathbf{R}^{-1}\Big)| = \frac{1}{|\det(\mathbf{R})|} f_{\mathbf{U}}(\mathbf{R}^{-1}\mathbf{z})$$

Since the densities for all the U_i's are i.i.d, and $\mathbf{U} = \mathbf{R}^{-1}\mathbf{Z}$, we can write the joint density function of Z as

$$\begin{aligned} f_{\mathbf{Z}}(\mathbf{z}) &= \frac{1}{|\det(\mathbf{R})|} f_{\mathbf{U}}(\mathbf{R}^{-1}\mathbf{z}) \\ &= \frac{1}{|\det(\mathbf{R})|} \prod_{i=1}^{n} f_{U_i}((\mathbf{R}^{-1}\mathbf{z})_i) \\ &= \frac{1}{|\det(\mathbf{R})|} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}(\mathbf{R}^{-1}\mathbf{z})^\top (\mathbf{R}^{-1}\mathbf{z})} \\ &= \frac{1}{|\det(\mathbf{R})|} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\mathbf{z}^\top \mathbf{R}^{-T} \mathbf{R}^{-1}\mathbf{z}} \\ &= \frac{1}{|\det(\mathbf{R})|} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\mathbf{z}^\top (\mathbf{R}\mathbf{R}^\top)^{-1}\mathbf{z}} \end{aligned}$$

Note that $(\mathbf{R}\mathbf{R}^\top)^{-1}$ is simply the covariance matrix for $\mathbf{Z}$:

$$\text{Cov}[\mathbf{Z}] = \mathbb{E}[\mathbf{Z}\mathbf{Z}^\top] = \mathbb{E}[\mathbf{R}\mathbf{U}\mathbf{U}^\top\mathbf{R}^\top] = \mathbf{R}\mathbb{E}[\mathbf{U}\mathbf{U}^\top]\mathbf{R}^\top = \mathbf{R}\mathbf{I}\mathbf{R}^\top = \mathbf{R}\mathbf{R}^\top$$

Thus the density function of $\mathbf{Z}$ can be written as

$$f_{\mathbf{Z}}(\mathbf{z}) = \frac{1}{|\det(\mathbf{R})|} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\mathbf{z}^\top \mathbf{\Sigma}_Z^{-1}\mathbf{z}}$$

Furthermore, we know that

$$\begin{aligned} |\det(\mathbf{\Sigma}_Z)| &= |\det\Big(\mathbf{R}\mathbf{R}^\top\Big)| \\ &= |\det(\mathbf{R}) \cdot \det\Big(\mathbf{R}^\top\Big)| \end{aligned}$$

$$= |\det(\mathbf{R}) \cdot \det(\mathbf{R})| = |\det(\mathbf{R})|^2$$

and therefore

$$f_{\mathbf{Z}}(\mathbf{z}) = \frac{1}{\sqrt{\det(\mathbf{\Sigma}_Z)}} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}\mathbf{z}^\top \mathbf{\Sigma}_Z^{-1} \mathbf{z}}$$

Estimating Gaussians from Data

For a particular multivariate Gaussian distribution $f(.)$, if we do not have the true means and covariances $\boldsymbol{\mu}, \mathbf{\Sigma}$, then our best bet is to use MLE to estimate them empirically with i.i.d. samples $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$:

$$\hat{\boldsymbol{\mu}} = \frac{1}{n} \sum_{t_i = k} \mathbf{x}_i$$

$$\hat{\mathbf{\Sigma}} = \frac{1}{n} \sum_{t_i = k} (\mathbf{x}_i - \hat{\boldsymbol{\mu}})(\mathbf{x}_i - \hat{\boldsymbol{\mu}})^T$$

Note that the above formulas are not necessarily trivial and must be formally proven using MLE. Just to present a glimpse of the process, let's prove that these formulas hold for the case where we are dealing with 1-d data points. For notation purposes, assume that $\mathcal{D} = \{x_1, x_2, \dots, x_n\}$ is the set of all training data points that belong to class k. Note that the data points are i.i.d. Our goal is to solve the following MLE problem:

$$\begin{aligned}
\hat{\mu}, \hat{\sigma}^2 &= \arg\max_{\mu, \sigma^2} P(x_1, x_2, ..., x_n \mid \mu, \sigma^2) \\
&= \arg\max_{\mu, \sigma^2} \ln\Big(P(x_1, x_2, ..., x_n \mid \mu, \sigma^2)\Big) \\
&= \arg\max_{\mu, \sigma^2} \sum_{i=1}^{n} \ln\Big(P(x_i \mid \mu, \sigma^2)\Big) \\
&= \arg\max_{\mu, \sigma^2} \sum_{i=1}^{n} -\frac{(x_i - \mu)^2}{2\sigma^2} - \ln(\sigma) - \frac{1}{2}\ln(2\pi) \\
&= \arg\min_{\mu, \sigma^2} \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma)
\end{aligned}$$

Note that the objective above is not jointly convex, so we cannot simply take derivatives and set them to 0! Instead, we decompose the minimization over σ^2 and μ into a nested optimization problem:

$$\min_{\mu, \sigma^2} \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma) = \min_{\sigma^2} \min_{\mu} \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma)$$

The optimization problem has been decomposed into an inner problem that optimizes for μ given a fixed σ^2, and an outer problem that optimizes for σ^2 given the optimal value $\hat{\mu}$. Let's first solve the inner optimization problem. Given a fixed σ^2, the objective is convex in μ, so we can simply take a partial derivative w.r.t μ and set it equal to 0:

$$\frac{\partial}{\partial \mu}\Big(\sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma)\Big) = \sum_{i=1}^{n} \frac{-(x_i - \mu)}{\sigma^2} = 0 \implies \hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Having solved the inner optimization problem, we now have that

$$\min_{\sigma^2} \min_{\mu} \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma) = \min_{\sigma^2} \sum_{i=1}^{n} \frac{(x_i - \hat{\mu})^2}{2\sigma^2} + \ln(\sigma)$$

Note that this objective is not convex in σ, so we must instead find the critical point of the objective that minimizes the objective. Assuming that $\sigma \geq 0$, the critical points are:

- $\sigma = 0$: assuming that not all of the points x_i are equal to $\hat{\mu}$, there are two terms that are at odds with each other: a $1/\sigma^2$ term that blows off to ∞, and a $\ln(\sigma)$ term that blows off to $-\infty$ as $\sigma \to 0$. Note that the $1/\sigma^2$ term blows off at a faster rate, so we conclude that

$$\lim_{\sigma \to 0} \sum_{i=1}^{n} \frac{(x_i - \hat{\mu})^2}{2\sigma^2} + \ln(\sigma) = \infty$$

- $\sigma = \infty$: this case does not lead to the solution, because it gives a maximum, not a minimum.

$$\lim_{\sigma \to \infty} \sum_{i=1}^{n} \frac{(x_i - \hat{\mu})^2}{2\sigma^2} + \ln(\sigma) = \infty$$

- Points at which the derivative w.r.t σ is 0

$$\frac{\partial}{\partial \sigma}\Big(\sum_{i=1}^{n} \frac{(x_i - \hat{\mu})^2}{2\sigma^2} + \ln(\sigma)\Big) = \sum_{i=1}^{n} -\frac{(x_i - \hat{\mu})^2}{\sigma^3} + \frac{1}{\sigma} = 0 \implies \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$$

We conclude that the optimal point is

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$$

Isocontours

Let's try to understand in detail how to visualize a multivariate Gaussian distribution. For simplicity, let's consider a zero-mean Gaussian distribution $\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$, which just leaves us with the covariance matrix $\mathbf{\Sigma}$. Since $\mathbf{\Sigma}$ is a symmetric, positive semidefinite matrix, we can decompose it by the spectral theorem into $\mathbf{\Sigma} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$, where the columns of $\mathbf{V}$ form an orthonormal basis in $\mathbb{R}^d$, and $\mathbf{\Lambda}$ is a diagonal matrix with real, non-negative values. We wish to find its **level set**

$$f(\mathbf{x}) = k$$

or simply the set of all points $\mathbf{x}$ such that the probability density $f(\mathbf{x})$ evaluates to a fixed constant k. This is equivalent to the level set $\ln\big(f(\mathbf{x})\big) = \ln(k)$ which further reduces to

$$\mathbf{x}^T \mathbf{\Sigma}^{-1} \mathbf{x} = c$$

for some constant c. Without loss of generality, assume that this constant is 1. The level set $\mathbf{x}^T \mathbf{\Sigma}^{-1} \mathbf{x} = 1$ is an ellipsoid with axes $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d$, with lengths $\sqrt{\lambda_1}, \sqrt{\lambda_2}, \ldots, \sqrt{\lambda_d}$, respectively. Each axis of the ellipsoid is the vector $\sqrt{\lambda_i}\mathbf{v}_i$, and we can verify that

$$(\sqrt{\lambda_i}\mathbf{v}_i)^T \mathbf{\Sigma}^{-1} (\sqrt{\lambda_i}\mathbf{v}_i) = \lambda_i \mathbf{v}_i^T \mathbf{\Sigma}^{-1} \mathbf{v}_i = \lambda_i \mathbf{v}_i^T (\mathbf{\Sigma}^{-1} \mathbf{v}_i) = \lambda_i \mathbf{v}_i^T (\lambda_i^{-1} \mathbf{v}_i) = \mathbf{v}_i^T \mathbf{v}_i = 1$$

The entries of $\mathbf{\Lambda}$ dictate how elongated or shrunk the distribution is along each direction. In the case of **isotropic** distributions, the entries of $\mathbf{\Lambda}$ are all identical, meaning the the axes of the ellipsoid form a circle. In the case of **anisotropic** distributions, the entries of $\mathbf{\Lambda}$ are not necessarily identical, meaning that the resulting ellipsoid may be elongated/shrunken and also rotated.

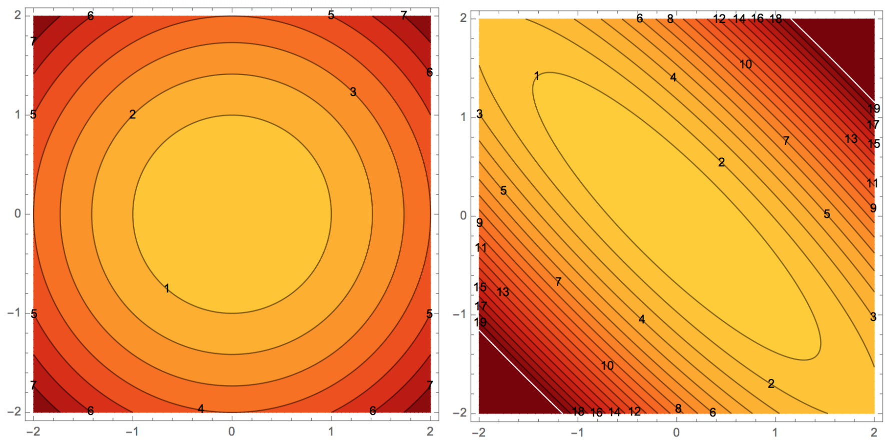


Figure 2.1: Isotropic (left) vs Anisotropic (right) contours are ellipsoids with axes $\sqrt{\lambda_i}\mathbf{v}_i$. Images courtesy Professor Shewchuk's notes

Properties

Let's state some well-known properties of Multivariate Gaussians. Given a JG random vector $\mathbf{Z} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{Z}, \mathbf{\Sigma}_\mathbf{Z})$, the linear transformation $\mathbf{AZ}$ (where $\mathbf{A}$ is an appropriately dimensioned constant matrix) is also JG:

$$\mathbf{AZ} \sim \mathcal{N}(\mathbf{A}\boldsymbol{\mu}_\mathbf{Z}, \mathbf{A}\mathbf{\Sigma}_\mathbf{Z}\mathbf{A}^\top)$$

We can derive the mean and covariance of $\mathbf{AZ}$ using the linearity of expectations:

$$\boldsymbol{\mu}_\mathbf{AZ} = \mathbb{E}[\mathbf{AZ}] = \mathbf{A}\mathbb{E}[\mathbf{Z}] = \mathbf{A}\boldsymbol{\mu}_\mathbf{Z}$$

and

$$\begin{aligned}\mathbf{\Sigma}_\mathbf{AZ} &= \mathbb{E}[(\mathbf{AZ} - \mathbb{E}[\mathbf{AZ}])(\mathbf{AZ} - \mathbb{E}[\mathbf{AZ}])^\top] \\ &= \mathbb{E}[\mathbf{A}(\mathbf{Z} - \mathbb{E}[\mathbf{Z}])(\mathbf{Z} - \mathbb{E}[\mathbf{Z}])^\top\mathbf{A}^\top] \\ &= \mathbf{A}\mathbb{E}[(\mathbf{Z} - \mathbb{E}[\mathbf{Z}])(\mathbf{Z} - \mathbb{E}[\mathbf{Z}])^\top]\mathbf{A}^\top \\ &= \mathbf{A}\mathbf{\Sigma}_\mathbf{Z}\mathbf{A}^\top\end{aligned}$$

Note that the statements above did not rely on the fact that $\mathbf{Z}$ is JG, so this reasoning applies to all random vectors. We know that $\mathbf{AZ}$ is JG itself, because it can be expressed as a linear transformation of i.i.d. Gaussians: $\mathbf{AZ} = \mathbf{ARU}$.

Now suppose that we have the partition $\mathbf{Z} = \begin{bmatrix}\mathbf{X}\\ \mathbf{Y}\end{bmatrix}$ whose distribution is given by $\mathbf{Z} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{Z}, \mathbf{\Sigma}_\mathbf{Z})$ and

$$\boldsymbol{\mu}_\mathbf{Z} = \begin{bmatrix}\boldsymbol{\mu}_\mathbf{X}\\ \boldsymbol{\mu}_\mathbf{Y}\end{bmatrix}, \mathbf{\Sigma}_\mathbf{Z} = \begin{bmatrix}\mathbf{\Sigma}_\mathbf{XX} & \mathbf{\Sigma}_\mathbf{XY}\\ \mathbf{\Sigma}_\mathbf{YX} & \mathbf{\Sigma}_\mathbf{YY}\end{bmatrix}$$

It turns out that the **marginal distribution** of the individual random vector $\mathbf{X}$ (and $\mathbf{Y}$) is JG:

$$\mathbf{X} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{X}, \mathbf{\Sigma}_\mathbf{XX})$$

However, the converse is not necessarily true: if $\mathbf{X}$ and $\mathbf{Y}$ are each individually JG, it is not necessarily the case that $\begin{bmatrix}\mathbf{X}\\ \mathbf{Y}\end{bmatrix}$ is JG! To see why, let's suppose that $\mathbf{X}$ and $\mathbf{Y}$ are individually JG. Thus, we can express each as a linear transformation of i.i.d. Gaussian random variables:

$$\mathbf{X} = \mathbf{R_X U_X}, \mathbf{Y} = \mathbf{R_Y U_Y}$$

we would expect that the expression for the joint distribution would be JG:

$$\begin{bmatrix}\mathbf{X}\\ \mathbf{Y}\end{bmatrix} = \begin{bmatrix}\mathbf{R_X} & \mathbf{0}\\ \mathbf{0} & \mathbf{R_Y}\end{bmatrix}\begin{bmatrix}\mathbf{U_X}\\ \mathbf{U_Y}\end{bmatrix}$$

However, since we cannot guarantee that the entries of $\mathbf{U_X}$ are independently distributed from the entries of $\mathbf{U_Y}$, we cannot conclude that the joint distribution is JG. If the entries are independently distributed, then we would be able to conclude that the joint distribution is JG.

Let's now transition back to our discussion of $\mathbf{Z}$. The **conditional distribution** of $\mathbf{X}$ given $\mathbf{Y}$ (and vice versa) is also JG:

$$\mathbf{X}|\mathbf{Y} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{X} + \boldsymbol{\Sigma}_\mathbf{XY}\boldsymbol{\Sigma}_\mathbf{YY}^{-1}(\mathbf{Y} - \boldsymbol{\mu}_\mathbf{Y}), \boldsymbol{\Sigma}_\mathbf{XX} - \boldsymbol{\Sigma}_\mathbf{XY}\boldsymbol{\Sigma}_\mathbf{YY}^{-1}\boldsymbol{\Sigma}_\mathbf{YX})$$

If $\mathbf{X}$ and $\mathbf{Y}$ are uncorrelated (that is, if $\boldsymbol{\Sigma}_\mathbf{XY} = \boldsymbol{\Sigma}_\mathbf{YX} = \mathbf{0}$), we can say that they are independent. Namely, the conditional distribution of $\mathbf{X}$ given $\mathbf{Y}$ does not depend on $\mathbf{Y}$:

$$\mathbf{X}|\mathbf{Y} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{X} + \mathbf{0}\boldsymbol{\Sigma}_\mathbf{YY}^{-1}(\mathbf{Y} - \boldsymbol{\mu}_\mathbf{Y}), \boldsymbol{\Sigma}_\mathbf{XX} - \mathbf{0}\boldsymbol{\Sigma}_\mathbf{YY}^{-1}\mathbf{0}) = \mathcal{N}(\boldsymbol{\mu}_\mathbf{X}, \boldsymbol{\Sigma}_\mathbf{XX})$$

This also follows from the multivariate Gaussian pdf:

$$\begin{aligned}
f_\mathbf{Z}\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} &= \frac{1}{(\sqrt{2\pi})^n}\begin{vmatrix}\boldsymbol{\Sigma}_\mathbf{XX} & \mathbf{0}\\ \mathbf{0} & \boldsymbol{\Sigma}_\mathbf{YY}\end{vmatrix}^{-\frac{1}{2}} \exp\left(-\frac{1}{2}\begin{bmatrix}\mathbf{x} & \mathbf{y}\end{bmatrix}^\top \begin{bmatrix}\boldsymbol{\Sigma}_\mathbf{XX} & \mathbf{0}\\ \mathbf{0} & \boldsymbol{\Sigma}_\mathbf{YY}\end{bmatrix}^{-1}\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix}\right)\\
&= \frac{1}{(\sqrt{2\pi})^{n_x}}\left|\boldsymbol{\Sigma}_\mathbf{XX}\right|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}\mathbf{x}^\top\boldsymbol{\Sigma}_\mathbf{XX}^{-1}\mathbf{x}\right)\cdot\frac{1}{(\sqrt{2\pi})^{n_y}}\left|\boldsymbol{\Sigma}_\mathbf{YY}\right|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}\mathbf{y}^\top\boldsymbol{\Sigma}_\mathbf{YY}^{-1}\mathbf{y}\right)\\
&= f_\mathbf{X}(\mathbf{x})\cdot f_\mathbf{Y}(\mathbf{y})
\end{aligned}$$

Note the significance of this statement. Given any two general random vectors, we cannot necessarily say "if they are uncorrelated, then they are independent". However in the case of random vectors from the same JG joint distribution, we can make this claim.

2.4 MLE and MAP for Regression (Part II)

The power of probabilistic thinking is that it allows us a way to model situations that arise and adapt our approaches in a reasonably principled way. This is particularly true when it comes to incorporating information about the situation that comes from the physical context of the data gathering process. In this note, we will explore what happens as we vary our assumptions about the noise in our data and the priors for our parameters, as well as the "importance" of certain training points.

So far we have used MLE and MAP to justify the optimization formulation of OLS and ridge regression, respectively. The MLE formulation assumes that the observation Y_i is a noisy version of the true underlying output:

$$Y_i = f(\mathbf{x}_i) + Z_i$$

where the noise for each datapoint is crucially i.i.d. The MAP formulation assumes that the model parameter W_j is according to an i.i.d. Gaussian prior

$$W_j \overset{\text{iid}}{\sim} \mathcal{N}(\mu_j, \sigma_h^2)$$

.

So far, we have restricted ourselves to the case when the noise/parameters are i.i.d:

$$\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I}), \quad \mathbf{W} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{W}, \sigma_h^2\mathbf{I})$$

However, what about the case when N_i's/W_j's are non-identical or dependent on one another? We would like to explore the case when the observation noise and underlying parameters are jointly Gaussian with arbitrary individual covariance matrices, but are independent of each other.

$$\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \Sigma_\mathbf{Z}), \quad \mathbf{W} \sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{W}, \boldsymbol{\Sigma}_\mathbf{W})$$

It turns out that via a change of coordinates, we can reduce these non-i.i.d. problems back to the i.i.d. case and solve them using the original techniques we used to solve OLS and Ridge Regression! Changing coordinates is a powerful tool in thinking about machine learning.

Weighted Least Squares

The basic idea of **weighted least squares** is the following: we place more emphasis on the loss contributed from certain data points over others - that is, we care more about fitting some data points over others. It turns out that this weighted perspective is very useful as a building block when we go beyond traditional least-squares problems.

Optimization View

From an optimization perspective, the problem can be expressed as

$$\hat{\mathbf{w}}_{\text{WLS}} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} \left(\sum_{i=1}^{n} \omega_i (y_i - \mathbf{x}_i{}^\top \mathbf{w})^2 \right)$$

This objective is the same as OLS, except that each term in the sum is weighted by a positive coefficient ω_i. As always, we can vectorize this problem:

$$\hat{\mathbf{w}}_{\text{WLS}} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y} - \mathbf{X}\mathbf{w})^\top \boldsymbol{\Omega} (\mathbf{y} - \mathbf{X}\mathbf{w})$$

Where the i'th row $\mathbf{X}$ is $\mathbf{x}_i{}^\top$, and $\boldsymbol{\Omega} \in \mathbb{R}^{n\times n}$ is a diagonal matrix with $\boldsymbol{\Omega}_{i,i} = \omega_i$.

We rewrite the WLS objective to an OLS objective:

$$\begin{aligned}
\hat{\mathbf{w}}_{\text{WLS}} &= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y} - \mathbf{X}\mathbf{w})^\top \boldsymbol{\Omega} (\mathbf{y} - \mathbf{X}\mathbf{w}) \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y} - \mathbf{X}\mathbf{w})^\top \boldsymbol{\Omega}^{1/2}\boldsymbol{\Omega}^{1/2} (\mathbf{y} - \mathbf{X}\mathbf{w}) \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\boldsymbol{\Omega}^{1/2}\mathbf{y} - \boldsymbol{\Omega}^{1/2}\mathbf{X}\mathbf{w})^\top (\boldsymbol{\Omega}^{1/2}\mathbf{y} - \boldsymbol{\Omega}^{1/2}\mathbf{X}\mathbf{w}) \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} \|\boldsymbol{\Omega}^{1/2}\mathbf{y} - \boldsymbol{\Omega}^{1/2}\mathbf{X}\mathbf{w}\|^2
\end{aligned}$$

This formulation is identical to OLS except that we have scaled the data matrix and the observation vector by $\mathbf{\Omega}^{1/2}$, and we conclude that

$$\hat{\mathbf{w}}_{\text{WLS}} = \left((\mathbf{\Omega}^{1/2}\mathbf{X})^\top(\mathbf{\Omega}^{1/2}\mathbf{X})\right)^{-1}\left(\mathbf{\Omega}^{1/2}\mathbf{X}\right)^\top\mathbf{\Omega}^{1/2}\mathbf{y} = (\mathbf{X}^\top\mathbf{\Omega}\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{\Omega}\mathbf{y}$$

Probabilistic View

As in MLE, we assume that our observations $\mathbf{y}$ are noisy, but now suppose that some of the y_i's are more noisy than others. How can we take this into account in our learning algorithm so we can get a better estimate of the weights? Our probabilistic model looks like

$$Y_i = \mathbf{x}_i{}^\top\mathbf{w} + Z_i$$

where the Z_i's are still independent Gaussians random variables, but not necessarily identical: $Z_i \sim \mathcal{N}(0, \sigma_i^2)$. Jointly, we have that $\mathbf{Z} \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{Z}}, \mathbf{\Sigma}_{\mathbf{Z}})$, where

$$\mathbf{\Sigma}_{\mathbf{Z}} = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \sigma_n^2 \end{bmatrix}$$

We can morph the problem into an MLE one by scaling the data to make sure all the Z_i's are identically distributed, by dividing by σ_i:

$$\frac{Y_i}{\sigma_i} = \frac{\mathbf{x}_i{}^\top}{\sigma_i}\mathbf{w} + \frac{Z_i}{\sigma_i}$$

Note that the scaled noise entries are now i.i.d:

$$\frac{Z_i}{\sigma_i} \overset{\text{iid}}{\sim} \mathcal{N}(0,1)$$

Jointly, we can express this change of coordinates as

$$\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{y} \sim \mathcal{N}(\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{X}\mathbf{w}, \mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{\Sigma}_{\mathbf{Z}}\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{\top}{2}}) = \mathcal{N}(\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{X}\mathbf{w}, \mathbf{I})$$

This change of variable is sometimes called the **reparameterization trick**. Now that the noise is i.i.d. using the change of coordinates, we rewrite our original problem as a scaled MLE problem:

$$\begin{aligned}\hat{\mathbf{w}}_{\text{WLS}} &= \arg\min_{\mathbf{w}\in\mathbb{R}^d}\left(\sum_{i=1}^n \frac{(\frac{y_i}{\sigma_i} - \frac{\mathbf{x}_i{}^\top}{\sigma_i}\mathbf{w})^2}{2}\right) + n\log\sqrt{2\pi} \\ &= \arg\min_{\mathbf{w}\in\mathbb{R}^d}\sum_{i=1}^n \frac{1}{\sigma_i^2}(y_i - \mathbf{x}_i{}^\top\mathbf{w})^2\end{aligned}$$

The MLE estimate of this scaled problem is equivalent to the WLS estimate of the original problem:

$$\hat{\mathbf{w}}_{\text{WLS}} = (\mathbf{X}^\top\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{\Sigma}_{\mathbf{Z}}^{-\frac{1}{2}}\mathbf{y} = (\mathbf{X}^\top\mathbf{\Sigma}_{\mathbf{Z}}^{-1}\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{\Sigma}_{\mathbf{Z}}^{-1}\mathbf{y}$$

As long as no σ is 0, $\mathbf{\Sigma}_{\mathbf{Z}}$ is invertible. Note that ω_i from the optimization perspective is directly related to σ_i^2 from the probabilistic perspective: $\omega_i = \frac{1}{\sigma_i^2}$. Or at the level of matrices, $\mathbf{\Omega} = \mathbf{\Sigma}_{\mathbf{Z}}{}^{-1}$. As the variance σ_i^2 of the noise corresponding to data point i decreases, the weight ω_i increases: we are more concerned about fitting data point i because it is likely to match the true underlying denoised point. Inversely, as the variance σ_i^2 increases, the weight ω_i decreases: we are less concerned about fitting data point i because it is noisy and should not be trusted.

Generalized Least Squares

Now let's consider the case when the noise random variables are dependent on one another. We have

$$\mathbf{Y} = \mathbf{X}\mathbf{w} + \mathbf{Z}$$

where $\mathbf{Z}$ is now a jointly Gaussian random vector. That is,

$$\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma_Z}), \quad \mathbf{Y} \sim \mathcal{N}(\mathbf{X}\mathbf{w}, \mathbf{\Sigma_Z})$$

This problem is known as **generalized least squares**. Our goal is to maximize the probability of our data over the set of possible $\mathbf{w}$'s:

$$\begin{aligned}\hat{\mathbf{w}}_{\text{GLS}} &= \arg\max_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{\sqrt{\det(\mathbf{\Sigma_Z})}} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}(\mathbf{y}-\mathbf{X}\mathbf{w})^\top \mathbf{\Sigma_Z}^{-1}(\mathbf{y}-\mathbf{X}\mathbf{w})} \\ &= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y}-\mathbf{X}\mathbf{w})^\top \mathbf{\Sigma_Z}^{-1}(\mathbf{y}-\mathbf{X}\mathbf{w})\end{aligned}$$

The optimization problem is therefore given by

$$\hat{\mathbf{w}}_{\text{GLS}} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y}-\mathbf{X}\mathbf{w})^\top \mathbf{\Sigma_Z}^{-1}(\mathbf{y}-\mathbf{X}\mathbf{w})$$

Since $\mathbf{\Sigma_Z}$ is symmetric, we can decompose it into its eigen structure using the spectral theorem:

$$\mathbf{\Sigma_Z} = \mathbf{Q} \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \sigma_n^2 \end{bmatrix} \mathbf{Q}^\top$$

where $\mathbf{Q}$ is orthonormal. As before with weighted least squares, our goal is to find an appropriate linear transformation so that we can reduce the problem into the i.i.d. case.

Consider

$$\mathbf{\Sigma_Z}^{-\frac{1}{2}} = \mathbf{Q} \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \frac{1}{\sigma_n} \end{bmatrix} \mathbf{Q}^\top$$

We can scale the data to morph the problem into an MLE problem with i.i.d. noise variables, by premultiplying the data matrix $\mathbf{X}$ and the observation vector $\mathbf{y}$ by $\mathbf{\Sigma_Z}^{-\frac{1}{2}}$. Jointly, we can express this change of coordinates as

$$\mathbf{\Sigma_Z}^{-\frac{1}{2}}\mathbf{y} \sim \mathcal{N}(\mathbf{\Sigma_Z}^{-\frac{1}{2}}\mathbf{X}\mathbf{w}, \mathbf{\Sigma_Z}^{-\frac{1}{2}}\mathbf{\Sigma_Z}\mathbf{\Sigma_Z}^{-\frac{\top}{2}}) = \mathcal{N}(\mathbf{\Sigma_Z}^{-\frac{1}{2}}\mathbf{X}\mathbf{w}, \mathbf{I}).$$

Consequently, in a very similar fashion to the independent noise problem, the MLE of the scaled dependent noise problem is

$$\hat{\mathbf{w}}_{\text{GLS}} = (\mathbf{X}^\top \mathbf{\Sigma_Z}^{-1}\mathbf{X})^{-1}\mathbf{X}^\top \mathbf{\Sigma_Z}^{-1}\mathbf{y}.$$

"Ridge Regression" with Dependent Parameters

In the ordinary least squares (OLS) statistical model, we assume that the output $\mathbf{Y}$ is a linear function of the input, plus some Gaussian noise. We take this one step further in MAP estimation, where we assume that the weights are a random variable. The new statistical model is

$$\mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{Z}$$

where $\mathbf{Y}$ and $\mathbf{Z}$ are n-dimensional random vectors, $\mathbf{W}$ is a d-dimensional random vector, and $\mathbf{X}$ is a fixed $n \times d$ matrix. Note that random vectors are not notationally distinguished from matrices here, so keep in mind what each symbol represents.

We have seen that ridge regression can be derived by assuming a prior distribution on $\mathbf{W}$ in which W_i are i.i.d. (univariate) Gaussian, or equivalently,

$$\mathbf{W} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

But more generally, we can allow $\mathbf{W}$ to be any multivariate Gaussian:

$$\mathbf{W} \sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{W}}, \boldsymbol{\Sigma}_{\mathbf{W}})$$

Recall that we can rewrite a multivariate Gaussian variable as an affine transformation of a standard Gaussian variable:

$$\mathbf{W} = \boldsymbol{\Sigma}_{\mathbf{W}}^{1/2}\mathbf{V} + \boldsymbol{\mu}_{\mathbf{W}}, \quad \mathbf{V} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

Plugging this parameterization into our previous statistical model gives

$$\mathbf{Y} = \mathbf{X}(\boldsymbol{\Sigma}_{\mathbf{W}}^{1/2}\mathbf{V} + \boldsymbol{\mu}_{\mathbf{W}}) + \mathbf{Z}$$

But this can be re-written

$$\mathbf{Y} - \mathbf{X}\boldsymbol{\mu}_{\mathbf{W}} = \mathbf{X}\boldsymbol{\Sigma}_{\mathbf{W}}^{1/2}\mathbf{V} + \mathbf{Z}$$

which we see has the form of the statistical problem that underlies traditional Ridge Regression with $\lambda = 1$, and therefore

$$\hat{\mathbf{v}} = (\boldsymbol{\Sigma}_{\mathbf{W}}^{\top/2}\mathbf{X}^\top\mathbf{X}\boldsymbol{\Sigma}_{\mathbf{W}}^{1/2} + \mathbf{I})^{-1}\boldsymbol{\Sigma}_{\mathbf{W}}^{\top/2}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_{\mathbf{W}})$$

However $\mathbf{V}$ is not what we care about – we need to convert back to the actual weights $\mathbf{W}$ in order to make predictions. Since $\mathbf{W}$ is completely determined by $\mathbf{V}$ (assuming fixed mean and covariance),

$$\begin{aligned}
\hat{\mathbf{w}} &= \boldsymbol{\Sigma}_{\mathbf{W}}^{1/2}\hat{\mathbf{v}} + \boldsymbol{\mu}_{\mathbf{W}} \\
&= \boldsymbol{\mu}_{\mathbf{W}} + \boldsymbol{\Sigma}_{\mathbf{W}}^{1/2}(\boldsymbol{\Sigma}_{\mathbf{W}}^{\top/2}\mathbf{X}^\top\mathbf{X}\boldsymbol{\Sigma}_{\mathbf{W}}^{1/2} + \mathbf{I})^{-1}\boldsymbol{\Sigma}_{\mathbf{W}}^{\top/2}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_{\mathbf{W}}) \\
&= \boldsymbol{\mu}_{\mathbf{W}} + (\mathbf{X}^\top\mathbf{X} + \underbrace{\boldsymbol{\Sigma}_{\mathbf{W}}^{-\top/2}\boldsymbol{\Sigma}_{\mathbf{W}}^{-1/2}}_{\boldsymbol{\Sigma}_{\mathbf{W}}^{-1}})^{-1}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_{\mathbf{W}}) \\
&= \boldsymbol{\mu}_{\mathbf{W}} + (\mathbf{X}^\top\mathbf{X} + \boldsymbol{\Sigma}_{\mathbf{W}}^{-1})^{-1}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_{\mathbf{W}})
\end{aligned}$$

Note that there are two terms: the prior mean $\boldsymbol{\mu}_{\mathbf{W}}$, plus another term that depends on both the data and the prior. The positive-definite precision matrix of $\mathbf{W}$'s prior ($\boldsymbol{\Sigma}_{\mathbf{W}}^{-1}$) controls how the data fit error affects our estimate. This is called Tikhonov regularization in the literature and generalizes ridge regularization.

To gain intuition, let us consider the simplified case where

$$\mathbf{\Sigma_W} = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d^2 \end{bmatrix}$$

When the prior variance σ_j^2 for dimension j is large, the prior is telling us that W_j may take on a wide range of values. Thus we do not want to penalize that dimension as much, preferring to let the data fit sort it out. And indeed the corresponding entry in $\mathbf{\Sigma_W^{-1}}$ will be small, as desired.

Conversely if σ_j^2 is small, there is little variance in the value of W_j, so $W_j \approx \mu_j$. As such we penalize the magnitude of the data-fit contribution to $\hat{W}_j$ more heavily.

If all the σ_j^2 are the same, then we have traditional ridge regularization.

Alternative derivation: directly conditioning jointly Gaussian random variables

In an explicitly probabilistic perspective, MAP with colored noise (and known $\mathbf{X}$) can be expressed as:

$$\mathbf{U}, \mathbf{V} \overset{\text{iid}}{\sim} \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{2.6}$$

$$\begin{bmatrix} \mathbf{Y} \\ \mathbf{W} \end{bmatrix} = \begin{bmatrix} \mathbf{R_Z} & \mathbf{XR_W} \\ \mathbf{0} & \mathbf{R_W} \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{V} \end{bmatrix} \tag{2.7}$$

where $\mathbf{R_Z}$ and $\mathbf{R_W}$ are relationships with $\mathbf{W}$ and $\mathbf{Z}$, respectively. Note that the $\mathbf{R_W}$ appears twice because our model assumes $\mathbf{Y} = \mathbf{XW} + \text{noise}$, so if $\mathbf{W} = \mathbf{R_W V}$, then we must have $\mathbf{Y} = \mathbf{XR_W V} + \text{noise}$.

We want to find the posterior $\mathbf{W} \mid \mathbf{Y} = \mathbf{y}$. The formulation above makes it relatively easy to find the posterior of $\mathbf{Y}$ conditioned on $\mathbf{W}$ (see below), but not vice-versa. So let's pretend instead that

$$\mathbf{U}', \mathbf{V}' \overset{\text{iid}}{\sim} \mathcal{N}(\mathbf{0}, \mathbf{I})$$
$$\begin{bmatrix} \mathbf{W} \\ \mathbf{Y} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{U}' \\ \mathbf{V}' \end{bmatrix}$$

Now $\mathbf{W} \mid \mathbf{Y} = \mathbf{y}$ is straightforward. Since $\mathbf{V}' = \mathbf{D}^{-1}\mathbf{Y}$, the conditional mean and variance of $\mathbf{W} \mid \mathbf{Y} = \mathbf{y}$ can be computed as follows:

$$\begin{aligned}
\mathbb{E}[\mathbf{W} \mid \mathbf{Y} = \mathbf{y}] &= \mathbb{E}[\mathbf{AU}' + \mathbf{BV}' \mid \mathbf{Y} = \mathbf{y}] \\
&= \mathbb{E}[\mathbf{AU}' \mid \mathbf{Y} = \mathbf{y}] + \mathbb{E}[\mathbf{BD}^{-1}\mathbf{Y} \mid \mathbf{Y} = \mathbf{y}] \\
&= \mathbf{A}\underbrace{\mathbb{E}[\mathbf{U}']}_{\mathbf{0}} + \mathbb{E}[\mathbf{BD}^{-1}\mathbf{Y} \mid \mathbf{Y} = \mathbf{y}] \\
&= \mathbf{BD}^{-1}\mathbf{y} \\
\text{Var}(\mathbf{W} \mid \mathbf{Y} = \mathbf{y}) &= \mathbb{E}[(\mathbf{W} - \mathbb{E}[\mathbf{W}])(\mathbf{W} - \mathbb{E}[\mathbf{W}])^\top \mid \mathbf{Y} = \mathbf{y}] \\
&= \mathbb{E}[(\mathbf{AU}' + \mathbf{BD}^{-1}\mathbf{Y} - \mathbf{BD}^{-1}\mathbf{Y})(\mathbf{AU}' + \mathbf{BD}^{-1}\mathbf{Y} - \mathbf{BD}^{-1}\mathbf{Y})^\top \mid \mathbf{Y} = \mathbf{y}] \\
&= \mathbb{E}[(\mathbf{AU}')(\mathbf{AU}')^\top \mid \mathbf{Y} = \mathbf{y}] \\
&= \mathbb{E}[\mathbf{AU}'(\mathbf{U}')^\top\mathbf{A}^\top]
\end{aligned}$$

$$
\begin{aligned}
&= \mathbf{A}\,\underbrace{\mathbb{E}[\mathbf{U}'(\mathbf{U}')^\top]}_{=\mathrm{Var}(\mathbf{U}')=\mathbf{I}}\,\mathbf{A}^\top \\
&= \mathbf{A}\mathbf{A}^\top
\end{aligned}
$$

In both cases above where we drop the conditioning on $\mathbf{Y}$, we are using the fact $\mathbf{U}'$ is independent of $\mathbf{V}'$ (and thus independent of $\mathbf{Y} = \mathbf{D}\mathbf{V}'$). Therefore

$$\mathbf{W} \mid \mathbf{Y} = \mathbf{y} \sim \mathcal{N}(\mathbf{B}\mathbf{D}^{-1}\mathbf{y}, \mathbf{A}\mathbf{A}^\top)$$

Recall that a Gaussian distribution is completely specified by its mean and covariance matrix. We see that the covariance matrix of the joint distribution is

$$
\begin{aligned}
\mathbb{E}\left[\begin{bmatrix}\mathbf{W}\\ \mathbf{Y}\end{bmatrix}\begin{bmatrix}\mathbf{W}^\top & \mathbf{Y}^\top\end{bmatrix}\right] &= \begin{bmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{0} & \mathbf{D}\end{bmatrix}\begin{bmatrix}\mathbf{A}^\top & \mathbf{0}\\ \mathbf{B}^\top & \mathbf{D}^\top\end{bmatrix} \\
&= \begin{bmatrix}\mathbf{A}\mathbf{A}^\top + \mathbf{B}\mathbf{B}^\top & \mathbf{B}\mathbf{D}^\top\\ \mathbf{D}\mathbf{B}^\top & \mathbf{D}\mathbf{D}^\top\end{bmatrix} \\
&= \begin{bmatrix}\boldsymbol{\Sigma}_{\mathbf{W}} & \boldsymbol{\Sigma}_{\mathbf{W},\mathbf{Y}}\\ \boldsymbol{\Sigma}_{\mathbf{Y},\mathbf{W}} & \boldsymbol{\Sigma}_{\mathbf{Y}}\end{bmatrix}
\end{aligned}
$$

Matching the corresponding terms, we can express the conditional mean and variance of $\mathbf{W} \mid \mathbf{Y} = \mathbf{y}$ in terms of these (cross-)covariance matrices:

$$
\begin{aligned}
\mathbf{B}\mathbf{D}^{-1}\mathbf{Y} &= \mathbf{B}\,\underbrace{\mathbf{D}^\top\mathbf{D}^{-\top}}_{\mathbf{I}}\,\mathbf{D}^{-1}\mathbf{Y} = (\mathbf{B}\mathbf{D}^\top)(\mathbf{D}\mathbf{D}^\top)^{-1}\mathbf{Y} = \boldsymbol{\Sigma}_{\mathbf{W},\mathbf{Y}}\boldsymbol{\Sigma}_{\mathbf{Y}}^{-1}\mathbf{Y} \\
\mathbf{A}\mathbf{A}^\top &= \mathbf{A}\mathbf{A}^\top + \mathbf{B}\mathbf{B}^\top - \mathbf{B}\mathbf{B}^\top \\
&= \mathbf{A}\mathbf{A}^\top + \mathbf{B}\mathbf{B}^\top - \mathbf{B}\,\underbrace{\mathbf{D}^\top\mathbf{D}^{-\top}}_{\mathbf{I}}\,\underbrace{\mathbf{D}^{-1}\mathbf{D}}_{\mathbf{I}}\,\mathbf{B}^\top \\
&= \mathbf{A}\mathbf{A}^\top + \mathbf{B}\mathbf{B}^\top - (\mathbf{B}\mathbf{D}^\top)(\mathbf{D}\mathbf{D}^\top)^{-1}\mathbf{D}\mathbf{B}^\top \\
&= \boldsymbol{\Sigma}_{\mathbf{W}} - \boldsymbol{\Sigma}_{\mathbf{W},\mathbf{Y}}\boldsymbol{\Sigma}_{\mathbf{Y}}^{-1}\boldsymbol{\Sigma}_{\mathbf{Y},\mathbf{W}}
\end{aligned}
$$

We can then apply the same reasoning to the original setup:

$$
\begin{aligned}
\mathbb{E}\left[\begin{bmatrix}\mathbf{Y}\\ \mathbf{W}\end{bmatrix}\begin{bmatrix}\mathbf{Y}^\top & \mathbf{W}^\top\end{bmatrix}\right] &= \begin{bmatrix}\mathbf{R}_{\mathbf{Z}}\mathbf{R}_{\mathbf{Z}}{}^\top + \mathbf{X}\mathbf{R}_{\mathbf{W}}\mathbf{R}_{\mathbf{W}}{}^\top\mathbf{X}^\top & \mathbf{X}\mathbf{R}_{\mathbf{W}}\mathbf{R}_{\mathbf{W}}{}^\top\\ \mathbf{R}_{\mathbf{W}}\mathbf{R}_{\mathbf{W}}{}^\top\mathbf{X}^\top & \mathbf{R}_{\mathbf{W}}\mathbf{R}_{\mathbf{W}}{}^\top\end{bmatrix} \\
&= \begin{bmatrix}\boldsymbol{\Sigma}_{\mathbf{Y}} & \boldsymbol{\Sigma}_{\mathbf{Y},\mathbf{W}}\\ \boldsymbol{\Sigma}_{\mathbf{W},\mathbf{Y}} & \boldsymbol{\Sigma}_{\mathbf{W}}\end{bmatrix}
\end{aligned}
$$

Therefore after defining $\boldsymbol{\Sigma}_{\mathbf{Z}} = \mathbf{R}_{\mathbf{Z}}\mathbf{R}_{\mathbf{Z}}{}^\top$, we can read off

$$
\begin{aligned}
\boldsymbol{\Sigma}_{\mathbf{W}} &= \mathbf{R}_{\mathbf{W}}\mathbf{R}_{\mathbf{W}}{}^\top \\
\boldsymbol{\Sigma}_{\mathbf{Y}} &= \boldsymbol{\Sigma}_{\mathbf{Z}} + \mathbf{X}\boldsymbol{\Sigma}_{\mathbf{W}}\mathbf{X}^\top \\
\boldsymbol{\Sigma}_{\mathbf{Y},\mathbf{W}} &= \mathbf{X}\boldsymbol{\Sigma}_{\mathbf{W}} \\
\boldsymbol{\Sigma}_{\mathbf{W},\mathbf{Y}} &= \boldsymbol{\Sigma}_{\mathbf{W}}\mathbf{X}^\top
\end{aligned}
$$

Plugging this into our estimator yields

$$
\begin{aligned}
\hat{\mathbf{w}} &= \mathbb{E}[\mathbf{W} \mid \mathbf{Y} = \mathbf{y}] \\
&= \boldsymbol{\Sigma}_{\mathbf{W},\mathbf{Y}}\boldsymbol{\Sigma}_{Y}^{-1}\mathbf{y}
\end{aligned}
$$

$$= \mathbf{\Sigma_W X^\top (\Sigma_Z + X \Sigma_W X^\top)^{-1} y}$$

One may be concerned because this expression does not take the form we expect – the inverted matrix is hitting $\mathbf{y}$ directly, unlike in other solutions we've seen. Although this form will turn out to be quite informative when we introduce the idea of the kernel trick in machine, learning, it is still disconcertingly different from what we are used to.

However, by using a lot of algebra together with the Woodbury matrix identity[1], it turns out that we can rewrite this expression as

$$\hat{\mathbf{w}} = (\mathbf{X^\top \Sigma_Z^{-1} X + \Sigma_W^{-1}})^{-1} \mathbf{X^\top \Sigma_Z^{-1} y}$$

which looks more familiar. In fact, you can recognize this as the general solution when we have both a generic Gaussian prior on the parameters and colored noise in the observations.

Summary of Linear Gaussian Statistical Models

We have seen a number of related linear models, with varying assumptions about the randomness in the observations and the weights. We summarize these below:

$\mathbf{W} \backslash \mathbf{Z}$	$\mathcal{N}(\mathbf{0}, \mathbf{I})$	$\mathcal{N}(\mathbf{0}, \mathbf{\Sigma_Z})$
No prior	$\hat{\mathbf{w}}_{\text{OLS}} = (\mathbf{X^\top X})^{-1}\mathbf{X^\top y}$	$\hat{\mathbf{w}}_{\text{GLS}} = (\mathbf{X^\top \Sigma_Z^{-1} X})^{-1}\mathbf{X^\top \Sigma_Z^{-1} y}$
$\mathcal{N}(\mathbf{0}, \lambda^{-1}\mathbf{I})$	$\hat{\mathbf{w}}_{\text{RIDGE}} = (\mathbf{X^\top X} + \lambda \mathbf{I})^{-1}\mathbf{X^\top y}$	$(\mathbf{X^\top \Sigma_Z^{-1} X} + \lambda \mathbf{I})^{-1}\mathbf{X^\top \Sigma_Z^{-1} y}$
$\mathcal{N}(\boldsymbol{\mu}_\mathbf{W}, \mathbf{\Sigma_W})$	$\boldsymbol{\mu}_\mathbf{W} + (\mathbf{X^\top X + \Sigma_W^{-1}})^{-1}\mathbf{X^\top}(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_\mathbf{W})$	$\boldsymbol{\mu}_\mathbf{W} + (\mathbf{X^\top \Sigma_Z^{-1} X + \Sigma_W^{-1}})^{-1}\mathbf{X^\top \Sigma_Z^{-1}}(\mathbf{y} - \mathbf{X}\boldsymbol{\mu}_\mathbf{W})$

2.5 Kernels and Ridge Regression

In ridge regression, we given a vector $\mathbf{y} \in \mathbb{R}^n$ and a matrix $\mathbf{X} \in \mathbb{R}^{n \times \ell}$, where n is the number of training points and ℓ is the dimension of the raw data points. In most settings we don't want to work with just the raw feature space, so we augment features to the data points and replace $\mathbf{X}$ with $\mathbf{\Phi} \in \mathbb{R}^{n \times d}$, where $\boldsymbol{\phi}_i^\top = \phi(\mathbf{x}_i) \in \mathbb{R}^d$. Then we solve a well-defined optimization problem that involves $\mathbf{\Phi}$ and $\mathbf{y}$, over the parameters $\mathbf{w} \in \mathbb{R}^d$. Note the problem that arises here. If we have polynomial features of degree at most p in the raw ℓ dimensional space, then there are $d = \binom{\ell+p}{p}$ terms that we need to optimize, which can be very, very large (much larger than the number of training points n). Wouldn't it be useful, if instead of solving an optimization problem over d variables, we could solve an equivalent problem over (potentially much smaller) n variables, and achieve a computational runtime independent of the number of augmented features? As it turns out, the concept of kernels (in addition to a technique called the kernel trick) will allow us to achieve this goal. Recall the solution to ridge regression:

$$\mathbf{w}^* = (\mathbf{\Phi^\top \Phi} + \lambda \mathbf{I})^{-1}\mathbf{\Phi^\top y}$$

This operation involves calculating $\mathbf{\Phi^\top \Phi}$, which is a $d \times d$ matrix and takes $O(d^2 n)$ time to compute. The matrix inversion operation takes an additional $O(d^3)$ time to compute. What we would really like is to have an $n \times n$ matrix that takes $O(n^3)$ to invert. Here's a simple observation: if we flip the order of $\mathbf{\Phi}^\top$ and $\mathbf{\Phi}$, we end up with an $n \times n$ matrix $\mathbf{\Phi \Phi^\top}$. In fact, the matrix $\mathbf{\Phi \Phi^\top}$ has a very intuitive meaning: it is the matrix of inner products between all of the augmented datapoints, which in loose terms measures the "similarity" among of the datapoints and captures their relationship. Now let's see if we could somehow express the solution to ridge regression using the matrix $\mathbf{\Phi \Phi^\top}$.

[1] $(\mathbf{A + UCV})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{C}^{-1} + \mathbf{VA}^{-1}\mathbf{U})^{-1}\mathbf{VA}^{-1}$

Derivation

For simplicity of notation, let's revert back to using $\mathbf{X}$ instead of $\mathbf{\Phi}$ (pretend that we are only working with raw features, our analysis of kernel ridge regression still holds if we use just the raw features). Rearranging the terms of the original ridge regression solution, we have

$$\begin{aligned}
\mathbf{w} &= (\mathbf{X}^\top\mathbf{X} + \lambda I)^{-1}\mathbf{X}^\top\mathbf{y} \\
(\mathbf{X}^\top\mathbf{X} + \lambda I)\mathbf{w} &= \mathbf{X}^\top\mathbf{y} \\
\mathbf{X}^\top\mathbf{X}\mathbf{w} + \lambda\mathbf{w} &= \mathbf{X}^\top\mathbf{y} \\
\lambda\mathbf{w} &= \mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\mathbf{w} \\
\mathbf{w} &= \frac{1}{\lambda}(\mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\mathbf{w}) \\
\mathbf{w} &= \frac{\mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\mathbf{w}}{\lambda} \\
\mathbf{w} &= \mathbf{X}^\top\frac{\mathbf{y} - \mathbf{X}\mathbf{w}}{\lambda}
\end{aligned}$$

which says that *whatever* $\mathbf{w}$ *is*, it is some linear combination of the training points $\mathbf{x}_i$ (because anything of the form $\mathbf{X}^\top\mathbf{v}$ is a linear combination of the columns of $\mathbf{X}^\top$, which are the training points). To find $\mathbf{w}$ it suffices to find $\mathbf{v}$, where $\mathbf{w} = \mathbf{X}^\top\mathbf{v}$.

Recall that the relationship we have to satisfy is $\mathbf{X}^\top\mathbf{X}\mathbf{w} - \lambda\mathbf{w} = \mathbf{X}^\top\mathbf{y}$. Let's assume that we had $\mathbf{v}$, and just substitute $\mathbf{X}^\top\mathbf{v}$ in for all the $\mathbf{w}$'s.

$$\begin{aligned}
\mathbf{X}^\top\mathbf{X}(\mathbf{X}^\top\mathbf{v}) + \lambda(\mathbf{X}^\top\mathbf{v}) &= \mathbf{X}^\top\mathbf{y} \\
\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v} + \mathbf{X}^\top(\lambda\mathbf{v}) &= \mathbf{X}^\top\mathbf{y} \\
\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top\mathbf{v} + \lambda\mathbf{v}) &= \mathbf{X}^\top(\mathbf{y})
\end{aligned}$$

We can't yet isolate $\mathbf{v}$ and have a closed-form solution for it, but we *can* make the observation that if we found an $\mathbf{v}$ such that we had

$$\mathbf{X}\mathbf{X}^\top\mathbf{v} + \lambda\mathbf{v} = \mathbf{y}$$

that would *imply* that this $\mathbf{v}$ also satisfies the above equation. Note that we did not "cancel the $\mathbf{X}^\top$'s on both sides of the equation." We saw that having $\mathbf{v}$ satisfy one equation implied that it satisfied the other as well. So, indeed, we can isolate $\mathbf{v}$ in this new equation:

$$(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})\mathbf{v} = \mathbf{y} \implies \mathbf{v}^* = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

and have that the $\mathbf{v}$ which satisfies this equation will be such that $\mathbf{X}^\top\mathbf{v}$ equals $\mathbf{w}$. We conclude that the optimal $\mathbf{w}$ is

$$\mathbf{w}^* = \mathbf{X}^\top\mathbf{v}^* = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

Recall that previously, we derived ridge regression and ended up with

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$$

In fact, these two are equivalent expressions! The question that now arises is which expression should you pick? Which is *more efficient* to calculate? We will answer this question after we introduce kernels.

Linear Algebra Derivation

The previous derivation involved using some intuitive manipulations to achieve the desired answer. Let's formalize our derivation using more principled arguments from linear algebra and optimization Before we do so, we must first introduce the **Fundamental Theorem of Linear Algebra (FTLA)**: Suppose that there is a matrix (linear map) $\mathbf{X}$ that maps $\mathbb{R}^\ell$ to $\mathbb{R}^n$. Denote $\mathcal{N}(\mathbf{X})$ as the nullspace of $\mathbf{X}$, and $\mathcal{R}(\mathbf{X})$ as the range of $\mathbf{X}$. Then the following properties hold:

1. $\mathcal{N}(\mathbf{X}) \overset{\perp}{\oplus} \mathcal{R}(\mathbf{X}^\top) = \mathbb{R}^\ell$ and $\mathcal{N}(\mathbf{X}^\top) \overset{\perp}{\oplus} \mathcal{R}(\mathbf{X}) = \mathbb{R}^n$ by symmetry

   The symbol $\oplus$ indicates that we taking a **direct sum** of $\mathcal{N}(\mathbf{X})$ and $\mathcal{R}(\mathbf{X}^\top)$, which means that $\forall u \in \mathbb{R}^\ell$ there exist unique elements $u_1 \in \mathcal{N}(\mathbf{X})$ and $u_2 \in \mathcal{R}(\mathbf{X}^\top)$ such that $u = u_1 + u_2$. Furthermore, the symbol $\perp$ indicates that $\mathcal{N}(\mathbf{X})$ and $\mathcal{R}(\mathbf{X}^\top)$ are orthogonal subspaces.

2. $\mathcal{N}(\mathbf{X}^\top\mathbf{X}) = \mathcal{N}(\mathbf{X})$ and $\mathcal{N}(\mathbf{X}\mathbf{X}^\top) = \mathcal{N}(\mathbf{X}^\top)$ by symmetry

3. $\mathcal{R}(\mathbf{X}^\top\mathbf{X}) = \mathcal{R}(\mathbf{X}^\top)$ and $\mathcal{R}(\mathbf{X}\mathbf{X}^\top) = \mathcal{R}(\mathbf{X})$ by symmetry.

Here's where FTLA comes, in the context of kernel ridge regression. We know that we can express any $\mathbf{w} \in \mathbb{R}^\ell$ as a unique combination $\mathbf{w} = \mathbf{w}_1 + \mathbf{w}_2$, where $\mathbf{w}_1 \in \mathcal{R}(\mathbf{X}^\top)$ and $\mathbf{w}_2 \in \mathcal{N}(\mathbf{X})$. Equivalently we can express this as $\mathbf{w} = \mathbf{X}^\top\mathbf{v} + \mathbf{r}$, where $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{r} \in \mathcal{N}(\mathbf{X})$. Now, instead of optimizing over $\mathbf{w} \in \mathbb{R}^\ell$, we can optimize over $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{r} \in \mathbb{R}^\ell$, which equates to optimizing over $n + \ell$ variables. However, as we shall see, the optimization over $\mathbf{r}$ will be trivial so we just have to optimize an n dimensional problem.

We know that $\mathbf{w} = \mathbf{X}^\top\mathbf{v} + \mathbf{r}$, where $\mathbf{v} \in \mathbb{R}^n$ and $\mathbf{r} \in \mathcal{N}(\mathbf{X})$. Let's now solve ridge regression by optimizing over the variables $\mathbf{v}$ and $\mathbf{r}$ instead of $\mathbf{w}$:

$$\begin{aligned}
\mathbf{v}^*, \mathbf{r}^* &= \underset{\mathbf{v}\in\mathbb{R}^n, \mathbf{r}\in\mathcal{N}(\mathbf{X})}{\arg\min} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2 \\
&= \underset{\mathbf{v}\in\mathbb{R}^n, \mathbf{r}\in\mathcal{N}(\mathbf{X})}{\arg\min} \|\mathbf{X}(\mathbf{X}^\top\mathbf{v} + \mathbf{r}) - \mathbf{y}\|_2^2 + \lambda\|\mathbf{X}^\top\mathbf{v} + \mathbf{r}\|_2^2 \\
&= \underset{\mathbf{v}\in\mathbb{R}^n, \mathbf{r}\in\mathcal{N}(\mathbf{X})}{\arg\min} \|\mathbf{X}\mathbf{X}^\top\mathbf{v} + \cancel{\mathbf{X}\mathbf{r}} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{X}^\top\mathbf{v} + \mathbf{r}\|_2^2 \\
&= \underset{\mathbf{v}\in\mathbb{R}^n, \mathbf{r}\in\mathcal{N}(\mathbf{X})}{\arg\min} \left(\mathbf{v}^\top\mathbf{X}\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v} - 2\mathbf{v}^\top\mathbf{X}\mathbf{X}^\top\mathbf{y} + \mathbf{y}^\top\mathbf{y}\right) + \lambda\left(\mathbf{v}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v} + \cancel{2\mathbf{v}^\top\mathbf{X}\mathbf{r}} + \mathbf{r}^\top\mathbf{r}\right) \\
&= \underset{\mathbf{v}\in\mathbb{R}^n, \mathbf{r}\in\mathcal{N}(\mathbf{X})}{\arg\min} \left(\mathbf{v}^\top\mathbf{X}\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v} - 2\mathbf{v}^\top\mathbf{X}\mathbf{X}^\top\mathbf{y}\right) + \lambda\left(\mathbf{v}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v} + \mathbf{r}^\top\mathbf{r}\right)
\end{aligned}$$

We crossed out $\mathbf{X}\mathbf{r}$ and $2\mathbf{v}^\top\mathbf{X}\mathbf{r}$ because $\mathbf{r} \in \mathcal{N}(\mathbf{X})$ and therefore $\mathbf{X}\mathbf{r} = \mathbf{0}$. Now we are optimizing over $L(\mathbf{v}, \mathbf{r})$, which is **jointly convex** in $\mathbf{v}$ and $\mathbf{r}$, because its Hessian is PSD. Let's show that this is indeed the case:

$$\nabla_{\mathbf{r}}^2 L(\mathbf{v}, \mathbf{r}) = 2\mathbf{I} \succeq \mathbf{0}$$

$$\nabla_{\mathbf{r}}\nabla_{\mathbf{v}} L(\mathbf{v}, \mathbf{r}) = \nabla_{\mathbf{v}}\nabla_{\mathbf{r}} L(\mathbf{v}, \mathbf{r}) = \mathbf{0}$$

$$\nabla_{\mathbf{v}}^2 L(\mathbf{v}, \mathbf{r}) = 2\mathbf{X}\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top + 2\lambda\mathbf{X}\mathbf{X}^\top \succeq \mathbf{0}$$

Since the cross terms of the Hessian are $\mathbf{0}$, it suffices that $\nabla_{\mathbf{r}}^2 L(\mathbf{v}, \mathbf{r})$ and $\nabla_{\mathbf{v}}^2 L(\mathbf{v}, \mathbf{r})$ are PSD to establish joint convexity. With joint convexity established, we can set the gradient to $\mathbf{0}$ w.r.t $\mathbf{r}$ and $\mathbf{v}$ and obtain the global minimum:

$$\nabla_{\mathbf{r}} L(\mathbf{v}, \mathbf{r}^*) = 2\mathbf{r}^* = \mathbf{0} \implies \mathbf{r}^* = \mathbf{0}$$

Note that $\mathbf{r}^* = \mathbf{0}$ just so happens in to be in $\mathcal{N}(\mathbf{X})$, so it is a feasible point.

$$\begin{aligned}\nabla_{\mathbf{v}} L(\mathbf{v}^*, \mathbf{r}^*) = 2\mathbf{X}\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v}^* - 2\mathbf{X}\mathbf{X}^\top\mathbf{y} + 2\lambda\mathbf{X}\mathbf{X}^\top\mathbf{v}^* = \mathbf{0}\\ \implies \mathbf{X}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})\mathbf{v}^* = \mathbf{X}\mathbf{X}^\top(\mathbf{y})\\ \implies \mathbf{v}^* = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}\end{aligned}$$

Note that $\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I}$ is positive definite and therefore invertible, so we can compute $(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$. Even though $(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$ is a critical point for which the gradient is $\mathbf{0}$, it must achieve the global minimum because the objective is jointly convex. We conclude that

$$\mathbf{w}^* = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

and arrive at the same solution as in the previous derivation.

Non-i.i.d. Case

So far we have assumed the special i.i.d. case of ridge regression, where

$$\mathbf{Y}|\mathbf{W} \sim \mathcal{N}(\mathbf{X}\mathbf{W}, \sigma^2\mathbf{I}), \quad \mathbf{W} \sim \mathcal{N}(\mathbf{0}, \sigma_h^2\mathbf{I})$$

In the non-i.i.d case we consider arbitrary covariance matrices:

$$\mathbf{Y}|\mathbf{W} \sim \mathcal{N}(\mathbf{X}\mathbf{W}, \Sigma_{\mathbf{Z}}), \quad \mathbf{W} \sim \mathcal{N}(\mathbf{0}, \Sigma_{\mathbf{W}})$$

As we've seen already, the solution in this case can be expressed in two forms, either the familiar case

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{\Sigma}_{\mathbf{Z}}^{-1}\mathbf{X} + \mathbf{\Sigma}_{\mathbf{W}}^{-1})^{-1}\mathbf{X}^\top\mathbf{\Sigma}_{\mathbf{Z}}^{-1}\mathbf{y}$$

or the case that we desire in kernel ridge regression

$$\mathbf{w}^* = \mathbf{\Sigma}_{\mathbf{W}}\mathbf{X}^\top(\mathbf{X}\mathbf{\Sigma}_{\mathbf{W}}\mathbf{X}^\top + \mathbf{\Sigma}_{\mathbf{Z}})^{-1}\mathbf{y}$$

The principal difference in the non-i.i.d case is that we are computing $\mathbf{X}\mathbf{\Sigma}_{\mathbf{W}}\mathbf{X}^\top$ as opposed to $\mathbf{X}\mathbf{X}^\top$.

Kernels

Having derived the kernel ridge regression formulation for the raw data matrix $\mathbf{X}$, we can apply the exact same logic to the augmented data matrix $\mathbf{\Phi}$ and replace the optimal expression with

$$\mathbf{w}^* = \mathbf{\Phi}^\top(\mathbf{\Phi}\mathbf{\Phi}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

Let's explore the $\mathbf{\Phi}\mathbf{\Phi}^\top$ term in kernel ridge regression in more detail:

$$\mathbf{\Phi}\mathbf{\Phi}^\top = \begin{pmatrix} \text{---} & \boldsymbol{\phi}_1^\top & \text{---} \\ \text{---} & \boldsymbol{\phi}_2^\top & \text{---} \\ & \vdots & \\ \text{---} & \boldsymbol{\phi}_n^\top & \text{---} \end{pmatrix} \begin{pmatrix} | & | & & | \\ \boldsymbol{\phi}_1 & \boldsymbol{\phi}_2 & \dots & \boldsymbol{\phi}_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} \boldsymbol{\phi}_1^\top\boldsymbol{\phi}_1 & \boldsymbol{\phi}_1^\top\boldsymbol{\phi}_2 & \dots \\ \boldsymbol{\phi}_2^\top\boldsymbol{\phi}_1 & \ddots & \\ \vdots & & \boldsymbol{\phi}_n^\top\boldsymbol{\phi}_n \end{pmatrix}$$

Each entry $\mathbf{\Phi}\mathbf{\Phi}_{ij}^\top$ is a dot product between $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_j)$ and can be interpreted as a similarity measure:

$$\mathbf{\Phi}\mathbf{\Phi}_{ij}^\top = \langle \boldsymbol{\phi}_i, \boldsymbol{\phi}_j \rangle = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle = k(\mathbf{x}_i, \mathbf{x}_j)$$

where $k(.,.)$ is the **kernel** function. The kernel function takes raw-feature inputs and outputs their inner product in the augmented feature space. We denote the matrix of $k(\mathbf{x}_i, \mathbf{x}_j)$ terms as the **Gram matrix** and denote it as $\mathbf{K}$:

$$\mathbf{K} = \mathbf{\Phi}\mathbf{\Phi}^\top = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \dots \\ k(\mathbf{x}_2, \mathbf{x}_1) & \ddots & \\ \vdots & & k(\mathbf{x}_n, \mathbf{x}_n) \end{pmatrix}$$

Formally, $k(\mathbf{x}_i, \mathbf{x}_j)$ is defined to be a valid kernel function if either of the following definitions are met:

- There exists a feature map $\phi(.)$ such that $\forall \mathbf{x}_i, \mathbf{x}_j, \quad k(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$
- For all sets $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, the Gram matrix $\mathbf{K}(\mathcal{D})$ is PSD

We will now state some basic properties of kernels.

- Given two valid kernels k_a and k_b, their linear combination

  $$k(\mathbf{x}_i, \mathbf{x}_j) = \alpha k_a(\mathbf{x}_i, \mathbf{x}_j) + \beta k_b(\mathbf{x}_i, \mathbf{x}_j)$$

  where $\alpha, \beta \geq 0$ is also a valid kernel. We can show this from the second property:

  $$\forall \mathbf{v} \in \mathbb{R}^n, \mathbf{v}^\top(\alpha \mathbf{K}_a + \beta \mathbf{K}_b)\mathbf{v} = \alpha \mathbf{v}^\top \mathbf{K}_a \mathbf{v} + \beta \mathbf{v}^\top \mathbf{K}_b \mathbf{v} \geq 0$$

- Given a positive semidefinite matrix $\mathbf{\Sigma}$,

  $$k(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^\top \mathbf{\Sigma} \phi(\mathbf{x}_j)$$

  is a valid kernel. We can show this from the first property: $\widetilde{\phi}(\mathbf{x}_i) = \mathbf{\Sigma}^{\frac{1}{2}} \phi(\mathbf{x}_i)$

- Given a valid kernel k_a,

  $$k(\mathbf{x}_i, \mathbf{x}_j) = f(\mathbf{x}_i) f(\mathbf{x}_j) k_a(\mathbf{x}_i, \mathbf{x}_j)$$

  is a valid kernel. We can show this from the first property: $\widetilde{\phi}(\mathbf{x}_i) = f(\mathbf{x}_i)\phi(\mathbf{x}_i)$

Computing the each Gram matrix entry $k(\mathbf{x}_i, \mathbf{x}_j)$ can be done in a straightforward fashion if we apply the feature map to $\mathbf{x}_i$ and $\mathbf{x}_j$ and then take their dot product in the augmented feature space — this takes $O(d)$ time, where d is the dimensionality of the problem in the augmented feature space. However, if we use the **kernel trick**, we can perform this operation in $O(\ell + \log p)$ time, where ℓ is the dimensionality of the problem in the raw feature space and p is the degree of the polynomials in the augmented feature space.

Kernel Trick

Suppose that we are computing $k(\mathbf{x}, \mathbf{z})$, using a p-degree polynomial feature map that maps ℓ dimensional inputs to $d = O(\ell^p)$ dimensional outputs. Let's take $p = 2$ and $\ell = 2$ as an example. Define the polynomial feature map as

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1 x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix}^\top$$

the kernel function can be expressed as

$$\begin{aligned}
k(\mathbf{x},\mathbf{z}) &= \phi(\mathbf{x})^\top \phi(\mathbf{z}) \\
&= \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix}^\top \begin{bmatrix} z_1^2 & z_2^2 & \sqrt{2}z_1z_2 & \sqrt{2}z_1 & \sqrt{2}z_2 & 1 \end{bmatrix} \\
&= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2 + 2x_1z_1 + 2x_2z_2 + 1 \\
&= (x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2) + 2x_1z_1 + 2x_2z_2 + 1 \\
&= (x_1z_1 + x_2z_2)^2 + 2(x_1z_1 + x_2z_2) + 1 \\
&= (\mathbf{x}^\top\mathbf{z})^2 + 2\mathbf{x}^\top\mathbf{z} + 1 \\
&= (\mathbf{x}^\top\mathbf{z} + 1)^2
\end{aligned}$$

We can compute $k(\mathbf{x},\mathbf{z})$ either by

1. Raising the inputs to the augmented feature space and take their inner product
2. Computing $(\mathbf{x}^\top\mathbf{z}+1)^2$, which involves an inner product of the raw-feature inputs

Clearly, the latter option is much cheaper to calculate, taking $O(\ell + \log p)$ time, instead of $O(\ell^p)$ time. In fact, this concept generalizes for any arbitrary ℓ and p, and for p-degree polynomial features, we have that

$$k(\mathbf{x},\mathbf{z}) = (\mathbf{x}^\top\mathbf{z}+1)^p$$

The kernel trick makes computations significantly cheaper to perform, making kernelization much more appealing! The takeaway here is that no matter what the degree p is, the computational complexity is the same — it is only dependent on the dimensionality of the raw feature space!

Note that we can equivalently express the degree-2 polynomial features problem using the more natural mapping

$$\widetilde{\phi(\mathbf{x})} = \begin{bmatrix} x_1^2 & x_2^2 & x_1x_2 & x_1 & x_2 & 1 \end{bmatrix}^\top$$

in which case the kernel function would be expressed as

$$k(\mathbf{x},\mathbf{z}) = \widetilde{\phi(\mathbf{x})}^\top \mathbf{\Sigma} \widetilde{\phi(\mathbf{z})} = (\mathbf{x}^\top\mathbf{z}+1)^2, \quad \mathbf{\Sigma} = \text{Diag}\begin{pmatrix} 1 & 1 & 2 & 2 & 2 & 1 \end{pmatrix}$$

Thus we can view kernel ridge regression with the kernel trick in two ways:

1. i.i.d. prior $\mathbf{W} \sim \mathcal{N}\left(\mathbf{0}, \text{Diag}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}\right)$, using the feature mapping $\phi(\mathbf{x})$
2. non-i.i.d prior $\mathbf{W} \sim \mathcal{N}\left(\mathbf{0}, \text{Diag}\begin{pmatrix} 1 & 1 & 2 & 2 & 2 & 1 \end{pmatrix}\right)$, using the feature mapping $\widetilde{\phi(\mathbf{x})}$ (note that the kernel trick is only applicable for this specific setting of $\mathbf{\Sigma}$ — it does not necessarily apply to arbitrary $\mathbf{\Sigma}$.)

Computational Analysis

Back to the original question: in ridge regression, should we compute

$$\mathbf{w}^* = \mathbf{\Phi}^\top(\mathbf{\Phi}\mathbf{\Phi}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

or

$$\mathbf{w}^* = (\mathbf{\Phi}^\top\mathbf{\Phi} + \lambda\mathbf{I})^{-1}\mathbf{\Phi}^\top\mathbf{y}$$

Let's compare their computational complexities. Suppose you are given an arbitrary test point $\mathbf{z} \in \mathbb{R}^\ell$, and you would like to compute its predicted value $\hat{\mathbf{y}}$. Let's see how these values are calculated in each case:

1. Kernelized

$$\hat{\mathbf{y}} = \langle \phi(\mathbf{z}), \mathbf{w}^* \rangle = \phi(\mathbf{z})^\top \mathbf{\Phi}^\top (\mathbf{\Phi}\mathbf{\Phi}^\top + \lambda \mathbf{I})^{-1} \mathbf{y} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{z}) & \dots & k(\mathbf{x}_n, \mathbf{z}) \end{bmatrix} (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y}$$

   Computing the $\mathbf{K}$ term takes $O(n^2(\ell + \log p))$, and inverting the matrix takes $O(n^3)$. These two computations dominate, for a total computation time of $O(n^3 + n^2(\ell + \log p))$.

2. Non-kernelized

$$\hat{\mathbf{y}} = \langle \phi(\mathbf{z}), \mathbf{w}^* \rangle = \phi(\mathbf{z})^\top (\mathbf{\Phi}^\top \mathbf{\Phi} + \lambda \mathbf{I})^{-1} \mathbf{\Phi}^\top \mathbf{y}$$

   Computing the $\mathbf{\Phi}^\top \mathbf{\Phi}$ term takes $O(d^2 n)$, and inverting the matrix takes $O(d^3)$. These two computations dominate, for a total computation time of $O(d^3 + d^2 n)$.

Here is the takeaway: if $d \ll n$, the non-kernelized method is preferable. Otherwise if $n \ll d$, the kernelized method is preferable.

2.6 Sparse Least Squares

Suppose we want to solve the least squares objective, subject to a constraint that $\mathbf{w}$ is sparse. Mathematically this is expressed as

$$\begin{aligned} \min_{\mathbf{w}} \quad & \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \\ \text{s.t.} \quad & \|\mathbf{w}\|_0 \leq k \end{aligned}$$

where the ℓ_0 norm of $\mathbf{w}$ is simply the number of non-zero elements in $\mathbf{w}$. This quantity is otherwise known as the **Hamming Distance** between $\mathbf{w}$ and $\mathbf{0}$, the vector of zeros.

There are several motivations for designing optimization problems with sparse solutions. One advantage is that sparse weights speed up testing time. In the context of primal problems, if the weight vector $\mathbf{w}$ is sparse, then after we compute $\mathbf{w}$ in training, we can discard features/dimensions with 0 weight, as they will contribute nothing to the evaluation of the hypothesized regression values of test points. A similar reasoning applies to dual problems with dual weight vector $\mathbf{v}$, allowing us to discard the training points corresponding to dual weight 0, ultimately allowing for faster evaluation of our hypothesis function on test points.

Note that the ℓ_0 norm does not actually satisfy the properties of a norm, evident by the fact that the it is not convex, a property that all norms share. Solving this optimization problem is NP-hard, so we instead aim to find a computationally feasible alternative method that can approximate the optimal solution. We will present two such methods: LASSO, a relaxed version of the problem that replaces the ℓ_0 norm with a ℓ_1 norm, and Matching pursuit, a greedy algorithm that iteratively updates one entry of $\mathbf{w}$ at a time until the sparsity constraint can not longer be satisfied.

LASSO

The **least absolute shrinkage and selection operator (LASSO)**, introduced in 1996 by Robert Tibshirani, is identical to the sparse least squares objective, except that the ℓ_0 norm penalizing $\mathbf{w}$ is now changed to an ℓ_1 norm:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \\ \text{s.t.} \quad & \|\mathbf{w}\|_1 \leq k \end{aligned}$$

(The k in the constraint is not necessarily the same k in the sparse least squares objective.) The ℓ_1 norm of $\mathbf{w}$ is the sum of absolute values of its entries:

$$\|\mathbf{w}\|_1 = \sum_{i=1}^{d} |w_i|$$

Unlike the ℓ_0 norm, the ℓ_1 norm actually satisfies the properties of norms. The relaxation from the ℓ_0 to ℓ_1 norm is desirable, because it makes the optimization problem convex, and is no longer NP-hard to solve. But does the ℓ_1 norm still induce sparsity like the ℓ_0? As we will see, the answer is yes!

Due to strong duality, we can equivalently express the LASSO problem in the unconstrained form

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \lambda \|\mathbf{w}\|_1$$

We make a striking observation here: LASSO is identical to the ridge regression objective, except that the ℓ_2 norm (squared) penalizing $\mathbf{w}$ is now changed to an ℓ_1 norm (with no squaring term).

Recall that the ℓ_2 norm squared of $\mathbf{w}$, the sum of squared values of its entries:

$$\|\mathbf{w}\|_2^2 = \sum_{i=1}^{d} w_i^2$$

As it turns out, the simple change from the ℓ_2 to ℓ_1 norm inherently leads to a sparse solution. In fact, the sparsity inducing properties of the ℓ_1 norm are not just unique to least squares. To illustrate the point, let's take a step away from least squares for a moment and discuss the ℓ_1 norm in the context of SVMs. Recall the soft-margin SVM problem (constraints omitted for brevity):

$$\min_{\mathbf{w},\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i$$

The slack ξ_i is constrained to be either positive or zero. Note that if a point $\mathbf{x}_i$ has a nonzero slack $\xi_i > 0$, by definition it must lie inside the margin. Due to the heavy penalty factor C for violating the margin there are relatively few such points, and thus the slack vector ξ is sparse — most of its entries are 0. We are interested in explaining why this phenomenon occurs in this specific optimization problem, and identifying the key properties that determine sparse solutions for arbitrary optimization problems.

To reason about the SVM case, let's see how changing some arbitrary slack variable ξ_i affects the loss. A unit decrease in ξ_i results in a "reward" of C, and is captured by the partial derivative $\frac{\partial L}{\partial \xi_i}$. Note that no matter what the current value of ξ_i is, the reward for decreasing ξ_i is constant. Of course, decreasing ξ_i may change the boundary and thus the cost attributed to the size of the margin $\|\mathbf{w}\|^2$. The overall reward for decreasing ξ_i is either going to be worth the effort (greater than cost incurred from $\mathbf{w}$) or not worth the effort (less than cost incurred from $\mathbf{w}$). Intuitively, ξ_i will continue to decrease until it hits a lower-bound "equilibrium" — which is often just 0.

Now consider the following formulation (constraints omitted for brevity again):

$$\min_{\mathbf{w},\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n} \xi_i^2$$

The reward for decreasing ξ_i is no longer constant — at any point, a unit decrease in ξ_i results in a "reward" of $2C\xi_i$. As ξ_i approaches 0, the rewards get smaller and smaller, reaching infinitesimal values. On the other hand, decreasing ξ_i causes a finite increase in the cost incurred by the $\|\mathbf{w}\|^2$ — the same increase in cost as in the previous example. Intuitively, we can reason that there will be a threshold value ξ_i^* such that decreasing ξ_i further will no longer outweigh the cost incurred by the size of the margin, and that the ξ_i's will halt their descent before they hit zero.

The same reasoning applies to least squares as well. For any particular component w_i of $\mathbf{w}$, the corresponding loss in LASSO is the absolute value $|w_i|$, while the loss in ridge regression is the squared term w_i^2. In the case of LASSO the "reward" for decreasing w_i by a unit amount is a constant λ, while for ridge regression the equivalent "reward" is $2\lambda w_i$, which depends on the value of w_i. There is a compelling geometric argument behind this reasoning as well.

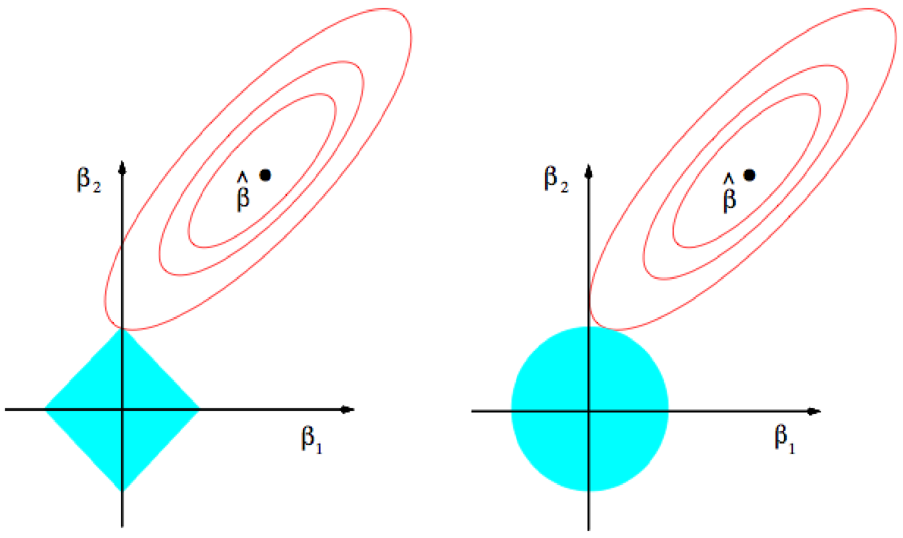


Figure 2.2: Comparing contour plots for LASSO (left) vs. ridge regression (right).

Suppose for simplicity that we are only working with 2-dimensional data points and are thus optimizing over two weight variables w_1 and w_2. In both figures above, the red ellipses represent isocontours in w-space of the squared loss $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$. In ridge regression, each isocontour of $\lambda\|\mathbf{w}\|_2^2$ is represented by a circle, one of which is shown in the right figure. Note that the optimal $\mathbf{w}$ will only occur at points of tangency between the red ellipse and the blue circle. Otherwise we could always move along the isocontour of one of the functions (keeping its overall cost fixed) while improving the value of the the other function, thereby improving the overall value of the loss function. We can't really infer much about these points of tangency other than the fact that the blue circle centered at the origin draws the optimal point closer to the origin (ridge regression penalizes large weights).

Now, let's examine the LASSO case. The red ellipses represent the same objective $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$, but now the ℓ_1 regularization term $\lambda\|\mathbf{w}\|_1$ is represented by diamond isocontours. As with ridge regression, note that the optimal point in w-space must occur at points of tangency between the ellipse and the diamond. Due to the "pointy" property of the diamonds, tangency is very likely to happen at the *corners* of the diamond because they are single points from which the rest of the diamond draws away from. And what are the corners of the diamond? Why, they are points at which one component of $\mathbf{w}$ is 0!

Solving LASSO

Convinced that LASSO achieves sparsity, now let's find the optimal solution to LASSO. Unlike ridge regression, it is not exactly clear what the closed form solution is through linear algebra or gradient methods, since the objective function not differentiable (due to the "pointiness" of the ℓ_1 norm). Specifically, LASSO zeros out features, and once these weights are set to 0 the objective function becomes non-differentiable. Note however, that the objective is still convex, and we could use an iterative method such as subgradient descent or line search to solve the problem. Here, we will use a line search method called **coordinate descent**.

While SGD focuses on iteratively optimizing the value of the objective $L(\mathbf{w})$ for each *sample* in the training set, coordinate descent iteratively optimizes the value of the objective for each *feature*.

Algorithm 1: Coordinate Descent

```
while $\mathbf{w}$ has not converged do
    pick a feature index $i$
    update $w_i$ to $\arg\min_{w_i} L(\mathbf{w})$
```

Coordinate descent is guaranteed to find the global minimum if L is **jointly convex**. No such guarantees can be made however if L is only **elementwise convex**, since it may have local minima. To understand why, let's start by understanding elementwise vs joint convexity. Suppose we are trying to minimize $f(x, y)$, a function of two scalar variables x and y. For simplicity, assume that f is twice differentiable, so we can take its Hessian. $f(x, y)$ is element-wise convex in x if its Hessian is psd when y is fixed:

$$\frac{\partial^2}{\partial x \partial x} f(x, y) \geq 0$$

Same goes for element-wise convexity in y.

$f(x, y)$ is jointly convex in x and y if its Hessian $\nabla^2 f(x, y)$ is psd. Note that being element-wise convex in both x and y does not imply joint convexity in x and y (consider $f(x, y) = x^2 + y^2 - 4xy$ as an example). However, being joint convexity in x and y does imply being element-wise convex in both x and y.

Now, if $f(x, y)$ was jointly convex, then we could find the gradient wrt. x and y individually, set them to 0, and be guaranteed that would be the global minimum. Can we do this if $f(x, y)$ is element-wise convex in both x and y? Even though it is true that $\min_{x,y} f(x, y) = \min_x \min_y f(x, y)$, we can't always just set gradients to 0 if $f(x, y)$ is not jointly convex. While the inner optimization problem over y is convex, the outer optimization problem over x may no longer be convex. In the case when joint convexity is not reached, there is no clean strategy to find global minimum and we must analyze all of the critical points to find the minimum.

In the case of LASSO, the objective function is jointly convex, so we can use coordinate descent. There are a few details to be filled in, namely the choice of which feature to update and how w_i is updated. One simple way is to just pick a random feature i each iteration. After choosing the feature, we have to update $w_i \leftarrow \arg\min_{w_i} L(\mathbf{w})$. For LASSO, it turns out there is a closed-form solution (note that we are only minimizing with respect to one feature instead of all the features).

Let's solve the line search problem $\min_{w_i} L(\mathbf{w})$. For convenience, let's separate the terms that depend on w_i from those that don't. Denoting $\mathbf{x}_j$ as the j-th column of $\mathbf{X}$, we have

$$L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_1$$

$$
\begin{aligned}
&= \left\| \sum_{j=1}^{d} w_j \mathbf{x}_j - \mathbf{y} \right\|_2^2 + \lambda |w_i| + \lambda \sum_{j \neq i} |w_j| \\
&= \|w_i \mathbf{x}_i + \mathbf{r}\|_2^2 + \lambda |w_i| + C
\end{aligned}
$$

where $\mathbf{r} = \sum_{j\neq i} w_j \mathbf{x}_j - \mathbf{y}$ and $C = \lambda \sum_{j\neq i} |w_j|$. The objective can in turn be written as

$$L(\mathbf{w}) = \lambda |w_i| + C + \sum_{j=1}^{n} (w_i x_{ji} + r_j)^2$$

where x_{ji} is the i'th feature of the j'th datapoint.

Suppose that the optimal w_i^* is strictly positive: $w_i^* > 0$. Setting the partial derivative of the objective wrt. w_i (the partial derivative in this case is defined since $w_i^* \neq 0$) to 0, we obtain

$$\frac{\partial L}{\partial w_i} = \lambda + \sum_{j=1}^{n} 2x_{ji}(w_i x_{ji} + r_j) = 0 \implies w_i^* = \frac{-\lambda - \sum_{j=1}^{n} 2x_{ji} r_j}{\sum_{j=1}^{n} 2x_{ji}^2}$$

Denoting $a = -\sum_{j=1}^{n} 2x_{ji} r_j$ and $b = \sum_{j=1}^{n} 2x_{ji}^2$, we have

$$w_i^* = \frac{-\lambda + a}{b}$$

But this only holds if the right hand side, $\frac{-\lambda+a}{b}$, is actually positive. If it is negative or 0, then this means there is no optimum in $(0, \infty)$.

When $w_i^* < 0$, then similar calculations will lead to

$$w_i^* = \frac{\lambda + a}{b}$$

Again, this only holds if $\frac{\lambda+a}{b}$ is actually negative. If it is positive or 0, then there is no optimum in $(-\infty, 0)$.

If neither the conditions $\frac{-\lambda+a}{b} > 0$ or $\frac{\lambda+a}{b} < 0$ hold, then there is no optimum in $(-\infty, 0)$ or $(0, \infty)$. But the LASSO objective is convex in w_i and has an optimum somewhere, thus in this case $w_i^* = 0$. In order for this case to hold, we must have that $\frac{-\lambda+a}{b} \leq 0$ and $\frac{\lambda+a}{b} \geq 0$. Rearranging, we can see this is equivalent to $|a| \leq \lambda$.

Examine each of the following cases:

- $\frac{-\lambda+a}{b} \leq 0$ and $\frac{\lambda+a}{b} \geq 0$: $w_i^* = 0$
- $\frac{-\lambda+a}{b} \leq 0$ and $\frac{\lambda+a}{b} < 0$: $w_i^* < 0$
- $\frac{-\lambda+a}{b} > 0$ and $\frac{\lambda+a}{b} \geq 0$: $w_i^* > 0$
- $\frac{-\lambda+a}{b} > 0$ and $\frac{\lambda+a}{b} < 0$: impossible since this implies $\frac{-\lambda+a}{b} > \frac{\lambda+a}{b}$ and λ and b are non-negative

The cases above imply the optimal solution w_i^*:

$$w_i^* = \begin{cases} 0 & \text{if } |a| \leq \lambda \\ \frac{-\lambda+a}{b} & \text{if } \frac{-\lambda+a}{b} > 0 \\ \frac{\lambda+a}{b} & \text{if } \frac{\lambda+a}{b} < 0 \end{cases}$$

where

$$a = -\sum_{j=1}^{n} 2x_{ji}r_j, \quad b = \sum_{j=1}^{n} 2x_{ji}^2$$

This is not a gradient-descent update — we have a closed-form solution for the optimum w_i, given that all of the other weights are fixed constants. We can see explicitly how the LASSO objective induces sparsity — a is some function of the data and the other weights, and when $|a| \leq \lambda$, we set $w_i = 0$ in this iteration of coordinate descent. By increasing λ, we increase the threshold of $|a|$ for w_i to be set to 0, and our solution becomes more sparse. Also note that the term $\frac{a}{b}$ is the least squares solution (without regularization), so we can see that the regularization term tries to pull the least squares update towards 0.

One subtle point: during coordinate descent, weights can be "reactivated" after having been set to 0 in a previous iteration, since a is affected by factors other than w_i.

Matching Pursuit

Rather than relaxing the ℓ_0 constraint (as seen in LASSO), the **matching pursuit** algorithm keeps the constraint, and instead finds an approximate solution to the sparse least squares problem in a greedy fashion. The algorithm starts with with a completely sparse solution ($\mathbf{w}^0 = \mathbf{0}$), and iteratively updates $\mathbf{w}$ until the sparsity constraint $\|\mathbf{w}\|_0 \leq k$ can no longer be met. At iteration t, the algorithm can only update one entry of $\mathbf{w}^{t-1}$, and it chooses the feature that minimizes the (squared) norm of the resulting *residual* $\|\mathbf{r}^t\|^2 = \|\mathbf{y} - \mathbf{X}\mathbf{w}^t\|^2$.

Algorithm 2: Matching Pursuit

```
initialize the weights $\mathbf{w}^0 = \mathbf{0}$ and the residual $\mathbf{r}^0 = \mathbf{y} - \mathbf{X}\mathbf{w}^0 = \mathbf{y}$
while $\|\mathbf{w}\|_0 < k$ do
    find the feature $i$ for which the length of the projected residual onto $\mathbf{x}_i$ is maximized:
        $i = \arg\min_j (\min_\nu \|\mathbf{r}^{t-1} - \nu\mathbf{x}_j\|) = \arg\max_j \frac{|\langle \mathbf{r}^{t-1}, \mathbf{x}_j \rangle|}{\|\mathbf{x}_j\|}$
    update the $i$'th feature entry of the weight vector:
        $w_i^t = w_i^{t-1} + \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_i \rangle}{\|\mathbf{x}_i\|^2}$
    update the residual vector: $\mathbf{r}^t = \mathbf{y} - \mathbf{X}\mathbf{w}^t$
```

At iteration t, we pick the coordinate i such that the distance from the residual $\mathbf{r}^{t-1}$ to $\mathbf{x}_i$ (the i'th column of $\mathbf{X}$ corresponding to feature i, *not* datapoint i) is minimized:

$$i = \arg\min_j (\min_\nu \|\mathbf{r}^{t-1} - \nu\mathbf{x}_j\|)$$

This equates to finding the index i for which the length of the projection onto $\mathbf{x}_i$ is maximized:

$$i = \arg\max_j \frac{|\langle \mathbf{r}^{t-1}, \mathbf{x}_j \rangle|}{\|\mathbf{x}_j\|}$$

Let's see why this is true. The inner optimization problem $\min_\nu \|\mathbf{r}^{t-1} - \nu\mathbf{x}_i\|$ is simply a projection problem, and its solution is

$$\nu^* = \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j \rangle}{\langle \mathbf{x}_j, \mathbf{x}_j \rangle}$$

which gives a value of

$$\left\|\mathbf{r}^{t-1} - \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle}\mathbf{x}_i\right\|$$

The outer optimization problem selects the feature that minimizes this quantity:

$$\begin{aligned}
i &= \arg\min_j \left\|\mathbf{r}^{t-1} - \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle}\mathbf{x}_i\right\| \\
&= \arg\min_j \left\|\mathbf{r}^{t-1} - \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle}\mathbf{x}_i\right\|^2 \\
&= \arg\min_j \|\mathbf{r}^{t-1}\|^2 + \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle^2}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle} - 2\frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle^2}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle} \\
&= \arg\min_j \|\mathbf{r}^{t-1}\|^2 - \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle^2}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle} \\
&= \arg\max_j \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle^2}{\langle \mathbf{x}_j, \mathbf{x}_j\rangle} \\
&= \arg\max_j \frac{\left|\langle \mathbf{r}^{t-1}, \mathbf{x}_j\rangle\right|}{\|\mathbf{x}_j\|}
\end{aligned}$$

Now that we have found the feature i that maximizes the length of the projection, we must update corresponding weight i and the residual vector. The updated residual $\mathbf{r}^t$ is the result of projecting $\mathbf{r}^{t-1}$ onto feature $\mathbf{x}_i$:

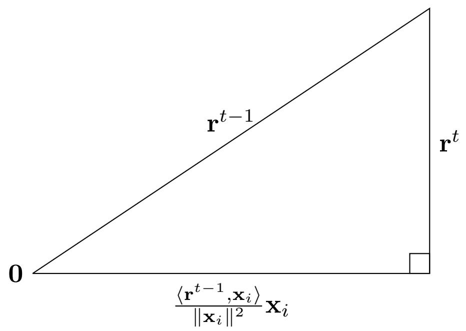

$\mathbf{r}^{t-1}$
$\mathbf{r}^t$
$\mathbf{0}$
$\frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_i\rangle}{\|\mathbf{x}_i\|^2}\mathbf{x}_i$

Figure 2.3: The updated residual $\mathbf{r}^t$, current residual $\mathbf{r}^{t-1}$, and scaled feature $\mathbf{x}_i$ form a right triangle.

The updated residual is given by

$$\mathbf{r}^t = \mathbf{r}^{t-1} - \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_i\rangle}{\|\mathbf{x}_i\|^2}\mathbf{x}_i$$

which corresponds to adding $\frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_i\rangle}{\|\mathbf{x}_i\|^2}$ to the corresponding weight w_i^t:

$$w_i^t = w_i^{t-1} + \frac{\langle \mathbf{r}^{t-1}, \mathbf{x}_i\rangle}{\|\mathbf{x}_i\|^2}$$

We update w_i to the optimum projection value and repeat greedily at each iteration. At each iteration, the (squared) length of the residual $\|\mathbf{X}\mathbf{w}^t - \mathbf{y}\|^2$ monotonically decreases since $\|\mathbf{r}^{t-1}\|^2 > \|\mathbf{r}^t\|^2$. While matching pursuit is not guaranteed to find the optimal $\mathbf{w}^*$, in practice it works well for most applications.

Orthogonal Matching Pursuit

The **Orthogonal Matching Pursuit (OMP)** algorithm is an extension to the standard Matching Pursuit algorithm with the following difference: at iteration t, we maintain a set I^t of all features selected by the algorithm so far, and instead of updating just the one weight corresponding to feature i found at iteration t, we update all weights corresponding to the features in I^t using Least Squares.

Algorithm 3: Orthogonal Matching Pursuit

```
initialize the weights $\mathbf{w}^0 = \mathbf{0}$ and the residual $\mathbf{r}^0 = \mathbf{y} - \mathbf{X}\mathbf{w}^0 = \mathbf{y}$
initialize a set of features $I^0 = \emptyset$
while $\|\mathbf{w}\|_0 < k$ do
    find the feature $i$ for which the length of the projected residual onto $\mathbf{x}_i$ is maximized:
        $$i = \arg\min_j (\min_\nu \|\mathbf{r}^{t-1} - \nu\mathbf{x}_j\|) = \arg\max_j \frac{|\langle \mathbf{r}^{t-1}, \mathbf{x}_j \rangle|}{\|\mathbf{x}_j\|}$$
    add feature $i$ to the set of features:
        $$I^t = I^{t-1} \cup \{i\}$$
    Estimate the best linear fit of the target $\mathbf{y}$ using the features obtained so far. Given that
    we have found $t$ good features, we now find the best linear fit for the target $\mathbf{y}$ using these
    $t$-features. Define $\mathbf{X}_t = [\mathbf{x}_{i^1}, \dots, \mathbf{x}_{i^t}]$ made up of these $t$-features. Then we determine $\mathbf{w}^t$
    as the solution for the following least-squares problem:
        $$\mathbf{w}^t = \arg\min_{\mathbf{w} \in \mathbb{R}^t} \|\mathbf{y} - \mathbf{X}_t\mathbf{w}\|_2^2$$
    update the residual vector: $\mathbf{r}^t = \mathbf{y} - \mathbf{X}\mathbf{w}^t$
```

The motivation for OMP is as follows: if we do not refit on all features updated so far after choosing a new feature, the new residual will not necessarily be orthogonal to the span of the canonical basis vectors corresponding to those chosen coordinates, and is therefore not optimal. The OMP algorithm ensures that $\mathbf{w}^t$ corresponds to an optimal Least Squares solution if we restricted our features to just those in I^t.

2.7 Total Least Squares

Previously, we have covered **Ordinary Least Squares (OLS)** which assumes that the dependent variable y is noisy but the independent variables $\mathbf{x}$ are noise-free. We now discuss **Total Least Squares (TLS)**, where we assume that our independent variables are also corrupted by noise. For this reason, TLS is considered an **errors-in-variables** model.

A probabilistic motivation?

We might begin with a probabilistic formulation and fit the parameters via maximum likelihood estimation, as before. Consider for simplicity a one-dimensional linear model

$$y_{\text{true}} = wx_{\text{true}}$$

where the observations we receive are corrupted by Gaussian noise

$$(x, y) = (x_{\text{true}} + z_x, y_{\text{true}} + z_y) \qquad\qquad z_x, z_y \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$$

Combining the previous two relations, we obtain

$$\begin{aligned} y &= y_{\text{true}} + z_y \\ &= w x_{\text{true}} + z_y \\ &= w(x - z_x) + z_y \\ &= wx \underbrace{-wz_x + z_y}_{\sim \mathcal{N}(0, w^2+1)} \end{aligned}$$

The likelihood for a single point is then given by

$$P(x, y; w) = \frac{1}{\sqrt{2\pi(w^2+1)}} \exp\left(-\frac{1}{2}\frac{(y - wx)^2}{w^2+1}\right)$$

Thus the log likelihood is

$$\log P(x, y; a) = \text{constant} - \frac{1}{2}\log\Big(w^2+1\Big) - \frac{1}{2}\frac{(y-wx)^2}{w^2+1}$$

Observe that the parameter w shows up in three places, unlike the form that we are familiar with, where it only appears in the quadratic term. Our usual strategy of setting the derivative equal to zero to find a maximizer will not yield a nice system of linear equations in this case, so we'll try a different approach.

Low-rank formulation

To solve the TLS problem, we develop another formulation that can be solved using the singular value decomposition. To motivate this formulation, recall that in OLS we attempt to minimize $\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$, which is equivalent to

$$\min_{\mathbf{w}, \boldsymbol{\epsilon}} \|\boldsymbol{\epsilon}\|_2^2 \quad \text{subject to} \quad \mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$$

This only accounts for errors in the dependent variable, so for TLS we introduce a second residual $\epsilon_{\mathbf{X}} \in \mathbb{R}^{n\times d}$ to account for independent variable error:

$$\min_{\mathbf{w}, \epsilon_{\mathbf{X}}, \boldsymbol{\epsilon}_{\mathbf{y}}} \left\| \begin{bmatrix} \epsilon_{\mathbf{X}} & \boldsymbol{\epsilon}_{\mathbf{y}} \end{bmatrix} \right\|_{\text{F}}^2 \quad \text{subject to} \quad (\mathbf{X} + \epsilon_{\mathbf{X}})\mathbf{w} = \mathbf{y} + \boldsymbol{\epsilon}_{\mathbf{y}}$$

For comparison to the OLS case, note that the Frobenius norm is essentially the same as the 2-norm, just applied to the elements of a matrix rather than a vector.

From a probabilistic perspective, finding the most likely value of a Gaussian corresponds to minimizing the squared distance from the mean. Since we assume the noise is 0-centered, we want to minimize the sum of squares of each entry in the error matrix, which corresponds exactly to minimizing the Frobenius norm.

In order to separate out the terms being minimized, we rearrange the constraint equation as

$$\underbrace{\begin{bmatrix} \mathbf{X} + \epsilon_{\mathbf{X}} & \mathbf{y} + \boldsymbol{\epsilon}_{\mathbf{y}} \end{bmatrix}}_{\in \mathbb{R}^{n\times(d+1)}} \begin{bmatrix} \mathbf{w} \\ -1 \end{bmatrix} = \mathbf{0}$$

This expression tells us that the vector $\begin{bmatrix}\mathbf{w}^\top & -1\end{bmatrix}^\top$ lies in the nullspace of the matrix on the left. However, if the matrix is full rank, its nullspace contains only $\mathbf{0}$, and thus the equation cannot be satisfied (since the last component, -1, is always nonzero). Therefore we must choose the perturbations $\epsilon_\mathbf{X}$ and $\boldsymbol{\epsilon}_\mathbf{y}$ in such a way that the matrix is not full rank.

It turns out that there is a mathematical result, the **Eckart-Young theorem**, that can help us pick these perturbations. This theorem essentially says that the best low-rank approximation (in terms of the Frobenius norm[2]) is obtained by throwing away the smallest singular values.

Theorem. *Suppose* $\mathbf{A} \in \mathbb{R}^{m\times n}$ *has rank* $r \leq \min(m,n)$, *and let* $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top = \sum_{i=1}^r \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\top$ *be its singular value decomposition. Then*

$$\mathbf{A}_k = \sum_{i=1}^k \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\top = \mathbf{U}\begin{bmatrix}\sigma_1 & \cdots & 0 & \cdots & 0\\ \vdots & \ddots & 0 & \cdots & 0\\ 0 & 0 & \sigma_k & \cdots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & 0 & \cdots & 0\end{bmatrix}\mathbf{V}^\top$$

where $k \leq r$, *is the best rank-k approximation to* $\mathbf{A}$ *in the sense that*

$$\|\mathbf{A} - \mathbf{A}_k\|_\text{F} \leq \|\mathbf{A} - \tilde{\mathbf{A}}\|_\text{F}$$

for any $\tilde{\mathbf{A}}$ *such that* $\operatorname{rank}(\tilde{\mathbf{A}}) \leq k$.

Let us assume that the data matrix $\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}$ is full rank.[3] Write its singular value decomposition:

$$\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix} = \sum_{i=1}^{d+1} \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\top$$

Then the Eckart-Young theorem tells us that the best rank-d approximation to this matrix is

$$\begin{bmatrix}\mathbf{X} + \epsilon_\mathbf{X} & \mathbf{y} + \boldsymbol{\epsilon}_\mathbf{y}\end{bmatrix} = \sum_{i=1}^{d} \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\top$$

which is achieved by setting

$$\begin{bmatrix}\epsilon_\mathbf{X} & \boldsymbol{\epsilon}_\mathbf{y}\end{bmatrix} = -\sigma_{d+1}\mathbf{u}_{d+1}{\mathbf{v}_{d+1}}^\top$$

The nullspace of our resulting matrix is then

$$\operatorname{null}\left(\begin{bmatrix}\mathbf{X} + \epsilon_\mathbf{X} & \mathbf{y} + \boldsymbol{\epsilon}_\mathbf{y}\end{bmatrix}\right) = \operatorname{null}\left(\sum_{i=1}^{d} \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\top\right) = \operatorname{span}\{\mathbf{v}_{d+1}\}$$

where the last equality holds because $\{\mathbf{v}_1, \dots, \mathbf{v}_{d+1}\}$ form an orthogonal basis for $\mathbb{R}^{d+1}$. To get the weight $\mathbf{w}$, we find a scaling α such that $\begin{bmatrix}\mathbf{w}^\top & -1\end{bmatrix}^\top$ is in the nullspace, i.e.

$$\begin{bmatrix}\mathbf{w}\\ -1\end{bmatrix} = \alpha\mathbf{v}_{d+1}$$

Note that this requires the $(d+1)$st component of $\mathbf{v}_{d+1}$ to be nonzero. (See Sectionfor details.)

[2] There is a more general version that holds for any unitary invariant norm.

[3] This should be the case in practice because the noise will cause $\mathbf{y}$ not to lie in the columnspace of $\mathbf{X}$.

Noise, regularization, and reverse-regularization

In a sense, above we have solved the problem of total least squares by reducing it to computing an appropriate SVD. Once we have $\mathbf{v}_{d+1}$, or any scalar multiple of it, we simply rescale it so that the last component is -1, and then the first d components give us $\mathbf{w}$. However, we can look at this more closely to uncover the relationship between TLS and the ideas of regularization that we have seen earlier in the course.

Since $\mathbf{v}_{d+1}$ is a right-singular vector of $\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}$, it is an eigenvector of the matrix

$$\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}^\top \begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix} = \begin{bmatrix}\mathbf{X}^\top\mathbf{X} & \mathbf{X}^\top\mathbf{y} \\ \mathbf{y}^\top\mathbf{X} & \mathbf{y}^\top\mathbf{y}\end{bmatrix}$$

So to find it we solve

$$\begin{bmatrix}\mathbf{X}^\top\mathbf{X} & \mathbf{X}^\top\mathbf{y} \\ \mathbf{y}^\top\mathbf{X} & \mathbf{y}^\top\mathbf{y}\end{bmatrix}\begin{bmatrix}\mathbf{w} \\ -1\end{bmatrix} = \sigma_{d+1}^2\begin{bmatrix}\mathbf{w} \\ -1\end{bmatrix}$$

From the top line we see that $\mathbf{w}$ satisfies

$$\mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{X}^\top\mathbf{y} = \sigma_{d+1}^2\mathbf{w}$$

which can be rewritten as

$$(\mathbf{X}^\top\mathbf{X} - \sigma_{d+1}^2\mathbf{I})\mathbf{w} = \mathbf{X}^\top\mathbf{y}$$

Thus, assuming $\mathbf{X}^\top\mathbf{X} - \sigma_{d+1}^2\mathbf{I}$ is invertible (see the next section), we can solve for the weights as

$$\hat{\mathbf{w}}_{\text{TLS}} = (\mathbf{X}^\top\mathbf{X} - \sigma_{d+1}^2\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$$

This result is like ridge regression, but with a *negative* regularization constant!

Why does this make sense? One of the original motivations of ridge regression was to ensure that the matrix being inverted is in fact nonsingular, and subtracting a scalar multiple of the identity seems like a step in the opposite direction. We can make sense of this by recalling our original model:

$$\mathbf{X} = \mathbf{X}_{\text{true}} + \mathbf{Z}$$

where $\mathbf{X}_{\text{true}}$ are the actual values before noise corruption, and $\mathbf{Z}$ is a zero-mean noise term with i.i.d. entries. Then

$$\begin{aligned}
\mathbb{E}[\mathbf{X}^\top\mathbf{X}] &= \mathbb{E}[(\mathbf{X}_{\text{true}} + \mathbf{Z})^\top(\mathbf{X}_{\text{true}} + \mathbf{Z})] \\
&= \mathbb{E}[\mathbf{X}_{\text{true}}{}^\top\mathbf{X}_{\text{true}}] + \mathbb{E}[\mathbf{X}_{\text{true}}{}^\top\mathbf{Z}] + \mathbb{E}[\mathbf{Z}^\top\mathbf{X}_{\text{true}}] + \mathbb{E}[\mathbf{Z}^\top\mathbf{Z}] \\
&= \mathbf{X}_{\text{true}}{}^\top\mathbf{X}_{\text{true}} + \mathbf{X}_{\text{true}}{}^\top\underbrace{\mathbb{E}[\mathbf{Z}]}_{\mathbf{0}} + \underbrace{\mathbb{E}[\mathbf{Z}]}_{\mathbf{0}}{}^\top\mathbf{X}_{\text{true}} + \mathbb{E}[\mathbf{Z}^\top\mathbf{Z}] \\
&= \mathbf{X}_{\text{true}}{}^\top\mathbf{X}_{\text{true}} + \mathbb{E}[\mathbf{Z}^\top\mathbf{Z}]
\end{aligned}$$

Observe that the off-diagonal terms of $\mathbb{E}[\mathbf{Z}^\top\mathbf{Z}]$ terms are zero because the ith and jth rows of $\mathbf{Z}$ are independent for $i \neq j$, and the on-diagonal terms are essentially variances. Thus the $-\sigma_{d+1}^2\mathbf{I}$ term is there to compensate for the extra noise introduced by our assumptions regarding the independent variables.

For another perspective, note that

$$\mathbb{E}[\mathbf{X}^\top] = \mathbb{E}[(\mathbf{X}_{\text{true}} + \mathbf{Z})^\top] = \mathbb{E}[\mathbf{X}_{\text{true}}{}^\top + \mathbf{Z}^\top] = \mathbb{E}[\mathbf{X}_{\text{true}}{}^\top] + \mathbb{E}[\mathbf{Z}^\top] = \mathbf{X}_{\text{true}}{}^\top$$

If we plug this into the OLS solution (where we have assumed no noise in the independent variables), we see

$$\hat{\mathbf{w}}_{\text{OLS}} = ({\mathbf{X}_{\text{true}}}^\top \mathbf{X}_{\text{true}})^{-1} {\mathbf{X}_{\text{true}}}^\top \mathbf{y} = (\mathbb{E}[\mathbf{X}^\top \mathbf{X}] - \mathbb{E}[\mathbf{Z}^\top \mathbf{Z}])^{-1} \mathbb{E}[\mathbf{X}]^\top \mathbf{y}$$

which strongly resembles the TLS solution, but expressed in terms of expectations over the noise $\mathbf{Z}$.

So, is this all just a mathematical trick or is there a practical sense in which ridge regularization itself is related to adding noise? The math above suggests that we can take the original training data set and instead of working with that data set, just sample lots of points (say r times each) with i.i.d. zero-mean Gaussian noise with variance λ added to each of their features. Call this the $\mathbf{X}$ and have the corresponding $\mathbf{y}$ just keep the original y values. Then, doing ordinary least squares on this noisily degraded data set will end up behaving like ridge regression since the laws of large numbers will make $\frac{1}{r}\mathbf{X}^\top \mathbf{X}$ concentrate around ${\mathbf{X}_{\text{true}}}^\top \mathbf{X}_{\text{true}} + \lambda I$. Meanwhile $\mathbf{X}^\top \mathbf{y}$ will concentrate to $r{\mathbf{X}_{\text{true}}}^\top \mathbf{y}_{orig}$ with $O(\sqrt{r})$ noise on top of this by the Central Limit Theorem (if we used other-than-Gaussian noise to noisily resample), and straight variance-$O(r)$ Gaussian noise if we indeed used Gaussian noise. Putting them together means that the result of OLS with noisily augmented training data will result in approximately the same solution as ridge-regression, with the solutions approaching each other as the number of noisy copies r goes to infinity.

Why does this make intuitive sense? How can adding noise make learning more reliable? The intuitive reason is that this added noise destroys inadvertent conspiracies. Overfitting happens because the learning algorithm sees some degree of conspiracies between the observed training labels y and the input features. By adding lots of copies of the training data with additional noise added into them, many of these conspiracies will be masked by the added noise because they are fundamentally sensitive to small details — this is why they manifest as large weights $\mathbf{w}$. We know from our studies of the bias/variance tradeoff that having more training samples reduces this variance. Adding our own noisy samples exploits this variance reduction.

In many practical machine learning situations, appropriately adding noise to your training data can be an important tool in helping generalization performance.

Existence of the solution

In the discussion above, we have in some places made assumptions to move the derivation forward. These do not always hold, but we can provide sufficient conditions for the existence of a solution.

Proposition. *Let $\sigma_1, \dots, \sigma_{d+1}$ denote the singular values of $\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}$, and $\tilde{\sigma}_1, \dots, \tilde{\sigma}_d$ denote the singular values of $\mathbf{X}$. If $\sigma_{d+1} < \tilde{\sigma}_d$, then the total least squares problem has a solution, given by*

$$\hat{\mathbf{w}}_{\text{TLS}} = (\mathbf{X}^\top \mathbf{X} - \sigma_{d+1}^2 \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

Proof. Let $\sum_{i=1}^{d+1} \sigma_i \mathbf{u}_i {\mathbf{v}_i}^\top$ be the SVD of $\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}$, and suppose $\sigma_{d+1} < \tilde{\sigma}_d$. We first show that the $(d+1)$st component of $\mathbf{v}_{d+1}$ is nonzero. To this end, suppose towards a contradiction that $\mathbf{v}_{d+1} = \begin{bmatrix}\mathbf{a}^\top & 0\end{bmatrix}^\top$ for some $\mathbf{a} \neq \mathbf{0}$. Since $\mathbf{v}_{d+1}$ is a right-singular vector of $\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}$, i.e. an eigenvector of $\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}^\top \begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}$, we have

$$\begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix}^\top \begin{bmatrix}\mathbf{X} & \mathbf{y}\end{bmatrix} \begin{bmatrix}\mathbf{a} \\ 0\end{bmatrix} = \begin{bmatrix}\mathbf{X}^\top \mathbf{X} & \mathbf{X}^\top \mathbf{y} \\ \mathbf{y}^\top \mathbf{X} & \mathbf{y}^\top \mathbf{y}\end{bmatrix} \begin{bmatrix}\mathbf{a} \\ 0\end{bmatrix} = \sigma_{d+1}^2 \begin{bmatrix}\mathbf{a} \\ 0\end{bmatrix}$$

Then

$$\mathbf{X}^\top \mathbf{X} \mathbf{a} = \sigma_{d+1}^2 \mathbf{a}$$

i.e. $\mathbf{a}$ is an eigenvector of $\mathbf{X}^\top\mathbf{X}$ with eigenvalue σ_{d+1}^2. However, this contradicts the fact that

$$\tilde{\sigma}_d^2 = \lambda_{\min}(\mathbf{X}^\top\mathbf{X})$$

since we have assumed $\sigma_{d+1} < \tilde{\sigma}_d$. Therefore the $(d+1)$st component of $\mathbf{v}_{d+1}$ is nonzero, which guarantees the existence of a solution.

We have already derived the given expression for $\hat{\mathbf{w}}_{\text{TLS}}$, but it remains to show that the matrix $\mathbf{X}^\top\mathbf{X} - \sigma_{d+1}^2\mathbf{I}$ is invertible. This is fairly immediate from the assumption that $\sigma_{d+1} < \tilde{\sigma}_d$, since this implies

$$\sigma_{d+1}^2 < \tilde{\sigma}_d^2 = \lambda_{\min}(\mathbf{X}^\top\mathbf{X})$$

giving

$$\lambda_{\min}(\mathbf{X}^\top\mathbf{X} - \sigma_{d+1}^2\mathbf{I}) = \lambda_{\min}(\mathbf{X}^\top\mathbf{X}) - \sigma_{d+1}^2 > 0$$

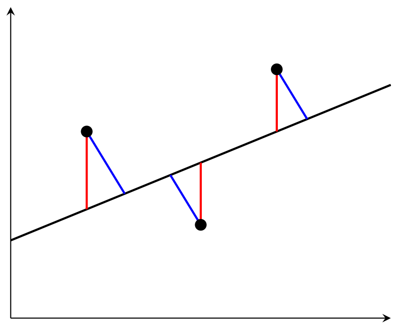
which guarantees that the matrix is invertible. □

This gives us a nice mathematical characterization of the existence of a solution, showing that the two technical requirements we raised earlier (the last entry of $\mathbf{v}_{d+1}$ being nonzero, and the matrix $\mathbf{X}^\top\mathbf{X} - \sigma_{d+1}^2$ being invertible) happen together. However, is the assumption of the proof likely to hold in practice? We give an intuitive argument that it is.

Consider that in solving the TLS problem, we have determined the error term $\epsilon_\mathbf{X}$. In principle, we could use this to denoise $\mathbf{X}$, as in $\hat{\mathbf{X}}_{\text{true}} = \mathbf{X} - \epsilon_\mathbf{X}$, and then perform OLS as normal. This process is essentially the same as TLS if we compare the original formulations. Assuming the error is drawn from a continuous distribution, the probability that the denoised matrix $\hat{\mathbf{X}}_{\text{true}}$ has collinear columns is zero.

TLS minimizes perpendicular distance

Recall that OLS tries to minimize the vertical distance between the fitted line and data points. TLS, on the other hand, tries to minimize the perpendicular distance. For this reason, TLS may sometimes be referred to as **orthogonal regression**.



The red lines represent vertical distance, which OLS aims to minimize. The blue lines represent perpendicular distance, which TLS aims to minimize. Note that all blue lines are perpendicular to the black line (hypothesis model), while all red lines are perpendicular to the x axis.

Chapter 3

Dimensionality Reduction

In machine learning, the data we have are often very high-dimensional. In fact, when we introduced the idea of features (like polynomial features), these made the dimensionality of the data even higher. The kernel trick was something that let us partially deal with this by working with vectors only as long as there are training samples.

However, there are a number of reasons why we might want to work with a lower-dimensional representation:

- Visualization (if we can get it down to 2 or 3 dimensions), e.g. for exploratory data analysis
- Reduce computational load
- Reduce variance in estimation — regularize the problem

So, how can we reduce the dimensionality of data? There are obvious ways — just keeping a subset of features. But which features? In general, that presumably depends on what you are trying to do. What could you do if you didn't know what you were trying to predict with those features? This corresponds to unsupervised dimensionality reduction. There are a couple of intuitive choices. First, just pick some features at random to keep. This is appealing for its symmetry, but it makes you wonder if we could do better by actually looking at the data before deciding which features to keep.

Consequently, another thing that you could do is to just keep the few features that have the most variability — which you could measure by the variance of that feature. But what if two of the most variable features were actually very correlated to each other? Should we really be including both of them? Maybe we should focus on "fresh" variability somehow. To do this, maybe it would be helpful to allow ourselves to synthesize linear combinations of features and keep some of these synthesized features.

3.1 Principal Component Analysis

Principal Component Analysis (PCA) is exactly such an unsupervised dimensionality reduction technique. Given a matrix of data points, it finds one or more orthogonal directions that capture the largest amount of variance in the data. Intuitively, the directions with less variance contain less information and may be discarded without introducing too much error. One of the practical motivations for taking this kind of unsupervised approach to dimensionality reduction is that labeled training data might be hard or expensive to get, but unlabeled training data (i.e. no y just $\mathbf{x}$) might be more easily available. PCA is able to extract meaningful directions from such unlabeled data.

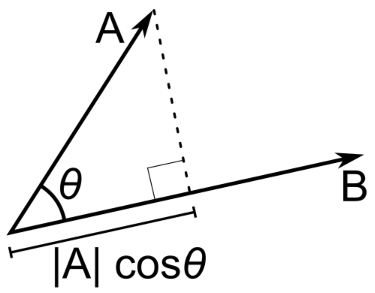
Not coincidentally, PCA turns out to be intimately connected to the ideas of Total Least Squares.

Projection

Let us first review the meaning of scalar projection of one vector onto another. If $\mathbf{v} \in \mathbb{R}^d$ is a unit vector, i.e. $\|\mathbf{v}\| = 1$, then the scalar projection of another vector $\mathbf{x} \in \mathbb{R}^d$ onto $\mathbf{v}$ is given by $\mathbf{x}^\top \mathbf{v}$. This quantity tells us roughly how much of the projected vector $\mathbf{x}$ lies along the direction given direction $\mathbf{v}$. Why does this expression make sense? Recall the slightly more general formula which holds for vectors of any length:

$$\mathbf{x}^\top \mathbf{v} = \|\mathbf{x}\| \|\mathbf{v}\| \cos\theta$$

where θ is the angle between the vectors. In this case, since $\|\mathbf{v}\| = 1$, the expression simplifies to $\mathbf{x}^\top \mathbf{v} = \|\mathbf{x}\| \cos\theta$. But since cosine gives the ratio of the adjacent side (the projection we want to find) to the hypotenuse ($\|\mathbf{x}\|$), this is exactly what we want:



One approach to dimensionality reduction by using projections is to choose projections at random — sample from an iid Gaussian and then normalize the vector to get our $\mathbf{v}$. This creates a degree of fairness across the individual features since an iid Gaussian is uniform over directions in d-dimensional space. As you will see in homework, this approach to dimensionality reduction actually has many interesting properties. By construction, however, it does not look at any data itself and thus is unable to prioritize important vs unimportant feature directions.

The first principal component

Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be our matrix of data, where each row is a d-dimensional datapoint. These are to be thought of as i.i.d. samples from some random vector $\mathbf{x}$.

We will assume that the data points have mean zero; if this is not the case, we can make it so by subtracting the average of all the rows, $\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^n \mathbf{x}_i$, from each row. The motivation for this is that we want to find directions of high variance within the data, and variance is defined relative to the mean of the data. If we did not zero-center the data, the directions found would be heavily influenced by where the data lie relative to the origin, rather than where they lie relative to the other data, which is more useful.

Since $\mathbf{X}$ is zero-mean, the sample variance of the datapoints' projections onto a unit vector $\mathbf{v}$ is given by

$$\frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i{}^\top \mathbf{v})^2 = \frac{1}{n}\|\mathbf{X}\mathbf{v}\|^2 = \frac{1}{n}\mathbf{v}^\top \mathbf{X}^\top \mathbf{X}\mathbf{v}$$

where $\mathbf{v}$ is constrained to have unit norm.[1]

With this motivation, we define the **first loading vector** $\mathbf{v}_1$ as the solution to the constrained optimization problem

$$\max_{\mathbf{v}} \mathbf{v}^\top \mathbf{X}^\top \mathbf{X}\mathbf{v} \quad \text{subject to} \quad \mathbf{v}^\top\mathbf{v} = 1$$

Note that we have discarded the positive constant factor $1/n$ which does not affect the optimal value of $\mathbf{v}$.

To reduce this constrained optimization problem to an unconstrained one, we write down its Lagrangian:

$$\mathcal{L}(\mathbf{v}) = \mathbf{v}^\top \mathbf{X}^\top \mathbf{X}\mathbf{v} - \lambda(\mathbf{v}^\top\mathbf{v} - 1)$$

First-order necessary conditions for optima imply that

$$\mathbf{0} = \nabla\mathcal{L}(\mathbf{v}_1) = 2\mathbf{X}^\top\mathbf{X}\mathbf{v}_1 - 2\lambda\mathbf{v}_1$$

Hence $\mathbf{X}^\top\mathbf{X}\mathbf{v}_1 = \lambda\mathbf{v}_1$, i.e. $\mathbf{v}_1$ is an eigenvector of $\mathbf{X}^\top\mathbf{X}$ with eigenvalue λ. Since we constrain ${\mathbf{v}_1}^\top\mathbf{v}_1 = 1$, the value of the objective is precisely

$${\mathbf{v}_1}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v}_1 = {\mathbf{v}_1}^\top(\lambda\mathbf{v}_1) = \lambda{\mathbf{v}_1}^\top\mathbf{v}_1 = \lambda$$

so the optimal value is $\lambda = \lambda_{\max}(\mathbf{X}^\top\mathbf{X})$, which is achieved when $\mathbf{v}_1$ is a unit eigenvector of $\mathbf{X}^\top\mathbf{X}$ corresponding to its largest eigenvalue.

Finding more principal components

We have seen how to find the first loading vector, which is the unit vector that maximizes the variance of the projected data points. However, in most applications, we want to find more than one direction. We want the subsequent directions found to also be directions of high variance, but they ought to be orthogonal to the existing directions in order to minimize redundancy in the information captured. Thus we define the kth loading vector $\mathbf{v}_k$ as the solution to the constrained optimization problem

$$\begin{aligned}\max_{\mathbf{v}} \mathbf{v}^\top \mathbf{X}^\top \mathbf{X}\mathbf{v} \quad \text{subject to} \quad & \mathbf{v}^\top\mathbf{v} = 1 \\ & \mathbf{v}^\top\mathbf{v}_i = 0, \quad i = 1, \dots, k-1\end{aligned}$$

We claim that $\mathbf{v}_k$ is a unit eigenvector of $\mathbf{X}^\top\mathbf{X}$ corresponding to its kth largest eigenvalue.

Proof. By induction on k. We have already shown that the claim is true for the base case $k = 1$ (where there are no orthogonality constraints). Now assume that it is true for the first k loading vectors $\mathbf{v}_1, \dots, \mathbf{v}_k$, and consider the problem of finding $\mathbf{v}_{k+1}$.

[1] To make sense of the sample variance, recall that for any random variable Z,

$$\mathrm{Var}(Z) = \mathbb{E}[(Z - \mathbb{E}[Z])^2]$$

so if $\mathbb{E}[Z] = 0$ then $\mathrm{Var}(Z) = \mathbb{E}[Z^2]$. In practice we will not have the true random variable Z, but rather i.i.d. observations $z_1, \dots, z_n$ of Z. The expected value can then be approximated by a sample average, i.e.

$$\mathbb{E}[Z^2] \approx \frac{1}{n}\sum_{i=1} z_i^2$$

which is justified by the law of large numbers, which states that (under mild conditions) the sample average converges to the expected value as $n \to \infty$. In our case the random variable Z is the principal component $\mathbf{v}^\top\mathbf{x}$, and the i.i.d. observations are the projections of our datapoints, i.e. $z_i = \mathbf{v}^\top\mathbf{x}_i$.

By the inductive hypothesis, we know that $\mathbf{v}_1, \dots, \mathbf{v}_k$ are orthonormal eigenvectors of $\mathbf{X}^\top\mathbf{X}$. Denote the ith largest eigenvalue of $\mathbf{X}^\top\mathbf{X}$ by λ_i, noting that $\mathbf{X}^\top\mathbf{X}\mathbf{v}_i = \lambda_i\mathbf{v}_i$.

The Lagrangian of the objective function is

$$\mathcal{L}(\mathbf{v}) = \mathbf{v}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v} - \lambda(\mathbf{v}^\top\mathbf{v} - 1) + \sum_{i=1}^{k} \eta_i \mathbf{v}^\top\mathbf{v}_i$$

First-order necessary conditions for optima imply that

$$\mathbf{0} = \nabla\mathcal{L}(\mathbf{v}_{k+1}) = 2\mathbf{X}^\top\mathbf{X}\mathbf{v}_{k+1} - 2\lambda\mathbf{v}_{k+1} + \sum_{i=1}^{k} \eta_i \mathbf{v}_i$$

This implies that, if $\mathbf{v}_{k+1}$ is orthogonal to $\mathbf{v}_1, \dots, \mathbf{v}_k$ (as we constrain it to be), then

$$\begin{aligned}
0 &= {\mathbf{v}_j}^\top\mathbf{0} \\
&= 2{\mathbf{v}_j}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v}_{k+1} - 2\lambda \underbrace{{\mathbf{v}_j}^\top\mathbf{v}_{k+1}}_{0} + \sum_{i=1}^{k} \eta_i \underbrace{{\mathbf{v}_j}^\top\mathbf{v}_i}_{\delta_{ij}} \\
&= 2(\mathbf{X}^\top\mathbf{X}\mathbf{v}_j)^\top\mathbf{v}_{k+1} + \eta_j \\
&= 2(\lambda_j\mathbf{v}_j)^\top\mathbf{v}_{k+1} + \eta_j \\
&= 2\lambda_j \underbrace{{\mathbf{v}_j}^\top\mathbf{v}_{k+1}}_{0} + \eta_j \\
&= \eta_j
\end{aligned}$$

for all $j = 1, \dots, k$.

Plugging these values back into the optimality equation above, we see that $\mathbf{v}_{k+1}$ must satisfy $\mathbf{X}^\top\mathbf{X}\mathbf{v}_{k+1} = \lambda\mathbf{v}_{k+1}$, i.e. $\mathbf{v}_{k+1}$ is an eigenvector of $\mathbf{X}^\top\mathbf{X}$ with eigenvalue λ. As before, the value of the objective function is then λ. To maximize, we want the largest eigenvalue, but we must respect the constraints that $\mathbf{v}_{k+1}$ is orthogonal to $\mathbf{v}_1, \dots, \mathbf{v}_k$. Clearly if $\mathbf{v}_{k+1}$ is equal to any of these eigenvectors (up to sign), then one of these constraints will not be satisfied. Thus to maximize the expression, $\mathbf{v}_{k+1}$ should be a unit eigenvector of $\mathbf{X}^\top\mathbf{X}$ corresponding to its $(k+1)$st largest eigenvalue. By the spectral theorem, we can always choose this vector in such a way that it is orthogonal to $\mathbf{v}_1, \dots, \mathbf{v}_k$, so we are done. □

We have shown that the loading vectors are orthonormal eigenvectors of $\mathbf{X}^\top\mathbf{X}$. In other words, they are right-singular vectors of $\mathbf{X}$, so they can all be found simultaneously by computing the SVD of $\mathbf{X}$.

Projecting onto the PCA coordinate system

Once we have computed the loading vectors, we can use them as a new coordinate system. The kth principal component of a datapoint $\mathbf{x}_i \in \mathbb{R}^d$ is defined as the scalar projection of $\mathbf{x}_i$ onto the kth loading vector $\mathbf{v}_k$, i.e. ${\mathbf{x}_i}^\top\mathbf{v}_k$. We can compute all the principal components of all the datapoints at once using a matrix-matrix multiplication:

$$\mathbf{Z}_k = \mathbf{X}\mathbf{V}_k$$

where $\mathbf{V}_k \in \mathbb{R}^{d\times k}$ is a matrix whose columns are the first k loading vectors $\mathbf{v}_1, \dots, \mathbf{v}_k$.

Below we plot the result of such a projection in the case $d = k = 2$:

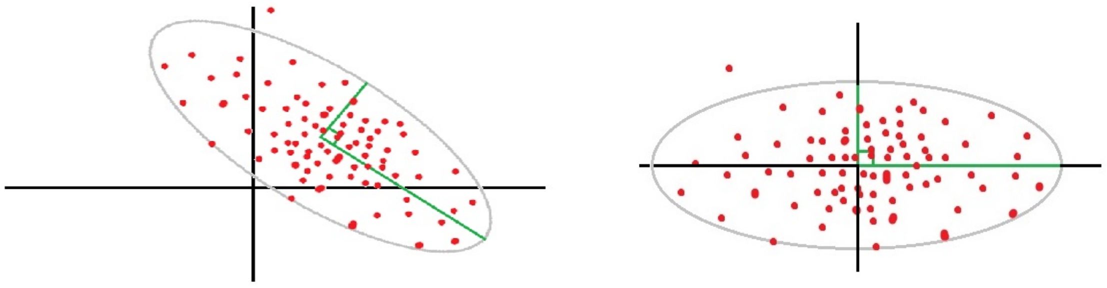


Figure 3.1: Left:data points; Right: PCA projection of data points

Observe that the data are uncorrelated in the projected space. Also note that this example does not show the power of PCA since we have not reduced the dimensionality of the data at all – the plot is merely to show the PCA coordinate transformation.

Once we've computed the principal components, we can approximately reconstruct the original points by

$$\tilde{\mathbf{X}}_k = \mathbf{Z}_k \mathbf{V}_k{}^\top = \mathbf{X}\mathbf{V}_k \mathbf{V}_k{}^\top$$

The rows of $\tilde{\mathbf{X}}_k$ are the projections of the original rows of $\mathbf{X}$ onto the subspace spanned by the loading vectors.

Other derivations of PCA

We have given the most common derivation of PCA above, but it turns out that there are other ways to solve the optimization problem, or to arrive at the same formulation. These give us helpful additional perspectives on what PCA is doing.

Changing coordinates

In PCA we want to find the unit length $\mathbf{v}$ that maximizes $\mathbf{v}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v}$. It turns out that there is a result, sometimes referred to as the **variational characterization of eigenvalues**, that tells us which vectors $\mathbf{v}$ achieve this. The key idea in the proof is a length-preserving change of coordinates.

Theorem. *Let $\mathbf{A} \in \mathbb{R}^{d\times d}$ be symmetric. Then for any $\mathbf{v} \in \mathbb{R}^d$ satisfying $\|\mathbf{v}\|_2 = 1$,*

$$\lambda_{\min}(\mathbf{A}) \le \mathbf{v}^\top\mathbf{A}\mathbf{v} \le \lambda_{\max}(\mathbf{A})$$

where for both bounds, equality holds if and only if $\mathbf{v}$ is a corresponding eigenvector.

Proof. We show only the max case because the argument for the min case is entirely analogous.

Since $\mathbf{A}$ is symmetric, we can decompose it as $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$, where $\mathbf{Q} \in \mathbb{R}^{d\times d}$ is orthogonal and $\mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \dots, \lambda_d)$ contains the eigenvalues of $\mathbf{A}$. For any $\mathbf{v}$ satisfying $\|\mathbf{v}\|_2 = 1$, define $\mathbf{z} = \mathbf{Q}^\top\mathbf{v}$, noting that the relationship between $\mathbf{v}$ and $\mathbf{z}$ is one-to-one because $\mathbf{Q}$ is invertible and that $\|\mathbf{z}\|_2 = 1$ because $\mathbf{Q}$ is orthogonal. Hence

$$\max_{\|\mathbf{v}\|_2=1} \mathbf{v}^\top\mathbf{A}\mathbf{v} = \max_{\|\mathbf{z}\|_2=1} \mathbf{z}^\top\mathbf{\Lambda}\mathbf{z} = \max_{\|\mathbf{z}\|_2^2=1} \sum_{i=1}^{d} \lambda_i z_i^2$$

We note that

$$\sum_{i=1}^{d} \lambda_i z_i^2 \leq \sum_{i=1}^{d} \lambda_{\max}(\mathbf{A}) z_i^2 = \lambda_{\max}(\mathbf{A}) \sum_{i=1}^{d} z_i^2$$

so the constraint $\|\mathbf{z}\|_2^2 = \sum_{i=1}^{d} z_i^2 = 1$ implies

$$\sum_{i=1}^{d} \lambda_i z_i^2 \leq \lambda_{\max}(\mathbf{A})$$

Defining $I = \{i : \lambda_i = \lambda_{\max}(\mathbf{A})\}$, the index set of the largest eigenvalue, we see that the bound is achieved with equality if and only if $\sum_{i \in I} z_i^2 = 1$ and $z_j = 0$ for $j \notin I$. Suppose $\mathbf{z}^*$ satisfies this condition. Then writing $\mathbf{q}_1, \dots, \mathbf{q}_d$ for the columns of $\mathbf{Q}$, we have

$$\mathbf{v}^* = \mathbf{Q}\mathbf{z}^* = \sum_{i=1}^{d} z_i^* \mathbf{q}_i = \sum_{i \in I} z_i^* \mathbf{q}_i$$

Recall that $\mathbf{q}_1, \dots, \mathbf{q}_d$ are eigenvectors of $\mathbf{A}$ and form an orthonormal basis for $\mathbb{R}^d$. Therefore by construction, the set $\{\mathbf{q}_i : i \in I\}$ forms an orthonormal basis for the eigenspace of $\lambda_{\max}(\mathbf{A})$. Hence $\mathbf{v}^*$, which is a linear combination of these, lies in that eigenspace and thus is an eigenvector of $\mathbf{A}$ corresponding to $\lambda_{\max}(\mathbf{A})$.

Conversely, suppose $\mathbf{v} \in \mathbb{R}^d$ is unit-length but not an eigenvector corresponding to $\lambda_{\max}(\mathbf{A})$. The vectors $\mathbf{q}_1, \dots, \mathbf{q}_d$ are still a basis for $\mathbb{R}^d$, so we have a unique expansion

$$\mathbf{v} = z_1\mathbf{q}_1 + \cdots + z_d\mathbf{q}_d$$

Since $\mathbf{v}$ does not lie in the eigenspace of $\lambda_{\max}(\mathbf{A})$, one of the components z_j must be nonzero for an index $j \notin I$, so equality does not hold in the bound above. $\square$

With this result established, we see that the vector we seek (which maximizes $\mathbf{v}^\top\mathbf{X}^\top\mathbf{X}\mathbf{v}$) must be an eigenvector corresponding to $\lambda_{\max}(\mathbf{X}^\top\mathbf{X})$. This is the same solution we derived via the Lagrangian formulation above.

Minimizing reconstruction error

Recall that ordinary least squares minimizes the vertical distance between the fitted line and the data points. We show that PCA can be interpreted as minimizing the perpendicular distance between the data points and the subspace onto which we are projecting them.

The orthogonal projection of a vector $\mathbf{x}$ onto the subspace spanned by a unit vector $\mathbf{v}$ equals $\mathbf{v}$ scaled by the scalar projection of $\mathbf{x}$ onto $\mathbf{v}$:

$$P_{\mathbf{v}}\mathbf{x} = (\mathbf{x}^\top\mathbf{v})\mathbf{v}$$

Suppose we want to minimize the total reconstruction error:

$$\sum_{i=1}^{n} \|\mathbf{x}_i - P_{\mathbf{v}}\mathbf{x}_i\|^2$$

For any $\mathbf{x} \in \mathbb{R}^d$, we know $\mathbf{x} - P_{\mathbf{v}}\mathbf{x} \perp P_{\mathbf{v}}\mathbf{x}$, so the Pythagorean Theorem tells us that

$$\|\mathbf{x} - P_{\mathbf{v}}\mathbf{x}\|^2 + \|P_{\mathbf{v}}\mathbf{x}\|^2 = \|\mathbf{x}\|^2$$

Thus

$$\begin{aligned}\sum_{i=1}^{n}\|\mathbf{x}_i - P_{\mathbf{v}}\mathbf{x}_i\|^2 &= \sum_{i=1}^{n}\left(\|\mathbf{x}_i\|^2 - \|P_{\mathbf{v}}\mathbf{x}_i\|^2\right) \\ &= \sum_{i=1}^{n}\|\mathbf{x}_i\|^2 - \sum_{i=1}^{n}\|(\mathbf{x}_i{}^\top\mathbf{v})\mathbf{v}\|^2 \\ &= \sum_{i=1}^{n}\|\mathbf{x}_i\|^2 - \sum_{i=1}^{n}(\mathbf{x}_i{}^\top\mathbf{v})^2\end{aligned}$$

Then since the first term $\sum_{i=1}^{n}\|\mathbf{x}_i\|^2$ is constant with respect to $\mathbf{v}$, minimizing reconstruction error is equivalent to maximizing $\sum_{i=1}^{n}(\mathbf{x}_i{}^\top\mathbf{v})^2$, which is (up to an irrelevant positive constant factor $1/n$) the projected variance.

Another way to write this interpretation is that the reconstructed matrix $\tilde{\mathbf{X}}_k$ is the best rank-k approximation to $\mathbf{X}$ in the Frobenius norm. To see this, first note that (writing $\mathbf{X} = \sum_{i=1}^{d}\sigma_i\mathbf{u}_i\mathbf{v}_i{}^\top$)

$$\tilde{\mathbf{X}}_k = \mathbf{X}\mathbf{V}_k\mathbf{V}_k{}^\top = \sum_{i=1}^{d}\sigma_i\mathbf{u}_i\mathbf{v}_i{}^\top\mathbf{V}_k\mathbf{V}_k{}^\top$$

By orthonormality, the product $\mathbf{v}_i{}^\top\mathbf{V}_k$ results in a k-dimensional row vector with 1 in the ith place and 0 everywhere else, i.e. $\mathbf{e}_i{}^\top$, as long as $i \leq k$. In this case,

$$\mathbf{v}_i{}^\top\mathbf{V}_k\mathbf{V}_k{}^\top = \mathbf{e}_i{}^\top\mathbf{V}_k{}^\top = (\mathbf{V}_k\mathbf{e}_i)^\top = \mathbf{v}_i{}^\top$$

If $i > k$, $\mathbf{v}_i{}^\top\mathbf{V}_k = \mathbf{0}^\top$, so the term disappears. Therefore we see that

$$\tilde{\mathbf{X}}_k = \sum_{i=1}^{d}\sigma_i\mathbf{u}_i\mathbf{v}_i{}^\top\mathbf{V}_k\mathbf{V}_k{}^\top = \sum_{i=1}^{k}\sigma_i\mathbf{u}_i\mathbf{v}_i{}^\top$$

which is the best rank-k approximation to $\mathbf{X}$ by the Eckart-Young theorem.

Probabilistic PCA

We have seen probabilistic motivations or derivations of many of the methods discussed so far in this class. In a similar vein, **probabilistic PCA** (PPCA) is a **generative** model for PCA. Here we make the following assumptions about how the data were generated: for each datapoint i, there is a k-dimensional **latent variable**

$$\mathbf{z}_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

which we cannot observe, and the actual d-dimensional observation is distributed conditionally on this latent variable as

$$\mathbf{x}_i|\mathbf{z}_i \sim \mathcal{N}(\boldsymbol{\Lambda}\mathbf{z}_i + \boldsymbol{\mu}, \boldsymbol{\Psi})$$

Here $\boldsymbol{\Lambda} \in \mathbb{R}^{d\times k}$ and $\boldsymbol{\mu} \in \mathbb{R}^d$ are parameters to be estimated. Since $\mathbf{z}_i$ is Gaussian and $\mathbf{x}_i|\mathbf{z}_i$ is Gaussian, $\begin{bmatrix}\mathbf{x}_i \\ \mathbf{z}_i\end{bmatrix}$ is Gaussian, so its marginal $\mathbf{x}_i$ is Gaussian. In particular, by integrating out the latent variable

$$p(\mathbf{x}_i) = \int_{\mathbf{z}} p(\mathbf{x}_i, \mathbf{z})\,\mathrm{d}\mathbf{z} = \int_{\mathbf{z}} p(\mathbf{x}_i|\mathbf{z})p(\mathbf{z})\,\mathrm{d}\mathbf{z}$$

one can show that

$$\mathbf{x}_i \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Lambda}\boldsymbol{\Lambda}^\top + \boldsymbol{\Psi})$$

It is common to assume $\mathbf{\Psi} = \sigma^2\mathbf{I}$. In this case, if we let $\sigma^2 \to 0$, we recover the original PCA solution in the sense that the columspace of $\hat{\mathbf{\Lambda}}_{\text{MLE}}$ approaches the PCA subspace (i.e. the columnspace of $\mathbf{V}_k$). [2]

3.2 Canonical Correlation Analysis

PCA provided us with a dimensionality-reduction approach that didn't use the labels y in any way. In that way, it was fundamentally unsupervised by nature. However, we can imagine that there can be situations in which the most relevant directions in $\mathbf{x}$ for understanding $\mathbf{y}$ are not necessarily the directions of greatest variation in $\mathbf{x}$. For example, what if the $\mathbf{x}$ data by nature was contaminated with a strong correlated noise signal? PCA would find the noise dimensions to be those that have the greatest variation and keep them, throwing away those dimensions where we could actually hope to get information relevant for predicting $\mathbf{y}$!

The other potentially troublesome aspect of PCA is that it is not invariant to a change of units or scaling. If we changed the units of some feature from meters to millimeters, then all the values for that feature would increase by a factor of a thousand, and suddenly, this direction might be favored by PCA. This is unavoidable because there is no natural reference point that would allow us to treat units as arbitrary.

Consequently, it is important to have an approach to dimensionality reduction and the discovery of linear structure from data that does take advantage of paired $(\mathbf{x}, \mathbf{y})$ data, preferably in a way that is robust to linear transformations of both $\mathbf{x}$ and $\mathbf{y}$ individually.

A latent space view with Gaussian random variables

What does it mean to extract the linear structure establishing the underlying relationship between $\mathbf{X}$ and $\mathbf{Y}$, two vector-valued quantites of which we have many paired samples. To understand what this should mean, we need to construct a model. The first thing that we do is assume we have a joint distribution for X and Y as random variables. In practice, we won't have the random variables in distribution, just paired samples of them. But it is easier to start understanding what we want by assuming that we have the entire distribution. This corresponds to how well we think we can do given infinite amounts of training data. The next we do is assume a particular form for the random variables. Since we are interested in linear structure, jointly Gaussian random variables are a useful model.

Our goal is to extract the underlying relationship or commonality between $\mathbf{X}$ and $\mathbf{Y}$. To do this, we assume that we have three underlying iid standard Gaussian random vectors $\mathbf{Z}_J$ (representing the common/joint part), $\mathbf{Z}_X$ (representing the randomness that is purely in $\mathbf{X}$ and not shared by $\mathbf{Y}$), and $\mathbf{Z}_Y$ (representing the randomness that is purely in $\mathbf{Y}$ and not shared by $\mathbf{X}$). Then we can assume that they are related by an underlying linear relationship:

$$\begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{0} \\ \mathbf{0} & \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{Z}_X \\ \mathbf{Z}_J \\ \mathbf{Z}_Y \end{bmatrix} \tag{3.1}$$

As is typical in these situations, there is going to be some ambiguity in choosing the $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ matrices. But the important thing is that somehow the $\mathbf{B}$ and $\mathbf{C}$ matrices together capture the joint relationship between $\mathbf{X}$ and $\mathbf{Y}$.

[2] See Tipping and Bishop's original paper for derivations and more information.

How will such a joint relationship manifest in the joint distributions for **X** and **Y**? To understand that, we should first consider the scalar case.

Correlation and Scalar Gaussians

For the scalar case, A, B, C, D are just real numbers. So, the joint distribution of X, Y is $\mathcal{N}(\mathbf{0}, \Sigma)$ where $\Sigma = \begin{bmatrix} A^2+B^2 & BC \\ BC & C^2+D^2 \end{bmatrix}$. The first thing that we notice is that we cannot disentangle B and C. The second is that the information about the joint relationship (which we know is encoded by B and C) is all in the cross-covariance term, not in the individual variance term. Recall that we want to pull out the relationship in a way that does not depend on any individual scaling or linear transformation that we apply to X and Y.

Here's a neat fact: if X and Y are jointly Gaussian, i.e.

$$\begin{bmatrix} X \\ Y \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \Sigma)$$

then we can define a distribution on *individually normalized* X and Y and have their joint inter-relationship entirely captured by $\rho(X, Y)$. First write

$$\rho(X, Y) = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Then

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma y^2 \end{bmatrix} = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_y^2 \end{bmatrix}$$

so

$$\begin{aligned} \begin{bmatrix} \sigma_x^{-1} & 0 \\ 0 & \sigma_y^{-1} \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} &\sim \mathcal{N}\left(0, \begin{bmatrix} \sigma_x^{-1} & 0 \\ 0 & \sigma_y^{-1} \end{bmatrix} \Sigma \begin{bmatrix} \sigma_x^{-1} & 0 \\ 0 & \sigma_y^{-1} \end{bmatrix}^\top \right) \\ &\sim \mathcal{N}\left(0, \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}\right) \end{aligned}$$

This ρ quantity is the signature of the joint inter-relationship of the X and Y random variables.

To make things explicit, once we have the $\rho = \frac{BC}{\sqrt{(A^2+B^2)(C^2+D^2)}}$, we can come up with many possible backstories for the latent picture behind the observed random variables. Here is one that splits the influence of the latent space proportionately.

$$A = \sigma_x \sqrt{1 - |\rho|} \tag{3.2}$$

$$B = \sigma_x \sqrt{|\rho|} \tag{3.3}$$

$$C = \sigma_y \text{sign}(\rho) \sqrt{|\rho|} \tag{3.4}$$

$$D = \sigma_y \sqrt{1 - |\rho|} \tag{3.5}$$

Because $A^2 + B^2 = \sigma_x^2$, $C^2 + D^2 = \sigma_y^2$, and $\rho = \frac{BC}{\sqrt{(A^2+B^2)(C^2+D^2)}}$, this works.

Pearson Correlation

Although we defined this ρ above for a pair of jointly Gaussian random variables, it is really about linear structure. The **Pearson Correlation Coefficient** $\rho(X, Y)$ is effectively a way to measure how linearly related (in other words, how well a linear model captures the relationship between) random variables X and Y.

$$\rho(X,Y) = \frac{\operatorname{Cov}(X,Y)}{\sqrt{\operatorname{Var}(X)\operatorname{Var}(Y)}}$$

Here are some important facts about it:

- It is commutative: $\rho(X,Y) = \rho(Y,X)$
- It always lies between -1 and 1: $-1 \le \rho(X,Y) \le 1$
- It is completely invariant to affine transformations: for any $a, b, c, d \in \mathbb{R}$,

$$\begin{aligned}\rho(aX+b, cY+d) &= \frac{\operatorname{Cov}(aX+b, cY+d)}{\sqrt{\operatorname{Var}(aX+b)\operatorname{Var}(cY+d)}} \\ &= \frac{\operatorname{Cov}(aX, cY)}{\sqrt{\operatorname{Var}(aX)\operatorname{Var}(cY)}} \\ &= \frac{a \cdot c \cdot \operatorname{Cov}(X,Y)}{\sqrt{a^2\operatorname{Var}(X)\cdot c^2\operatorname{Var}(Y)}} \\ &= \frac{\operatorname{Cov}(X,Y)}{\sqrt{\operatorname{Var}(X)\operatorname{Var}(Y)}} \\ &= \rho(X,Y)\end{aligned}$$

The correlation is defined in terms of random variables rather than observed data. Assume now that $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are vectors containing n independent observations of X and Y, respectively. Recall the **law of large numbers**, which states that for i.i.d. X_i with mean μ,

$$\frac{1}{n}\sum_{i=1}^{n} X_i \xrightarrow{\text{a.s.}} \mu \quad \text{as } n \to \infty$$

We can use this law to justify a sample-based approximation to the mean:

$$\operatorname{Cov}(X,Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \approx \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

where the bar indicates the sample average, i.e. $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. Then as a special case we have

$$\operatorname{Var}(X) = \operatorname{Cov}(X,X) = \mathbb{E}[(X - \mathbb{E}[X])^2] \approx \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

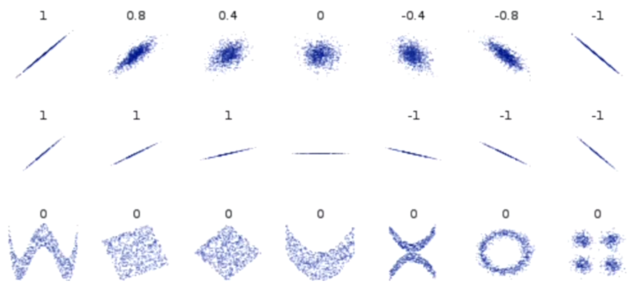
$$\operatorname{Var}(Y) = \operatorname{Cov}(Y,Y) = \mathbb{E}[(Y - \mathbb{E}[Y])^2] \approx \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$$

Plugging these estimates into the definition for correlation and canceling the factor of $1/n$ leads us to the **Sample Pearson Correlation Coefficient** $\hat{\rho}$:

$$\hat{\rho}(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \cdot \sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

$$= \frac{\tilde{x}^\top \tilde{y}}{\sqrt{\tilde{x}^\top \tilde{x} \cdot \tilde{y}^\top \tilde{y}}} \quad \text{where } \tilde{x} = x - \bar{x}, \tilde{y} = y - \bar{y}$$

Here are some 2-D scatterplots and their corresponding correlation coefficients:



You should notice that:

- The magnitude of $\hat{\rho}$ increases as X and Y become more linearly correlated.
- The sign of $\hat{\rho}$ tells whether X and Y have a positive or negative relationship.
- The correlation coefficient is undefined if either X or Y has 0 variance (horizontal line).

Canonical Correlation Analysis

Canonical Correlation Analysis (CCA) is a method of modeling the relationship between two point sets by making use of the correlation coefficients.

As in PCA, it is useful to start with trying to find the directions that represent the most correlation. You can think of this as finding the parts of $\mathbf{X}_{\text{rv}}$ and $\mathbf{Y}_{\text{rv}}$ that depend on the first coordinate of $\mathbf{Z}_J$, where we choose the convention that the first coordinate represents the most shared dimension. We will then see how to move on to get the rest.

Formally, given zero-mean random vectors $\mathbf{X}_{\text{rv}} \in \mathbb{R}^p$ and $\mathbf{Y}_{\text{rv}} \in \mathbb{R}^q$, we want to find projection vectors $\mathbf{u} \in \mathbb{R}^p$ and $\mathbf{v} \in \mathbb{R}^q$ that maximizes the correlation between ${\mathbf{X}_{\text{rv}}}^\top\mathbf{u}$ and ${\mathbf{Y}_{\text{rv}}}^\top\mathbf{v}$:

$$\max_{\mathbf{u},\mathbf{v}} \rho({\mathbf{X}_{\text{rv}}}^\top\mathbf{u}, {\mathbf{Y}_{\text{rv}}}^\top\mathbf{v}) = \max_{\mathbf{u},\mathbf{v}} \frac{\operatorname{Cov}({\mathbf{X}_{\text{rv}}}^\top\mathbf{u}, {\mathbf{Y}_{\text{rv}}}^\top\mathbf{v})}{\sqrt{\operatorname{Var}({\mathbf{X}_{\text{rv}}}^\top\mathbf{u}) \operatorname{Var}({\mathbf{Y}_{\text{rv}}}^\top\mathbf{v})}}$$

Observe that

$$\begin{aligned}
\operatorname{Cov}({\mathbf{X}_{\text{rv}}}^\top\mathbf{u}, {\mathbf{Y}_{\text{rv}}}^\top\mathbf{v}) &= \mathbb{E}[({\mathbf{X}_{\text{rv}}}^\top\mathbf{u} - \mathbb{E}[{\mathbf{X}_{\text{rv}}}^\top\mathbf{u}])({\mathbf{Y}_{\text{rv}}}^\top\mathbf{v} - \mathbb{E}[{\mathbf{Y}_{\text{rv}}}^\top\mathbf{v}])] \\
&= \mathbb{E}[\mathbf{u}^\top(\mathbf{X}_{\text{rv}} - \mathbb{E}[\mathbf{X}_{\text{rv}}])(\mathbf{Y}_{\text{rv}} - \mathbb{E}[\mathbf{Y}_{\text{rv}}])^\top\mathbf{v}] \\
&= \mathbf{u}^\top\mathbb{E}[(\mathbf{X}_{\text{rv}} - \mathbb{E}[\mathbf{X}_{\text{rv}}])(\mathbf{Y}_{\text{rv}} - \mathbb{E}[\mathbf{Y}_{\text{rv}}])^\top]\mathbf{v} \\
&= \mathbf{u}^\top \operatorname{Cov}(\mathbf{X}_{\text{rv}}, \mathbf{Y}_{\text{rv}})\mathbf{v}
\end{aligned}$$

which also implies (since $\operatorname{Var}(Z) = \operatorname{Cov}(Z, Z)$ for any random variable Z) that

$$\begin{aligned}
\operatorname{Var}({\mathbf{X}_{\text{rv}}}^\top\mathbf{u}) &= \mathbf{u}^\top \operatorname{Cov}(\mathbf{X}_{\text{rv}}, \mathbf{X}_{\text{rv}})\mathbf{u} \\
\operatorname{Var}({\mathbf{Y}_{\text{rv}}}^\top\mathbf{v}) &= \mathbf{v}^\top \operatorname{Cov}(\mathbf{Y}_{\text{rv}}, \mathbf{Y}_{\text{rv}})\mathbf{v}
\end{aligned}$$

so the correlation can be written

$$\rho(\mathbf{X}_{\text{rv}}{}^\top \mathbf{u}, \mathbf{Y}_{\text{rv}}{}^\top \mathbf{v}) = \frac{\mathbf{u}^\top \operatorname{Cov}(\mathbf{X}_{\text{rv}}, \mathbf{Y}_{\text{rv}})\mathbf{v}}{\sqrt{\mathbf{u}^\top \operatorname{Cov}(\mathbf{X}_{\text{rv}}, \mathbf{X}_{\text{rv}})\mathbf{u} \cdot \mathbf{v}^\top \operatorname{Cov}(\mathbf{Y}_{\text{rv}}, \mathbf{Y}_{\text{rv}})\mathbf{v}}}$$

Unfortunately, we do not have access to the true distributions of $\mathbf{X}_{\text{rv}}$ and $\mathbf{Y}_{\text{rv}}$, so we cannot compute these covariances matrices. However, we can estimate them from data. Assume now that we are given zero-mean data matrices $\mathbf{X} \in \mathbb{R}^{n\times p}$ and $\mathbf{Y} \in \mathbb{R}^{n\times q}$, where the rows of the matrix $\mathbf{X}$ are i.i.d. samples $\mathbf{x}_i \in \mathbb{R}^p$ from the random variable $\mathbf{X}_{\text{rv}}$, and correspondingly for $\mathbf{Y}_{\text{rv}}$. Then

$$\operatorname{Cov}(\mathbf{X}_{\text{rv}}, \mathbf{Y}_{\text{rv}}) = \mathbb{E}[(\mathbf{X}_{\text{rv}} - \underbrace{\mathbb{E}[\mathbf{X}_{\text{rv}}]}_{0})(\mathbf{Y}_{\text{rv}} - \underbrace{\mathbb{E}[\mathbf{Y}_{\text{rv}}]}_{0})^\top] = \mathbb{E}[\mathbf{X}_{\text{rv}}\mathbf{Y}_{\text{rv}}{}^\top] \approx \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i{}^\top = \frac{1}{n}\mathbf{X}^\top\mathbf{Y}$$

where again the sample-based approximation is justified by the law of large numbers. Similarly,

$$\begin{aligned}
\operatorname{Cov}(\mathbf{X}_{\text{rv}}, \mathbf{X}_{\text{rv}}) &= \mathbb{E}[\mathbf{X}_{\text{rv}}\mathbf{X}_{\text{rv}}{}^\top] \approx \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i{}^\top = \frac{1}{n}\mathbf{X}^\top\mathbf{X} \\
\operatorname{Cov}(\mathbf{Y}_{\text{rv}}, \mathbf{Y}_{\text{rv}}) &= \mathbb{E}[\mathbf{Y}_{\text{rv}}\mathbf{Y}_{\text{rv}}{}^\top] \approx \frac{1}{n}\sum_{i=1}^{n} \mathbf{y}_i \mathbf{y}_i{}^\top = \frac{1}{n}\mathbf{Y}^\top\mathbf{Y}
\end{aligned}$$

Plugging these estimates in for the true covariance matrices, we arrive at the problem

$$\max_{\mathbf{u},\mathbf{v}} \frac{\mathbf{u}^\top \left(\frac{1}{n}\mathbf{X}^\top\mathbf{Y}\right)\mathbf{u}}{\sqrt{\mathbf{u}^\top \left(\frac{1}{n}\mathbf{X}^\top\mathbf{X}\right)\mathbf{u} \cdot \mathbf{v}^\top \left(\frac{1}{n}\mathbf{Y}^\top\mathbf{Y}\right)\mathbf{v}}} = \max_{\mathbf{u},\mathbf{v}} \underbrace{\frac{\mathbf{u}^\top\mathbf{X}^\top\mathbf{Y}\mathbf{v}}{\sqrt{\mathbf{u}^\top\mathbf{X}^\top\mathbf{X}\mathbf{u} \cdot \mathbf{v}^\top\mathbf{Y}^\top\mathbf{Y}\mathbf{v}}}}_{\hat{\rho}(\mathbf{X}\mathbf{u}, \mathbf{Y}\mathbf{v})}$$

Let's try to massage the maximization problem into a form that we can reason with more easily. Our strategy is to choose matrices to transform $\mathbf{X}$ and $\mathbf{Y}$ such that the maximization problem is equivalent but easier to understand.

1. First, let's choose matrices $\mathbf{W}_x, \mathbf{W}_y$ to **whiten** $\mathbf{X}$ and $\mathbf{Y}$. This will make the (co)variance matrices $(\mathbf{X}\mathbf{W}_x)^\top(\mathbf{X}\mathbf{W}_x)$ and $(\mathbf{Y}\mathbf{W}_y)^\top(\mathbf{Y}\mathbf{W}_y)$ become identity matrices and simplify our expression. To do this, note that $\mathbf{X}^\top\mathbf{X}$ is positive definite (and hence symmetric), so we can employ the eigendecomposition

   $$\mathbf{X}^\top\mathbf{X} = \mathbf{U}_x\mathbf{S}_x\mathbf{U}_x{}^\top$$

   Since

   $$\mathbf{S}_x = \operatorname{diag}(\lambda_1(\mathbf{X}^\top\mathbf{X}), \dots, \lambda_d(\mathbf{X}^\top\mathbf{X}))$$

   where all the eigenvalues are positive, we can define the "square root" of this matrix by taking the square root of every diagonal entry:

   $$\mathbf{S}_x^{1/2} = \operatorname{diag}\left(\sqrt{\lambda_1(\mathbf{X}^\top\mathbf{X})}, \dots, \sqrt{\lambda_d(\mathbf{X}^\top\mathbf{X})}\right)$$

   Then, defining $\mathbf{W}_x = \mathbf{U}_x\mathbf{S}_x^{-1/2}\mathbf{U}_x{}^\top$, we have

   $$\begin{aligned}
   (\mathbf{X}\mathbf{W}_x)^\top(\mathbf{X}\mathbf{W}_x) &= \mathbf{W}_x{}^\top\mathbf{X}^\top\mathbf{X}\mathbf{W}_x \\
   &= \mathbf{U}_x\mathbf{S}_x^{-1/2}\mathbf{U}_x{}^\top\mathbf{U}_x\mathbf{S}_x\mathbf{U}_x{}^\top\mathbf{U}_x\mathbf{S}_x^{-1/2}\mathbf{U}_x{}^\top \\
   &= \mathbf{U}_x\mathbf{S}_x^{-1/2}\mathbf{S}_x\mathbf{S}_x^{-1/2}\mathbf{U}_x{}^\top
   \end{aligned}$$

$$
\begin{aligned}
&= \mathbf{U}_x \mathbf{U}_x{}^\top \\
&= \mathbf{I}
\end{aligned}
$$

which shows that $\mathbf{W}_x$ is a whitening matrix for $\mathbf{X}$. The same process can be repeated to produce a whitening matrix $\mathbf{W}_y = \mathbf{U}_y \mathbf{S}_y^{-1/2} \mathbf{U}_y{}^\top$ for $\mathbf{Y}$.

Let's denote the whitened data $\mathbf{X}_w = \mathbf{X}\mathbf{W}_x$ and $\mathbf{Y}_w = \mathbf{Y}\mathbf{W}_y$. Then by the change of variables $\mathbf{u}_w = \mathbf{W}_x^{-1}\mathbf{u}, \mathbf{v}_w = \mathbf{W}_y^{-1}\mathbf{v}$,

$$
\begin{aligned}
\max_{\mathbf{u},\mathbf{v}} \hat{\rho}(\mathbf{X}\mathbf{u}, \mathbf{Y}\mathbf{v}) &= \max_{\mathbf{u},\mathbf{v}} \frac{(\mathbf{X}\mathbf{u})^\top \mathbf{Y}\mathbf{v}}{\sqrt{(\mathbf{X}\mathbf{u})^\top \mathbf{X}\mathbf{u} (\mathbf{Y}\mathbf{v})^\top \mathbf{Y}\mathbf{v}}} \\
&= \max_{\mathbf{u},\mathbf{v}} \frac{(\mathbf{X}\mathbf{W}_x\mathbf{W}_x^{-1}\mathbf{u})^\top Y \mathbf{W}_y \mathbf{W}_y^{-1}\mathbf{v}}{\sqrt{(\mathbf{X}\mathbf{W}_x\mathbf{W}_x^{-1}\mathbf{u})^\top \mathbf{X}\mathbf{W}_x\mathbf{W}_x^{-1}\mathbf{u} (Y\mathbf{W}_y\mathbf{W}_y^{-1}\mathbf{v})^\top \mathbf{Y}\mathbf{W}_y\mathbf{W}_y^{-1}\mathbf{v}}} \\
&= \max_{\mathbf{u}_w,\mathbf{v}_w} \frac{(\mathbf{X}_w\mathbf{u}_w)^\top \mathbf{Y}_w\mathbf{v}_w}{\sqrt{(\mathbf{X}_w\mathbf{u}_w)^\top \mathbf{X}_w\mathbf{u}_w (\mathbf{Y}_w\mathbf{v}_w)^\top \mathbf{Y}_w\mathbf{v}_w}} \\
&= \max_{\mathbf{u}_w,\mathbf{v}_w} \frac{\mathbf{u}_w{}^\top \mathbf{X}_w{}^\top \mathbf{Y}_w\mathbf{v}_w}{\sqrt{\mathbf{u}_w{}^\top \mathbf{X}_w{}^\top \mathbf{X}_w\mathbf{u}_w \cdot \mathbf{v}_w{}^\top \mathbf{Y}_w{}^\top \mathbf{Y}_w\mathbf{v}_w}} \\
&= \max_{\mathbf{u}_w,\mathbf{v}_w} \underbrace{\frac{\mathbf{u}_w{}^\top \mathbf{X}_w{}^\top \mathbf{Y}_w\mathbf{v}_w}{\sqrt{\mathbf{u}_w{}^\top \mathbf{u}_w \cdot \mathbf{v}_w{}^\top \mathbf{v}_w}}}_{\hat{\rho}(\mathbf{X}_w\mathbf{u}_w, \mathbf{Y}_w\mathbf{v}_w)}
\end{aligned}
$$

Note we have used the fact that $\mathbf{X}_w{}^\top \mathbf{X}_w$ and $\mathbf{Y}_w{}^\top \mathbf{Y}_w$ are identity matrices by construction.

2. Second, let's choose matrices $\mathbf{D}_x$, $\mathbf{D}_y$ to **decorrelate** $\mathbf{X}_w$ and $\mathbf{Y}_w$. This will let us simplify the covariance matrix $(\mathbf{X}_w\mathbf{D}_x)^\top(\mathbf{Y}_w\mathbf{D}_y)$ into a **diagonal** matrix.

   Recall that our ultimate goal is to understand the underlying latent structure behind $\mathbf{X}_{\text{rv}}$ and $\mathbf{Y}_{\text{rv}}$. The whitening was a normalizing change of coordinates. The decorrelation is there so that we can pick out independent underlying components of $\mathbf{Z}_J$. (Since jointly Gaussian random variables are independent if they are uncorrelated.) Alternatively, you can consider decorrelation as reducing the problem to a sequence of scalar problems.

   To do this, we'll make use of the SVD:

$$
\mathbf{X}_w{}^\top \mathbf{Y}_w = \mathbf{U}\mathbf{S}\mathbf{V}^\top
$$

   The choice of $\mathbf{U}$ for $\mathbf{D}_x$ and $\mathbf{V}$ for $\mathbf{D}_y$ accomplishes our goal, since

$$
(\mathbf{X}_w\mathbf{U})^\top(\mathbf{Y}_w\mathbf{V}) = \mathbf{U}^\top \mathbf{X}_w{}^\top \mathbf{Y}_w\mathbf{V} = \mathbf{U}^\top(\mathbf{U}\mathbf{S}\mathbf{V}^\top)\mathbf{V} = \mathbf{S}
$$

Let's denote the decorrelated data $\mathbf{X}_d = \mathbf{X}_w\mathbf{D}_x$ and $\mathbf{Y}_d = \mathbf{Y}_w\mathbf{D}_y$. Then by the change of variables $\mathbf{u}_d = \mathbf{D}_x^{-1}\mathbf{u}_w = \mathbf{D}_x{}^\top \mathbf{u}_w, \mathbf{v}_d = \mathbf{D}_y^{-1}\mathbf{v}_w = \mathbf{D}_y{}^\top \mathbf{v}_w$,

$$
\begin{aligned}
\max_{\mathbf{u}_w,\mathbf{v}_w} \hat{\rho}(\mathbf{X}_w\mathbf{u}_w, \mathbf{Y}_w\mathbf{v}_w) &= \max_{\mathbf{u}_w,\mathbf{v}_w} \frac{(\mathbf{X}_w\mathbf{u}_w)^\top \mathbf{Y}_w\mathbf{v}_w}{\sqrt{\mathbf{u}_w{}^\top \mathbf{u}_w \cdot \mathbf{v}_w{}^\top \mathbf{v}_w}} \\
&= \max_{\mathbf{u}_w,\mathbf{v}_w} \frac{(\mathbf{X}_w\mathbf{D}_x\mathbf{D}_x^{-1}\mathbf{u}_w)^\top \mathbf{Y}_w\mathbf{D}_y\mathbf{D}_y^{-1}\mathbf{v}_w}{\sqrt{(\mathbf{D}_x\mathbf{u}_w)^\top \mathbf{D}_x\mathbf{u}_w \cdot (\mathbf{D}_y\mathbf{v}_w)^\top \mathbf{D}_y\mathbf{v}_w}} \\
&= \max_{\mathbf{u}_d,\mathbf{v}_d} \frac{(\mathbf{X}_d\mathbf{u}_d)^\top \mathbf{Y}_d\mathbf{v}_d}{\sqrt{\mathbf{u}_d{}^\top \mathbf{u}_d \cdot \mathbf{v}_d{}^\top \mathbf{v}_d}}
\end{aligned}
$$

$$= \max_{\mathbf{u}_d,\mathbf{v}_d} \underbrace{\frac{\mathbf{u}_d^\top \mathbf{X}_d \mathbf{Y}_d \mathbf{v}_d}{\sqrt{\mathbf{u}_d^\top \mathbf{u}_d \cdot \mathbf{v}_d^\top \mathbf{v}_d}}}_{\hat{\rho}(\mathbf{X}_d\mathbf{u}_d, \mathbf{Y}_d\mathbf{v}_d)}$$
$$= \max_{\mathbf{u}_d,\mathbf{v}_d} \frac{\mathbf{u}_d^\top \mathbf{S} \mathbf{v}_d}{\sqrt{\mathbf{u}_d^\top \mathbf{u}_d \cdot \mathbf{v}_d^\top \mathbf{v}_d}}$$

Without loss of generality, suppose $\mathbf{u}_d$ and $\mathbf{v}_d$ are unit vectors[3] so that the denominator becomes 1, and we can ignore it:

$$\max_{\mathbf{u}_d,\mathbf{v}_d} \frac{\mathbf{u}_d^\top S \mathbf{v}_d}{\sqrt{\mathbf{u}_d^\top \mathbf{u}_d \cdot \mathbf{v}_d^\top \mathbf{v}_d}} = \max_{\substack{\|\mathbf{u}_d\|=1 \\ \|\mathbf{v}_d\|=1}} \frac{\mathbf{u}_d^\top \mathbf{S} \mathbf{v}_d}{\|\mathbf{u}_d\|\|\mathbf{v}_d\|} = \max_{\substack{\|\mathbf{u}_d\|=1 \\ \|\mathbf{v}_d\|=1}} \mathbf{u}_d^\top \mathbf{S} \mathbf{v}_d$$

The diagonal nature of $\mathbf{S}$ implies $S_{ij} = 0$ for $i \neq j$, so our simplified objective expands as

$$\mathbf{u}_d^\top \mathbf{S} \mathbf{v}_d = \sum_i \sum_j (\mathbf{u}_d)_i S_{ij} (\mathbf{v}_d)_j = \sum_i S_{ii} (\mathbf{u}_d)_i (\mathbf{v}_d)_i$$

where S_{ii}, the singular values of $\mathbf{X}_w{}^\top \mathbf{Y}_w$, are arranged in descending order. Thus we have a weighted sum of these singular values, where the weights are given by the entries of $\mathbf{u}_d$ and $\mathbf{v}_d$, which are constrained to have unit norm. To maximize the sum, we "put all our eggs in one basket" and extract S_{11} by setting the first components of $\mathbf{u}_d$ and $\mathbf{v}_d$ to 1, and the rest to 0:

$$\mathbf{u}_d = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^p \qquad\qquad \mathbf{v}_d = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^q$$

Any other arrangement would put weight on S_{ii} at the expense of taking that weight away from S_{11}, which is the largest, thus reducing the value of the sum.

Finally we have an analytical solution, but it is in a different coordinate system than our original problem! In particular, $\mathbf{u}_d$ and $\mathbf{v}_d$ are the best weights in a coordinate system where the data has been whitened and decorrelated. To bring it back to our original coordinate system and find the vectors we actually care about ($\mathbf{u}$ and $\mathbf{v}$), we must invert the changes of variables we made:

$$\mathbf{u} = \mathbf{W}_x \mathbf{u}_w = \mathbf{W}_x \mathbf{D}_x \mathbf{u}_d \qquad\qquad \mathbf{v} = \mathbf{W}_y \mathbf{v}_w = \mathbf{W}_y \mathbf{D}_y \mathbf{v}_d$$

More generally, to get the best k directions, we choose

$$\mathbf{U}_d = \begin{bmatrix} \mathbf{I}_k \\ \mathbf{0}_{p-k,k} \end{bmatrix} \in \mathbb{R}^{p \times k} \qquad\qquad \mathbf{V}_d = \begin{bmatrix} \mathbf{I}_k \\ \mathbf{0}_{q-k,k} \end{bmatrix} \in \mathbb{R}^{q \times k}$$

where I_k denotes the k-dimensional identity matrix. Then

$$\mathbf{U} = \mathbf{W}_x \mathbf{D}_x \mathbf{U}_d \qquad\qquad \mathbf{V} = \mathbf{W}_y \mathbf{D}_y \mathbf{V}_d$$

Note that $\mathbf{U}_d$ and $\mathbf{V}_d$ have orthogonal columns. The columns of $\mathbf{U}$ and $\mathbf{V}$, which are the projection directions we seek, will in general not be orthogonal, but they will be linearly independent (since they come from the application of invertible matrices to the columns of $\mathbf{U}_d, \mathbf{V}_d$).

[3] Why can we assume this? Observe that the value of the objective does not change if we replace $\mathbf{u}_d$ by $\alpha\mathbf{u}_d$ and $\mathbf{v}_d$ by $\beta\mathbf{v}_d$, where α and β are any positive constants. Thus if there are maximizers $\mathbf{u}_d, \mathbf{v}_d$ which are not unit vectors, then $\mathbf{u}_d/\|\mathbf{u}_d\|$ and $\mathbf{v}_d/\|\mathbf{v}_d\|$ (which are unit vectors) are also maximizers.

Following (3.2), (3.3), (3.4), and (3.5), it is also possible to use what we have calculated to give an explicit learned latent-space realization for the $\mathbf{X}_{\text{rv}}, \mathbf{Y}_{\text{rv}}$ in terms of standard Gaussian random variables $\mathbf{Z}_X$, $\mathbf{Z}_J$, $\mathbf{Z}_Y$. In particular, matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ of the appropriate sizes. This is left as an exercise to the reader once you realize that after whitening and decorrelating (both invertible transformations), we are left with a collection of scalar problems that would represent independent random variables if all the variables were indeed jointly Gaussian.

CCA thus illustrates how it is possible to learn a latent representation for common (linear) structure given paired data. This is a powerful idea not limited to the specific case of CCA. In effect, CCA shows how we can discover (synthesize) features that distill what aspects of input data is relevant for understanding output data.

This is subtly different from what happens in ordinary least squares because in ordinary least squares, each individual element of $\mathbf{y}$ is predicted independently. In OLS, the different output variables are not used collectively to distill the most relevant dimensions of the input. By contrast, in CCA, the different output variables do vote collectively to determine relevant dimensions in the input.

Comparison with PCA

An advantage of CCA over PCA is that it is invariant to scalings and affine transformations of $\mathbf{X}$ and $\mathbf{Y}$. Consider a simplified scenario in which two matrix-valued random variables $\mathbf{X}, \mathbf{Y}$ satisfy $\mathbf{Y} = \mathbf{X} + \epsilon$ where the noise ϵ has huge variance. What happens when we run PCA on $\mathbf{Y}$? Since PCA maximizes variance, it will actually project $\mathbf{Y}$ (largely) into the column space of ϵ! However, we're interested in $\mathbf{Y}$'s relationship to $\mathbf{X}$, not its dependence on noise. How can we fix this? As it turns out, CCA solves this issue. Instead of maximizing variance of $\mathbf{Y}$, we maximize correlation between $\mathbf{X}$ and $\mathbf{Y}$. In some sense, we want the maximize "predictive power" of information we have.

CCA regression

Once we've computed the CCA coefficients, one application is to use them for regression tasks, predicting $\mathbf{Y}$ from $\mathbf{X}$ (or vice-versa). Recall that the correlation coefficient attains a greater value when the two sets of data are *more linearly correlated.* Thus, it makes sense to find the $k \times k$ weight matrix $\mathbf{A}$ that linearly relates $\mathbf{XU}$ and $\mathbf{YV}$. We can accomplish this with ordinary least squares.

Denote the projected data matrices by $\mathbf{X}_c = \mathbf{XU}$ and $\mathbf{Y}_c = \mathbf{YV}$. Observe that $\mathbf{X}_c$ and $\mathbf{Y}_c$ are zero-mean because they are linear transformations of $\mathbf{X}$ and $\mathbf{Y}$, which are zero-mean. Thus we can fit a linear model relating the two:

$$\mathbf{Y}_c \approx \mathbf{X}_c\mathbf{A}$$

The least-squares solution is given by

$$\begin{aligned}\mathbf{A} &= (\mathbf{X}_c{}^\top\mathbf{X}_c)^{-1}\mathbf{X}_c{}^\top\mathbf{Y}_c \\ &= (\mathbf{U}^\top\mathbf{X}^\top\mathbf{XU})^{-1}\mathbf{U}^\top\mathbf{X}^\top\mathbf{YV}\end{aligned}$$

However, since what we *really* want is an estimate of $\mathbf{Y}$ given new (zero-mean) observations $\tilde{\mathbf{X}}$ (or vice-versa), it's useful to have the entire series of transformations that relates the two. The predicted canonical variables are given by

$$\hat{\mathbf{Y}}_c = \tilde{\mathbf{X}}_c\mathbf{A} = \tilde{\mathbf{X}}\mathbf{U}(\mathbf{U}^\top\mathbf{X}^\top\mathbf{XU})^{-1}\mathbf{U}^\top\mathbf{X}^\top\mathbf{YV}$$

Then we use the canonical variables to compute the actual values:

$$\begin{aligned}\hat{\mathbf{Y}} &= \hat{\mathbf{Y}}_c(\mathbf{V}^\top\mathbf{V})^{-1}\mathbf{V}^\top \\ &= \tilde{\mathbf{X}}\mathbf{U}(\mathbf{U}^\top\mathbf{X}^\top\mathbf{X}\mathbf{U})^{-1}(\mathbf{U}^\top\mathbf{X}^\top\mathbf{Y}\mathbf{V})(\mathbf{V}^\top\mathbf{V})^{-1}\mathbf{V}^\top\end{aligned}$$

We can collapse all these terms into a single matrix $\mathbf{A}_{\text{eq}}$ that gives the prediction $\hat{\mathbf{Y}}$ from $\tilde{\mathbf{X}}$:

$$\mathbf{A}_{\text{eq}} = \underbrace{\mathbf{U}}_{\text{projection}}\underbrace{(\mathbf{U}^\top\mathbf{X}^\top\mathbf{X}\mathbf{U})^{-1}}_{\text{whitening}}\underbrace{(\mathbf{U}^\top\mathbf{X}^\top\mathbf{Y}\mathbf{V})}_{\text{decorrelation}}\underbrace{(\mathbf{V}^\top\mathbf{V})^{-1}\mathbf{V}^\top}_{\text{projection back}}$$

Chapter 4

Beyond Least Squares: Optimization and Neural Networks

4.1 Nonlinear Least Squares

Up to this point, we've restricted ourselves to *linear* regression models. That is, our prediction $\hat{y} = \mathbf{w}^\top \mathbf{x}$ is a linear function of the input $\mathbf{x}$. This holds even in the case of least-squares polynomial regression — while the predicted value is not a linear function of the raw input $\mathbf{x}$, it is still a linear function of the augmented polynomial feature input $\phi(\mathbf{x})$.

Effectively, we have been able to form nonlinear models by manually augmenting features to the input. Now what if instead of using a linear function of the augmented input, we could use an arbitrary nonlinear function $f(\mathbf{x}; \mathbf{w})$ *directly* of the raw input $\mathbf{x}$? This approach is often more expressive and robust, because it removes the burden of augmenting expressive features to the input. As a motivating example, consider the problem of estimating the 2D position $\mathbf{w} = (w_1, w_2)$ of a robot. We are given noisy distance estimates $Y_i \in \mathbb{R}$ from n sensors whose positions $\mathbf{x}_i \in \mathbb{R}^2$ are fixed and known. Since we are predicting *distance*, it is reasonable to use the model $f(\mathbf{x}; \mathbf{w}) = \|\mathbf{x} - \mathbf{w}\|_2$. This model is clearly more appropriate than restricting ourselves to a linear model with augmented features — in that case, what exactly would the augmented features be?

Note however that for most problems, we are not given the form or structure of the model. Consider the following example: we are trying to predict a user's income based on their occupation, age, education, etc... It is not exactly clear what model we should use. Rather than specifying a specific family of nonlinear functions, we are instead interested in a universal function appropriator $f(\mathbf{x}; \mathbf{w})$ which can approximate any function $f(\mathbf{x})$ with appropriate parameters $\mathbf{w}$. This will be the basis for **neural networks**, which we will study in detail later.

For the purposes of our discussion, let us assume that we are given a model f, an arbitrary (nonlinear) differentiable function parameterized by $\mathbf{w}$:

$$Y_i = f(\mathbf{x}_i; \mathbf{w}) + Z_i, \quad Z_i \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2), \quad i = 1, \ldots, n$$

which can equivalently be expressed as $Y_i \mid \mathbf{x}_i \sim \mathcal{N}(f(\mathbf{x}_i; \mathbf{w}), \sigma^2)$. We are interested in finding the parameters $\hat{\mathbf{w}}_{\text{MLE}}$ that maximize the likelihood of the data:

$$\hat{\mathbf{w}}_{\text{MLE}} = \arg\max_{\mathbf{w}} \ell(\mathbf{w}; \mathbf{X}, \mathbf{y})$$

$$
\begin{aligned}
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{n} \log p(y_i \mid \mathbf{x}_i, \mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f(\mathbf{x}_i;\mathbf{w}))^2}{2\sigma^2}\right) \\
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{n} \left[-\frac{1}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}(y_i - f(\mathbf{x}_i;\mathbf{w}))^2\right] \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i;\mathbf{w}))^2
\end{aligned}
$$

Observe that the objective function is a sum of squared residuals as we've seen before, but now the function f is nonlinear. For this reason this method is called **nonlinear least squares**.

Motivated by the MLE formulation above, our goal is to solve the following optimization problem:

$$\min_{\mathbf{w}} L(\mathbf{w}) = \min_{\mathbf{w}} \frac{1}{2}\sum_{i=1}^{n} (y_i - f(\mathbf{x}_i;\mathbf{w}))^2$$

One way to solve this optimization problem is to find all of its critical points and choose the point that minimizes the objective. From **first-order optimality conditions**, the gradient of the objective function at any minimum must be zero:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i;\mathbf{w}))\nabla_{\mathbf{w}} f(\mathbf{x}_i;\mathbf{w}) = \mathbf{0}$$

In compact matrix notation:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = J(\mathbf{w})^\top (\mathbf{y} - F(\mathbf{w})) = \mathbf{0}$$

where

$$F(\mathbf{w}) = \begin{bmatrix} f(\mathbf{x}_1;\mathbf{w}) \\ \vdots \\ f(\mathbf{x}_n;\mathbf{w}) \end{bmatrix}, \quad J(\mathbf{w}) = \begin{bmatrix} \nabla_{\mathbf{w}} f(\mathbf{x}_1;\mathbf{w})^\top \\ \vdots \\ \nabla_{\mathbf{w}} f(\mathbf{x}_n;\mathbf{w})^\top \end{bmatrix}$$

J is also referred to as the **Jacobian** of F. Observe that in the special case when f is linear in $\mathbf{w}$ (i.e. $f(\mathbf{x}_i;\mathbf{w}) = \mathbf{w}^\top \mathbf{x}_i$), the gradient $\nabla_{\mathbf{w}} L(\mathbf{w})$ will only depend $\mathbf{w}$ in $F(\mathbf{w})$ because the term $\nabla_{\mathbf{w}} f(\mathbf{x}_i;\mathbf{w})$ will only depend on $\mathbf{x}_i$:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i)\nabla_{\mathbf{w}}(\mathbf{w}^\top \mathbf{x}_i) = \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i)\mathbf{x}_i = \mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$$

and we can derive a **closed-form** solution for $\mathbf{w}$, arriving at the OLS solution:

$$
\begin{aligned}
\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) &= \mathbf{0} \\
\mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\mathbf{w} &= \mathbf{0} \\
\mathbf{X}^\top\mathbf{y} &= \mathbf{X}^\top\mathbf{X}\mathbf{w} \\
\mathbf{w} &= (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}
\end{aligned}
$$

In the general case where f is nonlinear in $\mathbf{w}$, it is not necessarily possible to derive a closed-form solution for $\mathbf{w}$, for a few reasons. First of all, without additional assumptions on f, the

NLS objective may not be convex. Therefore there may exist values of $\mathbf{w}$ that are not global minima, but nonetheless $\nabla_{\mathbf{w}} L(\mathbf{w}) = \mathbf{0}$ — they could be local minima, saddle points, or worse, local maxima! Second of all, even if the objective is convex, we may not be able to solve the equation $J(\mathbf{w})^\top(\mathbf{y} - F(\mathbf{w})) = \mathbf{0}$ for $\mathbf{w}$. Given the challenges that nonlinear least squares introduces over linear least squares, we need a principled approach to solve problems that have no closed-form solution, preferably agnostic of the specific objective itself.

4.2 Optimization

In the specific case of nonlinear least squares the objective is

$$\min_{\mathbf{w}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i; \mathbf{w}))^2$$

but more generally, as we move into the realm of neural networks and beyond, we will be solving arbitrary problems of the form

$$\min_{\mathbf{w} \in \mathcal{X}} f(\mathbf{w})$$

over an arbitrary *continuous* objective function $f : \mathbb{R}^d \to \mathbb{R}$ and arbitrary domain $\mathcal{X}$. If we are able to solve this more general class of problems, then we can solve nonlinear least squares as a specific instance of the problem. Solving such problems is the focus of **optimization**, an extensive field that has applications in control theory, finance, and machine learning.

In optimization we are interested in finding the **global minimum** of a function. In the pursuit of finding the global minimum, we may encounter **local minima** along the way, which are suboptimal but may actually be close enough to the global minimum. More broadly, such points belong to the class of **critical points**, the "interesting" points of deflection that we may want to consider when finding minima:

(i) **local minimum**: a differentiable point $\mathbf{w} \in \mathcal{X}$ such that there exists a *neighborhood* around $\mathbf{w}$ where $f(\mathbf{w})$ attains the minimum value

(ii) **local maximum**: a differentiable point $\mathbf{w} \in \mathcal{X}$ such that there exists a *neighborhood* around $\mathbf{w}$ where $f(\mathbf{w})$ attains the maximum value

(iii) **saddle point**: a differentiable point $\mathbf{w} \in \mathcal{X}$ such that for all *neighborhoods* around $\mathbf{w}$, there exists $\mathbf{u}, \mathbf{v}$ such that $f(\mathbf{u}) \leq f(\mathbf{w}) \leq f(\mathbf{v})$

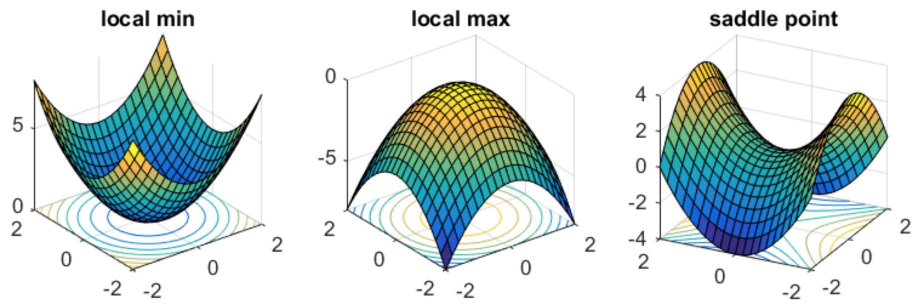


Figure 4.1: Source: Off the Convex Path

Technically, local minima must exist within a *neighborhood* of the domain and be *differentiable*, and our analysis of minima isn't complete without also considering the following as potential minima:

(i) **boundary points**: points $\mathbf{w} \in \mathcal{X}$ that can be approached from both $\mathcal{X}$ and outside $\mathcal{X}$, or more intuitively, points that lie of the "boundary" of $\mathcal{X}$

(ii) **non-differentiable points**: points at which the derivative is undefined, such as points with "kinks"

For the remainder of our discussion, we will assume that we are solving unconstrained optimization problems with differentiable functions, so that we will only have to consider critical points in our analysis.

The categorization in the previous section is helpful, but how exactly can we determine which points in the domain are critical points? As it turns out, the set of all critical points is simply the set of points at which the gradient is zero. Given that f is continuously differentiable, the **gradient** is defined as the vector of partial derivatives of f, denoted by

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \vdots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}$$

Since the set of points for which the gradient is zero in turn define the set of critical points, the gradient being zero is a necessary condition for local minima.

Proposition 1. *If $\mathbf{w}^*$ is a local minimum of f and f is continuously differentiable in a neighborhood of $\mathbf{w}^*$, then $\nabla f(\mathbf{w}^*) = \mathbf{0}$.*

Proof. See math4ml. □

This justifies the technique we have been using on numerous occasions so far to solve least squares problems: setting the gradient of the objective function to zero and solving the corresponding equation. Note however, that while setting the gradient to zero is a necessary condition for local minima, it is not a *sufficient* condition. In many circumstances, the function that we are optimizing may not have a local minima, and generally setting the gradient to zero could yield local maxima or saddle points. Even if all critical points were minima, we would still have to solve the corresponding equation $\nabla f(\mathbf{w}) = \mathbf{0}$, which is not always trivial. In the cases when solving this equation is intractable, we say that no **closed-form solution** exists, and therefore an *iterative algorithm* is needed to solve the optimization problem. Even if a problem does have a closed form solution that we can directly find, it may still be much more computationally efficient to solve the problem with iterative algorithms.

4.3 Gradient Descent

Rather than using gradients to find the closed-form solution, we can use gradients to "creep toward" a local minimum in an iterative fashion. **Gradient Descent** is an algorithm that iteratively takes small steps in the direction of *steepest descent* of the objective f. Intuitively, we can view gradient descent as a ball rolling down a hill. If we place the ball somewhere at the top of the hill, it will naturally roll down the direction of steepest descent until it reaches the bottom of the hill, at which point it may oscillate around until it eventually comes to a stop at the bottom.

Gradient descent is a simple, intuitive method that works remarkably well in practice. One question that remains is: how exactly do we determine the direction of steepest descent of a multivariate function, and what does this method have to do with gradients? Given that we are currently at a point $\mathbf{w}^{(t)}$ in the domain of the function, the direction of steepest descent is the negative of the gradient at that point, $-\nabla f(\mathbf{w}^{(t)})$. To see why, recall that the directional derivative in a unit direction $\mathbf{u}$ at $\mathbf{w}^{(t)}$ is defined as the inner product of the gradient and the direction:

$$D_{\mathbf{u}} f(\mathbf{w}^{(t)}) = \langle \nabla f(\mathbf{w}^{(t)}), \mathbf{u} \rangle = \|\nabla f(\mathbf{w}^{(t)})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta)$$

where θ is the angle between $\nabla f(\mathbf{w}^{(t)})$ and $\mathbf{u}$. Finding the direction of steepest descent entails finding the direction that minimizes the directional derivative. We can minimize the directional derivative by setting $\theta = -\pi$, which will mean that the direction $\mathbf{u}$ and $\nabla f(\mathbf{w}^{(t)})$ are opposite to each other, and thus the direction of steepest descent $\mathbf{u}^*$ is $-\nabla f(\mathbf{w}^{(t)})$ (similarly the direction of steepest *ascent* is $\nabla f(\mathbf{w}^{(t)})$). The gradient descent algorithm will take an arbitrary step in this direction, scaling the gradient by a scalar α_t.

Algorithm 4: Gradient Descent

```
Initialize $\mathbf{w}^{(0)}$ to a random point
while $f(\mathbf{w}^{(t)})$ not converged do
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla f(\mathbf{w}^{(t)})$
```

Determining this scaling α_t is dependent on the attributes of the function f. Sometimes we can set the scaling to a constant value and converge to the optimum value, whereas in other instances we need to determine an *adaptive stepsize*. A scaling that is too high may cause the algorithm to diverge from the optimal solution, whereas a scaling that is too low may cause the algorithm to converge too slowly. For certain classes of functions, there are theoretical guarantees that establish convergence, which we will state later.

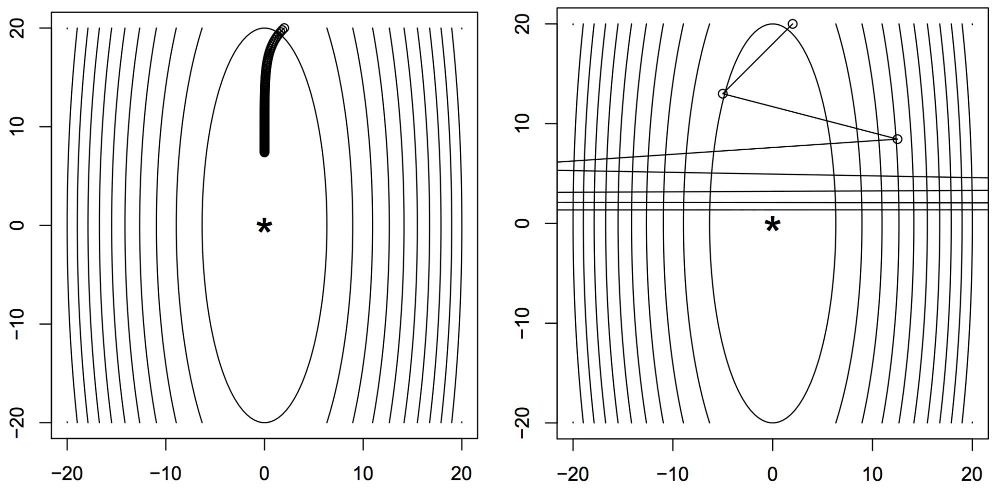


Figure 4.2: In gradient descent, stepsize matters. A small stepsize (left) will never converge to the optimal point, and a large stepsize (right) will lead to divergence. Source: CMU 10-725

Stochastic Gradient Descent

In many machine learning applications, the loss f that we want to minimize can be decomposed into a sum of functions, that is $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w})$. This holds in particular for problems involving an average over the training data, which is for example the case when we want to find the maximum likelihood estimator given i.i.d. data in a generative probabilistic model. In the standard gradient descent update, computing the gradient effectively entails computing and adding n separate gradients:

$$\nabla f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{w})$$

This standard form of gradient descent is commonly referred to as **batch gradient descent**, because it computes a full "batch" of gradients in each update. Assuming that the objective function f is deterministic, and given a fixed initial iterate $\mathbf{w}^{(0)}$, batch gradient descent is a deterministic algorithm.

One major issue with batch gradient descent is that it can be computationally expensive, because it requires computing and adding n separate gradients. In addition, due to the deterministic nature of the algorithm, it can easily get stuck at local minima and saddle points. We can mitigate these issues by deploying **stochastic gradients**. Given a fixed $\mathbf{w}$, the stochastic gradient $G(\mathbf{w})$ is a random vector-valued function which is equal to the gradient $\nabla f(\mathbf{w})$ *in expectation*, i.e. $\mathbb{E}_G[G(\mathbf{w})] = \nabla f(\mathbf{w})$, where the expectation is over the stochasticity of the gradient. We can therefore say that the stochastic gradient is an *unbiased estimate* of the true gradient. The stochastic gradient is used in place of the true gradient in the update rule:

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla G(\mathbf{w}^{(t)})$$

Mini-batch gradient descent is a stochastic variant of batch gradient descent, that instead of summing an entire "batch" of n gradients, samples and adds a random "mini-batch" of gradients over $k < n$ indices drawn from $\{1, \dots, n\}$:

$$G(\mathbf{w}) = \frac{1}{k}\sum_{i=1}^{k} \nabla f_i(\mathbf{w})$$

A major advantage of mini-batch gradient descent is that each iteration is now more computationally efficient, leading to greater progress and allowing us to monitor the performance of the algorithm faster. In addition, mini-batch gradient descent can escape local minima with more ease compared to batch gradient descent, due the noisy nature of its gradients. However note that this can also lead to instability if the stochastic gradients have high variance. For this reason, mini-batch gradient descent generally requires a higher number of overall iterations to match the performance of batch gradient descent, which can lead to expensive computational overhead. Given the appropriate choice of k, mini-batch gradient descent can be significantly more computationally efficient overall than batch gradient descent.

The special case of mini-batch gradient descent with $k = 1$ is called **stochastic gradient descent (SGD)**. In this case, we can define the stochastic gradient by just drawing an index i uniformly at random from $\{1, \dots, n\}$ and setting

$$G(\mathbf{w}) = \nabla f_i(\mathbf{w})$$

We can verify that the stochastic gradient is indeed an unbiased estimate of the true gradient:

$$\mathbb{E}_i[G(\mathbf{w})] = \mathbb{E}_i[\nabla f_i(\mathbf{w})] = \sum_{j=1}^{n} P(i = j)\nabla f_j(\mathbf{w})$$

$$= \frac{1}{n}\sum_{j=1}^{n} \nabla f_j(\mathbf{w}) = \nabla f(\mathbf{w})$$

Algorithm 5: Stochastic Gradient Descent

```
Initialize $\mathbf{w}^{(0)}$ to a random point
while $f(\mathbf{w}^{(t)})$ not converged do
    Sample a random index $i_t$ from $\{1, \ldots, n\}$
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla f_{i_t}(\mathbf{w}^{(t)})$
```

Compared to batch gradient descent, the gradient updates in SGD are significantly faster, but SGD often requires a significantly higher number of updates. In practice, mini-batch gradient descent is more effective than SGD and batch gradient descent, capturing the stability of batch gradient descent while at the same time injecting enough stochasticity to escape local minima and saddle points.

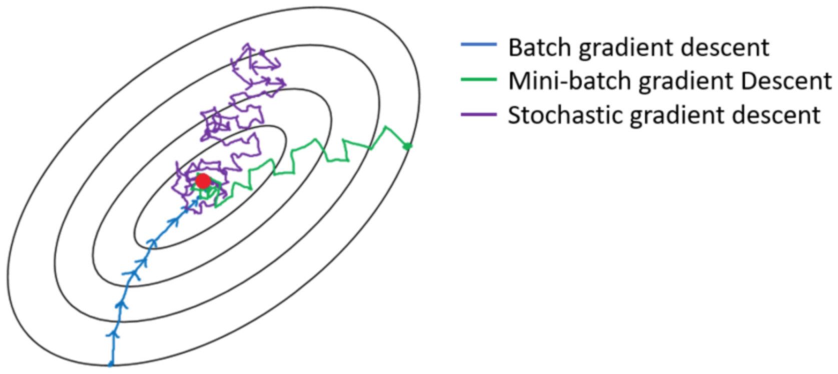


Figure 4.3: Increasing the batch size will lead to more stability at the cost of higher computational costs. Source: Towards Data Science

A fair metric that we can use to compare batch, mini-batch, and stochastic gradient descent is through the concept of **epochs**. An epoch is a measure of time — it is defined as the number of iterations in order to traverse the training data once. In the case of batch gradient descent, since all n examples comprising the training data are used to compute the gradient at each iteration, an epoch is simply equivalent to one iteration. In the case of SGD, since we only sample one example at each iteration, an epoch is equivalent to n iterations. In the case of mini-batch gradient descent, as epoch comprises of $\frac{n}{k}$ iterations. In practice, given the same number of epochs, mini-batch gradient descent tends to perform the best.

Momentum

Just as mini-batch gradient descent can lead us to escape local minima and saddle points, the stochastic nature of the algorithm can often lead to oscillations that cause instability and slow convergence. These issues are not just unique to stochastic gradients and can arise in the deterministic case, for example when the objective function is disproportionately scaled — ie. the function is elongated along one axis while being contracted along along another, giving the illusion of “ravines” in the function landscape. The disproportionate scalings cause the algorithm to make large leaps in contracted directions, while making very slow progress along elongated directions. The resulting behavior is a series of oscillations that may reach the optimal point very slowly.

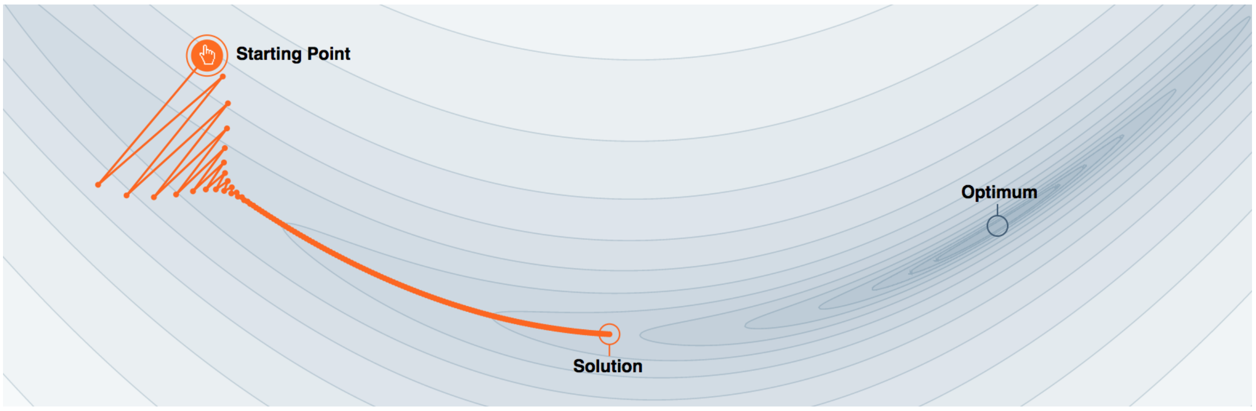


Figure 4.4: Standard gradient descent cannot converge to the optimum when the objective function is disproportionately scaled. Source: distill.pub

Polyak's heavy ball method addresses these issues by introducing a *momentum* term that adds inertia to the iterates and prevents them from deviating from the overall direction of the updates. Rather than updating the iterate $\mathbf{w}^{(t)}$ using the gradient $\nabla f(\mathbf{w}^{(t)})$, Polyak's heavy ball method uses $\nabla f(\mathbf{w}^{(t)})$ along with a history of all the gradients from the iterates seen so far. Specifically, it updates the iterates via a velocity term $\mathbf{v}^{(t)}$ that represents an exponential moving average of all of the gradients seen so far.

```
Algorithm 6: Polyak's Heavy Ball Method
Initialize $\mathbf{w}^{(0)}$ to a random point
Initialize $\mathbf{v}^{(0)}$ to $-\alpha_0 \nabla f(\mathbf{w}^{(0)})$
while $f(\mathbf{w}^{(t)})$ not converged do
    $\mathbf{v}^{(t)} \leftarrow \beta_t \mathbf{v}^{(t-1)} - \alpha_t \nabla f(\mathbf{w}^{(t)})$
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \mathbf{v}^{(t)}$
```

The velocity term is updated in the following recursive fashion:

$$\mathbf{v}^{(t)} \leftarrow \beta_t \mathbf{v}^{(t-1)} - \alpha_t \nabla f(\mathbf{w}^{(t)})$$

which when unrolled, is equivalent to

$$\begin{aligned}\mathbf{v}^{(t)} \leftarrow & - \beta_t \cdot \beta_{t-1} \dots \beta_1 \alpha_0 \nabla f(\mathbf{w}^{(0)}) \\ & - \beta_t \cdot \beta_{t-1} \dots \beta_2 \alpha_1 \nabla f(\mathbf{w}^{(1)}) \\ & \dots \\ & - \alpha_t \nabla f(\mathbf{w}^{(t)})\end{aligned}$$

which in the case when the β's and α's are constant is equivalent to

$$\mathbf{v}^{(t)} \leftarrow -\beta^t \alpha \nabla f(\mathbf{w}^{(0)}) - \beta^{t-1} \alpha \nabla f(\mathbf{w}^{(1)}) - \dots - \alpha \nabla f(\mathbf{w}^{(t)})$$

The motivation for using a moving average of the gradients is as follows: we want to downplay the directions for which the gradient is oscillating over time and boost the directions for which the gradient is constant over time. When we are in a "ravine" this has the effect of "killing" the gradient in constricted directions whose derivatives oscillate over time, accelerating convergence.

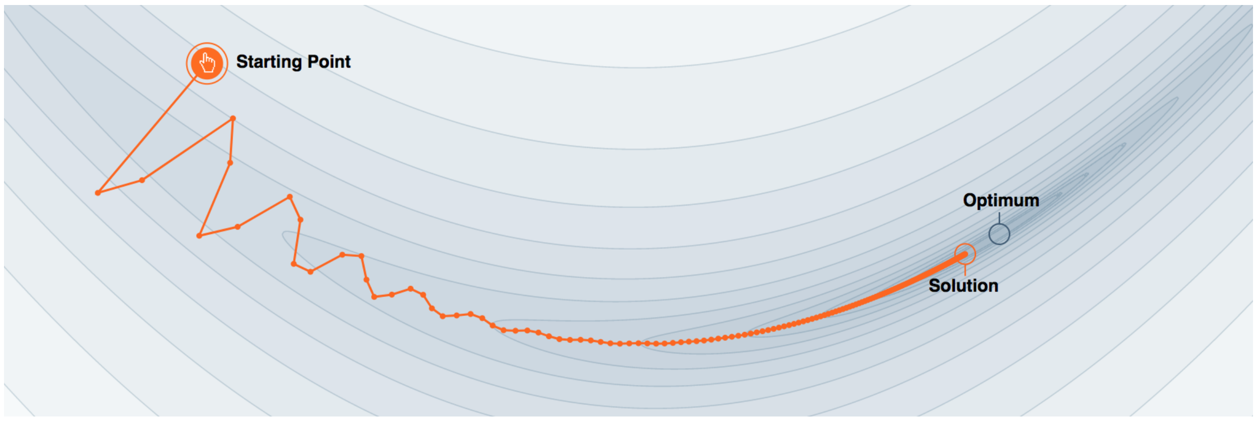


Figure 4.5: Polyak's heavy ball method uses momentum to dampen oscillations, accelerating convergence to the optimum point. Source: distill.pub

There is an alternative interpretation of Polyak's heavy ball method that is condensed to just one line:

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla f(\mathbf{w}^{(t)}) + \beta_t(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})$$

We can establish equivalence through the following manipulations:

$$\begin{aligned}\mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \mathbf{v}^{(t)} \\ &= \mathbf{w}^{(t)} + (-\alpha_t \nabla f(\mathbf{w}^{(t)}) + \beta_t \mathbf{v}^{(t-1)}) \\ &= \mathbf{w}^{(t)} - \alpha_t \nabla f(\mathbf{w}^{(t)}) + \beta_t(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})\end{aligned}$$

Polyak's heavy ball method uses information about past iterates to determine the descent direction. **Nesterov's accelerated gradient descent** improves on this reasoning, incorporating information about potential future iterates as well. The only difference in Nesterov's accelerated gradient descent is that it computes a "lookahead gradient" $\nabla f(\mathbf{w}^{(t)} + \beta_t \mathbf{v}^{(t-1)})$ instead of the gradient at the current iterate $\nabla f(\mathbf{w}^{(t)})$. Effectively, we are performing a one step "look ahead" of the gradient and moving in that direction, potentially correcting for oscillations ahead of us.

Algorithm 7: Nesterov's Accelerated Gradient Descent

```
Initialize $\mathbf{w}^{(0)}$ to a random point
Initialize $\mathbf{v}^{(0)}$ to $-\alpha_0 \nabla f(\mathbf{w}^{(0)})$
while $f(\mathbf{w}^{(t)})$ not converged do
    $\mathbf{v}^{(t)} \leftarrow \beta_t \mathbf{v}^{(t-1)} - \alpha_t \nabla f(\mathbf{w}^{(t)} + \beta_t \mathbf{v}^{(t-1)})$
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \mathbf{v}^{(t)}$
```

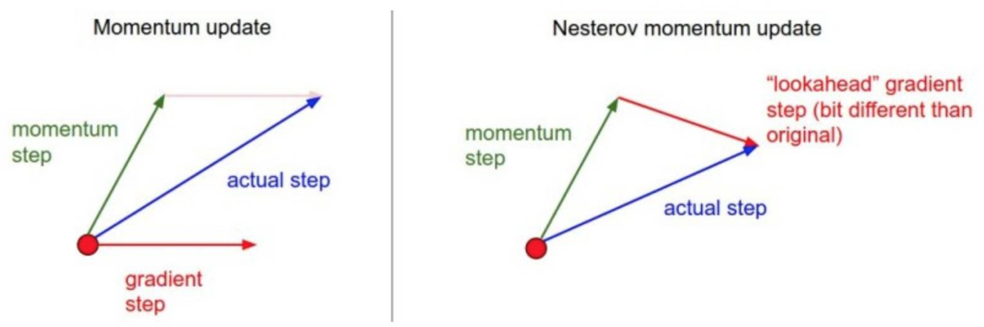


Figure 4.6: Polyak's heavy ball method applies gradient before update, while Nesterov's accelerated gradient descent applies gradient after update. Source: Stanford CS 231n

Through the same manipulations that we showed for Polyak's heavy ball method, we derive the one-line update for Nesterov's accelerated gradient descent:

$$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \nabla f(\mathbf{w}^{(t)} + \beta(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})) + \beta_t(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})$$

4.4 Line Search

Line search is another iterative optimization algorithm that, instead of taking small gradient steps, repeatedly slices the function across a 1 dimensional line and finds the minimum. Normally, finding the (global) minimum of d functions is an extremely difficult problem, but in this case we are only doing so for 1 dimensional "sliced" functions, which is a much more trivial task. Each iteration of line search entails three steps: (1) choosing a promising descent direction (or sometimes a random direction), (2) looking ahead in that direction and (roughly) finding the minimum, and (3) going to that minimum.

Algorithm 8: Line Search

```
Initialize $\mathbf{w}^{(0)}$ to a random point
while $f(\mathbf{w}^{(t)})$ not converged do
    Find a descent direction $\mathbf{u}^{(t)}$
    Find $\alpha_t \in \mathbb{R}_+$ to minimize $h(\alpha) = f(\mathbf{w}^{(t)} + \alpha \mathbf{u}^{(t)})$
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \alpha_t \mathbf{u}^{(t)}$
```

There are several options for the direction, such as the negative gradient (which is used in gradient descent). Broadly, we can pick any descent direction — a direction which entails to a negative directional derivate:

$$D_{\mathbf{u}} f(\mathbf{w}^{(t)}) = \langle \nabla f(\mathbf{w}^{(t)}), \mathbf{u} \rangle < 0$$

Another common choice is to choose an arbitrary coordinate (for example, the x/y/z coordinate in 3D), a line search variant called **coordinate descent**. Note that the minimization in the second step does not necessarily need to be exact. A simple approach is to sample several points across the line and choose the minimum, in a grid search fashion.

Line search methods offer a few advantages over gradient descent methods. For one, they do not necessarily require gradients, which can be particularly helpful in non-differentiable domains. Also, they are potentially more robust to local minima, because they find the global minima of the 1D functions ahead of them.

4.5 Convex Optimization

A critical issue with the methods we have presented so far is that they can get stuck in local minima. With gradient descent for example, moving in the direction of steepest descent is a *greedy* choice that can cause convergence to a poor local minimum, depending on the initial starting point of the algorithm.

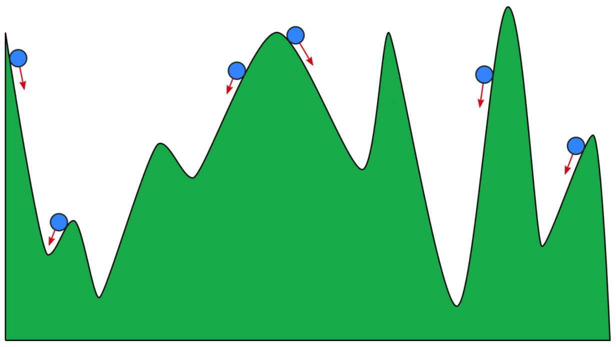


Figure 4.7: Depending on the initialization of gradient descent, the algorithm will converge to different local minima. Source: Towards Data Science

Convex functions conveniently eliminate this problem due to their "bowl shape," which ensures that all local minima are global minima.

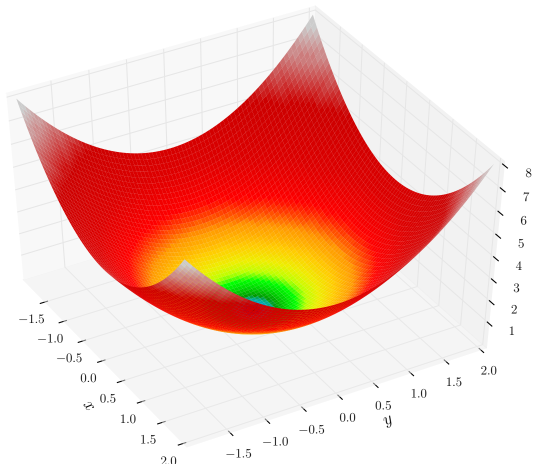


Figure 4.8: Source: Wikipedia

Due to this property, optimizing convex functions entails nice theoretical convergence rates that are otherwise not guaranteed for non-convex functions. For these reasons, there is a dedicated subfield of optimization called **convex optimization** that focuses on optimization problems with convex functions and convex constraints.

Given that $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable, the following are equivalent conditions of convexity:

(i) $f(t\mathbf{w}_1 + (1-t)\mathbf{w}_2) \le tf(\mathbf{w}_1) + (1-t)f(\mathbf{w}_2), \quad \forall \mathbf{w}_1, \mathbf{w}_2, t \in [0,1]$

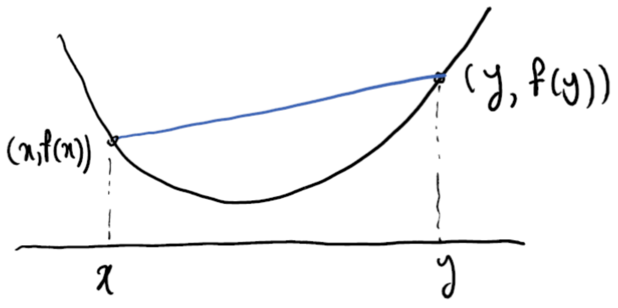


Figure 4.9: Any line segment connecting two points of a convex function must lie above the function. Source: Princeton University ORF 523

(ii) $f(\mathbf{w}_2) \geq f(\mathbf{w}_1) + \nabla f(\mathbf{w}_1)^\top(\mathbf{w}_2 - \mathbf{w}_1), \quad \forall \mathbf{w}_1, \mathbf{w}_2$

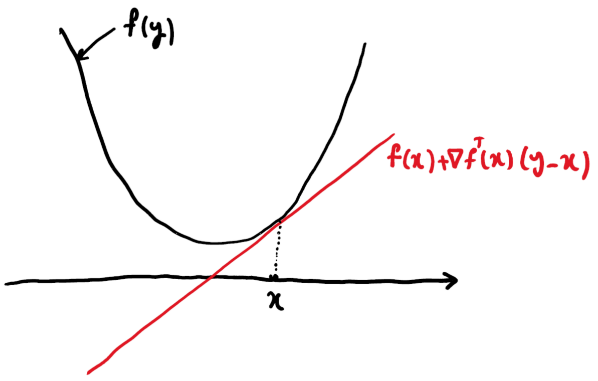


Figure 4.10: Any line tangent to a convex function must lie below the function. Source: Princeton University ORF 523

(iii) $(\nabla f(\mathbf{w}_2) - \nabla f(\mathbf{w}_1))^\top(\mathbf{w}_2 - \mathbf{w}_1) \geq 0, \quad \forall \mathbf{w}_1, \mathbf{w}_2$

(iv) $\nabla^2 f(\mathbf{w}) \succeq \mathbf{0}, \quad \forall \mathbf{w}$

Let's study these properties closely. The first condition states that for any two points $\mathbf{w}_1, \mathbf{w}_2$, the function lies below the line segment connecting $\mathbf{w}_1$ and $\mathbf{w}_2$. The next condition states that any tangent line to f must lie below the entire function. The third condition intuitively states that if $\mathbf{w}_2$ is greater than $\mathbf{w}_1$, then the derivative of $\mathbf{w}_2$ is also greater than the derivative of $\mathbf{w}_1$.

Finally, the last condition states that the "second derivative" of f is always non-negative. More rigorously, we can generalize the concept of second derivatives in higher dimensions with the Hessian. Given that f is twice continuously differentiable, we define the **Hessian** as the matrix of second partial derivatives of f, denoted by

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial w_1^2} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial w_d \partial w_1} & \cdots & \frac{\partial^2 f}{\partial w_d^2} \end{bmatrix}$$

The Hessian being PSD is a necessary condition for local minima:

Proposition 2. *If $\mathbf{w}^*$ is a local minimum of f and f is twice continuously differentiable in a neighborhood of $\mathbf{w}^*$, then $\nabla^2 f(\mathbf{w}^*)$ is positive semi-definite and $\nabla f(\mathbf{w}^*) = \mathbf{0}$.*

Proof. See math4ml. □

Unfortunately, the gradient being zero and the Hessian being PSD together are necessary but not sufficient conditions local minima (consider the function $f(w) = w^3$ or $f(w) = -w^4$). However, the gradient being zero and the Hessian being PSD in a *neighborhood* are sufficient conditions.

Proposition 3. *Suppose f is twice continuously differentiable with $\nabla^2 f$ positive semi-definite in a neighborhood of $\mathbf{w}^*$, and that $\nabla f(\mathbf{w}^*) = \mathbf{0}$. Then $\mathbf{w}^*$ is a local minimum of f.* [1]

Proof. See math4ml. □

Since for convex functions the Hessian is PSD at all points in the domain, any critical point is a local minimum. In fact, any local minimum is also a global minimum, so any point for which the gradient is zero must be the global minimum.

Proposition 4. *Let $\mathcal{X}$ be a convex set. If f is convex, then any local minimum of f in $\mathcal{X}$ is also a global minimum.*

Proof. See math4ml. □

Consequently we can find any point for which the gradient is zero and guarantee that it is the global minimum (this is exactly the case in OLS and Ridge Regression since the objective function is PSD and therefore convex). Note however, that this does not imply that the global minimum is unique — there could be several different points which achieve the global minimum.

Strong Convexity

While convex functions guarantee that all local minima are global minima, they do not guarantee that the global minimum is satisfied uniquely. Strongly convexity is an extension that guarantees this property. For a strictly positive $m \in \mathbb{R}$, a function is m-**strongly convex** if the following equivalent conditions hold:

(i) $f(t\mathbf{w}_1 + (1-t)\mathbf{w}_2) \leq tf(\mathbf{w}_1) + (1-t)f(\mathbf{w}_2) - \frac{t(1-t)m}{2}\|\mathbf{w}_2 - \mathbf{w}_1\|^2, \quad \forall \mathbf{w}_1, \mathbf{w}_2, t \in [0,1]$

(ii) $g(\mathbf{w}) = f(\mathbf{w}) - \frac{m}{2}\|\mathbf{w}\|^2$ is convex

(iii) $f(\mathbf{w}_2) \geq f(\mathbf{w}_1) + \nabla f(\mathbf{w}_1)^\top(\mathbf{w}_2 - \mathbf{w}_1) + \frac{m}{2}\|\mathbf{w}_2 - \mathbf{w}_1\|^2, \quad \forall \mathbf{w}_1, \mathbf{w}_2$

(iv) $(\nabla f(\mathbf{w}_2) - \nabla f(\mathbf{w}_1))^\top(\mathbf{w}_2 - \mathbf{w}_1) \geq m\|\mathbf{w}_2 - \mathbf{w}_1\|^2, \quad \forall \mathbf{w}_1, \mathbf{w}_2$

(v) $\nabla^2 f(\mathbf{w}) \succeq m\mathbf{I}, \quad \forall \mathbf{w}$

The conditions for strong convexity are identical to those for convex functions, but with an additional term involving m. Strongly convex functions provide several advantages over general convex functions. From the third condition, we see that strongly convex functions can be lower bounded by a quadratic function, which establishes the *uniqueness* of a global minimum.

[1] A subtle point: if $\nabla^2 f(\mathbf{w}^*)$ is *positive definite* and $\nabla f(\mathbf{w}^*) = \mathbf{0}$, then $\mathbf{w}^*$ is a strict local minimum. We do not have to check that the Hessian is PSD in a neighborhood of $\mathbf{w}^*$, as this condition is implied from the fact that f is twice continuously differentiable.

Proposition 5. *Let $\mathcal{X}$ be a convex set. If f is strongly convex, then there exists at exactly one local minimum of f in $\mathcal{X}$. Consequently, it is the unique global minimum of f in $\mathcal{X}$.*

Proof. See math4ml. □

If the Hessian of $\nabla^2 f$ has eigenvalues that are all strictly positive at all points, then f is m-strongly convex with m equal to the the smallest eigenvalue of $\nabla^2 f$ (over all points $\mathbf{w}$). Recall from our discussion of OLS vs. Ridge Regression that while OLS may have several solutions, Ridge Regression has a unique solution. This is because the Ridge Regression formulation is positive definite and thus strongly convex, while OLS is positive semi-definite and not necessarily strongly convex.

Smoothness

While strongly convex functions are *lower bounded* by a quadratic function, smooth functions are *upper bounded* by a quadratic function. [2]

An M**-smooth** (or more formally **Lipschitz continuous gradient**) function is one for which there exists a strictly positive $M \in \mathbb{R}$ such that

$$\|\nabla f(\mathbf{w}_2) - \nabla f(\mathbf{w}_1)\| \leq M\|\mathbf{w}_2 - \mathbf{w}_1\|, \quad \forall \mathbf{w}_1, \mathbf{w}_2 \tag{4.1}$$

This definition does not assume that f is convex. 4.1 implies all of the following equivalent conditions:

(i) $f(t\mathbf{w}_1 + (1-t)\mathbf{w}_2) \geq tf(\mathbf{w}_1) + (1-t)f(\mathbf{w}_2) - \frac{t(1-t)M}{2}\|\mathbf{w}_2 - \mathbf{w}_1\|^2, \quad \forall \mathbf{w}_1, \mathbf{w}_2, t \in [0,1]$

(ii) $f(\mathbf{w}_2) \leq f(\mathbf{w}_1) + \nabla f(\mathbf{w}_1)^\top(\mathbf{w}_2 - \mathbf{w}_1) + \frac{M}{2}\|\mathbf{w}_2 - \mathbf{w}_1\|^2, \quad \forall \mathbf{w}_1, \mathbf{w}_2$

(iii) $(\nabla f(\mathbf{w}_2) - \nabla f(\mathbf{w}_1))^\top(\mathbf{w}_2 - \mathbf{w}_1) \leq M\|\mathbf{w}_2 - \mathbf{w}_1\|^2, \quad \forall \mathbf{w}_1, \mathbf{w}_2$

(iv) $\nabla^2 f(\mathbf{w}) \preceq M\mathbf{I}, \quad \forall \mathbf{w}$

When f is convex, then the above conditions also imply 4.1, establishing equivalence among all of the conditions. Roughly speaking, smoothness is the counterpart to strong convexity, with the inequality signs flipped. If the Hessian of $\nabla^2 f$ has eigenvalues that are bounded from above, f is M-smooth with M equal to the the maximum eigenvalue of $\nabla^2 f$ (over all points $\mathbf{x}$).

Gradient Descent Convergence under Convexity

While gradient descent does not have convergence guarantees in general, we can make theoretical guarantees when the function is convex. Furthermore, strong convexity and smoothness will provide lower and upper bounds for f respectively, allowing us to achieve a significantly faster convergence rate. Assuming that the distance from the initial point $\mathbf{w}^{(0)}$ and the optimal point $\mathbf{w}^*$ is R, we have the following:

properties of f	stepsize α_t	convergence rate to $f(\mathbf{w}^*)$
convex, L-Lipschitz	$\frac{R}{L\sqrt{t}}$	$O(\frac{1}{\sqrt{t}})$
m-strongly convex, L-Lipschitz	$\frac{2}{m(t+1)}$	$O(\frac{1}{t})$
convex, M-smooth	$\frac{1}{M}$	$O(\frac{1}{t})$
m-strongly convex, M-smooth	$\frac{1}{M}$	$O(\exp(-t\frac{m}{M}))$

[2]Not to be confused with smooth functions in the context of real analysis

For detailed proofs of rates above, refer to the EE 227C lecture notes. Individually, strong convexity and smoothness will allow us to accelerate the rate of convergence from $O(\frac{1}{\sqrt{t}})$ to $O(\frac{1}{t})$. Put together, they allow us to achieve an exponential convergence rate — a significant acceleration! The quantity $\kappa = \frac{M}{m}$ is known as the **condition number** — the ratio of the largest over smallest singular value of the Hessian of f. Recall from our discussion of OLS vs. Ridge Regression that Ridge Regression adds a small penalty term $\lambda\|\mathbf{w}\|^2$ to the objective, effectively making the problem strongly convex. Since the OLS is already smooth as well, then gradient descent can achieve an exponential rate of convergence to the optimal value. The higher the value of λ, the lower the value of the condition number κ, which leads to an even faster convergence rate. This of course, comes at the costs of regularization.

4.6 Newton's Method

Up until this point, we have only considered first-order methods to optimize functions. Now, we will present **Newton's method**, an iterative method that utilizes second-order information to achieve a faster rate of convergence than existing first-order methods. Given an arbitrary twice continuously differentiable objective function f, Newton's Method iteratively minimizes the second-order Taylor expansion of the objective function. Given the current iterate $\mathbf{w}^{(t)}$, it minimizes the following objective:

$$\min_{\mathbf{w}} \bar{f}(\mathbf{w}) = f(\mathbf{w}^{(t)}) + \nabla f(\mathbf{w}^{(t)})^\top(\mathbf{w} - \mathbf{w}^{(t)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(t)})^\top \nabla^2 f(\mathbf{w}^{(t)})(\mathbf{w} - \mathbf{w}^{(t)})$$

We can minimize $\bar{f}(\mathbf{w})$ by setting its gradient to zero:

$$\nabla \bar{f}(\mathbf{w}) = \nabla f(\mathbf{w}^{(t)}) + \nabla^2 f(\mathbf{w}^{(t)})(\mathbf{w} - \mathbf{w}^{(t)}) = \mathbf{0}$$

Which leads to the update rule (otherwise known as *Newton step*)

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \nabla^2 f(\mathbf{w}^{(t)})^{-1} \nabla f(\mathbf{w}^{(t)})$$

The updates for Newton's method and gradient descent are nearly identical:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha_t \nabla f(\mathbf{w}^{(t)}) \qquad \text{(Gradient descent)}$$
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \nabla^2 f(\mathbf{w}^{(t)})^{-1} \nabla f(\mathbf{w}^{(t)}) \qquad \text{(Newton's method)}$$

We can think of gradient descent as a Newton update in which we approximate $\nabla^2 f(\mathbf{w}^{(t)})^{-1}$ by a scaled version of the identity. That is, gradient descent is equivalent to Newton's method when $\nabla^2 f(\mathbf{w}^{(t)})^{-1} = \alpha_t \mathbf{I}$ where $\mathbf{I}$ is the identity matrix.

The algorithm is as follows:

Algorithm 9: Newton's Method

```
Initialize $\mathbf{w}^{(0)}$ to a random point
while $f(\mathbf{w}^{(t)})$ not converged do
    $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \nabla^2 f(\mathbf{w}^{(t)})^{-1} \nabla f(\mathbf{w}^{(t)})$
```

Alternative Interpretation

Newton's method can equivalently be viewed as a a *root-finding algorithm* — specifically it finds the "roots" of the gradient by iteratively approximating the gradient and finding the root of the approximation. Newton's method is agnostic to the type of function that it optimizes — whether it is the gradient function, or just the objective function. At its simplest form, Newton's method can be used to find the roots of a single variable function $\varphi\colon \mathbb{R} \to \mathbb{R}$. Our goal is to find a root of the non-linear equation $\varphi(w) = 0$. Suppose we have a current estimate of the root $\varphi(w^{(t)})$. From Taylor's theorem, we can express the first-order form of $\varphi(w)$ with respect to $\varphi(w^{(t)})$ as

$$\varphi(w) = \varphi(w^{(t)}) + \varphi'(w) \cdot (w - w^{(t)}) + o(\left|w - w^{(t)}\right|)$$

given $\delta = w - w^{(t)}$ we equivalently have that

$$\varphi(w^{(t)} + \delta) = \varphi(w^{(t)}) + \varphi'(w) \cdot \delta + o(|\delta|)$$

Disregarding the $o(|\delta|)$ term, we solve (over δ) the following objective:

$$\varphi(w^{(t)}) + \varphi'(w^{(t)})\delta = 0$$

Then, $\delta = -\frac{\varphi(w^{(t)})}{\varphi'(w^{(t)})}$, leading to the iteration $w^{(t+1)} = w^{(t)} - \frac{\varphi(w^{(t)})}{\varphi'(w^{(t)})}$. We can similarly make an argument for a multivariate function $F : \mathbb{R}^d \to \mathbb{R}^d$. Our goal is to solve $F(\mathbf{w}) = \mathbf{0}$. Again, from Taylor's theorem we have that

$$F(\mathbf{w} + \Delta) = F(\mathbf{w}) + J_F(\mathbf{w})\Delta + o(\|\Delta\|)$$

where J_F is the Jacobian. This gives us $\Delta = -J_F^{-1}(\mathbf{w})F(\mathbf{w})$, and the iteration

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - J_F^{-1}(\mathbf{w}^{(t)})F(\mathbf{w}^{(t)})$$

In the context of optimization, Newton's method is a special application of this root-finding method, applied to the gradient function. That is, given that we are minimizing $f\colon \mathbb{R}^d \to \mathbb{R}$, Newton's method finds the roots of the gradient function $\nabla f\colon \mathbb{R}^d \to \mathbb{R}^d$. It uses the update rule

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \nabla^2 f(\mathbf{w}^{(t)})^{-1} \nabla f(\mathbf{w}^{(t)})$$

as the Hessian $\nabla^2 f(\mathbf{w}^{(t)})$ of the objective function corresponds to the Jacobian $J_F^{-1}(\mathbf{w})$ of the gradient. Let's understand the motivation of Newton's method in close detail. Our goal is to find local minima for f, points for which it is necessarily true that $\nabla f(\mathbf{w}) = \mathbf{0}$. Consequently, we wish to find points for which $\nabla f(\mathbf{w}) = \mathbf{0}$. The gradient $\nabla f(\mathbf{w})$ can be difficult or even intractable to work with, so instead we work with a first-order Taylor approximation of the gradient with respect to our current iterate $\mathbf{w}^{(t)}$. We solve for the roots of the first-order gradient, update our iterate, and repeat the process. Note that while solving $\nabla f(\mathbf{w}) = \mathbf{0}$ may yield local maxima or even saddle points, we are finding the roots of the linearized gradient, which is convex — therefore any point for which the first-order approximation of the gradient is zero yields a global minimum for the approximation.

Issues with Newton's Method

There are a few issues with Newton's method that we glossed over in our analysis. In general, there are no guarantees that Newton's method can converge, and even more concerning, the algorithm may get stuck as the Hessian $\nabla^2 f(\mathbf{w}^{(t)})$ may not be invertible. Placing invertibility issues aside, the most concerning issue is that Newton's method may not even be attempting to minimize the objective function. To see why, recall that the goal of each Newton step is to minimize the second-order approximation, which we do so by setting the gradient of the approximation to zero. This is not a sound step, as it may yield saddle points or maxima. This can happen when the Hessian $\nabla^2 f(\mathbf{w}^{(t)})$ has non-positive eigenvalues. In order to ensure that the second order approximation $\bar{f}(\mathbf{w})$ yields a unique global minimum, we must ensure that it is strongly convex. We can do so by regularizing the objective $f(\mathbf{w})$ with an additional $\lambda\|\mathbf{w}\|^2$ term, with an appropriately chosen λ that shifts all of the eigenvalues of the objective to be positive.

Even when the objective is strongly convex, Newton's method can be quite unpredictable. For example, consider the function

$$f(w) = \sqrt{w^2+1}$$

essentially a smoothed version of the absolute value $|x|$. Clearly, the function is minimized at $w^* = 0$. Calculating the necessary derivatives for Newton's method, we find

$$\begin{aligned} f'(w) &= \frac{w}{\sqrt{w^2+1}} \\ f''(w) &= (1+w^2)^{-3/2}\,. \end{aligned}$$

Note that $f(w)$ is strongly convex since its second derivative strictly positive and 1-smooth ($|f'(w)| < 1$). The Newton step for minimizing $f(w)$ is

$$w^{(t+1)} = w^{(t)} - \frac{f'(w^{(t)})}{f''(w^{(t)})} = -{w^{(t)}}^3\,.$$

The behavior of this algorithm depends on the magnitude of $w^{(t)}$. In particular, we have the following three regimes

$$\begin{cases} |w^{(t)}| < 1 & \text{Algorithm converges } \textit{cubically} \\ |w^{(t)}| = 1 & \text{Algorithm oscillates between } -1 \text{ and } 1 \\ |w^{(t)}| > 1 & \text{Algorithm diverges} \end{cases}$$

This example shows that even for strongly convex functions with Lipschitz gradients that Newton's method is only guaranteed to converge locally. To avoid divergence, a popular technique is to use a *damped* step–size:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha_t \nabla^2 f(\mathbf{w}^{(t)})^{-1} \nabla f(\mathbf{w}^{(t)})$$

Convergence Analysis

We can ensure that Newton's method converges, if all of the following conditions are met:

1. $\nabla^2 f(\mathbf{w})$ is Lipschitz: $\|\nabla^2 f(\mathbf{w}) - \nabla^2 f(\mathbf{w}')\| \leq \|\mathbf{w} - \mathbf{w}'\|$
2. $\exists \mathbf{w}^*$ s.t. $\nabla f(\mathbf{w}^*) = \mathbf{0}$ and $\nabla^2 f(\mathbf{w}^*) \succeq \alpha \mathbf{I}$ and $\|\mathbf{w}^{(0)} - \mathbf{w}^*\| \leq \frac{\alpha}{2}$

These conditions combined establish *local convergence* of Newton's method to a local minimum. That is, given that the initial point $\mathbf{w}^{(0)}$ is sufficiently close to the local minimum, the Hessian is positive definite at the local minimum, and the Hessian is Lipschitz (meaning that its rate of change can be bounded), we can ensure a quadratic convergence rate of $O(e^{-e^t})$, which is significantly faster than the fastest rate for gradient descent that we have seen, $O(e^{-t})$. Note however, that each Newton step will involve inverting the Hessian, which itself is an expensive $O(d^3)$ operation that becomes impractical for high dimensional functions.

4.7 Gauss-Newton Algorithm

Let's revisit the nonlinear least squares problem. We can try to apply all of the techniques and approaches we have covered so far to solve this problem, but there is a specialized algorithm for solving the nonlinear least squares problem, called **Gauss-Newton**. The Gauss-Newton algorithm has parallels to Newton's method, as they both repeatedly make linearly approximations of an objective and solve that approximation. At each iteration, this method linearly approximates the function F about the current iterate and solves a least-squares problem involving the linearization in order to compute the next iterate.

Let's say that we have a "guess" for $\mathbf{w}$ at iteration k, which we denote $\mathbf{w}^{(k)}$. We consider the first-order approximation of $F(\mathbf{w})$ about $\mathbf{w}^{(k)}$:

$$\begin{aligned} F(\mathbf{w}) \approx \tilde{F}(\mathbf{w}) &= F(\mathbf{w}^{(k)}) + \frac{\partial}{\partial \mathbf{w}} F(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) \\ &= F(\mathbf{w}^{(k)}) + J(\mathbf{w}^{(k)})\Delta\mathbf{w} \end{aligned}$$

where $\Delta\mathbf{w} := \mathbf{w} - \mathbf{w}^{(k)}$.

Now that $\tilde{F}$ is linear in $\Delta\mathbf{w}$ (the Jacobian and F are just constants: functions evaluated at $\mathbf{w}^{(k)}$), our objective is convex and we can perform linear least squares to form the closed form solution for $\Delta\mathbf{w}$. Applying the first-order optimality condition to the objective $\tilde{F}$ yields the following equation:

$$\mathbf{0} = J_{\tilde{F}}(\mathbf{w})^\top(\mathbf{y} - \tilde{F}(\mathbf{w})) = J(\mathbf{w}^{(k)})^\top \left(\mathbf{y} - \left(F(\mathbf{w}^{(k)}) + J(\mathbf{w}^{(k)})\Delta\mathbf{w}\right)\right)$$

Note that the Jacobian of the linearized function $\tilde{F}$, evaluated at any $\mathbf{w}$, is precisely $J(\mathbf{w}^{(k)})$. Denoting $\mathbf{J} = J(\mathbf{w}^{(k)})$ and $\Delta\mathbf{y} := \mathbf{y} - F(\mathbf{w}^{(k)})$ for brevity, we have

$$\begin{aligned} \mathbf{J}^\top(\Delta\mathbf{y} - \mathbf{J}\Delta\mathbf{w}) &= \mathbf{0} \\ \mathbf{J}^\top\Delta\mathbf{y} &= \mathbf{J}^\top\mathbf{J}\Delta\mathbf{w} \\ \Delta\mathbf{w} &= (\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\Delta\mathbf{y} \end{aligned}$$

Comparing this solution to OLS, we see that it is effectively solving

$$\Delta\mathbf{w} = \arg\min_{\delta\mathbf{w}} \|\mathbf{J}\delta\mathbf{w} - \Delta\mathbf{y}\|^2$$

where $\mathbf{J}$ represents $\mathbf{X}$ in OLS, $\Delta\mathbf{y}$ represents $\mathbf{y}$ in OLS, and $\delta\mathbf{w}$ represents $\mathbf{w}$ in OLS. At each iteration we are effectively minimizing the objective with respect to the linearization of F at the current iterate $\mathbf{w}^{(k)}$. Since $\delta F \approx \mathbf{J}\delta\mathbf{w}$, we can expect that the minimization with respect to $\tilde{F}$ is also optimal with respect to F in the local region around $\mathbf{w}^{(k)}$. Recalling that $\Delta\mathbf{w} = \mathbf{w} - \mathbf{w}^{(k)}$, we can improve upon our current guess $\mathbf{w}^{(k)}$ with the update

$$\begin{aligned} \mathbf{w}^{(k+1)} &= \mathbf{w}^{(k)} + \Delta\mathbf{w} \\ &= \mathbf{w}^{(k)} + (\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\Delta\mathbf{y} \end{aligned}$$

Algorithm 10: Gauss-Newton

```
Initialize $\mathbf{w}^{(0)}$ with some guess
while $\mathbf{w}^{(k)}$ has not converged do
    Compute Jacobian with respect to the current iterate: $\mathbf{J} = J(\mathbf{w}^{(k)})$
    Compute $\Delta\mathbf{y} = \mathbf{y} - F(\mathbf{w}^{(k)})$
    Update: $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + (\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\Delta\mathbf{y}$
```

Note that the solution will depend on the initial value $\mathbf{w}^{(0)}$ in general. There are several choices for measuring convergence. Some common choices include testing changes in the objective value:

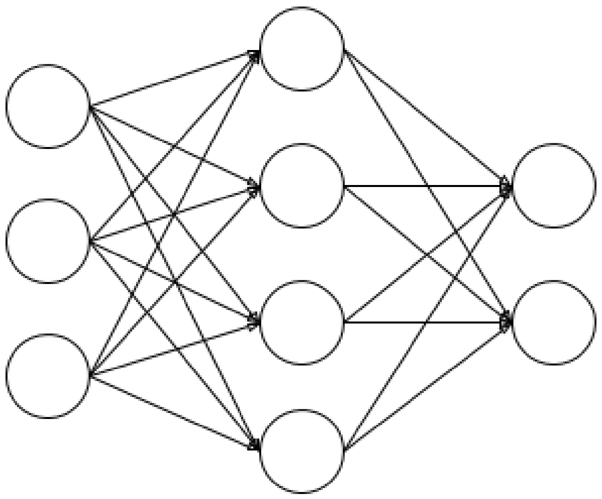
$$\left|\frac{L^{(k+1)} - L^{(k)}}{L^{(k)}}\right| \leq \text{threshold}$$

or in the iterates themselves:

$$\max_j \left|\frac{\Delta w_j}{w_j^{(k)}}\right| \leq \text{threshold}$$

4.8 Neural Networks

Neural networks are a class of compositional function approximators. They come in a variety of shapes and sizes. In this class, we will only discuss **feedforward** neural networks, those networks whose computations can be modeled by a directed acyclic graph.[3] The most basic (but still commonly used) class of feedforward neural networks is the **multilayer perceptron**. Such a network might be drawn as follows:



Computation flows left-to-right. The circles represent **nodes**, a.k.a. **units** or **neurons**, which are loosely based on the behavior of actual neurons in the brain. Observe that the nodes are organized into **layers**. The first (left-most) layer is called the **input layer**, the last (right-most) layer is called the **output layer**, and any other layers (there is only one here, but there could be multiple) are referred to as **hidden layers**. The dimensionality of the input and output layers is determined by the function we want the network to compute. For a function from $\mathbb{R}^d$ to $\mathbb{R}^k$, we should have d input nodes and k output nodes. The number and sizes of the hidden layers are hyperparameters to be chosen by the network designer.

Note that in the diagram above, each non-input layer has the following property: every node in that layer is connected to every node in the previous layer. Layers that have this property are described as **fully connected**.[4] Each edge in the graph has an associated weight, which is the

[3] There are also **recurrent** neural networks whose computation graphs have cycles.
[4] Later we will learn about **convolutional layers**, which have a different connectivity structure.

strength of the connection from the input node in one layer to the node in the next layer. Each node computes a weighted sum of its inputs, with these connection strengths being the weights, and then applies a nonlinear function which is variously referred to as the **activation function** or the **nonlinearity**. Concretely, if $\mathbf{w}_i$ denotes the weights and σ_i denotes the activation function of node i, it computes the function

$$\mathbf{x} \mapsto \sigma_i({\mathbf{w}_i}^\top \mathbf{x})$$

Let us denote the number of (non-input) layers by L, the number of units in layer $\ell \in \{0, \dots, L\}$ by n_ℓ (here n_0 is the size of the input layer), and the nonlinearity for layer $\ell \in \{1, \dots, L\}$ by $\boldsymbol{\sigma}_\ell : \mathbb{R}^{n_\ell} \to \mathbb{R}^{n_\ell}$. The weights for every node in layer ℓ can be stacked (as rows) into a matrix of weights $\mathbf{W}_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$. Then layer ℓ performs the computation

$$\mathbf{x} \mapsto \boldsymbol{\sigma}_\ell(\mathbf{W}_\ell \mathbf{x})$$

Since the output of each layer is passed as input to the next layer, the function represented by the entire network can be written

$$\mathbf{x} \mapsto \boldsymbol{\sigma}_L(\mathbf{W}_L \sigma_{L-1}(\cdots \boldsymbol{\sigma}_2(\mathbf{W}_2 \boldsymbol{\sigma}_1(\mathbf{W}_1 \mathbf{x})) \cdots))$$

This is what we mean when we describe neural networks as *compositional.*

Note that in most layers, the nonlinearity will be the same for each node within that layer, so it makes sense to refer to a scalar function $\sigma_\ell : \mathbb{R} \to \mathbb{R}$ as "the" nonlinearity for layer ℓ, and apply it element-wise:

$$\boldsymbol{\sigma}_\ell(\mathbf{x}) = \begin{bmatrix} \sigma_\ell(x_1) \\ \vdots \\ \sigma_\ell(x_{n_\ell}) \end{bmatrix}$$

The principle exception here is the **softmax** function $\boldsymbol{\sigma} : \mathbb{R}^k \to \mathbb{R}^k$ defined by

$$\boldsymbol{\sigma}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$

which is often used to produce a discrete probability distribution over k classes. Note that every entry of the softmax output depends on every entry of the input. Also, softmax preserves ordering, in the sense that sorting the indices $i = 1, \dots, k$ by the resulting value $\boldsymbol{\sigma}(\mathbf{x})_i$ yields the same ordering as sorting by the input value x_i. In other words, more positive x_i leads to larger $\boldsymbol{\sigma}(\mathbf{x})_i$. This nonlinearity is used most commonly (but not always) at the output layer of the network.

Expressive power

It is the repeated combination of nonlinearities that gives deep neural networks their remarkable expressive power. Consider what happens when we remove the activation functions (or equivalently, set them to the identity function): the function computed by the network is

$$\mathbf{x} \mapsto \underbrace{\mathbf{W}_L \mathbf{W}_{L-1} \cdots \mathbf{W}_2 \mathbf{W}_1}_{=:\widetilde{\mathbf{W}}} \mathbf{x}$$

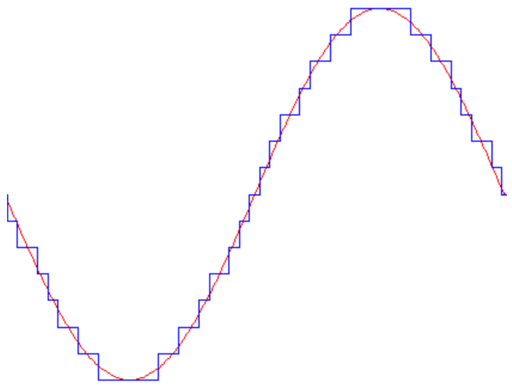
which is linear in its input! Moreover, the size of the smallest layer restricts the rank of $\widetilde{\mathbf{W}}$, as

$$\operatorname{rank}(\widetilde{\mathbf{W}}) \leq \min_{\ell \in \{1, \dots, L\}} \operatorname{rank}(\mathbf{W}_\ell) \leq \min_{\ell \in \{0, \dots, L\}} n_\ell$$

Despite having many layers of computation, this class of networks is not very expressive; it can only represent linear functions.

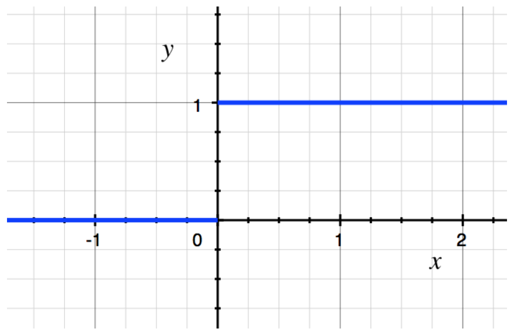
We would like to produce a class of networks that are **universal function approximators**. This essentially means that given any continuous function, we can choose a network in this class such that the output of the circuit can be made arbitrarily close to the output of the given function for all given inputs. We make a more precise statement later.

A key observation is that piecewise-constant functions are universal function approximators:



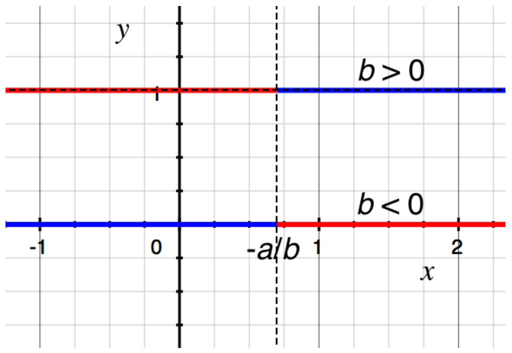
The nonlinearity we use, then, is the step function:

$$\sigma(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$



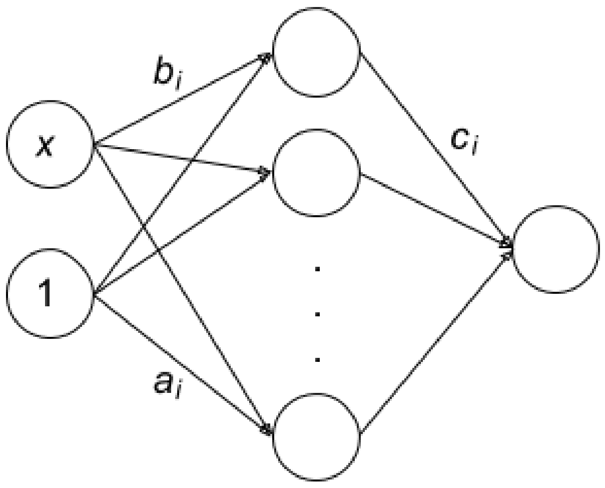
We can build very complicated functions from this simple step function by combining translated and scaled versions of it. Observe that

- If $a, b \in \mathbb{R}$, the function $x \mapsto \sigma(a + bx)$ is a translated (and, depending on the sign of b, possibly flipped) version of the step function:



- If $c \neq 0$, the function $x \mapsto c\sigma(x)$ is a vertically scaled version of the step function.

It turns out that only one hidden layer is needed for universal approximation, and for simplicity we assume a one-dimensional input. Thus from here on we consider networks with the following structure:



The input x is one-dimensional, and the weight on x to node j is b_j. We also introduce a constant 1, whose weight into node j is a_j. (This is referred to as the *bias*, but it has nothing to do with bias in the sense of the bias-variance tradeoff. It's just there to provide the node with the ability to shift its input.) The function implemented by the network is

$$h(x) = \sum_{j=1}^{k} c_j \sigma(a_j + b_j x)$$

where k is the number of hidden units.

Choosing weights

With a proper choice of a_j, b_j, and c_j, this function can approximate any continuous function we want. But the question remains: given some target function, how do we choose these parameters in the appropriate way?

Let's try a familiar technique: least squares. Assume we have training data $\{(x_i, y_i)\}_{i=1}^n$. We aim to solve the optimization problem

$$\min_{\mathbf{a},\mathbf{b},\mathbf{c}} \underbrace{\sum_{i=1}^n (y_i - h(x_i))^2}_{f(\mathbf{a},\mathbf{b},\mathbf{c})}$$

To run gradient descent, we need derivatives of the loss with respect to our optimization variables. We compute via the chain rule

$$\frac{\partial (y_i - h(x_i))^2}{\partial c_j} = -2(y_i - h(x_i))\frac{\partial h(x_i)}{\partial c_j} = 2(y_i - h(x_i))\sigma(a_j + b_j x_i)$$

We see that if this particular step is "off", as in $\sigma(a_j + b_j x_i) = 0$, then

$$\frac{\partial (y_i - h(x_i))^2}{\partial c_j} = 2(y_i - h(x_i))\underbrace{\sigma(a_j + b_j x_i)}_{0} = 0$$

so no update will be made for that example. More egregiously, consider the derivatives with respect to a_j[5]:

$$\frac{\partial f}{\partial a_j} = \sum_{i=1}^n -2(y_i - h(x_i))\underbrace{\frac{\partial h(x_i)}{\partial a_j}}_{0} = 0$$

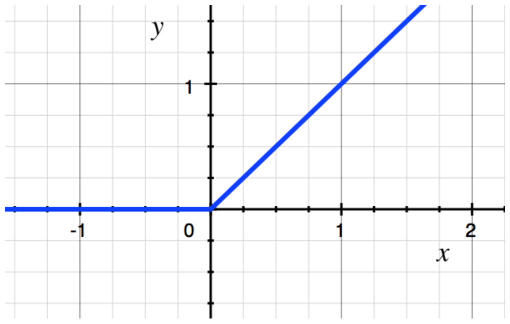
and b_j:

$$\frac{\partial f}{\partial b_j} = \sum_{i=1}^n -2(y_i - h(x_i))\underbrace{\frac{\partial h(x_i)}{\partial b_j}}_{0} = 0$$

Since gradient descent changes weights in proportion to their gradient, it will never modify **a** or **b**! Even though the step function is useful for the purpose of showing the approximation capabilities of neural networks, it is seldom used in practice because it cannot be trained by conventional gradient-based methods.

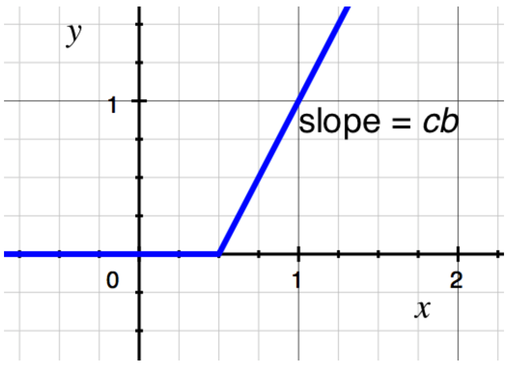
The next simplest universal approximator is the class of **piecewise-linear functions**. Just as piecewise-constant functions can be achieved by combinations of the step function as a nonlinearity, piecewise-linear functions can be achieved by combinations of the *rectified linear unit* (ReLU) function

$$\sigma(x) = \max\{0, x\}$$



[5] Technically, the derivative of σ is not defined at zero, where there is a discontinuity. However it is defined (and zero) everywhere else. In practice, we will almost never hit the point of discontinuity because it is a set of measure zero.

Depending on the weights **a** and **b**, our ReLUs can move to the left or right, increase or decrease their slope, and flip direction.



Let us calculate the gradients again, assuming we replace the step functions by ReLUs:

$$\frac{\partial f}{\partial c_j} = \sum_{i=1}^{n} -2(y_i - h(x_i)) \max\{0, a_j + b_j x_i\}$$

$$\frac{\partial f}{\partial a_j} = \sum_{i=1}^{n} -2(y_i - h(x_i)) c_j \frac{\partial}{\partial a_j} \max\{0, a_j + b_j x_i\} = \sum_{k=1}^{n} -2(y_i - h(x_i)) c_j \left(\begin{cases} 0 & \text{if } a_j + b_j x_i < 0 \\ 1 & \text{if } a_j + b_j x_i > 0 \end{cases} \right)$$

$$\frac{\partial f}{\partial b_j} = \sum_{i=1}^{n} -2(y_i - h(x_i)) c_j \frac{\partial}{\partial b_j} \max\{0, a_j + b_j x_i\} = \sum_{i=1}^{n} -2(y_i - h(x_i)) c_j \left(\begin{cases} 0 & \text{if } a_j + b_j x_i < 0 \\ x_i & \text{if } a_j + b_j x_i > 0 \end{cases} \right)$$

Crucially, we see that the gradient with respect to **a** and **b** is not uniformly zero, unlike with the step function.

Later we will discuss **backpropagation**, a dynamic programming algorithm for efficiently computing gradients with respect to a neural network's parameters.

Neural networks are universal function approximators

The celebrated neural network universal approximation theorem, due to Kurt Hornik[6], tells us that neural networks are universal function approximators in the following sense.

Theorem. Suppose $\sigma : \mathbb{R} \to \mathbb{R}$ is nonconstant, bounded, nondecreasing, and continuous[7], and let $S \subseteq \mathbb{R}^d$ be closed and bounded. Then for any continuous function $f : S \to \mathbb{R}$ and any $\epsilon > 0$, there exists a neural network with one hidden layer containing finitely many nodes, which we can write

$$h(\mathbf{x}) = \sum_{j=1}^{k} c_j \sigma(a_j + \mathbf{b}_j^\top \mathbf{x})$$

such that

$$|h(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

[6] See *Approximation Capabilities of Multilayer Feedforward Networks.*
[7] Both ReLU and sigmoid satisfy these requirements.

for all $\mathbf{x} \in S$.

There's some subtlety in the theorem that's worth noting. It says that for any given continuous function, there exists a neural network of finite size that uniformly approximates the given function. However, it says nothing about how well any *particular* architecture you're considering will approximate the function. It also doesn't tell us how to compute the weights.

It's also worth pointing out that in the theorem, the network consists of just one hidden layer. In practice, people find that using more layers works better.

4.9 Training Neural Networks

We have seen that first-order optimization techniques such as **gradient descent** and **stochastic gradient descent** are effective tools for minimizing differentiable cost functions. In order to implement these techniques, we need to be able to compute the gradient of the cost function with respect to the weights. The chain rule allows us to compute these derivatives in principle, but as we will see, the order of the computations matters in neural networks. The **backpropagation** algorithm takes advantage of the directed acyclic graph (DAG) nature of feedforward neural networks to calculate these derivatives efficiently.

Computational graphs

We assume that the our network can be expressed as a finite directed acyclic graph $G = (V, E)$, sometimes called the **computational graph** of the network. Each vertex $v_i \in V$ represents the result of some differentiable[8] computation. Each edge represents a computational dependency: there is an edge $(v_i, v_j) \in E$ if and only if the value computed at v_i is used to compute v_j. We denote the set of outgoing neighbors of a node v_i by

$$\text{out}(v_i) = \{v_j \in V : (v_i, v_j) \in E\}$$

Furthermore, some of these vertices have special significance. There is a vertex $\ell \in V$, representing the loss function, which contains no outgoing edges (i.e. $\text{out}(\ell) = \varnothing$). There is also some subset of vertices $W \subset V$ representing the trainable parameters of the network. Our objective is to efficiently calculate $\frac{\partial \ell}{\partial w_i}$ for each $w_i \in W$.

The primary mathematical tool employed in backpropagation is the chain rule. This allows us to write

$$\frac{\partial \ell}{\partial v_i} = \sum_{v_j \in \text{out}(v_i)} \frac{\partial \ell}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

The intuition here is that the value computed at v_i affects potentially all of the vertices to which it is an input, and each of those vertices affects the loss in some way. The total contribution of v_i to the loss must be summed over these downstream effects.

We could expand recursively to get an expression for each weight:

$$\frac{\partial \ell}{\partial w_i} = \sum_{v_j \in \text{out}(w_i)} \frac{\partial \ell}{\partial v_j} \frac{\partial v_j}{\partial w_i}$$

[8] A number of common neural network operations, such as the ReLU activation function, are not everywhere differentiable. In practice it is sufficient to be differentiable except at finitely many points.

$$= \sum_{v_j \in \text{out}(w_i)} \sum_{v_k \in \text{out}(v_j)} \frac{\partial \ell}{\partial v_k} \frac{\partial v_k}{\partial v_j} \frac{\partial v_j}{\partial w_i}$$

$$\vdots$$

$$= \sum_{\text{paths } v^{(1)}, \dots, v^{(k)} \text{ from } w_i \text{ to } \ell} \frac{\partial \ell}{\partial v^{(k)}} \frac{\partial v^{(k)}}{\partial v^{(k-1)}} \cdots \frac{\partial v^{(2)}}{\partial v^{(1)}} \frac{\partial v^{(1)}}{\partial w_i}$$

However, computing the derivative by evaluating this expression is quite inefficient, as many terms appear in more than one path from w_i to ℓ, so we are doing more work than necessary.

Backpropagation

The backpropagation algorithm combines the chain rule with the principles of **dynamic programming**: dividing a large problem into simpler subproblems, solving these and storing their solutions, and combining the stored solutions to solve larger subproblems or the original problem. In this context, the large problem is computing $\nabla \ell(W)$, and the subproblems are computing the individual terms $\frac{\partial \ell}{\partial w_i}$. The key observation from the first chain rule expression above is that we can reuse work by computing $\frac{\partial \ell}{\partial v_i}$ in a "back to front" order. That is, before computing $\frac{\partial \ell}{\partial v_i}$, we should compute $\frac{\partial \ell}{\partial v_j}$ for each $v_j \in \text{out}(v_i)$. Because our computational graph is a DAG, such a **topological ordering** can always and efficiently[9] be computed via a **topological sort**.[10] Then the subproblem of computing $\frac{\partial \ell}{\partial v_i}$ can be easily accomplished by combining the stored values $\frac{\partial \ell}{\partial v_j}$ with the terms $\frac{\partial v_j}{\partial v_i}$, which can typically be computed analytically based on our knowledge of what mathematical computations each vertex performs. Let us consider a few examples of computations that the vertices of neural network computation graphs perform, to get a concrete sense of what these $\frac{\partial v_j}{\partial v_i}$ terms look like.

Derivatives of common neural network elements

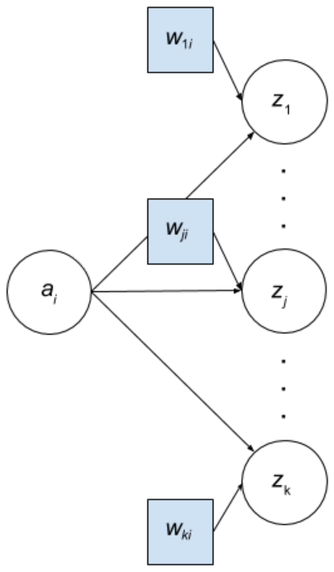
Fully connected layers

In a standard fully-connected layer, each vertex calculates z_j as a linear combination of the activations a_i of the previous layer, with weights w_{ji}:

$$z_j = \sum_i w_{ji} a_i$$

We have omitted layer indexing to keep the notation simple, but keep in mind that this a_i is the result of some computation performed at the previous layer[11], and these z_j are likely used as inputs to vertices at later layers. This part of the computational graph looks like

[9] In time linear in the size of the graph: $O(|V| + |E|)$.
[10] See CS 170!
[11] unless it is the input layer



In the image above, $\text{out}(w_{ji}) = \{z_j\}$, and it is straightforward to see that

$$\frac{\partial z_j}{\partial w_{ji}} = a_i$$

so

$$\frac{\partial \ell}{\partial w_{ji}} = \frac{\partial \ell}{\partial z_j} a_i$$

Observe that we must use the activations a_i that were previously computed in the forward pass.

We must also compute the derivatives z_j with respect to a_i so that we can pass these backward to earlier layers. In the image above, $\text{out}(a_i) = \{z_1, \dots, z_k\}$, and it is straightforward to see that

$$\frac{\partial z_j}{\partial a_i} = w_{ji}$$

so

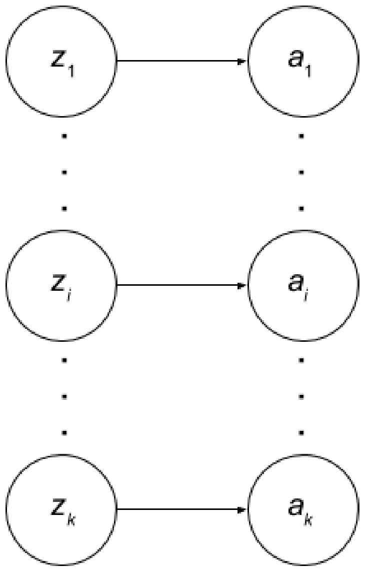
$$\frac{\partial \ell}{\partial a_i} = \sum_{j=1}^{k} \frac{\partial \ell}{\partial z_j} w_{ji}$$

Element-wise nonlinearities

After taking linear combinations, it is typical to insert a nonlinearity. (Recall from the previous note that nonlinearities are at the heart of neural networks' expressive power.) In most cases, this nonlinearity is applied elementwise. Again omitting layer indexing, we might write such a computation as

$$a_i = \sigma(z_i)$$

where z_i is the value from the previous layer, and σ is the activation function. This part of the computational graph looks like



In the image above, $\text{out}(z_i) = \{a_i\}$, and it is straightforward to see that

$$\frac{\partial a_i}{\partial z_i} = \sigma'(z_i)$$

so

$$\frac{\partial \ell}{\partial z_i} = \frac{\partial \ell}{\partial a_i}\sigma'(z_i)$$

Chapter 5

Classification

5.1 Generative vs. Discriminative Classification

The task of **classification** differs from **regression** in that we are now interested in assigning a d-dimensional data point one of a *discrete* number of **classes**, instead of assigning it a *continuous* value. Thus, the task is simpler in that there are fewer choices of labels per data point but more complicated in that we now need to somehow factor in information about each class to obtain the classifier that we want.

Given a training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ of n points, where each data point $\mathbf{x}_i \in \mathbb{R}^d$ is paired with a known discrete class label $y_i \in \{1, 2, ..., K\}$, our goal is to train a classifier which, when fed any arbitrary d-dimensional data point, classifies that data point as one of the K discrete classes.

There are two main types of classification models: generative models and discriminative models. **Generative models** have strong roots in probabilistic modeling. The idea is that we form a joint probability distribution $p(\mathbf{X}, Y)$ over the input $\mathbf{X}$ (which we treat as a random vector) and label Y (which we treat as a random variable), and we classify an arbitrary datapoint $\mathbf{x}$ with the class label that maximizes the joint probability:

$$\hat{y} = \arg\max_k \; p(\mathbf{x}, Y = k)$$

Generative models typically form the joint distribution by explicitly forming the following:

- A prior probability distribution over all classes:

$$P(k) = P(\text{class} = k)$$

- A conditional probability distribution for each class $k \in \{1, 2, ..., K\}$:

$$p_k(\mathbf{X}) = p(\mathbf{X}|\text{class } k)$$

Using the prior and the conditional distributions in conjunction, we have (from Bayes' rule) that maximizing the joint probability over the class labels is equivalent to maximizing the posterior probability of the class label:

$$\hat{y} = \arg\max_k \; p(\mathbf{x}, Y = k) = \arg\max_k \; P(k)\; p_k(\mathbf{x}) = \arg\max_k \; P(Y = k|\mathbf{x})$$

Maximizing the posterior will induce regions in the feature space in which one class has the highest posterior probability, and **decision boundaries** in between classes where the posterior probability of two classes are equal.

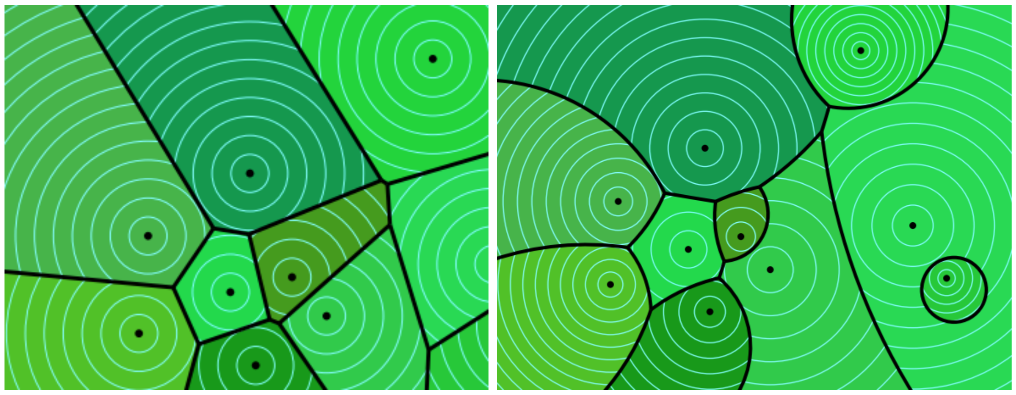


Figure 5.1: A collection (in dark black) of linear (left) vs quadratic (right) level set boundaries in a 2D feature space

Generative classifiers are flexible, quick to train, and can generate new samples (in order to augment the training dataset). However, they are also inefficient, because they require estimation of a large number of parameters (ie. the covariance matrices of the conditional distributions, which have $\frac{d(d+1)}{2}$ parameters). Typically, the decision boundary only requires $O(d)$ parameters, but generative models typically estimate $O(d^2)$ parameters in order to to determine the class-conditional probability distributions. As d increases, generative models tend to loose their effectiveness, as the number of parameters starts to dominate in comparison to the number of datapoints, and as a result the variance of the model increases.

This leads us to the concept of **discriminative models**, where we bypass learning a generative model altogether and directly learn a decision boundary. Discriminative models are parameterized by weights that either (1) form a posterior distribution $P(Y|\mathbf{X})$ without considering the prior or conditional distributions, or (2) directly form a hard decision boundary without considering any probabilities in the first place. In the former case, discriminative models choose the class that maximizes the posterior probability distribution:

$$\hat{y} = \arg\max_k P(Y = k|\mathbf{x})$$

Generative models also choose the class that maximizes the posterior probability distribution. The only difference is in the way generative and discriminative models form the posterior.

Bayes' Decision Rule

While both generative and discriminative models by default maximize the posterior probability over classes, this strategy may not necessarily be desirable at all times. Rather than maximizing the posterior probability, we would really like to minimize the **risk** of our model. Recall that the risk for a given classifier h is defined as the expected loss over $\mathbf{X}$ and Y:

$$R(h) = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\ell(h(\mathbf{x}), y)]$$

where $\ell(h(\mathbf{x}), y)$ measures the loss between the predicted label $h(\mathbf{x})$ and the true label y. In the context of regression, the loss function was the squared error $\ell(h(\mathbf{x}), y) = (h(\mathbf{x}) - y)^2$. In the context of classification, the loss function can take many forms, but the simplest is the standard step function

$$\ell(h(\mathbf{x}), y) = \begin{cases} 0 & \text{if } h(\mathbf{x}) = y \\ 1 & \text{if } h(\mathbf{x}) \neq y \end{cases}$$

Our goal is to find a classifier that minimizes the risk, given the loss function. We can equivalently express the risk as

$$R(h) = \int \left(\sum_{k=1}^{K} L(r(\mathbf{x}), k) P(Y = k|\mathbf{x}) \right) p(\mathbf{x}) d\mathbf{x}$$

The **Bayes' classifier** h^* will minimize the risk. Given an arbitrary $\mathbf{x}$, the Bayes' classifier will pick

$$h^*(\mathbf{x}) = \arg\min_j \sum_{k=1}^{K} L(j, k) P(Y = k|\mathbf{x})$$

Effectively, the Bayes' classifier will pick the class that minimizes the expected loss for the given $\mathbf{x}$. In the special case where the loss function is the standard step function (as described above),

$$h^*(\mathbf{x}) = \arg\min_j \sum_{k \neq j} P(Y = k|\mathbf{x}) = \arg\min_j 1 - P(Y = j|\mathbf{x}) = \arg\max_j P(Y = j|\mathbf{x})$$

This is equivalent to selecting the class that maximizes the posterior distribution!

Depending on which loss function we are using, the optimal classifier may or may not maximize the posterior probability. For example, consider the case of cancer diagnosis, where a patient's diagnosis for cancer can come up as positive or negative. There are four possible cases:

1. Classify the patient cancer +, and in reality the patient is cancer + (Correct Classification)
2. Classify the patient cancer −, and in reality the patient is cancer − (Correct Classification)
3. Classify the patient cancer +, but in reality the patient is cancer − (False positive)
4. Classify the patient cancer −, but in reality the patient is cancer + (False negative)

Classifying the patient's condition correctly is ideal, so we can reasonably set the loss for those cases to 0. The false positive and false negative cases are bad, and there should be a loss for these cases. But should these cases have the same loss value or should we weigh them differently? A false negative diagnosis would be significantly worse than a false positive, because a false negative diagnosis would go undiagnosed and would probably be fatal. Therefore, the associated loss for the false negative case should be higher than the associated loss for the false positive case. In this case, the goal is no longer to maximize the posterior probability, because otherwise we would be treating the false negative and false positive cases the same.

5.2 Least Squares Support Vector Machine

As a first example of a simple, non-probabilistic discriminative model, we discuss the **Least Squares Support Vector Machine (LS-SVM)**. Consider the binary classification problem where the classes are represented by −1 and +1. One way to classify a data point $\mathbf{x}$ is to estimate parameters $\mathbf{w}$, compute $\mathbf{w}^\top\mathbf{x}$, and classify $\mathbf{x}$ to be $\text{sign}(\mathbf{w}^\top\mathbf{x})$. Geometrically, the decision boundary this produces is a hyperplane, $\mathbf{w}^\top\mathbf{x} = 0$.

We need to figure out how to optimize the parameter $\mathbf{w}$. One simple procedure we can try is to fit a least squares objective:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \text{sign}(\mathbf{w}^\top \mathbf{x}_i)\|^2 + \lambda\|\mathbf{w}\|^2$$

Where $\mathbf{x}_i, \mathbf{w} \in \mathbb{R}^{d+1}$. Note that we have not forgotten about the bias term! Even though we are dealing with d dimensional data, $\mathbf{x}_i$ and $\mathbf{w}$ are $d+1$ dimensional: we add an extra "feature" of 1 to $\mathbf{x}$, and a corresponding bias term of k in $\mathbf{w}$. Note that in practice, we do not want to penalize the bias term in the regularization term, because the we should be able to work with any affine transformation of the data and still end up with the same decision boundary. Therefore, rather than taking the norm of $\mathbf{w}$, we often take the norm of $\mathbf{w}'$, which is every term of $\mathbf{w}$ excluding the corresponding bias term. For simplicity of notation however, let's just take the norm of $\mathbf{w}$.

Without the regularization term, this would be equivalent to minimizing the number of misclassified training points. Unfortunately, the "sign" term makes this optimization problem non-convex, and in fact this optimization problem is NP-hard (computationally intractable). Instead we can solve a relaxed version of this problem:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \mathbf{w}^\top \mathbf{x}_i\|^2 + \lambda\|\mathbf{w}\|^2$$

This method is called the binary **least squares support vector machine** (LS-SVM). Note that in this relaxed version, we care about the magnitude of $\mathbf{w}^\top \mathbf{x}_i$ and not just the sign.

One drawback of LS-SVM is that the hyperplane decision boundary it computes does not necessarily make sense for the sake of classification. For example, consider the following set of data points, color-coded according to the class:

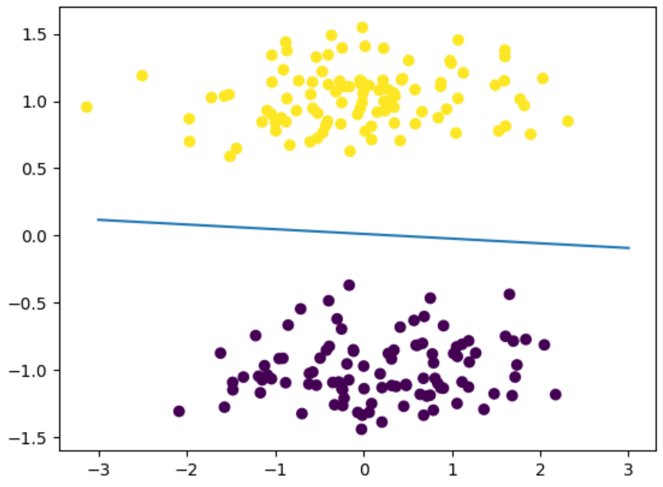


Figure 5.2: Reasonable fit LS-SVM

LS-SVM will classify every data point correctly, since all the $+1$ points lie on one side of the decision boundary and all the -1 points lie on the other side. Now if we add another cluster of points as follows:

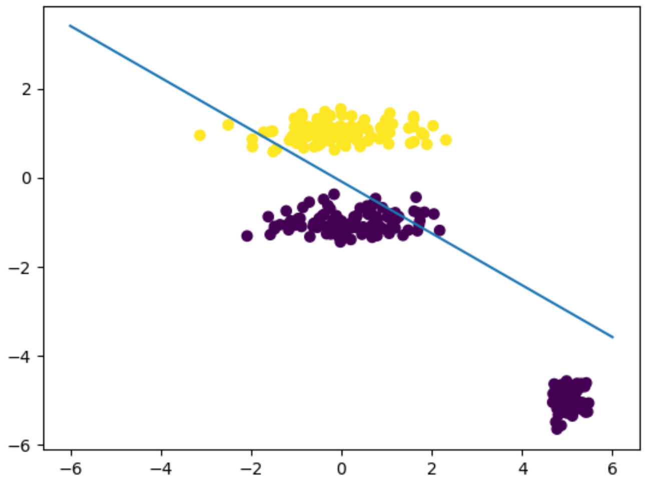


Figure 5.3: Poor fit LS-SVM

The original LS-SVM fit would still have classified every point correctly, but now the LS-SVM gets confused and decides that the points at the bottom right are contributing too much to the loss (perhaps for these points, $\mathbf{w}^\top \mathbf{x}_i = -5$ for the original choice of $\mathbf{w}$ so even though they are on the correct side of the original separating hyperplane, they incur a high squared loss and thus the hyperplane is shifted to accommodate). This problem will be solved when we introduce **general Support Vector Machines (SVM's)**.

Feature Extension

Working with linear classifiers in the raw feature space may be extremely limiting, so we may consider adding features that that allow us to come up with nonlinear classifiers (note that we are still working with linear classifiers in the augmented feature space). For example, adding quadratic features allows us to find a linear decision boundary in the augmented quadratic space that corresponds to a nonlinear "circle" decision boundary projected down into the raw feature space.

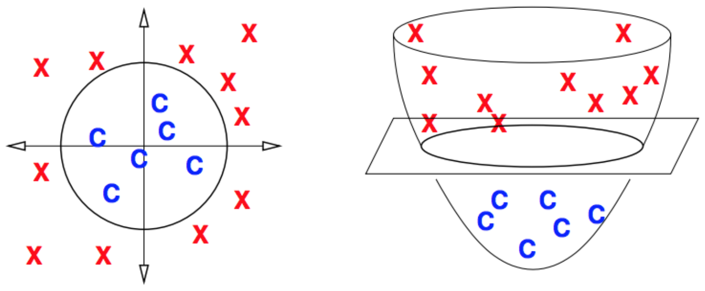


Figure 5.4: Augmenting Features, image courtesy of Prof. Shewchuk

In order implement this idea, we re-express our objective as

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \mathbf{w}^\top \phi(\mathbf{x}_i)\|^2 + \lambda \|\mathbf{w}\|^2$$

Note that ϕ is a function that takes as input the data in raw feature space, and outputs the data in augmented feature space.

Neural Network Extension

Instead of using the linear function $\mathbf{w}^\top \mathbf{x}$ or augmenting features to the data, we can also directly use a non-linear function of our choice in the original feature space, such as a neural network. One can imagine a whole family of discriminative binary classifiers that minimize

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - g_{\mathbf{w}}(\mathbf{x}_i)\|^2 + \lambda \|\mathbf{w}\|^2$$

where $g_{\mathbf{w}}(\mathbf{x}_i)$ can be any function that is easy to optimize. Then we can classify using the rule

$$\hat{y}_i = \begin{cases} 1 & g_{\mathbf{w}}(\mathbf{x}_i) > \theta \\ -1 & g_{\mathbf{w}}(\mathbf{x}_i) \leq \theta \end{cases}$$

Where θ is some threshold. In LS-SVM, $g_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{x}^\top \mathbf{w}_i$ and $\theta = 0$. Using a neural network with non-linearities as $g_{\mathbf{w}}$ can produce complex, non-linear decision boundaries.

Multiclass Extension

We can also adapt this approach to the case where we have multiple classes. Suppose there are K classes, labeled $1, 2, ..., K$. One possible way to extend the approach from binary classification is to compute $g_{\mathbf{w}}(\mathbf{x}_i)$ and round it to the nearest number from 1 to K. However, this approach gives an "ordering" to the classes, even if the classes themselves have no natural ordering. This is clearly a problem. For example, in fruit classification, suppose 1 is used to represent "peach," 2 is used to represent "banana," and 3 is used to represent "apple." In our numerical representation, it would appear that peaches are less than bananas, which are less than apples. As a result, if we have an image that looks like some cross between an apple and a peach, we may simply end up classifying it as a banana.

The typical way to get around this issue is as follows: if the i'th observation has class k, instead of using the representation $y_i = k$, we can use the representation $\mathbf{y}_i = \mathbf{e}_k$, the k'th canonical basis vector. Now there is no relative ordering in the representations of the classes. This method is called **one-hot vector encoding**.

When we have multiple classes, each $\mathbf{y}_i$ is a K-dimensional one-hot vector, so for LS-SVM, we instead have a $K \times (d+1)$ weight matrix to optimize over:

$$\arg\min_{\mathbf{W}} \sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{W}\mathbf{x}_i\|^2 + \lambda \|\mathbf{w}\|^2$$

To classify an arbitrary input $\mathbf{x}$, we compute $\mathbf{W}\mathbf{x}$ and see which component k is the largest:

$$\hat{y} = \max_k \mathbf{w}_k{}^\top \mathbf{x}$$

5.3 Logistic Regression

Logistic regression is a discriminative classification technique that has a direct probabilistic interpretation. We will first present the binary class case, and then we can easily extend the logic to the multiclass case.

Binary Logistic Regression

Suppose that we have the binary classification problem where classes are represented by 0 and 1. Note that we instead of using $-1/+1$ labels (as in LS-SVM), in binary logistic regression we use 0/1 labels. Logistic regression makes more sense this way because it directly outputs a probability, which belongs in the range of values between 0 and 1.

In binary logistic regression, we would like our model to output the probability that a data point is in class 0 or 1. We can start with the raw linear "score" $\mathbf{w}^\top\mathbf{x}$ and convert it to a probability between 0 and 1 by applying a sigmoid transformation $s(\mathbf{w}^\top\mathbf{x})$, where $s(z) = \frac{1}{1+e^{-z}}$. To classify an arbitrary point $\mathbf{x}$, we use the sigmoid function to output a probability distribution $P(\hat{Y})$ over the classes 0 and 1:

$$P(\hat{Y} = 1 \mid \mathbf{x}, \mathbf{w}) = s(\mathbf{w}^\top\mathbf{x}), \quad P(\hat{Y} = 0 \mid \mathbf{x}, \mathbf{w}) = 1 - s(\mathbf{w}^\top\mathbf{x})$$

we classify $\mathbf{x}$ as the class with the maximum probability:

$$\hat{y} = \max_k P(\hat{Y} = k \mid \mathbf{x}, \mathbf{w}) = \begin{cases} 1 & \text{if } s(\mathbf{w}^\top\mathbf{x}) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

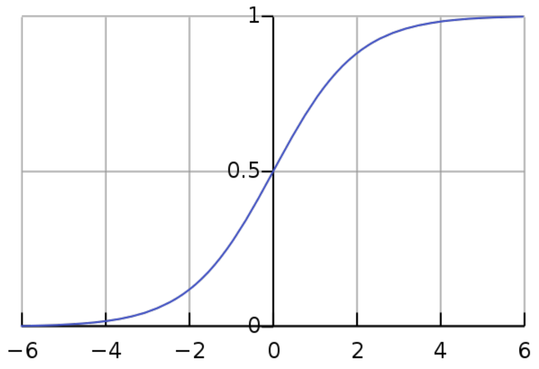


Figure 5.5: Logistic function. For our purposes, the horizontal axis is the output of the linear function $\mathbf{w}^\top\mathbf{x}_i$ and the vertical axis is the output of the logistic function, which can be interpreted as a probability between 0 and 1.

Equivalently, we classify $\mathbf{x}$ as

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w}^\top\mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Loss Function

Suppose we are given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$. In order to train our model, we need a loss function to optimize. One possibility is least squares:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - s(\mathbf{w}^\top \mathbf{x}_i)\|^2 + \lambda \|\mathbf{w}\|^2$$

However, this may not be the best choice. Ordinary least squares regression has theoretical justifications such as being the maximum likelihood objective under Gaussian noise. Least squares for this classification problem does not have a similar justification.

Instead, the loss function we use for logistic regression is called the log-loss, or **cross entropy**:

$$L(\mathbf{w}) = \sum_{i=1}^{n} y_i \ln\left(\frac{1}{s(\mathbf{w}^\top \mathbf{x}_i)}\right) + (1 - y_i) \ln\left(\frac{1}{1 - s(\mathbf{w}^\top \mathbf{x}_i)}\right)$$

If we define $p_i = s(\mathbf{w}^\top \mathbf{x}_i)$, then using the properties of logs we can express this as

$$L(\mathbf{w}) = -\sum_{i=1}^{n} y_i \ln p_i + (1 - y_i) \ln(1 - p_i)$$

For each $\mathbf{x}_i$, p_i represents our predicted probability that its corresponding class is 1. Because $y_i \in \{0, 1\}$, the loss corresponding to the i'th data point is

$$L_i(\mathbf{w}) = \begin{cases} -\ln p_i & \text{when } y_i = 1 \\ -\ln(1 - p_i) & \text{when } y_i = 0 \end{cases}$$

Intuitively, if $p_i = y_i$, then we incur 0 loss. However, this is never actually the case. The logistic function can never actually output a value of exactly 0 or 1, and we will therefore always incur some loss. If the actual label is $y_i = 1$, then as we lower p_i towards 0, the loss for this data point approaches infinity.

The loss function can be derived from a maximum likelihood perspective or an information-theoretic perspective. First let's present the maximum likelihood perspective. We view each observations y_i as an independent sample from a Bernoulli distribution $\hat{Y}_i \sim \text{Bern}(p_i)$ (technically we mean $\hat{Y}_i \mid \mathbf{x}_i, \mathbf{w}$, but we remove the conditioning terms for notational brevity), where p_i is a function of $\mathbf{x}_i$. Thus our observation y_i, which we can view as a "sample," has probability

$$P(\hat{Y}_i = y_i) = \begin{cases} p_i & \text{if } y_i = 1 \\ 1 - p_i & \text{if } y_i = 0 \end{cases}$$

One convenient way to write the likelihood of a single data point is

$$P(\hat{Y}_i = y_i) = p_i^{y_i} (1 - p_i)^{(1 - y_i)}$$

which holds no matter what y_i is.

We need a model for the dependency of p_i on $\mathbf{x}_i$. We have to enforce that p_i is a transformation of $\mathbf{x}_i$ that results in a number from 0 to 1 (ie. a valid probability). Hence p_i cannot be, say, linear in $\mathbf{x}_i$. One way to do achieve the 0-1 normalization is by using the sigmoid function

$$p_i = s(\mathbf{w}^\top \mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}}$$

Now we can estimate the parameters $\mathbf{w}$ via maximum likelihood. We have the problem

$$\begin{aligned}
\hat{\mathbf{w}}_{\text{LR}} &= \arg\max_{\mathbf{w}} P(\hat{Y}_1 = y_1, \ldots, \hat{Y}_n = y_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{n} P(\hat{Y}_i = y_i \mid \mathbf{x}_i, \mathbf{w}) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{(1-y_i)} \\
&= \arg\max_{\mathbf{w}} \ln\Big[\prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{(1-y_i)}\Big] \\
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i) \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i)
\end{aligned}$$

which exactly matches the cross-entropy formulation from earlier. The logistic regression loss function can also be justified from an information-theoretic perspective. To motivate this approach, we introduce **Kullback-Leibler (KL) divergence** (also called **relative entropy**), which measures the amount that one distribution diverges from another. Given *any* two discrete random variables P and Q, the KL divergence from Q to P is defined as

$$D_{\text{KL}}(P \parallel Q) = \sum_x P(x) \ln \frac{P(x)}{Q(x)}$$

Note that D_{KL} is not a true distance metric, because it is not symmetric, ie. $D_{\text{KL}}(P \parallel Q) \neq D_{\text{KL}}(Q \parallel P)$ in general. It also does not satisfy the triangle inequality. However, it is always positive, ie. $D_{\text{KL}}(P \parallel Q) \geq 0$, with equality iff $P = Q$.

In the context of classification, if the class label y_i is interpreted as the probability of being class 1, then logistic regression provides an estimate p_i of the probability that the data is in class 1. The true class label can be viewed as a sampled value from the "true" distribution $Y_i \sim \text{Bern}(y_i)$. $P(Y_i)$ is not a particularly interesting distribution because all values sampled from it will yield y_i: $P(Y_i = y_i) = 1$. Logistic regression yields a distribution $\hat{Y}_i \sim \text{Bern}(p_i)$, which is the posterior probability that estimates the true distribution $P(Y_i)$.

The KL divergence from $P(\hat{Y}_i)$ to $P(Y_i)$ provides a measure of how closely logistic regression can match the true label. We would like to minimize this KL divergence, and ideally we would try to choose our parameters so that $D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i)) = 0$. Again, this is impossible for two reasons. First, if we want $D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i)) = 0$, then we would need $p_i = y_i$, which is impossible because p_i is the output of a logistic function that can never actually reach 0 or 1. Second, even if we tried tuning the parameters so that $D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i)) = 0$, that's only optimizing one of the data points – we need to tune the parameters so that we can collectively minimize the totality of all of the KL divergences contributed by all data points.

Therefore, our goal is to tune the parameters $\mathbf{w}$ (which indirectly affect the p_i values and therefore the estimated distribution $P(\hat{Y}_i)$), in order to minimize the total sum of KL divergences contributed by all data points:

$$\hat{\mathbf{w}}_{\text{LR}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i))$$

$$
\begin{aligned}
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} y_i \ln \frac{y_i}{p_i} + (1-y_i)\ln\frac{(1-y_i)}{(1-p_i)} \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} y_i(\ln y_i - \ln p_i) + (1-y_i)(\ln(1-y_i) - \ln(1-p_i)) \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} (-y_i \ln p_i - (1-y_i)\ln(1-p_i)) + (y_i \ln y_i + (1-y_i)\ln(1-y_i)) \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i) \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} H(P(Y_i), P(\hat{Y}_i))
\end{aligned}
$$

Note that the $y_i \ln y_i + (1-y_i)\ln(1-y_i)$ component of the KL divergence is a constant, independent of our changes to p_i. Therefore, we are effectively minimizing the sum of the **cross entropies** $H(P(Y_i), P(\hat{Y}_i))$. We conclude our discussion of KL Divergence by noting the relation between KL divergence and cross entropy:

$$D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i)) = H(P(Y_i), P(\hat{Y}_i)) - H(P(Y_i))$$

where:

1. $D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i))$ is the KL divergence from $P(\hat{Y}_i)$ to $P(Y_i)$:

$$D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i)) = y_i \ln \frac{y_i}{p_i} + (1-y_i)\ln\frac{(1-y_i)}{(1-p_i)}$$

2. $H(P(Y_i), P(\hat{Y}_i))$ is the cross entropy between Y_i and $\hat{Y}_i$:

$$H(P(Y_i), P(\hat{Y}_i)) = -y_i \ln p_i - (1-y_i)\ln(1-p_i)$$

3. $H(Y_i)$ is the entropy of Y_i:

$$H(Y_i) = -y_i \ln y_i - (1-y_i)\ln(1-y_i)$$

Since the parameters $\mathbf{W}$ do not affect the entropy, we can optimize the cross entropy instead of the KL divergence.

Multiclass Logistic Regression

Let's generalize logistic regression to the case where there are K classes. Similarly to our discussion of the multi-class LS-SVM, it is important to note that there is no inherent ordering to the classes, and predicting a class in the continuous range from 1 to K would be a poor choice. To see why, recall our fruit classification example. Suppose 1 is used to represent "peach," 2 is used to represent "banana," and 3 is used to represent "apple." In our numerical representation, it would appear that peaches are less than bananas, which are less than apples. As a result, if we have an image that looks like some cross between an apple and a peach, we may simply end up classifying it as a banana.

The solution is to use a **one-hot vector encoding** to represent all of our labels. If the i'th observation has class k, instead of using the representation $y_i = k$, we can use the representation $\mathbf{y}_i = \mathbf{e}_k$, the k'th canonical basis vector. For example, in our fruit example, if the i'th image is classified as "banana", its label representation would be

$$\mathbf{y}_i = [0\ 1\ 0]$$

(Be careful to distinguish between the class label $y_i \in \mathbb{R}$ and its one-hot encoding $\mathbf{y}_i \in \mathbb{R}^K$). Now there is no relative ordering in the representations of the classes. We must modify our parameter representation accordingly to the one-hot vector encoding. Now, there are a set of $d+1$ parameters associated with every class, which amounts to a matrix $\mathbf{W} \in \mathbb{R}^{K \times (d+1)}$. For each input $\mathbf{x}_i \in \mathbb{R}^{d+1}$, each class k is given a "score"

$$z_k = {\mathbf{w}_k}^\top \mathbf{x}_i$$

Where $\mathbf{w}_k \in \mathbb{R}^{d+1}$ is the k'th row of the $\mathbf{W}$ matrix. In total there are K raw linear scores for an arbitrary input $\mathbf{x}$:

$$\begin{bmatrix} {\mathbf{w}_1}^\top \mathbf{x} & {\mathbf{w}_2}^\top \mathbf{x} & \dots & {\mathbf{w}_K}^\top \mathbf{x} \end{bmatrix}$$

The higher the score for a class, the more likely logistic regression will pick that class. Now that we have a score system, we must transform all of these scores into a posterior probability distribution $P(\hat{Y})$. For binary logistic regression, we used the logistic function, which takes the value $\mathbf{w}^\top \mathbf{x}$ and squashes it to a value between 0 and 1. The generalization to the the logistic function for the multi-class case is the **softmax function**. The softmax function takes as input all K scores (formally known as **logits**) and an index j, and outputs the probability that the corresponding softmax distribution takes value j:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

The logits induce a **softmax distribution**, which we can verify is indeed a probability distribution:

1. The entries are between 0 and 1.
2. The entries add up to 1.

Using the softmax function, we generate probabilities for each class:

$$\hat{\mathbf{y}} = \sigma(\mathbf{W}\mathbf{x}) = \begin{bmatrix} \sigma(\mathbf{W}\mathbf{x})_1 & \sigma(\mathbf{W}\mathbf{x})_2 & \dots & \sigma(\mathbf{W}\mathbf{x})_K \end{bmatrix}$$

and our prediction is the class with the maximum probability:

$$\hat{y} = \max_k \sigma(\mathbf{W}\mathbf{x})_k$$

On inspection, this softmax distribution is reasonable, because the higher the score of a class, the higher its probability. In fact, we can verify that the logistic function used in the binary case is a special case of the softmax function used in the multiclass case. Assuming that the corresponding parameters for class 0 and 1 are $\mathbf{w}_0$ and $\mathbf{w}_1$, we have that:

$$P(\hat{Y}_i = 1 \mid \mathbf{x}_i, \mathbf{W}) = \sigma(\mathbf{W}\mathbf{x}_i)_1 = \frac{e^{{\mathbf{w}_1}^\top \mathbf{x}_i}}{e^{{\mathbf{w}_0}^\top \mathbf{x}_i} + e^{{\mathbf{w}_1}^\top \mathbf{x}_i}} = s((\mathbf{w}_1 - \mathbf{w}_0)^\top \mathbf{x}_i)$$
$$P(\hat{Y}_i = 0 \mid \mathbf{x}_i, \mathbf{W}) = \sigma(\mathbf{W}\mathbf{x}_i)_0 = \frac{e^{{\mathbf{w}_0}^\top \mathbf{x}_i}}{e^{{\mathbf{w}_0}^\top \mathbf{x}_i} + e^{{\mathbf{w}_1}^\top \mathbf{x}_i}} = 1 - s((\mathbf{w}_1 - \mathbf{w}_0)^\top \mathbf{x}_i)$$

In the 2-class case, because we are only interested in the difference between $\mathbf{w}_1$ and $\mathbf{w}_0$, we just use a change of variables $\mathbf{w} = \mathbf{w}_1 - \mathbf{w}_0$. We don't need to know $\mathbf{w}_1$ and $\mathbf{w}_0$ individually, because once we know $P(\hat{Y}_i = 1)$, we know by default that $P(\hat{Y}_i = 0) = 1 - P(\hat{Y}_i = 1)$.

Loss Function

Let's derive the loss function for multiclass logistic regression, first using the information-theoretic perspective. The "true" or more formally the **target distribution** in this case is $P(Y_i = j) = \delta_{j,y_i}$. In other words, the entire distribution is concentrated on the label for the training example. The estimated distribution $P(\hat{Y}_i)$ comes from multiclass logistic regression, and in this case is the softmax distribution:

$$P(\hat{Y}_i = j) = \frac{e^{\mathbf{w}_j{}^\top \mathbf{x}_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^\top \mathbf{x}_i}}$$

Now let's proceed to deriving the loss function. The objective, as always, is to minimize the sum of the KL divergences contributed by all of the training examples.

$$\begin{aligned}
\hat{\mathbf{W}}_{\text{MCLR}} &= \arg\min_{\mathbf{W}} \sum_{i=1}^{n} D_{\text{KL}}(P(Y_i) \parallel P(\hat{Y}_i)) \\
&= \arg\min_{\mathbf{W}} \sum_{i=1}^{n}\sum_{j=1}^{K} P(Y_i = j) \ln\left(\frac{P(Y_i = j)}{P(\hat{Y}_i = j)}\right) \\
&= \arg\min_{\mathbf{W}} \sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \ln\left(\frac{\delta_{j,y_i}}{\sigma(\mathbf{W}\mathbf{x}_i)_j}\right) \\
&= \arg\min_{\mathbf{W}} \sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \cdot \ln \delta_{j,y_i} - \delta_{j,y_i} \cdot \ln\left(\sigma(\mathbf{W}\mathbf{x}_i)_j\right) \\
&= \arg\min_{\mathbf{W}} -\sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \cdot \ln\left(\sigma(\mathbf{W}\mathbf{x}_i)_j\right) \\
&= \arg\min_{\mathbf{W}} -\sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \cdot \ln\left(\frac{e^{\mathbf{w}_j{}^\top \mathbf{x}_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^\top \mathbf{x}_i}}\right) \\
&= \arg\min_{\mathbf{W}} \sum_{i=1}^{n} H(P(Y_i), P(\hat{Y}_i))
\end{aligned}$$

Just like binary logistic regression, we can justify the loss function with MLE as well:

$$\begin{aligned}
\hat{\mathbf{W}}_{\text{MCLR}} &= \arg\max_{\mathbf{w}} P(\hat{Y}_1 = y_1, \ldots, \hat{Y}_n = y_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{w}) \\
&= \arg\max_{\mathbf{W}} \prod_{i=1}^{n} P(\hat{Y}_i = y_i \mid \mathbf{x}_i, \mathbf{W}) \\
&= \arg\max_{\mathbf{W}} \prod_{i=1}^{n}\prod_{j=1}^{K} P(\hat{Y}_i = j \mid \mathbf{x}_i, \mathbf{W})^{\delta_{j,y_i}} \\
&= \arg\max_{\mathbf{W}} \sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \ln P(\hat{Y}_i = j \mid \mathbf{x}_i, \mathbf{W}) \\
&= \arg\max_{\mathbf{W}} \sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \ln\left(\frac{e^{\mathbf{w}_j{}^\top \mathbf{x}_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^\top \mathbf{x}_i}}\right)
\end{aligned}$$

$$= \arg\min_{\mathbf{W}} -\sum_{i=1}^{n}\sum_{j=1}^{K}\delta_{j,y_i}\ln\left(\frac{e^{\mathbf{w}_j{}^\top\mathbf{x}_i}}{\sum_{k=1}^{K}e^{\mathbf{w}_k{}^\top\mathbf{x}_i}}\right)$$

We conclude that the loss function for multiclass logistic regression is

$$L(\mathbf{W}) = -\sum_{i=1}^{n}\sum_{j=1}^{K}\delta_{j,y_i}\cdot\ln P(\hat{Y}_i = j \mid \mathbf{x}_i, \mathbf{W})$$

Training

The logistic regression loss function has no known analytic closed-form solution. Therefore, in order to minimize it, we can use gradient descent, either in batch form or stochastic form. Let's examine the case for batch gradient descent.

Binary Case

Recall the loss function

$$L(\mathbf{w}) = -\sum_{i=1}^{n} y_i\ln p_i + (1-y_i)\ln(1-p_i)$$

where

$$p_i = s(\mathbf{w}^\top\mathbf{x}_i) = \frac{1}{1+e^{-\mathbf{w}^\top\mathbf{x}_i}}$$

$$\begin{aligned}
\nabla_{\mathbf{w}}L(\mathbf{w}) &= \nabla_{\mathbf{w}}\left(-\sum_{i=1}^{n} y_i\ln p_i + (1-y_i)\ln(1-p_i)\right)\\
&= -\sum_{i=1}^{n} y_i\nabla_{\mathbf{w}}\ln p_i + (1-y_i)\nabla_{\mathbf{w}}\ln(1-p_i)\\
&= -\sum_{i=1}^{n}\frac{y_i}{p_i}\nabla_{\mathbf{w}}p_i - \frac{1-y_i}{1-p_i}\nabla_{\mathbf{w}}p_i\\
&= -\sum_{i=1}^{n}\left(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i}\right)\nabla_{\mathbf{w}}p_i
\end{aligned}$$

Note that $\nabla_{\mathbf{z}}s(\mathbf{z}) = s(\mathbf{z})(1-s(\mathbf{z}))$, and from the chain rule we have that

$$\nabla_{\mathbf{w}}p_i = \nabla_{\mathbf{w}}s(\mathbf{w}^\top\mathbf{x}_i) = s(\mathbf{w}^\top\mathbf{x}_i)(1-s(\mathbf{w}^\top\mathbf{x}_i))\mathbf{x}_i = p_i(1-p_i)\mathbf{x}_i$$

Plugging in this gradient value, we have

$$\nabla_{\mathbf{w}}L(\mathbf{w}) = -\sum_{i=1}^{n}\left(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i}\right)\nabla_{\mathbf{w}}p_i$$

$$
\begin{aligned}
&= -\sum_{i=1}^{n} \left(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i} \right) p_i(1-p_i)\mathbf{x}_i \\
&= -\sum_{i=1}^{n} \big(y_i(1-p_i) - (1-y_i)(p_i) \big) \mathbf{x}_i \\
&= -\sum_{i=1}^{n} (y_i - p_i)\, \mathbf{x}_i
\end{aligned}
$$

The gradient descent update is thus

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \epsilon \sum_{i=1}^{n} (y_i - p_i)\, \mathbf{x}_i$$

It does not matter what initial values we pick for $\mathbf{w}$, because the loss function $L(\mathbf{w})$ is convex and does not have any local minima. Let's prove this, by first finding the Hessian of the loss function. The k, lth entry of the Hessian is the partial derivative of the gradient with respect to $\mathbf{w}_k$ and $\mathbf{w}_\ell$:

$$
\begin{aligned}
H_{kl} &= \frac{\partial^2 L(\mathbf{w})}{\partial w_k \partial w_\ell} \\
&= \frac{\partial}{\partial w_k} - \sum_{i=1}^{n} (y_i - p_i)\, x_{il} \\
&= \sum_{i=1}^{n} \frac{\partial}{\partial w_k} p_i x_{il} \\
&= \sum_{i=1}^{n} p_i(1-p_i) x_{ik} x_{il}
\end{aligned}
$$

We conclude that

$$H(\mathbf{w}) = \sum_{i=1}^{n} p_i(1-p_i)\mathbf{x}_i \mathbf{x}_i^\top$$

To prove that $L(\mathbf{w})$ is convex in $\mathbf{w}$, we need to show that $\forall \mathbf{v},\ \mathbf{v}^\top H(\mathbf{w})\mathbf{v} \geq 0$:

$$\mathbf{v}^\top H(\mathbf{w})\mathbf{v} = \mathbf{v}^\top \sum_{i=1}^{n} p_i(1-p_i)\mathbf{x}_i \mathbf{x}_i^\top \mathbf{v} = \sum_{i=1}^{n} (\mathbf{v}^\top \mathbf{x}_i)^2 p_i(1-p_i) \geq 0$$

Multiclass Case

Instead of finding the gradient with respect to all of the parameters of the matrix $\mathbf{W}$, let's find them with respect to one row of $\mathbf{W}$ at a time:

$$
\begin{aligned}
\nabla_{\mathbf{w}_\ell} L(\mathbf{W}) &= \nabla_{\mathbf{w}_\ell} - \sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \cdot \ln\left(\frac{e^{\mathbf{w}_j{}^\top \mathbf{x}_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^\top \mathbf{x}_i}} \right) \\
&= -\sum_{i=1}^{n}\sum_{j=1}^{K} \delta_{j,y_i} \cdot \nabla_{\mathbf{w}_\ell} \left(\ln \frac{e^{\mathbf{w}_j{}^\top \mathbf{x}_i}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^\top \mathbf{x}_i}} \right)
\end{aligned}
$$

$$= -\sum_{i=1}^{n}\sum_{j=1}^{K}\delta_{j,y_i}\cdot\left(\nabla_{\mathbf{w}_\ell}\mathbf{w}_j{}^\top\mathbf{x}_i - \nabla_{\mathbf{w}_\ell}\ln\sum_{k=1}^{K}e^{\mathbf{w}_k{}^\top\mathbf{x}_i}\right)$$

$$= -\sum_{i=1}^{n}\sum_{j=1}^{K}\delta_{j,y_i}\cdot\left(\delta_{\ell,y_i} - \frac{e^{\mathbf{w}_\ell{}^\top\mathbf{x}_i}}{\sum_{k=1}^{K}e^{\mathbf{w}_k{}^\top\mathbf{x}_i}}\right)\mathbf{x}_i$$

$$= -\sum_{i=1}^{n}\left(\delta_{\ell,y_i} - P(\hat{Y}_i=\ell)\right)\mathbf{x}_i$$

The gradient descent update for $\mathbf{w}_\ell$ is thus

$$\mathbf{w}_\ell^{(t+1)} = \mathbf{w}_\ell^{(t+1)} + \epsilon\sum_{i=1}^{n}\left(\delta_{\ell,y_i} - P(\hat{Y}_i=\ell)\right)\mathbf{x}_i$$

Just as with binary logistic regression, it does not matter what initial values we pick for $\mathbf{W}$, because the loss function $L(\mathbf{W})$ is convex and does not have any local minima.

5.4 Gaussian Discriminant Analysis

Recall the idea of **generative models**: we classify an arbitrary datapoint $\mathbf{x}$ with the class label that maximizes the joint probability $p(\mathbf{x}, Y)$ over the label Y:

$$\hat{y} = \arg\max_k\, p(\mathbf{x}, Y=k)$$

Generative models typically form the joint distribution by explicitly forming the following:

- A prior probability distribution over all classes:

$$P(k) = P(\text{class} = k)$$

- A conditional probability distribution for each class $k \in \{1, 2, ..., K\}$:

$$p_k(\mathbf{X}) = p(\mathbf{X}|\text{class } k)$$

In total there are $K+1$ probability distributions: 1 for the prior, and K for all of the individual classes. Note that the prior probability distribution is a categorical distribution over the K discrete classes, whereas each class conditional probability distribution is a continuous distribution over $\mathbb{R}^d$ (often represented as a Gaussian). Using the prior and the conditional distributions in conjunction, we have (from Bayes' rule) that maximizing the joint probability over the class labels is equivalent to maximizing the posterior probability of the class label:

$$\hat{y} = \arg\max_k\, p(\mathbf{x}, Y=k) = \arg\max_k\, P(k)\; p_k(\mathbf{x}) = \arg\max_k\, P(Y=k|\mathbf{x})$$

Consider the example of digit classification. Suppose we are given dataset of images of handwritten digits each with known values in the range $\{0, 1, 2, \ldots, 9\}$. The task is, given an image of a handwritten digit, to classify it to the correct digit. A generative classifier for this this task would effectively form a the prior distribution and conditional probability distributions over the 10 possible digits and choose the digit that maximizes posterior probability:

$$\hat{y} = \underset{k\in\{0,1,2...,9\}}{\arg\max}\ p(\text{digit} = k|\text{image}) = \underset{k\in\{0,1,2...,9\}}{\arg\max}\ P(\text{digit} = k)\ p(\text{image}|\text{digit} = k)$$

Maximizing the posterior will induce regions in the feature space in which one class has the highest posterior probability, and **decision boundaries** in between classes where the posterior probability of two classes are equal.

Gaussian Discriminant Analysis (GDA) is a specific generative method in which the class conditional probability distributions are Gaussian: $(\mathbf{X}|Y = k) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. (Caution: the term "discriminant" in GDA is misleading; GDA is a generative method, it is not a discriminative method!)

Assume that we are given a training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ of n points. Estimating the prior distribution is the same for any other generative model. The probability of a class k is

$$P(k) = \frac{n_k}{n}$$

where n_k is the number of training points that belong to class k. We can estimate the parameters of the conditional distributions with MLE. Once we form the estimated prior conditional distributions, we use Bayes' Rule to directly solve the optimization problem

$$\begin{aligned}
\hat{y} &= \arg\max_k\ p(k \mid \mathbf{x}) \\
&= \arg\max_k\ P(k)\ p_k(\mathbf{x}) \\
&= \arg\max_k\ \ln\big(P(k)\big) + \ln\Big((\sqrt{2\pi})^d p_k(\mathbf{x})\Big) \\
&= \arg\max_k\ \ln\big(P(k)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k)^\top \hat{\boldsymbol{\Sigma}}_k^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_k|\Big) = Q_k(\mathbf{x})
\end{aligned}$$

For future reference, let's use $Q_k(\mathbf{x}) = \ln\Big(\sqrt{2\pi}\Big)^d P(k)\ p_k(\mathbf{x})$ to simplify our notation.

We classify an arbitrary test point

$$\hat{y} = \underset{k\in\{1,2,...,K\}}{\arg\max}\ Q_k(\mathbf{x})$$

GDA comes in two flavors: Quadratic Discriminant Analysis (QDA) in which the decision boundary is quadratic, and Linear Discriminant Analysis (LDA) in which the decision boundary is linear. We will now present both and compare them in detail.

QDA Classification

In **Quadratic Discriminant Analysis (QDA)**, the class conditional probability distributions are independent Gaussians — namely, the covariance $\boldsymbol{\Sigma}_k$ of class k has no dependence/relation to that of the other classes.

Due to this independence property, we can estimate the true mean and covariance $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ for each class conditional probability distribution $p_k(\mathbf{X})$ independently, with the n_k samples in our training data that are classified as class k. The MLE estimate for the parameters of $p_k(\mathbf{X})$ is:

$$\begin{aligned}
\hat{\boldsymbol{\mu}}_k &= \frac{1}{n_k} \sum_{i:y_i=k} \mathbf{x}_i \\
\hat{\boldsymbol{\Sigma}}_k &= \frac{1}{n_k} \sum_{i:y_i=k} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)^\top
\end{aligned}$$

LDA Classification

While QDA is a reasonable approach to classification, we might be interested in simplifying our model to reduce the number of parameters we have to learn. One way to do this is through **Linear Discriminant Analysis (LDA)** classification. Just as in QDA, LDA assumes that the class conditional probability distributions are normally distributed with different means $\boldsymbol{\mu}_k$, but LDA is different from QDA in that it requires all of the distributions to share the same covariance matrix $\boldsymbol{\Sigma}$. This is a simplification which, in the context of the Bias-Variance tradeoff, increases the bias of our method but may help decrease the variance.

The training and classification procedures for LDA are almost identical that of QDA. To compute the within-class means, we still want to take the empirical mean. However, the empirical covariance for all classes is now computed as

$$\hat{\boldsymbol{\Sigma}} = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{y_i})(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{y_i})^\top$$

One way to understand this formula is as a weighted average of the within-class covariances. Here, assume we have sorted our training data by class and we can index through the $\mathbf{x}_i$'s by specifying a class k and the index within that class j:

$$\begin{aligned}
\hat{\boldsymbol{\Sigma}} &= \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{y_i})(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{y_i})^\top \\
&= \frac{1}{n} \sum_{k=1}^{K} \sum_{i:y_i=k}^{n_k} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)^\top \\
&= \frac{1}{n} \sum_{k=1}^{K} n_k \; \boldsymbol{\Sigma}_k \\
&= \sum_{k=1}^{K} \frac{n_k}{n} \boldsymbol{\Sigma}_k
\end{aligned}$$

LDA and QDA Decision Boundary

Let's now derive the form of the decision boundary for QDA and LDA. As we will see, the term *quadratic* in QDA and *linear* in LDA actually signify the shape of the decision boundary. We will prove this claim using binary (2-class) examples for simplicity (class A and class B). An arbitrary point $\mathbf{x}$ is classified according to the following cases:

$$\hat{y} = \begin{cases} A & Q_A(\mathbf{x}) > Q_B(\mathbf{x}) \\ B & Q_A(\mathbf{x}) < Q_B(\mathbf{x}) \\ \text{Either } A \text{ or } B & Q_A(\mathbf{x}) = Q_B(\mathbf{x}) \end{cases}$$

The decision boundary is the set of all points in $\mathbf{x}$-space that are classified according to the third case.

Identical Conditional Distributions with Identical Priors

The simplest case is when the two classes are equally likely in prior, and their conditional probability distributions are isotropic with identical covariances. Recall that isotropic Gaussian distributions have covariances of the form of $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$, which means that their isocontours are circles. In this case, $p_A(\mathbf{X})$ and $p_B(\mathbf{X})$ have identical covariances of the form $\boldsymbol{\Sigma}_A = \boldsymbol{\Sigma}_B = \sigma^2\mathbf{I}$.

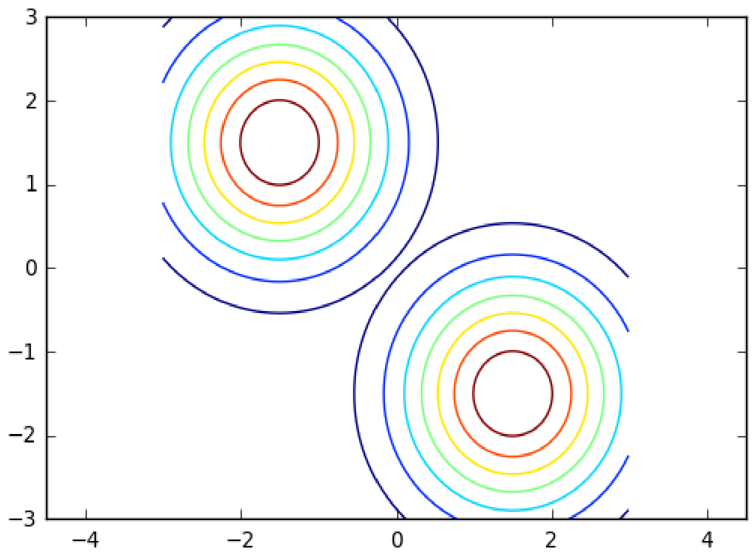


Figure 5.6: Contour plot of two isotropic, identically distributed Gaussians in $\mathbb{R}^2$. The circles are the level sets of the Gaussians.

Geometrically, we can see that the task of classifying a 2-D point into one of the two classes amounts simply to figuring out which of the means it's closer to. Using our notation of $Q_k(\mathbf{x})$ from before, this can be expressed mathematically as:

$$Q_A(\mathbf{x}) = Q_B(\mathbf{x})$$
$$\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_A)^\top \hat{\boldsymbol{\Sigma}}_A^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_A) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_A|\Big) = \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_B)^\top \hat{\boldsymbol{\Sigma}}_B^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_B) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_B|\Big)$$
$$\ln\left(\frac{1}{2}\right) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_A)^\top \sigma^{-2}\mathbf{I}(\mathbf{x} - \hat{\boldsymbol{\mu}}_A) - \frac{1}{2}\ln\Big(|\sigma^2\mathbf{I}|\Big) = \ln\left(\frac{1}{2}\right) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_B)^\top \sigma^{-2}\mathbf{I}(\mathbf{x} - \hat{\boldsymbol{\mu}}_B) - \frac{1}{2}\ln\Big(|\sigma^2\mathbf{I}|\Big)$$
$$(\mathbf{x} - \hat{\boldsymbol{\mu}}_A)^\top(\mathbf{x} - \hat{\boldsymbol{\mu}}_A) = (\mathbf{x} - \hat{\boldsymbol{\mu}}_B)^\top(\mathbf{x} - \hat{\boldsymbol{\mu}}_B)$$

The decision boundary is the set of points $\mathbf{x}$ for which $\|\mathbf{x} - \hat{\boldsymbol{\mu}}_A\|_2 = \|\mathbf{x} - \hat{\boldsymbol{\mu}}_B\|_2$, which is simply the set of points that are equidistant from $\hat{\boldsymbol{\mu}}_A$ and $\hat{\boldsymbol{\mu}}_B$. This decision boundary is linear because the set of points that are equidistant from $\hat{\boldsymbol{\mu}}_A$ and $\hat{\boldsymbol{\mu}}_B$ are simply the perpendicular bisector of the segment connecting $\hat{\boldsymbol{\mu}}_A$ and $\hat{\boldsymbol{\mu}}_B$.

The next case is when the two classes are equally likely in prior, and their conditional probability distributions are anisotropic with identical covariances.

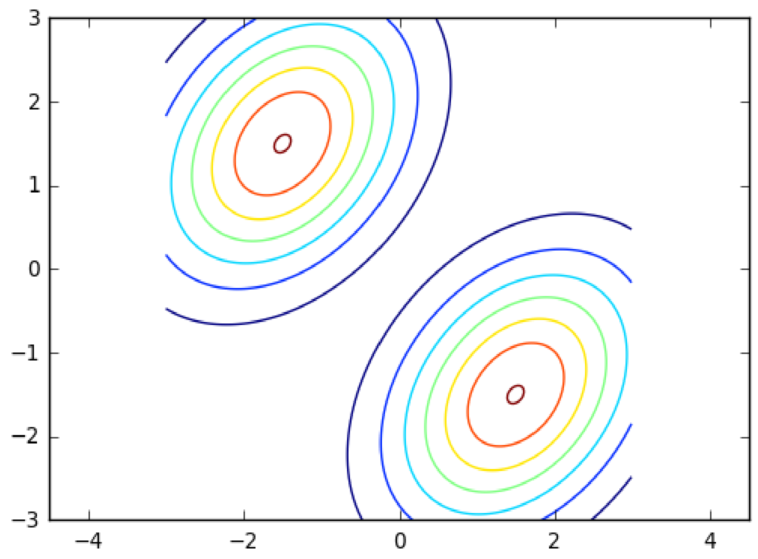


Figure 5.7: Two anisotropic, identically distributed Gaussians in $\mathbb{R}^2$. The ellipses are the level sets of the Gaussians.

The anisotropic case can be reduced to the isotropic case simply by performing a linear change of coordinates that transforms the ellipses back into circles, which induces a linear decision boundary both in the transformed and original space. Therefore, the decision boundary is still the set of points that are equidistant from $\hat{\boldsymbol{\mu}}_A$ and $\hat{\boldsymbol{\mu}}_B$.

Identical Conditional Distributions with Different Priors

Now, let's find the decision boundary when the two classes still have identical covariances but are not necessarily equally likely in prior:

$$\begin{gathered}
Q_A(\mathbf{x}) = Q_B(\mathbf{x}) \\
\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_A)^\top \hat{\boldsymbol{\Sigma}}_A^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_A) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_A|\Big) = \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_B)^\top \hat{\boldsymbol{\Sigma}}_B^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_B) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_B|\Big) \\
\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_A)^\top \hat{\boldsymbol{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_A) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}|\Big) = \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_B)^\top \hat{\boldsymbol{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_B) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}|\Big) \\
\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_A)^\top \hat{\boldsymbol{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_A) = \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_B)^\top \hat{\boldsymbol{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_B) \\
2\ln\big(P(A)\big) - \mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}\mathbf{x} + 2\mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_A - \hat{\boldsymbol{\mu}}_A{}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_A = 2\ln\big(P(B)\big) - \mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}\mathbf{x} + 2\mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_B - \hat{\boldsymbol{\mu}}_B{}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_B \\
2\ln\big(P(A)\big) + 2\mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_A - \hat{\boldsymbol{\mu}}_A{}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_A = 2\ln\big(P(B)\big) + 2\mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_B - \hat{\boldsymbol{\mu}}_B{}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_B
\end{gathered}$$

Simplifying, we have that

$$\mathbf{x}^\top \hat{\boldsymbol{\Sigma}}^{-1}(\hat{\boldsymbol{\mu}}_A - \hat{\boldsymbol{\mu}}_B) + \left(\ln\left(\frac{P(A)}{P(B)}\right) - \frac{\hat{\boldsymbol{\mu}}_A{}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_A - \hat{\boldsymbol{\mu}}_B{}^\top \hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{\mu}}_B}{2}\right) = 0$$

The decision boundary is the level set of a linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} - b$. In fact, the decision boundary is the level set of a linear function (which itself is linear) as long as the two class conditional probability distributions share the same covariance matrices. This is the reason for why LDA has a linear decision boundary.

Nonidentical Conditional Distributions with Different Priors

This is the most general case. We have that:

$$Q_A(\mathbf{x}) = Q_B(\mathbf{x})$$
$$\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_A)^\top \hat{\boldsymbol{\Sigma}}_A^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_A) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_A|\Big) = \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_B)^\top \hat{\boldsymbol{\Sigma}}_B^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_B) - \frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_B|\Big)$$

Here, unlike in LDA when $\boldsymbol{\Sigma}_A = \boldsymbol{\Sigma}_B$, we *cannot* cancel out the quadratic terms in $\mathbf{x}$ from both sides of the equation, and thus our decision boundary is now represented by the level set of an arbitrary quadratic function.

It should now make sense why QDA is short for *quadratic* discriminant analysis and LDA is short for *linear* discriminant analysis!

LDA and Logistic Regression

As it turns out, LDA and logistic regression share the same type of posterior distribution. We already showed that the posterior distribution in logistic regression is

$$P(Y = A|\mathbf{x}) = \frac{1}{1 + e^{\mathbf{w}^\top \mathbf{x} - b}} = s(\mathbf{w}^\top \mathbf{x} - b)$$

for some appropriate vector $\mathbf{w}$ and bias b. Now let's derive the posterior distribution for LDA. From Bayes' rule we have that

$$\begin{aligned} P(Y = A|\mathbf{x}) &= \frac{p(\mathbf{x}|Y = A)P(Y = A)}{p(\mathbf{x}|Y = B)P(Y = B) + p(\mathbf{x}|Y = B)P(Y = B)} \\ &= \frac{e^{Q_A(\mathbf{x})}}{e^{Q_A(\mathbf{x})} + e^{Q_B(\mathbf{x})}} \\ &= \frac{1}{1 + e^{Q_A(\mathbf{x}) - Q_B(\mathbf{x})}} \end{aligned}$$

We already showed the the decision boundary in LDA is linear — it is the set of points $\mathbf{x}$ such that

$$Q_A(\mathbf{x}) - Q_B(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} - b = 0$$

for some appropriate vector $\mathbf{w}$ and bias b. We therefore have that

$$P(Y = A|\mathbf{x}) = \frac{1}{1 + e^{\mathbf{w}^\top \mathbf{x} - b}} = s(\mathbf{w}^\top \mathbf{x} - b)$$

As we can see, even though logistic regression is a discriminative method and LDA is a generative method, both methods complement each other, arriving at the same form for the posterior distribution.

Generalizing to Multiple Classes

The analysis on the decision boundary in QDA and LDA can be extended to the general case when there are more than two classes. In the multiclass setting, the decision boundary is a collection of linear boundaries in LDA and quadratic boundaries in QDA. The following **Voronoi** diagrams illustrate the point:

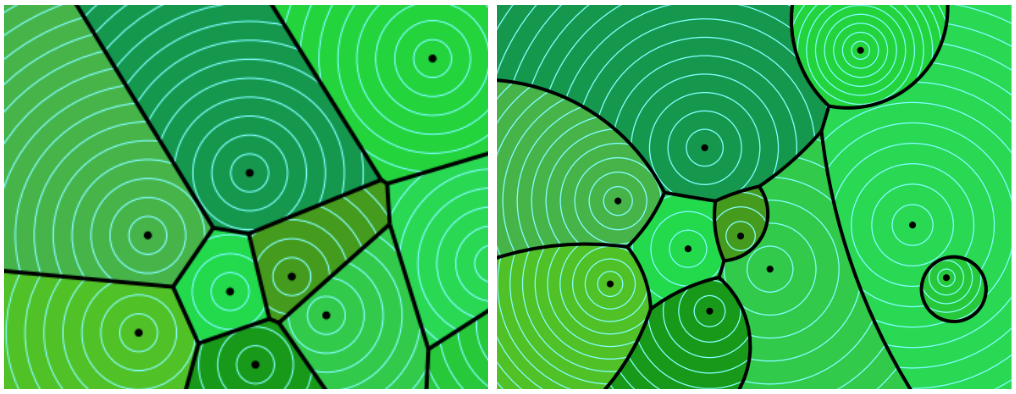


Figure 5.8: LDA (left) vs QDA (right): a collection of linear vs quadratic level set boundaries. Source: Professor Shewchuk's notes

5.5 Support Vector Machines

So far we've explored **generative classifiers** (LDA) and **discriminative classifiers** (logistic regression), but in both of these methods, we tasked ourselves with modeling some kind of probability distribution. One observation about classification is that in the end, if we only care about assigning each data point a class, all we really need to know do is find a "good" decision boundary, and we can skip thinking about the distributions. **Support Vector Machines (SVMs)** are an attempt to model decision boundaries directly in this spirit.

Here's the setup for the problem. We are given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, +1\}$. Our goal is to find a $d-1$ dimensional **hyperplane** decision boundary H which separates the $+1$'s from the -1's.

Motivation for SVMs

In order to motivate SVMs, we first have to understand the simpler **perceptron** algorithm and its shortcomings. Given that the training data is **linearly separable**, the perceptron algorithm finds a $d-1$ dimensional hyperplane that perfectly separates the $+1$'s from the -1's. Mathematically, the goal is to learn a set of parameters $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, that satisfy the linear separability constraints:

$$\forall i, \quad \begin{cases} \mathbf{w}^\top \mathbf{x}_i - b \geq 0 & \text{if } y_i = 1 \\ \mathbf{w}^\top \mathbf{x}_i - b \leq 0 & \text{if } y_i = -1 \end{cases}$$

Equivalently,

$$\forall i, \quad y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 0$$

The resulting decision boundary is a hyperplane $H = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} - b = 0\}$. All points on the positive side of the hyperplane are classified as $+1$, and all points on the negative side are classified as -1.

Perceptrons have two major shortcomings that as we shall see, SVMs can overcome. First of all, if the data is not linearly separable, the perceptron fails to find a stable solution. As we shall see, soft-margin SVMs fix this issue by allowing best-fit decision boundaries even when the data is not linearly separable. Second, if the data is linearly separable, the perceptron could find infinitely many hyperplanes that the perceptron could pick — if $(\mathbf{w}, b)$ is a pair that separates the data points, then the perceptron could also end up choosing a slightly different $(\mathbf{w}, b + \epsilon)$ pair that still separates the data points. Some hyperplanes are better than others, but the perceptron cannot distinguish between them. This leads to generalization issues.

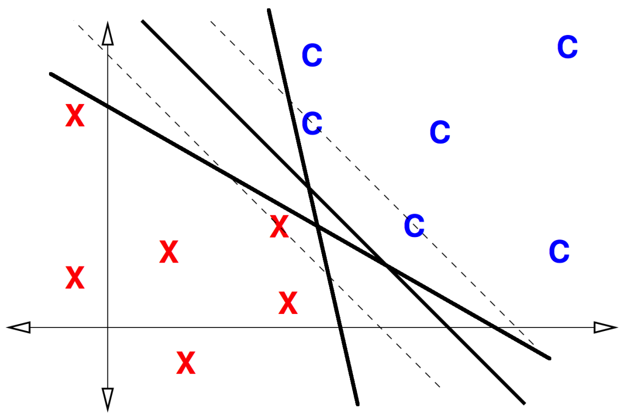


Figure 5.9: Several possible decision boundaries under the perceptron. The X's and C's represent the +1's and −1's respectively.

In the figure above, we consider three potential linear separators that satisfy the constraints. To the eyes of the perceptron algorithm, all three are perfectly valid linear separators. Ideally, we should not treat all linear separators equally — some are better than others. One could imagine that if we observed new test points that are nearby the region of C's (or X's) in the training data, they should also be of class C (or X). The two separators close to the training points would incorrectly classify some of these new test points, while the third separator which maintains a large distance to the points would classify them correctly. The perceptron algorithm does not take this reasoning into account, and may find a classifier that does not generalize well to unseen data.

Hard-Margin SVMs

Hard-Margin SVMs address the generalization problem of perceptrons by maximizing the **margin**, formally defined as the minimum distance from the decision boundary to the training points.

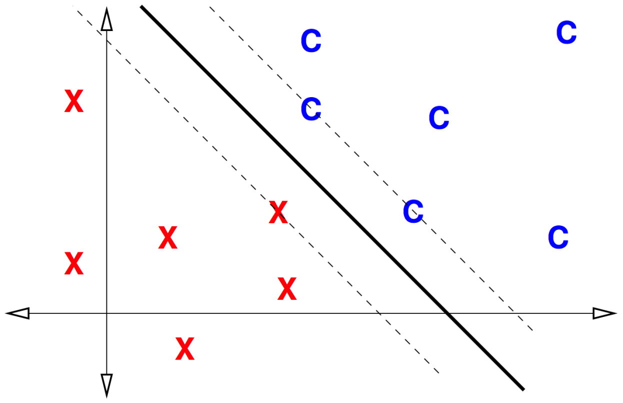


Figure 5.10: The optimal decision boundary (as shown) maximizes the margin.

Intuitively, maximizing the margin allows us to generalize better to unseen data, because the decision boundary with the maximum margin is as far away from the training data as possible and the boundary cannot be violated unless the unseen data contains outliers.

Simply put, the goal of hard-margin SVMs is to find a hyperplane H that maximizes the margin m. Let's formalize an optimization problem for hard-margin SVMs. The variables we are trying to optimize over are the margin m and the parameters of the hyperplane, $\mathbf{w}$ and b. The objective is to maximize the margin m, subject to the following constraints:

- All points classified as $+1$ are to the positive side of the hyperplane and their distance to H is greater than the margin
- All points classified as -1 are to the negative side of the hyperplane and their distance to H is greater than the margin
- The margin is non-negative.

Let's express the first two constraints mathematically. First, note that the vector $\mathbf{w}$ is perpendicular to the hyperplane $H = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} - b = 0\}$.

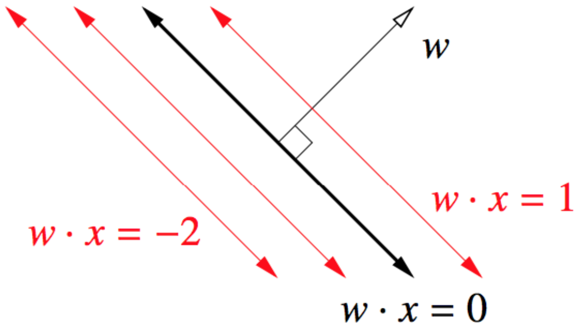


Figure 5.11: Image courtesy Professor Shewchuk's notes.

Proof: consider any two points on H, $\mathbf{x}_0$ and $\mathbf{x}_1$. We will show that $(\mathbf{x}_1 - \mathbf{x}_0) \perp \mathbf{w}$. Note that

$$(\mathbf{x}_1 - \mathbf{x}_0)^\top(\mathbf{w}) = (\mathbf{x}_1 - \mathbf{x}_0)^\top((\mathbf{x}_1 + \mathbf{w}) - \mathbf{x}_1) = \mathbf{x}_1{}^\top\mathbf{w} - \mathbf{x}_0{}^\top\mathbf{w} = b - b = 0$$

Since $\mathbf{w}$ is perpendicular to H, the (shortest) distance from any arbitrary point $\mathbf{z}$ to the hyperplane H is determined by a scaled multiple of $\mathbf{w}$. If we take any point on the hyperplane $\mathbf{x}_0$, the distance from $\mathbf{z}$ to H is the length of the projection from $\mathbf{z} - \mathbf{x}_0$ to the vector $\mathbf{w}$, which is

$$D = \frac{|\mathbf{w}^\top(\mathbf{z} - \mathbf{x}_0)|}{\|\mathbf{w}\|_2} = \frac{|\mathbf{w}^\top\mathbf{z} - \mathbf{w}^\top\mathbf{x}_0|}{\|\mathbf{w}\|_2} = \frac{|\mathbf{w}^\top\mathbf{z} - b|}{\|\mathbf{w}\|_2}$$

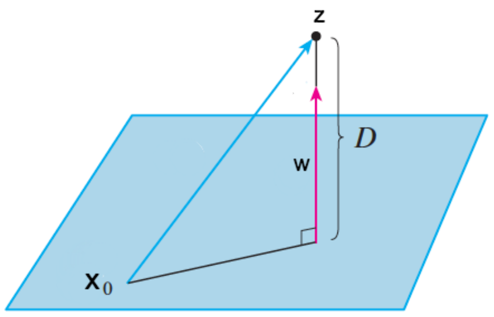


Figure 5.12: Shortest distance from z to H is determined by projection of $z - \mathbf{x}_0$ onto $\mathbf{w}$

Therefore, the distance from any of the training points $\mathbf{x}_i$ to H is

$$\frac{|\mathbf{w}^\top\mathbf{x}_i - b|}{\|\mathbf{w}\|_2}$$

In order to ensure that positive points are on the positive side of the hyperplane outside a margin of size m, and that negative points are on the negative side of the hyperplane outside a margin of size m, we can express the constraint

$$y_i \frac{(\mathbf{w}^\top\mathbf{x}_i - b)}{\|\mathbf{w}\|_2} \geq m$$

Putting everything together, we have the following optimization problem:

$$\begin{aligned} \max_{m,\mathbf{w},b} \quad & m \\ \text{s.t.} \quad & y_i \frac{(\mathbf{w}^\top\mathbf{x}_i - b)}{\|\mathbf{w}\|_2} \geq m \quad \forall i \\ & m \geq 0 \end{aligned} \tag{5.1}$$

Maximizing the margin m implies that there exists at least one point on the positive side of the hyperplane and at least one point on the negative side whose distance to the hyperplane is exactly equal to m. These points are the **support vectors**, hence the name “support vector machines.” They are called support vectors because they literally hold/support the margin planes in place.

Through a series of optimization steps, we can simplify the problem by removing the margin variable and just optimizing the parameters of the hyperplane. Note that the current optimization formulation does not induce a unique choice of $\mathbf{w}$ and b: if $(m^*, \mathbf{w}^*, b^*)$ is a solution, then $(m^*, \alpha\mathbf{w}^*, \alpha b^*)$ is also a solution, for any $\alpha > 0$. In order to ensure that $\mathbf{w}$ and b are unique (without changing the nature of the optimization problem), we can add an additional constraint for the norm of $\mathbf{w}$:

$\|\mathbf{w}\|_2 = \alpha$, for some $\alpha > 0$. In particular, we can add the constraint $\|\mathbf{w}\|_2 = \frac{1}{m}$ or equivalently, $m = \frac{1}{\|\mathbf{w}\|_2}$:

$$\begin{aligned} \max_{m,\mathbf{w},b} \quad & m \\ \text{s.t.} \quad & y_i \frac{(\mathbf{w}^\top \mathbf{x}_i - b)}{\|\mathbf{w}\|_2} \geq m \quad \forall i \\ & m \geq 0 \\ & m = \frac{1}{\|\mathbf{w}\|_2} \end{aligned} \tag{5.2}$$

Now, we can substitute $m = \frac{1}{\|\mathbf{w}\|_2}$ and eliminate m from the optimization:

$$\begin{aligned} \max_{\mathbf{w},b} \quad & \frac{1}{\|\mathbf{w}\|_2} \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 \quad \forall i \end{aligned} \tag{5.3}$$

At last, we have formulated the hard-margin SVM optimization problem! The standard formulation of hard-margin SVMs is

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|_2^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 \quad \forall i \end{aligned} \tag{5.4}$$

Soft-Margin SVMs

The hard-margin SVM optimization problem has a unique solution only if the data are linearly separable, but it has no solution otherwise. This is because the constraints are impossible to satisfy if we can't draw a hyperplane that separates the $+1$'s from the -1's. In addition, hard-margin SVMs are very sensitive to outliers — for example, if our data is class-conditionally distributed Gaussian such that the two Gaussians are far apart, if we witness an outlier from class $+1$ that crosses into the typical region for class -1, then hard-margin SVM will be forced to compromise a more generalizable fit in order to accommodate for this point. Our next goal is to come up with a classifier that is not sensitive to outliers and can work even in the presence of data that is not linearly separable. To this end, we'll talk about **Soft-Margin SVMs**.

A soft-margin SVM modifies the constraints from the hard-margin SVM by allowing some points to violate the margin. It introduces **slack variables** ξ_i, one for each training point, into the constraints:

$$\begin{aligned} y_i(\mathbf{w}^\top \mathbf{x}_i - b) &\geq 1 - \xi_i \\ \xi_i &\geq 0 \end{aligned}$$

The constraints are now a less-strict, *softer* version of the hard-margin SVM constraints, because each point $\mathbf{x}_i$ need only be a "distance" of $1 - \xi_i$ of the separating hyperplane instead of a hard "distance" of 1.

(By the way, the Greek letter ξ is spelled "xi" and pronounced "zai." ξ_i is pronounced "zai-eye.")

These constraints would be fruitless if we didn't bound the values of the ξ_i's — by setting them to large values, we are saying that any point may violate the margin by an arbitrarily large distance, which makes our choice of $\mathbf{w}$ meaningless. Therefore we modify the objective function to penalize the slacks:

$$\min_{\mathbf{w},b,,\xi_i} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i$$

Where C is a hyperparameter tuned through cross-validation. Putting the objective and constraints together, the soft-margin SVM optimization problem is

$$\begin{aligned} \min_{\mathbf{w},b,\xi_i} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned} \tag{5.5}$$

The table below compares the effects of having a large C versus a small C. As C goes to infinity, the penalty for having non-zero ξ_i goes to infinity, and thus we force the ξ_i's to be zero, which is exactly the setting of the hard-margin SVM.

	small C	large C
Desire	maximize margin	keep ξ_i's small or zero
Danger	underfitting	overfitting
Outliers	less sensitive	more sensitive

SVMs as Tikhonov Regularization Learning

The constrained version of soft-margin SVM optimization problem

$$\begin{aligned} \min_{\mathbf{w},b,\xi_i} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned} \tag{5.6}$$

can equivalently be expressed in an unconstrained fashion:

$$\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^n \max(1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b), 0) + \lambda\|\mathbf{w}\|^2$$

Let's see why. Manipulating the first constraint of constraints, we have that

$$\xi_i \geq 1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b)$$

Combining with the constraint $\xi_i \geq 0$, we have that

$$\xi_i \geq \max(1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b), 0)$$

At the optimal value of the optimization problem, these inequalities must be tight. Otherwise, we could lower each ξ_i to equal $\max(1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b), 0)$ and decrease the value of the objective function. Thus we can rewrite the soft-margin SVM optimization problem as

$$\begin{aligned} \min_{\mathbf{w},b,\xi_i} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i \\ \text{s.t.} \quad & \xi_i = \max(1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b), 0) \quad \forall i \end{aligned} \tag{5.7}$$

Simplifying further, we can remove the constraints:

$$\min_{\mathbf{w},b} \quad C\sum_{i=1}^{n} \max(1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b), 0) + \frac{1}{2}\|\mathbf{w}\|^2 \tag{5.8}$$

If we divide by Cn (which does not change the optimal solution of the optimization problem), we can see that this formulation is equivalent to the regularized regression problem, with $\lambda = \frac{1}{2Cn}$. Thus we have two interpretations of soft-margin SVM: either as finding a max-margin hyperplane that is allowed to make some mistakes via slack variables ξ_i, or as regularized empirical risk minimization.

Through some manipulations, we can formulate SVMs and many other classification problems as an optimization over the objective

$$\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n} L(y_i, \mathbf{w}^\top \mathbf{x}_i - b) + \lambda\|\mathbf{w}\|^2$$

over some specified loss function and a possible regularization term (sometimes we may set $\lambda = 0$). The simplest loss function that we can optimize is the 0-1 **step loss**:

$$L_{\text{STEP}}(y, \mathbf{w}^\top \mathbf{x} - b) = \begin{cases} 1 & y(\mathbf{w}^\top \mathbf{x} - b) < 0 \\ 0 & y(\mathbf{w}^\top \mathbf{x} - b) \geq 0 \end{cases}$$

The 0-1 loss is 0 if $\mathbf{x}$ is correctly classified and 1 otherwise. Minimizing $\frac{1}{n}\sum_{i=1}^{n} L(y_i, \mathbf{w}^\top \mathbf{x}_i - b)$ directly minimizes classification error on the training set. However, the 0-1 loss is difficult to optimize: it is neither convex nor differentiable (see Figure 7.1). Furthermore, if we exclude the regularization term, we do not penalize the classifier for being close to the training points, which leads to generalization issues.

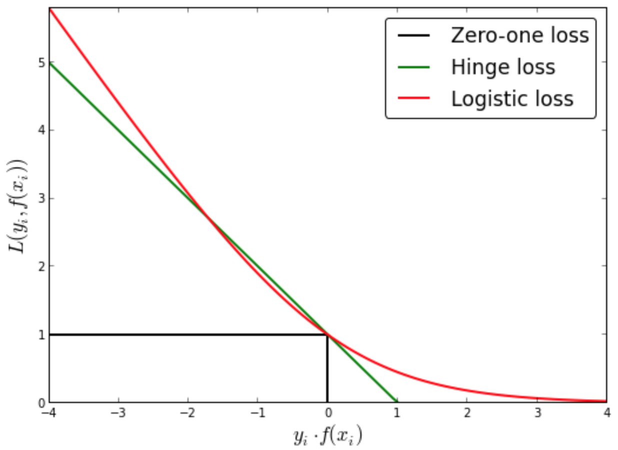


Figure 5.13: Step (0-1) loss, hinge loss, and logistic loss. Logistic loss is convex and differentiable, hinge loss is only convex, and step loss is neither.

Another loss function that we have seen is the **logistic loss**, which is used in logistic regression:

$$L_{\text{LR}}(y, \mathbf{w}^\top \mathbf{x} - b) = y \ln\left(\frac{1}{s(\mathbf{w}^\top \mathbf{x} - b)}\right) + (1 - y)\ln\left(\frac{1}{1 - s(\mathbf{w}^\top \mathbf{x} - b)}\right)$$

The logistic loss is convex and differentiable, and is optimized using gradient descent methods. The logistic loss is the basis for logistic regression, and it works well without regularization:

$$\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n} L_{\text{LR}}(y_i, \mathbf{w}^\top \mathbf{x}_i - b)$$

The **hinge loss** modifies the 0-1 loss to be convex. The points with $y(\mathbf{w}^\top \mathbf{x} - b) \geq 0$ should remain at 0 loss, but we may consider allowing a linear penalty "ramp" for misclassified points. This leads us to the hinge loss, as illustrated in Figure 7.1:

$$L_{\text{HINGE}}(y, \mathbf{w}^\top \mathbf{x} - b) = \max(1 - y(\mathbf{w}^\top \mathbf{x} - b), 0)$$

The ramp ensures that misclassified points that are close to the boundary are penalized less than misclassified points that are far from the boundary. The perceptron algorithm optimizes over the sum of hinge losses contributed from all of the training points:

$$\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n} L_{\text{HINGE}}(y_i, \mathbf{w}^\top \mathbf{x}_i - b)$$

The SVM formulation is an optimization over the same problem, with the addition of a regularization term:

$$\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n} L_{\text{HINGE}}(y_i, \mathbf{w}^\top \mathbf{x}_i - b) + \lambda\|\mathbf{w}\|^2$$

The regularization term allows for better generalization, in this case by penalizing choices of $\mathbf{w}$ for which the margin is small.

5.6 Duality

As we have seen in our discussion of kernels, ridge regression can be viewed in two ways: (1) an optimization problem over the weights $\mathbf{w} \in \mathbb{R}^d$ which scales according to the dimensionality of the augmented feature space, and (2) an optimization problem over the weights $\boldsymbol{\alpha} \in \mathbb{R}^n$ which scales according to the the number of training points. These two viewpoints give rise to two equivalent solutions:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^\top \mathbf{y} \quad \text{and} \quad \mathbf{w}^* = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}\mathbf{y}$$

The second (kernelized) expression is much more efficient to calculate when the number of training points n is significantly smaller than the number of augmented features d. Recall that the derivation for the kernelized expression relied on invoking the fundamental theorem of linear algebra and solving for a set of dual variables. While this approach is certainly valid, it may not be applicable for kernelizing all problems. Rather, a more principled approach is to apply Lagrangian duality and solve the dual problem. In this section we will introduce duality for arbitrary optimization problems, and then use duality to derive the kernelized versions for ridge regression and SVMs.

Primal and Dual Problem

All optimization problems can be expressed in the standard form

$$\begin{aligned} \min_{\mathbf{x}} \quad & f_0(\mathbf{x}) \\ \text{s.t.} \quad & f_i(\mathbf{x}) \leq 0 \quad i = 1, \dots, m \\ & h_j(\mathbf{x}) = 0 \quad j = 1, \dots, n \end{aligned} \tag{5.9}$$

For the purposes of our discussion, assume that $\mathbf{x} \in \mathbb{R}^d$. The components of an optimization problem are:

- The **objective function** $f_0(\mathbf{x})$
- The **inequality constraints**: expressions involving $f_i(\mathbf{x})$
- The **equality constraints**: expressions involving $h_j(\mathbf{x})$

Working with the constraints can be cumbersome and challenging to manipulate, and it would be ideal if we could somehow turn this constrained optimization problem into an unconstrained one. One idea is to re-express the optimization problem into

$$\min_{\mathbf{x}} \mathcal{L}(\mathbf{x})$$

where

$$\mathcal{L}(\mathbf{x}) = \begin{cases} f_0(\mathbf{x}) & \text{if} \quad f_i(\mathbf{x}) \leq 0, \ \forall i \in \{1 \mathrel{..} m\} \quad \text{and} \quad h_j(\mathbf{x}) = 0, \ \forall j \in \{1 \mathrel{..} n\} \\ \infty & \text{otherwise} \end{cases} \tag{5.10}$$

Note that the unconstrained optimization problem above is equivalent to the original constrained problem. Even though the unconstrained problem considers values that violate the constraints (and therefore are not in the feasible set for the constrained optimization problem), it will effectively ignore them because they are treated as ∞ in a minimization problem.

Even though we are now dealing with an unconstrained problem, it still is difficult to solve the optimization problem, because we still have to deal with all of the casework in the objective function $\mathcal{L}(\mathbf{x})$. In order to solve this issue, we have to introduce dual variables, specifically one set of dual variables for the equality constraints, and one set for the inequality constraints. If we only take into account the dual variables for the equality constraints, the optimization problem now becomes

$$\min_{\mathbf{x}} \max_{\boldsymbol{\nu}} \mathcal{L}(\mathbf{x}, \boldsymbol{\nu})$$

where

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\nu}) = \begin{cases} f_0(\mathbf{x}) + \sum_{j=1}^{n} \nu_j h_j(\mathbf{x}) & \text{if} \quad f_i(\mathbf{x}) \leq 0, \ \forall i \in \{1 \mathrel{..} m\} \\ \infty & \text{otherwise} \end{cases} \tag{5.11}$$

We are still working with an unconstrained optimization problem, except that now, we are optimizing over two sets of variables: the **primal variables** $\mathbf{x} \in \mathbb{R}^d$ and the **dual variables** $\boldsymbol{\nu} \in \mathbb{R}^n$. Also note that the optimization problem has now become a nested one, with an inner optimization problem the maximizes over the dual variables, and an outer optimization problem that minimizes over the primal variables. Let's examine why this optimization problem is equivalent to the original constrained optimization problem:

- Any $\mathbf{x}$ that violates the inequality constraints is still treated as ∞ by the outer minimization problem over $\mathbf{x}$ and therefore ignored
- For any $\mathbf{x}$ that violates the equality constraints (meaning that $\exists j$ s.t. $h_j(\mathbf{x}) \neq 0$), the inner maximization problem over $\boldsymbol{\nu}$ can choose ν_j as ∞ if $h_j(\mathbf{x}) > 0$ (or ν_j as $-\infty$ if $h_j(\mathbf{x}) < 0$) to cause the inner maximization to go to ∞, therefore being ignored by the outer minimization over $\mathbf{x}$

- For any $\mathbf{x}$ that does not violate any of the equality or inequality constraints, the inner maximization problem over $\boldsymbol{\nu}$ is simply equal to $f_0(\mathbf{x})$

This solution comes at a cost — in an effort to remove the equality constraints, we had to add in dual variables, one for each equality constraint. With this in mind, let's try to do the same for the inequality constraints. Adding in dual variable λ_i to represent each inequality constraint, we now have

$$\begin{aligned} \min_{\mathbf{x}} \max_{\boldsymbol{\lambda},\boldsymbol{\nu}} \quad & \mathcal{L}(\mathbf{x},\boldsymbol{\lambda},\boldsymbol{\nu}) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^{n} \nu_j h_j(\mathbf{x}) \\ \text{s.t.} \quad & \lambda_i \geq 0 \quad i = 1, \ldots, m \end{aligned} \tag{5.12}$$

For convenience, we can place the constraints involving $\boldsymbol{\lambda}$ into the optimization variable.

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}\geq\mathbf{0},\boldsymbol{\nu}} \mathcal{L}(\mathbf{x},\boldsymbol{\lambda},\boldsymbol{\nu}) = f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^{n} \nu_j h_j(\mathbf{x}) \tag{5.13}$$

This optimization problem above is otherwise known as the **primal** (not to be confused with the *primal variables*), and its optimal value is indeed *equivalent* to that of the original constrained optimization problem.

$$p^* = \min_{\mathbf{x}} \max_{\boldsymbol{\lambda}\geq\mathbf{0},\boldsymbol{\nu}} \mathcal{L}(\mathbf{x},\boldsymbol{\lambda},\boldsymbol{\nu}) \tag{5.14}$$

We can verify that this is indeed the case:

- For any $\mathbf{x}$ that violates the inequality constraints (meaning that $\exists i \in \{1 \,..\, m\}$ s.t. $f_i(\mathbf{x}) > 0$), the inner maximization problem over $\boldsymbol{\lambda}$ can choose λ_i as ∞ to cause the inner maximization go to ∞, therefore being ignored by the outer minimization over $\mathbf{x}$
- For any $\mathbf{x}$ that violates the equality constraints (meaning that $\exists j$ s.t. $h_j(\mathbf{x}) \neq 0$), the inner maximization problem over $\boldsymbol{\nu}$ can choose ν_j as ∞ if $h_j(\mathbf{x}) > 0$ (or ν_j as $-\infty$ if $h_j(\mathbf{x}) < 0$) to cause the inner maximization go to ∞, therefore being ignored by the outer minimization over $\mathbf{x}$
- For any $\mathbf{x}$ that does not violate any of the equality or inequality constraints, in the inner maximization problem over $\boldsymbol{\nu}$, the expression $\sum_{j=1}^{n} \nu_j h_j(\mathbf{x})$ evaluates to 0 no matter what the value of $\boldsymbol{\nu}$ is, and in the inner maximization problem over $\boldsymbol{\lambda}$, the expression $\sum_{i=1}^{m} \lambda_i f_i(\mathbf{x})$ can at maximum be 0, because λ_i is constrained to be non-negative, and $f_i(\mathbf{x})$ is non-positive. Therefore, at best, the maximization problem sets $\lambda_i f_i(\mathbf{x}) = 0$, and

$$\max_{\boldsymbol{\lambda}\geq\mathbf{0},\boldsymbol{\nu}} \mathcal{L}(\mathbf{x},\boldsymbol{\lambda},\boldsymbol{\nu}) = f_0(\mathbf{x})$$

In its full form, the objective $\mathcal{L}(\mathbf{x},\boldsymbol{\lambda},\boldsymbol{\nu})$ is called the **Lagrangian**, and it takes into account the unconstrained set of primal variables $\mathbf{x} \in \mathbb{R}^d$, the constrained set of dual variables $\boldsymbol{\lambda} \in \mathbb{R}^n$ corresponding to the inequality constraints, and the unconstrained set of dual variables $\boldsymbol{\nu} \in \mathbb{R}^m$ corresponding to the equality constraints. Note that our dual variables λ_i are in fact constrained, so ultimately we were not able to turn the original optimization problem into an unconstrained one, but our constraints are much simpler than before.

The **dual** of this optimization problem is still over the same optimization objective, except that now we swap the order of the maximization of the dual variables and the minimization of the primal variables.

$$d^* = \max_{\boldsymbol{\lambda}\geq\mathbf{0},\boldsymbol{\nu}} \min_{\mathbf{x}} \mathcal{L}(\mathbf{x},\boldsymbol{\lambda},\boldsymbol{\nu}) = \max_{\boldsymbol{\lambda}\geq\mathbf{0},\boldsymbol{\nu}} g(\boldsymbol{\lambda},\boldsymbol{\nu}) \tag{5.15}$$

The dual is effectively a maximization problem (over the dual variables):

$$d^* = \max_{\boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\nu}} g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \tag{5.16}$$

where

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) \tag{5.17}$$

The dual is very useful to work with, because now the inner optimization problem over $\mathbf{x}$ is an unconstrained problem! Furthermore, the dual $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ is always a concave function, regardless of the primal objective function or its constraints. This is because the dual is a pointwise minimum of concave functions, which itself is a concave function. Specifically $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ is a pointwise minimum of functions $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ that are affine in the dual variables (which are both concave and convex at the same time).

Strong Duality and KKT Conditions

Let's examine the relationship between the primal and dual problem. It is *always* true that the solution to the primal problem is at least as large as the solution to the dual problem:

$$p^* \geq d^* \tag{5.18}$$

This condition is know as **weak duality**.

Proof. We know that

$$\forall \mathbf{x}, \boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\nu} \quad \max_{\tilde{\boldsymbol{\lambda}} \geq 0, \tilde{\boldsymbol{\nu}}} \mathcal{L}(\mathbf{x}, \tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}}) \geq \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) \geq \min_{\tilde{\mathbf{x}}} \mathcal{L}(\tilde{\mathbf{x}}, \boldsymbol{\lambda}, \boldsymbol{\nu})$$

More compactly,

$$\forall \mathbf{x}, \boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\nu} \quad \max_{\tilde{\boldsymbol{\lambda}} \geq \mathbf{0}, \tilde{\boldsymbol{\nu}}} \mathcal{L}(\mathbf{x}, \tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}}) \geq \min_{\tilde{\mathbf{x}}} \mathcal{L}(\tilde{\mathbf{x}}, \boldsymbol{\lambda}, \boldsymbol{\nu})$$

Since this is true for all $\mathbf{x}, \boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\nu}$ this is true in particular when we set

$$\mathbf{x} = \arg\min_{\tilde{\mathbf{x}}} \max_{\tilde{\boldsymbol{\lambda}} \geq \mathbf{0}, \tilde{\boldsymbol{\nu}}} \mathcal{L}(\tilde{\mathbf{x}}, \tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}})$$

and

$$\boldsymbol{\lambda}, \boldsymbol{\nu} = \arg\max_{\tilde{\boldsymbol{\lambda}} \geq \mathbf{0}, \tilde{\boldsymbol{\nu}}} \min_{\tilde{\mathbf{x}}} \mathcal{L}(\tilde{\mathbf{x}}, \tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}})$$

We therefore know that

$$p^* = \min_{\tilde{\mathbf{x}}} \max_{\tilde{\boldsymbol{\lambda}} \geq \mathbf{0}, \tilde{\boldsymbol{\nu}}} \mathcal{L}(\tilde{\mathbf{x}}, \tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}}) \geq \max_{\tilde{\boldsymbol{\lambda}} \geq \mathbf{0}, \tilde{\boldsymbol{\nu}}} \min_{\tilde{\mathbf{x}}} \mathcal{L}(\tilde{\mathbf{x}}, \tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}}) = d^*$$

□

The difference $p^* - d^*$ is known as the **duality gap**. In the case of **strong duality**, the duality gap is 0. That is, we can swap the order of the minimization and maximization and up with the same optimal value:

$$p^* = d^* \tag{5.19}$$

There are several useful theorems detailing the existence of strong duality, such as **Slater's theorem**, which states that if the primal problem is convex, and there exists an $\mathbf{x}$ that can *strictly* meet the inequality constraints and meet the equality constraints, then strong duality holds. Given that strong duality holds, the **Karush-Kuhn-Tucker (KKT) conditions** can help us find the solution to the dual variables of the optimization problem. The KKT conditions are composed of:

1. Primal feasibility (inequalities)
$$f_i(\mathbf{x}) \leq 0,\ \forall i \in \{1 \mathinner{..} m\}$$
2. Primal feasibility (equalities)
$$h_j(\mathbf{x}) = 0,\ \forall j \in \{1 \mathinner{..} n\}$$
3. Dual feasibility
$$\lambda_i \geq 0,\ \forall i \in \{1 \mathinner{..} m\}$$
4. Complementary Slackness
$$\lambda_i f_i(\mathbf{x}) = 0,\ \forall i \in \{1 \mathinner{..} m\}$$
5. Stationarity
$$\nabla_{\mathbf{x}} f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} f_i(\mathbf{x}) + \sum_{j=1}^{n} \nu_j \nabla_{\mathbf{x}} h_j(\mathbf{x}) = 0$$

Let's see how the KKT conditions relate to strong duality.

Theorem 1. *If $\mathbf{x}^*$ and $\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ are the primal and dual solutions respectively, with zero duality gap (i.e. strong duality holds), then $\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ also satisfy the KKT conditions.*

Proof. KKT conditions 1, 2, 3 are trivially true, because the primal solution $\mathbf{x}^*$ must satisfy the primal constraints, and the dual solution $\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ must satisfy the dual constraints. Now, let's prove conditions 4 and 5. We know that since strong duality holds, we can say that

$$\begin{align}
p^* = f_0(\mathbf{x}^*) &= g(\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) = d^* \tag{5.20}\\
&= \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) \tag{5.21}\\
&\leq \mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) \tag{5.22}\\
&= f_0(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i^* f_i(\mathbf{x}^*) + \cancel{\sum_{j=1}^{n} \nu_j^* h_j(\mathbf{x}^*)} \tag{5.23}\\
&= f_0(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i^* f_i(\mathbf{x}^*) \tag{5.24}\\
&\leq f_0(\mathbf{x}^*) \tag{5.25}
\end{align}$$

We cancel the terms involving $h_j(\mathbf{x}^*)$ because we know that the primal solution must satisfy $h_j(\mathbf{x}^*) = 0$. Furthermore, we know that $\lambda_i^* f_i(\mathbf{x}^*) \leq 0$, because $\lambda_i^* \geq 0$ in order to satisfy the dual constraints, and $f_i(\mathbf{x}^*) \leq 0$ in order to satisfy the primal constraints. Since we established that $f_0(\mathbf{x}^*) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) \leq \mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) \leq f_0(\mathbf{x}^*)$, we know that all of the inequalities hold with equality and therefore $\mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) = \min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*)$. This implies KKT condition 5 (stationarity), that
$$\nabla_{\mathbf{x}} f_0(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i^* \nabla_{\mathbf{x}} f_i(\mathbf{x}^*) + \sum_{j=1}^{n} \nu_j^* \nabla_{\mathbf{x}}^* h_j(\mathbf{x}^*) = 0$$
Finally, note that due to the equality $f_0(\mathbf{x}^*) + \sum_{i=1}^{m} \lambda_i^* f_i(\mathbf{x}^*) = f_0(\mathbf{x}^*)$, we know that $\sum_{i=1}^{m} \lambda_i^* f_i(\mathbf{x}^*) = 0$. This combined with the fact that $\forall i \quad \lambda_i^* f_i(\mathbf{x}^*) \leq 0$, establishes KKT condition 4 (complementary slackness):
$$\lambda_i^* f_i(\mathbf{x}^*) = 0,\ \forall i \in \{1 \mathinner{..} m\}$$
□

The theorem above establishes that in the presence of strong duality, if the solutions are optimal, then they satisfy the KKT conditions. Let's prove a statement that is almost (but not quite) the converse, which will be much more helpful for solving optimization problems.

Theorem 2. *If $\bar{\mathbf{x}}$ and $\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}$ satisfy the KKT conditions, and the primal problem is convex, then they are the optimal solutions to the primal and dual problems with zero duality gap.*

Proof. If $\bar{\mathbf{x}}$ and $\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}$ satisfy KKT conditions 1, 2, 3 we know that they are at least feasible for the primal and dual problem. From the KKT stationarity condition we know that

$$\nabla_{\mathbf{x}} f_0(\bar{\mathbf{x}}) + \sum_{i=1}^{m} \bar{\lambda_i} \nabla_{\mathbf{x}} f_i(\bar{\mathbf{x}}) + \sum_{j=1}^{n} \bar{\nu_j} \nabla_{\mathbf{x}} h_j(\bar{\mathbf{x}}) = 0$$

Since the primal problem is convex, we know that $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ is convex in $\mathbf{x}$, and if the gradient of $\mathcal{L}(\mathbf{x}, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}})$ at $\bar{\mathbf{x}}$ is $\mathbf{0}$, we know that

$$\bar{\mathbf{x}} = \arg\min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}})$$

Therefore, we know that the optimal primal values for the primal problem optimize the inner optimization problem of the dual problem, and

$$g(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}) = f_0(\bar{\mathbf{x}}) + \sum_{i=1}^{m} \bar{\lambda_i} f_i(\bar{\mathbf{x}}) + \sum_{j=1}^{n} \bar{\nu_j} h_j(\bar{\mathbf{x}})$$

By the primal feasibility conditions for $h_j(\mathbf{x})$ and the complementary slackness condition, we know that

$$g(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}) = f_0(\bar{\mathbf{x}})$$

Now, all we have to do is to prove that $\bar{\mathbf{x}}$ and $\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}$ are primal and dual optimal, respectively. Note that since weak duality always holds, we know that

$$p^* \geq d^* = \max_{\boldsymbol{\lambda} \geq \mathbf{0}, \boldsymbol{\nu}} g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \geq g(\tilde{\boldsymbol{\lambda}}, \tilde{\boldsymbol{\nu}}), \quad \forall \tilde{\boldsymbol{\lambda}} \geq \mathbf{0}, \tilde{\boldsymbol{\nu}}$$

Since we know that $p^* \geq g(\boldsymbol{\lambda}, \boldsymbol{\nu})$, we can also say that

$$f_0(\mathbf{x}) - p^* \leq f_0(\mathbf{x}) - g(\boldsymbol{\lambda}, \boldsymbol{\nu})$$

And if we have that $f_0(\bar{\mathbf{x}}) = g(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}})$ as we deduced earlier, then

$$f_0(\bar{\mathbf{x}}) - p^* \leq f_0(\bar{\mathbf{x}}) - g(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}) = 0 \implies p^* \geq f_0(\bar{\mathbf{x}})$$

Since p^* is the minimum value for the primal problem, we can go further by saying that $p^* \geq f_0(\bar{\mathbf{x}})$ holds with equality and

$$p^* = f_0(\bar{\mathbf{x}}) = g(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}) \leq d^*$$

since it always holds that $p^* \geq d^*$ we conclude that

$$p^* = f_0(\bar{\mathbf{x}}) = g(\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}) = d^*$$

Therefore, we have proven that $\bar{\mathbf{x}}$ and $\bar{\boldsymbol{\lambda}}, \bar{\boldsymbol{\nu}}$ are primal and dual optimal respectively, with zero duality gap. We eventually arrived at the conclusion that strong duality does indeed hold. □

Let's pause for a second to understand what we've found so far. Given an optimization problem, its primal problem is an optimization problem over the primal variables, and its dual problem is an optimization problem over the dual variables. If strong duality holds, then we can solve the dual problem and arrive at the same optimal value. In order to solve the dual, we have to first solve the unconstrained inner optimization problem over the primal variables and then solve the constrained outer optimization problem over the dual variables. But how do we even know in the first place that strong duality holds? This is where KKT comes into play. If the the primal problem is convex and the KKT conditions hold, we can solve for the dual variables easily and also verify strong duality does indeed hold. We shall do just that, in our discussion of dual ridge regression and dual SVMs.

Dual Ridge Regression

Let's derive kernel ridge regression again, using duality this time. Recall the unconstrained ridge regression formulation:

$$\min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \lambda\|\mathbf{w}\|^2$$

This formulation is not conducive to dualization, because it lacks constraints. We will add constraints by introducing a dummy variable $\mathbf{z} = \mathbf{X}\mathbf{w} - \mathbf{y}$ that corresponds to equality constraints:

$$\begin{aligned} \min_{\mathbf{w},\mathbf{z}} \quad & \|\mathbf{z}\|^2 + \lambda\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & \mathbf{z} = \mathbf{X}\mathbf{w} - \mathbf{y} \end{aligned} \tag{5.26}$$

Now we proceed to forming the dual problem. For the purposes of notation, note that we are using $\boldsymbol{\alpha}$ in place of $\boldsymbol{\nu}$, and there are no dual variables corresponding to $\boldsymbol{\lambda}$ because there are no inequality constraints. The Lagrangian is

$$\mathcal{L}(\mathbf{w}, \mathbf{z}, \boldsymbol{\alpha}) = \|\mathbf{z}\|^2 + \lambda\|\mathbf{w}\|^2 + \boldsymbol{\alpha}^\top(\mathbf{X}\mathbf{w} - \mathbf{y} - \mathbf{z})$$

The dual problem is

$$\max_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}) \tag{5.27}$$

where

$$g(\boldsymbol{\alpha}) = \min_{\mathbf{w},\mathbf{z}} \|\mathbf{z}\|^2 + \lambda\|\mathbf{w}\|^2 + \boldsymbol{\alpha}^\top(\mathbf{X}\mathbf{w} - \mathbf{y} - \mathbf{z}) \tag{5.28}$$

Since the $g(\boldsymbol{\alpha})$ is a convex minimization problem over the variables $\mathbf{w}$ and $\mathbf{z}$, we can simply set the derivative to $\mathbf{0}$ w.r.t. $\mathbf{w}$ and $\mathbf{z}$:

- $\nabla_{\mathbf{w}}\mathcal{L} = 2\lambda\mathbf{w} + \mathbf{X}^\top\boldsymbol{\alpha} = \mathbf{0} \implies \mathbf{w}^*(\boldsymbol{\alpha}) = -\frac{1}{2\lambda}\mathbf{X}^\top\boldsymbol{\alpha}$. This tells us that $\mathbf{w}^*$ is going to be a linear combination of the $\mathbf{x}_i$'s.
- $\nabla_{\mathbf{z}}\mathcal{L} = 2\mathbf{z} - \boldsymbol{\alpha} = \mathbf{0} \implies \mathbf{z}^*(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}$.

Plugging these optimal values back into the optimization problem, we have that

$$\begin{aligned} g(\boldsymbol{\alpha}) &= \min_{\mathbf{w},\mathbf{z}} \mathcal{L}(\mathbf{w}, \mathbf{z}, \boldsymbol{\alpha}) && (5.29) \\ &= \mathcal{L}(\mathbf{w}^*(\boldsymbol{\alpha}), \mathbf{z}^*(\boldsymbol{\alpha}), \boldsymbol{\alpha}) && (5.30) \\ &= \left\|-\frac{1}{2}\boldsymbol{\alpha}\right\|^2 + \lambda\left\|-\frac{1}{2\lambda}\mathbf{X}^\top\boldsymbol{\alpha}\right\|^2 + \boldsymbol{\alpha}^\top\left(\mathbf{X}\left(-\frac{1}{2\lambda}\mathbf{X}^\top\boldsymbol{\alpha}\right) - \mathbf{y} - \frac{1}{2}\boldsymbol{\alpha}\right) && (5.31) \end{aligned}$$

$$= -\frac{1}{4}\boldsymbol{\alpha}^\top\boldsymbol{\alpha} - \frac{1}{4\lambda}\boldsymbol{\alpha}^\top\mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha} - \boldsymbol{\alpha}^\top\mathbf{y} \tag{5.32}$$

Now, the dual problem is

$$\max_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} -\frac{1}{4}\boldsymbol{\alpha}^\top\boldsymbol{\alpha} - \frac{1}{4\lambda}\boldsymbol{\alpha}^\top\mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha} - \boldsymbol{\alpha}^\top\mathbf{y}$$

Note that this problem is a maximization over a concave problem (similar to a minimization over a convex problem) and we can take the derivative w.r.t $\boldsymbol{\alpha}$ and set it to $\mathbf{0}$:

$$\nabla_{\boldsymbol{\alpha}} g(\boldsymbol{\alpha}) = -\frac{1}{2}\boldsymbol{\alpha} - \frac{1}{2\lambda}\mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha} - \mathbf{y} = \mathbf{0} \implies \boldsymbol{\alpha}^* = -2\lambda(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

The optimal $\mathbf{w}^*$ is therefore given by

$$\mathbf{w}^* = -\frac{1}{2\lambda}\mathbf{X}^\top\boldsymbol{\alpha}^* = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y}$$

Which exactly matches the expression we previously derived for kernel ridge regression! Note that while this solution is dual optimal, it may not be optimal for the primal problem. In order to ensure that it is primal optimal, we need to establish that strong duality holds. In this case the primal problem is convex, so we simply need to ensure that the KKT conditions hold. Since we are not dealing with any inequality conditions here, the only applicable conditions are primal feasibility for the equalities and stationarity. Indeed the primal equality constraints are met, since

$$\begin{aligned}
\mathbf{X}\mathbf{w}^* - \mathbf{y} - \mathbf{z}^* &= -\frac{1}{2\lambda}\mathbf{X}\mathbf{X}^\top\boldsymbol{\alpha}^* - \mathbf{y} - \frac{1}{2}\boldsymbol{\alpha}^* \\
&= -\frac{1}{2\lambda}(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})\boldsymbol{\alpha}^* - \mathbf{y} \\
&= (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}\mathbf{y} - \mathbf{y} \\
&= \mathbf{0}
\end{aligned}$$

We already showed the stationarity conditions are met, when we were solving $g(\boldsymbol{\alpha}) = \min_{\mathbf{w},\mathbf{z}} \mathcal{L}(\mathbf{w}, \mathbf{z}, \boldsymbol{\alpha})$. We conclude that $\mathbf{w}^*$ is indeed the optimal solution to the primal problem.

Dual SVMs

Previously in our investigation of SVMs, we formulated a constrained optimization problem that we can solve to find the optimal parameters for our hyperplane decision boundary. Recall the setup of soft-margin SVMs:

- y_i's: ± 1, representing positive or negative class
- $\mathbf{x}_i$'s: feature vectors in $\mathbb{R}^d$
- ξ_i's: slack variables representing how much an $\mathbf{x}_i$ is allowed to violate the margin
- C: a hyperparameter describing how severely we penalize slack
- The optimization problem for $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, the parameters of the SVM:

$$\begin{aligned}
\min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^n \xi_i \\
\text{s.t.} \quad & y_i(\mathbf{w}^\top\mathbf{x}_i - b) \geq 1 - \xi_i \quad \forall i \\
& \xi_i \geq 0 \quad \forall i
\end{aligned} \tag{5.33}$$

Now, let's investigate the dual of this problem. The primal problem in standard form is

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & (1-\xi_i) - y_i(\mathbf{w}^\top\mathbf{x}_i - b) \leq 0 \quad \forall i \\ & -\xi_i \leq 0 \quad \forall i \end{aligned} \tag{5.34}$$

Let's identify the primal and dual variables for the SVM problem. We will have

- Primal variables $\mathbf{w}$, b, and ξ_i
- Dual variables α_i corresponding to each constraint of the form $y_i(\mathbf{w}^\top\mathbf{x}_i - b) \geq 1 - \xi_i$
- Dual variables β_i corresponding to each constraint of the form $\xi_i \geq 0$

For the purposes of notation, note that we are using $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ in place of $\boldsymbol{\lambda}$, and there are no dual variables corresponding to $\boldsymbol{\nu}$ because there are no equality constraints. The Lagrangian for the SVM problem is

$$\mathcal{L}(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha},\boldsymbol{\beta}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i + \sum_{i=1}^{n}\alpha_i((1-\xi_i) - y_i(\mathbf{w}^\top\mathbf{x}_i - b)) + \sum_{i=1}^{n}\beta_i(-\xi_i) \tag{5.35}$$

$$= \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n}\alpha_i y_i(\mathbf{w}^\top\mathbf{x}_i - b) + \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i \tag{5.36}$$

Thus, the dual is

$$\max_{\boldsymbol{\alpha}\geq\mathbf{0},\boldsymbol{\beta}\geq\mathbf{0}} g(\boldsymbol{\alpha},\boldsymbol{\beta}) \tag{5.37}$$

where

$$g(\boldsymbol{\alpha},\boldsymbol{\beta}) = \min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n}\alpha_i y_i(\mathbf{w}^\top\mathbf{x}_i - b) + \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i \tag{5.38}$$

Let's use the KKT conditions to find the optimal dual variables. Verify that the primal problem is convex in the primal variables. We know that from the stationarity conditions, evaluated at the optimal dual values $\boldsymbol{\alpha}^*$ and $\boldsymbol{\beta}^*$, and the optimal primal values $\mathbf{w}^*, b^*, \xi_i^*$:

$$\frac{\partial\mathcal{L}}{\partial w_i} = \frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial \xi_i} = 0$$

- $\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w}^* - \sum_{i=1}^{n}\alpha_i^* y_i\mathbf{x}_i = \mathbf{0} \implies \mathbf{w}^* = \sum_{i=1}^{n}\alpha_i^* y_i\mathbf{x}_i$. This tells us that $\mathbf{w}^*$ is going to be a weighted combination of the positive-class $\mathbf{x}_i$'s and negative-class $\mathbf{x}_i$'s.
- $\frac{\partial\mathcal{L}}{\partial b} = \sum_{i=1}^{n}\alpha_i^* y_i = 0$. This tells us that the weights α_i^* will be equally distributed among positive- and negative- class training points.
- $\frac{\partial\mathcal{L}}{\partial \xi_i} = C - \alpha_i^* - \beta_i^* = 0 \implies 0 \leq \alpha_i^* \leq C$. This tells us that the weights α_i^* are restricted to being less than the hyperparameter C.

Verify that the other KKT also hold, establishing strong duality. Using these observations, we can eliminate some terms of the dual problem.

$$\mathcal{L}(\mathbf{w},b,\boldsymbol{\xi},\boldsymbol{\alpha}^*,\boldsymbol{\beta}^*) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n}\alpha_i^* y_i(\mathbf{w}^\top\mathbf{x}_i - b) + \sum_{i=1}^{n}\alpha_i^* + \sum_{i=1}^{n}(C - \alpha_i^* - \beta_i^*)\xi_i \tag{5.39}$$

$$= \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^n \alpha_i^* y_i(\mathbf{w}^\top \mathbf{x}_i) + b\underbrace{\sum_{i=1}^n \alpha_i^* y_i}_{=0} + \sum_{i=1}^n \alpha_i^* + \underbrace{\sum_{i=1}^n (C - \alpha_i^* - \beta_i^*)\xi_i}_{=0} \tag{5.40}$$

$$= \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^n \alpha_i^* y_i(\mathbf{w}^\top \mathbf{x}_i) + \sum_{i=1}^n \alpha_i^* \tag{5.41}$$

Since the primal problem is convex, from the KKT conditions we have that the optimal primal variables $\mathbf{w}^*, b^*, \boldsymbol{\xi}^*$ minimize $\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*)$:

$$\begin{aligned} g(\boldsymbol{\alpha}^*, \boldsymbol{\beta}^*) &= \min_{\mathbf{w},b,\boldsymbol{\xi}} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*) && (5.42)\\ &= \mathcal{L}(\mathbf{w}^*, b^*, \boldsymbol{\xi}^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}^*) && (5.43)\\ &= \frac{1}{2}\|\sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i\|^2 - \sum_{i=1}^n \alpha_i^* y_i((\sum_{j=1}^n \alpha_j^* y_j \mathbf{x}_j)^\top \mathbf{x}_i) + \sum_{i=1}^n \alpha_i^* && (5.44)\\ &= \frac{1}{2}\|\sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i\|^2 - \sum_{i=1}^n (\alpha_i^* y_i \mathbf{x}_i{}^\top (\sum_{j=1}^n \alpha_j^* y_j \mathbf{x}_j)) + \sum_{i=1}^n \alpha_i^* && (5.45)\\ &= \boldsymbol{\alpha}^{*\top}\mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^{*\top}\mathbf{Q}\boldsymbol{\alpha}^* && (5.46) \end{aligned}$$

where $Q_{ij} = y_i(\mathbf{x}_i{}^\top \mathbf{x}_j)y_j$ (and $\mathbf{Q} = (\text{diag }\mathbf{y})\mathbf{X}\mathbf{X}^\top(\text{diag }\mathbf{y})$).

Now, we can write the final form of the dual, which is only in terms of $\boldsymbol{\alpha}$ and $\mathbf{X}$ and $\mathbf{y}$ (Note that we have eliminated all references to $\boldsymbol{\beta}$):

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \boldsymbol{\alpha}^\top \mathbf{1} - \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{Q}\boldsymbol{\alpha} \\ \text{s.t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C \quad i = 1, \dots, n \end{aligned} \tag{5.47}$$

Remember to account for the constraints $\sum_{i=1}^n \alpha_i y_i = 0$ and $0 \le \alpha_i \le C$ that arise from the stationarity conditions. After all of this effort, we have managed to turn a minimization problem over the primal variables into a maximization problem over the dual variables. One might ask, why go through the effort to formulate and solve the dual problem instead? For one, the dual is an optimization problem over the number of training points n rather than the number of augmented features d, making it particularity attractive when $n \ll d$. Second, it incorporates the term $\mathbf{X}\mathbf{X}^\top$ which is simply the Gram matrix $\mathbf{K}$ of kernel evaluations among all pairs of training points. We can apply the kernel trick to form this Gram matrix, effectively relying on the the dimensionality of the raw feature space rather than the augmented feature space. These are more or less the exact same justifications for kernel ridge regression.

Geometric intuition

We've formulated the dual SVM problem and used the KKT conditions to formulate an equivalent optimization problem, but what do these dual values α_i even mean? That's a good question!

We know that given optimal primal and dual values, the following KKT conditions are enforced:

- Stationarity

$$C - \alpha_i^* - \beta_i^* = 0$$

- Complementary slackness

$$\alpha_i^* \cdot ((1 - \xi_i^*) - y_i(\mathbf{w}^{*\top}\mathbf{x}_i - b^*)) = 0$$

$$\beta_i^* \cdot \xi_i^* = 0$$

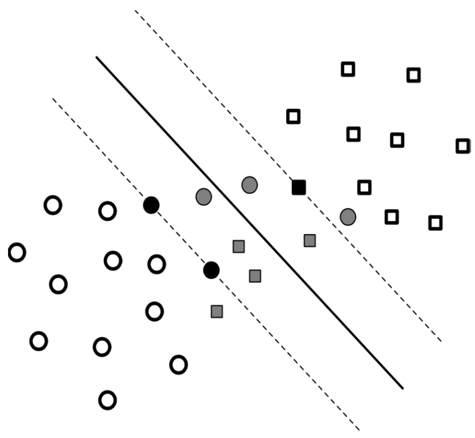
Here are some noteworthy relationships between α_i and the properties of the SVMs:

- Case 1: $\alpha_i^* = 0$. In this case, we know $\beta_i^* = C$, which is nonzero, and therefore $\xi_i^* = 0$. That is, if for point i we have that $\alpha_i^* = 0$ by the dual problem, then we know that there is no slack given to this point. Looking at the other complementary slackness condition, this makes sense because if $\alpha_i^* = 0$, then $y_i(\mathbf{w}^{*\top}\mathbf{x}_i - b^*) - (1 - \xi_i^*)$ may be any value, and if we're minimizing the sum of our ξ_i's, we should have $\xi_i^* = 0$. So, point i lies **on or outside the margin.**

- Case 2: α_i^* is nonzero. If this is the case, then we know $\beta_i^* = C - \alpha_i^* \geq 0$

  – Case 2.1: $\alpha_i^* = C$. If this is the case, then we know $\beta_i^* = 0$, and therefore ξ_i^* may be exactly 0 or nonzero. So, point i lies **on or violates the margin.**

  – Case 2.2: $0 < \alpha_i^* < C$. In this case, then β_i^* is nonzero and $\xi_i^* = 0$. But this is different from Case 1 because with α_i^* nonzero, we can divide by α_i^* in the complementary slackness condition and arrive at the fact that $1 - y_i(\mathbf{w}^{*\top}\mathbf{x}_i - b^*) = 0 \implies y_i(\mathbf{w}^{*\top}\mathbf{x}_i - b^*) = 1$, which means $\mathbf{x}_i$ lies exactly on the margin determined by $\mathbf{w}^*$ and b^*. So, point i lies **on the margin.**



$\alpha_i = 0 \Rightarrow$	$y_i f(x_i) \geq 1$: on or outside the margin
$0 < \alpha_i < C \Rightarrow$	$y_i f(x_i) = 1$: on the margin
$\alpha_i = C \Rightarrow$	$y_i f(x_i) \leq 1$: on or inside the margin
$\alpha_i = 0 \Leftarrow$	$y_i f(x_i) > 1$: outside the margin
$\alpha_i = C \Leftarrow$	$y_i f(x_i) < 1$: inside the margin

Using this information, let's reconstruct the optimal primal values $\mathbf{w}^*, b^*, \xi_i^*$ from the optimal dual values $\boldsymbol{\alpha}^*$:

$$\begin{aligned} \mathbf{w}^* &= \sum_{i=1}^n \alpha_i^* y_i \mathbf{x}_i \\ b^* &= \mathbf{w}^{*\top}\mathbf{x}_i - y_i \qquad \text{if } 0 < \alpha_i^* < C \\ \xi_i^* &= \begin{cases} 1 - y_i(\mathbf{w}^{*\top}\mathbf{x}_i - b^*) \text{ if } \alpha_i^* = C, \\ 0 \text{ otherwise} \end{cases} \end{aligned} \tag{5.48}$$

The principal takeaway is that the optimal $\mathbf{w}^*$ is a linear combination of the training points for which the corresponding dual weight α_i is non-zero. Such points are called **support vectors**, because they determine the optimal $\mathbf{w}^*$. There is a special relationship between the values of α_i and the position of $\mathbf{x}_i$ relative to the margin. All training points that violate the decision boundary have $\alpha_i > 0$ and are thus support vectors, while all training points that strictly do not violate the decision boundary (meaning that they do not lie on the boundary) have $\alpha_i = 0$ and are not support vectors. For training points which lie exactly on the boundary, some may have $\alpha_i > 0$ and some may have $\alpha_i = 0$; only the points that are critical to determining the decision boundary have $\alpha_i > 0$ and are thus support vectors. Intuitively, there are very few support vectors compared to the total number of training points, meaning that the dual vector $\boldsymbol{\alpha}^*$ is *sparse*. This is advantageous when predicting class for a test point:

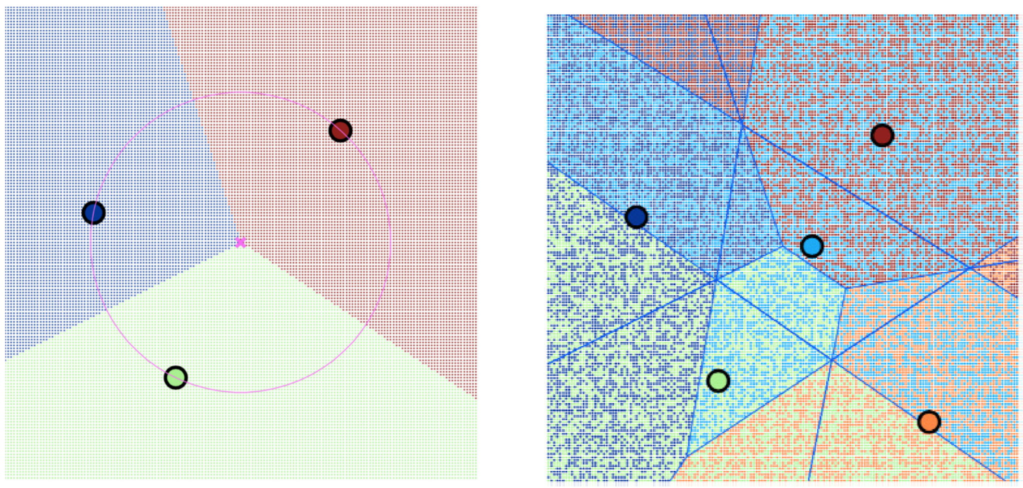
$$\mathbf{w}^{*\top}\phi(\mathbf{x}) + b^* = \sum_{i=1}^n \alpha_i^* y_i \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}) + b^* = \sum_{i=1}^n \alpha_i^* y_i k(\mathbf{x}_i, \mathbf{x}) + b^*$$

We only have to make $m \ll n$ kernel evaluations to predict a test point, where m is the number of support vectors. It should now be clear why the dual SVM problem is so useful: it allows us to use the kernel trick to eliminate dependence on the dimensionality of the argument feature space, while also allowing us to discard most training points because they have dual weight 0.

5.7 Nearest Neighbor Classification

In classification, it is reasonable to conjecture that data points that are sufficiently close to one another should be of the same class. For example, in fruit classification, perturbing a few pixels in an image of a banana should still result in something that looks like a banana. The **k-nearest-neighbors (k-NN)** classifier is based on this observation. Assuming that there is no preprocessing of the training data, the training time for k-NN is effectively $O(1)$. To train this classifier, we simply store our training data for future reference.[1] For this reason, k-NN is sometimes referred to as "lazy learning." The major work of k-NNs in done at testing time: to predict on a test data point $\mathbf{z}$, we compute the k closest training data points to $\mathbf{z}$, where "closeness" can be quantified in some distance function such as Euclidean distance — these are the *k nearest neighbors* to $\mathbf{z}$. We then find the most common class y among these k neighbors and classify $\mathbf{z}$ as y (that is, we perform a majority vote). For binary classification, k is usually chosen to be odd so we can break ties cleanly. Note that k-NN can also be applied to regression tasks — in that case k-NN would return the average label of the k nearest points.

[1]Sometimes we store the data in a specialized structure called a *k-d tree*. This data structure is out of scope for this course, but it usually allows for faster (average-case $O(\log n)$) nearest neighbors queries.

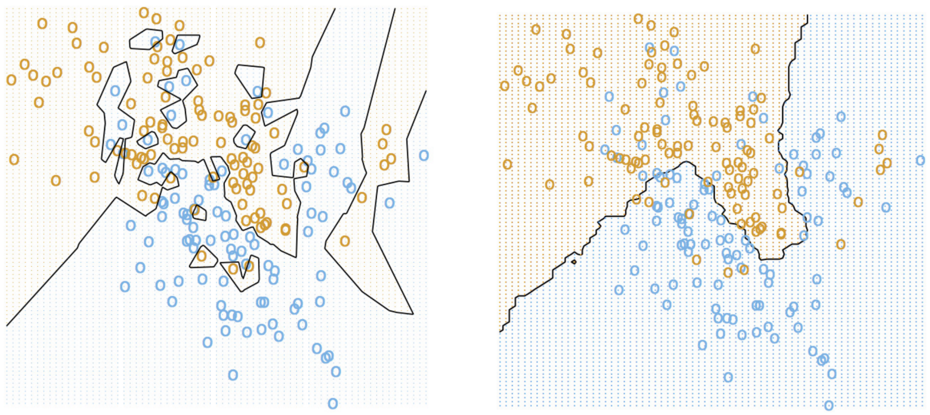


(a) 1-NN on 3 classes (b) 2-NN on 5 classes

Figure 5.14: **Voronoi diagram** for k-NN. Points in a region shaded a certain color will be classified as that color. Test points in a region shaded with a combination of 2 colors have those colors as their 2 nearest neighbors.

Choosing k

Nearest neighbors can produce very complex decision functions, and its behavior is highly dependent on the choice of k.



(a) $k = 1$ (b) $k = 15$

Figure 5.15: **Voronoi diagram** for $k = 1$ vs. $k = 15$. Figure from Introduction to Statistical Learning.

Choosing $k = 1$, we achieve an optimal training error of 0 because each training point will classify as itself, thus achieving 100% accuracy on itself. However, $k = 1$ overfits to the training data, and is a terrible choice in the context of the bias-variance tradeoff. Increasing k leads to an increase in training error, but a decrease in testing error and achieves better generalization. At one point, if k becomes too large, the algorithm will underfit the training data, and suffer from huge bias. In general, in order to select k we use cross-validation.

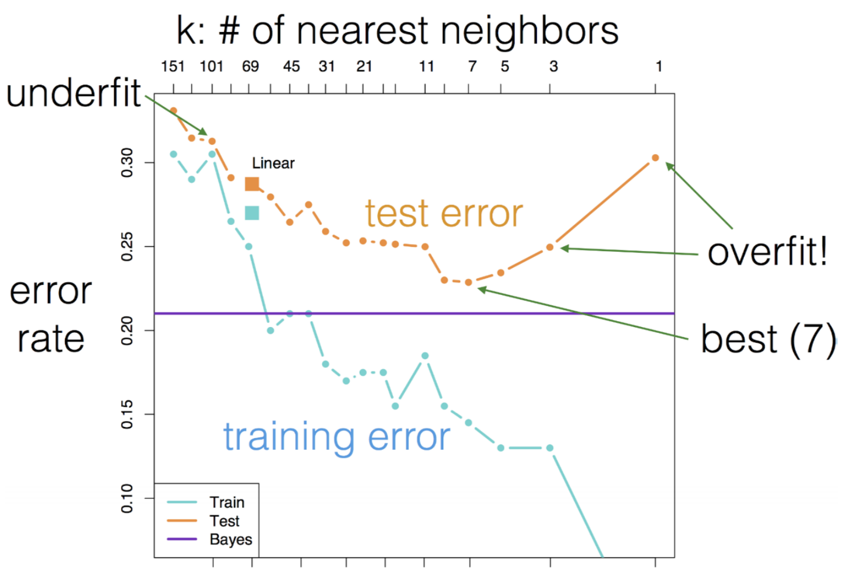


Figure 5.16: Training and Testing error as a function of k. Figure from Introduction to Statistical Learning.

Bias-Variance Analysis

Let's justify this reasoning formally for k-NN applied to regression tasks. Suppose we are given a training dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where the labels y_i are real valued scalars. We model our hypothesis $h(\mathbf{z})$ as

$$h(\mathbf{z}) = \frac{1}{k}\sum_{i=1}^{n} N(\mathbf{x}_i, \mathbf{z}, k)$$

where the function N is defined as

$$N(\mathbf{x}_i, \mathbf{z}, k) = \begin{cases} y_i & \text{if } \mathbf{x}_i \text{ is one of the } k \text{ closest points to } \mathbf{z} \\ 0 & o.w. \end{cases}$$

Suppose also we assume our labels $y_i = f(\mathbf{x}_i) + \epsilon$, where ϵ is the noise that comes from $\mathcal{N}(0, \sigma^2)$ and f is the true function. Without loss of generality, let $\mathbf{x}_1 \ldots \mathbf{x}_k$ be the k closest points. Let's first derive the bias^2 of our model for the given dataset $\mathcal{D}$.

$$\begin{aligned}
\big(\mathbb{E}[h(\mathbf{z})] - f(\mathbf{z})\big)^2 &= \left(\mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{n} N(\mathbf{x}_i, \mathbf{z}, k)\right] - f(\mathbf{z})\right)^2 = \left(\mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{k} y_i\right] - f(\mathbf{z})\right)^2 \\
&= \left(\frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[y_i] - f(\mathbf{z})\right)^2 = \left(\frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[f(\mathbf{x}_i) + \epsilon] - f(\mathbf{z})\right)^2 \\
&= \left(\frac{1}{k}\sum_{i=1}^{k} f(\mathbf{x}_i) - f(\mathbf{z})\right)^2
\end{aligned}$$

When $k \longrightarrow \infty$, then $\frac{1}{k}\sum_{i=1}^{k} f(\mathbf{x}_i)$ goes to the average label for $\mathbf{x}$. When $k = 1$, then the bias is simply $f(\mathbf{x}_1) - f(\mathbf{z})$. Assuming $\mathbf{x}_1$ is close enough to $f(\mathbf{z})$, the bias would likely be small when

$k = 1$ since it's likely to share a similar label. Meanwhile, when $k \longrightarrow \infty$, the bias doesn't depend on the training points at all which like will restrict it to be higher.

Now, let's derive the variance of our model.

$$\begin{aligned}\text{Var}[h(\mathbf{z})] = \text{Var}\left[\frac{1}{k}\sum_{i=1}^{k} y_i\right] &= \frac{1}{k^2}\sum_{i=1}^{k}\text{Var}[f(\mathbf{x}_i)+\epsilon] \\ &= \frac{1}{k^2}\sum_{i=1}^{k}\text{Var}[\epsilon] \\ &= \frac{1}{k^2}\sum_{i=1}^{k}\sigma^2 = \frac{1}{k^2}k\sigma^2 = \frac{\sigma^2}{k}\end{aligned}$$

The variance goes to 0 when $k \longrightarrow \infty$, and is maximized at $k = 1$.

Properties

Computational complexity: We require $O(n)$ space to store a training set of size n. There is no runtime cost during training if we do not use specialized data structures to store the data. However, predictions take $O(n)$ time, which is costly. There has been research into approximate nearest neighbors (ANN) procedures that quickly find an approximation for the nearest neighbor - some common ANN methods are *Locality-Sensitive Hashing* and algorithms that perform dimensionality reduction via randomized (Johnson-Lindenstrauss) distance-preserving projections.[2]

Flexibility: When $k > 1$, k-NN can be modified to output predicted probabilities $P(Y|X)$ by defining $P(Y|X)$ as the proportion of nearest neighbors to X in the training set that have class Y. k-NN can also be adapted for regression — instead of taking the majority vote, take the average of the y values for the nearest neighbors. k-NN can learn very complicated, non-linear decision boundaries.

Non-parametric: k-NN is a **non-parametric method**, which means that the number of parameters in the model grows with n, the number of training points. This is as opposed to parametric methods, for which the number of parameters is independent of n. Some examples of parametric models include linear regression, LDA, and neural networks.

Behavior in high dimensions: k-NN does not behave well in high dimensions. As the dimension increases, data points drift farther apart, so even the nearest neighbor to a point will tend to be very far away.

Theoretical properties: 1-NN has impressive theoretical guarantees for such a simple method. Cover and Hart, 1967 prove that as the number of training samples n approaches infinity, the expected prediction error for 1-NN is upper bounded by $2\epsilon^*$, where ϵ^* is the Bayes (optimal) error. Fix and Hodges, 1951 prove that as n and k approach infinity and if $\frac{k}{n} \rightarrow 0$, then the k nearest neighbor error approaches the Bayes error.

Curse of Dimensionality

To understand why k-NN does not perform well in high-dimensional space, we first need to understand the properties of metric spaces. In high-dimensional spaces, much of our low-dimensional

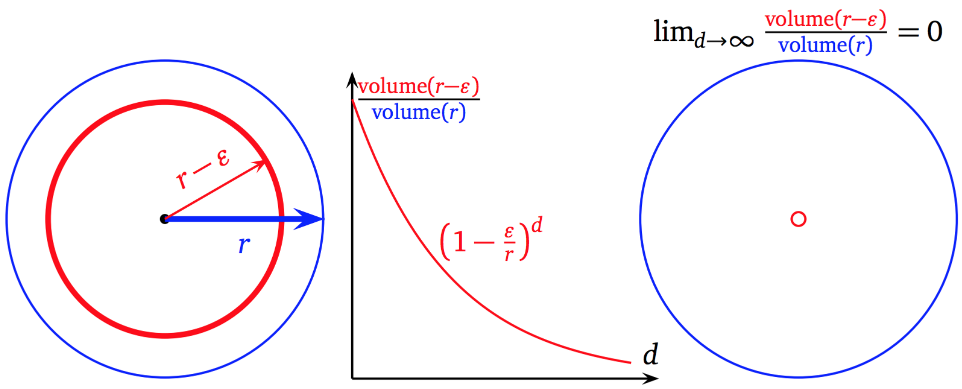
[2]ANN methods are beyond the scope of this course, but are useful in real applications.

intuition breaks down. Here is one classical example. Consider a ball in $\mathbb{R}^d$ centered at the origin with radius r, and suppose we have another ball of radius $r - \epsilon$ centered at the origin. In low dimensions, we can visually see that much of the volume of the outer ball is also in the inner ball.

In general, the volume of the outer ball is proportional to r^d, while the volume of the inner ball is proportional to $(r-\epsilon)^d$. Thus the ratio of the volume of the inner ball to that of the outer ball is

$$\frac{(r-\epsilon)^d}{r^d} = \left(1 - \frac{\epsilon}{r}\right)^d \approx e^{-\epsilon d/r} \xrightarrow[d\to\infty]{} 0$$

Hence as d gets large, most of the volume of the outer ball is concentrated in the annular region $\{x : r - \epsilon < x < r\}$ instead of the inner ball.


$r-\varepsilon$
r
$\frac{\text{volume}(r-\varepsilon)}{\text{volume}(r)}$
$\left(1-\frac{\varepsilon}{r}\right)^d$
d
$\lim_{d\to\infty} \frac{\text{volume}(r-\varepsilon)}{\text{volume}(r)} = 0$

High dimensions also make Gaussian distributions behave counter-intuitively. Suppose $X \sim \mathcal{N}(0, \sigma^2 I)$. If X_i are the components of X and R is the distance from X to the origin, then $R^2 = \sum_{i=1}^d X_i^2$. We have $E(R^2) = d\sigma^2$, so in expectation a random Gaussian will actually be reasonably far from the origin. If $\sigma = 1$, then R^2 is distributed chi-squared with d degrees of freedom. One can show that in high dimensions, with high probability $1 - O(e^{-d^\epsilon})$, this multivariate Gaussian will lie within the annular region $\{X : |R^2 - E(R^2)| \le d^{1/2+\epsilon}\}$ where $E(R^2) = d\sigma^2$ (one possible approach is to note that as $d \to \infty$, the chi-squared approaches a Gaussian by the CLT, and use a Chernoff bound to show exponential decay). This phenomenon is known as *concentration of measure.* Without resorting to more complicated inequalities, we can show a simple, weaker result:

Theorem: If $X_i \sim \mathcal{N}(0, \sigma^2)$, $i = 1, ..., d$ are independent and $R^2 = \sum_{i=1}^d X_i^2$, then for every $\epsilon > 0$, the following holds:

$$\lim_{d\to\infty} P(|R^2 - E(R^2)| \ge d^{\frac{1}{2}+\epsilon}) = 0$$

Thus in the limit, the squared radius is concentrated about its mean.

Proof. From the formula for the variance of a chi-squared distribution, we see that $\text{Var}(R^2) = 2d\sigma^4$. Applying a Chebyshev bound yields

$$P(|R^2 - E(R^2)| \ge d^{\frac{1}{2}+\epsilon}) \le \frac{2d\sigma^4}{d^{1+2\epsilon}} \xrightarrow[d\to\infty]{} 0$$

□

Thus a random Gaussian will lie within a thin annular region away from the origin in high dimensions with high probability, even though the mode of the Gaussian bell curve is at the origin. This illustrates the phenomenon in high dimensions where random data is spread very far apart. The k-NN classifier was conceived on the principle that nearby points should be of the same class - however, in high dimensions, even the nearest neighbors that we have to a random test point will tend to be far away, so this principle is no longer useful.

Improving k-NN

There are two main ways to improve k-NN and overcome the shortcomings we have discussed.

1. Obtain more training data.
2. Reduce the dimensionality of the features and/or pick better features. Consider other choices of distance function.

One example of reducing the dimensionality in image space is to lower the resolution of the image — while this is throwing some of the original pixel features away, we may still be able to get the same or better performance with a nearest neighbors method.

We can also modify the distance function. For example, we have a whole family of **Minkowski distances** that are induced by the L^p norms:

$$D_p(\mathbf{x}, \mathbf{z}) = \left(\sum_{i=1}^{d} |x_i - z_i|^p \right)^{\frac{1}{p}}$$

Without preprocessing the data, 1-NN with the L^3 distance outperforms 1-NN with L^2 on MNIST.

We can also use kernels to compute distances in a different feature space. For example, if k is a kernel with associated feature map Φ and we want to compute the Euclidean distance from $\Phi(x)$ to $\Phi(z)$, then we have

$$\begin{aligned}\|\Phi(\mathbf{x}) - \Phi(\mathbf{z})\|_2^2 &= \Phi(\mathbf{x})^\top \Phi(\mathbf{x}) - 2\Phi(\mathbf{x})^\top \Phi(\mathbf{z}) + \Phi(\mathbf{z})^\top \Phi(\mathbf{z}) \\ &= k(\mathbf{x}, \mathbf{x}) - 2k(\mathbf{x}, \mathbf{z}) + k(\mathbf{z}, \mathbf{z})\end{aligned}$$

Thus if we define $D(\mathbf{x}, \mathbf{z}) = \sqrt{k(\mathbf{x}, \mathbf{x}) - 2k(\mathbf{x}, \mathbf{z}) + k(\mathbf{z}, \mathbf{z})}$, then we can perform Euclidean nearest neighbors in Φ-space without explicitly representing Φ by using the kernelized distance function D.

Chapter 6

Clustering

In the problem of **clustering**, we are given a dataset comprised only of input features without labels. We wish to assign to each data point a discrete label indicating which "cluster" it belongs to, in such a way that the resulting cluster assignment "fits" the data. We are given flexibility to choose our notion of goodness of fit for cluster assignments.

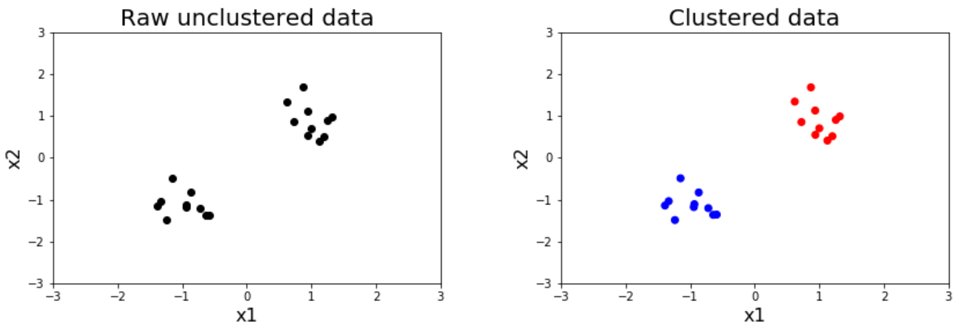


Figure 6.1: Left: unclustered raw data; Right: clustered data

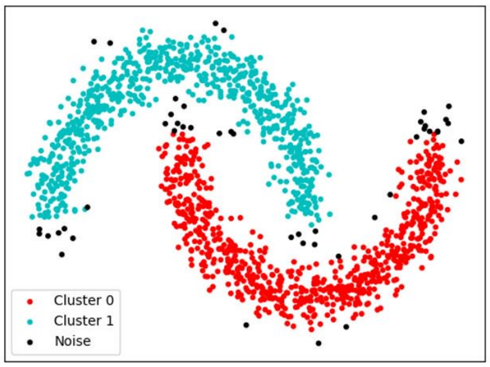


Figure 6.2: A nonspherical clustering assignment. Possible outliers are shown in black.[1]

In our discussion of LDA and QDA, we assumed that we had data which was conditionally Gaussian

[1]https://www.imperva.com/blog/2017/07/clustering-and-dimensionality-reduction-understanding-the-magic-behind-machine-learning/

given a discrete class label. When we observed a data point, we observed both its input features and its class label. These are **supervised** learning methods, which deal with prediction of observed outputs from observed inputs. Clustering is an example of **unsupervised learning**, where we are not given labels and desire to infer something about the underlying structure of the data. Another example of unsupervised learning is dimensionality reduction, where we desire to learn important features from the data.

Clustering is most often used in exploratory data visualization, as it allows us to see the different groups of similar data points within the data. Combined with domain knowledge, these clusters can have a physical interpretation - for example, different clusters can represent different species of plant in the biological setting, or types of consumers in a business setting. If desired, these clusters can be used as pre-processing to make the data more compact. Clustering is also used for outlier detection, as in Figure 6.2: data points that do not seem to belong in their assigned cluster may be flagged as outliers.

In order to create an algorithm for clustering, we first must determine what makes a good clustering assignment. Here are some possible desired properties:

1. High intra-cluster similarity - points within a given cluster are very similar.
2. Low inter-cluster similarity - points in different clusters are not very similar.

Of course, this depends on our notion of similarity. For now, we will say that points in $\mathbb{R}^d$ are similar if their L^2 distance is small, and dissimilar otherwise. A generalization of this notion is provided in the appendix.

6.1 K-means Clustering

Let X denote the set of N data points $\mathbf{x}_i \in \mathbb{R}^d$. A *cluster assignment* is a partition $C_1, ..., C_K \subseteq X$ such that the sets C_k are disjoint and $X = C_1 \cup \cdots \cup C_K$. A data point $\mathbf{x} \in X$ is said to belong to cluster k if it is in C_k.

One approach to the clustering problem is to represent each cluster C_k by a single point $\mathbf{c}_k \in \mathbb{R}^d$ in the input space - this is called the **centroid** approach. K-means is an example of centroid-based clustering where we choose centroids and a cluster assignment such that the total distance of each point to its assigned centroid is minimized. In this regard, K-means optimizes for high intra-cluster similarity, but the clusters do not necessarily need to be far apart, so we may also have high inter-cluster similarity.

Formally, K-means solves the following problem:

$$\underset{\{C_k\}_{k=1}^K, \{\mathbf{c}_k\}_{k=1}^K : X = C_1 \cup \cdots \cup C_K}{\arg\min} \sum_{k=1}^K \sum_{\mathbf{x} \in C_k} \|\mathbf{x} - \mathbf{c}_k\|^2$$

It has been shown that this problem is NP hard, so solving it exactly is intractable. However, we can come up with a simple algorithm to compute a candidate solution. If we knew the cluster assignment $C_1, ..., C_K$, then we would only need to determine the centroid locations. Since the choice of centroid location $\mathbf{c}_i$ does not affect the distances of points in C_j to $\mathbf{c}_j$ for $i \neq j$, we can consider each cluster separately and choose the centroid that minimizes the sum of squared distances to points in that cluster. The centroid we compute, $\hat{\mathbf{c}}_k$, is

$$\hat{\mathbf{c}}_k = \underset{\mathbf{c}_k}{\arg\min} \sum_{\mathbf{x} \in C_k} \|\mathbf{x} - c_k\|^2$$

But this is simply the mean of the data in C_k, that is,

$$\hat{\mathbf{c}}_k = \frac{1}{|C_k|} \sum_{\mathbf{x} \in C_k} \mathbf{x}$$

Similarly, if we knew the centroids $\mathbf{c}_k$, in order to choose the cluster assignment $C_1, ..., C_K$ that minimizes the sum of squared distances to the centroids, we simply assign each data point $\mathbf{x}$ to the cluster represented by its closest centroid, that is, we assign $\mathbf{x}$ to

$$\arg\min_k \|\mathbf{x} - \mathbf{c}_k\|^2$$

Now we can perform alternating minimization - on each iteration of our algorithm, we update the clusters using the current centroids, and then update the centroids using the new clusters. This algorithm is sometimes called **Lloyd's Algorithm**.

Algorithm 11: K-means Algorithm

```
Initialize $\mathbf{c}_k$, $k = 1, ..., K$
while K-means objective has not converged do
    Update partition $C_1 \cup \cdots \cup C_K$ given the $\mathbf{c}_k$ by assigning each $x \in X$ to the cluster represented by its nearest centroid
    Update centroids $\mathbf{c}_k$ given $C_1 \cup \cdots \cup C_K$ as $\mathbf{c}_k = \frac{1}{|C_k|} \sum_{\mathbf{x} \in C_k} \mathbf{x}$
```

This algorithm will always converge to some value. To show this, note the following facts:

1. There are only finitely many (say, M) possible partition/centroid pairs that can be produced by the algorithm. This is true since each partition chosen at some iteration in the algorithm completely determines the subsequent centroid assignment in that iteration.
2. Each update of the cluster assignment and centroids does not increase the value of the objective. This is true since each of these updates is a minimization of the objective which we solve exactly.

If the value of the objective has not converged after M iterations, then we have cycled through all the possible partition/centroid pairs attainable by the algorithm. On the next iteration, we would obtain a partition and centroid assignment that we have already seen, say on iteration $t \leq M$. But this means that the value of the objective at time $M + 1$ is the same as at time t, and because the value of the objective function never increases throughout the algorithm, the value is the same as at time M, so we have converged.

In practice, it is common to run the K-means algorithm multiple times with different initialization points, and the cluster corresponding to the minimum objective value is chosen. There are also ways to choose a smarter initialization than a random seed, which can improve the quality of the local optimum found by the algorithm.[2] It should be emphasized that no efficient algorithm for solving the K-means optimization is guaranteed to give a good cluster assignment, as the problem is NP hard and there are local optima.

Choosing the number of clusters k is similar to choosing the number of principal components for PCA - we can compute the value of the objective for multiple values of k and find the "elbow" in the curve.

[2] For example, K-means++.

We have noted that the main algorithm for solving K-means does not have to produce a good solution. Let us step back and consider some shortcomings of the K-means objective function itself:

1. There is no likelihood attached to K-means, which makes it harder to understand what assumptions we are making on the data.
2. Each feature is treated equally, so the clusters produced by K-means will look spherical. We can also infer this by looking at the sum of squares in the objective function, which we have seen to be related to spherical Gaussians.
3. Each cluster assignment in the optimization is a **hard assignment** - each point belongs in exactly one cluster. A **soft assignment** would assign each point to a distribution over the clusters, which can encode not only which cluster a point belongs to, but also how far it was from the other clusters.

Soft K-means

We can introduce soft assignments to our algorithm easily using the familiar softmax function. Recall that if $\mathbf{z} \in \mathbb{R}^d$, then the softmax function σ is defined as

$$\sigma(\mathbf{z})_j = \frac{e^{\mathbf{z}_j}}{\sum_{k=1}^d e^{\mathbf{z}_k}}$$

To compute the cluster assignment of a data point $\mathbf{x}_i$ in K-means, we computed

$$\arg\min_k \|\mathbf{x}_i - \mathbf{c}_k\|^2 = \arg\max_k -\|\mathbf{x}_i - \mathbf{c}_k\|^2$$

In soft K-means, we instead compute a soft assignment $r_i(k)$, $k = 1, ..., K$ where $\sum_k r_i(k) = 1$ as the softmax of the vector of $\mathbf{z} := -\beta\|\mathbf{x}_i - \mathbf{c}_k\|^2$, $k = 1, ..., K$:

$$r_i(k) = \sigma(\mathbf{z})_k$$

Here, β is a tunable parameter indicating the level of "softness" desired.

Once we have computed the soft assignments, we may use them to determine the centroids by using a weighted average. Before, we defined the new centroids as

$$\hat{\mathbf{c}}_k = \arg\min_{\mathbf{c}_k} \sum_{\mathbf{x} \in C_k} \|\mathbf{x} - \mathbf{c}_k\|^2$$

Now, we apply the $r_i(k)$ as weights for the minimization:

$$\hat{\mathbf{c}}_k = \arg\min_{\mathbf{c}_k} \sum_{i=1}^N r_i(k)\|\mathbf{x}_i - \mathbf{c}_k\|^2 = \frac{\sum_{i=1}^N r_i(k)\mathbf{x}_i}{\sum_{i=1}^N r_i(k)} \tag{6.1}$$

This is now a weighted average of the $\mathbf{x}_i$ - the weights reflect how much we believe each data point belongs to a particular cluster, and because we are using this information, our algorithm should not jump around between clusters, resulting in better convergence speed.

There are still a few issues with soft K-means. One is the choice of β - it is not so clear how to set this hyperparameter. Another issue is that our clusters are still spherical, since we are still weighting all features the same (note that we have weighted each *data point* differently with soft K-means). To solve these issues, we will use a fully probabilistic model.

6.2 Mixture of Gaussians

Suppose $\boldsymbol{\mu}_k \in \mathbb{R}^d, \boldsymbol{\Sigma}_k \in \mathbb{R}^{d\times d}$ are fixed parameters for $k = 1, ..., K$. Consider the following experiment: draw a value z from some distribution on the set of indices $\{1, ..., K\}$, and then draw $\mathbf{x} \in \mathbb{R}^d$ from the Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z)$. We can interpret $\mathbf{x}$ as belonging to cluster z. This model is called **Mixture of Gaussians (MoG)**, also known as a **Gaussian mixture model**.

If we have fit a MoG model to data (ie. we have determined values for $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, and the prior on z), then to perform clustering, we can use Bayes' rule to determine the posterior $P(z = k|\mathbf{x})$ and assign $\mathbf{x}$ to the cluster k that maximizes this quantity. In fact, this is exactly our decision rule with QDA using a prior - the difference is that QDA, a supervised method, is given labels to fit the mixture model, while in the unsupervised clustering setting we must fit the mixture model without the aid of labels. When $\boldsymbol{\Sigma}_k$ are not multiples of the identity, we can obtain non-spherical clusters, which was not possible with K-means.

MoG is an example of a **latent variable model**. A latent variable model is a probabilistic model in which some variables can be directly observed or measured, while other latent (hidden) variables cannot be observed directly; rather, we observe them indirectly through their influence on the observed variables. When we try to fit a MoG model to data, we only observe the data $\mathbf{x}_i$, which we presume to have been generated based on the latent variable z_i, the cluster assignment. Latent variable models are modular and can be used to model complex dependencies in a probabilistic model - however, the added flexibility can lead to difficulty in learning its parameters.

To illustrate this, we will examine the likelihood function for MoG. Suppose $\mathbf{x}_i$ has distribution $p(\mathbf{x}_i; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a set of all $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, p(z_i = k)$. The likelihood for the single data point $\mathbf{x}_i$ is

$$\begin{aligned}
\mathcal{L}_i(\boldsymbol{\theta}; \mathbf{x}_i) &= p(\mathbf{x}_i; \boldsymbol{\theta}) \\
&= \sum_{k=1}^{K} p(\mathbf{x}_i, z_i = k; \boldsymbol{\theta}) \\
&= \sum_{k=1}^{K} p(\mathbf{x}_i | z_i = k; \boldsymbol{\theta}) p(z_i = k; \boldsymbol{\theta})
\end{aligned}$$

Over all data points, the likelihood is

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}; \mathbf{x}) &= \prod_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\theta}; \mathbf{x}_i) \\
&= \prod_{i=1}^{N} \sum_{k=1}^{K} p(\mathbf{x}_i | z_i = k; \boldsymbol{\theta}) p(z_i = k; \boldsymbol{\theta})
\end{aligned}$$

Hence the log likelihood $\ell(\boldsymbol{\theta}; \mathbf{x})$ is given by

$$\ell(\boldsymbol{\theta}; \mathbf{x}) = \sum_{i=1}^{N} \log \left(\sum_{k=1}^{K} p(\mathbf{x}_i | z_i = k; \boldsymbol{\theta}) p(z_i = k; \boldsymbol{\theta}) \right) \tag{6.2}$$

When we perform QDA, we know the z_i are known, deterministic quantities and thus the likelihood (6.2) reduces to

$$\ell(\boldsymbol{\theta}; \mathbf{X}) = \sum_{i=1}^{N} \log p(\mathbf{x}_i | z_i; \boldsymbol{\theta})$$

Maximizing this is equivalent to fitting the individual class-conditional Gaussians via maximum likelihood, which is consistent with how we have described QDA in the past. When we fit the MoG without knowledge of the latent variables, the parameters $\boldsymbol{\theta}$ in (6.2) are now coupled together inside the log, which complicates the likelihood. While it is still possible to find the MLE by working out the gradient and using our descent methods, there is an alternative approach called Expectation-Maximization (EM), which takes advantage of the latent variable structure.

6.3 Expectation Maximization (EM) Algorithm

The **Expectation-Maximization (EM) Algorithm** is used to compute the MLE for latent variable models, such as MoG. First recall that soft K-means consisted of the following two alternating steps:

1. For each data point, compute a soft assignment $r_i(k)$ to the clusters - that is, a probability distribution over clusters. The soft assignment is obtained by using a softmax.
2. Update the centroids in an optimal way given the soft assignments computed in the first step. The resulting updates are a weighted average of the data points.

One could rephrase these updates as:

1. Soft imputation of the data - fill in the missing data ("impute") with a probability distribution over all its possible values ("soft").
2. Parameter updates given the imputed data

The EM algorithm alternates between these two steps in the same way as soft K-means, but the updates for each step are performed in a principled way to maximize certain "components" of the log likelihood (we will make this precise later). We will derive the following updates for EM when computing the MLE for MoG:

1. Using the current parameter estimates, estimate $p(z_i|\mathbf{x}_i;\boldsymbol{\theta})$. That is, perform soft imputation of the latent cluster variable.
2. Estimate the parameters via MLE, using the estimates of $p(z_i|\mathbf{x}_i;\boldsymbol{\theta})$ to make the computation tractable.

Both soft K-means and EM estimate $p(z_i|\mathbf{x}_i;\boldsymbol{\theta})$, though EM will do so in a more principled way. In the second alternating step, we will see that the EM update of the mean is exactly the same as the soft K-means centroid update. However, EM will also update the covariance estimates, which captures ellipsoidal structure in the data.

To derive the EM algorithm, it will be helpful to introduce the notion of the **complete log likelihood**, which we define as

$$L_c(\mathbf{x}_i, z_i;\boldsymbol{\theta}) := \log p(\mathbf{x}_i, z_i;\boldsymbol{\theta})$$

If we assumed z_i to be known, this would be the log likelihood of the data. In practice, we do not know the values of z_i, but if we are given a distribution $q(z_i|\mathbf{x}_i)$ over the latents z_i, we can marginalize over the possible values of z_i by taking the expectation

$$\mathbb{E}_q(L_c(\mathbf{x}_i, z_i;\boldsymbol{\theta})) = \sum_{k=1}^{K} L_c(\mathbf{x}_i, z_i = k;\boldsymbol{\theta})q(z_i = k|\mathbf{x}_i)$$

We call this the **expected complete log likelihood**. The distribution q is an estimate of the true conditional distribution $p(z_i|\mathbf{x}_i;\boldsymbol{\theta})$; in the EM algorithm, we will alternate between updating our q distribution to better estimate $p(z_i\mathbf{x}_i;\boldsymbol{\theta})$ and maximizing the expected complete log likelihood in place of the true likelihood function.

We are now in a position to derive the algorithm. We will need a well-known result called **Jensen's Inequality**:

Theorem 3. *If X is a random variable and f is convex, then $f(\mathbb{E}(X)) \leq \mathbb{E}(f(X))$.*

If f is concave, then using the fact $-f$ is convex immediately yields the conclusion $\mathbb{E}(f(X)) \leq f(\mathbb{E}(X))$. In particular, since log is concave, we have $\mathbb{E}(\log(X)) \leq \log\big(\mathbb{E}(X)\big)$.

Now we can derive the EM algorithm. Suppose $\mathbf{x}_i$ are random variables depending on z_i, and $\boldsymbol{\theta}$ are the parameters of interest. Given any conditional distribution over the latents $q(z_i = k|\mathbf{x}_i)$, the log likelihood for the i-th data point is

$$\begin{aligned}
\ell_i(\boldsymbol{\theta};\mathbf{x}_i) &= \log p(\mathbf{x}_i;\boldsymbol{\theta}) \\
&= \log \sum_{k=1}^{K} p(\mathbf{x}_i, z_i = k;\boldsymbol{\theta}) \\
&= \log \sum_{k=1}^{K} \frac{q(z_i = k|\mathbf{x}_i)p(\mathbf{x}_i, z_i = k;\boldsymbol{\theta})}{q(z_i = k|\mathbf{x}_i)} \\
&= \log \mathbb{E}_q\left[\frac{p(\mathbf{x}_i, z_i;\boldsymbol{\theta})}{q(z_i|\mathbf{x}_i)}\right] \\
&\geq \mathbb{E}_q\left[\log \frac{p(\mathbf{x}_i, z_i;\boldsymbol{\theta})}{q(z_i|\mathbf{x}_i)}\right] \qquad \text{(Jensen)} \\
&= \sum_{k=1}^{K} q(z_i = k|\mathbf{x}_i)\log[\frac{p(\mathbf{x}_i, z_i = k;\boldsymbol{\theta})}{q(z_i = k|\mathbf{x}_i)}] \\
&= -\sum_{k=1}^{K} q(z_i = k|\mathbf{x}_i)\log[q(z_i = k|\mathbf{x}_i)] + \sum_{k=1}^{K} q(z_i = k|\mathbf{x}_i)\log p(\mathbf{x}_i, z_i = k;\boldsymbol{\theta}) \\
&=: F_i(q,\boldsymbol{\theta})
\end{aligned}$$

We will define

$$H(q(z_i|\mathbf{x}_i)) := -\sum_{k=1}^{K} q(z_i = k|\mathbf{x}_i)\log[q(z_i = k|\mathbf{x}_i)]$$

and

$$L_c(\mathbf{x}_i, z_i;\boldsymbol{\theta}) := \log p(\mathbf{x}_i, z_i;\boldsymbol{\theta})$$

so that the above lower bound can be written as

$$F_i(q,\boldsymbol{\theta}) = H(q(z_i|\mathbf{x}_i)) + \mathbb{E}_q(L_c(\mathbf{x}_i, z_i;\boldsymbol{\theta}))$$

The first term $H(q(z_i|\mathbf{x}_i))$ has an information-theoretic interpretation - it is the **entropy** of the distribution $q(z_i|\mathbf{x}_i)$, a non-negative quantity that measures the amount of disorder encoded in the distribution. As mentioned earlier, the term $L_c(\mathbf{x}_i, z_i;\boldsymbol{\theta})$ is called the **complete log likelihood** of $\mathbf{x}_i$ - it will be easier to optimize $\mathbb{E}_q(L_c(\mathbf{x}_i, z_i;\boldsymbol{\theta}))$ than the original log likelihood, since we will not need to deal with the "marginalization problem" in the original log likelihood from Equation 6.2.

Since the log likelihood of the full data is the sum of the individual log likelihoods, we can take sums and find

$$\ell(\boldsymbol{\theta}; \mathbf{X}) \geq \sum_{i=1}^{N} F_i(q, \boldsymbol{\theta}) = H(q(\mathbf{z}|\mathbf{X})) + \mathbb{E}_q(L_c(\mathbf{X}, \mathbf{z}; \boldsymbol{\theta})) =: F(q, \boldsymbol{\theta}) \tag{6.3}$$

Here, $\mathbf{X}$ denotes the full dataset and $\mathbf{z}$ denotes the length N vector of latent variables. It is easy to check that if $q(z_i|\mathbf{x}_i) = p(z_i|\mathbf{x}_i; \boldsymbol{\theta})$ for all i, then the inequality (6.3) is tight (set $q(z_i|\mathbf{x}_i) = p(\mathbf{x}_i, z_i; \theta)$ in the application of Jensen's inequality and observe both sides of the inequality are equal). Thus it makes sense to perform an alternating maximization scheme, where we iteratively update $q(z_i|\mathbf{x}_i)$ to $p(z_i|\mathbf{x}_i; \boldsymbol{\theta})$ to make the inequality tight and then maximize over $\boldsymbol{\theta}$. Formally, the algorithm is as follows:

1. Initialize $\boldsymbol{\theta}^0$
2. Expectation (E) step (soft imputation): set $q^{t+1} = \arg\max_q F(q, \boldsymbol{\theta}^t)$, that is,

   $$q^{t+1}(z_i = k|\mathbf{x}_i) := p(z_i = k|\mathbf{x}_i; \boldsymbol{\theta}^t)$$

   This value of q is used to compute $\mathbb{E}_q(L_c(\mathbf{X}, \mathbf{z}; \boldsymbol{\theta}^t))$.
3. Maximization (M) step (parameter estimation): set

   $$\boldsymbol{\theta}^{t+1} = \arg\max_{\boldsymbol{\theta}} F(q^{t+1}, \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{q^{t+1}}(L_c(\mathbf{X}, \mathbf{z}; \boldsymbol{\theta}))$$
4. Repeat steps 2, 3 until convergence

A few remarks: when we maximize over q in the E step, there are nK values to be updated - one value of $q(z_i = k|\mathbf{x}_i)$ for every data index i and latent index k. After the E step, q^{t+1} is fixed and does not depend on $\boldsymbol{\theta}$, so the entropy term does not depend on $\boldsymbol{\theta}$ and maximizing $F(q^{t+1}, \boldsymbol{\theta})$ in the subsequent M step amounts to maximizing the expected complete log likelihood. The E step is what we have previously described as soft imputation of the latents: we fill in values for the hidden variables z_i by determining a conditional distribution $q(z_i|\mathbf{x}_i)$. The M step assumes the E step has done a reasonable job at imputing the data and uses this additional information to maximize the likelihood. Observe the connections between K-means, soft K-means, and EM - all perform alternating steps of data imputation and subsequent parameter optimization given the imputed data. In the data imputation step for K-means, each data point is given a hard assignment to a latent variable value, while in soft K-means and EM, each data point gets assigned a *distribution* over the latent variables.

From our derivation of EM, we can see that the value of the likelihood never decreases during the execution of the algorithm. It turns out that EM will converge to a parameter estimate with zero gradient, but will not necessarily find the global optimum. When the clusters are sufficiently separated, EM can exhibit Newtonian (second-order) convergence speed - however, if the clusters are close together, the posteriors will be very flat and EM can take longer than gradient descent methods to converge.

EM for MoG

As a concrete example, we derive the EM updates for fitting a mixture of Gaussians. Recall the MoG model

$$\mathbf{x}|z \sim \mathcal{N}(\boldsymbol{\mu}_z, \boldsymbol{\Sigma}_z), \quad p(z = k) =: \alpha_k$$

We define the parameter set $\boldsymbol{\theta}$ as the set of all $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k, \alpha_k$. Let $\mathbf{x}_1, ..., \mathbf{x}_N \in \mathbb{R}^d$ be our observed data. Define $q_{ki}^t := q^t(z_i = k|\mathbf{x}_i)$. The EM updates, derived below, are

E step:

$$q^{t+1}(z_i = k|\mathbf{x}_i) = \frac{\alpha_k^t p(\mathbf{x}_i|z_i = k; \boldsymbol{\theta}^t)}{\sum_{j=1}^K \alpha_j^t p(\mathbf{x}_i|z_i = j; \boldsymbol{\theta}^t)}$$

M step:

$$\begin{aligned}
\boldsymbol{\mu}_k^{t+1} &= \frac{\sum_{i=1}^N q_{ki}^{t+1}\mathbf{x}_i}{\sum_{i=1}^N q_{ki}^{t+1}} \\
\boldsymbol{\Sigma}_k^{t+1} &= \frac{\sum_{i=1}^N q_{ki}^{t+1}(\mathbf{x}_i - \boldsymbol{\mu}_k^{t+1})(\mathbf{x}_i - \boldsymbol{\mu}_k^{t+1})^T}{\sum_{i=1}^N q_{ki}^{t+1}} \\
\alpha_k^{t+1} &= \frac{1}{N}\sum_{i=1}^N q_{ki}^{t+1}
\end{aligned}$$

In the E step, we assign to each $\mathbf{x}_i$ a probability distribution over latents (that is, a soft assignment). This assignment is $p(\mathbf{x}_i|z_i = k; \boldsymbol{\theta}^t)$, the Gaussian likelihood of the data, but reweighted by the prior and normalized. In the M step, we are essentially computing the usual maximimum likelihood estimates of the parameters, but weighted by the posterior on z; indeed, if we set $q_{ki}^{t+1} = \frac{1}{N}$, then we are using the usual MLE. The update for $\boldsymbol{\mu}_k$ is entirely analogous to the update to the centroids for soft k-means (6.1). The main difference is that now we are also updating estimates of covariances and the prior and synthesizing this information in our posterior estimates in the E step, which will in turn influence the $\boldsymbol{\mu}_k$ assignments in the next E step.

We now derive these updates. Recall the log likelihood is

$$\ell(\boldsymbol{\theta}; \mathbf{x}) = \sum_{i=1}^N \log\left(\sum_{k=1}^K p(\mathbf{x}_i|z_i = k; \boldsymbol{\theta})p(z_i = k; \boldsymbol{\theta})\right)$$

For the E step, we set

$$\begin{aligned}
q^{t+1}(z_i = k|\mathbf{x}_i) &= p(z_i = k|\mathbf{x}_i; \boldsymbol{\theta}^t) \\
&= \frac{p(z_i = k, \mathbf{x}_i; \boldsymbol{\theta}^t)}{p(\mathbf{x}_i; \boldsymbol{\theta}^t)} \\
&= \frac{p(\mathbf{x}_i|z_i = k; \boldsymbol{\theta}^t)p(z_i = k; \boldsymbol{\theta}^t)}{\sum_{j=1}^K p(\mathbf{x}_i, z_i = j; \boldsymbol{\theta}^t)} \\
&= \frac{p(\mathbf{x}_i|z_i = k; \boldsymbol{\theta}^t)p(z_i = k; \boldsymbol{\theta}^t)}{\sum_{j=1}^K p(\mathbf{x}_i|z_i = j; \boldsymbol{\theta}^t)p(z_i = j; \boldsymbol{\theta}^t)}
\end{aligned}$$

For MoG, $p(\mathbf{x}|z = j; \boldsymbol{\theta}^t)$ is the pdf of $\mathcal{N}(\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$ evaluated at $\mathbf{x}$.

For the M step, we need to maximize the expected complete log-likelihood $\ell_q = \mathbb{E}_{q^{t+1}}(L_c(\mathbf{X}, \mathbf{z}; \boldsymbol{\theta}))$. The parameters to estimate are $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$, and α_k, the prior. The expected complete log likelihood is

$$\begin{aligned}
\mathbb{E}_{q^{t+1}}(L_c(x, z; \boldsymbol{\theta})) &= \sum_{i=1}^N \sum_{k=1}^K q_{ki}^{t+1}\left[\log\alpha_k - \frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_k)^T\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k) - \frac{1}{2}\log\Big((2\pi)^d|\boldsymbol{\Sigma}_k|\Big)\right] \\
&= \sum_{i=1}^N \sum_{k=1}^K q_{ki}^{t+1}\left[\log\alpha_k - \frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_k)^T\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_k) + \frac{1}{2}\log\Big((2\pi)^{-d}|\boldsymbol{\Sigma}_k^{-1}|\Big)\right]
\end{aligned}$$

We can take gradients with respect to the parameters:

$$\frac{\partial \ell_q}{\partial \boldsymbol{\mu}_k} = \boldsymbol{\Sigma}_k^{-1} \sum_{i=1}^{N} q_{ki}^{t+1} (\mathbf{x}_i - \boldsymbol{\mu}_k)$$

$$\frac{\partial \ell_q}{\partial \boldsymbol{\Sigma}_k^{-1}} = \frac{1}{2} \sum_{i=1}^{N} q_{ki}^{t+1} [\boldsymbol{\Sigma}_k - (\mathbf{x}_i - \boldsymbol{\mu}_k^{t+1})(\mathbf{x}_i - \boldsymbol{\mu}_k^{t+1})^T]$$

Setting the gradients to zero and solving these equations gives us the updates for μ_k, Σ_k shown above. To obtain the update for α_k, we need to introduce the constraint $\sum_{k=1}^{K} \alpha_k = 1$ via Lagrange multipliers. We thus maximize the Lagrangian $\ell_q' = \mathbb{E}_{q^{t+1}}(L_c(x, z; \boldsymbol{\theta})) - \lambda(\sum_{k=1}^{K} \alpha_k - 1)$. Taking gradients, we get

$$\frac{\partial \ell_q'}{\partial \alpha_k} = -\lambda + \frac{1}{\alpha_k} \sum_{i=1}^{N} q_{ki}^{t+1} = 0$$

We can rearrange to write

$$\alpha_k \lambda = \sum_{i=1}^{N} q_{ki}^{t+1}$$

Summing over all k, we obtain

$$\begin{aligned} \lambda \sum_{k=1}^{K} \alpha_k &= \sum_{k=1}^{K} \sum_{i=1}^{N} q_{ki}^{t+1} \\ &= \sum_{i=1}^{N} \sum_{k=1}^{K} q^{t+1}(z_i = k | \mathbf{x}_i) \\ &= \sum_{i=1}^{N} 1 = N \end{aligned}$$

using the fact that q^{t+1} is a distribution over z_i. Since $\sum_{k=1}^{K} \alpha_k = 1$, the left hand side reduces to λ, so we conclude $\lambda = N$. Substituting into the original gradient, we have

$$\frac{\partial \ell_q'}{\partial \alpha_k} = -N + \frac{1}{\alpha_k} \sum_{i=1}^{N} q_{ki}^{t+1}$$

Setting this to zero gives us the desired updates for α_k.

Appendix: Jensen's Inequality Proof

Let f be convex, and X be a random variable. Construct the tangent line $L(X) = aX + b$ to f at $\mathbb{E}(X)$ for some a, b - this means $L(\mathbb{E}(X)) = f(\mathbb{E}(X))$. By convexity, $L(X) \leq f(X)$ for all X.[3] Then by monotonicity of expectation, we have

$$f(\mathbb{E}(X)) = L(\mathbb{E}(X)) = a\mathbb{E}(X) + b = \mathbb{E}(aX + b) = \mathbb{E}(L(X)) \leq \mathbb{E}(f(X))$$

[3]Actually, it takes some extra work to prove this intuitive fact. We will take it for granted here.

Appendix: Distance and Similarity

In our discussion of K-means, we restricted ourselves to using L^2 distance as a distance function. Formally, a distance function $d(x, y)$ is defined as a non-negative function that satisfies the following properties:

1. $d(x, y) = 0$ iff $x = y$
2. $d(x, y) = d(y, x)$ for all x, y
3. $d(x, z) \leq d(x, y) + d(y, z)$ for all x, y, z (triangle inequality)

A **dissimilarity** measure $d(x, y)$ is a function satisfying the above properties except possibly the triangle inequality. One possible **similarity** measure $s(x, y)$ can be defined as $-d(x, y)$. In clustering, we are free to choose our notion of similarity. Different algorithms may or may not work differently depending on which similarity measure we choose.

For example, it does not make sense to use the K-means algorithm if we care about L^1 distances instead of L^2. However, we can apply the same principles used to derive the K-means algorithm: if we replace the L^2 norm in the objective by L^1 and do a similar alternating minimization, then on the centroid assignment step, we will set each centroid to the median of the data instead of the mean. On the cluster assignment step, we will assign each point to the closest centroid in L^1 distance. This variation is called K-medians.

Deciding which similarity measure to use is a modeling choice that typically depends on the data and desired clustering properties. For example, K-medians may be of use if the data has outliers and we desire a more robust estimator of the clusters. In certain domains, such as computer vision, the L^p distances are not appropriate measures of dissimilarity, so other measures may be used.

Chapter 7

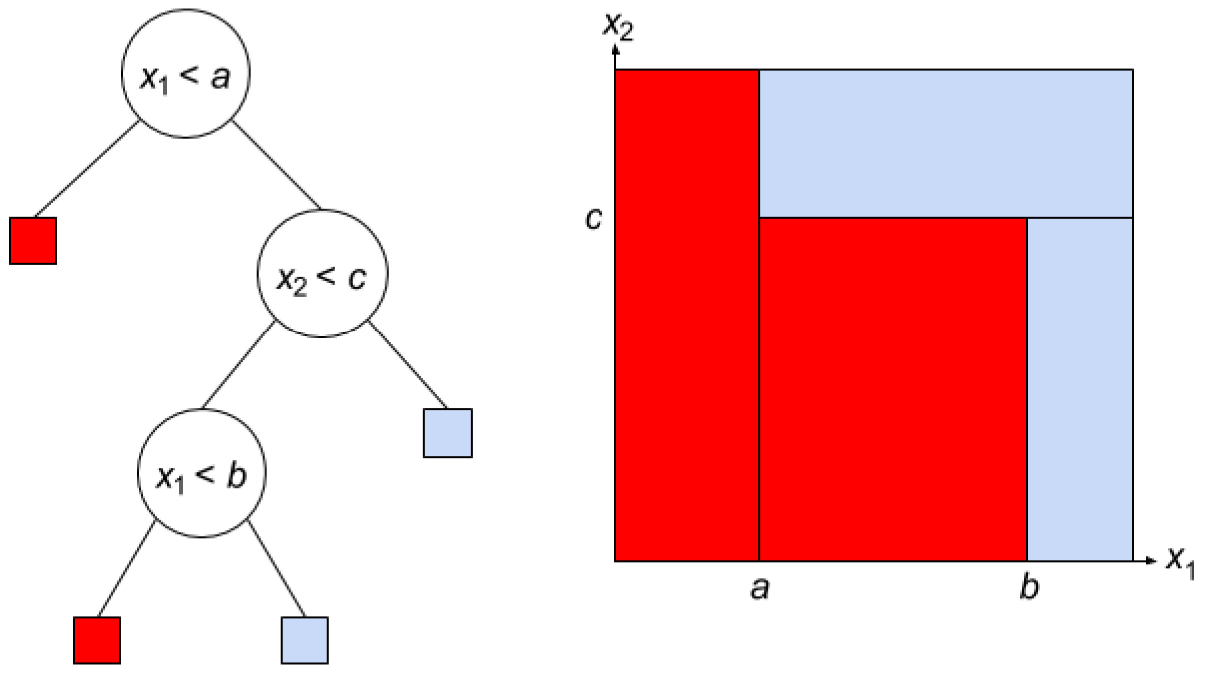
Decision Tree Learning

7.1 Decision Trees

A **decision tree** is a model that makes predictions by posing a series of simple tests on the given point. This process can be represented as a tree, with the non-leaf nodes of the tree representing these tests. At the leaves of this tree are the values to be predicted after descending the tree in the manner prescribed by each node's test. Decision trees can be used for both classification and regression, but we will focus exclusively on classification.

In the simple case which we consider here, the tests are of the form "Is feature j of this point less than the value v?"[1] These tests carve up the feature space in a nested rectangular fashion:

[1] One could use more complicated decision procedures. For example, in the case of a categorical variable, there could be a separate branch in the tree for each value that the variable could take on. Or multiple features could be used, e.g. "Is $x_1 > x_2$?" However, using more complicated decision procedures complicates the learning process. For simplicity we consider the single-feature binary case here.



Given sufficiently many splits, a decision tree can represent an arbitrarily complex classifier and perfectly classify any training set.[2]

Training

Decision trees are trained in a greedy, recursive fashion, downward from the root. After creating a node with some associated split, the children of that node are constructed by the same tree-growing procedure. However, the data used to train left subtree are only those points satisfying $x_j < v$, and similarly the right subtree is grown with only those points with $x_j \geq v$. Eventually we must each a base case where no further splits are to be made, and a prediction (rather than a split) is associated with the node.

We can see that the process of building the tree raises at least the following questions:

- How do we pick the split-feature, split-value pairs?
- How do we know when to stop growing the tree?

Typically decision trees will pick a split by considering all possible splits and choosing the one that is the best according to some criterion. We will discuss possible criteria later, but first it is worth asking what we mean by "all possible splits". It is clear that we should look at all features, but what about the possible values? Observe that in the case where tests are of the form $x_j < v$, there are infinitely many values of v we could choose, but only finitely many different resulting splits (since there are finitely many training points). Therefore it suffices to perform a one-dimensional sweep: sort the datapoints by their x_j values and only consider these values as split candidates.

Now let us revisit the criterion. Intuitively, we want to choose the split which most reduces our classifier's uncertainty about which class points belongs to. In the ideal case, the hyperplane $x_j = v$

[2] Unless two training points of different classes coincide.

perfectly splits the given data points such that all the instances of the positive class lie on one side and all the instances of the negative class lie on the other.

Entropy and information

One way to quantify the aforementioned "uncertainty", we'll use the ideas of **surprise** and **entropy**. The surprise of observing that a discrete random variable Y takes on value k is:

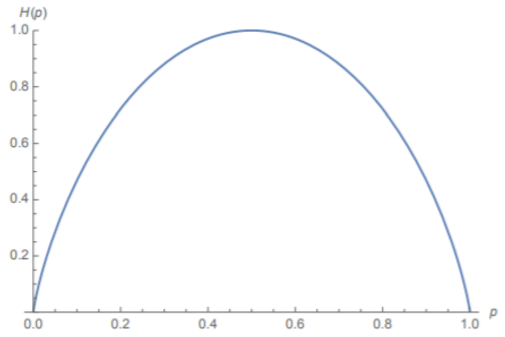
$$\log \frac{1}{P(Y=k)} = -\log P(Y=k)$$

As $P(Y=k) \to 0$, the surprise of observing that value approaches ∞, and conversely as $P(Y=k) \to 1$, the surprise of observing that value approaches 0.

The entropy of Y, denoted $H(Y)$, is the expected surprise:

$$\begin{aligned} H(Y) &= \mathbb{E}[-\log P(Y)] \\ &= -\sum_k P(Y=k)\log P(Y=k) \end{aligned}$$

Here's a graph of entropy vs. p for a Bernoulli(p) random variable:



Observe that as a function of p (the probability of the variable being 1), the entropy is strictly concave. Moreover, it is maximized at $p = \frac{1}{2}$, when the probability distribution is uniform with respect to the outcomes. That is to say, a coin that is completely fair ($P(Y=0) = P(Y=1) = \frac{1}{2}$) has more entropy than a coin that is biased. This is because we are less sure of the outcome of the fair coin than the biased coin *overall*. Even though we are more surprised when the biased coin comes up as its more unlikely outcome, the way that entropy is defined gives a higher uncertainty score to the fair coin. Generally speaking, a random variable has more entropy when the distribution of its outcomes is closer to uniform and less entropy when the distribution is highly skewed to one outcome.

This definition is for random variables, but in practice we work with data. The distribution is empirically defined by our training points $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$. Concretely, the probability of class k is the proportion of datapoints having class k:

$$P(Y=k) = \frac{|\{i \mid y_i = k\}|}{n}$$

We know that when we choose a split-feature, split-value pair, we want to reduce entropy in some way. Let $X_{j,v}$ be an indicator variable which is 1 when $x_j < v$, and 0 otherwise. There are a few entropies to consider:

- $H(Y)$
- $H(Y|X_{j,v}=1)$, the entropy of the distribution of points such that $x_j < v$.
- $H(Y|X_{j,v}=0)$, the entropy of the distribution of points such that $x_j \geq v$.

$H(Y)$ is not really under our control: we start with the set of points with labels represented by Y, this distribution has some entropy, and now we wish to carve up those points in a way to minimize the entropy remaining. Thus, the quantity we want to minimize is a weighted average of the two sides of the split, where the weights are (proportional to) the sizes of two sides:

$$\text{minimize } H(Y|X_{j,v}) := P(X_{j,v}=1)H(Y|X_{j,v}=1) + P(X_{j,v}=0)H(Y|X_{j,v}=0)$$

This quantity $H(Y|X_{j,v})$ is known as the **conditional entropy** of Y given $X_{j,v}$. An equivalent way of seeing this is that we want to maximize the information we've learned, which is represented by how much entropy is reduced after learning whether or not $x_j < v$:

$$\text{maximize } I(X_{j,v};Y) := H(Y) - H(Y|X_{j,v})$$

This quantity $I(X_{j,v};Y)$ is known as the **mutual information** between $X_{j,v}$ and Y. It is always nonnegative, and it's zero iff the resulting sides of the split have the same distribution of classes as the original set of points. Let's say you were using a decision tree to classify emails as spam and ham. For example, you gain no information if you take a set of (20 ham, 10 spam) and split it on some feature to give you sets of (12 ham, 6 spam); (8 ham, 4 spam) because the empirical distribution of those two resulting sets is equal to the original one.

Gini impurity

Another way to assess the quality of a split is **Gini impurity**, which measures how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset. It as defined as

$$G(Y) = \sum_k P(Y=k) \sum_{j \neq k} P(Y=j) = \sum_k P(Y=k)(1-P(Y=k)) = 1 - \sum_k P(Y=k)^2$$

Exactly as with entropy, we can define a version of this quantity which is dependent on the split. For example, $G(Y|X_{j,v}=1)$ would be the Gini impurity computed only on those points satisfying $x_j < v$. And we can define an analogous quantity

$$G(Y|X_{j,v}) := P(X_{j,v}=1)G(Y|X_{j,v}=1) + P(X_{j,v}=0)G(Y|X_{j,v}=0)$$

which is to be minimized.

Empirically, the Gini impurity is found to produce very similar results to entropy, and it is slightly faster to compute because we don't need to take logs.

Why not the misclassification rate?

Since we ultimately care about classification accuracy, it is natural to wonder why we don't directly use the misclassification rate by plurality vote as the measure of impurity:

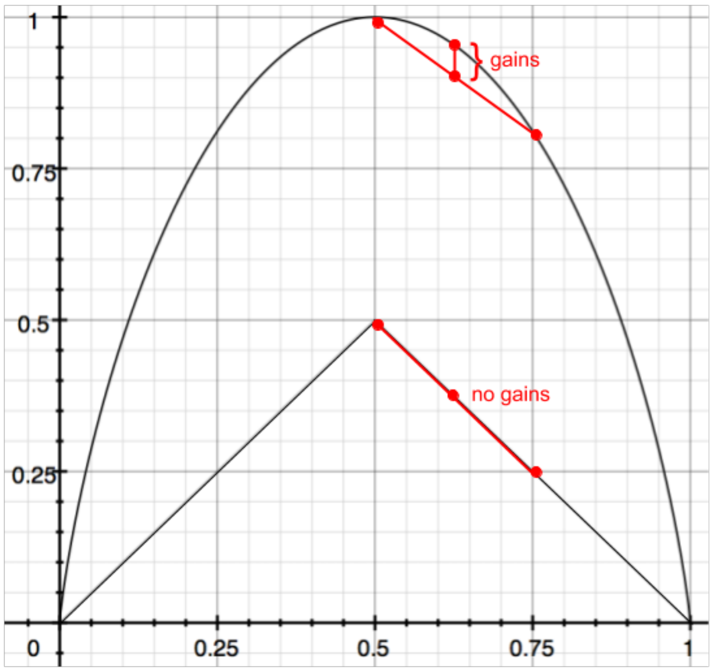
$$M(Y) = 1 - \max_k P(Y=k)$$

It turns out that this quantity is insensitive in the sense that the quantity it induces for evaluating splits ($M(Y|X_{j,v})$) may assign the same value to a variety of splits which are not, in fact, equally good. Suppose[3] the current node has 40 training points of class 1 and 40 of class 2. Here $M(Y) = 1 - \frac{1}{2} = \frac{1}{2}$. Now consider two possible splits:

1. Separate into a region $x_j < v$ with 30 points of class 1 and 10 of class 2 ($^{10}/_{40} = {}^{1}/_{4}$ misclassified), and a region $x_j \geq v$ with 10 points of class 1 and 30 of class 2 ($^{10}/_{40} = {}^{1}/_{4}$ misclassified). Since $^{40}/_{80} = {}^{1}/_{2}$ of the points lie satisfy $x_j < v$ and $^{40}/_{80} = {}^{1}/_{2}$ of the points lie satisfy $x_j \geq v$,
$$M(Y|X_{j,v}) = \frac{1}{2} \cdot \frac{1}{4} + \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{4}$$
2. Separate into a region $x_j < v$ with 20 points of class 1 and 40 of class 2 ($^{20}/_{60} = {}^{1}/_{3}$ misclassified), and a region $x_j \geq v$ with 20 points of class 1 and 0 of class 2 ($^{0}/_{20} = 0$ misclassified). Since $^{60}/_{80} = {}^{3}/_{4}$ of the points lie satisfy $x_j < v$ and $^{20}/_{80} = {}^{1}/_{4}$ of the points lie satisfy $x_j \geq v$,
$$M(Y|X_{j,v}) = \frac{3}{4} \cdot \frac{1}{3} + \frac{1}{4} \cdot 0 = \frac{1}{4}$$

We see that the criterion value is the same for both splits, even though the second one appears to do a better job reducing our uncertainty for many of the points. And indeed, the conditional entropy and Gini impurity scores are lower (so the information gain is higher) for the second split.[4]

The limitation of this criterion can be understood mathematically in terms of **strict concavity**. This property means that the graph of the function lies strictly below the tangent line[5] at every point (except the point of tangency, of course). Consider the plots of entropy and misclassification rate for a binary classification problem:



[3] This example is from https://sebastianraschka.com/faq/docs/decision-tree-binary.html
[4] Left as exercise (or see the source URL).
[5] Or, in higher dimensions, hyperplane

Both curves are concave, but the one for misclassification rate is not strictly so; it has only two unique tangent lines. Because the conditional quantity is a convex combination (since it is weighted by probabilities/proportions) of the children's values, the strictly convex functions always yield positive information gain as long as the children's distributions are not identical to the parent's. We have no such guarantee in either linear region of the misclassification rate curve; any convex combination of points on a line also lies on the line, yielding zero information gain.

Stopping criteria

We mentioned earlier that sufficiently deep decision trees can represent arbitrarily complex decision boundaries, but of course this will lead to overfitting if we are not careful. There are a number of heuristics we may consider to decide when to stop splitting:

- Limited depth: don't split if the node is beyond some fixed depth depth in the tree
- Node purity: don't split if the proportion of training points in some class is sufficiently high
- Information gain criteria: don't split if the gained information/purity is sufficiently close to zero

Note that these are not mutually exclusive, and the thresholds can be tuned with validation. As an alternative (or addition) to stopping early, you can **prune** a fully-grown tree by re-combining splits if doing so reduces validation error.

7.2 Random Forests

Another way to combat overfitting is to combine the predictions of many varied models into a single prediction, typically by plurality vote in the case of classification and averaging in the case of regression. This is a general technique known as **ensemble learning**. To understand the motivation for averaging, consider a set of uncorrelated random variables $\{Y_i\}_{i=1}^n$ with common mean $\mathbb{E}[Y_i] = \mu$ and variance $\text{Var}(Y_i) = \sigma^2$. The average of these has the same expectation

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^n Y_i\right] = \frac{1}{n}\sum_{i=1}^n \mathbb{E}[Y_i] = \frac{1}{n}\cdot n\mu = \mu$$

but reduced variance compared to each of the individual Y_i's:

$$\text{Var}\left(\frac{1}{n}\sum_{i=1}^n Y_i\right) = \left(\frac{1}{n}\right)^2 \sum_{i=1}^n \text{Var}(Y_i) = \frac{1}{n^2}\cdot n\sigma^2 = \frac{\sigma^2}{n}$$

In the context of ensemble methods, these Y_i are analogous to the prediction made by classifier i. The combined prediction has the same expected value as any individual prediction but lower variance. Real-world predictions will of course not be completely uncorrelated, but reducing correlation will generally reduce the final variance, so this is a goal to aim for.

Random forests are a specific ensemble method where the individual models are decision trees trained in a randomized way so as to reduce correlation among them. Because the basic decision tree building algorithm is deterministic, it will produce the same tree every time if we give it the same dataset and use the same algorithm hyperparameters (stopping conditions, etc.).

Random forests are typically randomized in the following ways:

- Per-classifier **bagging** (short for **bootstrap aggregating**): sample some number $m < n$ of datapoints uniformly with replacement, and use these as the training set.
- Per-split **feature randomization**: sample some number number $k < d$ of features as candidates to be considered for this split

Both the size of the random subsample of training points and the number of features at each split are hyperparameters which should be tuned through cross-validation.

7.3 Boosting

We have seen that in the case of random forests, combining many imperfect models can produce a single model that works very well. This is the idea of **ensemble methods**. However, random forests treat each member of the forest equally, taking a plurality vote or an average over their outputs. The idea of **boosting** is to combine the models (typically called *weak learners* in this context) in a more principled manner. The key idea is as follows: to improve our combined model, we should focus on finding learners that correctly predict the points which the overall boosted model is currently predicting inaccurately. Boosting algorithms implement this idea by associating a weight with each training point and iteratively reweighting so that mispredicted points have relatively high weights. Intuitively, some points are "harder" to predict than others, so the algorithm should focus its efforts on those.

These ideas also connect to matching pursuit. In both cases, our overall predictor is an additive combination of pieces which are selected one-by-one in a greedy fashion. The algorithm keeps track of residual prediction errors, chooses the "direction" to move based on these, and then performs a sort of line search to determine how far along that direction to move.

AdaBoost

There are many flavors of boosting. We will discuss one of the most popular versions, known as **AdaBoost** (short for *adaptive boosting*), which is a method for binary classification. Its developers won the prestgious Gödel Prize for this work.

Algorithm

We present the algorithm first, then derive it later. Assume access to a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$.

1. Initialize the weights $w_i = \frac{1}{n}$ for all $i = 1, \dots, n$ training points.
2. Repeat for $m = 1, \dots, M$:
   (a) Build a classifier $G_m : \mathbb{R}^d \to \{-1, 1\}$, where in the training process the data are weighted according to w_i.
   (b) Compute the weighted error $e_m = \frac{\sum_{i \text{ misclassified}} w_i}{\sum_i w_i}$.
   (c) Re-weight the training points as

$$w_i \leftarrow w_i \cdot \begin{cases} \sqrt{\frac{1-e_m}{e_m}} & \text{if misclassified by } G_m \\ \sqrt{\frac{e_m}{1-e_m}} & \text{otherwise} \end{cases}$$

(d) Optional: normalize the weights w_i to sum to 1.

We first address the issue of step (a): how do we train a classifier if we want to weight different samples differently? One common way to do this is to resample from the original training set every iteration to create a new training set that is fed to the next classifier. Specifically, we create a training set of size n by sampling n values from the original training data with replacement, according to the distribution w_i. (This is why we might renormalize the weights in step (d).) This way, data points with large values of w_i are more likely to be included in this training set, and the next classifier will place higher priority on such data points.

Suppose[6] that our weak learners always produce an error $e_m < \frac{1}{2}$. To make sense of the formulas we see in the algorithm, note that for step (c), if the i-th data point is misclassified, then the weight w_i gets increased by a factor of $\sqrt{\frac{1-e_m}{e_m}}$ (more priority placed on sample i), while if it is classified correctly, the priority gets decreased. AdaBoost does have a practical weakness in that this aggressive reweighting can cause the classifier to focus too much on certain training examples – if the data contains outliers or a lot of noise, the boosting algorithm's generalization performance may suffer as it overfits to a few challenging examples.

We have not yet discussed how to make a prediction on test points given our classifiers $G_1, \dots, G_M$. One conceivable method is to use logistic regression with $G_m(\mathbf{x})$ as features. However, a smarter choice that is based on the AdaBoost algorithm is to set

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1-e_m}{e_m} \right)$$

and classify $\mathbf{x}$ by

$$h(\mathbf{x}) = \operatorname{sgn} \left(\sum_{m=1}^{M} \alpha_m G_m(\mathbf{x}) \right)$$

Note that this choice of α_m (derived later) gives high weight to classifiers that have low error:

- As $e_m \to 0$, $\frac{1-e_m}{e_m} \to \infty$, so $\alpha_m \to \infty$.
- As $e_m \to 1$, $\frac{1-e_m}{e_m} \to 0$, so $\alpha_m \to -\infty$.

We now proceed to demystify the formulas in the algorithm above by presenting a matching pursuit interpretation of AdaBoost. This interpretation is also useful because it generalizes to a powerful technique called Gradient Boosting, of which AdaBoost is just one instance.

Derivation of AdaBoost

Suppose we have computed classifiers $G_1, \dots, G_{m-1}$ along with their corresponding weights α_k and we want to compute the next classifier G_m along with its weight α_m. The output of our model so far is $F_{m-1}(\mathbf{x}) := \sum_{i=1}^{m-1} \alpha_i G_i(\mathbf{x})$, and we want to minimize the risk:

$$\alpha_m, G_m = \arg\min_{\alpha, G} \sum_{i=1}^{n} L(y_i, F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i))$$

[6] This is a reasonable thing to ask. A classifier with error $e_m \geq \frac{1}{2}$ is even worse than the trivial classifier which predicts the class with the most total weight without regard for the input $\mathbf{x}_i$.

for some suitable loss function $L(y, \hat{y})$. Loss functions we have previously used include mean squared error for linear regression, cross-entropy loss for logistic regression, and hinge loss for SVM. For AdaBoost, we use the **exponential loss**:

$$L(y, \hat{y}) = e^{-y\hat{y}}$$

This loss function is illustrated in Figure 7.1. Observe that if $y\hat{y} > 0$ (i.e. $\hat{y}$ has the correct sign), the loss decreases exponentially in $|\hat{y}|$, which should be interpreted as the confidence of the prediction. Conversely, if $y\hat{y} < 0$, our loss is increasing exponentially in the confidence of the prediction.

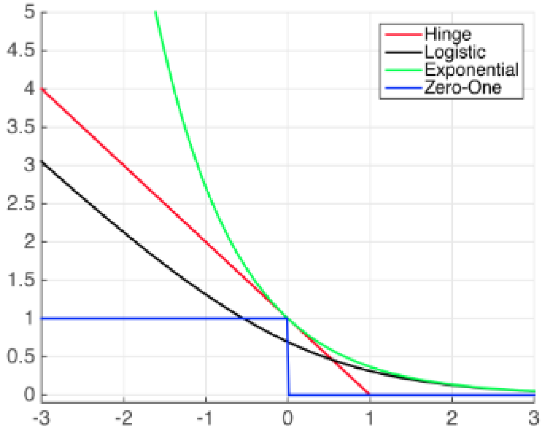


Figure 7.1: The exponential loss provides exponentially increasing penalty for confident incorrect predictions. This figure is from Cornell CS4780 notes.

Plugging the exponential loss into the general optimization problem above yields

$$\begin{aligned}\alpha_m, G_m &= \arg\min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i(F_{m-1}(\mathbf{x}_i)+\alpha G(\mathbf{x}_i))} \\ &= \arg\min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i F_{m-1}(\mathbf{x}_i)} e^{-y_i \alpha G(\mathbf{x}_i)}\end{aligned}$$

The term $w_i^{(m)} := e^{-y_i F_{m-1}(\mathbf{x}_i)}$ is a constant with respect to our optimization variables. We can split out this sum into the components with correctly classified points and incorrectly classified points:

$$\begin{aligned}\alpha_m, G_m &= \arg\min_{\alpha, G} \sum_{i=1}^{n} w_i^{(m)} e^{-y_i \alpha G(\mathbf{x}_i)} \\ &= \arg\min_{\alpha, G} \sum_{y_i = G(\mathbf{x}_i)} w_i^{(m)} e^{-\alpha} + \sum_{y_i \neq G(\mathbf{x}_i)} w_i^{(m)} e^{\alpha} \qquad (*) \\ &= \arg\min_{\alpha, G} e^{-\alpha} \left(\sum_{i=1}^{n} w_i^{(m)} - \sum_{y_i \neq G(\mathbf{x}_i)} w_i^{(m)} \right) + e^{\alpha} \sum_{y_i \neq G(\mathbf{x}_i)} w_i^{(m)} \\ &= \arg\min_{\alpha, G} (e^{\alpha} - e^{-\alpha}) \sum_{y_i \neq G(\mathbf{x}_i)} w_i^{(m)} + e^{-\alpha} \sum_{i=1}^{n} w_i^{(m)}\end{aligned}$$

To arrive at $(*)$ we have used the fact that $y_i G_m(\mathbf{x}_i)$ equals 1 if the prediction is correct, and -1 otherwise. For a fixed value of α, the second term in this last expression does not depend on G. Thus we can see that the best choice of $G_m(\mathbf{x})$ is the classifier that minimizes the total weight of the misclassified points. Let

$$e_m = \frac{\sum_{y_i \neq G_m(\mathbf{x}_i)} w_i^{(m)}}{\sum_i w_i^{(m)}}$$

Once we have obtained G_m, we can solve for α_m. Dividing $(*)$ by the constant $\sum_{i=1}^n w_i^{(m)}$, we obtain

$$\alpha_m = \arg\min_\alpha (1 - e_m)e^{-\alpha} + e_m e^{\alpha}$$

We can solve for the minimizer analytically using calculus. Setting the derivative of the objective function to zero gives

$$0 = -(1 - e_m)e^{-\alpha} + e_m e^{\alpha} = -e^{-\alpha} + e_m(e^{-\alpha} + e^{\alpha})$$

Multiplying through by e^{α} yields

$$0 = -1 + e_m(1 + e^{2\alpha})$$

Adding one to both sides and dividing by e_m, we have

$$\frac{1}{e_m} = 1 + e^{2\alpha}$$

i.e.

$$e^{2\alpha} = \frac{1}{e_m} - 1 = \frac{1 - e_m}{e_m}$$

Taking natural log on both sides and halving, we arrive at

$$\alpha_m = \frac{1}{2} \ln\left(\frac{1 - e_m}{e_m}\right)$$

as claimed earlier. From the optimal α_m, we can derive the weights:

$$\begin{aligned}
w_i^{(m+1)} &= \exp\big(-y_i F_m(\mathbf{x}_i)\big) \\
&= \exp\big(-y_i[F_{m-1}(\mathbf{x}_i) + \alpha_m G_m(\mathbf{x}_i)]\big) \\
&= w_i^{(m)} \exp\big(-y_i G_m(\mathbf{x}_i)\alpha_m\big) \\
&= w_i^{(m)} \exp\left(-y_i G_m(\mathbf{x}_i) \frac{1}{2} \ln\left(\frac{1 - e_m}{e_m}\right)\right) \\
&= w_i^{(m)} \exp\left(\ln\left[\left(\frac{1 - e_m}{e_m}\right)^{-\frac{1}{2} y_i G_m(\mathbf{x}_i)}\right]\right) \\
&= w_i^{(m)} \left(\frac{1 - e_m}{e_m}\right)^{-\frac{1}{2} y_i G_m(\mathbf{x}_i)} \\
&= w_i^{(m)} \sqrt{\frac{e_m}{1 - e_m}}^{\,y_i G_m(\mathbf{x}_i)}
\end{aligned}$$

Here we see that the multiplicative factor is $\sqrt{\frac{e_m}{1-e_m}}$ when $y_i = G_m(\mathbf{x}_i)$ and $\sqrt{\frac{1-e_m}{e_m}}$ otherwise. This completes the derivation of the algorithm.

As a final note about the intuition, we can view these α updates as pushing towards a solution in some direction until we can no longer improve our performance. More precisely, whenever we compute α_m (and thus $w^{(m+1)}$), for the incorrectly classified entries, we have

$$\sum_{y_i \neq G_m(\mathbf{x}_i)} w_i^{(m+1)} = \sum_{y_i \neq G_m(\mathbf{x}_i)} w_i^{(m)} \sqrt{\frac{1-e_m}{e_m}}$$

Dividing the right-hand side by $\sum_{i=1}^n w_i^{(m)}$, we obtain $e_m\sqrt{\frac{1-e_m}{e_m}} = \sqrt{e_m(1-e_m)}$. Similarly, for the correctly classified entries, we have

$$\frac{\sum_{y_i = G_m(\mathbf{x}_i)} w_i^{(m+1)}}{\sum_{i=1}^n w_i^{(m)}} = (1-e_m)\sqrt{\frac{e_m}{1-e_m}} = \sqrt{e_m(1-e_m)}$$

Thus these two quantities are the same once we have adjusted our α, so the misclassified and correctly classified sets both get equal total weight.

This observation has an interesting practical implication. Even after the training error goes to zero, the test error may continue to decrease. This may be counter-intuitive, as one would expect the classifier to be overfitting to the training data at this point. One interpretation for this phenomenon is that even though the boosted classifier has achieved perfect training error, it is still refining its fit in a max-margin fashion, which increases its generalization capabilities.

Gradient Boosting

AdaBoost assumes a particular loss function, the exponential loss function. **Gradient boosting** is a more general technique that allows an arbitrary differentiable loss function $L(y, \hat{y})$. Recall the general optimization problem we must solve when choosing the next model:

$$\min_{\alpha, G} \sum_{i=1}^n L(y_i, F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i))$$

Here G should no longer be assumed to be a classifier; it may be real-valued if we are solving a regression problem. By a Taylor expansion in the second argument,

$$L(y_i, F_{m-1}(\mathbf{x}_i) + \alpha G(\mathbf{x}_i)) \approx L(y_i, F_{m-1}(\mathbf{x}_i)) + \frac{\partial L}{\partial \hat{y}}(y_i, F_{m-1}(\mathbf{x}_i)) \cdot \alpha G(\mathbf{x}_i)$$

We can view the collection of predictions that a model G produces for the training set as a single vector $\mathbf{g} \in \mathbb{R}^n$ with components $g_i = G(\mathbf{x}_i)$. Then the overall cost function is approximated to first order by

$$\sum_{i=1}^n L(y_i, F_{m-1}(\mathbf{x}_i)) + \alpha \underbrace{\sum_{i=1}^n \frac{\partial L}{\partial \hat{y}}(y_i, F_{m-1}(\mathbf{x}_i)) \cdot g_i}_{\langle \nabla_{\hat{y}} L(\mathbf{y}, F_{m-1}(\mathbf{X})), \mathbf{g} \rangle}$$

where, in abuse of notation, $F_{m-1}(\mathbf{X})$ is a vector with $F_{m-1}(\mathbf{x}_i)$ as its ith element, and $\nabla_{\hat{y}} L(\mathbf{y}, \hat{\mathbf{y}})$ is a vector with $\frac{\partial L}{\partial \hat{y}}(y_i, \hat{y}_i)$ as its ith element. To decrease the cost in a steepest descent fashion, we seek the direction $\mathbf{g}$ which maximizes

$$\left|\langle -\nabla_{\hat{y}} L(\mathbf{y}, F_{m-1}(\mathbf{X})), \mathbf{g} \rangle\right|$$

subject to $\mathbf{g}$ being the output of some model G in the model class we are considering.[7]

Some comments are in order. First, observe that the loss need only be differentiable with respect to its inputs, not necessarily with respect to model parameters, so we can use non-differentiable models such as decision trees. Additionally, in the case of squared loss $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ we have

$$\frac{\partial L}{\partial \hat{y}}(y_i, F_{m-1}(\mathbf{x}_i)) = -(y_i - F_{m-1}(\mathbf{x}_i))$$

so

$$-\nabla_{\hat{y}} L(\mathbf{y}, F_{m-1}(\mathbf{X})) = \mathbf{y} - F_{m-1}(\mathbf{X})$$

This means the algorithm will follow the residual, as in matching pursuit.

[7] The absolute value may seem odd, but consider that after choosing the direction $\mathbf{g}_m$, we perform a line search to select α_m. This search may choose $\alpha_m < 0$, effectively flipping the direction. The key is to maximize the magnitude of the inner product.

Chapter 8

Deep Learning

8.1 Convolutional Neural Networks

Neural networks have been successfully applied to many real-world problems, such as predicting stock prices and learning robot dynamics in autonomous systems. The most general type of neural network is **multilayer perceptrons (MLPs)**, which we have already studied in detail and employed to classify 28×28 pixel images as digits (MNIST). While MLPs can be used to effectively classify small images (such as those in MNIST), they are impractical for large images. Let's see why. Given a $W \times H \times 3$ image (over 3 channels — red, green, blue), an MLP would take the flattened image as input, pass it through several fully connected (FC) layers and non-linearities, and finally output a vector of probabilities for each of the classes.

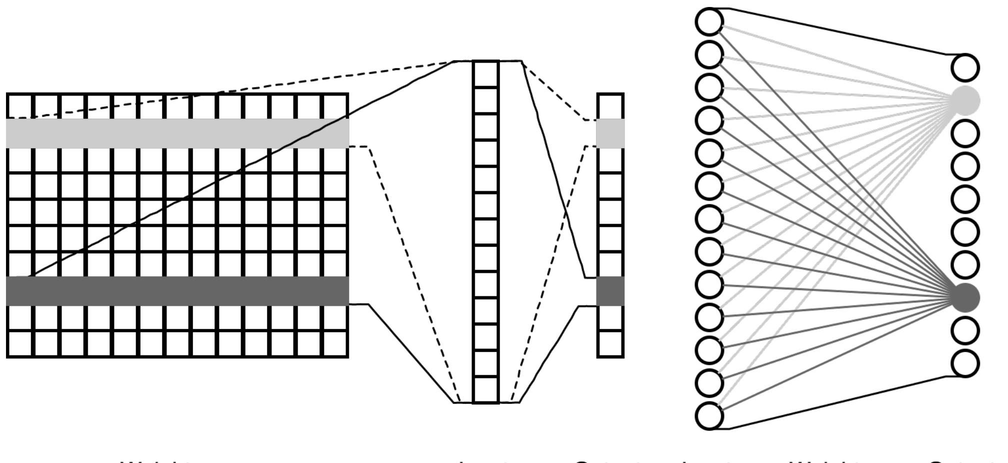


Figure 8.1: A Fully Connected layer connects every input neuron to every output neuron.

Associated with each FC layer is an $n_i \times n_o$ weight matrix that "connects" each of the n_i input neurons to each of the n_o output neurons, hence the term "fully connected layer". The first FC layer takes an image as input, with $n_i = W \times H \times 3$ input neurons. Assuming that there are $n_o \approx n_i$ output neurons, then there are $n_i \times n_o \approx W^2 \times H^2 \times 3^2$ weights — a prohibitively large number of weights (in the millions)! This analysis extends to all FC layers that have large inputs, not just the first layer. In the framework of image classification, MLPs are generally ineffective — not only are they computationally expensive to train (both in terms of time and memory usage), but they also have high variance due to the large number of weights.

Convolutional neural networks (CNNs, or ConvNets) are a different neural network architecture that significantly reduces the number of weights, and in turn reduces variance. Like MLPs, CNNs use FC layers and non-linearities, but they introduce two new types of layers — convolutional and pooling layers. Let's look at these two layers in detail.

Convolutional Layers

A **convolutional layer** takes a $W \times H \times D$ dimensional input $\mathbf{I}$ and *convolves* it with a $w \times h \times D$ dimensional **filter (or kernel) $\mathbf{G}$**. The weights of the filter can be hand designed, but in the context of machine learning we tune them automatically, just like we tune the weights of an FC layer. Mathematically, the convolution operator is defined as

$$(\mathbf{I} * \mathbf{G})[x, y] = \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} \sum_{c \in \{1 \cdots D\}} I_c[x + a, y + b] \cdot G_c[a, b]$$

The subscript in I_c indexes into the depth of the image, in this case for depth c. We can view convolution as either:

1. a 2-D operator over the width/height of the image, "broadcast" over the depth
2. a 3-D operator over the weight/height/depth of the image, with the convolution over the depth spanning the whole image with no room to move.

The output $\mathbf{L} = \mathbf{I} * \mathbf{G}$ is an array of $(W - w + 1) \times (H - h + 1) \times 1$ values.

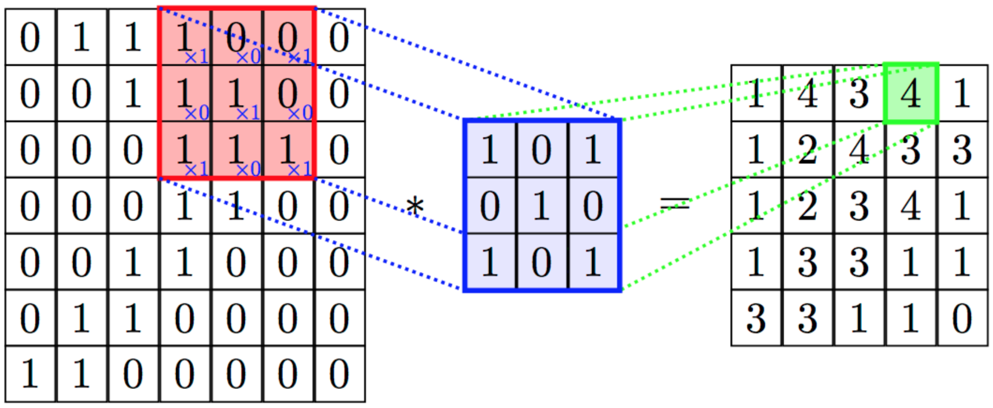


Figure 8.2: Convolving a filter with an image. In this example, we have $W = H = 7, w = h = 3, D = 1$. We extract $n_o = (W - w + 1) \times (H - h + 1) \times 1 = 25$ output values from $n_i = W \times H \times D = 49$ input values, via a filter with $3 \times 3 = 9$ weights.

What exactly is convolution useful for, and why do we use it in the context of image classification? In simple terms, convolutions help us *extract features.* On a low level, filters can be used to detect all kinds of edges in an image, and at a high level they can detect more complex shapes and objects that are critical to classifying an image. Consider a simple horizontal edge detector filter $[1 \quad -1]$. This filter will produce large negative values for inputs in which the left pixel is bright and the right pixel is dark; conversely, it will produce large positive values for inputs in which the left pixel is dark and the right pixel is bright.

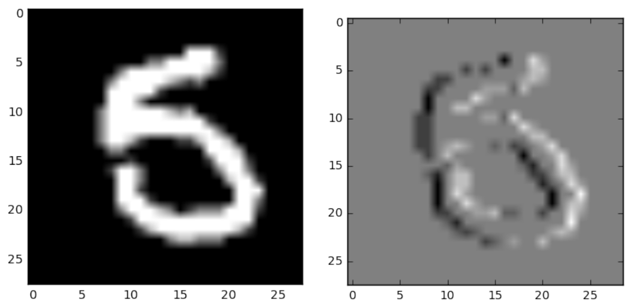


Figure 8.3: Left: a sample image.
Right: the output of convolving the $[1 \quad -1]$ filter about the image.

In general, a filter will produce large positive values in the areas of the image which appear most similar to it. As another example, here is a filter detects edges at a positive 45-degree angle:

$$\begin{bmatrix} 0.6 & 0.2 & 0 \\ 0.2 & 0 & 0.2 \\ 0 & 0.2 & 0.6 \end{bmatrix}$$

How do conv layers compare to FC layers? Let's revisit the example from figure 8.2, where the input is $n_i = 49$ units and the output is $n_o = 25$ units. In an FC layer, we would have used $n_i \times n_o = 1225$ weights, but in our conv layer the filter only has 9 weights! conv layers use a significantly smaller number of weights, which reduces the variance of the model significantly while retaining the expressiveness. This is because we make use of *weight sharing*:

1. the same weights are shared among all the pixels of the input
2. the individual units of the output layer are all determined by the same weights

Compare this to the fully-connected architecture where for each output unit, there is a separate weight to learn for each input-ouput weight. We can illustrate the point for a simple 1-D input-output example.

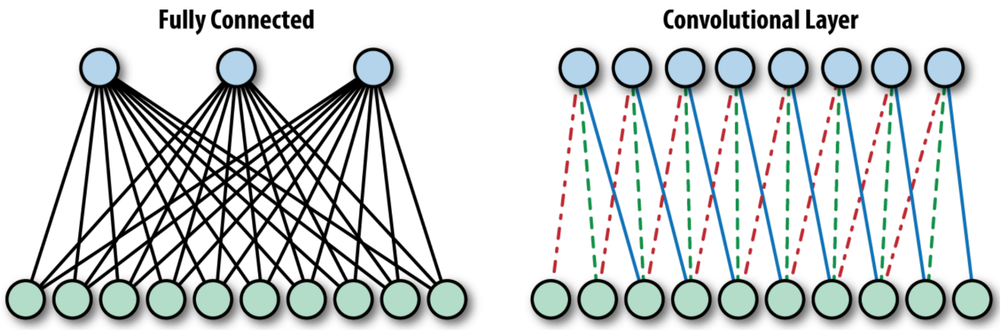


Figure 8.4: FC vs. conv layer. Conv layers are equivalent to FC layers, except that (1) all weights outside the receptive field are 0, and (2) the weights are shared.

This architecture not only decreases the complexity of our model (there are fewer weights), it is actually reasonable for image processing because there are repeated patterns in images — ie. a filter that can detect some kind of pattern in one area of the image can be used elsewhere in the image to detect the same pattern.

In practice, we can apply several different filters to the image to detect different patterns in the input image. For example, we can use a filter that detects horizontal edges, one that detects vertical edges, and another that detects diagonal edges all at once. Given a $W \times H \times D$ input image and k separate $w \times h \times D$ filters, each filter produces an $(W - w + 1) \times (H - h + 1) \times 1$ dimensional output. These individual outputs are stacked together for a $(W - w + 1) \times (H - h + 1) \times k$ combined output.

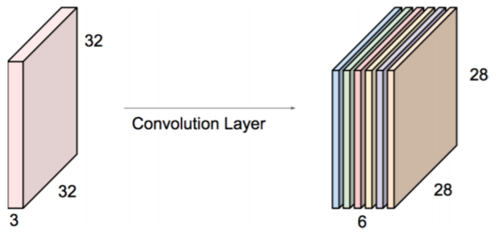
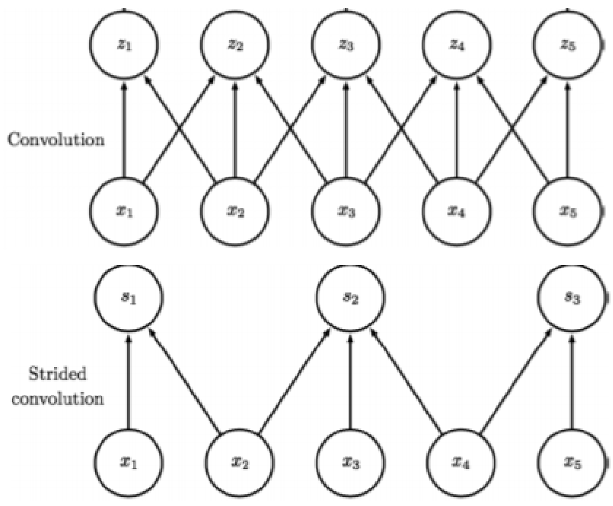


Figure 8.5: Here, we slid 6 independent $5 \times 5 \times 3$ filters across the original image to produce 6 activation maps in the next convolutional layer.

Stacking filters can incur high computational costs, and in order to mitigate this issue, we can stride our filter across the image by multiple pixels instead:


z_1 z_2 z_3 z_4 z_5
Convolution
x_1 x_2 x_3 x_4 x_5
s_1 s_2 s_3
Strided convolution
x_1 x_2 x_3 x_4 x_5

In conjunction to striding, zero-padding the borders of the image is sometimes used to control the exact dimensions of the convolutional layer.

So far, we have introduced convolutional layers as an intuitive and effective approach to extracting features from images, but one potential inspection a disadvantage is that they can only detect "local" features, which is not sufficient to capture complex, global patterns in images. This is not actually the case, because as we stack more convolutional layers, the effective **receptive field** of each successive layer increases. That is, as we go downstream (of the layers), the value of any single unit is informed by an increasingly large patch of the original image. For example, if we use two successive layers of 3×3 filters, any one unit in the first convolutional layer is informed by 9 separate image pixels. Any one unit in the second convolutional layer is informed by 9 separate units of the first convolutional layer, which could informed by up to $9 \times 9 = 81$ original pixels. The increasing receptive field of the successive layers means that the filters in the first few layers extract local low level features, and the later layers extract global high level features.

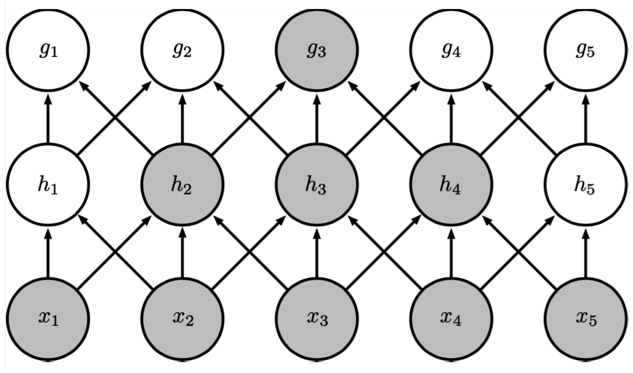
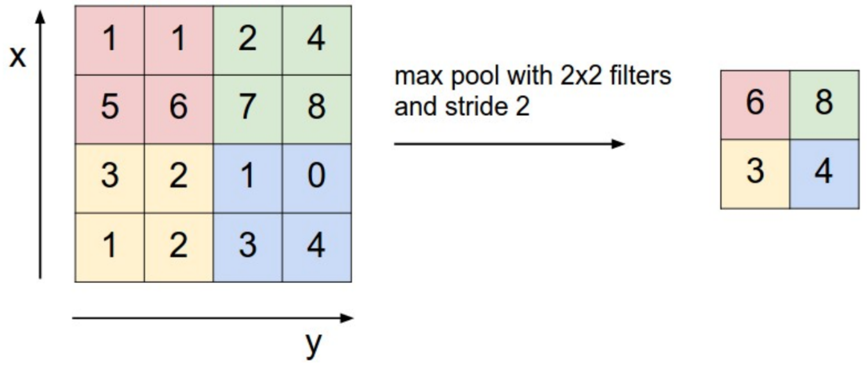


Figure 8.6: The highlighted unit in the downstream layer uses information from all the highlighted units in the input layer.

Pooling Layers

In line with convolutional layers reducing the number of weights in neural networks to reduce variance, **pooling layers** directly reduce the number of neurons in neural networks. The sole purpose of a pooling layer is to *downsample* (also known as *pool*, *gather*, *consolidate*) the previous layer, by sliding a fixed window across a layer and choosing one value that effectively "represents" all of the units captured by the window. There are two common implementations of pooling. In max-pooling, the representative value just becomes the largest of all the units in the window, while in average-pooling, the representative value is the average of all the units in the window. In practice, we stride pooling layers across the image with the stride equal to the size of the pooling layer. None of these properties actually involve any weights, unlike fully connected and convolutional layers.



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

Figure 8.7: Max-pooling layer

Orthogonal to the choice between max and average pooling is option between spatial and cross-channel pooling. Spatial pooling pools values within the same channel, which induces translational invariance in our model and adding generalization capabilities. In the following figure, we can see that even though the input layer of the right image is a translated version of the input layer of the left image, due to spacial pooling the next layer looks more or less the same.

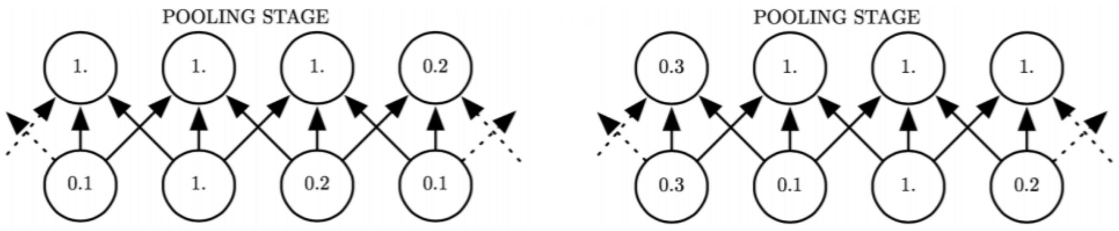


Figure 8.8: Spatial pooling

Cross-channel pooling pools values across different channels, which induces transformational invariance in our model, again adding generalization capabilities. To illustrate the point, consider an example with a convolutional layer represented by 3 filters. Suppose each can detect the number 5 in some degree of rotation. If we pooled across the three channels determined by these filters, then no matter what orientation of the number "5" we got as input to our CNN, the pooling layer would have a large response!

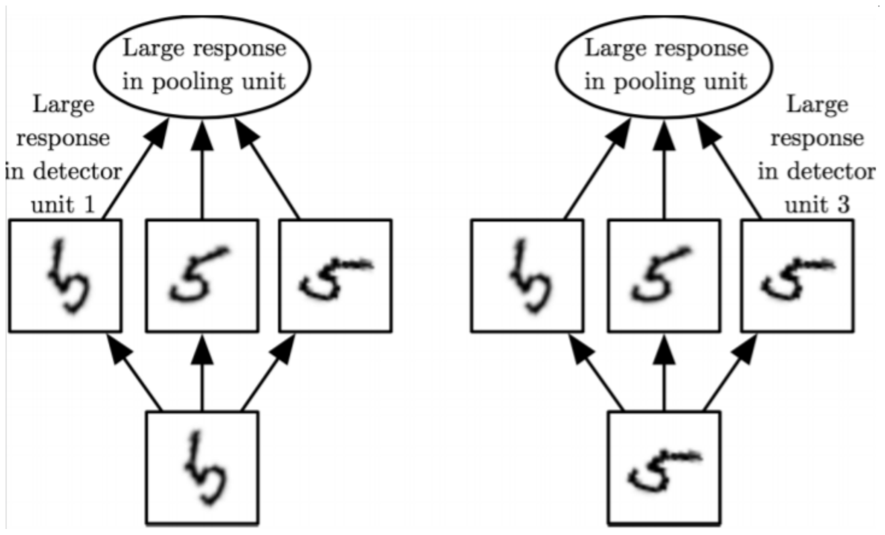

Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

Figure 8.9: Cross-channel pooling

Backpropagation for CNNs

Just like MLPs, we can use the Backpropagation algorithm to train CNNs as well. We simply have to compute partial derivatives for conv and pool layers, just as we did for FC layers and non-linearities.

Derivatives for conv layers

Let's denote the error function as f. In classification tasks, this is typically cross entropy loss. The forward pass will gives us the input I to the CNN. The backward pass will compute the partial derivatives of the the error f with respect the output of the layer L, $\frac{\partial f}{\partial L}$. Without additional knowledge about the error function after L, we can compute the derivatives with respect to elements in the input I and filter G using the chain rule. Specifically, we have

$$\frac{\partial f}{\partial G_c[x,y]} = \frac{\partial f}{\partial L}\frac{\partial L}{\partial G_c[x,y]}$$

and

$$\frac{\partial f}{\partial I_c[x,y]} = \frac{\partial f}{\partial L}\frac{\partial L}{\partial I_c[x,y]}$$

where $G_c[x,y]$ denotes the entry in the filter for color c at position (x,y) and similarly for $I_c[x,y]$.

From the equation for discrete convolution, we can compute the derivatives for each entry (i,j) in L as

$$\begin{aligned}\frac{\partial L[i,j]}{\partial G_c[x,y]} &= \frac{\partial}{\partial G_c[x,y]}\sum_{a=1}^{w}\sum_{b=1}^{h}\sum_{c\in\{r,g,b\}} I_c[i+a,j+b]\cdot G_c[a,b]\\ &= I_c[i+x,j+y]\end{aligned}$$

For the input image, we similarly compute the derivative as

$$\begin{aligned}\frac{\partial L[i,j]}{\partial I_c[x,y]} &= \frac{\partial}{\partial I_c[x,y]}\sum_{a=1}^{w}\sum_{b=1}^{h}\sum_{c\in\{r,g,b\}} I_c[i+a,j+b]\cdot G_c[a,b]\\ &= G_c[x-i,y-j]\end{aligned}$$

where we have $i+a=x$ and $j+b=y$. When $x-i$ or $y-j$ go outside the boundary of the filter, we can treat the derivative as zero.

We can collect the derivatives of the filter parameter for all $L[i,j]$ into a vector and multiply it by the derivatives we computed to get

$$\frac{\partial f}{\partial G_c[x,y]} = \frac{\partial f}{\partial L}\cdot\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}\frac{\partial L[i,j]}{\partial G_c[x,y]}] = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}I_c[i+x,j+y]$$

and for the image

$$\frac{\partial f}{\partial I_c[x,y]} = \frac{\partial f}{\partial L}\cdot\frac{\partial L}{\partial I_c[x,y]} = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}\frac{\partial L[i,j]}{\partial I_c[x,y]} = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}G_c[x-i,y-j]$$

Derivatives for pool layers

Since pooling layers do not involve any weights, we only need to calculate partial derivatives with respect to the input:

$$\frac{\partial f}{\partial I_c[x,y]}$$

Through the chain rule, we have that

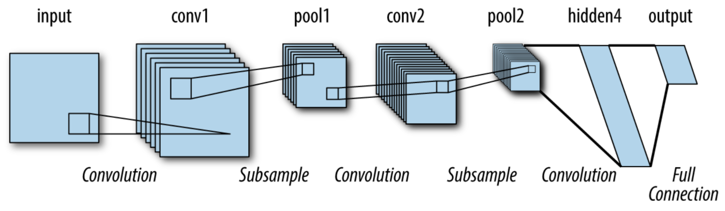
$$\frac{\partial f}{\partial I_c[x,y]} = \frac{\partial f}{\partial L}\cdot\frac{\partial L}{\partial I_c[x,y]}$$

and now the problem entails finding $\frac{\partial L}{\partial I_c[x,y]}$. Computing this derivative depends on the stride, orientation, and nature of the pooling, but in the case of max-pooling the output is simply a maximum of inputs: $L=\max(I_1,I_2,...,I_n)$ and in this case, we have $\frac{\partial L}{\partial I_j} = \mathbb{1}(I_j = \max(I_1,I_2,...,I_n))$.

8.2 CNN Architectures

Convolutional Neural Networks were first applied successfully to the ImageNet challenge in 2012 and continue to outperform computer vision techniques that do not use neural networks. Here are a few of the architectures that have been developed over the years.

LeNet (LeCun et al, 1998)



Key characteristics:

- Used to classify handwritten alphanumeric characters

AlexNet (Krizhevsky et al, 2012)

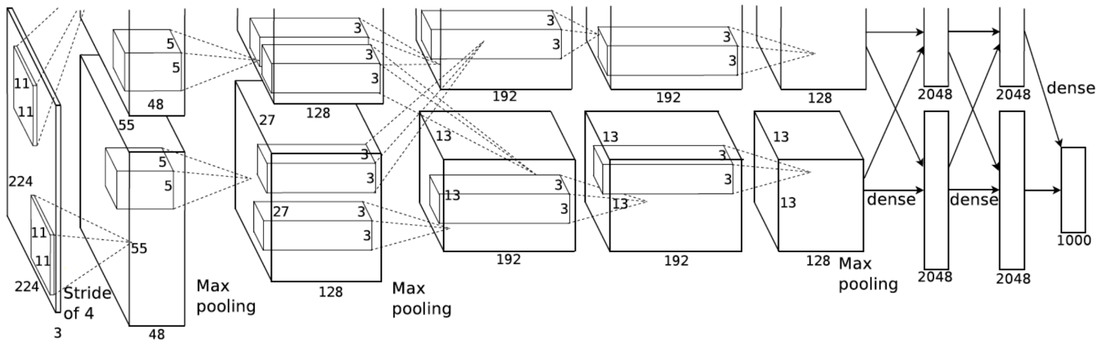


Figure 8.10: AlexNet architecture. Reference: “ImageNet Classification with Deep Convolutional Neural Networks,” NIPS 2012.

Key characteristics:

- Conv filters of varying sizes - for example, the first layer has 11×11 conv filters
- First use of ReLU, which fixed the problem of saturating gradients in the predominant tanh activation.
- Several layers of convolution, max pooling, some normalization. Three fully connected layers at the end of the network (these comprise the majority of the weights in the network).
- Around 60 million weights, over half of which are in the first fully connected layer following the last convolution.

- Trained over two GPU's — the top and bottom divisions in Figure 8.10 were due to the need to separate training onto two GPU's. There was limited communication between the GPU's, as illustrated by the arrows that go between the top and bottom.
- Dropout in first two FC layers — prevents overfitting
- Heavy data augmentation. One form is image translation and reflection: for example, an elephant facing the left is the same class as an elephant facing the right. The second form is altering the intensity of RGB color channels: different cameras can have different lighting on the same objects, so it is necessary to account for this.

VGGNet (Simonyan and Zisserman, 2014)

Reference paper: "Very Deep Convolutional Networks for Large-Scale Image Recognition," ICLR 2015.[1] Key characteristics:

- Only uses 3×3 convolutional filters. Blocks of conv-conv-conv-pool layers are stacked together, followed by fully connected layers at the end (the number of convolutional layers between pooling layers can vary). Note that a stack of 3 3×3 conv filters has the same effective receptive field as one 7×7 conv filter. To see this, imagine sliding a 3×3 filter over a 7×7 image - the result is a 5×5 image. Do this twice more and the result is a 1×1 cell - sliding one 7×7 filter over the original image would also result in a 1×1 cell. The computational cost of the 3×3 filters is lower - a stack of 3 such filters over C channels requires $3 * (3^2C)$ weights (not including bias weights), while one 7×7 filter would incur a higher cost of 7^2C learned weights. Deeper, more narrow networks can introduce more non-linearities than shallower, wider networks due to the repeated composition of activation functions.

GoogLeNet (Szegedy et al, 2014)

Also codenamed as "Inception."[2] Published in CVPR 2015 as "Going Deeper with Convolutions." Key characteristics:

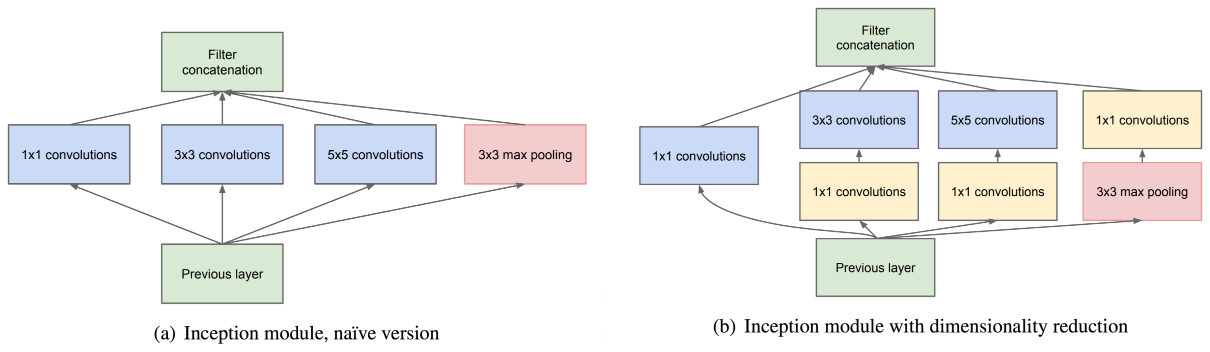


Figure 8.11: Inception Module

[1]VGG stands for the "Visual Geometry Group" at Oxford where this was developed.

[2]"In this paper, we will focus on an efficient deep neural network architecture for computer vision, codenamed Inception, which derives its name from the Network in network paper by Lin et al [12] in conjunction with the famous we need to go deeper internet meme [1]." The authors seem to be meme-friendly.

- Deeper than previous networks (22 layers), but more computationally efficient (5 million parameters - no fully connected layers).
- Network is composed of stacked sub-networks called "Inception modules." The naive Inception module (a) runs convolutional layers in parallel and concatenates the filters together. However, this can be computationally inefficient. The dimensionality reduction Inception module (b) performs 1×1 convolutions that act as dimensionality reduction. This lowers the computational cost and makes it tractable to stack many Inception modules together.

ResNet (He et al, 2015)

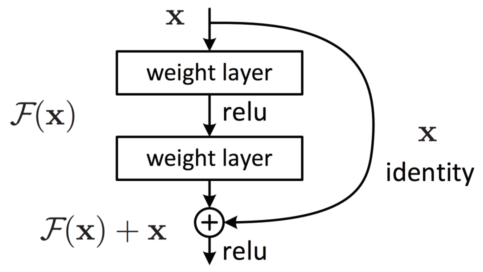


Figure 8.12: Building block for the ResNet from "Deep Residual Learning for Image Recognition," CVPR 2016. If the desired function to be learned is $\mathcal{H}(x)$, we instead learn the residual $\mathcal{F}(x) := \mathcal{H}(x) - x$, so the output of the network is $\mathcal{F}(x) + x = \mathcal{H}(x)$.

Key characteristics:

- Very deep (152 layers). Residual blocks (Figure 8.12) are stacked together - each individual weight layer in the residual block is implemented as a 3×3 convolution. There are no FC layers until the final layer.
- Residual blocks solve the "vanishing gradient" problem: the gradient signal diminishes in layers that are farther away from the end of the network. Let L be the loss, Y be the output at a layer, x be the input. Regular neural networks have gradients that look like

  $$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial x}$$

  but the derivative of Y with respect to x can be small. If we use a residual block where $Y = F(x) + x$, we have

  $$\frac{\partial Y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

  The $+x$ term in the residual block always provides some default gradient signal so the signal is still backpropagated to the front of the network. This allows the network to be very deep.

To conclude this section, we note that the winning ImageNet architectures have all increased in depth over the years. While both shallow and deep neural networks are known to be universal function approximators, there is growing empirical and theoretical evidence that deep neural networks can require fewer (even exponentially fewer) parameters than shallow nets to achieve the same approximation performance. There is also evidence that deep neural networks possess better generalization capabilities than their shallow counterparts. The performance, generalization, and optimization benefits of adding more layers is an ongoing component of theoretical research.

8.3 Visualizing and Understanding CNNs

We know that a convolutional net learns features, but these may not be directly useful to visualize. There are several methods available that enable us to better understand what convolutional nets actually learn. These include:

- Visualizing filters - can give an idea of what types of features the network learns, such as edge detectors. This only works in the first layer. Visualizing activations - can see sparsity in the responses as the depth increases. One can also visualize the feature map before a fully connected layer by conducting a nearest neighbor search in feature space. This helps to determine if the features learned by the CNN are useful - for example, in pixel space, an elephant on the left side of the image would not be a neighbor of an elephant on the right side of the image, but in a translation-invariant feature space these pictures might be neighbors.
- Reconstruction by deconvolution - isolate an activation and reconstruct the original image based on that activation alone to determine its effect.
- Activation maximization - Hubel and Wiesel's experiment, but computationally
- Saliency maps - find what locations in the image make a neuron fire
- Code inversion - given a feature representation, determine the original image
- Semantic interpretation - interpret the activations semantically (for example, is the CNN determining whether or not an object is shiny when it is trying to classify?)