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

March 13, 2018
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.

We owe gratitude to Professor Anant Sahai and Professor Stella Yu, 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 at http://snasiriany.me/cs189/. Please report any mistakes to snasiriany@berkeley.edu. Please contact the authors if you wish to redistribute this document.

Notation

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

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).
5.4 QDA and LDA ............................................................... 98
5.5 Support Vector Machines ............................................ 102

6 Duality, Nearest Neighbors, Sparsity .............................. 111
6.1 Duality ................................................................. 111
6.2 The Dual of SVMs ..................................................... 117
6.3 Nearest Neighbor Classification ................................... 120
6.4 Sparsity and LASSO ................................................... 125
6.5 Coordinate Descent .................................................... 127
6.6 Sparse Least-Squares .................................................. 129

7 Decision Tree Learning ................................................. 131
7.1 Decision Trees ......................................................... 131
7.2 Random Forests ....................................................... 134
7.3 Boosting ............................................................... 134

8 Deep Learning ............................................................ 139
8.1 Convolutional Neural Nets .......................................... 139
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 $y = f(x)$, where $y \in \mathbb{R}$ is some quantity that can be predicted from an input $x \in \mathbb{R}^d$, 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 $\hat{y} = h(x)$, called the hypothesis, that should approximate $f$. We assume that we have access to a dataset $D = \{(x_i, y_i)\}_{i=1}^n$, where each pair $(x_i, y_i)$ 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 $w \in \mathbb{R}^d$, the elements of which are known as parameters or weights, that controls the behavior of the function. That is,

$$h_w(x) = g(x, w)$$

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

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

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:

$$w^* = \arg \min_w L(w)$$

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 $h_w$ to be of the form $h_w(x) = \mathbf{x}^\top w$. We want

$$y_i \approx \hat{y}_i = h_w(x_i) = \mathbf{x}_i^\top w$$
for each $i = 1, \ldots, n$. This set of equations can be written in matrix form as

$$
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_n
\end{bmatrix} \approx \begin{bmatrix}
x_1^\top \\
x_2^\top \\
\vdots \\
x_n^\top
\end{bmatrix} \begin{bmatrix}
w_1 \\
w_2 \\
\vdots \\
w_d
\end{bmatrix}
$$

In words, the matrix $X \in \mathbb{R}^{n \times d}$ has the input datapoint $x_i$ as its $i$th 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 $y = Xw$ (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:

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

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 $w_{ols}^*$.

**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 $w^*$ satisfies $\nabla L(w^*) = 0$. In the OLS case,

$$
L(w) = \|Xw - y\|_2^2 \\
= (Xw - y)^\top (Xw - y) \\
= (Xw)^\top Xw - (Xw)^\top y - y^\top Xw + y^\top y \\
= w^\top X^\top Xw - 2w^\top X^\top y + y^\top y
$$

Using the following results from matrix calculus

$$
\nabla_x (a^\top x) = a \\
\nabla_x (x^\top Ax) = (A + A^\top)x
$$

the gradient of $L$ is easily seen to be

$$
\nabla L(w) = \nabla_w (w^\top X^\top Xw - 2w^\top X^\top y + y^\top y) \\
= \nabla_w (w^\top X^\top Xw) - 2\nabla_w (w^\top X^\top y) + \nabla_w (y^\top y) \\
= 2X^\top Xw - 2X^\top y
$$

where in the last line we have used the symmetry of $X^\top X$ to simplify $X^\top X + (X^\top X)^\top = 2X^\top X$.

Setting the gradient to 0, we conclude that any optimum $w_{ols}^*$ satisfies

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

\[
w_{\text{ols}}^* = (X^\top X)^{-1}X^\top y
\]

Note: Although we write \((X^\top X)^{-1}\), 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 \( \nabla L(w^*) = 0 \), 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

\[
\nabla^2 L(w) = 2X^\top X
\]

and show that this is positive semi-definite:

\[
\forall w, \quad w^\top(2X^\top X)w = 2(Xw)^\top Xw = 2\|Xw\|_2^2 \geq 0
\]

**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 \( v \in V \) can be decomposed uniquely in the form

\[
v = v_S + v_\perp
\]

where \( v_S \in S \) and \( v_\perp \in S^\perp \). 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 \( v \) to \( v_S \) in the decomposition above. An important property of the orthogonal projection is that

\[
\|v - P_Sv\| \leq \|v - s\|
\]

for all \( s \in S \), with equality if and only if \( s = P_Sv \). That is,

\[
P_Sv = \arg \min_{s \in S} \|v - s\|
\]

**Proof.** By the Pythagorean theorem,

\[
\|v - s\|^2 = \left\| \underbrace{v - P_Sv}_{\in S} + \underbrace{P_Sv - s}_{\in S} \right\|^2 = \|v - P_Sv\|^2 + \|P_Sv - s\|^2 \geq \|v - P_Sv\|^2
\]

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

Here is a visual representation of the argument above:
In the OLS case,

\[ w_{\text{OLS}}^* = \arg \min_w \|Xw - y\|_2^2 \]

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

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

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

To solve for \(w_{\text{OLS}}^*\), we need the following fact\(^1\): \[ \text{null}(X^\top) = \text{range}(X)^\perp \] Since we are projecting onto \(\text{range}(X)\), the orthogonality condition for optimality is that \(y - Py \perp \text{range}(X)\), i.e. \(y - Xw_{\text{OLS}}^* \in \text{null}(X^\top)\). This leads to the equation

\[ X^\top(y - Xw_{\text{OLS}}^*) = 0 \]

which is equivalent to

\[ X^\top Xw_{\text{OLS}}^* = X^\top y \]

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 $X$ 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 $X$:

$$X = U \Sigma V^T$$

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

$$X = U \begin{bmatrix} \Sigma_d & 0 \\ 0 & 0 \end{bmatrix} V^T$$

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

$$(X^TX)^{-1} = (V \begin{bmatrix} \Sigma_d & 0 \\ 0 & 0 \end{bmatrix} U^T \begin{bmatrix} \Sigma_d & 0 \\ 0 & 0 \end{bmatrix} V^T)^{-1}$$

$$= (V \begin{bmatrix} \Sigma_d & 0 \\ 0 & 0 \end{bmatrix} I \begin{bmatrix} \Sigma_d & 0 \\ 0 & 0 \end{bmatrix} V^T)^{-1}$$

$$= (V \Sigma_d^2 V^T)^{-1} = (V^T)^{-1}(\Sigma_d^{-2})^{-1}V^{-1} = V \Sigma_d^{-2} V^T$$

This means that $(X^TX)^{-1}$ will have singular values that are the squared inverse of the singular values of $X$, potentially leading to extremely large singular values when the singular value of $X$ 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 $w$ solution would prevent OLS from generalizing to unseen data.

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

$$\min_w \|Xw - y\|_2^2 + \lambda \|w\|_2^2$$

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

$$L(w) = \|Xw - y\|_2^2 + \lambda \|w\|_2^2$$

$$= w^T X^T X w - 2w^T X^T y + y^T y + \lambda w^T w$$

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

$$\nabla_w L(w) = 0$$

$$2X^T X w - 2X^T y + 2\lambda w = 0$$

$$(X^T X + \lambda I)w = X^T y$$

$$w^{*_{\text{ridge}}} = (X^T X + \lambda I)^{-1}X^T y$$

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

$$\nabla^2 L(w) = 2 X^T X + 2\lambda I$$
and show that this is positive definite (PD):

\[ \forall w \neq 0, \ w^\top (X^\top X + \lambda I) w = (Xw)^\top Xw + \lambda w^\top w = \|Xw\|^2 + \lambda \|w\|^2 > 0 \]

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 \( X^\top X \) 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 \( X^\top Xw = X^\top y \), because the range of \( X^\top X \) and the range space of \( X^\top \) are equivalent; since \( X^\top y \) lies in the range of \( X^\top \), it must equivalently lie in the range of \( X^\top X \) and therefore there always exists a \( w \) that satisfies the equation \( X^\top Xw = X^\top y \).)

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

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

Now with our slight tweak, the matrix \( X^\top X + \lambda I \) has become full rank and thus invertible. The singular values have become \( \frac{1}{\sigma^2_r + \lambda} \) 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 \( x \), 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
\[
\min_w \| Xw - y \|^2
\]
represents the “best-fit” linear model, by projecting \( y \) onto the subspace spanned by the columns of \( X \). However, the true input-output relationship \( y = f(x) \) 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 \( \phi : \mathbb{R}^\ell \to \mathbb{R}^d \), called a feature map, that maps each raw data point \( x \in \mathbb{R}^\ell \) into a vector of features \( \phi(x) \). The hypothesis function then writes
\[
h_w(x) = \sum_{j=1}^d w_j \phi_j(x) = w^\top \phi(x)
\]
Note that the resulting model is still linear with respect to the features, but it is nonlinear with respect to the original data if \( \phi \) is nonlinear. The component functions \( \phi_j \) 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 \( x \) as features (i.e. \( \phi_j(x) = x_j \)), 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 \( w \), just as before. To do this, we replace the original data matrix \( X \in \mathbb{R}^{n \times \ell} \) by \( \Phi \in \mathbb{R}^{n \times d} \), which has \( \phi(x_i)^\top \) as its \( i \)th row:
\[
\min_w \| \Phi w - y \|^2
\]

Example: Fitting Ellipses

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

Assume that we have \( n \) data points \( D = \{(x_{1i}, x_{2i})\}_{i=1}^n \), 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:
\[
\min_w \| \Phi w - 1 \|^2
\]
where
\[
\Phi = \begin{bmatrix}
    x_{1,1} & x_{2,1} & x_{1,1} x_{2,1} & x_{1,1} & x_{2,1} \\
    x_{1,2} & x_{2,2} & x_{1,2} x_{2,2} & x_{1,2} & x_{2,2} \\
    \vdots & \vdots & \vdots & \vdots & \vdots \\
    x_{1,n} & x_{2,n} & x_{1,n} x_{2,n} & x_{1,n} & x_{2,n}
\end{bmatrix}
\]
In this case, the feature map \( \phi \) is given by
\[
\phi(x) = (x_1^2, x_2^2, x_1 x_2, x_1, x_2)
\]
Note that there is no “target” vector \( y \) 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 a linear combination of monomial basis terms. Monomials can be classified in two ways, by their degree and dimension:

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Degree</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (univariate)</td>
<td></td>
<td>1</td>
<td>x</td>
<td>$x^2$</td>
<td>$x^3$</td>
<td>...</td>
</tr>
<tr>
<td>2 (bivariate)</td>
<td></td>
<td>1</td>
<td>$x_1,x_2$</td>
<td>$x_1^2,x_2^2,x_1x_2$</td>
<td>$x_1^3,x_2^3,x_1^2x_2,x_1x_2^2$</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

A big reason we care polynomial features is that any smooth function can be approximated arbitrarily closely by some polynomial.\textsuperscript{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\textsuperscript{3} to show that a polynomial of degree $d$ in $\ell$ 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:

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$d$</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>10</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>20</td>
<td>56</td>
<td>286</td>
<td>3276</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>56</td>
<td>252</td>
<td>3003</td>
<td>142506</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>286</td>
<td>3003</td>
<td>184756</td>
<td>183579396</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>3276</td>
<td>142506</td>
<td>183579396</td>
<td>126410606437752</td>
<td></td>
</tr>
</tbody>
</table>

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

$$h_w(x) = \sum_{j=1}^{d} w_j \phi_j(x) = w^T \phi(x)$$

\textsuperscript{2} Taylor’s theorem gives more precise statements about the approximation error.

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

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

The number of unique ways to arrange these stars and bars is the number of ways to choose the positions of the $\ell$ bars out of the total $\ell + d$ slots, i.e. $\ell + d$ choose $\ell$. (You could also pick the positions of the $d$ stars out of the total $\ell + d$ slots; the expression is symmetric in $\ell$ 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 $\ell^2$ penalty on the parameters $w$:

$$\min_w \|Xw - y\|^2 + \lambda\|w\|^2_2$$

The regularization weight $\lambda$ is also a hyperparameter, as it is fixed during the minimization above. However $\lambda$, 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 $D$, and that we have a loss function $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, which is to measure the error between the true output $y$ and our estimate $\hat{y} = h(x)$. The risk (or true error) of a particular hypothesis $h \in H$ is the expected loss over the whole data distribution:

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

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

$$h^* = \arg\min_{h \in H} R(h)$$

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

$$\hat{R}_{\text{train}}(h) = \frac{1}{n} \sum_{i=1}^n \ell(h(x_i), y_i)$$

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
\[ \hat{R}_{\text{val}}(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(x_i^{\text{val}}), y_i^{\text{val}}) \]

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 \( \lambda \) 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 \( \lambda \) 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 \( \lambda \) grows:
1.4. HYPERPARAMETERS AND VALIDATION

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 $i = 1, \ldots, k$,
   - Train the model on all the data except block $i$.
   - Evaluate the model (i.e. compute the validation error) using block $i$.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>train</td>
<td>validate</td>
<td>train</td>
<td>validate</td>
<td>train</td>
<td>validate</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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:

\[
\hat{w}_{\text{OLS}} = \arg\min_w \| y - Xw \|_2^2
\]

\[
\hat{w}_{\text{RIDGE}} = \arg\min_w \| y - Xw \|_2^2 + \lambda \| w \|_2^2
\]

One question that arises is why we specifically use the \( \ell^2 \) 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:

\( f : x \rightarrow f(x) \)

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

\[ \mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n} \]

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

\[ Y_i = f(x_i) + Z_i \]

We assume that \( x_i \) is a fixed value (which implies that \( f(x_i) \) 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

\[ Y_i \overset{\text{iid}}{\sim} \mathcal{N}(f(x_i), \sigma^2) \]
Now that we have defined the model and data, we wish to find a hypothesis model $h_{\theta}$ (parameterized by $\theta$) 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 $\theta$, we can express the problem as

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

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

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

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 $f(x_i)$). We can further simplify the problem by working with the log likelihood $\ell(\theta; \mathbf{X}, \mathbf{y}) = \log L(\theta; \mathbf{X}, \mathbf{y})$

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

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:

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

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 $p(y_i \mid \mathbf{x}_i, \theta)$ comes from a Gaussian

$$Y_i \mid \theta \sim \mathcal{N}(h_{\theta}(x_i), \sigma^2)$$

Continuing with logs:

$$\hat{\theta}_{\text{MLE}} = \arg \max_{\theta} \ell(\theta; \mathbf{X}, \mathbf{y}) = \arg \max_{\theta} \sum_{i=1}^{n} \log[p(y_i \mid \mathbf{x}_i, \theta)]$$

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

$$= \arg \max_{\theta} -\left(\sum_{i=1}^{n} \frac{(y_i - h_{\theta}(x_i))^2}{2\sigma^2}\right) - n \log \sqrt{2\pi\sigma}$$
2.1. MLE AND MAP FOR REGRESSION (PART I)

\[ \hat{\theta}_{\text{MLE}} = \arg \min_{\theta} \sum_{i=1}^{n} \left( \frac{(y_i - h_{\theta}(x_i))^2}{2\sigma^2} \right) + n \log \sqrt{2\pi\sigma} \] (2.4)

\[ \hat{\theta}_{\text{MLE}} = \arg \min_{\theta} \sum_{i=1}^{n} (y_i - h_{\theta}(x_i))^2 \] (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 \( h_{\theta}(x_i) = x_i^\top \theta \), where \( \theta \in \mathbb{R}^d \), where \( d \) is the number of dimensions of our featurized datapoints. For this specific setting, the problem becomes:

\[ \hat{\theta}_{\text{MLE}} = \arg \min_{\theta \in \mathbb{R}^d} \sum_{i=1}^{n} (y_i - x_i^\top \theta)^2 \]

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:

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

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.

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

\[ = \arg \max_{\theta} \frac{p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\theta}) \cdot p(\text{true model} = h_{\theta})}{p(\text{data} = \mathcal{D})} \]

\[ = \arg \max_{\theta} p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\theta}) \cdot p(\text{true model} = h_{\theta}) \]

\[ = \arg \max_{\theta} \log p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\theta}) + \log p(\text{true model} = h_{\theta}) \]

\[ = \arg \min_{\theta} - \log p(\text{data} = \mathcal{D} \mid \text{true model} = h_{\theta}) - \log p(\text{true model} = h_{\theta}) \]

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 \( p(\text{true model} = h_{\theta}) \) 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

\[ \hat{\theta}_{\text{MAP}} = \arg \min_{\theta} - \left( \sum_{i=1}^{n} \log[p(y_i \mid x_i, \theta)] \right) - \log[p(\theta)] \]
Again, just as in MLE, notice that we implicitly condition on the $x_i$’s because we treat them as constants. Also, let us assume as before that the noise terms are i.i.d. Gaussians: $N_i \sim N(0, \sigma^2)$. For the prior term $P(\Theta)$, we assume that the components $\theta_j$ are i.i.d. Gaussians:

$$\theta_j \sim N(\theta_{j0}, \sigma_h^2)$$

Using this specific information, we now have:

$$\hat{\Theta}_{\text{MAP}} = \arg \min_{\Theta} \left( \frac{\sum_{i=1}^{n} (y_i - h_\theta(x_i))^2}{2\sigma^2} + \frac{\sum_{j=1}^{d} (\theta_j - \theta_{j0})^2}{2\sigma_h^2} \right)$$

$$= \arg \min_{\Theta} \left( \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 \right) + \frac{\sigma^2}{\sigma_h^2} \left( \sum_{j=1}^{d} (\theta_j - \theta_{j0})^2 \right)$$

Let’s look again at the case for linear regression to illustrate the effect of the prior term when $\theta_{j0} = 0$. In this context, we refer to the linear hypothesis function $h_\theta(x) = \theta^\top x$.

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

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:
2.1. MLE AND MAP FOR REGRESSION (PART I)

The diagram above shows 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 of 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 \( x \) the corresponding measurement \( Y \) is a noisy measurement of the true underlying response \( f(x) \):

\[
Y = f(x) + Z
\]

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 a 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:

\[
\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}
\]

In that context, we treated the \( x_i \)'s in our dataset \( \mathcal{D} \) as fixed values. In this case however, we treat the \( x_i \)'s as values sampled from random variables \( X_i \). That is, \( \mathcal{D} \) is a random variable, consisting of random variables \( X_i \) and \( Y_i \). For some arbitrary test input \( x \), \( h(x; \mathcal{D}) \) 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 \( h(x; \mathcal{D}) \) every time we use a new dataset. Note that \( \mathcal{x} \) and \( \mathcal{D} \) are completely independent from one another — \( x \) is a test point, while \( \mathcal{D} \) consists of the training data.

**Metric**

Our objective is to, for a fixed test point \( x \), evaluate how closely the hypothesis can estimate the noisy observation \( Y \) corresponding to \( x \). Note that we have denoted \( x \) 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 \( x \) and \( Y \) have no relation to the set of \( X_i \)'s and \( Y_i \)'s in \( \mathcal{D} \). Again, we can view \( \mathcal{D} \) as the training data, and \( (x, Y) \) as a test point — the test point \( x \) 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 \( Y = f(x) + Z \):

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

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

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

Note that the error is w.r.t the observation \( Y \) and not the true underlying model \( f(x) \), 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 $Y$:

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

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

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:

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

$$
= \left( \text{Var}(h(x; D)) + \mathbb{E}[h(x; D)^2] \right) + \left( \text{Var}(Y) + \mathbb{E}[Y]^2 \right) - 2\mathbb{E}[h(x; D)] \cdot \mathbb{E}[Y]
$$

$$
= \left( \mathbb{E}[h(x; D)^2] - 2\mathbb{E}[h(x; D)] \cdot \mathbb{E}[Y] + \mathbb{E}[Y]^2 \right) + \text{Var}(h(x; D)) + \text{Var}(Y)
$$

$$
= \left( \mathbb{E}[h(x; D)] - \mathbb{E}[Y] \right)^2 + \text{Var}(h(x; D)) + \text{Var}(Y)
$$

$$
= \underbrace{\left( \mathbb{E}[h(x; D)] - f(x) \right)^2}_{\text{bias}^2 \text{ of method}} + \underbrace{\text{Var}(h(x; D))}_{\text{variance of method}} + \underbrace{\text{Var}(Z)}_{\text{irreducible error}}
$$

Recall that for any two independent random variables $D$ and $Y$, $g_1(D)$ and $g_2(Y)$ are also independent, for any functions $g_1, g_2$. This implies that $h(x; D)$ and $Y$ are independent, allowing us to express $\mathbb{E}[h(x; D) \cdot Y] = \mathbb{E}[h(x; D)] \cdot \mathbb{E}[Y]$ 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 $f(x)$, for a fixed value of $x$. A low bias means that on average the regressor $h(x)$ accurately estimates $f(x)$.

- **Variance of method**: Measures the variance of the hypothesis (over all possible training sets), for a fixed value of $x$. 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 $f(x)$ very well, but have very high variance and are very far off from $f(x)$ on an individual basis.

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

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 \( E[(Z - Y)^2] = E[((Z - E[Z]) + (E[Z] - Y))^2] \) which decomposes to easily give us the variance of \( Z \) and other terms.

\[
\varepsilon(x; h) = E[(h(x; D) - Y)^2] \\
= E\left[ (h(x; D) - E[h(x; D)]) + E[h(x; D)] - Y \right]^2 \\\n= E\left[ (h(x; D) - E[h(x; D)])^2 \right] + E\left[ (E[h(x; D)] - Y)^2 \right] + 2E\left[ (h(x; D) - E[h(x; D)]) \cdot (E[h(x; D)] - Y) \right] \\
= E\left[ (h(x; D) - E[h(x; D)])^2 \right] + E\left[ (E[h(x; D)] - Y)^2 \right] + 2E[h(x; D) \cdot E[h(x; D)]] \cdot E[h(x; D)] - Y \right] \\
= E\left[ (h(x; D) - E[h(x; D)])^2 \right] + E\left[ (E[h(x; D)] - Y)^2 \right] \\
= \text{Var}((h(x; D)) + E\left[ (E[h(x; D)] - Y)^2 \right] \\
= \text{Var}((h(x; D)) + E\left[ (E[h(x; D)] - Y)^2 \right] + E\left[ (Y - E[Y])^2 \right] + 2(E[h(x; D)] - E[Y]) \cdot E[E[Y] - Y] \\
= \text{Var}((h(x; D)) + E\left[ (E[h(x; D)] - Y)^2 \right] + E\left[ (Y - E[Y])^2 \right] \\
= \text{Var}((h(x; D)) + (E[h(x; D)] - E[Y])^2 + \text{Var}(Y) \\
= \text{Var}((h(x; D)) + (E[h(x; D)] - f(x))^2 + \text{Var}(Z) \\
= (E[h(x; D)] - f(x))^2 + \text{Var}(h(x; D)) + \text{Var}(Z) \\
\]

bias\(^2\) of method \quad \text{variance of method} \quad \text{irreducible error}

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

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 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 x.

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

The bias is now very low bias for all x’s. The variance is low for 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.

Fitting A Model over Multiple Datasets: $p = 2$

Bias and Variance in Model Selection: $p = 2$

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
2.2. **BIAS-VARIANCE TRADEOFF**

variance over all values of $x$, 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 $Z_1, Z_2, \ldots, Z_k$, 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 $U_i \overset{iid}{\sim} N(0,1)$ 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)

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

where $U_i \overset{iid}{\sim} N(0,1)$

In general, if we can represent the random vector $Z = (Z_1, Z_2, \ldots, Z_k)$ as

$$Z = RU$$

where $Z \in \mathbb{R}^n$, $R \in \mathbb{R}^{n \times n}$, $U \in \mathbb{R}^n$, and $U_i \overset{iid}{\sim} N(0,1)$, we refer to $Z$ 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 $Z = (Z_1, Z_2, \ldots, Z_k)$ is JG if there exists a base random vector $U = (U_1, U_2, \ldots, U_l)$ whose components are independent standard normal random variables, a transition matrix $R \in \mathbb{R}^{k \times l}$, and a mean vector $\mu \in \mathbb{R}^k$, such that $Z = RU + \mu$. 

2. A random vector $Z = (Z_1, Z_2, \ldots, Z_k)^\top$ 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 $Z = (Z_1, Z_2, \ldots, Z_k)^\top$ is JG if

$$f_Z(z) = \frac{1}{\sqrt{\det(\Sigma)}} \frac{1}{(\sqrt{2\pi})^k} e^{-\frac{1}{2}(z-\mu)^\top \Sigma^{-1}(z-\mu)}$$

Where $\Sigma = \mathbb{E}[(Z - \mu)(Z - \mu)^\top] = \mathbb{E}[(RU)(RU)^\top] = RE[UU^\top]R^\top = RR^\top$

$\Sigma$ is also called the covariance matrix of $Z$.

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. $Z = RU$ (in this case $k = l = n$), and we would like to derive the probability density of $Z$. Note that here we removed the $\mu$ 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 $\mu$.

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

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

$$f_U(u) = |\det(R)| f_Z(Ru)$$

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

$$P(U \subseteq R') = \int_{R'} f_U(u_1, u_2) du_1 du_2 \approx \sum_{R'} f_U(u_1, u_2) \Delta u_1 \Delta u_2$$

Now, let’s apply the linear transformation $$Z = RU$$, mapping the region $$R’$$ in $$U$$ space, to the region $$T(R')$$ in $$Z$$ space.

The graph on the right is now $$Z$$ space, the 2D space with axes $$Z_1$$ and $$Z_2$$. Assuming that the matrix $$R$$ is invertible, there is a one-to-one correspondence between points in $$U$$ space to points in $$Z$$ space. As we can note in the diagram above, each unit square in $$U$$ space maps to a parallelogram in $$Z$$ 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:

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

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 $$\Delta A$$, we say that

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

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

$$P(Z \subseteq T(R')) = \int_{T(R')} f_Z(z_1, z_2) dz_1 dz_2$$

$$\approx \sum_{T(R')} f_Z(z) \Delta A$$

$$= \sum_{R'} f_Z(Ru) |\det(R)| \Delta u_1 \Delta u_2$$

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

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

$$P(U \subseteq R') = \int \ldots \int_{R'} f_U(u) du_1 du_2 \ldots du_n$$
and
\[ P(Z \subseteq T(R')) = \int \ldots \int_{R'} f_Z(Ru) \left| \det(R) \right| du_1 du_2 \ldots du_n \]

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

Therefore, we can say that
\[ P(U \subseteq R') = \int \ldots \int_{R'} f_U(u) du_1 du_2 \ldots du_n \]
\[ = \int \ldots \int_{R'} f_Z(Ru) \left| \det(R) \right| du_1 du_2 \ldots du_n \]
\[ = P(Z \subseteq T(R')) \]

We conclude that
\[ f_U(u) = f_Z(Ru) \left| \det(R) \right| \]

An almost identical argument will allow us to state that
\[ f_Z(z) = f_U(R^{-1}z) \left| \det(R^{-1}) \right| = \frac{1}{\left| \det(R) \right|} f_U(R^{-1}z) \]

Since the densities for all the \( U_i \)’s are i.i.d, and \( U = R^{-1}Z \), we can write the joint density function of \( Z \) as
\[ f_Z(z) = \frac{1}{\left| \det(R) \right|} f_U(R^{-1}z) \]
\[ = \frac{1}{\left| \det(R) \right|} \prod_{i=1}^{n} f_U_i((R^{-1}z)_i) \]
\[ = \frac{1}{\left| \det(R) \right|} \left( \frac{1}{\sqrt{2\pi}} \right)^n e^{-\frac{1}{2}(R^{-1}z)^\top (R^{-1}z)} \]
\[ = \frac{1}{\left| \det(R) \right|} \left( \frac{1}{\sqrt{2\pi}} \right)^n e^{-\frac{1}{2}z^\top R^{-1}R^{-1}z} \]
\[ = \frac{1}{\left| \det(R) \right|} \left( \frac{1}{\sqrt{2\pi}} \right)^n e^{-\frac{1}{2}z^\top (RR^\top)^{-1}z} \]

Note that \((RR^\top)^{-1}\) is simply the covariance matrix for \( Z \):
\[ \text{Cov}[Z] = E[ZZ^\top] = E[RUU^\top R^\top] = RE[UU^\top]R^\top = RIR^\top = RR^\top \]

Thus the density function of \( Z \) can be written as
\[ f_Z(z) = \frac{1}{\left| \det(R) \right|} \left( \frac{1}{\sqrt{2\pi}} \right)^n e^{-\frac{1}{2}z^\top \Sigma^{-1}z} \]

Furthermore, we know that
\[ \left| \det(\Sigma_Z) \right| = \left| \det(RR^\top) \right| \]
\[ = \left| \det(R) \cdot \det(R^\top) \right| \]
\[ = |\det(R) \cdot \det(R)| = |\det(R)|^2 \]

and therefore
\[
f_Z(z) = \frac{1}{\sqrt{\det(\Sigma_Z)}} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}z'\Sigma_Z^{-1}z} \]

**Estimating Gaussians from Data**

For a particular multivariate Gaussian distribution \(f(.)\), if we do not have the true means and covariances \(\mu, \Sigma\), then our best bet is to use MLE to estimate them empirically with i.i.d. samples \(x_1, x_2, \ldots, x_n\):

\[
\hat{\mu} = \frac{1}{n} \sum_{i=k} x_i \\
\hat{\Sigma} = \frac{1}{n} \sum_{i=k} (x_i - \hat{\mu})(x_i - \hat{\mu})^T
\]

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 \(D = \{x_1, x_2, \ldots, x_n\}\) 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:

\[
\hat{\mu}, \sigma^2^2 = \arg \max_{\mu, \sigma^2} P(x_1, x_2, \ldots, x_n \mid \mu, \sigma^2) \\
= \arg \max_{\mu, \sigma^2} \ln \left( P(x_1, x_2, \ldots, x_n \mid \mu, \sigma^2) \right) \\
= \arg \max_{\mu, \sigma^2} \sum_{i=1}^{n} \ln \left( P(x_i \mid \mu, \sigma^2) \right) \\
= \arg \max_{\mu, \sigma^2} \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} - \ln(\sigma) - \frac{1}{2} \ln(2\pi) \\
= \arg \min_{\mu, \sigma^2} \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma)
\]

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 \(\sigma^2\) and \(\mu\) 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 \(\mu\) given a fixed \(\sigma^2\), and an outer problem that optimizes for \(\sigma^2\) given the optimal value \(\hat{\mu}\). Let’s first solve the inner optimization problem. Given a fixed \(\sigma^2\), the objective is convex in \(\mu\), so we can simply take a partial derivative w.r.t \(\mu\) and set it equal to 0:

\[
\frac{\partial}{\partial \mu} \left( \sum_{i=1}^{n} \frac{(x_i - \mu)^2}{2\sigma^2} + \ln(\sigma) \right) = \sum_{i=1}^{n} \frac{-(x_i - \mu)}{\sigma^2} = 0 \implies \hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i
\]
2.3. MULTIVARIATE GAUSSIANS

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 \(\sigma\), 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 \(\infty\), 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 \(\sigma\) is 0

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

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 \(N(0, \Sigma)\), which just leaves us with the covariance matrix \(\Sigma\). Since \(\Sigma\) is a symmetric, positive semidefinite matrix, we can decompose it by the spectral theorem into \(\Sigma = V \Lambda V^T\), where the columns of \(V\) form an orthonormal basis in \(\mathbb{R}^d\), and \(\Lambda\) is a diagonal matrix with real, non-negative values. We wish to find its level set

\[
f(x) = k
\]

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

\[
x^T \Sigma^{-1} x = c
\]

for some constant \(c\). Without loss of generality, assume that this constant is 1. The level set \(x^T \Sigma^{-1} x = 1\) is an ellipsoid with axes \(v_1, v_2, \ldots, v_d\), with lengths \(\sqrt{\lambda_1}, \sqrt{\lambda_2}, \ldots, \sqrt{\lambda_d}\), respectively. Each axis of the ellipsoid is the vector \(\sqrt{\lambda_i} v_i\), and we can verify that

\[
(\sqrt{\lambda_i} v_i)^T \Sigma^{-1} (\sqrt{\lambda_i} v_i) = \lambda_i v_i^T \Sigma^{-1} v_i = \lambda_i v_i^T (\Sigma^{-1} v_i) = \lambda_i v_i^T (\lambda_i^{-1} v_i) = v_i^T v_i = 1
\]
The entries of \( \Lambda \) dictate how elongated or shrunk the distribution is along each direction. In the case of isotropic distributions, the entries of \( \Lambda \) are all identical, meaning the axes of the ellipsoid form a circle. In the case of anisotropic distributions, the entries of \( \Lambda \) 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 \( \sqrt{\lambda_i} v_i \). Images courtesy Professor Shewchuk’s notes

Properties

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

\[
\mathbf{A Z} \sim \mathcal{N}(\mathbf{A} \mu_{\mathbf{Z}}, \mathbf{A} \Sigma_{\mathbf{Z}} \mathbf{A}^\top)
\]

We can derive the mean and covariance of \( \mathbf{A Z} \) using the linearity of expectations:

\[
\mu_{\mathbf{A Z}} = \mathbb{E}[\mathbf{A Z}] = \mathbf{A} \mathbb{E}[\mathbf{Z}] = \mathbf{A} \mu_{\mathbf{Z}}
\]

and

\[
\Sigma_{\mathbf{A Z}} = \mathbb{E}[(\mathbf{A Z} - \mathbb{E}[\mathbf{A Z}]) (\mathbf{A Z} - \mathbb{E}[\mathbf{A Z}])^\top] = \mathbb{E}[\mathbf{A} (\mathbf{Z} - \mathbb{E}[\mathbf{Z}]) (\mathbf{Z} - \mathbb{E}[\mathbf{Z}])^\top \mathbf{A}^\top] = \mathbf{A} \Sigma_{\mathbf{Z}} \mathbf{A}^\top
\]

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 \( \mathbf{A Z} \) is JG itself, because it can be expressed as a linear transformation of i.i.d. Gaussians: \( \mathbf{A Z} = \mathbf{A} \mathbf{R} \mathbf{U} \).

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

\[
\mu_{\mathbf{Z}} = \begin{bmatrix} \mu_{\mathbf{X}} \\ \mu_{\mathbf{Y}} \end{bmatrix}, \Sigma_{\mathbf{Z}} = \begin{bmatrix} \Sigma_{\mathbf{XX}} & \Sigma_{\mathbf{XY}} \\ \Sigma_{\mathbf{YX}} & \Sigma_{\mathbf{YY}} \end{bmatrix}
\]

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

\[
\mathbf{X} \sim \mathcal{N}(\mu_{\mathbf{X}}, \Sigma_{\mathbf{XX}})
\]
However, the converse is not necessarily true: if \( X \) and \( Y \) are each individually JG, it is not necessarily the case that \( \begin{bmatrix} X \\ Y \end{bmatrix} \) is JG! To see why, let’s suppose that \( X \) and \( Y \) are individually JG. Thus, we can express each as a linear transformation of i.i.d. Gaussian random variables:

\[
X = R_X U_X, \quad Y = R_Y U_Y
\]

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

\[
\begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} R_X & 0 \\ 0 & R_Y \end{bmatrix} \begin{bmatrix} U_X \\ U_Y \end{bmatrix}
\]

However, since we cannot guarantee that the entries of \( U_X \) are independently distributed from the entries of \( U_Y \), 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 \( Z \). The conditional distribution of \( X \) given \( Y \) (and vice versa) is also JG:

\[
X|Y \sim \mathcal{N}(\mu_X + \Sigma_{XY} \Sigma_{YY}^{-1}(Y - \mu_Y), \Sigma_{XX} - \Sigma_{XY} \Sigma_{YY}^{-1} \Sigma_{YX})
\]

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

\[
X|Y \sim \mathcal{N}(\mu_X + 0 \Sigma_{YY}^{-1}(Y - \mu_Y), \Sigma_{XX} - 0 \Sigma_{YY}^{-1} 0) = \mathcal{N}(\mu_X, \Sigma_{XX})
\]

This also follows from the multivariate Gaussian pdf:

\[
f_Z(x, y) = \frac{1}{(\sqrt{2\pi})^{n_x + n_y}} |\Sigma_{XX}|^{-\frac{1}{2}} \cdot |\Sigma_{YY}|^{-\frac{1}{2}} \exp \left( -\frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^\top \left[ \begin{bmatrix} \Sigma_{XX} & 0 \\ 0 & \Sigma_{YY} \end{bmatrix} \right]^{-1} \begin{bmatrix} x \\ y \end{bmatrix} \right)
\]

\[
= \frac{1}{(\sqrt{2\pi})^{n_x}} |\Sigma_{XX}|^{-\frac{1}{2}} \exp \left( -\frac{1}{2} x^\top \Sigma_{XX}^{-1} x \right) \cdot \frac{1}{(\sqrt{2\pi})^{n_y}} |\Sigma_{YY}|^{-\frac{1}{2}} \exp \left( -\frac{1}{2} y^\top \Sigma_{YY}^{-1} y \right)
\]

\[
= f_X(x) \cdot f_Y(y)
\]

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)

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:

\[
Y_i = f(x_i) + Z_i
\]

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

\[
W_j \sim \mathcal{N}(\mu_j, \sigma_h^2)
\]
So far, we have restricted ourselves to the case when the noise/parameters are i.i.d:

\[ Z \sim \mathcal{N}(0, \sigma^2 I), \quad W \sim \mathcal{N}(\mu_W, \sigma^2 h I) \]

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 noise and parameters are jointly Gaussian with arbitrary covariance matrices:

\[ Z \sim \mathcal{N}(0, \Sigma_Z), \quad W \sim \mathcal{N}(\mu_W, \Sigma_W) \]

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!

**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.

**Optimization View**

From an optimization perspective, the problem can be expressed as

\[
\hat{w}_{WLS} = \arg \min_{w \in \mathbb{R}^d} \left( \sum_{i=1}^{n} \omega_i(y_i - x_i^\top w)^2 \right)
\]

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

\[
\hat{w}_{WLS} = \arg \min_{w \in \mathbb{R}^d} (y - Xw)^\top \Omega (y - Xw)
\]

Where the \( i \)’th row \( X \) is \( x_i^\top \), and \( \Omega \in \mathbb{R}^{n \times n} \) is a diagonal matrix with \( \Omega_{i,i} = \omega_i \).

We rewrite the WLS objective to an OLS objective:

\[
\hat{w}_{WLS} = \arg \min_{w \in \mathbb{R}^d} (\Omega^{1/2} y - \Omega^{1/2} Xw)^\top (\Omega^{1/2} y - \Omega^{1/2} Xw)
\]

\[
= \arg \min_{w \in \mathbb{R}^d} \| \Omega^{1/2} y - \Omega^{1/2} Xw \|^2
\]

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

\[
\hat{w}_{WLS} = \left((\Omega^{1/2} X)^\top (\Omega^{1/2} X)\right)^{-1} (\Omega^{1/2} X)^\top \Omega^{1/2} y = (X^\top \Omega X)^{-1} X^\top \Omega y
\]
Probabilistic View

As in MLE, we assume that our observations $y$ 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

$$Y_i = x_i^\top w + Z_i$$

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 $Z \sim \mathcal{N}(\mu_Z, \Sigma_Z)$, where

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

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 $\sigma_i$:

$$\frac{Y_i}{\sigma_i} = \frac{x_i^\top}{\sigma_i} w + \frac{Z_i}{\sigma_i}$$

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

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

Jointly, we can express this change of coordinates as

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

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:

$$\hat{w}_{WLS} = \arg \min_{w \in \mathbb{R}^d} \left( \sum_{i=1}^n \frac{(y_i - x_i^\top w)^2}{\sigma_i^2} \right) + n \log \sqrt{2\pi}$$

$$= \arg \min_{w \in \mathbb{R}^d} \sum_{i=1}^n \frac{1}{\sigma_i^2} (y_i - x_i^\top w)^2$$

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

$$\hat{w}_{WLS} = (X^\top \Sigma_Z^{-\frac{1}{2}} \Sigma_Z^{-\frac{1}{2}} X)^{-1} X^\top \Sigma_Z^{-\frac{1}{2}} \Sigma_Z^{-\frac{1}{2}} y = (X^\top \Sigma_Z^{-1} X)^{-1} X^\top \Sigma_Z^{-1} y$$

As long as no $\sigma$ is 0, $\Sigma_Z$ is invertible. Note that $\omega_i$ from the optimization perspective is directly related to $\sigma_i^2$ from the probabilistic perspective: $\omega_i = \frac{1}{\sigma_i^2}$. As the variance $\sigma_i^2$ of the noise corresponding to data point $i$ decreases, the weight $\omega_i$ increases: we are more concerned about fitting data point $i$ because it is likely to match the true underlying de-noised point. Inversely, as the variance $\sigma_i^2$ increases, the weight $\omega_i$ 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

\[
Y = Xw + Z
\]

where \( Z \) is now a jointly Gaussian random vector. That is,

\[
Z \sim \mathcal{N}(0, \Sigma_Z), \quad Y \sim \mathcal{N}(Xw, \Sigma_Z)
\]

This problem is known as **generalized least squares**. Our goal is to maximize the probability of our data over the set of possible \( w \)’s:

\[
\hat{w}_{\text{gls}} = \arg \max_{w \in \mathbb{R}^d} \frac{1}{\sqrt{\det(\Sigma_Z)}} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}(y-Xw)^\top \Sigma_Z^{-1}(y-Xw)}
\]

\[
= \arg \min_{w \in \mathbb{R}^d} (y - Xw)^\top \Sigma_Z^{-1}(y - Xw)
\]

The optimization problem is therefore given by

\[
\hat{w}_{\text{gls}} = \arg \min_{w \in \mathbb{R}^d} (y - Xw)^\top \Sigma_Z^{-1}(y - Xw)
\]

Since \( \Sigma_Z \) is symmetric, we can decompose it into its eigen structure using the spectral theorem:

\[
\Sigma_Z = Q \begin{bmatrix}
\sigma_1^2 & 0 & \cdots & 0 \\
0 & \sigma_2^2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & \sigma_n^2
\end{bmatrix} Q^\top
\]

where \( Q \) 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

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

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

\[
\Sigma_Z^{-\frac{1}{2}} y \sim \mathcal{N}(\Sigma_Z^{-\frac{1}{2}} Xw, \Sigma_Z^{-\frac{1}{2}} \Sigma_Z \Sigma_Z^{-\frac{1}{2}}) = \mathcal{N}(\Sigma_Z^{-\frac{1}{2}} Xw, I)
\]

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

\[
\hat{w}_{\text{gls}} = (X^\top \Sigma_Z^{-1} X)^{-1} X^\top \Sigma_Z^{-1} y
\]
Ridge Regression with Dependent Parameters

In the ordinary least squares (OLS) statistical model, we assume that the output $Y$ 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

$$Y = XW + Z$$

where $Y$ and $Z$ are $n$-dimensional random vectors, $W$ is a $d$-dimensional random vector, and $X$ 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 $W$ in which $W_i$ are i.i.d. (univariate) Gaussian, or equivalently,

$$W \sim N(0, I)$$

But more generally, we can allow $W$ to be any multivariate Gaussian:

$$W \sim N(\mu_W, \Sigma_W)$$

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

$$W = \Sigma_W^{1/2}V + \mu_W, \quad V \sim N(0, I)$$

Plugging this parameterization into our previous statistical model gives

$$Y = X(\Sigma_W^{1/2}V + \mu_W) + Z$$

But this can be re-written

$$Y - X\mu_W = X\Sigma_W^{1/2}V + Z$$

which we see has the form of the statistical problem that underlies OLS, and therefore

$$\hat{v} = (\Sigma_W^{T/2}X^T \Sigma_W^{1/2}X + I)^{-1}\Sigma_W^{T/2}X^T(y - X\mu_W)$$

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

$$\hat{w} = \Sigma_W^{1/2}v + \mu_W$$

$$= \mu_W + \Sigma_W^{1/2}(\Sigma_W^{T/2}X^T \Sigma_W^{1/2}X + I)^{-1}\Sigma_W^{T/2}X^T(y - X\mu_W)$$

$$= \mu_W + (X^T X + \Sigma_W^{-T/2} \Sigma_W^{-1/2})^{-1}X^T(y - X\mu_W)$$

Note that there are two terms: the prior mean $\mu_W$, plus another term that depends on both the data and the prior. The precision matrix of $W$’s prior ($\Sigma_W^{-1}$) controls how the data fit error affects our estimate.

To gain intuition, let us consider the simplified case where

$$\Sigma_W = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d^2 \end{bmatrix}$$
When the prior variance \( \sigma_j^2 \) 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 \( \Sigma^{-1}_W \) will be small, as desired.

Conversely if \( \sigma_j^2 \) 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.

**Alternative derivation**

MAP with colored noise can be expressed as:

\[
U, V \overset{iid}{\sim} \mathcal{N}(0, I) \quad (2.6)
\]

\[
\begin{bmatrix}
Y \\
W
\end{bmatrix} =
\begin{bmatrix}
R_Z & XR_W \\
0 & R_W
\end{bmatrix}
\begin{bmatrix}
U \\
V
\end{bmatrix} \quad (2.7)
\]

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

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

\[
U', V' \overset{iid}{\sim} \mathcal{N}(0, I)
\]

\[
\begin{bmatrix}
W \\
Y
\end{bmatrix} =
\begin{bmatrix}
A & B \\
0 & D
\end{bmatrix}
\begin{bmatrix}
U' \\
V'
\end{bmatrix}
\]

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

\[
\mathbb{E}[W \mid Y = y] = \mathbb{E}[AU' + BV' \mid Y = y]
\]

\[
= \mathbb{E}[AU' \mid Y = y] + \mathbb{E}[BD^{-1}Y \mid Y = y]
\]

\[
= A \mathbb{E}[U'] + D^{-1}y
\]

\[
\text{Var}(W \mid Y = y) = \mathbb{E}[(W - \mathbb{E}[W])(W - \mathbb{E}[W])^\top \mid Y = y]
\]

\[
= \mathbb{E}[(AU' + BD^{-1}Y - BD^{-1}Y)(AU' + BD^{-1}Y - BD^{-1}Y)^\top \mid Y = y]
\]

\[
= \mathbb{E}[(AU')(AU')^\top \mid Y = y]
\]

\[
= A \mathbb{E}[U'(U')^\top] A^\top
\]

\[
= AA^\top
\]

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

\[
W \mid Y = y \sim \mathcal{N}(BD^{-1}y, AA^\top)
\]
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

\[
E\begin{bmatrix} W \\ Y \end{bmatrix} E\begin{bmatrix} W^\top \\ Y^\top \end{bmatrix} = \begin{bmatrix} A & B \\ 0 & D \end{bmatrix} \begin{bmatrix} A^\top & 0 \\ B^\top & D^\top \end{bmatrix} = \begin{bmatrix} AA^\top + BB^\top & BD^\top \\ DB^\top & DD^\top \end{bmatrix} = \begin{bmatrix} \Sigma_W & \Sigma_{W,Y} \\ \Sigma_{Y,W} & \Sigma_Y \end{bmatrix}
\]

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

\[
BD^{-1}Y = BD^{-1}D^{-1}Y = (BD^\top)(DD^{-1})Y = \Sigma_{W,Y}\Sigma_Y^{-1}Y
\]

\[
AA^\top = AA^\top + BB^\top - BB^\top
\]

\[
= AA^\top + BB^\top - BD^\top D^{-1}DB^\top
\]

\[
= AA^\top + BB^\top - (BD^\top)(DD^{-1})DB^\top
\]

\[
= \Sigma_W - \Sigma_{W,Y}\Sigma_Y^{-1}\Sigma_{Y,W}
\]

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

\[
E\begin{bmatrix} Y \\ W \end{bmatrix} E\begin{bmatrix} Y^\top \\ W^\top \end{bmatrix} = \begin{bmatrix} R_ZR_Z^\top + XR_WR_W^\top X^\top & XR_WR_W^\top \\ R_WR_W^\top X^\top & R_WR_W^\top \end{bmatrix} = \begin{bmatrix} \Sigma_Y & \Sigma_{Y,W} \\ \Sigma_{W,Y} & \Sigma_W \end{bmatrix}
\]

Therefore after defining \( \Sigma_Z = R_ZR_Z^\top \), we can read off

\[
\Sigma_W = R_WR_W^\top
\]

\[
\Sigma_Y = \Sigma_Z + XX\Sigma_WX^\top
\]

\[
\Sigma_{Y,W} = XX\Sigma_W
\]

\[
\Sigma_{W,Y} = XX^\top \Sigma_W
\]

Plugging this into our estimator yields

\[
\hat{w} = E[W \mid Y = y]
\]

\[
= \Sigma_{W,Y}\Sigma_Y^{-1}y
\]

\[
= \Sigma_WX^\top(\Sigma_Z + XX\Sigma_WX^\top)^{-1}y
\]

One may be concerned because this expression does not take the form we expect – the inverted matrix is hitting \( y \) directly, unlike in other solutions we’ve seen. But using the Woodbury matrix identity\(^1\), it turns out that we can rewrite this expression as

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

which looks more familiar.

\(^1\) \( (A + UCV)^{-1} = A^{-1} - A^{-1}UA^{-1}(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \)
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:

<table>
<thead>
<tr>
<th>$\mathbf{W}$</th>
<th>$\mathcal{N}(0, \mathbf{I})$</th>
<th>$\mathcal{N}(0, \Sigma_{\mathbf{Z}})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prior</td>
<td>$\hat{\mathbf{w}}_{\text{OLS}} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$</td>
<td>$\hat{\mathbf{w}}<em>{\text{GLS}} = (\mathbf{X}^\top\Sigma</em>{\mathbf{Z}}^{-1}\mathbf{X})^{-1}\mathbf{X}^\top\Sigma_{\mathbf{Z}}^{-1}\mathbf{y}$</td>
</tr>
<tr>
<td>$\mathcal{N}(0, \lambda^{-1}\mathbf{I})$</td>
<td>$\hat{\mathbf{w}}_{\text{RIDGE}} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{N}(\mathbf{\mu}<em>\mathbf{W}, \Sigma</em>{\mathbf{W}})$</td>
<td>$\hat{\mathbf{w}} = \mathbf{\mu}<em>\mathbf{W} + (\mathbf{X}^\top\mathbf{X} + \Sigma</em>{\mathbf{W}}^{-1})^{-1}\mathbf{X}^\top(\mathbf{y} - \mathbf{X}\mathbf{\mu}_\mathbf{W})$</td>
<td></td>
</tr>
</tbody>
</table>

2.5 Kernel Ridge Regression

In ridge regression, we are 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 $\ell$ 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 $\Phi \in \mathbb{R}^{n \times d}$, where $\phi_i^\top = \phi(x_i) \in \mathbb{R}^d$. Then we solve a well-defined optimization problem that involves $\Phi$ 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 $\ell$ dimensional space, then there are $d = (\ell + p)$ 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:

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

This operation involves calculating $\Phi^\top\Phi$, which is a $d \times d$ matrix and takes $O(d^2n)$ time to compute. The matrix inversion operation takes an additional $O(d^3)$ time to compute. What would we 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 $\Phi^\top$ and $\Phi$, we end up with an $n \times n$ matrix $\Phi\Phi^\top$. In fact, the matrix $\Phi\Phi^\top$ 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 $\Phi\Phi^\top$.

**Derivation**

For simplicity of notation, let’s revert back to using $\mathbf{X}$ instead of $\Phi$ (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

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

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

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

$$\lambda\mathbf{w} = \mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\mathbf{w}$$

$$\mathbf{w} = \frac{1}{\lambda}(\mathbf{X}^\top\mathbf{y} - \mathbf{X}^\top\mathbf{X}\mathbf{w})$$
2.5. KERNEL RIDGE REGRESSION

\[ w = \frac{X^\top y - X^\top Xw}{\lambda} \]

\[ w = \frac{X^\top y - Xw}{\lambda} \]

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

Recall that the relationship we have to satisfy is \( X^\top Xw - \lambda w = X^\top y \). Let’s assume that we had \( v \), and just substitute \( X^\top v \) in for all the \( w \)’s.

\[ X^\top X(X^\top v) + \lambda (X^\top v) = X^\top y \]

\[ X^\top XX^\top v + X^\top (\lambda v) = X^\top y \]

\[ X^\top (XX^\top v + \lambda v) = X^\top (y) \]

We can’t yet isolate \( v \) and have a closed-form solution for it, but we can make the observation that if we found an \( v \) such that we had \( XX^\top v + \lambda v = y \) that would imply that this \( v \) also satisfies the above equation. Note that we did not “cancel the \( X^\top \)’s on both sides of the equation.” We saw that having \( v \) satisfy one equation implied that it satisfied the other as well. So, indeed, we can isolate \( v \) in this new equation:

\[ (XX^\top + \lambda I)v = y \implies v^* = (XX^\top + \lambda I)^{-1}y \]

and have that the \( v \) which satisfies this equation will be such that \( X^\top v \) equals \( w \). We conclude that the optimal \( w \) is

\[ w^* = X^\top v^* = X^\top (XX^\top + \lambda I)^{-1}y \]

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

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

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) \( X \) that maps \( \mathbb{R}^\ell \) to \( \mathbb{R}^n \). Denote \( \mathcal{N}(X) \) as the nullspace of \( X \), and \( \mathcal{R}(X) \) as the range of \( X \). Then the following properties hold:

1. \( \mathcal{N}(X) \bigoplus \mathcal{R}(X^\top) = \mathbb{R}^\ell \) and \( \mathcal{N}(X^\top) \bigoplus \mathcal{R}(X) = \mathbb{R}^n \) by symmetry

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

2. \( \mathcal{N}(X^\top X) = \mathcal{N}(X) \) and \( \mathcal{N}(XX^\top) = \mathcal{N}(X^\top) \) by symmetry
3. \( \mathcal{R}(X^\top X) = \mathcal{R}(X) \) and \( \mathcal{R}(XX^\top) = \mathcal{R}(X) \) by symmetry.

Here’s where FTLA comes, in the context of kernel ridge regression. We know that we can express any \( \mathbf{w} \in \mathbb{R}^d \) as a unique combination \( \mathbf{w} = \mathbf{w}_1 + \mathbf{w}_2 \), where \( \mathbf{w}_1 \in \mathcal{R}(X) \) and \( \mathbf{w}_2 \in \mathcal{N}(X) \). Equivalently we can express this as \( \mathbf{w} = X^\top \mathbf{v} + \mathbf{r} \), where \( \mathbf{v} \in \mathbb{R}^n \) and \( \mathbf{r} \in \mathcal{N}(X) \). Now, instead of optimizing over \( \mathbf{w} \in \mathbb{R}^d \), we can optimize over \( \mathbf{v} \in \mathbb{R}^n \) and \( \mathbf{r} \in \mathbb{R}^d \), 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 \( \mathbf{w} = X^\top \mathbf{v} + \mathbf{r} \), where \( \mathbf{v} \in \mathbb{R}^n \) and \( \mathbf{r} \in \mathcal{N}(X) \). Let’s now solve ridge regression by optimizing over the variables \( \mathbf{v} \) and \( \mathbf{r} \) instead of \( \mathbf{w} \):

\[
\mathbf{v}^*, \mathbf{r}^* = \arg\min_{\mathbf{v} \in \mathbb{R}^n, \mathbf{r} \in \mathcal{N}(X)} \| X \mathbf{v} - \mathbf{y} \|^2_2 + \lambda \| \mathbf{r} \|^2_2
\]

\[
= \arg\min_{\mathbf{v} \in \mathbb{R}^n, \mathbf{r} \in \mathcal{N}(X)} \| X(X^\top \mathbf{v} + \mathbf{r}) - \mathbf{y} \|^2_2 + \lambda \| X^\top \mathbf{v} + \mathbf{r} \|^2_2
\]

\[
= \arg\min_{\mathbf{v} \in \mathbb{R}^n, \mathbf{r} \in \mathcal{N}(X)} (\mathbf{v}^\top XX^\top \mathbf{v} - 2\mathbf{v}^\top XX^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}) + \lambda (\mathbf{v}^\top XX^\top \mathbf{v} + 2\mathbf{v}^\top XX^\top \mathbf{r} + \mathbf{r}^\top \mathbf{r})
\]

\[
= \arg\min_{\mathbf{v} \in \mathbb{R}^n, \mathbf{r} \in \mathcal{N}(X)} (\mathbf{v}^\top XX^\top \mathbf{v} - 2\mathbf{v}^\top XX^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}) + \lambda (\mathbf{v}^\top XX^\top \mathbf{v} + \mathbf{r}^\top \mathbf{r})
\]

We crossed out \( X\mathbf{r} \) and \( 2\mathbf{v}^\top X\mathbf{r} \) because \( \mathbf{r} \in \mathcal{N}(X) \) and therefore \( X\mathbf{r} = \mathbf{0} \). 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:

\[
\nabla^2_r L(\mathbf{v}, \mathbf{r}) = 2I \succeq 0
\]

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

\[
\nabla^2_\mathbf{v} L(\mathbf{v}, \mathbf{r}) = 2XX^\top XX^\top + 2\lambda XX^\top \succeq 0
\]

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

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

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

\[
\nabla_\mathbf{v} L(\mathbf{v}^*, \mathbf{r}^*) = 2XX^\top XX^\top \mathbf{v}^* - 2XX^\top \mathbf{y} + 2\lambda XX^\top \mathbf{v}^* = 0
\]

\[
\implies XX^\top (XX^\top + \lambda I)\mathbf{v}^* = XX^\top (\mathbf{y})
\]

\[
\implies \mathbf{v}^* = (XX^\top + \lambda I)^{-1} \mathbf{y}
\]

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

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

and arrive at the same solution as in the previous derivation.
2.5. KERNEL RIDGE REGRESSION

Non-i.i.d. Case

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

\[ Y | W \sim N(XW, \sigma^2 I), \quad W \sim N(0, \sigma^2_n I) \]

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

\[ Y | W \sim N(XW, \Sigma_Z), \quad W \sim N(0, \Sigma_W) \]

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

\[ w^* = (X^\top \Sigma_Z^{-1} X + \Sigma_W^{-1})^{-1} X^\top \Sigma_Z^{-1} y \]

or the case that we desire in kernel ridge regression

\[ w^* = \Sigma_W X^\top (X \Sigma_W X^\top + \Sigma_Z)^{-1} y \]

The principal difference in the non-i.i.d case is that we are computing \(X \Sigma_W X^\top\) as opposed to \(XX^\top\).

Kernels

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

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

Let’s explore the \(\Phi \Phi^\top\) term in kernel ridge regression in more detail:

\[
\Phi \Phi^\top = \left( \begin{array}{ccc}
\phi_1^\top & \phi_2^\top & \cdots \\
\phi_2^\top & \phi_2^\top & \cdots \\
\vdots & \vdots & \ddots \\
\phi_n^\top & \phi_n^\top & \cdots \\
\end{array} \right) \left( \begin{array}{c}
| \\
\phi_1 \\
| \\
\phi_2 \\
| \\
\vdots \\
| \\
\phi_n \\
| \\
\end{array} \right) = \left( \begin{array}{cccc}
\phi_1^\top \phi_1 & \phi_1^\top \phi_2 & \cdots \\
\phi_2^\top \phi_1 & \phi_2^\top \phi_2 & \cdots \\
\vdots & \vdots & \ddots \\
\phi_n^\top \phi_1 & \phi_n^\top \phi_2 & \cdots & \phi_n^\top \phi_n \\
\end{array} \right)
\]

Each entry \(\Phi \Phi^\top_{ij}\) is a dot product between \(\phi(x_i)\) and \(\phi(x_j)\) and can be interpreted as a similarity measure:

\[
\Phi \Phi^\top_{ij} = \langle \phi_i, \phi_j \rangle = \langle \phi(x_i), \phi(x_j) \rangle = k(x_i, x_j)
\]

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 \(k(x_i, x_j)\) terms as the **Gram matrix** and denote it as \(K\):

\[
K = \Phi \Phi^\top = \begin{pmatrix}
  k(x_1, x_1) & k(x_1, x_2) & \cdots \\
  k(x_2, x_1) & \ddots & \\
  \vdots & & \ddots \\
  k(x_n, x_1) & \cdots & k(x_n, x_n)
\end{pmatrix}
\]

Formally, \(k(x_i, x_j)\) is defined to be a valid kernel function if either of the following definitions are met:

- There exists a feature map \(\phi(., )\) such that \(\forall x_i, x_j, \quad k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle\)
- For all sets \(D = \{x_1, x_2, \ldots, x_n\}\), the Gram matrix \(K(D)\) is PSD
We will now state some basic properties of kernels.

- Given two valid kernels \( k_a \) and \( k_b \), their linear combination
  \[
  k(x_i, x_j) = \alpha k_a(x_i, x_j) + \beta k_b(x_i, x_j)
  \]
  where \( \alpha, \beta \geq 0 \) is also a valid kernel. We can show this from the second property:
  \[
  \forall v \in \mathbb{R}^n, v^\top (\alpha K_a + \beta K_b)v = \alpha v^\top K_a v + \beta v^\top K_b v \geq 0
  \]

- Given a positive semidefinite matrix \( \Sigma \),
  \[
  k(x_i, x_j) = \phi(x_i)^\top \Sigma \phi(x_j)
  \]
  is a valid kernel. We can show this from the first property: \( \tilde{\phi}(x_i) = \Sigma^{1/2} \phi(x_i) \)

- Given a valid kernel \( k_a \),
  \[
  k(x_i, x_j) = f(x_i)f(x_j)k_a(x_i, x_j)
  \]
  is a valid kernel. We can show this from the first property: \( \tilde{\phi}(x_i) = f(x_i)\phi(x_i) \)

Computing the each Gram matrix entry \( k(x_i, x_j) \) can be done in a straightforward fashion if we apply the feature map to \( x_i \) and \( x_j \) 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 \( \ell \) is the dimensionality of the problem in the raw feature space and \( p \) is the degree of the polynomials in the augmented feature space.

### 2.6 Kernel Trick

Suppose that we are computing \( k(x, z) \), using a \( p \)-degree polynomial feature map that maps \( \ell \) 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

\[
\phi(x) = \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix}^\top
\]

the kernel function can be expressed as

\[
k(x, z) = \phi(x)^\top \phi(z)
\]

\[
= \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix}^\top \begin{bmatrix} z_1^2 & z_2^2 & \sqrt{2}z_1z_2 & \sqrt{2}z_1 & \sqrt{2}z_2 & 1 \end{bmatrix}
\]

\[
= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2 + 2x_1z_1 + 2x_2z_2 + 1
\]

\[
= (x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2) + 2x_1z_1 + 2x_2z_2 + 1
\]

\[
= (x_1^2 + x_2z_2)^2 + 2(x_1z_1 + x_2z_2) + 1
\]

\[
= (\mathbf{x}^\top \mathbf{z})^2 + 2\mathbf{x}^\top \mathbf{z} + 1
\]

\[
= (\mathbf{x}^\top \mathbf{z} + 1)^2
\]

We can compute \( k(x, z) \) either by

1. Raising the inputs to the augmented feature space and take their inner product
2. Computing \((x^\top z + 1)^2\), 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 \(\ell\) and \(p\), and for \(p\)-degree polynomial features, we have that

\[
k(x, z) = (x^\top z + 1)^p
\]

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

\[
\tilde{\phi}(x) = \begin{bmatrix} x_1^2 & x_2^2 & x_1 x_2 & x_1 & x_2 & 1 \end{bmatrix}^\top
\]

in which case the kernel function would be expressed as

\[
k(x, z) = \tilde{\phi}(x)^\top \Sigma \tilde{\phi}(z) = (x^\top z + 1)^2, \quad \Sigma = \text{Diag}(1, 1, 2, 2, 2, 1)
\]

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

1. i.i.d. prior \(W \sim \mathcal{N}(0, \text{Diag}(1, 1, 1, 1, 1))\), using the feature mapping \(\phi(x)\)

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

**Computational Analysis**

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

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

or

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

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

1. Kernelized

\[
\hat{y} = \langle \phi(z), w^* \rangle = \phi(z)^\top (\Phi \Phi^\top + \lambda I)^{-1} y = [k(x_1, z) \ldots k(x_n, z)] (K + \lambda I)^{-1} y
\]

Computing the \(K\) 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

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

Computing the \(\Phi^\top \Phi\) 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.
2.7 Total Least Squares

Previously, we have covered Ordinary Least Squares (OLS) which assumes that the dependent variable \( y \) is noisy but the independent variables \( x \) 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 \textit{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

\[ (x, y) = (x_{\text{true}} + z_x, y_{\text{true}} + z_y) \]

\[ z_x, z_y \sim \mathcal{N}(0, 1) \]

Combining the previous two relations, we obtain

\[
\begin{align*}
  y &= y_{\text{true}} + z_y \\
  &= wx_{\text{true}} + z_y \\
  &= w(x - z_x) + z_y \\
  &= wx - wz_x + z_y \\
  &\sim \mathcal{N}(0, w^2+1)
\end{align*}
\]

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

\[
\log P(x, y; a) = \text{constant} - \frac{1}{2} \log\left( w^2 + 1 \right) - \frac{1}{2} \frac{(y - wx)^2}{w^2 + 1}
\]

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 \( \|Xw - y\|^2 \), which is equivalent to

\[
\min_{w, \epsilon} \|\epsilon\|^2 \quad \text{subject to} \quad y = Xw + \epsilon
\]
This only accounts for errors in the dependent variable, so for TLS we introduce a second residual \( \epsilon_X \in \mathbb{R}^{n \times d} \) to account for independent variable error:

\[
\min_{w, \epsilon_X, \epsilon_y} \left\| \begin{bmatrix} \epsilon_X & \epsilon_y \end{bmatrix} \right\|_F^2 \quad \text{subject to} \quad (X + \epsilon_X)w = y + \epsilon_y
\]

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

\[
\begin{bmatrix} X + \epsilon_X & y + \epsilon_y \end{bmatrix} \begin{bmatrix} w \\ -1 \end{bmatrix} = 0
\]

This expression tells us that the vector \( \begin{bmatrix} w^T \\ -1 \end{bmatrix} \) 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 \( \epsilon_X \) and \( \epsilon_y \) 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 \( A \in \mathbb{R}^{m \times n} \) has rank \( r \leq \min(m, n) \), and let \( A = U \Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T \) be its singular value decomposition. Then

\[
A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U \begin{bmatrix} \sigma_1 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \sigma_k & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} V^T
\]

where \( k \leq r \), is the best rank-\( k \) approximation to \( A \) in the sense that

\[
\|A - A_k\|_F \leq \|A - \tilde{A}\|_F
\]

for any \( \tilde{A} \) such that \( \text{rank}(\tilde{A}) \leq k \).

Let us assume that the data matrix \( [X \quad y] \) is full rank.\(^3\) Write its singular value decomposition:

\[
[X \quad y] = \sum_{i=1}^{d+1} \sigma_i u_i v_i^T
\]

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

\[
[X + \epsilon_X \quad y + \epsilon_y] = \sum_{i=1}^{d} \sigma_i u_i v_i^T
\]

\(^2\) 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 \( y \) not to lie in the columnspace of \( X \).
which is achieved by setting
\[
\begin{bmatrix}
\epsilon X & \epsilon y
\end{bmatrix} = -\sigma_{d+1} u_{d+1} v_{d+1}^\top
\]
The nullspace of our resulting matrix is then
\[
\text{null}\left(\begin{bmatrix} X + \epsilon X & y + \epsilon y \end{bmatrix}\right) = \text{null}\left(\sum_{i=1}^d \sigma_i u_i v_i^\top\right) = \text{span}\{v_{d+1}\}
\]
where the last equality holds because \{v_1, \ldots, v_{d+1}\} form an orthogonal basis for \(\mathbb{R}^{d+1}\). To get the weight \(w\), we find a scaling \(\alpha\) such that \(\begin{bmatrix} w^\top & -1 \end{bmatrix}\) is in the nullspace, i.e.
\[
\begin{bmatrix} w \\ -1 \end{bmatrix} = \alpha v_{d+1}
\]
Note that this requires the \((d + 1)\)st component of \(v_{d+1}\) to be nonzero. (See the next section for details.)

Once we have \(v_{d+1}\), 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 \(w\). Since \(v_{d+1}\) is a right-singular vector of \([X \ y]\), it is an eigenvector of the matrix
\[
\begin{bmatrix} X & y \end{bmatrix}^\top \begin{bmatrix} X & y \end{bmatrix} = \begin{bmatrix} X^\top X & X^\top y \\ y^\top X & y^\top y \end{bmatrix}
\]
So to find it we solve
\[
\begin{bmatrix} X^\top X & X^\top y \\ y^\top X & y^\top y \end{bmatrix} \begin{bmatrix} w \\ -1 \end{bmatrix} = \sigma_{d+1}^2 \begin{bmatrix} w \\ -1 \end{bmatrix}
\]
From the top line we see that \(w\) satisfies
\[
X^\top X w - X^\top y = \sigma_{d+1}^2 w
\]
which can be rewritten as
\[
(X^\top X - \sigma_{d+1}^2 I)w = X^\top y
\]
Thus, assuming \(X^\top X - \sigma_{d+1}^2 I\) is invertible (see the next section), we can solve for the weights as
\[
\hat{w}_{\text{TLS}} = (X^\top X - \sigma_{d+1}^2 I)^{-1} X^\top y
\]
This result is like ridge regression, but with a negative regularization constant! Why does this make sense? One of the 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:
\[
X = X_{\text{true}} + Z
\]
where \(X_{\text{true}}\) are the actual values before noise corruption, and \(Z\) is a zero-mean noise term with i.i.d. entries. Then
\[
E[X^\top X] = E[(X_{\text{true}} + Z)^\top (X_{\text{true}} + Z)]
\]
\[
= E[X_{\text{true}}^\top X_{\text{true}}] + E[X_{\text{true}}^\top Z] + E[Z^\top X_{\text{true}}] + E[Z^\top Z]
\]
\[
= X_{\text{true}}^\top X_{\text{true}} + X_{\text{true}}^\top E[Z] + E[Z^\top X_{\text{true}}] + E[Z^\top Z]
\]
\[
= X_{\text{true}}^\top X_{\text{true}} + X_{\text{true}}^\top E[Z] + E[Z^\top X_{\text{true}}] + E[Z^\top Z]
\]

2.7. TOTAL LEAST SQUARES

\[ X_{\text{true}}^\top X_{\text{true}} + E[Z^\top Z] \]

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

For another perspective, note that

\[
E[X^\top] = E[(X_{\text{true}} + Z)^\top] = E[X_{\text{true}}^\top + Z^\top] = E[X_{\text{true}}^\top] + E[Z^\top] = X_{\text{true}}^\top
\]

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

\[ \hat{w}_{\text{ols}} = (X_{\text{true}}^\top X_{\text{true}})^{-1} X_{\text{true}}^\top y = (E[X^\top X] - E[Z^\top Z])^{-1} E[X^\top y] \]

which strongly resembles the TLS solution, but expressed in terms of expectations over the noise \( Z \).

**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 \( \sigma_1, \ldots, \sigma_{d+1} \) denote the singular values of \( [X \ y] \), and \( \tilde{\sigma}_1, \ldots, \tilde{\sigma}_d \) denote the singular values of \( X \). If \( \sigma_{d+1} < \tilde{\sigma}_d \), then the total least squares problem has a solution, given by

\[ \hat{w}_{\text{tls}} = (X^\top X - \sigma_{d+1}^2 I)^{-1} X^\top y \]

**Proof.** Let \( \sum_{i=1}^{d+1} \sigma_i u_i v_i^\top \) be the SVD of \( [X \ y] \), and suppose \( \sigma_{d+1} < \tilde{\sigma}_d \). We first show that the \((d + 1)\)st component of \( v_{d+1} \) is nonzero. To this end, suppose towards a contradiction that

\[ v_{d+1} = \begin{bmatrix} a^\top \\ 0 \end{bmatrix} \]

for some \( a \neq 0 \). Since \( v_{d+1} \) is a right-singular vector of \( [X \ y] \), i.e. an eigenvector of \( [X \ y]^\top [X \ y] \), we have

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

Then

\[ X^\top X a = \sigma_{d+1}^2 a \]

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

\[ \tilde{\sigma}_d^2 = \lambda_{\min}(X^\top X) \]

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

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

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

giving

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

which guarantees that the matrix is invertible. \( \square \)
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 $v_{d+1}$ being nonzero, and the matrix $X^\top X - \sigma^2_{d+1}$ 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 $\epsilon_X$. In principle, we could use this to denoise $X$, as in $\hat{X}_{true} = X - \epsilon_X$, 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 $\hat{X}_{true}$ 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

3.1 Principal Component Analysis

In machine learning, the data we have are often very high-dimensional. 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 noise

Principal Component Analysis (PCA) is 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.

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 $\theta$ 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:
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

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

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

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

---

\(^1\) To make sense of the sample variance, recall that for any random variable \( Z \),

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

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

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

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. \( z_i = \mathbf{v}^\top \mathbf{x}_i \).
Hence $X^TXv_1 = \lambda v_1$, i.e. $v_1$ is an eigenvector of $X^TX$ with eigenvalue $\lambda$. Since we constrain $v_1^Tv_1 = 1$, the value of the objective is precisely

$$v_1^TX^TXv_1 = v_1^T(\lambda v_1) = \lambda v_1^Tv_1 = \lambda$$

so the optimal value is $\lambda = \lambda_{\text{max}}(X^TX)$, which is achieved when $v_1$ is a unit eigenvector of $X^TX$ 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 $k$th loading vector $v_k$ as the solution to the constrained optimization problem

$$\max_{v} v^TX^TXv \quad \text{subject to} \quad v^Tv = 1$$

$$v^Tv_i = 0, \quad i = 1, \ldots, k - 1$$

We claim that $v_k$ is a unit eigenvector of $X^TX$ corresponding to its $k$th 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 $v_1, \ldots, v_k$, and consider the problem of finding $v_{k+1}$.

By the inductive hypothesis, we know that $v_1, \ldots, v_k$ are orthonormal eigenvectors of $X^TX$. Denote the $i$th largest eigenvalue of $X^TX$ by $\lambda_i$, noting that $X^TXv_i = \lambda_i v_i$.

The Lagrangian of the objective function is

$$L(v) = v^TX^TXv - \lambda(v^Tv - 1) + \sum_{i=1}^{k} \eta_i v_i^Tv_i$$

First-order necessary conditions for optima imply that

$$0 = \nabla L(v_{k+1}) = 2X^TXv_{k+1} - 2\lambda v_{k+1} + \sum_{i=1}^{k} \eta_i v_i$$

This implies that, if $v_{k+1}$ is orthogonal to $v_1, \ldots, v_k$ (as we constrain it to be), then

$$0 = v_{j}^T0 = 2v_{j}^TX^TXv_{k+1} - 2\lambda v_{j}^Tv_{k+1} + \sum_{i=1}^{k} \eta_i v_{i}^Tv_{j}$$

$$= 2(v_{j}^TXv_{j})^Tv_{k+1} + \eta_j$$

$$= 2(\lambda_j v_{j})^Tv_{k+1} + \eta_j$$

$$= 2\lambda_j v_{j}^Tv_{k+1} + \eta_j$$
for all \( j = 1, \ldots, k \).

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 \( \lambda \). As before, the value of the objective function is then \( \lambda \). To maximize, we want the largest eigenvalue, but we must respect the constraints that \( \mathbf{v}_{k+1} \) is orthogonal to \( \mathbf{v}_1, \ldots, \mathbf{v}_k \). 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 \( \mathbf{v}_1, \ldots, \mathbf{v}_k \), 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 \( k \)th principal component of a datapoint \( \mathbf{x}_i \in \mathbb{R}^d \) is defined as the scalar projection of \( \mathbf{x}_i \) onto the \( k \)th loading vector \( \mathbf{v}_k \), i.e. \( \mathbf{x}_i \mathbf{v}_k^\top \). 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 \( \mathbf{v}_1, \ldots, \mathbf{v}_k \).

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](image)

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.
3.1. PRINCIPAL COMPONENT ANALYSIS

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 \( v \) that maximizes \( v^\top X^\top X v \). It turns out that there is a result, sometimes referred to as the variational characterization of eigenvalues, that tells us which vectors \( v \) achieve this. The key idea in the proof is a length-preserving change of coordinates.

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

\[
\lambda_{\min}(A) \leq v^\top A v \leq \lambda_{\max}(A)
\]

where for both bounds, equality holds if and only if \( v \) is a corresponding eigenvector.

**Proof.** We show only the max case because the argument for the min case is entirely analogous. Since \( A \) is symmetric, we can decompose it as \( A = Q \Lambda Q^\top \), where \( Q \in \mathbb{R}^{d \times d} \) is orthogonal and \( \Lambda = \text{diag}(\lambda_1, \ldots, \lambda_d) \) contains the eigenvalues of \( A \). For any \( v \) satisfying \( \|v\|_2 = 1 \), define \( z = Q^\top v \), noting that the relationship between \( v \) and \( z \) is one-to-one because \( Q \) is invertible and that \( \|z\|_2 = 1 \) because \( Q \) is orthogonal. Hence

\[
\max_{\|v\|_2 = 1} v^\top A v = \max_{\|z\|_2 = 1} z^\top A z = \max_{\|z\|_2 = 1} \sum_{i=1}^d \lambda_i z_i^2
\]

We note that

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

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

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

Defining \( I = \{i : \lambda_i = \lambda_{\max}(A)\} \), 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 \( j \not\in I \). Suppose \( z^* \) satisfies this condition. Then writing \( q_1, \ldots, q_d \) for the columns of \( Q \), we have

\[
v^* = Qz^* = \sum_{i=1}^d z_i^* q_i = \sum_{i \in I} z_i^* q_i
\]

Recall that \( q_1, \ldots, q_d \) are eigenvectors of \( A \) and form an orthonormal basis for \( \mathbb{R}^d \). Therefore by construction, the set \( \{q_i : i \in I\} \) forms an orthonormal basis for the eigenspace of \( \lambda_{\max}(A) \). Hence \( v^* \), which is a linear combination of these, lies in that eigenspace and thus is an eigenvector of \( A \) corresponding to \( \lambda_{\max}(A) \).

Conversely, suppose \( v \in \mathbb{R}^d \) is unit-length but not an eigenvector corresponding to \( \lambda_{\max}(A) \). The vectors \( q_1, \ldots, q_d \) are still a basis for \( \mathbb{R}^d \), so we have a unique expansion

\[
v = z_1 q_1 + \cdots + z_d q_d
\]
Since $v$ does not lie in the eigenspace of $\lambda_{\text{max}}(A)$, one of the components $z_j$ must be nonzero for an index $j \notin I$, so equality does not hold in the bound above.

With this result established, we see that the vector we seek (which maximizes $v^\top X^\top Xv$) must be an eigenvector corresponding to $\lambda_{\text{max}}(X^\top X)$. 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 $x$ onto the subspace spanned by a unit vector $v$ equals $v$ scaled by the scalar projection of $x$ onto $v$:

$$P_v x = (x^\top v)v$$

Suppose we want to minimize the total reconstruction error:

$$\sum_{i=1}^{n} \|x_i - P_v x_i\|^2$$

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

$$\|x - P_v x\|^2 + \|P_v x\|^2 = \|x\|^2$$

Thus

$$\sum_{i=1}^{n} \|x_i - P_v x_i\|^2 = \sum_{i=1}^{n} \left(\|x_i\|^2 - \|P_v x_i\|^2\right)$$

$$= \sum_{i=1}^{n} \|x_i\|^2 - \sum_{i=1}^{n} \|(x_i^\top v)v\|^2$$

$$= \sum_{i=1}^{n} \|x_i\|^2 - \sum_{i=1}^{n} (x_i^\top v)^2$$

Then since the first term $\sum_{i=1}^{n} \|x_i\|^2$ is constant with respect to $v$, minimizing reconstruction error is equivalent to maximizing $\sum_{i=1}^{n} (x_i^\top v)^2$, 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 $\tilde{X}_k$ is the best rank-$k$ approximation to $X$ in the Frobenius norm. To see this, first note that (writing $X = \sum_{i=1}^{d} \sigma_i u_i v_i^\top$)

$$\tilde{X}_k = XV_k V_k^\top = \sum_{i=1}^{d} \sigma_i u_i v_i^\top V_k V_k^\top$$

By orthonormality, the product $v_i^\top V_k$ results in a $k$-dimensional row vector with 1 in the $i$th place and 0 everywhere else, i.e. $e_i^\top$, as long as $i \leq k$. In this case,

$$v_i^\top V_k V_k^\top = e_i^\top V_k^\top = (V_k e_i)^\top = v_i^\top$$
If $i > k$, $v_i^\top V_k = 0^\top$, so the term disappears. Therefore we see that

$$
\hat{X}_k = \sum_{i=1}^{d} \sigma_i u_i v_i^\top V_k V_k^\top = \sum_{i=1}^{k} \sigma_i u_i v_i^\top
$$

which is the best rank-$k$ approximation to $X$ 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 $z_i \sim \mathcal{N}(0, I)$ which we cannot observe, and the actual $d$-dimensional observation is distributed conditionally on this latent variable as $x_i | z_i \sim \mathcal{N}(\Lambda z_i + \mu, \Psi)$

Here $\Lambda \in \mathbb{R}^{d \times k}$ and $\mu \in \mathbb{R}^d$ are parameters to be estimated. Since $z_i$ is Gaussian and $x_i | z_i$ is Gaussian, so its marginal $x_i$ is Gaussian. In particular, by integrating out the latent variable

$$p(x_i) = \int_z p(x_i, z) \, dz = \int_z p(x_i | z) p(z) \, dz$$

one can show that $x_i \sim \mathcal{N}(\mu, \Lambda \Lambda^\top + \Psi)$

It is common to assume $\Psi = \sigma^2 I$. In this case, if we let $\sigma^2 \to 0$, we recover the original PCA solution in the sense that the columnspace of $\Lambda_{\text{MLE}}$ approaches the PCA subspace (i.e. the columnspace of $V_k$).  

### 3.2 Canonical Correlation Analysis

The **Pearson Correlation Coefficient** $\rho(X, Y)$ is a way to measure how linearly related (in other words, how well a linear model captures the relationship between) random variables $X$ and $Y$.

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

Here are some important facts about it:

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

$$\rho(aX + b, cY + d) = \frac{\text{Cov}(aX + b, cY + d)}{\sqrt{\text{Var}(aX + b) \text{Var}(cY + d)}}$$

$^2$ See Tipping and Bishop’s original paper for derivations and more information.
\[
\text{Cov}(aX, cY) = \frac{a \cdot c \cdot \text{Cov}(X, Y)}{\sqrt{a^2 \text{Var}(X) \cdot c^2 \text{Var}(Y)}}
\]

The correlation is defined in terms of random variables rather than observed data. Assume now that \(x, y \in \mathbb{R}^n\) 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 \(\mu\),

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

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

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

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

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

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

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

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

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

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

Correlation and Gaussians
Here’s a neat fact: if $X$ and $Y$ are jointly Gaussian, i.e.

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

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

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

Then

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

so

$$\begin{bmatrix} \sigma^{-1}_x & 0 \\ 0 & \sigma^{-1}_y \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} \sim \mathcal{N} \left( 0, \begin{bmatrix} \sigma^{-1}_x & 0 \\ 0 & \sigma^{-1}_y \end{bmatrix} \Sigma \begin{bmatrix} \sigma^{-1}_x & 0 \\ 0 & \sigma^{-1}_y \end{bmatrix}^\top \right)$$

$$\sim \mathcal{N} \left( 0, \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix} \right)$$

Canonical Correlation Analysis
Canonical Correlation Analysis (CCA) is a method of modeling the relationship between two point sets by making use of the correlation coefficient. Formally, given zero-mean random vectors $X_{rv} \in \mathbb{R}^p$ and $Y_{rv} \in \mathbb{R}^q$, we want to find projection vectors $u \in \mathbb{R}^p$ and $v \in \mathbb{R}^q$ that maximizes the correlation between $X_{rv}^\top u$ and $Y_{rv}^\top v$:

$$\max_{u,v} \rho(X_{rv}^\top u, Y_{rv}^\top v) = \max_{u,v} \frac{\text{Cov}(X_{rv}^\top u, Y_{rv}^\top v)}{\sqrt{\text{Var}(X_{rv}^\top u) \text{Var}(Y_{rv}^\top v)}}$$

Observe that

$$\text{Cov}(X_{rv}^\top u, Y_{rv}^\top v) = \mathbb{E}[(X_{rv}^\top u - \mathbb{E}[X_{rv}^\top u])(Y_{rv}^\top v - \mathbb{E}[Y_{rv}^\top v])]$$

$$= \mathbb{E}[u^\top (X_{rv} - \mathbb{E}[X_{rv}]) (Y_{rv} - \mathbb{E}[Y_{rv}])^\top v]$$

$$= u^\top \mathbb{E}[(X_{rv} - \mathbb{E}[X_{rv}]) (Y_{rv} - \mathbb{E}[Y_{rv}])] v$$

$$= u^\top \text{Cov}(X_{rv}, Y_{rv}) v$$

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

$$\text{Var}(X_{rv}^\top u) = u^\top \text{Cov}(X_{rv}, X_{rv}) u$$

$$\text{Var}(Y_{rv}^\top v) = v^\top \text{Cov}(Y_{rv}, Y_{rv}) v$$
so the correlation writes
\[ \rho(X_{rv}^\top u, Y_{rv}^\top v) = \frac{u^\top \text{Cov}(X_{rv}, Y_{rv})v}{\sqrt{u^\top \text{Cov}(X_{rv}, X_{rv})u \cdot v^\top \text{Cov}(Y_{rv}, Y_{rv})v}} \]

Unfortunately, we do not have access to the true distributions of \( X_{rv} \) and \( Y_{rv} \), so we cannot compute these covariances matrices. However, we can estimate them from data. Assume now that we are given zero-mean data matrices \( X \in \mathbb{R}^{n \times p} \) and \( Y \in \mathbb{R}^{n \times q} \), where the rows of the matrix \( X \) are i.i.d. samples \( x_i \in \mathbb{R}^p \) from the random variable \( X_{rv} \), and correspondingly for \( Y_{rv} \). Then
\[ \text{Cov}(X_{rv}, Y_{rv}) = \mathbb{E}[(X_{rv} - \mathbb{E}[X_{rv}])(Y_{rv} - \mathbb{E}[Y_{rv}])^\top] = \mathbb{E}[X_{rv}Y_{rv}^\top] \approx \frac{1}{n} \sum_{i=1}^{n} x_i y_i^\top = \frac{1}{n} X^\top Y \]
where again the sample-based approximation is justified by the law of large numbers. Similarly,
\[ \text{Cov}(X_{rv}, X_{rv}) = \mathbb{E}[X_{rv}X_{rv}^\top] \approx \frac{1}{n} \sum_{i=1}^{n} x_i x_i^\top = \frac{1}{n} X^\top X \]
\[ \text{Cov}(Y_{rv}, Y_{rv}) = \mathbb{E}[Y_{rv}Y_{rv}^\top] \approx \frac{1}{n} \sum_{i=1}^{n} y_i y_i^\top = \frac{1}{n} Y^\top Y \]

Plugging these estimates in for the true covariance matrices, we arrive at the problem
\[ \max_{u,v} \frac{u^\top \left( \frac{1}{n} X^\top Y \right) u}{\sqrt{u^\top \left( \frac{1}{n} X^\top X \right) u \cdot v^\top \left( \frac{1}{n} Y^\top Y \right) v}} = \max_{u,v} \frac{u^\top X^\top Y v}{\sqrt{u^\top X^\top X u \cdot v^\top Y^\top Y v}} \]

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 \( X \) and \( Y \) such that the maximization problem is equivalent but easier to understand.

1. First, let’s choose matrices \( W_x, W_y \) to whiten \( X \) and \( Y \). This will make the (co)variance matrices \( (XW_x)^\top (XW_x) \) and \( (YW_y)^\top (YW_y) \) become identity matrices and simplify our expression. To do this, note that \( X^\top X \) is positive definite (and hence symmetric), so we can employ the eigendecomposition
\[ X^\top X = U_x S_x U_x^\top \]
Since
\[ S_x = \text{diag}(\lambda_1(X^\top X), \ldots, \lambda_d(X^\top X)) \]
where all the eigenvalues are positive, we can define the “square root” of this matrix by taking the square root of every diagonal entry:
\[ S_x^{1/2} = \text{diag} \left( \sqrt{\lambda_1(X^\top X)}, \ldots, \sqrt{\lambda_d(X^\top X)} \right) \]
Then, defining \( W_x = U_x S_x^{-1/2} U_x^\top \), we have
\[ (XW_x)^\top (XW_x) = W_x^\top X^\top X W_x \]
\[ = U_x S_x^{-1/2} U_x^\top U_x S_x U_x^\top U_x S_x^{-1/2} U_x^\top \]
\[ = U_x S_x^{-1/2} S_x S_x^{-1/2} U_x^\top \]
which shows that $W_x$ is a whitening matrix for $X$. The same process can be repeated to produce a whitening matrix $W_y = U_yS_y^{-1/2}U_y^\top$ for $Y$.

Let’s denote the whitened data $X_w = XW_x$ and $Y_w = YW_y$. Then by the change of variables $u_w = W_x^{-1}u$, $v_w = W_y^{-1}v$,

$$\max_{u,w} \hat{\rho}(Xu, Yv) = \max_{u,w} \frac{(Xu)^\top Yv}{\sqrt{(Xu)^\top Xu}(Yv)^\top Yv}$$

$$= \max_{u,w} \frac{(XW_x^{-1}u)^\top YW_y^{-1}v}{\sqrt{(XW_x^{-1}u)^\top XW_x^{-1}u}(YW_y^{-1}v)^\top YW_y^{-1}v}$$

$$= \max_{u,w} \frac{(Xw_u)^\top Yw_v}{\sqrt{(Xw_u)^\top Xw_u}(Yw_v)^\top Yw_v}$$

$$= \max_{u,w} \frac{u_w^\top Xw_u}{\sqrt{u_w^\top u_w \cdot v_w^\top v_w}}$$

$$\hat{\rho}(Xw_u, Yw_v)$$

Note we have used the fact that $X_w^\top X_w$ and $Y_w^\top Y_w$ are identity matrices by construction.

2. Second, let’s choose matrices $D_x$, $D_y$ to decorrelate $X_w$ and $Y_w$. This will let us simplify the covariance matrix $(X_wD_x)^\top(Y_wD_y)$ into a diagonal matrix. To do this, we’ll make use of the SVD:

$$X_w^\top Y_w = USV^\top$$

The choice of $U$ for $D_x$ and $V$ for $D_y$ accomplishes our goal, since

$$(X_wU)^\top(Y_wV) = U^\top X_w^\top Y_wV = U^\top(USV^\top)V = S$$

Let’s denote the decorrelated data $X_d = X_wD_y$ and $Y_d = Y_wW_y$. Then by the change of variables $u_d = D_x^{-1}u_w = D_x^\top u_w$, $v_d = D_y^{-1}v_w = D_y^\top v_w$,

$$\max_{u_d,v_d} \hat{\rho}(X_d u_d, Y_d v_d) = \max_{u_d,v_d} \frac{(X_d u_d)^\top Y_d v_d}{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}}$$

$$= \max_{u_d,v_d} \frac{(X_d D_x^{-1}u_w)^\top Y_d u_d}{\sqrt{(D_x u_w)^\top D_x u_w \cdot (D_y v_w)^\top D_y v_w}}$$

$$= \max_{u_d,v_d} \frac{D_y v_d}{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}}$$

$$= \max_{u_d,v_d} \frac{u_d^\top S v_d}{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}}$$
Without loss of generality, suppose $u_d$ and $v_d$ are unit vectors\(^3\) so that the denominator becomes 1, and we can ignore it:

$$\max_{u_d, v_d} \frac{u_d^T Sv_d}{\sqrt{u_d^T u_d \cdot v_d^T v_d}} = \max_{\|u_d\| = 1 \|v_d\| = 1} \frac{u_d^T Sv_d}{\|u_d\| \|v_d\|} = \max_{\|u_d\| = 1 \|v_d\| = 1} u_d^T Sv_d$$

The diagonal nature of $S$ implies $S_{ij} = 0$ for $i \neq j$, so our simplified objective expands as

$$u_d^T Sv_d = \sum_i \sum_j (u_d)_i S_{ij} (v_d)_j = \sum_i S_{ii} (u_d)_i (v_d)_i$$

where $S_{ii}$, the singular values of $X_w^T Y_w$, are arranged in descending order. Thus we have a weighted sum of these singular values, where the weights are given by the entries of $u_d$ and $v_d$, 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 $u_d$ and $v_d$ to 1, and the rest to 0:

$$u_d = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^p \quad \quad \quad \quad \quad v_d = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^q$$

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, $u_d$ and $v_d$ 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 ($u$ and $v$), we must invert the changes of variables we made:

$$u = W_x u_w = W_x D_x u_d \quad \quad \quad \quad \quad v = W_y v_w = W_y D_y v_d$$

More generally, to get the best $k$ directions, we choose

$$U_d = \begin{bmatrix} I_k \\ 0_{p-k,k} \end{bmatrix} \in \mathbb{R}^{p \times k} \quad \quad \quad \quad \quad V_d = \begin{bmatrix} I_k \\ 0_{q-k,k} \end{bmatrix} \in \mathbb{R}^{q \times k}$$

where $I_k$ denotes the $k$-dimensional identity matrix. Then

$$U = W_x D_x U_d \quad \quad \quad \quad \quad V = W_y D_y V_d$$

Note that $U_d$ and $V_d$ have orthogonal columns. The columns of $U$ and $V$, 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 $U_d, V_d$).

**Comparison with PCA**

An advantage of CCA over PCA is that it is invariant to scalings and affine transformations of $X$ and $Y$. Consider a simplified scenario in which two matrix-valued random variables $X, Y$ satisfy $Y = X + \epsilon$ where the noise $\epsilon$ has huge variance. What happens when we run PCA on $Y$? Since

---

\(^3\) Why can we assume this? Observe that the value of the objective does not change if we replace $u_d$ by $\alpha u_d$ and $v_d$ by $\beta v_d$, where $\alpha$ and $\beta$ are any positive constants. Thus if there are maximizers $u_d, v_d$ which are not unit vectors, then $u_d/\|u_d\|$ and $v_d/\|v_d\|$ (which are unit vectors) are also maximizers.
PCA maximizes variance, it will actually project $Y$ (largely) into the column space of $\epsilon$! However, we’re interested in $Y$’s relationship to $X$, not its dependence on noise. How can we fix this? As it turns out, CCA solves this issue. Instead of maximizing variance of $Y$, we maximize correlation between $X$ and $Y$. 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 $Y$ from $X$ (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 $A$ that linearly relates $XU$ and $YV$. We can accomplish this with ordinary least squares.

Denote the projected data matrices by $X_c = XU$ and $Y_c = YV$. Observe that $X_c$ and $Y_c$ are zero-mean because they are linear transformations of $X$ and $Y$, which are zero-mean. Thus we can fit a linear model relating the two:

$$Y_c \approx X_c A$$

The least-squares solution is given by

$$A = (X_c^T X_c)^{-1} X_c^T Y_c = (U^T X^T X U)^{-1} U^T X^T Y V$$

However, since what we really want is an estimate of $Y$ given new (zero-mean) observations $\tilde{X}$ (or vice-versa), it’s useful to have the entire series of transformations that relates the two. The predicted canonical variables are given by

$$\hat{Y}_c = \tilde{X}_c A = \tilde{X} U (U^T X^T X U)^{-1} U^T X^T Y V$$

Then we use the canonical variables to compute the actual values:

$$\hat{Y} = \hat{Y}_c (V^T V)^{-1} V^T = \tilde{X} U (U^T X^T X U)^{-1} (U^T X^T Y V) (V^T V)^{-1} V^T$$

We can collapse all these terms into a single matrix $A_{eq}$ that gives the prediction $\hat{Y}$ from $\tilde{X}$:

$$A_{eq} = \underbrace{\left[ \sum_{\text{projection}} \right]}_{\text{projection}} \underbrace{\left[ \sum_{\text{whitening}} \right]}_{\text{whitening}} \underbrace{\left[ \sum_{\text{decorrelation}} \right]}_{\text{decorrelation}} \underbrace{\left[ \sum_{\text{projection back}} \right]}_{\text{projection back}}$$
4.1 Nonlinear Least Squares

All the models we’ve seen so far are linear in the parameters we’re trying to learn. That is, our prediction \( \hat{y} = f(x; \theta) \) is some linear function of the parameters \( \theta \). For example, in OLS, \( \theta = w \) and the residuals \( r_i \) are computed by \( y_i - w^\top x_i \), which is linear in the components of \( w \). In the case of least-squares polynomial regression, the predicted value is not a linear function of the input \( x \), but it is still a linear function of the augmented input \( \phi(x_i) \).

However, we may have models which are nonlinear functions of their parameters. Let’s consider a motivating example. Suppose we want to estimate the 2D position \( \theta = (\theta_1, \theta_2) \) of some entity, for example a robot. The information we have to work with are noisy distance estimates \( Y_i \in \mathbb{R} \) from \( n \) sensors whose positions \( x_i \in \mathbb{R}^2 \) are fixed and known. If we assume i.i.d. Gaussian noise as usual, our statistical model has the form

\[
Y_i = \|x_i - \theta\|_2 + Z_i, \quad Z_i \sim \mathcal{N}(0, \sigma^2), \quad i = 1, \ldots, n
\]

where

\[
\|x_i - \theta\|_2 = \sqrt{(x_{i1} - \theta_1)^2 + (x_{i2} - \theta_2)^2}
\]

Our prediction is

\[
\hat{y} = f(x; \theta) = \|x - \theta\|_2
\]

which is clearly not linear in \( \theta \).

MLE Formulation

More generally, let us assume a model where \( f \) is now some arbitrary differentiable function and \( \theta \in \mathbb{R}^d \):

\[
Y_i = f(x_i; \theta) + Z_i, \quad Z_i \sim \mathcal{N}(0, \sigma^2), \quad i = 1, \ldots, n
\]

Note that this implies \( Y_i \mid x_i \sim \mathcal{N}(f(x_i; \theta), \sigma^2) \). The maximum likelihood estimator is given by

\[
\hat{\theta}_{\text{MLE}} = \arg \max_{\theta} \ell(\theta; X, y)
\]

\[
= \arg \max_{\theta} \sum_{i=1}^{n} \log p(y_i \mid x_i, \theta)
\]
\[
= \arg \max_{\theta} \sum_{i=1}^{n} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left( -\frac{(y_i - f(x_i; \theta))^2}{2\sigma^2} \right)
\]
\[
= \arg \max_{\theta} \sum_{i=1}^{n} \left[ -\frac{1}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} (y_i - f(x_i; \theta))^2 \right]
\]
\[
= \arg \min_{\theta} \sum_{i=1}^{n} (y_i - f(x_i; \theta))^2
\]

Observe that the objective function is a sum of squared residuals as we’ve seen before, even though the function \(f\) is nonlinear in general. For this reason this method is called **nonlinear least squares**.

Motivated by the MLE formulation above, we consider the following optimization problem:

\[
\min_{\theta} \epsilon(\theta) = \min_{\theta} \sum_{i=1}^{n} (y_i - f(x_i; \theta))^2
\]

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:

\[
\nabla_\theta \epsilon = 2 \sum_{i=1}^{n} (y_i - f(x_i; \theta)) \nabla_\theta f(x_i; \theta) = 0
\]

In compact matrix notation:

\[
\nabla_\theta \epsilon = J(\theta)^\top (y - F(\theta)) = 0
\]

where

\[
F(\theta) = \begin{bmatrix}
    f(x_1; \theta) \\
    \vdots \\
    f(x_n; \theta)
\end{bmatrix}, \quad J(\theta) = \begin{bmatrix}
    \nabla_\theta f(x_1; \theta) \\
    \vdots \\
    \nabla_\theta f(x_n; \theta)
\end{bmatrix}
\]

\(J\) is also referred to as the **Jacobian** of \(F\). Observe that in the special case when \(f\) is linear in \(\theta\) (i.e. \(f(x_i; \theta) = \theta^\top x_i\)), the gradient \(\nabla_\theta \epsilon\) will only have \(\theta\) in one place because the term \(\nabla_\theta f(x_i; \theta)\) will only depend on \(x_i\):

\[
\nabla_\theta \epsilon = 2 \sum_{i=1}^{n} (y_i - \theta^\top x_i) \nabla_\theta (\theta^\top x_i) = 2 \sum_{i=1}^{n} (y_i - \theta^\top x_i) x_i
\]

and we can derive a closed-form solution for \(\theta\), arriving at the OLS solution:

\[
2X^\top (y - X\theta) = 0
\]

\[
2X^\top y - 2X^\top X\theta = 0
\]

\[
X^\top y = X^\top X\theta
\]

\[
\theta = (X^\top X)^{-1} X^\top y
\]

However, in the general case where \(f\) is nonlinear in \(\theta\), it would not be possible to write out a closed-form solution for \(\theta\).

**Remark:** Without more assumptions on \(f\), the NLS objective is not convex in general. This means that the first-order optimality condition is a **necessary** but not **sufficient** condition for a local minimum. That is, it is possible that the derivative is zero for some value of \(\theta\), but that value is not a local minimum. It could be a saddle point, or worse, a local maximum! Even if it is a minimum, it may not be the global minimum.


Gauss-Newton Algorithm

Since there is no closed-form solution to the nonlinear least squares optimization problem, we must resort to an iterative algorithm instead. One such algorithm is the *Gauss-Newton algorithm*. 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 $\theta$ at iteration $k$, which we denote $\theta^{(k)}$. We consider the first-order approximation of $F(\theta)$ about $\theta^{(k)}$:

$$F(\theta) \approx \bar{F}(\theta) = F(\theta^{(k)}) + \nabla_\theta F(\theta^{(k)})(\theta - \theta^{(k)})$$

$$= F(\theta^{(k)}) + J(\theta^{(k)})\Delta \theta$$

where $\Delta \theta := \theta - \theta^{(k)}$.

Now since $\bar{F}$ is linear in $\Delta \theta$ (the Jacobian and $F$ are just constants: functions evaluated at $\theta^{(k)}$), we can use the closed form solution for $\Delta \theta$ from the optimality condition to update our current guess $\theta^{(k)}$.

Applying the first-order optimality condition to the objective $\bar{F}$ yields the following equation:

$$0 = J_{\bar{F}}(\theta)^\top (y - \bar{F}(\theta)) = J(\theta^{(k)})^\top \left( y - \left( F(\theta^{(k)}) + J(\theta^{(k)})\Delta \theta \right) \right)$$

Note that the Jacobian of the linearized function $\bar{F}$, evaluated at any $\theta$, is precisely $J(\theta^{(k)})$. Writing $J = J(\theta^{(k)})$ for brevity, we have

$$J^\top y = J^\top (F(\theta^{(k)}) + J\Delta \theta)$$

$$J^\top (y - F(\theta^{(k)})) = J^\top J (\Delta \theta)$$

$$\Delta \theta = (J^\top J)^{-1}J^\top (y - F(\theta^{(k)}))$$

$$= (J^\top J)^{-1}J^\top \Delta y$$

where $\Delta y := y - F(\theta^{(k)})$. By comparing this solution to OLS, we see that it is effectively solving

$$\Delta \theta = \arg \min_{\delta \theta} ||J\delta \theta - \Delta y||^2$$

Since $\delta F \approx J\delta \theta$ is close to $\theta^{(k)}$, this is saying that we choose a change to the weights that corrects for the current error in the function values, but it bases this calculation on the linearization of $F$. Recalling that $\Delta \theta = \theta - \theta^{(k)}$, we can improve upon our current guess $\theta^{(k)}$ with the update

$$\theta^{(k+1)} = \theta^{(k)} + \Delta \theta$$

$$= \theta^{(k)} + (J^\top J)^{-1}J^\top \Delta y$$

Note that the solution will depend on the initial value $\theta^{(0)}$ in general. Here is the entire process laid out in steps:

The choice for measuring convergence is up to the practitioner. Some common choices include testing changes in the objective value:

$$\left| \frac{\epsilon^{(k+1)} - \epsilon^{(k)}}{\epsilon^{(k)}} \right| \leq \text{threshold}$$
Algorithm 1 Gauss-Newton

Initialize $\theta^{(0)}$ with some guess

while $\theta^{(k)}$ has not converged do
    Compute Jacobian with respect to the current iterate, $J = J(\theta^{(k)})$
    Compute $\Delta y = y - F(\theta^{(k)})$
    Update: $\theta^{(k+1)} = \theta^{(k)} + (J^T J)^{-1} J^T \Delta y$

end while

or in the iterates themselves:

$$\max_j \left| \frac{\Delta \theta_j}{\theta_j^{(k)}} \right| \leq \text{threshold}$$

4.2 Gradient Descent

Gradient descent is an iterative algorithm for finding local minima of differentiable functions. For an analogy, imagine walking downhill surrounded a thick fog that prevents you from seeing the height of the land around you, other than being able to tell which direction is steepest.

Recall that the gradient of $f$ at $x$, denoted $\nabla f(x)$, points in the direction of steepest ascent. Conversely, the negative gradient points in the direction of steepest descent. Therefore, if we take a small step in the direction of the negative gradient, we will decrease the value of the function.

The update performed is

$$x_{i+1} = x_i - \alpha_i \nabla f(x_i)$$

where $\alpha_i > 0$ is the step size. We may choose $\alpha_i$ to be a fixed constant, but in many cases it is decayed to zero over the course of training.

4.3 Neural Networks

Neural networks are a class of compositional function approximators. Unlike other function approximators we have seen (e.g. polynomials), they are nonlinear in their parameters.

(Aside) In information processing we have several perspectives:

- Procedural perspective – thinking in terms of an imperative programming language
- Functional perspective – mathematical equations and reasoning
• Circuit/graph perspective – information is processed as it flows through the system

We will consider neural nets from the circuit/graph perspective.

Consider the circuits below:

These circuits represent matrix multiplications, with the weights on the edges being the entries of the matrix. Assuming that the flow of information is from left to right, the circuit on the left is multiplication by a $3 \times 4$ matrix, and the circuit on the right is multiplication by a $2 \times 4$ matrix followed by multiplication by a $3 \times 2$ matrix.

Are these two circuits equally expressive? Certainly not, as the one on the left has rank at most 2, while the one on the right may have rank 3. However, the one on the left has more layers of processing, so it seems like it should be more expressive. The key thing that is missing is nonlinearity.

Let’s insert a nonlinear function, called an activation function, after these linear computations. We would like to choose this activation function to make the circuit a universal function approximator.

A key observation is that piecewise-constant functions are universal function approximators:

\[
\mathcal{u}(x) = \begin{cases} 
1 & x \geq 0 \\
0 & x < 0
\end{cases}
\]

The nonlinearity we use, then, is the step function:

\[
\mathcal{u}(x) = \begin{cases} 
1 & x \geq 0 \\
0 & x < 0
\end{cases}
\]

\[\text{This essentially means that given any continuous function, we can choose the weights such that the output of the circuit can be made arbitrarily close to the output of the given function for all inputs.}\]
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 \( x \mapsto u(a + bx) \) is a translated (and, depending on the sign of \( b \), possibly flipped) version of the step function:

- If \( c \neq 0 \), the function \( x \mapsto cu(x) \) is a vertically scaled version of the step function.

Assume that our circuit has the following structure:

The input \( x \) is one-dimensional, and the weight on \( x \) to neuron \( i \) is \( b_i \). We also introduce a constant 1, whose weight \( a_i \) into neuron \( i \) is \( a_i \). (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 network with the ability to shift the function.) The outputs of the intermediate nodes are \( a_i + b_i x \), and then we pass each of these through the activation function \( u \). The output of the network is a linear combination of the outputs of the activation functions:

\[
h(x) = \sum_{i=1}^{d} c_i u(a_i + b_i x)
\]
where $d$ is the number of intermediate neurons.

### Choosing weights

With a proper choice of $a_i$, $b_i$, and $c_i$, 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 $\{(x_k, y_k)\}_{k=1}^n$. We aim to solve the optimization problem

$$\min_{a,b,c} \sum_{k=1}^n e_k \text{subject to } f(a,b,c)$$

where

$$e_k = (y_k - h(x_k))^2$$

To run gradient descent, we need derivatives of the loss with respect to our optimization variables. We compute via the chain rule

$$\frac{\partial f}{\partial c_i} = \sum_{k=1}^n -2(y_k - h(x_k)) \frac{\partial h}{\partial c_i}(x_k) = u(a_i + b_i x_k)$$

We see that if this particular step is “off”, as in $u(a_i + b_i x_k) = 0$, then

$$\frac{\partial e_k}{\partial c_i} = 0$$

so no update will be made for that example. More notably, consider the derivative with respect to $a_i$:

$$\frac{\partial f}{\partial a_i} = \sum_{k=1}^n -2(y_k - h(x_k)) \frac{\partial h}{\partial a_i}(x_k) = 0$$

Similarly,

$$\frac{\partial f}{\partial b_i} = 0$$

The derivative at the jump is undefined, but in practice we will never hit that point of discontinuity. The bigger issue is that gradient descent will do nothing to optimize the $a_i$ and $b_i$ weights. 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

$$u(x) = \max(0, x)$$
Now we can calculate the gradients:

$$\frac{\partial f}{\partial c_i} = \sum_{k=1}^{n} -2(y_k - h(x_k)) \max(0, a_i + b_i x)$$

$$\frac{\partial f}{\partial a_i} = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \frac{\partial}{\partial a_i} \max(0, a_i + b_i x) = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \begin{cases} 0 & \text{if } a_i + b_i x < 0 \\ 1 & \text{if } a_i + b_i x > 0 \end{cases}$$

$$\frac{\partial f}{\partial b_i} = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \frac{\partial}{\partial b_i} \max(0, a_i + b_i x) = \sum_{k=1}^{n} -2(y_k - h(x_k))c_i \begin{cases} 0 & \text{if } a_i + b_i x < 0 \\ x_i & \text{otherwise} \end{cases}$$

Crucially, we see that the gradient with respect to $a$ and $b$ is not uniformly zero, unlike with the step function.

If the ReLU is active, both weights are adjustable. Depending on the gradient of the objective function, our ReLUs can move to the left or right, increase or decrease their slope, and flip direction.

If the weight initialization turns off the ReLU for every training point, then the gradient descent updates will not change the parameters of the neuron, and we say it is dead. Random initialization should result in a reasonable number of active neurons. There are more sophisticated initialization methods, such as “Glorot” initialization\(^2\), which take into account the number of connections and are more effective in practice. Leaky ReLUs, which have a small slope in the section where the ReLU is flat (say, $u(x) = 0.01 x$ when $x < 0$) are sometimes used to provide some small gradient signal to avoid dead neurons.

\(^2\) See *Understanding the difficulty of training deep feedforward neural networks*. 

Neural networks are universal function approximators

The celebrated neural network universal approximation theorem, due to Kurt Hornik\textsuperscript{3}, tells us that neural networks are universal function approximators in the following sense.

**Theorem.** Suppose \( u : \mathbb{R} \rightarrow \mathbb{R} \) is nonconstant, bounded, nondecreasing, and continuous\textsuperscript{4}, and let \( S \subseteq \mathbb{R}^n \) be closed and bounded. Then for any continuous function \( f : S \rightarrow \mathbb{R} \) and any \( \epsilon > 0 \), there exists a neural network with one hidden layer and a finite number of neurons, which we can write

\[
    h(x) = \sum_{i=1}^{d} c_i u(a_i + b_i^\top x)
\]

such that

\[
|h(x) - f(x)| < \epsilon
\]

for all \( x \in S \).

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.

\[\text{\ }\]

### 4.4 Training Neural Networks

**ReLU as Universal Function Approximators**

Last time we saw that the second-most simple universal function approximator was the **piecewise linear function**. Specifically, we talked about a specific component of piecewise linear functions called the **ReLU**, which is defined as \( f(x) = \max(0, x) \).

In our discussion of neural nets, we saw that we would have the ReLUs act on linear combinations of neural net units to introduce nonlinearity to the hypothesis function encoded by the neural net. For example, when acting on one input (and a bias term) our ReLUs will take in arguments of the form \( a + bx \). Let’s see an example of how expressive they can be. Suppose we wanted to build this function from ReLUs:

\[\text{\ }\]

\textsuperscript{3} See *Approximation Capabilities of Multilayer Feedforward Networks.*

\textsuperscript{4} Both ReLU and sigmoid satisfy these requirements.
CHAPTER 4. GRADIENT DESCENT, NEURAL NETWORKS

Figure 4.1: \( f(x_1, x_2) = \max(0, 2x_1 + 3x_2) \)

All we would need to do is center a ReLU at each hinge of the function and give it the appropriate parameters. For example, to match \( f \) from 0 to 3, we would only need the ReLU defined by \( \max(0, x - 2) \). The full function can be matched with this linear combination of ReLUs (and a constant bias term):

\[
f(x) = -1 + \max(0, x - 2) - \max(0, x - 3) + \max(0, 6 - x) - \max(0, 5 - x) + \max(0, 2x - 12)
\]

Here’s the plot on Wolfram Alpha.

In higher dimensions, i.e. when a ReLU takes in an arbitrarily long dot-product as input: \( f(x) = \max(0, w^Tx) \), the unit can be viewed as representative of a “ramp” in the higher-dimensional space. Here’s a plot of a 3-D ramp:
4.4. TRAINING NEURAL NETWORKS

Derivatives in Neural Nets

We have a very powerful tool at our disposal in neural nets, but it does us no good if we can’t train them. Let’s talk about how this happens. The output units of a neural net can be thought of as akin to a regression or hypothesis function determined by the parameters of some model. For example, in ordinary least-squares regression, we learned a hypothesis function \( f(x) = w^\top x \) determined by the parameter \( w \). It is just the same with neural nets, except that our hypothesis function can be arbitrarily complex. Consider the following neural net:

![Neural Net Diagram](image)

The hypothesis function that this neural net encodes is represented by the two outputs, \( O = [O_1, O_2] \).

Since neural net outputs are not linear functions of their inputs, there is no closed-form solution for the minimum to any reasonable loss function defined on them. Thus, we resort to using gradient descent to train them. To run gradient descent, we must calculate the derivative of the loss function with respect to the parameters of the neural net. In the network depicted above, the parameters are \( W = [w_1, w_2, w_3], a, \) and \( V = [v_1, v_2] \). These are the values of the model which we are allowed to tweak. Backpropagation is an efficient manner of computing these gradients that exploits the nested structure of neural nets.

The Chain Rule

This section is an aside meant to recall your knowledge about the chain rule in multivariable calculus. Let’s take a look at two slices of our neural net from above.

![Chain Rule Diagram](image)
If you want to compute the derivatives of upstream stuff with respect to the weights on these connections, you need only consider the input of a single connection at a time. That is to say:

$$\frac{\partial L}{\partial w_1} = \text{upstream terms} \cdot \frac{\partial H_1}{\partial w_1}$$

completely independent of $x_2$ and $x_3$.

If you want to compute the derivatives of upstream stuff with respect to the weights downstream of these connections, you’ll need to sum over the contributions of the inputs to these connections. That is to say:

$$\frac{\partial L}{\partial a} = \sum_i \text{upstream terms}_i \cdot \frac{\partial O_i}{\partial a}$$

Backpropagation

A naive way of calculating the gradients might be to differentiate the loss with respect to each parameter we’re interested in updating, one at a time. However, because the later layers of a neural net are just functions of the earlier layers, doing this would be wasteful. We can see this by taking a look at the derivatives of $L$ with respect to $v_1$, $a$, and $w_1$ in our example:

$$\frac{\partial L}{\partial v_1} = \frac{\partial L}{\partial O_1} \frac{\partial O_1}{\partial v_1}$$
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial O_1} \frac{\partial O_1}{\partial a} + \frac{\partial L}{\partial O_2} \frac{\partial O_2}{\partial a}$$
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial O_1} \frac{\partial O_1}{\partial w_1} + \frac{\partial L}{\partial O_2} \frac{\partial O_2}{\partial w_1}$$

You should notice that by invoking the chain rule, we see that the term $\frac{\partial L}{\partial O_1}$ is common to all three derivatives, and the term $\frac{\partial L}{\partial O_2}$ is common to the second two. This suggests that we should consider caching the results of the derivatives of weights in the later layers and reuse them in our computation of the derivatives of the weights of earlier layers: a dynamic programming approach.

The following is a general outline of how backpropagation might be implemented. Keep in mind that the specifics, especially those pertaining to the structure of each successive layer, will depend heavily on the architecture of the neural net in question.
4.4. TRAINING NEURAL NETWORKS

1. The **forward pass**: populate each unit of the neural net with the value it’s supposed to have (i.e., invoke all your dot-products and ReLUs).

2. Start at the upstream end, i.e. the outputs. Compute the gradient of the loss function with respect to the outputs (these are just numbers), and memoize/cache them.

3. Go back one layer. Now, treat your outputs $O_i$ as endpoints of your neural net, and compute the gradients w.r.t. the previous layer, caching these as well. The contributions to the final loss should be summed appropriately over the paths through which they influence the loss.

4. Repeat until you hit the last layer (the inputs). You should now have all the necessary components to compute the derivative of the loss function with respect to any parameter of the neural net.

**Speeding up Gradient Descent**

Backpropagation efficiently computes gradients, so we can run gradient descent to optimize neural nets. However, computing the full gradient may be expensive, particularly if we have a lot of data. Consider that the loss function in machine learning typically is an average (or sum, but this is the same up to a constant factor) of errors over the training points:

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(x_i, w), y_i)$$

By the linearity of differentiation, we easily see that

$$\nabla L(w) = \frac{1}{n} \sum_{i=1}^{n} \nabla_w \ell(h(x_i, w), y_i)$$

So computing the gradient takes time in linear in $n$. In the “big data” regime, $n$ is very large, so this cost may be unacceptable.

For this reason, we have reason to try to approximate the gradient of the loss function of a neural net by **taking a representative random sample** of the inputs to calculate the gradient over. Since gradient descent is an iterative process, we might consider forfeiting the exactness of the update of each individual iteration in exchange for better speed of computation. If we take the gradient over a random sample of these training samples at each step, we will get a noisy but unbiased estimate of the true gradient. The noise in each step is often an acceptable tradeoff in exchange for the opportunity to take many more steps.

In the regular gradient descent update, we have the policy:

$$w^{(k+1)} \leftarrow w^{(k)} - \alpha_k \nabla w L(w^{(k)})$$

In **stochastic gradient descent**, we have the update rule:

$$w^{(k+1)} \leftarrow w^{(k)} - \alpha_k G_k$$

where $G_k$ is a random variable which satisfies $\mathbb{E}[G_k] = \nabla L(w^{(k)})$. Typically $G_k$ is constructed by sampling $m < n$ training points uniformly at random (say the index is $I_k \subset \{1, \ldots, n\}$) and computing the gradient on these points only:

$$G_k = \frac{1}{m} \sum_{i \in I_k} \nabla_w \ell(h(x_i), y_i)$$
We begin by expanding the desired term:

$$\lim_{k \to \infty} \mathbb{E}[\|w^{(k)} - w^*\|^2] = 0$$

We make the following assumptions in the analysis below:

1. The loss function $f$ is $\ell$-strongly convex; that is, there exists a constant $\ell > 0$ such that $(\nabla f(x) - \nabla f(y))'(x - y) \geq \ell\|x - y\|^2$ for all $x, y$.

2. The expected squared norm of the stochastic gradient is bounded as $\mathbb{E}[\|G_k\|^2] \leq M^2 < \infty$.

We begin by expanding the desired term:

$$\mathbb{E}[\|w^{(k+1)} - w^*\|^2] = \mathbb{E}[(w^{(k+1)} - w^*)'(w^{(k+1)} - w^*)]$$

$$= \mathbb{E}[(w^{(k)} - w^* - \alpha_k G_k)'(w^{(k)} - w^* - \alpha_k G_k)]$$

$$= \mathbb{E}[\|w^{(k)} - w^*\|^2] - 2\alpha_k \mathbb{E}[(w^{(k)} - w^*)'G_k] + \alpha_k^2 \mathbb{E}[\|G_k\|^2]$$

For brevity, define

$$d_k = \mathbb{E}[\|w^{(k)} - w^*\|^2]$$

Our assumption on the expected squared norm of $G_k$ implies

$$d_{k+1} \leq d_k - 2\alpha_k \mathbb{E}[(w^{(k)} - w^*)'G_k] + \alpha_k^2 M^2$$

To evaluate the expectation, we condition on the past (i.e. all random decisions that contributed to $w^{(k)}$, including past choices of points to evaluate the gradient at). By the law of iterated expectation, we have

$$\mathbb{E}[(w^{(k)} - w^*)'G_k] = \mathbb{E}_p [\mathbb{E}[(w^{(k)} - w^*)'G_k | \text{past}]]$$

Here the inner expectation is taken over the choice of training points used to compute the stochastic gradient. But given the past (which includes $w^{(k)}$), we already know $w^{(k)}$, so

$$\mathbb{E}[(w^{(k)} - w^*)'G_k | \text{past}] = (w^{(k)} - w^*)'\mathbb{E}[G_k | \text{past}]$$

The current gradient $G_k$ depends on our new choice of evaluation point, which is presumed independent of the past and thus $\mathbb{E}[G_k | \text{past}] = \mathbb{E}[G_k] = \nabla f(w^{(k)})$, where the second equality holds because $G_k$ is an unbiased estimator for the true gradient at $w^{(k)}$. Putting all this together, we have

$$\mathbb{E}[(w^{(k)} - w^*)'G_k] = \mathbb{E}_p [(w^{(k)} - w^*)'\nabla f(w^{(k)})]$$

First order necessary conditions for optimality imply that $\nabla f(w^*) = 0$. Then by the assumption of $\ell$-strong convexity, we have

$$(w^{(k)} - w^*)'\nabla f(w^{(k)}) = (w^{(k)} - w^*)'(\nabla f(w^{(k)}) - \nabla f(w^*)) \geq \ell\|w^{(k)} - w^*\|^2$$

Taking expectations yields

$$\mathbb{E}[(w^{(k)} - w^*)'\nabla f(w^{(k)})] \geq \ell\mathbb{E}[\|w^{(k)} - w^*\|^2] = \ell d_k$$

Putting this back into our inequality for $d_{k+1}$, we get

$$d_{k+1} \leq d_k - 2\alpha_k \ell d_k + \alpha_k^2 M^2 = (1 - 2\alpha_k \ell)d_k + \alpha_k^2 M^2$$
4.4. TRAINING NEURAL NETWORKS

The $\alpha_k^2 M^2$ term was incurred by the randomness in our updates. We can try to send this term to 0 by diminishing the step size $\alpha_k$ over time – but decreasing $\alpha_k$ will also decrease the effect of the $(1 - 2\alpha_k \ell) d_k$ term, so we need to choose our step size carefully.

Setting $\alpha_k = \frac{1}{2\ell k}$ gives

$$d_{k+1} \leq \left(1 - \frac{1}{k}\right) d_k + \frac{1}{(2\ell k)^2} M^2$$

Let $S = \frac{M^2}{(2\ell)^2}$ so that this inequality becomes

$$d_{k+1} \leq \left(1 - \frac{1}{k}\right) d_k + \frac{1}{k^2} S$$

To analyze this recurrence, we expand the first few terms:

$$d_2 \leq S$$
$$d_3 \leq \left(1 - \frac{1}{2}\right) d_2 + \frac{1}{2^2} S = \frac{1}{1 \cdot 2} S + \frac{1}{2^2} S$$
$$d_4 \leq \left(1 - \frac{1}{3}\right) d_3 + \frac{1}{3^2} S = \frac{1}{1 \cdot 3} S + \frac{1}{2 \cdot 3} S + \frac{1}{3^2} S$$
$$d_5 \leq \left(1 - \frac{1}{4}\right) d_4 + \frac{1}{4^2} S = \frac{1}{1 \cdot 4} S + \frac{1}{2 \cdot 4} S + \frac{1}{3 \cdot 4} S + \frac{1}{4^2} S$$

Inductively, we find that

$$d_n \leq \frac{S}{n - 1} \sum_{j=1}^{n-1} \frac{1}{j}$$

Because the harmonic sum $\sum_{j=1}^{n-1} \frac{1}{j}$ behaves as $\ln(n)$, we see that $d_n$ is upper bounded in the limit by $S \frac{\ln(n)}{n} \to 0$. Hence $\lim_{n \to \infty} d_n = 0$ as desired.
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 \( D = \{(x_i, y_i)\}_{i=1}^{n} \) of \( n \) points, where each data point \( x_i \in \mathbb{R}^d \) 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 \( p(X, Y) \) over the inputs \( X \) and labels \( Y \), and we classify an arbitrary datapoint \( x \) with the class label that maximizes the joint probability:

\[
\hat{y} = \arg \max_k p(x, y = k)
\]

In practice, forming the joint distribution involves explicitly forming:

- A prior probability distribution over all classes \( P(k) = P(\text{class} = k) \)
- A conditional probability distribution for each class \( f_k(x) = f(x | \text{class} k) \)

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:

\[
\hat{y} = \arg \max_k p(x, y = k) = \arg \max_k P(k) \ f_k(x) = \arg \max_k p(y = k | x)
\]

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 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:

\[
\hat{y} = \arg \max_{k \in \{0,1,2,\ldots,9\}} P(\text{digit} = k) f(\text{image}|\text{digit} = k)
\]

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.

![Figure 5.1: A collection (in dark black) of linear (left) vs quadratic (right) level set boundaries in a 2D feature space](image)

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 (i.e., 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 determine the class-conditional probability distributions.

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 without considering the prior or conditional distributions, or (2) directly form a hard decision boundary without considering any probabilities in the first place.

### 5.2 Least Squares Support Vector Machine

As a first example of a simple 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 \( x \) is to estimate parameters \( w \), compute \( w^T x \), and classify \( x \) to be \( \text{sign}(w^T x) \). Geometrically, the decision boundary this produces is a hyperplane, \( w^T x = 0 \).

We need to figure out how to optimize the parameter \( w \). One simple procedure we can try is to fit
5.2. LEAST SQUARES SUPPORT VECTOR MACHINE

a least squares objective:

\[
\arg \min_w \sum_{i=1}^n \| y_i - \text{sign}(w^T x_i) \|^2 + \lambda \| w \|^2
\]

Where \( x_i, w \in \mathbb{R}^{d+1} \). Note that we have not forgotten about the bias term! Even though we are dealing with \( d \) dimensional data, \( x_i \) and \( w \) are \( d + 1 \) dimensional: we add an extra “feature” of 1 to \( x_i \), and a corresponding bias term of \( k \) in \( w \). 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 \( w \), we often take the norm of \( w' \), which is every term of \( w \) excluding the corresponding bias term. For simplicity of notation however, let’s just take the norm of \( w \).

Without the regularization term, this would be equivalent to minimizing the number of misclassified training points. Unfortunately, optimizing this is NP-hard (computationally intractable). Instead we can solve a relaxed version of this problem:

\[
\arg \min_w \sum_{i=1}^n \| y_i - w^T x_i \|^2 + \lambda \| w \|^2
\]

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 \( w^T x_i \) 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](image)

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:
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, $\mathbf{w}^T \mathbf{x}_i = -5$ 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 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.
In order to implement this idea, we re-express our objective as

$$\arg \min_w \sum_{i=1}^n \| y_i - w^T \phi(x_i) \|^2 + \lambda \|w\|^2$$

Note that $\phi$ 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 $w^T x$ 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

$$\arg \min_w \sum_{i=1}^n \| y_i - g_w(x_i) \|^2 + \lambda \|w\|^2$$

where $g_w(x_i)$ can be any function that is easy to optimize. Then we can classify using the rule

$$\hat{y}_i = \begin{cases} 
1 & g_w(x_i) > \theta \\
-1 & g_w(x_i) \leq \theta 
\end{cases}$$

Where $\theta$ is some threshold. In LS-SVM, $g_w(x_i) = x^T w$; and $\theta = 0$. Using a neural network with non-linearities as $g_w$ 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 $g_w(x_i)$ 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 $y_i = e_k$, 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 $y_i$ is a $K$-dimensional one-hot vector, so for LS-SVM, we instead have a $K \times (d + 1)$ weight matrix to optimize over:

$$\arg \min_W \sum_{i=1}^n \| y_i - W x_i \|^2 + \lambda \|W\|^2$$

To classify an arbitrary input $x$, we compute $W x$ and see which component $k$ is the largest:

$$\hat{y} = \max_k w_k^T x$$
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 $\hat{Y}$ 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 $\mathbf{w}^\top \mathbf{x}$, 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 $D = \{(x_i, y_i)\}_{i=1}^n$. In order to train our model, we need a loss function to optimize. One possibility is least squares:

$$\arg\min_w \sum_{i=1}^n \|y_i - s(w^\top x_i)\|^2 + \lambda \|w\|^2$$

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:

$$L(w) = \sum_{i=1}^n y_i \ln \left( \frac{1}{s(w^\top x_i)} \right) + (1 - y_i) \ln \left( \frac{1}{1 - s(w^\top x_i)} \right)$$

If we define $p_i = s(w^\top x_i)$, then using the properties of logs we can express this as

$$L(w) = -\sum_{i=1}^n y_i \ln p_i + (1 - y_i) \ln(1 - p_i)$$

For each $x_i$, $p_i$ 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

$$L_i(w) = \begin{cases} -\ln p_i & \text{when } y_i = 1 \\ -\ln(1 - p_i) & \text{when } y_i = 0 \end{cases}$$

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 $\hat{Y}_i \sim Bern(p_i)$ (technically we mean $\hat{Y}_i \mid x_i, w$, but we remove the conditioning terms for notational brevity), where $p_i$ is a function of $x_i$. 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 $x_i$. We have to enforce that $p_i$ is a transformation of $x_i$ that results in a number from 0 to 1 (ie. a valid probability). Hence $p_i$ cannot be, say, linear in $x_i$. One way to do achieve the 0-1 normalization is by using the sigmoid function

$$p_i = s(w^\top x_i) = \frac{1}{1 + e^{-w^\top x_i}}.$$
CHAPTER 5. CLASSIFICATION

Now we can estimate the parameters $\mathbf{w}$ via maximum likelihood. We have the problem

$$\hat{\mathbf{w}}_{\text{LR}} = \arg \max_{\mathbf{w}} P(\hat{Y}_1 = y_1, \ldots, \hat{Y}_n = y_n \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{w})$$

$$= \arg \max_{\mathbf{w}} \prod_{i=1}^n P(\hat{Y}_i = y_i \mid \mathbf{x}_i, \mathbf{w})$$

$$= \arg \max_{\mathbf{w}} \prod_{i=1}^n p_i^{y_i} (1 - p_i)^{(1 - y_i)}$$

$$= \arg \max_{\mathbf{w}} \ln \left[ \prod_{i=1}^n p_i^{y_i} (1 - p_i)^{(1 - y_i)} \right]$$

$$= \arg \max_{\mathbf{w}} \sum_{i=1}^n y_i \ln p_i + (1 - y_i) \ln (1 - p_i)$$

$$= \arg \min_{\mathbf{w}} - \sum_{i=1}^n y_i \ln p_i + (1 - y_i) \ln (1 - p_i)$$

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

$$D_{KL}(P||Q) = \sum_x P(x) \ln \frac{P(x)}{Q(x)}$$

Note that $D_{KL}$ is not a true distance metric, because it is not symmetric, ie. $D_{KL}(P||Q) \neq D_{KL}(Q||P)$ in general. It also does not satisfy the triangle inequality. However, it is always positive, ie. $D_{KL}(P||Q) \geq 0$, 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 $Y_i \sim \text{Bern}(y_i)$. $Y_i$ 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 $Y_i$.

The KL divergence from $\hat{Y}_i$ to $Y_i$ 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 $D_{KL}(Y_i||\hat{Y}_i) = 0$. Again, this is impossible for two reasons. First, if we want $D_{KL}(Y_i||\hat{Y}_i) = 0$, 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 $D_{KL}(Y_i||\hat{Y}_i) = 0$, 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 $\hat{Y}_i$), in order to minimize the total sum of KL divergences contributed by all data points:

$$\hat{\mathbf{w}}_{\text{LR}} = \arg \min_{\mathbf{w}} \sum_{i=1}^n D_{KL}(Y_i||\hat{Y}_i)$$
5.3. LOGISTIC REGRESSION

\[
= \arg \min_w \sum_{i=1}^n y_i \ln \frac{y_i}{p_i} + (1 - y_i) \ln \frac{1 - y_i}{1 - p_i}
\]

\[
= \arg \min_w \sum_{i=1}^n y_i (\ln y_i - \ln p_i) + (1 - y_i)(\ln(1 - y_i) - \ln(1 - p_i))
\]

\[
= \arg \min_w \sum_{i=1}^n (-y_i \ln p_i - (1 - y_i) \ln(1 - p_i)) + (y_i \ln y_i + (1 - y_i) \ln(1 - y_i))
\]

\[
= \arg \min_w \sum_{i=1}^n H(Y_i, \hat{Y}_i)
\]

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 \(H(Y_i, \hat{Y}_i)\). We conclude our discussion of KL divergence by noting the relation between KL divergence and cross entropy:

\[
D_{KL}(Y_i||\hat{Y}_i) = H(Y_i, \hat{Y}_i) - H(Y_i)
\]

where \(D_{KL}(Y_i||\hat{Y}_i)\) is the KL divergence from \(\hat{Y}_i\) to \(Y_i\), \(H(Y_i, \hat{Y}_i)\) is the cross entropy between \(Y_i\) and \(\hat{Y}_i\), and \(H(Y_i)\) is the entropy of \(Y_i\). Since the parameters \(W\) 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 \(y_i = e_k\), 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

\[
y_i = [0 \, 1 \, 0]
\]

(Be careful to distinguish between the class label \(y_i \in \mathbb{R}\) and its one-hot encoding \(y_i \in \mathbb{R}^K\)). 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 \(W \in \mathbb{R}^{K \times (d+1)}\). For each input \(x_i \in \mathbb{R}^{d+1}\), each class \(k\) is given a “score”

\[
z_k = w_k^\top x_i
\]

Where \(w_k \in \mathbb{R}^{d+1}\) is the \(k\)’th row of the \(W\) matrix. In total there are \(K\) raw linear scores for an arbitrary input \(x\):

\[
\begin{bmatrix}
w_1^\top x & w_2^\top x & \ldots & w_K^\top x
\end{bmatrix}
\]
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 $\hat{Y}$.

For binary logistic regression, we used the logistic function, which takes the value $w^\top x$ 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 $j$:

$$
\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}
$$

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:

$$
\hat{y} = \sigma(Wx) = [\sigma(Wx)_1 \sigma(Wx)_2 \ldots \sigma(Wx)_K]
$$

and our prediction is the class with the maximum probability:

$$
\hat{y} = \max_k \sigma(Wx)_k
$$

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 $w_0$ and $w_1$, we have that:

$$
P(\hat{Y}_i = 1 \mid x_i, W) = \sigma(Wx_i)_1 = \frac{e^{w_1^\top x_i}}{e^{w_0^\top x_i} + e^{w_1^\top x_i}} = s((w_1 - w_0)^\top x_i)
$$

$$
P(\hat{Y}_i = 0 \mid x_i, W) = \sigma(Wx_i)_0 = \frac{e^{w_0^\top x_i}}{e^{w_0^\top x_i} + e^{w_1^\top x_i}} = 1 - s((w_1 - w_0)^\top x_i)
$$

In the 2-class case, because we are only interested in the difference between $w_1$ and $w_0$, we just use a change of variables $w = w_1 - w_0$. We don’t need to know $w_1$ and $w_0$ 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 $P(Y_i = j) = \delta_{ij}$. In other words, the entire distribution is concentrated on the label for the training example. The estimated distribution $\hat{Y}_i$ comes from multiclass logistic regression, and in this case is the softmax distribution:

$$
P(\hat{Y}_i = j) = \frac{e^{w_j^\top x_i}}{\sum_{k=1}^{K} e^{w_k^\top x_i}}
$$

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{equation}
\hat{W}_{\text{MCLR}} = \arg\min_W \sum_{i=1}^n D_{KL}(Y_i || \hat{Y}_i)
\end{equation}

\begin{align*}
&= \arg\min_W \sum_{i=1}^n \sum_{j=1}^K P(Y_i = j) \ln \left( \frac{P(Y_i = j)}{P(Y_i = j)} \right) \\
&= \arg\min_W \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \ln \left( \frac{\delta_{j,y_i}}{\sigma(Wx_i)_j} \right) \\
&= \arg\min_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 (\sigma(Wx_i)_j) \\
&= \arg\min_W - \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \cdot \ln (\sigma(Wx_i)_j)
\end{align*}

\begin{align*}
&= \arg\min_W - \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \cdot \ln \left( \frac{e^{w_j^\top x_i}}{\sum_{k=1}^K e^{w_k^\top x_i}} \right) \\
&= \arg\min_W \sum_{i=1}^n H(Y_i, \hat{Y}_i)
\end{align*}

Just like binary logistic regression, we can justify the loss function with MLE as well:

\begin{equation}
\hat{W}_{\text{MCLR}} = \arg\max_W P(\hat{Y}_1 = y_1, \ldots, \hat{Y}_n = y_n | x_1, \ldots, x_n, W)
\end{equation}

\begin{align*}
&= \arg\max_W \prod_{i=1}^n P(\hat{Y}_i = y_i | x_i, W) \\
&= \arg\max_W \prod_{i=1}^n \prod_{j=1}^K P(\hat{Y}_i = j | x_i, W)^{\delta_{j,y_i}} \\
&= \arg\max_W \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \ln P(\hat{Y}_i = j | x_i, W) \\
&= \arg\max_W \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \ln \left( \frac{e^{w_j^\top x_i}}{\sum_{k=1}^K e^{w_k^\top x_i}} \right) \\
&= \arg\max_W - \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \ln \left( \frac{e^{w_j^\top x_i}}{\sum_{k=1}^K e^{w_k^\top x_i}} \right)
\end{align*}

We conclude that the loss function for multiclass logistic regression is

\begin{equation}
L(W) = - \sum_{i=1}^n \sum_{j=1}^K \delta_{j,y_i} \cdot \ln P(\hat{Y}_i = j | x_i, W)
\end{equation}
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

\[ L(w) = - \sum_{i=1}^{n} y_i \ln p_i + (1 - y_i) \ln(1 - p_i) \]

where

\[ p_i = s(w^\top x_i) = \frac{1}{1 + e^{-w^\top x_i}} \]

\[ \nabla_w L(w) = \nabla_w \left( - \sum_{i=1}^{n} y_i \ln p_i + (1 - y_i) \ln(1 - p_i) \right) \]

\[ = - \sum_{i=1}^{n} y_i \nabla_w \ln p_i + (1 - y_i) \nabla_w \ln(1 - p_i) \]

\[ = - \sum_{i=1}^{n} \frac{y_i}{p_i} \nabla_w p_i - \frac{1 - y_i}{1 - p_i} \nabla_w p_i \]

\[ = - \sum_{i=1}^{n} \left( \frac{y_i}{p_i} - \frac{1 - y_i}{1 - p_i} \right) \nabla_w p_i \]

Note that \( \nabla_z s(z) = s(z)(1 - s(z)) \), and from the chain rule we have that

\[ \nabla_w p_i = \nabla_w s(w^\top x_i) = s(w^\top x_i)(1 - s(w^\top x_i))x_i = p_i(1 - p_i)x_i \]

Plugging in this gradient value, we have

\[ \nabla_w L(w) = - \sum_{i=1}^{n} \left( \frac{y_i}{p_i} - \frac{1 - y_i}{1 - p_i} \right) p_i(1 - p_i)x_i \]

\[ = - \sum_{i=1}^{n} (y_i(1 - p_i) - (1 - y_i)(p_i)) x_i \]

\[ = - \sum_{i=1}^{n} (y_i - p_i) x_i \]

The gradient descent update is thus

\[ w^{(t+1)} = w^{(t)} + \epsilon \sum_{i=1}^{n} (y_i - p_i) x_i \]
It does not matter what initial values we pick for \( w \), because the loss function \( L(w) \) is convex and does not have any local minima. Let’s prove this, by first finding the Hessian of the loss function. The \( k,l \)th entry of the Hessian is the partial derivative of the gradient with respect to \( w_k \) and \( w_\ell \):

\[
H_{kl} = \frac{\partial^2 L(w)}{\partial w_k \partial w_\ell} = \frac{\partial}{\partial w_k} - \sum_{i=1}^{n} (y_i - p_i) x_{il} = \sum_{i=1}^{n} \frac{\partial}{\partial w_k} p_i x_{il} = \sum_{i=1}^{n} p_i (1 - p_i) x_{il}
\]

We conclude that

\[ H(w) = \sum_{i=1}^{n} p_i (1 - p_i) x_i x_i^\top \]

To prove that \( L(w) \) is convex in \( w \), we need to show that \( \forall v, v^\top H(w) v \geq 0 \):

\[
v^\top H(w) v = v^\top \sum_{i=1}^{n} p_i (1 - p_i) x_i x_i^\top v = \sum_{i=1}^{n} (v^\top x_i)^2 p_i (1 - p_i) \geq 0
\]

**Multiclass Case**

Instead of finding the gradient with respect to all of the parameters of the matrix \( W \), let’s find them with respect to one row of \( W \) at a time:

\[
\nabla_{w_\ell} L(W) = \nabla_{w_\ell} - \sum_{i=1}^{n} \sum_{j=1}^{K} \delta_{j,y_i} \cdot \ln \left( \frac{e^{w_j^\top x_i}}{\sum_{k=1}^{K} e^{w_k^\top x_i}} \right)
\]

\[
= - \sum_{i=1}^{n} \sum_{j=1}^{K} \delta_{j,y_i} \cdot \nabla_{w_\ell} \left( \ln \left( \frac{e^{w_j^\top x_i}}{\sum_{k=1}^{K} e^{w_k^\top x_i}} \right) \right)
\]

\[
= - \sum_{i=1}^{n} \sum_{j=1}^{K} \delta_{j,y_i} \cdot \left( \nabla_{w_\ell} w_j^\top x_i - \nabla_{w_\ell} \ln \sum_{k=1}^{K} e^{w_k^\top x_i} \right)
\]

\[
= - \sum_{i=1}^{n} \sum_{j=1}^{K} \delta_{j,y_i} \cdot \left( \delta_{\ell,y_i} - \frac{e^{w_\ell^\top x_i}}{\sum_{k=1}^{K} e^{w_k^\top x_i}} x_i \right)
\]

\[
= - \sum_{i=1}^{n} \left( \delta_{\ell,y_i} - P(\hat{Y}_i = \ell) \right) x_i
\]

The gradient descent update for \( w_\ell \) is thus

\[ w_\ell^{(t+1)} = w_\ell^{(t+1)} + \epsilon \sum_{i=1}^{n} \left( \delta_{\ell,y_i} - P(\hat{Y}_i = \ell) \right) x_i \]

Just as with binary logistic regression, it does not matter what initial values we pick for \( W \), because the loss function \( L(W) \) is convex and does not have any local minima.
5.4 QDA and LDA

QDA Classification

Quadratic Discriminant Analysis (QDA) is a specific generative method in which the class conditional probability distributions are independent Gaussians: \( f_k(.) \sim \mathcal{N}(\mu_k, \Sigma_k) \).

(Note: the term “discriminant” in QDA is misleading: remember that QDA is not a discriminative method, it is a generative method!)

For a particular class conditional probability distribution \( f_k(.) \), if we do not have the true means and covariances \( \mu_k, \Sigma_k \), then our best bet is to estimate them empirically with the samples in our training data that are classified as class \( k \). Recall that the MLE estimate for the parameters of \( f_k(.) \) is:

\[
\hat{\mu}_k = \frac{1}{n_k} \sum_{t_i=k} x_i \\
\hat{\Sigma}_k = \frac{1}{n_k} \sum_{t_i=k} (x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T
\]

Assuming that we know the means and covariances for all the classes, we can use Bayes’ Rule to directly solve the optimization problem

\[
\hat{y} = \arg \max_k p(k \mid x) \\
= \arg \max_k P(k) f_k(x) \\
= \arg \max_k \ln(P(k)) + \ln\left((\sqrt{2\pi})^d f_k(x)\right) \\
= \arg \max_k \ln(P(k)) - \frac{1}{2}(x - \hat{\mu}_k)^T \hat{\Sigma}_k^{-1}(x - \hat{\mu}_k) - \frac{1}{2} \ln(|\hat{\Sigma}_k|) = Q_k(x)
\]

For future reference, let’s use \( Q_k(x) = \ln\left((\sqrt{2\pi})^d P(k) f_k(x)\right) \) to simplify our notation.

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 \( \mu_k \), but LDA is different from QDA in that it requires all of the distributions to share the same covariance matrix \( \Sigma \). 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 is now computed with

\[
\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^n (x_i - \hat{\mu}_t)(x_i - \hat{\mu}_t)^T
\]
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 \( x_i \)'s by specifying a class \( k \) and the index within that class \( j \):

\[
\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \hat{\mu}_t)(x_i - \hat{\mu}_t)^T
\]

\[
= \frac{1}{n} \sum_{k=1}^{K} \sum_{j=1}^{n_k} (x_{j,k} - \hat{\mu}_k)(x_{j,k} - \hat{\mu}_k)^T
\]

\[
= \frac{1}{n} \sum_{k=1}^{K} n_k \Sigma_k
\]

\[
= \sum_{k=1}^{K} \frac{n_k}{n} \Sigma_k
\]

**LDA vs. QDA**

Up to this point, we have used the term **quadratic** in QDA and **linear** in LDA. These terms signify the shape of the **decision boundary** in \( x \)-space. Given any two classes, the decision boundary represents the points in \( x \)-space at which the two classes are equally likely.

Let’s study binary (2-class) examples for simplicity. Assume that the two classes in question are class \( A \) and class \( B \). An arbitrary point \( x \) can be classified according to three cases:

\[
\hat{y} = \begin{cases} 
  A & P(\text{class} = A \mid x) > P(\text{class} = B \mid x) \\
  B & P(\text{class} = A \mid x) < P(\text{class} = B \mid x) \\
  \text{Either} \ A \ \text{or} \ B & P(\text{class} = A \mid x) = P(\text{class} = B \mid x)
\end{cases}
\]

The decision boundary is the set of all points in \( x \)-space that are classified according to the third case. Let’s look at the form of the decision boundary according to the different scenarios possible under QDA and LDA.

**Identical Distributions**

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 \( \Sigma = \sigma^2 I \), which means that their isocontours are circles. In this case, \( f_A(.) \) and \( f_B(.) \) have identical covariances of the form \( \Sigma_A = \Sigma_B = \sigma^2 I \).
CHAPTER 5. CLASSIFICATION

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 $Q_k(x)$ from before, this can be expressed mathematically as:

$$Q_A(x) = Q_B(x)$$

$$\ln \left( \frac{1}{2} \right) - \frac{1}{2}(x - \hat{\mu}_A)^T \sigma^2 I (x - \hat{\mu}_A) - \frac{1}{2} \ln \left( |\sigma^2 I| \right) = \ln \left( \frac{1}{2} \right) - \frac{1}{2}(x - \hat{\mu}_B)^T \sigma^2 I (x - \hat{\mu}_B) - \frac{1}{2} \ln \left( |\sigma^2 I| \right)$$

$$(x - \hat{\mu}_A)^T (x - \hat{\mu}_A) = (x - \hat{\mu}_B)^T (x - \hat{\mu}_B)$$

The decision boundary is the set of points $x$ for which $\|x - \hat{\mu}_A\|_2 = \|x - \hat{\mu}_B\|_2$, which is simply the set of points that are equidistant from $\mu_A$ and $\mu_B$. This decision boundary is linear because the set of points that are equidistant from $\mu_A$ and $\mu_B$ are simply the perpendicular bisector of the segment connecting $\mu_A$ and $\mu_B$.

The next case is when the two classes are equally likely in prior, and their conditional probability distributions are anisotropic with identical covariances.
5.4. **QDA AND LDA**

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.

**Identical Distributions with Priors**

Now, let’s find the decision boundary when the two classes still have identical covariances but are not necessarily equally likely in prior:

$$Q_A(x) = Q_B(x)$$

$$\ln(P(A)) - \frac{1}{2}(x - \hat{\mu}_A)^T \hat{\Sigma}^{-1}(x - \hat{\mu}_A) - \frac{1}{2} \ln(|\hat{\Sigma}|) = \ln(P(B)) - \frac{1}{2}(x - \hat{\mu}_B)^T \hat{\Sigma}^{-1}(x - \hat{\mu}_B)$$

$$\ln(P(A)) - \frac{1}{2}(x - \hat{\mu}_A)^T \hat{\Sigma}^{-1}(x - \hat{\mu}_A) = \ln(P(B)) - \frac{1}{2}(x - \hat{\mu}_B)^T \hat{\Sigma}^{-1}(x - \hat{\mu}_B)$$

$$2 \ln(P(A)) - x^T \hat{\Sigma}^{-1} x + 2x^T \hat{\Sigma}^{-1} \hat{\mu}_A - \hat{\mu}_A^T \hat{\mu}_A = 2 \ln(P(B)) - x^T \hat{\Sigma}^{-1} x + 2x^T \hat{\Sigma}^{-1} \hat{\mu}_B - \hat{\mu}_B^T \hat{\mu}_B$$

$$2 \ln(P(A)) + 2x^T \hat{\Sigma}^{-1} \hat{\mu}_A - \hat{\mu}_A^T \hat{\mu}_A = 2 \ln(P(B)) + 2x^T \hat{\Sigma}^{-1} \hat{\mu}_B - \hat{\mu}_B^T \hat{\mu}_B$$

Simplifying, we have that

$$x^T (\hat{\Sigma}^{-1}(\hat{\mu}_A - \hat{\mu}_B)) + \left( \ln\left(\frac{P(A)}{P(B)}\right) - \frac{\hat{\mu}_A^T \hat{\mu}_A - \hat{\mu}_B^T \hat{\mu}_B}{2} \right) = 0$$

The decision boundary is the level set of a linear function $f(x) = w^T x + k$. Notice the pattern: the decision boundary is always 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 Distributions with Priors

We have that:

\[
\ln(P(A)) - \frac{1}{2}(x - \mu_A)^T \Sigma_A^{-1}(x - \mu_A) = \ln(P(B)) - \frac{1}{2}(x - \mu_B)^T \Sigma_B^{-1}(x - \mu_B)
\]

Here, unlike in LDA when \( \Sigma_A = \Sigma_B \), we cannot cancel out the quadratic terms in \( x \) 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!

Generalizing to Multiple Classes

The quadratic nature of the decision boundary in QDA and the linear nature of the decision boundary in LDA still apply to the general case when there are more than two classes. The following excellent figures from Professor Shewchuk’s notes illustrate this point:

![Figure 5.8: LDA (left) vs QDA (right): a collection of linear vs quadratic level set boundaries](image)

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 \( \mathcal{D} = \{(x_i, y_i)\}_{i=1}^n \), where \( x_i \in \mathbb{R}^d \) 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 classifier 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:

$$\forall i, \begin{cases} \mathbf{w}^\top \mathbf{x}_i - b \geq 0 & \text{if } y_i = 1 \\ \mathbf{w}^\top \mathbf{x}_i - b \leq 0 & \text{if } y_i = -1 \end{cases}$$

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.

Note that 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 decision boundaries — 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. 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.](image)

In the figure above, we consider three potential linear separators that satisfy the constraints. One could imagine that if we observed new test points that are nearby the region of C’s in the training data, they should also be of class C. One separator would incorrectly classify some of these new test points, while the others would most likely still be able to classify them correctly. To the eyes of the perceptron algorithm, all lines are perfectly valid decision boundaries. Therefore, the perceptron may not be able to generalize well to unseen data.

**Hard-Margin SVMs**

**Hard-Margin SVMs** solve the generalization problem of perceptrons by maximizing the *margin*, formally known as the minimum distance from the decision boundary to any of the training points.
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, $w$ 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 $w$ is perpendicular to the hyperplane $H = \{x : w^\top x - b = 0\}$.

Proof: consider any two points on $H$, $x_0$ and $x_1$. We will show that $(x_1 - x_0) \perp w$. Note that

$$(x_1 - x_0)^\top(w) = (x_1 - x_0)^\top((x_1 + w) - x_1) = x_1^\top w - x_0^\top w = b - b = 0$$
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 $\mathbf{z}$ to $H$ is determined by projection of $\mathbf{z} - \mathbf{x}_0$ 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:

$$\max_{m, \mathbf{w}, b} \quad m$$

s.t. \quad $y_i \frac{(\mathbf{w}^\top \mathbf{x}_i - b)}{\|\mathbf{w}\|_2} \geq m \quad \forall i$

$$m \geq 0 \quad (5.1)$$

Maximizing the margin $m$ means 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 \textit{support vectors}, hence the name “support vector machines.”

Through a series of optimization steps, we can simplify the problem by removing the margin variable and just optimizing the parameters of the hyperplane. In order to do so, we have to first introduce two new variables $\mathbf{w}'$ and $b'$ that capture the relationship between the three original variables $m$, $\mathbf{w}$, and $b$. The optimization problem then becomes:

$$\max_{\mathbf{w}', b'} \quad m$$

s.t. \quad $y_i \frac{(\mathbf{w}'^\top \mathbf{x}_i - b')}{\|\mathbf{w}'\|_2} \geq m \quad \forall i$

$$m \geq 0 \quad (5.2)$$

This simplified problem can be solved using various optimization techniques, such as quadratic programming.
Having introduced the new variables $w'$ and $b'$, the old variables $m, w,$ and $b$ are no longer relevant to the optimization problem, and we can remove them. The previous optimization problem is equivalent to

$$
\max_{w', b'} \frac{1}{||w'||_2} \quad \text{s.t.} \quad y_i(w'^\top x_i - b') \geq 1 \quad \forall i
$$

Let’s verify that (2) and (3) are equivalent. We will show that

1. The optimal value of (2) is at least as good as the optimal value of (3). Assume that the optimal values for (3) are $w'^*,$ and $b'^*$. One feasible point for (2) is $(m, w, b, w', b') = (\frac{1}{||w'||_2}, w'^*, b'^*, w'^*, b'^*)$, which leads to the same objective value as (3). Therefore, the optimal value of (2) is at least as good as that of (3).

2. The optimal value of (3) is at least as good as the optimal value of (2). Assume that the optimal values for (2) are $(m^*, w^*, b^*, w'^*, b'^*)$. One feasible point for (3) is $(w', b') = (w'^*, b'^*)$ which leads to the same objective value as (2). Therefore, the optimal value of (3) is at least as good as that of (2).

We can rewrite objective so that the problem is a minimization rather than a maximization:

$$
\min_{w', b'} \frac{1}{2}||w'||_2^2 \quad \text{s.t.} \quad y_i(w'^\top x_i - b') \geq 1 \quad \forall i
$$

At last, we have formulated the hard-margin SVM optimization problem! Using the notation $w$ and $b$, the objective of hard-margin SVMs is

$$
\min_{w, b} \frac{1}{2}||w||_2^2 \quad \text{s.t.} \quad y_i(w^\top x_i - b) \geq 1 \quad \forall i
$$

**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
5.5. SUPPORT VECTOR MACHINES

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. Formally, it introduces **slack variables** $\xi_i$, one for each training point, into the constraints:

$$
y_i(w^T x_i - b) \geq 1 - \xi_i
$$

$$
\xi_i \geq 0
$$

which is a less-strict, softer version of the hard-margin SVM constraints because it says that each point $x_i$ 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 $\xi$ is spelled “xi” and pronounced “zai.” $\xi_i$ is pronounced “zai-eye.”)

These constraints would be fruitless if we didn’t bound the values of the $\xi_i$’s, because by setting them to large values, we are essentially saying that any point may violate the margin by an arbitrarily large distance...which makes our choice of $w$ meaningless. We modify the objective function to be:

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

Where $C$ is a hyperparameter tuned through cross-validation. Putting the objective and constraints together, the soft-margin SVM optimization problem is

$$
\min_{w, b, \xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \xi_i \quad \text{s.t. \quad } y_i(w^T x_i - b) \geq 1 - \xi_i \quad \forall i
$$

$$
\xi_i \geq 0 \quad \forall i
$$

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 $\xi_i$ goes to infinity, and thus we force the $\xi_i$’s to be zero, which is exactly the setting of the hard-margin SVM.

<table>
<thead>
<tr>
<th></th>
<th>small $C$</th>
<th>large $C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desire</td>
<td>maximize margin</td>
<td>keep $\xi_i$’s small or zero</td>
</tr>
<tr>
<td>Danger</td>
<td>underfitting</td>
<td>overfitting</td>
</tr>
<tr>
<td>Outliers</td>
<td>less sensitive</td>
<td>more sensitive</td>
</tr>
</tbody>
</table>

**SVMs as Tikhonov Regularization Learning**

Consider the following regularized regression problem:

$$
\min_{w, b} \frac{1}{n} \sum_{i=1}^{n} L(y_i, w^T x_i - b) + \lambda \|w\|^2
$$

In the context of classification, the loss function that we would like to optimize is 0-1 **step loss**:

$$
L_{\text{step}}(y, w^T x - b) = \begin{cases} 1 & y(w^T x - b) < 0 \\ 0 & y(w^T x - b) \geq 0 \end{cases}
$$
The 0-1 loss is 0 if \( x \) is correctly classified and 1 otherwise. Thus minimizing \( \frac{1}{n} \sum_{i=1}^{n} L(y_i, w^\top x_i - b) \)
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).

![Step (0-1) loss, hinge loss, and squared loss. Squared loss is convex and differentiable, hinge loss is only convex, and step loss is neither.](image)

We can try to modify the 0-1 loss to be convex. The points with \( y(w^\top x - b) \geq 0 \) 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:

\[
L_{\text{hinge}}(y, w^\top x + b) = \max(1 - y(w^\top x - b), 0)
\]

Thus the regularized regression problem becomes

\[
\min_{w, b} \frac{1}{n} \sum_{i=1}^{n} \max(1 - y_i(w^\top x_i - b), 0) + \lambda\|w\|^2
\]

Recall that the original soft-margin SVM optimization problem is

\[
\min_{w, b, \xi_i} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi_i \\
\text{s.t.} \quad y_i(w^\top x_i - b) \geq 1 - \xi_i \quad \forall i \\
\xi_i \geq 0 \quad \forall i
\]

We claim these two formulations are actually equivalent. Manipulating the first constraint, we have that

\[
\xi_i \geq 1 - y_i(w^\top x_i - b)
\]

Combining with the constraint \( \xi_i \geq 0 \), we have that

\[
\xi_i \geq \max(1 - y_i(w^\top x_i - b), 0)
\]

At the optimal value of the optimization problem, these inequalities must be tight. Otherwise, we could lower each \( \xi_i \) to equal \( \max(1 - y_i(w^\top x_i - b), 0) \) and decrease the value of the objective function. Thus we can rewrite the soft-margin SVM optimization problem as

\[
\min_{w, b, \xi_i} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi_i \\
\text{s.t.} \quad \xi_i = \max(1 - y_i(w^\top x_i - b), 0) \quad \forall i
\]
5.5. SUPPORT VECTOR MACHINES

Simplifying further, we can remove the constraints:

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{n} \max(1 - y_i(w^\top x_i - b), 0)$$ (5.9)

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 $\xi_i$, or as regularized empirical risk minimization with the hinge loss.
Chapter 6

Duality, Nearest Neighbors, Sparsity

6.1 Duality

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:

- $y_i$: $\pm 1$, representing positive or negative class
- $x_i$: feature vectors in $\mathbb{R}^d$
- $\xi_i$: slack variables representing how much an $x_i$ is allowed to violate the margin
- $C$: a hyperparameter describing how severely we penalize slack
- The optimization problem for $w \in \mathbb{R}^d$ and $t \in \mathbb{R}$, the parameters of the SVM:

$$
\begin{align*}
\text{min}_{w,t,\xi_i} & \quad \frac{1}{2}||w||^2 + C \sum_{i=1}^{n} \xi_i \\
\text{s.t.} & \quad y_i(w^T x_i - t) \geq 1 - \xi_i \quad \forall i \\
& \quad \xi_i \geq 0 \quad \forall i
\end{align*}
$$

Now, we will investigate the dual of this problem, which will motivate our discussion of kernels. Before we do so, we first have to understand the primal and dual of an optimization problem.

Primal and Dual of an Optimization Problem

All optimization problems and be expressed in the standard form

$$
\begin{align*}
\text{min}_x & \quad f_0(x) \\
\text{s.t.} & \quad f_i(x) \leq 0 \quad i = 1, \ldots, m \\
& \quad h_j(x) = 0 \quad j = 1, \ldots, n
\end{align*}
$$

For the purposes of our discussion, assume that $x \in \mathbb{R}^d$. The main components of an optimization problem are:
• The **objective function** \( f_0(x) \)

• The **inequality constraints**: expressions involving \( f_i(x) \)

• The **equality constraints**: expressions involving \( h_j(x) \)

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

\[
\min_x \mathcal{L}(x) \quad (6.2)
\]

where

\[
\mathcal{L}(x) = \begin{cases} 
  f_0(x) & \text{if } f_i(x) \leq 0, \forall i \in [1, m] \text{ and } h_j(x) = 0, \forall j \in [1, n] \\
  \infty & \text{otherwise}
\end{cases}
\]

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 \( \infty \) 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 \( \mathcal{L}(x) \). 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

\[
\min_x \max_\nu \mathcal{L}(x, \nu) \quad (6.3)
\]

where

\[
\mathcal{L}(x, \nu) = \begin{cases} 
  f_0(x) + \sum_{j=1}^n \nu_j h_j(x) & \text{if } f_i(x) \leq 0, \forall i \in [1, m] \\
  \infty & \text{otherwise}
\end{cases}
\]

We are still working with an unconstrained optimization problem, except that now, we are optimizing over two sets of variables: the **primal variables** \( x \in \mathbb{R}^d \) and the **dual variables** \( \nu \in \mathbb{R}^n \). 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 \( x \) that violates the inequality constraints is still treated as \( \infty \) by the outer minimization problem over \( x \) and therefore ignored

• For any \( x \) that violates the equality constraints (meaning that \( \exists j \text{ s.t. } h_j(x) \neq 0 \)), the inner maximization problem over \( \nu \) can choose \( \nu_j \) as \( \infty \) if \( h_j(x) > 0 \) (or \( \nu_j \) as \( -\infty \) if \( h_j(x) < 0 \)) to cause the inner maximization blow off to \( \infty \), therefore being ignored by the outer minimization over \( x \)

• For any \( x \) that does not violate any of the equality or inequality constraints, the inner maximization problem over \( \nu \) is simply equal to \( f_0(x) \)
This solution comes at a cost – in an effort to remove the equality constraints, we had to add in dual variables, one for each inequality constraint. With this in mind, let’s try to do the same for the inequality constraints. Adding in dual variable \( \lambda_i \) to represent each inequality constraint, we now have

\[
\min_{x} \max_{\lambda, \nu} \mathcal{L}(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{j=1}^{n} \nu_j h_j(x)
\]

s.t. \( \lambda_i \geq 0 \quad i = 1, \ldots, m \) \hfill (6.4)

For convenience, we can place the constraints involving \( \lambda \) into the optimization variable.

\[
\min_{x} \max_{\lambda \geq 0, \nu} \mathcal{L}(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{j=1}^{n} \nu_j h_j(x)
\]

This optimization problem above is otherwise known as the \textbf{primal} (not to be confused with the \textit{primal variables}), and its optimal value is indeed \textit{equivalent} to that of the original constrained optimization problem.

\[
p^* = \min_{x} \max_{\lambda \geq 0, \nu} \mathcal{L}(x, \lambda, \nu)
\]

We can verify that this is indeed the case:

- For any \( x \) that violates the inequality constraints (meaning that \( \exists i \in [1, m] \) s.t. \( f_i(x) > 0 \)), the inner maximization problem over \( \lambda \) can choose \( \lambda_i \) as \( \infty \) to cause the inner maximization blow off to \( \infty \), therefore being ignored by the outer minimization over \( x \).

- For any \( x \) that violates the equality constraints (meaning that \( \exists j \) s.t. \( h_j(x) \neq 0 \)), the inner maximization problem over \( \nu \) can choose \( \nu_j \) as \( \infty \) if \( h_j(x) > 0 \) (or \( \nu_j \) as \( -\infty \) if \( h_j(x) < 0 \)) to cause the inner maximization blow off to \( \infty \), therefore being ignored by the outer minimization over \( x \).

- For any \( x \) that does not violate any of the equality or inequality constraints, in the inner maximization problem over \( \nu \), the expression \( \sum_{j=1}^{n} \nu_j h_j(x) \) evaluates to 0 no matter what the value of \( \nu \) is, and in the inner maximization problem over \( \lambda \), the expression \( \sum_{i=1}^{m} \lambda_i f_i(x) \) can at maximum be 0, because \( \lambda_i \) is constrained to be non-negative, and \( f_i(x) \) is non-positive. Therefore, at best, the maximization problem sets \( \lambda_i f_i(x) = 0 \), and

\[
\max_{\lambda \geq 0, \nu} \mathcal{L}(x, \lambda, \nu) = f_0(x)
\]

In its full form, the objective \( \mathcal{L}(x, \lambda, \nu) \) is called the \textbf{Lagrangian}, and it takes into account the unconstrained set of primal variables \( x \in \mathbb{R}^d \), the constrained set of dual variables \( \lambda \in \mathbb{R}^n \) corresponding to the inequality constraints, and the unconstrained set of dual variables \( \nu \in \mathbb{R}^m \) corresponding to the equality constraints. Note that our dual variables \( \lambda_i \) 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 \textbf{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.

\[
d^* = \max_{\lambda \geq 0, \nu} \min_{x} \mathcal{L}(x, \lambda, \nu) = \max_{\lambda \geq 0, \nu} g(\lambda, \nu)
\]

The dual is effectively a maximization problem (over the dual variables):

\[
d^* = \max_{\lambda \geq 0, \nu} g(\lambda, \nu)
\]
where
\[ g(\lambda, \nu) = \min_x L(x, \lambda, \nu) \]
The dual is very useful to work with, because now the inner optimization problem over \( x \) is an unconstrained problem!

**Strong Duality and KKT Conditions**

It is *always* true that the solution to the primal problem is at least as large as the solution to the dual problem:
\[ p^* \geq d^* \]  
(6.5)

This condition is known as **weak duality**.

*Proof.* We know that
\[ \forall x, \lambda \geq 0, \nu \quad \max_{\lambda \geq 0, \tilde{\nu}} L(x, \tilde{\lambda}, \tilde{\nu}) \geq L(x, \lambda, \nu) \geq \min_{\tilde{x}} L(\tilde{x}, \lambda, \nu) \]

More compactly,
\[ \forall x, \lambda \geq 0, \nu \quad \max_{\lambda \geq 0, \tilde{\nu}} L(x, \tilde{\lambda}, \tilde{\nu}) \geq \min_{\tilde{x}} L(\tilde{x}, \lambda, \nu) \]

Since this is true for all \( x, \lambda \geq 0, \nu \) this is true in particular when we set
\[ x = \arg \min_{\tilde{x}} \max_{\lambda \geq 0, \tilde{\nu}} L(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) \]

and
\[ \lambda, \nu = \arg \max_{\tilde{\lambda}, \tilde{\nu}} \min_{\tilde{x}} L(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) \]

We therefore know that
\[ p^* = \min_{\tilde{x}} \max_{\lambda \geq 0, \tilde{\nu}} L(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) \geq \max_{\lambda \geq 0, \tilde{\nu}} \min_{\tilde{x}} L(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) = d^* \]

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:
\[ p^* = d^* \]  
(6.6)

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 \( x \) 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 solutions to the dual variables of the optimization problem. The KKT conditions are composed of:

1. Primal feasibility (inequalities)
   \[ f_i(x) \leq 0, \forall i \in [1, m] \]
2. Primal feasibility (equalities)
   \[ h_j(x) = 0, \forall j \in [1, n] \]
3. Dual feasibility
\[ \lambda_i \geq 0, \forall i \in [1, m] \]

4. Complementary Slackness
\[ \lambda_i f_i(x) = 0, \forall i \in [1, m] \]

5. Stationarity
\[ \nabla_x f_0(x) + \sum_{i=1}^{m} \lambda_i \nabla_x f_i(x) + \sum_{j=1}^{n} \nu_j \nabla_x h_j(x) = 0 \]

Let’s see how the KKT conditions relate to strong duality.

**Theorem 1.** If \( x^* \) and \( \lambda^*, \nu^* \) are the primal and dual solutions respectively, with zero duality gap (i.e. strong duality holds), then \( x^*, \lambda^*, \nu^* \) also satisfy the KKT conditions.

**Proof.** KKT conditions 1, 2, 3 are trivially true, because the primal solution \( x^* \) must satisfy the primal constraints, and the dual solution \( \lambda^*, \nu^* \) must satisfy the dual constraints. Now, let’s prove conditions 4 and 5. We know that since strong duality holds, we can say that
\[ p^* = f_0(x^*) = g(\lambda^*, \nu^*) = d^* \]
\[ = \min_x L(x, \lambda^*, \nu^*) \]
\[ \leq L(x^*, \lambda^*, \nu^*) \]
\[ = f_0(x^*) + \sum_{i=1}^{m} \lambda^*_i f_i(x^*) + \sum_{j=1}^{n} \nu^*_j h_j(x^*) \]
\[ = f_0(x^*) + \sum_{i=1}^{m} \lambda^*_i f_i(x^*) \]
\[ \leq f_0(x^*) \]

In the fourth step, we can cancel the terms involving \( h_j(x^*) \) because we know that the primal solution must satisfy \( h_j(x^*) = 0 \). In the fifth step, we know that \( \lambda^*_i f_i(x^*) \leq 0 \), because \( \lambda^*_i \geq 0 \) in order to satisfy the dual constraints, and \( f_i(x^*) \leq 0 \) in order to satisfy the primal constraints. Since we established that \( f_0(x^*) = \min_x L(x, \lambda^*, \nu^*) \leq L(x^*, \lambda^*, \nu^*) \leq f_0(x^*) \), we know that all of the inequalities hold with equality and therefore \( L(x^*, \lambda^*, \nu^*) = \min_x L(x, \lambda^*, \nu^*) \). This implies KKT condition 5 (stationarity), that
\[ \nabla_x f_0(x^*) + \sum_{i=1}^{m} \lambda^*_i \nabla_x f_i(x^*) + \sum_{j=1}^{n} \nu^*_j \nabla_x h_j(x^*) = 0 \]

Finally, note that due to the equality \( f_0(x^*) + \sum_{i=1}^{m} \lambda^*_i f_i(x^*) = f_0(x^*) \), we know that \( \sum_{i=1}^{m} \lambda^*_i f_i(x^*) = 0 \). This combined with the fact that \( \forall i \lambda^*_i f_i(x^*) \leq 0 \), establishes KKT condition 4 (complementary slackness):
\[ \lambda^*_i f_i(x^*) = 0, \forall i \in [1, m] \]

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 $\bar{x}$ and $\bar{\lambda}, \bar{\nu}$ 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 $\bar{x}$ and $\bar{\lambda}, \bar{\nu}$ 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

$$\nabla_x f_0(\bar{x}) + \sum_{i=1}^m \bar{\lambda}_i \nabla_x f_i(\bar{x}) + \sum_{j=1}^n \bar{\nu}_j \nabla_x h_j(\bar{x}) = 0$$

Since the primal problem is convex, we know that $L(x, \lambda, \nu)$ is convex in $x$, and if the gradient of $L(x, \bar{\lambda}, \bar{\nu})$ at $\bar{x}$ is 0, we know that

$$\bar{x} = \arg \min_x L(x, \bar{\lambda}, \nu^*)$$

Therefore, we know that the optimal primal values for the primal problem optimize the inner optimization problem of the dual problem, and

$$g(\bar{\lambda}, \bar{\nu}) = f_0(\bar{x}) + \sum_{i=1}^m \bar{\lambda}_i f_i(\bar{x}) + \sum_{j=1}^n \bar{\nu}_j h_j(\bar{x})$$

By the primal feasibility conditions for $h_j(x)$ and the complementary slackness condition, we know that

$$g(\bar{\lambda}, \bar{\nu}) = f_0(\bar{x})$$

Now, all we have to do is to prove that $\bar{x}$ and $\bar{\lambda}, \bar{\nu}$ are primal and dual optimal, respectively. Note that since weak duality always holds, we know that

$$p^* \geq d^* = \max_{\lambda \geq 0, \nu} g(\lambda, \nu) \geq g(\bar{\lambda}, \bar{\nu}), \quad \forall \bar{\lambda} \geq 0, \bar{\nu}$$

Since we know that $p^* \geq g(\lambda, \nu)$, we can also say that

$$f_0(x) - p^* \leq f_0(x) - g(\lambda, \nu)$$

And if we have that $f_0(\bar{x}) = g(\bar{\lambda}, \bar{\nu})$ as we deduced earlier, then

$$f_0(\bar{x}) - p^* \leq f_0(\bar{x}) - g(\bar{\lambda}, \bar{\nu}) = 0 \implies p^* \geq f_0(\bar{x})$$

Since $p^*$ is the minimum value for the primal problem, we can go further by saying that $p^* \geq f_0(\bar{x})$ holds with equality and

$$p^* = f_0(\bar{x}) = g(\bar{\lambda}, \bar{\nu}) = d^*$$

Therefore, we have proven that $\bar{x}$ and $\bar{\lambda}, \bar{\nu}$ are primal and dual optimal respectively, with zero duality gap. We eventually arrived at the conclusion that strong duality does indeed hold. $\square$

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 SVMs.
6.2 The Dual of SVMs

Let’s apply our knowledge of duality to find the dual of the soft-margin SVM optimization problem.

\[
\begin{align*}
\min_{w,t,\xi} & \quad \frac{1}{2}||w||^2 + C \sum_{i=1}^{n} \xi_i \\
\text{s.t.} & \quad (1 - \xi_i) - y_i(w^T x_i - t) \leq 0 \quad \text{and} \quad -\xi_i \leq 0 \\
g(w,t,\xi) \leq 0
\end{align*}
\]

Let’s identify the primal and dual variables for the SVM problem. We will have

- Primal variables \(w, t,\) and \(\xi_i\)
- Dual variables \(\alpha_i\) corresponding to each constraint of the form \(y_i(w^T x_i - t) \geq 1 - \xi_i\)
- Dual variables \(\beta_i\) corresponding to each constraint of the form \(\xi_i \geq 0\)

For the purposes of notation, note that we are using \(\alpha\) and \(\beta\) in place of \(\lambda\), and there are no dual variables corresponding to \(\nu\) because there are no equality constraints. The lagrangian for the SVM problem is:

\[
L(w, t, \xi, \alpha, \beta) = \frac{1}{2}||w||^2 + C \sum_{i=1}^{n} \alpha_i((1 - \xi_i) - y_i(w^T x_i - t)) + \sum_{i=1}^{n} \beta_i(-\xi_i)
\]

Thus, the dual is:

\[
\max_{\alpha, \beta \geq 0} \quad g(\alpha, \beta) = \min_{w, t, \xi} \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i y_i(w^T x_i - t) + \sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n} (C - \alpha_i - \beta_i)\xi_i
\]

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 \(\alpha^*\) and \(\beta^*\), and the optimal primal values \(w^*, t^*, \xi^*_i\):

\[
\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial t} = \frac{\partial L}{\partial \xi_i} = 0
\]

- \(\nabla_w L = w^* - \sum_{i=1}^{n} \alpha^*_i y_i x_i = 0 \implies w^* = \sum_{i=1}^{n} \alpha^*_i y_i x_i\). This tells us that \(w^*\) is going to be a weighted combination of the positive-class \(x_i\)'s and negative-class \(x_i\)'s.
- \(\frac{\partial L}{\partial t} = \sum_{i=1}^{n} \alpha^*_i y_i = 0\). This tells us that the weights \(\alpha^*_i\) will be equally distributed among positive- and negative-class training points.
- \(\frac{\partial L}{\partial \xi_i} = C - \alpha^*_i - \beta^*_i = 0 \implies 0 \leq \alpha^*_i \leq C\). This tells us that the weights \(\alpha^*_i\) 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.

\[
\mathcal{L}(w, t, \xi, \alpha^*, \beta^*) = \frac{1}{2} ||w||^2 - \sum_{i=1}^{n} \alpha_i^* y_i (w^T x_i - t) + \sum_{i=1}^{n} \alpha_i^* + \sum_{i=1}^{n} (C - \alpha_i^* - \beta_i^*) \xi_i
\]

\[
= \frac{1}{2} ||w||^2 - \sum_{i=1}^{n} \alpha_i^* y_i (w^T x_i) + t \sum_{i=1}^{n} \alpha_i^* y_i + \sum_{i=1}^{n} \alpha_i^* + \sum_{i=1}^{n} (C - \alpha_i^* - \beta_i^*) \xi_i
\]

\[
= \frac{1}{2} ||w||^2 - \sum_{i=1}^{n} \alpha_i^* y_i (w^T x_i) + \sum_{i=1}^{n} \alpha_i^*
\]

We can also rewrite all the optimal primal variables \(w^*, t^*, \xi^*\) in terms of the optimal dual variables \(\alpha_i^*\):

\[
g(\alpha^*, \beta^*) = \min_{w, t, \xi} \mathcal{L}(w, t, \xi, \alpha^*, \beta^*)
\]

\[
= \mathcal{L}(w^*, t^*, \xi^*, \alpha^*, \beta^*)
\]

\[
= \frac{1}{2} ||\sum_{i=1}^{n} \alpha_i^* y_i x_i||^2 - \sum_{i=1}^{n} \alpha_i^* y_i ((\sum_{j=1}^{n} \alpha_j^* y_j x_j)^T x_i) + \sum_{i=1}^{n} \alpha_i^*
\]

\[
= \frac{1}{2} ||\sum_{i=1}^{n} \alpha_i^* y_i x_i||^2 - \sum_{i=1}^{n} (\alpha_i^* y_i x_i^T (\sum_{j=1}^{n} \alpha_j^* y_j x_j)) + \sum_{i=1}^{n} \alpha_i^*
\]

\[
= \alpha^{*T} 1 - \frac{1}{2} \alpha^{*T} Q \alpha^*
\]

where \(Q_{ij} = y_i (x_i^T x_j) y_j\) (and \(Q = (\text{diag } y) XX^T (\text{diag } y)\)).

Now, we can write the final form of the dual, which is only in terms of \(\alpha\) and \(x\) and \(y\) (Note that we have eliminated all references to \(\beta\)):

\[
\max_{\alpha} \quad \alpha^T 1 - \frac{1}{2} \alpha^T Q \alpha
\]

\[
\text{s.t.} \quad \sum_{i=1}^{n} \alpha_i y_i = 0
\]

\[
0 \leq \alpha_i \leq C \quad i = 1, \ldots, n
\]

(6.9)

**Geometric intuition**

We’ve seen that the optimal value of the dual problem in terms of \(\alpha\) is equivalent to the optimal value of the primal problem in terms of \(w, t, \xi\). But what do these dual values \(\alpha_i\) even mean? That’s a good question!

We know that the following KKT conditions are enforced:

- **Stationarity**
  
  \[ C - \alpha_i^* - \beta_i^* = 0 \]

- **Complementary slackness**

  \[ \alpha_i^* \cdot ((1 - \xi_i^*) - y_i (w^* x_i - t^*)) = 0 \]
\[ \beta^*_i : \xi^*_i = 0 \]

Here are some noteworthy relationships between \( \alpha_i \) 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 \( y_i(w^*^T x_i - t^*) - (1 - \xi_i^*) \) may be any value, and if we’re minimizing the sum of our \( \xi_i \)'s, we should have \( \xi_i^* = 0 \). So, point \( i \) lies **on or outside the margin**.

- **Case 2:** \( \alpha_i^* \) 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 \( \xi_i^* \) may be exactly 0 or nonzero. So, point \( i \) lies **on or inside the margin**.
  - **Case 2.2:** \( 0 < \alpha_i^* < C \). In this case, then \( \beta_i^* \) is nonzero and \( \xi_i^* = 0 \). But this is different from Case 1 because with \( \alpha_i^* \) nonzero, we can divide by \( \alpha_i^* \) in the complementary slackness condition and arrive at the fact that \( 1 - y_i(w^*^T x_i - t^*) = 0 \implies y_i(w^*^T x_i - t^*) = 1 \), which means \( x_i \) lies exactly on the margin determined by \( w^* \) and \( t^* \). So, point \( i \) lies **on the margin**.

Finally, let’s reconstruct the optimal primal values \( w^* \), \( t^* \), \( \xi_i^* \) from the optimal dual values \( \alpha^* \):

\[
\begin{align*}
  w^* &= \sum_{i=1}^{n} \alpha_i^* y_i x_i \\
  t^* &= \text{mean}(w^*^T x_i \quad \forall i : \quad 0 < \alpha_i^* < C) \\
  \xi_i^* &= \begin{cases} 
    1 - y_i(w^*^T x_i - t^*) & \text{if } \alpha_i^* = C, \\
    0 & \text{otherwise}
  \end{cases}
\end{align*}
\]
6.3 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 \textbf{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.\footnote{Sometimes we store the data in a specialized structure called a \textit{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.} For this reason, k-NN is sometimes referred to as “lazy learning.” The major work of k-NNs is done at testing time: to predict on a test data point $z$, we compute the $k$ closest training data points to $z$, where “closeness” can be quantified in some distance function such as Euclidean distance - these are the \textit{k nearest neighbors} to $z$. We then find the most common class $y$ among these $k$ neighbors and classify $z$ 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.

![Voronoi diagram for k-NN](image)

Figure 6.1: \textbf{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) 1-NN on 3 classes  
(b) 2-NN on 5 classes
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.

Let’s justify this reasoning formally for k-NN applied to regression tasks. Suppose we are given a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, where the labels $y_i$ are real valued scalars. We model our
hypothesis \( h(z) \) as

\[
h(z) = \frac{1}{k} \sum_{i=1}^{n} N(x_i, z, k)
\]

where the function \( N \) is defined as

\[
N(x_i, z, k) = \begin{cases} y_i & \text{if } x_i \text{ is one of the } k \text{ closest points to } z \\
0 & \text{o.w.} \end{cases}
\]

Suppose also we assume our labels \( y_i = f(x_i) + \epsilon \), where \( \epsilon \) is the noise that comes from \( \mathcal{N}(0, \sigma^2) \) and \( f \) is the true function. Let \( x_1 \ldots x_k \) be the \( k \) closest points. Let’s first derive the bias\(^2\) of our model for the given dataset \( \mathcal{D} \).

\[
(\mathbb{E}[h(z)] - f(z))^2 = \left( \mathbb{E} \left[ \frac{1}{k} \sum_{i=1}^{n} N(x_i, z, k) \right] - f(z) \right)^2 = \left( \mathbb{E} \left[ \frac{1}{k} \sum_{i=1}^{k} y_i \right] - f(z) \right)^2
\]

\[
= \left( \frac{1}{k} \sum_{i=1}^{k} \mathbb{E}[y_i] - f(z) \right)^2 = \left( \frac{1}{k} \sum_{i=1}^{k} \mathbb{E}[f(x_i) + \epsilon] - f(z) \right)^2
\]

\[
= \left( \frac{1}{k} \sum_{i=1}^{k} f(x_i) - f(z) \right)^2
\]

When \( k \to \infty \), then \( \frac{1}{k} \sum_{i=1}^{k} f(x_i) \) goes to the average label for \( x \). When \( k = 1 \), then the bias is simply \( f(x_1) - f(z) \). Assuming \( x_1 \) is close enough to \( f(z) \), the bias would likely be small when \( k = 1 \) since it’s likely to share a similar label. Meanwhile, when \( k \to \infty \), 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.

\[
\text{Var}[h(z)] = \text{Var} \left[ \frac{1}{k} \sum_{i=1}^{k} y_i \right] = \frac{1}{k^2} \sum_{i=1}^{k} \text{Var}[f(x_i) + \epsilon]
\]

\[
= \frac{1}{k^2} \sum_{i=1}^{k} \text{Var}[\epsilon]
\]

\[
= \frac{1}{k^2} \sum_{i=1}^{k} \sigma^2 = \frac{1}{k^2} \sigma^2 = \frac{\sigma^2}{k}
\]

The variance goes to 0 when \( k \to \infty \), 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.\(^2\)

\(^2\)ANN methods are beyond the scope of this course, but are useful in real applications.
6.3. NEAREST NEIGHBOR CLASSIFICATION

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 \( \epsilon^* \) is the Bayes (optimal) error. Fix and Hodges, 1951 prove that as \( n \) and \( k \) approach infinity and if \( k/n \to 0 \), 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 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

\[
\frac{(r - \epsilon)^d}{r^d} = \left(1 - \frac{\epsilon}{r}\right)^d \approx e^{-\epsilon d/r} \quad \text{for} \quad d \to \infty
\]

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.

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 $1 - O(e^{-d})$, this multivariate Gaussian will lie within the annular region $\{X : |R^2 - E(R^2)| \leq d^{1/2+\epsilon}\}$ 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 $X_i \sim \mathcal{N}(0, \sigma^2)$, $i = 1, \ldots, d$ are independent and $R^2 = \sum_{i=1}^d X_i^2$, then for every $\epsilon > 0$, the following holds:

$$\lim_{d \to \infty} P(|R^2 - E(R^2)| \geq d^{\frac{1}{2}+\epsilon}) = 0$$

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 $Var(R^2) = 2d\sigma^4$. Applying a Chebyshev bound yields

$$P(|R^2 - E(R^2)| \geq d^{\frac{1}{2}+\epsilon}) \leq \frac{2d\sigma^4}{d^{1+2\epsilon}} \xrightarrow{d \to \infty} 0$$

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:

$$D_p(x, z) = \left( \sum_{i=1}^d |x_i - z_i|^p \right)^{\frac{1}{p}}$$

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 \( \Phi \) and we want to compute the Euclidean distance from \( \Phi(x) \) to \( \Phi(z) \), then we have

\[
\| \Phi(x) - \Phi(z) \|^2 = \Phi(x)^T \Phi(x) - 2 \Phi(x)^T \Phi(z) + \Phi(z)^T \Phi(z)
\]

\[
= k(x, x) - 2k(x, z) + k(z, z)
\]

Thus if we define \( D(x, z) = \sqrt{k(x, x) - 2k(x, z) + k(z, z)} \), then we can perform Euclidean nearest neighbors in \( \Phi \)-space without explicitly representing \( \Phi \) by using the kernelized distance function \( D \).

### 6.4 Sparsity and LASSO

#### Sparsity for SVMs

Recall the objective function of the soft-margin SVM problem:

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

Note that if a point \( x_i \) 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 \( \xi \) 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 \( \xi_i \) affects the loss. A unit decrease in \( \xi_i \) 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 \( \xi_i \) is, the reward for decreasing \( \xi_i \) is constant. Of course, decreasing \( \xi_i \) may change the boundary and thus the cost attributed to the size of the margin \( \|w\|^2 \). The overall reward for decreasing \( \xi_i \) is either going to be worth the effort (greater than cost incurred from \( w \)) or not worth the effort (less than cost incurred from \( w \)). Intuitively, \( \xi_i \) 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):

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

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

There are many motivations for designing optimization problems with sparse solutions. One advantage of sparse solutions is that they speed up testing time. In the context of primal problems, if the weight vector \( w \) is sparse, then after we compute \( w \) 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 \( v \),
allowing us to discard the training points corresponding to dual weight 0, ultimately allowing for
tfaster evaluation of our hypothesis function on test points.

LASSO

Given the motivations for sparsity in SVMs, let’s modify the ridge regression objective to achieve
sparsity as well. The least absolute shrinkage and selection operator (LASSO) developed
in 1996 by Robert Tibshirani, is one method that achieves this desired effect. LASSO is identical
to the ridge regression objective, except that the L2-norm (squared) penalizing \( w \) is now changed
to an L1-norm (with no squaring term):

\[
\min_w \| Xw - y \|^2 + \lambda \| w \|_1
\]

The L1-norm of \( w \) is a sum of absolute values of its entries:

\[
\| z \|_1 = \sum_{i=1}^{d} |w_i|
\]

Compare this to the L2-norm squared of \( w \), the sum of squared values of its entries:

\[
\| z \|_2 = \sum_{i=1}^{d} w_i^2
\]

Just as in ridge regression, there is a statistical justification for using the L1-norm. Whereas
ridge regression assumes a Gaussian prior over the weights \( w \), LASSO assumes a Laplace prior
(otherwise known as a double exponential distribution) over the weights \( w \).

Let’s understand why such a simple change from the L2 to L1-norm inherently leads to a sparse
solution. For any particular component \( w_i \) of \( w \), note that 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 \( \lambda \), while for ridge regression
the equivalent “reward” is \( 2\lambda w_i \), which depends on the value of \( w_i \). Thus for the same reasons
as we presented for SVMs, LASSO achieves sparsity while ridge regression does not. There is a
compelling geometric argument behind this reasoning as well.
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 \( \|Xw - y\|^2 \). In ridge regression, each isocontour of \( \lambda \|w\|^2 \) is represented by a circle, one of which is shown in the right figure. Note that the optimal \( w \) 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 \( \|Xw - y\|^2 \), but now the L1 regularization term \( \lambda \|w\|_1 \) 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 \( w \) is 0!

### 6.5 Coordinate Descent

Note that the LASSO objective function is convex but not differentiable, due to the “pointiness” of the L1-norm. This is a problem for gradient descent techniques — in particular, LASSO zeros out features, and once these weights are set to 0 the objective function becomes non-differentiable. **Coordinate descent** is an alternative optimization algorithm that can solve convex but non-differentiable problems such as LASSO.

While SGD focuses on iteratively optimizing the value of the objective \( L(w) \) for each sample in the training set, coordinate descent iteratively optimizes the value of the objective for each feature.
Algorithm 2 Coordinate Descent

while $w$ has not converged do
    pick a feature index $i$
    update $w_i$ to arg min$_{w_i} L(w)$
end while

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 you 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 $y$? Actually not always. 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. The reason for this is that 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 $w_i \leftarrow \arg\min_{w_i} L(w)$. 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).

The LASSO objective is $\|Xw - y\|_2^2 + \lambda \|w\|_1$. It will be convenient to separate the terms that depend on $w_i$ from those that don’t. Denoting $x_j$ as the $j$-th column of $X$, we have

$$L(w) = \|Xw - y\|_2^2 + \lambda \|w\|_1$$

$$= \|\sum_{j=1}^d w_j x_j - y\|_2^2 + \lambda |w_i| + \lambda \sum_{j \neq i} |w_j|$$

$$= \|w_i x_i + C^{(1)}\|_2^2 + \lambda |w_i| + C^{(2)}$$

where $C^{(1)} = \sum_{j \neq i} w_j x_j - y$ and $C^{(2)} = \lambda \sum_{j \neq i} |w_j|$. The objective can in turn be written as

$$L(w) = \lambda |w_i| + C^{(2)} + \sum_{j=1}^n (w_i x_j + C^{(1)}_j)^2$$

Suppose the optimal $w_i$ is strictly positive: $w_i > 0$. Setting the partial derivative of the objective
wrt \( w_i^* \) to 0, we obtain

\[
\frac{\partial L}{\partial w_i} = \lambda + \sum_{j=1}^n 2x_{j,i}(w_i^*x_{j,i} + C_j^{(1)}) = 0 \implies w_i^* = \frac{-\lambda - \sum_{j=1}^n 2x_{j,i}C_j^{(1)}}{\sum_{j=1}^n 2x_{j,i}^2}
\]

Denoting \( a = -\sum_{j=1}^n 2x_{j,i}C_j^{(1)} \) and \( b = \sum_{j=1}^n 2x_{j,i}^2 \), 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 \( w_i^* = 0 \).

For this to happen, \( -\frac{\lambda + a}{b} \leq 0 \) and \( \frac{\lambda + a}{b} \geq 0 \). Rearranging, we can see this is equivalent to \( |a| \leq \lambda \).

To summarize:

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

where

\[
a = -\sum_{j=1}^n 2x_{j,i}C_j^{(1)}, \quad b = \sum_{j=1}^n 2x_{j,i}^2
\]

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. This is not a gradient-descent based update — we have a closed-form solution for the optimum \( w_i \), given all the other weights are fixed constants. We can also 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 \( \lambda \), we increase the threshold for \( |a| \) to be set to 0, and our solution becomes more sparse.

Note that during the optimization, weights can be set to 0, but they can also be “reactivated” after they have been set to 0 in a previous iteration, since \( a \) is affected by factors other than \( w_i \).

### 6.6 Sparse Least-Squares

Suppose we want to solve the least squares objective, subject to a constraint that \( w \) is sparse. Mathematically this is expressed as

\[
\min_w \quad \|Xw - y\|_2^2 \\
\text{s.t.} \quad \|w\|_0 \leq k
\]
Note that the **L0-norm** of \( w \) is simply the number of non-zero elements in \( w \). This optimization problem aims to minimize the residual error between our prediction \( Xw \) and \( y \) while ensuring that the solution \( w \) is sparse. Solving this optimization problem is NP-hard, so we instead aim to find a computationally feasible alternative method that can approximate the optimal solution. **Matching pursuit** is a greedy algorithm that achieves this goal.

### Matching Pursuit Algorithm

Recall that in ordinary least squares, we minimize the residual error \( \| Xw - y \|_2^2 \) by projecting \( y \) onto the subspace spanned by the columns of \( X \), thereby obtaining a linear combination \( w^* \) of the columns of \( X \) that minimizes the residual error. Matching pursuit is a greedy algorithm that starts with a completely sparse solution (\( w = 0 \)) and iteratively “builds up” \( w \) until the sparsity constraint \( \| w \|_0 \leq k \) can no longer be met. The algorithm is as follows:

**Algorithm 3** Matching Pursuit

1. Initialize the residual \( r = y \)
2. Initialize the weights \( w = 0 \)
3. While \( \| w \|_0 < k \) do
   1. Find the index \( i \) for which the residual is minimized: \( i = \arg \max_j \frac{\langle r, x_j \rangle}{\| x_j \|} \)
   2. Update the \( i \)'th entry of the weight vector to \( w_i = \frac{\langle r, x_i \rangle}{\| x_i \|^2} \)
   3. Update the new residual value: \( r = y - Aw \)
4. End while

At each step of the algorithm, we pick the coordinate \( i \) such that \( x_i \) (the \( i \)-th column of \( X \) corresponding to feature \( i \), not datapoint \( i \)) minimizes the residual \( r \). This equates to finding the index \( i \) for which the length of the projection onto \( x_i \) is maximized:

\[
i = \arg \max_j \frac{\langle r, x_j \rangle}{\| x_j \|}
\]

Let’s see why this is true. When we project the residual \( r \) on the vector \( x_j \), the resulting projection has length \( \frac{\langle r, x_j \rangle}{\| x_j \|} \). The length of the new residual follows from pythagoras theorem:

\[
\| r_{\text{old}} \|^2 = \| r_{\text{new}} \|^2 + \left( \frac{\langle r, x_j \rangle}{\| x_j \|} \right)^2 \implies i = \arg \max_j \frac{\langle r, x_j \rangle}{\| x_j \|}
\]

We move \( w_i \) to the optimum projection value and repeat greedily at each iteration. While matching pursuit is not guaranteed to find the optimal \( w^* \), in practice it works well for most applications. Setting the number of iterations is typically determined through cross-validation.
Chapter 7

Decision Tree Learning

7.1 Decision Trees

A decision tree is a type of classifier which classifies things by repeatedly posing questions of the form, “Is feature $x_i$ greater than value $v$?” You can visualize such a thing as a tree, where each node consists of a split-feature, split-value pair $(x_i, v)$ and the leaves are possible classes. Here’s a picture:

Note that a class may appear multiple times at the leaves of the decision tree. Picking a split-feature, split-value pair essentially carves up the feature space into chunks, and you can imagine that having enough of these pairs lets you create an arbitrarily complex classifier that can perfectly overfit any training set (unless two training points of different classes coincide).

When we consider the task of building this tree, we want to ask ourselves a couple of things:

- How do we pick the split-feature, split-value pairs?
- How do we know when to stop growing the tree?

Let’s address the first point. Intuitively, when we pick a feature to split on, we want choose the one which maximizes how sure we are of the classes of the two resulting chunks. If we had a bunch
of data points and the barrier represented by the line $x_i = v$ perfectly split them such that all the instances of the positive class were on one side and all the instances of the negative class were on the other, then the split $(x_i, v)$ is a pretty good one.

Now, to concretely talk about the idea of “how sure” we are, we’ll use the ideas of surprise and entropy. The numerical surprise of observing that a random variable $X$ takes on value $k$ is:

$$\log \frac{1}{P(X = k)} = -\log P(X = k)$$

The entropy of a random variable $H(X)$ is:

$$H(X) = E[\text{surprise}] = E_X[\log \frac{1}{P(X = k)}] = \sum_k P(X = k) \log \frac{1}{P(X = k)}$$

Here’s a graph of entropy vs. $P(X = k)$ for a binary discrete random variable $X$ (as in, $k \in \{0, 1\}$):

Observe that (1) it’s convex and (2) it’s maximized when the probability distribution is even with respect to the outcomes. That is to say, a coin that is completely fair ($P(X = 0) = P(X = 1) = 0.5$) 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. In general, 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.

**An assumption**

By the way, it’s strange that entropy is defined for random variables when there are no random variables in a training set. To address this, we make the assumption that a random variable $X$ on the possible classes is representative of our training set. The distribution of $X$ is defined by our training points $(x_i, y_i)$, where $P(X = c_j) = \frac{n_j}{n}$ with $n_j$ being the number of training points whose $y_i$ is $c_j$.

So now we know that when we choose a split-feature, split-value pair, we want to reduce entropy in some way. First, let $Y_{f_i,v}$ be an indicator function that is 1 for a data point if its feature $f_i$ is less than $v$. Note that $Y$ is not a random variable, though we will use it in some notations that make
it appear like one. When we consider splitting one set of points represented by random variable \( X \) with split-key \((f_i, v)\), there are a few entropies we should consider.

- \( H(X) \)
- \( H(X|Y = 1) \). That is, the entropy of the set of points whose \( f_i \) is less than \( v \).
- \( H(X|Y = 0) \). That is, the entropy of the set of points whose \( f_i \) is not less than \( v \).

In some ways \( H(X) \) is non-negotiable: we start with the set of points represented by \( X \), and they have some entropy, and now we wish to carve up those points in a way to minimize the entropy remaining. Thus, the thing 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:

\[
\text{Minimize } P(X < v)H(X|Y = 1) + P(X \geq v)H(X|Y = 0)
\]

An equivalent way of seeing this is that we want to maximize the information we’ve learned, which represented by how much \( H(X) \) gets reduced after learning \( Y \):

\[
\text{Maximize } H(X) - \frac{P(Y = 1)H(X|Y = 1) + P(Y = 0)H(X|Y = 0)}{H(X|Y)}
\]

For general indicator functions \( Y \), we want to maximize something called the **mutual information** \( I(X; Y) \). This depends on the **conditional entropy** of \( X \) given \( Y \):

\[
H(X|Y = y) = \sum_k P(X = k|Y = y) \log \frac{1}{P(X = k|Y = y)}
\]

\[
H(X|Y) = \sum_y P(Y = y)H(X|Y = y)
\]

\[
I(X; Y) = H(X) - H(X|Y)
\]

Note that \( I(X; Y) \) is nonnegative and it’s zero only if 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. The preceding statement says, 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.

Now that we have addressed how to pick splits, let’s talk about when to stop growing the tree. We know that decision trees are powerful tools because they can be used to represent any arbitrarily complex decision boundaries. Thus, it is very easy to overfit them. One way to do this is to keep carving up the feature space until the leaves are entirely pure (as in, they only contain points of a single class). As a result, when training decision trees, we always want to keep in mind a couple of heuristics for stopping early:

- **Limited depth**: keep track of the depth of each split, and don’t go past some depth \( t \)
- **Information gain criteria**: stop carving once your splits don’t reward you with enough information (i.e., set a threshold for \( I(X; Y) \)
• Pruning: This isn’t a method for stopping early. Rather it’s the strategy of growing your tree out as far as you can, and then combining splits back together if doing so reduces your validation error.

All of these thresholds can be tuned with cross-validation.

7.2 Random Forests

Another way to combat overfitting is the use of random forests. A random forest is a set of decision trees whose outputs are averaged together in some way (usually majority vote) to produce the final answer to a classification problem. In addition, each tree of the forest will have some elements of randomness to them. Let’s think about why this randomness is necessary.

Imagine you took the same set of training points and you built $k$ decision trees on them with the method of choosing splits described in the previous section. What could you say about these $k$ trees? Answer: They would all be the exact same trees in terms of what splits are chosen! Why: because there is only one best split of a set of training points, and if we use the same algorithm on the same set of training points, we’ll get the same result.

There are two options: (1) change our algorithm for each tree, or (2) change our training points. The algorithm is pretty good, so let’s change the training points. In particular, we’ll use the method of bagging: for each tree, we sample with replacement some constant number of training points to “bag” and use for training. This will avoid the problem of having identical decision trees...

...Or will it? Imagine that you had an extremely predictive feature, like the appearance of the word “viagra” for classifying spam, for your classification task. Then, even if you took a random subsample of training points, your $k$ trees would still be very similar, most likely choosing to split on the same features. Thus, the randomness of our trees will come from:

• bagging: random subsample of training points per tree
• enforcing that a random subsample of features be used for each tree

Both the size of the random subsample of training points and the number of features per tree are hyperparameters intended to be tuned through cross-validation.

Remember why this is a good idea: One decision tree by itself is prone to overfitting to the training set. However, if we have a bunch of them that are diverse and uncorrelated, then we can be more sure that their average is closer to the true classifier.

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 majority 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. Roughly, we want to:

• Treat each of the weak classifiers (e.g., trees in a random forest) as “features”
• Hunt for better “features” based on the current performance of the overall classifier
The key idea is as follows: to improve our classifier, we should focus on finding learners that correctly classify the points which the overall boosted classifier is currently getting wrong. Boosting algorithms implement this idea by associating a weight with each training point and iteratively reweighting so that misclassified points have relatively high weights. Intuitively, some points are “harder” to classify than others, so the algorithm should focus its efforts on those.

**AdaBoost**

There are many flavors of boosting. We will discuss one of the most popular versions, known as AdaBoost (short for adaptive boosting). Its developers won the Gödel Prize for this work.

**Algorithm**

We present the algorithm first, then derive it later.

1. Initialize the weight of all $n$ training points to $\frac{1}{n}$.

2. Repeat for $m = 1, \ldots, M$:
   (a) Build a classifier $G_m$ with data weighted according to $w_i$.
   (b) Compute the weighted error $e_m = \frac{\sum_{i\text{ misclassified}} w_i}{\sum w_i}$.
   (c) Re-weight the training points as

   $$w_i \leftarrow w_i \cdot \begin{cases} \sqrt{\frac{1}{1-e_m}} & \text{if misclassified} \\ \sqrt{\frac{e_m}{1-e_m}} & \text{otherwise} \end{cases}$$

   (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 that our weak learners always produce an error $e_m < 1/2$. 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 weakness in that this aggressive reweighting can cause the classifier to focus too much on certain training examples – if the data is noisy with outliers, then this will weaken the boosting algorithm’s performance.

We have not yet discussed how to make a prediction given our classifiers (say, $G_1, \ldots, G_M$). One conceivable method is to use logistic regression with $G_m(x)$ 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 \( x \) by \( \text{sign}(\sum \alpha_m G_m(x)) \). Note that this choice of \( \alpha_m \) (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 by presenting a matching pursuit interpretation of AdaBoost.

**Derivation of AdaBoost**

Suppose we have computed classifiers \( G_1, \ldots, G_{m-1} \) along with their corresponding weights \( \alpha_k \) and we want to compute the next classifier \( G_m \) along with its weight \( \alpha_m \). The output of our model so far is \( F_{m-1}(x) := \sum_{i=1}^{m-1} \alpha_i G_i(x) \), and we want to minimize the risk:

\[
\arg \min_{\alpha_m, G_m} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \alpha_m G_m(x_i))
\]

for some suitable loss function \( L \). 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:

\[
L(y, h(x)) = e^{-y h(x)}
\]

This loss function is illustrated in Figure 7.1. We have exponential decay as we increase the input – thus if \( y h(x) \) is large and positive (so \( h(x) \) has the correct sign and high magnitude), our loss is decreasing exponentially. Conversely, if \( y h(x) \) is a large negative value, our loss is increasing exponentially, and thus we are heavily penalized for confidently making an incorrect prediction.

![Graph showing exponential loss](image)

**Figure 7.1:** The exponential loss provides exponentially increasing penalty for confident incorrect predictions. This figure is from Cornell CS4780 notes.

We can write the AdaBoost optimization problem with exponential loss as follows:

\[
\alpha_m, G_m = \arg \min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i(F_{m-1}(x_i) + \alpha G(x_i))}
\]
7.3. BOOSTING

\[
= \arg \min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i F_{m-1}(x_i)} e^{-y_i \alpha G(x_i)}
\]

The term \( w_i^{(m)} := e^{-y_i F_{m-1}(x)} \) 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:

\[
\alpha_m, G_m = \arg \min_{\alpha, G} \sum_{y_i = G(x)} w_i^{(m)} e^{-\alpha} + \sum_{y_i \neq G(x)} w_i^{(m)} e^{+\alpha} \quad (\ast)
\]

For a fixed value of \( \alpha \), the second term does not depend on \( G \). Thus we can see that the best choice of \( G_m(x) \) is the classifier that minimizes the error given the weights \( w_i^{(m)} \). Let

\[
e_m = \frac{\sum_{y_i \neq G_m(x)} w_i^{(m)}}{\sum_i w_i^{(m)}}
\]

Once we have obtained \( G_m \), we can solve for \( \alpha_m \): by dividing \( \ast \) by the constant \( \sum_{i=1}^{n} w_i^{(m)} \), we obtain

\[
\alpha_m = \arg \min_{\alpha} (1 - e_m) e^{-\alpha} + e_m e^{\alpha}
\]

which has the solution (left as exercise)

\[
\alpha_m = \frac{1}{2} \ln \left( \frac{1 - e_m}{e_m} \right)
\]

as claimed earlier.

From the optimal \( \alpha_m \), we can derive the weights:

\[
w_i^{(m+1)} = \exp \left( -y_i F_m(x_i) \right) = \exp \left( -y_i [F_{m-1}(x_i) + \alpha_m G_m(x_i)] \right) = w_i^{(m)} \exp \left( -y_i G_m(x_i) \alpha_m \right) = w_i^{(m)} \exp \left( -y_i G_m(x_i) \frac{1}{2} \ln \left( \frac{1 - e_m}{e_m} \right) \right) = w_i^{(m)} \exp \left( \ln \left( \frac{1 - e_m}{e_m} \right) - \frac{1}{2} y_i G_m(x_i) \right) = w_i^{(m)} \left( \frac{1 - e_m}{e_m} \right)^{-\frac{1}{2} y_i G_m(x_i)}
\]

Here we see that the multiplicative factor is \( \sqrt{\frac{e_m}{1 - e_m}} \) when \( y_i = G_m(x_i) \) 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 \( \alpha \) updates as pushing towards a solution in some direction until we can no longer improve our performance. More precisely, whenever we
compute \( \alpha_m \) (and thus \( w^{(m+1)} \)), for the incorrectly classified entries, we have

\[
\sum_{y_i \neq G_m(x_i)} w_i^{(m+1)} = \sum_{y_i \neq G_m(x_i)} w_i^{(m)} \sqrt{\frac{1 - e_m}{e_m}}
\]

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

\[
\frac{\sum_{y_i = G_m(x_i)} w_i^{(m+1)}}{\sum_{i=1}^{n} w_i^{(m)}} = (1 - e_m) \sqrt{\frac{e_m}{1 - e_m}} = \sqrt{e_m(1 - e_m)}
\]

Thus these two quantities are the same once we have adjusted our \( \alpha \), 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 AdaBoost 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.
Chapter 8

Deep Learning

8.1 Convolutional Neural Nets

A convolutional neural net is just like a regular neural net, except more complicated (convoluted). But in some ways, a convolutional neural net can actually be a simpler model than a fully-connected neural net.

We'll be talking about convolutional neural networks (ConvNets) mostly in the framework of image classification. First, let's recall what one layer of a regular neural net looks like:

Remember that each layer consists of units which represent a single number. The values on the units of some layer are determined by the values of the units behind them and the weights on the edges in between the layers. More specifically, the vector of numbers representing layer 2, say $y$, can be represented by linear combinations $Wx$ of the vector representing layer 1. If we were dealing with images, each unit of the input layer might be the intensity of a single pixel of the image.

The fully-connected architecture of units has a lot of weights to learn. For just a small $32 \times 32$ image, there would be 1024 input units and at least as many weights just between the first two layers. Now, let's look at how a ConvNet which deals with images uses its weights differently.
Notice that some of the layers are called “conv” and “pool.” These layers are the new ideas that ConvNets introduce to the neural net architecture.

**Convolutional Layers**

A **convolutional layer** is determined by **convolving** a **kernel** about the previous layer. The kernel is just a fancy name for an array of weights, and convolving means that we slide the array of weights across the pixels of the previous layer and compute the sum of the elementwise products (kind of like a 2D dot-product). Here is a picture that illustrates this:

In the above picture, we extracted 6 activation values from 12 input values (we would supposedly pass the dot-products through some kind of nonlinearity as well). In a regular fully-connected neural net, we would have used $6 \times 12 = 72$ weights to accomplish this. However, in this convolutional layer, we only used 4 weights. This is because we made use of **weight sharing**, as in:

1. the weights $w, x, y, z$ were shared among all the pixels of the input
2. the individual units of the output layer were all determined by the same weights $w, x, y, z$

Compare this to the fully-connected architecture where for each output unit, there is a separate weight to learn for each input unit. This kind of strategy decreases the complexity of our model (there are fewer weights), but it makes sense for image processing because there are lots of repeated
patterns in images, and if you have one kernel which can detect some kind of phenomenon, then it would make sense to use it elsewhere in the image.

How do kernels detect things, anyways? The short answer is: they will produce large values in the areas of the image which appear most similar to them. Consider a simple kernel $[1 \ -1]$. This kernel will have produce large values for which the left pixel is bright and the right pixel is dark. Conversely, it will produce small values for which the left pixel is dark and the right pixel is bright.

![Figure 8.1: Left: a sample image. Right: the output of convolving the $[1 \ -1]$ kernel about the image.](image)

Notice how the output image has high values (white) in the areas where the original image turned from bright to dark (like the right hand side of the figure), and it has low values (black) in the areas where the original image turned from dark to bright (like the left hand side of the figure). This kernel can be thought of as a simple edge detector! As another example, consider the kernel:

$$
\begin{bmatrix}
0.6 & 0.2 & 0 \\
0.2 & 0 & 0.2 \\
0 & 0.2 & 0.6
\end{bmatrix}
$$

If this was convolved about an image, it would detect edges at a positive 45-degree angle.

Just a few more things on convolutional layers:

1. You can stack the outputs of multiple kernels together to form a convolutional layer.

![Figure 8.2: Here, we slid 6 separate $5 \times 5 \times 3$ kernels across the original image to produce 6 activation maps in the next convolutional layer. Each activation map can also be referred to as a channel.](image)

2. To save memory, you can have your kernel stride across the image by multiple pixels.
3. Zero-padding is sometimes used to control the exact dimensions of the convolutional layer.

4. As you add more convolutional layers, the effective receptive field of each successive layer increases. That is to say, as you 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 you use two successive layers of $3 \times 3$ kernels, 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.

5. You can think of convolutional layers as a complexity-regularized version of fully-connected layers.

### Pooling Layers

A pooling layer does not involve more weights. Instead, it is a layer whose purpose is solely to downsample AKA pool AKA gather AKA consolidate the previous layer. It does this by sliding a window of some size across the units of a layer of the ConvNet and choosing (somehow) one value to effectively “represent” all the units captured by the window. You can tweak the nature of a pooling layer in a few orthogonal ways.

1. **How to pool?** In max-pooling, the representative value just becomes the largest of all the units in the window. In average-pooling, the representative value is the average of all the units in the window.
2. *In which direction to pool?*

(a) Spatial pooling pools values within the same channel. This has the capability of introducing translational invariance to your model.

![Spatial Pooling Diagram](image)

Figure 8.4: Here, the input layer of the right image is a translated version of the input layer of the left image, but because of max-pooling, the next layer looks more or less the same.

(b) Cross-channel pooling pools values across different channels. This has the capability of introducing transformational invariance to your model.

![Cross-Channel Pooling Diagram](image)

Figure 8.5: Here, we have an example where our convolutional layer is represented by 3 kernels. Suppose they were each good for detecting the number 5 in some degree of rotation. If we pooled across the three channels determined by these kernels, then no matter what orientation of the number “5” we got as input to our ConvNet, the pooling layer would have a large response!

3. *“Lossiness” of pooling.* This is determined by the stride of the pooling window. If you stride by a large amount, you potentially lose more information, but you conserve memory.

If you now look back at the picture of the ConvNet near the beginning of this note, you should have a better idea of what each layer is doing. The ConvNet in that picture is Yann Lecun’s *LeNet*, which is used to classify handwritten alphanumeric characters!

**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.

**AlexNet (Krizhevsky et al, 2012)**

Key characteristics:

- Conv filters of varying sizes - for example, the first layer has $11 \times 11$ 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 8.6 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.

\(^1\)VGG stands for the “Visual Geometry Group” at Oxford where this was developed.
**8.1. CONVOLUTIONAL NEURAL NETS**

GoogLeNet (Szegedy et al, 2014)

Also codenamed as “Inception.”\(^2\) Published in CVPR 2015 as “Going Deeper with Convolutions.”

Key characteristics:

- 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 \(1 \times 1\) 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)

![Building block for the ResNet](image)

Figure 8.8: Building block for the ResNet from “Deep Residual Learning for Image Recognition,” CVPR 2016. If the desired function to be learned is \(H(x)\), we instead learn the residual \(F(x) := H(x) - x\), so the output of the network is \(F(x) + x = H(x)\).

Key characteristics:

\(^2\)“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.
CHAPTER 8. DEEP LEARNING

• Very deep (152 layers). Residual blocks (Figure 8.8) are stacked together - each individual weight layer in the residual block is implemented as a $3 \times 3$ 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.

Towards an Understanding of Convolutional Nets

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?)

See Stella’s slides for images of these techniques in practice.