A Comprehensive Guide to Machine Learning

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

January 17, 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

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

Other notes:

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

Contents

Chapter 1

Regression, Validation

1.1 Machine Learning Principles

Have you ever tailored your exam preparation, to optimize exam performance? It seems likely that you would, and in fact, those preparations may range from "taking one practice exam" to "wearing lucky underwear". The question were implicitly answering is: which of these actually betters exam performance? This is where machine learning comes in. In machine learning, we strive to find patterns in data, then make predictions using those patterns, just as the average student would do to better exam scores. We start by structuring an approach to understanding machine learning.

Levels of Abstraction

We will take an example from astronomy, below. How exactly do we define the parameters of the problem, and how do we leverage the machine learning framework to help with this problem? See below, for a concrete outline of what to think about.

1. **Applications and Data**: What are you trying to do? What is your data like? Here, identify your problem and the nature of your observations. Your data may be Cartesian coordinates, an matrix of RGB values etc.

   Example: We have (x_1, y_1) coordinates. We want to compute each planet's orbit.

2. **Model**: What kind of pattern do you want to find? This could be a polynomial, a geometric figure, a set of dynamics governing a self-feedback loop etc.

   Example: An ellipse for an orbit.

3. **Optimization Problem**: Whatever problem you have, turn it into an optimization problem. We could minimize losses or maximize gains, but in either case, define an objective function that formally specifies what you care about.

   Example: $\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2^2$

4. **Optimization Algorithm**: How do we minimize? Determine how exactly to solve the optimization problem that you've now proposed.

   Example: Solve $\mathbf{Cx} = \mathbf{d}$*, a system of linear equations.*

In this course, we will study both **models** and **optimization problems** extensively. Being able to compartmentalize each of these will make it easier for you to frame and understand machine learning.

1.2 Ordinary Least Squares

Let us now use the framework specified above to discuss ordinary least squares, a canonical optimization problem that we'll study in-depth from various perspectives. We'll then slowly add complexity to least squares, with a number of different techniques. We're most familiar with the 2-dimensional case, commonly called "linear regression". So, let's start there.

1. **Applications and Data**: We have n data points (a_i, b_i) and wish to determine how the a_i's determine the b_i's.

2. **Model**: With least squares, we assume a_i, b_i are linearly related. More concretely, we wish to find a model m, c such that $b_i \approx ma_i + c$ where $b_i, a_i, m, c \in \mathbb{R}$. In other words, we want to learn how the data determines m, c.

3. **Optimization Problem**: To solve this problem, we turn it into an optimization problem:

$$\min_{m,c} \|(m\mathbf{a} + c\mathbb{1}) - \mathbf{b}\|_2^2$$

4. **Optimization Algorithm**: As it turns out, ordinary least squares has a closed-form solution for the optimal value of $\mathbf{x}$. We can solve for this a number of ways, and below, we'll see perspectives that each of these methods belies.

What if our samples are vectors, instead? Our model is then a matrix $\mathbf{X}$, and we'll need to be more careful about our formulation of the least squares framework.

1. **Applications and Data**: We have n data points $(\mathbf{a}_i, \mathbf{b}_i)$ and wish to determine how the $\mathbf{a}_i$'s determine the $\mathbf{b}_i$'s.

2. **Model**: Assume $\mathbf{a}_i, \mathbf{b}_i$ are linearly related. Find $\mathbf{X}$ such that $\mathbf{b}_i \approx \mathbf{X}\mathbf{a}_i$ where $\mathbf{b_i} \in \mathbb{R}^m, \mathbf{a_i} \in \mathbb{R}^l, \mathbf{X} \in \mathbb{R}^{m \times l}$.

3. **Optimization Problem**: To solve this problem, we turn it into an optimization problem. We rewrite in the least squares framework by flattening X (i.e., stacking each row vector of X on top of each other) and using the $\mathbf{a}_i$'s to form diagonal block matrices as follows, to give us:

$$\min_{\mathbf{x}} \| \begin{bmatrix} \mathbf{a}_1^T & 0 & 0 & 0 \\ 0 & \mathbf{a}_1^T & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \mathbf{a}_1^T \\ \mathbf{a}_2^T & 0 & 0 & 0 \\ 0 & \mathbf{a}_2^T & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \mathbf{a}_2^T \\ \mathbf{a}_3^T & 0 & 0 & 0 \\ 0 & \mathbf{a}_3^T & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \mathbf{a}_3^T \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{a}_n^T & 0 & 0 & 0 \\ 0 & \mathbf{a}_n^T & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \mathbf{a}_n^T \end{bmatrix} \begin{bmatrix} \\ -- \\ \\ \\ -- \\ \\ \\ -- \\ \\ \\ -- \\ \vdots \\ -- \\ \\ \\ \end{bmatrix} - \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \\ \vdots \\ \\ \mathbf{b}_n \end{bmatrix} \|^2$$

Using the notation $M_{m\times n}$ to denote a matrix with m rows and n columns (an m by n matrix), we see that the matrices and vectors above have the following dimensions:

$$\mathbf{A}_{mn\times ml}\mathbf{x}_{ml\times 1} = \mathbf{b}_{mn\times 1}$$

4. **Optimization Algorithm**: Again, we cover this in the next section.

Vector Calculus Approach

We consider the brute-force approach to deriving the solution, first. Just as you would find optimum for scalar-valued functions, take the derivative, set it equal to 0, and solve. Before we begin, keep in mind the following convention.

$$\frac{\partial \mathbf{w}^T\mathbf{x}}{\partial \mathbf{x}} = \mathbf{w}, \frac{\partial \mathbf{x}^T\mathbf{A}\mathbf{x}}{\partial \mathbf{x}} = (\mathbf{A} + \mathbf{A}^T)\mathbf{x}$$

Using these conventions, we can now take the gradient of our objective function. Apply the definition of the L2-norm to start, that $\|\mathbf{x}\|_2^2 = \mathbf{x}^T\mathbf{x}$.

$$\begin{aligned}
&\frac{\partial}{\partial \mathbf{x}}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 \\
&= \frac{\partial}{\partial \mathbf{x}}((\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{A}\mathbf{x} - \mathbf{b})) \\
&= \frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - (\mathbf{A}\mathbf{x})^T\mathbf{b} - \mathbf{b}^T(\mathbf{A}\mathbf{x}) + \mathbf{b}^T\mathbf{b}) \\
&= \frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{x}^T\mathbf{A}^T\mathbf{b} + \mathbf{b}^T\mathbf{b}) \qquad \text{recall } \mathbf{b}^T\mathbf{a} = \mathbf{a}^T\mathbf{b} \text{ for } \mathbf{a}, \mathbf{b} \in \mathbb{R}^d \\
&= 2\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{A}^T\mathbf{b}
\end{aligned}$$

Now, set the gradient to $\mathbf{0}$, and solve for $\mathbf{x}^*$.

$$\begin{aligned} 2\mathbf{A}^T\mathbf{A}\mathbf{x} &= 2\mathbf{A}^T\mathbf{b} \\ \mathbf{A}^T\mathbf{A}\mathbf{x} &= \mathbf{A}^T\mathbf{b} \\ \mathbf{x}^* &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \end{aligned}$$

Finally, we have our closed form solution.

Projection Approach

We want to find something, denoted $\hat{\mathbf{x}}$, spanned by the columns of $\mathbf{A}$, that is closest to $\mathbf{b}$. As it turns out, this is precisely $\mathbf{b}$ projected onto the column space of A. Then, the error $\mathbf{b} - \mathbf{A}\mathbf{x}$ is perpendicular to everything spanned by the columns of A. This gives us the following formulation:

$$\begin{aligned} \mathbf{a}_i^T(\mathbf{b} - \mathbf{A}\mathbf{x}) &= 0, \quad \forall \mathbf{x} \\ \mathbf{A}^T(\mathbf{b} - \mathbf{A}\mathbf{x}) &= \mathbf{0} \\ \mathbf{A}^T\mathbf{b} &= \mathbf{A}^T\mathbf{A}\mathbf{x} \\ \mathbf{x}^* &= (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \end{aligned}$$

1.3 Feature Engineering

We've seen that the least-squares optimization problem

$$\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

represents the "best-fit" *linear* model, by projecting $\mathbf{b}$ onto the subspace spanned by the columns of A. However, sometimes we would like to fit not just linear models, but also *nonlinear* models such as ellipses and cubic functions. We can still do this under the framework of least-squares, by augmenting (in addition to the raw features) new arbitrary features to the data. Note that the resulting models are still linear w.r.t the augmented features, but they are nonlinear w.r.t the original, raw features.

Example: Fitting Ellipses

Let's use least-squares to fit a model for data points that come from an ellipse.

1. **Applications and Data**: We have n data points (x_i, y_i), which may be noisy (could be off the actual orbit). Our goal is to determine how the x_i's determine the y_i's.
2. **Model**: Model ellipse in the form $a_0 + a_1x^2 + a_2y^2 + a_3xy + a_4x + a_5y = 1$. There are 6 unknown coefficients (or 'weights').

3. **Optimization Problem**: We formulate the problem with least-squares as before:

$$\min_{\mathbf{a}} \|\mathbf{Aa} - \mathbf{b}\|_2^2$$

Since every single data point has the same prediction value of 1, we can represent $\mathbf{b} \in \mathbb{R}^n$ as all 1's. Each coefficient a_i can be interpreted as the weight of the i'th feature. In expanded form, the optimization problem can be formulated as the following:

$$\min_{a_0,a_1,a_2,a_3,a_4,a_5} \| \begin{bmatrix} 1 & x_1^2 & y_1^2 & x_1y_1 & x_1 & y_1 \\ 1 & x_2^2 & y_2^2 & x_2y_2 & x_2 & y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n^2 & y_n^2 & x_ny_n & x_n & y_n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_5 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \|^2$$

Specifically:

- Each column represents a feature
- Each row represents a data point
- A **feature** is a building block of a model
- Assuming the columns are linearly independent, we can fit the data with OLS
- Augmenting more features in addition to the raw features makes the columns less and less linearly independent, making OLS more and more infeasible

Model Notation

A model has the following form:

$$y = \sum_{i=1}^{P} \alpha_i \Phi_i(\mathbf{x})$$

- y is the prediction from the model (note that y is a linear combination of the features)
- Φ_i represents the i'th feature - which depends on the raw features of the data point x
- α_i is the fixed coefficient/weight corresponding to the i'th feature

Polynomial Features

There are many different kinds of features, and an important class is **polynomial features**. Polynomial features can be interpreted as polynomials of the raw features.

Remember that polynomials are merely linear combination of monomial basis terms. Monomials can be classified in two ways, by their degree and dimension:

degree $\rightarrow$	0	1	2	3	$\cdots$
$\downarrow$ **dimension**					
univariate	1	x	x^2	x^3	$\cdots$
bivariate	1	x_1, x_2	x_1^2, x_2^2, x_1x_2	$x_1^3, x_2^3, x_1^2x_2, x_1x_2^2$	$\cdots$

Why are polynomial features so important? Note that under **Taylor' Theorem**, any sufficiently smooth function can be approximated arbitrarily closely by a high enough degree polynomial. This

is true for both univariate polynomials and multivariate polynomials, giving them the title of **universal approximators**. In this sense, polynomial features are building blocks for many kinds of functions.

One downside of polynomials is that as their degree increases, they increase exponentially in the number of terms. That is, for a polynomial of degree at most d in l dimensional space,

$$\#\text{ of terms} = O(l^d)$$

The justification is as follows: there are d "slots" (corresponding to the degree), and l "choices" for each slot (corresponding to which of the l original "raw" features will be used).

Due to their exponential number of terms, polynomial features suffer from the curse of dimensionality. Note however that we can bypass this issue by employing kernels, allowing us to even deal with infinite-diemsional polynomials! More on this later in the course.

Types of Error

One question when using polynomial features is how do we choose the right degree for the polynomial? In order to answer this question we would idealy measure the prediction error of our model for different choices of the degree and choose the degree that minimizes the error.

Here we must make a distinction between training error and true error. **Training error** is the prediction error when the model is evaluated on the training data, in other words the same data that was used to train the model is being used to evaluate it. **True error** is the prediction error when the model is evaluated on unseen data that comes from the true underlying model.

For different choices of the degree, let's observe what happens to the training error versus the true error. We observe that as degree increases, both training error and true error decrease initially due to using a more complex model. Training error continues to always decrease as the degree increases. This is true, because as the degree increases, the polynomial becomes strictly more expressive, fitting the training data more accurately. However, the true error eventually increases as the degree increases. This indicates the presence of **overfitting**: the model is too complex, and it is fitting the noise too much, leading to a low training error but high true error. Overfitting highlights that getting a better fit for your training data does not mean that you've learned the correct model.

1.4 Ridge Regression

Motivation

While OLS can be used for solving linear least squares problems, it falls short for two reasons: numerical instabilty and overfitting.

Numerical Instability: Numerical instability arises when the features of the data are close to collinear (leading to linearly dependent feature columns), causing the feature matrix A to have singular values that are either 0 or very close to 0. Why are small singular values bad? Let us illustrate this via the singular value decomposition (SVD) of A:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

Now let's try to expand the $(\mathbf{A}^T\mathbf{A})^{-1}$ term in OLS using the SVD of A:

$$(\mathbf{A}^T\mathbf{A})^{-1} = (\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^{-1} = (\mathbf{V}\mathbf{\Sigma}(\mathbf{I})\mathbf{\Sigma}\mathbf{V}^T)^{-1} = (\mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T)^{-1} = (\mathbf{V}^T)^{-1}(\mathbf{\Sigma}^2)^{-1}\mathbf{V}^{-1} = \mathbf{V}\mathbf{\Sigma}^{-2}\mathbf{V}^T$$

This means that $(\mathbf{A}^T\mathbf{A})^{-1}$ will have singular values that are the squared inverse of the singular values of $\mathbf{A}$, leading to either extremely large or infinite singular values (when the singular value of $\mathbf{A}$ is 0). Such excessively large singular values can be very problematic for numerical stability purposes.

Overfitting: Overfitting arises when the features of the A matrix are too complex and prevent OLS from producing a model that generalizes to unseen data. We want to be able to still keep these complex features, but somehow penalize them in our objective function so that we do not overfit.

Solution

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

$$\min_{\mathbf{x}}(\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 + \lambda^2\|\mathbf{x}\|^2)$$

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

$$\begin{aligned}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 + \lambda^2\|\mathbf{x}\|^2 &= (\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{A}\mathbf{x} - \mathbf{b}) + \lambda^2\mathbf{x}^T\mathbf{x} \\ &= \mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{A}^T\mathbf{b} + \mathbf{b}^T\mathbf{b} + \lambda^2\mathbf{x}^T\mathbf{x}\end{aligned}$$

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

$$\begin{aligned}\mathbf{0} &= 2\mathbf{x}(\mathbf{A}^T\mathbf{A} + \lambda^2\mathbf{I}) - 2\mathbf{b}^T\mathbf{A} \\ \mathbf{x} &= \mathbf{b}^T\mathbf{A}(\mathbf{A}^T\mathbf{A} + \lambda^2\mathbf{I})^{-1} \\ \mathbf{x} &= (\mathbf{A}^T\mathbf{A} + \lambda^2\mathbf{I})^{-1}\mathbf{A}^T\mathbf{b}\end{aligned}$$

The technique we just described is known as **ridge regression**, aka **Tikhonov regularization**. Note that now, the SVD of $(\mathbf{A}^T\mathbf{A} + \lambda^2\mathbf{I})^{-1}$ becomes:

$$(\mathbf{A}^T\mathbf{A} + \lambda^2\mathbf{I})^{-1} = (\mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T + \lambda^2\mathbf{I})^{-1} = (\mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T + \mathbf{V}\lambda^2\mathbf{I}\mathbf{V}^T)^{-1} = (\mathbf{V}(\mathbf{\Sigma}^2 + \lambda^2\mathbf{I})\mathbf{V}^T)^{-1}$$

Now with our slight tweak, the singular values have become $1/(\sigma^2 + \lambda^2)$, meaning that even if $\sigma = 0$, the singular values are guaranteed to be at least $1/\lambda^2$, solving our numerical instability issues. Furthermore, we have partially solved the overfitting issue. By penalizing the norm of $\mathbf{x}$, we encourage the weights corresponding to relevant features that capture the main structure of the true model, and penalized the weights corresponding to complex features that only serve to fine tune the model and fit noise in the data.

1.5 Hyperparameters and Validation

Recall the supervised regression setting in which we attempt to learn a mapping $f : \mathbb{R}^d \to \mathbb{R}$ from labeled examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where $y_i \approx f(\mathbf{x}_i)$. The typical strategy in regression is to pick some

parametric model and then fit the parameters of the model to the data. Here we consider a model of the form

$$f_{\boldsymbol{\alpha}}(\mathbf{x}) = \overbrace{\sum_{j=1}^{p}}^{\text{model order}} \underbrace{\alpha_j}_{\text{weights}} \underbrace{\phi_j(\mathbf{x})}_{\text{features}}$$

The functions ϕ_j compute **features** of the input $\mathbf{x}$. They are sometimes called **basis functions** because our model $f_{\boldsymbol{\alpha}}$ is a linear combination of them. In the simplest case, we could just use the components of $\mathbf{x}$ as features (i.e. $\phi_j(\mathbf{x}) = x_j$) but in future lectures it will sometimes be helpful to disambiguate the features of an example from the example itself, so we encourage this way of thinking now.

Once we've fixed a model, we can the optimal value of $\boldsymbol{\alpha}$ by minimizing some cost function L that measures how poorly the model fits the data:

$$\min_{\boldsymbol{\alpha}} L(\{\mathbf{x}_i, y_i\}_{i=1}^n, \boldsymbol{\alpha}) \overset{e.g.}{=} \min_{\boldsymbol{\alpha}} \sum_{i=1}^{n} (f_{\boldsymbol{\alpha}}(\mathbf{x}) - y_i)^2$$

Observe that the model order p is not one of the decision variables being optimized in the minimization above. For this reason p is called a **hyperparameter**. We might say more specifically that it is a **model hyperparameter**, since it determines the structure of the model.

Let's consider another example. **Ridge regression** is like ordinary least squares except that we add an ℓ^2 penalty on the parameters $\boldsymbol{\alpha}$:

$$\min_{\boldsymbol{\alpha}} \|A\boldsymbol{\alpha} - \mathbf{b}\|^2 + \lambda \|\boldsymbol{\alpha}\|_2^2$$

The solution to this optimization problem is obtained by solving the linear system

$$(A^T A + \lambda I)\boldsymbol{\alpha} = A^T \mathbf{b}$$

The regularization weight λ is also a hyperparameter, as it is fixed during the minimization above. However λ, unlike the previously discussed hyperparameter p, 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.

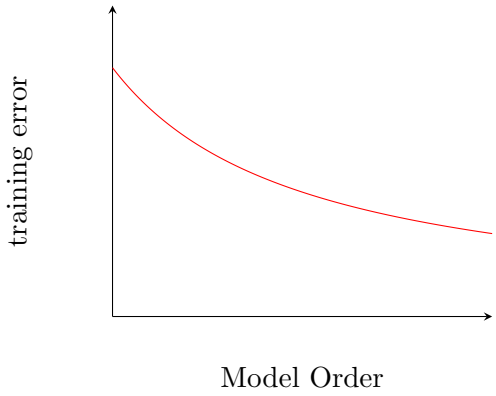
Types of Error

Let us define the **mean-squared training error** (or just **training error**) as

$$\widehat{\text{MSE}}_{\text{train}}(\boldsymbol{\alpha}) = \frac{1}{n} \sum_{i=1}^{n} (f_{\boldsymbol{\alpha}}(\mathbf{x}_i) - y_i)^2$$

This is a measure of how well the model fits the data.

Note that as we add more features, training error can only decrease. This is because the optimizer can set $\alpha_j = 0$ if feature j cannot be used to reduce training error.

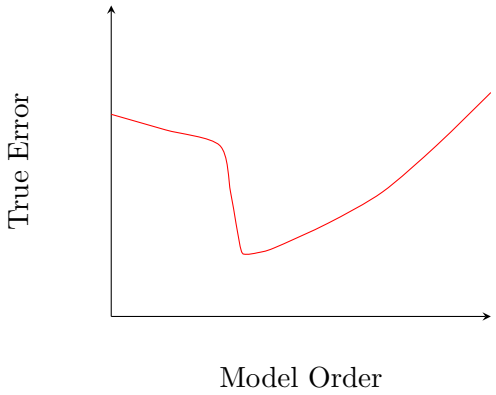


But training error is not exactly the kind of error we care about. In particular, training error only reflects the data in the training set, whereas we really want the model to perform well on new points sampled from the same data distribution as the training data. With this motivation we define the true error as

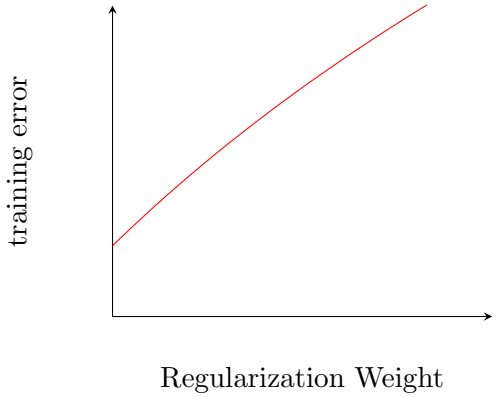
$$\text{MSE}(\boldsymbol{\alpha}) = \mathbb{E}[(f_{\boldsymbol{\alpha}}(\mathbf{x}) - y)^2]$$

where the expectation is taken over all pairs $(\mathbf{x}, y)$ from the data distribution. We generally cannot compute the true error because we do not have access to the data distribution, but we will see later that we can estimate it.

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

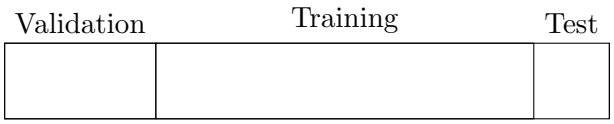


Validation

We've given some examples of hyperparameters and mentioned that they are not determined by the data-fitting optimization procedure. How, then, should we choose the values of p and λ?

Validation Error

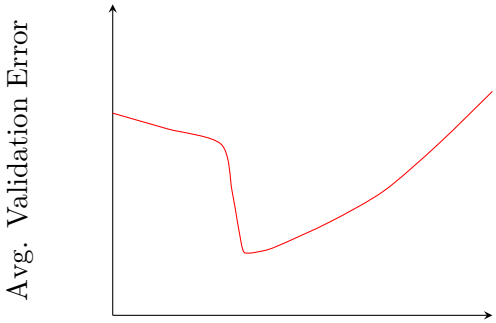
The key idea is to set aside some portion (say 30%) of the training data, called the **validation set**, which is *not* allowed to be used when fitting the model:



We can use this validation set to estimate the true error, as in

$$\widehat{\text{MSE}}_{\text{validate}}(\boldsymbol{\alpha}) = \frac{1}{m}\sum_{k=1}^{m}(f_\lambda(\mathbf{x}_{v,k}) - y_{v,k})^2 \approx \mathbb{E}[(f_\lambda(\mathbf{x}) - y)^2]$$

where f_λ is the model obtained by solving the regularized problem with value λ. The validation error tracks the true error reasonably well as long as m is large:



Note that this approximation only works because the validation set is disjoint from the training set. If we trained and evaluated the model using the same data points, the estimate would be very biased, since the model has been constructed specifically to perform well on those points.

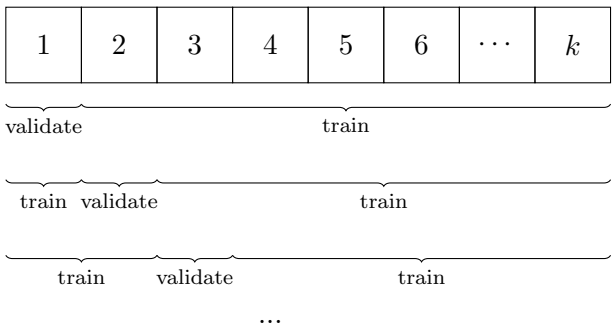
Now that we can estimate the true error, we have a simple method for choosing hyperparameter values: simply try a bunch of configurations of the hyperparameters and choose the one that yields the lowest validation error.

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.



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

Note 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.

1.6 Bias-Variance Tradeoff

Recall from our previous note that for a fixed input x, our measurement Y is a noisy measurement of the true underlying response $f(x)$:

$$Y = f(x) + N$$

Where N is a zero-mean random variable, and is typically represented as a Gaussian distribution. Our goal in regression is to recover the underlying model $f(.)$ as closely as possible. We previously mentioned MLE and MAP as two techniques that try to find of reasonable approximation to $f(.)$

by solving a probabilistic objective. 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 D that consisted of n (x_i, y_i) pairs. In that context, we treated the x_i's in our dataset D as *fixed* values. In this case however, we treat the x_i's as **random variables**. Specifically:

$$D = \{X_1, Y_1; X_2, Y_2; \ldots; X_n, Y_n\}$$

D is a random variable, because it consists of random variables X_i and Y_i. (Notice that because these quantities are random variables, they are all represented by uppercase letters.) Our hypothesis model $h(.)$ is also a random variable – $h(x, D)$ depends on some arbitrary test input x, as well as the random variable D that was used to train h. Since D is random, we will have a slightly different hypothesis model $h(., D)$ everytime we use a new dataset.

Metric

Our objective is to, for a fixed value of x, evaluate how closely the hypothesis can estimate the *noisy* observation 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 D as uppercase letters because we are treating them as random variables. Y and 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 D. In a way, we can think of D as the training data, and (x, Y) as the test data – in fact, x is often not even in the training set D! Mathematically, we represent our metric as the expected squared error between the hypothesis and the observation $Y = f(x) + N$:

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

The expectation here is over two random variables, D and Y:

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

Note that the error is w.r.t the observation and not the true underlying model, 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:

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

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

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

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

Let's use these facts to decompose the error:

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

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 bias 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: $\mathbb{E}_X[\varepsilon(x;h)]$.

Alternative Decomposition

The previous derivation is short, but may seem somewhat hacky and arbitrary. Here is an alternative derivation that is longer, but more intuitive. At its core, it uses the technique that $\mathbb{E}\Big[\Big(Z - Y\Big)^2\Big] = \mathbb{E}\Big[\Big((Z - \mathbb{E}[Z]) + (\mathbb{E}[Z] - Y)\Big)^2\Big]$ which decomposes to easily give us the variance

of Z and other terms.

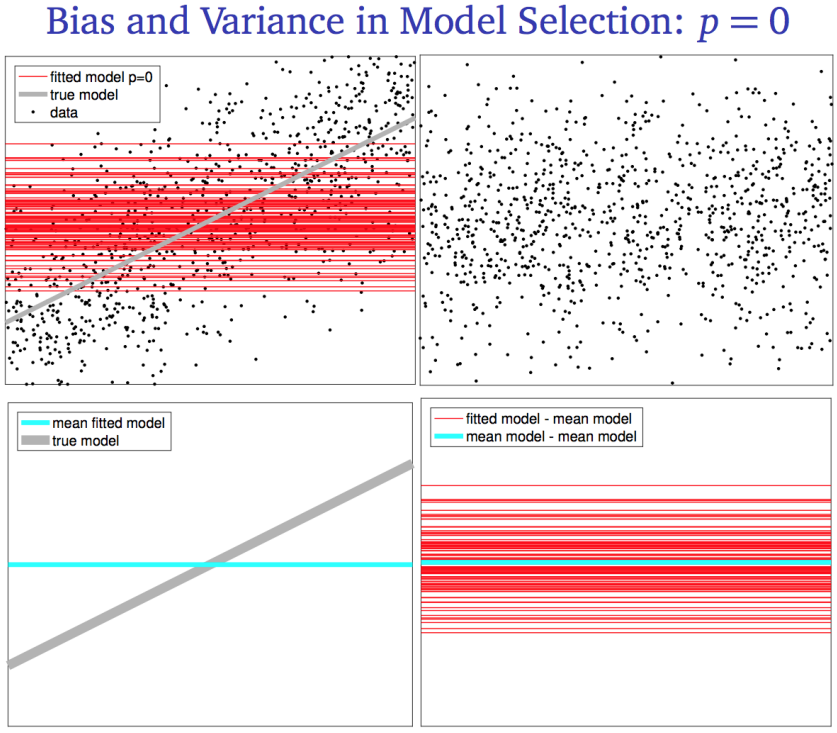
$$
\begin{aligned}
&\varepsilon(x;h) = \mathbb{E}[(h(x;D) - Y)^2] = \mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)] + \mathbb{E}[h(x;D)] - Y\Big)^2\Big] \\
&= \mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)]\Big)^2\Big] + \mathbb{E}\Big[\Big(\mathbb{E}[h(x;D)] - Y\Big)^2\Big] + 2\mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)]\Big) \cdot \Big(\mathbb{E}[h(x;D)] - Y\Big)\Big] \\
&= \mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)]\Big)^2\Big] + \mathbb{E}\Big[\Big(\mathbb{E}[h(x;D)] - Y\Big)^2\Big] + 2\mathbb{E}[h(x;D) - \mathbb{E}[h(x;D)]] \cdot \mathbb{E}[\mathbb{E}[h(x;D)] - Y] \\
&= \mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)]\Big)^2\Big] + \mathbb{E}\Big[\Big(\mathbb{E}[h(x;D)] - Y\Big)^2\Big] + 2\Big(\mathbb{E}[h(x;D)] - \mathbb{E}[h(x;D)]\Big) \cdot \mathbb{E}[\mathbb{E}[h(x;D)] - Y] \\
&= \mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)]\Big)^2\Big] + \mathbb{E}\Big[\Big(\mathbb{E}[h(x;D)] - Y\Big)^2\Big] + 2 \cdot (0) \cdot \mathbb{E}[\mathbb{E}[h(x;D)] - Y] \\
&= \mathbb{E}\Big[\Big(h(x;D) - \mathbb{E}[h(x;D)]\Big)^2\Big] + \mathbb{E}\Big[\Big(\mathbb{E}[h(x;D)] - Y\Big)^2\Big] \\
&= \mathrm{Var}((h(x;D)) + \mathbb{E}\Big[\Big(\mathbb{E}[h(x;D)] - Y\Big)^2\Big] \\
&= \mathrm{Var}((h(x;D)) + \Big(\mathrm{Var}\Big(\mathbb{E}[h(x;D)] - Y\Big) + \mathbb{E}\Big[\mathbb{E}[h(x;D)] - Y\Big]^2\Big) \\
&= \mathrm{Var}((h(x;D)) + \mathrm{Var}(Y) + \Big(\mathbb{E}[h(x;D)] - \mathbb{E}[Y]\Big)^2 \\
&= \mathrm{Var}((h(x;D)) + \mathrm{Var}(N) + \Big(\mathbb{E}[h(x;D)] - f(x)\Big)^2 \\
&= \underbrace{\Big(\mathbb{E}[h(x;D)] - f(x)\Big)^2}_{bias^2 \text{ of method}} + \underbrace{\mathrm{Var}(h(x;D))}_{\text{variance of method}} + \underbrace{\mathrm{Var}(N)}_{\text{irreducible error}}
\end{aligned}
$$

Experiments

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

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

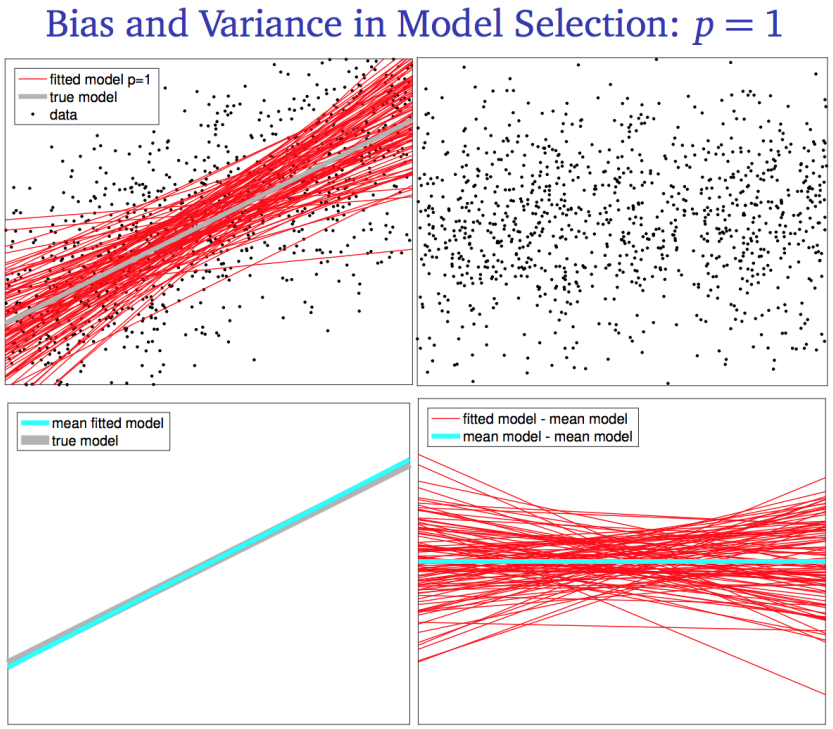
Let's combine all of these hypotheses together into one picture to see the bias and variance of our model.



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.

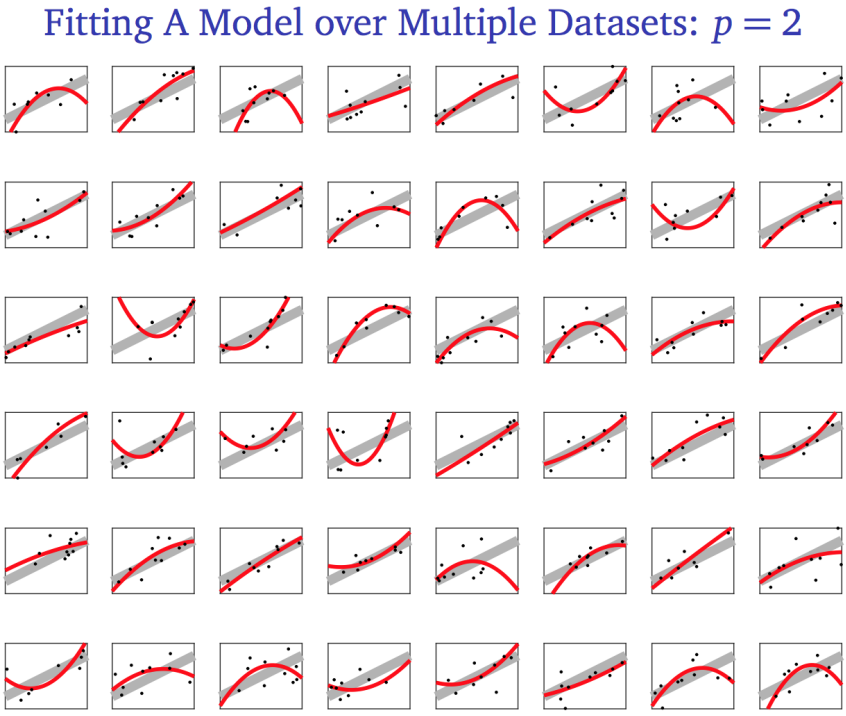
Now let's look at a 1st degree (linear) regression model.

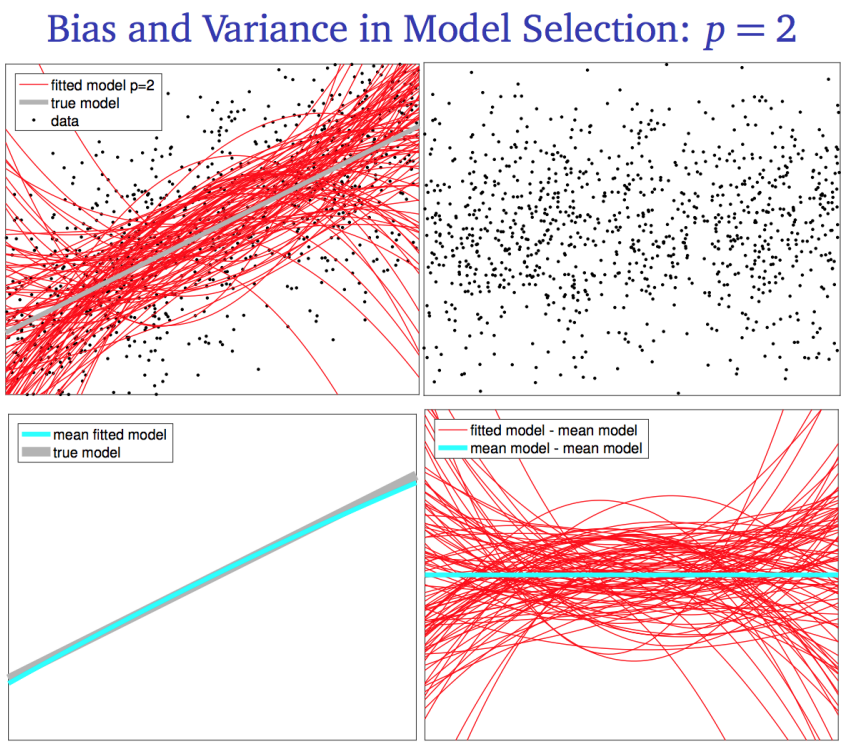


The bias is now very low bias for all x's. The variance is low for x's in the middle of the graph,

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.

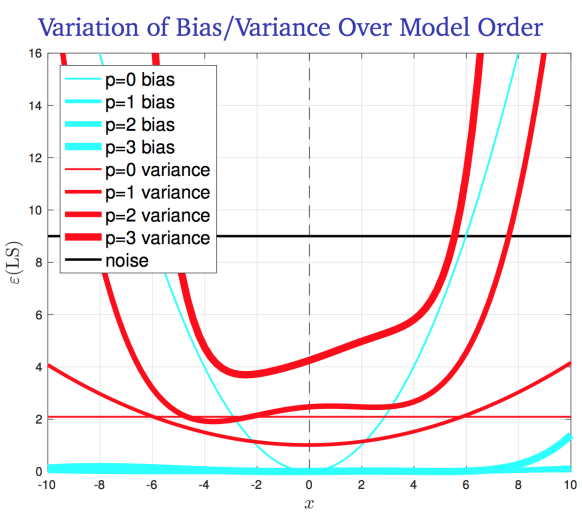


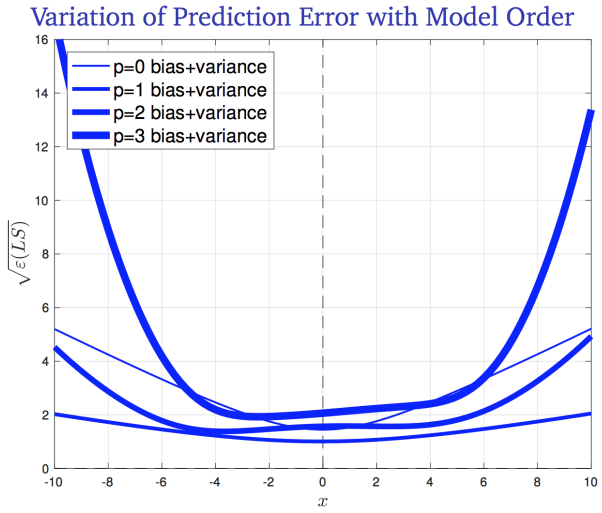


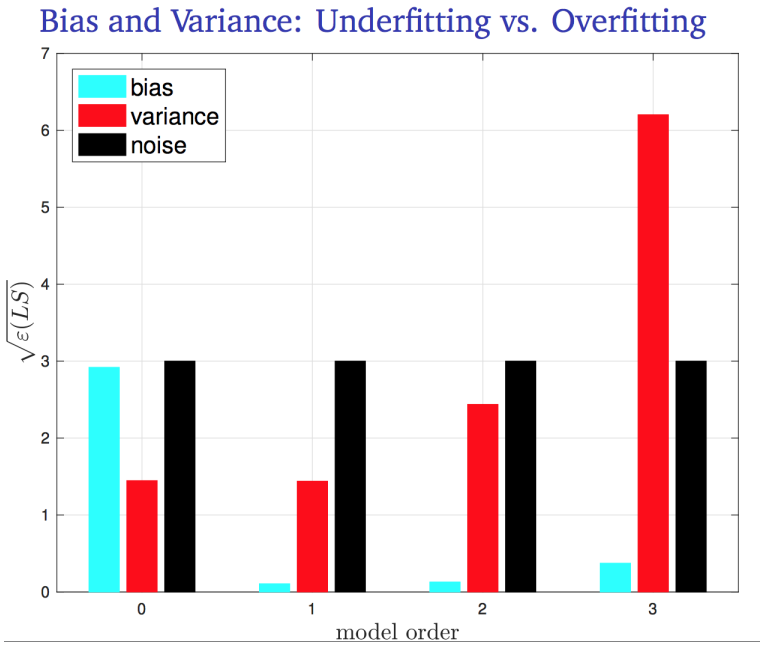
The bias is still very low for all x's. However, the variance is much higher for all values of x.

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

variance over all values of x, as shown in the third graph.







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 $\text{Var}(N)$, 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.

Chapter 2

MLE, MAP, and Gaussians

2.1 MLE and MAP

So far, we've explored a number of ideas under the least squares framework:

- Ordinary Least Least Squares

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \phi(x_i)^T \mathbf{w})^2 = \|\mathbf{y} - \mathbf{A}\mathbf{w}\|_2^2 \implies \mathbf{w}_{OLS}^* = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{y}$$

- Ridge Regression

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \phi(x_i)^T \mathbf{w})^2 + \lambda^2 \sum_{j=1}^{d} w_j^2 = \|\mathbf{y} - \mathbf{A}\mathbf{w}\|_2^2 + \lambda^2 \|\mathbf{w}\|_2^2 \implies \mathbf{w}_{Ridge}^* = (\mathbf{A}^T\mathbf{A} + \lambda^2\mathbf{I})^{-1}\mathbf{A}^T\mathbf{y}$$

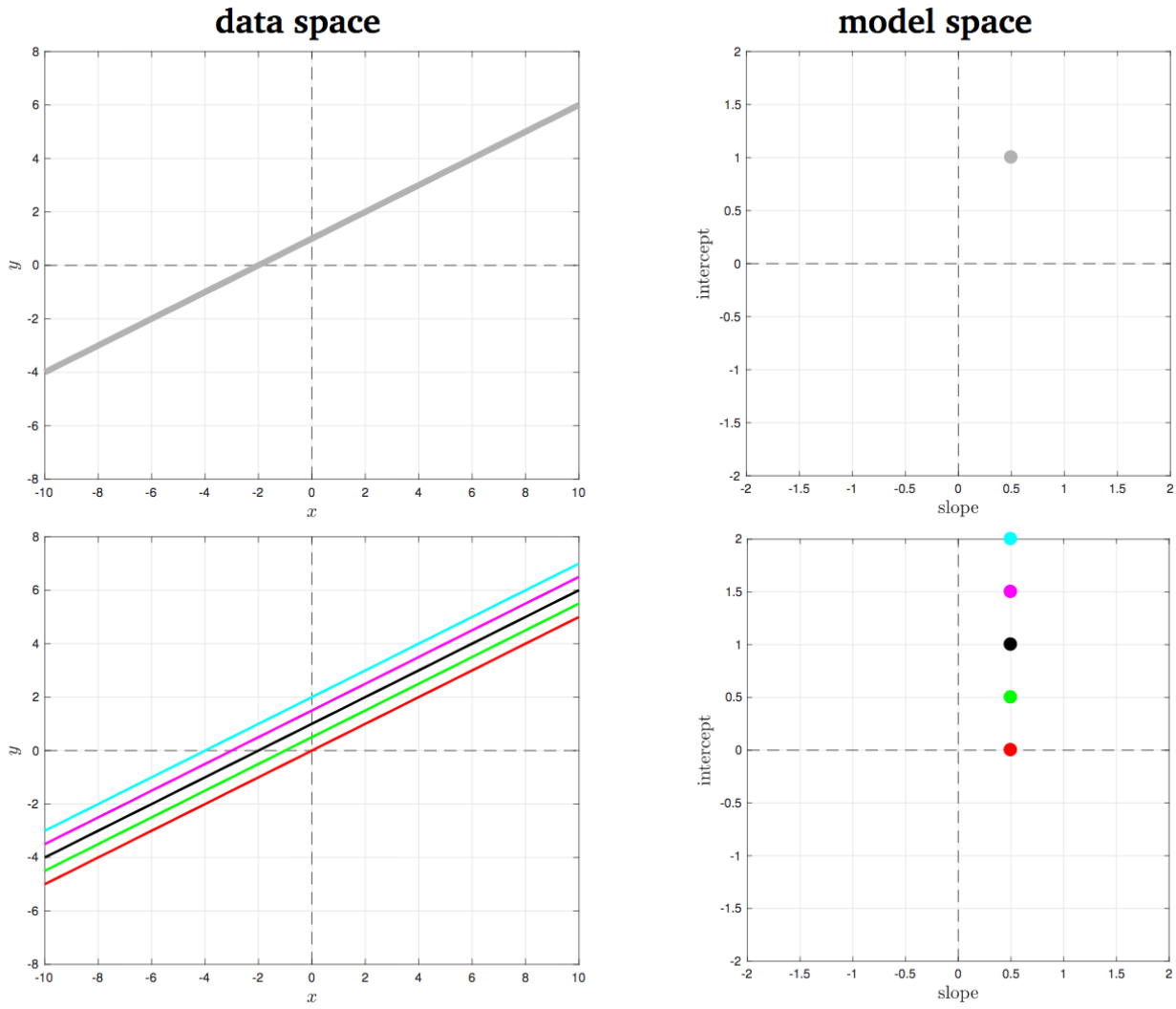
One question that you may have been asking yourself is why we are using the squared error to measure the effectiveness of our model, and why we use the $\ell 2$ norm for the model weights (and not some other norm). We will justify all of these design choices by exploring the statistical interpretations of supervised learning methods and in particular, regression methods. In more concrete terms, we will use a variety of concepts from probability, such as Gaussians, MLE and MAP, in order to validate what we've done so far through a different lens.

Optimization in Model Space

Let's look at a simple regression model:

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

Our goal is to find the optimal slope and intercept values that we believe best describe the data. Each arbitrary (slope, intercept) pair forms a line in **data space**. However, it can also be viewed as a single point in **model space**. Learning the optimal model amounts to fitting a line to the data points in the data space; it's equivalent to locating the optimal parameter point in the model space:

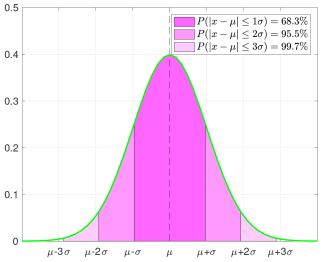


Data as Samples from Distributions

The data that we observe are random samples with different distributions, coverages, and densities. There are many different distributions that we will encounter, such as **Unifrom**, **Gaussian**, and **Laplacian**.

However, it is arguable that Gaussians distributions are by far the most prevalent. For the purposes of this section we will assume that you already have had exposure to Gaussian distributions before. Let's review their properties for convenience. Gaussians are particularly appealing because they occur quite frequently in natural settings. Noise distributions in particular are often assumed to be Gaussian distributions, partly because most of noise is captured within 1 or 2 standard deviations of the mean:

random variable:	$X \sim p(x)$
probability distribution:	$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
mean:	$E[X] = \int_{-\infty}^{+\infty} x p(x) dx = \mu$
variance:	$V[X] = E[(X-\mu)^2] = \int_{-\infty}^{+\infty} (x-\mu)^2 p(x) dx = \sigma^2$
parameters:	$X \sim \mathcal{N}(\mu, \sigma^2)$
log-likelihood:	$\log P(x) = -\frac{(x-\mu)^2}{2\sigma^2} - \log(\sqrt{2\pi}\sigma)$



$$X \sim \mathcal{N}(\mu_X, \sigma_X^2), \qquad Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$$

linear combinations:	$E[aX + bY] = a\mu_X + b\mu_Y$
if X, Y are independent:	$V[aX + bY] = a^2\sigma_X^2 + b^2\sigma_Y^2$
	$aX + bY \sim \mathcal{N}(a\mu_x + b\mu_Y, a^2\sigma_X^2 + b^2\sigma_Y^2)$

MLE and MAP for Model Selection

In the context of regression (and all of supervised learning for that matter), we assume a **true underlying model** that maps inputs to outputs. Our goal as machine learning experts is to find a **hypothesis model** that best represents the true underlying model, by using the data we are given.

Let's define more concretely our definition of the data and the model. Our data takes consists of n (x, y) pairs, just as we have seen before:

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

The true underlying model f is a function that maps the inputs x_i to the true outputs $f(x_i)$. Each **observation** y_i is simply a noisy version of $f(x_i)$:

$$y_i = f(x_i) + N_i$$

Note that $f(x_i)$ is a constant, while N_i is a random variable. We always assume that N_i has zero mean, because otherwise there would be systematic bias in our observations. The N_i's could be gaussian, uniform, laplacian, etc.. Here, let us assume that they are **i.i.d** and gaussian: $N_i \overset{\text{i.i.d}}{\sim} \mathcal{N}(0, \sigma^2)$. We can therefore say that $y_i | x_i \sim \mathcal{N}(f(x_i), \sigma^2)$.

Now that we have introduced the data and model, we wish to find a hypothesis model that best describes 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 model that maximizes the probability of the data. If we denote the set of hypothesis models as $\mathcal{H}$, we can represent the

problem as:

$$h^*_{MLE} = \arg\max_{h \in \mathcal{H}} P(\text{data} = \mathcal{D}| \text{ true model} = h)$$

More concretely:

$$h^*_{MLE} = \arg\max_{h \in \mathcal{H}} P(y_1, \ldots, y_n | x_1, \ldots, x_n, h)$$

Note that we actually conditioned on the 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 $f(x_i)$).

From the chain rule of probability, we can expand the probability statement:

$$P(y_1, \ldots, y_n | x_1, \ldots, x_n, h) = P(y_1 | x_1, \ldots, x_n, h) \cdot P(y_2 | y_1, x_1, x_2 \ldots, x_n, h) \cdot \ldots \cdot P(y_n | y_1, \ldots, y_{n-1}, x_1, \ldots, x_n, h)$$

We can simplify this expression by viewing the problem as a graphical model. Note that y_i only depends on its parent in the graphical model, x_i. It does not depend on the other y_j's, since all y's have independent noise terms. We can therefore simplify the objective:

$$h^*_{MLE} = \arg\max_{h \in \mathcal{H}} P(y_1, \ldots y_n | x_1, \ldots x_n, h) = \prod_{i=1}^{n} P(y_i | x_i, h)$$

Now let's focus on each individual term $P(y_i|x_i, h)$. We know that $y_i|x_i, h \sim \mathcal{N}(h(x_i), \sigma^2)$, which is cumbersome to work with because gaussians have exponential terms. So instead we wish to work with logs, which eliminate the exponential terms:

$$h^*_{MLE} = \arg\max_{h \in \mathcal{H}} log[P(y_1, \ldots y_n | x_1, \ldots x_n, h)] = \sum_{i=1}^{n} log[P(y_i | x_i, h)]$$

Note that with logs we are still working with the same problem, because logarithms are monotonic functions. Let's try to understand this mathematically through calculus:

$$\frac{d}{dx} log(f(x)) = 0 \iff \frac{1}{f(x)} \frac{df}{dx} = 0 \iff \frac{df}{dx} = 0$$

The last statement is true in this case since $f(x)$ is a gaussian function and thus can never equal 0. Continuing with logs:

$$\begin{aligned}
h^*_{MLE} &= \arg\max_{h \in \mathcal{H}} \sum_{i=1}^{n} log[P(y_i | x_i, h)] && (2.1) \\
&= \arg\max_{h \in \mathcal{H}} -\Big(\sum_{i=1}^{n} \frac{(y_i - h(x_i))^2}{2\sigma^2} \Big) - n \log \sqrt{2\pi}\sigma && (2.2) \\
&= \arg\min_{h \in \mathcal{H}} \Big(\sum_{i=1}^{n} \frac{(y_i - h(x_i))^2}{2\sigma^2} \Big) + n \log \sqrt{2\pi}\sigma && (2.3) \\
&= \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} (y_i - h(x_i))^2 && (2.4)
\end{aligned}$$

Note that in step (3) we turned the problem from a maximization problem to a minimization problem by negating the objective. In step (4) 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. In that case our hypothesis has the form $h(x_i) = \phi(x_i)^T \mathbf{w}$, where $\mathbf{w} \in \mathbb{R}^d$, where d is the number of dimensions of our featurized datapoints. For this specific setting, the problem becomes:

$$\mathbf{w}^*_{MLE} = \arg\min_{\mathbf{w} \in \mathbb{R}^d} \Big(\sum_{i=1}^{n} \frac{(y_i - \phi(x_i)^T \mathbf{w})^2}{2\sigma^2} \Big)$$

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:

$$\begin{aligned}
h^*_{MAP} &= \arg\max_{h \in \mathcal{H}} P(\text{true model} = h | \text{ data} = \mathcal{D}) && (2.1)\\
&= \arg\max_{h \in \mathcal{H}} \frac{P(\text{true model} = h, \text{ data} = \mathcal{D})}{P(\text{ data} = \mathcal{D})} && (2.2)\\
&= \arg\max_{h \in \mathcal{H}} c \cdot P(\text{true model} = h, \text{ data} = \mathcal{D}) && (2.3)\\
&= \arg\max_{h \in \mathcal{H}} P(\text{true model} = h, \text{ data} = \mathcal{D}) && (2.4)\\
&= \arg\max_{h \in \mathcal{H}} P(\text{ data} = \mathcal{D} | \text{true model} = h) \cdot P(\text{true model} = h) && (2.5)
\end{aligned}$$

Here, we used **Bayes' Rule** to reexpress the objective. In step (3) we represent $P(\text{ data} = \mathcal{D})$ 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 $P(h)$ to our objective. This term is the **prior** over our true model.

More concretely, we have (just as we did with MLE):

$$\begin{aligned}
h^*_{MAP} &= \arg\max_{h \in \mathcal{H}} P(h | y_1, \ldots, y_n, x_1, \ldots, x_n) && (2.1)\\
&= \arg\max_{h \in \mathcal{H}} \frac{P(h, y_1, \ldots, y_n | x_1, \ldots, x_n)}{P(y_1, \ldots, y_n | x_1, \ldots, x_n)} && (2.2)\\
&= \arg\max_{h \in \mathcal{H}} c \cdot P(h, y_1, \ldots, y_n | x_1, \ldots, x_n) && (2.3)\\
&= \arg\max_{h \in \mathcal{H}} P(h, y_1, \ldots, y_n | x_1, \ldots, x_n) && (2.4)\\
&= \arg\max_{h \in \mathcal{H}} P(y_1, \ldots, y_n | h, x_1, \ldots, x_n) \cdot P(h) && (2.5)\\
&= \arg\max_{h \in \mathcal{H}} log[P(y_1, \ldots, y_n | h, x_1, \ldots, x_n) \cdot P(h)] && (2.6)\\
&= \arg\max_{h \in \mathcal{H}} log[P(y_1, \ldots, y_n | h, x_1, \ldots, x_n)] + log[P(h)] && (2.7)\\
&= \arg\max_{h \in \mathcal{H}} \Big(\sum_{i=1}^{n} log[P(y_i | x_i, h)] \Big) + log[P(h)] && (2.8)
\end{aligned}$$

Again, just as in MLE, notice that we condition on the x_i's in the whole process because we treat them as constants. Also, let us assume as before that the noise terms are i.i.d and gaussian: $N_i \overset{\text{i.i.d}}{\sim} \mathcal{N}(0, \sigma^2)$. For the prior term $P(h)$, we assume that it follows a shifted and scaled version of the standard Multivariate Gaussian distribution: $h \sim \mathcal{N}(h_0, \sigma_h^2 I)$. Using this specific information, we now have:

$$h^*_{MAP} = \arg\max_{h \in \mathcal{H}} \Big(\sum_{i=1}^{n} log[P(y_i|x_i, h)] \Big) + log[P(h)] \tag{2.1}$$

$$= \arg\max_{h \in \mathcal{H}} \Big(- \frac{\sum_{i=1}^{n} (y_i - h(x_i))^2}{2\sigma^2} \Big) + \Big(\frac{-\|h - h_0\|^2}{2\sigma_h^2} \Big) \tag{2.2}$$

$$= \arg\min_{h \in \mathcal{H}} \Big(\frac{\sum_{i=1}^{n} (y_i - h(x_i))^2}{2\sigma^2} \Big) + \Big(\frac{\|h - h_0\|^2}{2\sigma_h^2} \Big) \tag{2.3}$$

$$= \arg\min_{h \in \mathcal{H}} \Big(\sum_{i=1}^{n} (y_i - h(x_i))^2 \Big) + \frac{\sigma^2}{\sigma_h^2} \Big(\|h - h_0\|^2 \Big) \tag{2.4}$$

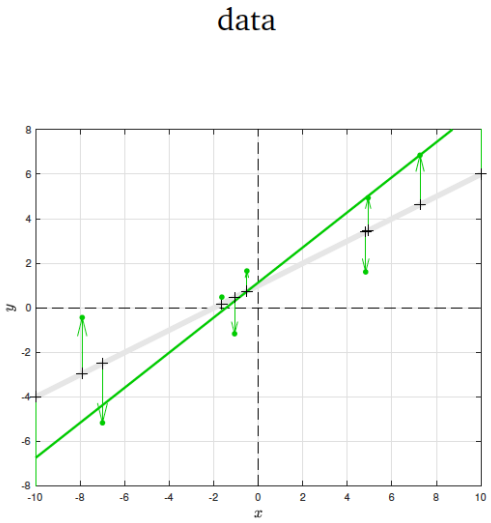
Let's look again at the case for linear regression to illustrate the effect of the prior term when $h_0 = 0$:

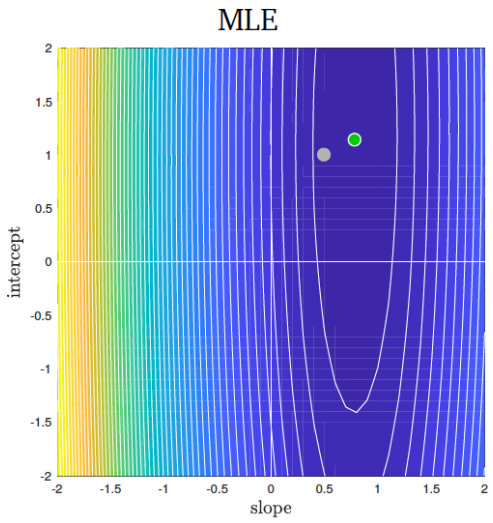
$$\mathbf{w}^*_{MAP} = \arg\min_{\mathbf{w} \in \mathbb{R}^d} \Big(\sum_{i=1}^{n} (y_i - \phi(x_i)^T \mathbf{w}))^2 \Big) + \frac{\sigma^2}{\sigma_h^2} \|\mathbf{w}\|^2$$

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 $\lambda = \frac{\sigma}{\sigma_h}$. We conclude that MAP is a probabilistic justification for adding the penalized ridge term in Ridge Regression.

MLE vs. MAP

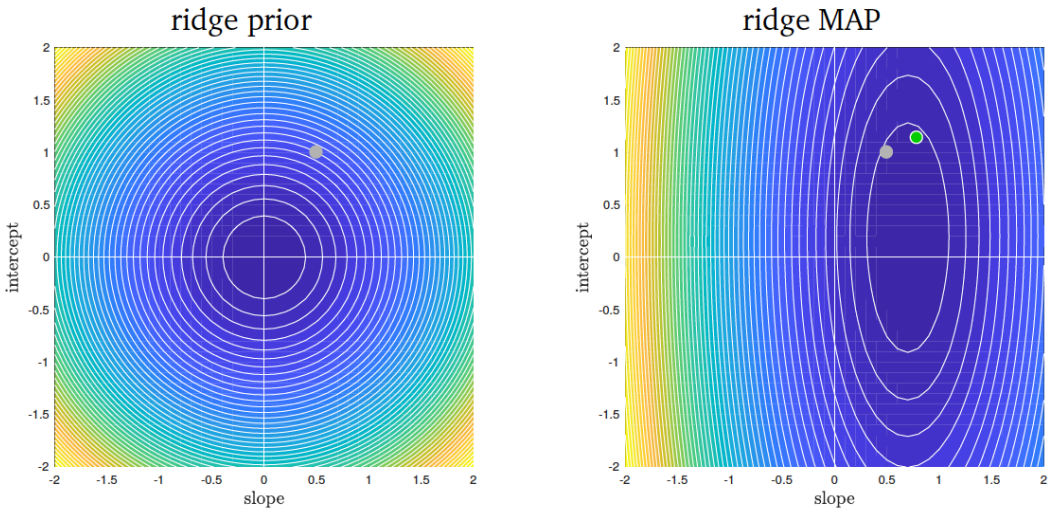
Based on our analysis of Ordinary Least Squares Regression and Ridge Regression, we should expect to see MAP perform better than MLE. But is that always the case? Let us revisit at the (slope, intercept) example from earlier. We already know the true underlying model parameters, and we will compare them to the values that MLE and MAP select. Let's start with MLE:





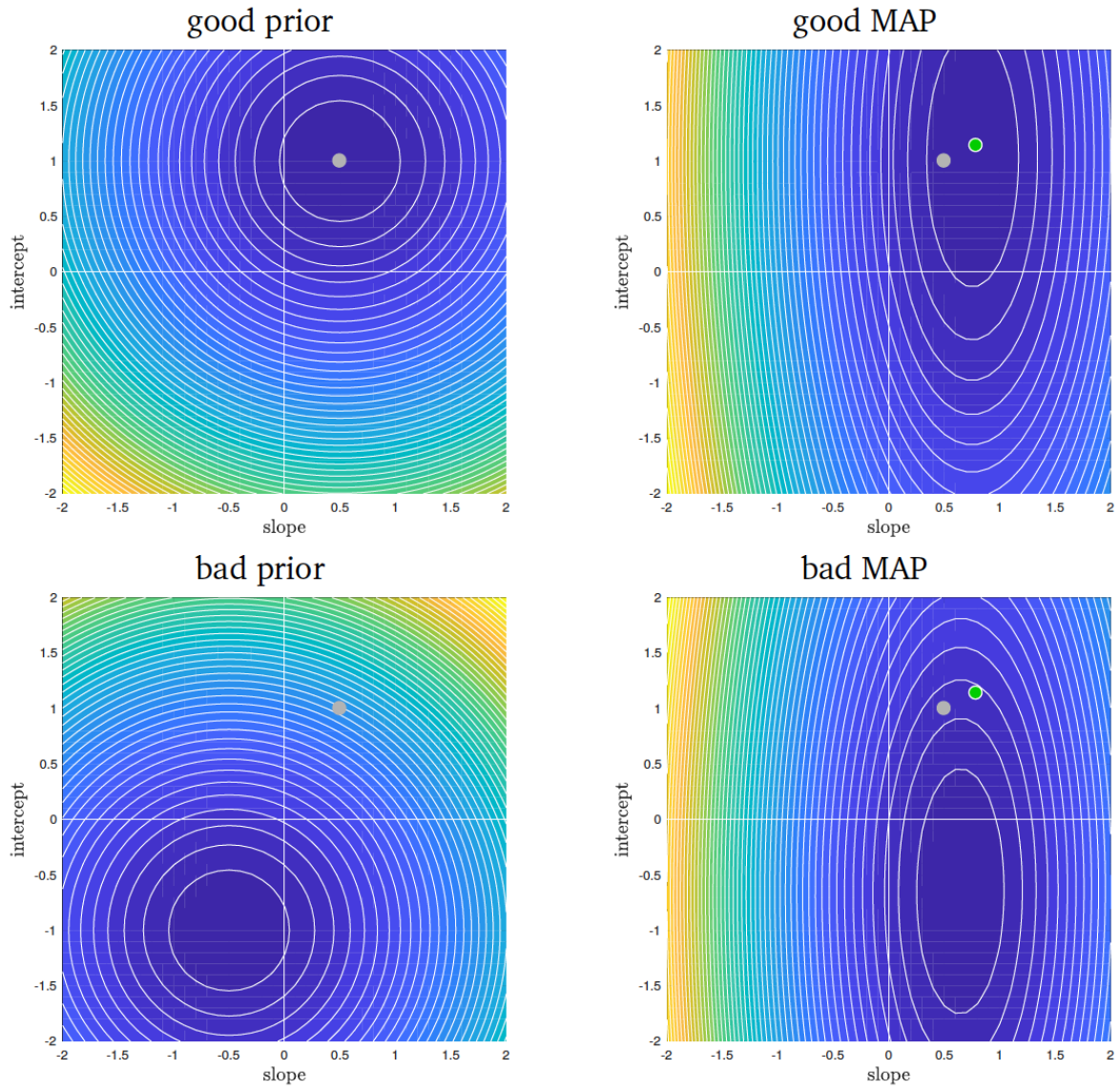
The diagram on the left shows the data space representation of the problem, and the diagram on the right shows the model space representation. The gray line in the left diagram and the gray dot in the right diagram are the true underlying model. Using noisy data samples, MLE predicts a reasonable hypothesis model (as indicated by the green line in the left diagram and the green dot in the right diagram).

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 your belief of what the true underlying model is. If you have reason to believe that the model weights should all be small, then the prior should be centered at zero. Let's look at MAP for a prior that is centered at zero:



For reference, we have marked the MLE estimation from before as a green point and the true model as a gray point. As we can see from the right diagram, using a prior centered at zero leads us to skew our prediction of the model weights toward the origin, leading to a less accurate model than MLE.

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. As the first set of diagrams below show, our prediction would be better than MLE. 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 Weighted Least Squares

So far we have used MLE in the context of Gaussian noise to justify the optimization formulation of regression problems, such as OLS. Let's apply this dual optimization-probability philosophy to other regression problems, such as **Weighted Least Squares**.

Optimization View

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. From an optimization perspective, the problem can be expressed as

$$\mathbf{w}^*_{WLS} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} \Big(\sum_{i=1}^{n} \omega_i (y_i - \phi(x_i)^T \mathbf{w})^2 \Big)$$

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

$$\mathbf{w}^*_{WLS} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y} - \mathbf{A}\mathbf{w})^T \mathbf{\Omega} (\mathbf{y} - A\mathbf{w})$$

Where the i'th row $\mathbf{A}$ is $\phi(x_i)^T$, and $\mathbf{\Omega} \in \mathbb{R}^{n\times n}$ is a diagonal matrix with $\mathbf{\Omega}_{i,i} = \omega_i$.

We rewrite the WLS objective to an OLS objective:

$$\begin{aligned}
\mathbf{w}^*_{WLS} &= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y} - \mathbf{A}\mathbf{w})^T \mathbf{\Omega} (\mathbf{y} - \mathbf{A}\mathbf{w}) && (2.1)\\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y} - \mathbf{A}\mathbf{w})^T \mathbf{\Omega}^{1/2} \mathbf{\Omega}^{1/2} (\mathbf{y} - \mathbf{A}\mathbf{w}) && (2.2)\\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{\Omega}^{1/2}\mathbf{y} - \mathbf{\Omega}^{1/2}\mathbf{A}\mathbf{w})^T (\mathbf{\Omega}^{1/2}\mathbf{y} - \mathbf{\Omega}^{1/2}\mathbf{A}\mathbf{w}) && (2.3)
\end{aligned}$$

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

$$\mathbf{w}^*_{WLS} = \Big((\mathbf{\Omega}^{1/2}\mathbf{A})^T (\mathbf{\Omega}^{1/2}\mathbf{A}) \Big)^{-1} \Big(\mathbf{\Omega}^{1/2}\mathbf{A} \Big)^T \mathbf{\Omega}^{1/2}\mathbf{y} = (\mathbf{A}^T\mathbf{\Omega}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{\Omega}\mathbf{y}$$

Probabilistic View

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

Our probabilistic model looks like

$$y_i = \phi(x_i)^T \mathbf{w} + Z_i$$

where the Z_i's are still independent Gaussians random variables, but not necessarily identical: $Z_i \sim \mathcal{N}(0, \sigma_i^2)$. We can morph the problem into an MLE one by scaling the data to make sure all the Z_i's are identically distributed, by dividing by σ_i:

$$\frac{y_i}{\sigma_i} = \frac{\phi(x_i)^T \mathbf{w}}{\sigma_i} + \frac{Z_i}{\sigma_i}$$

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

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

We rewrite our original problem as a scaled MLE problem:

$$\mathbf{w}^*_{WLS} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} \Big(\sum_{i=1}^{n} \frac{(\frac{y_i}{\sigma_i} - \frac{\phi(x_i)^T\mathbf{w}}{\sigma_i})^2}{2(1)} \Big) + n \log\sqrt{2\pi}(1)$$

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

$$\mathbf{w}^*_{WLS} = (\mathbf{A}^T\boldsymbol{\Sigma}_z^{-\frac{1}{2}}\boldsymbol{\Sigma}_z^{-\frac{1}{2}}\mathbf{A})^{-1}\mathbf{A}^T\boldsymbol{\Sigma}_z^{-\frac{1}{2}}\boldsymbol{\Sigma}_z^{-\frac{1}{2}}\mathbf{y} = (\mathbf{A}^T\boldsymbol{\Sigma}_z^{-1}\mathbf{A})^{-1}\mathbf{A}^T\boldsymbol{\Sigma}_z^{-1}\mathbf{y}$$

where

$$\boldsymbol{\Sigma}_z = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \sigma_n^2 \end{bmatrix}$$

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

Dependent Noise

What if the Z_i's aren't independent? This usually happens for time series data. Again, we assume we have a model for how the noise behaves. The noise entries are not independent, but there is a known process.
e.g. $Z_{i+1} = rZ_i + U_i$ where $U_i \sim \mathcal{N}(0,1)$, i.i.d, $-1 \leq r \leq 1$ (so that it doesn't blow up)
or a "sliding window" example (like echo of audio) where $Z_i = \boldsymbol{\Sigma} r_j U_{i-j}$, $U_i \sim \mathcal{N}(0,1)$.

In general, we can always represent the noise vector as

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

where $\mathbf{Z} \in \mathbb{R}^n$, $\mathbf{R} \in \mathbb{R}^{n\times n}$, $\mathbf{U} \in \mathbb{R}^n$, and $U_i \overset{\text{i.i.d}}{\sim} \mathcal{N}(0,1)$.

$\mathbf{Z}$ is a **Jointly Gaussian Random Vector**. Our goal now is to derive its probability density formula.

2.3 Multivariate Gaussians

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

1. A random vector $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_k)^T$ is JG if there exists a base random vector $\mathbf{U} = (U_1, U_2, \ldots, U_l)^T$ whose components are independent standard normal random variables, a transition matrix $\mathbf{R} \in \mathbb{R}^{k\times l}$, and a mean vector $\boldsymbol{\mu} \in \mathbb{R}^k$, such that $\mathbf{Z} = \mathbf{RU} + \boldsymbol{\mu}$.
2. A random vector $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_k)^T$ is JG if $\sum_{i=1}^k a_i Z_i$ is normally distributed for every $a = (a_1, a_2, \ldots, a_k)^T \in \mathbb{R}^k$.
3. (Non-degenerate case only) A random vector $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_k)^T$ is JG if

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

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

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

Proving (1) $\implies$ (3)

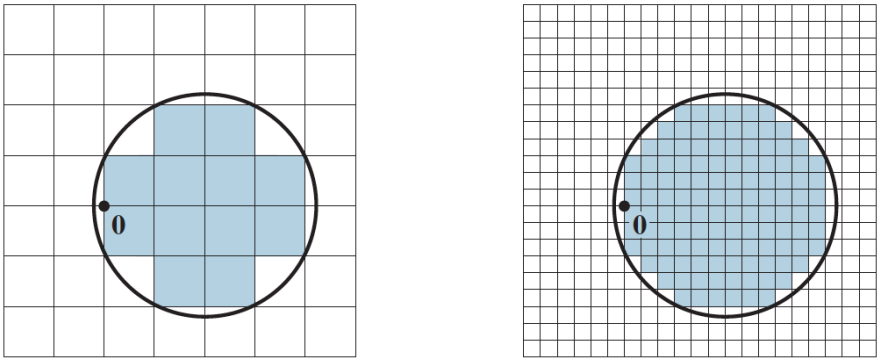
In the context of the noise problem we defined earlier, we are starting with condition (1), ie. $\mathbf{Z} = \mathbf{RU}$ (in this case $k = l = n$), and we would like to derive the probability density of $\mathbf{Z}$. Note that here we removed the $\boldsymbol{\mu}$ from consideration because in machine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

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

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

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

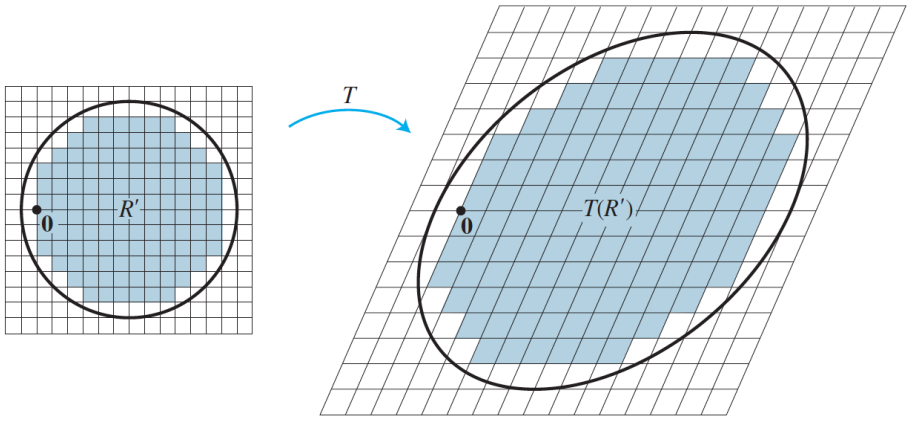
Let's see why this is true, with a simple 2D geometric explanation. Define $\mathbf{U}$ space to be the 2D space with axes U_1 and U_2. Now take any arbitrary region $\mathbf{R}'$ in $\mathbf{U}$ space (note that this $\mathbf{R}'$ is different from the matrix $\mathbf{R}$ that relates $\mathbf{U}$ to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



Mathematically, we have that

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

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



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

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

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

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

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

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

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

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

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

and

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

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

Therefore, we can say that

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

We conclude that

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

An almost identical argument will allow us to state that

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

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

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

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

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

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

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

Furthermore, we know that $|\det(\mathbf{\Sigma}_Z)| = |\det\Big(\mathbf{R}\mathbf{R}^T\Big)| = |\det(\mathbf{R})\cdot\det\Big(\mathbf{R}^T\Big)| = |\det(\mathbf{R})\cdot\det(\mathbf{R})| = |\det(\mathbf{R})|^2$ and therefore

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

2.4 MLE with Dependent Noise

Up to this point, we have been able to easily view regression problems from an optimization perspective. In the context of dependent noise however, it is much easier to view the problem from a probabilistic perspective. Note as before that:

$$\mathbf{y} = \mathbf{A}\mathbf{w} + \mathbf{Z}$$

Where $\mathbf{Z}$ is now a jointly Gaussian random vector. That is, $\mathbf{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_Z)$, and $\mathbf{y} \sim \mathcal{N}(\mathbf{A}\mathbf{w}, \mathbf{\Sigma}_Z)$. Our goal is to maximize the probability of our data over the set of possible $\mathbf{w}$'s:

$$\begin{aligned}
\mathbf{w}^* &= \arg\max_{\mathbf{w}\in\mathbb{R}^d} \frac{1}{\sqrt{|\det(\mathbf{\Sigma}_Z)|}} \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{1}{2}(\mathbf{y}-\mathbf{A}\mathbf{w})^T\mathbf{\Sigma}_Z^{-1}(\mathbf{y}-\mathbf{A}\mathbf{w})} && (2.1) \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^d} (\mathbf{y}-\mathbf{A}\mathbf{w})^T\mathbf{\Sigma}_Z^{-1}(\mathbf{y}-\mathbf{A}\mathbf{w}) && (2.2)
\end{aligned}$$

Notice that $\mathbf{\Sigma}_Z$ is symmetric, which means it has a good eigen structure, therefore we can take the advantage interpret this geometrically. $\mathbf{\Sigma}_Z$ can be written as

$$\mathbf{\Sigma}_Z = \mathbf{Q} \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \sigma_n^2 \end{bmatrix} \mathbf{Q}^T$$

where $\mathbf{Q}$ is orthonormal. This means a multivariate Gaussian can be thought of having its level sets be the ellipsoid having axis given by $\mathbf{Q}$ and amount of stretch given by σs.

Let

$$\mathbf{R}_Z = \mathbf{\Sigma}_Z^{\frac{1}{2}} = \mathbf{Q} \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & \sigma_n \end{bmatrix}$$

As before, we can scale the data to morph the problem into an MLE problem with i.i.d noise variables, by premultiplying the data matrix $\mathbf{A}$ and the observation vector $\mathbf{y}$ by $\mathbf{R}_Z^{-1}$.

In a very similar fashion to the independent noise problem, the MLE of the scaled dependent noise problem is $\mathbf{w}^* = (\mathbf{A}^T\mathbf{\Sigma}_Z^{-1}\mathbf{A})^{-1}\mathbf{A}^T\mathbf{\Sigma}_Z^{-1}\mathbf{y}$.

2.5 MAP with Colored Noise

Recall the ordinary least squares (OLS) model. We have a dataset $\mathcal{D} = \{(\mathbf{a}_i, y_i)\}_{i=1}^n$ and assume that each y_i is a linear function of $\mathbf{a}_i$, plus some independent Gaussian noise, which we have rescaled to have variance 1:

$$z_i \overset{\text{iid}}{\sim} \mathcal{N}(0, 1) \tag{2.3}$$

$$y_i = \mathbf{a}_i^\top \mathbf{w} + z_i \tag{2.4}$$

Initially we used the geometric interpretation of OLS to solve for $\mathbf{w}$. The previous two lectures showed how we can find $\mathbf{w}$ with **estimators** instead:

1. Maximum likelihood estimation (**MLE**):

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \log P(\mathcal{D} \mid \mathbf{w})$$

2. Maximum a posteriori estimation (**MAP**):

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \log P(\mathbf{w} \mid \mathcal{D}) = \arg\max_{\mathbf{w}} \log P(\mathcal{D} \mid \mathbf{w}) + \log P(\mathbf{w})$$

When deriving ridge regression via MAP estimation, our prior assumed that w_i were i.i.d. (univariate) Gaussian, but more generally, we can allow $\mathbf{w}$ to be any multivariate Gaussian:

$$\mathbf{w} \sim \mathcal{N}(\boldsymbol{\mu}_w, \Sigma_w)$$

Recall (see Discussion 4) that we can rewrite a multivariate Gaussian variable as an affine transformation of a standard Gaussian variable:

$$\mathbf{w} = \Sigma_w^{1/2} \underbrace{\mathbf{v}}_{\text{noise}} + \underbrace{\boldsymbol{\mu}_\mathbf{w}}_{\text{mean}} \qquad\qquad \mathbf{v} \sim \mathcal{N}(0, I)$$

This change of variable is sometimes called the *reparameterization trick*.

Plugging this reparameterization into our approximation $\mathbf{Y} \approx A\mathbf{w}$ gives

$$\begin{aligned}\mathbf{Y} &\approx A\Sigma_w^{1/2}\mathbf{v} + A\boldsymbol{\mu}_\mathbf{w} \\ A\Sigma_w^{1/2}\mathbf{v} &\approx \mathbf{Y} - A\boldsymbol{\mu}_\mathbf{w} \\ \hat{\mathbf{v}} &= (\Sigma_w^{T/2}A^\top A\Sigma_w^{1/2} + I)^{-1}\Sigma_w^{T/2}A^\top(\mathbf{y} - A\boldsymbol{\mu}_\mathbf{w})\end{aligned}$$

Since the variance from data and prior have both been normalized, the noise-to-signal ratio (λ) is equal to 1.

However $\mathbf{v}$ is not what we care about – we need to convert back to the actual weights $\mathbf{w}$ in order to make predictions. Using our identity again,

$$\begin{aligned}\hat{\mathbf{w}} &= \boldsymbol{\mu}_w + \Sigma_w^{1/2}(\Sigma_w^{T/2}A^\top A\Sigma_w^{1/2} + I)^{-1}\Sigma_w^{T/2}A^\top(\mathbf{y} - A\boldsymbol{\mu}_\mathbf{w}) \\ &= \boldsymbol{\mu}_w + (A^\top A + \underbrace{\Sigma_w^{-T/2}\Sigma_w^{-1/2}}_{\Sigma_w^{-1}})^{-1}A^\top(\mathbf{y} - A\boldsymbol{\mu}_\mathbf{w})\end{aligned}$$

Note that there are two terms: the prior mean $\boldsymbol{\mu}_w$, plus another term that depends on both the data and the prior. The precision matrix of $\mathbf{w}$'s prior (Σ_w^{-1}) controls how the data fit error affects our estimate.

To gain intuition, let us consider the simplified case where

$$\Sigma_w = \begin{bmatrix}\sigma_{w,1}^2 & 0 & \cdots & 0 \\ 0 & \sigma_{w,2}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{w,n}^2\end{bmatrix}$$

When the prior variance $\sigma_{w,j}^2$ 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 Σ_w^{-1} will be small, as desired.

Conversely if $\sigma_{w,j}^2$ 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:

$$\mathbf{u}, \mathbf{v} \overset{\text{iid}}{\sim} \mathcal{N}(\mathbf{0}, I) \tag{2.5}$$

$$\begin{bmatrix}\mathbf{Y} \\ \mathbf{w}\end{bmatrix} = \begin{bmatrix}R_z & AR_w \\ 0 & R_w\end{bmatrix}\begin{bmatrix}\mathbf{u} \\ \mathbf{v}\end{bmatrix} \tag{2.6}$$

where R_z and R_w are relationships with w and z, respectively. Note that the R_w appears twice because our model assumes $\mathbf{Y} = A\mathbf{w} + \text{noise}$, so if $\mathbf{w} = R_w\mathbf{v}$, then we must have $\mathbf{Y} = AR_w\mathbf{v} + \text{noise}$.

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

$$\begin{aligned}u', v' &\overset{\text{iid}}{\sim} \mathcal{N}(\mathbf{0}, I) \\ \begin{bmatrix}\mathbf{w} \\ \mathbf{Y}\end{bmatrix} &= \begin{bmatrix}A & B \\ 0 & D\end{bmatrix}\begin{bmatrix}u' \\ v'\end{bmatrix}\end{aligned}$$

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

$$\begin{aligned}
\mathbb{E}[\mathbf{w} \mid \mathbf{Y}] &= \mathbb{E}[Au' + Bv' \mid \mathbf{Y}] \\
&= \mathbb{E}[Au' \mid \mathbf{Y}] + \mathbb{E}[BD^{-1}\mathbf{Y} \mid \mathbf{Y}] \\
&= A\underbrace{\mathbb{E}[u']}_{\mathbf{0}} + \mathbb{E}[BD^{-1}\mathbf{Y} \mid \mathbf{Y}] \\
&= BD^{-1}\mathbf{Y} \\
\mathrm{Var}(\mathbf{w} \mid \mathbf{Y}) &= \mathbb{E}[(\mathbf{w} - \mathbb{E}[\mathbf{w}])(\mathbf{w} - \mathbb{E}[\mathbf{w}])^\top \mid \mathbf{Y}] \\
&= \mathbb{E}[(Au' + BD^{-1}\mathbf{Y} - BD^{-1}\mathbf{Y})(Au' + BD^{-1}\mathbf{Y} - BD^{-1}\mathbf{Y})^\top \mid \mathbf{Y}] \\
&= \mathbb{E}[(Au')(Au')^\top \mid \mathbf{Y}] \\
&= \mathbb{E}[Au'(u')^\top A^\top] \\
&= A\underbrace{\mathbb{E}[u'(u')^\top]}_{=\mathrm{Var}(u')=I} A^\top \\
&= AA^\top
\end{aligned}$$

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

$$\mathbf{w} \mid \mathbf{Y} \sim \mathcal{N}(BD^{-1}\mathbf{Y}, AA^\top)$$

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

$$\begin{aligned}
\mathbb{E}\left[\begin{bmatrix}\mathbf{w} \\ \mathbf{Y}\end{bmatrix} \quad \begin{bmatrix}\mathbf{w}^\top & \mathbf{Y}^\top\end{bmatrix}\right] &= \begin{bmatrix}A & B \\ 0 & D\end{bmatrix}\begin{bmatrix}A^\top & 0 \\ B^\top & D^\top\end{bmatrix} \\
&= \begin{bmatrix}AA^\top + BB^\top & BD^\top \\ DB^\top & DD^\top\end{bmatrix} \\
&= \begin{bmatrix}\Sigma_w & \Sigma_{w,Y} \\ \Sigma_{Y,w} & \Sigma_Y\end{bmatrix}
\end{aligned}$$

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

$$\begin{aligned}
BD^{-1}\mathbf{Y} &= B\underbrace{D^\top D^{-T}}_{I}D^{-1}\mathbf{Y} = (BD^\top)(DD^\top)^{-1}\mathbf{Y} = \Sigma_{w,Y}\Sigma_Y^{-1}\mathbf{Y} \\
AA^\top &= AA^\top + BB^\top - BB^\top \\
&= AA^\top + BB^\top - B\underbrace{D^\top D^{-T}}_{I}\underbrace{D^{-1}D}_{I}B^\top \\
&= AA^\top + BB^\top - (BD^\top)(DD^\top)^{-1}DB^\top \\
&= \Sigma_w - \Sigma_{w,Y}\Sigma_Y^{-1}\Sigma_{Y,w}
\end{aligned}$$

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

$$\begin{aligned}
\mathbb{E}\left[\begin{bmatrix}\mathbf{Y} \\ \mathbf{w}\end{bmatrix} \quad \begin{bmatrix}\mathbf{Y}^\top & \mathbf{w}^\top\end{bmatrix}\right] &= \begin{bmatrix}R_z R_z^\top + AR_w R_w^\top A^\top & AR_w R_w^\top \\ R_w R_w^\top A^\top & R_w R_w^\top\end{bmatrix} \\
&= \begin{bmatrix}\Sigma_Y & \Sigma_{Y,w} \\ \Sigma_{w,Y} & \Sigma_w\end{bmatrix}
\end{aligned}$$

Therefore after defining $\Sigma_z = R_z R_z^\top$, we can read off

$$\begin{aligned}
\Sigma_w &= R_w R_w^\top \\
\Sigma_Y &= \Sigma_z + A\Sigma_w A^\top \\
\Sigma_{Y,w} &= A\Sigma_w \\
\Sigma_{w,Y} &= \Sigma_w A^\top
\end{aligned}$$

Plugging this into our estimator yields

$$\begin{aligned}
\hat{\mathbf{w}} &= \mathbb{E}[\mathbf{w} \mid \mathbf{Y} = \mathbf{y}] \\
&= \Sigma_{w,Y}\Sigma_Y^{-1}\mathbf{y} \\
&= \Sigma_w A^\top (\Sigma_z + A\Sigma_w A^\top)^{-1}\mathbf{y}
\end{aligned}$$

One may be concerned because this expression does not take the form we expect – the inverted matrix is hitting $\mathbf{y}$ directly, unlike in other solutions we've seen. But using the Woodbury matrix identity[1], it turns out that we can rewrite this expression as

$$\hat{\mathbf{w}} = (A^\top \Sigma_z^{-1} A + \Sigma_w^{-1})^{-1} A^\top \Sigma_z^{-1}\mathbf{y}$$

which looks more familiar.

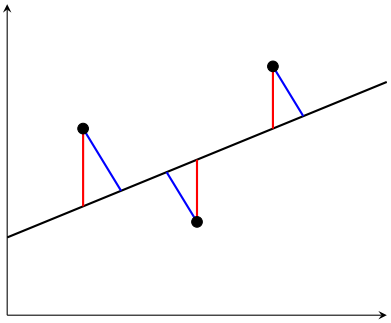
[1] $(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$

Chapter 3

Low-Rank approximation

3.1 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**), which arises in the case where we assume that our x data is also corrupted by noise. Both LS methods want to get a model that produce an approximation closest to all the points, but they measure the distance differently. OLS tries to minimize the vertical distance between the fitted line and data points, while TLS tries to minimize the perpendicular distance.



The **red** line represents **vertical distance**, which OLS aims to minimize. The **blue** line represents **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.

We might begin with a probabilistic formulation and fit the parameters via maximum likelihood estimation, as before. Suppose on the plane, we have a true model that we want to recover from some data points:

$$y_i = ax_i \tag{1}$$

and we observe data points in the form

$$(x_i + z_{xi}, y_i + z_{yi}) \tag{2}$$

where the noise terms are normally distributed, i.e. $z_{xi}, z_{yi} \overset{\text{iid}}{\sim} \mathcal{N}(0, 1)$.

Applying the equation (1) that comes from the true model mentioned above, we rewrite (2) in the

form $(x_i + z_{xi}, ax_i + z_{yi})$, giving

$$y = ax \underbrace{- az_x + z_y}_{\sim \mathcal{N}(0, a^2+1)} \tag{3}$$

Given these assumptions, we can derive the likelihood for just 1 point under hypothesis a:

$$P(x_i, y_i; a) = \frac{1}{\sqrt{2\pi(a^2+1)}} \exp\left(-\frac{1}{2}\frac{(y_i - ax_i)^2}{a^2+1}\right) \tag{4}$$

Thus the log likelihood is

$$\log P(x_i, y_i; a) = \text{constant} - \frac{1}{2}\log\Big(a^2+1\Big) - \frac{1}{2}\frac{(y_i - ax_i)^2}{a^2+1} \tag{5}$$

Observe that a shows up in three places, unlike the form that we are familiar with, where a 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.

Solution

To solve the TLS problem, we develop another formulation that can be solved using the singular value decomposition.

Assume we have n data points, $\mathbf{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R}$, and we stack them to get

$$(X + \mathbf{e})\mathbf{w} = \mathbf{y} + \mathbf{f} \tag{6}$$

where $\mathbf{w} \in \mathbb{R}^d$ is the weight, and E and $\mathbf{f}$ are noise terms that we add to explain the error in the model. Our goal is to minimize the Frobenius norm[1] of the matrix composed of these error vectors. Recall that 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. Thus we arrive at the following constrained optimization problem:

$$\min_{E,\mathbf{f}} \left\| \begin{bmatrix} E & \mathbf{f} \end{bmatrix} \right\|_F^2 \quad \text{subject to} \quad (X + E)\mathbf{w} = \mathbf{y} + \mathbf{f}$$

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

$$\underbrace{\Big(\begin{bmatrix} X & \mathbf{y} \end{bmatrix} + \begin{bmatrix} E & \mathbf{f} \end{bmatrix}\Big)}_{\in \mathbb{R}^{n \times (d+1)}} \begin{bmatrix} \mathbf{w} \\ -1 \end{bmatrix} = 0 \tag{7}$$

In linear algebraic terms, this expression says that the vector $(\mathbf{w}, -1)$ lies in the nullspace of the matrix on the left. Note that $\begin{bmatrix} X & \mathbf{y} \end{bmatrix}$ would not be full rank if we observe the data with no noise, since we would have $\mathbf{y} = X\mathbf{w}$, which implies that the columns are linearly dependent. But the observations we get have noise, which makes the training data matrix full rank. To compensate,

[1] Recall that the Frobenius norm is like the standard Euclidean norm but applied to the elements of a matrix instead of a vector:

$$\|A\|_F^2 = \sum_i \sum_j A_{ij}^2$$

we must add something to it so that it loses rank, since otherwise the nullspace is just $\{\mathbf{0}\}$ and the equation cannot be solved. We use the SVD coordinate system to achieve this:

$$\begin{bmatrix} X & \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_{d+1} & | & U_{rest} \end{bmatrix} \begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_{d+1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^\top \\ \vdots \\ \mathbf{v}_{d+1}^\top \end{bmatrix} \tag{8}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{d+1} > 0$, and U and V are orthogonal matrices. Recall that this implies that multiplication by U or V does not change the Frobenius norm, so minimizing $\| \begin{bmatrix} E & \mathbf{f} \end{bmatrix} \|_F^2$ is equivalent to minimizing $\| \begin{bmatrix} E' & \mathbf{f}' \end{bmatrix} \|_F^2$ where $E', \mathbf{f}'$ are $E, \mathbf{f}$ expressed in the SVD coordinates. Now our problem reduces to finding $E', \mathbf{f}'$ such that

$$\begin{bmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_{d+1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} + \begin{bmatrix} E' & \mathbf{f}' \end{bmatrix} \tag{9}$$

is not full rank and $\| \begin{bmatrix} E' & \mathbf{f}' \end{bmatrix} \|_F^2$ is as small as possible. Since the matrix on the left is diagonal, we can reduce its rank by simply zeroing out one of its diagonal elements. Therefore our perturbation $\begin{bmatrix} E' & \mathbf{f}' \end{bmatrix}$ will have $-\sigma_j$ in the (j, j) position for some j, and zeros everywhere else. To minimize the size of the perturbation, we decide to eliminate the smallest σ_j by taking

$$\begin{bmatrix} E' & \mathbf{f}' \end{bmatrix} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & -\sigma_{d+1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

Such a perturbation in SVD coordinates corresponds to a perturbation of

$$\begin{bmatrix} E & \mathbf{f} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_1 & \dots & \mathbf{u}_{d+1} & | & U_{rest} \end{bmatrix} \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & -\sigma_{d+1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} \mathbf{v}_1^\top \\ \vdots \\ \mathbf{v}_{d+1}^\top \end{bmatrix} = -\sigma_{d+1} \mathbf{u}_{d+1} \mathbf{v}_{d+1}^\top$$

in the original coordinate system. It turns out that this choice is optimal, as guaranteed by the Eckart-Young theorem, which is stated at the end for reference.

The nullspace of our resulting matrix is then

$$\text{null}\left(\begin{bmatrix} X & \mathbf{y} \end{bmatrix} + \begin{bmatrix} E & \mathbf{f} \end{bmatrix} \right) = \text{null}\left(\sum_{j=1}^{d} \sigma_j \mathbf{u}_j \mathbf{v}_j^\top \right) = \text{span}\{\mathbf{v}_{d+1}\}$$

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

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

Once we have $\mathbf{v}_{d+1}$, or any scalar multiple of it, we simply rescale it so that the second component is -1, and then the first component gives us w. Since $\mathbf{v}_{d+1}$ is a right-singular vector of $\begin{bmatrix} X & \mathbf{y} \end{bmatrix}$, it is an eigenvector of the matrix

$$\begin{bmatrix} X & \mathbf{y} \end{bmatrix}^\top \begin{bmatrix} X & \mathbf{y} \end{bmatrix} = \begin{bmatrix} X^\top X & X^\top \mathbf{y} \\ \mathbf{y}^\top X & \mathbf{y}^\top \mathbf{y} \end{bmatrix} \tag{11}$$

So to find it we solve

$$\begin{bmatrix} X^\top X & X^\top \mathbf{y} \\ \mathbf{y}^\top X & \mathbf{y}^\top \mathbf{y} \end{bmatrix} \begin{bmatrix} \mathbf{w} \\ -1 \end{bmatrix} = \sigma_{d+1}^2 \begin{bmatrix} \mathbf{w} \\ -1 \end{bmatrix} \tag{12}$$

To gain an extra perspective, ignore the bottom equation (we can do this because we have an extra degree of freedom) and consider the solution of the top equation:

$$X^\top X \mathbf{w} - X^\top \mathbf{y} = \sigma_{d+1}^2 \mathbf{w} \tag{13}$$

which can be rewritten as

$$(X^\top X - \sigma_{d+1}^2 I)\mathbf{w} = X^\top \mathbf{y} \tag{14}$$

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:

$$X = X_{\text{true}} + Z_x$$

where X_{true} are the actual values before noise corruption, and Z_x is a noise term. Then

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

Observe that the off-diagonal terms of $\mathbb{E}[Z_x^\top Z_x]$ terms are zero because the ith and jth rows of Z_x 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.

Eckart-Young Theorem

The Eckart-Young theorem essentially says that the best low-rank approximation (in terms of the Frobenius norm) 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 $A = U\Sigma V^\top = \sum_{i=1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$ be its singular value decomposition. Then*

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

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 $\text{rank}(\tilde{A}) \leq k$.

3.2 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

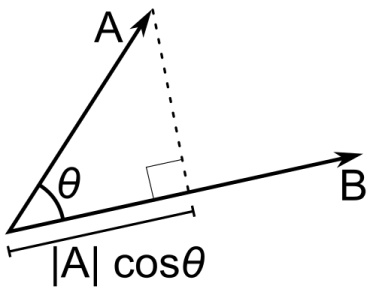
Principal Component Analysis (PCA) is an unsupervised dimensionality reduction technique. Given a matrix of data points, it finds one or more orthogonal directions that capture the largest amount of variance in the data. Intuitively, the directions with less variance contain less information and may be discarded without introducing too much error.

Projection

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

$$x^\top u = \|x\|\|u\| \cos\theta$$

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



Formulating PCA

Let $X \in \mathbb{R}^{n \times d}$ be our matrix of data, where each row is a d-dimensional data point. We will assume that the data points have mean zero; otherwise we subtract the mean to make them zero-mean:

$$X - \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top X, \text{ where } \mathbf{1}_n = \begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}$$

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. For example, translating all the data by some fixed vector could completely change the principal components if we did not center.

Recall that for any random variable Z,

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

so if $\mathbb{E}[Z] = 0$ then $\text{Var}(Z) = \mathbb{E}[Z^2]$.

Hence once X is zero-mean, the variance of the projections is given by

$$\epsilon_{PCA\_var}(u) = \sum_{i=1}^{n}(x_i^\top u)^2 = \|Xu\|^2$$

where u is constrained to have unit norm. We want to maximize the variance, so the objective becomes

$$\max_{\|u\|=1} \epsilon_{PCA\_var}(u) = \max_{\|u\|=1} \|Xu\|^2 = \max_{u \neq 0} \frac{\|Xu\|^2}{\|u\|^2} = \max_{u \neq 0} \frac{u^\top X^\top X u}{u^\top u}$$

The ratio on the right is known as a *Rayleigh quotient*. We will see that Rayleigh quotients are heavily related to eigenvalues, so anytime you see one, your eigensense should tingle.

Rayleigh Quotients

Suppose $M \in \mathbb{R}^{d \times d}$ is a real, symmetric ($M = M^\top$) matrix. The Rayleigh quotient of M is defined as

$$R(u; M) = \frac{u^\top M u}{u^\top u}$$

Denote the eigenvalues of M by $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$, with corresponding eigenvectors $v_1, \ldots, v_d$. That is, $Mv_j = \lambda_j v_j$ for $j = 1, \cdots, d$. If we stack the v_j as columns of a matrix V:

$$V = \begin{bmatrix} v_1 & \cdots & v_d \end{bmatrix}$$

then the eigenvector equations can be simulatneously written as

$$MV = V\Lambda$$

where

$$\Lambda = \text{diag}(\lambda_1, \cdots, \lambda_d)$$

Then since V is an orthogonal matrix, it is invertible with $V^{-1} = V^\top$, so

$$V^\top M V = \Lambda$$

Let u be a unit length vector. Since $\{v_j\}$ form a basis, we can write

$$u = \sum_{j=1}^{d} \alpha_j v_j = V\alpha$$

Then since V is an orthogonal matrix, $\|\alpha\| = 1$ as well. Now

$$R(u; M) = \frac{u^\top M u}{u^\top u} = \alpha^\top V^\top M V \alpha = \alpha^\top \Lambda \alpha = \sum_{j=1}^{d} \lambda_j \alpha_j^2$$

Since we have the constraint $\alpha_1^1 + \cdots + \alpha_d^2 = 1$, the right-most expression is a weighted average of the eigenvalues and hence bounded by the smallest and largest of these:

$$\lambda_d \leq R(u; M) \leq \lambda_1$$

The lower bound is achieved by putting $\alpha_d = \pm 1$, $\alpha_{j\neq d} = 0$, and the upper bound by $\alpha_1 = \pm 1$, $\alpha_{j\neq 1} = 0$. The maximizing value can then be recovered by

$$\sum_{j=1}^{d} \alpha_j v_j = \pm v_1$$

That is, it is an eigenvector corresponding to the largest eigenvalue! Hence

$$\lambda_d = R(v_d; M) \leq R(u; M) \leq R(v_1; M) = \lambda_1$$

Finally, note that since the Rayleigh quotient is scale invariant, i.e. $R(\gamma u; M) = R(u; M)$ for any $\gamma \neq 0$, the inequality above holds for any scaling of the vectors, not just unit-length vectors.

Calculating the first principal component

Armed with our knowledge of Rayleigh quotients, the solution to the PCA problem is immediate:

$$\max_{\|u\|=1} \epsilon_{PCA\_var}(u) = \max_{u\neq 0} \underbrace{\frac{u^\top X^\top X u}{u^\top u}}_{R(u, X^\top X)} = \lambda_1(X^\top X)$$

where the maximizer u^* is a unit eigenvector corresponding to this eigenvalue. Writing $X = U\Sigma V^\top$, we have

$$X^\top X = V\Sigma^\top \underbrace{U^\top U}_{I} \Sigma V^\top = V\Sigma^\top \Sigma V^\top$$

The expression on the right is an eigendecomposition of $X^\top X$, so

$$\lambda_1(X^\top X) = [\Sigma^\top \Sigma]_{11} = \sigma_1^2(X)$$

with corresponding eigenvector $\mathbf{v}_1$, which is the first principal component.

Finding multiple principal components

We have seen how to derive the first principal component, which maximizes the variance of the projected data. But usually we will need more than one direction, since one direction is unlikely

to capture the data well. The basic idea here is to subtract off the contributions of the previously computed principal components, and then apply the same rule as before to what remains. If $u_{(1)}, \ldots, u_{(k-1)}$ denote the principal components already computed, this subtracting off is accomplished by

$$\hat{X} = X - \sum_{j=1}^{k-1} X u_{(j)} u_{(j)}^\top = X \left(I - \sum_{j=1}^{k-1} u_{(j)} u_{(j)}^\top \right)$$

This expression should be understood as applying the same subtracting transformation to each row of the data[2]:

$$\hat{x}_i = \left(I - \sum_{j=1}^{k-1} u_{(j)} u_{(j)}^\top \right) x_i = x_i - \sum_{j=1}^{k-1} u_{(j)} u_{(j)}^\top x_i$$

The vector $u_{(j)} u_{(j)}^\top x_i$ should be recognized as the orthogonal projection of x_i onto the subspace spanned by $u_{(j)}$. Hence $\hat{x}_i$ is what's left when you start with x_i and then remove all the components that belong to the subspaces spanned by each $u_{(j)}$.[3]

We want to find the direction of largest variance subject to the constraint that it must be orthogonal to all the previously computed directions. Thus we have a constrained problem of the form

$$u_{(k)} = \underset{\forall j<k:\ u^\top u_{(j)}=0}{\arg\max} \frac{u^\top \hat{X}^\top \hat{X} u}{u^\top u}$$

But we don't want to actually compute $\hat{X}$. Fortunately, we don't have to! Consider that if u is orthogonal to $u_{(1)}, \ldots, u_{(k-1)}$ (as we constrain it to be), then

$$\hat{X} u = \left(X - \sum_{j=1}^{k-1} X u_{(j)} u_{(j)}^\top \right) u = X u - \sum_{j=1}^{k-1} X u_{(j)} \underbrace{u_{(j)}^\top u}_{0} = X u$$

Thus we can write the optimization problem above as

$$u_{(k)} = \underset{\forall j<k:\ u^\top u_{(j)}=0}{\arg\max} \frac{u^\top X^\top X u}{u^\top u}$$

eliminating the need to compute $\hat{X}$. Unsurprisingly, the solution to this problem is given by $u_{(k)} = v_k$, that is, a unit eigenvector corresponding to the kth largest eigenvalue of $X^\top X$.

Rather than iteratively computing each new $u_{(j)}$, we can view the problem of finding the first k principal components as a joint optimization problem over all k directions simultaneously. This amounts to maximizing the variance as projected onto a k-dimensional subspace:

$$U = \underset{U^\top U = I}{\arg\max} \sum_{j=1}^{k} u_{(j)}^\top X^\top X u_{(j)} = \underset{U^\top U = I}{\arg\max} \operatorname{tr}\Big(U^\top X^\top X U \Big)$$

For matrices U with orthogonal columns we can define

$$R(U; M) = R([u_1, \ldots, u_k]; M) = \sum_{j=1}^{k} R(u_j; M)$$

[2] To see this, take the transpose of both sides and use the symmetry of $I - \sum_j u_{(j)} u_{(j)}^\top$.

[3] This is exactly the same idea as the Gram-Schmidt process.

As before, the bounds for this expression are given in terms of the smallest and largest eigenvalues, but now there are k of them:

$$\begin{aligned}\lambda_1 + \lambda_2 + \cdots + \lambda_k &= R([v_1, v_2, \ldots, v_k]; M) \\ &\geq R(U; M) \\ &\geq R([v_{d-k+1}, \ldots, v_{d-1}, v_d]; M) \\ &= \lambda_{d-k+1} + \cdots + \lambda_{d-1} + \lambda_d\end{aligned}$$

Hence, projection onto the subspace spanned by the first k leading eigenvectors maximizes the variance of the projected data. We can find k principal components by computing the SVD, $X = U\Sigma V^\top$, and then taking the first k columns of the matrix V.

Projecting onto the PCA coordinate system

Once we have the principal components, we can use them as a new coordinate system. To do this we must project the data onto this coordinate system, which can be done in the same way as above (taking inner products). Each data point $x_i \in \mathbb{R}^d$ becomes a new vector $\tilde{x}_i \in \mathbb{R}^k$, where k is the number of principal components. The components of the projection write

$$[\tilde{x}_i]_j = x_i^\top u_j$$

We can compute all these vectors at once more efficiently using a matrix-matrix multiplication

$$\tilde{X} = XU$$

where $U \in \mathbb{R}^{d \times k}$ is a matrix whose columns are the principal components.

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

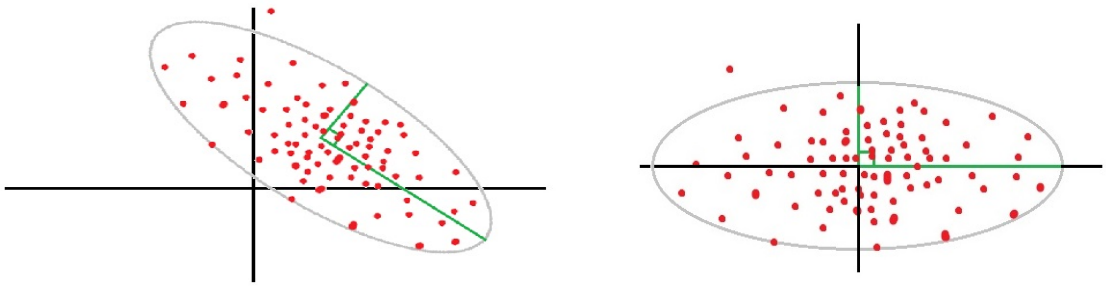


Figure 3.1: Left:data points; Right: PCA projection of data points

Observe that the data are uncorrelated in the projected space. Also note that this example does not show the full 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.

Other derivations of PCA

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

Gaussian assumption

Let us assume that the data are generated by a multivariate Gaussian distribution:

$$x_i \overset{\text{iid}}{\sim} \mathcal{N}(\mu, \Sigma)$$

Then the maximum likelihood estimate of the covariance matrix Σ is

$$\hat{\Sigma} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})^\top = \frac{1}{n}X^\top X$$

where $\bar{x}$ is the sample average and the matrix X is assumed to be zero-mean as before. The eigenvectors of $\hat{\Sigma}$ and $X^\top X$ are the same since they are positive scalar multiples of each other.

The contours of the multivariate Gaussian density form ellipsoids (see Figure 1). The direction of largest variance (i.e. the first principal component) is the eigenvector corresponding to the smallest eigenvalue of Σ^{-1}, which is the largest eigenvalue of Σ. We do not know Σ in general, so we use $\hat{\Sigma}$ in its place. Thus the principal component is an eigenvector corresponding to the largest eigenvalue of $\hat{\Sigma}$. As mentioned earlier, this matrix has the same eigenvalues and eigenvectors as $X^\top X$, so we arrive at the same solution.

Minimizing reconstruction error

Ordinary least squares minimizes the vertical distance between the fitted line and the data points:

$$\|y - Xu\|^2$$

We show that PCA can be interpreted as minimizing the perpendicular distance between the principal component subspace and the data points, so in this sense it is doing the same thing as total least squares.

The orthogonal projection of a vector x onto the subspace spanned by a unit vector u equals u scaled by the scalar projection of x onto u:

$$P_u x = (uu^\top)x = (x^\top u)u$$

Suppose we want to minimize the total reconstruction error:

$$\begin{aligned}
\epsilon_{PCA_Err}(u) &= \sum_{i=1}^{n} \|x_i - P_u x_i\|^2 \\
&= \sum_{i=1}^{n} \left(\|x_i\|^2 - \|P_u x_i\|^2 \right) && (*) \\
&= \sum_{i=1}^{n} \|x_i\|^2 - \sum_{i=1}^{n} \|(x_i^\top u)u\|^2 \\
&= \sum_{i=1}^{n} \|x_i\|^2 - \underbrace{\sum_{i=1}^{n} (x_i^\top u)^2}_{\epsilon_{PCA_Var}(u)}
\end{aligned}$$

where $(*)$ holds by the Pythagorean theorem

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

since $x - P_u x \perp P_u x$. Then since the first term $\sum_i \|x_i\|^2$ is constant with respect to u, we have

$$\arg\min_u \epsilon_{PCA\_Err}(u) = \arg\min_u \text{ constant} - \epsilon_{PCA\_Var}(u) = \arg\max_u \epsilon_{PCA\_Var}(u)$$

Hence minimizing reconstruction error is equivalent to maximizing projected variance.

3.3 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 $a, b, c, d \in \mathbb{R}$,

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

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

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

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

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

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

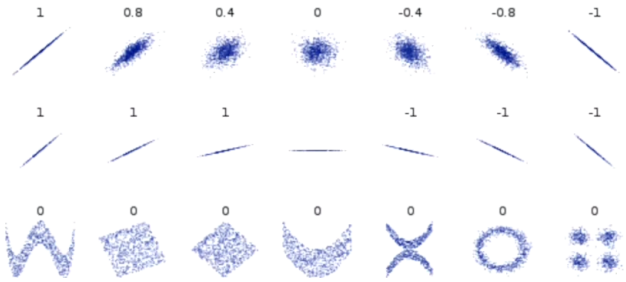
$$\text{Var}(X) = \text{Cov}(X, X) = \mathbb{E}[(X - \mathbb{E}[X])^2] \approx \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

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

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

$$\begin{aligned}\hat{\rho}(x,y) &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \cdot \sum_{i=1}^n (y_i - \bar{y})^2}} \\ &= \frac{\tilde{x}^\top \tilde{y}}{\sqrt{\tilde{x}^\top \tilde{x} \cdot \tilde{y}^\top \tilde{y}}} \quad \text{where } \tilde{x} = x - \bar{x}, \tilde{y} = y - \bar{y}\end{aligned}$$

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



You should notice that:

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

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

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

so

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

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 $X_{\text{rv}} \in \mathbb{R}^p$ and $Y_{\text{rv}} \in \mathbb{R}^q$, we want to find projection vectors $u \in \mathbb{R}^p$ and $v \in \mathbb{R}^q$ that maximizes the correlation between $X_{\text{rv}}^\top u$ and $Y_{\text{rv}}^\top v$:

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

Observe that

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

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

$$\begin{aligned}
\operatorname{Var}(X_{\text{rv}}^\top u) &= u^\top \operatorname{Cov}(X_{\text{rv}}, X_{\text{rv}}) u \\
\operatorname{Var}(Y_{\text{rv}}^\top v) &= v^\top \operatorname{Cov}(Y_{\text{rv}}, Y_{\text{rv}}) v
\end{aligned}$$

so the correlation writes

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

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

$$\operatorname{Cov}(X_{\text{rv}}, Y_{\text{rv}}) = \mathbb{E}[(X_{\text{rv}} - \underbrace{\mathbb{E}[X_{\text{rv}}]}_{0})(Y_{\text{rv}} - \underbrace{\mathbb{E}[Y_{\text{rv}}]}_{0})^\top] = \mathbb{E}[X_{\text{rv}} Y_{\text{rv}}^\top] \approx \frac{1}{n} \sum_{i=1}^{n} x_i y_i^\top = \frac{1}{n} X^\top Y$$

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

$$\begin{aligned}
\operatorname{Cov}(X_{\text{rv}}, X_{\text{rv}}) &= \mathbb{E}[X_{\text{rv}} X_{\text{rv}}^\top] \approx \frac{1}{n} \sum_{i=1}^{n} x_i x_i^\top = \frac{1}{n} X^\top X \\
\operatorname{Cov}(Y_{\text{rv}}, Y_{\text{rv}}) &= \mathbb{E}[Y_{\text{rv}} Y_{\text{rv}}^\top] \approx \frac{1}{n} \sum_{i=1}^{n} y_i y_i^\top = \frac{1}{n} Y^\top Y
\end{aligned}$$

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

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

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
$$S_x = \operatorname{diag}(\lambda_1(X^\top X), \ldots, \lambda_d(X^\top X))$$
where all the eigenvalues are positive, we can define the "square root" of this matrix by taking the square root of every diagonal entry:
$$S_x^{1/2} = \operatorname{diag}\left(\sqrt{\lambda_1(X^\top X)}, \ldots, \sqrt{\lambda_d(X^\top X)}\right)$$
Then, defining $W_x = U_x S_x^{-1/2} U_x^\top$, we have
$$\begin{aligned}(XW_x)^\top(XW_x) &= W_x^\top X^\top X W_x \\ &= U_x S_x^{-1/2} U_x^\top U_x S_x U_x^\top U_x S_x^{-1/2} U_x^\top \\ &= U_x S_x^{-1/2} S_x S_x^{-1/2} U_x^\top \\ &= U_x U_x^\top \\ &= I\end{aligned}$$
which shows that W_x is a whitening matrix for X. The same process can be repeated to produce a whitening matrix $W_y = U_y S_y^{-1/2} U_y^\top$ for Y.
Let's denote the whitened data $X_w = XW_x$ and $Y_w = YW_y$. Then by the change of variables $u_w = W_x^{-1}u, v_w = W_y^{-1}v$,
$$\begin{aligned}\max_{u,v} \hat{\rho}(Xu, Yv) &= \max_{u,v} \frac{(Xu)^\top Yv}{\sqrt{(Xu)^\top Xu (Yv)^\top Yv}} \\ &= \max_{u,v} \frac{(XW_xW_x^{-1}u)^\top YW_yW_y^{-1}v}{\sqrt{(XW_xW_x^{-1}u)^\top XW_xW_x^{-1}u (YW_yW_y^{-1}v)^\top YW_yW_y^{-1}v}} \\ &= \max_{u_w,v_w} \frac{(X_w u_w)^\top Y_w v_w}{\sqrt{(X_w u_w)^\top X_w u_w (Y_w v_w)^\top Y_w v_w}} \\ &= \max_{u_w,v_w} \frac{u_w^\top X_w^\top Y_w v_w}{\sqrt{u_w^\top X_w^\top X_w u_w \cdot v_w^\top Y_w^\top Y_w v_w}} \\ &= \max_{u_w,v_w} \underbrace{\frac{u_w^\top X_w^\top Y_w v_w}{\sqrt{u_w^\top u_w \cdot v_w^\top v_w}}}_{\hat{\rho}(X_w u_w, Y_w v_w)}\end{aligned}$$
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 $(X_w D_x)^\top (Y_w D_y)$ 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
$$(X_w U)^\top (Y_w V) = U^\top X_w^\top Y_w V = U^\top (USV^\top) V = S$$

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

$$\begin{aligned}
\max_{u_w, v_w} \hat{\rho}(X_w u_w, Y_w v_w) &= \max_{u_w, v_w} \frac{(X_w u_w)^\top Y_w v_w}{\sqrt{u_w^\top u_w \cdot v_w^\top v_w}} \\
&= \max_{u_w, v_w} \frac{(X_w D_x D_x^{-1} u_w)^\top Y_w D_y D_y^{-1} v_w}{\sqrt{(D_x u_w)^\top D_x u_w \cdot (D_y v_w)^\top D_y v_w}} \\
&= \max_{u_d, v_d} \frac{(X_d u_d)^\top Y_d v_d}{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}} \\
&= \max_{u_d, v_d} \frac{u_d^\top X_d Y_d v_d}{\underbrace{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}}_{\hat{\rho}(X_d u_d, Y_d v_d)}} \\
&= \max_{u_d, v_d} \frac{u_d^\top S v_d}{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}}
\end{aligned}$$

Without loss of generality, suppose u_d and v_d are unit vectors[4] so that the denominator becomes 1, and we can ignore it:

$$\max_{u_d, v_d} \frac{u_d^\top S v_d}{\sqrt{u_d^\top u_d \cdot v_d^\top v_d}} = \max_{\substack{\|u_d\|=1 \\ \|v_d\|=1}} \frac{u_d^\top S v_d}{\|u_d\| \|v_d\|} = \max_{\substack{\|u_d\|=1 \\ \|v_d\|=1}} u_d^\top S v_d$$

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

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

where S_{ii}, the singular values of $X_w^\top Y_w$, 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:

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

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:

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

[4] Why can we assume this? Observe that the value of the objective does not change if we replace u_d by αu_d and v_d by βv_d, where α and β 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.

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

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

where I_k denotes the k-dimensional identity matrix. Then

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

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 ϵ has huge variance. What happens when we run PCA on Y? Since PCA maximizes variance, it will actually project Y (largely) into the column space of ϵ! 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

$$\begin{aligned} A &= (X_c^\top X_c)^{-1} X_c^\top Y_c \\ &= (U^\top X^\top X U)^{-1} U^\top X^\top Y V \end{aligned}$$

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

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

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

$$\begin{aligned} \hat{Y} &= \hat{Y}_c (V^\top V)^{-1} V^\top \\ &= \tilde{X} U (U^\top X^\top X U)^{-1} (U^\top X^\top Y V) (V^\top V)^{-1} V^\top \end{aligned}$$

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

$$A_{\text{eq}} = \underbrace{U}_{\text{projection}} \underbrace{(U^\top X^\top X U)^{-1}}_{\text{whitening}} \underbrace{(U^\top X^\top Y V)}_{\text{decorrelation}} \underbrace{(V^\top V)^{-1} V^\top}_{\text{projection back}}$$

3.4 Dimensionality Reduction

There are many issues with working in high dimensions. As the dimension d grows, machine learning algorithms can become more computationally intensive. It also becomes more difficult to visualize our data - humans are notoriously bad at visualizing beyond 3 dimensions. Additionally, redundant features can add more noise than signal. There is a widely held belief that most natural data in high dimensions (for example, data used in genomics) can be represented in a lower dimensional space.

Dimensionality reduction is an *unsupervised* method - unlike supervised learning, there are no labels that we need to match, no classes to predict. As such, defining problems becomes more subjective and heuristic. One approach to dimensionality reduction is feature selection, in which we remove features that we deem to be irrelevant based on some criteria. For example, the LASSO provides this feature selection using L^1 regularization.

Another approach to dimensionality reduction is learning latent features. This approach seeks to find new latent features that are transformations of our given features that represent the data well.

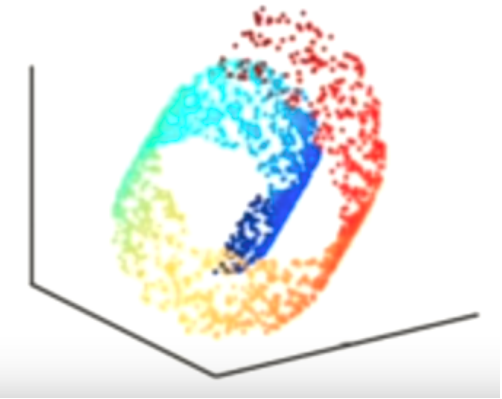


Figure 3.2: The Swiss Roll dataset is a 2-dimensional manifold embedded in 3 dimensional space. We should be able to represent it with a 2-dimensional feature space.

Principal Components Analysis

PCA can be used for dimensionality reduction. Recall that finding the PCA decomposition of $X \in \mathbb{R}^{n \times D}$ amounts to finding the SVD

$$X = USV^T$$

Suppose $d << D$, and let $\tilde{U}, \tilde{S}, \tilde{V}$ contain the first d columns of U, S, V respectively. Then $\tilde{V}^T x$ projects data x into a d-dimensional latent space. This projection into a lower-dimensional space

will typically cause some loss of information, but if the data really does live in a d-dimensional subspace, then we should be able to reconstruct x via $\tilde{V}\tilde{V}^T x$ and obtain something that is close to the original x. PCA can be seen as finding a basis for a subspace that minimizes this reconstruction error (low-rank approximation property).

Sometimes, it does not make sense to find orthogonal directions that capture the maximum variance, as PCA does. Independent Components Analysis instead seeks directions that are statistically independent, and is more suitable for some applications.

Nonnegative Matrix Factorization

NMF takes a non-negative matrix X, where each column is a data entry, and approximately factors it into $X = BH$, where B is skinny, H is fat, and both these matrices are non-negative. The k-th column of X is the sum of the columns of B, weighted according to the entries in the k-th column of H. In this regard, each column of B can be seen as a feature that contributes to the data in a meaningful way - the number of columns in B is the number of features we are allowed to reconstruct the data with. If each column of X is a vectorized image, then each column of B will be a vectorized image. NMF learns a part-based representation of the data, since the reconstruction is a non-negative linear combination of features of the same dimension as the data. It turns out that NMF will tend to produce sparser features than PCA. NMF has applications in computer vision and recommender systems, among other fields.

Multidimensional Scaling

MDS seeks to learn a low-dimensional representation such that pairwise distances between points are exactly preserved in the latent representation. Specifically, if $X_i \in \mathbb{R}^D$ and $W_{ij} = \|X_i - X_j\|^2$, then MDS finds Y_i such that $\|Y_i - Y_j\|^2 \approx W_{ij}$. More generally, one can use any dissimilarity measure $d(X_i, X_j)$ in place of the norm. If the dissimilarity measure is a metric, then MDS is equivalent to PCA. MDS is typically used to visualize high dimensional data.

Note that while MDS captures interpoint distances in its low-dimensional embedding, it is incapable of capturing local geometric structure - for example, in the Swiss roll dataset shown above, points in the red region are dissimilar from points in the blue region, but are relatively close to these points in distance, so MDS will provide a representation where red and blue points are not separated.

Nonlinear Methods

Linear methods such as PCA are not well-suited to capture intrinsic geometric structure in data. There are several approaches to solving this problem:

- Kernel methods: it is possible to derive a kernelized version of PCA, for example.
- Manifold learning: an n-manifold[5] is a surface that locally resembles n-dimensional Euclidean space. For example, the Swiss roll is a 2-manifold embedded in 3-dimensional space. In manifold learning, we learn a mapping from data to a low-dimensional manifold - for example, a mapping from Swiss roll to 2-dimensional plane.

[5]For the mathematically inclined, a manifold is a second-countable Hausdorff topological space such that each point has a neighborhood homeomorphic to a neighborhood in Euclidean space. Most machine learning researchers do not care for these definitions and will call almost anything a manifold.

Isometric Feature Mapping (IsoMap)

IsoMap performs MDS on the geodesic distances between points, as follows:

(1) Construct a local neighborhood graph by connecting each point with its k nearest neighbors.

(2) Compute all-pairs shortest path distances (these are the geodesic distances).

(3) Apply MDS on geodesic distances.

If we apply IsoMap to the Swiss roll, we see that while the Euclidean distance between the red and blue regions is low, the geodesic distance is high - the geodesic distance between points represents how far we would have to go if we were walking on the Swiss roll manifold in 2 dimensions.

IsoMap has been used effectively for dimensionality reduction in facial data.

Next time: Laplacian Eigenmaps, t-SNE.

Chapter 4

Gradient Descent, Newton's Method

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 θ. 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 parameters.

However, we may have need for models which are nonlinear function of their parameters. We consider a motivating example first.

Noisy Distance Readings

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 m 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

$$Y_i = \|X_i - \theta\| + N_i, \qquad N_i \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2), \qquad i = 1, \ldots, m$$

where

$$\|X_i - \theta\| = \sqrt{(X_{i1} - \theta_1)^{\mathbf{2}} + (X_{i2} - \theta_2)^{\mathbf{2}}}$$

Here our prediction is

$$\hat{y} = f(x; \theta) = \|x - \theta\|$$

which is clearly not linear in θ.

Formulation from MLE

More generally, let us assume a model similar to the one above, but where f is now some arbitrary differentiable function and $\theta \in \mathbb{R}^d$:

$$Y_i = f(X_i; \theta) + N_i, \qquad N_i \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2), \qquad i = 1, \ldots, m$$

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

$$\begin{aligned}
\hat{\theta}_{\text{MLE}} &= \arg\max_{\theta} \ \log P(y_1, \ldots, y_m \mid x_1, \ldots, x_m; \theta, \sigma) \\
&= \arg\max_{\theta} \ \log \prod_{i=1}^{m} P(y_i \mid x_i; \theta, \sigma) \\
&= \arg\max_{\theta} \ \sum_{i=1}^{m} \log P(y_i \mid x_i; \theta, \sigma) \\
&= \arg\max_{\theta} \ \sum_{i=1}^{m} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f(x_i;\theta))^2}{2\sigma^2}\right) \\
&= \arg\max_{\theta} \ \sum_{i=1}^{m} \left[-\frac{1}{2}\log\Big(2\pi\sigma^2\Big) - \frac{1}{2\sigma^2}(y_i - f(x_i;\theta))^2\right] \\
&= \arg\min_{\theta} \ \sum_{i=1}^{m} (y_i - f(x_i;\theta))^2
\end{aligned}$$

The last step holds because the first term in the sum is constant w.r.t. the optimization variable θ, and we flip from max to min because of the negative sign.

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 the method is called **nonlinear least squares**.

Unfortunately, there is no closed-form solution for $\hat{\theta}_{\text{MLE}}$ in general. Later we will see an iterative method for computing it.

Solutions to Nonlinear Least Squares

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

$$\min_{\theta} \epsilon_{LS}(\theta) = \min_{\theta} \sum_{i} (y_i - f(x_i;\theta))^2$$

One way to minimize a function is to use calculus. We know that the gradient of the objective function at any minimum must be zero, because if it isn't, we can take a sufficiently small step in the direction of the negative gradient that the objective function's value will be reduced.

Thus, the **first-order optimality condition** that needs to be satisfied is:

$$\nabla_{\theta}\epsilon_{LS} = 2\sum_{i} (y_i - f(x_i;\theta))\nabla_{\theta} f(x_i;\theta) = 0$$

In compact matrix notation:

$$J(\theta)^{\top}(Y - F(\theta)) = 0$$

where

$$F(\theta) = \begin{bmatrix} f(x_1;\theta) \\ \vdots \\ f(x_n;\theta) \end{bmatrix}$$

$$J(\theta) = \begin{bmatrix} \nabla_\theta f(x_1;\theta)^\top \\ \vdots \\ \nabla_\theta f(x_n;\theta)^\top \end{bmatrix} = \nabla_\theta F, \text{ the } \textbf{Jacobian} \text{ of F}$$

Observe that when f is linear in θ (i.e. $f(x_i;\theta) = \theta^\top x_i$), the gradient $\nabla_\theta \epsilon_{LS}$ will only have θ in one place because the term $\nabla_\theta f(x_i;\theta)$ will only depend on x_i:

$$\nabla_\theta \epsilon_{LS} = 2\sum_i (y_i - \theta^\top x_i)\nabla_\theta(\theta^\top x_i) = 2\sum_i (y_i - \theta^\top x_i)x_i$$

and it is easy to derive a closed-form solution for θ in terms of the y_i's and x_i's:

$$\begin{aligned} 2X^\top(Y - X\theta) &= 0 \\ 2X^\top Y - 2X^\top X\theta &= 0 \\ X^\top Y &= X^\top X\theta \\ \theta &= (X^\top X)^{-1}X^\top Y \end{aligned}$$

It's just OLS!

If, however, f were not linear in θ, the term $\nabla_\theta f(x_i;\theta)$ would contain more θ terms (since differentiating once wouldn't be enough to make them go away), and it would not be possible to write out a closed-form solution for θ.

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 θ, 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.

The Gauss-Newton algorithm

Since there is no closed-form solution to the nonlinear least squares optimization problem, we resort to an iterative algorithm, the **Gauss-Newton algorithm**[1], to tackle it. 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 θ at iteration k, which we denote $\theta^{(k)}$. We can then approximate $F(\theta)$ to first order using a Taylor expansion about $\theta^{(k)}$:

$$\begin{aligned} F(\theta) \approx \tilde{F}(\theta) &:= F(\theta^{(k)}) + \nabla_\theta F(\theta^{(k)})(\theta - \theta^{(k)}) \\ &= F(\theta^{(k)}) + J(\theta^{(k)})\Delta\theta \end{aligned}$$

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

[1] For some reason this algorithm was called gradient descent in lecture, but it is not really gradient descent. However, like gradient descent, it is an iterative, first-order optimization algorithm. Another popular method for solving nonlinear least squares, the **Levenberg-Marquardt algorithm**, is a sort of interpolation between Gauss-Newton and gradient descent.

Now since $\tilde{F}$ 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 from earlier to the objective $\tilde{F}$ yields the following equation:

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

Note that we have used the fact that the Jacobian of the linearized function $\tilde{F}$, evaluated at any θ, is precisely $J(\theta^{(k)})$. This is because $\tilde{F}$ is affine where the linear map is $J(\theta^{(k)})$, so the best linear approximation is just that.

Writing $J = J(\theta^{(k)})$ for brevity, we have

$$\begin{aligned} J^\top Y &= J^\top (F(\theta^{(k)}) + J\Delta\theta) \\ J^\top (Y - F(\theta^{(k)})) &= J^\top J(\Delta\theta) \\ \Delta\theta &= (J^\top J)^{-1} J^\top (Y - F(\theta^{(k)})) \\ &= (J^\top J)^{-1} J^\top \Delta Y \end{aligned}$$

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 $\delta F \approx J\delta\theta$ 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

$$\begin{aligned} \theta^{(k+1)} &= \theta^{(k)} + \Delta\theta \\ &= \theta^{(k)} + (J^\top J)^{-1} J^\top \Delta Y \end{aligned}$$

Here's the entire process laid out in steps:

1. Initialize $\theta^{(0)}$ with some guess
2. Repeat until convergence:
   (a) Compute Jacobian with respect to the current iterate, $J = J(\theta^{(k)})$
   (b) Compute $\Delta Y = Y - F(\theta^{(k)})$
   (c) Update: $\theta^{(k+1)} = \theta^{(k)} + (J^\top J)^{-1} J^\top \Delta Y$

Note that the solution found will depend on the initial value $\theta^{(0)}$ in general.

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

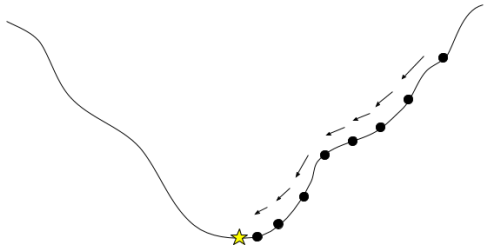
$$\left| \frac{\epsilon^{(k+1)} - \epsilon^{(k)}}{\epsilon^{(k)}} \right| \leq \text{threshold}$$

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

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha_i \nabla f(\mathbf{x}_i)$$

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

Chapter 5

Neural Networks

5.1 Neural Networks

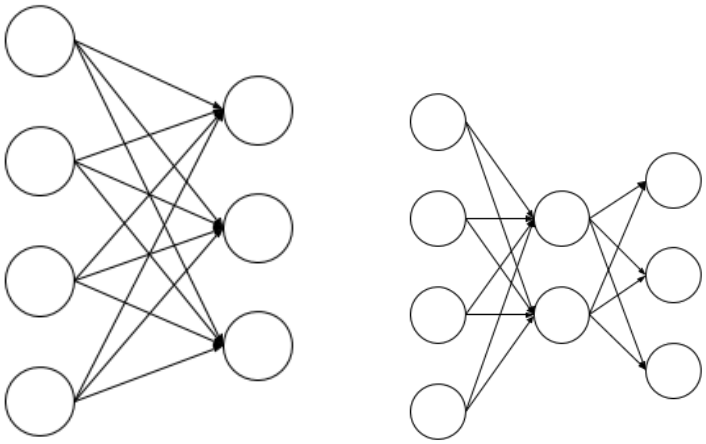
Neural networks are a class of compositional function approximators. Unlike other function approximators we have seen (e.g. polynomials), they are nonlinear in their parameters.

(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:

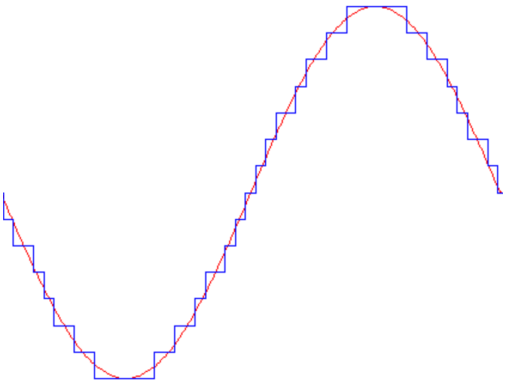


These circuits represent matrix multiplications, with the weights on the edges being the entries of the matrix. Assuming that the flow of information is from left to right, the circuit on the left is multiplication by a 3×4 matrix, and the circuit on the right is multiplication by a 2×4 matrix followed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

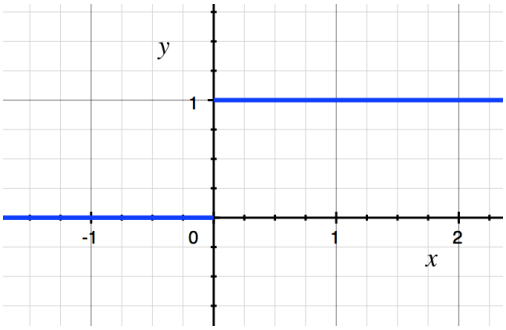
of processing, so it seems like it should be more expressive. The key thing that is missing is *nonlinearity*.

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.[1] A key observation is that piecewise-constant functions are universal function approximators:



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

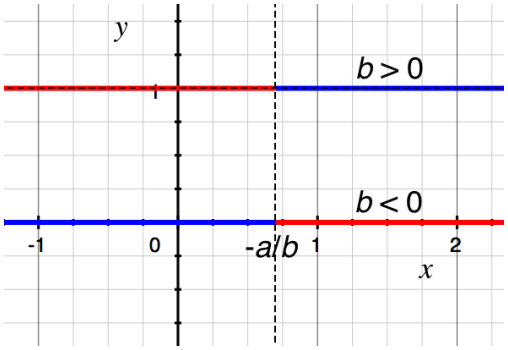
$$u(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$



We can build very complicated functions from this simple step function by combining translated and scaled versions of it. Observe that

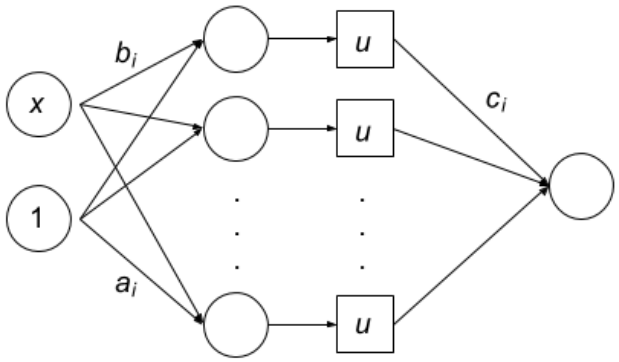
- If $a, b \in \mathbb{R}$, the function $x \mapsto u(a + bx)$ is a translated (and, depending on the sign of b, possibly flipped) version of the step function:

[1] 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.



- 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

$$\min_{\mathbf{a},\mathbf{b},\mathbf{c}} \underbrace{\sum_{k=1}^{n} e_k}_{f(\mathbf{a},\mathbf{b},\mathbf{c})}$$

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

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

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:

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

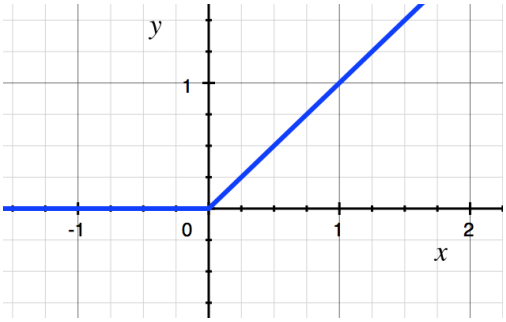
Similarly,

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

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:

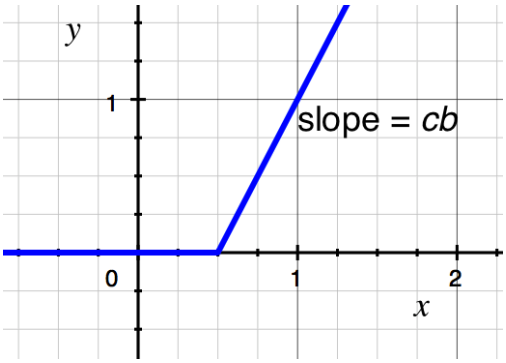
$$\frac{\partial f}{\partial c_i} = \sum_{k=1}^{n} -2(y_k - h(x_k)) \max(0, a_i + b_i x)$$

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

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

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, $u(x) = .01x$ when $x < 0$) are sometimes used to provide some small gradient signal to avoid dead neurons.

Neural networks are universal function approximators

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

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

$$h(\mathbf{x}) = \sum_{i=1}^{d} c_i u(a_i + \mathbf{b}_i^\top \mathbf{x})$$

[2] See *Understanding the difficulty of training deep feedforward neural networks.*
[3] See *Approximation Capabilities of Multilayer Feedforward Networks.*
[4] Both ReLU and sigmoid satisfy these requirements.

such that

$$|h(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

for all $\mathbf{x} \in S$.

There's some subtlety in the theorem that's worth noting. It says that for any given continuous function, there exists a neural network of finite size that uniformly approximates the given function. However, it says nothing about how well any particular architecture you're considering will approximate the function. It also doesn't tell us how to compute the weights.

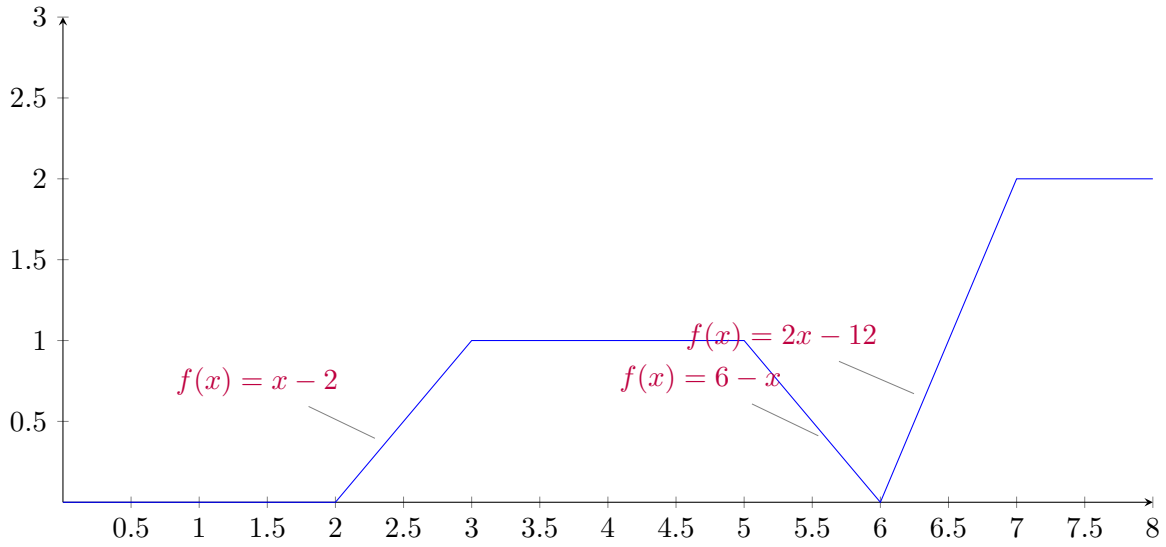
It's also worth pointing out that in the theorem, the network consists of just one hidden layer. In practice, people find that using more layers works better.

5.2 Training Neural Networks

ReLUs 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:



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):

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

Here's the plot on Wolfram Alpha.

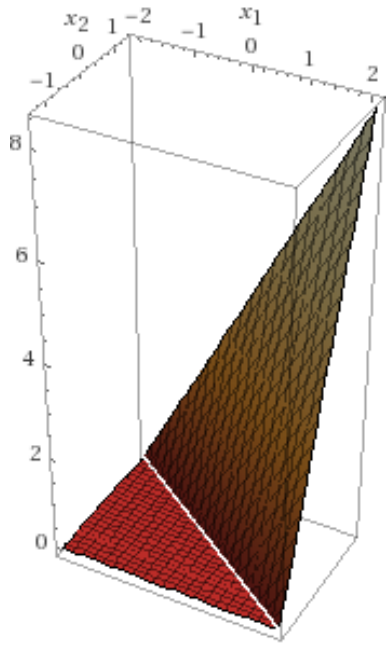
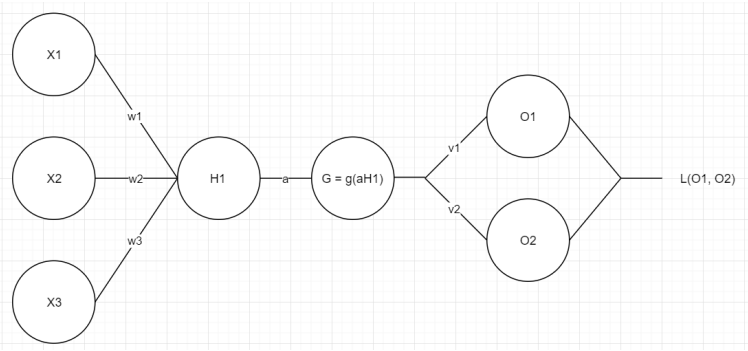


Figure 5.1: $f(x_1, x_2) = \max(0, 2x_1 + 3x_2)$

In higher dimensions, i.e. when a ReLU takes in an arbitrarily long dot-product as input: $f(x) = \max(0, w^\top x)$, the unit can be viewed as representative of a "ramp" in the higher-dimensional space. Here's a plot of a 3-D ramp:

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:

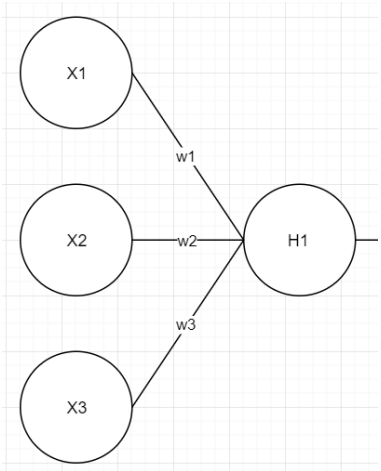


The hypothesis function that this neural net encodes is represented by the two outputs, $O = [O_1, O_2]$.

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

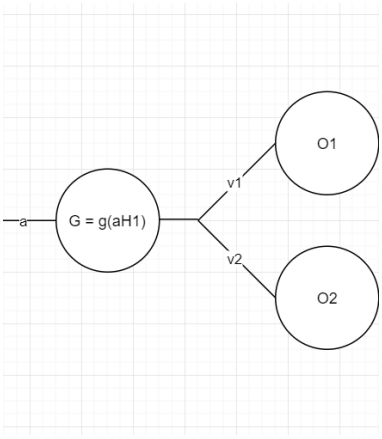
This section is an aside meant to recall your knowledge about the chain rule in multivariable calculus. Let's take a look at two slices of our neural net from above.



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:

$$\begin{aligned}
\frac{\partial L}{\partial v_1} &= \frac{\partial L}{\partial O_1}\frac{\partial O_1}{\partial v_1} \\
\frac{\partial L}{\partial a} &= \frac{\partial L}{\partial O_1}\frac{\partial O_1}{\partial a} + \frac{\partial L}{\partial O_2}\frac{\partial O_2}{\partial a} \\
\frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial O_1}\frac{\partial O_1}{\partial w_1} + \frac{\partial L}{\partial O_2}\frac{\partial O_2}{\partial w_1}
\end{aligned}$$

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.

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 denote the true optimum by w^*. Our goal is to show the following:

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

We make the following assumptions in the analysis below:

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

$$\begin{aligned}
&\mathbb{E}[\|w^{(k+1)} - w^*\|^2] \\
&= \mathbb{E}[(w^{(k+1)} - w^*)^\top (w^{(k+1)} - w^*)] \\
&= \mathbb{E}[(w^{(k)} - w^* - \alpha_k G_k)^\top (w^{(k)} - w^* - \alpha_k G_k)] \\
&= \mathbb{E}[\|w^{(k)} - w^*\|^2] - 2\alpha_k \mathbb{E}[(w^{(k)} - w^*)^\top G_k] + \alpha_k^2 \mathbb{E}[\|G_k\|^2]
\end{aligned}$$

For brevity, define

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

Our assumption on the expected squared norm of G_k implies

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

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

$$\mathbb{E}[(w^{(k)} - w^*)^\top G_k] = \mathbb{E}_{\text{past}}\left[\mathbb{E}[(w^{(k)} - w^*)^\top G_k \mid \text{past}]\right]$$

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

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

The current gradient G_k depends on our new choice of evaluation point, which is presumed independent of the past and thus $\mathbb{E}[G_k \mid \text{past}] = \mathbb{E}[G_k] = \nabla f(w^{(k)})$, 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

$$\mathbb{E}[(w^{(k)} - w^*)^\top G_k] = \mathbb{E}_{\text{past}}[(w^{(k)} - w^*)^\top \nabla f(w^{(k)})]$$

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

$$(w^{(k)} - w^*)^\top \nabla f(w^{(k)}) = (w^{(k)} - w^*)^\top (\nabla f(w^{(k)}) - \nabla f(w^*)) \geq \ell \|w^{(k)} - w^*\|_2^2$$

Taking expectations yields

$$\mathbb{E}[(w^{(k)} - w^*)^\top \nabla f(w^{(k)})] \geq \ell \mathbb{E}[\|w^{(k)} - w^*\|_2^2] = \ell d_k$$

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

$$d_{k+1} \leq d_k - 2\alpha_k \ell d_k + \alpha_k^2 M^2 = (1 - 2\alpha_k \ell) d_k + \alpha_k^2 M^2$$

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 α_k over time – but decreasing α_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:

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

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 6

Classification

6.1 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.

Here's a formal definition: Given a training set $\mathcal{D} = \{(\mathbf{x}_i, t_i)\}_{i=1}^n$ of n points, with each data point $\mathbf{x}_i \in \mathbb{R}^d$ paired with a known discrete class label $t_i \in \{1, 2, ..., K\}$, train a classifier which, when fed any arbitrary d-dimensional data point, classifies that data point as one of the K discrete classes.

Classifiers have strong roots in probabilistic modeling. The idea is that given an arbitrary datapoint $\mathbf{x}$, we classify $\mathbf{x}$ with the class label k^* that maximizes the posterior probability of the class label given the data point:

$$k^* = \arg\max_k P(\text{class} = k|\mathbf{x})$$

Consider the example of digit classification. Suppose we are given dataset of images of handwritten digits each with known values in the range $\{0, 1, 2, \ldots, 9\}$. The task is, given an image of a handwritten digit, to classify it to the correct digit. A generic classifier for this this task would effectively form a posterior probability distribution over the 10 possible digits and choose the digit that achieves the maximum posterior probability:

$$k^* = \arg\max_{k \in \{0,1,2...,9\}} P(\text{digit} = k|\text{image})$$

There are two main types of models that can be used to train classifiers: **generative models** and **discriminative models**.

6.2 Generative Models

Generative models involve explicitly forming:

1. A prior probability distribution over all classes $k \in \{1, 2, \ldots, K\}$

$$P(k) = P(\text{class} = k)$$

2. A conditional probability distribution for each class k

$$f_k(\mathbf{x}) = f(\mathbf{x}|\text{class } k)$$

In total there are $K+1$ probability distributions: 1 for the prior, and K for all of the individual classes. Note that the prior probability distribution is a categorical distribution over the K discrete classes, whereas each class conditional probability distribution is a continuous distribution over $\mathbb{R}^d$ (often represented as a Gaussian). Using the prior and the conditional distributions in conjunction, we conclude (from Bayes' rule) that we are effectively solving

$$k^* = \arg\max_k P(\text{class} = k|\mathbf{x}) = \arg\max_k \frac{P(k)\ f_k(\mathbf{x})}{f(\mathbf{x})} = \arg\max_k P(k)\ f_k(\mathbf{x})$$

In the case of the digit classification, we are solving for

$$k^* = \arg\max_{k\in\{0,1,2...,9\}} P(\text{digit} = k)\ f(\text{image}|\text{digit} = k)$$

6.3 QDA Classification

Quadratic Discriminant Analysis (QDA) is a specific generative method in which the class conditional probability distributions are independent Gaussians: $f_k(.) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$.

Note: the term "discriminant" in QDA is misleading: remember that QDA is not a discriminative method, it is a generative method!

Estimating $f_k(.)$

For a particular class conditional probability distribution $f_k(.)$, if we do not have the true means and covariances $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$, then our best bet is to estimate them empirically with the samples in our training data that are classified as class k:

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

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

is to solve the following MLE problem:

$$\begin{aligned}
\hat{\mu_k}, \hat{\sigma_k}^2 &= \arg\max_{\mu_k, \sigma_k^2} P(x_1, x_2, ..., x_{n_k} | \mu_k, \sigma_k^2) \\
&= \arg\max_{\mu_k, \sigma_k^2} \ln\Big(P(x_1, x_2, ..., x_{n_k} | \mu_k, \sigma_k^2)\Big) \\
&= \arg\max_{\mu_k, \sigma_k^2} \sum_{i=1}^{n_k} \ln\Big(P(x_i | \mu_k, \sigma_k^2)\Big) \\
&= \arg\max_{\mu_k, \sigma_k^2} \sum_{i=1}^{n_k} -\frac{(x_i - \mu_k)^2}{2\sigma_k^2} - \ln(\sigma_k) - \frac{1}{2}\ln(2\pi) \\
&= \arg\min_{\mu_k, \sigma_k^2} \sum_{i=1}^{n_k} \frac{(x_i - \mu_k)^2}{2\sigma_k^2} + \ln(\sigma_k)
\end{aligned}$$

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

$$\min_{\mu_k, \sigma_k^2} \sum_{i=1}^{n_k} \frac{(x_i - \mu_k)^2}{2\sigma_k^2} + \ln(\sigma_k) = \min_{\sigma_k^2} \min_{\mu_k} \sum_{i=1}^{n_k} \frac{(x_i - \mu_k)^2}{2\sigma_k^2} + \ln(\sigma_k)$$

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

$$\frac{\partial}{\partial \mu_k}\Big(\sum_{i=1}^{n_k} \frac{(x_i - \mu_k)^2}{2\sigma_k^2} + \ln(\sigma_k)\Big) = \sum_{i=1}^{n_k} \frac{-(x_i - \mu_k)}{\sigma_k^2} = 0 \implies \hat{\mu_k} = \frac{1}{n_k}\sum_{i=1}^{n_k} x_i$$

Having solved the inner optimization problem, we now have that

$$\min_{\sigma_k^2} \min_{\mu_k} \sum_{i=1}^{n_k} \frac{(x_i - \mu_k)^2}{2\sigma_k^2} + \ln(\sigma_k) = \min_{\sigma_k^2} \sum_{i=1}^{n_k} \frac{(x_i - \hat{\mu_k})^2}{2\sigma_k^2} + \ln(\sigma_k)$$

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

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

$$\lim_{\sigma_k \to 0} \sum_{i=1}^{n_k} \frac{(x_i - \hat{\mu_k})^2}{2\sigma_k^2} + \ln(\sigma_k) = \infty$$

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

$$\lim_{\sigma_k \to \infty} \sum_{i=1}^{n_k} \frac{(x_i - \hat{\mu_k})^2}{2\sigma_k^2} + \ln(\sigma_k) = \infty$$

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

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

We conclude that the optimal point is

$$\hat{\sigma_k}^2=\frac{1}{n_k}\sum_{i=1}^{n_k}(x_i-\hat{\mu_k})^2$$

QDA Optimization Formulation

Assuming that we know the means and covariances for all the classes, we can use Bayes' Rule to directly solve the optimization problem

$$\begin{aligned}
k^* &= \arg\max_k P(k)\ f_k(\mathbf{x}) \\
&= \arg\max_k (\sqrt{2\pi})^d P(k)\ f_k(\mathbf{x}) \\
&= \arg\max_k \ln\big(P(k)\big)+\ln\Big((\sqrt{2\pi})^d f_k(\mathbf{x})\Big) \\
&= \arg\max_k \ln\big(P(k)\big)-\frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}}_k)^T\hat{\boldsymbol{\Sigma}}_k^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}}_k)-\frac{1}{2}\ln\Big(|\hat{\boldsymbol{\Sigma}}_k|\Big)=Q_k(\mathbf{x})
\end{aligned}$$

For future reference, let's use $Q_k(\mathbf{x})=\ln\Big(\sqrt{2\pi}\Big)^d P(k)\ f_k(\mathbf{x}))$ to simplify our notation.

6.4 LDA Classification

While QDA is a reasonable approach to classification, we might be interested in simplifying our model to reduce the number of parameters we have to learn. One way to do this is through **Linear Discriminant Analysis (LDA)** classification. Just as in QDA, LDA assumes that the class conditional probability distributions are normally distributed with different means $\boldsymbol{\mu}_k$, but LDA is different from QDA in that it requires all of the distributions to share the same covariance matrix $\boldsymbol{\Sigma}$. This is a simplification which, in the context of the Bias-Variance tradeoff, increases the bias of our method but may help decrease the variance.

Estimating $f_k(.)$

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

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

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 $\mathbf{x}_i$'s by specifying

a class k and the index within that class j:

$$\begin{aligned}\hat{\mathbf{\Sigma}} &= \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{t_i})(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_{t_i})^T \\ &= \frac{1}{n}\sum_{k=1}^{K}\sum_{j=1}^{n_k}(\mathbf{x}_{j,k} - \hat{\boldsymbol{\mu}}_k)(\mathbf{x}_{j,k} - \hat{\boldsymbol{\mu}}_k)^T \\ &= \frac{1}{n}\sum_{k=1}^{K} n_k \ \mathbf{\Sigma}_k \\ &= \sum_{k=1}^{K}\frac{n_k}{n}\mathbf{\Sigma}_k\end{aligned}$$

6.5 LDA vs. QDA: Differences and Decision Boundaries

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 $\mathbf{x}$-space. Given any two classes, the decision boundary represents the points in $\mathbf{x}$-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 $\mathbf{x}$ can be classified according to three cases:

$$k^* = \begin{cases} A & P(\text{class} = A|\mathbf{x}) > P(\text{class} = B|\mathbf{x}) \\ B & P(\text{class} = A|\mathbf{x}) < P(\text{class} = B|\mathbf{x}) \\ \text{Either } A \text{ or } B & P(\text{class} = A|\mathbf{x}) = P(\text{class} = B|\mathbf{x}) \end{cases}$$

The decision boundary is the set of all points in $\mathbf{x}$-space that are classified according to the third case. Let's look at the form of the decision boundary according to the different scenarios possible under QDA and LDA.

Identical Isotropic Gaussian Distributions

The simplest case is when the two classes are equally likely in prior, and their conditional probability distributions are isotropic with identical covariances. **Isotropic** Gaussian distributions have covariances of the form of $\mathbf{\Sigma} = \sigma^2\mathbf{I}$, which means that their isocontours are circles. In this case, $f_A(.)$ and $f_B(.)$ have identical covariances of the form $\mathbf{\Sigma}_A = \mathbf{\Sigma}_B = \sigma^2\mathbf{I}$.

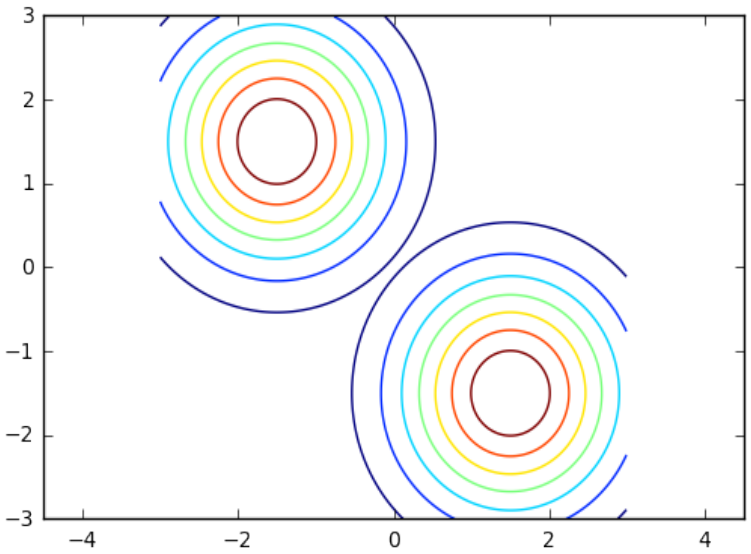


Figure 6.1: Contour plot of two isotropic, identically distributed Gaussians in $\mathbb{R}^2$. The circles are the level sets of the Gaussians.

Geometrically, we can see that the task of classifying a 2-D point into one of the two classes amounts simply to figuring out which of the means it's closer to. Using our notation of $Q_k(\mathbf{x})$ from before, this can be expressed mathematically as:

$$\begin{aligned} Q_A(\mathbf{x}) &= Q_B(\mathbf{x}) \\ \ln\left(\frac{1}{2}\right) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}_A})^T\sigma^2\mathbf{I}(\mathbf{x}-\hat{\boldsymbol{\mu}_A}) - \frac{1}{2}\ln\Big(|\sigma^2\mathbf{I}|\Big) &= \ln\left(\frac{1}{2}\right) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}_B})^T\sigma^2\mathbf{I}(\mathbf{x}-\hat{\boldsymbol{\mu}_B}) - \frac{1}{2}\ln\Big(|\sigma^2\mathbf{I}|\Big) \\ (\mathbf{x}-\hat{\boldsymbol{\mu}_A})^T(\mathbf{x}-\hat{\boldsymbol{\mu}_A}) &= (\mathbf{x}-\hat{\boldsymbol{\mu}_B})^T(\mathbf{x}-\hat{\boldsymbol{\mu}_B}) \end{aligned}$$

The decision boundary is the set of points $\mathbf{x}$ for which $||\mathbf{x}-\hat{\boldsymbol{\mu}_A}||_2 = ||\mathbf{x}-\hat{\boldsymbol{\mu}_B}||_2$, which is simply the set of points that are equidistant from $\boldsymbol{\mu}_A$ and $\boldsymbol{\mu}_B$. This decision boundary is linear because the set of points that are equidistant from $\boldsymbol{\mu}_A$ and $\boldsymbol{\mu}_B$ are simply the perpendicular bisector of the segment connecting $\boldsymbol{\mu}_A$ and $\boldsymbol{\mu}_B$.

Identical Anisotropic Gaussian Distributions

The next case is when the two classes are equally likely in prior, and their conditional probability distributions are anisotropic with identical covariances. **Anisotropic** Gaussian distributions are simply all Gaussian distributions that are not isotopic.

In order to understand the difference, let's take a closer look at the covariance matrix $\boldsymbol{\Sigma}$. Since $\boldsymbol{\Sigma}$ is a symmetric, positive semidefinite matrix, we can decompose it by the spectral theorem into $\boldsymbol{\Sigma} = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^T$, where the columns of $\mathbf{V}$ form an orthonormal basis in $\mathbb{R}^d$, and $\boldsymbol{\Lambda}$ is a diagonal matrix with real, non-negative values. The entries of $\boldsymbol{\Lambda}$ dictate how elongated or shrunk the distribution is along each direction. To see why this is the case, let's consider a zero-mean Gaussian distribution $\mathcal{N}(0, \boldsymbol{\Sigma})$. We wish to find its **level set** $f(\mathbf{x}) = k$, or simply the set of all points $\mathbf{x}$ such that the probability density $f(\mathbf{x})$ evaluates to a fixed constant k. This is equivalent to the level set $\ln\big(f(\mathbf{x})\big) = \ln(k)$, which further reduces to $\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\mathbf{x} = c$, for some constant c. Without loss of generality, assume that this constant is 1. The level set $\mathbf{x}^T\boldsymbol{\Sigma}^{-1}\mathbf{x} = 1$ is an ellipsoid with axes

$\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d$, with lengths $\sqrt{\lambda_1}, \sqrt{\lambda_2}, \ldots, \sqrt{\lambda_d}$, respectively. Each axis of the ellipsoid is the vector $\sqrt{\lambda_i}\mathbf{v}_i$, and we can verify that

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

In the case of isotropic distributions, the entries of $\mathbf{\Lambda}$ are all identical, meaning the the axes of the ellipsoid form a circle. In the case of anisotropic distributions, the entries of $\mathbf{\Lambda}$ are not necessarily identical, meaning that the resulting ellipsoid may be elongated/shruken and also rotated.

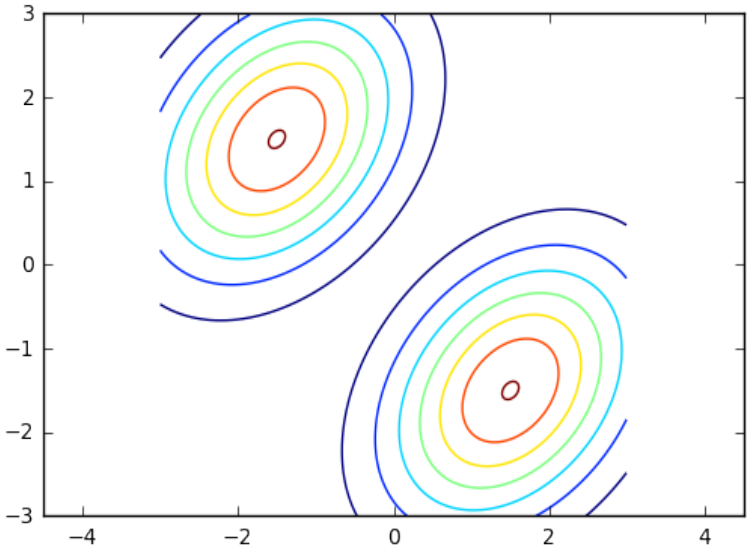


Figure 6.2: Two anisotropic, identically distributed Gaussians in $\mathbb{R}^2$. The ellipses are the level sets of the Gaussians.

The case when the two classes are identical anisotropic distributions can be reduced to the isotropic case simply by performing a change of coordinates that transforms the ellipses back into circles. Thus, the decision boundary is still linear.

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:

$$\begin{aligned}
Q_A(\mathbf{x}) &= Q_B(\mathbf{x}) \\
\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}_A})^T\hat{\mathbf{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}_A}) - \frac{1}{2}\ln\Big(|\hat{\mathbf{\Sigma}}|\Big) &= \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}_B})^T\hat{\mathbf{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}_B}) - \frac{1}{2}\ln\Big(|\hat{\mathbf{\Sigma}}|\Big) \\
\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}_A})^T\hat{\mathbf{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}_A}) &= \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x}-\hat{\boldsymbol{\mu}_B})^T\hat{\mathbf{\Sigma}}^{-1}(\mathbf{x}-\hat{\boldsymbol{\mu}_B}) \\
2\ln\big(P(A)\big) - \mathbf{x}^T\hat{\mathbf{\Sigma}}^{-1}\mathbf{x} + 2\mathbf{x}^T\hat{\mathbf{\Sigma}}^{-1}\hat{\boldsymbol{\mu}_A} - \hat{\boldsymbol{\mu}_A}^T\hat{\boldsymbol{\mu}_A} &= 2\ln\big(P(B)\big) - \mathbf{x}^T\hat{\mathbf{\Sigma}}^{-1}\mathbf{x} + 2\mathbf{x}^T\hat{\mathbf{\Sigma}}^{-1}\hat{\boldsymbol{\mu}_B} - \hat{\boldsymbol{\mu}_B}^T\hat{\boldsymbol{\mu}_B} \\
2\ln\big(P(A)\big) + 2\mathbf{x}^T\hat{\mathbf{\Sigma}}^{-1}\hat{\boldsymbol{\mu}_A} - \hat{\boldsymbol{\mu}_A}^T\hat{\boldsymbol{\mu}_A} &= 2\ln\big(P(B)\big) + 2\mathbf{x}^T\hat{\mathbf{\Sigma}}^{-1}\hat{\boldsymbol{\mu}_B} - \hat{\boldsymbol{\mu}_B}^T\hat{\boldsymbol{\mu}_B}
\end{aligned}$$

Simplifying, we have that

$$\mathbf{x}^T(\hat{\mathbf{\Sigma}}^{-1}(\hat{\boldsymbol{\mu}_A} - \hat{\boldsymbol{\mu}_B})) + \Big(\ln\big(\frac{P(A)}{P(B)}\big) - \frac{\hat{\boldsymbol{\mu}_A}^T\hat{\boldsymbol{\mu}_A} - \hat{\boldsymbol{\mu}_B}^T\hat{\boldsymbol{\mu}_B}}{2}\Big) = 0$$

The decision boundary is the level set of a linear function $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + k$. 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

This is certainly *not* the case in LDA. We have that:

$$\ln\big(P(A)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}_A})^T \hat{\boldsymbol{\Sigma}_A}^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}_A}) = \ln\big(P(B)\big) - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}_B})^T \hat{\boldsymbol{\Sigma}_B}^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}_B})$$

Here, unlike the case when $\boldsymbol{\Sigma}_A = \boldsymbol{\Sigma}_B$, we *cannot* cancel out the quadratic terms in $\mathbf{x}$ from both sides of the equation, and thus our decision boundary is now represented by the level set of an arbitrary quadratic function.

It should now make sense why QDA is short for **quadratic** discriminant analysis and LDA is short for **linear** discriminant analysis!

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:

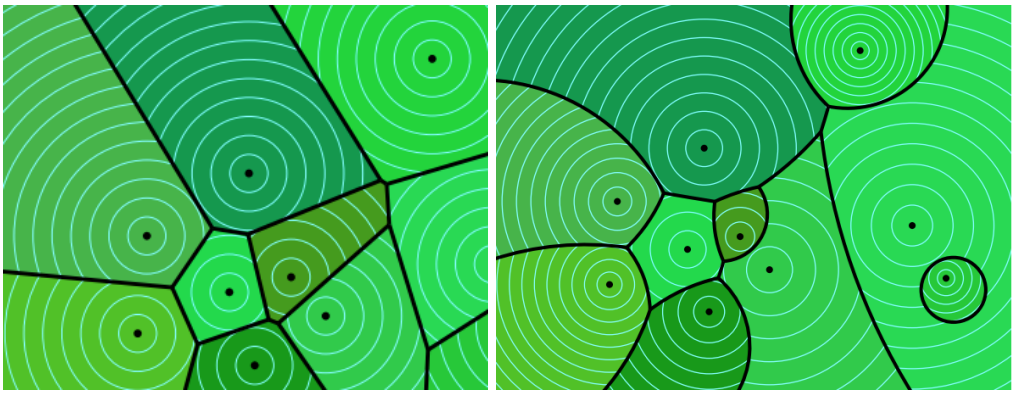


Figure 6.3: LDA (left) vs QDA (right): a collection of linear vs quadratic level set boundaries

6.6 Discriminative Models

In our discussion of LDA and QDA, we focused on **generative models**, where we explicitly model the probability of the data $P(\mathbf{X}, Y)$ and choose the class c^* that maximizes the posterior probability: $c^* = \arg\max_c P(Y = c|\mathbf{X})$. For example, in QDA we model $P(\mathbf{X}|Y = c)$ as a Gaussian with an estimated mean $\boldsymbol{\mu}_c$ and covariance matrix $\boldsymbol{\Sigma}_c$. If we choose a prior $P(Y)$, then our predicted class for a new test point $\mathbf{x}$ is:

$$c^* = \arg\max_c P(Y = c|\mathbf{X} = \mathbf{x}) = \arg\max_c P(\mathbf{X} = \mathbf{x}|Y = c)P(Y = c)$$

The generative approach is flexible and we can actually use our model to generate new samples. However, LDA and QDA can be inefficient in that they require estimation of a large number of parameters (ie. the covariance matrices, which have $\frac{d(d-1)}{2}$ parameters). For 2-class LDA, the decision boundary is of the form

$$\mathbf{w}^T\mathbf{x} + k = 0$$

where $k, \mathbf{w}$ are estimated parameters. The decision boundary only requires $d+1$ parameters, but we ended up estimating $O(d^2)$ parameters because we needed to determine the class-conditional Gaussian generative models. LDA is an indirect approach to classification - it estimates parameters for $P(\mathbf{X}|Y)$ and use Bayes' rule to compute a decision rule, which leads to the discrepancy between the number of parameters required to specify the generative model and the number of parameters required to perform classification.

This leads us to the concept of **discriminative models**, where we bypass learning a generative model altogether and directly learn a decision boundary for the purpose of classification. The process of constructing a discriminative model is composed of two key design choices:

1) Representation: how we represent the output of the model (for example, the output of a model could be any real-valued number that we threshold to perform classification, or the output could represent a probability)

2) Loss function: how we train and penalize errors

6.7 Least Squares Support Vector Machine

Model and Training

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

We need to figure out how to optimize the parameter $\mathbf{w}$. One simple procedure we can try is to fit a least squares objective:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \text{sign}(\mathbf{w}^T\mathbf{x}_i)\|^2 + \lambda\|\mathbf{w}\|^2$$

Where $\mathbf{x}_i, \mathbf{w} \in \mathbb{R}^{d+1}$. Note that we have not forgotten about the bias term! Even though we are dealing with d dimensional data, $\mathbf{x}_i$ and $\mathbf{w}$ are $d+1$ dimensional: we add an extra "feature" of 1 to $\mathbf{x}$, and a corresponding bias term of k in $\mathbf{w}$. Note that in practice, we do not want to penalize the bias term in the regularization term, because the we should be able to work with any affine transformation of the data and still end up with the same decision boundary. Therefore, rather than taking the norm of $\mathbf{w}$, we often take the norm of $\mathbf{w}'$, which is every term of $\mathbf{w}$ excluding the corresponding bias term. For simplicity of notation however, let's just take the norm of $\mathbf{w}$.

Without the regularization term, this would be equivalent to minimizing the number of misclassified training points. Unfortunately, optimizing this is NP-hard (computationally intractable). However,

we can solve a relaxed version of this problem:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \mathbf{w}^T \mathbf{x}_i\|^2 + \lambda \|\mathbf{w}\|^2$$

This method is called the *2-class least squares support vector machine* (LS-SVM). Note that in this relaxed version, we care about the magnitude of $\mathbf{w}^T \mathbf{x}_i$ and not just the sign.

One drawback of LS-SVM is that the hyperplane decision boundary it computes does not necessarily make sense for the sake of classification. For example, consider the following set of data points, color-coded according to the class:

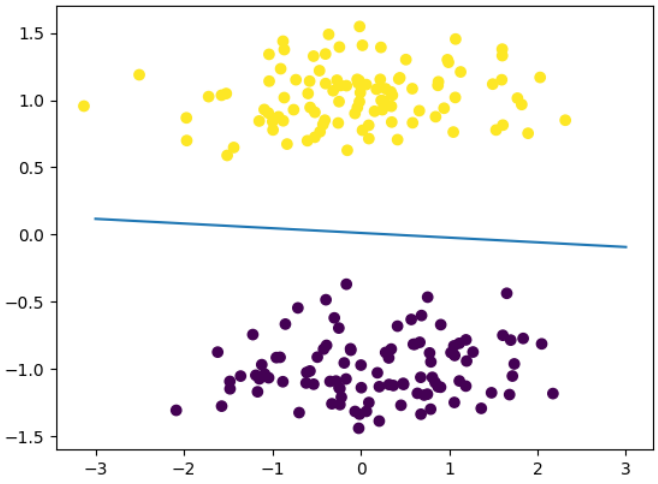


Figure 6.4: Reasonable fit LS-SVM

LS-SVM will classify every data point correctly, since all the $+1$ points lie on one side of the decision boundary and all the -1 points lie on the other side. Now if we add another cluster of points as follows:

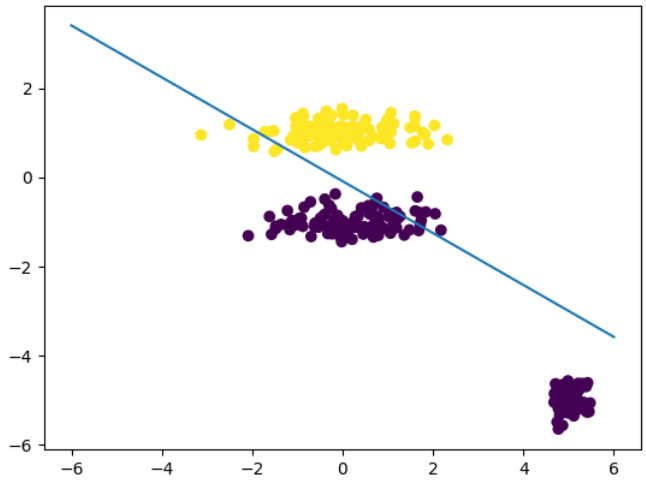


Figure 6.5: Poor fit LS-SVM

The original LS-SVM fit would still have classified every point correctly, but now the LS-SVM gets confused and decides that the points at the bottom right are contributing too much to the loss (perhaps for these points, $\mathbf{w}^T\mathbf{x}_i = -5$ 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 that allow us to come up with nonlinear classifiers (note that we are still working with linear classifiers in the augmented feature space). For example, adding quadratic features allows us to find a linear decision boundary in the augmented quadratic space that corresponds to a nonlinear "circle" decision boundary projected down into the raw feature space.

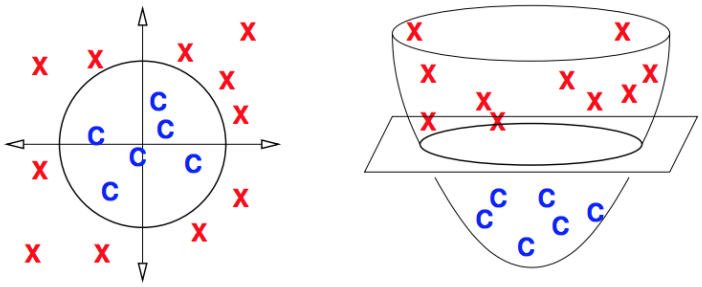


Figure 6.6: Augmenting Features, image courtesy of Prof. Shewchuk

In order implement this idea, we re-express our objective as

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - \mathbf{w}^T \phi(\mathbf{x}_i)\|^2 + \lambda \|\mathbf{w}'\|^2$$

Note that ϕ is a function that takes as input the data in raw feature space, and outputs the data in augmented feature space.

Neural Network Extension

Instead of using the linear function $\mathbf{w}^T\mathbf{x}$ or augmenting features to the data, we can also directly use a non-linear function of our choice in the original feature space, such as a neural network. One can imagine a whole family of discriminative binary classifiers that minimize

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - g_{\mathbf{w}}(\mathbf{x}_i)\|^2 + \lambda \|\mathbf{w}\|^2$$

where $g_{\mathbf{w}}(\mathbf{x}_i)$ can be any function that is easy to optimize. Then we can classify using the rule

$$\hat{y}_i = \begin{cases} 1 & g_{\mathbf{w}}(\mathbf{x}_i) > \theta \\ -1 & g_{\mathbf{w}}(\mathbf{x}_i) \leq \theta \end{cases}$$

Where θ is some threshold. In LS-SVM, $g_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{x}^T\mathbf{w}_i$ and $\theta = 0$. Using a neural network with non-linearities as $g_{\mathbf{w}}$ can produce complex, non-linear decision boundaries.

Multi-Class Extension

We can also adapt this approach to the case where we have multiple classes. Suppose there are K classes, labeled $1, 2, ..., K$. One possible way to extend the approach from binary classification is to compute $g_{\mathbf{w}}(\mathbf{x}_i)$ and round it to the nearest number from 1 to K. However, this approach gives an "ordering" to the classes, even if the classes themselves have no natural ordering. This is clearly a problem. For example, in fruit classification, suppose 1 is used to represent "peach," 2 is used to represent "banana," and 3 is used to represent "apple." In our numerical representation, it would appear that peaches are less than bananas, which are less than apples. As a result, if we have an image that looks like some cross between an apple and a peach, we may simply end up classifying it as a banana.

The typical way to get around this issue is as follows: if the i-th observation has class k, instead of using the representation $y_i = k$, we can use the representation $y_i = e_k$, 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:

$$\arg\min_{\mathbf{W}} \sum_{i=1}^{n} \|y_i - \mathbf{W}\mathbf{x}_i\|^2 + \lambda \|\mathbf{w}\|^2$$

To classify a data point, we compute $\mathbf{W}\mathbf{x}_i$ and see which component j is the largest - we then predict $\mathbf{x}_i$ to be in class j.

6.8 Logistic Regression

Logistic regression is a discriminative classification technique that has a direct probabilistic interpretation. 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 logitistic 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 logistic regression, we would like our model to output an estimate of the probability that a data point is in class 1. We can start with the linear function $\mathbf{w}^T\mathbf{x}$ and convert it to a number between 0 and 1 by applying a sigmoid transformation $s(\mathbf{w}^T\mathbf{x})$, where $s(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}}}$. Thus to classify $\mathbf{x}_i$ after learning the weights $\mathbf{w}$, we would estimate the probability as

$$P(y_i = 1|\mathbf{x}_i) = s(\mathbf{w}^T\mathbf{x}_i)$$

and classify $\mathbf{x}_i$ as

$$\hat{y}_i = \begin{cases} 1 & \text{if } s(\mathbf{w}^T\mathbf{x}_i) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

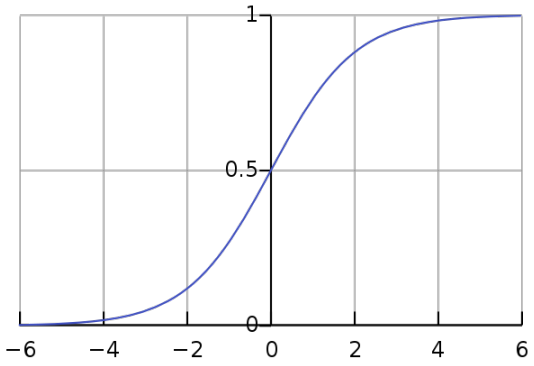


Figure 6.7: Logistic function. For our purposes, the horizontal axis is the output of the linear function $\mathbf{w}^T\mathbf{x}_i$ and the vertical axis is the output of the logistic function, which can be interpreted as a probability between 0 and 1.

The classifier equivalently classifies $\mathbf{x}_i$ as

$$\hat{y}_i = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x}_i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

In order to train our model, we need a loss function to optimize. One possibility is least squares:

$$\arg\min_{\mathbf{w}} \sum_{i=1}^{n} \|y_i - s(\mathbf{w}^T\mathbf{x}_i)\|^2 + \lambda\|\mathbf{w}\|^2$$

However, this may not be the best choice. Ordinary least squares regression has theoretical justifications such as being the maximum likelihood objective under Gaussian noise. Least squares for this classification problem does not have a similar justification.

Instead, the loss function we use for logistic regression is called the log-loss, or **cross entropy**:

$$L(\mathbf{w}) = \sum_{i=1}^{n} y_i \ln\left(\frac{1}{s(\mathbf{w}^T\mathbf{x}_i)}\right) + (1 - y_i)\ln\left(\frac{1}{1 - s(\mathbf{w}^T\mathbf{x}_i)}\right)$$

If we define $p_i = s(\mathbf{w}^T \mathbf{x}_i)$, then using the properties of logs we can write this as

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

For each $\mathbf{x}_i$, p_i represents our predicted probability that its corresponding class is 1. Because $y_i \in \{0, 1\}$, the loss corresponding to the i-th data point is

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

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.

Logistic Loss: Maximum Likelihood Approach

Logistic regression can be derived from a maximum likelihood model. Suppose we observe $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, and each observation is sampled from a distribution $Y_i \sim Bernoulli(p_i)$, where p_i is a function of x_i. We can view $P(Y_i = 1) = P(Y = 1|\mathbf{x}_i)$ as the posterior probability that $\mathbf{x}_i$ is classified as 1, and similarly $P(Y_i = 0)$ as the posterior probability that x_i is classified as 0. The observation y_i is simply sampled from this posterior probability distribution Y_i. Thus our observation y_i, which we can view as a "sample," has probability:

$$P(Y_i = y_i) = \begin{cases} p_i & \text{if } y_i = 1 \\ 1 - p_i & \text{if } y_i = 0 \end{cases}$$

One convenient way to write the likelihood of a single data point is

$$P(Y_i = y_i) = p_i^{y_i}(1 - p_i)^{(1-y_i)}$$

which holds no matter what y_i is.

We need a model for the dependency of p_i on $\mathbf{x}_i$. We have to enforce that p_i is a transformation of $\mathbf{x}_i$ that results in a number from 0 to 1 (ie. a valid probability). Hence p_i cannot be, say, linear in $\mathbf{x}_i$. One way to do achieve the 0-1 normalization is by using the sigmoid function

$$p_i = s(\mathbf{w}^T \mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}_i}}$$

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

$$\begin{aligned}
\mathbf{w}^*_{LR} &= \arg\max_{\mathbf{w}} \prod_{i=1}^{n} P(Y_i = y_i) \\
&= \arg\max_{\mathbf{w}} \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{(1-y_i)} \\
&= \arg\max_{\mathbf{w}} \ln\Big[\prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{(1-y_i)}\Big] \\
&= \arg\max_{\mathbf{w}} \sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i) \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i)
\end{aligned}$$

Logistic Loss: Information-Theoretic Perspective

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 to be

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

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.

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 $P_i \sim Bernoulli(y_i)$. P_i is not a particularly interesting distribution because all values sampled from it will yield y_i: $P(P_i = y_i) = 1$. Logistic regression yields a distribution $Q_i \sim Bernoulli(p_i)$, which is the posterior probability that estimates the true distribution P_i. Be careful not to mix up the notation between the random variable P_i and the quantity p_i! In the context of this problem p_i is associated with the estimated distribution Q_i, not the true distribution P_i.

The KL divergence from Q_i to P_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 weights so that $D_{KL}(P_i||Q_i) = 0$. However, this is impossible for two reasons. First, if we want $D_{KL}(P_i||Q_i) = 0$, 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 weights so that $D_{KL}(P_i||Q_i) = 0$, that's only optimizing one of the data points – we need to tune the weights 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 weights $\mathbf{w}$ (which indirectly affect the p_i values and therefore the estimated distribution Q_i), in order to minimize the total sum of KL divergences contributed by

all data points:

$$\begin{aligned}
\mathbf{w}_{LR}^* &= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} D_{KL}(P_i||Q_i) \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} y_i \ln \frac{y_i}{p_i} + (1-y_i) \ln \frac{(1-y_i)}{(1-p_i)} \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} y_i(\ln y_i - \ln p_i) + (1-y_i)(\ln(1-y_i) - \ln(1-p_i)) \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} (-y_i \ln p_i - (1-y_i)\ln(1-p_i)) + (y_i \ln y_i + (1-y_i)\ln(1-y_i)) \\
&= \arg\min_{\mathbf{w}} -\sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i) \\
&= \arg\min_{\mathbf{w}} \sum_{i=1}^{n} H(P_i, Q_i)
\end{aligned}$$

Note that the $y_i \ln y_i + (1-y_i)\ln(1-y_i)$ component of the KL divergence is a constant, independent of our changes to p_i. Therefore, we are effectively minimizing the sum of the **cross entropies** $H(P_i, Q_i)$. We conclude our discussion of KL Divergence by noting the relation between KL divergence and cross entropy:

$$D_{KL}(P_i||Q_i) = H(P_i, Q_i) - H(P_i)$$

where $D_{KL}(P_i||Q_i)$ is the KL divergence from Q_i to P_i, $H(P_i, Q_i)$ is the cross entropy between P_i and Q_i, and $H(P_i)$ is the **entropy** of P_i. Intuitively, we can think of entropy as the expected amount of "surprise" of a distribution. Mathematically, we have that

$$H(P) = \sum_x -P(x) \ln P(x) = \sum_x P(x) \ln \frac{1}{P(x)} = \mathbb{E}_P\Big[\ln \frac{1}{P(x)}\Big]$$

To simplify the argument, assume that P is a Bernoulli distribution. If $P \sim Bernoulli(1)$ or $P \sim Bernoulli(0)$, there is no surprise at all because we always expect the same value every time we sample from the distribution. This is justified mathematically by the fact that entropy is minimized: $H(P) = 0$. However, when $P \sim Bernoulli(0.5)$, there is a lot of uncertainty/surprise and in fact, entropy is maximized.

6.9 Multiclass Logistic Regression

Recall that in logistic regression, we are tuning a weight vector $w \in \mathbb{R}^{d+1}$, which leads to a posterior distribution $Q_i \sim Bernoulli(p_i)$ over the binary classes 0 and 1:

$$\begin{aligned}
P(Q_i = 1) = p_i = s(w^T x_i) &= \frac{1}{1 + e^{-w^T x_i}} \\
P(Q_i = 0) = 1 - s(w^T x_i) &= \frac{e^{-w^T x_i}}{1 + e^{-w^T x_i}}
\end{aligned}$$

Let's generalize this concept to **Multiclass Logistic Regression**, 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

$$y_i = [0\ 1\ 0]^T$$

Now there is no relative ordering in the representations of the classes. We must modify our weight representation accordingly to the one-hot vector encoding. Now, there are a set of $d+1$ weights 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"

$$z_k = w_k^T x_i$$

Where w_k is the k-th row of the W matrix. In total there are K scores for an input x_i:

$$[w_1^T x_i \quad w_2^T x_i \quad \dots \quad w_K^T x_i]$$

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 Q. For binary logistic regression, we used the logistic function, which takes the value $w^T x_i$ 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:

$$\text{softmax}(j, \{z_1, z_2, \dots, z_K\}) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

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.

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 is a special case of the softmax function. Assuming that the corresponding weights for class 0 and 1 are w_0 and w_1, we have that:

$$P(Q_i = 1) = \frac{e^{w_1^T x_i}}{e^{w_0^T x_i} + e^{w_1^T x_i}} = \frac{e^{(w_1 - w_1)^T x_i}}{e^{(w_0 - w_1)^T x_i} + e^{(w_1 - w_1)^T x_i}} = \frac{1}{1 + e^{-(w_1 - w_0)^T x_i}} = s((w_1 - w_0)^T x_i)$$

$$P(Q_i = 0) = \frac{e^{w_0^T x_i}}{e^{w_0^T x_i} + e^{w_1^T x_i}} = \frac{e^{(w_0 - w_1)^T x_i}}{e^{(w_0 - w_1)^T x_i} + e^{(w_1 - w_1)^T x_i}} = \frac{e^{-(w_1 - w_0)^T x_i}}{1 + e^{-(w_1 - w_0)^T x_i}} = 1 - s((w_1 - w_0)^T x_i)$$

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 $P(Q_i = 1)$, we know by default that $P(Q_i = 0) = 1 - P(Q_i = 1)$.

Multiclass Logistic Regression Loss Function

Let's derive the loss function for multiclass logistic regression, using the information-theoretic perspective. The "true" or more formally the **target distribution** in this case is $P(P_i = j) = y_i[j]$. In other words, the entire distribution is concentrated on the label for the training example. The estimated distribution Q comes from multiclass logistic regression, and in this case is the softmax distribution:

$$P(Q_i = j) = \frac{e^{w_j^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}}$$

Now let's proceed to deriving the loss function. The objective, as always, is to minimize the sum of the KL divergences contributed by all of the training examples.

$$\begin{aligned}
W^*_{MCLR} &= \arg\min_W \sum_{i=1}^{n} D_{KL}(P_i||Q_i) \\
&= \arg\min_W \sum_{i=1}^{n} \sum_{j=1}^{K} P(P_i = j) \ln \frac{P(P_i = j)}{P(Q_i = j)} \\
&= \arg\min_W \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \ln \frac{y_i[j]}{\text{softmax}(j, \{w_1^T x_i, w_1^T x_i, \ldots, w_K^T x_i\})} \\
&= \arg\min_W \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \ln y_i[j] - y_i[j] \cdot \ln \big(\text{softmax}(j, \{w_1^T x_i, w_2^T x_i, \ldots, w_K^T x_i\})\big) \\
&= \arg\min_W - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \ln \big(\text{softmax}(j, \{w_1^T x_i, w_1^T x_i, \ldots, w_K^T x_i\})\big) \\
&= \arg\min_W - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \ln \Big(\frac{e^{w_j^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}}\Big) \\
&= \arg\min_W \sum_{i=1}^{n} H(P_i, Q_i)
\end{aligned}$$

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

$$\begin{aligned}
W^*_{MCLR} &= \arg\max_W \prod_{i=1}^{n} P(Y_i = y_i) \\
&= \arg\max_W \prod_{i=1}^{n} \prod_{j=1}^{K} P(Q_i = j)^{y_i[j]} \\
&= \arg\max_W \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \ln P(Q_i = j) \\
&= \arg\max_W \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \ln \left(\frac{e^{w_j^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}} \right) \\
&= \arg\min_W - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \ln \left(\frac{e^{w_j^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}} \right)
\end{aligned}$$

We conclude that the loss function for multiclass logistic regression is

$$L(W) = - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \ln P(Q_i = j)$$

6.10 Training Logistic Regression

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 Logistic Regression

Recall the loss function

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

where

$$p_i = s(w^T x_i) = \frac{1}{1 + e^{-w^T x_i}}$$

$$\begin{aligned}
\nabla_w L(w) &= \nabla_w \left(-\sum_{i=1}^{n} y_i \ln p_i + (1-y_i)\ln(1-p_i) \right) \\
&= -\sum_{i=1}^{n} y_i \nabla_w \ln p_i + (1-y_i)\nabla_w \ln(1-p_i) \\
&= -\sum_{i=1}^{n} \frac{y_i}{p_i}\nabla_w p_i - \frac{1-y_i}{1-p_i}\nabla_w p_i \\
&= -\sum_{i=1}^{n} \left(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i} \right) \nabla_w p_i
\end{aligned}$$

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

$$\nabla_w p_i = \nabla_w s(w^T x_i) = s(w^T x_i)(1 - s(w^T x_i))x_i = p_i(1-p_i)x_i$$

Plugging in this gradient value, we have

$$\begin{aligned}
\nabla_w L(w) &= -\sum_{i=1}^{n} \left(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i} \right) \nabla_w p_i \\
&= -\sum_{i=1}^{n} \left(\frac{y_i}{p_i} - \frac{1-y_i}{1-p_i} \right) p_i(1-p_i)x_i \\
&= -\sum_{i=1}^{n} \big(y_i(1-p_i) - (1-y_i)(p_i) \big) x_i \\
&= -\sum_{i=1}^{n} (y_i - p_i)\, x_i
\end{aligned}$$

We conclude that the gradient of the loss function with respect to the parameters is

$$\nabla_w L(w) = -\sum_{i=1}^{n} (y_i - p_i)\, x_i = -X^T(y-p)$$

where $y, p \in \mathbb{R}^n$. The gradient descent update is thus

$$w = w - \epsilon \nabla_w L(w)$$

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, lth entry of the Hessian is the partial derivative of the gradient with respect to w_k and w_l:

$$\begin{aligned} H_{kl} &= \frac{\partial^2 L(w)}{\partial w_k \partial w_l} \\ &= \frac{\partial}{\partial w_k} - \sum_{i=1}^{n} (y_i - p_i)\, x_{il} \\ &= \sum_{i=1}^{n} \frac{\partial}{\partial w_k} p_i x_{il} \\ &= \sum_{i=1}^{n} p_i(1-p_i) x_{ik} x_{il} \end{aligned}$$

We conclude that

$$H = \sum_{i=1}^{n} p_i(1-p_i) x_i x_i^T$$

To prove that $L(w)$ is convex in w, we need to show that $w^T H w \geq 0, \quad \forall w$:

$$w^T H w = w^T \sum_{i=1}^{n} p_i(1-p_i) x_i x_i^T w = \sum_{i=1}^{n} (w^T x_i)^2 p_i(1-p_i) \geq 0$$

Multiclass Logistic Regression

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:

$$\begin{aligned} \nabla_{w_l} L(W) &= \nabla_{w_l} \left(- \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \ln \left(\frac{e^{w_j^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}} \right) \right) \\ &= - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \nabla_{w_l} \left(\ln \frac{e^{w_j^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}} \right) \\ &= - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \left(\nabla_{w_l} w_j^T x_i - \nabla_{w_l} \ln \sum_{k=1}^{K} e^{w_k^T x_i} \right) \\ &= - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \left(\mathbb{1}\{j = l\} x_i - \frac{e^{w_l^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}} x_i \right) \\ &= - \sum_{i=1}^{n} \sum_{j=1}^{K} y_i[j] \cdot \left(\mathbb{1}\{j = l\} - \frac{e^{w_l^T x_i}}{\sum_{k=1}^{K} e^{w_k^T x_i}} \right) x_i \\ &= - \sum_{i=1}^{n} \left(\mathbb{1}\{y_i = l\} - P(Q_i = l) \right) x_i \\ &= -X^T \left(\mathbb{1}\{y_i = l\} - P(Q = l) \right) \end{aligned}$$

Note the use of indicator functions: $\mathbb{1}\{j = l\}$ evaluates to 1 if $j = l$, otherwise 0. Also note that since y_i is a one-hot vector encoding, it evaluates to 1 only for one entry and 0 for all other entries.

We can therefore simplify the expression by only considering the j for which $y_i[j] = 1$. The gradient descent update for w_l is

$$w_l = w_l - \epsilon \nabla_{w_l} L(W)$$

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.

6.11 Support Vector Machines

So far we've explored **generative models** (LDA) and **discriminative models** (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:

- Given: training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, +1\}$
- Goal: 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 much 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 weight vector $w \in \mathbb{R}^d$ and a bias term $b \in \mathbb{R}$, that satisfy the linear separability constraints:

$$\forall i, \quad \begin{cases} w^T x_i - b \geq 0 & \text{if } y_i = 1 \\ w^T x_i - b \leq 0 & \text{if } y_i = -1 \end{cases}$$

Equivalently,

$$\forall i, \quad y_i(w^T x_i - b) \geq 0$$

The resulting decision boundary is a hyperplane $H = \{x : w^T x - b = 0\}$. 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 (w, b) is a pair that separates the data points, then the perceptron could also end up choosing a slightly different $(w, b + \epsilon)$ pair. Some hyperplanes are better than others, but the perceptron cannot distinguish between them. This leads to generalization issues.

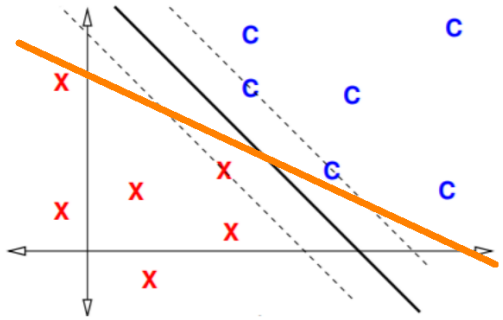


Figure 6.8: Two possible decision boundaries under the perceptron. The X's and C's represent the +1's and −1's respectively.

In the figure above, we consider two 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. The orange separator would incorrectly classify some of these new test points, while the black separator would most likely still be able to classify them correctly. To the eyes of the perceptron algorithm, both the orange and black 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.

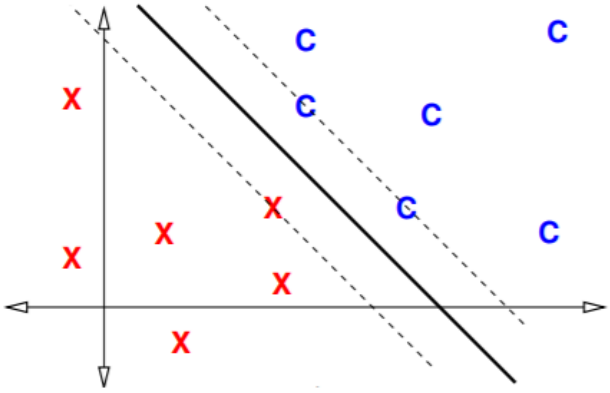


Figure 6.9: The optimal decision boundary (as shown) maximizes the margin.

Intuitively, maximizing the margin allows us to generalize better to unseen data, because the decision boundary with the maximum margin is as far away from the training data as possible and the boundary cannot be violated unless the unseen data contains outliers.

Simply put, the goal of hard-margin SVMs is to find a hyperplane H that maximizes the margin m. Let's formalize an optimization problem for hard-margin SVMs. The variables we are trying to optimize over are the margin m and the parameters of the hyperplane, 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 $H = \{x : w^T x - b = 0\}$.

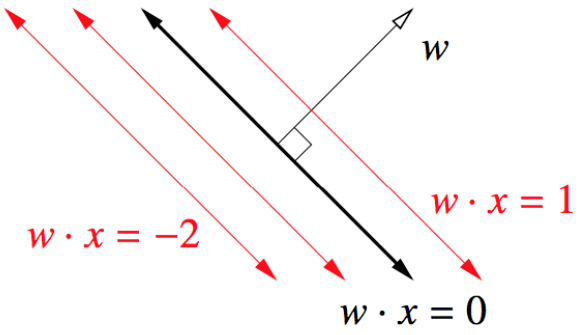


Figure 6.10: Image courtesy Professor Shewchuk's notes.

Proof: consider any two points on H, x_0 and x_1. We will show that $(x_1 - x_0) \perp ((x_1 + w) - x_1)$. Note that

$$(x_1 - x_0)^T((x_1 + w) - x_1) = (x_1 - x_0)^T w = x_1^T w - x_0^T w = b - b = 0$$

Since w is perpendicular to H, the (shortest) distance from any arbitrary point z to the hyperplane H is determined by a scaled multiple of w. If we take any point on the hyperplane x_0, the distance from z to H is the length of the projection from $z - x_0$ to the vector w, which is

$$D = \frac{|w^T(z - x_0)|}{\|w\|_2} = \frac{|w^T z - w^T x_0|}{\|w\|_2} = \frac{|w^T z - b|}{\|w\|_2}$$

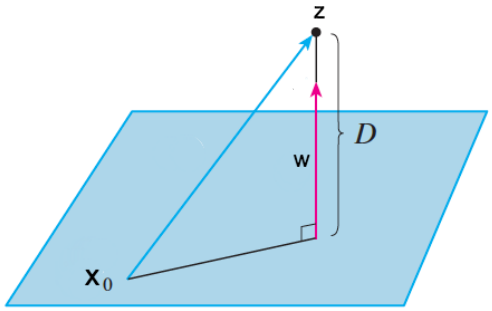


Figure 6.11: Shortest distance from z to H is determined by projection of $z - x_0$ onto w

Therefore, the distance from any of the training points x_i to H is

$$\frac{|w^T x_i - b|}{\|w\|_2}$$

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

$$y_i \frac{(w^T x_i - b)}{\|w\|_2} \geq m$$

Putting everything together, we have the following optimization problem:

$$\begin{aligned} \max_{m,w,b} \quad & m \\ \text{s.t.} \quad & y_i \frac{(w^T x_i - b)}{\|w\|_2} \geq m \quad \forall i \\ & m \geq 0 \end{aligned} \tag{6.1}$$

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 **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 w' and b' that capture the relationship between the three original variables m, w, and b.

$$\begin{aligned} \max_{m,w,b,w',b'} \quad & \frac{1}{\|w'\|_2} \\ \text{s.t.} \quad & y_i(w'^T x_i - b') \geq 1 \quad \forall i \\ & m \geq 0 \\ & w' = \frac{w}{\|w\|_2 m} \\ & b' = \frac{b}{\|w\|_2 m} \end{aligned} \tag{6.2}$$

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

$$\begin{aligned} \max_{w',b'} \quad & \frac{1}{\|w'\|_2} \\ \text{s.t.} \quad & y_i(w'^T x_i - b') \geq 1 \quad \forall i \end{aligned} \tag{6.3}$$

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:

$$\begin{aligned} \min_{w',b'} \quad & \frac{1}{2}\left\|w'\right\|_2^2 \\ \text{s.t.} \quad & y_i(w'^T x_i - b') \geq 1 \quad \forall i \end{aligned} \tag{6.4}$$

At last, we have formulated the hard-margin SVM optimization problem! Using the notation w and b, the objective of hard-margin SVMs is

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|_2^2 \\ \text{s.t.} \quad & y_i(w^T x_i - b) \geq 1 \quad \forall i \end{aligned} \tag{6.5}$$

Soft-Margin SVMs

The hard-margin SVM optimization problem has a unique solution only if the data are linearly separable, but it has no solution otherwise. This is because the constraints are impossible to satisfy if we can't draw a hyperplane that separates the $+1$'s from the -1's. In addition, hard-margin SVMs are very sensitive to outliers – for example, if our data is class-conditionally distributed Gaussian such that the two Gaussians are far apart, if we witness an outlier from class $+1$ that crosses into the typical region for class -1, then hard-margin SVM will be forced to compromise a more generalizable fit in order to accommodate for this point. Our next goal is to come up with a classifier that is not sensitive to outliers and can work even in the presence of data that is not linearly separable. To this end, we'll talk about **Soft-Margin SVMs**.

A soft-margin SVM modifies the constraints from the hard-margin SVM by allowing some points to violate the margin. Formally, it introduces **slack variables** ξ_i, one for each training point, into the constraints:

$$\begin{aligned} y_i(w^T x_i - b) &\geq 1 - \xi_i \\ \xi_i &\geq 0 \end{aligned}$$

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 ξ is spelled "xi" and pronounced "zai". ξ_i is pronounced "zai-eye.")

These constraints would be fruitless if we didn't bound the values of the ξ_i's, 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:

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

Where C is a hyperparameter tuned through cross-validation. Putting the objective and constraints together, the soft-margin SVM optimization problem is

$$\begin{aligned} \min_{w,b,\xi_i} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i(w^T x_i - b) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned} \tag{6.6}$$

The table below compares the effects of having a large C versus a small C. As C goes to infinity, the penalty for having non-zero ξ_i goes to infinity, and thus we force the ξ_i's to be zero, which is exactly the setting of the hard-margin SVM.

	small C	large C
Desire	maximize margin	keep ξ_i's small or zero
Danger	underfitting	overfitting
Outliers	less sensitive	more sensitive

SVMs as Tikhonov Regularization Learning

Consider the following regularized regression problem:

$$\min \frac{1}{n} \sum_{i=1}^{n} L(y_i, w^T x_i - b) + \lambda \|w\|^2$$

In the context of classification, the loss function that we would like to optimize is 0-1 **step loss**:

$$L_{step}(y, w^T x - b) = \begin{cases} 1 & y(w^T x - b) < 0 \\ 0 & y(w^T x - b) \geq 0 \end{cases}$$

The 0-1 loss is 0 if x is correctly classified and 1 otherwise. Thus minimizing $\frac{1}{n} \sum_{i=1}^{n} L(y_i, w^T x_i - b)$ 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 9.1).

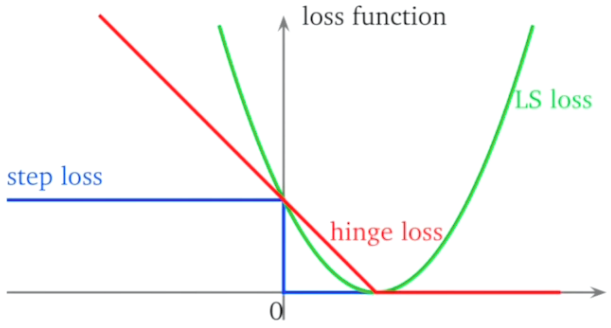


Figure 6.12: Step (0-1) loss, hinge loss, and squared loss. Squared loss is convex and differentiable, hinge loss is only convex, and step loss is neither.

We can try to modify the 0-1 loss to be convex. The points with $y(w^T x - b) \geq 0$ 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 9.1:

$$L_{hinge}(y, w^T x + b) = \max(1 - y(w^T x - b), 0)$$

Thus the regularized regression problem becomes

$$\min_{w,b} \frac{1}{n} \sum_{i=1}^{n} \max(1 - y_i(w^T x_i - b), 0) + \lambda \|w\|^2$$

Recall that the original soft-margin SVM optimization problem is

$$\begin{aligned} \min_{w,b,\xi_i} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & y_i(w^T x_i - b) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned}$$

We claim these two formulations are actually equivalent. Manipulating the first constraint, we have that

$$\xi_i \geq 1 - y_i(w^T x_i - b)$$

Combining with the constraint $\xi_i \geq 0$, we have that

$$\xi_i \geq \max(1 - y_i(w^T x_i - b), 0)$$

At the optimal value of the optimization problem, these inequalities must be tight. Otherwise, we could lower each ξ_i to equal $\max(1 - y_i(w^T x_i - b), 0)$ and decrease the value of the objective function. Thus we can rewrite the soft-margin SVM optimization problem as

$$\begin{aligned} \min_{w,b,\xi_i} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{s.t.} \quad & \xi_i = \max(1 - y_i(w^T x_i - b), 0) \quad \forall i \end{aligned} \tag{6.7}$$

Simplifying further, we can remove the constraints:

$$\min_{w,b} \quad \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\max(1 - y_i(w^T x_i - b), 0) \tag{6.8}$$

If we divide by Cn (which does not change the optimal solution of the optimization problem), we can see that this formulation is equivalent to the regularized regression problem, with $\lambda = \frac{1}{2Cn}$. Thus we have two interpretations of soft-margin SVM: either as finding a max-margin hyperplane that is allowed to make some mistakes via slack variables ξ_i, or as regularized empirical risk minimization with the hinge loss.

Chapter 7

Duality, Kernels

7.1 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's: ± 1, representing positive or negative class
- x_i's: feature vectors in $\mathbb{R}^d$
- ξ_i's: 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:

$$\begin{aligned} \min_{w,t,\xi_i} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i(w^T x_i - t) \geq 1 - \xi_i \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned}$$

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

$$\begin{aligned} \min_{x} \quad & f_0(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad i = 1, \ldots, m \\ & h_j(x) = 0 \quad j = 1, \ldots, n \end{aligned} \tag{7.1}$$

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

$$\min_x \mathcal{L}(x) \tag{7.2}$$

where

$$\mathcal{L}(x) = \begin{cases} f_0(x) & \text{if} \quad f_i(x) \leq 0, \ \forall i \in [1, m] \quad \text{and} \quad h_j(x) = 0, \ \forall j \in [1, n] \\ \infty & \text{otherwise} \end{cases}$$

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 ∞ 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

$$\min_x \max_\nu \mathcal{L}(x, \nu) \tag{7.3}$$

where

$$\mathcal{L}(x, \nu) = \begin{cases} f_0(x) + \sum_{j=1}^n \nu_j h_j(x) & \text{if} \quad f_i(x) \leq 0, \ \forall i \in [1, m] \\ \infty & \text{otherwise} \end{cases}$$

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 ∞ by the outer minimization problem over x and therefore ignored
- For any x that violates the equality constraints (meaning that $\exists j$ s.t. $h_j(x) \neq 0$), the inner maximization problem over ν can choose ν_j as ∞ if $h_j(x) > 0$ (or ν_j as $-\infty$ if $h_j(x) < 0$) to cause the inner maximization blow off to ∞, 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 ν 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 λ_i to represent each inequality constraint, we now have

$$\begin{aligned} \min_x \max_{\lambda,\nu} \quad & \mathcal{L}(x,\lambda,\nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{j=1}^{n} \nu_j h_j(x) \\ \text{s.t.} \quad & \lambda_i \geq 0 \quad i = 1,\dots,m \end{aligned} \tag{7.4}$$

For convenience, we can place the constraints involving λ 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 **primal** (not to be confused with the *primal variables*), and its optimal value is indeed *equivalent* to that of the original constrained optimization problem.

$$p^* = \min_x \max_{\lambda \geq 0,\nu} \mathcal{L}(x,\lambda,\nu)$$

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 λ can choose λ_i as ∞ to cause the inner maximization blow off to ∞, 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 ν can choose ν_j as ∞ if $h_j(x) > 0$ (or ν_j as $-\infty$ if $h_j(x) < 0$) to cause the inner maximization blow off to ∞, 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 ν, the expression $\sum_{j=1}^{n} \nu_j h_j(x)$ evaluates to 0 no matter what the value of ν is, and in the inner maximization problem over λ, the expression $\sum_{i=1}^{m} \lambda_i f_i(x)$ can at maximum be 0, because λ_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 **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 λ_i are in fact constrained, so ultimately we were not able to turn the original optimization problem into an unconstrained one, but our constraints are much simpler than before.

The **dual** of this optimization problem is still over the same optimization objective, except that now we swap the order of the maximization of the dual variables and the minimization of the primal variables.

$$d^* = \max_{\lambda \geq 0,\nu} \min_x \mathcal{L}(x,\lambda,\nu) = \max_{\lambda \geq 0,\nu} g(\lambda,\nu)$$

The dual is effectively a maximization problem (over the dual variables):

$$d^* = \max_{\lambda \geq 0,\nu} g(\lambda,\nu)$$

where

$$g(\lambda, \nu) = \min_x \mathcal{L}(x, \lambda, \nu)$$

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:

$$p^* \geq d^* \tag{7.5}$$

This condition is know as **weak duality**.

Proof. We know that

$$\forall x, \lambda \geq 0, \nu \quad \max_{\tilde{\lambda} \geq 0, \tilde{\nu}} \mathcal{L}(x, \tilde{\lambda}, \tilde{\nu}) \geq \mathcal{L}(x, \lambda, \nu) \geq \min_{\tilde{x}} \mathcal{L}(\tilde{x}, \lambda, \nu)$$

More compactly,

$$\forall x, \lambda \geq 0, \nu \quad \max_{\tilde{\lambda} \geq 0, \tilde{\nu}} \mathcal{L}(x, \tilde{\lambda}, \tilde{\nu}) \geq \min_{\tilde{x}} \mathcal{L}(\tilde{x}, \lambda, \nu)$$

Since this is true for all $x, \lambda \geq 0, \nu$ this is true in particular when we set

$$x = \arg\min_{\tilde{x}} \max_{\tilde{\lambda} \geq 0, \tilde{\nu}} \mathcal{L}(\tilde{x}, \tilde{\lambda}, \tilde{\nu})$$

and

$$\lambda, \nu = \arg\max_{\tilde{\lambda} \geq 0, \tilde{\nu}} \min_{\tilde{x}} \mathcal{L}(\tilde{x}, \tilde{\lambda}, \tilde{\nu})$$

We therefore know that

$$p^* = \min_{\tilde{x}} \max_{\tilde{\lambda} \geq 0, \tilde{\nu}} \mathcal{L}(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) \geq \max_{\tilde{\lambda} \geq 0, \tilde{\nu}} \min_{\tilde{x}} \mathcal{L}(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) = d^*$$

□

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:

$$p^* = d^* \tag{7.6}$$

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 λ^*, ν^* are the primal and dual solutions respectively, with zero duality gap (i.e. strong duality holds), then x^*, λ^*, ν^* 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 λ^*, ν^* must satisfy the dual constraints. Now, let's prove conditions 4 and 5. We know that since strong duality holds, we can say that

$$\begin{aligned}
p^* = f_0(x^*) &= g(\lambda^*, \nu^*) = d^* \\
&= \min_x \mathcal{L}(x, \lambda^*, \nu^*) \\
&\leq \mathcal{L}(x^*, \lambda^*, \nu^*) \\
&= f_0(x^*) + \sum_{i=1}^{m} \lambda_i^* f_i(x^*) + \cancel{\sum_{j=1}^{n} \nu_j^* h_j(x^*)} \\
&= f_0(x^*) + \sum_{i=1}^{m} \lambda_i^* f_i(x^*) \\
&\leq f_0(x^*)
\end{aligned}$$

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 $f_0(x^*) = \min_x \mathcal{L}(x, \lambda^*, \nu^*) \leq \mathcal{L}(x^*, \lambda^*, \nu^*) \leq f_0(x^*)$, we know that all of the inequalities hold with equality and therefore $\mathcal{L}(x^*, \lambda^*, \nu^*) = \min_x \mathcal{L}(x, \lambda^*, \nu^*)$. 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 $\forall i \quad \lambda_i^* f_i(x^*) \leq 0$, 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 $\mathcal{L}(x, \lambda, \nu)$ is convex in x, and if the gradient of $\mathcal{L}(x, \bar{\lambda}, \bar{\nu})$ at $\bar{x}$ is 0, we know that

$$\bar{x} = \arg\min_x \mathcal{L}(x, \bar{\lambda}, \nu^*)$$

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

$$p^* \geq d^* = \max_{\lambda \geq 0, \nu} g(\lambda, \nu) \geq g(\tilde{\lambda}, \tilde{\nu}), \quad \forall \tilde{\lambda} \geq 0, \tilde{\nu}$$

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. □

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.

7.2 The Dual of SVMs

Let's apply our knowledge of duality to find the dual of the soft-margin SVM optimization problem.

$$\overbrace{\min_{w,t,\xi} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i}^{\min\ f(w,t,\xi)}$$
$$\underbrace{(1-\xi_i) - y_i(w^T x_i - t) \leq 0 \ \text{ and } \ -\xi_i \leq 0}_{g(w,t,\xi)\leq 0}$$

Let's identify the primal and dual variables for the SVM problem. We will have

- Primal variables w, t, and ξ_i
- Dual variables α_i corresponding to each constraint of the form $y_i(w^T x_i - t) \geq 1 - \xi_i$
- Dual variables β_i corresponding to each constraint of the form $\xi_i \geq 0$

For the purposes of notation, note that we are using α and β in place of λ, and there are no dual variables corresponding to ν because there are no equality constraints. The lagrangian for the SVM problem is:

$$\begin{aligned}\mathcal{L}(w,t,\xi,\alpha,\beta) &= \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i + \sum_{i=1}^{n}\alpha_i((1-\xi_i) - y_i(w^T x_i - t)) + \sum_{i=1}^{n}\beta_i(-\xi_i) \\ &= \frac{1}{2}||w||^2 - \sum_{i=1}^{n}\alpha_i y_i(w^T x_i - t) + \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i\end{aligned} \tag{7.7}$$

Thus, the dual is:

$$\max_{\alpha,\beta\geq 0} g(\alpha,\beta) = \min_{w,t,\xi} \frac{1}{2}||w||^2 - \sum_{i=1}^{n}\alpha_i y_i(w^T x_i - t) + \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i \tag{7.8}$$

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 α^* and β^*, and the optimal primal values w^*, t^*, ξ_i^*:

$$\frac{\partial\mathcal{L}}{\partial w_i} = \frac{\partial\mathcal{L}}{\partial t} = \frac{\partial\mathcal{L}}{\partial\xi_i} = 0$$

- $\nabla_w\mathcal{L} = w^* - \sum_{i=1}^{n}\alpha_i^* y_i x_i = 0 \implies w^* = \sum_{i=1}^{n}\alpha_i^* y_i x_i$. 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.
- $\frac{\partial\mathcal{L}}{\partial t} = \sum_{i=1}^{n}\alpha_i^* y_i = 0$. This tells us that the weights α_i^* will be equally distributed among positive- and negative- class training points.
- $\frac{\partial\mathcal{L}}{\partial\xi_i} = C - \alpha_i^* - \beta_i^* = 0 \implies 0 \leq \alpha_i^* \leq C$. This tells us that the weights α_i^* are restricted to being less than the hyperparameter C.

Verify that the other KKT also hold, establishing strong duality. Using these observations, we can eliminate some terms of the dual problem.

$$\begin{aligned}
\mathcal{L}(w,t,\xi,\alpha^*,\beta^*) &= \frac{1}{2}||w||^2 - \sum_{i=1}^n \alpha_i^* y_i(w^T x_i - t) + \sum_{i=1}^n \alpha_i^* + \sum_{i=1}^n (C - \alpha_i^* - \beta_i^*)\xi_i \\
&= \frac{1}{2}||w||^2 - \sum_{i=1}^n \alpha_i^* y_i(w^T x_i) + t\underbrace{\sum_{i=1}^n \alpha_i^* y_i}_{=0} + \sum_{i=1}^n \alpha_i^* + \underbrace{\sum_{i=1}^n (C - \alpha_i^* - \beta_i^*)\xi_i}_{=0} \\
&= \frac{1}{2}||w||^2 - \sum_{i=1}^n \alpha_i^* y_i(w^T x_i) + \sum_{i=1}^n \alpha_i^*
\end{aligned}$$

We can also rewrite all the optimal primal variables w^*, t^*, ξ^* in terms of the optimal dual variables α_i^*:

$$\begin{aligned}
g(\alpha^*,\beta^*) &= \min_{w,t,\xi} \mathcal{L}(w,t,\xi,\alpha^*,\beta^*) \\
&= \mathcal{L}(w^*,t^*,\xi^*,\alpha^*,\beta^*) \\
&= \frac{1}{2}||\sum_{i=1}^n \alpha_i^* y_i x_i||^2 - \sum_{i=1}^n \alpha_i^* y_i((\sum_{j=1}^n \alpha_j^* y_j x_j)^T x_i) + \sum_{i=1}^n \alpha_i^* \\
&= \frac{1}{2}||\sum_{i=1}^n \alpha_i^* y_i x_i||^2 - \sum_{i=1}^n (\alpha_i^* y_i x_i^T(\sum_{j=1}^n \alpha_j^* y_j x_j)) + \sum_{i=1}^n \alpha_i^* \\
&= \alpha^{*T}\mathbf{1} - \frac{1}{2}\alpha^{*T} Q \alpha^*
\end{aligned}$$

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 α and x and y (Note that we have eliminated all references to β):

$$\begin{aligned}
\max_{\alpha} \quad & \alpha^T\mathbf{1} - \frac{1}{2}\alpha^T Q \alpha \\
\text{s.t.} \quad & \sum_{i=1}^n \alpha_i y_i = 0 \\
& 0 \le \alpha_i \le C \quad i = 1,\dots,n
\end{aligned} \tag{7.9}$$

Geometric intuition

We've seen that the optimal value of the dual problem in terms of α is equivalent to the optimal value of the primal problem in terms of w, t, and ξ. But what do these dual values α_i even mean? That's a good question!

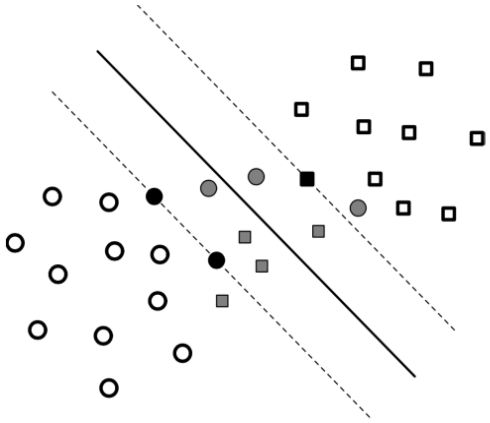
We know that the following KKT conditions are enforced:

- Stationarity
$$C - \alpha_i^* - \beta_i^* = 0$$
- Complementary slackness
$$\alpha_i^* \cdot ((1 - \xi_i^*) - y_i(w^{*T} x_i - t^*)) = 0$$

$$\beta_i^* \cdot \xi_i^* = 0$$

Here are some noteworthy relationships between α_i and the properties of the SVMs:

- Case 1: $\alpha_i^* = 0$. In this case, we know $\beta_i^* = C$, which is nonzero, and therefore $\xi_i^* = 0$. That is, if for point i we have that $\alpha_i^* = 0$ by the dual problem, then we know that there is no slack given to this point. Looking at the other complementary slackness condition, this makes sense because if $\alpha_i^* = 0$, then $y_i(w^{*T}x_i - t^*) - (1 - \xi_i^*)$ may be any value, and if we're minimizing the sum of our ξ_i's, we should have $\xi_i^* = 0$. So, point i lies **on or outside the margin.**
- Case 2: α_i^* is nonzero. If this is the case, then we know $\beta_i^* = C - \alpha_i^* \geq 0$
  - Case 2.1: $\alpha_i^* = C$. If this is the case, then we know $\beta_i^* = 0$, and therefore ξ_i^* may be exactly 0 or nonzero. So, point i lies **on or inside the margin.**
  - Case 2.2: $0 < \alpha_i^* < C$. In this case, then β_i^* is nonzero and $\xi_i^* = 0$. But this is different from Case 1 because with α_i^* nonzero, we can divide by α_i^* in the complementary slackness condition and arrive at the fact that $1 - y_i(w^{*T}x_i - t^*) = 0 \implies y_i(w^{*T}x_i - t^*) = 1$, which means x_i lies exactly on the margin determined by w^* and t^*. So, point i lies **on the margin.**



$\alpha_i = 0 \Rightarrow$ $y_i f(x_i) \geq 1$: on or outside the margin
$0 < \alpha_i < C \Rightarrow$ $y_i f(x_i) = 1$: on the margin
$\alpha_i = C \Rightarrow$ $y_i f(x_i) \leq 1$: on or inside the margin
$\alpha_i = 0 \Leftarrow$ $y_i f(x_i) > 1$: outside the margin
$\alpha_i = C \Leftarrow$ $y_i f(x_i) < 1$: inside the margin

Finally, let's reconstruct the optimal primal values w^*, t^*, ξ_i^* from the optimal dual values α^*:

$$\begin{aligned} w^* &= \sum_{i=1}^{n} \alpha_i^* y_i x_i \\ t^* &= \text{mean}(w^{*T}x_i \qquad \forall i : \quad 0 < \alpha_i^* < C) \\ \xi_i^* &= \begin{cases} 1 - y_i(w^{*T}x_i - t^*) \text{ if } \alpha_i^* = C, \\ 0 \text{ otherwise} \end{cases} \end{aligned} \tag{7.10}$$

7.3 Kernels

Fundamental Theorem of Linear Algebra

Before we begin our discussion of kernels, let's first introduce the **Fundamental Theorem of Linear Algebra (FTLA)**. Suppose that there is a matrix (linear map) A that maps $\mathbb{R}^n$ to $\mathbb{R}^m$. Denote $\mathcal{N}(A)$ as the nullspace of A, and $\mathcal{R}(A)$ as the range of A. Then the following properties hold:

1. $\mathcal{N}(A) \overset{\perp}{\oplus} \mathcal{R}(A^T) = \mathbb{R}^n$

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

   Proof. We can equivalently prove that $\mathcal{N}(A) = \mathcal{R}(A^T)^\perp$.

   $$\begin{aligned} u \in \mathcal{N}(A) &\iff Au = 0 \\ &\iff \langle v | Au \rangle = 0 \quad \forall v \in \mathbb{R}^m \\ &\iff \left\langle A^T v \middle| u \right\rangle = 0 \quad \forall v \in \mathbb{R}^m \\ &\iff u \in \mathcal{R}(A^T)^\perp \end{aligned}$$

   □

   $\mathcal{N}(A^T) \overset{\perp}{\oplus} \mathcal{R}(A) = \mathbb{R}^m$, by symmetry.

2. $\mathcal{N}(A^T A) = \mathcal{N}(A)$

   Proof. Let's prove $\mathcal{N}(A) \subseteq \mathcal{N}(A^T A)$ and $\mathcal{N}(A^T A) \subseteq \mathcal{N}(A)$:

   $$\begin{aligned} u \in \mathcal{N}(A) &\iff Au = 0 \\ &\implies A^T Au = 0 \\ &\iff u \in \mathcal{N}(A^T A) \end{aligned}$$

   $$\begin{aligned} u \in \mathcal{N}(A^T A) &\iff A^T Au = 0 \\ &\iff \left\langle u_1 \middle| A^T Au \right\rangle = 0 \quad \forall u_1 \in \mathbb{R}^n \\ &\iff \langle Au_1 | Au \rangle = 0 \quad \forall u_1 \in \mathbb{R}^n \\ &\implies \langle Au | Au \rangle = 0 \\ &\iff Au = 0 \\ &\iff u \in \mathcal{N}(A) \end{aligned}$$

   □

   $\mathcal{N}(AA^T) = \mathcal{N}(A^T)$, by symmetry.

3. $\mathcal{R}(A^T A) = \mathcal{R}(A^T)$

Proof.

$$\begin{aligned} u \in \mathcal{R}(A^T) &\iff \exists v \in \mathbb{R}^m \quad A^T v = u \\ &\iff \exists v_1 \in \mathcal{R}(A), v_2 \in \mathcal{N}(A^T) \quad A^T(v_1 + v_2) = u \\ &\iff \exists v_1 \in \mathcal{R}(A) \quad A^T v_1 = u \\ &\iff \exists u_1 \in \mathbb{R}^n \quad A^T(Au_1) = u \\ &\iff u \in \mathcal{R}(A^T A) \end{aligned}$$

□

$\mathcal{R}(AA^T) = \mathcal{R}(A)$, by symmetry.

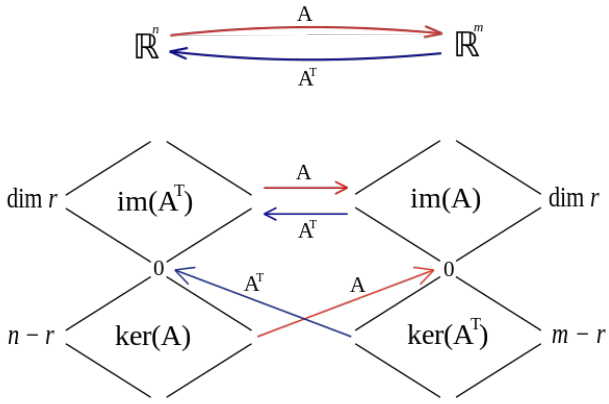


Figure 7.1: This diagram follows from the properties of FTLA

What exactly is the fundamental theorem of linear algebra useful for in the context of kernels? In most machine learning problems such as regression and SVM, we given a vector $y \in \mathbb{R}^n$ and a matrix $X \in \mathbb{R}^{n \times m}$, where n is the number of training points and m 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 end up with a matrix $A \in \mathbb{R}^{n \times d}$, where $a_i = \phi(x_i) \in \mathbb{R}^d$. Then we solve a well-defined optimization problem that involves A and y, over the weights $w \in \mathbb{R}^d$. Note the problem that arises here. If we have polynomial features of degree at most p in the raw m dimensional space, then there are $d = O(m^p)$ terms that we need to optimize, which can be very very very large, in fact much larger than the number of training points n. Wouldn't it be useful, if instead of solving a optimization problem over d variables, we could solve an equivalent problem over (potentially much smaller) n variables? Here's where FTLA comes in. We know that we can express any $w \in \mathbb{R}^d$ as a unique combination $w = w_1 + w_2$, where $w_1 \in \mathcal{R}(A^T)$ and $w_2 \in \mathcal{N}(A)$. Equivalently we can express this as $w = A^T v + r$, where $v \in \mathbb{R}^n$ and $r \in \mathcal{N}(A)$. Now, instead of optimizing over $w \in \mathbb{R}^d$, we can optimize over $v \in \mathbb{R}^n$ and $r \in \mathbb{R}^d$, which equates to optimizing over $n + d$ variables. However, as we shall see, the optimization over r will be trivial so we just have to optimize an n dimensional problem. Let's first see how this plays our in ridge regression.

Kernel Ridge Regression

We know that $w = A^T v + r$, where $v \in \mathbb{R}^n$ and $r \in \mathcal{N}(A)$. Let's now solve ridge regression by optimizing over the variables v and r instead of w:

$$\begin{aligned}
v^*, r^* &= \underset{v,r \in \mathcal{N}(A)}{\arg\min} ||Aw - y||_2^2 + \lambda ||w||_2^2 \\
&= \underset{v,r \in \mathcal{N}(A)}{\arg\min} ||A(A^T v + r) - y||_2^2 + \lambda ||A^T v + r||_2^2 \\
&= \underset{v,r \in \mathcal{N}(A)}{\arg\min} ||AA^T v + \cancel{Ar} - y||_2^2 + \lambda ||A^T v + r||_2^2 \\
&= \underset{v,r \in \mathcal{N}(A)}{\arg\min} \left(v^T AA^T AA^T v - 2v^T AA^T y + y^T y \right) + \lambda \left(v^T AA^T v + \cancel{2v^T Ar} + r^T r \right) \\
&= \underset{v,r \in \mathcal{N}(A)}{\arg\min} \left(v^T AA^T AA^T v - 2v^T AA^T y \right) + \lambda \left(v^T AA^T v + r^T r \right)
\end{aligned}$$

We crossed out Ar and $2v^T Ar$ because $r \in \mathcal{N}(A)$ and therefore $Ar = 0$. Now we are optimizing over $L(v, r)$, which is **jointly convex** in v and r, because its hessian is positive semi-definite. Let's show that this is indeed the case:

$$\nabla_r^2 L(v, r) = 2I \succeq 0$$

$$\nabla_r \nabla_v L(v, r) = \nabla_v \nabla_r L(v, r) = 0$$

$$\nabla_v^2 L(v, r) = 2AA^T AA^T + 2\lambda AA^T \succeq 0$$

Since the cross terms of the hessian are 0, it suffices that $\nabla_r^2 L(v, r)$ and $\nabla_v^2 L(v, r)$ are psd to establish joint convexity. With joint convexity established, we can set the gradient to 0 wrt r and v and obtain the global minimum:

$$\nabla_r L(v, r) = 2r = 0 \implies r^* = 0$$

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

$$\begin{aligned}
\nabla_v L(v, r) = 2AA^T AA^T v - 2AA^T y + 2\lambda AA^T v = 0 \\
\implies AA^T (AA^T + \lambda I) v = AA^T (y) \\
\implies v^* = (AA^T + \lambda I)^{-1} y
\end{aligned}$$

Note that $AA^T + \lambda I$ is positive definite and therefore invertible, so we can compute $(AA^T + \lambda I)^{-1} y$. Even though $(AA^T + \lambda I)^{-1} y$ 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

$$w^* = A^T (AA^T + \lambda I)^{-1} y$$

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

$$w^* = (A^T A + \lambda I)^{-1} Ay$$

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.

Alternative Derivation

We can arrive at the same expression for w^* with some clever algebraic manipulations. Our previous derivation of ridge regression was $w^* = (A^TA+\lambda I)^{-1}A^TY$. Rearranging the terms of this equation, we have

$$\begin{aligned}(A^TA + \lambda I)w &= A^Ty \\ A^TAw + \lambda w &= A^Ty \\ \lambda w &= A^Ty - A^TAw \\ w &= \frac{1}{\lambda}(A^Ty - A^TAw) \\ w &= \frac{A^Ty - A^TAw}{\lambda} \\ w &= A^T\frac{y - Aw}{\lambda}\end{aligned}$$

which says that *whatever* w *is*, it is some linear combination of the training points a_i (because anything of the form A^Tv is a linear combination of the columns of A^T, which are the training points). To find w it suffices to find v, where $w = A^Tv$.

Recall that the relationship we have to satisfy is $A^TAw - \lambda w = A^Ty$. Let's assume that we had v, and just substitute A^Tv in for all the w's.

$$\begin{aligned}A^TA(A^Tv) + \lambda(A^Tv) &= A^Ty \\ A^TAA^Tv + A^T(\lambda v) &= A^Ty \\ A^T(AA^Tv + \lambda v) &= A^T(y)\end{aligned}$$

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

$$AA^Tv + \lambda v = y$$

that would *imply* that this v also satisfies the above equation. Note that we did not "cancel the A^T'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:

$$(AA^T + \lambda I)v = y \implies v^* = (AA^T + \lambda I)^{-1}y$$

and have that the v which satisfies this equation will be such that A^Tv equals w. We conclude that the optimal w is

$$w^* = A^Tv^* = A^T(AA^T + \lambda I)^{-1}y$$

Kernels

Recall that in the solution for ridge regression we have an AA^T term. If we expand this term we have that

$$AA^T = \begin{pmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_n^T & — \end{pmatrix}\begin{pmatrix} | & | & & | \\ a_1 & a_2 & \dots & a_n \\ | & | & & | \end{pmatrix} = \begin{pmatrix} a_1^Ta_1 & a_1^Ta_2 & \dots \\ a_2^Ta_1 & \ddots & \\ \vdots & & a_n^Ta_n \end{pmatrix}$$

Each entry AA^T_{ij} is a dot product between a_i and a_j and can be interpreted as a similarity measure. In fact, we can express

$$AA^T_{ij} = \langle a_i | a_j \rangle = \langle \phi(x_i) | \phi(x_j) \rangle = k(x_i, x_j)$$

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:

$$K = AA^T = \begin{pmatrix} k(x_1,x_1) & k(x_1,x_2) & \dots \\ k(x_2,x_1 & \ddots & \\ \vdots & & k(x_n,x_n) \end{pmatrix}$$

Formally, $k(x_1, x_2)$ is defined to be a valid kernel function if there exists a feature map $\phi(.)$ such that $\forall x_1, x_2$,

$$k(x, y) = \langle \phi(x_1) | \phi(x_2) \rangle$$

Equivalently, we can also say that for all sets $\mathcal{D} = \{a_1, a_2, \dots, a_n\}$, the Gram matrix $K(\mathcal{D})$ is positive semi-definite.

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 $O(m + \log p)$ time, where m is the dimensionality of the problem in the raw feature space and k is the degree of the polynomials in the augmented feature space.

7.4 Kernel Trick

Suppose that you are computing $k(x, z)$, using a p-degree polynomial feature map that maps m dimensional inputs to $d = O(m^p)$ dimensional outputs. Let's take $p = 2$ and $m = 2$ as an example. We have that

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

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

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

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

features, we have that

$$k(x, z) = (x^T z + 1)^p$$

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!

Computational Analysis

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

$$w^* = A^T(AA^T + \lambda I)^{-1}y$$

or

$$w^* = (A^T A + \lambda I)^{-1}Ay$$

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

1. Kernelized

   $$\hat{y} = \langle \phi(z) | w^* \rangle = \phi(z)^T A^T (AA^T + \lambda I)^{-1} y = \begin{bmatrix} k(x_1, z) & \dots & k(x_n, z) \end{bmatrix}^T (K + \lambda I)^{-1} y$$

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

2. Non-kernelized

   $$\hat{y} = \langle \phi(z) | w^* \rangle = \phi(z)^T (A^T A + \lambda I)^{-1} A^T y$$

   Computing the $A^T A$ term takes $O(d^2 n)$, and inverting the matrix takes $O(d^3)$. These two computations dominate, for a total computation time of $O(d^3 + d^2 n) = O(m^{3p} + m^{2p} n)$.

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

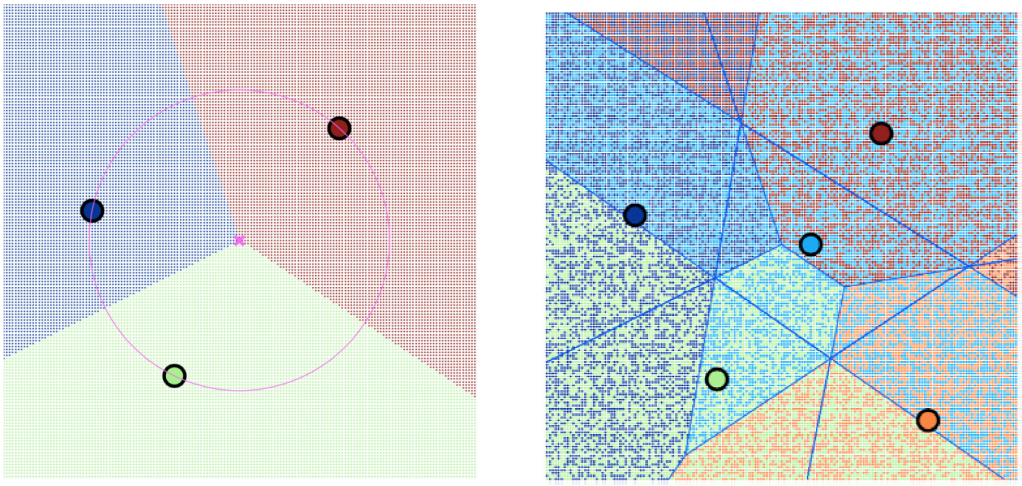
This discussion now motivates a major reason for dual SVMs. Recall that the dual SVM formulation is an optimization problem over n variables instead of d variables. It incorporates a $Q = (\text{diag } y)AA^T(\text{diag } y)$ term, and using the kernel trick, the AA^T term takes $O(n^2(m + \log p))$ instead of $O(n^2 m^p)$ to calculate, making the dual SVM formulation a very attractive choice when $n << d$.

Kernelization can be applied to pretty much any machine learning problem, such as logistic regression, k nearest neighbors, and even PCA.

7.5 Nearest Neighbor Classification

In classification, it is reasonable to conjecture that data points that are sufficiently close to one another should be of the same class. For example, in fruit classification, perturbing a few pixels in an image of a banana should still result in something that looks like a banana. The **k-nearest-neighbors (k-NN)** classifier is based on this observation. Assuming that there is no preprocessing of the training data, the training time for k-NN is effectively $O(1)$. To train this classifier, we simply

store our training data for future reference.[1] For this reason, k-NN is sometimes referred to as "lazy learning." The major work of k-NNs in done at testing time: to predict on a test data point 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 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.



(a) 1-NN on 3 classes

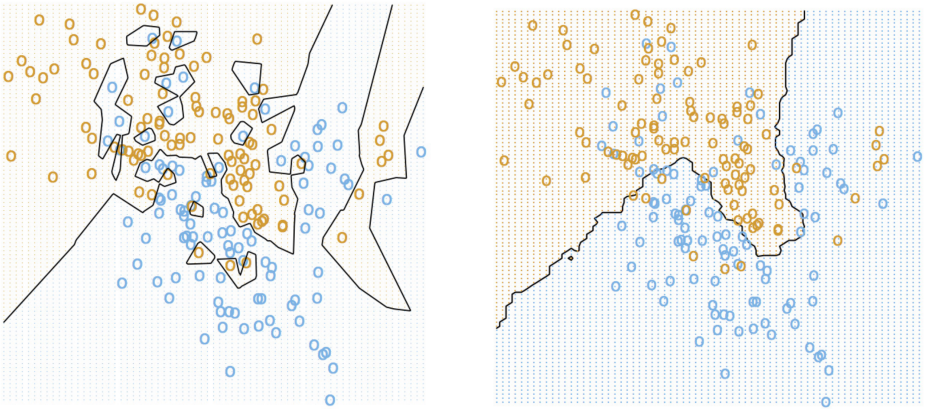
(b) 2-NN on 5 classes

Figure 7.2: **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.

[1] Sometimes we store the data in a specialized structure called a *k-d tree*. This data structure is out of scope for this course, but it usually allows for faster (average-case $O(\log n)$) nearest neighbors queries.



(a) $k = 1$ (b) $k = 15$

Figure 7.3: **Voronoi diagram** for $k = 1$ vs. $k = 15$. Figure from Introduction to Statistical Learning.

Choosing $k = 1$, we achieve an optimal training error of 0 because each training point will classify as itself, thus achieving 100% accuracy on itself. However, $k = 1$ overfits to the training data, and is a terrible choice in the context of the bias-variance tradeoff. Increasing k leads to an increase in training error, but a decrease in testing error and achieves better generalization. At one point, if k becomes too large, the algorithm will underfit the training data, and suffer from huge bias. In general, in order to select k we use cross-validation.

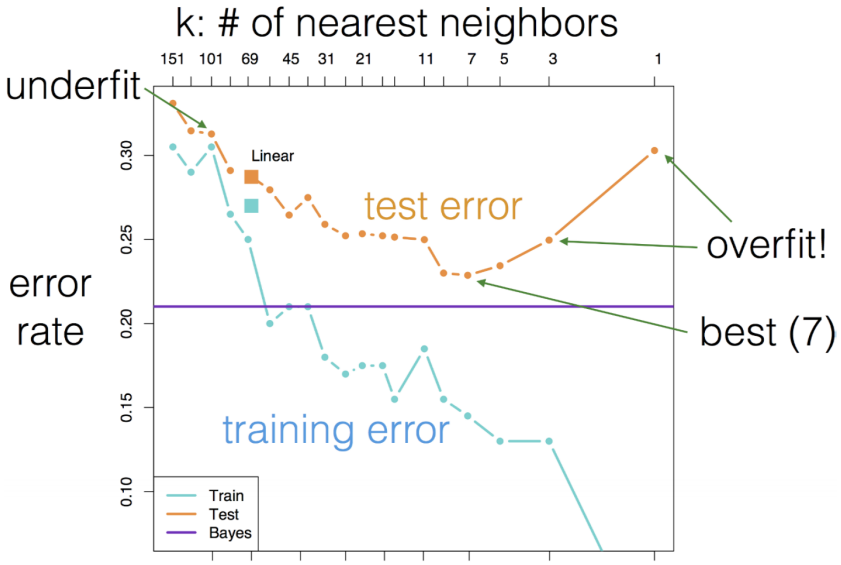


Figure 7.4: Training and Testing error as a function of k. Figure from Introduction to Statistical Learning.

Bias-Variance Analysis

Let's justify this reasoning formally for k-NN applied to regression tasks. Suppose we are given a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, 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

$$N(x_i, z, k) = \begin{cases} y_i & \text{if } x_i \text{ is one of the } k \text{ closest points to } z \\ 0 & o.w. \end{cases}$$

Suppose also we assume our labels $y_i = f(x_i) + \epsilon$, where ϵ is the noise that comes from $\mathcal{N}(0, \sigma^2)$ and f is the true function. Let $x_1 \dots x_k$ be the k closest points. Let's first derive the bias[2] of our model for the given dataset $\mathcal{D}$.

$$\begin{aligned}
\big(\mathbb{E}[h(z)] - f(z)\big)^2 &= \left(\mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{n} N(x_i, z, k)\right] - f(z)\right)^2 = \left(\mathbb{E}\left[\frac{1}{k}\sum_{i=1}^{k} y_i\right] - f(z)\right)^2 \\
&= \left(\frac{1}{k}\sum_{i=1}^{k} \mathbb{E}[y_i] - f(z)\right)^2 = \left(\frac{1}{k}\sum_{i=1}^{k} \mathbb{E}[f(x_i) + \epsilon] - f(z)\right)^2 \\
&= \left(\frac{1}{k}\sum_{i=1}^{k} f(x_i) - f(z)\right)^2
\end{aligned}$$

When $k \longrightarrow \infty$, 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 $k \longrightarrow \infty$, the bias doesn't depend on the training points at all which like will restrict it to be higher.

Now, let's derive the variance of our model.

$$\begin{aligned}
Var[h(z)] &= Var\left[\frac{1}{k}\sum_{i=1}^{k} y_i\right] = \frac{1}{k^2}\sum_{i=1}^{k} Var[f(x_i) + \epsilon] \\
&= \frac{1}{k^2}\sum_{i=1}^{k} Var[\epsilon] \\
&= \frac{1}{k^2}\sum_{i=1}^{k} \sigma^2 = \frac{1}{k^2} k\sigma^2 = \frac{\sigma^2}{k}
\end{aligned}$$

The variance goes to 0 when $k \longrightarrow \infty$, and is maximized at $k = 1$.

Properties

Computational complexity: We require $O(n)$ space to store a training set of size n. There is no runtime cost during training if we do not use specialized data structures to store the data. However, predictions take $O(n)$ time, which is costly. There has been research into approximate nearest neighbors (ANN) procedures that quickly find an approximation for the nearest neighbor - some common ANN methods are *Locality-Sensitive Hashing* and algorithms that perform dimensionality reduction via randomized (Johnson-Lindenstrauss) distance-preserving projections.[2]

[2] ANN methods are beyond the scope of this course, but are useful in real applications.

Flexibility: When $k > 1$, k-NN can be modified to output predicted probabilities $P(Y|X)$ by defining $P(Y|X)$ as the proportion of nearest neighbors to X in the training set that have class Y. k-NN can also be adapted for regression — instead of taking the majority vote, take the average of the y values for the nearest neighbors. k-NN can learn very complicated, non-linear decision boundaries.

Non-parametric: k-NN is a **non-parametric method**, which means that the number of parameters in the model grows with n, the number of training points. This is as opposed to parametric methods, for which the number of parameters is independent of n. Some examples of parametric models include linear regression, LDA, and neural networks.

Behavior in high dimensions: k-NN does not behave well in high dimensions. As the dimension increases, data points drift farther apart, so even the nearest neighbor to a point will tend to be very far away.

Theoretical properties: 1-NN has impressive theoretical guarantees for such a simple method. Cover and Hart, 1967 prove that as the number of training samples n approaches infinity, the expected prediction error for 1-NN is upper bounded by $2\epsilon^*$, where ϵ^* is the Bayes (optimal) error. Fix and Hodges, 1951 prove that as n and k approach infinity and if $\frac{k}{n} \to 0$, then the k nearest neighbor error approaches the Bayes error.

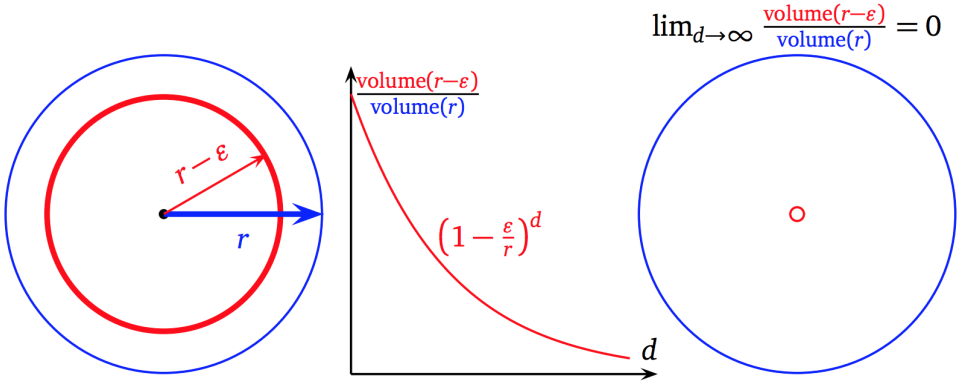
Curse of Dimensionality

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

$$\frac{(r-\epsilon)^d}{r^d} = \left(1 - \frac{\epsilon}{r}\right)^d \approx e^{-\epsilon d/r} \underset{d\to\infty}{\longrightarrow} 0$$

Hence as d gets large, most of the volume of the outer ball is concentrated in the annular region $\{x : r - \epsilon < x < r\}$ instead of the inner ball.



High dimensions also make Gaussian distributions behave counter-intuitively. Suppose $X \sim \mathcal{N}(0, \sigma^2 I)$. If X_i are the components of X and R is the distance from X to the origin, then $R^2 = \sum_{i=1}^d X_i^2$. We

have $E(R^2) = d\sigma^2$, so in expectation a random Gaussian will actually be reasonably far from the origin. If $\sigma = 1$, then R^2 is distributed chi-squared with d degrees of freedom. One can show that in high dimensions, with high probability $1 - O(e^{-d^\epsilon})$, this multivariate Gaussian will lie within the annular region $\{X : |R^2 - E(R^2)| \leq d^{1/2+\epsilon}\}$ 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 $X_i \sim \mathcal{N}(0, \sigma^2)$, $i = 1, ..., d$ 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

$$P(|R^2 - E(R^2)| \geq d^{\frac{1}{2}+\epsilon}) \leq \frac{2d\sigma^4}{d^{1+2\epsilon}} \underset{d\to\infty}{\longrightarrow} 0$$

□

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 Φ and we want to compute the Euclidean distance from $\Phi(x)$ to $\Phi(z)$, then we have

$$\begin{aligned}\|\Phi(x) - \Phi(z)\|_2^2 &= \Phi(x)^T\Phi(x) - 2\Phi(x)^T\Phi(z) + \Phi(z)^T\Phi(z) \\ &= k(x,x) - 2k(x,z) + k(z,z)\end{aligned}$$

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 Φ-space without explicitly representing Φ by using the kernelized distance function D.

Chapter 8

Sparsity

8.1 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 ξ is **sparse** — most of its entries are 0. We are interested in explaining why this phenomenon occurs in this specific optimization problem, and identifying the key properties that determine sparse solutions for arbitrary optimization problems.

To reason about the SVM case, let's see how changing some arbitrary slack variable ξ_i affects the loss. A unit decrease in ξ_i results in a "reward" of C, and is captured by the partial derivative $\frac{\partial L}{\partial \xi_i}$. Note that no matter what the current value of ξ_i is, the reward for decreasing ξ_i is constant. Of course, decreasing ξ_i may change the boundary and thus the cost attributed to the size of the margin $\|w\|^2$. The overall reward for decreasing ξ_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, ξ_i will continue to decrease until it hits a lower-bound "equilibrium" — which is often just 0.

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

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

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

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 faster 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 λ, 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.

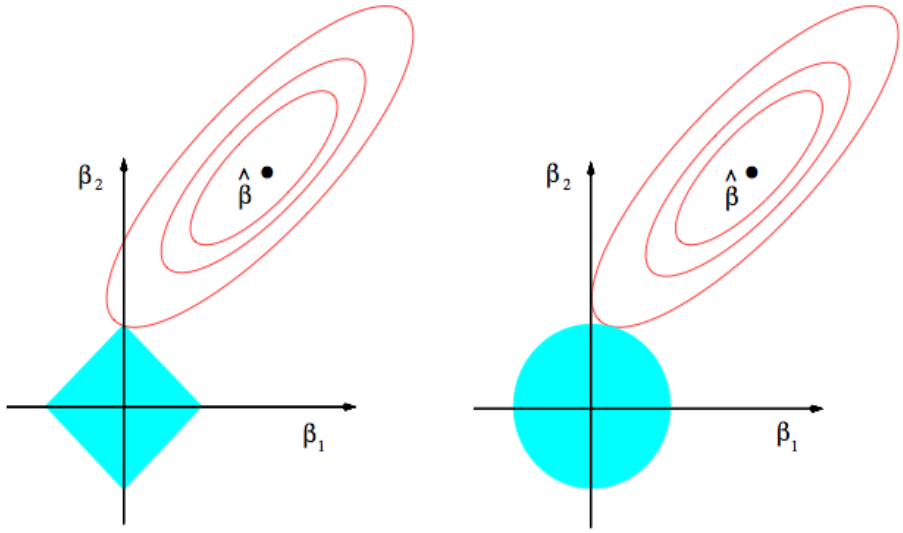


Figure 8.1: Comparing contour plots for LASSO (left) vs. ridge regression (right).

Suppose for simplicity that we are only working with 2-dimensional data points and are thus optimizing over two weight variables w_1 and w_2. In both figures above, the red ellipses represent isocontours in w-space of the squared loss $\|Xw - y\|^2$. In ridge regression, each isocontour of $\lambda\|w\|_2^2$ 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!

8.2 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 1 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

$$\begin{aligned} L(w) &= \|Xw - y\|_2^2 + \lambda\|w\|_1 \\ &= \|\sum_{j=1}^{d} w_j x_j - y\|_2^2 + \lambda|w_i| + \lambda\sum_{j\neq i}|w_j| \\ &= \|w_i x_i + C^{(1)}\|_2^2 + \lambda|w_i| + C^{(2)} \end{aligned}$$

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

$$L(w) = \lambda|w_i| + C^{(2)} + \sum_{j=1}^{n}(w_i x_{j,i} + C_j^{(1)})^2$$

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 λ, 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.

8.3 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

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

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 with a completely sparse solution ($w = 0$) and iteratively "builds up" w until the the sparsity constraint $\|w\|_0 \leq k$ can no longer be met. The algorithm is as follows:

Algorithm 2 Matching Pursuit

initialize the residual $r = y$
initialize the weights $w = 0$
while $\|w\|_0 < k$ **do**
  find the index i for which the residual is minimized: $i = \arg\max_j \langle r, x_j \rangle / \|x_j\|$
  update the i'th entry of the weight vector to $w_i = \langle r, x_i \rangle / \|x_i\|^2$
  update the new residual value: $r = y - Aw$
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 $\langle r, x_j \rangle / \|x_j\|$. The length of the new residual follows from pythagoras theorem:

$$\|r_{old}\|^2 = \|r_{new}\|^2 + \left(\frac{\langle r, x_j \rangle}{\|x_j\|} \right)^2 \implies i = \arg\max_j \frac{\langle r, x_j \rangle}{\|x_j\|}$$

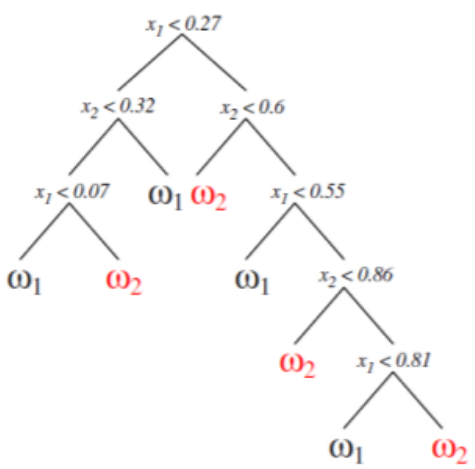
We move w_i to the optimum projection value and repeat greedily at each iteration. While matching pursuit is not guaranteed to find the optimal w^*, in practice it works well for most applications. Setting the number of iterations is typically determined through cross-validation.

Chapter 9

Decision Tree Learning

9.1 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.

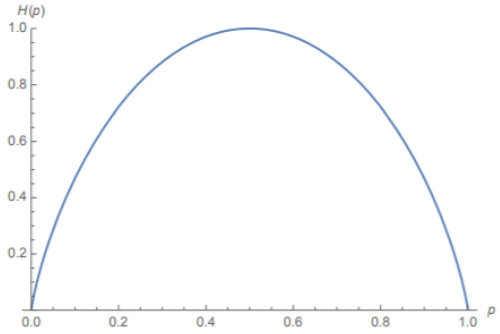
Now, to concretely talk about the idea of "how sure" we are, we'll use the ideas of **surprise** and **entropy**. The numerical surprise of observing that a random variable X takes on value k is:

$$\log \frac{1}{P(X = k)} = -\log P(X = k)$$

The entropy of a random variable $H(X)$ is:

$$\begin{aligned} H(X) &= E[\text{surprise}] \\ &= E_X[\log \frac{1}{P(X = k)}] \\ &= \sum_k P(X = k) \log \frac{1}{P(X = k)} \end{aligned}$$

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:

$$\begin{aligned}\textbf{Minimize } & P(X<v)H(X|Y=1) + P(X\geq v)H(X|Y=0) \\ =& P(Y=1)H(X|Y=1) + P(Y=0)H(X|Y=0)\end{aligned}$$

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:

$$\textbf{Maximize } H(X) - \underbrace{P(Y=1)H(X|Y=1) + P(Y=0)H(X|Y=0)}_{H(X|Y)}$$

.

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:

$$\begin{aligned}H(X|Y=y) &= \sum_k P(X=k|Y=y)\log\frac{1}{P(X=k|Y=y)} \\ H(X|Y) &= \sum_y P(Y=y)H(X|Y=y) \\ I(X;Y) &= H(X) - H(X|Y)\end{aligned}$$

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 precedng 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.

9.2 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.

9.3 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 $e_m = \frac{\sum_{i \text{ misclassified}} w_i}{\sum_i w_i}$.
   (c) Re-weight the training points as

$$w_i \leftarrow w_i \cdot \begin{cases} \sqrt{\frac{1-e_m}{e_m}} & \text{if misclassified} \\ \sqrt{\frac{e_m}{1-e_m}} & \text{otherwise} \end{cases}$$

   (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 $\text{sign}(\sum_m \alpha_m G_m(x))$. Note that this choice of α_m (derived later) gives high weight to classifiers that have low error:

- As $e_m \to 0$, $\frac{1-e_m}{e_m} \to \infty$, so $\alpha_m \to \infty$.
- As $e_m \to 1$, $\frac{1-e_m}{e_m} \to 0$, so $\alpha_m \to -\infty$.

We now proceed to demystify the formulas in the algorithm by presenting a matching pursuit interpretation of AdaBoost.

Derivation of AdaBoost

Suppose we have computed classifiers $G_1, ..., G_{m-1}$ along with their corresponding weights α_k and we want to compute the next classifier G_m along with its weight α_m. The output of our model so far is $F_{m-1}(x) := \sum_{i=1}^{m-1} \alpha_i G_i(x)$, 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*:

$$L(y, h(x)) = e^{-yh(x)}$$

This loss function is illustrated in Figure 9.1. We have exponential decay as we increase the input – thus if $yh(x)$ is large and positive (so $h(x)$ has the correct sign and high magnitude), our loss is decreasing exponentially. Conversely, if $yh(x)$ is a large negative value, our loss is increasing exponentially, and thus we are heavily penalized for confidently making an incorrect prediction.

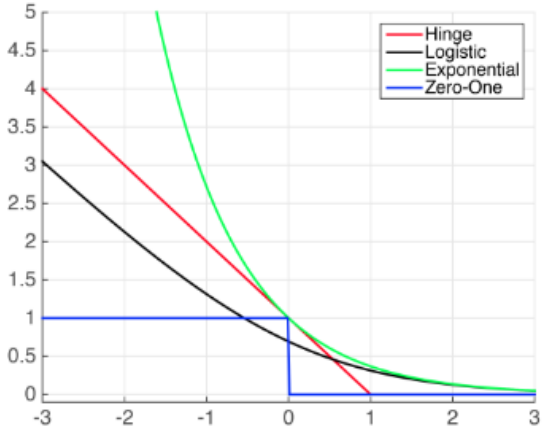


Figure 9.1: The exponential loss provides exponentially increasing penalty for confident incorrect predictions. This figure is from Cornell CS4780 notes.

We can write the AdaBoost optimization problem with exponential loss as follows:

$$\begin{aligned}\alpha_m, G_m &= \arg\min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i(F_{m-1}(x_i)+\alpha G(x_i))} \\ &= \arg\min_{\alpha, G} \sum_{i=1}^{n} e^{-y_i F_{m-1}(x_i)} e^{-y_i \alpha G(x_i)}\end{aligned}$$

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:

$$\begin{aligned}\alpha_m, G_m &= \arg\min_{\alpha, G} \sum_{y_i = G(x)} w_i^{(m)} e^{-\alpha} + \sum_{y_i \neq G(x)} w_i^{(m)} e^{+\alpha} \qquad (*) \\ &= \arg\min_{\alpha, G} (e^{\alpha} - e^{-\alpha}) \sum_{y_i \neq G(x_i)} w_i^{(m)} + e^{-\alpha} \sum_{i=1}^{n} w_i^{(m)}\end{aligned}$$

For a fixed value of α, 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

$$e_m = \frac{\sum_{y_i \neq G_m(x_i)} w_i^{(m)}}{\sum_i w_i^{(m)}}$$

Once we have obtained G_m, we can solve for α_m: by dividing $(*)$ by the constant $\sum_{i=1}^{n} w_i^{(m)}$, we obtain

$$\alpha_m = \arg\min_{\alpha} (1 - e_m) e^{-\alpha} + e_m e^{\alpha}$$

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 α_m, we can derive the weights:

$$\begin{aligned}w_i^{(m+1)} &= \exp\left(-y_i F_m(x_i)\right) \\ &= \exp\left(-y_i [F_{m-1}(x_i) + \alpha_m G_m(x_i)]\right) \\ &= w_i^{(m)} \exp\left(-y_i G_m(x_i) \alpha_m\right) \\ &= w_i^{(m)} \exp\left(-y_i G_m(x_i) \frac{1}{2} \ln \left(\frac{1 - e_m}{e_m} \right)\right) \\ &= w_i^{(m)} \exp\left(\ln \left[\left(\frac{1 - e_m}{e_m} \right)^{-\frac{1}{2} y_i G_m(x_i)} \right]\right) \\ &= w_i^{(m)} \left(\frac{1 - e_m}{e_m} \right)^{-\frac{1}{2} y_i G_m(x_i)}\end{aligned}$$

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 α updates as pushing towards a solution in some direction until we can no longer improve our performance. More precisely, whenever we compute α_m (and thus $w^{(m+1)}$), for the incorrectly classified entries, we have

$$\sum_{y_i \neq G_m(x_i)} w_i^{(m+1)} = \sum_{y_i \neq G_m(x_i)} w_i^{(m)} \sqrt{\frac{1-e_m}{e_m}}$$

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 α, so the misclassified and correctly classified sets both get equal total weight.

This observation has an interesting practical implication. Even after the training error goes to zero, the 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.

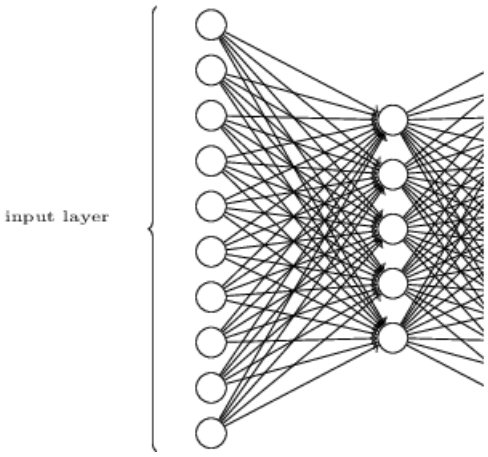
Chapter 10

Deep Learning

10.1 Convolutional Neural Nets

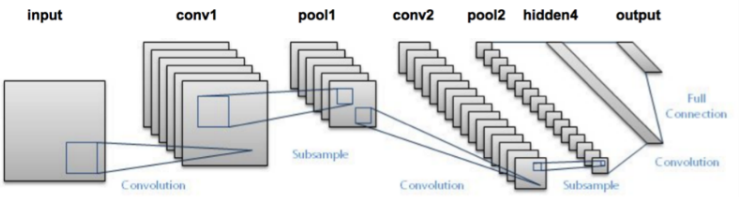
A **convolutional neural net** is just like a regular neural net, except more complicated (convoluted). But in some ways, a convolutional neural net can actually be a simpler model than a fully-connected neural net.

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.

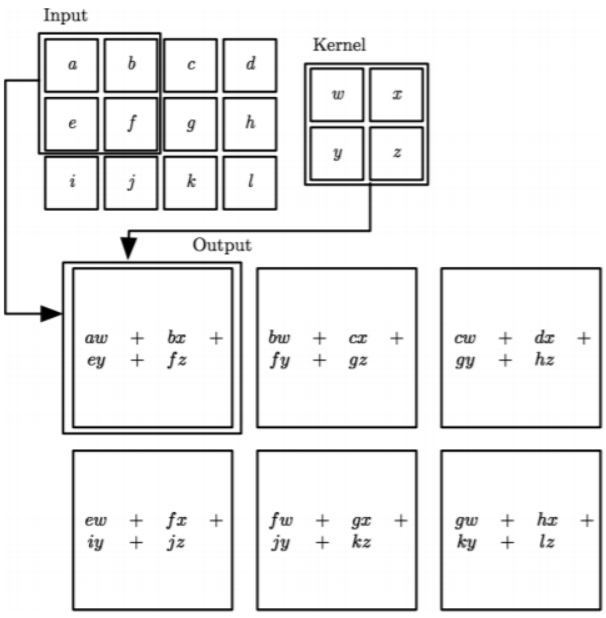
The fully-connected architecture of units has a *lot* of weights to learn. For just a small 32×32 image, there would be 1024 input units and at least as many weights just between the first two layers. Now, let's look at how a ConvNet which deals with images uses its weights differently.



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 $[1 \quad -1]$. 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.

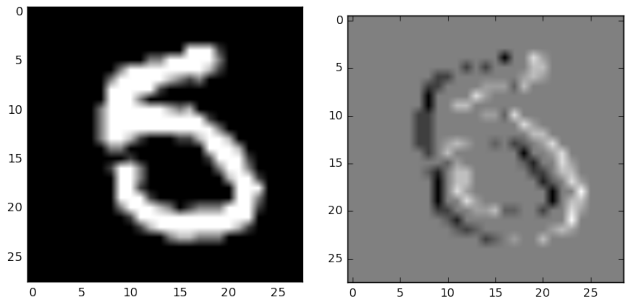


Figure 10.1: Left: a sample image.
Right: the output of convolving the $[1 \quad -1]$ kernel about the image.

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.

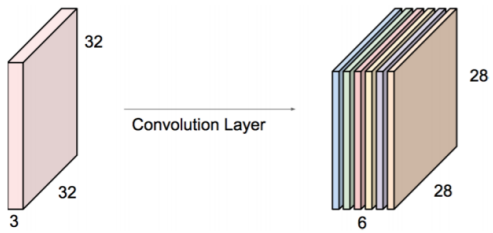
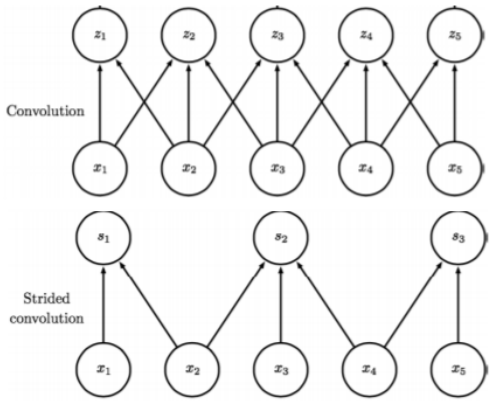


Figure 10.2: Here, we slid 6 separate $5 \times 5 \times 3$ kernels across the original image to produce 6 activation maps in the next convolutional layer. Each activation map can also be referred to as a **channel**.

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 3×3 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.

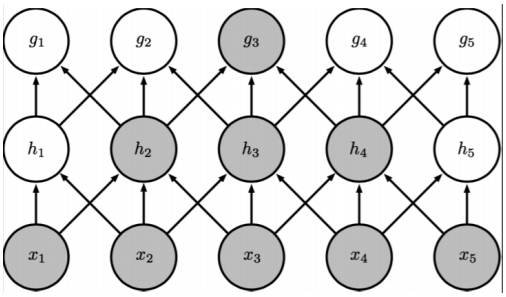


Figure 10.3: The highlighted unit in the downstream layer uses information from all the highlighted units in the input layer.

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.

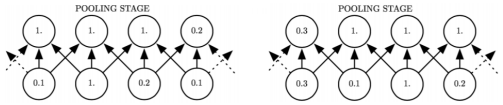


Figure 10.4: Here, the input layer of the right image is a translated version of the input layer of the left image, but because of max-pooling, the next layer looks more or less the same.

   (b) Cross-channel pooling pools values across different channels. This has the capability of introducing transformational invariance to your model.

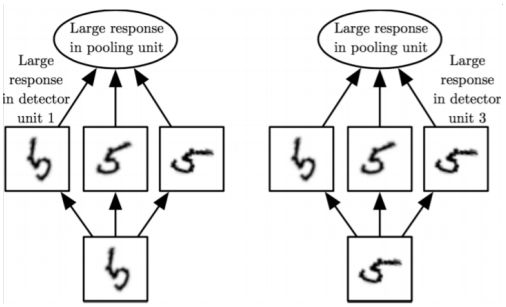


Figure 10.5: Here, we have an example where our convolutional layer is represented by 3 kernels. Suppose they were each good for detecting the number 5 in some degree of rotation. If we pooled across the three channels determined by these kernels, then no matter what orientation of the number "5" we 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 11×11 conv filters
- First use of ReLU, which fixed the problem of saturating gradients in the predominant tanh activation.

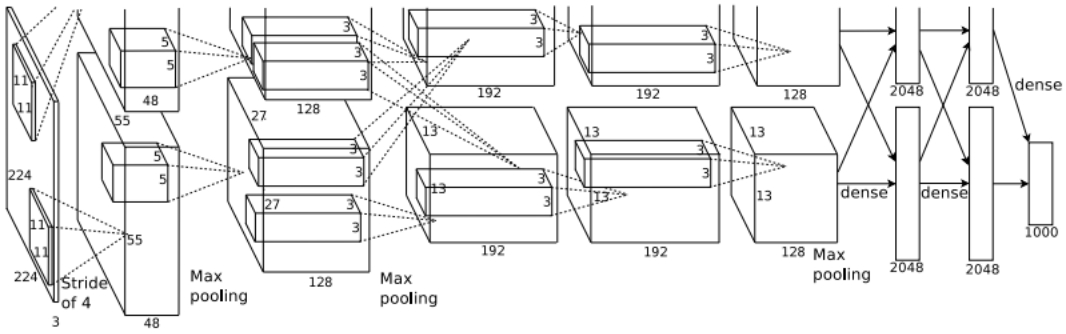


Figure 10.6: AlexNet architecture. Reference: "ImageNet Classification with Deep Convolutional Neural Networks," NIPS 2012.

- 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 10.6 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)

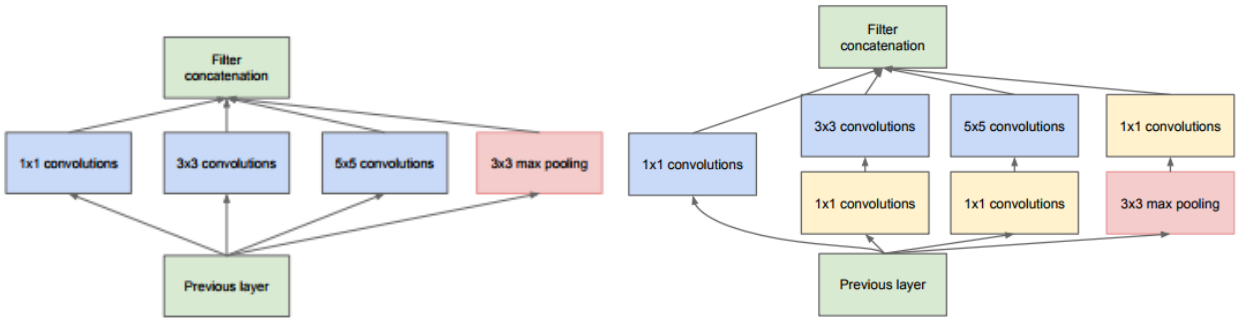
Reference paper: "Very Deep Convolutional Networks for Large-Scale Image Recognition," ICLR 2015.[1] Key characteristics:

- 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 $3 * (3^2 C)$ weights (not including bias weights), while one 7×7 filter would incur a higher cost of $7^2 C$ 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.

GoogLeNet (Szegedy et al, 2014)

Also codenamed as "Inception."[2] Published in CVPR 2015 as "Going Deeper with Convolutions." Key characteristics:



(a) Inception module, naïve version (b) Inception module with dimensionality reduction

Figure 10.7: Inception Module

- Deeper than previous networks (22 layers), but more computationally efficient (5 million parameters - no fully connected layers).
- Network is composed of stacked sub-networks called "Inception modules." The naive Inception module (a) runs convolutional layers in parallel and concatenates the filters together. However, this can be computationally inefficient. The dimensionality reduction Inception module (b) performs 1×1 convolutions that act as dimensionality reduction. This lowers the computational cost and makes it tractable to stack many Inception modules together.

ResNet (He et al, 2015)

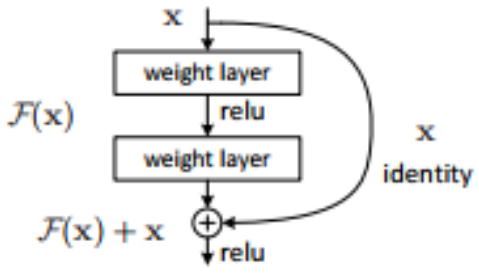


Figure 10.8: Building block for the ResNet from "Deep Residual Learning for Image Recognition," CVPR 2016. If the desired function to be learned is $\mathcal{H}(x)$, we instead learn the residual $\mathcal{F}(x) := \mathcal{H}(x) - x$, so the output of the network is $\mathcal{F}(x) + x = \mathcal{H}(x)$.

Key characteristics:

[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.

- Very deep (152 layers). Residual blocks (Figure 10.8) are stacked together - each individual weight layer in the residual block is implemented as a 3×3 convolution. There are no FC layers until the final layer.
- Residual blocks solve the "vanishing gradient" problem: the gradient signal diminishes in layers that are farther away from the end of the network. Let L be the loss, Y be the output at a layer, x be the input. Regular neural networks have gradients that look like

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial x}$$

but the derivative of Y with respect to x can be small. If we use a residual block where $Y = F(x) + x$, we have

$$\frac{\partial Y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

The $+x$ term in the residual block always provides some default gradient signal so the signal is still backpropagated to the front of the network. This allows the network to be very deep.

To conclude this section, we note that the winning ImageNet architectures have all increased in depth over the years. While both shallow and deep neural networks are known to be universal function approximators, there is growing empirical and theoretical evidence that deep neural networks can require fewer (even exponentially fewer) parameters than shallow nets to achieve the same approximation performance. There is also evidence that deep neural networks possess better generalization capabilities than their shallow counterparts. The performance, generalization, and optimization benefits of adding more layers is an ongoing component of theoretical research.

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.