Undergraduate Fundamentals of Machine Learning

The initial version of this textbook was created by **William J. Deuschle** for his senior thesis, based on his notes of CS181 during the Spring of 2017. This textbook has since been maintained by the CS181 course staff with bug fixes from many CS181 students.

Contents

1 Introduction to Machine Learning 1

1.1 What is Machine Learning? 1

1.2 What Will This Book Teach Me? 1

1.3 Our Machine Learning Framework 2

1.4 This Book's Notation 3

Mathematical and Statistical Notation 3

Textbook Specific Notation 3

2 Regression 4

2.1 Defining the Problem 4

2.2 Solution Options 4

2.2.1 K-Nearest-Neighbors 5

2.2.2 Neural Networks 5

2.2.3 Random Forests 5

2.2.4 Gradient Boosted Trees 5

2.2.5 Turning to Linear Regression 5

2.3 Introduction to Linear Regression 6

2.4 Basic Setup 6

2.4.1 Merging of Bias 7

2.4.2 Visualization of Linear Regression 7

2.5 Finding the Best Fitting Line 8

2.5.1 Objective Functions and Loss 8

2.5.2 Least Squares Loss 9

2.6 Linear Regression Algorithms 10

2.6.1 Optimal Weights via Matrix Differentiation 10

2.6.2 Bayesian Solution: Maximum Likelihood Estimation 11

2.6.3 Alternate Interpretation: Linear Regression as Projection 13

2.7 Model Flexibility 13

2.7.1 Basis Functions 13

2.7.2 Regularization 16

2.7.3 Generalizing Regularization . . . 19
2.7.4 Bayesian Regularization . . . 20
2.8 Choosing Between Models . . . 22
2.8.1 Bias-Variance Tradeoff and Decomposition . . . 22
2.8.2 Cross-Validation . . . 26
2.8.3 Making a Model Choice . . . 26
2.8.4 Bayesian Model Averaging . . . 27
2.9 Linear Regression Extras . . . 27
2.9.1 Predictive Distribution . . . 27
2.10 Conclusion . . . 28

3 Classification 29
3.1 Defining the Problem . . . 29
3.2 Solution Options . . . 29
3.3 Discriminant Functions . . . 30
3.3.1 Basic Setup: Binary Linear Classification . . . 30
3.3.2 Multiple Classes . . . 31
3.3.3 Basis Changes in Classification . . . 31
3.4 Numerical Parameter Optimization and Gradient Descent . . . 33
3.4.1 Gradient Descent . . . 35
3.4.2 Batch Gradient Descent versus Stochastic Gradient Descent . . . 36
3.5 Objectives for Decision Boundaries . . . 36
3.5.1 0/1 Loss . . . 36
3.5.2 Least Squares Loss . . . 37
3.5.3 Hinge Loss . . . 38
3.6 Probabilistic Methods . . . 39
3.6.1 Probabilistic Discriminative Models . . . 39
Logistic Regression . . . 40
Multi-Class Logistic Regression and Softmax . . . 42
3.6.2 Probabilistic Generative Models . . . 44
Classification in the Generative Setting . . . 44
MLE Solution . . . 45
Naive Bayes . . . 47
3.7 Conclusion . . . 49

4 Neural Networks 50
4.1 Motivation . . . 50
4.1.1 Comparison to Other Methods . . . 51
4.1.2 Universal Function Approximation . . . 51
4.2 Feed-Forward Networks . . . 52

4.3 Neural Network Basics and Terminology 52
4.3.1 Adaptive Basis Functions 53
4.4 Network Training 56
4.4.1 Objective Function 56
4.4.2 Optimizing Parameters 56
4.4.3 Backpropagation 57
4.4.4 Computing Derivatives Using Backpropagation 57
4.5 Choosing a Network Structure 61
4.5.1 Cross Validation for Neural Networks 61
4.5.2 Preventing Overfitting 62
Regularization 62
Data Augmentation 62
4.6 Specialized Forms of Neural Networks 63
4.6.1 Convolutional Neural Networks (CNNs) 63
4.6.2 Recurrent Neural Networks (RNNs) 63
4.6.3 Bayesian Neural Networks (BNNs) 64

5 Support Vector Machines 65
5.1 Motivation 65
5.1.1 Max Margin Methods 65
5.1.2 Applications 66
5.2 Hard Margin Classifier for Linearly Separable Data 67
5.2.1 Why the Hard Margin 67
5.2.2 Deriving our Optimization Problem 67
5.2.3 What is a Support Vector 69
5.3 Soft Margin Classifier 70
5.3.1 Why the Soft Margin? 70
5.3.2 Updated Optimization Problem for Soft Margins 71
5.3.3 Soft Margin Support Vectors 72
5.4 Conversion to Dual Form 72
5.4.1 Lagrange Multipliers 73
5.4.2 Deriving the Dual Formulation 74
5.4.3 Making Predictions 75
5.4.4 Why is the Dual Formulation Helpful? 76
5.4.5 Kernel Composition 77

6 Clustering 78
6.1 Motivation 78
6.1.1 Applications 79
6.2 K-Means Clustering 79

6.2.1 Lloyd's Algorithm . . . 79
6.2.2 Example of Lloyd's . . . 81
6.2.3 Number of Clusters . . . 84
6.2.4 Initialization and K-Means++ . . . 84
6.2.5 K-Medoids Alternative . . . 86
6.3 Hierarchical Agglomerative Clustering . . . 86
6.3.1 HAC Algorithm . . . 87
6.3.2 Linkage Criterion . . . 89
Min-Linkage Criteria . . . 89
Max-Linkage Criterion . . . 89
Average-Linkage Criterion . . . 90
Centroid-Linkage Criterion . . . 90
Different Linkage Criteria Produce Different Clusterings . . . 90
6.3.3 How HAC Differs from K-Means . . . 91

7 Dimensionality Reduction 92
7.1 Motivation . . . 92
7.2 Applications . . . 93
7.3 Principal Component Analysis . . . 93
7.3.1 Reconstruction Loss . . . 94
7.3.2 Minimizing Reconstruction Loss . . . 96
7.3.3 Multiple Principal Components . . . 97
7.3.4 Identifying Directions of Maximal Variance in our Data . . . 97
7.3.5 Choosing the Optimal Number of Principal Components . . . 98
7.4 Conclusion . . . 101

8 Graphical Models 102
8.1 Motivation . . . 102
8.2 Directed Graphical Models (Bayesian Networks) . . . 103
8.2.1 Joint Probability Distributions . . . 105
8.2.2 Generative Models . . . 106
8.2.3 Generative Modeling vs. Discriminative Modeling . . . 107
8.2.4 Understanding Complexity . . . 108
8.2.5 Independence and D-Separation . . . 109
8.3 Example: Naive Bayes . . . 113
8.4 Conclusion . . . 114

9 Mixture Models 115
9.1 Motivation . . . 115
9.2 Applications . . . 117

9.3 Fitting a Model 117
9.3.1 Maximum Likelihood for Mixture Models 117
9.3.2 Complete-Data Log Likelihood 118
9.4 Expectation-Maximization (EM) 118
9.4.1 Expectation Step 119
9.4.2 Maximization Step 120
9.4.3 Full EM Algorithm 121
9.4.4 The Math of EM 121
The Evidence Lower Bound (ELBO) 122
Optimization 123
Correctness 125
Equivalence to Prior Formulation 125
9.4.5 Connection to K-Means Clustering 125
9.4.6 Dice Example: Mixture of Multinomials 126
9.5 Gaussian Mixture Models (GMM) 128
9.6 Admixture Models: Latent Dirichlet Allocation (LDA) 129
9.6.1 LDA for Topic Modeling 129
9.6.2 Applying EM to LDA 130
9.7 Conclusion 132

10 Hidden Markov Models 133
10.1 Motivation 133
10.2 Applications 134
10.3 HMM Data, Model, and Parameterization 135
10.3.1 HMM Data 135
10.3.2 HMM Model Assumptions 135
10.3.3 HMM Parameterization 136
10.4 Inference in HMMs 136
10.4.1 The Forward-Backward Algorithm 137
10.4.2 Using α's and β's for Training and Inference 139
p(Seq) 139
Prediction 140
Smoothing 140
Transition 140
Filtering 140
Best path 141
10.5 Using EM to Train a HMM 141
10.5.1 E-Step 142
10.5.2 M-Step 142
10.6 Conclusion 143

11 Markov Decision Processes **144**

11.1 Formal Definition of an MDP . . . 145

11.2 Finite Horizon Planning . . . 146

11.3 Infinite Horizon Planning . . . 146

11.3.1 Value iteration . . . 147

Bellman Consistency Equation and Bellman Optimality . . . 147

Bellman Operator . . . 149

Value Iteration Algorithm . . . 149

11.3.2 Policy Iteration . . . 150

Policy Evaluation . . . 150

12 Reinforcement Learning **151**

12.1 Motivation . . . 151

12.2 General Approaches to RL . . . 151

12.3 Model-Free Learning . . . 152

12.3.1 SARSA and Q-Learning . . . 153

Convergence Conditions . . . 154

12.3.2 Deep Q-Networks . . . 154

12.3.3 Policy Learning . . . 155

12.4 Model-Based Learning . . . 156

12.5 Conclusion . . . 157

Chapter 1

Introduction to Machine Learning

1.1 What is Machine Learning?

There is a great deal of misunderstanding about what machine learning is, fueled by recent success and at times sensationalist media coverage. While its applications have been and will continue to be extraordinarily powerful under the right circumstances, it's important to gain some sense of where and why the tools presented in this book will be applicable. Broadly, machine learning is the application of statistical, mathematical, and numerical techniques to derive some form of knowledge from data. This 'knowledge' may afford us some sort of summarization, visualization, grouping, or even predictive power over data sets.

With all that said, it's important to emphasize the limitations of machine learning. It is not nor will it ever be a replacement for critical thought and methodical, procedural work in data science. Indeed, machine learning can be reasonably characterized a loose collection of disciplines and tools. Where the lines begin that separate machine learning from statistics or mathematics or probability theory or any other handful of fields that it draws on are not clear. So while this book is a synopsis of the basics of machine learning, it might be better understood as a collection of tools that can be applied to a specific subset of problems.

1.2 What Will This Book Teach Me?

The purpose of this book is to provide you the reader with the following: a framework with which to approach problems that machine learning learning might help solve. You will hopefully come away with a sense of the strengths and weaknesses of the tools presented within and, even more importantly, gain an understanding of how these tools are situated among problems of interest. To that end, we will aim to develop systems for thinking about the structure of the problems we work on. That way, as we continue to add new tools and techniques to our repertoire, we will always have a clear view of the context in which we can expect to use them. This will not only create a nice categorization of the different practices in machine learning, it will also help motivate why these techniques exist in the first place.

You will not be an expert in any individual ML concept after reading this text. Rather, you should come away with three different levels of understanding. First, you should gain a general contextual awareness of the different problem types that ML techniques may be used to solve. Second, you should come away with a practical awareness of how different ML techniques operate. This means that after you have successfully identified an appropriate ML technique for a given problem, you will also know how that method actually accomplishes the goal at hand. If you only come away with these two levels of understanding, you will be off to a good start. The third level of understanding relates to having a derivational awareness of the algorithms and methods we will make use of. This level of understanding is not strictly necessary to successfully interact with existing machine learning capabilities, but it will be required if you desire to go further and deepen existing knowledge. Thus, we will be presenting derivations, but it will be secondary to a high level understanding of problem types and the practical intuition behind available solutions.

1.3 Our Machine Learning Framework

Let's consider for the first time what we will call the **Machine Learning Cube**. The purpose of this cube is to describe the domain of problems we will encounter, and it will be a useful way to delineate the techniques we will apply to different types of problems. Understanding the different facets of the cube will aid you in understanding machine learning as a whole, and can even give you intuition about techniques that you have never encountered before. Let's now describe the features of the cube.

Our cube has three axes. On the first axis we will put the output domain. The domain of our output can take on one of two forms: **discrete** or **continuous**. Discrete, or categorical data, is data that can only fall into one of n specific classes. For example: male or female, integer values between 1 and 10, or different states in the U.S. are all examples of categorical data. Continuous data is that which falls on the real number line.

The second axis of the cube is reserved for the statistical nature of the machine learning technique in question. Specifically, it will fall into one of two broad categories: **probabilistic** or **non-probabilistic** techniques. Probabilistic techniques are those for which we incorporate our data using some form of statistical distribution or summary. In general, we are then able to discard some or all of our data once we have finished tuning our probabilistic model. In contrast, non-probabilistic techniques are those that use the data directly to perform some action. A very common and general example of this is comparing how close a new data point is to other points in your existing data set. Non-probabilistic techniques potentially make fewer assumptions, but they do require that you keep around some or all of your data. They are also potentially slower techniques at runtime because they may require touching all of the data in your dataset to perform some action. These are a very broad set of guidelines for the distinction between probabilistic and non-probabilistic techniques - you should expect to see some exceptions and even some techniques that fit into both of these categories to some degree. Having a sense for their general benefits and drawbacks is useful, and you will gain more intuition about the distinction as we begin to explore different techniques.

The third and final axis of the cube describes the type of training we will use. There are two major classes of machine learning techniques: **supervised** and **unsupervised**. In fact, these two classes of techniques are so important to describing the field of machine learning that we will roughly divide this textbook into two halves dedicated to techniques found within each of these categories. A supervised technique is one for which we get to observe a data set of both the inputs and the outputs ahead of time, to be used for training. For example, we might be given a data set about weather conditions and crop production over the years. Then, we could train a machine learning model that learns a relationship between the input data (weather) and output data (crop production). The implication here is that given new input data, we will be able to predict the unseen output data. An unsupervised technique is one for which we only get a data set of 'inputs' ahead of time. In fact, we don't even need to consider these as inputs anymore: we can just consider them to be a set of data points that we wish to summarize or describe. Unsupervised techniques revolve around clustering or otherwise describing our data.

We will see examples of all of these as we progress throughout the book, and you will gain an intuition for where different types of data and techniques fall in our cube. Eventually, given just the information in the cube for a new technique, you will have a solid idea of how that technique operates.

1.4 This Book's Notation

The machine learning community uses a number of different conventions, and learning to decipher the different versions of those conventions is important to understanding work done in the field. For this book, we will try to stick to a standard notation that we define here in part. In addition to mathematical and statistical notation, we will also describe the conventions used in this book for explaining and breaking up different concepts.

Mathematical and Statistical Notation

We will describe the dimensionality of variables when necessary, but generic variables will often be sufficient when explaining new techniques. Boldface variables ($\mathbf{x}$) represent vectors, capital boldface characters ($\mathbf{X}$) represent matrices, and standard typeface variables (x) describe scalars.

Statistical distributions will sometimes be described in terms of their probability density function (PDF), e.g. $Y \sim \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$. Alternatively, in the case of a well known probability distribution, we will describe those in terms of their standard notation, e.g. $Y \sim \mathcal{N}(\mu, \sigma^2)$

Textbook Specific Notation

We have also introduced a few conventions to make consumption of this material easier. We have boxes dedicated to definitions, explaining techniques in the context of the ML Framework Cube, and for explaining common misconceptions:

Definition 1.4.1 (Definition Explanation): You will find definitions in these dark gray boxes.

Derivation 1.4.1 (Derivation Explanation): You will find derivations in these light gray boxes.

ML Framework Cube: ML Framework Cube

You will find ML Framework Cube explanations in these blue wrapped boxes.

⋆ You will find explanations for subtle or confusing concepts in these red wrapped boxes.

Chapter 2

Regression

A major component of machine learning, the one that most people associate with ML, is dedicated to making predictions about a target given some inputs, such as predicting how much money an individual will earn in their lifetime given their demographic information. In this chapter, w!e're going to focus on the case where our prediction is a continuous, real number. When the target is a real number, we call this prediction procedure **regression**.

2.1 Defining the Problem

In order to understand regression, let's start in a more natural place: what types of problems are we trying to solve? What exactly does it mean to make a prediction that is a *continuous, real number*? To build some intuition, here are a few examples of problems that regression could be used for:

1. Predicting a person's height given the height of their parents.

2. Predicting the amount of time someone will take to pay back a loan given their credit history.

3. Predicting what time a package will arrive given current weather and traffic conditions.

Hopefully you are starting to see the pattern emerging here. Given some inputs, we need to produce a prediction for a continuous output. That is exactly the purpose of **regression**. Notice that regression isn't any one technique in particular. It's just a class of methods that helps us achieve our overall goal of predicting a continuous output.

Definition 2.1.1 (Regression): A class of techniques that seeks to make predictions about unknown continuous target variables given observed input variables.

2.2 Solution Options

Now that you understand the type of problems we are trying to solve with regression, we can start to think at a high level of the different ways we might arrive at a solution. Here is a short, nonexhaustive list of regression techniques with brief explanations:

2.2.1 K-Nearest-Neighbors

K-Nearest-Neighbors is an extremely intuitive, non-parametric technique for regression or classification. It works as follows in the regression case:

1. Identify the K points in our data set that are closest to the new data point. 'Closest' is some measure of distance, usually Euclidean.

2. Average the value of interest for those K data points.

3. Return that averaged value of interest: it is the prediction for our new data point.

⋆ A *non-parametric* model simply means we don't make any assumptions about the form of our data. We only need to use the data itself to make predictions.

2.2.2 Neural Networks

A neural network works by scaling and combining input variables many times. Furthermore, at each step of scaling and combining inputs, we typically apply some form of *nonlinearity* over our data values. The proof is beyond the scope of this textbook, but neural networks are known to be *universal function approximators*. This means that given enough scaling and combining steps, we can approximate any function to an arbitrarily high degree of accuracy using a neural network.

2.2.3 Random Forests

Random forests are what's known as an *ensemble method*. This means we average the results of many smaller regressors known as *decision trees* to produce a prediction. These decision trees individually operate by partitioning our original data set with respect to the value of predictive interest using a subset of the features present in that data set. Each decision tree individually may not be a reliable regressor; by combining many of them we achieve a model with better performance.

2.2.4 Gradient Boosted Trees

Gradient boosted trees are another technique built on decision trees. Assume we start with a set of decision trees that together achieve a certain level of performance. Then, we iteratively add new trees to improve performance on the hardest examples in our data set, and reweight our new set of decision trees to produce the best level of performance possible.

2.2.5 Turning to Linear Regression

You've likely never even heard of some of these preceding techniques - that's okay. The point is that we have a number of existing methods that take different approaches to solving regression problems. Each of these will have their own strengths and weaknesses that contribute to a decision about whether or not you might use them for a given regression problem. We obviously do not cover these methods in great detail here; what's more important is the understanding that there are a variety of ways to go about solving regression problems. From here on out, we will take a deeper dive into linear regression. There are several reasons for which we are exploring this specific technique in greater detail. The two main reasons are that it has been around for a long time and thus is very well understood, and it also will introduce many concepts and terms that will be utilized extensively in other machine learning topics.

2.3 Introduction to Linear Regression

In this chapter, we're specifically going to focus on **linear regression**, which means that our goal is to find some linear combination of the $x_1, ..., x_D$ input values that predict our target y.

Definition 2.3.1 (Linear Regression): Suppose we have an input $\mathbf{x} \in \mathbb{R}^D$ and a continuous target $y \in \mathbb{R}$. Linear regression determines weights $w_i \in \mathbb{R}$ that combine the values of x_i to produce y:

$$y = w_0 + w_1 x_1 + ... + w_D x_D \tag{2.1}$$

⋆ Notice w_0 in the expression above, which doesn't have a corresponding x_0 value. This is known as the *bias* term. If you consider the definition of a line $y = mx + b$, the bias term is corresponds to the intercept b. It accounts for data that has a non-zero mean.

Let's illustrate how linear regression works using an example, considering the case of 10 year old Sam. She is curious about how tall she will be when she grows up. She has a data set of parents' heights and the final heights of their children. The inputs $\mathbf{x}$ are:

$$x_1 = \text{height of mother (cm)}$$
$$x_2 = \text{height of father (cm)}$$

Using linear regression, she determines the weights $\mathbf{w}$ to be:

$$\mathbf{w} = [34, 0.39, 0.33]$$

Sam's mother is 165 cm tall and her father is 185 cm tall. Using the results of the linear regression solution, Sam solves for her expected height:

$$\text{Sam's height} = 34 + 0.39(165) + 0.33(185) = \mathbf{159.4\ cm}$$

ML Framework Cube: Linear Regression

Let's inspect the categories linear regression falls into for our ML framework cube. First, as we've already stated, linear regression deals with a **continuous** output domain. Second, our goal is to make predictions on future data points, and to construct something capable of making those predictions we first need a labeled data set of inputs and outputs. This makes linear regression a **supervised** technique. Third and finally, linear regression is **non-probabilistic**. Note that there also exist probabilistic interpretations of linear regression which we will discuss later in the chapter.

Domain	***Training***	***Probabilistic***
Continuous	Supervised	No

2.4 Basic Setup

The most basic form of linear regression is a simple weighted combination of the input variables $\mathbf{x}$, which you will often see written as:

$$f(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x_1 + ... + w_D x_D \tag{2.2}$$

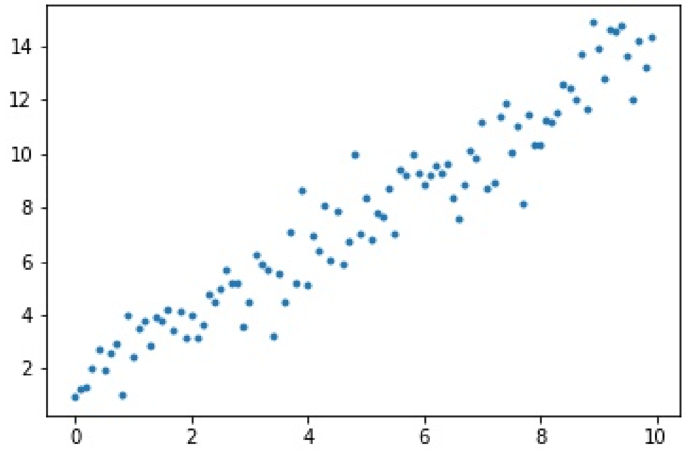


Figure 2.1: Data set with clear trend.

2.4.1 Merging of Bias

We're going to introduce a common notational trick here for making the bias term, w_0, easier to handle. At the moment w_0 is unwieldy because it is not being multiplied by an x_i value. A simple way to make our bias term easier to handle is to simply introduce another variable, x_0, that is always 1 for every data point. For example, considering the case of Sam's height from above, we have the height of her parents, $\mathbf{x}$:

$$\mathbf{x} = (165, 185)$$

We now add a 1 in the first position of the data point to make it:

$$\mathbf{x'} = (1, 165, 185)$$

We do this for every point in our data set. This bias trick lets us write:

$$f(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \mathbf{x} = w_0 x_0 + w_1 x_1 + ... + w_D x_D \tag{2.3}$$

This is more compact, easier to reason about, and makes properties of linear algebra nicer for the calculations we will be performing.

2.4.2 Visualization of Linear Regression

Let's try to build some intuition about how linear regression works. Our algorithm is given a collection of data points: inputs $\mathbf{x}$ and corresponding targets $\mathbf{y}$. Our goal is to find the best set of weights $\mathbf{w}$ such that given a new data point $\mathbf{x}$, we can accurately predict the true target value y. This is visualizable in the simple case where $\mathbf{x}$ is a 1-dimensional input variable, as in Figure 2.1.

Our eyes are naturally able to detect a very clear trend in this data. If we were given a new $\mathbf{x}^*$ data point, how would be predict its target value y? We would first fit a line to our data, as in Figure 2.2, and then find where on that line the new $\mathbf{x}^*$ value sits.

That is the entirety of linear regression! It fits the 'best' line to our data, and then uses that line to make predictions. In 1-D input space, this manifests itself as the simple problem seen above,

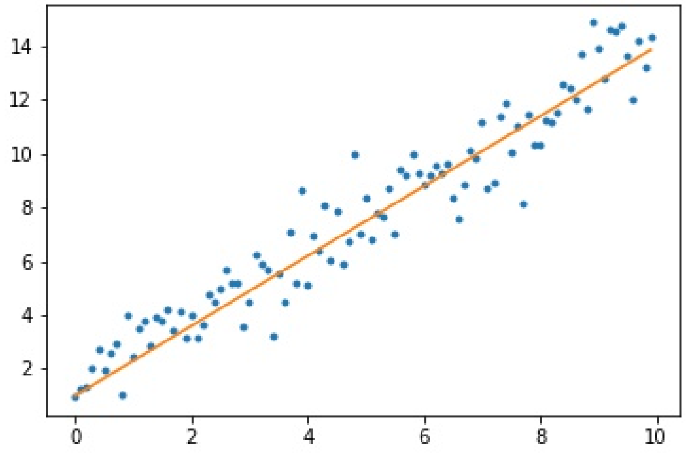


Figure 2.2: Data set with clear trend, best fitting line included.

where we need only find a single bias term w_0 (which acts as the intercept of the line) and single weight w_1 (which acts as the slope of the line). However, the same principle applies to higher dimensional data as well. We're always fitting the hyperplane that best predicts the data.

$\star$ Although our input data points $\mathbf{x}$ can take on multiple dimensions, our output data y is always a 1-dimensional real number when dealing with regression problems.

Now that we have some intuition for what linear regression is, a natural question arises: how do we find the optimal values for $\mathbf{w}$? That is the remaining focus of this chapter.

2.5 Finding the Best Fitting Line

2.5.1 Objective Functions and Loss

Now that we've defined our model as a weighted combination of our input variables, we need some way to choose our value of $\mathbf{w}$. To do this, we need an **objective function**.

Definition 2.5.1 (Objective Function): A function that measures the 'goodness' of a model. We can optimize this function to identify the best possible model for our data.

As the definition explains, the purpose of an objective function is to measure how good a specific model is. We can therefore optimize this function to find a good model. Note that in the case of linear regression, our 'model' is just a setting of our parameters $\mathbf{w}$.

An objective function will sometimes be referred to as *loss*. Loss actually measures how bad a model is, and then our goal is to minimize it. It is common to think in terms of loss when discussing linear regression, and we incur loss when the hyperplane we fit is far away from our data.

So how do we compute the loss for a specific setting of $\mathbf{w}$? To do this, we often use **residuals**.

Definition 2.5.2 (Residual): The residual is the difference between the target (y) and predicted ($y(\mathbf{x}, \mathbf{w})$) value that a model produces:

$$\text{residual} = \text{target} - \text{prediction} = y - f(\mathbf{x}, \mathbf{w}) = \boxed{y - \mathbf{w}^\top \mathbf{x}}$$

Commonly, loss is a function of the residuals produced by a model. For example, you can imagine taking the absolute value of all of the residuals and adding those up to produce a measurement of loss. This is sometimes referred to as *L1 Loss.* Or, you might square all of the residuals and then add those up to produce loss, which is called *L2 loss* or *least squares loss.* You might also use some combination of L1 and L2 loss. For the most part, these are the two most common forms of loss you will see when discussing linear regression.

When minimized, these distinct measurements of loss will produce solutions for $\mathbf{w}$ that have different properties. For example, L2 loss is not robust to outliers due to the fact that we are squaring residuals. Furthermore, L2 loss will produce only a single solution while L1 loss can potentially have many equivalent solutions. Finally, L1 loss produces unstable solutions, meaning that for small changes in our data set, we may see large changes in our solution $\mathbf{w}$.

Loss is a concept that we will come back to very frequently in the context of supervised machine learning methods. Before exploring exactly how we use loss to fit a line, let's consider least squares loss in greater depth.

2.5.2 Least Squares Loss

As we mentioned above, there are different methods for computing loss. One of the most commonly used measurements is known as **least squares or L2 loss**. Least squares, as it is often abbreviated, says that the loss for a given data point is the square of the difference between the target and predicted values:

$$\mathcal{L}_n(\mathbf{w}) = (y - \mathbf{w}^\top \mathbf{x}_n)^2 \tag{2.4}$$

⋆ The notation $\mathcal{L}_n(\mathbf{w})$ is used to indicate the loss incurred by a model $\mathbf{w}$ for a single data point $(\mathbf{x}_n, \text{y})$. $\mathcal{L}(\mathbf{w})$ indicates the loss incurred for an entire data set by the model $\mathbf{w}$. Be aware that this notation is sometimes inconsistent between different sources.

There is a satisfying statistical interpretation for using this loss function which we will explain later in this chapter, but for now it will suffice to discuss some of the properties of this loss function that make it desirable.

First, notice that it will always take on positive values. This is convenient because we can focus exclusively on minimizing our loss, and it also allows us to combine the loss incurred from different data points without worrying about them cancelling out.

A more subtle but enormously important property of this loss function is that we know a lot about how to efficiently optimize quadratic functions. This is not a textbook about optimization, but some quick and dirty intuition that we will take advantage of throughout this book is that we can easily and reliably take the derivative of quadratic functions because they are continuously differentiable. We also know that optima of a quadrative function will be located at points where the derivative of the function is equal to 0, as seen in Figure 2.3. In contrast, L1 loss is not continuously differentiable over the entirety of its domain.

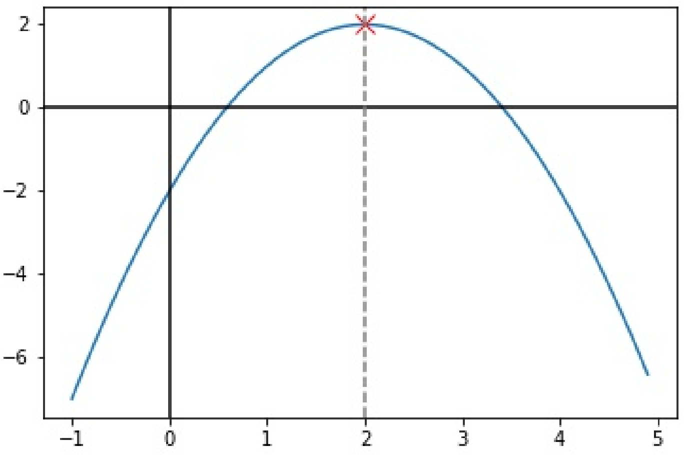


Figure 2.3: Quadratic function with clear optimum at $x = 2$, where the derivative of the function is 0.

2.6 Linear Regression Algorithms

Now that we have our least squares loss function, we can finally begin to fit a line to our data. We will walk through the derivation of how this is done, in the process revealing the algorithm used for linear regression.

2.6.1 Optimal Weights via Matrix Differentiation

Let $\mathbf{X}$ denote a $N \times D$ dimension matrix (the "design matrix") where the row n is $\mathbf{x}_n^\top$, i.e., the features corresponding to the nth example. Let $\mathbf{Y}$ denote a $N \times 1$ vector corresponding to the target values, where the nth entry corresponds to y_n.

First, we can define the loss incurred by parameters $\mathbf{w}$ over our entire data set $\mathbf{X}$ as follows:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 \tag{2.5}$$

⋆ Note that we added a constant $\frac{1}{2}$ to the beginning of our loss expression. This scales the loss, which will not change our final result for the optimal parameters. It has the benefit of making our calculations cleaner once we've taken the gradient of the loss.

We now want to solve for the values of $\mathbf{w}$ that minimize this expression.

Derivation 2.6.1 (Least Squares Optimal Weights Derivation): We find the optimal weights $\mathbf{w}^*$ as follows:

Start by taking the gradient of the loss with respect to our parameter $\mathbf{w}$:

$$\nabla\mathcal{L}(\mathbf{w}) = \sum_{n=1}^{N} (y_n - \mathbf{w}^\top \mathbf{x}_n)(-\mathbf{x}_n)$$

Setting this gradient to 0, and multiplying both sides by -1, we have:

$$\sum_{n=1}^{N} y_n \mathbf{x}_n - \sum_{n=1}^{N} (\mathbf{w}^\top \mathbf{x}_n)\mathbf{x}_n = 0. \tag{2.6}$$

Since $(\mathbf{w}^\top \mathbf{x}_n)\mathbf{x}_n = (\mathbf{x}_n^\top \mathbf{w})\mathbf{x}_n = \mathbf{x}_n(\mathbf{x}_n^\top \mathbf{w})$, recognizing that for scalar a and vector $\mathbf{v}$ we have $a^\top = a$ and $a\mathbf{v} = \mathbf{v}a$. Substituting, we want to solve for $\mathbf{w}$ such that

$$\sum_{n=1}^{N} y_n \mathbf{x}_n - \sum_{n=1}^{N} \mathbf{x}_n(\mathbf{x}_n^\top \mathbf{w}) = 0.$$

At this point, it is convenient to rewrite these summations as matrix operations, making use of design matrix $\mathbf{X}$ $(N \times D)$ and target values $\mathbf{y}$ $(N \times 1)$. We have

$$\mathbf{X}^\top \mathbf{y} = \sum_{n=1}^{N} y_n \mathbf{x}_n, \qquad \mathbf{X}^\top \mathbf{X}\mathbf{w} = \sum_{n=1}^{N} \mathbf{x}_n(\mathbf{x}_n^\top \mathbf{w})$$

Substituting, we have

$$\mathbf{X}^\top \mathbf{y} - \mathbf{X}^\top \mathbf{X}\mathbf{w} = 0.$$

Solving for the optimal weight vector, we have

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}. \tag{2.7}$$

For this to be well defined we need $\mathbf{X}$ to have full column rank (features are not colinear) so that $\mathbf{X}^\top \mathbf{X}$ is positive definite and the inverse exists.

The quantity $(\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top$ in Derivation 2.6.1 has a special name: the ***Moore-Penrose pseudo inverse***. You can think of it as a generalization of a matrix inversion operation to a non-square matrix.

2.6.2 Bayesian Solution: Maximum Likelihood Estimation

We've thus far been discussing linear regression exclusively in terms of a loss function that helps us fit a set of weights to our data. In particular, we have been working with least squares, which has nice properties that make it a reasonable loss function.

In a very satisfying fashion, least squares also has a statistical foundation. In fact, you can recover the least squares loss function purely from a statistical derivation that we present here.

Consider our data set $\mathbf{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$, $\mathbf{x}_n \in \mathbb{R}^m$, $y \in \mathbb{R}$. Let's imagine that our data was generated according to the following process:

$$y_n \sim \mathcal{N}(\mathbf{w}^\top \mathbf{x}_n, \beta^{-1})$$

Which can be written equivalently as:

$$p(y_n|\mathbf{x}_n, \mathbf{w}, \beta) = \mathcal{N}(\mathbf{w}^\top \mathbf{x}_n, \beta^{-1}) \tag{2.8}$$

The interpretation of this is that our target value y is generated according to a linear combination of our inputs $\mathbf{x}$, but there is also some noise in the data generating process described by the variance parameter β^{-1}. It's an acknowledgement that some noise exists naturally in our data.

⋆ It's common to write variance as an inverse term, such as β^{-1}. The parameter β is then known as the *precision*, which is sometimes easier to work with than the variance.

As before, we now ask the question: how do we solve for the optimal weights $\mathbf{w}$? One approach we can take is to maximize the probability of observing our target data $\mathbf{y}$. This technique is known as *maximum likelihood estimation*.

Derivation 2.6.2 (Maximum Likelihood Estimation for Bayesian Linear Regression): The likelihood of our data set is given by:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} \mathcal{N}(\mathbf{w}^\top \mathbf{x}_n, \beta^{-1})$$

We then take the logarithm of the likelihood, and since the logarithm is a strictly increasing, continuous function, this will not change our optimal weights $\mathbf{w}$:

$$\ln p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta) = \sum_{n=1}^{N} \ln \mathcal{N}(\mathbf{w}^\top \mathbf{x}_n, \beta^{-1})$$

Using the density function of a univariate Gaussian:

$$\begin{aligned}\ln p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta) &= \sum_{n=1}^{N} \ln \frac{1}{\sqrt{2\pi\beta^{-1}}} e^{-(y_n - \mathbf{w}^\top \mathbf{x}_n)^2/2\beta^{-1}} \\ &= \frac{N}{2}\ln\beta - \frac{N}{2}\ln(2\pi) - \frac{\beta}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top \mathbf{x}_n)^2\end{aligned}$$

Notice that this is a quadratic function in $\mathbf{w}$, which means that we can solve for it by taking the derivative with respect to $\mathbf{w}$, setting that expression to 0, and solving for $\mathbf{w}$:

$$\begin{aligned}\frac{\partial \ln p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta)}{\partial \mathbf{w}} &= -\beta \sum_{n=1}^{N}(y_n - \mathbf{w}^\top \mathbf{x}_n)(-\mathbf{x}_n) \\ \Leftrightarrow \quad \sum_{n=1}^{N} y_n \mathbf{x}_n &- \sum_{n=1}^{N}(\mathbf{w}^\top \mathbf{x}_n)\mathbf{x}_n = 0.\end{aligned}$$

Notice that this is exactly the same form as Equation 2.6. Solving for $\mathbf{w}$ as before, we have:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y} \tag{2.9}$$

Notice that our final solution is exactly the same form as the solution in Equation 2.7, which we solved for by minimizing the least squares loss! **The takeaway here is that minimizing a least squares loss function is equivalent to maximizing the probability under the assumption of a linear model with Gaussian noise**.

2.6.3 Alternate Interpretation: Linear Regression as Projection

Another common interpretation of linear regression is that of a projection of our targets, $\mathbf{y}$, onto the column space of our inputs $\mathbf{X}$. This can be useful for building intuition.

We showed above that the quantity $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ can be thought of as the pseudoinverse for our inputs $\mathbf{X}$. Let's now consider the case where $\mathbf{X}$ is square and the pseudoinverse is equal to the true inverse: $\mathbf{X}^{-1} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$. We have for our optimal $\mathbf{w}^*$:

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

which simplifies as

$$\mathbf{w}^* = \mathbf{X}^{-1}\mathbf{y}$$

We can recover our target values $\mathbf{y}$ by multiplying either side by $\mathbf{X}$:

$$\begin{aligned}\mathbf{X}\mathbf{w}^* &= \mathbf{X}\mathbf{X}^{-1}\mathbf{y}\\ \mathbf{X}\mathbf{w}^* &= \mathbf{y}\end{aligned}$$

We were able to recover our targets $\mathbf{y}$ exactly because $\mathbf{X}$ is an invertible tranformation. However, in the general case where $\mathbf{X}$ is not invertible and we have to use the approximate pseudoinverse $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$, we instead recover $\hat{\mathbf{y}}$:

$$\begin{aligned}\mathbf{X}\mathbf{w}^* &= \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}\\ \mathbf{X}\mathbf{w}^* &= \hat{\mathbf{y}}\end{aligned}$$

where $\hat{\mathbf{y}}$ can be thought of as the closest projection of $\mathbf{y}$ onto the column space of $\mathbf{X}$.

Furthermore, this motivates the intuition that $\mathbf{w}^*$ is the set of coefficients that best transforms our input space $\mathbf{X}$ into our target values $\mathbf{y}$.

2.7 Model Flexibility

Occasionally, linear regression will fail to recover a good solution for a data set. While this may be because our data doesn't actually have predictive power, it might also just indicate that our data is provided in a format unsuitable for linear regression. This section explores that problem, in particular focusing on how we can manipulate the flexibility of our models to make them perform better.

2.7.1 Basis Functions

There are some situations where linear regression is not the best choice of model for our input data $\mathbf{X}$. Because linear regression only scales and combines input variables, it is unable to apply more complex transformations to our data such as a *sin* or square root function. In those situations where we need to transform our input variable somehow prior to performing linear regression (which is known as moving our data into a new *basis*), we can apply a **basis function**.

Definition 2.7.1 (Basis Function): Typically denoted by the symbol $\phi(\cdot)$, a basis function is a transformation applied to an input data point $\mathbf{x}$ to move our data into a different *input basis*, which is another phrase for *input domain*.

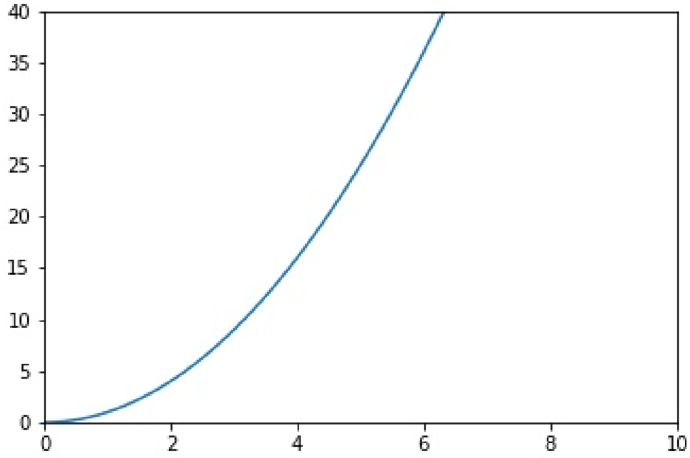


Figure 2.4: Data with no basis function applied.

For example, consider our original data point:

$$\mathbf{x} = (x^{(1)}, x^{(2)})'$$

We may choose our basis function $\phi(\mathbf{x})$ such that our transformed data point in its new basis is:

$$\phi(\mathbf{x}) = (x^{(1)}, {x^{(1)}}^2, x^{(2)}, \sin(x^{(2)}))'$$

Using a basis function is so common that we will sometimes describe our input data points as $\boldsymbol{\phi} = (\phi^{(1)}, \phi^{(2)}, ..., \phi^{(D)})'$.

⋆ The notation $\mathbf{x} = (x^{(1)}, x^{(2)})'$ is a way to describe the dimensions of a single data point $\mathbf{x}$. The term $\mathbf{x}^{(1)}$ is the first dimension of a data point $\mathbf{x}$, while $\mathbf{x}_1$ is the first data point in a data set.

Basis functions are very general - they could specify that we just keep our input data the same. As a result, it's common to rewrite the least squares loss function from Equation 2.4 for linear regression in terms of the basis function applied to our input data:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top \boldsymbol{\phi}_n)^2 \tag{2.10}$$

To motivate why we might need basis functions for performing linear regression, let's consider this graph of 1-dimensional inputs $\mathbf{X}$ along with their target outputs $\mathbf{y}$, presented in Figure 2.4.

As we can see, we're not going to be able to fit a good line to this data. The best we can hope to do is something like that of Figure 2.5.

However, if we just apply a simple basis function to our data, in this case the square root function, $\phi(\mathbf{x}) = (\sqrt{x_1})'$, we then have the red line in Figure 2.6. We now see that we can fit a very good line to our data, thanks to basis functions.

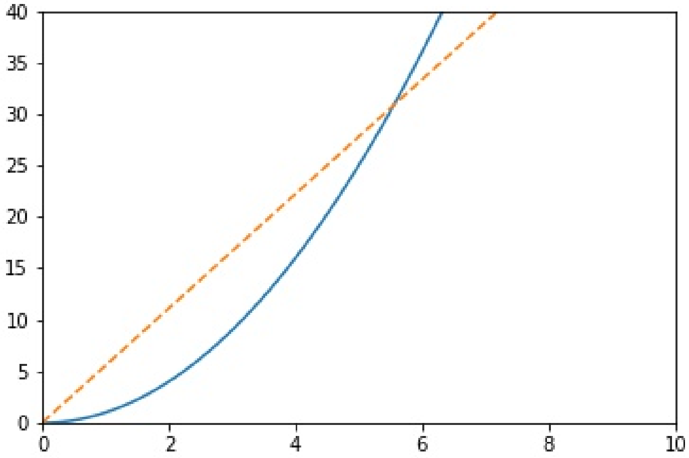


Figure 2.5: Data with no basis function applied, attempt to fit a line.

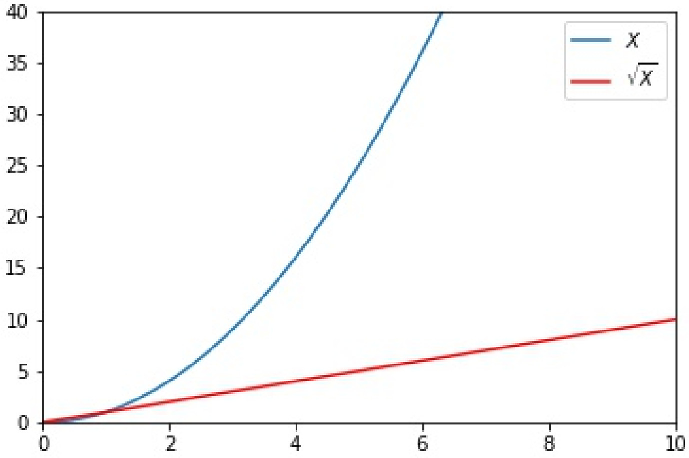


Figure 2.6: Data with square root basis function applied.

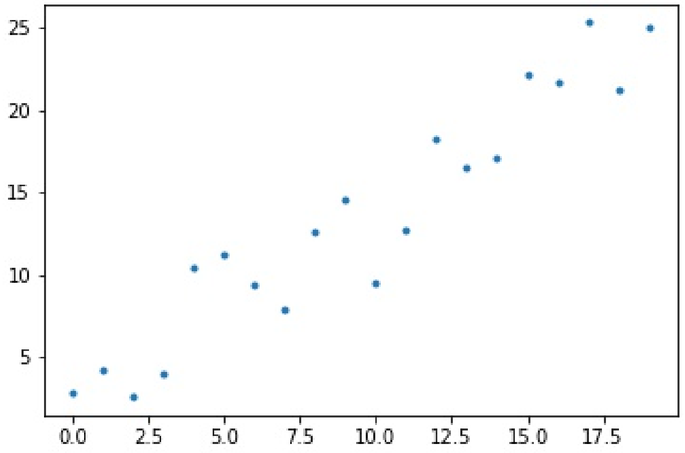


Figure 2.7: Data set with a clear trend and Gaussian noise.

Still, the logical question remains: how can I choose the appropriate basis function? This toy example had a very obviously good basis function, but in general with high-dimensional, messy input data, how do we choose the basis function we need?

The answer is that this is not an easy problem to solve. Often, you may have some domain specific knowledge that tells you to try a certain basis, such as if you're working with chemical data and know that an important equation involves a certain function of one of your inputs. However, more often than not we won't have this expert knowledge either. Later, in the chapter on neural networks, we will discuss methods for discovering the best basis functions for our data automatically.

2.7.2 Regularization

When we introduced the idea of basis functions above, you might have wondered why we didn't just try adding many basis transformations to our input data to find a good transformation. For example, we might use this large basis function on a D-dimensional data point $\mathbf{z}$:

$$\phi(\mathbf{z}) = ({z^{(1)}}, {z^{(1)}}^2, ..., {z^{(1)}}^{100}, {z^{(2)}}, {z^{(2)}}^2, ..., {z^{(2)}}^{100}, ..., {z^{(D)}}, {z^{(D)}}^2, ..., {z^{(D)}}^{100})'$$

where you can see that we expand the dimensions of the data point to be 100 times its original size.

Let's say we have an input data point $\mathbf{x}$ that is 1-dimensional, and we apply the basis function described above, so that after the transformation each data point is represented by 100 values. Say we have 100 data points on which to perform linear regression, and because our transformed input space has 100 values, we have 100 parameters to fit. In this case, with one parameter per data point, it's possible for us to fit our regression line perfectly to our data so that we have no loss! But is this a desirable outcome? The answer is no, and we'll provide a visual example to illustrate that.

Imagine Figure 2.7 is our data set. There is a very clear trend in this data, and you would likely draw a line that looks something like that of Figure 2.8 to fit it.

However, imagine we performed a large basis transformation like the one described above. If we do that, it's possible for us to fit our line perfectly, threading every data point, like that in Figure 2.9.

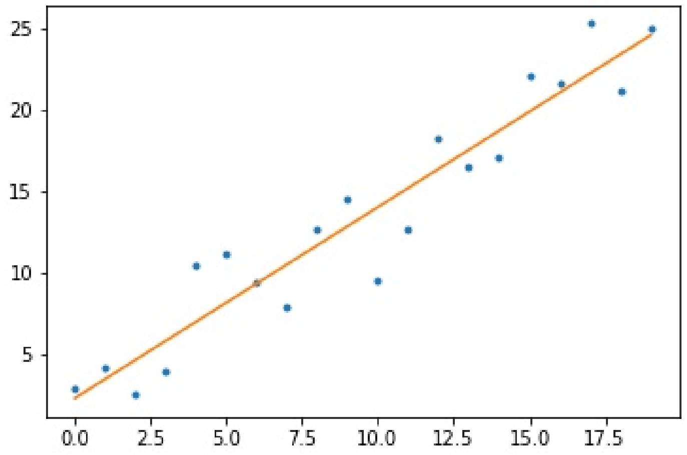


Figure 2.8: Natural fit for this data set.

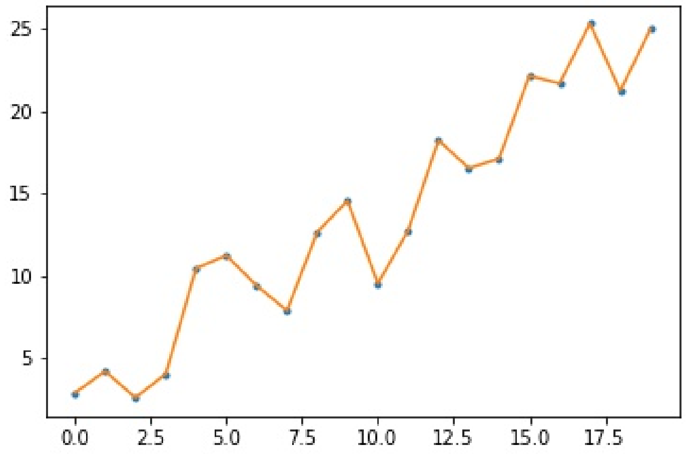


Figure 2.9: Unnatural fit for this data set.

Let's see how both of these would perform on new data points. With our first regression line, if we have a new data point $\mathbf{x} = (10)'$, we would predict a target value of **14.1**, which most people would agree is a pretty good measurement. However, with the second regression line, we would predict a value of **9.5**, which most people would agree does not describe the general trend in the data. So how can we handle this problem elegantly?

Examining our loss function, we see that right now we're only penalizing predictions that are not correct in training. However, what we ultimately care about is doing well on new data points, not just our training set. This leads us to the idea of **generalization**.

Definition 2.7.2 (Generalization): Generalization is the ability of a model to perform well on new data points outside of the training set.

A convoluted line that matches the noise of our training set exactly isn't going to generalize well to new data points that don't look exactly like those found in our training set. If wish to avoid recovering a convoluted line as our solution, we should also penalize the total size of our weights $\mathbf{w}$. The effect of this is to discourage many complex weight values that produce a messy regression line. By penalizing large weights, we favor simple regression lines like the one in Figure 2.8 that take advantage of only the most important basis functions.

The concept that we are introducing, penalizing large weights, is an example of what's known as **regularization**, and it's one that we will see come up often in different machine learning methods.

Definition 2.7.3 (Regularization): Applying penalties to parameters of a model.

There is obviously a tradeoff between how aggressively we regularize our weights and how tightly our solution fits to our data, and we will formalize this tradeoff in the next section. However, for now, we will simply introduce a regularization parameter λ to our least squares loss function:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top \boldsymbol{\phi}_n)^2 + \frac{\lambda}{2}\mathbf{w}^\top \mathbf{w} \tag{2.11}$$

The effect of λ is to penalize large weight parameters. The larger λ is, the more we will favor simple solutions. In the limit $\lim_{\lambda \to \infty} \mathcal{L}(\mathbf{w})$, we will drive all weights to 0, while with a nonexistant $\lambda = 0$ we will apply no regularization at all. Notice that we're squaring our weight parameters - this is known as *L2 norm regularization* or **ridge regression**. While L2 norm regularization is very common, it is just one example of many ways we can perform regularization.

To build some intuition about the effect of this regularization parameter, examine Figure 2.10. Notice how larger values of λ produce less complex lines, which is the result of applying more regularization. This is very nice for the problem we started with - needing a way to choose which basis functions we wanted to use. With regularization, we can select many basis functions, and then allow regularization to 'prune' the ones that aren't meaningful (by driving their weight parameters to 0). While this doesn't mean that we should use as many basis transformations as possible (there will be computational overhead for doing this), it does allow us to create a much more flexible linear regression model without creating a convoluted regression line.

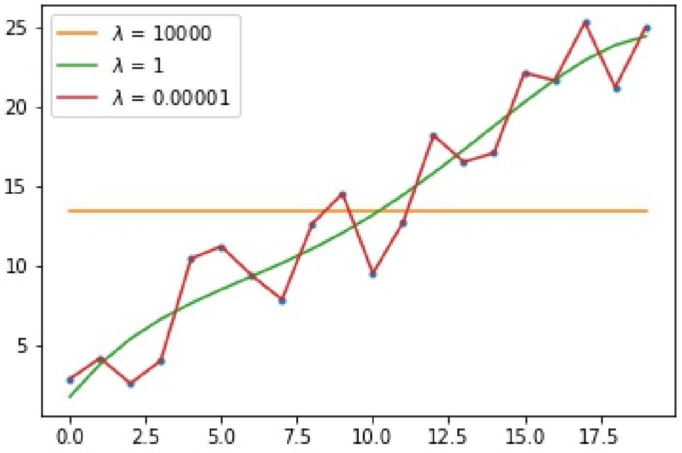


Figure 2.10: Effect of different regularization parameter values on final regression solution.

2.7.3 Generalizing Regularization

We've thus far only discussed one form of regularization: ridge regression. Remember that under ridge regression, the loss function takes the form:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top \boldsymbol{\phi}_n)^2 + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w},$$

where the $(\lambda/2)\mathbf{w}^\top\mathbf{w}$ term is for the regularization. We can generalize our type of regularization by writing it as:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top \boldsymbol{\phi}_n)^2 + \frac{\lambda}{2}||\mathbf{w}||_h^h$$

where h determines the type of regularization we are using and thus the form of the optimal solution that we recover. For example, if $h = 2$ then we add $\lambda/2$ times the square of the L2 norm. The three most commonly used forms of regularization are lasso, ridge, and elastic net.

Ridge Regression
This is the case of $h = 2$, which we've already discussed, but what type of solutions does it tend to recover? Ridge regression prevents any individual weight from growing too large, providing us with solutions that are generally moderate.

Lasso Regression
Lasso regression is the case of $h = 1$. Unlike ridge regression, lasso regression will drive some parameters w_i to zero if they aren't informative for our final solution. Thus, lasso regression is good if you wish to recover a sparse solution that will allow you to throw out some of your basis functions. You can see the forms of ridge and lasso regression functions in Figure 2.11. If you think about how Lasso is L1 Norm (absolute value) and Ridge is L2 Norm (squared distance), you can think of those shapes as being the set of points (w_1, w_2) for which the norm takes on a constant

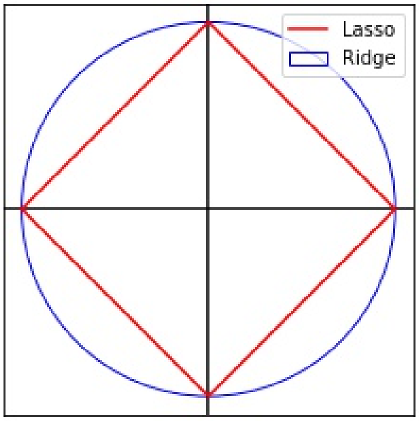


Figure 2.11: Form of the ridge (blue) and lasso (red) regression functions.

value.

Elastic Net

Elastic net is a middle ground between ridge and lasso regression, which it achieves by using a linear combination of the previous two regularization terms. Depending on how heavily each regularization term is weighted, this can produce results on a spectrum between lasso and ridge regression.

2.7.4 Bayesian Regularization

We've seen regularization in the context of loss functions, where the goal is to penalize large weight values. How does the concept of regularization apply to Bayesian linear regression?

The answer is that we can interpret regularizing our weight parameters as adding a prior distribution over $\mathbf{w}$. Note that this is a different conception of regularization than we saw in the previous section. In the Bayesian framework, we are averaging over different models specified by different values of $\mathbf{w}$. Therefore, in this context regularization entails weighting models with smaller values of $\mathbf{w}$ more heavily.

Derivation 2.7.1 (Bayesian Regularization Derivation): Because we wish to shrink our weight values toward 0 (which is exactly what regularization does), we will select a Normal prior with mean 0 and variance $\mathbf{S}_0^{-1}$:

$$\mathbf{w} \sim \mathcal{N}(0, \mathbf{S}_0^{-1})$$

Remember from Equation 2.8 that the distribution over our observed data is Normal as well, written here in terms of our entire data set:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}, \beta) = \mathcal{N}(\mathbf{X}\mathbf{w}, \beta^{-1}\mathbf{I})$$

We want to combine the likelihood and the prior to recover the posterior distribution of $\mathbf{w}$, which follows directly from Bayes' Theorem:

$$\underbrace{p(\mathbf{w}|\mathbf{X},\mathbf{y},\beta)}_{\text{posterior}} \propto \underbrace{p(\mathbf{y}|\mathbf{X},\mathbf{w},\beta)}_{\text{likelihood}}\underbrace{p(\mathbf{w})}_{\text{prior}}$$

We now wish to find the value of $\mathbf{w}$ that maximizes the posterior distribution. We can maximize the log of the posterior with respect to $\mathbf{w}$, which simplifies the problem slightly:

$$\ln p(\mathbf{w}|\mathbf{X},\mathbf{y},\beta) = \ln p(\mathbf{y}|\mathbf{X},\mathbf{w},\beta) + \ln p(\mathbf{w}) + C_1$$

where C_1 is some constant free of $\mathbf{w}$. Let's handle $\ln p(\mathbf{y}|\mathbf{X},\mathbf{w},\beta)$ first:

$$\begin{aligned}\ln p(\mathbf{y}|\mathbf{X},\mathbf{w},\beta) &= \ln \prod_{n=1}^{N} \mathcal{N}(y_n|\mathbf{w}^\top\mathbf{x}_n, \beta^{-1}) \\ &= \ln \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\beta^{-1}}} \exp\left\{ -\frac{\beta}{2}(y_n - \mathbf{w}^\top\mathbf{x}_n)^2 \right\} \\ &= C_2 - \frac{\beta}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top\mathbf{x}_n)^2 + \frac{1}{\sqrt{2\pi\beta^{-1}}}\end{aligned}$$

where C_2 collects more constant terms that don't depend on $\mathbf{w}$. Let's now handle $\ln p(\mathbf{w})$:

$$\begin{aligned}\ln p(\mathbf{w}) &= \ln \mathcal{N}(0, \mathbf{S}_0^{-1}) \\ &= \ln \frac{1}{(|2\pi\mathbf{S}_0^{-1}|)^{\frac{1}{2}}} \exp\left\{ -\frac{1}{2}\mathbf{w}^\top\mathbf{S}_0\mathbf{w} \right\} \\ &= C_3 - \frac{1}{2}\mathbf{w}^\top\mathbf{S}_0\mathbf{w}\end{aligned}$$

combining the terms for $\ln p(\mathbf{y}|\mathbf{X},\mathbf{w},\beta)$ and $\ln p(\mathbf{w})$:

$$\begin{aligned}\ln p(\mathbf{w}|\mathbf{X},\mathbf{y},\beta) &= -\frac{\beta}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top\mathbf{x}_n)^2 - \frac{1}{2}\mathbf{w}^\top\mathbf{S}_0\mathbf{w} + C \\ &= -\frac{1}{\beta}\left(\frac{1}{2}\sum_{n=1}^{N}(y_n - \mathbf{w}^\top\mathbf{x}_n)^2 + \frac{1}{2\beta}\mathbf{w}^\top\mathbf{S}_0\mathbf{w} \right) + C\end{aligned}$$

Since $\beta > 0$, notice that maximizing the posterior probability with respect to $\mathbf{w}$ is equivalent to *minimizing* the sum of squared errors $(y_n - \mathbf{w}^\top\mathbf{x}_n)^2$ and the regularization term $\frac{1}{\beta}\mathbf{w}^\top\mathbf{S}_0\mathbf{w}$.

The interpretation of this in the case $\mathbf{S}_0 = s\mathbf{I}$ is that adding a prior over the distribution of our weight parameters $\mathbf{w}$ and then maximizing the resulting posterior distribution is equivalent to ridge regression with $\lambda = \frac{s}{\beta}$, as the regularization term simplifies to $\frac{s}{\beta}\mathbf{w}^\top\mathbf{S}_0\mathbf{w}$. Recall that both s and β are precisions. This states that if the precision on the observations is small, then the regularization term is relatively large, meaning our posterior leans more towards our prior. If the precision on our prior is small, then the regularization is small, meaning our posterior leans more towards the data.

2.8 Choosing Between Models

2.8.1 Bias-Variance Tradeoff and Decomposition

Now that you know about regularization, you might have some intuition for why we need to find a balance between complex and simple regression solutions. A complex solution, while it might fit all of our training data, may not generalize well to future data points. On the other hand, a line that is too simple might not vary enough to provide good predictions at all. This phenomenon is not unique to linear regression- it's actually a very fundamental concept in machine learning that's known as the **bias-variance tradeoff**.

Definition 2.8.1 (Bias-Variance Tradeoff): When constructing machine learning models, we have a choice somewhere on a spectrum between two extremes: fitting exactly to our training data or not varying in response to our training data at all. The first extreme, fitting all of our training data, is a situation of high *variance*, because our output changes heavily in reponse to our input data (see the red line in Figure 2.10). At the other extreme, a solution that doesn't change in response to our training data at all is a situation of high *bias* (see the yellow line in Figure 2.10). This means our model heavily favors a specific form regardless of the training data, so our target outputs don't fluctuate between distinct training sets.

Obviously a good solution will fall somewhere in between these two extremes of high variance and high bias. Indeed, we have techniques like regularization to help us balance the two extremes (improving generalization), and we have other techniques like *cross-validation* that help us determine when we have found a good balance (measuring generalization).

⋆ In case you are not familiar with the terms *bias* and *variance*, we provide their statistical definitions here:

$$\text{bias}(\theta) = \text{E}[\theta] - \theta$$

$$\text{variance}(\theta) = \text{E}[(\theta - \text{E}[\theta])^2]$$

Before we discuss how to effectively mediate between these opposing forces of error in our models, we will first show that the bias-variance tradeoff is not only conceptual but also has probabilistic underpinnings. Specifically, any loss that we incur over our training set using a given model can be described in terms of bias and variance, as we will demonstrate now.

Derivation 2.8.1 (Bias-Variance Decomposition): Let's begin by asserting that we have a model $f(\cdot)$ that makes a prediction of our target y given input data point $\mathbf{x}$. We wish to break down the squared error of f into terms involving bias and variance.

Start with the expected squared error (MSE), where the expectation is taken with respect to both our data set $\mathbf{D}$ (variation in our modeling error comes from what data set we get), which is a random variable of ($\mathbf{x}$, y) pairs sample from a distribution F, and our conditional distribution $y|\mathbf{x}$ (there may be additional error because the data are noisy):

$$MSE = \text{E}_{\mathbf{D},y|x}[(y - f_{\mathbf{D}}(\mathbf{x}))^2]$$

where we use the notation $f_{\mathbf{D}}$ to explicitly acknowledge the dependence of our fitted model f on the dataset $\mathbf{D}$. For reasons that will become clear in a few steps, add and subtract our target mean

$\bar{y}$, which is the true conditional mean given by $\bar{y} = \mathrm{E}_{y|\mathbf{x}}[y]$, inside of the squared term:

$$MSE = \mathrm{E}_{\mathbf{D},y|x}[(y - \bar{y} + \bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2]$$

Group together the first two terms and the last two terms:

$$MSE = \mathrm{E}_{\mathbf{D},y|x}[((y - \bar{y}) + (\bar{y} - f_{\mathbf{D}}(\mathbf{x})))^2]$$

Expanding this expression and using linearity of expectation:

$$MSE = \mathrm{E}_{\mathbf{D},y|x}[(y - \bar{y})^2] + \mathrm{E}_{\mathbf{D},y|x}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] + 2\mathrm{E}_{\mathbf{D},y|x}[(y - \bar{y})(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))] \tag{2.12}$$

Let's examine the last term, $2\mathrm{E}[(y - \bar{y})(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))]$. Notice that $(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))$ does not depend on the conditional distribution $y|\mathbf{x}$ at all. Thus, we are able to move one of those expecations in, which makes this term:

$$2\mathrm{E}_{\mathbf{D},y|x}[(y - \bar{y})(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))] = 2\mathrm{E}_{\mathbf{D}}[\mathrm{E}_{y|\mathbf{x}}[(y - \bar{y})](\bar{y} - f_{\mathbf{D}}(\mathbf{x}))]$$

And note that:

$$\mathrm{E}_{y|\mathbf{x}}[(y - \bar{y})] = 0$$

Which eliminates this last term entirely:

$$2\mathrm{E}_{\mathbf{D},y|x}[(y - \bar{y})(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))] = 2\mathrm{E}_{\mathbf{D}}[0 \cdot (\bar{y} - f_{\mathbf{D}}(\mathbf{x}))] = 0$$

We can now write Equation 2.12 as:

$$\begin{aligned} MSE &= \mathrm{E}_{\mathbf{D},y|x}[(y - \bar{y})^2] + \mathrm{E}_{\mathbf{D},y|x}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] \\ &= \mathrm{E}_{y|x}[(y - \bar{y})^2] + \mathrm{E}_{\mathbf{D}}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] \end{aligned} \tag{2.13}$$

where we have removed expectations that do not apply (e.g. $\bar{y}$ does not depend on the dataset $\mathbf{D}$). We now have two terms contributing to our squared error. We will put aside the first term $\mathrm{E}_{y|x}[(y - \bar{y})^2]$, as this is unidentifiable *noise* in our data set. In other words, our data will randomly deviate from the mean in ways we cannot predict. On the other hand, we can work with the second term $\mathrm{E}_{\mathbf{D}}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2]$ as it involves our model function $f(\cdot)$

As before, for reasons that will become clear in a few steps, let's add and subtract our prediction mean $\bar{f}(\cdot) = \mathrm{E}_{\mathbf{D}}[f_{\mathbf{D}}(\mathbf{x})]$, which is the expectation of our model function taken with respect to our random data set.

$$\mathrm{E}_{\mathbf{D}}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] = \mathrm{E}_{\mathbf{D}}[(\bar{y} - \bar{f}(\mathbf{x}) + \bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))^2]$$

Expanding this squared term, we have:

$$\mathrm{E}_{\mathbf{D}}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] = (\bar{y} - \bar{f}(\mathbf{x}))^2 + \mathrm{E}_{\mathbf{D}}[(\bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))^2] + 2\mathrm{E}_{\mathbf{D}}[(\bar{y} - \bar{f}(\mathbf{x}))(\bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))]$$

As before, the third term here is 0:

$$2\mathrm{E}_{\mathbf{D}}[(\bar{y} - \bar{f}(\mathbf{x}))(\bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))] = 2(\bar{y} - \bar{f}(\mathbf{x}))\mathrm{E}_{\mathbf{D}}[(\bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))] = 2(\bar{y} - \bar{f}(\mathbf{x}))(0) = 0$$

Leaving us with these two terms:

$$\mathrm{E}_{\mathbf{D}}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] = (\bar{y} - \bar{f}(\mathbf{x}))^2 + \mathrm{E}_{\mathbf{D}}[(\bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))^2]$$

Notice the form of these two terms. The first one, $(\bar{y} - \bar{f}(\mathbf{x}))^2$, is the squared *bias* of our model, since it is the square of the average difference between our prediction and the true target value. The second one, $\mathrm{E}_{\mathbf{D}}[(\bar{f}(\mathbf{x}) - f_{\mathbf{D}}(\mathbf{x}))^2]$, is the *variance* of our model, since it is the expected squared difference between our model and its average value. Thus:

$$\mathrm{E}_{\mathbf{D}}[(\bar{y} - f_{\mathbf{D}}(\mathbf{x}))^2] = \mathit{bias}(f(\mathbf{x}))^2 + \mathit{variance}(f(\mathbf{x}))$$

Thus, our total squared error, plugging in to Equation 2.13 can be written as:

$$\boxed{\mathit{MSE} = \mathit{noise}(\mathbf{x}) + \mathit{bias}(f(\mathbf{x}))^2 + \mathit{variance}(f(\mathbf{x}))}$$

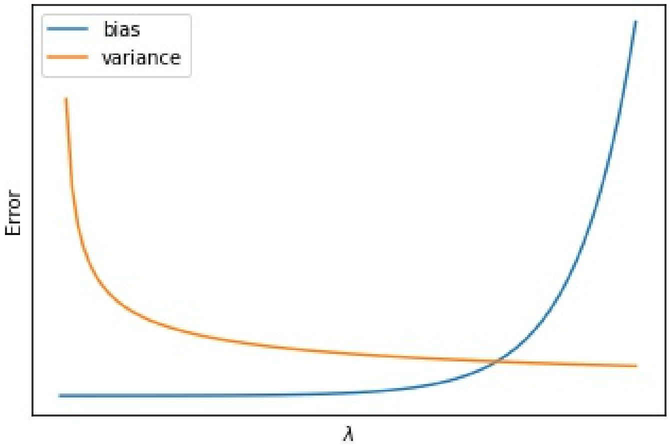


Figure 2.12: Bias and variance both contribute to the overall error of our model.

The key takeaway of the bias-variance decomposition is that the controllable error in our model is given by the squared bias and variance. Holding our error constant, to decrease bias requires increasing the variance in our model, and vice-versa. In general, a graph of the source of error in our model might look something like Figure 2.12.

For a moment, consider what happens on the far left side of this graph. Our variance is very high, and our bias is very low. In effect, we're fitting perfectly to all of the data in our data set. This is exactly why we introduced the idea of regularization from before - we're fitting a very convoluted line that is able to pass through all of our data but which doesn't generalize well to new data points. There is a name for this: **overfitting**.

Definition 2.8.2 (Overfitting): A phenomenon where we construct a convoluted model that is able to predict every point in our data set perfectly but which doesn't generalize well to new data points.

The opposite idea, **underfitting**, is what happens at the far right of the graph: we have high bias and aren't responding to the variation in our data set at all.

Definition 2.8.3 (Underfitting): A phenomenon where we construct a model that doesn't respond to variation in our data.

So you can hopefully now see that the bias-variance tradeoff is important to managing the problem of overfitting and underfitting. Too much variance in our model and we'll overfit to our data set. Too much bias and we won't account for the trends in our data set at all.

In general, we would like to find a sweet spot of moderate bias and variance that produces minimal error. In the next section, we will explore how we find this sweet spot.

2.8.2 Cross-Validation

We've seen that in choosing a model, we incur error that can be described in terms of bias and variance. We've also seen that we can regulate the source of error through regularization, where heavier regularization increases the bias of our model. A natural question then is how do we know how much regularization to apply to achieve a good balance of bias and variance?

Another way to look at this is that we've traded the question of finding the optimal number of basis functions for finding the optimal value of the regularization parameter λ, which is often an easier problem in most contexts.

One very general technique for finding the sweet spot of our regularization parameter, other hyperparameters, or even for choosing among entirely different models is known as **cross-validation**.

Definition 2.8.4 (Cross-Validation): A subsampling procedure used over a data set to tune hyperparameters and avoid over-fitting. Some portion of a data set (10-20% is common) is set aside, and training is performed on the remaining, larger portion of data. When training is complete, the smaller portion of data left out of training is used for testing. The larger portion of data is sometimes referred to as the *training set*, and the smaller portion is sometimes referred to as the *validation set*.

Cross-validation is often performed more than once for a given setting of hyperparameters to avoid a skewed set of validation data being selected by chance. In **K-Folds cross-validation**, you perform cross-validation $\mathbf{K}$ times, allocating $\frac{1}{\mathbf{K}}$ of your data for the validation set at each iteration.

Let's tie this back into finding a good regularization parameter. For a given value of λ, we will incur a certain amount of error in our model. We can measure this error using cross-validation, where we train our model on the training set and compute the final error using the validation set. To find the optimal value for λ, we perform cross-validation using different values of λ, eventually settling on the value that produces the lowest final error. This will effectively trade off bias and variance, finding the value of λ that minimizes the total error.

You might wonder why we need to perform cross-validation at all - why can't we train on the entire data set and then compute the error over the entire data set as well?

The answer is again overfitting. If we train over the entire data set and then validate our results on the exact same data set, we are likely to choose a regularization parameter that encourages our model to conform to the exact variation in our data set instead of finding the generalizable trends. By training on one set of data, and then validating on a completely different set of data, we force our model to find good generalizations in our data set. This ultimately allows us to pick the regularization term λ that finds the sweet spot between bias and variance, overfitting and underfitting.

2.8.3 Making a Model Choice

Now that we're aware of overfitting, underfitting, and how those concepts relate to the bias-variance tradeoff, we still need to come back to the question of how we actually select a model. Intuitively, we are trying to find the middle ground between bias and variance: picking a model that fits our data but that is also general enough to perform well on yet unseen data. Furthermore, there is no such thing as the 'right' model choice. Instead, there are only model options that are either better or worse than others. To that end, it can be best to rely on the techniques presented above, specifically cross-validation, to make your model selection. Then, although you will not be able to make any sort of guarantee about your selection being the 'best' of all possible models, you can at least have confidence your model achieved the best generalizability that could be proven through cross-validation.

2.8.4 Bayesian Model Averaging

We can also handle model selection using a Bayesian approach. This means we account for our uncertainty about the true model by averaging over the possible candidate models, weighting each model by our prior certainty that it is the one producing our data. If we have M models indexed by $m = 1, ..., M$, we can write the likelihood of observing our data set $\mathbf{X}$ as follows:

$$p(\mathbf{X}) = \sum_{m=1}^{M} p(\mathbf{X}|m)p(m)$$

where $p(m)$ is our prior certainty for a given model and $p(\mathbf{X}|m)$ is the likelihood of our data set given that model. The elegance of this approach is that we don't have to pick any particular model, instead choosing to marginalize out our uncertainty.

2.9 Linear Regression Extras

With most of linear regression under our belt at this point, it's useful to drill down on a few concepts to come to a deeper understanding of how we can use them in the context of linear regression and beyond.

2.9.1 Predictive Distribution

Remaining in the setting of Bayesian Linear Regression, we may wish to get a distribution over our weights $\mathbf{w}$ instead of a point estimator for it using maximum likelihood. As we saw in Section 2.7.4, we can introduce a prior distribution over $\mathbf{w}$, then together with our observed data, we can produce a posterior distribution over $\mathbf{w}$ as desired.

Derivation 2.9.1 (Posterior Predictive Derivation): For the sake of simplicity and ease of use, we will select our prior over $\mathbf{w}$ to be a Normal distribution with mean $\boldsymbol{\mu}_0$ and variance $\boldsymbol{S}_0^{-1}$:

$$p(\mathbf{w}) = \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{S}_0^{-1})$$

Remembering that the observed data is normally distributed, and accounting for Normal-Normal conjugacy, our posterior distribution will be Normal as well:

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}, \beta) = \mathcal{N}(\boldsymbol{\mu}_N, \boldsymbol{S}_N^{-1})$$

where

$$\boldsymbol{S}_N = (\boldsymbol{S}_0^{-1} + \beta\mathbf{X}^\top\mathbf{X})^{-1}$$
$$\boldsymbol{\mu}_N = \boldsymbol{S}_N(\boldsymbol{S}_0^{-1}\boldsymbol{\mu}_0 + \beta\mathbf{X}\mathbf{y})$$

We now have a posterior distribution over $\mathbf{w}$. However, usually this distribution is not what we care about. We're actually interested in making a point prediction for the target y^* given a new input $\mathbf{x}^*$. How do we go from a posterior distribution over $\mathbf{w}$ to this prediction?

The answer is using what's known as the **posterior predictive** over y^* given by:

$$\begin{aligned} p(y^*|\mathbf{x}^*, \mathbf{X}, \mathbf{y}) &= \int_{\mathbf{w}} p(y^*|\mathbf{x}^*, \mathbf{w}) p(\mathbf{w}|\mathbf{X}, \mathbf{y}) d\mathbf{w} \\ &= \int_{\mathbf{w}} \mathcal{N}(y^*|\mathbf{w}^\top \mathbf{x}^*, \beta^{-1}) \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}_N, \boldsymbol{S}_N^{-1}) d\mathbf{w} \end{aligned} \tag{2.14}$$

The idea here is to average the probability of y^* over all the possible setting of $\mathbf{w}$, weighting the probabilities by how likely each setting of $\mathbf{w}$ is according to its posterior distribution.

2.10 Conclusion

In this chapter, we looked at a specific tool for handling regression problems known as linear regression. We've seen linear regression described in terms of loss functions, probabilistic expressions, and geometric projections, which reflects the deep body of knowledge that we have around this very common technique.

We've also discussed many concepts in this chapter that will prove useful in other areas of machine learning, particularly for other supervised techniques: loss functions, regularization, bias and variance, over and underfitting, posterior distributions, maximum likelihood estimation, and cross-validation among others. Spending time to develop an understanding of these concepts now will pay off going forward.

It may or may not be obvious at this point that we are missing a technique for a very large class of problems: those where the solution is not just a continuous, real number. How do we handle situations where we need to make a choice between different discrete options? This is the question we will turn to in the next chapter.

Chapter 3

Classification

In the last chapter we explored ways of predicting a continuous, real-number target. In this chapter, we're going to think about a different problem- one where our target output is discrete-valued. This type of problem, one where we make a prediction by choosing between finite class options, is known as **classification**.

3.1 Defining the Problem

As we did when studying regression, let's begin by thinking about the type of problems we are trying to solve. Here are a few examples of classification tasks:

1. Predicting whether a given email is spam.

2. Predicting the type of object in an image.

3. Predicting whether a manufactured good is defective.

The point of classification is hopefully clear: we're trying to identify the most appropriate class for an input data point.

Definition 3.1.1 (Classification): A set of problems that seeks to make predictions about unobserved target classes given observed input variables.

3.2 Solution Options

There are several different means by which we can solve classification problems. We're going to discuss three in this chapter: discriminant functions, probabilistic discriminative models (e.g. logistic regression), and probabilistic generative models. Note that these are not the only methods for performing classification tasks, but they are similar enough that it makes sense to present and explore them together. Specifically, these techniques all use some linear combination of input variables to produce a class prediction. For that reason, we will refer to these techniques as **generalized linear models**.

ML Framework Cube: Generalized Linear Models

Since we are using these techniques to perform classification, generalized linear models deal with a **discrete** output domain. Second, as with linear regression, our goal is to make predictions on future data points given an initial set of data to learn from. Thus, generalized linear models are **supervised** techniques. Finally, depending on the type of generalized linear model, they can be either **probabilistic or non-probabilistic**.

Domain	***Training***	***Probabilistic***
Discrete	Supervised	Yes / No

3.3 Discriminant Functions

Generalized linear models for classification come in several different flavors. The most straightforward method carries over very easily from linear regression: **discriminant functions**. As we will see, with discriminant functions we are linearly separating the input space into sections belonging to different target classes. We will explore this method first. One thing to keep in mind is that it's generally easiest to initially learn these techniques in the case where we have only two target classes, but there is typically a generalization that allows us to handle the multi-class case as well.

As with linear regression, discriminant functions $h(\mathbf{x}, \mathbf{w})$ seek to find a weighted combination of our input variables to make a prediction about the target class:

$$h(\mathbf{x}, \mathbf{w}) = w^{(0)}x^{(0)} + w^{(1)}x^{(1)} + ... + w^{(D)}x^{(D)} \tag{3.1}$$

where we are using the bias trick of appending $x^{(0)} = 1$ to all of our data points.

3.3.1 Basic Setup: Binary Linear Classification

The simplest use case for a discriminant function is when we only have two classes that we are trying to decide between. Let's denote these two classes **1** and **-1**. Our discriminant function in Equation 3.1 will then predict class **1** if $h(\mathbf{x}, \mathbf{w}) \geq 0$ and class **-1** if $h(\mathbf{x}, \mathbf{w}) < 0$:

$$\begin{cases} 1 & \text{if } h(\mathbf{x}, \mathbf{w}) \geq 0 \\ -1 & \text{if } h(\mathbf{x}, \mathbf{w}) < 0 \end{cases}$$

Geometrically, the linear separation between these two classes then looks like that of Figure 3.1. Notice the line where our prediction switches from class 1 to class -1. This is precisely where $h(\mathbf{x}, \mathbf{w}) = 0$, and it is known as the **decision boundary**.

Definition 3.3.1 (Decision Boundary): The decision boundary is the line that divides the input space into different target classes. It is learned from an initial data set, and then the target class of new data points can be predicted based on where they fall relative to the decision boundary. At the decision boundary, the discriminant function takes on a value of 0.

⋆ You will sometimes see the term **decision surface** in place of decision boundary, particularly if the input space is larger than two dimensions.

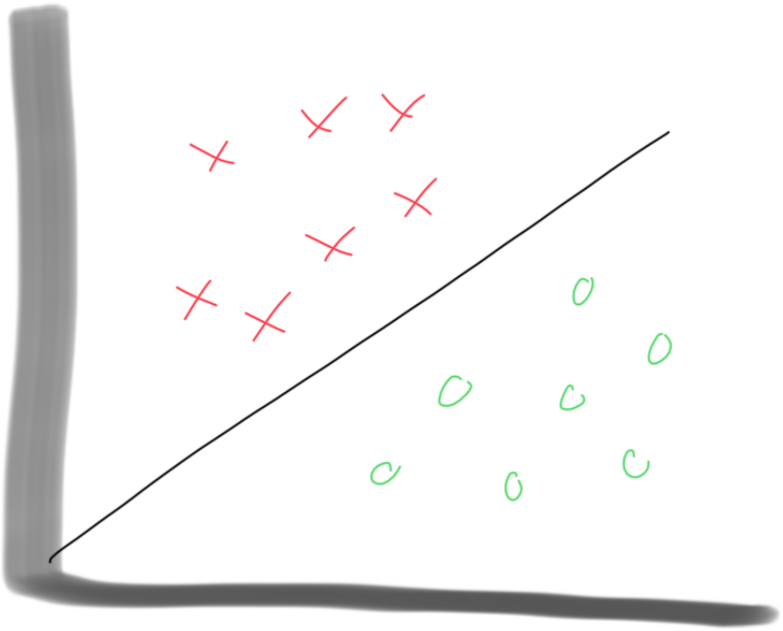


Figure 3.1: Clear separation between classes.

3.3.2 Multiple Classes

Now consider the case that we have $K > 2$ classes $C_1, C_2, ..., C_K$ to choose between. One obvious approach we might try is to use K different discriminant functions that each determine whether or not a given input is in that class C_k. This is known as a *one-versus-all* approach, and it doesn't work properly because we end up with ambiguous regions as demonstrated in Figure 3.2. Intuitively, several of the discriminator functions could claim that a data point is a part of their class, which is an undesirable result.

Another obvious approach we might employ is to use $\binom{K}{2}$ discriminant functions that each determine whether a given point is more likely to be in class C_j or class C_k. This is known as a *one-versus-one* approach, and it also doesn't work because we again end up with ambiguous regions as demonstrated in Figure 3.3.

Instead, we can avoid these ambiguities in the multi-class case by using K different linear classifiers $h_k(\mathbf{x}, \mathbf{w}_k)$, and then assigning new data points to the class C_k for which $h_k(\mathbf{x}, \mathbf{w}_k) > h_j(\mathbf{x}, \mathbf{w}_j)$ for all $j \neq k$. Then, similar to the two-class case, the decision boundaries are described by the surface along which $h_k(\mathbf{x}, \mathbf{w}_k) = h_j(\mathbf{x}, \mathbf{w}_j)$.

Now that we've explored the multi-class generalization, we can consider how to learn the weights $\mathbf{w}$ that define the optimal discriminant functions. However, prior to solving for $\mathbf{w}$, we need to discuss how basis transformations apply to classification problems.

3.3.3 Basis Changes in Classification

We initially discussed basis changes in the context of linear regression, and they are equally important for classification tasks. For example, consider the data set in Figure 3.4.

It's obviously not possible for us to use a linear classifier to separate this data set. However, if we apply a basis change by squaring one of the data points, we instead have Figure 3.5, which is

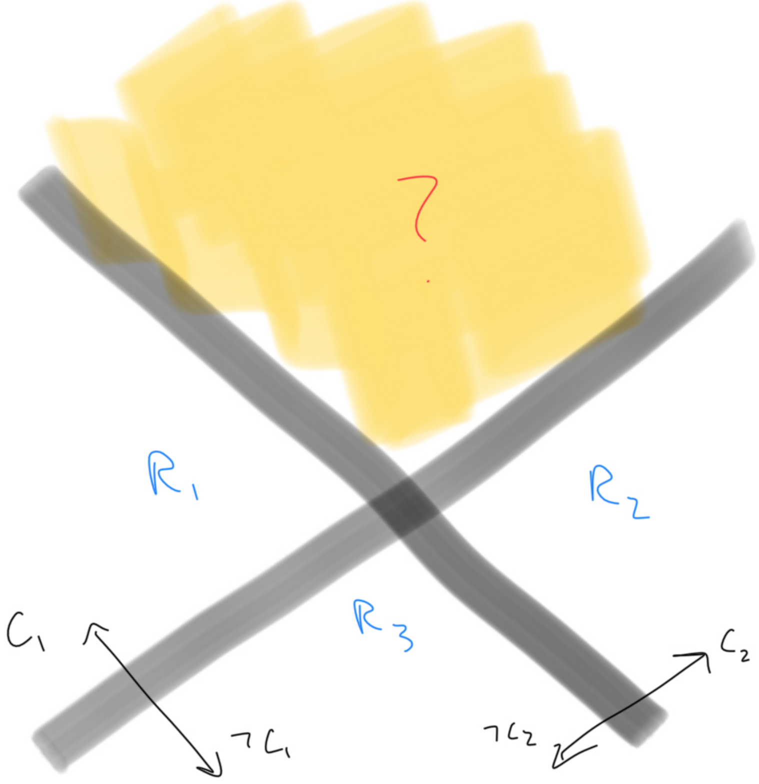


Figure 3.2: Ambiguities arise from one-versus-all method.

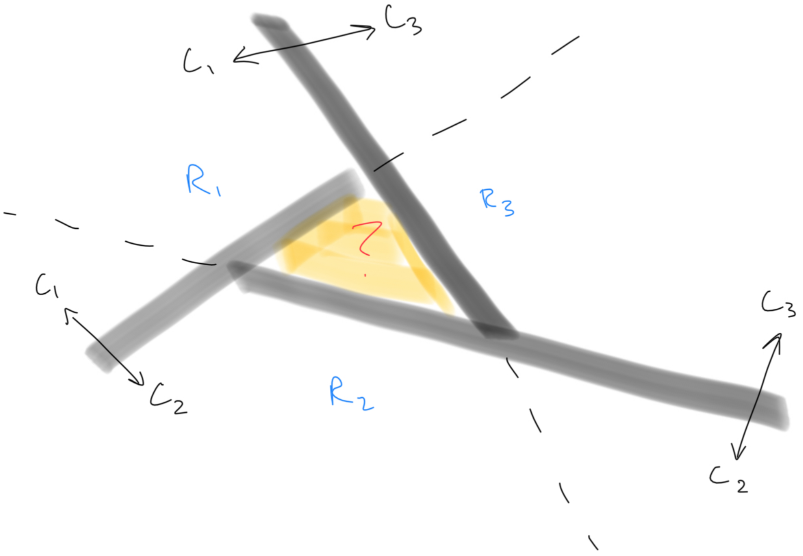


Figure 3.3: Ambiguities arise from one-versus-one method.

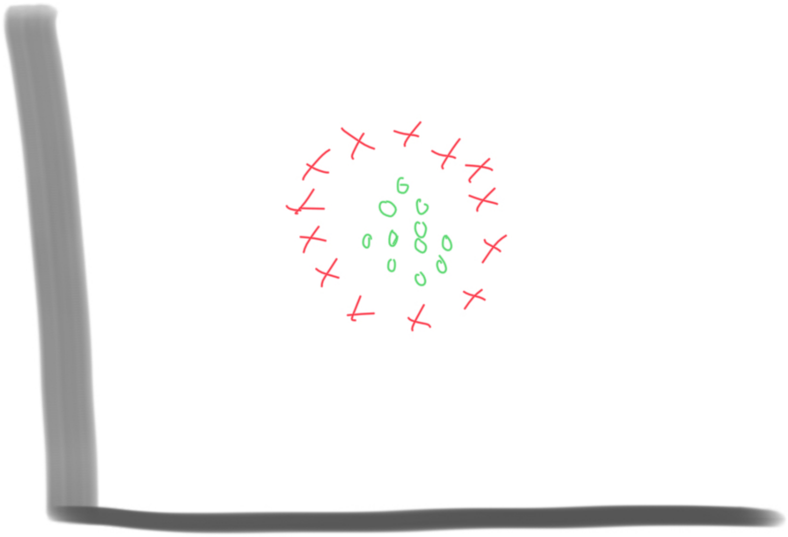


Figure 3.4: Data set without any basis functions applied, not linearly separable.

now linearly separable by a plane between the two classes. Applying a generic basis change $\phi(\cdot)$, we can write our generalized linear model as:

$$h_k(\mathbf{x}, \mathbf{w}_k) = \mathbf{w}_k^\top \phi(\mathbf{x}) = \mathbf{w}_k^\top \boldsymbol{\phi} \tag{3.2}$$

For the sake of simplicity in the rest of this chapter, we will leave out any basis changes in our derivations, but you should recognize that they could be applied to any of our input data to make the problems more tractable.

⋆ For an input matrix $\mathbf{X}$, there is a matrix generalization of our basis transformed inputs: $\mathbf{\Phi} = \phi(\mathbf{X})$, where $\mathbf{\Phi}$ is known as the *design matrix*.

3.4 Numerical Parameter Optimization and Gradient Descent

Recall from the previous chapter on linear regression that when it comes to optimizing our model's weight parameters $\mathbf{w}$, the goal is to minimize our loss function $\mathcal{L}(\mathbf{w})$ (also called the objective function). We did this by taking the derivative of our objective function with respect to $\mathbf{w}$, setting that expression equal to 0, and solving for $\mathbf{w}$. We were previously able to perform that procedure with confidence because the least squares loss function was *convex* with respect to the weight parameters, which meant it had a global solution we could solve for directly. Thus, the point of minimization for the objective function would occur where $\nabla\mathcal{L}(\mathbf{w}) = 0$.

Unfortunately, it's not always the case that our objective function will be convex with respect to our weight parameters. In fact, in the next section, we will consider an objective function that is not convex, and as a result we will need a new way to optimize our parameters. Typically, when facing a non-convex objective function, we will need to resort to a *numerical* procedure.

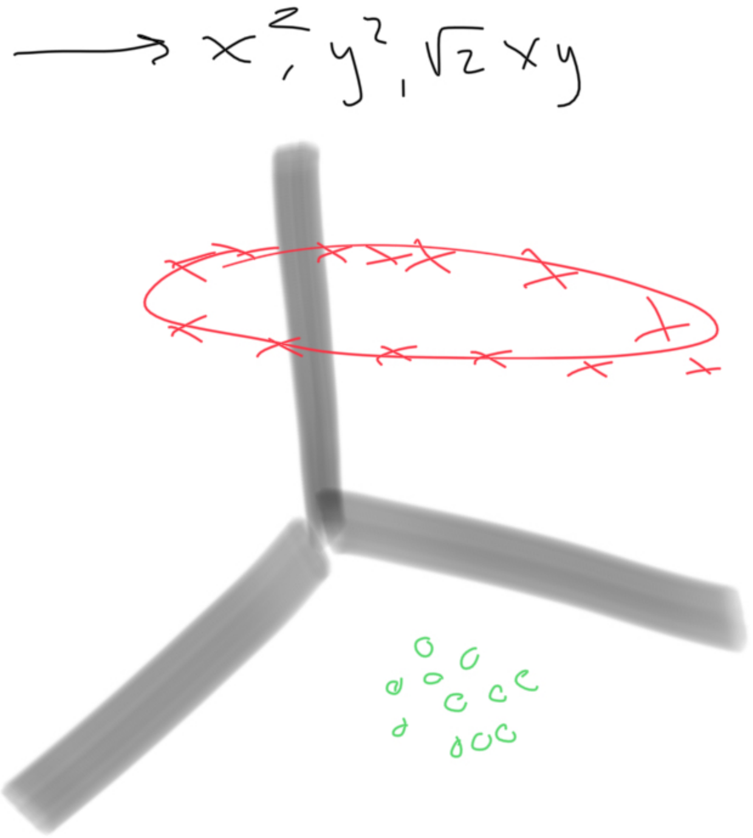


Figure 3.5: Data set with basis functions applied, now linearly separable.

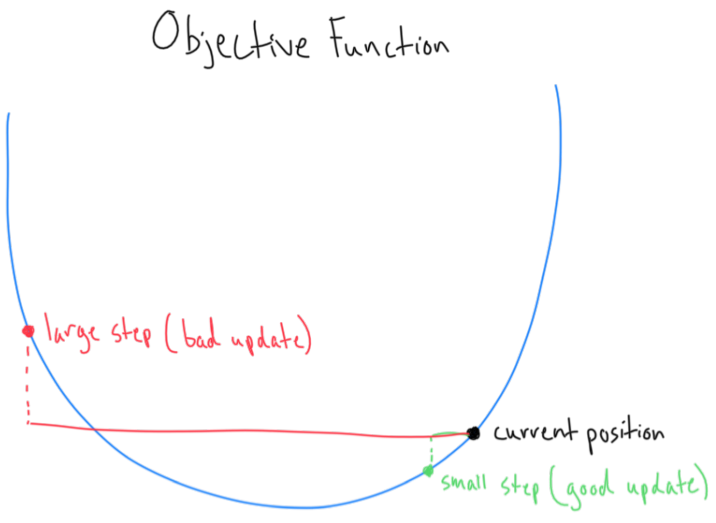


Figure 3.6: Step Size in Gradient Descent.

⋆ The terms *numerical* and *analytical* procedures come up very frequently in machine learning literature. An analytical solution typically utilizes a closed form equation that accepts your model and input data and returns a solution in the form of optimized model parameters. On the other hand, numerical solutions are those that require some sort of iteration to move toward an ever better solution, eventually stopping once the solution is deemed 'good enough'. Analytical solutions are typically more desirable than numerical solutions due to computational efficiency and performance guarantees, but they often are not possible for complex problems due to non-convexity.

Gradient descent is one such numerical optimization technique.

3.4.1 Gradient Descent

Definition 3.4.1 (Gradient Descent): Gradient descent is a numerical, iterative optimization technique for finding the minimum of a function. It is often used to fit complex model parameters.

The high level idea behind gradient descent is as follows: to update our parameters, we take a small step in the opposite direction of the gradient of our objective function with respect to the weight parameters $\mathbf{w}^{(t)}$. Notationally, this looks like the following:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla \mathcal{L}(\mathbf{w}^{(t)}) \tag{3.3}$$

where $\mathbf{w}^{(t)}$ corresponds to the state of the parameters $\mathbf{w}$ at time t, $\mathcal{L}(\mathbf{w}^{(t)})$ is the gradient of our objective function, and $\eta > 0$ is known as the *learning rate*. Note that the parameter values at time $t = 0$ given by $\mathbf{w}^{(0)}$ are often initialized randomly.

⋆ In general, we want a learning rate that is large enough so that we make progress toward reaching a better solution, but not so large that we take a step that puts us in a worse place in the parameter space than we were at the previous step. Notice in Figure 3.6 that an appropriately small step size improves our objective function, while a large step size overshoots the update and leaves us in a worse position.

Why take a step in the opposite direction of the gradient of the objective function? You can think of the objective function as a hill, and the current state of our parameters $\mathbf{w}^{(t)}$ is our position on that hill. The gradient tells us the steepest direction of increase in the objective function (i.e. it specifies the direction that will make our model worse). Since we want to minimize the objective function, we choose to move away from the direction of the gradient, sending our model down the hill towards an area of lower error. We typically cease optimization when our updates become sufficiently small, indicating that we've reached a local minimum. Note that it's a good idea to run gradient descent multiple times to settle on a final value for $\mathbf{w}$, ideally initializing $\mathbf{w}^{(0)}$ to a different starting value each time, because we are optimizing a function with multiple local minima.

3.4.2 Batch Gradient Descent versus Stochastic Gradient Descent

There are different means by which we can compute the gradient of our objective function at each step. The first way, often called *batch gradient descent*, computes the gradient for our objective function at each step using the entire data set. In contrast, the technique known as *stochastic gradient descent* (also known as SGD) utilizes a subset of the data points at each step to compute the gradient, sometimes just a single data point. Stochastic gradient descent is typically a more popular technique for several reasons. First, the computation time is often significantly smaller as you don't need to pass over the entire data set at each iteration. Furthermore, it's less likely that you will get stuck in local minima while running SGD because a point in the parameter space that is a local minima for the entire data set combined is much less likely to be a local minima for each data point individually. Finally, SGD lends itself to being used for training online models (meaning models built on data points that are arriving at regular intervals) as the entirety of the data does not need to be present in order to train.

3.5 Objectives for Decision Boundaries

Now that we have a high-level understanding of what we're trying to accomplish with discriminant functions as well as a grasp on gradient descent, we can consider how to solve for the decision boundaries that will dictate our classification decisions. Similar to linear regression, we first need to establish an objective function to optimize. We begin with a very simple objective function known as **0/1 loss**.

3.5.1 0/1 Loss

Recall that a loss function penalizes mistakes made by our model. The idea behind the **0/1 loss** function is very simple: if our model misclassifies a point, we incur a loss of 1, and if our model classifies it correctly, we incur no loss.

While this is a very intuitive loss function, it does not have a closed form solution like least squares does, and it is non-convex so it is not easily optimized. Intuitively, because we incur a loss of 0 or 1 for every prediction, we have no sense of 'how good' a given prediction was. For example, one prediction could be very close to correct, while another could be way off, but they would both receive an equivalent loss of 1. Formally, because this loss function is not differentiable, we cannot get gradient information with which to optimize our model parameters $\mathbf{w}$. We will find a way around this in a moment when we discuss **hinge loss**, but before we get to that, let's consider using least squares loss as we did for linear regression.

3.5.2 Least Squares Loss

We are already familiar with the least squares loss function from linear regression, and we can apply it again in this context to find the set of weights $\mathbf{w}$ that form the optimal decision boundary between target classes.

We first need to introduce the idea of *one-hot encoding*, which simply means that the class of a given data point is described by a vector with K options that has a 1 in the position that corresponds to class C_k and 0s everywhere else (note that these classes aren't usually 0-indexed). For example, class C_1 of 4 classes would be represented by the vector:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.4}$$

While class C_2 would be represented by the vector:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.5}$$

and so on. Now that we have the idea of one-hot encoding, we can describe our target classes for each data point in terms of a one-hot encoded vector, which can then be used in our training process for least squares.

Each class C_k gets its own linear function with a different set of weights $\mathbf{w}_k$:

$$h_k(\mathbf{x}, \mathbf{w}_k) = \mathbf{w}_k^\top \mathbf{x}$$

We can combine the set of weights for each class into a matrix $\mathbf{W}$, which gives us our linear classifier:

$$h(\mathbf{x}, \mathbf{W}) = \mathbf{W}^\top \mathbf{x} \tag{3.6}$$

where each row in the transposed weight matrix $\mathbf{W}^\top$ corresponds to the linear function of an individual class, and matrix $\mathbf{W}$ is $D \times K$. We can use the results derived in the last chapter to find the solution for $\mathbf{W}$ that minimizes the least squares loss function. Assuming a data set of input data points $\mathbf{X}$ and one-hot encoded target vectors $\mathbf{Y}$ (where every row is a single target vector, so that $\mathbf{Y}$ is $N \times K$), the optimal solution for $\mathbf{W}$ is given by:

$$\mathbf{W}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y},$$

which we can then use in our discriminant function $h(\mathbf{x}, \mathbf{W}^*)$ to make predictions on new data points.

While least squares gives us an analytic solution for our discriminant function, it has significant limitations when used for classification. For one, least squares penalizes data points that are 'too good', meaning they fall too far on the correct side of the decision boundary. Furthermore, it is not robust to outliers, meaning the decision boundary significantly changes with the addition of just a few outlier data points, as seen in Figure 3.7.

We can help remedy the problems with least squares by using an alternative loss function for determining our weight parameters.

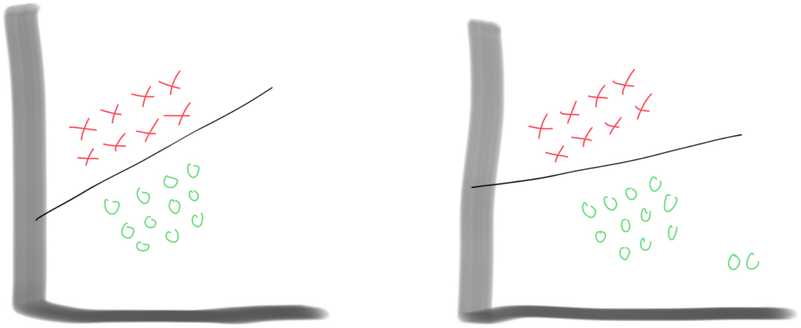


Figure 3.7: Outliers significantly impact our decision boundary.

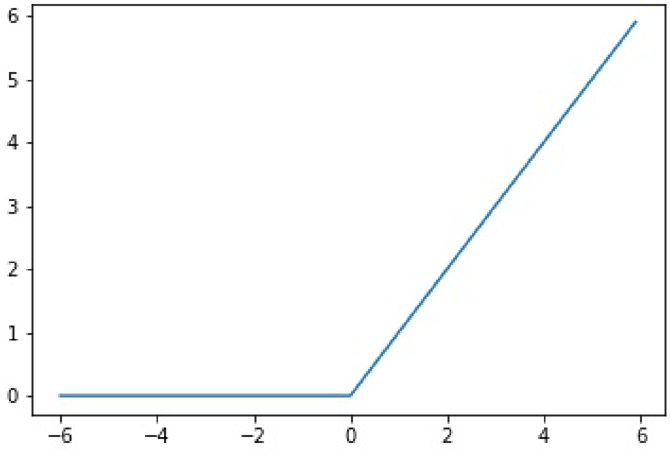


Figure 3.8: Form of the ReLU function.

3.5.3 Hinge Loss

Recall that the problem with 0/1 loss was that we couldn't use it optimize our model parameters **w**. It didn't produce a closed form solution like least squares loss, and it wasn't differentiable so we couldn't take gradients.

The hinge loss function is a modification of the 0/1 loss function that both provides more fine-grained information about the 'goodness' of a prediction and makes the loss function differentiable. To understand the hinge loss function, it's first necessary to introduce the *rectified linear activation unit*, known as ReLU, seen in Figure 3.8.

$$\text{ReLU}(z) = \max\{0, z\} \tag{3.7}$$

We can use the form of this function to our advantage in constructing the hinge loss by recognizing that we wish to incur error when we're wrong (which corresponds to $z > 0$, the right side of the graph that is continuously increasing), and we wish to incur 0 error if we are correct (which corresponds to the left side of the graph where $z < 0$).

Remember from the previous section on least squares that in the two-class case, we classify a data point $\mathbf{x}^*$ as being from class **1** if $h(\mathbf{x}^*, \mathbf{w}) \geq 0$, and class **-1** otherwise. We can combine this logic with ReLU by recognizing that $-h(\mathbf{x}^*, \mathbf{w})y^* \geq 0$ when there is a classification error, where y^* is the true class of data point $\mathbf{x}^*$. This has exactly the properties we described above: we incur error when we misclassify, and otherwise we do not incur error.

We can then write the entirety of the hinge loss function:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \text{ReLU}(-h(\mathbf{x}_i, \mathbf{w})y_i) \tag{3.8}$$

$$= -\sum_{y_i \neq \hat{y}_i}^{N} h(\mathbf{x}_i, \mathbf{w})y_i \tag{3.9}$$

$$= -\sum_{y_i \neq \hat{y}_i}^{N} \mathbf{w}^\top \mathbf{x}_i y_i \tag{3.10}$$

where $\hat{y}_i$ is our class prediction and y_i is the true class value. Notice that misclassified examples contribute positive loss, as desired. We can take the gradient of this loss function, which will allow us to optimize it using stochastic gradient descent. The gradient of the loss with respect to our parameters $\mathbf{w}$ is as follows:

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = -\sum_{y_i \neq \hat{y}_i}^{N} \mathbf{x}_i y_i$$

and then our update equation from time t to time $t+1$ for a single misclassified example and with learning rate η is given by:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \mathbf{w}^{(t)} + \eta \mathbf{x}_i y_i$$

To sum up, the benefits of the hinge loss function are its differentiability (which allows us to optimize our weight parameters), the fact that it doesn't penalize any correctly classified data points (unlike basic linear classification), and that it penalizes more heavily data points that are more poorly misclassified.

Using hinge loss with discriminant functions to solve classification tasks (and applying stochastic gradient descent to optimize the model parameters) is known as the **perceptron algorithm**. The perceptron algorithm guarantees that if there is separability between all of our data points and we run the algorithm for long enough, we will find a setting of parameters that perfectly separates our data set. The proof for this is beyond the scope of this textbook.

3.6 Probabilistic Methods

Unsurprisingly, we can also cast the problem of classification into a probabilistic context, which we now turn our attention to. Within this setting, we have a secondary choice to make between two distinct probabilistic approaches: discriminative or generative. We will explore both of these options.

3.6.1 Probabilistic Discriminative Models

Ultimately, our classification task can be summarized as follows: *given a new data point $\boldsymbol{x}^*$, can we accurately predict the target class y^*?*

Given this problem statement, it makes sense that we might try to model $p(y^*|\mathbf{x}^*)$. In fact, modeling this conditional distribution directly is what's known as **probabilistic discriminative modeling**.

Definition 3.6.1 (Probabilistic Discriminative Modeling): Probabilistic modeling is a classification technique whereby we choose to directly model the conditional class distribution in order to make classification predictions.

This means that we will start with the functional form of the generalized linear model described by Equation 3.2, convert this to a conditional distribution, and then optimize the parameters of the conditional distribution directly using a maximum likelihood procedure. From here, we will be able to make predictions on new data points $\mathbf{x}^*$. The key feature of this procedure, which is known as *discriminative training*, is that it optimizes the parameters of a conditional distribution directly. We describe a specific, common example of this type of procedure called **logistic regression** in the next section.

Logistic Regression

One problem we need to face in our discriminative modeling paradigm is that the results of our generalized linear model are not probabilities; they are simply real numbers. This is why in the previous paragraph we mentioned needing to convert our generalized linear model to a conditional distribution. That step boils down to somehow squashing the outputs of our generalized linear model onto the real numbers between 0 and 1, which will then correspond to probabilities. To do this, we will apply what is known as the **logistic sigmoid function**, $\sigma(\cdot)$.

Definition 3.6.2 (Logistic Sigmoid Function, $\sigma(\cdot)$): The logistic sigmoid function is commonly used to compress the real number line down to values between 0 and 1. It is defined functionally as:

$$\sigma(z) = \frac{1}{1 + \exp{(-z)}}$$

As you can see in Figure 3.9 where the logistic sigmoid function is graphed, it squashes our output domain between 0 and 1 as desired for a probability.

⋆ There is a more satisfying derivation for our use of the logistic sigmoid function in logistic regression, but understanding its squashing properties as motivation is sufficient for the purposes of this book.

Using the logistic sigmoid function, we now have a means of generating a probability that a new data point $\mathbf{x}^*$ is part of class y^*. Because we are currently operating in the two-class case, which in this context will be denoted C_1 and C_2, we'll write the probability for each of these classes as:

$$p(y^* = C_1|\mathbf{x}^*) = \sigma(\mathbf{w}^\top \mathbf{x}^*)$$
$$p(y^* = C_2|\mathbf{x}^*) = 1 - p(y^* = C_1|\mathbf{x}^*)$$

Now that we have such functions, we can apply the maximum likelihood procedure to determine the optimal parameters for our logistic regression model.

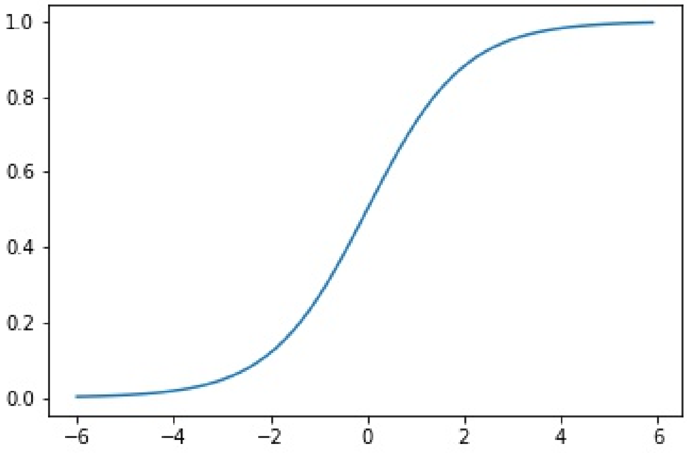


Figure 3.9: Logistic Sigmoid Function.

For a data set $\{\mathbf{x}_i, y_i\}$ where $i = 1..N$ and $y_i \in \{0, 1\}$, the likelihood for our setting of parameters $\mathbf{w}$ can be written as:

$$p(\{y_i\}_{i=1}^N|\mathbf{w}) = \prod_{i=1}^N \hat{y}_i^{y_i}\{1 - \hat{y}_i\}^{1-y_i} \tag{3.11}$$

where $\hat{y}_i = p(y_i = C_1|\mathbf{x}_i) = \sigma(\mathbf{w}^\top\mathbf{x}_i)$.

In general, we would like to maximize this probability to find the optimal setting of our parameters. This is exactly what we intend to do, but with two further simplifications. First, we're going to maximize the probability of the *logarithm* of the likelihood, as in Equation 3.12.

$$\ln(p(\{y_i\}_{i=1}^N|\mathbf{w})) = \sum_{i=1}^N \{y_i \ln \hat{y}_i + (1 - y_i) \ln (1 - \hat{y}_i)\} \tag{3.12}$$

As a monotonically increasing function, maximizing the logarithm of the likelihood (called the *log likelihood*) will result in the same optimal setting of parameters as if we had just optimized the likelihood directly. Furthermore, using the log likelihood has the nice effect of turning what is currently a product of terms from $1..N$ to a sum of terms from $1..N$, which will make our calculations nicer.

Second, we will turn our log likelihood into an *error function* by taking the negative of our log likelihood expression. Now, instead of maximizing the log likelihood, we will be minimizing the error function, which will again find us the same setting of parameters.

⋆ It's worth rereading the above paragraph again to understand the pattern presented there, which we will see several times throughout this book. Instead of maximizing a likelihood function directly, it is often easier to define an error function using the negative log likelihood, which we can then minimize to find the optimal setting of parameters for our model.

After taking the negative logarithm of the likelihood function defined by Equation 3.11, we are left with the following term, known as the *cross-entropy error function*, which we will seek to minimize:

$$\mathrm{E}(\mathbf{w}) = -\ln p(\{y_i\}|\mathbf{w}) = -\sum_{i=1}^{N}\{y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)\} \tag{3.13}$$

where as before $\hat{y}_i = p(y_i = C_1|\mathbf{x}_i) = \sigma(\mathbf{w}^\top \mathbf{x}_i)$. The cross-entropy error refers to the log likelhood of the labels conditioned on the examples. When used with the specific form of the logistic regression, this is also the *logistic loss*. Now, to solve for the optimal setting of parameters using a maximum likelihood approach as we've done previously, we start by taking the gradient of the cross-entropy error function with respect to $\mathbf{w}$:

$$\nabla\mathrm{E}(\mathbf{w}) = \sum_{i=1}^{N}(\hat{y}_i - y_i)\mathbf{x}_i \tag{3.14}$$

which we arrive at by recognizing that the derivative of the logistic sigmoid function can be written in terms of itself as:

$$\frac{\partial\sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$$

Let's inspect the form of Equation 3.14 for a moment to understand its implications. First, it's a summation over all of our data points, as we would expect. Then, for each data point, we are taking the difference between our predicted value $\hat{y}_i$ and the actual value y_i, and multiplying that difference by the input vector $\mathbf{x}_i$.

While a closed form solution does not present itself here as it did in the case of linear regression due to the nonlinearity of the logistic sigmoid function, we can still optimize the parameters $\mathbf{w}$ of our model using an iterative procedure like gradient descent, where the objective function is defined by Equation 3.13.

Multi-Class Logistic Regression and Softmax

As we saw when working with discriminant functions, we also need to account for multi-class problems, which are practically speaking more common than the simple two-class scenario.

In the logistic regression setting (which is a form of *discriminative modeling*, not to be confused with *discriminant functions*), we are now working with probabilities, which is why we introduced the 'probability squashing' sigmoidal function $\sigma(\cdot)$. Note that the sigmoidal function is also sometimes known as the sigmoidal activation function.

Similarly, in the multi-class logistic regression setting, we would like to also have a probability squashing function that generalizes beyond two classes. This generalization of the sigmoidal function is known as **softmax**.

Definition 3.6.3 (Softmax): Softmax is the multi-class generalization of the sigmoidal activation function. It accepts a vector of activations (inputs) and returns a vector of probabilities corresponding to those activations. It is defined as follows:

$$\mathrm{softmax}_k(\mathbf{z}) = \frac{\exp(z_k)}{\sum_{i=1}^{K}\exp(z_i)}, \text{ for all } k$$

Multi-class logistic regression uses softmax over a vector of activations to select the most likely target class for a new data point. It does this by applying softmax and then assigning the new data point to the class with the highest probability.

Example 3.1 (Softmax Example): Consider an example that has three classes: C_1, C_2, C_3. Let's say we have an activation vector $\mathbf{z}$ for our new data point $\mathbf{x}$ that we wish to classify, given by:

$$\mathbf{z} = \mathbf{W}^\top \mathbf{x} = \begin{bmatrix} 4 \\ 1 \\ 7 \end{bmatrix}$$

where

$$\mathbf{z}_j = \mathbf{w}_j^\top \mathbf{x}$$

Then, using our definition of softmax, we have:

$$\text{softmax}(\mathbf{z}) = \begin{bmatrix} 0.047 \\ 0.002 \\ 0.950 \end{bmatrix}$$

And therefore, we would assign our new data point $\mathbf{x}$ to class C_3, which has the largest activation.

As in the two-class logistic regression case, we now need to solve for the parameters $\mathbf{W}$ of our model, also written as $\{\mathbf{w}_j\}$. Assume we have an observed data set $\{\mathbf{x}_i, \mathbf{y}_i\}$ for $i = 1..N$ where $\mathbf{y}_i$ are one-hot encoded target vectors. We begin this process by writing the likelihood for our data, which is only slightly modified here to account for multiple classes:

$$p(\{\mathbf{y}_i\}_{i=1}^N | \mathbf{W}) = \prod_{i=1}^N \prod_{j=1}^K p(\mathbf{y}_i = C_j | \mathbf{x}_i)^{y_{ij}} = \prod_{i=1}^N \prod_{j=1}^K \hat{y}_{ij}^{y_{ij}} \tag{3.15}$$

where $\hat{y}_{ij} = \text{softmax}_j(\mathbf{W}\mathbf{x}_i)$

We can now take the negative logarithm to get the cross-entropy error function for the multi-class classification problem:

$$\text{E}(\mathbf{W}) = -\ln p(\{\mathbf{y}_i\}_{i=1}^N | \mathbf{W}) = -\sum_{i=1}^N \sum_{j=1}^K y_{ij} \ln \hat{y}_{ij} \tag{3.16}$$

As in the two-class case, we now take the gradient with respect to one of our weight parameter vectors $\mathbf{w}_j$:

$$\nabla_{\mathbf{w}_j} \text{E}(\mathbf{W}) = \sum_{i=1}^N (\hat{y}_{ij} - y_{ij}) \mathbf{x}_i \tag{3.17}$$

which we arrived at by recognizing that the derivative of the softmax function with respect to the input activations z_j can be written in terms of itself:

$$\frac{\partial \text{softmax}_k(z)}{\partial z_j} = \text{softmax}_k(z)(\text{I}_{kj} - \text{softmax}_j(z))$$

where I is the identity matrix.

As in the two-class case, now that we have this gradient expression, we can use an iterative procedure like gradient descent to optimize our parameters $\mathbf{W}$.

3.6.2 Probabilistic Generative Models

With the probabilistic discriminative modeling approach, we elected to directly model $p(y^*|\mathbf{x}^*)$. However, there was an alternative option: we could have instead modeled the joint distribution of the class y^* and the input data point $\mathbf{x}^*$ together as $p(y^*, \mathbf{x}^*)$. This approach is what's known as **probabilistic generative modeling** because we actually model the process by which the data was generated.

To model the data generating process in classification tasks generally acknowledges that a data point is produced by first selecting a class y^* from a categorical class prior $p(y^*)$ and then generating the data point $\mathbf{x}^*$ itself from the class-conditional distribution $p(\mathbf{x}^*|y^*)$, the form of which is problem specific. This generative approach is a particularly good idea if we want to create more data (by sampling from the joint distribution) or if we have some sort of expert knowledge about how the data was generated, which can make our model more powerful than the discriminative approach.

$\star$ Notice that with probabilistic generative modeling, we choose a specific distribution for our class-conditional densities instead of simply using a generalized linear model combined with a sigmoid/softmax function as we did in the logistic regression setting. This highlights the difference between discriminative and generative modeling: in the generative setting, we are modeling the production of the data itself instead of simply optimizing the parameters of a more general model that predicts class membership directly.

Classification in the Generative Setting

Now that we're situated in the generative setting, we turn our attention to the actual problem of using our model to predict class membership of new data points $\mathbf{x}^*$.

To perform classification, we will pick the class C_k that maximizes the probability of $\mathbf{x}^*$ being from that class as defined by $p(y^* = C_k|\mathbf{x}^*)$. We can relate this conditional density to the joint density $p(y^*, \mathbf{x}^*)$ through Bayes' Rule:

$$p(y^* = C_k|\mathbf{x}^*) = \frac{p(y^*, \mathbf{x}^*)}{p(\mathbf{x}^*)} = \frac{p(\mathbf{x}^*|y^* = C_k)p(y^* = C_k)}{p(\mathbf{x}^*)} \propto p(\mathbf{x}^*|y^* = C_k)p(y^* = C_k)$$

where $p(\mathbf{x}^*)$ is a constant that can be ignored as it will be the same for every conditional probability $p(y^* = C_k|\mathbf{x}^*)$.

Recall that the class prior $p(y)$ will always be a categorical distribution (the multi-class generalization of the Bernoulli distribution), while the class-conditional distribution can be specified using prior knowledge of the problem domain. Once we have specified this class conditional distribution, we can solve for the parameters of both that model and the categorical distribution by optimizing the likelihood function. Let's now derive that likelihood function.

Derivation 3.6.1 (Probabilistic Generative Model Likelihood Function):
We're going to derive the likelihood function for the parameters of our probabilistic generative model in the two-class setting, allowing that the multi-class generalization will be a straightforward exercise.
Let's start by assuming a Gaussian conditional distribution for our data $p(\mathbf{x}|y = C_k)$. Given a data set $\{\mathbf{x}_i, y_i\}$ for $i = 1..N$, where $y_i = 1$ corresponds to class C_1 and $y_i = 0$ corresponds to class C_2, we can construct our maximum likelihood solution. Let's first specify our class priors:

$$p(C_1) = \pi$$
$$p(C_2) = 1 - \pi$$

For simplicity, we'll assume a shared covariance matrix $\mathbf{\Sigma}$ between our two classes. Then, for data points $\mathbf{x}_i$ from class C_1, we have:

$$p(\mathbf{x}_i, C_1) = p(C_1)p(\mathbf{x}_i|C_1) = \pi\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma})$$

And for data points $\mathbf{x}_i$ from class C_2, we have:

$$p(\mathbf{x}_i, C_2) = p(C_2)p(\mathbf{x}_i|C_2) = (1-\pi)\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_2, \mathbf{\Sigma})$$

Using these two densities, we can construct our likelihood function:

$$p(\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \mathbf{\Sigma}) = \prod_{i=1}^{N} \Big(\pi\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma})\Big)^{y_i} \Big((1-\pi)\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_2, \mathbf{\Sigma})\Big)^{1-y_i}$$

As usual, we will take the logarithm which is easier to work with:

$$\ln p(\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \mathbf{\Sigma}) = \sum_{i=1}^{N} y_i \ln \Big(\pi\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma})\Big) + (1-y_i)\ln\Big((1-\pi)\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_2, \mathbf{\Sigma})\Big)$$

Now that we have specified the log-likelihood function for our model, we can go about optimizing our model by maximizing this likelihood. One way to do this is with a straightforward maximum likelihood estimation approach. We will optimize our parameters $\pi, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \text{and}, \mathbf{\Sigma}$ separately, using the usual procedure of taking the derivative, setting equal to 0, and then solving for the parameter of interest. We write down this MLE solution in the following section.

MLE Solution

Solving for π

Beginning with π, we'll concern ourselves only with the terms that depend on π which are:

$$\sum_{i=1}^{N} y_i \ln \pi + (1-y_i)\ln(1-\pi)$$

Taking the derivative with respect to π, setting equal to 0, rearranging, we get:

$$\pi = \frac{1}{N}\sum_{i=1}^{N} y_i = \frac{N_1}{N} = \frac{N_1}{N_1 + N_2}$$

where N_1 is the number of data points in our data set from class C_1, N_2 is the number of data points from class C_2, and N is just the total number of data points. This means that the maximum likelihood solution for π is the fraction of points that are assigned to class C_1, a fairly intuitive solution and one that will be commonly seen when working with maximum likelihood calculations.

Solving for $\boldsymbol{\mu}$

Let's now perform the maximization for $\boldsymbol{\mu}_1$. Start by considering the terms from our log likelihood involving $\boldsymbol{\mu}_1$:

$$\sum_{i=1}^{N} y_i \ln \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_1, \mathbf{\Sigma}) = -\frac{1}{2}\sum_{i=1}^{N} y_i(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top \mathbf{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_1) + c$$

where c are constants not involving the $\boldsymbol{\mu}_1$ term. Taking the derivative with respect to $\boldsymbol{\mu}_1$, setting equal to 0, and rearranging:

$$\boldsymbol{\mu}_1 = \frac{1}{N_1}\sum_{i=1}^{N} y_i \mathbf{x}_i$$

which is simply the average of all the data points $\mathbf{x}_i$ assigned to class C_1, a very intuitive result. By the same derivation, the maximum likelihood solution for $\boldsymbol{\mu}_2$ is:

$$\boldsymbol{\mu}_2 = \frac{1}{N_2}\sum_{i=1}^{N}(1-y_i)\mathbf{x}_i$$

Solving for $\boldsymbol{\Sigma}$

We can also the maximum likelihood solution for the shared covariance matrix $\boldsymbol{\Sigma}$. Start by considering the terms in our log likelihood expression involving $\boldsymbol{\Sigma}$:

$$-\frac{1}{2}\sum_{i=1}^{N} y_i \ln|\boldsymbol{\Sigma}| - \frac{1}{2}\sum_{i=1}^{N} y_i(\mathbf{x}_i-\boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_1) - \frac{1}{2}\sum_{i=1}^{N}(1-y_i)\ln|\boldsymbol{\Sigma}| \\ -\frac{1}{2}\sum_{i=1}^{N}(1-y_i)(\mathbf{x}_i-\boldsymbol{\mu}_2)^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_2)$$

We can use the following "matrix cookbook formulas" to help with taking the derivative with respect to $\boldsymbol{\Sigma}$. Also, we adopt convention $\mathbf{Z}^{-\top} := (\mathbf{Z}^\top)^{-1}$. The two helpful formulas are:

$$\frac{\partial \mathbf{a}^\top\mathbf{Z}^{-1}\mathbf{b}}{\partial \mathbf{Z}} = -\mathbf{Z}^{-\top}\mathbf{a}\mathbf{b}^\top\mathbf{Z}^{-\top} \\ \frac{\partial \ln|\det(\mathbf{Z})|}{\partial \mathbf{Z}} = \mathbf{Z}^{-\top}.$$

Now, taking the derivative with respect to $\boldsymbol{\Sigma}$ and collecting terms, we have:

$$-\frac{1}{2}N\boldsymbol{\Sigma}^{-\top} + \frac{1}{2}\sum_{i=1}^{N} y_i\boldsymbol{\Sigma}^{-\top}(\mathbf{x}_i-\boldsymbol{\mu}_1)(\mathbf{x}_i-\boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}^{-\top} + \frac{1}{2}\sum_{i=1}^{N}(1-y_i)\boldsymbol{\Sigma}^{-\top}(\mathbf{x}_i-\boldsymbol{\mu}_2)(\mathbf{x}_i-\boldsymbol{\mu}_2)^\top\boldsymbol{\Sigma}^{-\top}$$

Setting this to zero and multiplying both sides by $\boldsymbol{\Sigma}^\top$ from the left and right (with the effect of retaining $\boldsymbol{\Sigma}^\top$ in only the first term, since $\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma}^{-\top} = \boldsymbol{\Sigma}^\top\left(\boldsymbol{\Sigma}^\top\right)^{-1} = \mathbf{I}$ and $\boldsymbol{\Sigma}^{-\top}\boldsymbol{\Sigma}^\top = \left(\boldsymbol{\Sigma}^\top\right)^{-1}\boldsymbol{\Sigma}^\top = \mathbf{I}$), and multiplying by 2, we have

$$N\boldsymbol{\Sigma}^\top - \sum_{i=1}^{N}\left(y_i(\mathbf{x}_i-\boldsymbol{\mu}_1)(\mathbf{x}_i-\boldsymbol{\mu}_1)^\top + (1-y_i)(\mathbf{x}_i-\boldsymbol{\mu}_2)(\mathbf{x}_i-\boldsymbol{\mu}_2)^\top\right) = 0.$$

Rearranging to solve for $\boldsymbol{\Sigma}$, and recognizing that covariance matrices are symmetric, and so $\boldsymbol{\Sigma}^\top = \boldsymbol{\Sigma}$, we have:

$$\boldsymbol{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i(\mathbf{x}_i-\boldsymbol{\mu}_1)(\mathbf{x}_i-\boldsymbol{\mu}_1)^\top + (1-y_i)(\mathbf{x}_i-\boldsymbol{\mu}_2)(\mathbf{x}_i-\boldsymbol{\mu}_2)^\top\right).$$

This has the intuitive interpretation that the maximum likelihood solution for the shared covariance matrix is the weighted average of the two individual covariance matrices. Note that $(\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top$ is a matrix (the outer product of the two vectors). Also, y_i is a scalar, which means that each term is a sum of matrices. For any point i, only one of two matrices inside will contribute due to the use of y_i and $(1 - y_i)$.

It is relatively straightforward to extend these maximum likelihood derivations from their two-class form to their more general, multi-class form.

Naive Bayes

There exists a further simplification to probabilistic generative modeling in the context of classification known as **Naive Bayes**.

Definition 3.6.4 (Naive Bayes): Naive Bayes is a type of generative model for classification tasks. It imposes the simplifying rule that for a given class C_k, we assume that each feature of the data points $\mathbf{x}$ generated within that class are independent (hence the descriptor 'naive'). This means that the conditional distribution $p(\mathbf{x}|y = C_k)$ can be written as:

$$p(\mathbf{x}|y = C_k) = \prod_{i=1}^{D} p(x_i|y = C_k)$$

where D is the number of features in our data point $\mathbf{x}$ and C_k is the class. Note that Naive Bayes does not specify the form of the model $p(x_i|y = C_k)$, this decision is left up to us.

This is obviously not a realistic simplification for all scenarios, but it can make our calculations easier and may actually hold true in certain cases. We can build more intuition for how Naive Bayes works through an example.

Example 3.2 (Naive Bayes Example): Suppose you are given a biased two-sided coin and two biased dice. The coin has probabilities as follows:

Heads : 30%
Tails : 70%

The dice have the numbers 1 through 6 on them, but they are biased differently. Die 1 has probabilities as follows:

1 : 40%
2 : 20%
3 : 10%
4 : 10%
5 : 10%
6 : 10%

Die 2 has probabilities as follows:

1 : 20%
2 : 20%
3 : 10%
4 : 30%
5 : 10%
6 : 10%

Your friend is tasked with doing the following. First, they flip the coin. If it lands Heads, they select Die 1, otherwise they select Die 2. Then, they roll that die 10 times in a row, recording the results of the die rolls. After they have completed this, you get to observe the aggregated results from the die rolls. Using this information (and assuming you know the biases associated with the coin and dice), you must then classify which die the rolls came from. Assume your friend went through this procedure and produced the following counts:

1 : 3
2 : 1
3 : 2
4 : 2
5 : 1
6 : 1

Determine which die this roll count most likely came from.

Solution:
This problem is situated in the Naive Bayes framework: for a given class (dictated by the coin flip), the outcomes within that class (each die roll) are independent. Making a classification in this situation is as simple as computing the probability that the selected die produced the given roll counts. Let's start by computing the probability for Die 1:

$$\begin{aligned} p(\text{Die 1}) &= p(\text{Coin Flip} = \text{Heads}) * p(\text{Roll Count} = [3, 1, 2, 2, 1, 1]) \\ &\propto 0.3 * (0.4)^3 * (0.2)^1 * (0.1)^2 * (0.1)^2 * (0.1)^1 * (0.1)^1 \\ &\propto 3.84 * 10^{-9} \end{aligned}$$

Notice that we don't concern ourselves with the normalization constant for the probability of the roll count - this will not differ between the choice of dice and we can thus ignore it for simplicity. Now the probability for Die 2:

$$\begin{aligned} p(\text{Die 2}) &= p(\text{Coin Flip} = \text{Tails}) * p(\text{Roll Count} = [3, 1, 2, 2, 1, 1]) \\ &\propto 0.7 * (0.2)^3 * (0.2)^1 * (0.1)^2 * (0.3)^2 * (0.1)^1 * (0.1)^1 \\ &\propto 1.008 * 10^{-8} \end{aligned}$$

Therefore, we would classify this roll count as having come from Die 2.

Note that this problem asked us only to make a classification prediction after we already knew the parameters governing the coin flip and dice rolls. However, given a data set, we could have also used a maximum likelihood procedure under the Naive Bayes assumption to estimate the values of the parameters governing the probability of the coin flip and die rolls.

3.7 Conclusion

In this chapter, we looked at different objectives and techniques for solving classification problems, including discriminant functions, probabilistic discriminative models, and probabilistic generative models. In particular, we emphasized the distinction between two-class and multi-class problems as well as the philosophical differences between generative and discriminative modeling.

We also covered several topics that we will make use of in subsequent chapters, including sigmoid functions and softmax, maximum likelihood solutions, and further use of basis changes.

By now, you have a sound understanding of generative modeling and how it can be applied to classification tasks. In the next chapter, we will explore how generative modeling is applied to a still broader class of problems.

Chapter 4

Neural Networks

Despite how seemingly popular neural networks have become recently, they aren't actually a novel technique. The first neural networks were described in the early 1940s, and the only reason they weren't put into practice shortly thereafter was the fact that we didn't yet have access to the large amounts of storage and compute that complex neural network require. Over the last two decades, and particularly with the advent of cloud computing, we now have more and more access to the cheap processing power and memory required to make neural networks a viable option for model building.

As we will come to see in this chapter, neural networks are an extraordinarily flexible class of models used to solve a variety of different problem types. In fact, this flexibility is both what makes them so widely applicable and yet so difficult to use properly. We will explore the applications, underlying theory, and training schemes behind neural networks.

4.1 Motivation

For problems that fall into the category of regression or classification, we've already discussed the utility of basis functions. Sometimes, a problem that is intractable with our raw input data will be readily solvable with basis-transformed data. We often select these basis changes using expert knowledge. For example, if we were working with a data set that related to chemical information, and there were certain equations that a chemist told us to be important for the particular problem we were trying to solve, we might include a variety of the transformations that are present in those equations.

However, imagine now that we have a data set with no accompanying expert information. More often than not, complex problem domains don't come with a useful set of suggested transformations. How do we find useful basis functions in these situations? This is exactly the strength of neural networks - they identify the best basis for a data set!

Neural networks simultaneously solve for our model parameters and the best basis transformations. This makes them exceedingly flexible. Unfortunately, this flexibility is also the weakness of neural nets: while it enables us to solve difficult problems, it also creates a host of other complications. Chief among these complications is the fact that neural networks require a lot of computation to train. This is a result of the effective model space being so large - to explore it all takes time and resources. Furthermore, this flexibilty can cause rather severe overfitting if we are not careful.

In summary, neural networks identify good basis transformations for our data, and the strengths and weaknesses of neural networks stem from the same root cause: model flexibility. It will be our goal then to appropriately harness these properties to create useful models.

4.1.1 Comparison to Other Methods

In the previous two chapters, we explored two broad problem types: classification and regression, and it's natural to wonder where neural networks fit in. The answer is that they are applicable to both. The flexibility of neural networks even extends to the types of problems they can be made to handle. Thus, the tasks that we've explored over the last two chapters, such as predicting heights in the regression case or object category in the classification case, can be performed by neural networks.

Given that neural networks are flexible enough to be used as models for either regression or classification tasks, this means that every time you're faced with a problem that falls into one of these categories, you have a choice to make between the methods we've already covered or using a neural network. Before we've explored the specifics of neural networks, how can we discern at a high level when they will be a good choice for a specific problem?

One simple way to think about this is that if we never *needed* to use neural networks, we probably wouldn't. In other words, if a problem can be solved effectively by one of the techniques we've already described for regression or classification (such as linear regression, discriminant functions, etc.), we would prefer to use those. The reason is that neural networks are often more memory and processor intensive than these other techniques, and they are much more complex to train and debug.

The flip side of this is that hard problems are often too complex or too hard to engineer features for to use a simple regression or classification technique. Indeed, even if you eventually think you will need to use a neural network to solve a given problem, it makes sense to try a simple technique first both to get a baseline of performance and because it may just happen to be good enough.

What is so special about neural networks that they can solve problems that the other techniques we've explored may not be able to? And why are they so expensive to train? These questions will be explored over the course of the chapter, and a good place to start is with the status of neural networks as universal function approximators.

4.1.2 Universal Function Approximation

The flexibility of neural networks is a well-established phenomenon. In fact, neural networks are what are known as *universal function approximators*. This means that with a large enough network, it is possible to approximate any function. The proof of this is beyond the scope of this textbook, but it provides some context for why flexibility is one of the key attributes of neural networks.

ML Framework Cube: Neural Networks

As universal function approximators, neural networks can operate over discrete or continuous outputs. We primarily use neural networks to solve regression or classification problems, which involve training on data sets with example inputs and outputs, making this a **supervised** technique. Finally, while there exist probabilistic extensions for neural networks, they primarily operate in the **non-probabilistic** setting.

Domain	***Training***	***Probabilistic***
Continuous/Discrete	Supervised	No

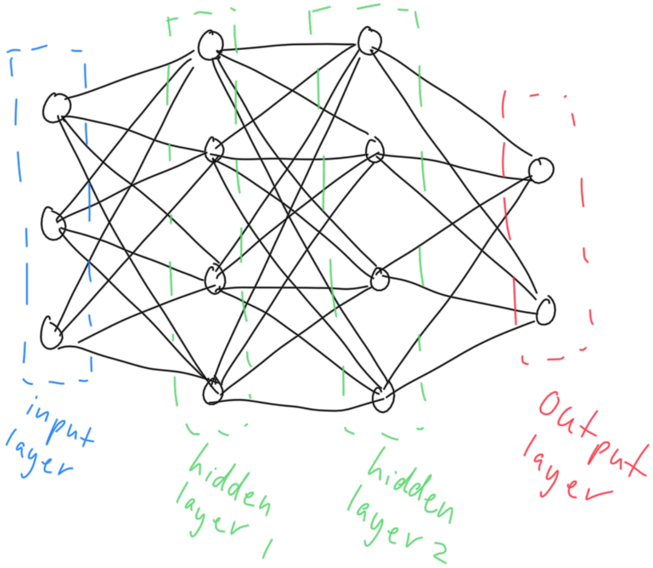


Figure 4.1: Simple Neural Network.

4.2 Feed-Forward Networks

The feed-forward neural network is the most basic setup for a neural network. Most of the logic behind neural networks can be explained using a feed-forward network, with additional bells and whistles typically added to form more complex networks. We will explore this basic neural network structure first.

4.3 Neural Network Basics and Terminology

Looking at Figure 4.1, we see that a feed-forward neural network is a series of connected **layers** that transform an input data point $\mathbf{x}$ into an output data point $\mathbf{y}$. Each layer is composed of **nodes**, the small black circles. Each node in the input layer corresponds to one dimension of a single data point $\mathbf{x}$ (meaning the first node is x_1, the second node x_2, etc.). The same is true of the nodes in the output layer, which represent each dimension of $\mathbf{y}$. For a binary classification problem there is a single output node, representing the predicted probability of the positive class (with K outputs for multi-classification). For a regression problem there may one or more nodes, depending on the dimensions of the output. The nodes in the **hidden** layers correspond to **activation functions**.

Let's zoom in on the first node in hidden layer 1, shown in Figure 4.2, to describe what happens at each node as we transform an input. For now we adopt simple notation and don't worry about indexing by later. Looking at Figure 4.2, notice that every node in the input layer is connected to this first node in hidden layer 1. Each of the lines is a **connection**, that has a weight w_d associated with it. We multiply all the nodes in the input layer by their corresponding weight, and then add them all together to produce the *activation* a, i..e, the input at this first node (we've included the bias term in the input vector):

$$a = x_1 w_1 + x_2 w_2 + x_3 w_3 \tag{4.1}$$

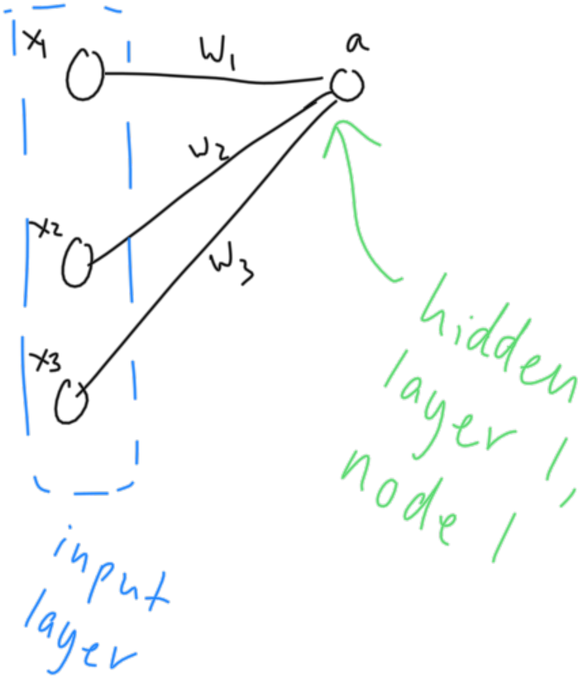


Figure 4.2: Zooming in on the inputs and the first node of the first layer.

⋆ Every node in every layer has distinct weights associated with it.

This gives us the activation for the first node in the first hidden layer. Once we've done this for every node in the first hidden layer, we make a non-linear transform of these activations, and then move on to computing the activations for the second hidden layer (which require the outputs from the first hidden layer, as indicated by the network of connections). We keep pushing values through the network in this manner until we have our complete output layer, at which point we are finished.

We've skipped over some important details in this high-level overview, but with this general information about what a neural network looks like and the terminology associated with it, we can now dig into the details a little deeper.

4.3.1 Adaptive Basis Functions

As we mentioned in the introduction, the strength of neural networks is that we can learn an effective basis for our problem domain at the same time as we train the parameters of our model. In fact, learning this basis becomes just another part of our parameter training. Let's make this notion of learning a basis more concrete.

Thinking back to our chapter on linear regression, we were training a model that made predictions using a functional form that looked like:

$$y(\mathbf{x}, \mathbf{w}) = \mathbf{w}\boldsymbol{\phi}^T = \sum_{d=1}^{D} w_d \boldsymbol{\phi}_d$$

where $\boldsymbol{\phi} = \phi(\mathbf{x})$, ϕ is the basis transformation function, and D is the dimensionality of the data point.

Typically with this linear regression setup, we are training our model to optimize the parameters $\mathbf{w}$. With neural networks this is no different— we still train to learn those parameters. However, the difference in the neural network setting is that the basis transformation function ϕ is no longer fixed. Instead, the transformations are incorporated into the model parameters, and thus learned at the same time.

This leads to a different functional form for neural networks. A neural network with M nodes in its first hidden layer performs M linear combinations of an input data point $\mathbf{x}$:

$$a_j^{(1)} = \sum_{d=1}^{D} w_{jd}^{(1)} x_d + w_{j0}^{(1)} \quad \forall j \in 1..M \tag{4.2}$$

Here, we use $a^{(1)}$ to denote the activation of a unit in layer 1 and notation $w^{(1)}$ denotes the weights used to determine the activations in layer 1. We also make the bias explicit. We will still use the bias trick in general, but we've left it out here to explicitly illustrate the bias term $w_{j0}^{(1)}$. Other than this, equation 4.2 describes what we've already seen in Figure 4.2. The only difference is that we index each node in the hidden layer (along with its weights) by j.

The M different values $a_j^{(1)}$ are the activations. We transform these activations with a non-linear activation function $h(\cdot)$ to give:

$$z_j^{(1)} = h(a_j) \tag{4.3}$$

⋆ Note that we didn't mention activation functions in the previous section only for the sake of simplicity. These non-linearities are crucial to the performance of neural networks because they allow for modeling of outcomes that vary non-linearly with their input variables.

These values $z_j^{(1)}$ correspond to the outputs of the hidden units, each of which is associated with an activation function. Superscript $^{(1)}$ indicates they are the outputs of units in layer 1. A typical activation function is the sigmoid function, but other common choices are the *tanh function* and *rectified linear unit (ReLU)*.

These output values $z_j^{(1)}$, for units $j \in \{1, \ldots, M\}$ in layer 1, form the inputs to the next layer. The activation of unit j' in layer 2 depends on the outputs from layer 1 and the weights $w_{j'0}^{(2)}, w_{j'1}^{(2)}, \ldots, w_{j'M}^{(2)}$ that define the linear sum at the input of unit j':

$$a_{j'}^{(2)} = \sum_{j=1}^{M} w_{j'm}^{(2)} z_j^{(1)} + w_{j'0}^{(2)} \tag{4.4}$$

We can connect many layers together in this way. They need not all have the same number of nodes but we will adopt M for the number of nodes in each layer for convenience of exposition. Eventually, we will reach the output layer, and each output is denoted y_k, for $k \in \{1, \ldots, K\}$. The final activation function may be the sigmoid function, softmax function, or just linear (and thus no transform).

We can now examine a more complete diagram of a feed-forward neural network, shown in Figure 4.3. It may be helpful to reread the previous paragraphs and use the diagram to visualize how a neural network transforms its inputs. This is a single hidden layer, or two-layer, network. Here, we use z to denote the output values of the units in the hidden layer.

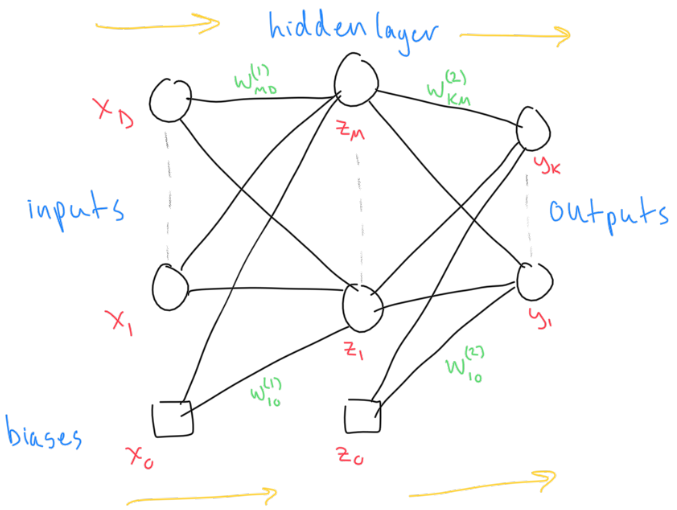


Figure 4.3: Feed-Forward Neural Network.

⋆ Different resources choose to count the number of layers in a neural net in different ways. We've elected to count each layer of non-input nodes, thus the two-layer network in Figure 4.3. However, some resources will choose to count every layer of nodes (three in this case) and still others count only the number of hidden layers (making this a one layer network).

Combining Figure 4.3 and our preceeding functional description, we can describe the operation performed by a two-layer neural network using a single functional transformation (with m to index a unit in the hidden layer):

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{m=1}^{M} w_{km}^{(2)} h\left(\sum_{d=1}^{D} w_{md}^{(1)} x_d + w_{m0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{4.5}$$

where we've elected to make the final activation function the sigmoid function $\sigma(\cdot)$, as is suitable for binary classification. We use h to denote the non-linear activation function for a hidden unit. Written like this, a neural network is simply a non-linear function that transforms an input $\mathbf{x}$ into an output $\mathbf{y}$ that is controlled by our set of parameters $\mathbf{w}$.

Furthermore, we see now why this basic variety of neural networks is a *feed-forward neural network*. We're simply feeding our input $\mathbf{x}$ forward through the network from the first layer to the last layer. Assuming we have a fully trained network, we can make predictions on new input data points by propagating them through the network to generate output predictions ("the forward pass").

We can also simplify this equation by utilizing the bias trick and appending an $x_0 = 1$ value to each of our data points such that:

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{m=1}^{M} w_{km}^{(2)} h\left(\sum_{d=1}^{D} w_{md}^{(1)} x_d\right)\right)$$

Finally, it's worth considering that while a neural network is a series of linear combinations, it is special because of the differentiable non-linearities applied at each of the hidden layers. Without these non-linearities, the successive application of different network weights would be equivalent to a single large linear combination.

4.4 Network Training

Now that we understand the structure of a basic feed-forward neural network and how they can be used to make predictions, we turn our attention to the training process.

4.4.1 Objective Function

To train our network, it's first necessary to establish an objective function. Remember that neural networks can be used to solve both regression and classification problems, which means that our choice of objective will depend on the type of problem and the properties we desire.

For the case of *linear regression*, a common objective function is the *least squares loss*:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(y(\mathbf{x}_n, \mathbf{w}) - y_n\Big)^2, \tag{4.6}$$

where y_n is the target value on example n. Sometimes we will have a regression problem with multiple outputs, in which case the loss would also take the sum over these different target values.

For a *binary classification* problem, which we model through a single, sigmoid output activation unit, then negated log-likelihood (or cross-entropy) is the typical loss function:

$$\mathcal{L}(\mathbf{w}) = -\sum_{n=1}^{N}\Big(y_n \ln \hat{y}_n + (1 - y_n)(\ln(1 - \hat{y}_n)\Big) \tag{4.7}$$

For a *multiclass classification problem*, produced by a softmax function in the output activation layer, we would use the negated log likelihood (cross entropy) loss:

$$\mathcal{L}(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{kn} \ln\left(\frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_{j=1}^{K}\exp(a_j(\mathbf{x}, \mathbf{w}))}\right) \tag{4.8}$$

⋆ Loss function and objective function all refer to the same concept: the function we optimize to train our model.

4.4.2 Optimizing Parameters

We want to find weight parameters $\mathbf{w}$ to minimize the objective function. The highly non-linear nature of neural networks means that this objective function will be non-convex with respect to our weight parameters. But still, we make use of stochastic gradient descent for optimization (refer to the previous chapter for a refresher).

In order to use gradient descent, we first need to figure out how to compute the gradient of our objective function with respect to our weights. That is the topic of the next section.

4.4.3 Backpropagation

Considering how our feed-forward neural network works, by propagating activations through our network to produce a final output, it's not immediately clear how we can compute gradients for the weights that lie in the middle of our networ. There is an elegant solution to this, which comes from "sending errors backwards" through our network, in a process known as *backpropagation*.

Definition 4.4.1 (Backpropagation):
Backpropagation is the procedure by which we pass errors backwards through a feed-forward neural network in order to compute gradients for the weight parameters of the network.

Backpropagation refers specifically to the portion of neural network training during which we compute the derivative of the objective function with respect to the weight parameters. This is done by propagating errors backwards through the network, hence the name.

⋆ Note that we still need to update the value of the weight parameters after computing their derivatives. This is typically done using gradient descent or some variant of it.

We now explore the details of backpropagation in greater depth.

4.4.4 Computing Derivatives Using Backpropagation

Recall that the activation $a_j^{(\ell)}$ for a node j in layer ℓ of a neural network can be described by the equation:

$$a_j^{(\ell)} = \sum_{m=1}^{M} w_{jm}^{(\ell)} z_m^{(\ell-1)}, \tag{4.9}$$

where there are M incoming nodes, each with corresponding output values $z_1^{(\ell-1)}, ..., z_M^{(\ell-1)}$, and with the weights in layer ℓ corresponding to node j denoted by $w_{j1}^{(\ell)}, ..., w_{jM}^{(\ell)}$. This activation is transformed by an activation function $h(\cdot)$ to give unit output $z_j^{(\ell)}$:

$$z_j^{(\ell)} = h(a_j^{(\ell)}). \tag{4.10}$$

Computing these values as we flow through the network constitutes the *forward pass* through our network.

We now wish to begin the process of computing derivatives of the objective function with respect to our weights. For the sake of simplicity, we'll assume that the current setting of our parameters $\mathbf{w}$ generates a loss of L for a single data point, as though we were performing stochastic gradient descent.

Let's consider how we could compute the derivative of L with respect to an individual weight in our network, $w_{jm}^{(\ell)}$ (the mth weight for activation j in layer ℓ):

$$\frac{\partial L}{\partial w_{jm}^{(\ell)}}. \tag{4.11}$$

We first need to figure out what the dependence of L is on this weight. This weight contributes to the final result only via its contribution to the activation $a_j^{(\ell)}$. This allows us to use the chain

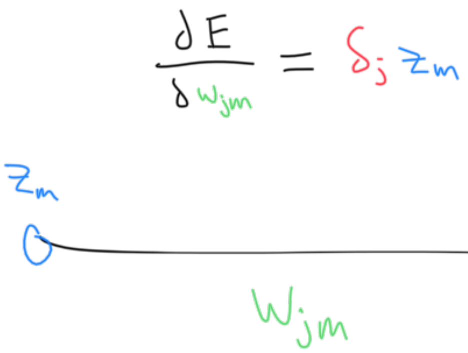


Figure 4.4: Gradient of the loss function in a neural network with respect to a weight. It depends on the input value z_m and the "error" corresponding to the activation value at the output end of the weight.

rule to simplify Equation 4.11 as:

$$\frac{\partial L}{\partial w_{jm}^{(\ell)}} = \frac{\partial L}{\partial a_j^{(\ell)}} \cdot \frac{\partial a_j^{(\ell)}}{\partial w_{jm}^{(\ell)}}. \tag{4.12}$$

The first part of this is the, typically non-linear, dependence of loss on activation. The second part is the linear dependence of activation on weight. Using Equation 4.9, we have that:

$$\frac{\partial a_j^{(\ell)}}{\partial w_{jm}^{(\ell)}} = z_m^{(\ell-1)},$$

and just the value of the input from the previous layer. We now introduce the following notation for the first term,

$$\delta_j^{(\ell)} = \frac{\partial L}{\partial a_j^{(\ell)}}, \tag{4.13}$$

where $\delta_j^{(\ell)}$ values are referred to as *errors*. We rewrite Equation 4.12 as:

$$\frac{\partial L}{\partial w_{jm}^{(\ell)}} = \delta_j^{(\ell)} z_m^{(\ell-1)}. \tag{4.14}$$

The implications of Equation 4.14 are significant for understanding backpropagation. The derivative of the loss with respect to an arbitrary weight in the network can be calculated as the product of the error $\delta_j^{(\ell)}$ at the "output end of that weight" and the value $z_m^{(\ell-1)}$ at the "input end of the weight." We visualize this property in Figure 4.4 (dropping the layer subscripting).

To compute the derivatives, it suffices to compute the values of δ_j for each node, also saving the output values z_m during the forward pass through the network (to be multiplied by the values of δ_j to get partials).

⋆ We will only have “errors values” δ_j for the hidden and output units of our network. This is logical because there is no notion of applying an error to our input data, which we have no control over.

We now consider how to compute these error values. For a unit in the output layer, indexing it here by k, and assuming the output activation function is linear and adopting least squares loss (i.e., regression), we have for the dependence of loss on the activation of this unit,

$$\delta_k^{(\ell)} = \frac{\partial L}{\partial a_k^{(\ell)}} = \frac{\partial L}{\partial \hat{y}_k} = \frac{\mathrm{d}\left(\frac{1}{2}(\hat{y}_k - y_k)^2\right)}{\mathrm{d}\hat{y}_k} = \hat{y}_k - y_k.$$

Here, we use shorthand $a_k^{(\ell)} = \hat{y}_k$, providing the kth dimension of the prediction of the model. Although a regression problem, we're imagining here that there are multiple regression targets (say, the height, weight and blood pressure of an individual). Here, y_k is the true target value for this data point. Note that this is for OLS. The expression would be different for a classification problem and negated log likelihood as the loss.

To compute the error $\delta_j^{(\ell)}$ for a hidden unit j in a layer ℓ, we again make use of the chain rule, and write:

$$\delta_j^{(\ell)} = \frac{\partial L}{\partial a_j^{(\ell)}} = \sum_{m=1}^{M} \frac{\partial L}{\partial a_m^{(\ell+1)}} \frac{\partial a_m^{(\ell+1)}}{\partial a_j^{(\ell)}}, \tag{4.15}$$

where the summation runs over all of the M nodes to which the node j in layer ℓ sends connections, as seen in Figure 4.5. This expression recognizes that the activation value of this unit contributes only via its contribution to the activation value of each unit to which it is connected in the next layer. The first term in one of the products in the summation is the, typically non-linear, dependence between loss and activation value of a unit in the next layer. The second term in one of the products captures the relationship between this activation and the subsequent activation.

Now, we can simplify by noticing that:

$$\frac{\partial L}{\partial a_m^{(\ell+1)}} = \delta_m^{(\ell+1)} \qquad \{\text{by definition}\} \tag{4.16}$$

$$\frac{\partial a_m^{(\ell+1)}}{\partial a_j^{(\ell)}} = \frac{\mathrm{d}h(a_j^{(\ell)})}{\mathrm{d}a_j^{(\ell)}} \cdot w_{mj}^{(\ell+1)} = h'(a_j^{(\ell)}) \cdot w_{mj}^{(\ell+1)}. \qquad \{\text{chain rule}\} \tag{4.17}$$

Substituting, and pulling forward the derivative of the activation function, we can rewrite the expression for the error on a hidden unit j in layer ℓ as:

$$\delta_j^{(\ell)} = h'(a_j^{(\ell)}) \sum_{m=1}^{M} w_{mj}^{(\ell+1)} \delta_m^{(\ell+1)}. \tag{4.18}$$

This is very useful, and is the key insight in backpropagation. It means that the value of the errors can be computed by “passing back” (backpropagating) the errors for nodes farther up in the network! Since we know the values of δ for the final layer of output node, we can recursively apply Equation 4.18 to compute the values of δ for all the nodes in the network.

Remember that all of these calculations were done for a single input data point that generated the loss L. If we were using SGD with mini-batches, then we would perform same calculation for each data point in mini-batch B, and average the gradients as follows:

$$\frac{\partial L}{\partial w_{jm}^{(\ell)}} = \frac{1}{|B|} \sum_{n \in B} \frac{\partial L_n}{\partial w_{jm}^{(\ell)}}, \tag{4.19}$$

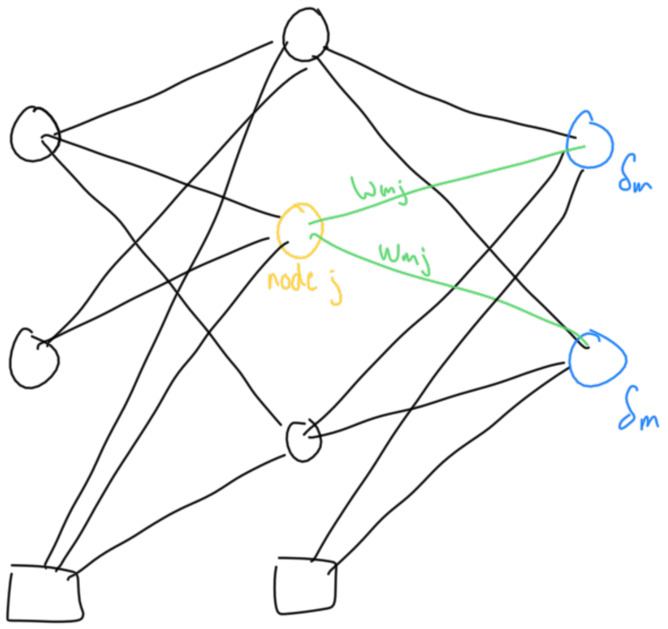


Figure 4.5: Summation over the nodes (blue) in layer $\ell + 1$ to which node j in layer ℓ (gold) sends connections (green). Note: read this as m and m'.

where L_n is the loss on example n.

To solidify our understanding of the backpropagation algorithm, it can be useful to try a concrete example.

Example 4.1 (Backpropagation Example): Imagine the case of a simple two layer neural network as in Figure 4.3, with K outputs (we denote them $\hat{y}_1$ through $\hat{y}_K$ for a given data point). We imagine this is a regression problem, but one with multiple dimensions to the output, and assume OLS. For a given data point, the loss is computed as:

$$L = \sum_{k=1}^{K} \frac{1}{2}(\hat{y}_k - y_k)^2,$$

where we write y_k for the kth dimension of the target value. For a unit in the hidden layer, with activation value a, we make use of the sigmoid activation, with

$$z = \sigma(a) = \frac{1}{1 + \exp(-a)},$$

whose derivative is given by:

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a)).$$

For an input data point $\mathbf{x}$, we forward propagate through the network to get the activations of the hidden layer, and for each m in this layer we have:

$$a_m^{(1)} = \sum_{d=0}^{D} w_{md}^{(1)} x_d,$$

given weights $w^{(1)}_{m0}, \dots, w^{(1)}_{mD}$, and with output value from unit m as,

$$z^{(1)}_m = \sigma(a^{(1)}_m).$$

We propagate these output values forward to get the outputs, and for each output unit k, we have:

$$\hat{y}_k = \sum_{m=0}^{M} w^{(2)}_{km} z^{(1)}_m,$$

where $w^{(2)}_{k0}, \dots, w^{(2)}_{kM}$ are the weights for unit k.

Now that we've propagated forward, we propagate our errors backwards! We start by computing the errors for the output layer as follows:

$$\delta^{(2)}_k = \frac{\partial L}{\partial \hat{y}_k} = \hat{y}_k - y_k.$$

We then backpropagate these errors back to each hidden unit m in layer 1 as follows:

$$\begin{aligned}\delta^{(1)}_m = \frac{\partial L}{\partial a^{(1)}_m} &= h'(a^{(1)}_m) \sum_{k=1}^{K} w^{(2)}_{km} \delta^{(2)}_k \\ &= \sigma(a^{(1)}_m)(1 - \sigma(a^{(1)}_m)) \sum_{k=1}^{K} w^{(2)}_{km}(\hat{y}_k - y_k) \\ &= z^{(1)}_m(1 - z^{(1)}_m) \sum_{k=1}^{K} w^{(2)}_{km}(\hat{y}_k - y_k).\end{aligned}$$

And now that we have our errors for the hidden and output layers, we can compute the derivative of the loss with respect to our weights as follows, for the dth weight on the mth unit in layer 1, and the mth weight on the kth unit in layer 2:

$$\frac{\partial L}{\partial w^{(1)}_{md}} = \delta^{(1)}_m x_d, \quad \frac{\partial L}{\partial w^{(2)}_{km}} = \delta^{(2)}_k z^{(1)}_m.$$

We then use these derivatives along with an optimization technique such as stochastic gradient descent to improve the model weights.

4.5 Choosing a Network Structure

Now that we know the general form of a neural network and how the training process works, we must step back and consider the question of how we actually arrive at an optimal network structure. We'll begin with an idea we've already seen before: cross validation.

4.5.1 Cross Validation for Neural Networks

We've previously discussed cross validation in the chapter on linear regression. We used it then to compare the performance of different models, attempting to identify the best model while also

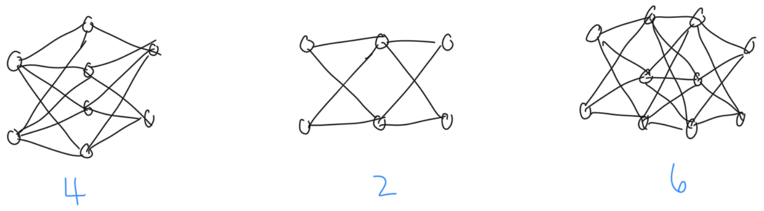


Figure 4.6: Networks with different structures and numbers of internal nodes.

avoiding overfitting. We can use a similar process to identify a reasonable network structure.

First of all, the input and output parameters of a neural network are generally decided for us: the dimensionality of our input data dictates the number of input units and the dimensionality of the required output dictates the number of output units. For example, if we have an 8-by-8 pixel image and need to predict whether it is a ‘0’ or a ‘1’, our input dimensions are fixed at 64 and our output dimensions are fixed at 2. Depending on whether you wish to perform some sort of pre or post-processing on the inputs/outputs of your network, this might not actually be the case, but in general when choosing a network structure we don’t consider the first or last layer of nodes as being a relevant knob that we can tune.

That leaves us to choose the structure of the hidden layers in our network. Unsurprisingly, the more hidden layers we have and the more nodes we have in each of those layers, the more variation we will produce in our results and the closer we will come to overfitting.

Thus, we can use cross validation in the same way we’ve done before: train our model with differing numbers of internal units and structures (as in Figure 4.6) and then select the model that performs best on the validation set.

⋆ There are other considerations at play beyond performance when choosing a network structure. For example, the more internal units you have in your network, the more storage and compute time you will need to train them. If either training time or response time after training a model is critical, you may need to consider consolidating your network at the expense of some performace.

4.5.2 Preventing Overfitting

Besides keeping your neural network small, there are other means of preventing it from overfitting.

Regularization

You can also apply regularization to the weights in your network to help prevent overfitting. For example, we could introduce a simple quadratic regularizer of the form $\frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$ to our objective function. There are other considerations to be made here, for example we would like our regularizer to be invariant to scaling, meaning that multiplying our input data by a constant would produce a proportionally equivalent network after training. The quadratic regularizer is not invariant to scaling, but the basic concept of avoiding extreme weights is the same nonetheless.

Data Augmentation

We can use transformations to augment our data sets, which helps prevent overfitting. This technique is not specific to neural networks, but often the types of unstructured data for which we use neural networks can benefit greatly from it.

Definition 4.5.1 (Data Augmentation): Data augmentation refers to the practice of increasing the size and diversity of your training data by applying transformations to the initial data set.

For example, if we are working with image data, we might choose to rotate or reflect the image, depending on the type of network we are trying to build and whether or not this would preserve the integrity of the image. We might also change something like the brightness or density of the image data. In this way, we can produce more and more varied training points, thus reducing the likelihood of overfitting.

4.6 Specialized Forms of Neural Networks

Simple neural networks are useful for a general set of tasks, and as universal function approximators, they *could* be useful for any task. However, there are certain data types and use cases for which we've developed more specialized forms of neural networks that perform even better in their respective domains. We will take a high level view of these different flavors of neural networks.

4.6.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks (abbreviated CNNs) are most often used for image data, but their underlying principles apply in other domains as well.

To understand why a CNN is useful, consider this specific problem: you are trying to determine whether or not there is a dog in an image. There are two general difficulties we have to deal with in solving this problem. First, while dogs have a lot of similar features (ears, tails, paws, etc.), we need some means of breaking an image down into smaller pieces that we can identify as being ears or tails or paws. Second, what happens if we train on images of dogs that are all in the center of the photo, and then we try to test our network on an image where the dog is in the upper left hand corner? It's going to fail miserably.

CNNs overcome these problems by extracting smaller local features from images via what's known as a *sliding window*. You can imagine this sliding window as a matrix kernel that moves over every subsection of an image, producing a summary of those subsections that feed into the next layer in our network. We do this over the entire image, and with several different sliding windows. Without going into too many details, this solves the two general problems we had above: our small sliding window can summarize a feature of interest (such as a dog ear) and it is also location invariant, meaning that we can identify that dog ear anywhere in an image.

4.6.2 Recurrent Neural Networks (RNNs)

As with CNNs, recurrent neural networks (abbreviated RNNs) are used to more efficiently solve a specific problem type. To motivate the structure of an RNN, we will turn again to a specific example.

Imagine we were building a tool with the goal of predicting what word comes next in a newspaper article. Obviously, the words that came before the word that we are currently trying to predict are crucial to predicting the next word. Imagine we propagate the preceding ten words through our network to predict which word we think will come next. It would also be useful if we could send some of the information at each layer backwards through the network to help with the next prediction - since we know the sequence of words matters. In this sense, our network is 'stateful'

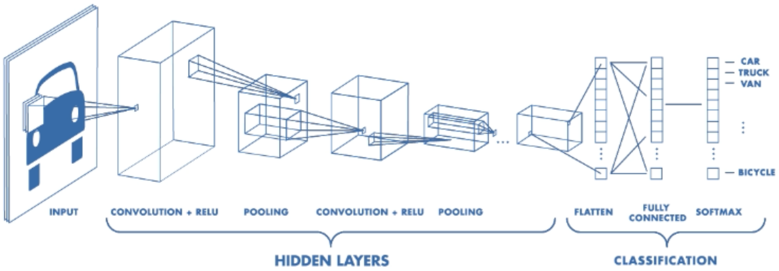


Figure 4.7: Excellent diagram of the structure of a CNN. source: https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks–1489512765771.html

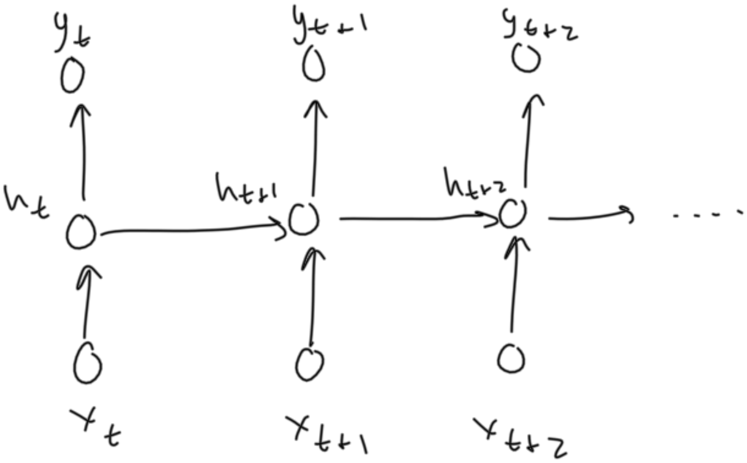


Figure 4.8: Simple example of an RNN.

because it's remembering what came before. We don't have this ability with a feed-forward network, which by design only propagates information forward through the network. RNNs add backward passing of activations into their network structure to improve predictions on data where there is some temporal dependence on what came previously.

4.6.3 Bayesian Neural Networks (BNNs)

Up until now, our training process has been one of maximum likelihood estimation, or a maximum posterior approach if we utilize a regularizer that can be interpreted as introducing a prior.

A Bayesian neural network, or BNN, does exactly what you might imagine: it introduces a distribution over the parameters of our model, which then requires marginalizing over those distributions in order to make a prediction. The specifics of how exactly a BNN is constructed are beyond the scope of this textbook, but the idea behind why we would utilize a BNN is the same as the reason we utilize Bayesian techniques in other domains, particularly the use of prior information to aid model performance.

Chapter 5

Support Vector Machines

In this chapter, we will explore what are known as a support vector machines, or SVMs for short. SVMs are broadly useful for problems in classification and regression, and they are part of a family of techniques known as *margin methods*. The defining goal of margin methods, and SVMs specifically, is to put as much distance as possible between data points and decision boundaries. We will dig deeper into what exactly this means over the course of the chapter. One of the most appealing aspects of SVMs is that they can be solved as convex optimization problems, for which we can find a global optimum with relative ease. We will explore the mathematical underpinnings of SVMs, which can be slightly more challenging than our previous topics, as well as their typical use cases.

5.1 Motivation

While SVMs can be used for classification or regression, we will reason about them in the classification case as it is more straightforward.

The grand idea behind SVMs is that we should construct a linear hyperplane in our feature space that maximally separates our classes, which means that the different classes should be as far from that hyperplane as possible. The distance of our data from the hyperplane is known as *margin*.

Definition 5.1.1 (Margin): Margin is the distance of the nearest data point from the separating hyperplane of an SVM model, as seen in Figure 5.1. Larger margins often lead to more generalizable models.

A larger margin tends to mean that our model will generalize better, since it provides more wiggle room to correctly classify unseen data (think about new data being a perturbation on current data).

This idea of the margin of a separator is quite intuitive. If you were presented with Figure 5.1 and were asked to separate the two classes, you would likely draw the line that keeps data points as far from it as possible. SVMs and other margin-based methods will attempt to algorithmically recreate this intuition.

5.1.1 Max Margin Methods

SVMs are a specific instance of a broader class of model known as *max margin methods*. Their name describes them well: they deal with creating a maximum margin between training data and

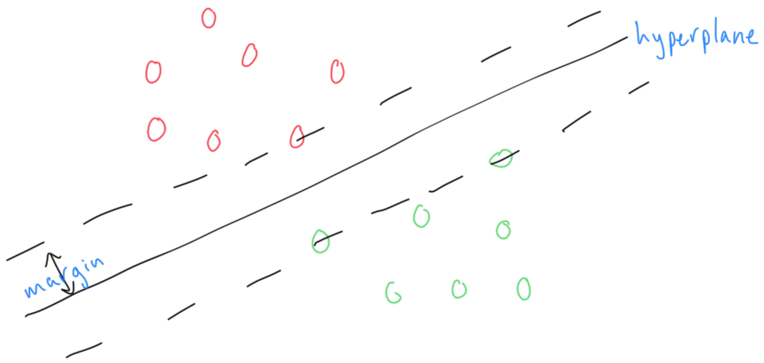


Figure 5.1: Hyperplane with margin between different classes.

decision boundary, with the idea that this leads to model generalizability.

Other max margin methods are outside the scope of this textbook. These alternative methods may differ from SVMs in a non-trivial way. For example, SVMs do not produce probabilities on different classes, but rather decision rules for handling new data points. If you needed probabilities, there are other max margin methods that can be used for the task.

ML Framework Cube: Support Vector Machines

SVMs are typically used in settings with discrete outputs. We need labeled training data to identify the relevant hyperplane in an SVM model. Finally, SVMs operate in a non-probabilistic setting.

Domain	***Training***	***Probabilistic***
Discrete	Supervised	No

5.1.2 Applications

The theory behind SVMs has been around for quite some time (since 1963), and prior to the rise of neural networks and other more computationally intensive techniques, SVMs were used extensively for image recognition, object categorization, and other typical machine learning tasks.

SVMs are still widely used in practice, for example for classification problems known as anomaly detection.

⋆ The purpose of anomaly detection is to identify unusual data points. For example, if we are manufacturing shoes, we may wish to inspect and flag any shoe that seems atypical with respect to the rest of the shoes we produce.

Anomaly detection can be as simple as a binary classification problem where the data set is comprised of anomalous and non-anomalous data points. As we will see, an SVM can be constructed from this data set to identify future anomalous points very efficiently. SVMs extend beautifully to settings where we want to use basis functions, and thus non-linear interactions on features. For this reason, they continue to be competitive in many real-world situations where these kinds of interactions are important to work with.

5.2 Hard Margin Classifier for Linearly Separable Data

We will learn the theory behind SVMs by starting with a simple two-class classification problem, as we've seen several times in previous chapters. We will constrain the problem even further by assuming, at least to get started, that the two classes are linearly separable, which is the basis of the *hard margin* formulation for SVMs.

⋆ The expression 'hard margin' simply means that we don't allow any data to be classified incorrectly. If it's not possible to find a hyperplane that perfectly separates the data based on class, then the hard margin classifier will return no solution.

5.2.1 Why the Hard Margin

The hard margin constraint, which assumes that our data is linearly separable, is not actually a requirement for constructing an SVM, but it simplifies the problem initially and makes our derivations significantly easier. After we've established the hard margin formulation, we will extend the technique to work in situations where our data is not linearly separable.

5.2.2 Deriving our Optimization Problem

Recall that our goal is to define a hyperplane that separates our data points and maintains the maximum possible distance between the hyperplane and nearest data points on either side of it. There are N examples $\mathbf{x}_1, ..., \mathbf{x}_N$ and there is a bias term w_0. Each example has a label $y_1, ..., y_N$ which is either 1 or -1. To uncover this hyperplane, we start with a simple linear model for a two-class classification problem:

$$h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + w_0. \tag{5.1}$$

This is the *discriminant function* and we classify a new example to class 1 or -1 according to the sign produced by our trained model $h(\mathbf{x})$. Later, we will also make this more general by using a basis function, $\phi(\mathbf{x})$ to transform to a higher dimensional feature space.

By specifying our model this way, we have implicity defined a hyperplane separating our two classes given by:

$$\mathbf{w}^\top \mathbf{x} + w_0 = 0 \tag{5.2}$$

Furthermore, we have that $\mathbf{w}$ is orthogonal to the hyperplane, which we demonstrate now:

Derivation 5.2.1 (Hyperplane Orthogonal to w): Imagine two data points $\mathbf{x}_1$ and $\mathbf{x}_2$ on the hyperplane defined by $\mathbf{w}^\top \mathbf{x} + w_0 = 0$. When we project their difference onto our model $\mathbf{w}$, we find:

$$\mathbf{w}^\top (\mathbf{x}_1 - \mathbf{x}_2) = \mathbf{w}^\top \mathbf{x}_1 - \mathbf{w}^\top \mathbf{x}_2 = -w_0 - (-w_0) = 0 \tag{5.3}$$

which means that $\mathbf{w}$ is orthogonal to our hyperplane. We can visualize this in Figure 5.2.

Remember that we're trying to maximize the margin between our training data and the hyperplane. The fact that $\mathbf{w}$ is orthogonal to our hyperplane will help with this.

To determine the distance between a data point $\mathbf{x}$ and the hyperplane, which we denote d, we need the distance in the direction of $\mathbf{w}$ between the point and the hyperplane. We denote $\mathbf{x}_p$ to be

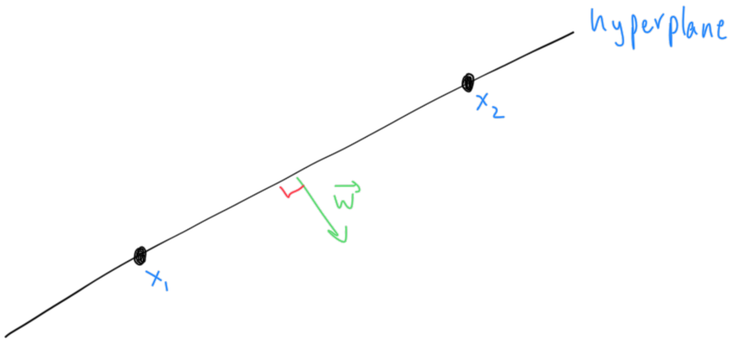


Figure 5.2: Our weight vector **w** is orthogonal to the separating hyperplane.

the projection of **x** onto the hyperplane, which allows us to decompose **x** as the following:

$$\mathbf{x} = \mathbf{x}_p + d\frac{\mathbf{w}}{||\mathbf{w}||_2} \tag{5.4}$$

which is the sum of the portion of the projection of **x** onto the hyperplane and the portion of **x** that is parallel to **w** (and orthogonal to the hyperplane). From here we can solve for d:

Derivation 5.2.2 (Distance from Hyperplane Derivation): We start by left multiplying Equation 5.4 with $\mathbf{w}^\top$,

$$\mathbf{w}^\top\mathbf{x} = \mathbf{w}^\top\mathbf{x}_p + d\frac{\mathbf{w}^\top\mathbf{w}}{||\mathbf{w}||_2}.$$

Simplifying (note that $\mathbf{w}^\top\mathbf{x}_p = -w_0$ from Equation 5.3):

$$\mathbf{w}^\top\mathbf{x} = -w_0 + d||\mathbf{w}||_2$$

Rearranging:

$$d = \frac{\mathbf{w}^\top\mathbf{x} + w_0}{||\mathbf{w}||_2}.$$

For each data point **x**, we now have the signed distance of that data point from the hyperplane.

For an example that is classified correctly, this signed distance d will be positive for class $y_n = 1$, and negative for class $y_n = -1$. Given this, we can make the distance unsigned (and always positive) for a correctly classified data point by multiplying by y_n. Then, the *margin for an correctly classified data point* $(\boldsymbol{x}_n, y_n)$ is given by:

$$\frac{y_n(\mathbf{w}^\top\mathbf{x}_n + w_0)}{||\mathbf{w}||_2}. \tag{5.5}$$

The margin for an entire data set is given by the margin to the closest point in the data set, and

$$\min_n \frac{y_n(\mathbf{w}^\top\mathbf{x}_n + w_0)}{||\mathbf{w}||_2}. \tag{5.6}$$

Then, it is our goal to maximize this margin with respect to our model parameters $\mathbf{w}$ and w_0. This is given by:

$$\max_{\mathbf{w},w_0} \frac{1}{||\mathbf{w}||_2} \left[\min_n y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \right] \tag{5.7}$$

Here, we pull the $1/||\mathbf{w}||_2$ term forward. Note carefully that w_0 does not play a role in the denominator $||\mathbf{w}||_2$.

This is a hard problem to optimize, but we can make it more tractable by recognizing some important features of Equation 5.7. First, rescaling $\mathbf{w} \to \alpha\mathbf{w}$ and $w_0 \to \alpha w_0$, for any $\alpha > 0$, has no impact on the margin for any correctly classified data point $\mathbf{x}_n$. This is because the effect of α cancels out in the numerator and denominator of Equation 5.5.

We can use this rescaling liberty to enforce

$$y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1, \qquad \text{for all } n. \tag{5.8}$$

This does not change the optimal margin because we can always scale up both $\mathbf{w}$ and w_0 by $\alpha > 0$ to achieve $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1$, and without affecting the margin. Moreover, since the problem is to maximize $1/||\mathbf{w}||_2$, an optimal solution will want $||\mathbf{w}||_2$ to be as small as possible, and thus at least one of these constraints (5.8) will be binding and equal to one in an optimal solution.

Thus our optimization problem now looks like:

$$\max_{\mathbf{w},w_0} \frac{1}{||\mathbf{w}||_2} \quad \text{s.t.} \quad y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1, \quad \text{for all } n. \tag{5.9}$$

Here, we recognized that $\min_n y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) = 1$ in an optimal solution with these new constraints, and adopted this in the objective. This simplifies considerably, removing the "min $_n$" part of the objective.

Notice that maximizing $\frac{1}{||\mathbf{w}||_2}$ is equivalent to minimizing $||\mathbf{w}||_2^2$. We will also add a constant term $\frac{1}{2}$ for convenience, leaving the **hard-margin formulation of the training problem**:

$$\min_{\mathbf{w},w_0} \frac{1}{2}||\mathbf{w}||_2^2 \quad \text{s.t.} \quad y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1, \quad \text{for all } n. \tag{5.10}$$

Note that Equation 5.10 is now a quadratic programming problem, which means we wish to optimize a quadratic function subject to a set of linear constraints on our parameters. Arriving at this form was the motivation for the preceding mathematic manipulations. We will discuss shortly how we actually optimize this function.

5.2.3 What is a Support Vector

Up until now, we have discussed Support Vector Machines without identifying what a support vector is. We now have enough information from the previous section to define them. Later, when we work with a dual formulation, we will see the precise role that they play.

Say that an example is on the *margin boundary* if $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) = 1$. These are the points that are closest to the hyperplane and have margin $1/||\mathbf{w}||_2$.

Definition 5.2.1 (Support Vector): A support vector in a hard-margin SVM formulation must be a data point that is on the margin boundary of the optimal solution, with $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) = 1$ and margin $1/||\mathbf{w}||_2$.

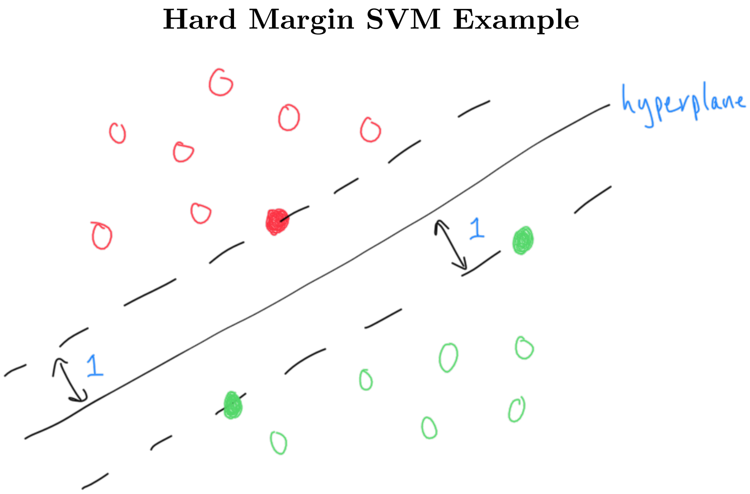


Figure 5.3: Example of the resulting hyperplane for a hard margin SVM. The filled in data points are support vectors in this example. A support vector for the hard-margin formulation must be on the margin boundary, with a discriminant value of +1 or -1.

In the hard margin case we have constrained the closest data points to have discriminant value $\mathbf{w}^\top \mathbf{x}_n + w_0 = 1$ (-1 for a negative example). Figure 5.3 shows a hard margin SVM solution with an illustration of corresponding support vectors.

⋆ After we have optimized an SVM in the hard margin case, we must have at least two support vectors with discriminant value that is 1 or -1, and thus a margin of $1/||\mathbf{w}||_2$.

⋆ Not every example on the margin boundary needs to be a support vector.

5.3 Soft Margin Classifier

Thus far, we've been operating under the assumption that our data is linearly separable in feature space, which afforded us several convenient guarantees in the derivations of the previous section. For example, given that our data was linearly separable, we could guarantee that every data point would be on the correct side of the hyperplane, which waswhat allowed us to enforce the constraint that $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1$. We now seek to generalize the work of the previous section to situations where our data is not separable.

5.3.1 Why the Soft Margin?

What if our data is not linearly separable in the feature space (even after applying a basis function)? This is the likely case in real applications! Unfortunately, our current hard margin SVM formulation would be useless with non-linearly separable data. That is why we need the soft margin SVM.

At a high level, the soft margin SVM allows for some of our data points to be closer to or even on the incorrect side of the hyperplane. This is desirable if our data set is not linearly separable or contains outliers, and it is also quite intuitive. Examining Figure 5.4, we see that we have a single outlier data point. We can still create a good model by just allowing this single data point to be

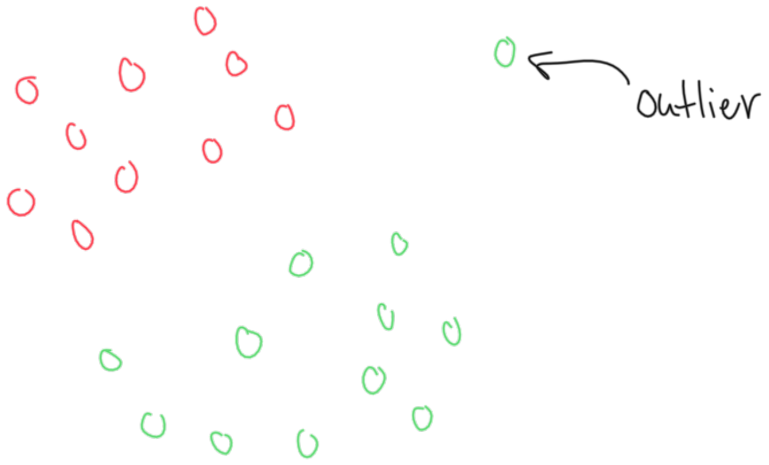


Figure 5.4: An outlier can make the hard margin formulation impossible or unable to generalize well.

close to the hyperplane (or in some cases, even on the wrong side of the hyperplane). That is what the soft margin formulation will allow for.

5.3.2 Updated Optimization Problem for Soft Margins

To enable the soft margin formulation, we introduce what are known as *slack variables* denoted $\xi_n \geq 0$, which simply relax the constraint from Equation 5.9 that we imposed in the hard margin formulation. Say that a data point is "inside the margin region" if its discriminant value is in the range $(-1, +1)$. There is a slack variable $\xi_n \geq 0$ for every data point $\mathbf{x}_n$, and they take the following values according to how we classify $\mathbf{x}_n$:

$$\xi_n = \begin{cases} = 0 & \text{if } \mathbf{x}_n \text{ is correctly classified} \\ \in (0, 1] & \text{if } \mathbf{x}_n \text{ is correctly classified but inside the margin region} \\ > 1 & \text{if } \mathbf{x}_n \text{ is incorrectly classified} \end{cases} \tag{5.11}$$

These slack variable penalize data points on the wrong side of the margin boundary, but they don't forbid us from allowing data points to be on the wrong side if this produces the best model. We now reformulate the optimization problem as follows. This is the **soft-margin training problem**:

$$\begin{aligned} \min_{\mathbf{w}, w_0} \; & \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_{n=1}^{N} \xi_n \\ \text{s.t.} \quad & y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1 - \xi_n, \quad \text{for all } n \\ & \xi_n \geq 0, \quad \text{for all } n. \end{aligned} \tag{5.12}$$

Here, C is a regularization parameter that determines how heavily we penalize violations of the hard margin constraints. A large C penalizes violation of the hard margin constraints more heavily, which means our model will follow the data closely and have small regularization. A small C won't heavily penalize having data points inside the margin region, relaxing the constraint and allowing our model to somewhat disregard more of the data. This means more regularization.

Soft Margin SVM Example

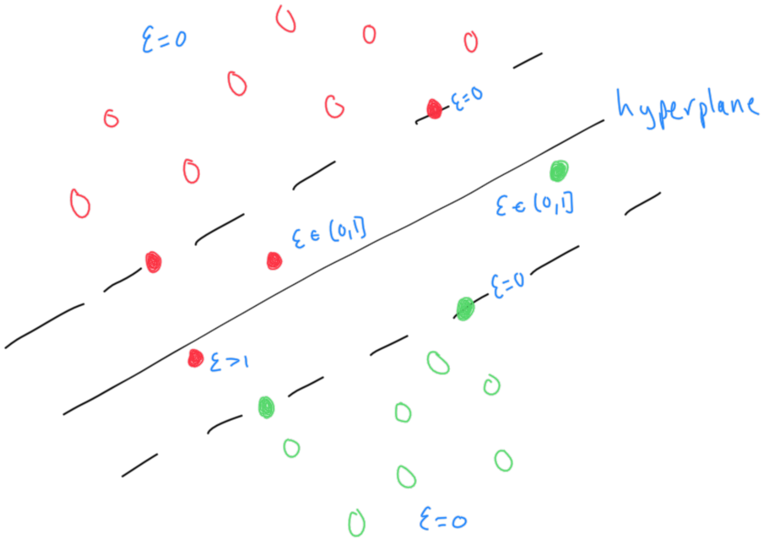


Figure 5.5: Example of the resulting hyperplane for a soft margin SVM. The filled in data points illustrate the support vectors in this example and must be either on the margin boundary or on the "wrong side" of the margin boundary.

⋆ Unlike most regularization parameters we've seen thus far, C increases regularization as it gets smaller.

5.3.3 Soft Margin Support Vectors

Under the the hard margin formulation, the support vectors were some subset of the data points with $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) = 1$, and so situated exactly on the margin boundary, and they points closest to the hyperplane.

The support vectors in the soft margin case must be data points that are either on the margin boundary (and thus $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) = 1$) or for which $\xi_n \in (0, 1]$ and in the margin region, or that are misclassified (when $\xi_n > 1$). We can visualize this in Figure 5.5.

⋆ Not every data point on the margin boundary, in the margin region, or that is misclassified needs to be a support vector in the soft-margin formulation. But those that become support vectors must meet one of these criteria.

5.4 Conversion to Dual Form

Now that we have the formulation of the optimization problem for SVMs, we need to discuss how we actually go about optimizing to produce a model solution. This will involve converting to a *dual form* of our problem. We will do this in the hard margin case for technical simplicity, but our solution can be easily modified to also apply to the soft margin formulation.

⋆ A dual form is an equivalent manner of representing an optimization problem, in this case the quadratic programming problem we need to optimize. Dual forms can be easier to work with than their initial form ("the primal form.")

The dual form will be useful because it will allow us to bring in a basis function into the SVM formulation in a very elegant and computationally efficient way.

5.4.1 Lagrange Multipliers

Before we get into deriving the dual form, we need to be aware of a critical piece of math that will enable us to solve our optimization problem: *Langrange multipliers.*

A Lagrange multiplier is used to find optima of a function subject to certain constraints. This is exactly what we need to solve the optimization problem described by Equation 5.10.

The underlying theory behind Lagrange multipliers is not overly difficult to understand, but it is beyond the scope of this textbook in its fully generality. We will simply offer the method by which you can use them to solve the optimization problem of interest here.

Recall from the "Section 0 math" that we understood there how to use the Lagrangian method to solve optimization problems with equality constraints, for example of the form min $f(\mathbf{w})$ s.t. $g_n(\mathbf{w}) = 0$, for each constraint n. There, a suitable Lagrangian function was $L(\mathbf{w}, \boldsymbol{\alpha}) = f(\mathbf{w}) + \sum_n \alpha_n g_n(\mathbf{w})$, where $\alpha_n \in \mathbb{R}$ were the Lagrangian variables, and this problem could be solved via the partial derivatives of L with respect to $\mathbf{w}$ and $\boldsymbol{\alpha}$, and setting them to zero.

Here, we have a slightly different problem to solve, which has the following form (recognizing that we can write a constraint $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) \geq 1$ as $-y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) + 1 \leq 0$, and thus as an "$\leq 0$" constraint):

$$\min_{\mathbf{w}} \; f(\mathbf{w}) \quad \text{s.t. } g_n(\mathbf{w}) \leq 0, \quad \text{all } n \tag{5.13}$$

To solve this, we make use of the same kind of Lagrangian function,

$$L(\mathbf{w}, \boldsymbol{\alpha}) = f(\mathbf{w}) + \sum_n \alpha_n g_n(\mathbf{w}). \tag{5.14}$$

But for the new "inequality form" of the constrained optimization problem, we also need to introduce a new subproblem, which for any fixed $\mathbf{w}$, solves

$$\max_{\boldsymbol{\alpha}} L(\mathbf{w}, \boldsymbol{\alpha}) \quad \text{s.t.} \quad \alpha_n \geq 0 \quad \text{for all } n \tag{5.15}$$

With this, we consider the problem

$$\min_{\mathbf{w}} \left[\max_{\boldsymbol{\alpha},\ \boldsymbol{\alpha} \geq 0} f(\mathbf{w}) + \sum_n \alpha_n g_n(\mathbf{w}) \right] \tag{5.16}$$

Now, if $\mathbf{w}$ violates one or more constraints in (5.13), then the subproblem (5.15) becomes unbounded, with α_n on the corresponding constraints driven arbitrarily large. Otherwise, if we have $g_n(\mathbf{w}) < 0$ then we will have $\alpha_n = 0$, and we conclude $\alpha_n g_n(\mathbf{x}) = 0$ in all optimal solutions to (5.16). Therefore, and assuming that problem (5.13) is feasible, we have $L(\mathbf{w}, \boldsymbol{\alpha}) = f(\mathbf{w})$ in an optimal solution to (5.16). Thus, we establish that (5.16) is an equivalent formulation to (5.13).

Substituting into our problem (5.10), the Lagrangian formulation becomes

$$\min_{\mathbf{w}, w_0} \left[\max_{\boldsymbol{\alpha},\ \boldsymbol{\alpha} \geq 0} \frac{1}{2} \mathbf{w}^\top \mathbf{w} + \sum_n \alpha_n (-y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) + 1) \right] \tag{5.17}$$

Equivalently, it is convenient to write this as

$$\min_{\mathbf{w},w_0}\left[\max_{\boldsymbol{\alpha},\ \boldsymbol{\alpha}\geq 0}\frac{1}{2}\mathbf{w}^\top\mathbf{w}-\sum_n \alpha_n(y_n(\mathbf{w}^\top\mathbf{x}_n+w_0)-1)\right], \tag{5.18}$$

with Lagrangian function

$$L(\mathbf{w},\boldsymbol{\alpha},w_0)=\frac{1}{2}\mathbf{w}^\top\mathbf{w}-\sum_n \alpha_n(y_n(\mathbf{w}^\top\mathbf{x}_n+w_0)-1). \tag{5.19}$$

5.4.2 Deriving the Dual Formulation

Using this Lagrangian function, we allow ourselves to switch from solving Equation 5.10 to instead solving:

$$\min_{\mathbf{w},w_0}\left[\max_{\boldsymbol{\alpha},\ \boldsymbol{\alpha}\geq 0} L(\mathbf{w},\boldsymbol{\alpha},w_0)\right] \tag{5.20}$$

⋆ The '$\min_{\mathbf{w},w_0}\max_{\boldsymbol{\alpha}\geq 0}$' in Equation 5.20 may be initially confusing. The way to read this is that for any choice of $\mathbf{w}$, w_0, the inner "max" problem then finds values of $\boldsymbol{\alpha}$ to try to "defeat" the outer minimization objective.

We now wish to convert the objective in Equation 5.20 to a dual objective. Under the sufficient conditions of strong duality which hold for this problem because Equation 5.10 has a quadratic objective and linear constraints (but whose explanation is beyond the scope of this textbook), we can equivalently reformulate the optimization problem (5.20) as:

$$\max_{\boldsymbol{\alpha},\ \boldsymbol{\alpha}\geq 0}\left[\min_{\mathbf{w},w_0} L(\mathbf{w},\boldsymbol{\alpha},w_0)\right]. \tag{5.21}$$

At this point, we can use first order optimality conditions to solve for $\mathbf{w}$, i.e., the inner minimization problem, for some choice of $\boldsymbol{\alpha}$ values. Taking the gradient, setting them equal to 0, and solving for $\mathbf{w}$, we have:

$$\begin{aligned}\nabla L(\mathbf{w},\boldsymbol{\alpha},w_0)&=\mathbf{w}-\sum_{n=1}^N \alpha_n y_n \mathbf{x}_n=0\\ \Leftrightarrow\qquad \mathbf{w}^*&=\sum_{n=1}^N \alpha_n y_n \mathbf{x}_n.\end{aligned} \tag{5.22}$$

When we do the same thing for w_0, we find the following:

$$\begin{aligned}\frac{\partial L(\mathbf{w},\boldsymbol{\alpha},w_0)}{\partial w_0}&=-\sum_{n=1}^N \alpha_n y_n=0\\ \Leftrightarrow\qquad \sum_{n=1}^N \alpha_n y_n&=0.\end{aligned} \tag{5.23}$$

This is interesting. If $\sum_n \alpha_n y_n < 0$, then $L(\cdot)$ is increasing with w_0 without bound, and the inner-minimization would choose $w_0=-\infty$ and this choice of $\boldsymbol{\alpha}$ cannot solve (5.21). If $\sum_n \alpha_n y_n > 0$, then $L(\cdot)$ is decreasing with w_0 without bound, and the inner-minimization would choose $w_0=+\infty$, and this choice of $\boldsymbol{\alpha}$ cannot solve (5.21). We need $\sum_n \alpha_n y_n = 0$ for an optimal solution to (5.21). So, we don't yet obtain the optimal value for w_0, but we do gain a new constraint on the α-values that will need to hold in an optimal solution.

Now we substitute for $\mathbf{w}^*$ into our Lagrangian function, and also assume (5.23), since this will be adopted as a new constraint in solving the optimization problem. Given this, we obtain:

$$\begin{aligned} L(\mathbf{w}, \boldsymbol{\alpha}, w_0) &= \frac{1}{2}\mathbf{w}^\top \mathbf{w} - \mathbf{w}^\top \sum_n \alpha_n y_n \mathbf{x}_n - w_0 \sum_n \alpha_n y_n + \sum_n \alpha_n \\ &= -\frac{1}{2}\mathbf{w}^\top \mathbf{w} + \sum_n \alpha_n \\ &= \sum_n \alpha_n - \frac{1}{2}\left(\sum_n \alpha_n y_n \mathbf{x}_n\right)^\top \left(\sum_{n'} \alpha_{n'} y_{n'} \mathbf{x}_{n'}\right) \end{aligned} \tag{5.24}$$

where the second equation follows from the first by using (5.22) and (5.23), and the third equation follows by using (5.22).

This is now entirely formulated in terms of $\boldsymbol{\alpha}$, and provides the **hard margin, dual formulation**:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_n \alpha_n - \frac{1}{2}\sum_n \sum_{n'} \alpha_n \alpha_{n'} y_n y_{n'} \mathbf{x}_n^\top \mathbf{x}_{n'} \\ \text{s.t.} \quad & \sum_n \alpha_n y_n = 0, \quad \text{for all n} \\ & \alpha_n \geq 0, \quad \text{for all n} \end{aligned} \tag{5.25}$$

Here, we add $\sum_n \alpha_n y_n = 0$ as a constraint. This is another quadratic objective, subject to linear constraints. This can be solved via SGD or another approach to solving convex optimization problems. With a little more work we can use the optimal $\boldsymbol{\alpha}$ values to make predictions.

Although the derivation is out of scope for this textbook, there is also a very similar **dual form for the soft-margin SVM training problem**:

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_n \alpha_n - \frac{1}{2}\sum_n \sum_{n'} \alpha_n \alpha_{n'} y_n y_{n'} \mathbf{x}_n^\top \mathbf{x}_{n'} \\ \text{s.t.} \quad & \sum_n \alpha_n y_n = 0, \quad \text{for all n} \\ & C \geq \alpha_n \geq 0, \quad \text{for all n} \end{aligned} \tag{5.26}$$

This puts an upper-bound on α_n to prevent the dual from being unbounded in the case where the hard-margin SVM problem is infeasible because the data cannot be separated. It is not yet clear why any of this has been useful. We will see the value of the dual formulation when working with basis functions.

⋆ By (5.22) we see how to find the weight vector $\mathbf{w}$ from a solution $\boldsymbol{\alpha}$. We didn't yet explain how to find w_0. That will be explained next.

5.4.3 Making Predictions

Substituting the optimal dual solution into the discriminant function, we have

$$h(\mathbf{x}) = \sum_{n=1}^{N} \alpha_n y_n \mathbf{x}_n^\top \mathbf{x} + w_0. \tag{5.27}$$

For data points with $\alpha_n > 0$, this is taking a weighted vote over examples in the training data based on the size of the inner product $\mathbf{x}_n^\top \mathbf{x}$.

Since $\boldsymbol{\alpha}$ are Lagrange multipliers, they are non-negative, and moreover, by reasoning about the "max subproblem" in the min-max formulation (5.16), we know that they take on value zero whenever $y_n h(\mathbf{x}_n) > 1$. The data points for which $\alpha_n > 0$ are known as **support vectors**, and they will must be data points that are either on the margin boundary, inside the margin region, or misclassified. For the hard-margin formulation they must be data points on the margin boundary.

This is a major takeaway for the usefulness of SVMs: once we've trained our model, we can discard most of our data. We only need to keep the support vectors to make predictions. Soon we also see the "kernel trick." This also illustrates why we need to solve for the values of $\boldsymbol{\alpha}$: those values dictate which data points are the support vectors for our model.

Solving for w_0

We can solve for w_0 by recognizing that $y_n(\mathbf{w}^\top \mathbf{x}_n + w_0) = 1$ for any data point on the margin boundary. For the hard-margin formulation we can solve for w_0 using any example for which $\alpha_n > 0$. For the soft-margin formulation, it can be shown that the only points with $\alpha_n = C$ are those inside the margin region or misclassified, and so that any point with $C > \alpha_n > 0$ is on the margin boundary. Any such point can be solved to solve for w_0.

5.4.4 Why is the Dual Formulation Helpful?

Beyond having a succinct formulation of the discriminant function (assuming the number of support vectors is small), you might be wondering what exactly we gained by moving to the dual formulation of this problem.

First, the complexity of the optimization problem we're solving changed from one that is dependent on the number of features D, i.e., the size of the weight vector, to one that is linearly dependent on the number of data points N. Thus, the number of variables to optimize over is now independent of the dimensionality of the feature space.

Second, in the dual formulation, we have the opportunity to take advantage of what's known as the *kernel trick* to map our data $\mathbf{x}_n$ into higher dimensions without incurring performance costs. This works as follows: notice that the only way in which a feature vector $\mathbf{x}$ is accessed during the training process (5.26) and at prediction time (5.27) is through the inner product $\mathbf{x}^\top \mathbf{z}$ between two data points. Suppose that we are using a basis function $\phi : \mathbb{R}^D \to \mathbb{R}^M$, so that this becomes $\phi(\mathbf{x})^\top \phi(\mathbf{z})$. Now, define *kernel function*

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z}). \tag{5.28}$$

The idea of the kernel trick is that *we might be able to compute $K(\cdot, \cdot)$ without actually working in the basis function space, $\mathbb{R}^M$, but rather be able to compute the Kernel function directly through algebra in the lower dimensional space, $\mathbb{R}^D$.*

For example, it can be shown that the polynomial kernel

$$K_{\text{poly}}(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^\top \mathbf{z})^q, \quad \text{for integers } q \geq 2 \tag{5.29}$$

corresponds to computing the inner product with a basis function that makes use of all terms up to degree q. When $q = 2$, then it is all constant, linear, and quadratic terms. The polynomial kernel function does this without needing to actually project the examples to the higher dimensional space. Rather it takes the inner product in the lower-dimensional space, adds 1 to this scalar, and then raises it to the power of q. The implicit basis is growing exponentially large in q!

⋆ The kernel trick can even be used to work in an infinite basis. This is the case with the Gaussian kernel. If that is of interest, you should look into Taylor series basis expansions and the Gaussian kernel.

The importance of the kernel trick is that when computations can be done efficiently in the initial space $\mathbb{R}^D$, then the training problem can be solved by computing the pairwise $K(\cdot,\cdot)$ values for all pairs of training examples, and then using SGD to solve the dual, soft-margin training problem (with N decision variables).

5.4.5 Kernel Composition

Now that we've seen the usefulness of the kernel trick for working in higher dimensional spaces without incurring performance costs or memory overhead, it's reasonable to wonder what sort of valid kernels we can construct.

To be explicit, by 'kernel' we mean a function $K(\mathbf{x},\mathbf{z}) = \phi(\mathbf{x})^\top \phi(\mathbf{z})$ that produces a scalar product from two vectors, as determined by some basis function ϕ.

Although it is beyond the scope of this textbook, the condition for a kernel function $K(\cdot,\cdot)$ to be valid is that the $N \times N$ matrix $\mathbf{K}$, where element $K_{n,n'} = K(\mathbf{x}_n, \mathbf{x}_{n'})$ should be positive semidefinite for *any* choices of the data set $\{\mathbf{x}_n\}$. This is known as Mercer's theorem.

⋆ The matrix $\mathbf{K}$ of elements $K(\mathbf{x}_n, \mathbf{x}_{n'})$ is known as the *Gram matrix* or *Kernel matrix*.

In practice, this provides for a logic for how different valid kernels can be composed (if they maintain a p.s.d. Gram matrix!).

There exists a set of rules that preserve the validity of kernels through transformations. These include such things as

- scalar multiplication $c \cdot K(\cdot,\cdot)$
- exponentiation $\exp\{K(\cdot,\cdot)\}$
- addition $K_1(\mathbf{x},\mathbf{z}) + K_2(\mathbf{x},\mathbf{z})$, and
- multiplication $K_1(\mathbf{x},\mathbf{z}) \cdot K_2(\mathbf{x},\mathbf{z})$.

It is always possible to test the validity of a given kernel by demonstrating that its Gram matrix $\mathbf{K}$ is positive semidefinite.

Chapter 6

Clustering

In this chapter, we will explore a technique known as clustering. This represents our first foray into unsupervised machine learning techniques. Unlike the previous four chapters, where we explored techniques that assumed a data set of inputs and targets, with the goal of eventually making predictions over unseen data, our data set will no longer contain explicit targets. Instead, these techniques are motivated by the goal of uncovering structure in our data. Identifying clusters of similar data points is a useful and ubiquitous unsupervised technique.

6.1 Motivation

The reasons for using an unsupervised technique like clustering are broad. We often don't have a specific task in mind; rather, we are trying to uncover more information about a potentially opaque data set. For clustering specifically, our unsupervised goal is to group data points that are similar.

There are many reasons why we might separate our data by similarity. For organizational purposes, it's convenient to have different classes of data. It can be easier for a human to sift through data if it's loosely categorized beforehand. It may be a preprocessing step for an inference method; for example, by creating additional features for a supervised technique. It can help identify which features make our data points most distinct from one another. It might even provide some idea of how many distinct data types we have in our set.

This idea of data being 'similar' means that we need some measure of *distance* between our data points. While there are a variety of clustering algorithms available, the importance of this distance measurement is consistent between them.

Distance is meant to capture how 'different' two data points are from each other. Then, we can use these distance measurements to determine which data points are similar, and thus should be clustered together. A common distance measurement for two data points $\mathbf{x}$ and $\mathbf{x}'$ is given by:

$$||\mathbf{x} - \mathbf{x'}||_{L2} = \sqrt{\sum_{d=1}^{D} (\mathbf{x}_d - \mathbf{x'}_d)^2} \tag{6.1}$$

where D is the dimensionality of our data. This is known as **L2** or **Euclidean distance**, and you can likely see the similarity to L2 regularization.

There are a variety of distance measurements available for data points living in a D-dimensional Euclidean space, but for other types of data (such as data with discrete features), we would need to select a different distance metric. Furthermore, the metrics we choose to use will have an impact on the final results of our clustering.

ML Framework Cube: Clustering

In unsupervised learning, the domain refers to the domain of the hidden variable z (which is analogous to y in supervised learning). In clustering, we have z's that represent the discrete clusters. Furthermore, the techniques that we explore in this chapter are fully unsupervised and non-probabilistic.

Domain	***Training***	***Probabilistic***
Discrete	Unsupervised	No

6.1.1 Applications

Here are a few specific examples of use cases for clustering:

1. Determining the number of phenotypes in a population.

2. Organizing images into folders according to scene similarity.

3. Grouping financial data as a feature for anticipating extreme market events.

4. Identifying similar individuals based on DNA sequences.

As we mentioned above, there are different methods available for clustering. In this chapter, we will explore two of the most common techniques: K-Means Clustering and Hierarchical Agglomerative Clustering. We also touch on the flavors available within each of these larger techniques.

6.2 K-Means Clustering

The high level procedure behind K-Means Clustering (known informally as k-means) is as follows:

1. Initialize cluster centers by randomly selecting points in our data set.

2. Using a distance metric of your choosing, assign each data point to the closest cluster.

3. Update the cluster centers based on your assignments and distance metric (for example, when using L2 distance, we update the cluster centers by averaging the data points assigned to each cluster).

4. Repeat steps 1 and 2 until convergence.

In the case where we are using the L2 distance metric, this is known as *Lloyd's algorithm*, which we derive in the next section.

6.2.1 Lloyd's Algorithm

Lloyd's algorithm, named after Stuart P. Lloyd who first suggested the algorithm in 1957, optimizes our cluster assignments via a technique known as coordinate descent, which we will learn more about in later chapters.

Derivation 6.2.1 (Lloyd's Algorithm Derivation):

We begin by defining the objective used by Lloyd's algorithm.

Objective

The loss function for our current assignment of data points to clusters is given by:

$$\mathcal{L}(\mathbf{X}, \{\boldsymbol{\mu}\}_{c=1}^{C}, \{\mathbf{r}\}_{n=1}^{N}) = \sum_{n=1}^{N}\sum_{c=1}^{C} r_{nc}||\mathbf{x}_n - \boldsymbol{\mu}_c||_2^2 \tag{6.2}$$

where $\mathbf{X}$ is our NxD data set (N is the number of data points and D is the dimensionality of our data), $\{\boldsymbol{\mu}\}_{c=1}^{C}$ is the CxD matrix of cluster centers (C is the number of clusters we chose), and $\{\mathbf{r}\}_{n=1}^{N}$ is our NxC matrix of *responsibility vectors.* These are one-hot encoded vectors (one per data point), where the 1 is in the position of the cluster to which we assigned the nth data point.

We now define the algorithmic portion of Lloyd's clustering procedure.

Algorithm

We first adjust our responsibility vectors to minimize each data point's distance from its cluster center. Formally:

$$r_{nc} = \begin{cases} = 1 & \text{if } c = \arg\min_{c'} ||\mathbf{x}_n - \boldsymbol{\mu}_{c'}|| \\ = 0 & \text{otherwise} \end{cases} \tag{6.3}$$

After updating our responsibility vectors, we now wish to minimize our loss by updating our cluster centers $\boldsymbol{\mu}_c$. The cluster centers which minimize our loss can be computed by taking the derivative of our loss with respect to $\boldsymbol{\mu}_c$, setting equal to 0, and solving for our new cluster centers $\boldsymbol{\mu}_c$:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_c} &= -2\sum_{n=1}^{N} r_{nc}(\mathbf{x}_n - \boldsymbol{\mu}_c) \\ \boldsymbol{\mu}_c &= \frac{\sum_{n=1}^{N} r_{nc}\mathbf{x}_n}{\sum_{n=1}^{N} r_{nc}} \end{aligned} \tag{6.4}$$

Intuitively, this is the average of all the data points $\mathbf{x}_n$ assigned to the cluster center $\boldsymbol{\mu}_c$.

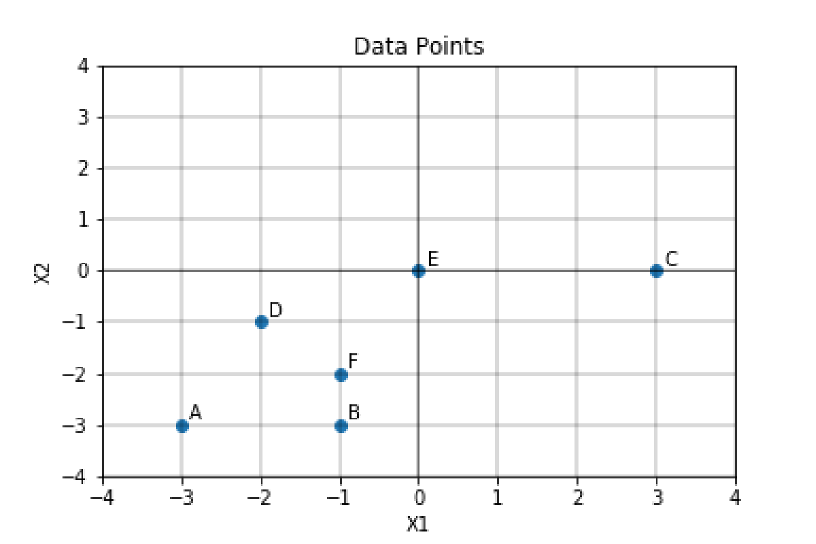
We then update our responsibility vectors based on the new cluster centers, update the cluster centers again, and continue this cycle until we have converged on a stable set of cluster centers and responsibility vectors.

Note that while Lloyd's algorithm is guaranteed to converge, it is only guaranteed to converge to a locally optimal solution. Finding the globally optimal set of assignments and cluster centers is an NP-hard problem. As a result, a common strategy is to execute Lloyd's algorithm several times with different random initializations of cluster centers, selecting the assignment that minimizes loss across the different trials. Furthermore, to avoid nonsensical solutions due to scale mismatch between features (which would throw our Euclidean distance measurements off), it makes sense to standardize our data in a preprocessing step. This is as easy as subtracting the mean and dividing by the standard deviation across each feature.

6.2.2 Example of Lloyd's

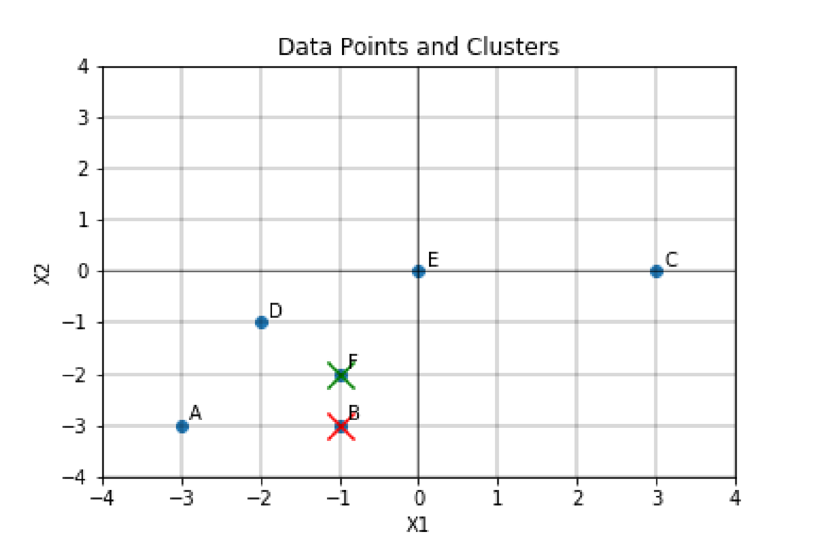
For some more clarity on exactly how Lloyd's algorithm works, let's walk through an example.

Example 6.1 (Lloyd's Algorithm Example): We start with a data set of size $N = 6$. Each data point is two-dimensional, with each feature taking on a value between -3 and 3. We also have a 'Red' and 'Green' cluster. Here is a table and graph of our data points, labelled A through F:

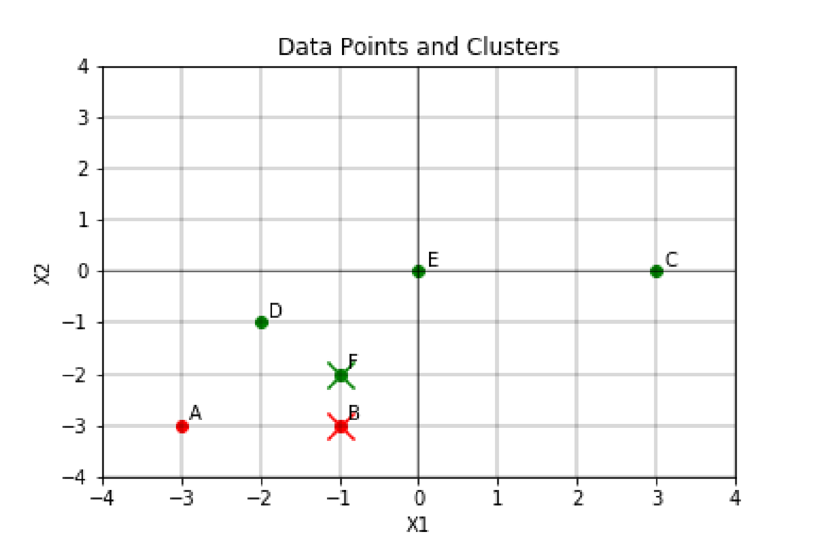


	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	n/a	n/a	n/a
B	(-1, -3)	n/a	n/a	n/a
C	(3, 0)	n/a	n/a	n/a
D	(-2, -1)	n/a	n/a	n/a
E	(0, 0)	n/a	n/a	n/a
F	(-1, -2)	n/a	n/a	n/a

Let's say we wish to have 2 cluster centers. We then randomly initialize those cluster centers by selecting two data points. Let's say we select B and F. We identify our cluster centers with a red and green 'X' respectively:

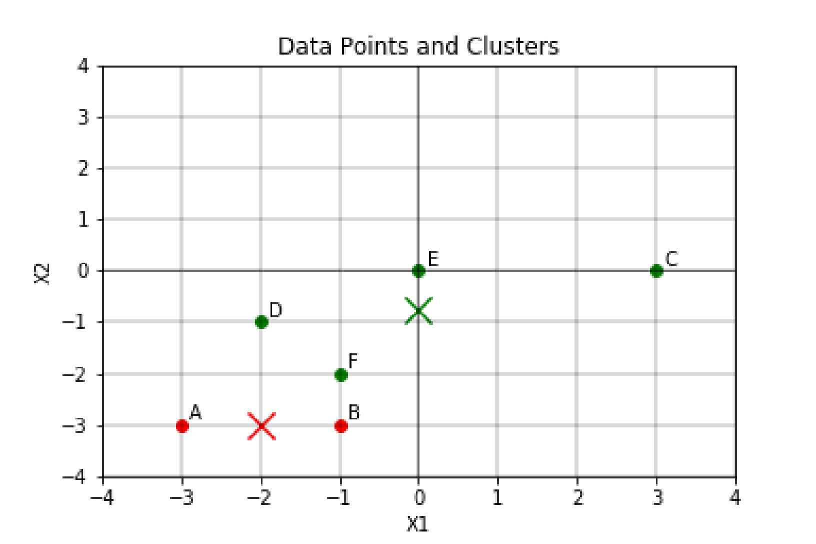


We now begin Lloyd's algorithm by assigning each data point to its closest cluster center:



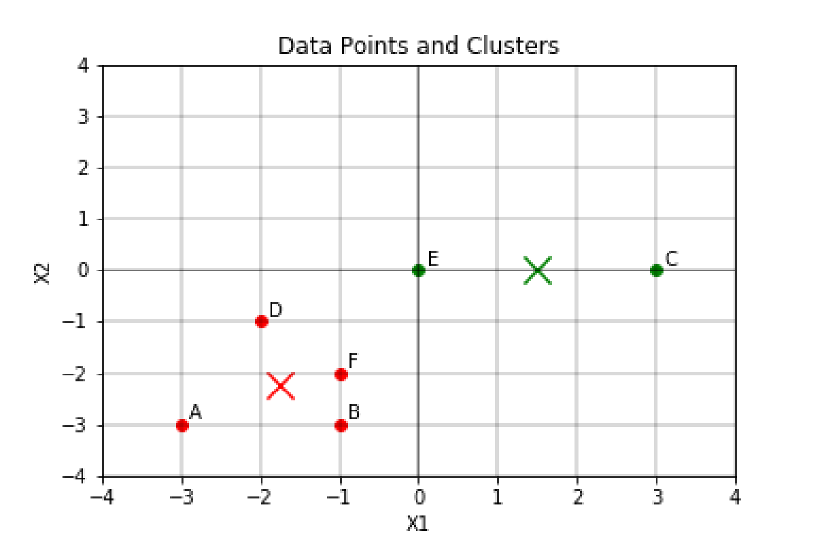
	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	n/a	n/a	Red
B	(-1, -3)	n/a	n/a	Red
C	(3, 0)	n/a	n/a	Green
D	(-2, -1)	n/a	n/a	Green
E	(0, 0)	n/a	n/a	Green
F	(-1, -2)	n/a	n/a	Green

We then update our cluster centers by averaging the data points assigned to each:



	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	2.00	2.24	Red
B	(-1, -3)	0.00	1.00	Red
C	(3, 0)	5.00	4.47	Green
D	(-2, -1)	2.24	1.41	Green
E	(0, 0)	3.16	2.24	Green
F	(-1, -2)	1.00	0.00	Green

We proceed like this, updating our cluster centers and assignments, until convergence. At convergence, we've achieved these cluster centers and assignments:



	Coordinates	Dist. to Red	Dist. to Green	Cluster Assgn.
A	(-3, -3)	1.46	5.41	Red
B	(-1, -3)	1.06	3.91	Red
C	(3, 0)	5.26	1.50	Green
D	(-2, -1)	1.27	3.64	Red
E	(0, 0)	2.85	1.50	Green
F	(-1, -2)	0.79	3.20	Red

Where our red cluster is at (-1.75, -2.25) and our green cluster is at (1.5 , 0). Note that for this random initialization of cluster centers, we deterministically identified the locally optimal set of assignments and cluster centers. For a specific initialization, running Lloyd's algorithm will always identify the same set of assignments and cluster centers. However, different initializations will produce different results

6.2.3 Number of Clusters

You may have wondered about a crucial, omitted detail: how do we choose the proper number of clusters for our data set? There doesn't actually exist a 'correct' number of clusters. The fewer clusters we have, the larger our loss will be, and as we add more clusters, our loss will get strictly smaller. That being said, there is certainly a tradeoff to be made here.

Having a single cluster is obviously useless - we will group our entire data set into the same cluster. Having N clusters is equally useless - each data point gets its own cluster.

One popular approach to identifying a good number of clusters is to perform K-Means with a varying number of clusters, and then to plot the number of clusters against the loss. Typically, that graph will look like Figure 6.1.

Notice that at $x = ...$ clusters, there appears to be a slight bend in the decrease of our loss. This is often called the **knee**, and it is common to choose the number of clusters to be where the knee occurs.

Intuitively, the idea here is that up to a certain point, adding another cluster significantly decreases the loss by more properly grouping our data points. However, eventually the benefit of adding another cluster stops being quite so significant. At this point, we have identified a natural number of groups for our data set.

6.2.4 Initialization and K-Means++

Up until now, we have assumed that we should randomly initialize our cluster centers and execute Lloyd's algorithm until convergence. We also suggested that since Lloyd's algorithm only produces a local minimum, it makes sense to perform several random initializations before settling on the most optimal assignment we've identified.

While this is a viable way to perform K-Means, there are other ways of initializing our original cluster centers that can help us find more optimal results without needing so many random initializations. One of those techniques is known as **K-Means++**.

The idea behind K-Means++ is that our cluster centers will typically be spread out when we've reached convergence. As a result, it might not make sense to initialize those cluster centers in an entirely random manner. For example, Figure 6.2 would be a poor initialization.

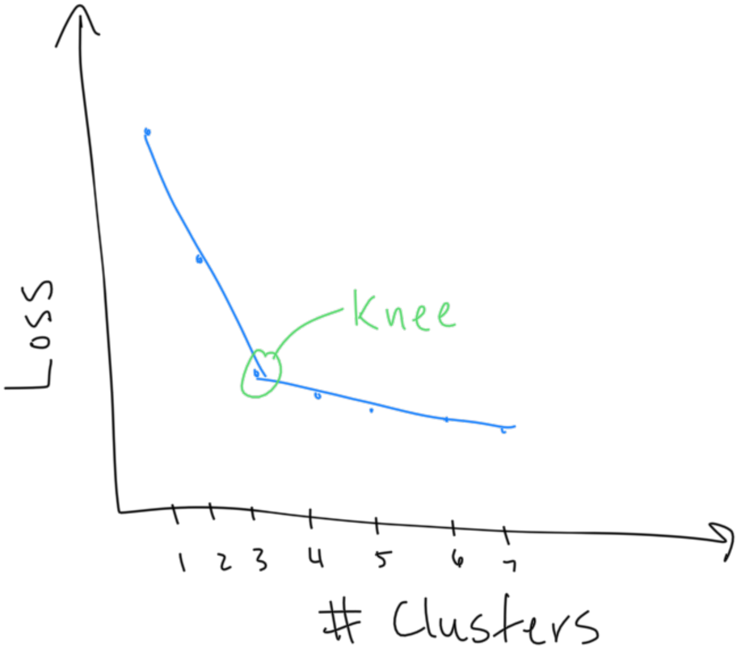


Figure 6.1: Finding the Knee.

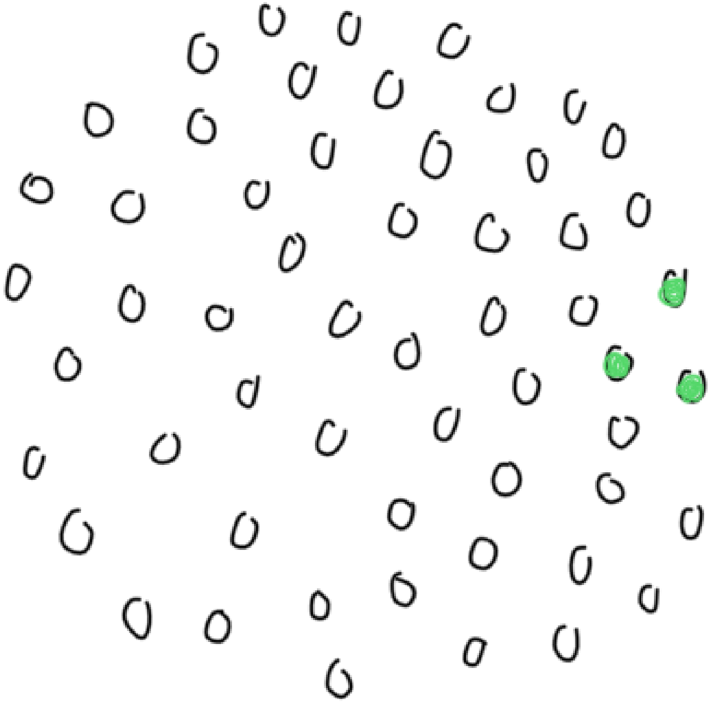


Figure 6.2: Bad Cluster Initialization.

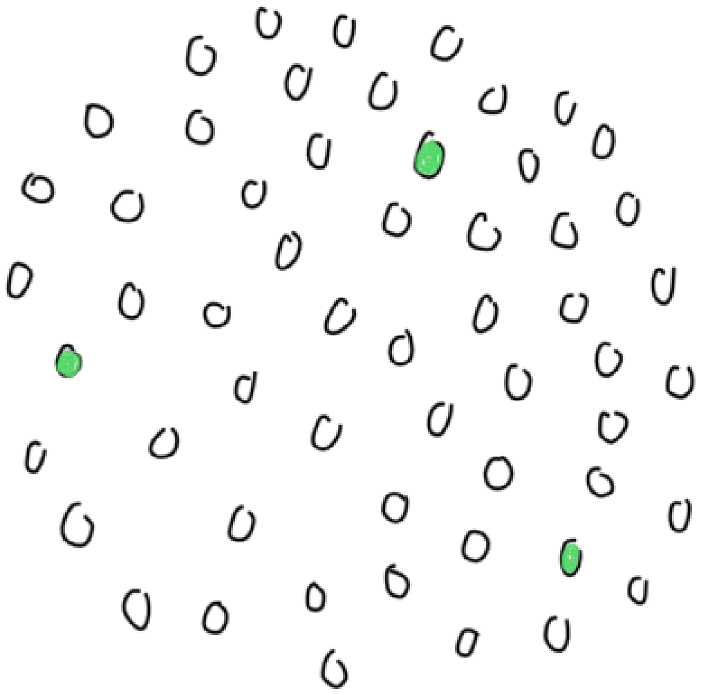


Figure 6.3: Good Cluster Initialization.

We would much rather start with a random initialization that looks like Figure 6.3.

We can use the hint that we want our cluster centers somewhat spread out to find a better random initialization. This is where the initialization algorithm presented by K-Means++ comes in.

For K-Means++, we choose the first cluster center by randomly selecting a point in our data set, same as before. However, for all subsequent cluster center initializations, we select points in our data set with probability proportional to the squared distance from their nearest cluster center. The effect of this is that we end up with a set of initializations that are relatively far from one another, as in Figure 6.3.

6.2.5 K-Medoids Alternative

Recall that in the cluster center update step (Equation 6.4) presented in the derivation of Lloyd's algorithm, we average the data points assigned to each cluster to compute the new cluster centers. Note that in some cases, this averaging step doesn't actually make sense (for example, if we have categorical variables as part of our feature set). In these cases, we can use an alternative algorithm known as **K-Medoids**. The idea behind K-Medoids is simple: instead of averaging the data points assigned to that cluster, update the new cluster center to be the data point assigned to that cluster which is most like the others.

6.3 Hierarchical Agglomerative Clustering

The motivating idea behind K-Means was that we could use a distance measurement to assign data points to a fixed number of clusters, iteratively improving our assignments and cluster locations

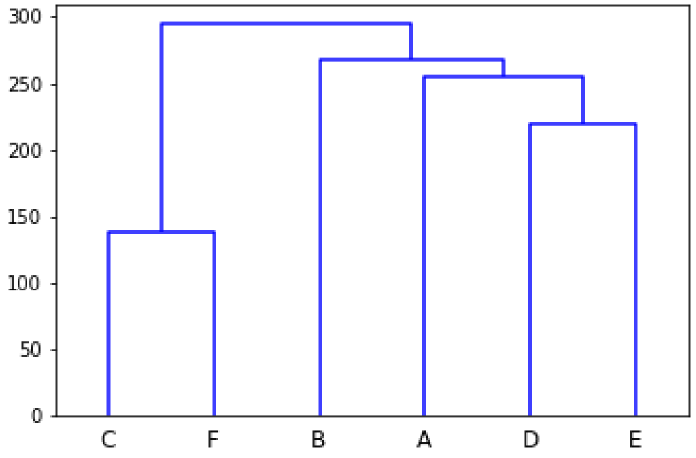


Figure 6.4: Dendrogram Example.

until convergence.

Moving on to Hierarchical Agglomerative Clustering (also known as **HAC** - pronounced ‘hack’), the motivating idea is instead to group data from the bottom up. This means every data point starts as its own cluster, and then we merge clusters together based on a distance metric that we define. This iterative merging allows us to construct a tree over our data set that describes relationships between our data. These trees are known as *dendrograms*, with an example found in Figure 6.4. Notice that the individual data points are the leaves of our tree, and the trunk is the cluster that contains the entirety of our data set.

We now formally define the HAC algorithm, and in the process, explain how we construct such a tree.

6.3.1 HAC Algorithm

1. Start with N clusters, one for each data point.

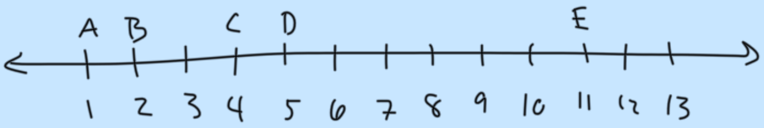
2. Measure the distance between clusters. This will require an inter-cluster distance measurement that we will define shortly.

3. Merge the two ‘closest’ clusters together, reducing the number of clusters by 1. Record the distance between these two merged clusters.

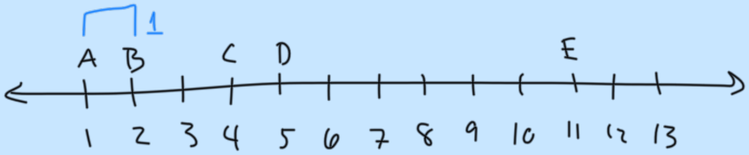
4. Repeat step 2 until we’re left with only a single cluster.

In the remainder of the chapter, we’ll describe this procedure in greater detail (including how to measure the distance between clusters), explain the clustering information produced by the tree, and discuss how HAC differs from K-Means. But first, to make this algorithm a little more clear, let’s perform HAC one step at a time on a toy data set, constructing the dendrogram as we go.

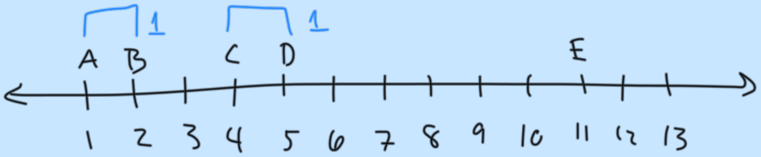
Example 6.2 (HAC Algorithm Example): Let’s say we have a data set of five points A, B, C, D, E that we wish to perform HAC on. These points will simply be scalar data that we can represent on a number line. We start with 5 clusters and no connections at all:



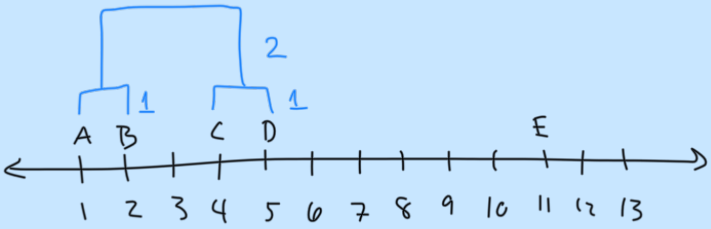
We find the closest two clusters to merge first. A and B are nearest (it's actually tied with C and D, but we can arbitrarily break these ties), so we start by merging them. Notice that we also annotate the distance between them in the tree, which in this case is 1:



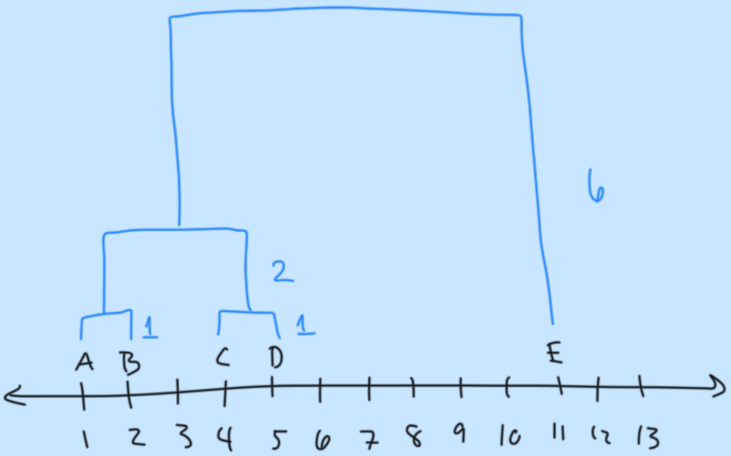
We now have four clusters: (A, B), C, D, and E. We again find the closest two clusters, which in this case is C and D:



We now have three remaining clusters: (A, B), (C, D), and E. We proceed as before, identifying the two closest clusters to be (A, B) and (C, D). Merging them:



Finally we are left with two clusters: (A, B, C, D) and E. The remaining two clusters are obviously the closest together, so we merge them:



At this point there is only a single cluster. We have constructed our tree and are finished with HAC.

Notice how the distance between two merged clusters manifests itself through the height of the dendrogram where they merge (which is why we tracked those distances as we constructed the tree). Notice also that we now have many layers of clustering: if we're only only interested in clusters whose elements are at least k units away from each other, we can 'cut' the dendrogram at that height and examine all the clusters that exist below that cut point.

Finally, we need to handle the important detail of how to compute the distance between clusters. In the preceding example, we designated the distance between two clusters to be the minimum distance between any two data points in the clusters. This is what is known as the **Min-Linkage Criterion**. However, there are certainly other ways we could have computed the distance between clusters, and using a different distance measurement can produce different clustering results. We now turn to these different methods and the properties of clusters they produce.

6.3.2 Linkage Criterion

Here are a few of the most common linkage criteria.

Min-Linkage Criteria

We've already seen the Min-Linkage Criterion in action from the previous example. Formally, the criterion says that the distance $d_{C,C'}$ between each cluster pair C and C' is given by

$$d_{C,C'} = \min_{k,k'} ||\mathbf{x}_k - \mathbf{x}_{k'}|| \tag{6.5}$$

where $\mathbf{x}_k$ are data points in cluster C and $\mathbf{x}_{k'}$ are data points in cluster C'. After computing these pairwise distances, we choose to merge the two clusters that are closest together.

Max-Linkage Criterion

We could also imagine defining the distance $d_{C,C'}$ between two clusters as being the distance between the two points that are farthest apart in each cluster. This is known as the Max-Linkage Criterion.

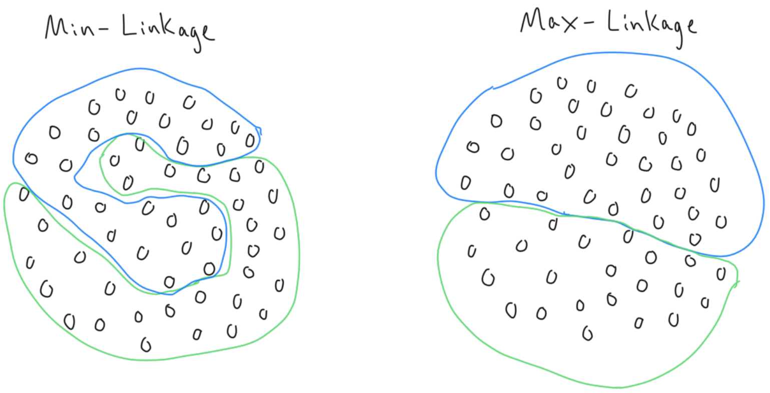


Figure 6.5: Different Linkage Criteria.

The distance between two clusters is then given by:

$$d_{C,C'} = \max_{k,k'} ||\mathbf{x}_k - \mathbf{x}_{k'}|| \tag{6.6}$$

As with the Min-Linkage Criterion, after computing these pairwise distances, we choose to merge the two clusters that are closest together.

⋆ Be careful not to confuse the linkage criterion with which clusters we choose to merge. We **always** merge the clusters that have the smallest distance between them. How we compute that distance is given by the linkage criterion.

Average-Linkage Criterion

The Average-Linkage Criterion averages the pairwise distance between each point in each cluster. Formally, this is given by:

$$d_{C,C'} = \frac{1}{KK'} \sum_{k=1}^{K} \sum_{k'=1}^{K'} ||\mathbf{x}_k - \mathbf{x}_{k'}|| \tag{6.7}$$

Centroid-Linkage Criterion

The Centroid-Linkage Criterion uses the distance between the centroid of each cluster (which is the average of the data points in a cluster). Formally, this is given by:

$$d_{C,C'} = ||\frac{1}{K} \sum_{k=1}^{K} \mathbf{x}_k - \frac{1}{K'} \sum_{k'=1}^{K'} \mathbf{x}_{k'}|| \tag{6.8}$$

Different Linkage Criteria Produce Different Clusterings

It's important to note that the linkage criterion you choose to use will influence your final clustering results. For example, the min-linkage criterion tends to produce 'stringy' clusters, while the max-linkage criterion tends to produce more compact clusters. You can see the difference between the results of these two linkage criteria in Figure 6.5.

⋆ You should convince yourself of the different flavors of linkage criteria. For example, when using the min-linkage criterion, we get these 'stringy' results because we're most inclined to extend existing clusters by grabbing whichever data points are closest.

6.3.3 How HAC Differs from K-Means

Now that we're aware of two distinct clustering techniques and their variants, we consider the differences between the two methods.

First of all, there is a fundamental difference in determinism between HAC and K-Means. In general, K-Means incurs a certain amount of randomness and needs to be run multiple times to ensure a good result. On the other hand, once you've selected a linkage criterion for HAC, the clusters you calculate are deterministic. You only need to run HAC a single time.

Another difference between HAC and K-Means comes from the assumptions we make. For K-Means, we need to specify the number of clusters up front before running our algorithm, potentially using something like the knee-method to decide on the number of clusters. On the other hand, you don't need to assume anything to run HAC, which simplifies its usage. However, the downside for HAC is that when you wish to present your final clustering results, you need to decide on the max distance between elements in each cluster (so that you can cut the dendrogram).

The fact that you need to make a decision about where to cut the dendrogram means that running HAC once gives you several different clustering options. Furthermore, the dendrogram in and of itself can be a useful tool for visualizing data. We don't get the same interactivity from K-Means clustering.

⋆ We often use dendrograms to visualize evolutionary lineage.

Chapter 7

Dimensionality Reduction

In previous chapters covering supervised learning techniques, we often used basis functions to project our data into higher dimensions prior to applying an inference technique. This allowed us to construct more expressive models, which ultimately produced better results. While it may seem counterintuitive, in this chapter we're going to focus on doing exactly the opposite: reducing the dimensionality of our data through a technique known as Principal Component Analysis (PCA). We will also explore why it is useful to reduce the dimensionality of some data sets.

7.1 Motivation

Real-world data is often very high dimensional, and it's common that our data sets contain information we are unfamiliar with because the dimensionality is too large for us to comb through all the features the by hand.

In these situations, it can be very difficult to manipulate or utilize our data effectively. We don't have a sense for which features are 'important' and which ones are just noise. Fitting a model to the data may be computationally expensive, and even if we were to fit some sort of model to our data, it may be difficult to interpret why we obtain specific results. It's also hard to gain intuition about our data through visualization since humans struggle to think in more than three dimensions. All of these are good reasons that we may wish to reduce the dimensionality of a data set.

ML Framework Cube: Dimensionality Reduction

Dimensionality reduction operates primarily on continuous feature spaces, is fully unsupervised, and is non-probabilistic for the techniques we explore in this chapter.

Domain	***Training***	***Probabilistic***
Continuous	Unsupervised	No

While dimensionality reduction is considered an unsupervised technique, it might also be thought of as a tool used to make data more manageable prior to taking some other action. In fact, it's an important preprocessing step for a host of use cases.

Bear Heights and Weights

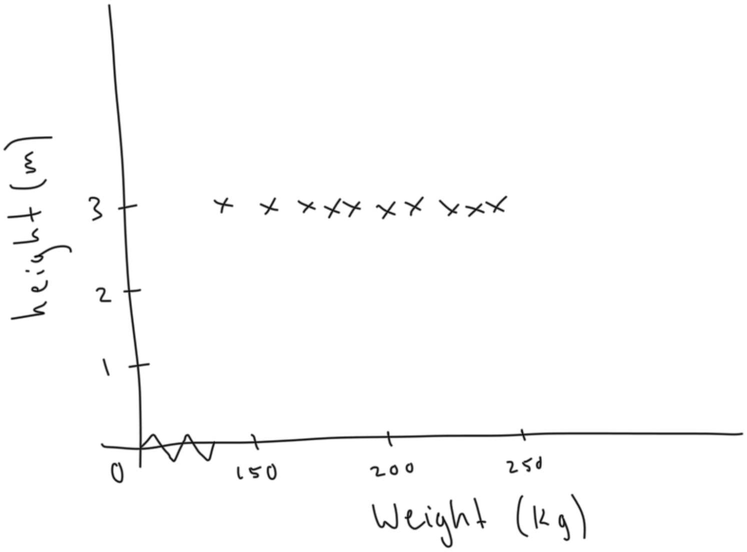


Figure 7.1: Graph of bear heights and weights.

7.2 Applications

As described above, we need a tool like dimensionality reduction in situations where high-dimensional data hinders us. Here are a few specific situations where we would use such a technique:

1. Presenting differences between complex molecules in two dimensions (via a graph).

2. Understanding the results of a credit trustworthiness algorithm.

3. Efficiently training a neural network to predict supermarket sales on a data set with many input features.

4. Identifying which costly measurements are worth collecting when experimenting with new chemicals.

With a few of these use cases in mind, we now turn to the math that underpins the dimensionality reduction technique known as Priniciple Component Analysis.

7.3 Principal Component Analysis

The main idea behind Principal Component Analysis (PCA) is that we can linearly project our data set onto a subspace without losing too much information. For example, three-dimensional data might primarily exist in a subspace that is actually a two-dimensional plane.

One way to think about this is to identify and preserve the features along which there is the most variance. For example, imagine we had a data set comprised of the heights and weights of individual bears. As an extreme case, let's suppose all the bears were exactly the same height but had a wide range of weights.

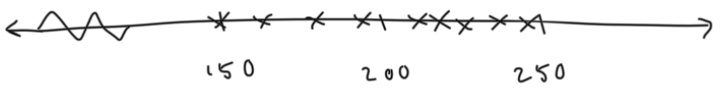

Bear Weights

Figure 7.2: Bear weights.

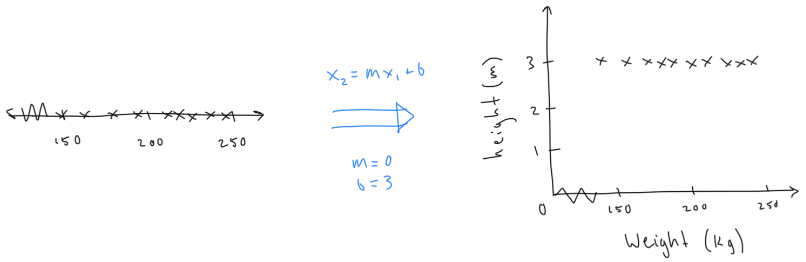

Converting Between Reduced and Original Data

Figure 7.3: Converting between the reduced data and original data.

To differentiate our data points, we obviously only need to report the weights of the bears. The variance of the heights is 0, and the variance of the weights is some non-zero number. Intuitively, the most interesting features from our data sets are those that vary the most.

⋆ In this simple example, the direction of maximal variance occurs exactly along the x_1 axis, but in general it will occur on a plane described by a combination of our input features.

The second way to think about PCA is that we are minimizing the error we incur when we move from the lower-dimensional representation back to the original representation. This is known as *reconstruction loss*. We can consider the meaning of this using our bear example.

Let's say we project the data set from Figure 7.1 here down to a single dimension by recording only the weights:

Then, to reconstruct our original graph, we need only to keep track of a slope and bias term in the form of the familiar equation $x_2 = mx_1 + b$. In this case our slope is $m = 0$ and our bias $b = 3$. Note that this storage overhead is constant (just remembering the slope and bias) regardless of how big our data set gets. Thus we can go from our low-dimensional representation back to our original data:

It will be our goal to determine a low-dimensional representation of our data that allows us to return to our high-dimensional data while losing as little information as possible. We wish to preserve everything salient about the data while discarding as much redundant information as possible. We now turn to how this can be achieved.

7.3.1 Reconstruction Loss

We identified a key tenet of dimensionality reduction in the previous section: finding subspaces of our data that preserve as much information as possible. Concretely, this means we want to convert our original data point $\mathbf{x}_n$ in D dimensions into a data point $\mathbf{x}'_n$ in D' dimensions where $D' < D$.

Data Reconstruction

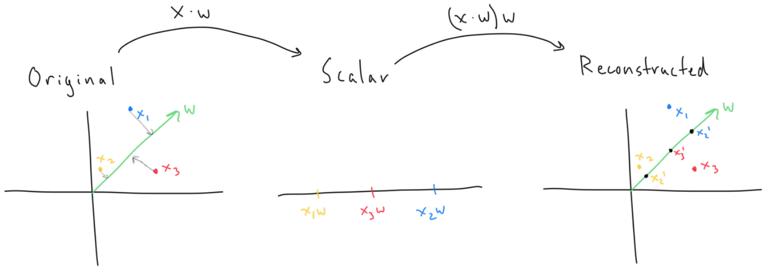


Figure 7.4: Far left: our original data. Middle: our reduced data in scalar form after the projection given by $\mathbf{x}{\cdot}\mathbf{w}$. Right: our reconstructed data points given by $(\mathbf{x}{\cdot}\mathbf{w})\mathbf{w}$. Notice that our reconstructed data points are not the same as our original data.

⋆ We're going to assume that our data set has been mean-centered such that each feature in $\mathbf{x}_n$ has mean 0. This will not affect the application of the method (we can always convert back to the uncentered data by adding back the mean of each feature), but will make our derivations more convenient to work with.

Let's consider a simple case first: $D' = 1$. This means that we're projecting our D dimensional data down onto just a single dimension, or in geometric terms, we're projecting our data points $\mathbf{x}_n$ onto a line through the origin. We can define this line as the unit vector $\mathbf{w} \in \mathbb{R}^{D\times 1}$, and the projection is given by the dot product $\mathbf{x} \cdot \mathbf{w}$.

⋆ The unit vector $\mathbf{w}$ onto which we project our data is known as a *principal component*, from which PCA gets its name.

This projection produces a scalar, and that scalar defines how far our projection $\mathbf{x} \cdot \mathbf{w}$ is from the origin. We can convert this scalar back to D dimensional space by multiplying it with the unit vector $\mathbf{w}$. This means that $(\mathbf{x} \cdot \mathbf{w})\mathbf{w}$ is the result of projecting our data point $\mathbf{x}$ down into one-dimension and then converting it to its coordinate location in D dimensions. We refer to these as our *projection vectors*, and we can observe what this looks like geometrically in Figure 7.4.

The projection vectors we recover from the expression $(\mathbf{x} \cdot \mathbf{w})\mathbf{w}$ will be in D dimensions, but they will obviously not be identical to the original D dimensional vectors (Figure 7.4 demonstrates why that is the case). This difference between the original and projection vectors can be thought of as error, since it is information lost from our original data. For a given data point $\mathbf{x}_n$ and unit vector $\mathbf{w}$, we can measure this error through the expression:

$$||\mathbf{x}_n - (\mathbf{x} \cdot \mathbf{w})\mathbf{w}||^2 \tag{7.1}$$

which is known as **reconstruction loss** because it measures the error incurred when reconstructing our original data from its projection.

Definition 7.3.1 (Reconstruction Loss): Reconstruction loss is the difference (measured via a distance metric such as Euclidean distance) between an original data set and its reconstruction from a lower dimensional representation. It indicates how much information is lost during dimensionality reduction.

Reconstruction loss is then a metric for evaluating how 'good' a subspace in D' dimensions is at representing our original data in D dimensions. The better it is, the less information we lose, and the reconstruction loss is lower as a result.

7.3.2 Minimizing Reconstruction Loss

We now know that our goal is to find a good subspace to project onto, and we also know that finding this good subspace is equivalent to minimizing the reconstruction loss it incurs. We can now formalize this as an optimization problem.

First, we simplify the reconstruction loss for a single data point $\mathbf{x}_n$ as follows:

$$\begin{aligned} ||\mathbf{x}_n - (\mathbf{x}_n \cdot \mathbf{w})\mathbf{w}||^2 &= (\mathbf{x}_n - (\mathbf{x}_n \cdot \mathbf{w})\mathbf{w})(\mathbf{x}_n - (\mathbf{x}_n \cdot \mathbf{w})\mathbf{w}) \\ &= ||\mathbf{x}_n||^2 - 2(\mathbf{x}_n \cdot \mathbf{w})^2 + (\mathbf{x}_n \cdot \mathbf{w})^2||\mathbf{w}||^2 \\ &= ||\mathbf{x}_n||^2 - (\mathbf{x}_n \cdot \mathbf{w})^2 \end{aligned}$$

where $||\mathbf{w}||^2 = 1$ because it is a unit vector. Note that we can define reconstruction loss over our entire data set as follows:

$$RL(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} ||\mathbf{x}_n||^2 - (\mathbf{x}_n \cdot \mathbf{w})^2 \tag{7.2}$$

Recall that our goal is to minimize reconstruction loss over our data set by optimizing the subspace defined by $\mathbf{w}$. Let's first rewrite Equation 7.2 as:

$$RL(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} ||\mathbf{x}_n||^2 - \frac{1}{M} \sum_{n=1}^{N} (\mathbf{x}_n \cdot \mathbf{w})^2$$

where we can see that our optimization will depend only on maximizing the second term:

$$\max_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^{N} (\mathbf{x}_n \cdot \mathbf{w})^2 \tag{7.3}$$

since it is the only one involving $\mathbf{w}$. Recall that the sample mean of a data set is given by the expression $\frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n$, and note that Equation 7.3 is the sample mean of $(\mathbf{x} \cdot \mathbf{w})^2$. Using the definition of variance for a random variable $\mathbf{Z}$ (which is given by $Var(\mathbf{Z}) = \mathbb{E}(\mathbf{Z}^2) - (\mathbb{E}(\mathbf{Z}))^2$), we can rewrite Equation 7.3 as:

$$\frac{1}{N} \sum_{n=1}^{N} (\mathbf{x}_n \cdot \mathbf{w})^2 = Var\big[\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^{N}\big] + \big(\mathbb{E}\big[\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^{N}\big]\big)^2$$

Recall that we centered our data $\mathbf{x}_n$ to have mean 0 such that the expression above simplifies to:

$$\frac{1}{N} \sum_{n=1}^{N} (\mathbf{x}_n \cdot \mathbf{w})^2 = Var\big[\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^{N}\big] \tag{7.4}$$

and therefore $Var\big[\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^{N}\big]$ is the term we wish to maximize. **This means that minimizing the reconstruction loss is equivalent to maximizing the variance of our projections $\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^{N}$.**

⋆ Note the intuitiveness of this result. We should like to find a subspace that maintains the spread in our data.

7.3.3 Multiple Principal Components

Up until now, we've been considering how we would project onto a single principal component $\mathbf{w} \in \mathbb{R}^{D\times 1}$. This will reduce our data down to one dimension, just a scalar. In general, we will wish to preserve more of our data than just a single dimension (in order to capture more of the variance in our data and reduce reconstruction loss), which means that we will need to have multiple principal components. For now we'll assume that our principal components are orthogonal (we will prove this later), which then allows us to describe the projection of our data $\mathbf{x}_n$ onto this subspace as the sum of the projections onto D' orthogonal vectors:

$$\sum_{d'=1}^{D'} (\mathbf{x}_n \cdot \mathbf{w}_{d'})\mathbf{w}_{d'} \tag{7.5}$$

7.3.4 Identifying Directions of Maximal Variance in our Data

We now know from the previous section that we find our principal components (and thus the subspace we will project onto) by identifying the directions of maximum variance in our projected data set. We know from Equation 7.4 that the variance is equivalent to:

$$\sigma^2_{\mathbf{w}} \equiv Var\big[\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^N\big] = \frac{1}{N}\sum_{n=1}^N (\mathbf{x}_n \cdot \mathbf{w})^2$$

Rewriting this in terms of matrix notation we have that:

$$\sigma^2_{\mathbf{w}} = \frac{1}{N}(\mathbf{X}\mathbf{w})^T(\mathbf{X}\mathbf{w})$$

We can further simplify this:

$$\begin{aligned}
\sigma^2_{\mathbf{w}} &= \frac{1}{N}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} \\
\sigma^2_{\mathbf{w}} &= \mathbf{w}^T\frac{\mathbf{X}^T\mathbf{X}}{N}\mathbf{w} \\
\sigma^2_{\mathbf{w}} &= \mathbf{w}^T\mathbf{S}\mathbf{w}
\end{aligned}$$

where, since we assume the design matrix $\mathbf{X}$ is **mean-centered**, then $\mathbf{S} = \frac{\mathbf{X}^T\mathbf{X}}{N}$ is the empirical covariance matrix of our data set.

⋆ Notice that by convention we describe the empirical covariance of a data set with the term **S** instead of the usual covariance term Σ.

Our goal is to maximize the term $\sigma^2_{\mathbf{w}} = Var\big[\{\mathbf{x}_n \cdot \mathbf{w}\}_{n=1}^N\big]$ with respect to $\mathbf{w}$. Furthermore, $\mathbf{w}$ is a unit vector, so we must optimize subject to the constraint $\mathbf{w}^T\mathbf{w} = 1$. Recalling the discussion of

Lagrange multipliers from Chapter 6 on Support Vector Machines, we incorporate this constraint by reformulating our optimization problem as the Lagrangian equation:

$$\mathcal{L}(\mathbf{w}, \lambda) = \mathbf{w}^T\mathbf{S}\mathbf{w} - \lambda(\mathbf{w}^T\mathbf{w} - 1)$$

As usual, we proceed by taking the derivative with respect to each parameter:

$$\frac{d\mathcal{L}(\mathbf{w}, \lambda)}{d\mathbf{w}} = 2\mathbf{S}\mathbf{w} - 2\lambda\mathbf{w}$$
$$\frac{d\mathcal{L}(\mathbf{w}, \lambda)}{d\lambda} = \mathbf{w}^T\mathbf{w} - 1$$

We can now set these equal to 0 and solve for the optimal values:

$$\mathbf{S}\mathbf{w} = \lambda\mathbf{w}$$
$$\mathbf{w}^T\mathbf{w} = 1$$

This result is very significant! As we knew already, we needed $\mathbf{w}$ to be a unit vector. However, we also see that $\mathbf{w}$ is an eigenvector of the empirical covariance matrix $\mathbf{w}$. Futhermore, the eigenvector that will maximize our quantity of interest $\sigma^2_{\mathbf{w}} = \mathbf{w}^T\mathbf{S}\mathbf{w}$ will be the eigenvector with the largest eigenvalue λ.

Linear algebra gives us many tools for finding eigenvectors, and as a result we can efficiently identify our principal components. Note also that the eigenvectors of a symmetric matrix are orthogonal, which proves our earlier assumption that our principal components are orthogonal.

⋆ Each eigenvector is a principal component. The larger the eigenvalue associated with that principal component, the more variance there is along that principal component.

To recap, we've learned that the optimal principal components (meaning the vectors describing our projection subspace) are the eigenvectors of the empirical covariance matrix of our data set. The vector preserving the most variance in our data (and thus minimizing the reconstruction loss) is given by the eigenvector with the largest eigenvalue, followed by the eigenvector with the next largest eigenvalue, and so on. Furthermore, while it is somewhat outside the scope of this textbook, we are guaranteed to have D distinct, orthogonal eigenvectors with eigenvalues ≥ 0. This is a result of linear algebra that hinges on the fact that our empirical covariance matrix $\mathbf{S}$ is symmetric and positive semi-definite.

7.3.5 Choosing the Optimal Number of Principal Components

We now know that the eigenvectors of the empirical covariance matrix $\mathbf{S}$ give us the principal components that form our projection subspace. Note that the exact procedure for finding these eigenvectors is a topic better suited for a book on linear algebra, but one approach is to use *singular value decomposition* (SVD).

The *eigenvalue decomposition* of a square matrix is $\mathbf{S} = \mathbf{V}\mathbf{L}\mathbf{V}^\top$, where the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{S}$, and $\mathbf{L}$ is a diagonal matrix whose entries are the corresponding eigenvalues $\{\lambda_i\}_i$. The SVD of a real-valued, square or rectangular matrix $\mathbf{X}$ is $\mathbf{X} = \mathbf{U}\mathbf{Z}\mathbf{V}^\top$ with diagonal

Four Dimensional Iris Data Set in Three Dimensions

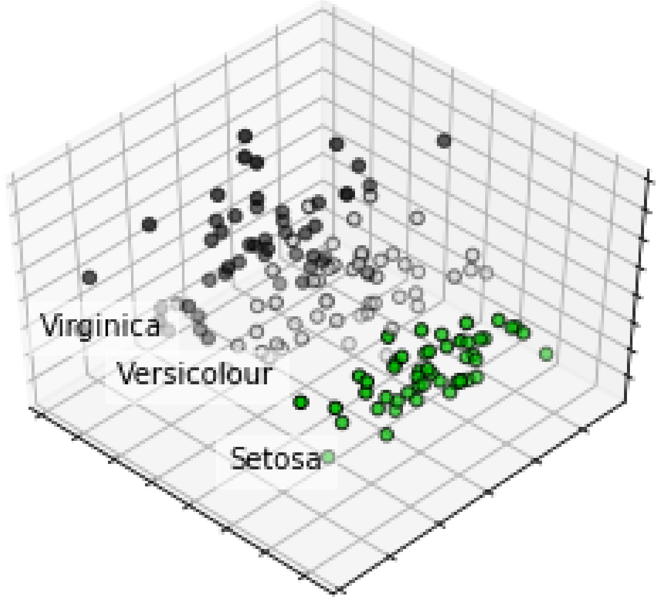


Figure 7.5: PCA applied to Fisher's Iris data set, which is originally in four dimensions. We reduce it to three dimensions for visualization purposes and label the different flower types. This example is taken from the sklearn documentation.

matrix $\mathbf{Z}$, and orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ (i.e., $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$). Matrix $\mathbf{V}$ is the *right singular matrix*, and its columns the *right singular vectors*. The entries in $\mathbf{Z}$ are the corresponding *singular values*. The column vectors in $\mathbf{V}$ also the eigenvectors of $\mathbf{X}^\top\mathbf{X}$. To see this, we have $\mathbf{X}^\top\mathbf{X} = \mathbf{V}\mathbf{Z}\mathbf{U}^\top\mathbf{U}\mathbf{Z}\mathbf{V}^\top = \mathbf{V}\mathbf{Z}^2\mathbf{V}^\top$, where we substitute the SVD for $\mathbf{X}$. We also see that the eigenvalues $\lambda_i = z_i^2$, so that they are the squared singular values.

For a mean-centered design matrix $\mathbf{X}$, so that $(1/N)\mathbf{X}^\top\mathbf{X} = \text{cov}(\mathbf{X})$, we can compute the SVD on the design matrix $\mathbf{X}$, and then read off the eigenvectors of the covariance as the columns of the right singular matrix and the eigenvalues as $\lambda_i = z_i^2/N$. Alternatively, you can first divide the mean-centered design matrix by $\sqrt{N}$ before taking its SVD, in which case the square of the singular values corresponds to the eigenvalues.

Because the principal components are orthogonal, the projections they produce will be entirely uncorrelated. This means we can project our original data onto each component individually and then sum those projections to create our lower dimensional data points. Note that it doesn't make sense that we would use every one of our D principal components to define our projection subspace, since that wouldn't lead to a reduction in the dimensionality of our data at all (the D orthogonal principal components span the entire D dimensional space of our original data set). We now need to decide how many principal components we will choose to include, and therefore what subspace we will be projecting onto.

The 'right' number of principal components to use depends on our goals. For example, if we simply wish to visualize our data, then we would project onto a 2D or 3D space. Therefore, we would choose the first 2 or 3 principal components, and project our original data onto the subspace defined by those vectors. This might look something like Figure 7.5.

However, it's more complicated to choose the optimal number of principal components when our goal is not simply visualization. We're now left with the task of trading off how much dimensionality

Reconstruction Loss vs. Number of Principal Components

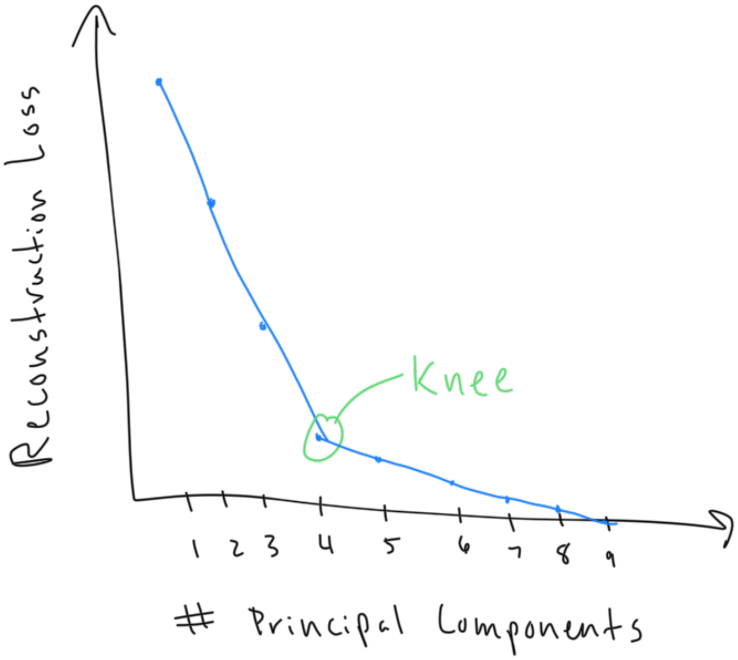


Figure 7.6: Reconstruction loss versus the number of principal components. Notice the similarity to the 'elbow' method of K-Means clustering.

reduction we wish to achieve with how much information we want to preserve in our data.

One way to do this is similar to the informal 'elbow' method described for K-Means clustering. We graph our reconstruction loss against the number of principal components used, as seen in Figure 7.6. The idea is to add principal components to our subspace one at a time, calculating the reconstruction loss as we go. The first few principal components will greatly reduce the reconstruction loss, before eventually leveling off. We can identify the 'elbow' where the reduction in loss starts to diminish, and choose to use that number of principal components.

Another way to do this is to consider how much variance we wish to preserve in our data. Each principal component is associated with an eigenvalue λ_d that indicates what proportion of the variance that principal component is responsible for in our data set. Then the fraction of variance retained from our data set if we choose to keep D' principal components is given by:

$$\text{retained variance} = \frac{\sum_{d'=1}^{D'} \lambda_{d'}}{\sum_{d=1}^{D} \lambda_d} \tag{7.6}$$

For different applications, there may be different levels of acceptable variance retention, which can help us decide how many principal components to keep.

Finally, once we have selected our principal components, we have also defined the subspace onto which we will be projecting our original data. And although this subspace is defined by the basis given by our principal components, these principal components are not a unique description of that subspace. We could choose to use any basis after we've identified our subspace through the principal components. The importance of this idea is simply that although our principal components are unique, they are not the only basis we could use to define the same projection subspace.

7.4 Conclusion

Principal component analysis is useful for visualization purposes, removing redundant information, or making a data set more computationally manageable. PCA is also a good tool for data exploration, particularly when digging into an unfamiliar data set for the first time.

It is not essential that we know by heart the exact derivation for arriving at the principal components of a data set. The same can be said of the linear algebra machinery needed to compute principal components. However, it is important to have an intuitve grasp over how variance in our data set relates to principal components, as well as an understanding of how subspaces in our data can provide compact representations of that data set. These are critical concepts for working effectively with real data, and they will motivate related techniques in machine learning.

Chapter 8

Graphical Models

Mathematics, statistics, physics, and other academic fields have useful notational systems. As a hybrid of these and other disciplines, machine learning borrows from many of the existing systems. Notational abstractions are important to enable consistent and efficient communication of ideas, for both teaching and knowledge creation purposes. Much of machine learning revolves around modeling data processes, and then performing inference over those models to generate useful insights. In this chapter, we will be introducing a notational system known as the directed graphical model (DGM) that will help us reason about a broad class of models.

8.1 Motivation

Up until this point, we've defined notation on the fly, relied on common statistical concepts, and used diagrams to convey meaning about the problem setup for different techniques. We've built up enough working knowledge and intuition at this point to switch to a more general abstraction for defining models: directed graphical models (DGMS). DGMs will allow us to both consolidate our notation and convey information about arbitrary problem formulations. An example of what a DGM looks like for a linear regression problem setup is given in Figure 8.1, and over the course of the chapter, we'll explain how to interpret the symbols in this diagram.

We need graphical models for a couple of reasons. First, and most importantly, a graphical model unambiguously conveys a problem setup. This is useful both to share models between people (communication) and to keep all the information in a specific model clear in your own head (consolidation). Once we understand the meaning of the symbols in a DGM, it will be far easier to

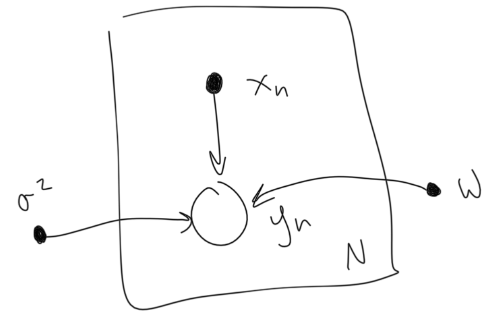


Figure 8.1: Linear regression model expressed with a DGM.

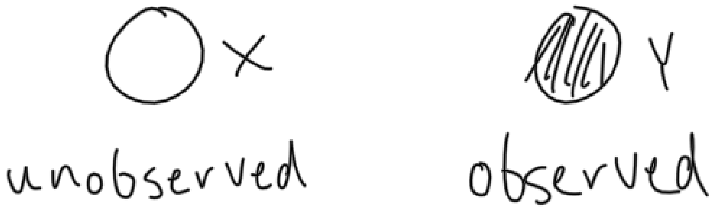


Figure 8.2: Random variables are denoted with an open circle, and it is shaded if the variable is observed.

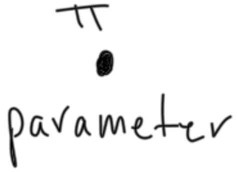


Figure 8.3: Deterministic parameters are denoted with a tight, small dot.

examine one of them than it will be to read several sentences describing the type of model we're imagining for a specific problem. Another reason we use DGMs is that they help us reason about independence properties between different parts of our model. For simple problem setups this may be easy to keep track of in our heads, but as we introduce more complicated models it will be useful to reason about independence properties simply by examining the DGM describing that model.

Ultimately, directed graphical models are a tool to boost efficiency and clarity. We'll examine the core components of a DGM, as well as some of their properties regarding independence and model complexity. The machinery we develop here will be used heavily in the coming chapters.

8.2 Directed Graphical Models (Bayesian Networks)

There are a few fundamental components to all of the models we've examined so far. In fact, we can model just about everything we've discussed to this point using just random variables, deterministic parameters, and arrows to indicate the relationships between them. Let's consider linear regression as a simple but comprehensive example. We have a random variable y_n, the object of predictive interest, which depend on deterministic parameters in the form of data $\mathbf{x}_n$ and weights $\mathbf{w}$. This results in the DGM given by Figure 8.1. There are four primary pieces of notation that the linear regression setup gives rise to, and these four components form the backbone of every DGM we would like to construct.

First, we have random variables represented by an open circle, shown in Figure 8.2. Note that if we observe a random variable of a given model, then we shade it in. Otherwise, it's left open.

Second, we have deterministic parameters represented by a tight, small dot, shown in Figure 8.3.

Third, we have arrows that indicate the dependence relationship between different random variables and parameters, shown in Figure 8.4. Note that an arrow from X into Y means that Y depends on X.

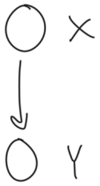


Figure 8.4: Arrows indicate dependence relationships.

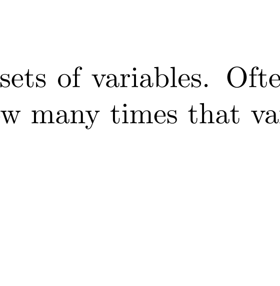


Figure 8.5: Plates indicate repeated sets of variables. Often there will be a number in one of the corners (N in this case) indicating how many times that variable is repeated.

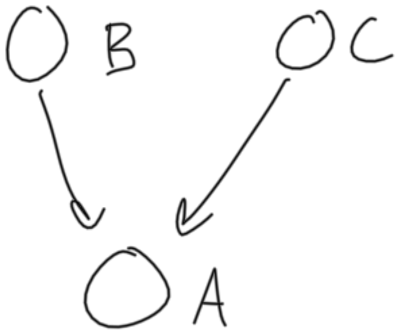


Figure 8.6: DGM for the joint distribution give by Equation 8.1.

And finally, we have plate notation to indicate that we have repeated sets of variables in our model setup, shown in Figure 8.5.

With these four simple constructs, we can describe complex model setups with a simple diagram. We can have an arbitrary number of component with any sort of dependence structure baked in. DGMs can be useful as a reference while working on a problem, and they also make it easy to iterate on an existing model setup.

8.2.1 Joint Probability Distributions

We'll now consider how DGMs simplify the task of reasoning about a joint probability distribution over multiple random variables. Note that for any joint probability distribution, regardless of our knowledge about the dependence structure in the model, it's always valid to write a generic joint probability distribution as follows:

$$p(A, B, C)$$

where in this setup, we are interested in the joint distribution between three random variables A, B, C. However, this doesn't tell us anything about the structure of the problem at hand: where there is independence and how we can use that to simplify our model. For example, if we knew that B and C were independent and we also knew the conditional distribution of $A|B, C$ then we would much rather setup our joint probability equation as:

$$p(A, B, C) = p(B)p(C)p(A|B, C) \tag{8.1}$$

DGMs assist in this process of identifying the appropriate factorization, as their structure allows us to read off valid factorizations directly. For example, the joint distribution given by Equation 8.1 can be read from Figure 8.6.

We can move between a DGM and a valid factorized joint distribution by identifying the arrows indicating dependencies. If a random variable has no dependencies (as neither B nor C do in this example), they can be written on their own as marginal probabilities $p(B)$ and $p(C)$. For random variables with arrows coming into them, indicating dependence, we include them in the joint factorization conditioned on the variables that they depend on, i.e. $P(A|B, C)$. In this way we can move back and forth between DGMs and factorized joint distributions with ease.

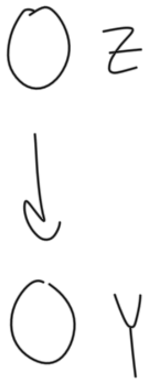


Figure 8.7: Example of a generative model.

8.2.2 Generative Models

While DGMs allow us to move quickly between a factorized joint distribution and a corresponding graphical model, they also show us the process by which data comes into existence (sometimes referred to as the data generating story or generative process). What this means is that if presented with a DGM, it is possible for us to identify how the data gets created, and if we have the proper tools, how we could generate new data ourselves.

Definition 8.2.1 (Generative Models): A generative model describes the entire process by which data comes into existence. While this is not always necessary if our goal is only to make predictions or perform some other kind of inference, it does have the added benefit of enabling the creation of new data by sampling from the generative model.

⋆ Note that we can create graphical models for both generative and discriminative models. Discriminative models will only model the conditional distribution $p(Y|Z)$, while generative models will model the full joint distribution $p(Z, Y)$.

Let's consider a simple example to see how this works in practice. Consider the flow of information present in Figure 8.7. First, there is some realization of the random variable Z. Then conditioned on that value of Z, there is some realization of the random variable Y. Obviously, the joint factorization for this DGM is given by $p(Z)p(Y|Z)$, but on a more intuitive level, the data is created by first sampling from Z's distribution, and then based on that value, sampling from the conditional distribution of Y.

To make this concrete, we could consider Z to be a random variable that determines a specific breed of dog, and Y to be the random variable that, conditioned on the breed of dog, determines the snout length of that type of dog. Notice that we have not specified anything about the specific distributional form from which Z and Y come, only the story of how they relate to each other.

This story also shows us that if we had some model for Z and $Y|Z$, we could generate data points ourselves. Our procedure would simply be to sample from Z and then to sample from Y conditioned on that value of Z. We could perform this process as many times as we like to generate new data. This is in contrast to sampling directly from the joint $p(Z, Y)$, which is difficult if we don't know the exact form of the joint distribution (which we often do not) or if the joint

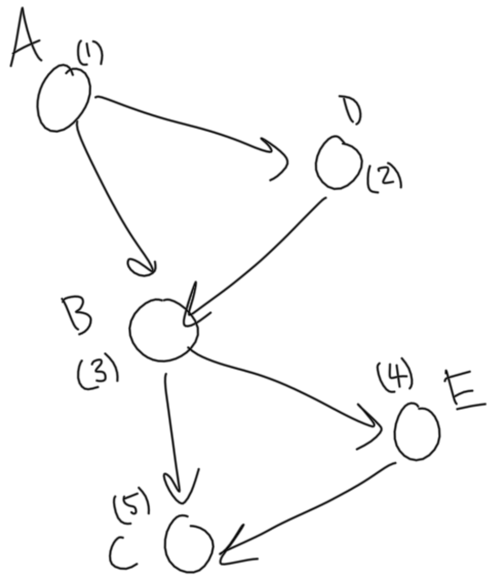


Figure 8.8: Notice we have to sample these in topological order: A, D, B, E, C.

distribution is hard to sample from directly.

The technique of sampling from distributions in the order indicated by their DGM is known as **ancestral sampling**, and it is a major benefit of generative models.

Definition 8.2.2 (Ancestral Sampling): Ancestral sampling is a technique used to generate data from an arbitrary DGM. It works by sampling from the random variables in topological order, meaning that we first sample from all random variables without any dependence, then from the random variables that depend on those intially sampled random variables, and so on until all the random variables have been sampled. This is demonstrated in Figure 8.8.

8.2.3 Generative Modeling vs. Discriminative Modeling

In the previous section, we described generative modeling. You may have wondered if there was any other type of modeling possible, if not the model that describes how data comes into existence. There is another type of model known as a discriminative model, and we have already seen several examples of them.

A discriminitive model skips the step of describing how data was generated (i.e. it doesn't model the joint $p(Z, Y)$), and instead it cuts right to our predictive objective (i.e. it models the conditional $p(Y|Z)$). For example, if we wish to predict what value a response variable will take on, we can consider the input data points to be parameters without an underlying distribution, and then our model is simply tasked with predicting the response variable based on those parameters. This is in contrast to a generative model, which would model how all the various data points came into existence by assigning a distribution to each of them.

Discriminative modeling is exactly what we did with linear and logistic regression, and it's commonly what we do with SVMs and neural networks as well. Let's consider the DGM describing linear regression again, from Figure 8.1. Notice that the data points $\mathbf{x}_n$ are not random variables

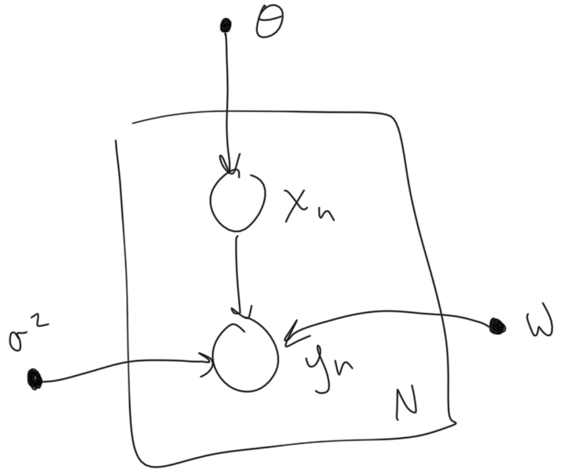


Figure 8.9: Linear regression DGM, modeling $\mathbf{x}_n$ as a random variable.

but merely parameters of the DGM. If we wanted a generative model, we would instead have a DGM that looks that of Figure 8.9.

The difference between these model setups is significant. The generative model for linear regression would allow us to generate new data points, but it is also significantly more complex to handle because now instead of having just a single response variable to predict, we also have to contend with how we model the generation of the data points $\mathbf{x}_n$. This may be difficult on both conceptual and computational levels. This doesn't mean we'll never try to do this, but if our goal is simply to make predictions about the response variable y_n, it may be overkill to use a generative model.

In essence, the distinction between generative and discriminative models comes down to whether or not the model tries to describe how the data is realized or if the model simply tries to perform a specific inference task without modeling the entirety of the data generating process. Neither one is better, they are just different techniques that will apply differently depending on your modeling and inferential needs.

8.2.4 Understanding Complexity

We've already motivated one of the primary uses of DGMs as being the ability to convert a joint distribution into a factorization. At the heart of that task was recognizing and exploiting independence properties in a model over multiple random variables. Another benefit of this process is that it allows us to easily reason about the size (also called 'complexity') of a joint factorization over discrete random variables. In other words, it allows us to determine how many parameters we will have to learn to describe the factorization for a given DGM.

Let's consider an example to make this concept clear. Let's say we have four categorical random variables A, B, C, D which take on 2, 4, 8, and 16 values respectively. If we were to assume full dependence between each of these random variables, then a joint distribution table over all of these random variables would require $(2 * 4 * 8 * 16) - 1$ total parameters (where each parameter corresponds to the probability of a specific permutation of the values A, B, C, D).

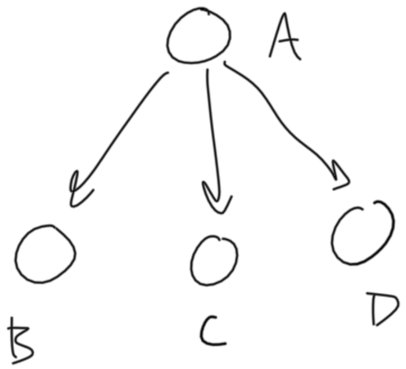


Figure 8.10: Conditioned on A, we have that B, C, and D are independent.

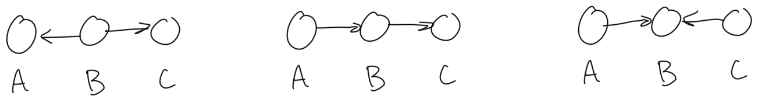


Figure 8.11: The three random variable relationships that will tell us about independence relationships.

⋆ Notice that the number of parameters we need is $(2*4*8*16)-1$ and not $(2*4*8*16)$. This is simply because if we know the first $(2*4*8*16)-1$ parameters, we can calculate the value of the final one exactly.

However, if we knew that some of these random variables were conditionally independent, then the number of parameters would change. For example, consider the joint distribution given by $p(A,B,C,D) = p(A)p(B|A)p(C|A)p(D|A)$. This would imply that conditioned on A, each of B,C,D were conditionally independent. This can also be shown by the DGM in Figure 8.10.

In this case, a table of parameters to describe this joint distribution would only require $2*((4-1)+(8-1)+(16-1))$ parameters, which is significantly less.

This leads to the natural conclusion that the more conditional independence we can identify in a given model, the easier it becomes from a modeling and computational perspective. This is another major benefit of DGMs: it's possible to visually reason about the independence properties of a given model. We turn to how this works next.

8.2.5 Independence and D-Separation

We can use the form of a graphical model directly to determine which variables are independent under specific observation assumptions. We have three base cases for the relationship between variables from which we can reason about the structure in any arbitrarily complex model. The three cases look like what's found in Figure 8.11. For each of these cases, there is a notion of information either flowing from one random variable to the other or being 'blocked' (implying independence) by an observation. Specifically, case 1 and case 2 have information between nodes A and C 'blocked' by observing node B, and case 3 has information between nodes A and C 'blocked' by *not* observing node B.

Let's consider each of these cases more carefully, and gain a better notion of what it means

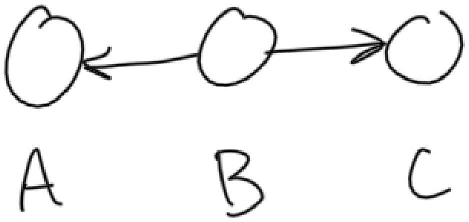


Figure 8.12: First structure, unobserved.

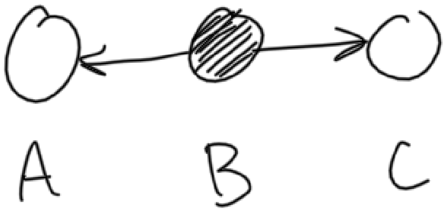


Figure 8.13: First structure, observed.

for observations to cause information to be blocked. This will lead naturally to an understanding of independence assumptions between random variables. Considering the first example random variable structure, shown in Figure 8.12.

We can factorize this as follows:

$$p(A, B, C) = p(B)p(A|B)p(C|B)$$

We know that for this case, A and C are not independent (note that we have not observed B). Therefore, we say that information flows from A to C. However, once we've observed B, we have that A and B are conditionally independent, shown in Figure 8.13.

We now say that the flow of information from A to C is 'blocked' by the observation of B. Thus they are conditionally independent. Intuitively, if we observe A but not B, then we have some information about what B might be and therefore also what C might be. The same applies in the other direction: observing C but not B.

Moving on to the second random variable structure, shown in Figure 8.14, we again consider the unobserved case first.

This allows us to write the joint distribution as:

$$p(A, B, C) = p(A)p(B|A)p(C|B)$$

We have that for this case, A and C are not independent if we have not observed C. Information is flowing from A to B through C. However, once we've observed B, then we have that A and C are again conditionally independent, shown in Figure 8.15.

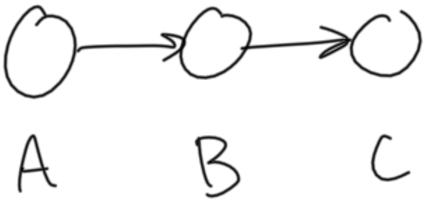


Figure 8.14: Second structure, unobserved.

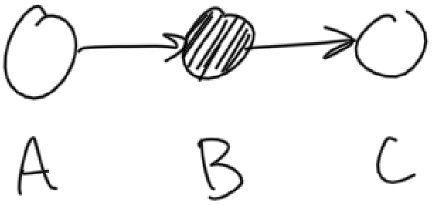


Figure 8.15: Second structure, observed.

We again say that the flow of information from A to C is 'blocked' by the observation of B. Intuitively, if we observed A but not B, we have some information about what B might be and therefore what C might be as well. The same applies in the other direction: observing C but not B.

Notice that these first two cases behave in the same manner: observing a random variable in between two other random variable 'blocks' information from flowing between the two outer random variables. In the third and final case the opposite is true. Not observing data in this case will 'block' information, and we will explain this shift through an idea known as 'explaining away'.

We have the third and final random variable structure, shown in Figure 8.16. We consider the unobserved case first.

In this setup, we say that information from A to C is being 'blocked' by the unobserved variable

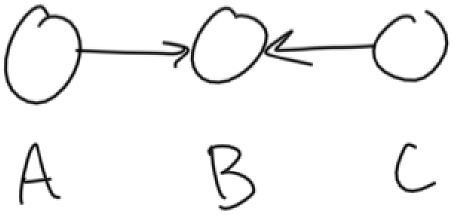


Figure 8.16: Third structure, unobserved.

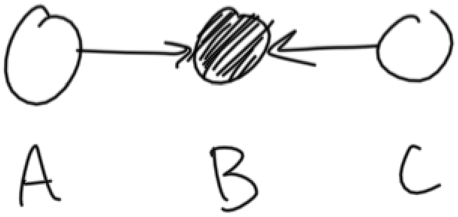


Figure 8.17: Third structure, observed.

B. Thus A and C are currently conditionally independent. However, once we've observed B, as shown in Figure 8.17, the information flow changes.

Now, information is flowing between A and C through the observed variable B, and thus A and C are now conditionally dependent. This phenomenon, where the observation of the random variable in the middle creates conditional dependence is known as *explaining away.* The idea relies on the fact that now that we know the value for B, we have some idea of how much A or C may have contributed to B adopting that value. An example might make this phenomenon more clear.

Let's consider that the random variable A corresponds to whether or not it rained on a certain day, the random variable B corresponds to the lawn being wet, and the random variable C corresponds to the sprinkler being on. Let's say we observe B: the lawn is wet. Let's say we then observe variable A: it has not rained today. Intuitively, we would now assume that variable C would have the value that the sprinkler has been on, because that's the only way for the lawn to be wet. This is exactly the phenomenon of explaining away. Observing B unblocks the flow of information between A and C because we can now use an observation to 'explain' how B got its value, and therefore determine what the other unobserved value might have been.

Notice that we've only described three simple cases relating dependence relationships between random variables in a DGM, but with just these three cases, we can determine the dependence structure of any arbitrarily complicated DGM. We just have to consider how information flows from node to node. If information gets 'blocked' at any point in our DGM network because of an observation (or lack thereof), then we gain some knowledge about independence within our model.

Consider the dependence between random variables A and F in Figure 8.18. Initially, before any observations are made, we can see that A and F are dependent (information flows from A through B). However, after observing B, A and F become independent (because information blocked at both the observed B and unobserved D). Finally, after observing D, A and F are once again dependent. Now we have information that flows from A through D.

⋆ In some other resources, you'll come upon the idea of 'D-separation' or 'D-connection'. D-separation is simply applying the principles outlined above to determine if two nodes are independent, or D-separated. On the other hand, two-nodes that are D-connected are dependent on one another.

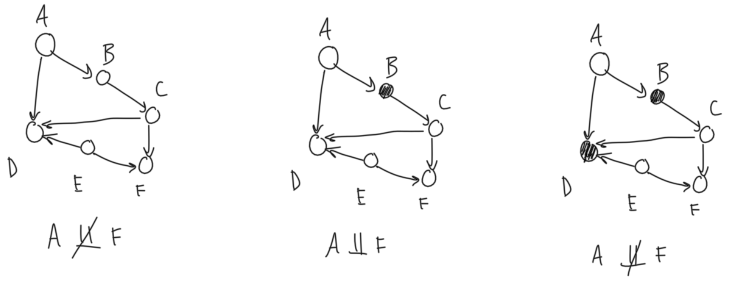


Figure 8.18: Notice how the independence between A and F depends on observations made within the network.

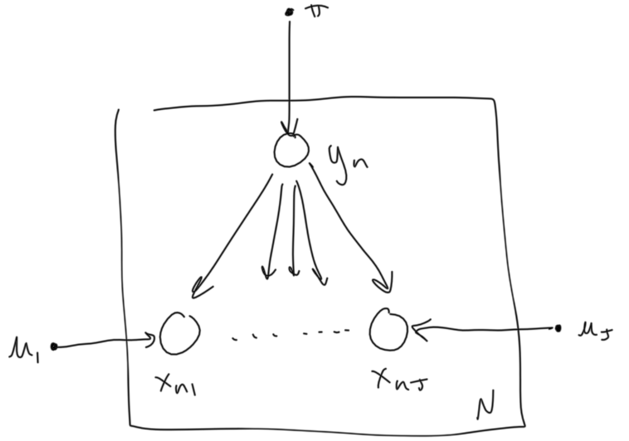


Figure 8.19: DGM for Naive Bayes problem setup.

8.3 Example: Naive Bayes

Recall from the chapter on classification the Naive Bayes model. As a quick recap, Naive Bayes makes the assumption that for a single observation coming from a specific class (for example, our classes could be different dog breeds), the data associated with that observation are independent (for example, the data could be fur color, snout length, weight, etc.).

We can set up the Naive Bayes problem specification using the DGM form, as we see in Figure 8.19.

Even if we hadn't heard of Naive Bayes before, we would understand this model and the implications of the model, simply by an examination of the DGM that describes it. We can directly read off the factorization described above that is the whole point of Naive Bayes:

$$p(y_n, x_{n1}, ..., x_{nJ}) = p(y_n)p(x_{n1}|y_n) \cdots p(x_{nJ}|y_n)$$

Writing this factorization is facilitated directly by our DGM, even if we've never heard of Naive Bayes before. It provides a common language for us to move fluidly between detailed probability factorizations and general modeling intuition.

8.4 Conclusion

Directed graphical models are indispensible for model visualization and effective communication of modeling ideas. With an understanding of what DGMs are meant to represent, it's much easier to dig into more complex probabilistic models. In a lot of ways, this chapter is preparation for where we head next. The topics of the following chapters will rely heavily on DGMs to explain their structure, use cases, and interesting variants.

Chapter 9

Mixture Models

The real world often generates observable data that falls into a combination of unseen categories. For example, at a specific moment in time I could record sound waves on a busy street that come from a combination of cars, pedestrians, and animals. If I were to try to model my data points, it would be helpful if I could group them by source, even though I didn't observe where each sound wave came from individually.

In this chapter we explore what are known as mixture models. Their purpose is to handle data generated by a combination of unobserved categories. We would like to discover the properties of these individual categories and determine how they mix together to produce the data we observe. We consider the statistical ideas underpinning mixture models, as well as how they can be used in practice.

9.1 Motivation

Mixture models are used to model data involving *latent variables*.

Definition 9.1.1 (Latent Variable): A latent variable is a piece of data that is not observed, but that influences the observed data. We often wish to create models that capture the behavior of our latent variables.

We are sometimes unable to observe all the data present in a given system. For example, if we measure the snout length of different animals but only get to see the snout measurements themselves, the latent variable would be the type of animal we are measuring for each data point. For most data generating processes, we will only have access to a portion of the data and the rest will be hidden from us. However, if we can find some way to also model the latent variables, our model will potentially be much richer, and we will also be able to probe it with more interesting questions. To build some intuition about latent variable models, we present a simple directed graphical model with a latent variable $\mathbf{z}_n$ in Figure 9.1.

One common means of modeling data involving latent variables, and the topic of this chapter, is known as a *mixture model*.

Definition 9.1.2 (Mixture Model): A mixture model captures the behavior of data coming from a combination of different distributions.

At a high level, a mixture model operates under the assumption that our data is generated by first sampling a discrete class, and then sampling a data point from within that category according

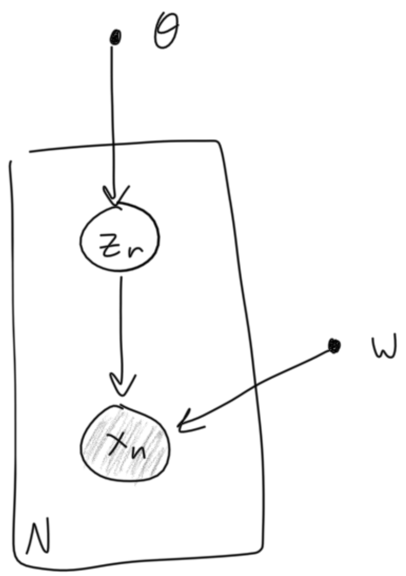


Figure 9.1: Directed graphical model with a latent variable **z**.

to the distribution for that category. For the example of animal snouts, we would first sample a species of animal, and then based on the distribution of snout lengths in that species, we would sample an observation to get a complete data point.

Probabilistically, sampling a class (which is our latent variable, since we don't actually observe it) happens according to a Categorical distribution, and we typically refer to the latent variable as **z**. Thus:

$$p(\mathbf{z} = C_k; \boldsymbol{\theta}) = \theta_k$$

where C_k is class k, and $\boldsymbol{\theta}$ is the parameter to the Categorical distribution that specifies the probability of drawing each class. We write the latent variable in bold **z** because we will typically consider it to be one-hot encoded (of dimension K, for K classes). Then, once we have a class, we have a distribution for the observed data point coming from that class:

$$p(\mathbf{x}|\mathbf{z} = C_k; \mathbf{w})$$

⋆ The distribution given by $p(\mathbf{x}|\mathbf{z} = C_k; \mathbf{w})$ is known as the *class-conditional distribution.*

This distribution depends on the type of data we are observing, and is parameterized by an arbitrary parameter **w** whose form depends on what is chosen as the class-conditional distribution. For the case of snout lengths, and many other examples, this conditional distribution is often modeled using a Gaussian distribution, in which case our model is known as a **Gaussian Mixture Model**. We will discuss Gaussian Mixture Models in more detail later in the chapter.

If we can effectively model the distribution of our observed data points and the latent variables responsible for producing the data, we will be able to ask interesting questions of our model. For example, upon observing a new data point $\mathbf{x}'$ we will be able to produce a probability that it came from a specific class $\mathbf{z}' = C_k$ using Bayes' rule and our model parameters:

$$p(\mathbf{z}' = C_k|\mathbf{x}') = \frac{p(\mathbf{x}'|\mathbf{z}' = C_k; \mathbf{w})p(\mathbf{z}' = C_k; \boldsymbol{\theta})}{\sum_{k'} p(\mathbf{x}'|\mathbf{z}' = C_{k'}; \mathbf{w})p(\mathbf{z}' = C_{k'}; \boldsymbol{\theta})}$$

Furthermore, after modeling the generative process, we will be able to generate new data points by sampling from our categorical class distribution, and then from the class-conditional distribution for that category:

$$\begin{aligned}\mathbf{z} &\sim Cat(\boldsymbol{\theta}) \\ \mathbf{x} &\sim p(\mathbf{x}|\mathbf{z} = C_k; \mathbf{w})\end{aligned}$$

Finally, it will also be possible for us to get a sense of the cardinality of $\mathbf{z}$ (meaning the number of classes our data falls into), even if that was not something we were aware of a priori.

ML Framework Cube: Mixture Models

The classes of data $\mathbf{z}$ in a mixture model will typically be discrete. Notice also that this is an unsupervised technique: while we have a data set $\mathbf{X}$ of observations, our goal is not to make predictions. Rather, we are trying to model the generative process of this data by accounting for the latent variables that generated the data points. Finally, this is a probabilistic model both for the latent variables and for our observed data.

Domain	***Training***	***Probabilistic***
Discrete	Unsupervised	Yes

9.2 Applications

Since much of the data we observe in our world has some sort of unobserved category associated with it, there are a wide variety of applications for mixture models. Here are just a few:

1. Handwriting image recognition. The categories are given by the characters (letters, numbers, etc.) and the class-conditional is a distribution over what each of those characters might look like.

2. Noise classification. The categories are given by the source of a noise (e.g. we could have different animal noises), and the class-conditional is a distribution over what the sound waves for each animal noise look like.

3. Vehicle prices. The categories are given by the brand of vehicle (we could alternatively categorize by size, safety, year, etc.), and the class-conditional is a distribution over the price of each brand.

9.3 Fitting a Model

We've defined the general form of a mixture model: we have a distribution $p(\mathbf{z}; \boldsymbol{\theta})$ over our classes and a distribution $p(\mathbf{x}|\mathbf{z} = C_k; \mathbf{w})$ as our class-conditional distribution. A natural approach would be to compute the maximum likelihood values for our parameters $\boldsymbol{\theta}$ and $\mathbf{w}$. Let's consider how we might go about this for a mixture model.

9.3.1 Maximum Likelihood for Mixture Models

Our goal is to maximize the likelihood of our observed data. Because we don't actually observe the latent variables $\mathbf{z}_n$ which determine the class of each observed data point $\mathbf{x}_n$, we can simply sum over the possible classes for each of our N data points as follows:

$$p(\mathbf{X}; \boldsymbol{\theta}, \mathbf{w}) = \prod_{n=1}^{N} \sum_{k=1}^{K} p(\mathbf{x}_n, z_{n,k}; \boldsymbol{\theta}, \mathbf{w})$$

This uses $p(\mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w}) = \sum_k p(\mathbf{x}_n, z_{n,k}; \boldsymbol{\theta}, \mathbf{w})$ (marginalizing out over the latent class). Taking the logarithm to get our log-likelihood as usual:

$$\log p(\mathbf{X}; \boldsymbol{\theta}, \mathbf{w}) = \sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} p(\mathbf{x}_n, z_{n,k}; \boldsymbol{\theta}, \mathbf{w}) \right] \tag{9.1}$$

It may not be immediately obvious, but under this setup, the maximum likelihood calculation for our parameters $\boldsymbol{\theta}$ and $\mathbf{w}$ is now intractable. The summation over the K classes of our latent variable $\mathbf{z}_n$, which is required because we don't actually observe those classes, is inside of the logarithm, which prevents us from arriving at an analytical solution (it may be helpful to try to solve this yourself, you'll realize that consolidating a summation inside of a logarithm is not possible). You could still try to use gradient descent, but the problem is non-convex and we'll see a much more elegant approach. The rest of this chapter will deal with how we can optimize our mixture model in the face of this challenge.

9.3.2 Complete-Data Log Likelihood

We have a problem with computing the MLE for our model parameters. If we only knew which classes our data points came from, i.e., if we had $\mathbf{z}_n$ for each example n, then we would be able to calculate $\log p(\mathbf{x}, \mathbf{z})$ with relative ease because we would no longer require a summation inside the logarithm:

$$\begin{aligned} \log p(\mathbf{X}, \mathbf{Z}) &= \sum_{n=1}^{N} \log p(\mathbf{x}_n, \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) && (9.2) \\ &= \sum_{n=1}^{N} \log[p(\mathbf{x}_n | \mathbf{z}_n; \mathbf{w}) p(\mathbf{z}_n; \boldsymbol{\theta})] && (9.3) \\ &= \sum_{n=1}^{N} \log p(\mathbf{x}_n | \mathbf{z}_n; \mathbf{w}) + \log p(\mathbf{z}_n; \boldsymbol{\theta}) && (9.4) \end{aligned}$$

Notice that because we've now observed $\mathbf{z}_n$, we don't have to marginalize over its possible values. This motivates an interesting approach that takes advantage of our ability to work with $p(\mathbf{x}, \mathbf{z})$ if we only knew $\mathbf{z}$.

The expression p($\mathbf{x}$, $\mathbf{z}$) is known as the *complete-data likelihood* because it assumes that we have both our observation $\mathbf{x}$ and the class $\mathbf{z}$ that $\mathbf{x}$ came from. Our ability to efficiently calculate the complete-data log likelihood $\log p(\mathbf{x}, \mathbf{z})$ is the crucial piece of the algorithm we will present to optimize our mixture model parameters. This algorithm is known as **Expectation-Maximization**, or **EM** for short.

9.4 Expectation-Maximization (EM)

The motivation for the EM algorithm, as presented in the previous section, is that we do not have a closed form optimization for our mixture model parameters due to the summation inside of the logarithm. This summation was required because we didn't observe a crucial piece of data, the class $\mathbf{z}$, and therefore we had to sum over its values.

EM uses an iterative approach to optimize our model parameters. It proposes a soft value for $\mathbf{z}$ using an expectation calculation (we can think about this as giving a distribution on $\mathbf{z}_n$ for each n), and then based on that proposed value, it maximizes the *expected* complete-data log likelihood with respect to the model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$ via a standard MLE procedure.

Notice that EM is composed of two distinct steps: an "E step" that finds the expected value of the latent class variables given the current set of parameters, and an "M step" that improves the model parameters by maximizing expected complete-data log likelihood given these soft assignments to class variables. These two steps give the algorithm its name, and more generally, this type of approach is also referred to as **coordinate ascent**. The idea behind coordinate ascent is that we can replace a hard problem (maximizing the log likelihood for our mixture model directly) with two easier problems, namely the E- and M-step. We alternate between the two easier problems, executing each of them until we reach a point of convergence or decide that we've done enough. We may also restart because EM will provide a local but not global optimum.

We'll walk through the details of each of these two steps and then tie them together with the complete algorithm.

⋆ K-Means, an algorithm we discussed in the context of clustering, is also a form of coordinate ascent. K-Means is sometimes referred to as a "maximization-maximization" algorithm because we iteratively maximize our assignments (by assigning each data point to just a single cluster) and then update our cluster centers to maximize their likelihood with respect to the new assignments. That is, it does a "max" in place of the E-etep, making a hard rather than soft assignment.

9.4.1 Expectation Step

The purpose of the E-step is to find expected values of the latent variables $\mathbf{z}_n$ for each example given the current parameter values. Let's consider what this looks like with a concrete example.

Let's say our data points $\mathbf{x}_n$ can come from one of three classes. Then, we can represent the latent variable $\mathbf{z}_n$ associated with each data point using a one-hot encoded vector. For example, if $\mathbf{z}_n$ came from class C_1, we would denote this:

$$\mathbf{z}_n = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

As we've already explained, we don't know the true value of this latent variable. Instead, we will compute its conditional expectation based on the current setting of our model parameters and our observed data $\mathbf{x}_n$. We denote the expectation of our latent variables as $\mathbf{q}_n$, and we calculate them as follows:

$$\mathbf{q}_n = \mathbb{E}[\mathbf{z}_n|\mathbf{x}_n] = \begin{bmatrix} p(\mathbf{z}_n = C_1|\mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w}) \\ p(\mathbf{z}_n = C_2|\mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w}) \\ p(\mathbf{z}_n = C_3|\mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w}) \end{bmatrix} \propto \begin{bmatrix} p(\mathbf{x}_n|\mathbf{z}_n = C_1; \mathbf{w})p(\mathbf{z}_n = C_1; \boldsymbol{\theta}) \\ p(\mathbf{x}_n|\mathbf{z}_n = C_2; \mathbf{w})p(\mathbf{z}_n = C_2; \boldsymbol{\theta}) \\ p(\mathbf{x}_n|\mathbf{z}_n = C_3; \mathbf{w})p(\mathbf{z}_n = C_3; \boldsymbol{\theta}) \end{bmatrix}$$

The expectation of a 1-hot encoded vector is equivalent to a distribution on the values that the latent variable might take on. Notice that we can switch from proportionality in our $\mathbf{q}_n$ values to actual probabilities by simply dividing each unnormalized value by the sum of all the unnormalized values. Then, our $\mathbf{q}_n$ values will look something like the following, where a larger number indicates a stronger belief that the data point $\mathbf{x}_n$ came from that class:

$$\mathbf{q}_n = \begin{bmatrix} 0.8 \\ 0.1 \\ 0.1 \end{bmatrix}$$

There are two important things to note about the expectation step. First, the model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$ are held fixed. We're computing the expectation of our latent variables based on the current setting of those model parameters. Those parameters are randomly initialized if this is our first time running the expectation step.

Second, we have a value of $\mathbf{q}_n$ for every data point $\mathbf{x}_n$ in our data set. As a result, $\mathbf{q}_n$ are sometimes called "local parameters," since there is one assigned to each data point. This is in contrast to our model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$, which are "global parameters." The size of the global model parameters doesn't fluctuate based on the size of our data set.

After performing the E-step, we now have an expectation for our latent variables, given by $\mathbf{q}_n$. In the maximization step, which we describe next, we use these $\mathbf{q}_n$ values to improve our global parameters.

9.4.2 Maximization Step

After the expectation step, we have a $\mathbf{q}_n \in [0, 1]^K$ (and summing to one) associated with each data point $\mathbf{x}_n$, which describes our belief that the data point came from each class C_k. Now that we have these expected 'class assignments', it's possible for us to maximize the expected complete-data likelihood with respect to our model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$.

Recall that optimizing our parameters using the complete-data log likelihood is tractable because we avoid summing over the classes inside of the logarithm. Although we do not have the actual complete-data, since we don't know the true $\mathbf{z}_n$ values, we now have a distribution over these latent variables (given by $\mathbf{q}_n$).

Notice that our $\mathbf{q}_n$ values are 'soft' assignments - meaning that unlike the $\mathbf{z}_n$ values, which are one-hot encodings of assignments to a class, the $\mathbf{q}_n$ values have a probability that a data point $\mathbf{x}_n$ came from each class. Recall the expression for complete-data log likelihood:

$$\log p(\mathbf{X}, \mathbf{Z}) = \sum_{n=1}^{N} \log p(\mathbf{x}_n | \mathbf{z}_n; \mathbf{w}) + \log p(\mathbf{z}_n; \boldsymbol{\theta}) \tag{9.5}$$

We work instead with the expected complete-data log likelihood, using $\mathbf{q}_n$ to provide the distribution on $\mathbf{z}_n$ for each example n:

$$\begin{aligned}
\mathrm{E}_{\mathbf{z}_n|\mathbf{x}_n}[\log p(\mathbf{X}, \mathbf{Z})] &= \mathrm{E}_{\mathbf{z}_n|\mathbf{x}_n}\left[\sum_{n=1}^{N} \log p(\mathbf{x}_n | \mathbf{z}_n; \mathbf{w}) + \log p(\mathbf{z}_n; \boldsymbol{\theta})\right] && (9.6) \\
&= \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n|\mathbf{x}_n}\left[\log p(\mathbf{x}_n | \mathbf{z}_n; \mathbf{w}) + \log p(\mathbf{z}_n; \boldsymbol{\theta})\right] && (9.7) \\
&= \sum_{n=1}^{N} \sum_{k=1}^{K} p(\mathbf{z}_n = C_k | \mathbf{x}_n)\big(\log p(\mathbf{x}_n | \mathbf{z}_n = C_k; w) + \log p(\mathbf{z}_n = C_k; \boldsymbol{\theta})\big) && (9.8) \\
&= \sum_{n=1}^{N} \sum_{k=1}^{K} q_{n,k}\big(\log p(\mathbf{x}_n | \mathbf{z}_n = C_k; \mathbf{w}) + \log p(\mathbf{z}_n = C_k; \boldsymbol{\theta})\big) && (9.9)
\end{aligned}$$

Notice the crucial difference between this summation and that of Equation 9.1: the summation over the classes is now outside of the logarithm! Recall that using the log-likelihood directly was intractable precisely because the summation over the classes was inside of the logarithm. This maximization became possible by taking the expectation over our latent variables (using the values we computed in the E-step), which moved the summation over the classes outside of the logarithm.

We can now complete the M-step by maximizing Equation 9.6 with respect to our model parameters $\boldsymbol{\theta}$ and $\mathbf{w}$. This has an analytical solution. We take the derivative with respect to the parameter of interest, set to 0, solve, and update the parameter with the result.

9.4.3 Full EM Algorithm

Now that we have a grasp on the purpose of the EM algorithm, as well as an understanding of the expectation and maximization steps individually, we are ready to put everything together to describe the entire EM algorithm.

1. Begin by initializing our model parameters $\mathbf{w}$ and $\boldsymbol{\theta}$, which we can do at random. Since the EM algorithm is performed over a number of iterative steps, we will denote these initial parameter values $\mathbf{w}^{(0)}$ and $\boldsymbol{\theta}^{(0)}$. We will increment those values as the algorithm proceeds.

2. E-step: compute the values of $\mathbf{q}_n$ based on the current setting of our model parameters.

$$\mathbf{q}_n = \mathbb{E}[\mathbf{z}_n|\mathbf{x}_n] = \begin{bmatrix} p(\mathbf{z}_n = C_1|\mathbf{x}_n; \boldsymbol{\theta}^{(i)}, \mathbf{w}^{(i)}) \\ \vdots \\ p(\mathbf{z}_n = C_K|\mathbf{x}_n; \boldsymbol{\theta}^{(i)}, \mathbf{w}^{(i)}) \end{bmatrix} \propto \begin{bmatrix} p(\mathbf{x}_n|\mathbf{z}_n = C_1; \mathbf{w}^{(i)})p(\mathbf{z}_n = C_1; \boldsymbol{\theta}^{(i)}) \\ \vdots \\ p(\mathbf{x}_n|\mathbf{z}_n = C_K; \mathbf{w}^{(i)})p(\mathbf{z}_n = C_K; \boldsymbol{\theta}^{(i)}) \end{bmatrix}$$

3. M-step: compute the values of $\mathbf{w}$ and $\boldsymbol{\theta}$ that maximize our expected complete-data log likelihood for the current setting of the values of $\mathbf{q}_n$:

$$\mathbf{w}^{(i+1)}, \boldsymbol{\theta}^{(i+1)} \in \arg\max_{\mathbf{w},\boldsymbol{\theta}} \mathbb{E}_{\mathbf{Z}|\mathbf{X}}[\log p(\mathbf{X}, \mathbf{Z}; \mathbf{w}, \boldsymbol{\theta})] \tag{9.10}$$

4. Return to step 2, repeating this cycle until our likelihood converges. Note that the likelihood is guaranteed to (weakly) increase at each step using this procedure.

It is also typical to re-start the procedure because we are guaranteed a local but not global optimum.

9.4.4 The Math of EM

The above sections supply an intuitive sense of why EM should work in scenarios where the latent variables follow categorical distributions. We now supply a more rigorous proof that iteratively computing the quantities derived above will increase the data log likelihood in a more general setting.

The Evidence Lower Bound (ELBO)

Despite our inability to directly optimize the observed data log likelihood, it remains the quantity of interest. Note the following equivalence:

$$\sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} p(\mathbf{x}_n, \mathbf{z}_n = C_k; \boldsymbol{\theta}, \mathbf{w}) \right] = \sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} p(\mathbf{x}_n \mid \mathbf{z}_n = C_k; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n = C_k \mid \boldsymbol{\theta}, \mathbf{w}) \right] \tag{9.11}$$

$$= \sum_{n=1}^{N} \log \mathrm{E}_{\mathbf{z}_n \sim p(\mathbf{z}_n \mid \boldsymbol{\theta})} \left[p(\mathbf{x}_n \mid \mathbf{z}_n, \mathbf{w}) \right] \tag{9.12}$$

We adopt this latter expression, since it also holds in cases where $\mathbf{z}_n$ is not discrete.

As touched on earlier, the main issues with directly optimizing this expression are as follows:

1. If we were to take a gradient, the distribution over which we're taking an expectation involves one of the parameters, $\boldsymbol{\theta}$, and hence computing the gradient with respect to $\boldsymbol{\theta}$ is difficult.

2. There's an expectation (previously a sum) inside the logarithm.

To solve this, we ultimately derive a lower bound on the observed data log likelihood which proves to be computationally tractable to optimize. First, let $q(\mathbf{z}_n)$ denote another probability distribution on $\mathbf{z}_n$. Then:

$$\begin{aligned}
&\sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} p(\mathbf{x}_n, \mathbf{z}_n = C_k; \boldsymbol{\theta}, \mathbf{w}) \right] \\
&= \sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} p(\mathbf{x}_n \mid \mathbf{z}_n = C_k; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n = C_k \mid \boldsymbol{\theta}, \mathbf{w}) \right] \\
&= \sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} p(\mathbf{x}_n \mid \mathbf{z}_n = C_k; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n = C_k \mid \boldsymbol{\theta}, \mathbf{w}) \cdot \frac{q(\mathbf{z}_n = C_k)}{q(\mathbf{z}_n = C_k)} \right] \\
&= \sum_{n=1}^{N} \log \left[\sum_{k=1}^{K} \frac{p(\mathbf{x}_n \mid \mathbf{z}_n = C_k; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n = C_k \mid \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n = C_k)} \cdot q(\mathbf{z}_n = C_k) \right] \\
&= \sum_{n=1}^{N} \log \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\frac{p(\mathbf{x}_n \mid \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n)} \right] \qquad (9.13)
\end{aligned}$$

The above derivation again restricts to discrete $\mathbf{z}_n$, but the equivalence between the expressions in (9.12) and (9.13) is in fact more general, holding whenever p is absolutely continuous with respect to the chosen q.

In any case, we've now fixed the first of the two issues. By introducing the distribution q, the expectation is no longer over some distribution depending on the parameter $\boldsymbol{\theta}$.

To fix the second issue, we must somehow pass the log into the expectation. This is accomplished using Jensen's inequality, at the cost of turning the equality into a lower bound:

$$\sum_{n=1}^{N} \log \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\frac{p(\mathbf{x}_n \mid \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n)} \right] \tag{9.14}$$

$$\geq \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \left(\frac{p(\mathbf{x}_n \mid \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n)} \right) \right] \tag{9.15}$$

In summary, the two issues with directly optimizing the observed data log likelihood were resolved as follows:

- Introducing auxiliary variables $q(\mathbf{z}_n)$ allow us to reparametrize the expectation to be over q rather than $p(\mathbf{z}_n \mid \boldsymbol{\theta})$.
- Jensen's inequality allows the log to pass through the expectation.

However, this came at the cost of converting our objective into a lower bound of the original quantity, as well as introducing the new parameter that is the distribution q.

It turns out that the iterative process given in the section above amounts to alternating between optimizing the parameters $\mathbf{w}, \boldsymbol{\theta}$ and the distribution q.

Optimization

Before delving into the optimization process, we first establish two identities involving the ELBO.

$$\mathrm{ELBO}(\mathbf{w}, q) = \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \left(\frac{p(\mathbf{x}_n \mid \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n)} \right) \right] \tag{9.16}$$

First,

$$\begin{aligned}
\mathrm{ELBO}(\mathbf{w}, q) &= \sum_{n=1}^{N} \left(\mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \left(p(\mathbf{x}_n \mid \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w}) \right) \right] - \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} [q(\mathbf{z}_n)] \right) \\
&= \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \left(p(\mathbf{x}_n \mid \mathbf{z}_n; \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w}) \right) \right] - \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} [q(\mathbf{z}_n)] \\
&= \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log p(\mathbf{x}_n, \mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w}) \right] - \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} [q(\mathbf{z}_n)] \qquad (9.17)
\end{aligned}$$

The first term in (9.17) is the expected complete data log likelihood, and the second is the entropy of the distribution q.

Second, using Bayes' rule:

$$\begin{aligned}
\mathrm{ELBO}(\mathbf{w}, q) &= \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \left(\frac{p(\mathbf{x}_n \mid \boldsymbol{\theta}, \mathbf{w}) p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n)} \right) \right] \\
&= \sum_{n=1}^{N} \left(\log p(\mathbf{x}_n \mid \boldsymbol{\theta}, \mathbf{w}) + \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \frac{p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w})}{q(\mathbf{z}_n)} \right] \right) \\
&= \sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \boldsymbol{\theta}, \mathbf{w}) - \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim q(\mathbf{z}_n)} \left[\log \frac{q(\mathbf{z}_n)}{p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w})} \right] \qquad (9.18)
\end{aligned}$$

The first term is exactly the observed data log likelihood, and the second is known as the *KL divergence* between $q(\mathrm{z}_n)$ and $p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w})$. In particular, the KL-divergence between distributions P and Q is defined as

$$\mathrm{D}_{\mathrm{KL}}(P \| Q) = \mathrm{E}_{x \sim P(x)} \left[\log \frac{P(x)}{Q(x)} \right] \tag{9.19}$$

The key property of KL-divergence is that it is nonnegative, that is $\mathrm{D}_{\mathrm{KL}}(P\|Q) \geq 0$ always, with equality if and only if $P = Q$:

$$\begin{aligned} -\mathrm{D}_{\mathrm{KL}}(P\|Q) = \mathrm{E}_{x\sim P(x)}\left[\log \frac{Q(x)}{P(x)}\right] &\leq \log \mathrm{E}_{x\sim P(x)}\left[\frac{Q(x)}{P(x)}\right] \\ &\leq \log \sum \frac{Q(x)}{P(x)} \cdot P(x) \\ &\leq 0 \end{aligned}$$

where we have equality when the equality condition of Jensen's is met, meaning $Q(x) = cP(x)$, which forces $Q(x) = P(x)$ since both are distributions. The proof above is only for discrete variables, but can again be extended to the continuous case.

Using this, we rewrite (9.18) as

$$\mathrm{ELBO}(\mathbf{w}, q) = \sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \boldsymbol{\theta}, \mathbf{w}) - \sum_{n=1}^{N} \mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}_n)\|p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w})) \tag{9.20}$$

We now describe the E and M steps of the procedure. Begin with some random assignment of $\boldsymbol{\theta}, \mathbf{w}$, and q.

1. The E-step: Fixing the parameters $\boldsymbol{\theta}, \mathbf{w}$, choose the distributions q that maximizes $\mathrm{ELBO}(\mathbf{w}, q)$.

   Observe that only the second term of (9.20) depends on the distributions q. Hence maximizing the ELBO is equivalent to minimizes the sum of KL divergences $\sum_{n=1}^{N} \mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}_n)\|p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w}))$. This immediately yields that

   $$q(\mathbf{z}_n) = p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w}).$$

2. The M-step: Fixing the distributions q, maximize the ELBO with respect to the parameters $\boldsymbol{\theta}, \mathbf{w}$.

   Observe that only the first term of (9.17) depends on the parameters, and hence this is equivalent to maximizing the expected complete data log likelihood,

   $$\sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n\sim q(\mathbf{z}_n)}\left[\log p(\mathbf{x}_n, \mathbf{z}_n \mid \boldsymbol{\theta}, \mathbf{w})\right]. \tag{9.21}$$

An equivalent formulation is then as follows. Define the auxiliary function $Q(\mathbf{w}, \boldsymbol{\theta} \mid \mathbf{w}^{\mathrm{old}}, \boldsymbol{\theta}^{\mathrm{old}})$ as

$$Q(\mathbf{w}, \boldsymbol{\theta} \mid \mathbf{w}^{\mathrm{old}}, \boldsymbol{\theta}^{\mathrm{old}}) = \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n\sim p(\mathbf{z}_n|\mathbf{x_n};\mathbf{w}^{\mathrm{old}},\boldsymbol{\theta}^{\mathrm{old}})}\left[\log p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}, \boldsymbol{\theta})\right] \tag{9.22}$$

Then as before, begin with some random assignment of $\mathbf{w}, \boldsymbol{\theta}, q$.

1. The E-step: compute the auxiliary function $Q(\mathbf{w}, \boldsymbol{\theta}; \mathbf{w}^{\mathrm{old}}, \boldsymbol{\theta}^{\mathrm{old}})$. Note that this essentially just requires computing the posterior distribution $p(\mathbf{z}_n \mid \mathbf{x_n}; \mathbf{w}^{\mathrm{old}}, \boldsymbol{\theta}^{\mathrm{old}})$, as before.

2. The M-step: maximize $Q(\mathbf{w}, \boldsymbol{\theta}; \mathbf{w}^{\mathrm{old}}, \boldsymbol{\theta}^{\mathrm{old}})$ with respect to $\mathbf{w}$ and $\boldsymbol{\theta}$.

Why are these equivalent? We've dropped the qs by noting that at each step, they will just be $p(\mathbf{z}_n \mid \mathbf{x}_n; \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})$.

It is important to note the significance of expression (9.20). Since the first term is exactly the observed data log likelihood, this indicates that the gap between the ELBO and the observed data log likelihood is exactly the KL divergence between $q(\mathbf{z}_n)$ and $p(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}, \mathbf{w})$. If this gap goes to zero, then we will have succeeded in maximizing the observed data log likelihood. This is why it is important to continually maximize over the distributions q - choosing the best q makes the bound as tight as possible every round.

Correctness

We will only show that EM increases the observed log likelihood at every iteration. We do this using the second of the two formulations.

$$\begin{aligned}
&\sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \mathbf{w}, \boldsymbol{\theta}) - \sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}}) \\
&= \sum_{n=1}^{N} \log \frac{p(\mathbf{x}_n \mid \mathbf{w}, \boldsymbol{\theta})}{p(\mathbf{x}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \\
&= \sum_{n=1}^{N} \log \frac{\sum_{k=1}^{K} p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}, \boldsymbol{\theta})}{p(\mathbf{x}_n \mid \mathbf{z}_n; \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \\
&= \sum_{n=1}^{N} \log \sum_{k=1}^{K} \frac{p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}, \boldsymbol{\theta})}{p(\mathbf{x}_n \mid \mathbf{z}_n; \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}}) p(\mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} p(\mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}}) \\
&= \sum_{n=1}^{N} \log \mathrm{E}_{\mathbf{z}_n \sim p(\mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \left[\frac{p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}, \boldsymbol{\theta})}{p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \right] \\
&\geq \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim p(\mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \left[\log \frac{p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}, \boldsymbol{\theta})}{p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \right] \\
&= \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim p(\mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \left[\log p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}, \boldsymbol{\theta}) \right] - \sum_{n=1}^{N} \mathrm{E}_{\mathbf{z}_n \sim p(\mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}})} \left[\log p(\mathbf{x}_n, \mathbf{z}_n \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}}) \right] \\
&= Q(\mathbf{w}, \boldsymbol{\theta} \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta}) - Q(\mathbf{w}^{\text{old}}, \boldsymbol{\theta}^{\text{old}} \mid \mathbf{w}^{\text{old}}, \boldsymbol{\theta})
\end{aligned}$$

We are maximizing Q at each step, and thus this difference is strictly positive.

Equivalence to Prior Formulation

It is not hard to see that the formulation presented above, centered around maximizing the ELBO, is equivalent to the one presented in section 9.4.3. Note that $q_{n,k}$ is exactly $p(\mathbf{z}_n = C_k \mid \mathbf{x}_n, \mathbf{w}, \boldsymbol{\theta})$, and thus the quantity being computed in the M-step (as in (9.10)) is exactly the same as in (9.21).

9.4.5 Connection to K-Means Clustering

At this point, it's worth considering the similarity between the EM algorithm and another coordinate ascent algorithm that we considered in the context of clustering: K-Means.

Recall that K-Means proceeds according to a similar iterative algorithm: we first make hard assignments of data points to existing cluster centers, and then we update the cluster centers based on the most recent data point assignments.

In fact, the main differences between K-Means clustering and the EM algorithm are that:

1. In the EM setting, we make soft cluster assignments through our $\mathbf{q}_n$ values, rather than definitively assigning each data point to only one cluster.

2. The EM algorithm is able to take advantage of flexible, class-conditional distributions to capture the behavior of the observed data, whereas K-Means clustering relies only on distance measurements to make assignments and update cluster centers.

In the context of a mixture-of-Gaussian model, which we get to later in the chapter, we can confirm that K-means is equal to the limiting case of EM where the variance of each class-conditional Gaussian goes to 0, the prior probability of each class is uniform, and the distributions are spherical.

9.4.6 Dice Example: Mixture of Multinomials

Consider the following example scenario: we have two biased dice (with 6 faces) and one biased coin (with 2 sides). Data is generated as follows: first, the biased coin is flipped. Suppose it lands heads. Then dice 1 is rolled $c = 10$ times. This gives the first example, i.e., $\mathbf{x}_1$ would correspond to the number of times of rolling each of a $1, 2, \ldots, 6$. Then we repeat, flipping the biased coin. Suppose it lands tails. Then Dice 2 is rolled 10 times. We record the result of the dice rolls as $\mathbf{x}_2$. We keep repeating, obtaining additional examples.

For example, our observations for the first 10 rolls may look like: $1, 5, 3, 4, 2, 2, 3, 1, 6, 2$ and we'd record as our first example

$$\mathbf{x}_1 = \begin{bmatrix} 2 \\ 3 \\ 2 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

We're going to try to infer the parameters of each of the dice based on these observations. Let's consider how this scenario fits into our idea of a mixture model. First, the latent variable $\mathbf{z}_n$ has a natural interpretation as being which dice was rolled for the n^{th} observed data point $\mathbf{x}_n$. We can represent $\mathbf{z}_n$ using a one-hot vector, so that if the n^{th} data point came from Dice 1, we'd denote that:

$$\mathbf{z}_n = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

We denote the probability vector associated with the biased coin as $\boldsymbol{\theta} \in [0, 1]^2$, summing to 1, with θ_1 being the probability of the biased coin landing heads and θ_2 being the probability of the biased coin landing tails. Furthermore, we need parameters to describe the behavior of the biased dice. We use $\boldsymbol{\pi}_\mathbf{1}, \boldsymbol{\pi}_2 \in [0, 1]^6$, summing to 1, where each 6-dimensional vector describes the probability that the respective dice lands on each face.

For a given dice, this defines a multinomial distribution. For c trials, and counts $x_1, \dots, x_6$ for each of 6 faces on a 6-sided dice, and probabilities $\boldsymbol{\pi}$, this is

$$p(\mathbf{x}; \boldsymbol{\pi}) = \frac{c!}{x_1! \cdot \ldots \cdot x_6!} \pi_1^{x_1} \cdot \ldots \cdot \pi_6^{x6} \tag{9.23}$$

For our purposes, let $p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ denote the multinomial distribution on observation $\mathbf{x}_{nj}$ when latent vector $\mathbf{z}_n = C_k$.

The model parameters are $\mathbf{w} = \{\boldsymbol{\theta}, \boldsymbol{\pi}_1, \boldsymbol{\pi}_2\}$. We can optimize the model parameters using EM. We start by initializing the parameters $\boldsymbol{\theta}^{(0)}, \boldsymbol{\pi}^{(0)}$.

In the E-step, we compute the soft assignment values, $\mathbf{q}_n$. For dice k, this given by

$$\begin{aligned} q_{nk} =& p(\mathbf{z}_n = C_k | \mathbf{x}_n; \boldsymbol{\theta}^{(i)}, \boldsymbol{\pi}^{(i)}) && (9.24) \\ =& \frac{p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) p(\mathbf{z}_n = C_k; \boldsymbol{\theta}^{(i)})}{\sum_{k=1}^K p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) p(\mathbf{z}_n = C_k; \boldsymbol{\theta}^{(i)})} && (9.25) \\ =& \frac{p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) \theta_k^{(i)}}{\sum_{k=1}^K p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) \theta_k^{(i)}} && (9.26) \end{aligned}$$

We could also use the "product trick" to write a single expression

$$\begin{aligned} p(\mathbf{z}_n | \mathbf{x}_n; \boldsymbol{\theta}^{(i)}, \boldsymbol{\pi}^{(i)}) &= \frac{\prod_{k=1}^K \left(p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) \theta_k^{(i)} \right)^{z_{nk}}}{\sum_{k=1}^K p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) \theta_k^{(i)}} \\ &= \frac{\prod_{k=1}^K p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)})^{z_{nk}} \prod_{k=1}^K (\theta_k^{(i)})^{z_{nk}}}{\sum_{k=1}^K p(\mathbf{x}_n | \mathbf{z}_n = C_k; \boldsymbol{\pi}^{(i)}) \theta_k^{(i)}} \end{aligned}$$

The vector $\mathbf{q}_n$ is defined as:

$$\mathbf{q}_n = \begin{bmatrix} p(\mathbf{z}_n = C_1 | \mathbf{x}_n; \boldsymbol{\theta}^{(i)}, \boldsymbol{\pi}^{(i)}) \\ p(\mathbf{z}_n = C_2 | \mathbf{x}_n; \boldsymbol{\theta}^{(i)}, \boldsymbol{\pi}^{(i)}) \end{bmatrix} \tag{9.27}$$

After computing the values of $\mathbf{q}_n$, we are ready to perform the M-step. Recall that we are maximizing the expected complete-data log likelihood, which takes the form:

$$\begin{aligned} \mathbb{E}_{\mathbf{Z}|\mathbf{X}}[\log p(\mathbf{X}, \mathbf{Z})] &= \mathbb{E}_{\mathbf{q}_n} \left[\sum_{n=1}^N \log p(\mathbf{z}_n; \boldsymbol{\theta}^{(i+1)}, \boldsymbol{\pi}^{(i+1)}) + \log p(\mathbf{x}_n | \mathbf{z}_n; \boldsymbol{\theta}^{(i+1)}, \boldsymbol{\pi}^{(i+1)}) \right] && (9.28) \\ &= \sum_{n=1}^N \mathbb{E}_{\mathbf{z}_n | \mathbf{x}_n} \left[\log p(\mathbf{z}_n; \boldsymbol{\theta}^{(i+1)}, \boldsymbol{\pi}^{(i+1)}) + \log p(\mathbf{x}_n | \mathbf{z}_n; \boldsymbol{\theta}^{(i+1)}, \boldsymbol{\pi}^{(i+1)}) \right] && (9.29) \end{aligned}$$

We can then substitute in for the multinomial expression and simplify, and dropping constants we have that we're looking for parameters that solve

$$\begin{aligned} &\arg \max_{\boldsymbol{\theta}^{(i+1)}, \boldsymbol{\pi}^{(i+1)}} \left\{ \sum_{n=1}^N \sum_{k=1}^2 q_{n,k} \log \theta_k^{(i+1)} + \sum_{n=1}^N \sum_{k=1}^2 q_{n,k} \log \left(\pi_{k,1}^{x_{n,1}} \cdot \ldots \cdot \pi_{k,6}^{x_{n,6}} \right) \right\} \\ &= \arg \max_{\boldsymbol{\theta}^{(i+1)}, \boldsymbol{\pi}^{(i+1)}} \left\{ \sum_{n=1}^N \sum_{k=1}^2 q_{n,k} \log \theta_k^{(i+1)} + \sum_{n=1}^N \sum_{k=1}^2 \sum_{j=1}^6 q_{n,k} x_{n,j} \log(\pi_{k,j}) \right\} && (9.30) \end{aligned}$$

To maximize the expected complete-data log likelihood, it's necessary to introduce Lagrange multipliers to enforce the constraints $\sum_k \theta_k^{(i+1)} = 1$ and $\sum_j \pi_{k,j}^{(i+1)} = 1$, for each k. After doing this, and solving, we recover the following update equations for the model parameters:

$$\theta_k^{(i+1)} \leftarrow \frac{\sum_{n=1}^N q_{n,k}}{N}$$

$$\boldsymbol{\pi}_k^{(i+1)} \leftarrow \frac{\sum_{n=1}^N q_{n,k}\mathbf{x}_n}{c\sum_{n=1}^N q_{n,k}},$$

where $c = 10$ in out example.

We now have everything we need to perform EM for this setup. After initializing our parameters $\mathbf{w}^{(0)}$, we perform the E-step by evaluating 9.27. After calculating our values of $\mathbf{q}_n$ in the E-step, we update our parameters $\mathbf{w} = \{\boldsymbol{\theta}, \boldsymbol{\pi}_1, \boldsymbol{\pi}_2\}$ in the M-step by maximizing 9.30 with respect to $\boldsymbol{\theta}, \boldsymbol{\pi}_1, \boldsymbol{\pi}_2$. We perform these two steps iteratively, until convergence of our parameters. We may also do a restart.

9.5 Gaussian Mixture Models (GMM)

Our previous example was a simple but somewhat restricted application of the EM algorithm to solving a latent variable problem. We now turn to a more practical example, used widely in different contexts, called a Gaussian Mixture Model (GMM). As you might expect, a GMM consists of a combination of multiple Gaussian distributions. Among other things, it is useful for modeling scenarios where the observed data is continuous.

Let's go over a more rigorous formulation of the GMM setup. First, we have observed continous data $\mathbf{x}_n \in \mathbb{R}^m$ and latent variables $\mathbf{z}_n$ which indicate which Gaussian 'cluster' our observed data point was drawn from. In other words:

$$p(\mathbf{x}_n|\mathbf{z}_n = C_k) = \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

where $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$ are the mean and covariance parameters respectively for the k^{th} cluster center.

The data generation process works as follows: we first sample a cluster center from a Categorical distribution parameterized by $\boldsymbol{\theta} \in \mathrm{R}^K$. Then, based on the sampled cluster center, we sample a data point $\mathbf{x}_n \in \mathrm{R}^m$, which is the only piece of data that we actually observe. As usual for a mixture model, it is our goal to use the observed data to determine the cluster means and covariances, as well as the parameters of the Categorical distribution that selects the cluster centers.

Fortunately, this problem setup is perfectly suited for EM. We can apply the same machinery we've discussed throughout the chapter and used in the previous example.

1. First, we randomly initialize our parameters $\boldsymbol{\theta}, \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$.

2. [E-Step] Calculate the posterior distribution over $\mathbf{z}_n$ given by $\mathbf{q}_n$:

$$\begin{aligned}\mathbf{q}_n = \mathrm{E}[\mathbf{z}_n|\mathbf{x}_n] &= \begin{bmatrix} p(\mathbf{z}_n = C_1|\mathbf{x}_n; \theta_1, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \\ \vdots \\ p(\mathbf{z}_n = C_K|\mathbf{x}_n; \theta_K, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K) \end{bmatrix} \\ &\propto \begin{bmatrix} \theta_1\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \\ \vdots \\ \theta_K\mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K) \end{bmatrix}\end{aligned}$$

This is the current expectation for our latent variables $\mathbf{z}_n$ given our data $\mathbf{x}_n$ and the current setting of our model parameters $\boldsymbol{\theta}, \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$.

3. [M-Step] Using our values of $\mathbf{q}_n$, calculate the expected complete-data log likelihood, and then use that term to optimize our model parameters:

$$\begin{aligned}\mathbb{E}_{\mathbf{q}_n}[\log p(\mathbf{X}, \mathbf{Z})] &= \mathbb{E}_{\mathbf{q}_n}\left[\sum_{n=1}^{N} \ln(p(\mathbf{x}_n, \mathbf{z}_n; \boldsymbol{\theta}, \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K))\right] \\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} q_{n,k} \ln \theta_k + q_{n,k} \ln \mathcal{N}(\mathbf{x}_n; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\end{aligned}$$

We can then use this expected complete-data log likelihood to optimize our model parameters $\boldsymbol{\theta}, \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$ by computing the MLE as usual. Using a Lagrange multiplier to enforce $\sum_{k=1}^K \theta_k = 1$, we recover the update equations:

$$\begin{aligned}\theta_k^{(i+1)} &\leftarrow \frac{\sum_{n=1}^N q_{n,k}}{N} \\ \boldsymbol{\mu}_k^{(i+1)} &\leftarrow \frac{\sum_{n=1}^N q_{n,k}\mathbf{x}_n}{\sum_{n=1}^N q_{n,k}} \\ \boldsymbol{\Sigma}_k^{(i+1)} &\leftarrow \frac{\sum_{n=1}^N q_{n,k}(\mathbf{x}_n - \boldsymbol{\mu}_k^{(i+1)})(\mathbf{x}_n - \boldsymbol{\mu}_k^{(i+1)})^T}{\sum_{n=1}^N q_{n,k}}\end{aligned}$$

4. Return to step 2. Repeat until convergence.

Finally, it's worth comparing EM and K-Means clustering as applied to GMMs. First, as discussed previously, EM uses soft assignments of data points to clusters rather than hard assignments. Second, the standard K-Means algorithm does not estimate the covariance of each cluster. However, if we enforce as a part of our GMM setup that the covariance matrices of all the clusters are given by $\epsilon\mathbf{I}$, then as $\epsilon \to 0$, EM and K-Means will in fact produce the same results.

9.6 Admixture Models: Latent Dirichlet Allocation (LDA)

With a grasp on mixture models, it is not too difficult to understand admixture models. In a sentence: an admixture model is a mixture of mixture models. Latent Dirichlet Allocation (LDA) is a common form of admixture models, and it is sometimes also referred to as *topic modeling*, for reasons that will become apparent shortly. Describing LDA using an example will hopefully make the idea of an admixture model more concrete.

9.6.1 LDA for Topic Modeling

Consider the following data generating process for a set of text documents. We have a Dirichlet distribution $\boldsymbol{\theta} \sim Dir(\boldsymbol{\alpha})$ over the possible topics a document can take on.

⋆ If you haven't seen the Dirichlet before, it is a distribution over an n-dimensional vector whose components sum to 1. For example, a sample from a dirichlet distribution in 3-dimensions could produce a sample that is the vector

$$\begin{bmatrix}0.2\\0.5\\0.3\end{bmatrix}$$

.

We sample from that Dirichlet distribution to determine the mixture of topics $\boldsymbol{\theta}_n$ in our document D_n:

$$\boldsymbol{\theta}_n \sim Dir(\boldsymbol{\alpha})$$

Then, for each possible topic, we sample from a Dirichlet distribution to determine the mixture of words $\boldsymbol{\phi}_k$ in that topic:

$$\boldsymbol{\phi}_k \sim Dir(\boldsymbol{\beta})$$

Then, for each word $\mathbf{w}_{n,j}$ in the document D_n, we first sample from a Categorical parameterized by the topic mixture $\boldsymbol{\theta}_n$ to determine which topic that word will come from:

$$\mathbf{z}_{n,j} \sim Cat(\boldsymbol{\theta}_n)$$

Then, now that we have a topic given by $\mathbf{z}_{n,j}$ for this word $\mathbf{w}_{n,j}$, we sample from a Categorical parameterized by that topic's mixture over words given by $\boldsymbol{\phi}_{\mathbf{z}_{n,j}}$:

$$\mathbf{w}_{n,j} \sim Cat(\boldsymbol{\phi}_{\mathbf{z}_{n,j}})$$

Notice the mixture of mixtures at play here: we have a mixture model over the topics to produce each document in our corpus, and then for every word in a given document, we have a mixture over the topics to generate each individual word.

The indexing is particularly confusing because there are several layers of mixtures here, but to clarify: $n \in 1..N$ indexes each document D_n in our corpus, $k \in 1..K$ indexes each possible topic, and $j \in 1..J$ indexes each word $\mathbf{w}_{n,j}$ in document D_n, and $e \in 1..E$ indexes each word in our dictionary (note that $\mathbf{w}_{n,j} \in \mathbb{R}^E$).

$\boldsymbol{\theta}_n$ specifies the distribution over topics in document D_n, and $\boldsymbol{\alpha}$ is the hyperparameter for the distribution that produces $\boldsymbol{\theta}_n$. Similarly, $\boldsymbol{\phi}_k$ specifies the distribution over words for the k^{th} topic, and $\boldsymbol{\beta}$ is the hyperparameter for the distribution that produces $\boldsymbol{\phi}_k$.

9.6.2 Applying EM to LDA

Now that the problem setup and notation are taken care of, let's consider how we can apply EM to optimize the parameters $\boldsymbol{\theta}_n$ (the mixture over topics in a document) and $\boldsymbol{\phi}_k$ (the mixture over words for a topic). Note that we can simplify the problem slightly by considering $\boldsymbol{\theta}_n$ and $\boldsymbol{\phi}_k$ to be deterministic parameters for optimization (rather than random variables parameterized by $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$). Then, EM proceeds as follows:

1. First, we randomly initialize our parameters $\{\boldsymbol{\theta}_n\}_{n=1}^N, \{\boldsymbol{\phi}_k\}_{k=1}^K$.

2. [E-Step] Fix the topic distribution of the document given by $\boldsymbol{\theta}_n$ and the word distribution under a topic given by $\boldsymbol{\phi}_k$. Calculate the posterior distribution $\mathbf{q}_{n,j} = p(\mathbf{z}_{n,j}|\mathbf{w}_{n,j})$, and note that this is the distribution over the possible topics of a word:

$$\begin{aligned}\mathbf{q}_{n,j} = \mathrm{E}[\mathbf{z}_{n,j}|\mathbf{w}_{n,j}] &= \begin{bmatrix} p(\mathbf{z}_{n,j} = C_1|\mathbf{w}_{n,j}; \boldsymbol{\theta}_n, \boldsymbol{\phi}_1) \\ \vdots \\ p(\mathbf{z}_{n,j} = C_K|\mathbf{w}_{n,j}; \boldsymbol{\theta}_n, \boldsymbol{\phi}_K) \end{bmatrix} \\ &\propto \begin{bmatrix} p(\mathbf{w}_{n,j}|\mathbf{z}_{n,j} = C_1; \boldsymbol{\phi}_1)p(\mathbf{z}_{n,j} = C_1; \boldsymbol{\theta}_n) \\ \vdots \\ p(\mathbf{w}_{n,j}|\mathbf{z}_{n,j} = C_K; \boldsymbol{\phi}_K)p(\mathbf{z}_{n,j} = C_K; \boldsymbol{\theta}_n) \end{bmatrix} \\ &= \begin{bmatrix} \phi_{1,\mathbf{w}_{n,j}} \cdot \theta_{n,1} \\ \vdots \\ \phi_{K,\mathbf{w}_{n,j}} \cdot \theta_{n,K} \end{bmatrix}\end{aligned}$$

3. [M-Step] Using our values of $\mathbf{q}_n$, calculate the expected complete-data log likelihood (which marginalizes over the unknown hidden variables $\mathbf{z}_{n,j}$), and then use that expression to optimize our model parameters $\boldsymbol{\theta}_n$ and $\boldsymbol{\phi}_k$:

$$\begin{aligned}\mathbb{E}_{\mathbf{q}_n}[\log p(\mathbf{W}, \mathbf{Z})] &= \mathbb{E}_{\mathbf{q}_n}\left[\sum_{n=1}^{N}\sum_{j=1}^{J} \ln(p(\mathbf{w}_{n,j}, \mathbf{z}_{n,j}; \{\boldsymbol{\theta}_n\}_{n=1}^{N}, \{\boldsymbol{\phi}_k\}_{k=1}^{K}\right] \\ &= \sum_{n=1}^{N}\sum_{j=1}^{J}\sum_{k=1}^{K} q_{n,j,k} \ln \theta_{n,k} + q_{n,j,k} \ln \phi_{k,\mathbf{w}_{n,j}}\end{aligned}$$

We can then use this expected complete-data log likelihood to optimize our model parameters $\{\boldsymbol{\theta}_n\}_{n=1}^{N}, \{\boldsymbol{\phi}_k\}_{k=1}^{K}$ by computing the MLE as usual. Using Lagrange multipliers to enforce $\forall n \sum_{k=1}^{K} \theta_{n,k} = 1$ and $\forall k \sum_{e=1}^{E} \phi_{k,e} = 1$ (where e indexes each word in our dictionary), we recover the update equations:

$$\begin{aligned}\theta_{n,k}^{(i+1)} &\leftarrow \frac{\sum_{j=1}^{J} q_{n,j,k}}{J} \\ \phi_{k,d}^{(i+1)} &\leftarrow \frac{\sum_{n=1}^{N}\sum_{j=1}^{J} q_{n,j,k} w_{n,j,d}}{\sum_{n=1}^{N}\sum_{j=1}^{J} q_{n,j,k}}\end{aligned}$$

4. Return to step 2. Repeat until convergence.

The largest headache for applying the EM algorithm to LDA is keeping all of the indices in order, and this is the result of working with a mixture of mixtures. Once the bookkeeping is sorted out, the actual updates are straightforward.

9.7 Conclusion

Mixture models are one common way of handling data that we believe is generated through a combination of unobserved, latent variables. We've seen that training these models directly is intractable (due to the marginalization over the latent variables), and so we turned to a coordinate ascent based algorithm known as Expectation-Maximization to get around this difficulty. We then explored a couple of common mixture models, including a multinomial mixture, Gaussian Mixture Model, and an admixture model known as Latent Dirichlet Allocation. Mixture models are a subset of a broader range of models known as latent variable models, and the examples seen in this chapter are just a taste of the many different mixture models available to us. Furthermore, EM is just a single algorithm for optimizing these models. A good grasp on the fundamentals of mixture models and the EM algorithm will be useful background for expanding to more complicated, expressive latent variable models.

Chapter 10

Hidden Markov Models

Many of the techniques we've considered so far in this book have been motivated by the *types* of data we could expect to work with. For example, the supervised learning techniques (forms of regression, neural networks, support vector machines, etc.) were motivated by the fact that we had labelled training data. We ventured into clustering to group unlabelled data and discussed dimensionality reduction to handle overly high-dimensional data. In the previous chapter, we examined techniques for managing *incomplete* data with latent variable models. In this chapter we turn to a technique for handling temporal data.

10.1 Motivation

One major type of data we have not yet paid explicit attention to is *time series* data. Most of the information we record comes with some sort of a timestamp. For example, any time we take an action online, there is a high probability that the database storing that data also tracks it with a timestamp. Physical sensors in the real world always record timestamps because it would be very difficult to make sense of their information if it is not indexed by time. When we undergo medical exams, the results are recorded along with a timestamp. It's almost inconceivable at this point that we would record information without also keeping track of when that data was generated, or at the very least when we saved that data.

For these reasons, its interesting to develop models that are specialized to temporal data. Certainly, time encodes a lot of information that we take for granted about the physical and digital worlds. For example, if the sensors on a plane record the position of the plane at a specific point in time, we would expect the surrounding data points to be relatively similar, or at least move in a consistent direction. In a more general sense, we expect that time constrains other aspects of the data it is attached to in specific ways.

In this chapter, we will focus on one such model known as a **Hidden Markov Model** or **HMM**. At a high level, the goal of an HMM is to model the state of an entity over time, with the caveat that we never actually observe the state itself. Instead, we observe a data point $\mathbf{x}_t$ at each time step (often called an 'emission' or 'observation') that depends on the state $\mathbf{s}_t$. For example, we could model the position of a robot over time given a noisy estimation of the robot's current position at each time step. Furthermore, we will assume that one state $\mathbf{s}_t$ transitions to the next state $\mathbf{s}_{t+1}$ according to a probabilistic model. Graphically, an HMM looks like Figure 10.1, which encodes the relationships between emissions and hidden states. Here, there are n time steps in total.

We will probe HMMs in more detail over the course of the chapter, but for now let's consider

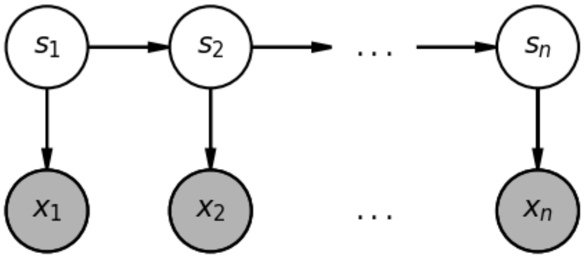


Figure 10.1: The directed graphical model for an HMM.

their high-level properties.

ML Framework Cube: Hidden Markov Models

First, HMMs handle discrete states, and for the purpose of this text, we will only consider discrete-valued emissions as well. Second, since a HMM does not have access to the states, these are hidden (or latent) and in this sense we can view training as "unsupervised." Finally, we assume a probabilistic relationship between hidden states and observed emissions as well as the transitions between hidden states.

Domain	***Training***	***Probabilistic***
Discrete	Unsupervised	Yes

⋆ Models that treat continuous state variables are commonly referred to as dynamical systems.

10.2 Applications

Unsurprisingly, there are many applications for models like HMMs that explicitly account for time and unobserved states, especially those that relate to the physical world. Examples include:

1. The position of a robot arm when its movements may be non-deterministic and sensor readings are noisy. [State= robot arm position; observation = sensor reading]

2. Speech recognition. [State = phoneme; observation = sound]

3. Analyzing sequences that occur in the natural world, such as DNA [State = codon, a genetic code in a DNA molecule; observation= one of the four bases, i.e., A, C, T, or G]

10.3 HMM Data, Model, and Parameterization

As explained above, HMMs model the state of an entity over time given some noisy observations, as shown in Figure 10.1.

10.3.1 HMM Data

The data for a HMM consists of the sequence of one-hot encoded emissions $\mathbf{x}_1, ..., \mathbf{x}_n$, for n total time steps. There corresponding states, $\mathbf{s}_1, ..., \mathbf{s}_n$, are latent and unobserved.

Each state corresponds to one of K possible options, i.e., with one-hot coding, $\mathbf{s}_t \in \{0, 1\}^K$. Each emission corresponds to one of M possible options, with one-hot coding, $\mathbf{x}_t \in \{0, 1\}^M$.

⋆ In general the observed emissions don't have to be discrete, but for the sake of being explicit, we present the discrete interpretation here.

A data set has N data points, meaning N sequences where each sequence is composed of n emissions (in general they can be of different lengths, we assume same length for simplicity). To summarize:

- A data set consists of N sequences.
- Each sequence is composed of n observed emissions $\mathbf{x}_1, ..., \mathbf{x}_n$.
- Each emission $\mathbf{x}_t$ takes on one of M possible values.
- Each hidden state $\mathbf{s}_t$ take on one of K possible values.

10.3.2 HMM Model Assumptions

In modeling the joint distribution over hidden states and observed emissions

$$p(\mathbf{s}_1, ..., \mathbf{s}_n, \mathbf{x}_1, ..., \mathbf{x}_n) = p(\mathbf{s}_1, ..., \mathbf{s}_n)p(\mathbf{x}_1, ..., \mathbf{x}_n|\mathbf{s}_1, ..., \mathbf{s}_n), \tag{10.1}$$

the HMM model makes the following two assumptions:

1. State $\mathbf{s}_{t+1}$ depends only on the previous state $\mathbf{s}_t$:

$$p(\mathbf{s}_{t+1}|\mathbf{s}_1, ..., \mathbf{s}_t, \mathbf{x}_1, ..., \mathbf{x}_t) = p(\mathbf{s}_{t+1}|\mathbf{s}_t)$$

This is the *Markov Property*, and it means that given knowledge of the state at the previous time step, we can ignore all other earlier states and emissions. Here, we also assume the transition is stationary, so that the transition model doesn't depend on time.

2. At each time step t, the observed emission $\mathbf{x}_t$ depends only on the current state $\mathbf{s}_t$:

$$p(\mathbf{x}_t|\mathbf{s}_1, ..., \mathbf{s}_t, \mathbf{x}_1, ..., \mathbf{x}_{t-1}) = p(\mathbf{x}_t|\mathbf{s}_t)$$

⋆ The Markovian assumption for transitions, as well as the fact that we don't observe the true states, gives rise to the *Hidden Markov Model* name.

These two assumptions allow us to factorize the large joint distribution given by Equation 10.1 as follows:

$$p(\mathbf{s}_1, ..., \mathbf{s}_n)p(\mathbf{x}_1, ..., \mathbf{x}_n|\mathbf{s}_1, ..., \mathbf{s}_n) = p(\mathbf{s}_1)\prod_{t=1}^{n-1} p(\mathbf{s}_{t+1}|\mathbf{s}_t)\prod_{t=1}^{n} p(\mathbf{x}_t|\mathbf{s}_t) \tag{10.2}$$

This factorization will prove important for making HMM training and inference tractable.

10.3.3 HMM Parameterization

Now that we understand the form of the data as well as the modeling assumptions made by a HMM, we can specify the model parameterization explicitly. Referencing the factorized joint distribution from Equation 10.2, we will need three distinct sets of parameters.

1. Parameters for the prior over the initial hidden state $p(\mathbf{s}_1)$. This will be denoted $\boldsymbol{\theta} \in [0,1]^K$, with $\sum_k \theta_k = 1$, such that:

   $$p(\mathbf{s}_1 = k) = \theta_k.$$

2. Parameters for the transition probabilities between states $p(\mathbf{s}_{t+1}|\mathbf{s}_t)$. This will be denoted $\mathbf{T} \in [0,1]^{K \times K}$, with $\sum_j T_{i,j} = 1$ for each i, such that:

   $$p(\mathbf{s}_{t+1} = j | \mathbf{s}_t = i) = T_{i,j},$$

   where $T_{i,j}$ is the probability of transitioning from state i to state j.

3. Parameters for the conditional probability of the emission, $p(\mathbf{x}_t|\mathbf{s}_t)$, given the state. This will be denoted $\boldsymbol{\pi} \in [0,1]^{K \times M}$, with $\sum_m \pi_{k,m} = 1$ for each k, such that:

   $$p(\mathbf{x}_t = m | \mathbf{s} = k) = \pi_{k,m}.$$

   For each state, there is a distribution on possible emissions.

In sum, we have three sets of parameters $\boldsymbol{\theta} \in [0,1]^K$, $\mathbf{T} \in [0,1]^{K \times K}$, and $\boldsymbol{\pi} \in [0,1]^{K \times M}$ that we need to learn from our data set. Then, using a trained model, we will be able to perform several types of inference over our hidden states, as detailed next.

10.4 Inference in HMMs

We will see how to estimate the parameters of a HMM ("training") in Section 10.5. For now we want to understand how to do efficient inference in HMMs. Given parameters $\boldsymbol{\theta}, \mathbf{T}$, and $\boldsymbol{\pi}$, and given a sequence of observations, $\mathbf{x}_1, ..., \mathbf{x}_n$ (or $\mathbf{x}_1, ..., \mathbf{x}_t$) there are various queries we may like to perform:

- "p(seq)" $p(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ (what is the distribution on sequences of emissions?)
- Prediction $p(\mathbf{x}_{t+1}|\mathbf{x}_1, \ldots, \mathbf{x}_t)$ (what is the prediction of the next emission given what is known so far?)
- Smoothing $p(\mathbf{s}_t|\mathbf{x}_1, \ldots, \mathbf{x}_n)$ (after the fact, what do we predict for some earlier state?)
- Transition $p(\mathbf{s}_t, \mathbf{s}_{t+1}|\mathbf{x}_1, \ldots, \mathbf{x}_n)$ (after the fact, what do we predict for the joint distribution on some pair of temporally adjacent states?)
- Filtering $p(\mathbf{s}_t|\mathbf{x}_1, \ldots, \mathbf{x}_t)$ (what is the prediction, in real-time, of the current state?)
- Best path $\max p(\mathbf{s}_1, \ldots, \mathbf{s}_n|\mathbf{x}_1, \ldots, \mathbf{x}_n)$ (after the fact, what is the most likely sequence of states?)

For just one example, let's consider smoothing $p(\mathbf{s}_t|\mathbf{x}_1, ..., \mathbf{x}_n)$. To compute this would require marginalizing over all the unobserved states other than $\mathbf{s}_t$, as follows:

$$\begin{aligned}
p(\mathbf{s}_t|\mathbf{x}_1, ..., \mathbf{x}_n) &\propto p(\mathbf{s}_t, \mathbf{x}_1, ..., \mathbf{x}_n) \\
&= \sum_{s_1,\ldots,s_{t-1},s_{t+1},\ldots,s_n} p(\mathbf{s}_1, \ldots, \mathbf{s}_n, \mathbf{x}_1, \ldots, \mathbf{x}_n) \\
&= \sum_{s_1,\ldots,s_{t-1},s_{t+1},\ldots,s_n} p(\mathbf{s}_1) \prod_{t=1}^{n-1} p(\mathbf{s}_{t+1}|\mathbf{s}_t) \prod_{t=1}^{n} p(\mathbf{x}_t|\mathbf{s}_t).
\end{aligned}$$

Without making use of variable elimination, this requires summing over all possible states other that t, which is very costly. Moreover, suppose we then query this for another state. We'd need to sum again over all the states except for this new state, which duplicates a lot of work. Rather than performing these summations over and over again, we can instead "memoize" (or reuse) these kinds of summations using the Forward-Backward algorithm. This algorithm also makes uses of variable elimination to improve the efficiency of inference.

10.4.1 The Forward-Backward Algorithm

The Forward-Backward algorithm uses variable elimination methods to compute two sets of quantities, that we refer to as the "alpha" and "beta" values. The algorithm makes elegant use of dynamic programming. It can be viewed as a preliminary inference step such that the alpha and beta values can then be used for all inference tasks of interest as well as within EM for training a HMM model.

The Forward-Backward algorithm is also an example of a *message-passing* scheme, which means we can conceptualize it is passing aound compact messages along edges of the graphical model that corresponds to a HMM. The algorithm passes messages forwards and backwards through 'time', meaning up and down the chain shown in the graphical model representation in Figure 10.1. The forward messages are defined at each state as $\alpha_t(\mathbf{s}_t)$, while the backward messages are defined at each state as $\beta_t(\mathbf{s}_t)$. Let's define these α and β values explicitly.

The α_t's represent the joint probability of all our observed emissions from time $1, ..., t$ as well as the state at time t:

$$\alpha_t(\mathbf{s}_t) = p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) \tag{10.3}$$

Graphically, the α_t's are capturing the portion of the HMM shown in Figure 10.2.

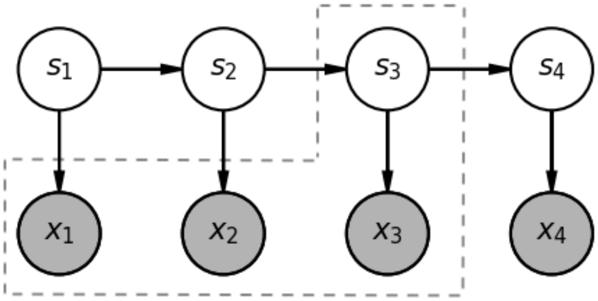


Figure 10.2: α_t's capture the joint probability for the boxed portion; shown for $\alpha_3(\mathbf{s}_3)$

We can factorize this joint probability using what we know about the conditional independence properties of HMMs as follows:

$$
\begin{aligned}
\alpha_t(\mathbf{s}_t) &= p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) \\
&= p(\mathbf{x}_t|\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{s}_t)p(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{s}_t) \\
&= p(\mathbf{x}_t|\mathbf{s}_t) \sum_{s_{t-1}} p(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{s}_{t-1}, \mathbf{s}_t) \qquad (10.4) \\
&= p(\mathbf{x}_t|\mathbf{s}_t) \sum_{s_{t-1}} p(\mathbf{s}_t|\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{s}_{t-1})p(\mathbf{x}_1, \dots, \mathbf{x}_{t-1}, \mathbf{s}_{t-1}) \qquad (10.5) \\
&= p(\mathbf{x}_t|\mathbf{s}_t) \sum_{s_{t-1}} p(\mathbf{s}_t|\mathbf{s}_{t-1})\alpha_{t-1}(\mathbf{s}_{t-1}) \qquad (10.6)
\end{aligned}
$$

The first term in Equation (10.4) follows from the Markov property, and for the second term we've expressed this joint probability by explicitly introducing $\mathbf{s}_{t-1}$ and marginalizing out over this variable. Equation 10.6 follows from the Markov property, and by substituting for the definition of the alpha value.

Notice that our expression for $\alpha_t(\mathbf{s}_t)$ includes the expression for $\alpha_{t-1}(\mathbf{s}_{t-1})$, which is the α from the previous time step. This means we can define our messages *recursively.* After we've computed the α's at one time step, we *pass* them forwards along the chain and use them in the computation of alpha values for the next time step. In other words, we compute the α values for period 1, then pass that message along to compute the α values in period 2, and so forth until we reah the end of the chain and have all the α's in hand.

$\star$ These α values will be useful for inference and will also be used during training a HMM via EM (in the E-step)

At this point, we've handled the forward messages, which send information from the beginning to the end of the chain. We also send information from the end of the chain back to the beginning, which constitutes the backwards portion. This is where we will compute our β values. The β_t's represent the joint probability over all the observed emissions from time $t+1, ..., n$ conditioned on the state at time t:

$$\beta_t(\mathbf{s}_t) = p(\mathbf{x}_{t+1}, ..., \mathbf{x}_n|\mathbf{s}_t) \qquad (10.7)$$

Graphically, this means that the β_t's are capturing the portion of the HMM shown in Figure 10.3.

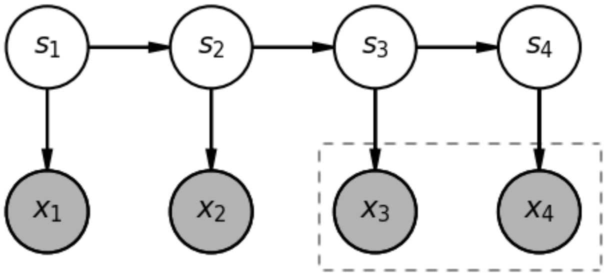


Figure 10.3: β_t's capture the joint probability for the boxed portion of the HMM. Shown here for $\beta_2(\mathbf{s}_2)$

We can factorize Equation 10.7 in a similar way to how we factorized the distribution described by the α's:

$$\begin{aligned}\beta_t(\mathbf{s}_t) &= p(\mathbf{x}_{t+1}, ..., \mathbf{x}_n | \mathbf{s}_t) \\ &= \sum_{\mathbf{s}_{t+1}} p(\mathbf{x}_{t+1}, ..., \mathbf{x}_n, \mathbf{s}_{t+1} | \mathbf{s}_t) && (10.8) \\ &= \sum_{\mathbf{s}_{t+1}} p(\mathbf{s}_{t+1}|\mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{s}_t, \mathbf{s}_{t+1}) p(\mathbf{x}_{t+2}, ..., \mathbf{x}_n | \mathbf{x}_{t+1}, \mathbf{s}_t, \mathbf{s}_{t+1}) && (10.9) \\ &= \sum_{\mathbf{s}_{t+1}} p(\mathbf{s}_{t+1}|\mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{s}_{t+1}) p(\mathbf{x}_{t+2}, ..., \mathbf{x}_n | \mathbf{s}_{t+1}) && (10.10) \\ &= \sum_{\mathbf{s}_{t+1}} p(\mathbf{s}_{t+1}|\mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{s}_{t+1}) \beta_{t+1}(\mathbf{s}_{t+1}). && (10.11)\end{aligned}$$

Here, Equation (10.8) introduces $\mathbf{s}_{t+1}$ and marginalizes out over this variable. Equation (10.9) is the product rule. Equation (10.10) makes use of the Markov property in two places. Equation (10.11) substitutes in for the beta values.

Again, as we saw with our calculation of the α's, we have our β's defined recursively. Once again, this means that we can propagate messages and compute the beta values efficiently. In this case, we start at the end of the chain, and compute our β's for each state by passing messages back toward the front.

To summarize, the Forward-Backward algorithm calculates the α and β values as follows:

$$\alpha_t(\mathbf{s}_t) = \begin{cases} p(\mathbf{x}_t|\mathbf{s}_t) \sum_{\mathbf{s}_{t-1}} p(\mathbf{s}_t|\mathbf{s}_{t-1}) \alpha_{t-1}(\mathbf{s}_{t-1}) & 1 < t \leq n \\ p(\mathbf{x}_1|\mathbf{s}_1) p(\mathbf{s}_1) & \text{otherwise} \end{cases}$$

$$\beta_t(\mathbf{s}_t) = \begin{cases} \sum_{\mathbf{s}_{t+1}} p(\mathbf{s}_{t+1}|\mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{s}_{t+1}) \beta_{t+1}(\mathbf{s}_{t+1}) & 1 \leq t < n \\ 1 & \text{otherwise} \end{cases}$$

⋆ Notice that the base case for the β's is 1. This is a quirk of our indexing, and it ensures we have valid messages when we pass messages back from the final state $\mathbf{s}_n$.

10.4.2 Using α's and β's for Training and Inference

Now that we know how to compute these α and β values, let's see how to use them for inference. Consider the product of the α and β value at a specific time t:

$$\alpha_t(\mathbf{s}_t)\beta_t(\mathbf{s}_t) = p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) p(\mathbf{x}_{t+1}, ..., \mathbf{x}_n | \mathbf{s}_t) = p(\mathbf{x}_1, ..., \mathbf{x}_n, \mathbf{s}_t).$$

This is the joint distribution over all emissions and the state at time t. Using this as a building block, this can support many kinds of inference.

p(Seq)

For example, we might like to evaluate the joint distribution over the emissions.

$$p(\mathbf{x}_1, ..., \mathbf{x}_n) = \sum_{\mathbf{s}_t} p(\mathbf{x}_1, ..., \mathbf{x}_n, \mathbf{s}_t) = \sum_{\mathbf{s}_t} \alpha_t(\mathbf{s}_t)\beta_t(\mathbf{s}_t) \tag{10.12}$$

where we can sum over the possible state values. This calculation be defined for any state $\mathbf{s}_t$.

Prediction

Another common task is to predict the value of the next emission given the previous emissions.

$$p(\mathbf{x}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_t)$$

To compute this we can sum over state $\mathbf{s}_t$ and the next state $\mathbf{s}_{t+1}$ as follows:

$$\begin{aligned}
p(\mathbf{x}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_t) &\propto p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{x}_{t+1}) \\
&= \sum_{\mathbf{s}_t} \sum_{\mathbf{s}_{t+1}} p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{x}_{t+1}, \mathbf{s}_t, \mathbf{s}_{t+1}) \qquad (10.13) \\
&= \sum_{\mathbf{s}_t} \sum_{\mathbf{s}_{t+1}} p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t, \mathbf{s}_{t+1}) \qquad (10.14) \\
&= \sum_{\mathbf{s}_t} \sum_{\mathbf{s}_{t+1}} \alpha_t(\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{s}_{t+1}). \qquad (10.15)
\end{aligned}$$

Here, Equation (10.13) follows by introducing states $\mathbf{s}_t$ and $\mathbf{s}_{t+1}$ and marginalizing out over them. Equation (10.14) follows from the product rule, and Equation (10.15) by using the Markov property in two places and substituting for $\alpha_t(\mathbf{s}_t)$.

Smoothing

Smoothing is the problem of predicting the state at time t given all the observed emissions. We can think about this as updating the beliefs that we would have had in real-time, given emissions up to and including t, given *all* observed evidence up to period n. Hence the phrasing "smoothing." For this, we have

$$p(\mathbf{s}_t|\mathbf{x}_1, ..., \mathbf{x}_n) \propto p(\mathbf{x}_1, ..., \mathbf{x}_n, \mathbf{s}_t) = \alpha_t(\mathbf{s}_t)\beta_t(\mathbf{s}_t). \qquad (10.16)$$

Transition

Finally, we may wish to understand the joint distribution on states $\mathbf{s}_t$ and $\mathbf{s}_{t+1}$ given all the observed evidence.

$$\begin{aligned}
p(\mathbf{s}_t, \mathbf{s}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_n) &\propto p(\mathbf{s}_t, \mathbf{s}_{t+1}, \mathbf{x}_1, ..., \mathbf{x}_n) \\
&= p(\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{x}_1, ..., \mathbf{x}_t, \mathbf{s}_t, \mathbf{s}_{t+1}) \\
&\qquad p(\mathbf{x}_{t+2}, ..., \mathbf{x}_n|\mathbf{x}_1, ..., \mathbf{x}_{t+1}, \mathbf{s}_t, \mathbf{s}_{t+1}) \qquad (10.17) \\
&= \alpha_t(\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t) p(\mathbf{x}_{t+1}|\mathbf{s}_{t+1}) \beta_{t+1}(\mathbf{s}_{t+1}). \qquad (10.18)
\end{aligned}$$

Here, Equation (10.17) follows from the product rule, and (10.18) by substituting for $\alpha_t(\mathbf{s}_t)$, applying the Markov property three times, and substituting for $\beta_{t+1}(\mathbf{s}_{t+1})$.

Filtering

For filtering, we have

$$p(\mathbf{s}_t|\mathbf{x}_1, ..., \mathbf{x}_t) \propto p(\mathbf{s}_t, \mathbf{x}_1, ..., \mathbf{x}_t) = \alpha_t(\mathbf{s}_t). \qquad (10.19)$$

Best path

For the best path problem, we want to solve

$$\arg\max_{\mathbf{s}_1,\ldots,\mathbf{s}_n} p(\mathbf{s}|\mathbf{x}) = \arg\max_{\mathbf{s}_1,\ldots,\mathbf{s}_n} p(\mathbf{s}_1,\ldots,\mathbf{s}_n,\mathbf{x}_1,\ldots,\mathbf{x}_n)$$

This sometimes referred to as the "decoding" (or explanation) problem. For this, we can define the following function:

$$\gamma_t(\mathbf{s}_t) = \max_{\mathbf{s}_1,\ldots,\mathbf{s}_{t-1}} p(\mathbf{s}_1,\ldots,\mathbf{s}_t,\mathbf{x}_1,\ldots,\mathbf{x}_t). \tag{10.20}$$

This is the likelihood of $\mathbf{x}_1,\ldots,\mathbf{x}_t$, if the current state is $\mathbf{s}_t$, and under the best explanation so far. Recall that the recurrence for alpha is as follows:

$$\forall \mathbf{s}_t: \quad \alpha_t(\mathbf{s}_t) = \begin{cases} p(\mathbf{x}_t|\mathbf{s}_t)\sum_{\mathbf{s}_{t-1}} p(\mathbf{s}_t|\mathbf{s}_{t-1})\alpha_{t-1}(\mathbf{s}_{t-1}) & \text{if } 1 < t \leq n \\ p(\mathbf{x}_1|\mathbf{s}_1)p(\mathbf{s}_1) & \text{o.w.} \end{cases}$$

Analogously, the recurrence for this γ-value can be shown to be:

$$\forall \mathbf{s}_t: \quad \gamma_t(\mathbf{s}_t) = \begin{cases} p(\mathbf{x}_t|\mathbf{s}_t)\max_{\mathbf{s}_{t-1}} p(\mathbf{s}_t|\mathbf{s}_{t-1})\gamma_{t-1}(\mathbf{s}_{t-1}) & \text{if } 1 < t \leq n \\ p(\mathbf{x}_1|\mathbf{s}_1)p(\mathbf{s}_1) & \text{o.w.} \end{cases} \tag{10.21}$$

To be able to find the optimal sequence, we also store, for each $\mathbf{s}_t$, the best choice of $\mathbf{s}_{t-1}$:

$$z_t^*(\mathbf{s}_t) = \arg\max_{\mathbf{s}_{t-1}} \left[p(\mathbf{s}_t|\mathbf{s}_{t-1})\gamma_{t-1}(\mathbf{s}_{t-1})\right] \tag{10.22}$$

This recursive procedure is known as the Viterbi algorithm and provides an efficient way to infer the "best path" through states given a sequence of observations.

10.5 Using EM to Train a HMM

Let's now turn to using EM to train a HMM. Recall the motivation for the Expectation-Maximization algorithm from the previous chapter: we had parameters we wished to optimize, but the presence of unobserved variables made direct optimization of those parameters intractable. We're faced with a similar problem in the context of HMMs.

Given a data set of observed emissions $\{\mathbf{x}^i\}_{i=1}^N$ where each data point $\mathbf{x}^i$ represents the sequence $(\mathbf{x}_1^i, ..., \mathbf{x}_n^i)$, our goal is to estimate the parameters $\boldsymbol{\theta}, \mathbf{T}$, and $\boldsymbol{\pi}$.

With knowledge of the hidden states, this would be a relatively straightforward problem of MLE for the parameters. However, the states are latent variables, and for this reason we use the EM algorithm.

This amounts to computing the probability on hidden states in the E-step, and then based on these probabilities, we update our parameters by maximizing the expected complete-data likelihood in the M-step. As usual, we perform these E and M steps iteratively until convergence. We will make use of the Forward-Backward algorithm for the inference on probabilities of the latent states in the E-Step.

We initialize the parameters in EM arbitrarily.

10.5.1 E-Step

For the E-Step, we take the parameters $\boldsymbol{\theta}, \mathbf{T}, \boldsymbol{\pi}$ as fixed. For each data point $\mathbf{x}^i$, we run Forward-Backward with these parameters to get the alpha and beta values for this data point.

The hidden variables are the states $\mathbf{s}_1, ..., \mathbf{s}_n$. For each data point $\mathbf{x}^i = (\mathbf{x}^i_1, ..., \mathbf{x}^i_n)$, we are interested in computing the values $\mathbf{q}^i \in [0,1]^{n \times K}$, for K possible hidden state values. This represents the predicted probability of each hidden state value for each time period, for this In particular, we have

$$q^i_{t,k} = p(\mathbf{s}^i_t = k \,|\, \mathbf{x}^i_1, ..., \mathbf{x}^i_n). \tag{10.23}$$

This is the probability that state $\mathbf{s}^i_t$ takes on value k given the data point $\mathbf{x}^i$. This is the smoothing operation described in the previous section and we can use Equation 10.16 to compute our $\mathbf{q}^i$ values.

Ordinarily, we'd be done with the E-step after computing the marginal probability of each latent variable. But in this case we will also want to estimate the transition probabilities between hidden states, i.e., parameter matrix $\mathbf{T}$. For this, we also need to calculate the joint distribution between temporally-adjacent pairs of latent variables. For data point $\mathbf{x}^i$, and for periods t and $t+1$, we denote this as $\mathbf{Q}^i_{t,t+1} \in [0,1]^{K \times K}$, where the entries in this matrix sum to 1. This represents the distribution on pairs of states in periods t and $t+1$ for this data point. We write $\mathbf{Q}^i$ to denote the corresponding values for all pairs of time periods.

To see how the entries are calculated, we can use $Q^i_{t,t+1,k,\ell}$ to denote the transition from state k at time step t to state ℓ at time step $t+1$,

$$Q^i_{t,t+1,k,\ell} = p(\mathbf{s}^i_t = k, \mathbf{s}^i_{t+1} = \ell \,|\, \mathbf{x}^i_1, ..., \mathbf{x}^i_n). \tag{10.24}$$

This is exactly the transition inference problem that we described in the previous section. Because of this, we can directly use our α and β values in the transition operation, as given by Equation (10.18).

With our $\mathbf{q}^i$ and $\mathbf{Q}^i$ values for each data point, we are ready to move on to the maximization step.

10.5.2 M-Step

We now solve for the expected complete-data log likelihood problem, making use of the $\mathbf{q}^i$ and $\mathbf{Q}^i$ values from the E-step.

Given knowledge of states, the complete-data likelihood for one data point with observations $\mathbf{x}$ and states $\mathbf{s}$ is

$$p(\mathbf{x}, \mathbf{s}) = p(\mathbf{s}_1; \boldsymbol{\theta}) \prod_{t=1}^{n-1} p(\mathbf{s}_{t+1} | \mathbf{s}_t; \mathbf{T}) \prod_{t=1}^{n} p(\mathbf{x}_t | \mathbf{s}_t; \boldsymbol{\pi}).$$

With one-hot coding of $\mathbf{x}_t$ and $\mathbf{s}_t$, and taking the log, this is

$$\ln[p(\mathbf{x}, \mathbf{s})] = \sum_{k=1}^{K} s_{1k} \ln \theta_k + \sum_{t=1}^{n-1} \sum_{k=1}^{K} \sum_{\ell=1}^{K} s_{t,k} s_{t+1,\ell} \ln T_{k,\ell} + \sum_{t=1}^{n} \sum_{k=1}^{K} s_{t,k} \sum_{m=1}^{M} x_{t,m} \ln(\pi_{k,m}).$$

From this, we would be able to solve for the MLE for the parameters for the complete-data log likelihood.

Now, the states are latent variables, and we need to work with the *expected complete-data log likelihood*, which for a single data point $\mathbf{x}^i$, is

$$\mathbb{E}_{\mathbf{s}^i}[\ln(p(\mathbf{x}^i, \mathbf{s}^i))] = \sum_{k=1}^{K} q^i_{1k} \ln \theta_k + \sum_{t=1}^{n-1} \sum_{k=1}^{K} \sum_{\ell=1}^{K} Q^i_{t,t+1,k,\ell} \ln T_{k,l} + \sum_{t=1}^{n} \sum_{k=1}^{K} q^i_{t,k} \sum_{m=1}^{M} x^i_{t,m} \ln \pi_{k,m}. \tag{10.25}$$

Applying the appropriate Lagrange multipliers, and maximizing with respect to each of the parameters of interest, we can show that we obtain the following update equations for each of the parameters (for N data points):

$$\theta_k = \frac{\sum_{i=1}^N q^i_{1,k}}{N}, \quad \text{for all states } k \tag{10.26}$$

$$T_{k,l} = \frac{\sum_{i=1}^N \sum_{t=1}^{n-1} Q^i_{t,t+1,k,l}}{\sum_{i=1}^N \sum_{t=1}^{n-1} q^i_{t,k}}, \quad \text{for all states } k, \ell \tag{10.27}$$

$$\pi_{k,m} = \frac{\sum_{i=1}^N \sum_{t=1}^{n} q^i_{t,k} x^i_{t,m}}{\sum_{i=1}^N \sum_{t=1}^{n} q^i_{t,k}}, \quad \text{for all states } k\text{, observations } m \tag{10.28}$$

After updating our parameter matrices $\boldsymbol{\theta}, \mathbf{T}$, and $\boldsymbol{\pi}$, we switch back to the E-step, continuing in this way until convergence. As with other uses of EM, it provides only a local optimum and it can be useful to try a few random restarts.

10.6 Conclusion

The Hidden Markov Model is a type of latent variable model motivated by the combination of time series and discrete observations and states. We relied on the Expectation-Maximization algorithm to train a HMM, and developed the Forward-Backward algorithm to make both inference and training (the E-step) computationally efficient. Many of the ideas developed in this chapter will offer good intuition for how to develop learning and inference methods for dynamical systems and other time series models.

Chapter 11

Markov Decision Processes

In the previous chapter we learned about Hidden Markov Models (HMMs) in which we modeled our underlying environment as being a Markov chain where each state was hidden, but produced certain observations that we could use to infer the state. The process of learning in this setting included finding the distribution over initial hidden states, the transition probabilities between states, and the probabilities of each state producing each measurement. The movement of the underlying Markov chain from one state to the next was a completely autonomous process and questions of interest focused mainly on inference.

Markov Decision Processes (MDPs) introduce somewhat of a paradigm shift. Similar to HMMs, in the MDP setting we model our underlying environment as a Markov chain. However, our environment doesn't transition from state to state autonomously like in an HMM, but rather requires that an *agent* in the environment takes some *action*. Thus, the probability of transitioning to some state at time $t+1$ depends on both the state *and* the action taken at time t. Furthermore, after taking some action the agent receives a *reward* which you can think of as characterizing the "goodness" of performing a certain action in a certain state. Thus, our overall objective in the MDP setting is different. Rather than trying to perform inference, we rather would like to find a sequence of actions that maximizes the agent's total reward.

While MDPs *can* have hidden states – these are called partially-observable MDPs (POMDPs) – for the purposes of this course we will only consider MDPs where the agent's state is known. If, like in HMMs, we do not have full knowledge of our environment and thus do not know transition probabilities between states or the rewards associated with each state and action we can try to find the optimal sequence of actions using *reinforcement learning* —a subfield of machine learning, distinct from supervised and unsupervised learning, that is characterized by problems in which an agent seeks to *explore* its environment, and simultaneously use that knowledge to perform actions that *exploit* its environment and obtain rewards. This chapter discusses how we find the optimal set of actions given that we *do* have full knowledge of the environment.

ML Framework Cube: Markov Decision Processes

Domain	***Training***	***Probabilistic***
Discrete or Continuous	?	Yes

Note that the question mark under the "training" category means that learning in MDPs is neither supervised nor unsupervised but rather falls under the umbrella of reinforcement learning - an entirely separate branch of ML.

11.1 Formal Definition of an MDP

In an MDP, we model our underlying environment as containing any one of a set of states $\mathcal{S}$. At each state, an agent can take any action from the set $\mathcal{A}$ which gives them a reward according to the function $r : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which we will often see written as $r(s, a)$. For the purposes of this class, we assume $\mathcal{S}$ and $\mathcal{A}$ are discrete and finite, i.e. contain finitely many different elements, and also assume that rewards are deterministic, i.e. $r(s, a)$ is fixed for any s and a.

Given a specified state and action (*state-action pair*), the environment will transition into a new state according to a transition function $p : \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$ which we will often seen written as $p(s'|s, a)$. That is, from our current state-action pair (s, a), for all $s' \in \mathcal{S}$, the environment will transition to the new state s' with probability $p(s'|s, a)$. If we fix the action taken, our model behaves like a Markov chain.

We let the action we take at each state be given by a policy $\pi : \mathcal{S} \to \mathcal{A}$ which we will often see written as $\pi(a|s)$. Our goal is to find some optimal policy π^* that *maximizes total rewards.* Interacting with the environment according to some policy produces a dataset

$$\mathcal{D} = \{s_0, a_0, r_0, s_1, a_1, r_1, \dots\}$$

where the subscripts indicate timesteps.

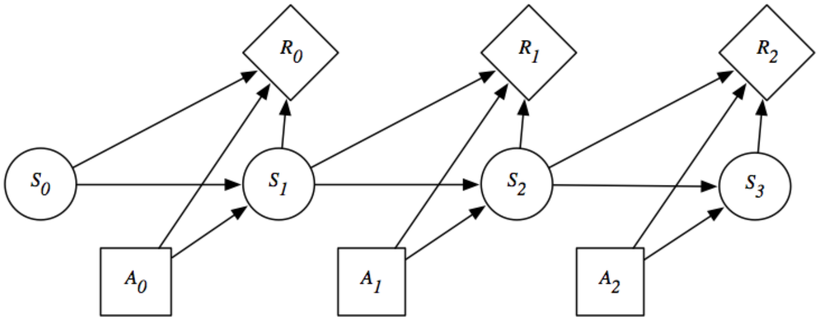


Figure 11.1: A graphical representation of an MDP.

Note that since our state transitions are modeled as a Markov chain, the *Markov assumption* holds:

$$p_t(s_{t+1}|s_1, \cdots s_t, a_1, \cdots a_t) = p_t(s_{t+1}|s_t, a_t)$$

In other words, the probability that the environment transitions to state s_{t+1} at time t only depends on the state the agent was in and the action the agent took at time t. Furthermore, we can assume that transition probabilities are *stationary*:

$$p_t(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_t, a_t)$$

In other words, the probability of transitioning from one state to another does not depend on the current timestep (note how we dropped the subscript t in the expression on the RHS).

We call the process of finding the optimal policy in an MDP given full knowledge of our environment *planning* (when we *don't* have prior knowledge about our environment but know that it behaves according to some MDP with unknown parameters we turn to Reinforcement Learning - see Chapter 12). Our approach to planning changes depending on whether there is a limit to the number of timesteps the agent may act for, which we call a *finite horizon*, or if they may act forever (or effectively forever), which we call an *infinite horizon.*

11.2 Finite Horizon Planning

In a finite horizon MDP, our objective function is as follows:

$$\max_{\pi} \mathbb{E}_{s\sim p}\left[\sum_{t=0}^{T} r_t|\pi\right]$$

Note that the only source of randomness in the system is the transition function.

We find the optimal policy for a finite horizon MDP using dynamic programming. To formalize this, let's define the optimal value function $V^*_{(t)}(s)$ to be the highest *value* achievable in state s (i.e. acting under the optimal policy π^*) with t timesteps remaining for the agent to act. Then, we know

$$V^*_{(1)}(s) = \max_a [r(s,a)]$$

since with only one timestep left to act, the best the agent can do is take the action that maximizes their immediate reward. We can define the optimal value function for $t > 1$ recursively as follows:

$$V^*_{(t+1)}(s) = \max_a [r(s,a) + \sum_{s'\in\mathcal{S}} p(s'|s,a)V^*_{(t)}(s')]$$

In other words, with more than one timestep to go we take the action that maximizes not only our immediate reward but also our *expected future reward*. This formulation makes use of the *principle of optimality* or the *Bellman consistency equation* which states that an optimal policy consists of taking an optimal first action and then following the optimal policy from the successive state (these concepts will be discussed further in section 11.3.1). Consequently, we have a different optimal policy at each timestep where

$$\pi^*_{(1)}(s) = \arg\max_a [r(s,a)]$$
$$\pi^*_{(t+1)}(s) = \arg\max_a [r(s,a) + \sum_{s'\in\mathcal{S}} p(s'|s,a)V^*_{(t)}(s')]$$

The computational complexity of this approach is $O(|S|^2|A|T)$ since for each state and each timestep we find the value-maximizing action which involves calculating the expected future reward requiring a summation over the value function for all states.

11.3 Infinite Horizon Planning

If the agent is able to act forever, a dynamic programming solution is no longer feasible as there is no base case (there will never only be 1 timestep remaining). Furthermore, we can't use the same objective function that we did when working with finite horizon MDPs since our expected reward could be infinite! Thus, we modify the objective function as follows:

$$\max_{\pi} \mathbb{E}_{s\sim p}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

where $\gamma \in [0,1]$ is called the *discount factor*. Multiplying the reward at time t in the infinite sum by γ^t makes the sum resolve to a finite number, since our rewards are bounded in $[0,1]$. As $\gamma \to 1$, rewards further in the future have more of an impact on the expected total reward and thus the optimal policy will be more "patient", acting in a way to achieve high rewards in the future rather than just maximizing short-term rewards. Conversely, as $\gamma \to 0$, rewards further in the future will have less of an impact on the expected total reward and thus the optimal policy will be less patient, preferring to maximize short-term rewards.

The *effective time horizon* of an infinite horizon MDP is the number of timesteps after which γ^t becomes so small that rewards after time t are negligible. Using the formula for the sum of a geometric series we find that this is approximately $\frac{1}{1-\gamma}$. This yields our first approach to planning in an infinite horizon MDP which is to convert it into a finite horizon MDP where $T = \frac{1}{1-\gamma}$. However, note that this fraction could be arbitrarily large and that the time complexity of the dynamic programming solution is *linear* with respect to T. This implores us to consider alternate solutions.

11.3.1 Value iteration

Define the value function of an infinite time horizon MDP under the policy π as the *expected discounted value* from policy π starting at state s:

$$V^\pi(s) = \mathbb{E}_{s\sim p}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)|s_0 = s, \pi\right]$$

Then, policy π is weakly better than policy π' if $V^\pi(s) \geq V^{\pi'}(s)\ \forall s$. The *optimal policy* π^* is that which is weakly better than all other policies.

Consequently, the optimal value function is the value function following policy π^*:

$$V^*(s) = \max_\pi V^\pi(s)$$

Bellman Consistency Equation and Bellman Optimality

The value iteration algorithm, which we will describe soon, is based on an alternate expression for the value function. The **Bellman consistency equation** gives us an alternate expression for $V^\pi(s)$:

$$V^\pi(s) = r(s, \pi(s)) + \gamma \mathbb{E}_{s'\sim p}[V^\pi(s)]$$

Proof: We begin by expanding our initial expression for $V^\pi(s)$:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_{s\sim p}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t)|s_0 = s, \pi\right] \\ &= r(s, \pi(s)) + \gamma \mathbb{E}_{s\sim p}\left[r(s_1, a_1) + \gamma r(s_2, a_2) + \cdots |s_0 = s, \pi\right] \end{aligned}$$

We now apply Adam's law (dropping the subscripts on the expectations to avoid notational clutter):

$$= r(s, \pi(s)) + \gamma \mathbb{E}[\mathbb{E}[r(s_1, a_1) + \gamma r(s_2, a_2) + \cdots |s_0 = s, s_1 = s', \pi]]$$

By the Markov property:

$$= r(s, \pi(s)) + \gamma \mathbb{E}[\mathbb{E}[r(s_1, a_1) + \gamma r(s_2, a_2) + \cdots |s_1 = s', \pi]]$$

By definition, this is equal to $r(s, \pi(s)) + \gamma \mathbb{E}_{s' \sim p}[V^{\pi}(s')]$. □

The **Bellman optimality** conditions are a set of two theorems that tell us important properties about the optimal value function V^*.

Theorem 1: V^* satisfies the following Bellman consistency equations:

$$V^*(s) = \max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')]] \quad \forall s$$

This theorem also tells us that if $\hat{\pi}(s) = \arg\max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')]]$ then $\hat{\pi}$ is the optimal policy.
Proof: Let $\hat{\pi}(s) = \arg\max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')]]$. To prove the claim it suffices to show that $V^*(s) \leq V^{\hat{\pi}}(s)$ $\forall s$. By the Bellman consistency equation we have:

$$\begin{aligned} V^*(s) &= r(s, \pi^*(s)) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')] \\ &\leq \max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')]] \\ &= r(s, \hat{\pi}(s)) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')] \end{aligned}$$

Proceeding recursively:

$$\begin{aligned} &\leq r(s, \hat{\pi}(s)) + \gamma \mathbb{E}_{s' \sim p}[r(s', \hat{\pi}(s')) + \gamma \mathbb{E}_{s'' \sim p}[V^*(s'')]] \\ &\dots \\ &\leq \mathbb{E}_{s,s',\dots \sim p}[r(s, \hat{\pi}(s)) + \gamma r(s', \hat{\pi}(s') + \cdots | \hat{\pi}] \\ &= V^{\hat{\pi}}(s) \end{aligned}$$

□

Theorem 2: For any value function V, if $V(s) = \max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V(s')]]$ $\forall s$ then $V = V^*$.
Proof: First, we define the maximal component distance between V ad V^* as follows:

$$||V - V^*||_\infty = \max_s |V(s) - V^*(s)|$$

For V which satisfies the Bellman optimality equations, if we could show that $||V - V^*||_\infty \leq \gamma ||V - V^*||_\infty$ then the proof would be complete since it would imply:

$$||V - V^*||_\infty \leq \gamma ||V - V^*||_\infty \leq \gamma^2 ||V - V^*||_\infty \leq \cdots \leq \lim_{k \to \infty} \gamma^k ||V - V^*||_\infty = 0$$

Thus we have:

$$\begin{aligned} |V(s) - V^*(s)| &= |max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V(s')]] - \max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V^*(s')]]| \\ &\leq \max_a |r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V(s')] - r(s,a) - \gamma \mathbb{E}_{s' \sim p}[V^*(s')]| \\ &= \gamma \max_a |\mathbb{E}_{s' \sim p}[V(s')] - \mathbb{E}_{s' \sim p}[V^*(s')]| \\ &\leq \max_a \mathbb{E}_{s' \sim p}[|V(s') - V^*(s')] \\ &\leq \gamma \max_{a,s'} |V(s') - V^*(s')| \\ &= \gamma \max_{s'} |V(s') - V^*(s')| \\ &= \gamma ||V - V^*||_\infty \end{aligned}$$

□

They key takeaways from the Bellman optimality conditions are twofold. Theorem 1 gives us a nice expression for the optimal value function that we will soon show allows us to *iteratively* improve an arbitrary value function towards optimality. It also tells us what the corresponding optimal policy is. Theorem 2 tells us that the optimal value function is *unique* so if we find *some* value function that satisfies the Bellman consistency equations given in Theorem 1 then it *must* be optimal.

Bellman Operator

In the previous section we mentioned that the formulation of the optimal value function we get from Theorem 1 will allow us to improve some arbitrary value function iteratively. In this section we will show why this is the case. Define the Bellman operator $B : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ to be a function that takes as input a value function and outputs another value function as follows:

$$B(V(s)) := \max_a [r(s,a) + \gamma \mathbb{E}_{s \sim p}[V(s')]]$$

By the Bellman optimality conditions we know that $B(V^*) = V^*$ and that this identity property doesn't hold for any other value function. The optimal value function V^* is thus the unique *fix-point* of the function B which means that passing V^* into B always returns V^*.

Furthermore, we know that B is a *contraction mapping* which means that $||B(V) - B(V')||_\infty \leq \gamma ||V - V'||_\infty$.
Proof:

$$\begin{aligned}
|B(V(s)) - B(V'(s))| &= |\max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V(s')] - \max_a [r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V'(s')]]] \\
&\leq \max_a |r(s,a) + \gamma \mathbb{E}_{s' \sim p}[V(s') - r(s,a) - \gamma \mathbb{E}_{s' \sim p} V'(s')]| \\
&= \gamma \max_a |\mathbb{E}_{s' \sim p}[V(s') - V'(s')]| \\
&\leq \gamma \max_a [\mathbb{E}_{s' \sim p}[|V(s') - V'(s')|]] \\
&\leq \gamma \max_{a,s} |V(s) - V'(s)| \\
&= \gamma ||V - V'||_\infty
\end{aligned}$$

□

Value Iteration Algorithm

From the previous section we learned that applying the Bellman operator to an arbitrary value function repeatedly will get it closer and closer to the optimal value function. Once we see that applying the Bellman operator no longer changes the input value function, we know we have found the optimal value function. This is the basis for the value iteration algorithm which works as follows:

1. Initialize $V(s) = 0 \ \forall s$

2. Repeat until convergence:
   $V'(s) \leftarrow \max_a [r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V(s')] \ \forall s$
   $V(s) \leftarrow V'(s) \ \forall s$

It is a theorem that value iteration converges to V^* asymptotically. We can also extract the optimal policy from the resulting value function asymptotically. Each iteration of value iteration takes $O(|S|^2|A|)$ time.

11.3.2 Policy Iteration

An alternate approach to finding the optimal policy in an infinite horizon MDP uses the same contraction properties of the Bellman operator but iterates on the *policy* directly rather than on the value function. The algorithm is as follows:

1. Initialize some arbitrary policy π
2. Repeat until convergence:
   (a) Policy evaluation: Calculate $V^\pi(s)\ \forall s$
   (b) Policy improvement:
   $\pi'(s) \leftarrow \arg\max_a [r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)V^\pi(s')]\ \forall s$
   $\pi(s) \leftarrow \pi'(s)\ \forall s$

Here, convergence means that the improvement step does not modify the policy from the previous iteration. It is a theorem that policy iteration converges to the optimal policy in a finite number of steps and also yields the optimal value function.

Policy Evaluation

In the previous section, we saw that one of the steps in the policy iteration algorithm was to calculate the value function for each state following a policy π. There are two ways to do this:

- **Exact policy evaluation:**
  We know that our value function must satisfy the Bellman consistency equations:
  $$V^\pi(s) = r(s,\pi(s)) + \gamma \sum_{s \in \mathcal{S}} p(s'|s,\pi(s))V^\pi(s') \quad \forall s$$
  This is a system of $|\mathcal{S}|$ linear equations that has a *unique solution.* Rearranging terms, representing functions as matrices and vectors, and replacing sums with matrix multiplication we get that:
  $$\mathbf{V}^\pi = (\mathbf{I} - \gamma \mathbf{P}^\pi)^{-1}\mathbf{R}^\pi$$
  where $\mathbf{V}^\pi$ is an $|\mathcal{S}| \times 1$ vector, $\mathbf{I}$ is the identity matrix, $\mathbf{P}^\pi$ is an $|\mathcal{S}| \times |\mathcal{S}|$ matrix where $\mathbf{P}^\pi_{s,s'} = p(s'|s,\pi(s)))$, and $\mathbf{R}^\pi$ is an $|S| \times 1$ vector where $\mathbf{R}^\pi_s = r(s,\pi(s))$. This subroutine runs in time $O(|\mathcal{S}|^3)$.
- **Iterative policy evaluation:**
  Rather than solving the system of equations described earlier exactly, we can instead do it iteratively. We perform the following two steps:
  1. Initialize $\mathbf{V}^0$ such that $||\mathbf{V}^0||_\infty \in \left[0, \frac{1}{1-\gamma}\right]$
  2. Repeat until convergence: $\mathbf{V}^{t+1} \leftarrow \mathbf{R} + \gamma \mathbf{P}^\pi \mathbf{V}^t$

  . This algorithm runs in time $\tilde{O}\left(\frac{|S|^2}{1-\gamma}\right)$.

Policy iteration requires fewer iterations than value iteration to solve for the optimal policy π^*, but requires more time per iteration: $O(|S|^2|A| + |S|^3)$ if we use exact policy evaluation.

Chapter 12

Reinforcement Learning

12.1 Motivation

In the last chapter, we discussed the *Markov Decision Process (MDP)*: a framework that models a learner's environment as a vector of states, actions, rewards, and transition probabilities. Given this model, we can solve for an optimal (reward-maximizing) policy using either value iteration or policy iteration. Sometimes, however, the learner doesn't have prior knowledge of the rewards they will get from each state or the probability distribution over states they could end up in after taking some action from their current state. Is it still possible to learn a policy that will maximize rewards? In this chapter, we will learn about **Reinforcement Learning (RL)** - a machine learning technique that addresses this problem.

ML Framework Cube: Reinforcement Learning

Domain	***Training***	***Probabilistic***
Discrete or Continuous	?	Yes

Note that we will only cover RL for discrete state spaces in this chapter. The question mark under the "training" category means that RL is neither supervised nor unsupervised.

12.2 General Approaches to RL

Imagine that you are vacationing in a town that has 20 restaurants. Suppose that *if* you had been able to try the food at every restaurant you would have had a definitive ranking of which was your favorite, second favorite, third favorite and so on. Your vacation is 10 days long and your budget allows you to eat at one restaurant per day. Your goal on your vacation is to have the best experience, which in this case means to eat at the restaurants you like the most. The problem is that you don't have enough time to figure out which restaurants you like the most since you don't have enough time to eat at each one. In this situation, you have two key considerations to balance:

1. Time spent **exploring**. If you spend the right amount of time exploring new restaurants, then there is a good chance that you will find one that you like a lot. However, spending too much time exploring will result in an average experience overall since some restaurants will be good and some will be bad.

2. Time spent **exploiting**. If you spend the right amount of time exploiting your knowledge of what restaurants you've liked the most from your exploration you will have a good experience. However, if you spend too long exploiting (and consequently less time exploring) you risk missing out on eating at all the restaurants that you could have liked *more*.

This is called the **exploration vs exploitation** tradeoff. The more time you spend exploring the less time you can spend exploiting, and vice versa. In order to have the best vacation experience, you need to find a balance between time spent exploring different restaurants and time spent exploiting what you have learned by eating at that restaurants that you liked the most thus far. A reinforcement learner's task is similar to the vacation restaurant task described above. They must balance time spent exploring the environment by taking actions and observing rewards and penalties, and time spent maximizing rewards based on what they know.

We now return to the problem described in the previous section: how do we learn a policy that will maximize rewards in an MDP where we know nothing about which rewards we will get from each state and which state we will end up in after taking an action? One approach is to try and *learn* the MDP. In other words, we can create a learner that will explore its environment for some amount of time in order to learn the rewards associated with each state and the transition probabilities associated with each action. Then, they can use an MDP planning algorithm like policy iteration or value iteration to come up with an optimal policy and maximize their rewards. This is called **model-based** learning. The main advantage of model-based learning is that it can inexpensively incorporate changes in the reward structure or transition function of the environment into the model. However, model-based learning is computationally expensive, and an incorrect model can yield a sub-optimal policy.

Conversely, **model-free** learning is a family of RL techniques where a learner tries to learn the optimal policy directly without modelling the environment. While this sort of learning is less adaptive to changes in the environment, it is far more computationally efficient than model-based learning.

12.3 Model-Free Learning

While there are many methods within the family of model-free learning, we will focus on three **value-based** methods - *SARSA* (section 12.3.1), *Q-learning* (section 12.3.1) and *Deep Q-learning* (section 12.3.2) - and one **policy-based** method - *Policy Learning* (section 12.3.3).

We will begin by discussing value-based methods. In this family of RL algorithms, the learner tries to calculate the expected reward they will receive from a state s upon taking action a. Formally, they are trying to learn a function that maps (s, a) to some value representing the expected reward. This is called the Q-function and is defined as follows for some policy π:

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s') \tag{12.1}$$

In words, the approximate expected reward (Q value) for taking action s from state a is the actual reward received from the environment by doing so in the current iteration plus the expectation taken over all reachable states of the highest value achievable starting at that state times the discount factor γ. The Q-function following the optimal policy is defined analogously:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^*(s') \tag{12.2}$$

Note, that $V^*(s') = max_{a'} Q^*(s', a')$ since the highest value achievable from state s' following policy $*$ is the Q value of taking the optimal action from s'. Substituting this in, we get the following *Bellman Equation*:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s}^{'} p(s'|s,a) \max_{a'} Q^*(s', a) \tag{12.3}$$

Note that we can't directly calculate the term $\gamma \sum_s^{\prime} p(s'|s,a) \max_a^{\prime} Q^*(s', a)$ since we don't know $p(s'|s,a)$. We will discuss how this is addressed by the two algorithms we will cover in the value-based family.

12.3.1 SARSA and Q-Learning

At a high level value-based model-free reinforcement learners work by initializing the values of $Q(s,a)$ for all states s and all actions a in an s x a matrix. They then repeat the following two steps until satisfactory performance is achieved:

1. *Act* based on Q values
2. Use s (current state), a (action), r (reward), s' (next state), a' (action taken from next state) in order to *update* the approximation of $Q(s,a)$

We will refer to (s, a, r, s', a') as an **experience**. Let $\pi(s)$ we the action that a learner takes from state s. One strategy for acting that attempts to balance exploration and exploitation is called **ϵ-greedy** and defined as follows:

$$\pi(s) = \begin{cases} \text{argmax}_a\, Q(s,a) \text{ with probability } 1 \text{ - } \epsilon \\ \text{random with probability } \epsilon \end{cases} \tag{12.4}$$

Here, ϵ is some number $\in [0, 1]$ which controls how likely the learner is to choose a random action as opposed to the currently known optimal action. Varying the value of ϵ changes the balance between exploration and exploitation.

Once the learner has had an experience, they can begin to learn Q^*. We will now describe two algorithms which perform this update differently for every new experience:

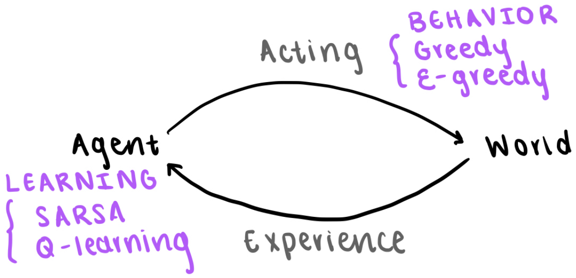


Figure 12.1: The process of model-free learning

1. **SARSA**: $Q(s,a) \leftarrow \alpha_t(s,a)[r + \gamma Q(s',a') - Q(s,a)]$ where $\alpha_t(s,a)$ is the **learning rate**, a parameter which controls how much the observation affects $Q(s,a)$.
   The expression $r + \gamma Q(s',a')$ is a 1-step estimate of $Q(s,a)$. The expression in the square brackets above is called the **temporal difference (TD) error** and represents the difference between the previous estimate of $Q(s,a)$ and the new one. Since the action a' that we use for our update is the one that was recommended by the policy π (recall that $a' = \pi(s')$), SARSA is an **on-policy** algorithm. This means that if there was no epsilon greedy action selection, the reinforcement learner would always act according to the policy π and SARSA would converge to V^π.

2. **Q-learning**: $Q(s,a) \leftarrow Q(s,a) + \alpha_t(s,a)[r + \gamma \max_a'(s',a') - Q(s,a)]$
   Similar to SARSA, the expression in the square brackets is the TD error. Note, that in Q-learning the update depends on the reward-maximizing action $\max_a'(s',a')$ corresponding to the Bellman equation Q^* rather than the policy-recommended action. This makes it an **off-policy** algorithm. In other words, Q-learning is computing V^* which may or may not correspond to actions recommended by the policy π.

This procedure of learning Q by updating $Q(s,a)$ by the TD error is called **TD updating**.

Convergence Conditions

Let $\alpha_t(s,a) = 0$ for all (s,a) that are not visited at time t. It is a theorem that Q-learning converges to Q^* (and hence π converges to π^*) as $t \to \infty$ as long as the following two conditions are met:

- $\sum_t \alpha_t(s,a) = \infty \ \forall s,a$
  The sum of the learning rate over infinitely many time steps must diverge. In order for this to happen, each state-action pair (s,a) must be visited infinitely often. Thus, we see the importance of an ϵ-greedy learner which forces the agent to probablistically take random actions in order to explore more of the state space.

- $\sum_t \alpha(s,a)^2 < \infty \ \forall s,a$
  The sum of the square of the learning rate over infinitely many time steps must converge. Note that in order for $\alpha_t(s,a)^2 < \infty$, the learning rate must be iteratively reduced. For example, we could set $\alpha_t(s,a) = \frac{1}{N_t(s,a)}$ where $N_t(s,a)$ is the number of times the learner took action a from state s.

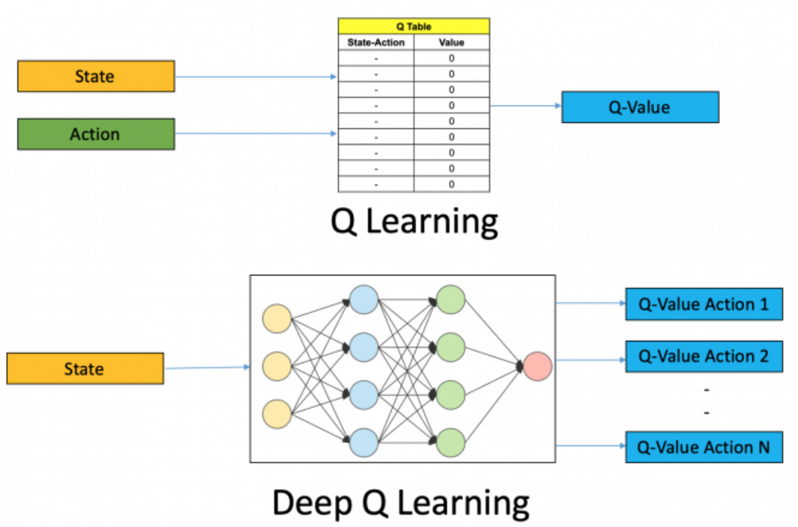
SARSA converges to Q^* if the above two conditions are met *and* behavior is greedy in the limit (ie. $\epsilon \to 0$ as $t \to \infty$). One common choice is $\epsilon_t(s) = \frac{1}{N_t(s)}$ where $N_t(s)$ is the number of times the learner visited state s. The notation $\epsilon_t(s)$ implies that a separate ϵ value is maintained for every state rather than just maintaining one value of ϵ that controls learner behavior across *all* states (though this is also an option).

Citation: Chapter 6 of *Reinforcement Learning: An Introduction* by Sutton and Barto.

12.3.2 Deep Q-Networks

Sometimes, we have scenarios where there are too many state-action pairs to learn via traditional TD updates in a reasonable amount of time. For example, consider the game of chess where there are mind bogglingly many possible configurations of the board. In cases like these, we can use a universal function approximator, like a convolutional neural network, to estimate the Q-function.

The neural network takes the learner's current state as input and outputs the approximated Q-values for taking each possible action from that state by training parameters w. The learner's next action is the maximum over all outputs of the neural network.



Q Table	
State-Action	Value
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0
-	0

Figure 12.2: Q-learning vs Deep Q-learning.(Image citation: https://www.mlq.ai/deep-reinforcement-learning-q-learning/)

While there are several specific variants depending on the problem that is trying to be solved, the general loss function that the neural network tries to minimize at iteration i is as follows:

$$\mathcal{L}(w_i) = E[(r + \gamma \max_{a'} Q(s', a'; w_{i-1}) - Q(s, a; w_i))^2] \tag{12.5}$$

Here, (s, a, r, s', a') and γ are defined the same as in regular Q-learning. w_i are the parameters that the neural net is training during the current iteration of the RL algorithm, and w_{i-1} are the optimal parameters from the previous iteration of the algorithm. The TD error is the term $r + \gamma \max_{a'} Q(s', a'; w_{i-1})$, and the squared term inside the expectation is the TD target. Since directly optimizing the loss in equation 12.5 is difficult, gradient descent, specifically stochastic gradient descent is used (in order to avoid having to calculate the entire expected value term in 12.5).

The Atari deep Q network that solved the game of brickbreaker used a technique called *experience replay* to make updates to the network more stable. The experience (s, a, r, s') was put into a *replay buffer* which was sampled from to perform minibatch gradient descent to minimize the loss function.

12.3.3 Policy Learning

There are a few situations in which the RL techniques described thus far don't work well or work relatively less well than alternatives:

1. The Q function is far more complex than the policy being learned.

2. The action space is continuous.

3. Expert knowledge needs to be introduced.

This presents the need for another model-free method called **policy learning** in which we adopt a differentiable policy $\pi_\theta(a|s)$ parametrized by θ, and try to learn θ directly without going through a Q function first.
We update θ iteratively according to the following equation:

$$\theta \leftarrow \theta + \alpha_t \nabla_\theta J(\theta)$$

where $J(\theta) = E[r(h)]$, $r(h) = \sum_t r_t$, and $h = (s, a, r, s', a', r', s'', a''...)$. The gradient term approximates the performance V^π. We define the likelihood $\mu_\theta(h) = \prod_t \pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a)$. We can expand $\nabla_\theta(\theta)$ as follows:

$$\nabla_\theta J(\theta) = \nabla_\theta E[r(n)] = \nabla_\theta \int_h \mu_\theta(h) r(h) \delta h = \int_h r(h) \nabla_\theta \mu_\theta(h) \delta h \tag{12.6}$$

We seem to have run into an issue here. The term $\nabla_\theta \mu_\theta(h)$ depends on the transition probability $p(*|s, a)$ by definition, but point of policy learning was to avoid having to learn these transition probabilities. However, it turns out we can circumvent this issue by applying the useful identity

$$\nabla_\theta \mu_\theta = \mu_\theta \frac{1}{\mu_\theta} \nabla_\theta \mu_\theta = \mu_\theta \nabla_\theta \ln \mu_\theta \tag{12.7}$$

Thus, we can rewrite the previous equation as follows:

$$\int_h r(h)\mu_\theta(h)\nabla_\theta \left[\sum_t \ln \pi_\theta(a_t|s_t) + \sum_t \ln p(s_{t+1}|s_t, a) \right] \delta h \tag{12.8}$$

$$= E\left[r(h) \sum_t \nabla_\theta \ln \pi_\theta(a_t|s_t) \right] \tag{12.9}$$

$$= E\left[\sum_t \nabla_\theta \ln \pi_\theta(a_t|s_t) r_t \right] \tag{12.10}$$

Equation 12.10 only involves $\pi_\theta(a_t|s_t)$ which we are trying to learn, and r_t which we observe from the environment. We can now perform SGD to find the optimal θ. Note policy learning is on-policy.

12.4 Model-Based Learning

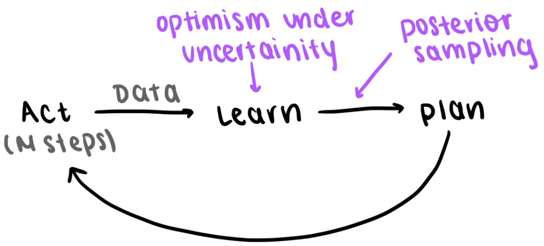


Figure 12.3: The process of model-based learning.

Recall that in model-based learning, the learner tries to learn the parameters of the MDP underlying their environment and then uses planning to come up with a policy. A model-based learner repeats the following three steps until satisfactory performance is achieved:

1. Act according to some policy for M steps.
2. Learn/update a model of the underlying MDP based on the experiences gained from step 1.
3. Use the model from step 2 to plan for the next M steps.

In many implementations, "learning" a model means coming up with a maximum likelihood estimate of the parameters of the underlying MDP: the reward function and transition probabilities (we will not cover the specifics of this process in this section). We can now plan according to this model and follow the recommended policy for the next M steps. One issue with this basic model is that it doesn't do anything to handle time spent exploring vs exploiting. While there are many ways to address this in practice, we will now present three common approaches:

- Incorporate exploration directly into the policy, for example, by using an ϵ-greedy strategy as discussed in 12.3.1.
- **Optimism under uncertainty**: Let $N(s, a)$ be the number of times we visited the state-action pair (s, a). Each "visit" is a time that the agent was in state s and chose to perform action a. When we are learning, we will assume that if $N(s, a)$ is small, then the next state will have higher reward than predicted by the maximum likelihood model. Since the model thinks taking visiting lesser-known state-action pairs will lead to high reward (we are *optimistic* when we are *uncertain*), we'll have a tendency to explore these lesser-known areas.
- **Posterior sampling/Thompson sampling**: We maintain a posterior distribution such as a Dirichlet over the transition function $p(*|s, a)$ $\forall s, a$ which we update using the experiences from the previous M steps of acting rather than coming up with a maximum likelihood point estimate. We then *sample* from this posterior to get a model, which we proceed to plan with. The idea is that when we are less certain about the outcome of the transition function, there's a greater likelihood of the sampling leading us to explore. When we are more certain about what the best outcome is, we're more likely to just stick with that best outcome.

12.5 Conclusion

To recap, reinforcement learning is a machine learning technique that allows a learner to find an optimal policy in an environment modeled by an MDP where the transition probabilities and reward function are unknown. The learner gains information by interacting with the environment and receiving rewards and uses this information to update the model they have of the underlying MDP (model-based) or their beliefs about the value of taking particular actions from particular states (value-based model-free). Q-learning and temporal difference updating are key topics in model-free learning that underlie many techniques in the area including SARSA, Q-learning and deep Q-learning. Contemporary RL models such as the one used by AlphaGo often use a combination of supervised and reinforcement learning methods which can be endlessly customized to meet the needs of a problem.