USC CSCI 467: Introduction to Machine Learning Lecture Notes

Robin Jia

Fall 2026

Contents

1 Linear Regression **4**
- 1.1 Setup . . . 4
  - 1.1.1 Making predictions . . . 4
  - 1.1.2 Learning . . . 5
  - 1.1.3 Summary of Notation . . . 6
- 1.2 Gradient Descent . . . 6
  - 1.2.1 Gradient Descent Intuition . . . 7
  - 1.2.2 Gradient Descent for Linear Regression . . . 8
- 1.3 Better Featurization . . . 9
- 1.4 Convexity: Why Gradient Descent Works . . . 11
- 1.5 Maximum Likelihood Estimation . . . 15
  - 1.5.1 MLE for coin flips . . . 16
  - 1.5.2 Deriving linear regression via MLE . . . 16

2 Classification with Logistic Regression and Softmax Regression **18**
- 2.1 Deriving Logistic Regression from MLE . . . 18
- 2.2 Gradient Descent for Logistic Regression . . . 21
- 2.3 Multi-class Classification with Softmax Regression . . . 21
  - 2.3.1 Objective and Cross Entropy . . . 23
  - 2.3.2 Gradients . . . 23
  - 2.3.3 Relationship to logistic regression . . . 24

3 Overfitting and Regularization **25**
- 3.1 Overfitting . . . 25
- 3.2 Splitting your data . . . 25
  - 3.2.1 Development Sets . . . 26
  - 3.2.2 Splitting your dataset . . . 27
- 3.3 Bias and Variance . . . 27
- 3.4 Regularization . . . 28
  - 3.4.1 L_2 Regularization . . . 28
  - 3.4.2 Maximum a Posteriori Estimation . . . 29
  - 3.4.3 L_1 Regularization . . . 30

4 Normal Equations for Linear Regression **32**
4.1 Deriving the Normal Equations . . . 32
4.2 Uniqueness and Non-invertibility . . . 34

5 Generative Classifiers and Naive Bayes **36**
5.1 Generative Classifiers Overview . . . 36
5.2 Naive Bayes for Text Classification . . . 37
5.2.1 Parameters . . . 37
5.2.2 Learning with MLE . . . 38
5.2.3 Laplace Smoothing . . . 39
5.2.4 Working in log space . . . 40
5.3 Naive Bayes for General Feature Vectors . . . 40
5.3.1 Parameters and Learning . . . 40
5.3.2 Laplace Smoothing . . . 41
5.4 Generative vs. Discriminative classifiers . . . 42

6 Non-parametric Methods **43**
6.1 k-Nearest Neighbors . . . 43
6.2 Kernel Methods . . . 45
6.2.1 Motivation . . . 45
6.2.2 Kernels and kernelized predictors . . . 45
6.2.3 A Second Look at Logistic Regression . . . 46
6.2.4 Kernels compute dot products in a different feature space . . . 47
6.3 Support Vector Machines . . . 49
6.3.1 SVM without Kernels . . . 49
6.3.2 A kernelized loss function . . . 50
6.3.3 What are support vectors? . . . 51
6.3.4 Optimization . . . 52

7 k-Means Clustering **53**
7.1 Introduction to Unsupervised Learning . . . 53
7.2 Clustering . . . 53
7.3 k-Means Clustering . . . 54
7.3.1 Setup . . . 54
7.3.2 Algorithm . . . 55
7.3.3 Choosing k . . . 57

8 Gaussian Mixture Models and Expectation-Maximization **58**
8.1 Multivariate Gaussians . . . 58
8.2 Gaussian Mixture Model . . . 61
8.3 Inference in GMMs . . . 62
8.4 Learning GMMs with Expectation-Maximization . . . 63

9 Dimensionality Reduction with Principal Component Analysis **67**
9.1 Dimensionality Reduction . . . 67
9.2 Principal Component Analysis . . . 68
9.2.1 Preliminaries . . . 68
9.2.2 Defining Reconstruction Loss . . . 69
9.2.3 Minimizing Reconstruction Error is Maximizing Variance . . . 70
9.2.4 Maximizing Variance via Eigendecomposition . . . 71

10 Bandits **73**
10.1 Introduction to Bandits . . . 74
10.2 Formal Definitions . . . 74
10.3 Upper Confidence Bound Algorithm . . . 75
10.3.1 Exploration and Exploitation . . . 75
10.3.2 UCB intuition . . . 76
10.3.3 Formal algorithm description . . . 76
10.3.4 Regret Bounds for UCB . . . 78

11 Reinforcement Learning **79**
11.1 Markov Decision Processes . . . 79
11.2 Q-Learning . . . 83
11.2.1 Reinforcement Learning Setup . . . 83
11.2.2 Tabular Q-Learning . . . 83
11.2.3 Exploration and ϵ-Greedy . . . 84
11.2.4 State discretization . . . 85
11.2.5 Q-Learning with Function Approximation . . . 85
11.2.6 Deep Q-Learning . . . 87
11.2.7 Policy Gradient Methods . . . 87

Chapter 1

Linear Regression

We wil begin our exploration of supervised learning with linear regression. Recall that a **regression** problem is one where we are trying to predict a real-valued quantity. Some examples of regression problems include:

- Predicting the tomorrow's high temperature given information about temperatures, precipitation, etc. today.
- Predicting the sale price of a house given the house's area, number of bedrooms, etc.
- Predicting the future price of a stock given current information about the company.

1.1 Setup

Our goal is to learn a function f that maps inputs x (e.g., information about houses) to outputs y (e.g., prices). In linear regression, we make the key design decision to model y with a **linear** function of the input x. Based on patterns in the **training data**, we will try to find the linear function f that best approximates y. The figure below shows a one-dimensional case, where we want to fit a line to the available data.

1.1.1 Making predictions

Now we're ready to specify our linear regression model. Since we are modeling y as a linear function of the features x, our predictions are of the form:

$$f(x) \triangleq \left(\sum_{i=1}^{d} w_i \cdot x_i \right) + b = w^\top x + b, \tag{1.1}$$

where "weight vector" $w = [w_1, \ldots, w_d] \in \mathbb{R}^d$ and "bias term" $b \in \mathbb{R}$ are **parameters** of our model, i.e., the components that we will learn from data.[1]

[1] Technically, the presence of b makes this is an affine function, but it's common to call this linear.

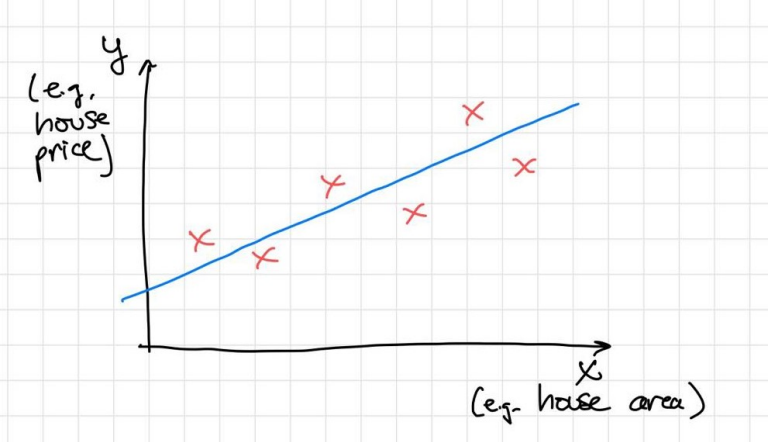


Figure 1.1: Linear regression in 1 dimension. Here we have six examples (in red), each of which is an (x, y) pair. We fit a line (in blue) which allows us to map any x to a predicted value for y.

What do these parameters mean? We can think of the bias as the baseline value (e.g., some baseline house price). Then, for each feature, there is an amount of increase/decrease per feature (e.g., for each bedroom, the price increases by 100k).

Throughout this class, we will use θ to denote all of the parameters of a model; for linear regression, $\theta = (w, b)$. We will write $f(x; \theta)$ or $f(x; w, b)$ in place of $f(x)$ to emphasize that f depends on θ.[2]

1.1.2 Learning

We have decided on the *form* or our predictions (i.e., a linear function of x); to complete the picture, we need to choose good values of w and b. The key idea in all of supervised learning is that the **training data** will help us determine which parameter values are good. Thus, we will assume access to a dataset D consisting of n training examples. Each training example is a pair $(x^{(i)}, y^{(i)})$, for $i = 1, \ldots, n$.[3]

How do we use D to choose θ? Our plan will be to write down a **loss function** $L(\theta)$ that describes how much our predictions $f(x; \theta)$ deviate from the true y's observed in D. We can then choose θ that minimize this loss. For linear regression, we will use the **squared loss** function—that is, we will measure the squared difference between our predictions $f(x; \theta)$ and true outputs y, averaged across the training examples.[4] Formally, we have

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(f(x^{(i)}; \theta) - y \right)^2 = \frac{1}{n} \sum_{i=1}^{n} \left(w^\top x^{(i)} + b - y \right)^2 . \tag{1.2}$$

[2]Note that this notation conveys something different than $f(x \mid \theta)$, which would signify "conditioning on" θ. This would be appropriate if θ were a random variable, but currently we are not viewing it as such. As we will discuss in a few classes, it is however possible to view things through a Bayesian lens, at which point we will think of θ as a random variable with some prior distribution.

[3]We use the notation $x^{(i)}$ to avoid confusion with x_i being the i-th component of the vector x.

[4]We could also use the sum instead of the average. The average has the slightly nicer property of being roughly the same magnitude regardless of how large the training dataset is.

While other loss functions are possible, we will see later why squared loss is a natural choice. The best θ would have the smallest loss, so our goal now becomes to *minimize* $L(\theta)$ with respect to θ.

We have now defined all the core concepts in linear regression! We started with the goal of learning a linear function that maps features of inputs x to outputs y. We have now reduced this problem to the problem of finding values of w and b that minimize this loss function—that is, we have rewritten a *learning* problem as an *optimization* problem. It turns out that we now have several options for solving this optimization. In this class we will cover two: Gradient descent, a general-purpose optimization strategy that we will see many times in this course, and the Normal Equations, a closed-form solution that works specifically for linear regression.

1.1.3 Summary of Notation

To summarize the new notation, we have:

- x: An input vector. We often refer to each component of x as a different **feature** (e.g., area, number of bedrooms, etc.)
- y: An output/response (e.g., price) in $\mathbb{R}$
- w, b: "Weight" and "bias" **parameters** of the model, jointly denoted as θ.
- Given a new input x, our prediction is $f(x; w, b) \triangleq w^\top x + b$.
- D: A training dataset consisting of n training examples, denoted $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$.
- $L(\theta)$: The **loss function** for linear regression, defined to be

$$\frac{1}{n} \sum_{i=1}^{n} \left(w^\top x^{(i)} + b - y^{(i)} \right)^2 .$$

1.2 Gradient Descent

Gradient descent is a general algorithm for minimizing a differentiable function. In other words, given a differentiable **objective function** $F(x)$ that maps from $\mathbb{R}^d$ to $\mathbb{R}$, gradient descent tries to find a value x^* that minimizes $F(x)$. The intuition behind gradient descent is straightforward (and much more general than just gradient descent). We will start from some initial value for x, identify a direction to step in that will lower the value of $F(x)$, take a small step in that direction, and repeat many times.

This can be illustrated succinctly in pseudocode:

Algorithm 1 High-level description of gradient descent (and many other algorithms)

1: Choose initial guess $x^{(0)}$
2: **for** $t = 1, \ldots, T$ **do**
3: Choose $x^{(t)}$ that slightly decreases $F(x)$ relative to $x^{(t-1)}$
return $x^{(T)}$

1.2.1 Gradient Descent Intuition

Rough intuition: Thinking one coordinate at a time. Before we think about how to update x, let's just think about how to update one coordinate x_i. There are only two ways to slightly change a scalar—we can increase it slightly or decrease it slightly. Which one is better? This is exactly what the derivative tells us!

Recall that the derivative of a function is positive when it's increasing, and negative when it's decreasing. So, to decide whether to increase or decrease x_i at each step, we just need to look at the derivative with respect to x_i. To be slightly more precise, we care about the derivative with respect to x_i when all other components of x are held fixed; this is known as the *partial derivative* with respect to x_i, denoted $\frac{\partial F}{\partial x_i}$. You compute partial derivatives exactly the same way you compute normal single-variable derivatives; you just have to remember to differentiate with respect to x_i and treat everything else as a constant.

Once we compute the partial derivative with respect to x_i, we have three cases:

- If $\frac{\partial F}{\partial x_i} > 0$, then F is increasing in x_i, so to make F smaller we need to make x_i smaller.
- If $\frac{\partial F}{\partial x_i} < 0$, then F is decreasing in x_i, so to make F smaller we need to make x_i bigger.
- If $\frac{\partial F}{\partial x_i} = 0$, then we don't need to change x_i at all.

Thus, the takeaway is that we always want to update w_i in the *opposite* direction of its associated partial derivative.

Finally, let's define the **gradient**. The **gradient** $\nabla_x F(x)$ is simply defined to be the vector of all partial derivatives with respect to each x_i:

$$\nabla_x F(x) \triangleq \begin{pmatrix} \frac{\partial F}{\partial x_1} \\ \frac{\partial F}{\partial x_2} \\ \vdots \\ \frac{\partial F}{\partial x_d} \end{pmatrix}$$

Note that if $x \in \mathbb{R}^d$, then $\nabla_x F(x)$ is also a vector $\in \mathbb{R}^d$.

Let's think about what this gradient vector represents. We know that for every $i = 1, \ldots, d$, we want to nudge x in the opposite direction of the gradient. In other words, given our previous guess $x^{(t-1)}$, we can generate a better new for x by *subtracting* a small multiple of the gradient from $x^{(t-1)}$.

We can formalize this intuition to arrive at the **gradient descent algorithm**:

Algorithm 2 Gradient descent

$x^{(0)} \leftarrow \mathbf{0} \in \mathbb{R}^d$
for $t = 1, \ldots, T$ **do**
    $x^{(t)} \leftarrow x^{(t-1)} - \eta \nabla_x F(x^{(t-1)})$
return $x^{(T)}$

We have initialized $x^{(0)}$ to simply be the zero vector—for problems like linear regression, it turns out the initialization does not matter very much, for reasons we will discuss in the

next class. η is the **learning rate**, a small number (e.g., 0.01) that determines how small of a step we want to take at each iteration. Note the negative sign, which ensures that we are always stepping in the direction that *minimizes* F; if that were a plus sign, this algorithm would instead maximize F. Finally, note that we are using both the sign and the magnitude of the derivative—we take a larger step if the derivative is larger in absolute value.

As written, this algorithm runs for some fixed number of steps T. Usually, you would try to choose T to be large enough that you get close to a stationary point (i.e., where the gradient is 0). There are also other possible criteria to use when deciding when to stop gradient descent, which we will discuss later in the course.

More precise intuition: Gradient as steepest ascent direction. The above argument doesn't precisely identify the gradient as the best direction to step in. Here, we more precisely show why the gradient is the most natural and efficient choice.

If you have taken multivariable calculus, you may have been taught that the gradient $\nabla_x F(x^{(t)})$ is the direction of steepest ascent—taking a step in that direction is the fastest way to increase the value of $F(x^{(t)})$. Since we want to minimize F, we will step in the exact opposite direction, i.e. $-\nabla_x F(x)$, the direction of steepest *descent*.

Why is the gradient the steepest ascent direction? One way to convince yourself of this is to think about the first-order Taylor expansion of F centered around $x^{(t)}$:

$$F(x) \approx F(x^{(t)}) + \left(\nabla_x F(x^{(t)})\right)^\top (x - x^{(t)}).$$

Re-arranging this, we have

$$F(x) - F(x^{(t)}) \approx \left(\nabla_x F(x^{(t)}\right)^\top (x - x^{(t)}). = \|\nabla_x F(x^{(t)})\| \cdot \|x - x^{(t)}\| \cdot \cos(\alpha)$$

where α is the angle between the vectors $\nabla_x F(x^{(t)})$ and $x - x^{(t)}$. The left hand side is just the amount by which F increases when we step from $x^{(t)}$ to x. If we want to take a step of a fixed size (i.e., fixed $\|x - x^{(t)}\|$), the way to have the biggest positive effect on F is thus to make $\cos(\alpha) = 1$, i.e., to make $x - x^{(t)}$ point in the same direction as $\nabla_x F(x^{(t)})$. Similarly, to have the biggest negative effect on F, we want to make $\cos(\alpha) = -1$, so $x - x^{(t)}$ should point in the exact opposite direction as the gradient.

1.2.2 Gradient Descent for Linear Regression

To apply gradient descent to linear regression, all that we need to do is derive the gradient for $L(\theta)$.

First, let's make one small change so we can stop thinking of w and b as different—they're actually essentially the same thing. I'm going to modify my dataset by adding an additional feature whose value is just always 1. The associated weight for this feature always just gets added to the final prediction, so it operates the exact same way that the bias does. This trick lets me omit the bias term—it's unnecessary now—so we can just use $w^\top x$ as the model's output.

Now let's remind ourselves of the loss function we want to optimize:

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} \left(w^\top x^{(i)} - y^{(i)}\right)^2.$$

All we need to do is take the gradient with respect to w. We can do this by applying the chain rule—it works just as well for the gradient as for the normal derivative, since the gradient is just a bunch of derivatives.

$$\nabla_w L(w) = \frac{1}{n} \sum_{i=1}^{n} 2 \cdot \left(w^\top x^{(i)} - y^{(i)}\right) \cdot x^{(i)}. \tag{1.3}$$

Here we have used the fact that

$$\nabla_w w^\top x^{(i)} = x^{(i)}.$$

You can convince yourself of this by taking the partial derivative with respect to each component w_j, and show that it indeed equals $x_j^{(i)}$.

Now, all we have to do is plug these gradient formulas into the gradient descent update rule to get our full algorithm.

Algorithm 3 Gradient descent for linear regression

1: $w^{(0)} \leftarrow \mathbf{0} \in \mathbb{R}^d$
2: **for** $t = 1, \ldots, T$ **do**
3: $\quad w^{(t)} \leftarrow w^{(t-1)} - \eta \cdot \frac{2}{n} \sum_{i=1}^{n} (w^{(t-1)\top} x^{(i)} - y^{(i)}) \cdot x^{(i)}$
return $w^{(T)}$

Understanding the gradient formula. Let's try to understand why this gradient formula makes sense. First, note that each term in the sum is simply a scalar multiplied by the vector $x^{(i)}$. So the update corresponding to the i-the example either adds or subtracts some multiple of $x^{(i)}$ to w. The sign of this update is determined by the quantity $w^\top x^{(i)} - y^{(i)}$. If this quantity is positive, that means our model has overestimated $y^{(i)}$. Thus, the gradient contains a positive multiple of $x^{(i)}$, so when we do gradient descent we subtract off a multiple of $x^{(i)}$. This makes intuitive sense: we should want to decrease the dot product between w and $x^{(i)}$, so we should be subtracting off multiples of $x^{(i)}$. Conversely, if $w^\top x^{(i)} - y^{(i)} < 0$, that means our model has underestimated $y^{(i)}$, so the gradient descent update will add a multiple of $x^{(i)}$ to w.

1.3 Better Featurization

So far, we've talked about learning functions that were linear in the input x. What if we want to learn more complex functions? It turns out we can actually do this rather easily, if we just change what x is!

Polynomial features Suppose we have a dataset that looks like this:

Clearly we cannot fit this data well with a straight line. But there's actually a simple solution! I can first modify x from a scalar to a vector with entries $[x, x^2]$, and run linear regression with these vectors as my inputs. Now that I have two features, I will have a separate parameter w_1 corresponding to x and w_2 corresponding to x^2, so I will be fitting a function of the form $w_1 x + w_2 x^2 + b$, i.e., a quadratic function. In general, I can apply any transformation to x I want, and then run linear regression on top of this transformed x.

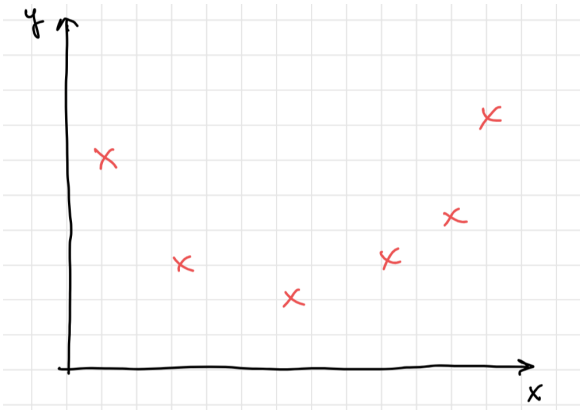


Figure 1.2: A regression dataset that cannot be fit with a straight line.

Indicator features. One common strategy for coming up with features is to create **indicator features**. These are binary features, meaning that they are always either 0 or 1. For any boolean expression over x, you can create an associated indicator function.

Indicator features are another way to help a linear regression model learn a non-linear function. For example, if we're trying to predict the sale price of a house, we might have a feature called "number of bedrooms," which is always an integer. If we were to directly use this feature in linear regression, the model could only learn a linear relationship between number of bedrooms and price. This is very limiting! Instead, we can create a group of indicator features, one for each possible number of bedrooms. That is, we have a feature for "does it have 1 bedroom," "does it have 2 bedrooms," etc. Each of these features would be 1 if the answer is yes, and 0 otherwise. This lets you learn a custom change in price for every possible number of bedrooms, instead of forcing that the difference in price between 1 and 2 is the same as the difference between 2 and 3.

Note that it makes sense at some point to start grouping values together, say, considering everything with 10+ bedrooms to be the same feature. This is because your data will become very sparse. How many houses do you think you'll see with 27 bedrooms? Probably very few, so you can't estimate the weight of a feature for "is there exactly 27 bedrooms" very well. But it's reasonable to pool all these together under "at least 10 bedrooms" so that you can at least estimate the weight associated with having a large number of bedrooms.

Indicator features are also commonly used to handle **categorical** features, which are things that take on some fixed set of discrete values (e.g., "house type"—is it a condo, or town house, or single-family home?). These features are not even numerical by nature, so in order to use them in a linear regression, you have to convert them to something numerical.

Feature engineering. Indicator features can be whatever you want, which opens the door to a lot of possibilities. Let's take the example of zip code. There are $10^5 = 100,000$ possible five-digit zip codes. So you could create $100,000$ indicator features, one for each zip code. But is this the best thing to do? If your dataset isn't that big, there might be some zip codes for which you have very few training examples, or even zero training examples, but you still want to do something reasonable if they show up at test time. So, a better idea might be to group zip codes that are geographically close to each other. You can assume that the

effect of zip code on price doesn't change that much if you move from one zip code to an adjacent one (e.g., any Los Angeles zip code will be very different from one in the middle of Illinois). There's no "right" or "wrong" answer here—depending on the task you're trying to solve and the data that's available, different choices of features may be preferable. The term **feature engineering** is commonly used to refer to the process of coming up with different ways to "featurize" your data to make your machine learning model more accurate.

So... is linear regression "linear"? The thing to remember is that linear regression learns a function that is **linear in your features**. However, you can choose features to be any function of the input, so the learned function can in fact be non-linear in your original input! You might be wondering if there is a compact way to automatically add a lot of complex features to a model. In fact the answer is yes, thanks to something called the kernel trick, which we will learn about in a few lectures.

1.4 Convexity: Why Gradient Descent Works

So far I haven't addressed a rather important question: How do we know that gradient descent will work? When we run gradient descent to minimize a function $f(x)$, will we actually find its global minimum?

Gradient descent is a very myopic algorithm—it just keeps taking steps to incrementally make our value a little better. Since it's so myopic, it can may converge to a *local* optimum that could be much worse than the *global* optimum. Luckily for us, there is a large class of objective functions for which all local optima are also global optima. When this property holds, we can guarantee that gradient descent converges to the global optimum. In particular, this holds if $f(x)$ is a **convex** function, and many objective functions we care about are in fact convex. Thus, we will now take a slight detour into the world of *convex optimization*.

An informal definition. Pictorially, a function is convex if it "holds water," whereas a concave function (i.e., a function whose negative is convex) "spills water." This is illustrated in Figure 1.3.

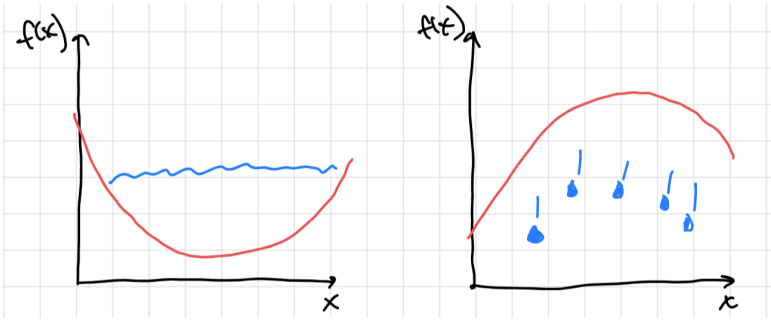


Figure 1.3: Left: A convex function "holds water." Right: A concave function "spills water." While this is not a precise definition, it can be a useful way to remember what a prototypical convex or concave function looks like.

This intuition is formalized in the following definition:

Definition 1.4.1. *A function $f : \mathbb{R}^d \to \mathbb{R}$ is **convex** if for all $x, x' \in \mathbb{R}^d$ and scalars $t \in [0, 1]$,*

$$f((1-t)x + tx') \leq (1-t)f(x) + tf(x'). \tag{1.4}$$

Geometrically, this definition says something very intuitive. Let's draw a function f and pick any two points x and x'. This inequality says that if you draw a line connecting these two points, the function has to stay below the line. The line is traced out by the right hand side of the inequality as t goes from 0 to 1, whereas the function itself is given by the left hand side.

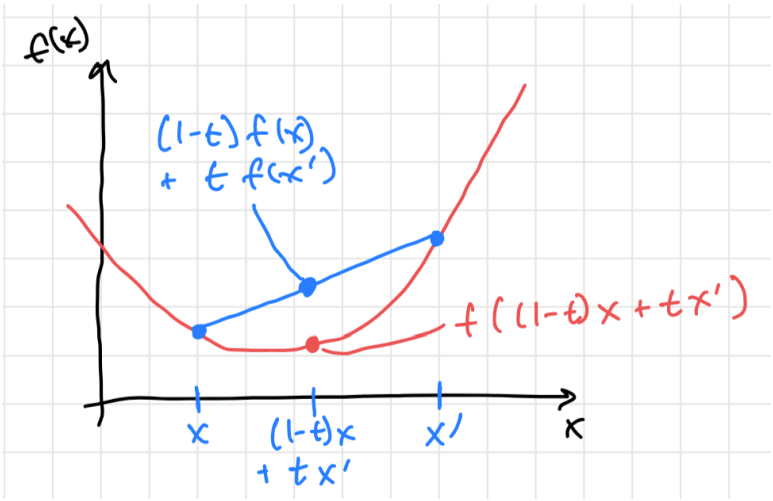


Figure 1.4: A function is convex if and only if every line segment you draw connecting two points on the function lies above the function itself.

All local minima of a convex function are global minima. With this definition, we can see geometrically that a convex function cannot have a local minimum that is not also a global minimum.[5] Figure 1.5 sketches out why this is the case.

Let's actually prove this. First, I need a formal definition of a local minimum:

Definition 1.4.2. *A point x is a **local minimum** of a function $f(x)$ if there exists $\epsilon > 0$ for which every $x' \in B_\epsilon(x)$ satisfies $f(x) \leq f(x')$. Note that $B_\epsilon(x)$ denotes the ball of radius ϵ centered around x, or equivalently, the set of points x' such that $\|x' - x\| < \epsilon$.*

Now let's suppose there are two points x and y that are both local minima of f, and $f(y) < f(x)$ (i.e., y is a "better" local minimum than x). We will show that if this is true, f cannot be convex. In other words, this situation is not possible when f is convex.

Since x is a local minimum, there is some ϵ for which every $x' \in B_\epsilon(x)$ satisfies $f(x) \leq f(x')$. Now, it is clear that we can choose some $t > 0$ such that $(1-t)x + ty = x + t(y - x) \in$

[5] Note that a convex function *can* have multiple global minima—for instance, the constant function $f(x) = 0$ is convex.

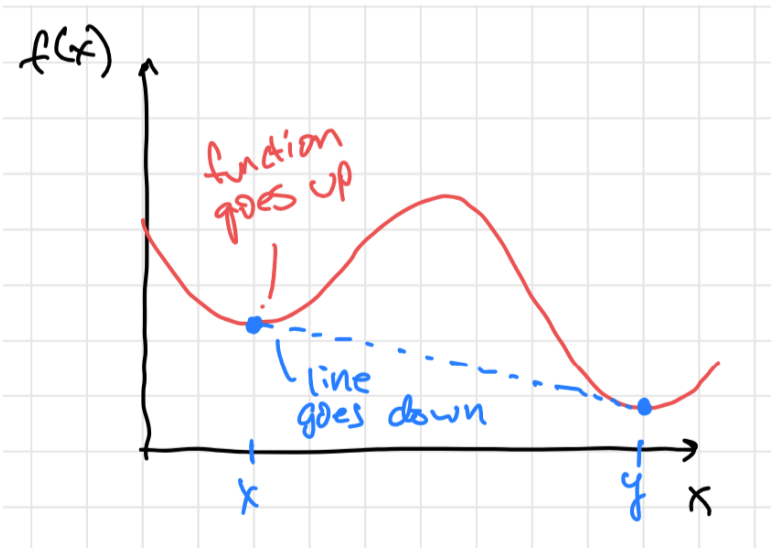


Figure 1.5: A sketch of why a convex function cannot have two distinct local minima (unless they are both equally good). If we draw a line connecting the worse local minimum (x) to the better one (y), that line must be going down, but the function around the worse local minimum must be going up. This is impossible if the function is convex.

$B_\epsilon(x)$. For example, we can choose $t = \frac{\epsilon}{2\|y-x\|}$, so that the distance from $x + t(y-x)$ to x is exactly $\epsilon/2$. By the definition of a local minimum, we know that

$$f((1-t)x + ty) \geq f(x).$$

Moreover, since $f(x) > f(y)$, we know

$$f((1-t)x + ty) > f(y).$$

Now let's multiply the top inequality by $(1-t)$ and the bottom equation by t. This gives us

$$f((1-t)x + ty) > (1-t)f(x) + tf(y).$$

Note that the left hand side is *strictly* greater than the right hand side because we knew $f(y) > f(x)$ and that $t > 0$. This is exactly the opposite of what the definition of convexity tells us, so f cannot be convex.

Showing that functions are convex. So far, I've shown you that convex functions have very nice properties. How do we know if a given function is convex? Luckily, there are rules we can use to prove a complex function is convex based on its building blocks. For our purposes right now, the following three rules will be sufficient:

1. **A univariate function $f : \mathbb{R} \to \mathbb{R}$ is convex if its second derivative is always ≥ 0.** Thus, for instance, the function $f(x) = x^2$ is convex because its second derivative $f''(x) = 2$.

Proof: I'll provide a quick proof sketch here. The basic idea is to apply the mean value theorem multiple times. We can use a proof by contradiction: suppose that for some $x, y \in \mathbb{R}$, with $y > x$, and $t \in [0, 1]$, the value of f at $(1-t)x + ty$ lies above the line connecting $(x, f(x))$ and $(y, f(y))$. For ease of notation, define $z = (1-t)x + ty$, and let m be the slope of the line connecting $(x, f(x))$ and $(y, f(y))$. The line connecting $(x, f(x))$ and $(z, f(z))$ must have a slope larger than m, since it reaches a point above the line of slope m (see Figure 1.6 for an illustration). Similarly, the line connecting $(z, f(z))$ and $(y, f(y))$ must have a slope less than m. Call these two other slopes m_1 and m_2, respectively. By the mean value theorem, there must be some point $a \in [x, z]$ where $f'(a) = m_1$, and another point $b \in [z, y]$ where $f'(b) = m_2$. So this means that over the interval $[a, b]$, the first derivative f' must be going down. By the mean value theorem again, this means there is another point c where $f''(c) < 0$, since f'' is just the derivative of f'. This is a contradiction, so the function must have actually been convex.

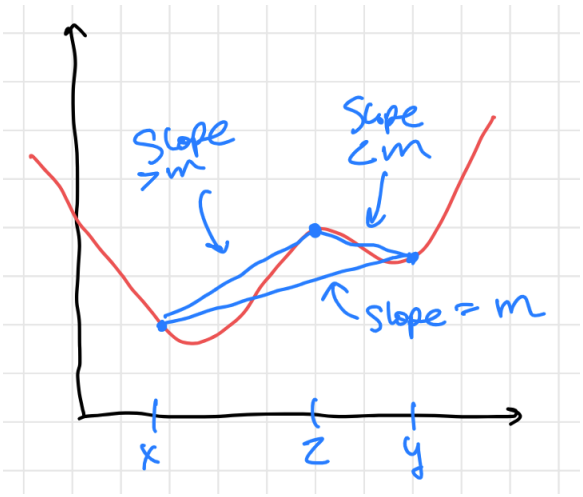


Figure 1.6: An illustration for the mean value theorem argument showing that a function with $f''(x) \geq 0$ everywhere must be convex.

2. **A convex function applied to an affine function is convex.** In other words, if $f(x)$ is convex, then so is $f(Ax + b)$ where A is a matrix/vector/scalar and b is a vector/scalar (depending on whether x is a vector or scalar, and whether f takes a vector or scalar as input). The intuition is clear in the univariate case: for any scalars a and b, $f(ax + b)$ just shifts the function along the x axis by $-b$ and scales it horizontally by a factor of a. This preserves the overall shape of the function, which is what determines convexity.

   Proof: Let $g(x) = f(Ax + b)$. For all $x, y \in \mathbb{R}^d$ and all $t \in [0, 1]$, we have:

$$\begin{aligned}
g((1-t)x + ty) &= f(A((1-t)x + ty) + b) && \text{By definition of } g \\
&= f((1-t)(Ax + b) + t(Ay + b)) && \text{Algebra} \\
&\leq (1-t)f(Ax + b) + tf(Ay + b) && \text{By convexity of } f \\
&= (1-t)g(x) + tg(y) && \text{By definition of } g
\end{aligned}$$

3. **If $f(x)$ and $g(x)$ are convex, then so is $f(x) + g(x)$.** Again, in the univariate case this makes sense given rule 1. The second derivative of $f(x) + g(x)$ is simply $f''(x) + g''(x)$, and so if each one is individually ≥ 0 then so is the sum.

   Proof: Let $h(x) = f(x) + h(x)$. For all $x, y \in \mathbb{R}^d$ and all $t \in [0, 1]$, we have:

$$\begin{aligned} h((1-t)x + ty) &= f((1-t)x + ty) + g((1-t)x + ty) && \text{By definition of } h \\ &\leq (1-t)f(x) + tf(y) + (1-t)g(x) + tg(y) && \text{By convexity of } f \text{ and } g \\ &= (1-t)h(x) + th(y) && \text{By definition of } h \end{aligned}$$

4. **If $f(x)$ is convex and $c > 0$, then so is $cf(x)$.** Again this makes intuitive sense in 1-D, since scaling by a positive constant just stretches the function along the y-axis. The proof is straightforward and similar to the last two.

This is by no means an exhaustive set of rules, but these are the most common ones that come up. My goal in this lecture is not to give you a comprehensive tour of convex functions, but just to know some key rules of thumb so that you can look at a new loss function and quickly surmise whether it is convex. If it's convex, you can be confident that you will not run into bad local minima when optimizing it.

Linear regression is convex. Let's apply what we learned to linear regression. Recall that our loss function was

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} \left(f(x^{(i)}; w) - y^{(i)} \right)^2 = \frac{1}{n} \sum_{i=1}^{n} \left(w^\top x^{(i)} - y^{(i)} \right)^2 . \tag{1.5}$$

We want to show that this is a convex function of w. Each term inside the sum is a convex function (i.e., the function $g(x) \to x^2$, which is convex by Rule 1) applied to an affine function of w. (Keep in mind that since this is a function of w, we just think of $x^{(i)}$ and $y^{(i)}$ as constants.) Thus, by Rule 2, each term of the sum is convex. Finally, by Rule 3, the sum of all these convex terms is also convex, and by Rule 4 we can safely multiply everything by $\frac{1}{n}$.

1.5 Maximum Likelihood Estimation

Finally, I want to come back to the question of why we use squared loss, and connect this to some much more general thoughts about how we can come up with machine learning algorithms. We will re-apply these principles in the next class on logistic regression, our first method for classification.

One general principle for designing machine learning algorithms is the framework of **Maximum Likelihood Estimation** (MLE). The idea is to view the observed data as being generated by some probabilistic process parameterized by our model parameters θ. The best θ is the one that best explains the data, i.e., the θ under which the data has the highest probability (hence, maximum likelihood).

1.5.1 MLE for coin flips

Let's warm up with a simple example. Suppose we have a coin that might be biased. Every time you flip it, it has some probability p of coming up heads (and thus probability $1-p$ of coming up tails).

Now suppose we flip it n times. Each time, we get an observation y_i that is either 1 (heads) or 0 (tails). Based on this observed data, we would like to infer what the value of p is. One principle by which we can make this inference is the principle of maximum likelihood estimation. Out of all possible p's (i.e., all possible numbers between 0 and 1), we want to figure out which one maximizes the probability of the observed data.

To do this, we need to do the following steps:

1. Write down the probability of the data as a function of our parameters (in this case, p).

2. Compute the value of p that maximizes this probability.

In Homework 0, you will carry out these two steps for this simple case of a weighted coin.

1.5.2 Deriving linear regression via MLE

Now we will derive linear regression by applying the principle of maximum likelihood estimation under a particular, natural probabilistic model. In particular, let's assume that each $y^{(i)}$ is drawn independently at random from a Gaussian distribution with variance σ^2 centered around $\theta^\top x^{(i)}$.[6] Why Gaussian? One can imagine that one reason we can't perfectly predict y is that there are many small, random effects that we are not modeling. By the Central Limit Theorem, the cumultative effective of many such small, independent, random effects will be normally distributed.

Remember that the Gaussian pdf for mean μ and variance σ^2 is given by

$$p(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

The overall probability of the $y^{(i)}$'s given all the $x^{(i)}$'s is

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)};\theta) \tag{1.6}$$

$$= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma}(y^{(i)} - \theta^\top x^{(i)})^2\right) \tag{1.7}$$

We take the product because we assumed the $y^{(i)}$'s are sampled independently.[7]

[6] Note that we are only modeling the $y^{(i)}$'s here, and ignoring the probability of the $x^{(i)}$'s.

[7] A quick note on terminology: $\mathcal{L}(\theta)$ denotes the *likelihood* of θ, which is equal to the *probability* of the data under θ. Following convention, we will try to be consistent that likelihood refers to a function of θ, even though colloquially "likelihood" and "probability" can often refer to the same thing.

In MLE, this is the objective function that we are trying to *maximize*. I now claim that this is exactly the same as minimizing our original linear regression objective. Let's start by taking the log of everything—since log is a monotonically increasing function, maximizing likelihood is the same as maximizing log-likelihood (which we denote $\mathcal{LL}(\theta)$). Taking the log often is useful because it converts products into sums, and sums are easy to differentiate. We have:

$$\mathcal{LL}(\theta) = \log \left(\prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma}(y^{(i)} - \theta^\top x^{(i)})^2 \right) \right) \tag{1.8}$$

$$= \sum_{i=1}^{n} \left(\log\left(\frac{1}{\sigma\sqrt{2\pi}} \right) - \frac{1}{2\sigma}(y^{(i)} - \theta^\top x^{(i)})^2 \right) \tag{1.9}$$

$$= n \log\left(\frac{1}{\sigma\sqrt{2\pi}} \right) - \frac{1}{2\sigma} \sum_{i=1}^{n} (y^{(i)} - \theta^\top x^{(i)})^2. \tag{1.10}$$

The first term is a constant that does not depend on θ, so we can ignore it when optimizing with respect to θ. The latter term is exactly our familiar squared loss from before, multiplied by a negative constant. This means that *maximizing* $\mathcal{LL}$ is indeed equivalent to *minimizing* squared loss.

Chapter 2

Classification with Logistic Regression and Softmax Regression

2.1 Deriving Logistic Regression from MLE

Last class, we derived the equation for linear regression by applying the principle of Maximum Likelihood Estimation: We choose the parameters θ to maximize the log-likelihood of the data, i.e.

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; \theta). \tag{2.1}$$

To come up with an algorithm for classification, let's use the same recipe but with a different probabilistic model. Logistic regression[1] starts with the following model:

$$p(y = 1 \mid x; w) = \frac{1}{1 + \exp(-w^\top x)}, \tag{2.2}$$

where both our feature vector x and parameter vector w are in $\mathbb{R}^d$.[2] Note that $p(y = -1 \mid x; w) = 1 - p(y = 1 \mid x; w)$ by definition.

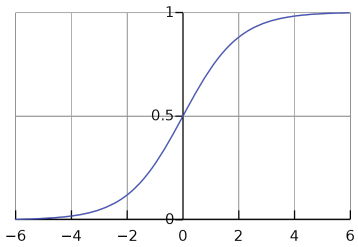
What's going on here? We can define the function $\sigma(z) = \frac{1}{1+\exp(-z)}$, called the "sigmoid" or "logistic" function, so we can now write:

$$p(y = 1 \mid x; w) = \sigma(w^\top x). \tag{2.3}$$

The function $\sigma(z)$ looks like the following:

[1] It is very unfortunate that this is called logistic *regression* because it is used for classification problems, not regression problems.

[2] Just as in linear regression, we can remove the need for a bias term by adding a feature whose value is 1 for every example.



The important things to note are:

- For large positive z, $\sigma(z)$ approaches 1.
- For small (very negative) z, $\sigma(z)$ approaches 0.
- When $z = 0$, $\sigma(z) = \frac{1}{2}$.
- $1 - \sigma(z) = \sigma(-z)$.

Now we see the point of $\sigma(z)$. We can view $w^\top x$ as a score of whether the example looks more like a positive example (if $w^\top x > 0$) or a negative example (if $w^\top x < 0$). σ transforms this raw score into a probability between 0 and 1.

Figure 2.1 below shows a geometric picture of what's going on. In two dimensions, the equation $w^\top x = 0$ defines a line (in d dimensions, it defines a $d-1$-dimensional hyperplane). This represents points where $p(y = 1 \mid x) = \frac{1}{2}$—that is, points that seem equally likely to be positive or negative. On one side of the line are points with positive dot product with w, which are more likely to be positive; on the other side, the points have negative dot product with w, and thus are more likely to be negative.

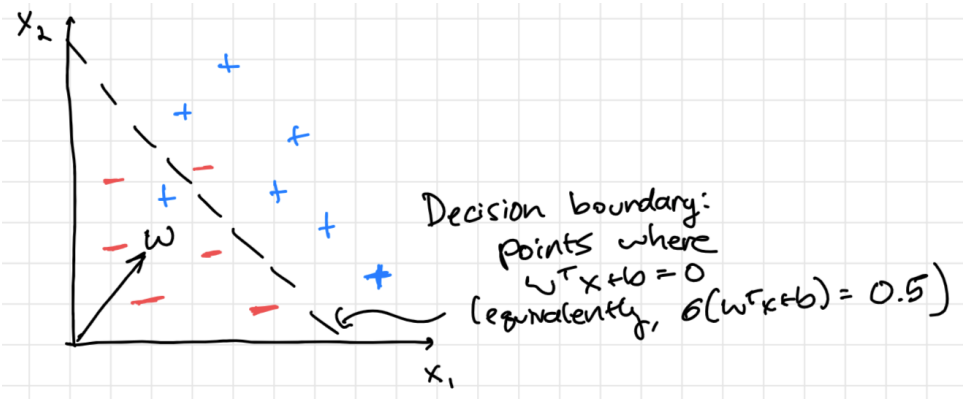


Figure 2.1: Illustration of logistic regression decision boundary. The decision boundary is the hyperplane defined by the equation $w^\top x + b = 0$. This hyperplane is always perpendicular to the weight vector w. Points on the decision boundary are predicted by the model to be equally likely to be positive or negative. Going further from the decision boundary, the model makes more confident predictions (i.e., the probabilities become closer to 1 or 0).

Since the decision boundary we learn is still a linear function of the features, logistic regression is another instance of a **linear model**, just like linear regression.[3]

Note that we can write $p(y = -1 \mid x; w) = 1 - p(y = 1 \mid x; w) = 1 - \sigma(w^\top x) = \sigma(-w^\top x)$. Thus, we can use this compact expression:

$$p(y \mid x; w) = \sigma(y \cdot w^\top x). \tag{2.4}$$

You can easily check that this is correct both when $y = 1$ and when $y = -1$.

Putting everything together, we arrive at this objective function:

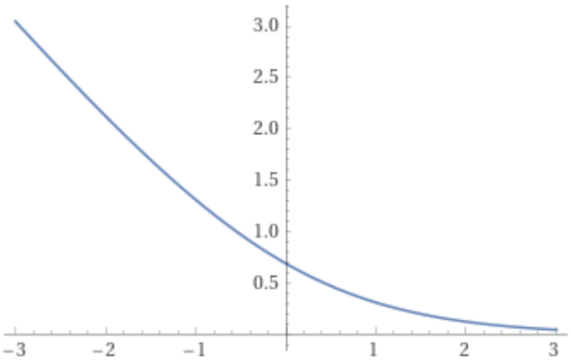
$$\log \mathcal{L}(w) = \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; w) \tag{2.5}$$

$$= \sum_{i=1}^{n} \log \sigma(y^{(i)} \cdot w^\top x^{(i)}). \tag{2.6}$$

Recall that we want to maximize (log-)likelihood. However, usually in machine learning convert problems into minimization problems. We can do this simply by adding a negative sign. I'll also multiply by $\frac{1}{n}$ just to make it easier to interpret this as an average across the dataset (this doesn't affect what w minimizes the loss):

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} -\log \sigma(y^{(i)} \cdot w^\top x^{(i)}) \tag{2.7}$$

We can interpret the function $\ell(z) = -\log \sigma(z)$ as a **loss function** similar to squared loss from linear regression. If it is high, that means our model is doing a bad job, and if it is close to zero, it is doing a good job. Here's what this function looks like:



The expression $y \cdot w^\top x$ is called the **margin**, and is a quantity that we always want to be > 0, as that means y and $w^\top x$ have the same sign, and thus that we are classifying correctly. The loss function $\ell(z)$ is our way of saying exactly how unhappy we are with any particular value of the marign. You see that we are very unhappy when the margin is very negative, and then this amount of unhappiness decreases.

[3]Note that just as in linear regression, we can define all sorts of complex features if we want to learn a complex decision boundary to separate the positive and negative examples.

2.2 Gradient Descent for Logistic Regression

Let's keep going with the same recipe we used for linear regression. Remember that we first wrote down an objective function that we wanted to minimize, then used gradient descent to optimize the function with respect to w. We can do the exact same thing here. The only thing that changes is the gradient itself.

First, let's prove a useful fact about the σ function:

$$\begin{aligned}\frac{d}{dz}\log\sigma(z) &= \frac{1}{\sigma(z)}\cdot\frac{d}{dz}\sigma(z)\\ &= (1+\exp(-z))\cdot\left(-(1+\exp(-z))^{-2}\right)\cdot(-\exp(-z))\\ &= \frac{\exp(-z)}{1+\exp(-z)}\\ &= \sigma(-z).\end{aligned}$$

Now we just combine this fact with the chain rule to get the gradient of the objective L:

$$\nabla_w L(w) = \nabla_w \frac{1}{n}\sum_{i=1}^{n} -\log\sigma(y^{(i)}\cdot w^\top x^{(i)}) \tag{2.8}$$

$$= \frac{1}{n}\sum_{i=1}^{n} -\sigma(-y^{(i)}\cdot w^\top x^{(i)})\cdot y^{(i)}\cdot x^{(i)}. \tag{2.9}$$

Finally, recall that we want to decrease L, so we will be stepping in the *negative* direction of the gradient. This means that we will our update rule will look like:

$$w^{(t+1)} = w^{(t)} + \eta\frac{1}{n}\sum_{i=1}^{n}\sigma(-y^{(i)}\cdot w^\top x^{(i)})\cdot y^{(i)}\cdot x^{(i)} \tag{2.10}$$

where η is the learning rate, as before.

Let's try to understand why this expression makes sense. First, let's check that the signs of everything make sense. σ is always > 0 so let's put it aside for now. For a positive example, we will *add* some multiple of $x^{(i)}$ to w. This makes sense because we want positive examples to have high dot product with w. Conversely, we will *subtract* some multiple of $x^{(i)}$ to w for each negative example, since we want them to have low dot product with w. Finally let's look at the σ term. This is σ of the *negative margin*. That means that if the margin is very low (bad), then this number is close to 1. If the margin is very large (good), then the number is close to 0. This means that if we're doing very badly on a particular example, we do a large update to w. If we're doing well on a particular example, then we don't need to change w in a big way (as far as that example is concerned).

2.3 Multi-class Classification with Softmax Regression

Logistic regression works for binary classification. What if we have more than two classes? For example, what if we have an image and want to classify what species of animal it is?

This is a **multi-class classification** problem. There is a natural way to extend logistic regression to multi-class settings, known as **softmax regression** or **multinomial logistic regression**.

Essentially, the idea is that if we have C classes, we will now have C parameter vectors of dimension d instead of a single vector. The j-th vector will score how well the input matches the j-th class. Formally, we have a set of parameters $W = \{w^{(1)}, \ldots, w^{(C)}\}$ where each $w^{(j)} \in \mathbb{R}^d$. Given an input x, the score for the j-th class is $w^{(j)\top}x$, where higher score corresponds to higher probability of x being from class j.

Now let's talk about how you get probabilities here. Intuitively, the model should predict whichever class j has the highest value of $w^{(j)\top}x$. But we can't directly use these dot product scores as probabilities, since some might be negative. A natural thing to do is to first apply the exponential function, ensuring the score for every class is positive. Then we can simply normalize by the sum to get a probability distribution.

Translated into math, we get the following:

$$p(y = j \mid x; W) = \frac{\exp(w^{(j)\top}x)}{\sum_{k=1}^{C} \exp(w^{(j)\top}x)}. \tag{2.11}$$

This function—exponentiating everything and then normalizing—has a special name, called the **softmax** function. Let's try to understand what this function does. Suppose $K = 3$ and we have the following:

$$\begin{aligned} w^{(1)\top}x &= 2 \\ w^{(2)\top}x &= 1 \\ w^{(3)\top}x &= -3 \end{aligned}$$

These pre-softmax values are often referred to as the **logits**. (This term can also be used to refer to $w^\top x$ in logistic regression, i.e., what you get before the sigmoid function.)

After we exponentiate, we get:

$$\begin{aligned} \exp(w^{(1)\top}x) &\approx 7.4 \\ \exp(w^{(2)\top}x) &\approx 2.7 \\ \exp(w^{(3)\top}x) &\approx 0.1 \\ \sum_{k=1}^{3} \exp(w^{(k)\top}x) &\approx 7.4 + 2.7 + 0.1 = 10.2 \\ p(y = 1 \mid x; W) &\approx 7.4/10.2 \approx .72 \\ p(y = 2 \mid x; W) &\approx 2.7/10.2 \approx .27 \\ p(y = 3 \mid x; W) &\approx 0.1/10.2 \approx .01 \end{aligned}$$

What has happened? The softmax function found the index with the largest value (here, $k = 1$) and put most of the probability mass on this value. This is another one of those cases where the name of the function is misleading. You might think that softmax should be a "soft" version of the max function, but this is not the case. It is more similar to a soft $\arg\max$.

2.3.1 Objective and Cross Entropy

What do we do now? Once again we apply the principle of Maximum Likelihood Estimation: We search for the W that maximizes the (log-)probability of the data with gradient descent. By now, we know that we will want to maximize likelihood, which is the same as maximizing log-likelihood, which is the same as minimizing negative log-likelihood, which is the same as minimizing $\frac{1}{n}$ times the negative log-likelihood So I'm just going to skip straight to the last part:

$$\begin{aligned} L(W) &= -\frac{1}{n}\sum_{i=1}^{n} \log P(y^{(i)} \mid x^{(i)}; W) \\ &= -\frac{1}{n}\sum_{i=1}^{n} \left(w^{y^{(i)}\top} x - \log \sum_{k=1}^{C} \exp(w^{(k)\top} x^{(i)}) \right) \end{aligned}$$

This objective is also sometimes called the **cross-entropy loss**, because we can view it as the cross-entropy between the true label distribution and our predicted label distribution. For two distributions p and q over $\{1, \ldots, C\}$, the cross-entropy is defined as

$$H(p, q) = -\sum_{k=1}^{C} p_k \log q_k.$$

Cross-entropy is a measurement of how well q "covers" the distribution of p, where lower cross-entropy signifies that q is close to p. It is minimized when $q = p$, in which case it equals the entropy of p.

If we define $p_k^{(i)}$ to be 1 when $k = y^{(i)}$ and 0 otherwise, and $q_k^{(i)}$ to be our model's predicted distribution on example i, then our objective is simply minimizing cross-entropy:

$$L(W) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{C} -p_k^{(i)} \log q_k^{(i)}$$

2.3.2 Gradients

We take a gradient in much the same way. Let's focus on the gradient with respect to $w^{(c)}$, the weight vector for class c, i.e., $\nabla_{w^{(c)}} L(W)$.

The gradient for the first term is easy—it's either $x^{(i)}$ if $y^{(i)} = c$ or 0 otherwise.

The second log-sum-exp term is a bit more complex, but we can handle it with the chain rule:

$$\nabla_{w_c} \log \sum_{k=1}^{C} \exp(w^{(k)\top} x^{(i)}) = \frac{1}{\sum_{k=1}^{C} \exp(w^{(k)\top} x^{(i)})} \cdot \exp(w^{(c)\top} x^{(i)}) \cdot x^{(i)} \tag{2.12}$$

All we had to do is apply the chain rule twice, first to log and then to exp. Rewriting this, this is simply

$$q_c^{(i)} \cdot x^{(i)}.$$

Putting this all together, the full gradient expression is (folding in the negative signs):

$$\frac{1}{n}\sum_{i=1}^{n}(q_c^{(i)} - \mathbb{I}[y^{(i)} = c]) \cdot x^{(i)}.$$

This is kind of similar to logistic regression, in that again we have a scalar times $x^{(i)}$. There are two cases now. If $y^{(i)} \neq c$, then the scalar is positive, so we will subtract some multiple of $x^{(i)}$ to w_c. If $y^{(i)} = c$, then the scalar is negative (since q_c is a probability distribution), its entries are always ≤ 1, so when we do gradient descent we add. This makes intuitive sense in both cases.

2.3.3 Relationship to logistic regression

You might be wondering whether we can also use softmax regression when $K = 2$. It winds up essentially being the same as a reparameterized version of logistic regression.

Let w_0 and w_1 be the weight vectors for the two classes in binary classification. Then we have:

$$\begin{aligned} P(y = 1 \mid x; w) &= \frac{\exp(w_1^\top x)}{\exp(w_1^\top x) + \exp(w_0^\top x)} \\ &= \frac{1}{1 + \exp((w_0 - w_1)^\top x)}. \end{aligned}$$

This is just the same as logistic regression with parameter $w = w_1 - w_0$. Intuitively this makes sense—$w_1 - w_0$ is the key vector of interest because it captures what differentiates a positive example from a negative example.

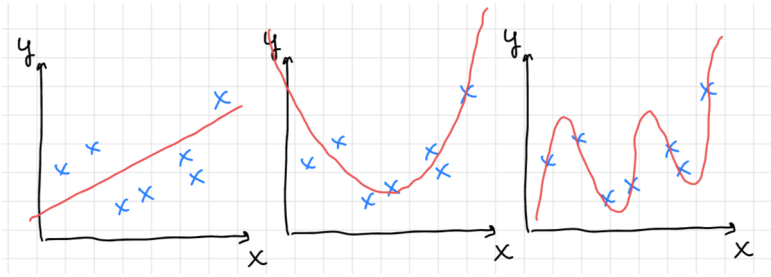
Chapter 3

Overfitting and Regularization

So far, we've talked a lot about training models—either regression or classification models. But just because we have run training correctly doesn't guarantee that the model is good or useful!

3.1 Overfitting

When we train a model, we find parameters that lead to low loss on the training data. Let's consider a simple example where we use linear regression with polynomial features to fit a dataset. We can use either linear features only, linear and quadratic features, or features up to degree 7. These three options might result in the following three predictors:



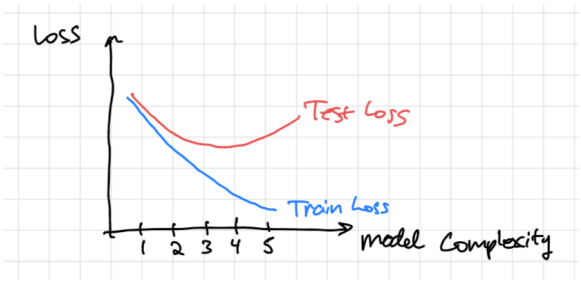
The first function seems to be too simple to fit the data well. We call this **Underfitting**. The second function seems to roughly capture the key trends. What about the last function? It perfectly fits all of the data, so it achieves zero training loss. But it seems like it has done so by fitting a lot of non-meaningful fluctuations. This is known as **Overfitting**. But by what principle do we avoid doing this? Based on training error, this is the best possible function.

3.2 Splitting your data

Okay, how do we detect that using a high-degree polynomial is a worse predictor, even though it fits the data better? The ultimate litmus test is whether our model makes good predictions on new, unseen examples. We use a **test dataset**, a separate dataset from the

training dataset that's only used for evaluation. Accuracy on the test dataset is a proxy for how well the model predicts on a *new* example, which is actually what we care about. We don't actually care about how well we fit the training data—we already know the training labels!

Plotting train vs. test loss. A common way to visualize the difference between training loss and test loss is to plot the loss as a function of how complex your model is:



For a while, increasing the complexity of your model improves both training loss and test loss. But if you keep increasing it more, you improve training loss but hurt test loss! This is the regime where you are overfitting.

3.2.1 Development Sets

Okay, so we now understand why we might not want to throw all possible features into our model. But how do we decide exactly which features to use? The best way to do this is with a **development set**, also known as a validation set. The point is that you should actually always have three separate datasets. The training dataset is for training the model. The development set helps you choose **hyperparameters**, such as how many features to use. You can also use it to decide what is the best learning rate for gradient descent, or other settings of your training method that we'll talk about later. Once you do this, you evaluate on the test set to estimate how good your model is on unseen examples.

Hopefully it's clear why we can't choose hyperparameters on the training set (we'll always choose to have all the features). But why is separating the dev set and test set important? If you fit hyperparameters on the test set and then evaluate on that test set, you will have "cheated" in some way. We can draw an analogy to taking an exam:

- If you use the same dataset for training and evaluation, that's like cheating on an exam by getting a copy of it ahead of time and memorizing all the answers. It's very obvious here that your performance on the exam does not reflect your actual ability.

- If you use the same dataset for choosing hyperparameters and evaluation, that's like if you got to take the exam 100 times, where you get your memory wiped in between tries, but then you get to keep the best score out of those 100 tries. (Each "try" represents a different hyperparameter setting.) There's going to be some random fluctuation in how well you do on that exam, so at some point you're going to get lucky and do better than your average score. So this is still an overestimate of your true ability.

3.2.2 Splitting your dataset

In the real world, usually you're not given a dataset with pre-defined train, development, and test subsets. For example, if you're predicting house prices, you just have one big dataset of houses that have been sold and what their sale price was. So, before you do any machine learning, you first need to **split** your dataset into train, development, and test subsets (more commonly referred to as "splits"). The simplest way to do this is to randomly partition the dataset into three subsets. Usually you would reserve the most data for training, since that determines how well your model will work. Something like 70% train, 10% development, and 20% test is often reasonable, although you also want to make sure your development and test sets aren't too small (otherwise you won't be able to reliably estimate how well your model is doing).

3.3 Bias and Variance

One way to conceptualize the danger of overfitting is the **bias-variance tradeoff,** as illustrated in Figure 3.3. Any machine learning model will make mistakes for one of two reasons:

- **Bias.** Bias refers to errors that arise because assumptions of the model do not match the reality of the task. For example, maybe your model assumes that y is a linear function of your features. This is probably not literally true, which puts some upper bound on how well your model can fit the data. In general, whenever we use a machine learning algorithm, we are restricted to a particular **model family**—the set of possible functions we can learn. For linear regression, the model family is the set of linear functions of the features. Bias arises when our model family is too small to represent the actual function we're trying to model. When we have large bias, our models will underfit the training data.

- **Variance.** Variance is error from sensitivity to small fluctuations in the training set. Suppose we assume we need polynomial features of degree 20. There might be a "best" 20-degree polynomial, but given a finite training dataset you might learn one that is not as good. The difference between what you find and the best possible thing within your function class is variance. The larger your model family is, the larger your variance will be—when you have more candidates, it's harder to identify the best one. When we have high variance, our models are likely to overfit the training data.

Overall, our test accuracy is determined by both bias and variance. So, we must always be careful to achieve a good balance to avoid having either very high bias (underfitting) or very high variance (overfitting).

Note that some bias is always necessary in the real world. In the extreme, you can consider the function class of "all possible functions." This always has zero bias, but you also can never learn anything—no training dataset can tell you how to choose between the infinite possible functions that perfectly fit your data but make different predictions on unseen data. The whole point of machine learning is that labels on seen examples must give

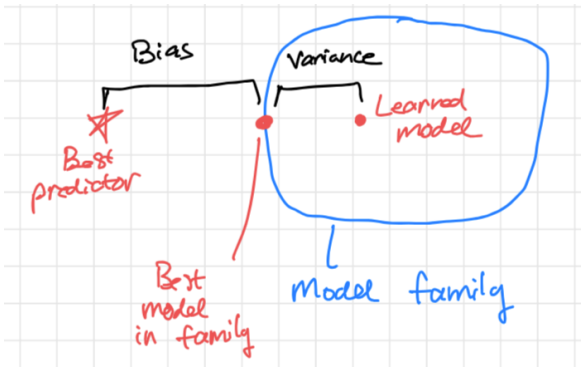


Figure 3.1: Illustration of the bias-variance trade-off. Bias describes the difference between the optimal predictor and the best predictor in the model family. Variance describes the difference between the best predictor in the model family and the predictor that we actually learn. Both together define how much worse our model is than the optimal function.

you *some* information about labels on unseen examples. This is known as **inductive bias**, and is captured by the saying, "All models are wrong, but some are useful."

3.4 Regularization

How else can we prevent overfitting? The bias-variance diagram suggests that in order to do this, we should try to restrict the set of functions under consideration. Reducing the number of features is one way to do this. Another common idea is **regularization**. The idea of regularization is to impose a soft constraint to encourage "simpler" functions.

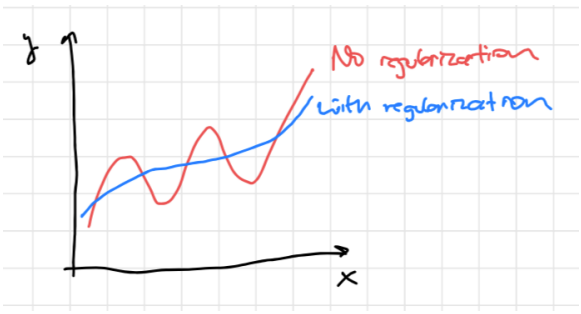
3.4.1 L_2 Regularization

Let's start with the most common version, called L_2 regularization. We simply add a term to the loss function that penalizes the L_2 norm of our parameters. For linear regression, this looks like:

$$L(w) = \left(\frac{1}{n} \sum_{i=1}^{n} (w^\top x^{(i)} - y^{(i)})^2 \right) + \lambda \|w\|^2.$$

Recall that $\|w\|^2 = \sum_{i=j}^{d} w_j^2$, just the sum of squared entries of w. The λ is a **hyperparameter** that determines how much we want to constrain w to have a small L_2 norm. $\lambda = 0$ corresponds to no regularization.

How does reducing the norm reduce the complexity of our function? In the linear regression case, this basically means we can only learn a polynomial whose coefficients aren't that large. This means it cannot have too many wiggles in it.



Note that L_2 regularization can be combined with any of the methods we've seen (logistic or softmax regression). Its overall effect is always the same—it encourages the w you learn to have smaller norm, which decreases the size of the model family and helps reduce overfitting.

Gradient for L_2 regularization. The L_2 regularization term just gets added to the original loss function, so its gradient also just gets added to the gradient for the original loss function. The gradient is simply:

$$\nabla_w \lambda \|w\|^2 = 2\lambda w.$$

Note that this is analogous to the derivative of x^2 in one dimension, since $\|w\|^2$ is just the sum of the squares of each element of w.

During gradient descent, we step in the direction of the negative gradient. Thus, the effect of L_2 regularization is that we always update w_j *towards* 0. If w_j is a large positive number, we will step in the negative direction; if it's a large negative number, we will step in the positive direction.

3.4.2 Maximum a Posteriori Estimation

There's yet another way to justify the use of regularization, and it relates to yet another probabilistic story. For this, we have to talk about Frequentism and Bayesianism.

So far, when we've done MLE, we've adopted a frequentist view of things: There's some true value of the parameters out there in nature, and our goal is to estimate that real value as close as possible.

An alternative is a Bayesian viewpoint: The parameters themselves are a random variable that are drawn from some distribution. In this view, we can view regularization as coming from a **prior distribution** on the parameter.

It's time to introduce the Bayesian version of MLE, called **Maximum a Posteriori Estimation** (MAP). In MAP, we imagine computing a posterior over the parameters conditioned on the data:

$$P(w \mid D) = \frac{P(w)P(D \mid w)}{P(D)}.$$

In MAP estimation, we find the w that maximizes $P(w \mid D)$—i.e., the most likely w according to the posterior distribution. The $P(D \mid w)$ term is the same thing we saw in MLE. The $P(w)$ term is the new thing, which is the prior. Finally, the denominator doesn't depend on w so it doesn't matter.

Where does L_2 regularization come from? Let's assume that $P(w)$ is a Gaussian, in particular that each component w_j is distributed as an independent Gaussian with variance σ^2 and mean 0. Then the $P(w)$ term is

$$\prod_{j=1}^{d} \frac{1}{\sigma\sqrt{2\pi}} \exp(-w_j^2/2\sigma^2).$$

Analogously to MLE, in MAP we want to find the w that is most likely under the posterior distribution $P(w \mid D)$; in other words, we want to maximize $P(w \mid D)$ with respect to w. As we know, this is equivalent to minimizing the negative log of $P(w \mid D)$. So we have

$$\begin{aligned}
-\log P(w \mid D) &= -\log P(w) - \log P(D \mid w) + \log P(D) \\
&= -\left(\sum_{j=1}^{d} -\log(\sigma\sqrt{2\pi}) - \frac{1}{2\sigma^2} w_j^2\right) - logP(D \mid w) + C_1 \\
&= \frac{1}{2\sigma^2} \sum_{j=1}^{d} w_j^2 - \log P(D \mid w) + C_2 \\
&= \frac{1}{2\sigma^2} \|w_j\|^2 - \log P(D \mid w) + C_2,
\end{aligned}$$

where C_1 and C_2 are constants that do not depend on w, so we can ignore them when trying to minimize this objective. Note that the second term is exactly the loss function for MLE (up to multiplication by a constant). THe first term is exactly L_2 regularization, where σ controls the strength of the regularization (i.e., $\lambda = \frac{1}{2\sigma^2}$). Smaller σ means a stronger prior towards small values, hence more regularization.

3.4.3 L_1 Regularization

Another common approach to regularization is to use L_1 regularization, which penalizes the 1-norm rather than the 2-norm. More formally, we add $\lambda\|w\|_1$ to the loss function, where

$$\|w\|_1 = \sum_{j=1}^{d} |w_j|.$$

Again, λ is a hyperparameter that controls how much regularization we want to add.

Gradient for L_1 regularization. To obtain the gradient, let's take the partial derivative of the L_1 regularization term with respect to w_j. We can show that

$$\frac{\partial}{\partial w_j} \lambda\|w\|_1 = \lambda \operatorname{sign}(w_j),$$

where the sign function is defined to be 1 if its input is positive, -1 if its input is negative, and 0 otherwise. So, overall we can write

$$\nabla_w \lambda\|w\|_1 = \lambda \operatorname{sign}(w)$$

where we understand $\operatorname{sign}(w)$ to denote element-wise application of sign to the vector w.

Comparison with L_2 regularization. Comparing the two gradient expressions helps us understand the difference between L_1 and L_2 regularization. In both cases, the gradient update rule always tells us to step towards 0. In L_2 regularization, we do a large gradient descent update if w_j is far from 0, and a small update when w_j is close to 0 (since the size of the step is just proportional to $|w_j|$). In L_1 regularization, we do a fixed size step no matter how big w_j is. Thus, L_1 has a **sparsifying effect**. It will keep pushing w_j to be smaller and smaller until it's at 0. But if w_j is already very large, it will not try to change it by a lot to get it closer to 0. In contrast, L_2 regularization prevents $|w_j|$ from getting too large by doing very large steps when that happens, but it does not really push it to be exactly 0.

In practice, L_2 regularization is often used by default unless there is a reason to want a sparse vector w (i.e., a w where many entries are exactly 0). Sparse vectors may be easier to understand, since we can just ignore certain features that correspond to a weight of 0. It is also possible to mix the two at the same time.

Chapter 4

Normal Equations for Linear Regression

So far, our recipe for coming up with machine learning problems has been the following:

1. Come up with a probabilistic story for how the data was generated.
2. Write down a loss function for which minimizing that function is equivalent to maximizing the likelihood of the data.
3. Optimize the loss function using gradient descent.

While gradient descent is undoubtedly the single most important and widely-used technique for optimizing loss functions in machine learning, it is not the only option. For linear regression, the optimal weight vector w actually has a closed-form solution given by the **Normal Equations**. In this chapter, we will sketch out how to derive this closed-form equation and what its implications are.

4.1 Deriving the Normal Equations

The basic strategy is something you may remember from your calculus class. We want to find the minimum of a convex function. We can do this by taking the gradient and setting it equal to zero. Recall the intuition for this: At the minimum of the function, there's no direction you can step in that would further decrease the value. That must mean that the derivative in every direction is zero, i.e., the gradient is zero.

So, let's start by writing down the gradient from the last lecture and setting it equal to zero.

$$\nabla_w L(w) = \frac{1}{n} \sum_{i=1}^{n} 2 \cdot \left(w^\top x^{(i)} - y^{(i)}\right) \cdot x^{(i)} = 0 \tag{4.1}$$

$$\sum_{i=1}^{n} w^\top x^{(i)} \cdot x^{(i)} = \sum_{i=1}^{n} y^{(i)} \cdot x^{(i)} \tag{4.2}$$

Keep in mind that both sides of this equation are vectors.

It turns out that both sides can be simplified quite a bit. To do this, I'm going to define the matrix $X \in \mathbb{R}^{n \times d}$ such that its i-th row is $x^{(i)}$. (Recall that n is the number of training examples and d is the number of features.) X is called the **design matrix**. Similarly, let's define the vector $y \in \mathbb{R}^n$ to be the vector of $y^{(i)}$'s.

Let's start by simplifying the right-hand side. This equation looks quite a bit like matrix multiplication—recall that one way to think about matrix multiplication is that you take a linear combination of *columns* of the matrix weighted by the entries of the vector. Since $x^{(i)}$'s are rows in X, but our expression is a linear combination of columns, we need to transpose X first. Thus, we can rewrite the right-hand side simply as $X^\top y$.

Now let's look at the left side. First I'm going to rearrange some pieces

$$\sum_{i=1}^{n} w^\top x^{(i)} \cdot x^{(i)} = \sum_{i=1}^{n} (x^{(i)\top} w) \cdot x^{(i)} \qquad \text{Dot product is symmetric} \tag{4.3}$$

$$= \sum_{i=1}^{n} x^{(i)} \cdot (x^{(i)\top} w) \qquad \text{Multiply scalar on right instead of left} \tag{4.4}$$

$$= \sum_{i=1}^{n} (x^{(i)} x^{(i)\top}) w \qquad \text{Matrix multiplication is associative} \tag{4.5}$$

$$= \left(\sum_{i=1}^{n} x^{(i)} x^{(i)\top} \right) w \qquad \text{Factor out } w \tag{4.6}$$

Essentially, we have carefully rearranged things so that we have a column vector (which can be thought of as a $d \times 1$ matrix) times a scalar (roughly speaking, a 1×1 matrix), so that we can safely multiply them as matrices and invoke the fact that matrix multiplication is associative.

Now, we just have to notice that

$$\sum_{i=1}^{n} x^{(i)} x^{(i)\top} = X^\top X. \tag{4.7}$$

I think the easiest way to convince yourself of this is simply to ask what the ij-th entry of each matrix is. In both cases, you'll find that it is equal to $\sum_{k=1}^{m} x_i^{(k)} x_j^{(k)}$. On the left side, $x_i^{(k)} x_j^{(k)}$ is the ij-th entry of each term in the sum. On the right side, the whole expression is the dot product of the i-th column of X (i.e., row of $X^\top$) with the j-th column of X.

Putting it all together. Okay, so what have we accomplished? Putting everything together, we have that setting the gradient equal to zero is equivalent to

$$(X^\top X)\theta = X^\top y. \tag{4.8}$$

Even if you don't remember this exact equation, you should remember the form—this is a linear equation with d equations and d unknowns.

How can we solve this? Well, we can invert the matrix $X^\top X$. This gives us the solution for w:

$$w = (X^\top X)^{-1} X^\top y. \tag{4.9}$$

4.2 Uniqueness and Non-invertibility

One thing that's immediately apparent from the Normal Equations formula is that we've implicitly assumed that $X^\top X$ is invertible. If that is true, then there is a unique w that minimzes the linear regression loss, and the Normal Equations find that w. But it's also possible for $X^\top X$ to not be invertible. Let's examine a few scenarios where that would happen.

More features than examples. One important cases is when there are more features than training examples. Recall that we use d to denote the number of features (i.e., the dimensionality of the data and of w) and n to denote the number of training examples. X is an $n \times d$ matrix. So, $X^\top X$ is a $d \times d$ matrix, but if $n < d$, then it can have rank at most n. This is because each column of $X^\top X$ is a linear combination of the columns of $X^\top$ (this is a basic property of matrix multiplication), and there are only n columns in $X^\top$ (it is a $d \times n$ matrix). Finally, recall that if a $d \times d$ matrix is not full-rank (i.e., its rank is $< d$), it is not invertible.

Dealing with non-invertibility To account for the fact that $X^\top X$ may not be invertible, the workaround is to use something called the pseudoinverse of $X^\top X$, denoted $(X^\top X)^+$. The pseudoinverse of a matrix M is equal to M^{-1} when M is invertible, and otherwise computes a generalization of the inverse. No matter whether $X^\top X$ is invertible or not, choosing

$$w = (X^\top X)^+ X^\top y$$

will always give **a** solution to the linear regression problem. If there are fewer examples than features, the resulting w will actually achieve 0 loss on the training set, which sounds good. Numpy has a special function (`numpy.linalg.pinv`) that will compute the pseudoinverse for you.

The problem from a machine learning is the issue of **generalization**. When $X^\top X$ is not invertible, there is not just a single w that minimizes the linear regression loss—there are many. Using the psuedo-inverse gives you one solution, but it's not the only one. We say that the training data **under-constrains** the choice of w: there is not enough information to choose between all these many w's that all look good based on training loss alone. But it's very likely that some of those w's have much better test loss than the other ones. In other words, when we have more features than examples, we will have **high variance**, following the bias-variance discussion from the previous chapter.

Redundant features. Another situation that can cause non-invertibility is the existence of redundant features. Suppose that the features at indices i and j that are identical—$x_i^{(k)} = x_j^{(k)}$ for all examples k. Then, $X^\top X$ would become non-invertible. To see this, note that this means that the i-th and j-th columns of X are identical. That means that the vector $v = e_i - e_j$, where e_i is the i-th basis vector, is in the null space, as Xv is just the i-th column minus the j-th column. A square matrix with non-trivial nullspace is not invertible.

The practical takeaway from this is that having redundant features can cause problems when solving for w. Essentially, when there are redundant features, there are many equally

good values of w (e.g., you could learn a very positive weight for w_i and very negative weight for w_j, and they would cancel out). This makes things potentially unstable!

Even if you're solving linear regression with gradient descent, redundant features are annoying, since they mean that your algorithm isn't converging to a single unique solution, but an entire subspace of solutions. Think of it this way: suppose you have two features that are *almost* identical. The inclusion of a single example could make you decide to rely on one feature vs. another feature. So the problem becomes very *sensitive* to small changes in the input dataset, which can lead to unintuitive behavior.

Chapter 5

Generative Classifiers and Naive Bayes

Today, we will learn a different approach to classification called generative classifiers. We will look at one prototypical example of a generative classifier, Naive Bayes. As before, we'll assume there are C classes, where C may equal 2 or may be > 2.

We will focus first on a particular variant of Naive Bayes that is used for text classification tasks (i.e., x is a piece of text). After that, we will look at the more general Naive Bayes method that applies to any feature vector x.

5.1 Generative Classifiers Overview

So far, everything we've seen is a **discriminative** classifier. That means we're only modeling $p(y \mid x)$. All of our probabilistic stories were only about how y was created given x. But there is another way.

In a **Generative** classifier, we model the joint process of generating the x's and the y's. This allows us to use Bayes Rule to compute the final $p(y \mid x)$:

$$P(y = k \mid x) = \frac{P(y = k)P(x \mid y = k)}{P(x)} = \frac{P(y = k)P(x \mid y = k)}{\sum_{k'=1}^{C} P(y = k')P(x \mid y = k')}.$$

Let's remind ourselves about how expressions like this work. The numerator has two terms:

- The prior distribution over y. For classification, this can usually be easily estimated just by counting the frequency of each label.
- The conditional probability of x given y. If x is some complicated object, this may not be easy to do! However, there are some helpful modeling assumptions we can make that are false, but still useful in practice. Remember that "All models are wrong, but some are useful."
- The denominator is simply a normalizing constant. For classification, it's very easy because we just sum over the possible labels.

At test time, we are given a new x. If we have a learned way to compute $P(y)$ and $P(x \mid y)$ for all possible values of y, then we can compute $P(y \mid x)$. We can then predict the most likely y, i.e. the k for which $P(y = k \mid x)$ is largest.

Now, we will discuss Naive Bayes, which is one family of methods for estimating $P(y)$ and $P(x \mid y)$.

5.2 Naive Bayes for Text Classification

Now let's focus on a classic generative classifier called Naive Bayes. First, let us assume we are doing **text classification**, which just means that our inputs x are pieces of text (I will also refer to these as "documents"). We write that each input $x = (x_1, \ldots, x_d)$ where each x_j is a word belonging to a vocabulary V (i.e., a set of possible words).

The key to Naive Bayes is to make the following simplifying assumption when modeling $P(x \mid y)$:

$$\textbf{Naive Bayes Assumption:} \quad P(x \mid y) = \prod_{j=1}^{d} p(x_j \mid y).$$

This says that conditioned on the label y, each word of x is sampled *independently*.

We will also invoke a second assumption that is specific to text documents: we will also assume that for every j, the distribution $p(x_j \mid y)$ is identical. In other words, the first word, second word, third word, etc. all are drawn from the same distribution. Together, these two assumptions define a particular version of Naive Bayes known as **Multinomial Naive Bayes**.

Note that we don't really believe that either of these assumptions is true. In fact, they are both clearly false, given the complexities of real human language. Nonetheless, for many practical tasks, these assumptions are good enough to give us useful algorithms. In particular, if we mainly expect that the set of words used in different documents will change from one label to the next, then Naive Bayes can detect these sorts of changes. In a hypothetical situation where two labels were different only because of the order of words used, for instance, Naive Bayes would fail to distinguish them, but you can imagine that such situations may be rare.

We can think of Naive Bayes as positing yet another probabilistic story. This time, instead of just thinking about how the y's are generated, we will model how a piece of text x and its corresponding label y are *jointly* generated:

1. The label is first sampled from the prior distribution $P(y)$.
2. Each of the d words of x are then sampled *independently* from the conditional distribution $P(x_j \mid y)$.

5.2.1 Parameters

What are the parameters of our model? In general, parameters are whatever we are trying to learn from data, and they determine the predictions made by our method. For logistic regression, our parameters were weight vectors w that got dot-producted with the inputs.

In the case of Naive Bayes, the parameters are just the probabilities associated with $P(y)$ and $P(x_j \mid y)$, which we will learn from our training data. So, the parameters are:

1. The prior term $p(y)$. If we have C classes, this is a distribution over the C classes, so the needed parameter vector is a vector $\pi \in \mathbb{R}^C$ where $P(y = k) = \pi_k$.

2. The posterior term $p(x \mid y)$. Given our Naive Bayes assumption, this boils down to knowing $p(x_j \mid y)$ for each possible value of y. This is a probability distribution over the vocabulary V, so it can be described by $|V|$ parameters, each of which denote the probability of a particular word $w \in V$ being generated conditioned on the label y. So, we have a total of $|V| \cdot C$ parameters. We can think of these as a $|V| \times C$ matrix called τ, where $\tau_{wk} = P(x_j = w \mid k)$.[1]

5.2.2 Learning with MLE

Okay, so how do we estimate these parameters? Let's apply the principle of maximum likelihood yet again (or equivalently, minimizing the the negative log-likelihood). But this time, we will be modeling the probability of the x's and the y's in our dataset.

$$\begin{aligned} L(\pi, \tau) &= -\sum_{i=1}^{n} \log P(x^{(i)}, y^{(i)}) \\ &= -\sum_{i=1}^{n} \log P(y^{(i)}) + \log P(x^{(i)} \mid y^{(i)}). \end{aligned}$$

It's apparent that the first term depends only on π and the second one only depends on τ, so we can find the optimal π and τ by looking only at the first term and second term, respectively.

Learning π. Let's start with the first one. Summing up all the first terms, we have:

$$-\sum_{i=1}^{n} \log P(y^{(i)}) = -\sum_{k=1}^{C} \text{count}(k) \log \pi_k$$

where $\text{count}(k)$ is the amount of times that label k appears among the n training examples.

When $C = 2$, you can see that this is exactly your homework problem! You have a "coin" that is either $y = 1$ or $y = 2$. So as you recall, the solution is to make

$$\pi_1 = \frac{\text{count}(y = 1)}{n}, \pi_2 = \frac{\text{count}(y = 2)}{n}.$$

This is intuitive—you just compute the empirical distribution over the classes.

Now what if $C > 2$? You can think of this as observing a bunch of dice rolls, as opposed to a bunch of coin flips. But it turns out that the solution is basically the same. Let's just

[1] Note that we're indexing into τ with actual words w rather than numerical indices. If this bothers you, you can equivalently imagine numbering all the words in the vocabulary from 1 to $|V|$ and using those numbers as indices.

consider a particular choice of k in isolation and think about the optimal choice of π_k. Well, we can think of every example as either being from class k or not from class k, and the probability of the former is π_k. So, this is just like the coin flip example, and so the MLE estimate for π_k is just

$$\boxed{\pi_k = \frac{\text{count}(y = k)}{n}}.$$

Learning τ. Now let's think about the other term, which only includes τ but not π. I won't go through the algebra since it's a bit tedious; instead I will argue that conceptually this is basically the same as the case with π.

First, recall that we have a completely separate probability distribution $P(w_j \mid y)$ for each choice of y. So we can think of our dataset being split into C smaller datasets, each of which only contains examples where $y = k$ for some k. The log-likelihood of all the data is sum of the log-likelihoods of each subset, and within each smaller dataset the only parameters that affect the likelihood are τ_{wk} for one choice of k. So we can just consider each value of k in isolation.

Now, we have assumed that each word of each example with $y = k$ is generated independently from the same distribution, namely the one where $P(x_j = w \mid y = k) = \tau_{wk}$. This is kind of like we have a bunch of observations from a $|V|$-sided dice.

Let $\text{count}(w, y = k)$ be the number of times we see the word w in an example whose label is k. By the same logic we used earlier, the MLE choice for τ_{wk} is

$$\tau_{wk} = \frac{\text{count}(w, y = k)}{\sum_{w \in V} \text{count}(w, y = k)}.$$

Note that the denominator is the total number of observations we have for this particular $|V|$-sided dice, i.e., the total number of all words we've seen in examples that have the label $y = k$.

5.2.3 Laplace Smoothing

If you just use the above model naively, you are very likely to get poor results. Why? Suppose you get a pretty long document to classify. At least one of the words might be such that in the training dataset, it happened to only appear when $y = 1$. And another word might be such that in the training dataset, it happened to only appear when $y = 2$. Now what happens? Well, $P(x \mid y = 1)$ will be 0, as will $P(x \mid y = 2)$! In other words, your model just says that your input is impossible, and will not tell you anything useful.

The important lesson here is this: **Never assign probability zero to events that could plausibly happen!**

We can fix this problem with **pseudocounts**. Let's just pretend that we've actually seen every possible word-label combination λ times before. This adds λ to every $\text{count}(w, y = k)$. As a result of seeing each word λ times with $y = k$, our total number of observations seen with $y = k$ increases by $\lambda \cdot |V|$. So our new formula becomes:

$$\tau_{wk} = \frac{\text{count}(w, y = k) + \lambda}{\left(\sum_{w \in V} \text{count}(w, y = k)\right) + |V| \cdot \lambda}.$$

λ is a hyperparameter of the method. In practice, you normally choose a very small λ. Note that you are even allowed to choose λ that is not an integer, for example $\lambda = 0.1$.

5.2.4 Working in log space

At test time, we're given a document x and want to estimate $P(y \mid x)$. This involves computing $P(x \mid y) = \prod_{j=1}^{d} P(x_j \mid y)$ for all possible values of y. Note that each term in this product is a probability that is probably quite small. When you multiply many small numbers together on a computer, you will run into **numerical underflow**: the numbers will become too small to represent with normal floating-point numbers, and your computed value for $P(x \mid y)$ will just be 0.

To solve this, we often "work in log space." That means that instead of computing $P(x \mid y)$, we will actually compute

$$\log P(x \mid y) = \sum_{j=1}^{d} \log P(x_j \mid y).$$

This sum will not underflow anymore, since we're just adding up a bunch of not-that-negative numbers.

In the end, to make a prediction, we can compute $\log\left(P(x \mid y)P(y)\right) = \log P(x \mid y) + \log P(y)$ for each possible value of y. We know that the value of y that makes this largest also has the largest $P(y \mid x)$, so that becomes our prediction.

5.3 Naive Bayes for General Feature Vectors

Naive Bayes is applicable to much more than just text classification. Let's now consider the case where x is just some generic feature vector $\in \mathbb{R}^d$, as in linear/logistic/softmax regression.

The Naive Bayes assumption still holds exactly as before—we assume that

$$P(x \mid y) = \prod_{i=1}^{d} P(x_j \mid y).$$

Note that here, each x_j is a different feature, not a different word, since we're no longer assuming that x's are documents.

5.3.1 Parameters and Learning

In the general case, we no longer assume that all of the d features are sampled from the same distribution (just that they're independently sampled). So, to model $P(x \mid y)$, we have to model each feature individually, i.e., we will have separate parameters for each feature.

- **Binary features.** Some features are binary, i.e., they are either 1 or 0. For example, if we are classifying black-and-white images, each pixel could be one component of x_j, i.e. a 28×28 image would lead to $28^2 = 784$-dimensional vector ($d = 784$). If x_j is

a binary feature, then for each possible $k \in \{1, \ldots, C\}$, $P(x_j \mid y = k)$ is completely described by a single parameter, which we can call p. To estimate p, we do the same thing we would do for estimating the probability of a coin landing heads:

$$p = \frac{\text{count}(x_j = 1, y = k)}{\sum_{i=1}^{n} \mathbb{I}[y^{(i)} = k]}$$

Note that here, for the denominator, we simply count the number of examples where $y = k$, as each feature shows up exactly once in each example; in contrast, for text, each example might contain a different number of words.

- **Categorical features.** Other features may take on more than two values. For example, suppose you are attempting to predict whether you will like a given song. You can frame this as binary classification—do you like the song or not? One relevant feature is the genre of the song, which can have M possible values (e.g., pop, country, classical, etc.). We can have one feature x_j whose value is the genre. Now, for each possible $k \in \{1, \ldots, C\}$, we would have a parameter vector $q \in \mathbb{R}^M$, where $q_m = P(x_j = m \mid y)$, the probability that x is of the m-th genre, conditioned on $y = k$. We would estimate this as:

$$q_m = \frac{\text{count}(x_j = m, y = k)}{\sum_{i=1}^{n} \mathbb{I}[y^{(i)} = k]}$$

- **Numerical features.** Note that you can also incorporate real-valued features into Naive Bayes by assuming some probability distribution that they are sampled from. For instance, you could assume that $P(x_j \mid y)$ is a Gaussian distribution, and estimate its mean and variance based on the training data.

Finally, note that for a single x vector, you could have features of all these types. That is not a problem, since we're fitting a separate independent distribution $P(x_j \mid y)$ for each coordinate j.

5.3.2 Laplace Smoothing

Laplace smoothing for binary and categorical features works in much the same way as with text. For binary features, we assume we have seen both $x_j = 0$ and $x_j = 1$ λ times with each label. This adds 2λ to the denominator:

$$p = \frac{\text{count}(x_j = 1, y = k) + \lambda}{(\sum_{i=1}^{n} \mathbb{I}[y^{(i)} = k]) + 2\lambda}$$

Similarly, for a categorical variable with M possible values, we assume we see each possible value λ times with each label, which adds $M \cdot \lambda$ to the denominator:

$$q_m = \frac{\text{count}(x_j = m, y = k) + \lambda}{(\sum_{i=1}^{n} \mathbb{I}[y^{(i)} = k]) + M\lambda}$$

5.4 Generative vs. Discriminative classifiers

Overall, how do generative classification methods like Naive Bayes compare with the discriminative methods we saw earlier in class? Here are a few general advantages and disadvantages.

Advantages of discriminative classifiers:

- Usually have higher accuracy, especially when the dataset is large. $P(y \mid x)$ is simpler to model than $P(x \mid y)$, and is thus in some sense easier to learn.
- Arbitrary feature preprocessing is allowed. We don't have to worry if the input features are related, since we don't make any assumptions of conditional independence between features.

Advantages of generative classifiers:

- Learning is easier because you don't have to use an optimization method like gradient descent—you just count feature occurrences.
- It is easy to handle missing input features (just exclude them from the computation).
- Each class is modeled separately, so there's no need to retrain everything when you add a new class. In contrast, adding one new class for softmax regression would require completely retraining the model.

Chapter 6

Non-parametric Methods

So far, all the methods we've studied are **parametric** methods: our goal was to learn some set of parameters whose dimensionality (i.e., the number of parameters to learn) is fixed regardless of the training dataset size. For example, in logistic regression, our parameter was a single vector w of dimension d. In Naive Bayes, we learned some number of parameters proportional to the number of possible labels and the size of our vocabulary.[1] In all these cases, we use the training data to learn parameters, then use the parameters to predict on the test data. Once learning is finished, the training data is no longer needed to make predictions.

Today we will start discussing **non-parametric** methods: methods for which our model requires storing the training data in memory. The predictions that non-parametric methods make are dependent directly on the training examples. Why would we want to do this? We'll start with the simplest non-parametric method called k-Nearest Neighbors. Then we'll study a more sophisticated family of methods with a similar intuition, called kernel methods.

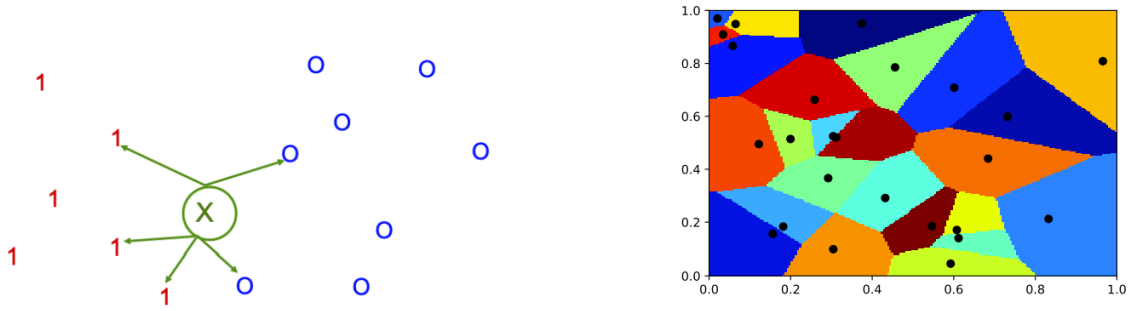
6.1 k-Nearest Neighbors

k-Nearest Neighbors (often abbreviated k-NN) is a classification algorithm motivated by a simple idea: similar examples are likely to have similar labels. Thus, when we get a new test input, we should predict the same label as the label for similar examples (i.e., "neighbors") from the training set.

Defining similarity. To define what examples are similar, we need to define some distance metric between examples. If our inputs x are vectors in $\mathbb{R}^d$, then one natural choice is to use the familiar Euclidean norm to define distance.

k-NN algorithm. The k-NN algorithm is very simple. The "training" step is simply to store the full training dataset in memory. At test time, we receive an input x that must be classified. We identify the k most similar points to x from the training dataset (i.e., the

[1] Technically one could argue that the vocabulary increases in size as the training set grows. However, at some point your vocabulary will stop growing as you add new documents, since eventually you will have already seen all valid English words.



(a) *(b)*

Figure 6.1: (a) An illustration of k-NN for $k = 5$. Since 3/5 of the neighbors are of the $y = 1$ class, the overall prediction is $y = 1$. (b) An illustration of how 1-nearest neighbor classifies points. For each example (black dots), the surrounding colored region is the set of points for which that dot is its nearest neighbor. Thus, all points in each colored region would be given the same label as the corresponding black dot by 1-NN. Figure taken from Kevin Murphy's *Probabilistic Machine Learning* textbook.

"nearest neighbors"), and return the most common label out of those nearest neighbors. Figure 6.1(a) shows an example with $k = 5$.

Effect of k. k is a hyper-parameter that defines how many neighbors we care about. If we choose $k = 1$, then we would just find the single closest example and return its label. This is reasonable, but is in some sense "risky" because you're hoping that one example is correct. However, if that example is unusual or labeled incorrectly in your training data, you will get all of its neighbors wrong. The advantage of choosing a larger k is that you can average over a larger number of examples, thus reducing the impact of individual outliers or wrongly labeled datapoints.

Bias and variance. As illustrated in Figure 6.1(b), k-NN can learn highly expressive decision boundaries. Compared with logistic regression, which makes a strong assumption that the decision boundary is linear, k-NN has much lower bias, as it does not make any such assumption.

On the other hand, k-NN can suffer from variance issues and may not always generalize well to test data. This is especially true in high-dimensional settings, i.e., if d is large, due to the "curse of dimensionality." The problem is that the amount of "space" grows exponentially with d, and thus in high dimensions the space of possible x's is very large. This means that the vast majority of possible test points x will be "far away" from all your training points (because the space is so vast). Essentially, this means that even your nearest neighbor is not that similar to you, so k-NN will struggle.

6.2 Kernel Methods

Now we will move on to a second type of non-parametric methods known as kernel methods. Kernel methods are widely used and often a good default choice when faced with a new classification problem. Pedagogically, kernel methods are also interesting because they combine ideas from linear classifiers (e.g., logistic regression) and k-Nearest Neighbors.

6.2.1 Motivation

We will continue with the same basic motivation we had with k-Nearest Neighbors: examples that are similar usually have the same labels. While this intuition is good, k-NN operationalizes this in quite a simplistic way. There are no parameters to tune, and very little way to add in regularization to control variance (besides changing k). So, we would like to combine the intuition of k-NN with the machinery we've established for methods like logistic regression, where we fit parameters from training data and can add regularization to manage the bias-variance trade-off.

6.2.2 Kernels and kernelized predictors

To achieve this goal, we will introduce the concept of a kernelized predictor. Given a training dataset $\{x^{(1)}, \ldots, x^{(n)}\}$ and test input x^{test}, a kernelized predictor makes a prediction by computing:

$$f(x) = \sum_{i=1}^{n} \alpha_i k(x^{(i)}, x^{\text{test}}).$$

Here, $k(x, z)$ is a **kernel function** that measures the similarity between point x and z. $k(x, z)$ should be large for points that are similar, and small for points that are dissimilar. $\alpha \in \mathbb{R}^n$ is a vector of learned parameters, one per training example. While kernelized predictors can be used for all sorts of tasks, we will focus on binary classification. We will predict the positive class if $f(x) > 0$ and the negative class otherwise. So, you can think of $f(x)$ as being analogous to the expression $w^\top x$ in logistic regression, since in logistic regression we classify positive if $w^\top x > 0$ and negative otherwise.

One of the most popular kernel functions is called the **radial basis function** (RBF) kernel. It is defined by the following equation:

$$k(x, z) = \exp\left(-\frac{\|x - z\|^2}{2\sigma^2}\right),$$

where σ^2 is a hyperparameter, also referred to as the "bandwidth."

This expression should look familiar to you, as it is basically the Gaussian pdf formula! When $x = z$, this function reaches its maximum possible value of 1. If x and z are very far apart from each other (in terms of Euclidean distance), then this tends to 0. The bandwidth parameter controls how quickly you go to 0, or in other words, how close x and z have to be to be considered "pretty similar." A smaller bandwidth means the points have to be very close together to be similar; a larger bandwidth means the points can be further apart and still be similar.

Note that even though we can refer to α as parameters of the model, this model is still a non-parametric, since the predictions depend on both α and the training data, and also the number of parameters in α grows with the size of the training data. "Parametric" means that the model is entirely described by a set of fixed-size parameters, and anything that is not parametric is "non-parametric."

6.2.3 A Second Look at Logistic Regression

So far, we've decided what form we want our model to have, and it has some parameters (the α_i's) that have to be learned. How can we learn these?

It turns out that the logistic regression algorithm we've already learned can also learn a kernelized predictor! To demonstrate this, I will show that logistic regression is already, in some sense, learning a kernelized predictor with a particular choice of kernel. By understanding exactly how this is true, we can figure out how to learn the kernelized predictor for any kernel function we want.

Let's focus on logistic regression with L_2 regularization. We can distill everything about this method into two important equations:

1. At training time, we repeatedly apply the gradient descent update rule:

$$w^{(t+1)} = w^{(t)} + \eta \sum_{i=1}^{m} \sigma(-y^{(i)} \cdot w^\top x^{(i)}) \cdot y^{(i)} \cdot x^{(i)} \tag{6.1}$$

   where λ is the hyperparameter that controls how much L_2 regularization we do.

2. At test time, when we get a new example x^{test}, we make a prediction based on $w^\top x^{\text{test}}$.

The key insight is that this algorithm, as well as many other typical linear classification algorithms, **only cares about the x's through their dot products with each other.** I'll make this more precise in a second. What this implies is that we can potentially run this same algorithm with a generalized notion of dot products, rather than the conventional one in which you add up the product of entries of two vectors. It turns out that the RBF kernel is one suitable replacement for the standard dot product, and this will give us a kernelized predictor.

But before we get ahead of ourselves, I want to show what I mean by only caring about dot products. Let's define another kernel, called the dot product kernel, whose value is simply $k(x, z) = x^\top z$. I claim that I can rewrite logistic regression as learning a kernelized predictor using this particular kernel.

The key step is to notice that w **is always a linear combination of $x^{(i)}$'s.** Here we are assuming that we initialize $w^{(0)}$ to the zero vector, which is routine. In every gradient update, we add some scalar times $x^{(i)}$ for each i. Thus, we can equivalently represent w with a vector $\alpha \in \mathbb{R}^n$, where n is the number of training examples, and we define $\theta = \sum_{i=1}^{n} \alpha_i x^{(i)}$.

What happens to our update rule now? We can directly update α instead of w:

$$\alpha_i^{(t+1)} = \alpha_i^{(t)} + \eta\sigma(-y^{(i)} \cdot w^\top x^{(i)}) \cdot y^{(i)}. \tag{6.2}$$

The rest is simply a matter of substituting in our new expression for w. In the training update, we get:

$$\alpha_i^{(t+1)} = \alpha_i^{(t)} + \eta\sigma\left(-y^{(i)} \cdot \sum_{j=1}^{n} \alpha_j x^{(j)\top} x^{(i)}\right) \cdot y^{(i)} \tag{6.3}$$

$$= \alpha_i^{(t)} + \eta\sigma\left(-y^{(i)} \cdot \sum_{j=1}^{n} \alpha_j k(x^{(j)}, x^{(i)})\right) \cdot y^{(i)}. \tag{6.4}$$

At test time, the output logit becomes:

$$w^\top x^{\text{test}} = \sum_{i=1}^{n} \alpha_i k(x^{(i)}, x^{\text{test}}). \tag{6.5}$$

To summarize:

1. We noticed that after every step of gradient descent, the current α is always a linear combination of the $x^{(i)}$'s.

2. Thus, we can represent α in terms of a vector of n numbers, one for each $x^{(i)}$ (we called this vector α).

3. We can then rewrite logistic regression so that we update α directly (which causes a corresponding update in w).

4. When we rewrite things in this way, we see that we don't actually need to know any of the $x^{(i)}$'s, as long as we know their dot product with every other $x^{(j)}$ as well as their dot product with any test example x^{test}.

Note that this is a different algorithm than standard logistic regression, since we maintain a vector $\alpha \in \mathbb{R}^n$ rather than $w \in \mathbb{R}^d$. This new version of logistic regression is called **kernel logistic regression**. When you use the dot product kernel, kernel logistic regression makes identical predictions to normal logistic regression. But, we don't have to use the dot product kernel only! We can choose a different kernel function, such as the RBF kernel, and run the exact same algorithm. This will learn α_i's that are suitable for this new kernel.

Why does this work? It turns out that the RBF kernel actually *does* compute a dot product between two vectors, just in a different feature space that is a transformation of the original feature space. This is why it is OK to simply replace every occurrence of dot product with the RBF kernel. Next time we will explain what this means, and use this observation to better understand why kernels are useful and come up with other useful kernels.

6.2.4 Kernels compute dot products in a different feature space

You're probably wondering what the point of this exercise was. Let's start with the following scenario. Recall from our linear regression discussion that we can apply any transformation to all of our x's to create a new, more complicated set of features. You can think of this transformation as a function ϕ that takes in an original x and outputs a larger vector with

more features concatenated on. For example, you could try to add all *quadratic* features: for every pair of indices $i, j \in \{1, \ldots, d\}$, you could add the feature $x_i x_j$. This adds expressivity to the model because it previously could only represent linear functions like $ax_i + bx_j$.

But if you do this, then suddenly you will have $O(d^2)$ features. If you wanted to add cubic features, you would need $O(d^3)$ features. In general, if you wanted all polynomials up to degree q, you would need $O(d^q)$ features. This could get very large very quickly!

Now let's look back at our new version of logistic regression. It tells us that we never actually have to construct these enormous feature vectors, as long as we can figure out what the dot product between two of those feature vectors is. Okay, you might be saying, how can we compute the dot product of two vectors without actually writing down those vectors? It turns out that for polynomial features of degree q, there is a nice trick: we can just use $k(x, x') = (x^\top x' + 1)^q$!

The general, formal statement is that $(x^\top x' + 1)^q = \phi(x)^\top \phi(x')$ for some feature function $\phi(x)$ whose entries include a term proportional to every monomial of x of degree up to q.[2] On Homework 2, you will prove this for the special case of quadratic features (i.e., $q = 2$) and when the dimensionality of the original x is also 2 (i.e., $d = 2$).

This trick of using kernels as an efficient way to represent a large feature space is often called the **kernel trick**.

What about the RBF kernel? It turns out that the RBF kernel corresponds to an *infinite-dimensional* feature vector. In Homework 2, you will show this for the special case where $d = 1$. Without the kernel trick, it is literally impossible to actually implement logistic regression using these features, since you would have to instantiate an infinite-dimensional vector. However, the kernel trick makes using the RBF kernel possible.

Runtime analysis. It's worth taking a moment to compare the runtime of standard logistic regression with kernel logistic regression using polynomial features of degree q.

In normal logistic regression, the most expensive part is computing dot products of these massive $O(d^q)$-dimensional vectors. This takes time $O(d^q)$. Each gradient step is just a single loop over the data, so a training iteration takes time $O(nd^q)$. At test time, you just compute one dot product, so making a prediction takes $O(d^q)$.

In kernel logistic regression, each training iteration must update each of the n components of α. Each update involves n computations of the kernel $k(x^{(i)}, x^{(j)})$. Using the kernel trick, each kernel evaluation can just be done by taking the dot product between $x^{(i)}$ and $x^{(j)}$, then doing some arithmetic on top (i.e., adding 1 and raising to the q-th power), so this is $O(d)$ in total. So, overall each iteration takes $O(n^2 d)$ time. At test time, you have to compute the kernel between the test input and every training example, which takes $O(nd)$ time.

So, using the kernelized method gives us a much better dependence on d, at the cost of quadratic runtime in the number of training examples for training (instead of linear) and linear runtime in the number of training examples for testing (instead of constant). Thus, kernel methods can be effective when your dataset is not too big, but the $O(n^2)$ runtime becomes prohibitive with very large datasets.

[2] A monomial is just a product of variables, so this says that it includes every way to multiply up to q coordinates of x together. E.g. $x_1^3 x_2 x_3^5$ is a monomial of degree $3 + 1 + 5 = 9$.

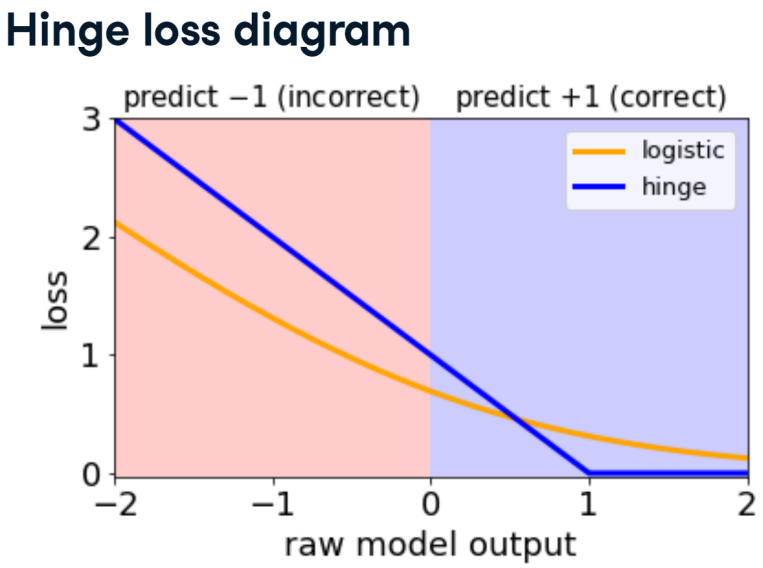


Figure 6.2: A comparison of hinge loss used in SVM and log-sigmoid loss used for logistic regression. (Note: what's called "raw model output" in this figure is what we've been calling the margin.)

6.3 Support Vector Machines

While I have focused on kernel logistic regression for pedagogical purposes, in practice you are much more likely to see a different (but highly related) linear classification method used with kernels called **Support Vector Machines** (SVM).

6.3.1 SVM without Kernels

There are many equivalent formulations of SVMs. In my opinion, the simplest one is to view SVM as a slight modification of logistic regression with L_2 regularization. In particular, SVMs without kernels are linear classifiers just like logistic regression, and they can be viewed as minimizing the following loss function:

$$L(w) = \left(\frac{1}{n}\sum_{i=1}^{n}[1 - y^{(i)}w^\top x^{(i)}]_+\right) + \lambda\|w\|^2.$$

The notation $[z]_+$ denotes the "positive" part of the number z and is simply equal to z if $z > 0$ and 0 otherwise. You can also equivalently write $[z]_+ = \max(z, 0)$.

This should be quite reminiscent of logistic regression! In logistic regression, we applied the log-sigmoid function to the margin $yw^\top x$. For SVMs, we apply the **hinge loss** function, defined as $f(z) = [1 - z]_+$, to the margin. We can plot the hinge loss and log-sigmoid functions on the same plot to understand the difference, as shown in Figure 6.2.

Both log-sigmoid and hinge loss want to encourage the margin to be large, so the larger your margin, the smaller the loss. For log-sigmoid, you can never reach exactly 0 loss, so you're incentivized to always make the margin higher. On the other hand, for hinge loss, once

your margin is ≥ 1, you get exactly 0 loss and there's no incentive to improve it further. Going the other direction, both functions are become linear in the margin as the margin tends towards negative infinity.

6.3.2 A kernelized loss function

Let's see how we can kernelize the SVM objective. While we can also try to kernelize the gradient descent step like we did with logistic regression, it turns out that with SVM's the best way to minimize the loss is actually not with normal gradient descent but with some more complicated optimization methods. I won't get into those here, so instead we'll just show how the *objective function* can be kernelized.

First, as before we can note that the optimal solution to w will be a linear combination of the $x^{(i)}$'s. One way to argue this is to again look at the gradient, as we did for logistic regression, and show that each gradient update always adds scalar multiples of the $x^{(i)}$'s and nothing more.[3]

Now that we know the optimal w will be a linear combination of $x^{(i)}$'s, we are justified in substituting

$$w = \sum_{i=1}^{n} \alpha_i x^{(i)}$$

for $\alpha \in \mathbb{R}^n$, and viewing the loss as a function of α rather than w.

Applying that substitution everywhere leads to the following:

$$\begin{aligned}
L(\alpha) &= \frac{1}{n}\sum_{i=1}^{n}\left[1 - y^{(i)}\left(\sum_{j=1}^{n}\alpha_j x^{(j)}\right)^\top x^{(i)}\right]_+ + \lambda\left\|\sum_{j=1}^{n}\alpha_j x^{(j)}\right\|^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\left[1 - y^{(i)}\sum_{j=1}^{n}\alpha_j x^{(j)\top} x^{(i)}\right]_+ + \lambda\left(\sum_{i=1}^{n}\alpha_i x^{(i)}\right)^\top\left(\sum_{j=1}^{n}\alpha_j x^{(j)}\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left[1 - y^{(i)}\sum_{j=1}^{n}\alpha_j x^{(j)\top} x^{(i)}\right]_+ + \lambda\left(\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j x^{(i)\top} x^{(j)}\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left[1 - y^{(i)}\sum_{j=1}^{n}\alpha_j k(x^{(j)}, x^{(i)})\right]_+ + \lambda\left(\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j k(x^{(i)}, x^{(j)})\right),
\end{aligned}$$

where in the last line we have made the substitution $x^\top z = k(x, z)$. Now we can replace any generalized definition of dot product, such as one that involves the quadratic or RBF kernel, for the original dot products between examples. Note also that just as in kernelized logistic regression, we have $O(n^2)$ runtime in the number of training examples, as there are these double sums over the training dataset.

[3] Gradient descent is still a valid way to minimize the SVM objective and will converge to the global minimum.

6.3.3 What are support vectors?

SVMs get their name from the concept of support vectors. Let's consider a simple, linearly separable classification dataset as shown in Figure 6.3.

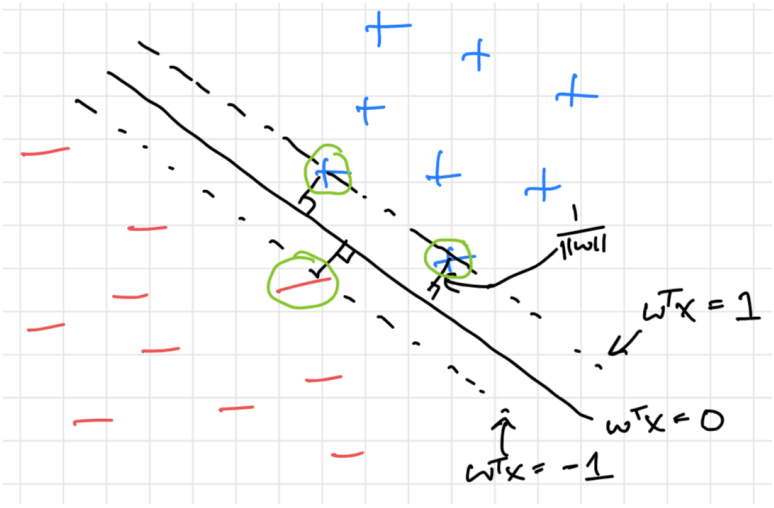


Figure 6.3: Illustration of a support vector machine. The decision boundary, defined by $w^\top x = 0$, is chosen to balance two considerations. The first is ensuring that all examples have margin of at least 1 (illustrated by the lines $w^\top x = 1$ and $w^\top x = -1$). The second is minimizing the L_2 norm of w, which corresponds to maximizing the distance (which is $1/\|w\|$) between the hyperplanes where the margin is 1 and the decision boundary. Thus, the SVM wants to classify all points correctly using a decision boundary along which the two classes are furthest apart. The resulting decision boundary is only dependent on the points with margin ≤ 1 (circled in green), which are known as the support vectors.

Here, the SVM objective will encourage all the margins to be at least 1, but there's no reason to go beyond 1. The decision boundary, defined by $w^\top x = 0$, is parallel to other hyperplanes defined by $w^\top x = -1$ and $w^\top x = 1$, which correspond to the place where the margin becomes 1 for negative examples and positive examples, respectively. It turns out that the distance between two adjacent parallel hyperplanes in the diagram is just $1/\|w\|$. So, minimizing the L_2 regularization term is equivalent to maximizing this distance.

Finally, note that the only points that really matter are the ones highlighted in green, which have margin of ≤ 1. These are referred to as **support vectors**. The other points are classified correctly with margin > 1, and don't have any effect on the loss. For example, if you took one of those points and moved it around a little bit, the optimal decision boundary wouldn't change. It turns out that at the minimum of the SVM objective function, the learned w is a linear combination of *just the support vectors*. No other datapoints contribute to w.

If we now look at the kernelized form, this means that $\alpha_i = 0$ for all $(x^{(i)}, y^{(i)})$ that are not support vectors. Thus, predictions of the SVM only depend on the dot products with the support vectors. When we want to predict on a new test point, we just have to compute its kernel with all the support vectors, which can be much less than the number of training examples, n. This is the big reason why SVMs and kernels are a good match!

Let's contrast this with logistic regression. Since the log-sigmoid loss never reaches 0, even points that are very far on the correct side of the decision boundary still affect the overall loss a little bit. Since every datapoint influences the loss a little bit, the final w you learn is going to depend on all of the datapoints, so making a prediction will take the full $O(n)$ time.

6.3.4 Optimization

It turns out the fact that the hinge loss is piecewise linear enables some more complex optimization techniques that we won't discuss here. But the important thing to know is that these optimization techniques also only care about dot products between examples, and thus we can use the same kernel trick to replace dot products with kernels.

Chapter 7

k-Means Clustering

7.1 Introduction to Unsupervised Learning

Let's take stock of where we are. In the first half of the class, we saw a variety of methods for doing supervised learning. In all of them, the basic recipe was the same. We obtained a training dataset that had examples of an input x paired with a correct output y, which we denoted $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$. We then trained some model with this data such that it can take in a new input x_{test} and predict what the correct output y_{test} should be for that previously unseen input.

Today we will start learning about *unsupervised* learning. In unsupervised learning, we are no longer concerned with learning a mapping from inputs to outputs. In fact, we no longer have a "correct output" associated with each example. Our dataset is only x's—we write it simply as $\{x^{(1)}, \ldots, x^{(n)}\}$. What can we do with this data? In unsupervised learning, we hope to learn something about the **structure** of our data. We will cover three types of unsupervised learning:

- **Clustering**: Are my datapoints naturally grouped into multiple **subgroups**?
- **Dimensionality Reduction**: For data that is high-dimensional, is there a **low-dimensional subspace** that actually captures (almost) everything that's happening in the data?
- **Embeddings**: For data about objects that are not vectors, can we learn a vector for each object that captures the **similarity relationships** between objects?

7.2 Clustering

The first type of unsupervised learning problem we will study is called *clustering*. The idea of clustering is that we are trying to learn **subgroup structure** within the dataset. A clustering algorithm takes in a dataset D that consists solely of x's, i.e., $D = \{x^{(1)}, \ldots, x^{(n)}\}$. It then assigns each x to one of k possible **clusters**, or subsets of the data. Formally, we can write the output of the clustering algorithm as an **assignment** $z_1, \ldots, z_n$ where each $z_i \in \{1, 2, \ldots, k\}$ denotes the cluster assigned to datapoint $x^{(i)}$. We can say that the "j-th

cluster" is just the set of all examples $x^{(i)}$ where $z_i = j$. For brevity, I will write $z_{1:n}$ to refer to the complete list of assignments $[z_1, \ldots, z_n]$.

Note that the maximum number of clusters k is a hyperparameter of the algorithm. In other words, before we run the clustering algorithm, we should already have a guess of how many total clusters there are in the data. If we don't know ahead of time, there are some heuristics we can use to choose a good value of k, just as we had ways of choosing hyperparameters in supervised learning. We'll discuss these after introducing our first clustering algorithm.

7.3 k-Means Clustering

Now we will dive into the most famous clustering and widely-used clustering algorithm, k-means clustering.

7.3.1 Setup

The main idea of k-means clustering is that we can represent each cluster of the dataset with a "cluster mean" or "cluster centroid," denoted μ_j for the j-th cluster. μ_j represents the center of all the points in cluster j. In order for something to be a good clustering, we should have all examples $x^{(i)}$ to be close to the corresponding cluster mean, μ_{z_i}.

To formalize this intuition, we can write down a loss function that we want to minimize in order to find the best cluster assignment (just like we did whenever we wanted to derive a new supervised learning algorithm). We want to minimize the sum of the distances between each point and its cluster centroid, which we can write as

$$L(z_{1:n}, \mu_{1:k}) = \sum_{i=1}^{n} \|x^{(i)} - \mu_{z_i}\|^2.$$

Let's unpack this equation. On the left hand side, note that our loss L is a function of both our assignment z_i for each $i = 1, 2, \ldots, n$, and our cluster centroid μ_j for each $j = 1, 2, \ldots, k$. When we run the k-means algorithm, we will want to find values of both $z_{1:n}$ and $\mu_{1:k}$ that minimize this loss. This accomplishes the main goal of a clustering algorithm—assigning each point to a cluster—but as a byproduct it also computes centroids for each cluster.

On the right hand side, we compute the squared Euclidean distance between each example $x^{(i)}$ and its assigned cluster centroid μ_{z_i}. This sort of loss function is sometimes referred to as the *reconstruction error*. The reason is that we can view clustering as a way of simplifying the dataset—instead of remembering the exact location of each datapoint, we can just remember the cluster centroids and what cluster each datapoint belongs to. If we did that, how well could we "reconstruct" the original dataset? Well, our best guess for the location of each datapoint is just that it's roughly located at its corresponding cluster centroid. So the error of this reconstruction is exactly how far away each original datapoint is from its corresponding cluster centroid.

7.3.2 Algorithm

Now that we have a loss function, we need to find an algorithmic way to minimize it. First, let's note that we cannot use gradient descent in this case. Why not? gradient descent requires us to compute the gradient of the loss with respect to all of our input variables. However, the z_i's are not continuous variables at all—they're a discrete choice of one of the k clusters. There's no way to slightly perturb these variables, so the concept of a derivative with respect to z_i doesn't make any sense.

Instead, we'll choose a different general strategy known as **alternating minimization**. You can think of alternating minimization as a high-level strategy rather than an algorithm itself. All it means is that if we have a loss function that depends on two variables, we can first minimize it with respect to the first variable while holding the second fixed, then minimize it with respect to the second variable while holding the first fixed, and repeat. This is often an effective strategy even if we just start from a random guess for both variables. In this way, philosophically there are some similarities between alternating minimization and gradient descent: in both cases, we start with a possibly bad guess, then go through many iterations of improvement.

In the case of k-means, alternating minimization will alternate between the following two steps until convergence (i.e., until the values stop changing):

1. Given our current guess for $\mu_{1:k}$, update our guess for $z_{1:n}$ to be the one that minimizes L.

2. Given our current guess for $z_{1:n}$, update our guess for $\mu_{1:k}$ to be the one that minimizes L.

The nice thing about k-means is that it turns out that both of these updates have very simple closed-form solutions. Let's see the full algorithm in detail. In addition to seeing how we do each update, we also need to know how to initialize our guess.

Choosing an initial guess for $\mu_{1:k}$. Note above that in the alternating minimization loop, we chose to start with updating $z_{1:n}$ given a guess for $\mu_{1:k}$. That means that to get everything started, we just need an initial guess of $\mu_{1:k}$ (and not $z_{1:n}$). There are several possible strategies here, but one simple one is simply to choose each μ_j to be a different random example from the dataset. This is reasonable because we start each cluster in a different place, even if it's not at the optimal position. The further steps of alternating minimization will refine this guess

Updating $z_{1:n}$. Given our current guess of the cluster centroids $\mu_{1:k}$, what is the best choice of z_i? Well, for each example, we should just assign it to the cluster centroid it is closest to. Formally, we can write

$$z_i = \arg\min_{j=1,\dots,k} \|x^{(i)} - \mu_j\|^2.$$

In other words, we compute the distance between $x^{(i)}$ and each μ_j, and choose whichever j gives the smallest distance as our new value of z_i.

Updating $\mu_{1:k}$. Now, given our current guess of the cluster assignments, what is the best choice of $\mu_{1:k}$? Intuitively, for each $j = 1, \ldots, k$, there are a bunch of points that have been assigned to cluster j. To minimize the distance between all those points and μ_j, we should want μ_j in the middle of all those points, so it should be something like the average of all those points. In fact, it turns out we have cleverly chosen our loss function so that μ_j should exactly be the average of the points assigned to cluster j.

We can prove this mathematically. First, let's rewrite the loss function by grouping all of the examples in terms of which cluster they're assigned to:

$$\sum_{i=1}^{n} \|x^{(i)} - \mu_{z_i}\|^2. = \sum_{j=1}^{k} \sum_{i:z_i=j} \|x^{(i)} - \mu_j\|^2.$$

Now, for each j, μ_j appears in exactly one of these terms, so we can focus on one j at a time. In general, for a fixed choice of j, we need to minimize

$$\sum_{i:z_i=j} \|x^{(i)} - \mu_j\|^2.$$

Now we can take the gradient with respect to μ_j and set it equal to 0:

$$\begin{aligned}
\nabla_{\mu_j} \sum_{i:z_i=j} \|x^{(i)} - \mu_j\|^2 &= \sum_{i:z_i=j} 2(x^{(i)} - \mu_j) \cdot \cancel{-1} = 0 \\
\sum_{i:z_i=j} x^{(i)} &= |\{i : z_i = 1\}| \cdot \mu_j \\
\mu_j &= \frac{1}{|\{i : z_i = 1\}|} \sum_{i:z_i=j} x^{(i)}
\end{aligned}$$

This final expression is simply the average of all $x^{(i)}$'s assigned to cluster j.

Convergence. In the full k-means algorithm, we keep alternating between these two minimization steps until we reach a fixed point—a point where the updates to $z_{1:n}$ don't change anything, and so the updates to $\mu_{1:k}$ also don't change anything.

Are we guaranteed to have reached the global minimum of the k-means loss? Not necessarily. Each step of alternating minimization reduces the loss, but we can get stuck at a local optimum. In fact, with the same dataset and different random initializations of the cluster centroids, you can arrive at different clusterings. It is not uncommon to do multiple random restarts (i.e., just try multiple random initializations) and choose the best one as the final clustering.

Comparison with supervised learning. Overall, how does this compare with supervised learning? In some ways, you might think that k-Means clustering reminds you a bit of classification. For each input $x^{(i)}$, we make a "prediction" of some integer between 1 and k, almost as if we were doing multi-class classification with k classes. Moreover, as we saw in the k-Means algorithm, the "prediction" z_i for each $x^{(i)}$ is completely determined by the

choice of $\mu_{1:k}$; thus, we can think of $\mu_{1:k}$ as learned parameters, and based on the learned parameters we make some predictions.

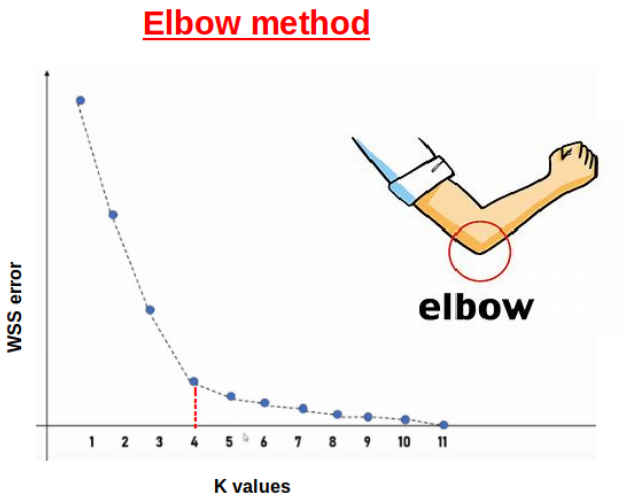
However, there are also a couple key differences to keep in mind. First, in classification we have explicit supervision of which class each training example belongs to. In clustering, we do not get told this. Second, in classification our goal is to learn a set of parameters that helps us make predictions on new test examples. In clustering, our goal is to divide up the data we are already given, or to make "predictions" on the training data. We usually do not think about a separate test dataset for clustering (more on that in the next section).

7.3.3 Choosing k

Note that the entire k-means clustering assumes that we have chosen a good value for the hyperparameter k (the number of clusters) ahead of time. In some applications, we may simply want to divide the dataset up into a fixed number of subgroups, so this is fine. But if we actually want to understand the structure of the data, we also need to determine how many distinct clusters are actually present in the data.

Let's start by thinking about how we did this for supervised learning. To choose hyperparameters, we tried many different values and measured the loss on a development set. Whichever hyperparameter value led to the lowest loss was considered the best. Does this also work for k-means? Unfortunately, no. In general, the k-means loss function will always decrease if you add more clusters. This is just because if there are more cluster centroids, then the closest centroid is almost certainly closer than if there were fewer centroids.

Instead, an informal rule we can use is called the "elbow criterion." To apply the elbow criterion, you try running k-means with different values of k and make a plot with the k-means loss on the y-axis and k on the x-axis. This will create a curve that slopes downwards to the right. The point where the curve bends and starts becoming flat is called the "elbow," and is usually a reasonable choice for k, because it signifies a point where increasing k no longer decreases the loss by that much.



Chapter 8

Gaussian Mixture Models and Expectation-Maximization

k-Means is a popular and simple clustering algorithm, but it has its drawbacks. Importantly assumes that each cluster can be described by a single parameter, namely its centroid. What if some clusters have different shapes? Or if some clusters are a lot larger than other clusters? k-Means doesn't allow us to learn these nuances. It assumes that all clusters are spherical, since it measures proximity to each cluster centroid with normal Euclidean distance. And it assumes all clusters are the same size, since it always assigns a point to the closest cluster centroid—what if one centroid is slightly further away but that cluster is much larger?

The Gaussian Mixture Model (GMM) is a probabilistic clustering approach that allows us to learn cluster shapes as well as centroids and assignments. While similar in some ways to k-Means, GMM is more complex and mathematically involved. At a high level, in GMM we will posit a probabilistic story for how our dataset was created, reminiscent of some of our early supervised learning lectures. We will use this to define a loss function, and we will fit the parameters of our clusters (i.e, their location, shape, and size) by minimizing this loss function. Fitting these parameters will enable us to infer the best assignment of each example to a cluster.

8.1 Multivariate Gaussians

In a GMM, each cluster is defined by a probability distribution, rather than a single centroid. In particular, each cluster has an associated multivariate Gaussian distribution. Before we talk about GMMs, let's talk about multivariate Gaussians.

We have already seen many examples in class of a single number $x \in \mathbb{R}$ being distributed as a Gaussian. It turns out there is a natural extension to multiple variables, i.e., to say that a vector $x \in \mathbb{R}^d$ is distributed as a "multivariate Gaussian." Figure 8.1 shows samples from a 2-dimensional multivariate Gaussian. Just like the familiar one-dimensional bell curve, the multivariate Gaussian has highest probability density in the center of the distribution, and gets lower the further you go from the middle.

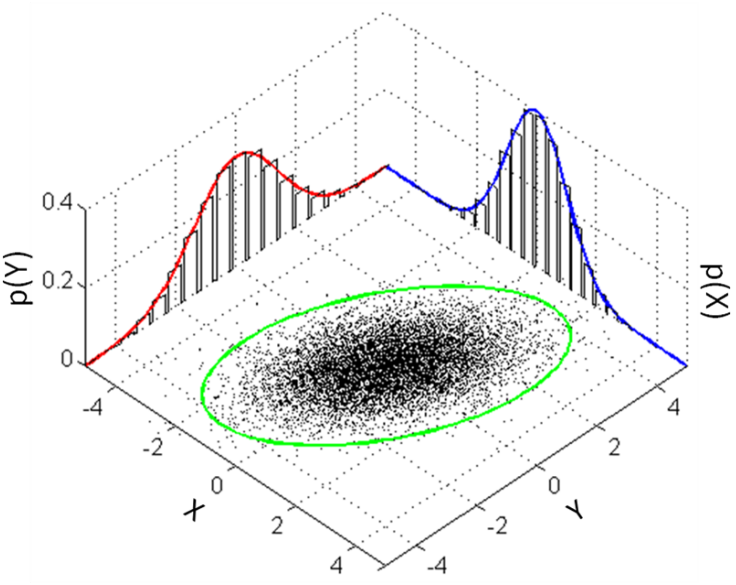


Figure 8.1: Samples from a 2-dimensional multivariate Gaussian distribution, with the marginal (single-variable) probability distributions displayed on the sides. Each coordinate of the multivariate Gaussian is distributed normally. The green ellipse denotes where most of the probability mass is.

Covariance. Before we see the full multivariate Gaussian, let's review covariance, a key statistical quantity whenever we are thinking about multiple random variables at the same time. Recall that the variance of a random variable X is defined as

$$\text{var}(X) = \mathbb{E}[(X - \mu)^2]$$

where $\mu = \mathbb{E}[X]$. This measures the expected squared distance between X and its mean value. Analogously, the **covariance** of two random variables X_1 and X_2 is

$$\text{cov}(X_1, X_2) = \mathbb{E}[(X_1 - \mu_1)(X_2 - \mu_2)]$$

where $\mu_1 = \mathbb{E}[X_1]$ and $\mu_2 = \mathbb{E}[X_2]$.

If we have d random variables $X_1, \ldots, X_d$, the **covariance matrix** Σ is the $d \times d$ matrix where Σ_{ij} is the covariance between X_i and X_j. Note that since the covariance between X_i and X_j is the same as the covariance between X_j and X_i, the covariance matrix is always symmetrical. Further, from the definition of covariance, we can see that the covariance of X with itself is just the variance of X, so the diagonal entries Σ_{ii} are simply equal to $\text{var}(X_i)$ for each i.

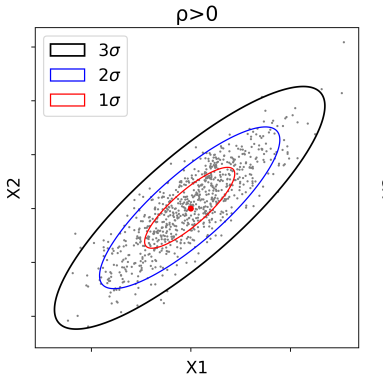
The covariance between two random variables is closely related to their correlation. If the covariance is positive, the variables are positively correlated; if it's negative, they are negatively correlated. In fact, the correlation between X_1 and X_2 is simply

$$\frac{\text{cov}(X_1, X_2)}{\sqrt{\text{var}(X_1) \times \text{var}(X_2)}}$$

So, a distribution with a covariance matrix of

$$\Sigma = \begin{pmatrix} 1 & 0.5 \\ 0.5 & 1 \end{pmatrix}$$

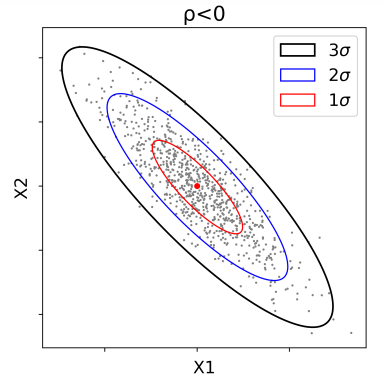
looks like this:



whereas distribution with a covariance matrix of

$$\Sigma = \begin{pmatrix} 1 & -0.5 \\ -0.5 & 1 \end{pmatrix}$$

looks like this:



Thus, the covariance tells us something about the "shape" of the distribution.

Matrix form. There is a nice matrix form for the covariance matrix. Suppose X is a vector of random variables of length d, and $\mu \in \mathbb{R}^d$ is the vector of the means for each X_i. Then the covariance matrix Σ can be written concisely as

$$\Sigma = \mathbb{E}[(X - \mu)(X - \mu)^\top].$$

Recall that whenever we see an expression like $vv^\top$, this is an *outer product* and it yields a $d \times d$ matrix whose ij-th entry is $v_i v_j$. So in our case, the ij-th entry of the matrix is just $(X_i - \mu_i)(X_j - \mu_j)$. You can see that the expected value of this exactly matches the covariance formula we had above.

Defining the multivariate Gaussian. Now we are ready to define the multivariate Gaussian distribution. Recall that the univariate Gaussian is parameterized by two parameters: the mean μ and variance σ^2. The mean simply tells us where the middle of the bell curve is. The variance tells us how spread out it is—in other words, it controls the shape of the bell curve. Something similar applies in multiple dimensions. The d-dimensional multivariate Gaussian is parameterized by a mean vector $\mu \in \mathbb{R}^d$ and a covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$.

The probability density function for the multivariate Gaussian is

$$P(X = x; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}\sqrt{\det(\Sigma)}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right),$$

where $\det(A)$ denotes the determinant of the matrix A. You can see that if $d = 1$, this simplifies to the familiar univariate Gaussian probability density function.

It's not at all important to remember this formula. We are just using it because the multivariate Gaussian is a reasonable probability distribution whose mean is μ and covariance matrix is Σ. If we have some data we think has a given mean and covariance, we can model it with a multivariate Gaussian.

8.2 Gaussian Mixture Model

Now that we have the language to describe clusters of different shapes, we can write down a *probabilistic story* for how our dataset was generated. As in k-Means, we will assume we have k clusters, and our dataset is denoted as $D = \{x^{(1)}, \ldots, x^{(n)}\}$ where each $x^{(i)} \in \mathbb{R}^d$.

The parameters of our probabilistic story are as follows:

- $\pi_1, \ldots, \pi_k$: π_j represents the probability of any given point being from cluster j, or in other words how "common" or how "large" the j-th cluster is.
- $\mu_1, \ldots, \mu_k$: μ_j is the mean vector for cluster j (similar to the k-Means centroids)
- $\Sigma_1, \ldots, \Sigma_k$: Σ_j is the covariance matrix for cluster j. This determines the shape of each cluster.

Based on these parameters, we assume the data was generated in the following two steps:

- For each $i = 1, \ldots, n$, we sampled a random variable $Z_i \in \{1, \ldots, k\}$ where $P(Z_i = j) = \pi_j$. This denotes which cluster the i-th example is actually from.
- For each $i = 1, \ldots, n$, we sample the random variable X_i from a multivariate Gaussian with mean μ_{Z_i} and covariance Σ_{Z_i}. In other words, we sample X_i from the probability distribution associated with the cluster Z_i.

When we actually work with GMMs, we observe some values of the random variables X_i: X_1 is observed to have value $x^{(1)}$, and so forth. However, we never observe the values of the Z_i's directly: they are **latent** random variables ("latent" just means "unobserved"). In fact, you might notice that this probabilistic story is quite reminiscent of Naive Bayes,

except that in Naive Bayes our training data included observations for both sets of random variables—there were no latent variables.

Now, what is our goal? We have two things we need to accomplish. We need to learn good values for $\pi_{1:k}$, $\mu_{1:k}$, and $\Sigma_{1:k}$ that match the observed data. We also need to infer values for all the Z_i's, since that tells us which which cluster each point belongs to.

8.3 Inference in GMMs

Before we talk about learning, let's first discuss the simpler problem of **inference**. You can roughly think of "inference" here as meaning, how do we make a prediction given learned parameters? In general, the term "inference" is used a lot in probabilistic modeling to refer to computing the conditional distribution of some random variable given values of other random variables and the parameters of your probability distribution.

For GMMs, we observe values of X_i's and want to *infer* the conditional distribution over Z_i's. More formally, given an example $x^{(i)}$ and already-learned values for parameters $\pi_{1:k}$, $\mu_{1:k}$ and $\Sigma_{1:k}$, we want to estimate

$$P(Z_i \mid X_i = x^{(i)}; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}).$$

Or in English: once we have learned the cluster parameters, we have to decide for each example $x^{(i)}$, what is the probability that it belongs to each possible cluster.

Doing this involves a direct application of Bayes Rule. Our generative story tells us how to compute $P(X_i = x^{(i)} \mid Z_i = b)$ for each possible value $b = 1, \ldots, k$. So, Bayes rule says that for any $j = 1, \ldots, k$,

$$P(Z_i = j \mid X_i = x^{(i)}; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}) = \frac{P(Z_i = j)P(X_i = x^{(i)} \mid Z_i = j)}{\sum_{b=1}^{k} P(Z_i = b)P(X_i = x^{(i)} \mid Z_i = b)}.$$

Note that the bottom is simply $P(X_i = x^{(i)})$, which we get by marginalizing out (i.e., summing over) all the possible values for Z_i.

To actually compute the values for $P(Z_i \mid X_i = x^{(i)}; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k})$, we just have to plug in the expressions for $P(Z_i = b)$ and $P(X_i = x \mid Z_i = b)$. Recall that they are:

$$P(Z_i = b) = \pi_b$$
$$P(X_i = x \mid Z_i = b) = \frac{1}{(2\pi)^{d/2}\sqrt{\det(\Sigma_b)}} \exp\left(-\frac{1}{2}(x - \mu_b)^\top \Sigma_b^{-1}(x - \mu_b)\right).$$

So our overall expression is

$$P(Z_i = j \mid X_i = x^{(i)}; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}) = \frac{\frac{\pi_j}{\sqrt{\det(\Sigma_j)}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^\top \Sigma_j^{-1}(x^{(i)} - \mu_j)\right).}{\sum_{b=1}^{k} \frac{\pi_b}{\sqrt{\det(\Sigma_b)}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_b)^\top \Sigma_b^{-1}(x^{(i)} - \mu_b)\right)}.$$

Remembering this equation is not at all important, but it is important to understand the steps we did to arrive at it.

Hard inference. What I just described is sometimes referred to as "soft" inference because we produce a probability distribution over possible clusters for each example, rather than a hard decision of which cluster each example belongs to. To ultimately produce a "hard" assignment $z_{1:n}$ where each example is assigned to exactly one cluster, we can simply choose z_i be the cluster j that maximizes $P(Z_i = j \mid X_i = x^{(i)}; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k})$, for each $i = 1, \ldots, n$. In other words, we just assign each example to the cluster that we *inferred* was the most likely one. This is just like how in softmax regression, we obtain a predicted distribution over the possible classes, and to make a prediction we just choose the most likely class.

8.4 Learning GMMs with Expectation-Maximization

Finally, let's address the full problem of learning GMMs. Recall that we have two things we need to learn:

- The unknown parameters $\pi_{1:k}$, $\mu_{1:k}$, and $\Sigma_{1:k}$
- The latent variables $Z_{1:n}$.

There is a very general technique used in situations like this where we want to both learn unknown parameters of some probability distribution, and perform inference on latent variables. This is known as the **Expectation-Maximization** (EM) algorithm. EM alternates between two steps:

- **E-step**: Infer latent variable's distributions given current guess of parameters
- **M-step**: Choose value of parameters that best fit the data based on the current inferred distribution of latent variables.

Note that this is very reminiscent of k-Means. The E-step is similar to how we assigned each example to a cluster given the cluster centroids, while the M-step is similar to how we updated the cluster centroids based on the current assignment of examples to clusters. In fact, it is possible to view k-Means as performing a type of "hard-EM" (because it makes hard assignments of examples to clusters, rather than having a distribution over possible clusters for each example) for a particular type of probabilistic model.

Let's now walk through the two steps of EM for GMMs.

E-step. This was essentially covered in the previous section. For each $i = 1, \ldots, n$, given our current guess of the parameters, we infer the distribution over Z_i. For ease of notation, let's define a matrix $R \in \mathbb{R}^{n \times k}$ where

$$R_{ij} = P(Z_i = j \mid X_i = x^{(i)}; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}).$$

Basically, each row of R stores the inferred probability distribution over what cluster example i was most likely to belong to (under our current guess for the parameters). R is the output of the E-step, and will be fed to the M-step.

M-step. After the E step, we have access to two things: (1) The actual values of all the X_i's, and (2) the inferred distribution for all the Z_i's. How can we use this to update our parameters?

Let's compare this with Naive Bayes. In Naive Bayes, we had actual values of all the X_i's and all the Y_i's (which are kind of like the Z_i's we have now). The best parameters were the ones that maximize the likelihood of the data, so we wrote down the log-likelihood of the data and maximized it.

In EM, we can't exactly do MLE since we don't know the actual values of Z_i's. But we have the values of the R_{ij}'s, which are very close. If we have $R_{i1} = 0.8$ and $R_{i2} = 0.2$, that says that we think example i has an 80% chance of being from cluster 1 and a 20% chance of being from cluster 2. So here's what we do: we pretend that our dataset has 0.8 examples where $X_i = x^{(i)}$ and $Z_i = 1$, and 0.2 examples where $X_i = x^{(i)}$ and $Z_i = 2$. Then we maximize the likelihood on this (slightly fictitious) dataset.

The loss function we get when we do this is called the **Expected Complete Log Likelihood** (ECLL). It is defined as follows:

$$\text{ECLL}(\pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}) = \sum_{i=1}^{n} \sum_{j=1}^{k} R_{ij} \log P(X_i = x^{(i)}, Z_i = j; \pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}).$$

Lets try to understand this. The quantity inside the log is simply the probability of observing $X_i = x^{(i)}$ and $Z_i = j$, i.e., observing the i-th example coming from cluster j. This is the "complete" log likelihood since it includes the probability of both X_i and Z_i, not just X_i. Now, how much do we care about the probability that $X_i = x^{(i)}$ and $Z_i = j$? Well, in our fictitious dataset, that occurred R_{ij} times. So we multiply that by R_{ij} and sum over all possible j. This is the "expected" part of the ECLL—we take the expectation over all possible values of Z_i, assuming Z_i is sampled from the distribution given by the R_{ij}'s.

If you're wondering why it's correct to multiply the complete-log-likelihood with R_{ij}, think about what we would do if $R_{ij} = 1$ for some $j^\star$ and 0 for all others. This would mean that we were certain that $Z_i = j^\star$. In that case, we would just compute the log-likelihood for that value of Z_i, and ignore all others. This is mathematically equivalent to multiplying by 1 when $j = j^\star$ and multiplying by 0 otherwise.

M-step update for μ_j. Now that we have decided to maximize the ECLL, the rest is a matter of calculus. I will explicitly show the formula we get for μ_j, then just tell you the formulas for the others and give the intuition.

To maximize the ECLL with respect to μ_j, we want to take the gradient with respect to μ_j, set it equal to 0, and then solve for μ_j. Let's first plug in our formulas for $P(Z_i = j)$ and $P(X_i \mid Z_i = j)$:

$$\text{ECLL}(\pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}) = \sum_{i=1}^{n} \sum_{j=1}^{k} R_{ij} \left(\log \pi_j + \log \left(\frac{1}{(2\pi)^{d/2} \sqrt{\det(\Sigma_j)}} \right) - \frac{1}{2} (x^{(i)} - \mu_j)^\top \Sigma_j^{-1} (x^{(i)} - \mu_j) \right).$$

Note that a lot of terms don't depend on μ_j, so we can ignore them when taking the gradient

with respect to μ_j. At the end, we'll get left with

$$\nabla_{\mu_j} \text{ECLL}(\pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}) = \sum_{i=1}^{n} R_{ij} \nabla \mu_j \left(-\frac{1}{2}(x^{(i)} - \mu_j)^\top \Sigma_j^{-1}(x^{(i)} - \mu_j) \right).$$

To take this gradient, we will use a fact that $\nabla_x x^\top A x = 2Ax$. You can prove this formula directly, or just note that it's very similar to the normal rule for quadratics ($x^\top Ax$ is essentially a multivariate quadratic). This gives us

$$\begin{aligned}
\nabla_{\mu_j} \text{ECLL}(\pi_{1:k}, \mu_{1:k}, \Sigma_{1:k}) &= -\frac{1}{2} \sum_{i=1}^{n} R_{ij} \cdot 2\Sigma_j^{-1}(x^{(i)} - \mu_j) \cdot (-1) = 0 \\
\sum_{i=1}^{n} R_{ij} \Sigma_j^{-1}(x^{(i)} - \mu_j) &= 0 \qquad \text{Simplify} \\
\Sigma_j^{-1} \sum_{i=1}^{n} R_{ij}(x^{(i)} - \mu_j) &= 0 \qquad \text{Factor out } \Sigma_j^{-1} \\
\sum_{i=1}^{n} R_{ij}(x^{(i)} - \mu_j) &= 0 \qquad \text{Multiply on left by } \Sigma_j \\
\sum_{i=1}^{n} R_{ij} x^{(i)} &= \mu_j \sum_{i=1}^{n} R_{ij} \qquad \text{Move } \mu_j \text{ terms to right} \\
\mu_j &= \boxed{\frac{\sum_{i=1}^{n} R_{ij} x^{(i)}}{\sum_{i=1}^{n} R_{ij}}}
\end{aligned}$$

Intuition and M-step updates for π_j and Σ_j. What is the intuition for this formula for μ_j? It can be related back to our fictitious dataset where each possible value of Z_i appears R_{ij} times. The denominator is the total number of (fractional) examples where $Z_i = j$. The numerator is the weighted sum of $x^{(i)}$ weighted by the probability that $Z_i = j$. So overall, this is just a weighted average of $x^{(i)}$'s based on the probability of being in cluster j. Inituitively this is quite similar to what we had for k-Means, except for k-Means all the R_{ij}'s were either 1 or 0, since we made hard assignments of points to clusters.

Based on this, we can intuit the updates for the other parameters. First,

$$\pi_j = \frac{\sum_{i=1}^{n} R_{ij}}{n}.$$

This is simply measuring the fraction of points where $Z_i = j$, which is exactly what π_j represents.

Finally, we have

$$\Sigma_j = \frac{\sum_{i=1}^{n} R_{ij}(x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^\top}{\sum_{i=1}^{n} R_{ij}}.$$

This is the formula for the covariance of a dataset (in matrix form), but again weighted by the R_{ij}'s, just like we had for the μ_j update.

So overall, the EM recipe is as follows:

- **E-step**: Run inference to estimate $R_{ij} = P(Z_i = j \mid X_i)$ for each i.
- **M-step**: Pretend the R_{ij}'s give you fractional counts of each value of Z_i. Use these fractional counts to compute a weighted mean and covariance for each cluster, as well as the frequency of each cluster.

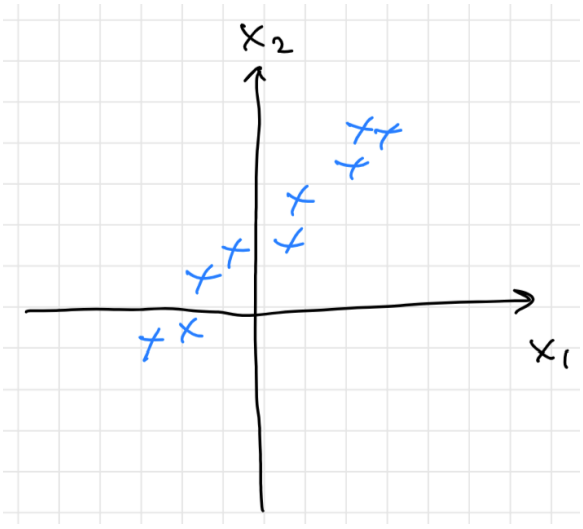
Chapter 9

Dimensionality Reduction with Principal Component Analysis

9.1 Dimensionality Reduction

The last two classes focused on clustering, which is one type of unsupervised learning task. Now we will learn about another type of unsupervised learning called dimensionality reduction. In dimensionality reduction, we are also given an unlabeled dataset $D = \{x^{(1)}, \ldots, x^{(n)}\}$, and we want to learn something about the dataset's underlying structure. But unlike in clustering, where we wanted to find subgroups of examples, in dimensionality reduction we want to find a **low-dimensional subspace** that best represents the data.

As a toy example, consider the following two-dimensional dataset:



Even though all the $x^{(i)}$'s are in two dimensions, it is clear that the dataset is "essentially" one-dimensional because all the points are very close to lying on a single straight line. Given this dataset, a dimensionality reduction technique should be able to identify this straight line. We could then simply the data by plotting the *projection* of the points onto this straight line in one dimension, rather than plotting the original two-dimensional data. In other words, we can "reduce" the dimensionality of the data from 2 to 1.

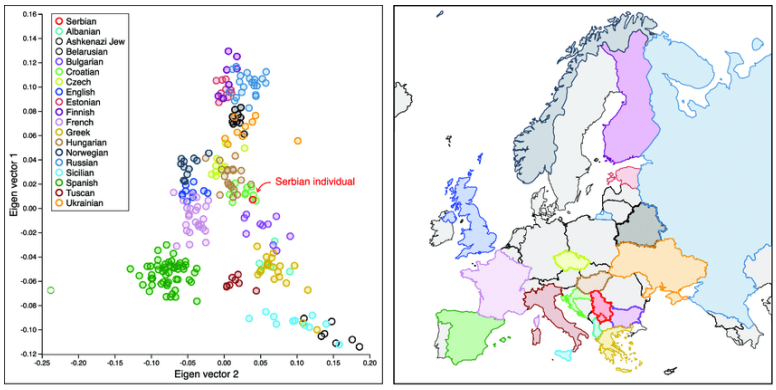


Figure 9.1: Result of running PCA on genetic data from European individuals. The top two principal components capture the latitude and longitude of the individual's country of origin, so the resulting plot is very similar to the map of Europe.

In practice, dimensionality reduction is a powerful tool when trying to understand high-dimensional data. It's very hard for humans to intuitively grasp something about four-dimensional space, let alone 1000-dimensional space. So if we have data that is high-dimensional, we often apply dimensionality reduction to extract the top two most important dimensions (i.e., to find a two-dimensional subspace that captures most of the structure in the dataset).

In Figure 9.1, we see the result of applying dimensionality reduction to a dataset of genetic data from different European individuals. In the original data, each individual's genome was tested for roughly 600,000 different single nucleotide variants (SNVs), or possible differences in genetic sequence between individuals. Thus, each individual's genome is represented by a 600,000-dimensional vector. To visualize this data, researchers applied a method called Principal Component Analysis, which we will talk about shortly, and used it to extract the top two dimensions. When they plotted this, they found a strong correlation between the individual's country of origin and where they fell on this two-dimensional plot. Individuals from the same country were more likely to have similar genomes, and individuals from neighboring countries were more similar than individuals from distant countries. Thus, after dimensionality reduction, the plot of all the genomes closely resembles the actual map of Europe.

9.2 Principal Component Analysis

In this class we will cover the most widely-used method for dimensionality reduction, **principal component analysis** (PCA).

9.2.1 Preliminaries

To start out, let's consider the goal of finding the best 1-dimensional projection of our data. It turns out our solution to this will naturally lead us to a way to identify the best k-dimensional projection for any k as well.

When doing PCA, we will assume that our data has mean 0, i.e.

$$\frac{1}{n}\sum_{i=1}^{n} x^{(i)} = 0.$$

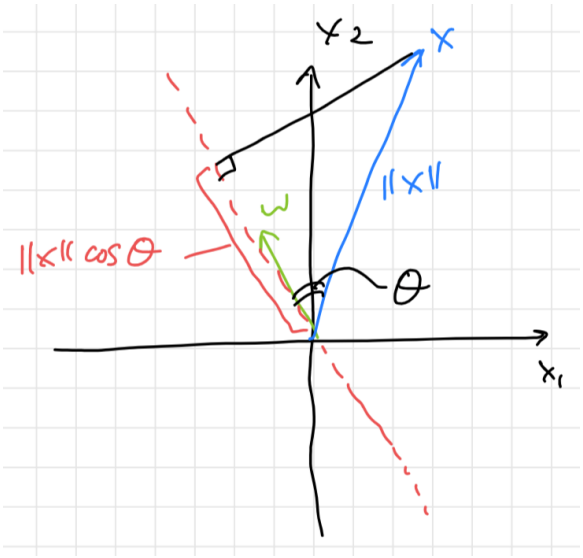
In practice, this is easy to ensure because we can first compute the mean of the data and then subtract it from every datapoint. This doesn't really meaningfully change the data (it just shifts everything over by a fixed amount) and ensures that the new dataset has mean 0.

9.2.2 Defining Reconstruction Loss

As usual, our strategy will be to write down a loss function for our parameters, and find the parameters that minimize that loss function. What are the parameters of a dimensionality reduction method? Well, we want to find a 1-dimensional subspace to project the data onto. A 1-dimensional subspace is defined by a unit vector w (i.e., a vector where $\|w\| = 1$); the subspace is simply the set of vectors that are some multiple of w. So we can think of this w as the parameter we want to learn.

Now we must write down a loss function that tells us how bad a given choice of w is. Similarly to k-Means, we can use the "reconstruction loss" idea to come up with a good loss function. We would say that a w is bad if when we project all points onto the subspace spanned by w, the resulting points are far from the original points. Conversely, w is good if when we do this projection, the resulting points are pretty close to the original points—in other words, the projection of all the data onto the subspace is a pretty good "reconstruction" of the original data.

Projections. Let's remember what it means to project onto a (one-dimensional) subspace. If we have a vector x and a unit vector w, then the projection of x onto the subspace spanned by w is $(w^\top x)w$. Note that since $w^\top x$ is a scalar, this is indeed a multiple of w. If you want to remember this formula, you can draw out this diagram:



The projection of x onto the subspace spanned by w is the point you get when you drop a perpendicular line from x onto the line in the direction of w. This forms a right triangle, and the distance of that point from the origin is exactly $\|x\| \cos\theta$, where θ is the angle between x and w. Now, recall the relationship between the dot product and the angle between two vectors:

$$\cos(\theta) = \frac{w^\top x}{\|w\|\|x\|}.$$

Since w is a unit vector, $\|w\| = 1$, and so $\|x\| \cos(\theta)$ is simply $w\top x$. So, the projection of x onto w is $w^\top x$ units from the origin in the direction of w, which means it is just the point $(w^\top x)w$.

The reconstruction loss. Now that we have a formula for the projection of x onto the subspace spanned by w, we can write down the reconstruction error loss:

$$L(w) = \sum_{i=1}^{n} \|x^{(i)} - (w^\top x^{(i)})w\|^2.$$

This simply compares the original $x^{(i)}$ to the projection of $x^{(i)}$ onto our chosen subspace, and penalizes us if they are far away (in terms of Euclidean distance).

9.2.3 Minimizing Reconstruction Error is Maximizing Variance

It turns out that for PCA, it is helpful to notice that minimizing the reconstruction loss is equivalent to maximizing a different objective function. The latter maximization problem will directly give us a closed-form solution for w.

Refer back to the previous diagram. In the reconstruction loss, we're computing the squared distance between x and $(w^\top x)w$. This is one leg of a right triangle (the one in black), where the other leg is what we already computed to have length $(w^\top x)w$.[1] The Pythagorean Theorem therefore tells us that for each x,

$$(w^\top x)^2 + \|x - (w^\top x)w\|^2 = \|x\|^2.$$

The x's all come from our dataset, so $\|x\|^2$ is fixed. The second term on the left is the reconstruction error for example x, which we want to minimize. So it is exactly equivalent to try to *maximize* the first term, $(w^\top x)^2$, since they must add up to some fixed number.

As a result, we can rewrite our goal as maximizing the quantity

$$\sum_{i=1}^{n} (w^\top x^{(i)})^2.$$

Note that if we multiply this by $\frac{1}{n}$, the quantity $\frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)})^2$ is simply the variance of $w^\top x$, since the mean of the data 0. This itself is an interesting observation: to minimize reconstruction error, we can alternatively try to maximize the variance of the data after projecting onto the subspace spanned by w. Intuitively, this preserves most of the "spread" of the data.

[1] Technically I've been ignoring the possibility that $w^\top x < 0$. The math still works out, but the distance should technically be $|w^\top x|$ rather than $w^\top x$.

9.2.4 Maximizing Variance via Eigendecomposition

We now have a new goal: finding a unit vector w that maximizes the variance $\frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)})^2$. Let's try to transform this into a more useful form:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)})^2 &= \frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)})(x^{(i)\top} w) && \text{Dot product is symmetric} \\
&= \frac{1}{n}\sum_{i=1}^{n} w^\top (x^{(i)} x^{(i)\top}) w && \text{Multiplication is associative} \\
&= \frac{1}{n} w^\top \left(\sum_{i=1}^{n} x^{(i)} x^{(i)\top}\right) w && \text{Factor out } w\text{'s from sum} \\
&= w^\top \Sigma w
\end{aligned}$$

where we define

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} x^{(i)} x^{(i)\top}.$$

Why have we named this Σ? If we remember our discussion of covariance matrices from GMM's, you may recall that this expression is exactly the covariance of the dataset (again, since the mean is 0).

So we now know that all we have to do is find the unit vector w that maximizes the simple expression $w^\top \Sigma w$. It turns out that linear algebra gives us a very nice solution to this problem. First, note that Σ is a symmetric matrix. We can see that because the outer product between a vector and itself always yields a symmetric matrix, and Σ is just the sum of all these outer products; alternatively, we can just remember that covariance matrices are always symmetric. Now, an important fact about symmetric matrices is that they can be **diagonalized**: for any symmetric $d \times d$ matrix Σ, there exists[2] a diagonal matrix D and orthonormal matrix U such that $\Sigma = UDU^\top$. The diagonal entries of D are denoted $\lambda_1, \ldots, \lambda_d$ and are the **eigenvalues** of Σ. U being orthonormal means that each column of U, which we'll denote u_j, is a unit vector and each pair of vectors u_i and u_j are orthogonal, i.e., $u_i^\top u_j = 0$ for all $i \neq j$. The vector u_j is the **eigenvector** associated with the eigenvalue λ_j. Note that the order of the d eigenvalues and eigenvectors can be chosen arbitrarily; for convenience later, let's order them so that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d$.

This means that now we can write our objective as maximizing $w^\top UDU^\top w$. Let's define $a = U^\top w$, so this is also $a^\top Da$. What do we know about a? There is a nice fact about orthonormal matrices: whenever you multiply an orthonormal matrix U by a vector v, the resulting vector Uv has the same norm as v. This can be shown with some direct calculation:

$$\|Uv\|^2 = (Uv)^\top Uv = v^\top U^\top Uv = v^\top Iv = v^\top v = \|v\|^2,$$

where we know that $U^\top U$ is the identity matrix I because U is orthonormal. (You may also remember some intuition from your linear algebra class that multiplying by an orthonormal matrix applies a change of basis that doesn't stretch the space in any way, so the lengths of

[2]There are library functions that will in fact compute these matrices for you, such as `numpy.linalg.eig`.

all vectors is preserved.) Armed with this fact, we now know that since $\|w\| = 1$, $\|a\|$ must be 1 as well.

Now we have further simplified our problem to maximizing $a^\top Da$ subject to $\|a\| = 1$. To do this, note that

$$a^\top Da = \sum_{j=1}^{d} \lambda_j a_j^2,$$

and we know that $\|a\|^2 = \sum_{j=1}^{d} a^2 = 1$. You can think of this as saying that we have a fixed budget—we can spend some money to invest in a_1^2, some other money to invest in a_2^2, and so on, but the total amount we can invest is 1. And for each dollar we invest in a_1^2, we get λ_1 in return; for each dollar we invest in a_2^2, we get λ_2 in return; and so on. When framed this way, it should be intuitively clear that the best choice for a should be to have $a_1 = 1$, since λ_1 is the largest eigenvalue, and 0 everywhere else. This is the most efficient way to "spend" the total budget that we have.

We're almost done! We just need to go back and figure out which value of w results in this optimal choice of $a = U^\top w$. Well, we need $u_1^\top w = 1$, and $u_j^\top w = 0$ whenever $j \neq 1$. We know of a vector that satisfies this—it's just u_1! So after all this work, we have finally arrived at a clean answer: the optimal choice of w is **the eigenvector of Σ corresponding to the largest eigenvalue**.

Going beyond one dimension. One of the most common reasons to use PCA (or any dimensionality reduction method) is to reduce a high-dimensional dataset down to two dimensions, so we can plot it. So how do we find the top two dimensions? The answer is very intuitive: we should just use the eigenvectors corresponding to the top *two* eigenvalues. In general, if we want to reduce the dimensionality down to k, we just use the top k eigenvectors. The projection of the data onto the j-th eigenvector is referred to as the **j-th principal component**.

Chapter 10

Bandits

It's time to discuss a third and final machine learning paradigm: **reinforcement learning**. In reinforcement learning, we have an **agent** that is interacting with a world by taking **actions**. Each time you take an action, you land in a new **state**, in which you then take another action; you also receive a **reward** for the action you just took. We want to learn how to take sequences of good actions that lead to large total rewards, but at the beginning of the process we don't know how the world works. So, we have to try out actions to learn about how the world works and which actions lead to high rewards. Out of everything we cover in this class, reinforcement learning is the closest to mirroring how humans have to learn from experience throughout life. We go about the world taking actions, learn from our experience, and use that to choose different actions in the future.

Reinforcement learning differs from both supervised and unsupervised learning in two key ways. First, in both supervised and unsupervised learning, all of our algorithms assumed that a dataset was handed to us by someone else. In supervised learning, the dataset looked like $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$; in unsupervised learning the dataset looked like $\{x^{(1)}, \ldots, x^{(n)}\}$. But in both cases, the learning algorithm itself received the dataset as an input. It did not have any influence over what data was there—it just tried to extract insights from the dataset handed to us. In contrast, in reinforcement learning we will start with no data, and our algorithm will collect data over time by interacting with the world. By making different choices, the algorithm will actually influence what data we collect, and therefore what we can learn. This is a very important dynamic that any good reinforcement learning algorithm must take into account.

Second, in reinforcement learning the supervision we receive is in the form of a reward, which basically tells us whether a sequence of actions resulted in something good or something bad. Even if an action we take receives high reward, we have no idea if there would have been a better series of actions that could give us higher reward. We never get told what the "best" or "correct" action would have been to take—that would make it supervised learning. But on the other hand, the reward does provide some (weaker) supervision, so we are not in an unsupervised learning setting either.

10.1 Introduction to Bandits

We will begin our exploration of reinforcement learning by considering **bandit** problems. The bandit setting is a special case of reinforcement learning in which there is no "state" that is controlled by the agent. In a bandit problem, the agent simply chooses an action at each timestep and receives some reward.

To explain this concept more, it helps to understand where the term "bandit" comes from. A "one-armed bandit" is a slang term for a slot machine. Now imagine you are in a casino with many slot machines. Some of the slot machines may give better payouts than others, but when you enter the casino you have no idea which slot machine is the best. So, you have to try each of the slot machines one at a time. Trying a slot machine doesn't change the "state" of the world (it doesn't change how good each slot machine is), but the more you try, the more you learn about which ones are better. Eventually, you hope to find the slot machine with the best payouts, without wasting too much money on the "bad" machines in the process. This problem setting is known as the **multi-armed bandit** problem in machine learning, which is often shortened to a "bandit problem."

Some other scenarios that can be cast as bandit problems include:

- **Medicine**: Suppose you have k different medicines that are hypothesized to help with a given disease. Initially, you don't know which one is most effective, so you need to do some sort of clinical trial. You could view this as a bandit problem. Each time you meet a patient with this disease, you prescribe them one of the k medicines (this is the "action"). You can observe whether they get better or not, which is your "reward." Over time, you learn which medicine is most effective at treating the disease.

- **Recommendation systems**: Now consider a social media company like TikTok. For each user, TikTok tries to recommend videos to maximize user engagement, but when a new user signs up they don't know anything about that user's preferences. So they need to keep recommending different videos and observe whether the user watches those videos or not. Over time, TikTok learns what actions (i.e., videos to recommend) are likely to lead to the "reward" of the user watching the video.

10.2 Formal Definitions

Let's now provide a more formal definition of the multi-armed bandit problem.

- The agent plays a game for T total rounds.

- In each round $t = 1, \ldots, T$, the agent gets to select one action A_t. We will assume there is a fixed list of k actions numbered $1, \ldots, k$, and that at every round, the agent can select any one of these actions.

- After playing an action A_t, the agent receives a reward R_t. This is sampled from an (unknown) distribution $p_a(R)$ for each action a. For example, in the casino example, each slot machine a has its own probability distribution over payouts. That distribution is the same for all timesteps, but differs between machines.

- The agent's goal is to maximize the total reward accrued over all timesteps, i.e.,

$$\sum_{t=1}^{T} R_t.$$

You can see that in order to achieve this goal, the agent should try to figure out which action has the largest expected reward, and then play that action all the time.

Regret. It is common to measure the success of a bandit algorithm not by the total reward, but by the difference between the total reward and the reward of the best possible strategy. This difference is known as the **regret**—intuitively, this is how much you "regret" not playing the best possible strategy. To define the best possible strategy, first let's define $\mu(a)$ to be the expected reward for each action a:

$$\mu(a) \triangleq \mathbb{E}_{R \sim p_a(R)}[R].$$

The optimal action a^* is the one with the largest expected reward:

$$a^* = \arg \max_{a \in \{1,\ldots,k\}} \mu(a).$$

So, we can now define the **expected regret** of any bandit algorithm as:

$$\mu(a^*) \cdot T - \mathbb{E}\left[\sum_{t=1}^{T} R_t\right].$$

The first term is the expected reward if you always played the optimal action a^* for all T timesteps; the second term is the expected reward of the algorithm. Having a regret of 0 means your algorithm matches the optimal strategy, which is ideal. In reality, we would always expect to have some regret > 0, because the algorithm does not know in advance which action is optimal, and has to try all of them first to figure that out.

Note that regret is really only useful as a theoretical concept, because in practice you don't actually know what the optimal action a^* is, so you can't directly compute it. But it turns out that we can still come up with algorithms that we can prove have low regret.

10.3 Upper Confidence Bound Algorithm

We will now introduce one prototypical algorithm for solving multi-armed bandit problems called the **Upper Confidence Bound (UCB)** algorithm. UCB is intuitive and achieves provably good regret.

10.3.1 Exploration and Exploitation

Before we dive into UCB, let's talk about a dynamic that any bandit algorithm has to navigate: the trade-off between exploration and exploitation. **Exploration** refers to trying

different actions to gain knowledge about which one is better. In the casino example, it makes sense to try every slot machine at least a few times. **Exploitation** refers to using the knowledge you have gained to choose the best action. In the casino example, once you have figured out the best slot machine, you should just keep playing it over and over. Note that these two are fundamentally at odds with each other: when you're exploring, you are trying actions that you don't necessarily think are best. If you're exploiting, you're ignoring the possibility that other actions might actually be worth trying. Bandit problems are a simple setting where we have to balance these two considerations.

10.3.2 UCB intuition

Let's see how UCB balances exploration and exploitation. The key idea of UCB is that for each action a, our algorithm will try to estimate the expected value $\mu(a)$ of taking that action. Since we only collect a finite amount of data, all of our estimates will have some amount of uncertainty. In UCB, we will represent this uncertainty as a confidence interval—we will estimate $\mu(a)$ as being between some lower bound and upper bound, based on our observations so far. For example, we might believe that for action a_1, $\mu(a_1)$ is between 0.2 and 0.8, while for another action a_2, $\mu(a_2)$ is between 0.3 and 0.4. At each timestep, the algorithm then chooses the action with the *largest upper bound*. Based on the observations, it then recomputes its estimates of $\mu(a)$.

You can think of the upper bound as the "optimistic" expected reward for each action. By choosing the action with the largest upper bound, UCB epitomizes the principle of **optimism in the face of uncertainty**. By being optimistic, we are open to the possibility that any action could be really good, so we will naturally try all of them at least a little bit (i.e., we will explore).

10.3.3 Formal algorithm description

We'll now see the formal description of the algorithm. Recall that at each timestep $t = 1, \ldots, T$, the algorithm chooses an action A_t and receives a reward R_t. For a time t, let $n_t(a)$ denote the number of times we have tried action a up until time t. In other words, $n_t(a)$ is how many "datapoints" we have about action a up until time t. Let $\hat{\mu}_t(a)$ denote the sample mean of the rewards over all of these times that we took action a in time t.

As we described in the last section, we want to maintain an upper and lower bound on the true mean $\mu(a)$ based on the observed data. We will get these bounds by reasoning about the standard deviation of our sample mean $\hat{\mu}_t(a)$. In general, when you draw many i.i.d. samples from some distribution, the sample mean is an unbiased estimate of the true mean, and its variance decreases based on how many samples you drew. In particular, when you have n examples, the variance of a sample mean is σ^2/n, where σ^2 is the variance of the original distribution. So, the standard deviation is $\sigma/\sqrt{n}$; most importantly, it is proportional to $1/\sqrt{n}$.

UCB creates a confidence interval whose width is proportional to the standard deviation

of the sample mean for each action a. In particular, we assume that

$$\mu(a) \in \left[\hat{\mu}_t(a) - \sqrt{\frac{2\log t}{n_t(a)}}, \hat{\mu}_t(a) + \sqrt{\frac{2\log t}{n_t(a)}}\right]$$

In other words, we believe that the true mean is within $\pm\sqrt{\frac{2\log t}{n_t(a)}}$ of the sample mean $\hat{\mu}_t(a)$. Note that the width of our confidence interval is proportional to $1/\sqrt{n_t(a)}$, as expected, since $n_t(a)$ is the number of samples used to compute our sample mean $\hat{\mu}_t(a)$. For convenience, we will define

$$\mathrm{UCB}_t(a) = \hat{\mu}_t(a) + \sqrt{\frac{2\log t}{n_t(a)}}$$

Finally, the UCB algorithm itself is very straightforward:

1. For $t = 1, \ldots, k$: try each action once. (Recall that k is the number of possible actions)
2. For $t = k+1, \ldots, T$: choose $A_t = \arg\max_a \mathrm{UCB}_t(a)$.

Let's take a minute to examine the formula for $\mathrm{UCB}_t(a)$, since it governs how the UCB algorithm chooses actions. The first term, $\hat{\mu}_t(a)$, represents our current best guess of the expected reward for taking action a. If we chose the action with the largest first term, that would be a purely exploitation-based strategy—we just greedily choose the action that seems best so far. Meanwhile, the second term represents how much uncertainty we have about our estimate of $\mu(a)$. If we chose the action with the largest second term, that would be a purely exploration-based strategy—we would always choose the action we have tried the least number of times before. The overall UCB formula thus balances exploration and exploitation simply by adding up an exploitation-promoting term and an exploration-promoting term, and maximizing the sum.

We can also further understand the logic of the exploration-promoting term,

$$\sqrt{\frac{2\log t}{n_t(a)}}.$$

The denominator gets larger over time, as we collect more samples of rewards after playing action a. Thus, this exploration-promoting term gets smaller over time, leading us to slowly do more exploitation. This is intuitive—after a while, we have a good sense of which actions are good, and should be exploiting that knowledge more. On the other hand, the numerator, $\sqrt{2\log t}$, gets bigger over time, albeit slowly (because of the log). This means that we will never *completely* rule out an action. If we go a long time without trying a particular action a, the denominator will stay the same, while the numerator will grow, so eventually its UCB will get larger and larger until we pick it again. This ensures that the algorithm always "keeps an open mind" to other actions.

10.3.4 Regret Bounds for UCB

We won't go into the details here, but it turns out that UCB has quite strong theoretical guarantees. Let's assume that the reward R_t is always bounded within $[0, 1]$ (you can often scale the rewards accordingly so they're always in this interval). Then, we can prove that the regret of UCB is $O(\sqrt{kT \log T})$. This is an exciting result because this function grows *more slowly than a linear function of* T. Recall that the regret is defined as

$$\mu(a^*) \cdot T - \mathbb{E}\left[\sum_{t=1}^{T} R_t\right].$$

Since we have a sum over all T timesteps, you would naturally expect this quantity to grow linearly with T. The fact that UCB has sublinear regret implies that over time, it converges to the optimal strategy.

Another way of viewing this is to think about the *average* regret, which is just our original regret formula divided by the total number of timesteps T. In English, this measures how much worse you were than the optimal action, on average across all timesteps. We can see that the average regret of UCB is

$$O\left(\frac{\sqrt{kT \log T}}{T}\right),$$

which tends to 0 as T tends to infinity. So, after enough time, the gap between UCB and the optimal strategy becomes negligible.

Chapter 11

Reinforcement Learning

Now that we have warmed up by studying bandits, it's time to look at the full reinforcement learning problem. Recall that bandits are a particular type of reinforcement learning in which the "state" of the world does not change when we take an action. In contrast, in a general RL problem, the agent starts in some state and every action they take causes them to "transition" into a new state. This adds an additional consideration compared to bandits: how can we make sure that the agent learns about all the possible states they could go to, and which states are desirable?

11.1 Markov Decision Processes

Before we can talk about reinforcement learning, we have to talk about what it means mathematically to be interacting with a *world*. When we do reinforcement learning, we assume that the world can be described formally by a **Markov Decision Process** (MDP). This is a formal description of what are the possible states an agent could be in, what actions they could take, how it's decided what state they end up in, etc. The agent can interact with the world for one or more **episodes**. In each episode, the agent starts at some state, and continuously takes actions and arrives in new states. This continues until it reaches an end state. Keep in mind that the learning agent itself does not know the MDP at the beginning; it must learn about the MDP by interacting with it.

Formal definition of MDP. Formally, an MDP is defined by the following ingredients:

- A set of states S. These represent all possible states of the agent.
- A starting state $s_{\text{start}} \in S$. This is where the agent starts each of its attempts. In some cases, you could have a probability distribution over states, rather than a single fixed start state.
- Actions(s): A function that returns the set of possible actions to take in every state s.
- $T(s, a, s')$: The probability of going from state s to state s' after taking action a. Since

this is a probability, it must be that

$$\sum_{s' \in S} T(s, a, s') = 1$$

for any state s and action $a \in \text{Actions}(s)$. The fact that this is a probability distribution captures the notion that the world has some randomness (e.g., things that the agent does not control), and taking the same action in the same state can lead to different results.

- Reward(s, a, s'): The (immediate) reward received when going from s to s' by taking action a. We also may refer to this as the reward associated with the *transition* from s to s' when taking action a.
- IsEnd(s): A function that returns whether or not the given state s is an end state. Once the agent reaches an end state, that particular **episode** is over.

Looking ahead to reinforcement learning, our agent will be dropped into the world without knowing the transition probabilities T or the rewards associated with each transition.

Note that embedded within this definition is a key assumption: every time you visit the same state s, the transition and reward probabilities are exactly the same. This is called a **Markovian** assumption: essentially, if you know that you're at state s, then that entirely defines the dynamics of the world going forward. Nothing about the past history gives you any more information about what is about to happen.

Agent policies. So far we have defined how the world works. Now let's take the perspective of an agent. An agent chooses actions in accordance with a **policy**. A policy is just a function $\pi(s)$ that maps each state s to a chosen action $a \in \text{Actions}(s)$. Note again that because of the Markovian nature of MDPs, the current state s is all the agent should need to choose a good action—it doesn't need anything about its past.

Value of a policy. The agent wants to choose actions that maximize the **total sum** of rewards across the whole episode. Thus, the best policy should not merely choose the action with the largest immediate reward; it should also choose actions that lead to states from which large rewards are possible. This again is analogous to life—you might take a class you don't like (immediate negative reward) so that in the future you can get a job you do like (future positive reward).

Formally, the agent wants to choose a policy π that maximizes the expected sum of rewards it receives, from start to end. It helps to define what we call the **value** of a state s under a policy π, denoted $V_\pi(s)$, which is the expected sum of rewards starting at state s and running policy π. I will similarly refer to V_π as the **value function**, as it is a function of the state s. Remember that when I say "expected," one way to think about this is just to imagine starting at state s many times, running the same policy until you hit an end state, and recording what total reward you get. The expected reward is just the average total reward if you do this many times.

If we know all the transitions and rewards of the MDP, we can write down a series of equations that tell us what the value of a policy is. When we run the policy π starting at a state s, the transition function tells us what new states s' we might end up in, and the reward function tells us what rewards we would get at that particular timestep. From there, we would keep running the policy π and obtain some further rewards, which in expectation would be $V_\pi(s')$ by definition. Therefore, the value function must satisfy the following equation:

$$V_\pi(s) = \sum_{s' \in S} T(s, \pi(s), s') \cdot (\text{Reward}(s, a, s') + V_\pi(s')) .$$

This is just the formula for the expected value of the immediate reward for taking action $\pi(s)$, plus the expected future reward we will get after landing in the new state s'.

Now, how useful is this equation if we are trying to compute $V_\pi(s)$? At first glance, it seems to be not that useful, because we would need to compute $V_\pi(s')$ for all other possible states s' first. However, there's another way to think about this: this equation is true for every state s, and so in total we have $|S|$ different equations. We also have $|S|$ different "unknowns", namely $V_\pi(s)$ for each $s \in S$. All of the equations are linear equations. Thus, this is simply a big system of linear equations with $|S|$ equations and $|S|$ variables, and we can use our favorite method of solving linear systems of equations to solve for all $V_\pi(s)$ values. One important caveat to remember here is that we can only do this if we know in advance what the transition and reward functions are—during reinforcement learning, we won't actually know these.

Discounting. There's one small caveat to the definition I just gave: usually we give a slight preference to current rewards than future rewards. The reason for this is partly mathematical: in theory the future could last forever, and so the "total" reward might grow to infinity, which causes some headaches. Another way to motivate this is that at each timestep, there might be a small probability of system failure/death/etc., so it's slightly better to get rewards now.

We operationalize this mathematically using something called a **discount factor**, denoted γ, which is just a number between 0 and 1. γ represents the probability of survival at each timestep—imagine something like $\gamma = 0.99$, so at each timestep there is a 0.1 chance of death. To receive a reward from a future timestep, you have to survive all the way until that time. So a reward one step in the future is only worth γ times the actual reward, a reward two steps in the future is only worth γ^2 times the actual reward (since you have to survive twice), and so on.

So, if our agent starts at state s, interacts with the world over many timesteps by playing policy π, and receives reward r_1 after its first action, r_2 after its second action and so on, then we would say its **discounted sum of rewards** is $r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots$. Similarly, the expected value of this discounted sum of rewards is the **discounted value** of state s under policy π. When I write $V_\pi(s)$, I will actually use this to denote the discounted value, rather than the un-discounted value.

Note that rewards that happen infinitely far in the future contribute 0 to the value, as long as $\gamma < 1$. So the discounted value is well-behaved even in games that could in theory go on forever.

One nice thing about discounting is that it only requires us to update our $V_\pi(s)$ formula slightly. The revised formula is:

$$V_\pi(s) = \sum_{s' \in S} T(s, \pi(s), s') \cdot (\text{Reward}(s, a, s') + \gamma V_\pi(s')) .$$

The only difference is that the future reward $V_\pi(s')$ is now multiplied by γ, so it is discounted relative to the immediate reward. Since s' is the state we will get to one step in the future, it is discounted by a single factor of γ; if we were to expand the formula for $V_\pi(s')$, you would see that the state two steps in the future will get a second factor of γ, and so forth, matching the geometric progression I wrote out above.

Optimal value. What policy should the agent play to maximize value? If we know all the details of the MDP (i.e., we know the transition probabilities and rewards), then there is a nice set of equations that characterizes the optimal policy. Let $V_{\text{OPT}}(s)$ denote the *maximum possible value* for any policy under state s. I'm also going to define the Q-value, denoted $Q_{\text{OPT}}(s, a)$, which is defined as the expected sum of rewards after taking action a in state s, and then behaving optimally for the rest of the episode. This may sound clunky at first, but it turns out that the Q-value is actually a very useful quantity to think about.

$V_{\text{OPT}}(s)$ and $Q_{\text{OPT}}(s, a)$ are closely related via recursive formulas. First, we have

$$V_{\text{OPT}}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{OPT}}(s, a) & \text{else} \end{cases}$$

Essentially, this equation says that if s is an end state, then the episode is over, so there's no more rewards we can collect. Thus, the value is simply 0. Otherwise, the optimal value is achieved by playing whichever action has the largest Q-value—this is true by definition of the Q-value.

Meanwhile, the Q-value is characterized by this equation:

$$Q_{\text{OPT}}(s, a) = \sum_{s' \in S} T(s, a, s') \left(\text{Reward}(s, a, s') + \gamma V_{\text{OPT}}(s')\right)$$

Why is this true? First, this is an expectation over the possible transitions that can happen after taking action a in state s; the transition probabilities are given by $T(s, a, s')$. The thing we're computing the expectation of is the total sum of rewards we obtain. First, there is the immediate reward we get, which is $\text{Reward}(s, a, s')$. This is given to us immediately, so there's no discount factor. Then there are the future rewards we can expect to get once we land in the new state s'. This is exactly $V_{\text{OPT}}(s')$, but since they are happening one step in the future, we multiply it by γ.

Finally, if we knew all the Q-values, then the optimal policy $\pi^\star$ is simply choosing the action that maximizes the Q-value at every state:

$$\pi^\star(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{OPT}}(s, a).$$

There are algorithms for solving MDP's that directly use these equations to compute $V_{\text{OPT}}(s)$ and $Q_{\text{OPT}}(s, a)$ for every state s and action a. These methods assume that we know

everything about the MDP. However, since our focus is on learning, we will not talk about those algorithms. Instead, we are more interested in the reinforcement learning setting, where we do not already know how the MDP works (i.e., the transition probabilities and rewards are unknown).

11.2 Q-Learning

It's time to talk about our first reinforcement learning algorithm, Q-learning. The basic idea of Q-learning is simple. We know from above that if we knew all the Q-values, we would immediately know the optimal policy. So, we will interact with the environment and use our observations to *learn* what the Q-values are. If we can estimate all the Q-values well, we can come up with the optimal policy (or at least something close).

11.2.1 Reinforcement Learning Setup

First, let's understand the reinforcement learning problem setup. The agent gets to experience the world in a series of **episodes**, each of which starts at the start state and goes until it reaches an end state. Think of each episode as a different "lifetime" or a different chance to play a game. Within each episode, time t starts at $t = 1$, where the initial state s_1 is the start state. The agent then repeatedly takes an action a_t based on its current state s_t using an "action policy" called $\pi_{\text{act}}(s)$. After taking an action, the agent receives a new reward r_t and transitions to a new state s_{t+1}. You can think of this as a new piece of data being collected—after taking action a_t in state s_t, the agent now observes that it gets r_t and transitions to s_{t+1}. We can think of this as a single example (s_t, a_t, r_t, s_{t+1}), kind of like a bigger version of the (x, y) examples we saw during supervised learning. Each time new data is observed, the agent has the opportunity to update its parameters. Finally, the episode ends if s_{t+1} is an end state.

Altogether, the psueodocde for this loop is as follows:

```
for e = 1, 2, ... do                          ▷ e is the episode number
    s_1 ← s_start                             ▷ Start each episode at the start state
    for t = 1, 2, ... do                      ▷ t is the timestep within each episode
        Agent chooses action a_t = π_act(s_t)
        Agent receives reward r_t and new state s_{t+1}
        Agent updates parameters
        if IsEnd(s_{t+1}) then break
```

11.2.2 Tabular Q-Learning

Let's go back to the MDP world when we knew all the transition and reward probabilities. We defined this quantity $Q_{\text{OPT}}(s, a)$, and we noted that if we knew this for every possible state s and action a you could take from that state, then we could immediately infer the optimal policy by just taking the a that maximizes $Q_{\text{OPT}}(s, a)$ at every state s.

When we do reinforcement learning, we no longer know the transition and reward probabilities, but it's still true that *if* we somehow knew all the $Q_{\text{OPT}}(s, a)$ values, then we would also know the optimal policy. So, let's try to learn all of these values!

The simplest way to learn these is simply to enumerate all the possible states and actions, and estimate the Q-value for each one independently. In other words, for every s and a, we have a parameter called $\hat{Q}(s, a)$ that is meant to approximate $Q_{\text{OPT}}(s, a)$, and we will learn its value from data. This is particular strategy is known as **Tabular Q-Learning**, since we are trying to learn the Q-values and you can think of the thing we're learning as a big table of size (Number of states) by (Number of actions).

How do we learn $\hat{Q}(s, a)$? Well, we're trying to approximate $Q_{\text{OPT}}(s, a)$, and that was defined as:

$$Q_{\text{OPT}}(s, a) = \sum_{s' \in S} T(s, a, s') \left(\text{Reward}(s, a, s') + \gamma V_{\text{OPT}}(s')\right).$$

This formula is an expectation over all possible transitions that can happen after taking action a in state s. We don't have enough information to compute this expectation. But we do have "data" that consists of 4-tuples of the form (s, a, r, s'), which tells us what happened *one time* we took action a in state s. So, one "sample" for the $Q_{\text{OPT}}(s, a)$ is $r + \gamma V_{\text{OPT}}(s')$—the expected value of this quantity is exactly the Q-value. And while we also don't know $V_{\text{OPT}}(s')$, we can also approximate it with a function $\hat{V}(s)$ by just re-appropriating the formula for $V_{\text{OPT}}(s)$:

$$\hat{V}(s) = \begin{array}{ll} 0 & \text{if IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} \hat{Q}(s, a) & \text{else} \end{array}$$

This just provides the estimated value of each state s based on our Q-value estimates $\hat{Q}(s, a)$.

Finally, how should we update our guess of $\hat{Q}(s, a)$? Well, we can "nudge" it a bit based on our new observed data by taking a weighted average between the old value and this new observed sample. This leads to the tabular Q-learning update rule:

$$\hat{Q}(s, a) \leftarrow (1 - \eta)\hat{Q}(s, a) + \eta(r + \gamma \hat{V}(s')).$$

You can think of η as a learning rate that determines how much we trust the new sample vs. how much we trust our old value. Note that $\hat{V}(s')$ uses the $\hat{Q}$ values for the new state s', so essentially we are incorporating information about our Q-value estimates for s' into our Q-value estimates for s.

Usually we will initialize our $\hat{Q}$ values to all zeroes, and then over time they will get populated as we visit different states and take different actions.

11.2.3 Exploration and ϵ-Greedy

There's one final thing we need to discuss: what policy $\pi_{\text{act}}(s)$ should we use during learning? The obvious candidate is to simply use our estimates $\hat{Q}(s, a)$ to guess the best action to take in every state, and always take that action. But it turns out this has a problem. Suppose we take a particular action a in state s once and get a negative reward. Thus, we will learn a negative Q-value for that (s, a) pair. If we used the naive policy to take actions, then we

would never try that action again. But maybe we just got unlucky and it was actually a good action. Perhaps it's an action that could lead to a state from which we can obtain high rewards. Or, suppose we take a and get a high reward. Since the other actions' Q-values are initialized to zero, we might myopically always choose action a and never even try the other actions, even though they could be even better. The point is, we shouldn't immediately rule out things because of one good or bad experience—we should sometimes try things even though we think they're not optimal.

What I've just described is a fundamental trade-off known as **exploration vs. exploitation**. If we completely trust our $\hat{Q}(s, a)$ estimates, we are **exploiting** all the knowledge we've already gained, and trusting it to be correct. This would be good if we were confident we actually have learned everything there is to learn about the environment. But if we're still trying to learn more, we need to occasionally **explore** new things that we don't think are optimal, but *could* be.

There are many possible solutions to this, but there's a basic strategy that's a surprisingly strong baseline. It's called ϵ-greedy, and the way it works is simple: with probability ϵ, take a random action. Otherwise, take the best action according to our current $\hat{Q}$ values. By making ϵ larger, we can prioritize exploration more instead of exploitation.

Note that this is a strategy to employ *during* learning/training. Once we have learned Q-values, we can then set ϵ to 0 to obtain the best total rewards.

11.2.4 State discretization

While tabular Q-learning works when the state space (i.e., set of possible states) is small, in many cases the state space is in fact very large. Imagine a robot that has a location (represented as an (x, y) coordinate), plus positions of all its joints (e.g., represented as angles). Each of these quantities is represented by one or more real numbers, so in theory the set of possible states is infinite.

In practice, though, we can *discretize* this state space to get something manageable. For each dimension (e.g., x position, y position, angle of joint 1, etc.), we can divide all the possible values into a set of B discrete buckets, then map each continuous value to the corresponding bucket. If we have d different real numbers, and divide each dimension into B buckets, then the total number of discrete states becomes B^d. If B and d are relatively small, this can be small enough that we can still run tabular Q-learning. However, this grows exponentially in d, so when d gets a bit larger we will run into issues.

11.2.5 Q-Learning with Function Approximation

To deal with more complex state spaces, we need to do something smarter than independently learning the Q-value for each state. Intuitively, this is where some machine learning models can come into play. In supervised learning, we usually dealt with a very large possible space of inputs (e.g., the set of all possible houses when predicting house prices), but we could make good predictions for those inputs because we identified relevant features and learned a model to map those features to predictions. We can do the same with with Q-learning.

Defining the model. Let's start with the simplest possible model (besides keeping a giant table), which is a linear model. Suppose we have some (hand-engineered) feature function ϕ that looks at a state-action pair (s, a) and produces $\phi(s, a)$, a feature vector in $\mathbb{R}^d$. Now, we could make the modeling choice to make our prediction $\hat{Q}(s, a)$ a linear function of $\phi(s, a)$. In other words, we will have a parameter vector $w \in \mathbb{R}^d$ and our predicted Q-value will be

$$\hat{Q}(s, a) = w^\top \phi(s, a).$$

By the way, note that tabular Q-learning is a special case of this: it corresponds to using a ϕ that has an indicator features for each possible (s, a) pair. If the state space is large, this introduces a ton of features, and thus a ton of parameters, so we will struggle to learn them well.

Training. How should we train our model? Well, it looks a lot like linear regression from supervised learning, except that we don't have any examples of inputs (s, a) paired with the "true" Q-value $Q_{\text{OPT}}(s, a)$. If we had these, then we could just run normal linear regression by minimizing the squared error between our prediction and the true value. What can we do instead?

Well, in tabular Q-learning, every time we observed a (s, a, r, s') tuple, we tried to "nudge" the predicted Q-value for (s, a) towards the quantity $r + \gamma \hat{V}(s')$. (Note that $\hat{V}(s')$ is still defined in terms of $\hat{Q}(s, a)$ as before.) So, we can pretend that this quantity is in fact the "right answer" for $Q_{\text{OPT}}(s, a)$, and perform a gradient update on it. Pretending that it's the right answer means trying to minimize the squared loss (with a $\frac{1}{2}$ thrown in to make the resulting gradient formula nicer):

$$\frac{1}{2}(\hat{Q}(s, a) - r - \gamma \hat{V}(s'))^2 = \frac{1}{2}(w^\top \phi(s, a) - r - \gamma \hat{V}(s'))^2.$$

Let's take the gradient of this with respect to w.[1]

$$\begin{aligned}\nabla_w \frac{1}{2}(w^\top \phi(s, a) - r - \gamma \hat{V}(s'))^2 &= 2 \cdot \frac{1}{2}(w^\top \phi(s, a) - r - \gamma \hat{V}(s')) \cdot \phi(s, a) \\ &= (w^\top \phi(s, a) - r - \gamma \hat{V}(s')) \cdot \phi(s, a).\end{aligned}$$

Thus, our gradient descent update will be

$$w \leftarrow w - \eta(w^\top \phi(s, a) - r - \gamma \hat{V}(s')) \cdot \phi(s, a),$$

where η is our learning rate.

If we look at this a little closer, we notice that this formula is in fact very similar to the tabular Q-learning update rule. We can rewrite that update rule as follows:

$$\begin{aligned}\hat{Q}(s, a) &\leftarrow (1 - \eta)\hat{Q}(s, a) + \eta(r + \gamma \hat{V}(s')) \\ &= \hat{Q}(s, a) + \eta(r + \gamma \hat{V}(s') - \hat{Q}(s, a)) \\ &= \hat{Q}(s, a) - \eta(\hat{Q}(s, a) - r - \gamma \hat{V}(s')).\end{aligned}$$

[1] Note that technically $\hat{V}(s')$ also depends on w, but we will ignore this. We just learned something about s, so we want to update $\hat{Q}(s, a)$. We have not learned anything about s' yet, so it is premature to update $\hat{V}(s')$. But in the next timestep, we will visit s' and take some action from there, at which point $\hat{Q}(s, a)$ will get updated, which in turn may change $\hat{V}(s')$.

In fact, you can show that tabular Q-learning is exactly doing gradient descent on this same squared loss function as well.

11.2.6 Deep Q-Learning

We've showed that we can replace the table in Tabular Q-Learning with a linear model. What if we take it a step further and replace the linear model with a neural network? This is another completely viable approach, which is often called **Deep Q-Learning**, and the neural network that is used is referred to as a **Deep Q Network**.

The idea is straightforward: now our prediction $\hat{Q}(s, a)$ will just be the output of a neural network that receives information about the state s and action a as input, and predicts the Q-value as a regression task. I'll call the parameters of the neural network θ, and write the prediction as $\hat{Q}_\theta(s, a)$ to show the dependence on θ. We can train it in exactly the same way we trained the linear Q-learning model—minimize squared error with $r + \gamma\hat{V}(s')$. By the chain rule, the gradient is:

$$\nabla_\theta \frac{1}{2}(\hat{Q}_\theta(s, a) - r - \gamma\hat{V}(s'))^2 = (\hat{Q}_\theta(s, a) - r - \gamma\hat{V}(s')) \cdot \nabla_\theta \hat{Q}_\theta(s, a).$$

Note that this is the product of two quantities. The left-hand side is straightforward to compute. (Note that computing $\hat{V}(s')$ will require running the Deep Q Network on the state s' and all possible actions a' that are doable in s'). The right-hand side is the gradient of $\hat{Q}_\theta(s, a)$, i.e., the gradient of the output of our Deep Q Network. This can be computed via backpropagation, as we've seen previously. From here, we just do gradient descent to update θ.

11.2.7 Policy Gradient Methods

Finally, let's take a look at a different way to do reinforcement learning. We can think of Q-learning as a way to turn RL into a regression-like problem, since it learns to predict a continuous value (the Q-value) for every (s, a) pair. This is useful because if we know the Q-value, we can infer what the best action to take is. Alternatively, we can try to directly *classify* what the best action is given the current state. **Policy gradient methods** are a class of RL methods that take this view—they train a model that looks like a classifier that looks at the current state s and predicts the best action a. This skips the intermediate step of learning the Q-value for each (s, a) pair.

We'll use the notation $\pi_\theta(a \mid s)$ to denote the classifier model's probability distribution over actions a given a current state s. θ denotes the parameters of the model. We use the notation π since π was how we denoted a policy earlier; this classifier is just a probabilistic policy since it chooses (a distribution over) actions for each state. Usually, the classifier is some sort of deep neural network.

Objective function. Let's follow our familiar recipe for deriving learning algorithms. First, what does it mean for a given choice of θ to be good? Well, recall that our original goal in reinforcement learning is to find a policy that has high expected rewards. We can translate that directly into a reward function.

First, define a **trajectory** to be any complete sequence of events that happen in an episode. In other words, a trajectory z is a big sequence of states, actions, and rewards:

$$z = [s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T, s_{T+1}].$$

This denotes that the agent started at state s_1, took action a_1, received reward r_1, and ended up in state s_2. It then took action a_2, and so on, until its final action a_T resulted in an end state s_{T+1}, where the whole episode ended. For convenience, let

$$R(z) = \sum_{t=1}^{T} \gamma^{t-1} r_t$$

denote the total discounted reward accrued during this trajectory.[2]

What we want to do is maximize the expected value of $R(z)$ when we play policy $\pi_\theta(a \mid s)$. Let $P(z;\theta)$ denote the probability of a given trajectory z occurring under the policy. (Note that we can't actually compute $P(z;\theta)$ since that would require knowing the transition probabilities of the world, but we'll see how to work around that issue later.) Our objective function is

$$V(\theta) = \sum_{\text{trajectories } z} P(z;\theta) R(z).$$

I use the letter V here since this can be thought of as the value of the policy $\pi_\theta(a \mid s)$.

Training. We want to maximize this function, so let's try to take the gradient and then apply gradient ascent (since we're maximizing, not minimizing). We have that

$$\nabla_\theta V(\theta) = \sum_z \nabla_\theta P(z;\theta) R(z).$$

There's a problem with this expression—how do we sum over all possible trajectories z? As you might imagine, the set of all possible trajectories is extremely large. There's no way compute this sum exactly. Instead, the best thing we could hope for is to convert this expression back into an *expected value* of some quantity over the probability distribution $P(z;\theta)$. That's because if the thing we want to compute is an expected value, we can just sample many trajectories by interacting with the world and playing policy $\pi_\theta(a \mid s)$, and compute the average.

It turns out that there's a clever way to convert this gradient into an expectation. First, it is straightforward to show the following:

$$\begin{aligned} \nabla_\theta \log P(z;\theta) &= \frac{1}{P(z;\theta)} \nabla_\theta P(z;\theta) \\ \nabla_\theta P(z;\theta) &= P(z;\theta) \cdot \nabla_\theta \log P(z;\theta). \end{aligned}$$

The first line follows from the chain rule; the second line is just rearranging the terms in the first line.

[2] Again, note that if we don't care about discounting, we can just set $\gamma = 1$.

Now let's see what happens when we plug this in to our expression for $\nabla_\theta V(\theta)$:

$$\begin{aligned}\nabla_\theta V(\theta) &= \sum_z \nabla_\theta P(z;\theta) R(z) \\ &= \sum_z P(z;\theta) R(z) \nabla_\theta \log P(z;\theta) \\ &= \mathbb{E}_\theta[R(z) \nabla_\theta \log P(z;\theta)],\end{aligned}$$

where $\mathbb{E}_\theta$ denotes the expected value when choosing actions according to the probability distribution $\pi_\theta(a \mid s)$. We have successfully written everything in terms of an expectation! So, any time we sample a trajectory, the expected value of the quantity $R(z)\nabla_\theta \log P(z;\theta)$ is exactly the gradient we should use to update θ.

Finally, let's return to the question of how to compute $\nabla_\theta \log P(z;\theta)$. The probability of z happening is just the probability of starting in s_1, then taking a_1 in state s_1, then transitioning to s_2, and so on. So, we have that

$$\log P(z;\theta) = \log P(s_1) + \log \pi_\theta(a_1 \mid s_1) + \log T(s_1, a_1, s_2) + \log \pi_\theta(a_2 \mid s_2) + \log T(s_2, a_2, s_3) + \cdots .$$

As I pointed out earlier, we can't actually compute this since we don't know the start state probabilities and transition probabilities $T(s, a, s')$. However, we *do* know that these things do not depend on θ. Thus, when we take the gradient with respect to θ, all those terms will just become 0, and we'll be left with

$$\nabla_\theta \log P(z;\theta) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t).$$

Each of these terms can be computed by backpropagating through the model. So overall, we have that the gradient we care about is

$$\nabla_\theta V(\theta) = \sum_z \nabla_\theta P(z;\theta) R(z) = \mathbb{E}_\theta \left[R(z) \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right].$$

Full algorithm. Putting all the pieces together, here is the full basic version of a policy gradient algorithm. This is also known as the REINFORCE algorithm.

> Initialize θ randomly
> **for** $e = 1, 2, \ldots$ **do**
> Sample trajectory z using $\pi_\theta(a \mid s)$ containing T total actions.
> $\theta \leftarrow \theta + \eta R(z) \cdot \sum_{t=1}^{T} \log \pi_\theta(a_t \mid s_t)$

This simply does the gradient ascent update with learning rate η every time a new trajectory is sampled.

There are many ways to improve this algorithm. The primary issue with the basic REINFORCE is that even though the quantity we use to estimate the gradient has the right

expected value, it also has very high variance. There are many techniques to lower this variance, such as actor-critic approaches, proximal policy optimization (PPO), etc. All of them use the same basic mathematical trick to estimate the gradient of $V(\theta)$, and thus are all policy gradient methods. We don't have time to cover these methods in this class, but more advanced classes on reinforcement learning will cover these.