USC CSCI 467: Introduction to Machine Learning
Lecture Notes

Robin Jia

Fall 2023

Contents

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.

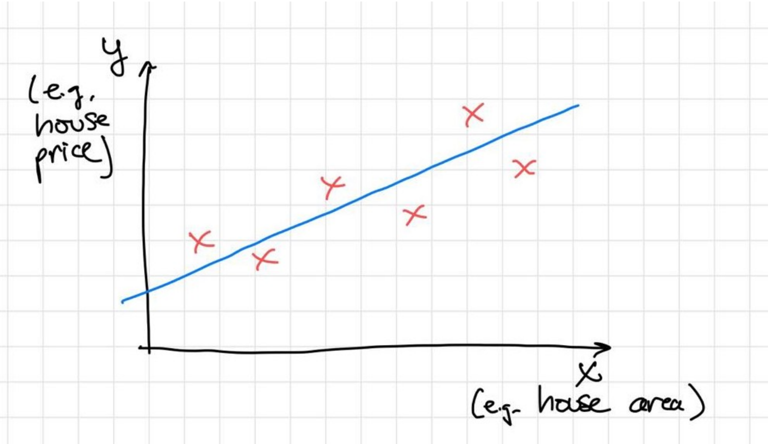


Figure 1.1: Linear regression in 1 dimension. Here we have six examples (in red), each of which is an (x, y) pair. We fit a line (in blue) which allows us to map any x to a predicted value for y.

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]

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

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

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

[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.

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)

```
Choose initial guess $x^{(0)}$
for $t = 1, \ldots, T$ do
    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

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

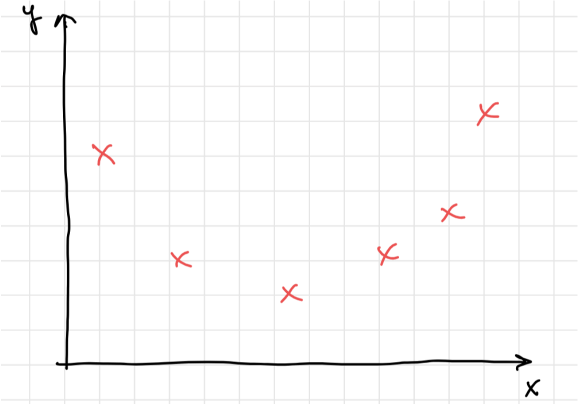


Figure 1.2: A regression dataset that cannot be fit with a straight line.

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.

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.

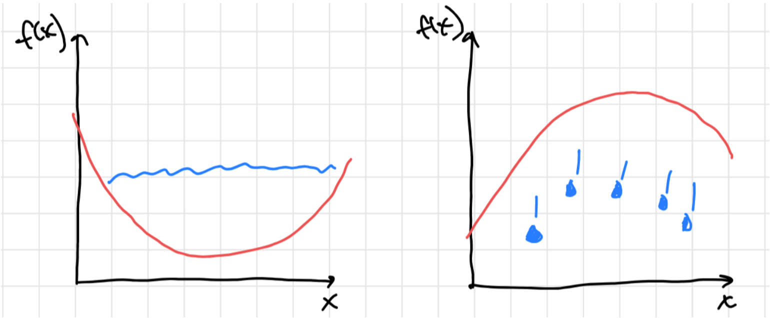


Figure 1.3: Left: A convex function "holds water." Right: A concave function "spills water." While this is not a precise definition, it can be a useful way to remember what a prototypical convex or concave function looks like.

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 $t \in [0, 1]$,*

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

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.

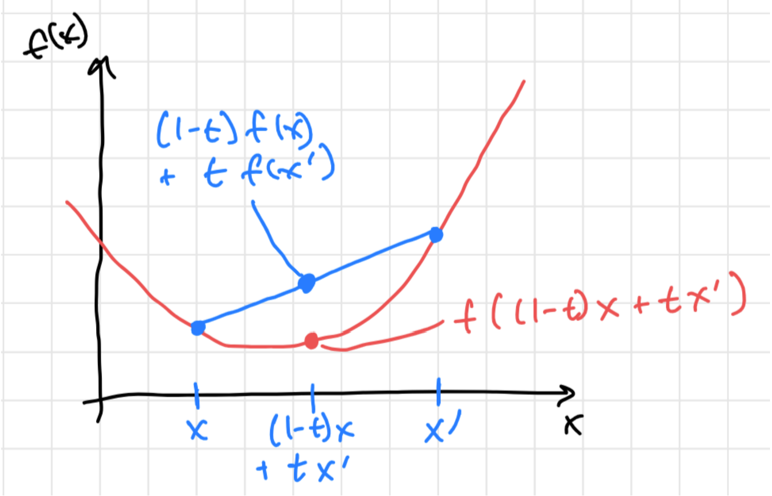


Figure 1.4: A function is convex if and only if every line segment you draw connecting two points on the function lies above the function itself.

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.

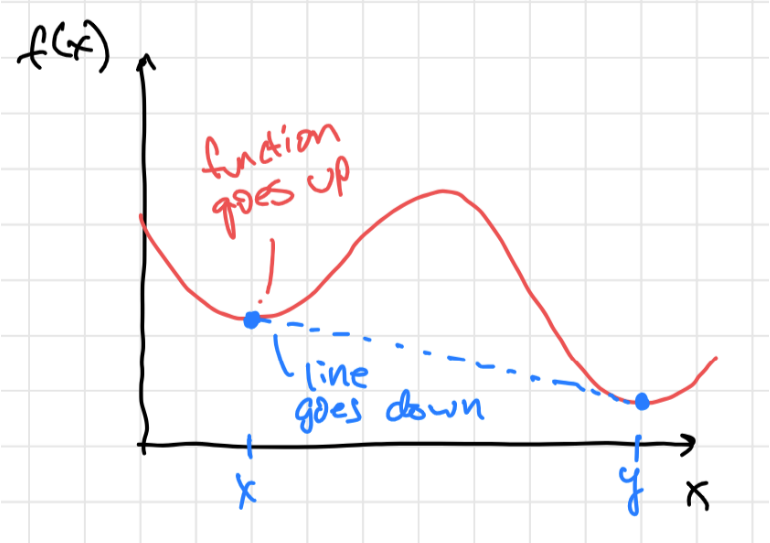


Figure 1.5: A sketch of why a convex function cannot have two distinct local minima (unless they are both equally good). If we draw a line connecting the worse local minimum (x) to the better one (y), that line must be going down, but the function around the worse local minimum must be going up. This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

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 $(1-t)x + ty = x + t(y-x) \in B_\epsilon(x)$. 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

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

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

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.

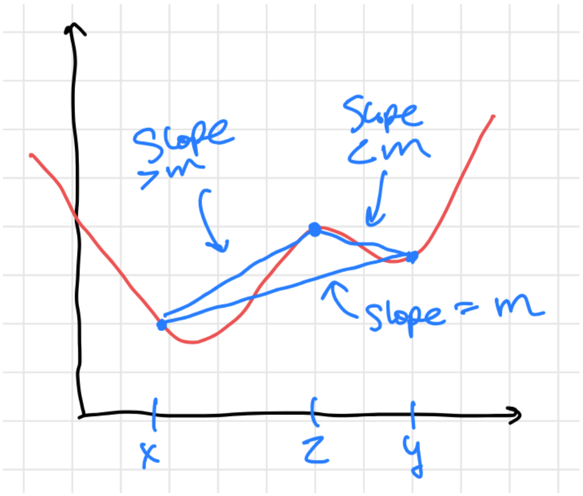


Figure 1.6: An illustration for the mean value theorem argument showing that a function with $f''(x) \geq 0$ everywhere must be convex.

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:

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

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:

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

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

$$\begin{aligned} \mathcal{L}(\theta) &= \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta) && (1.6) \\ &= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma}(y^{(i)} - \theta^\top x^{(i)})^2\right) && (1.7) \end{aligned}$$

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

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

[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.

it converts products into sums, and sums are easy to differentiate. We have:

$$\begin{aligned}
\mathcal{LL}(\theta) &= \log\left(\prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma}(y^{(i)} - \theta^\top x^{(i)})^2\right)\right) && (1.8)\\
&= \sum_{i=1}^{n} \left(\log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2\sigma}(y^{(i)} - \theta^\top x^{(i)})^2\right) && (1.9)\\
&= n\log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2\sigma}\sum_{i=1}^{n}(y^{(i)} - \theta^\top x^{(i)})^2. && (1.10)
\end{aligned}$$

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:

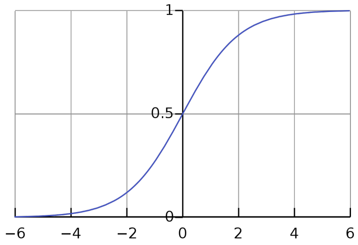
$$p(y = 1 \mid x; w) = \frac{1}{1 + \exp(-w^\top x)}, \tag{2.2}$$

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.

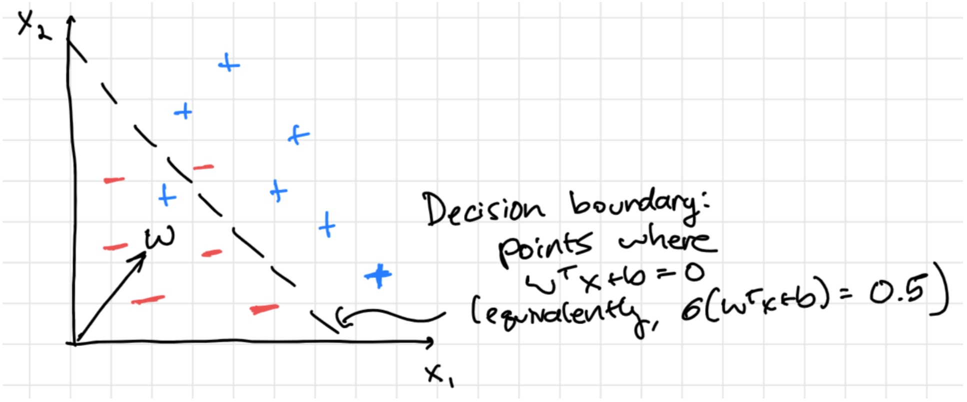


Figure 2.1: Illustration of logistic regression decision boundary. The decision boundary is the hyperplane defined by the equation $w^\top x + b = 0$. This hyperplane is always perpendicular to the weight vector w. Points on the decision boundary are predicted by the model to be equally likely to be 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 $y = -1$.

[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.

Putting everything together, we arrive at this objective function:

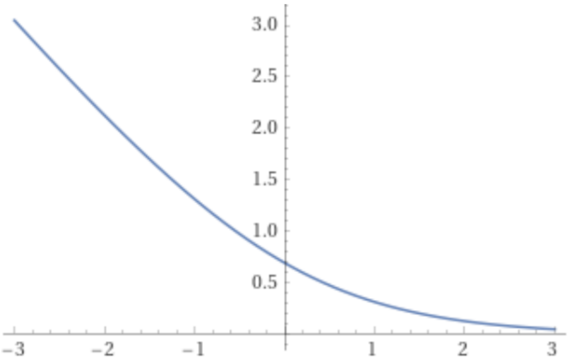
$$\log \mathcal{L}(w) = \sum_{i=1}^{n} \log p(y^{(i)} \mid x^{(i)}; w) \tag{2.5}$$

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

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 margin. You see that we are very unhappy when the margin is very negative, and then this amount of unhappiness decreases.

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:

$$\begin{aligned}\nabla_w L(w) &= \nabla_w \frac{1}{n}\sum_{i=1}^{n} -\log\sigma(y^{(i)}\cdot w^\top x^{(i)}) && (2.8)\\ &= \frac{1}{n}\sum_{i=1}^{n} -\sigma(-y^{(i)}\cdot w^\top x^{(i)})\cdot y^{(i)}\cdot x^{(i)}. && (2.9)\end{aligned}$$

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 $W = \{w^{(1)}, \dots, w^{(C)}\}$ 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 $\{1, \dots, C\}$, 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.

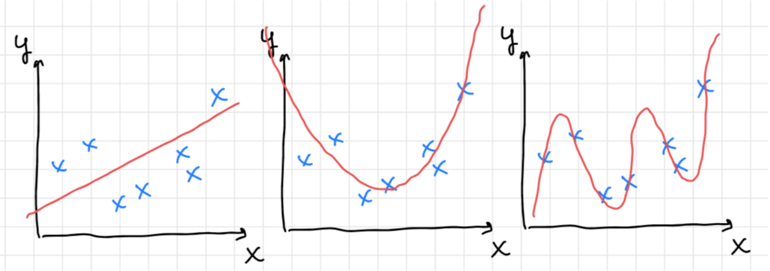
Chapter 3

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:

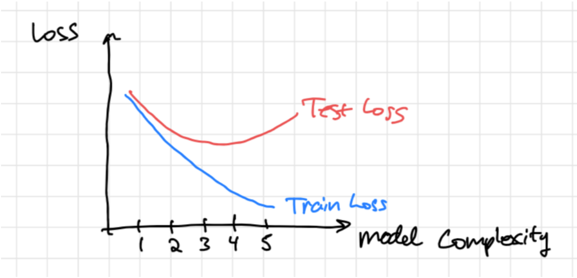


The first function seems to be too simple to fit the data well. We call this **Underfitting**. The second function seems to roughly capture the key trends. What about the last function? It perfectly fits all of the data, so it achieves zero training loss. But it seems like it has done so by fitting a 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

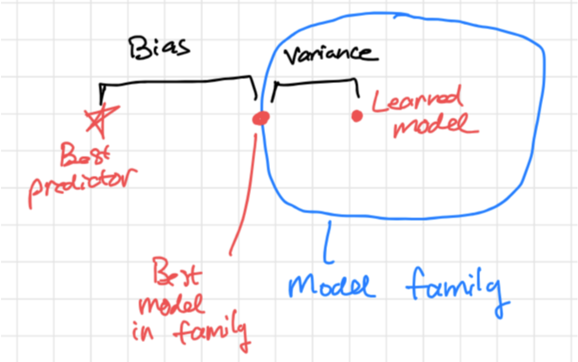


Figure 3.1: Illustration of the bias-variance trade-off. Bias describes the difference between the optimal predictor and the best predictor in the model family. Variance describes the difference between the best predictor in the model family and the predictor that we actually learn. Both together define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

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 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.

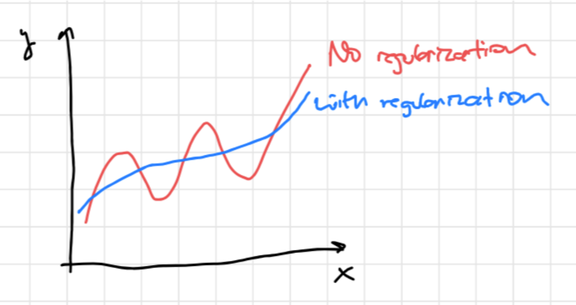
3.4.1 L_2 Regularization

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

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

We will maximize $P(w \mid D)$ by taking the log, so this becomes

$$\sum_{j=1}^{d} -\log(\sigma\sqrt{2\pi}) + \frac{1}{2\sigma^2} w_j^2 = C - \frac{1}{2\sigma^2}\sum_{j=1}^{d} w_j^2 = C - \frac{1}{2\sigma^2}\|w_j\|^2.$$

This is exactly L_2 regularization, where σ controls the strength of the regularization. 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.