USC CSCI 467: Introduction to Machine Learning
Lecture Notes

Robin Jia

Fall 2023

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.

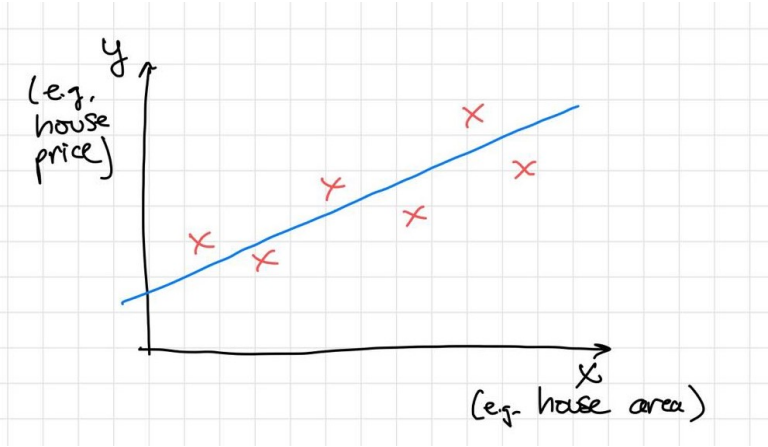

Figure 1.1: Linear regression in 1 dimension. Here we have six examples (in red), each of which is an (x, y) pair. We fit a line (in blue) which allows us to map any x to a predicted value for y.

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, ..., 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

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

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:

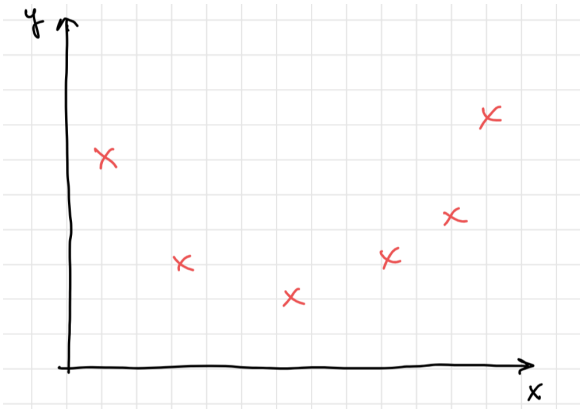

Figure 1.2: A regression dataset that cannot be fit with a straight line.

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.

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

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

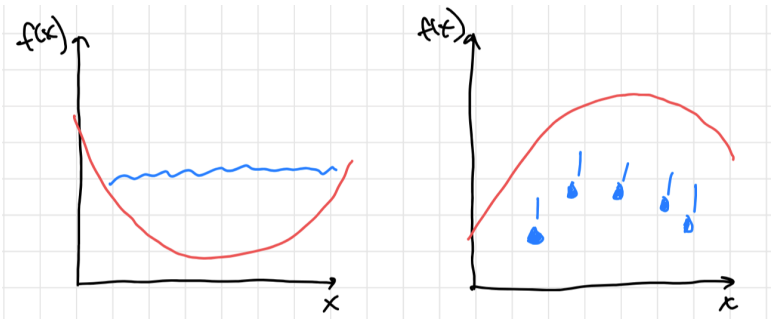

Figure 1.3: Left: A convex function "holds water." Right: A concave function "spills water." While this is not a precise definition, it can be a useful way to remember what a prototypical convex or concave function looks like.

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.

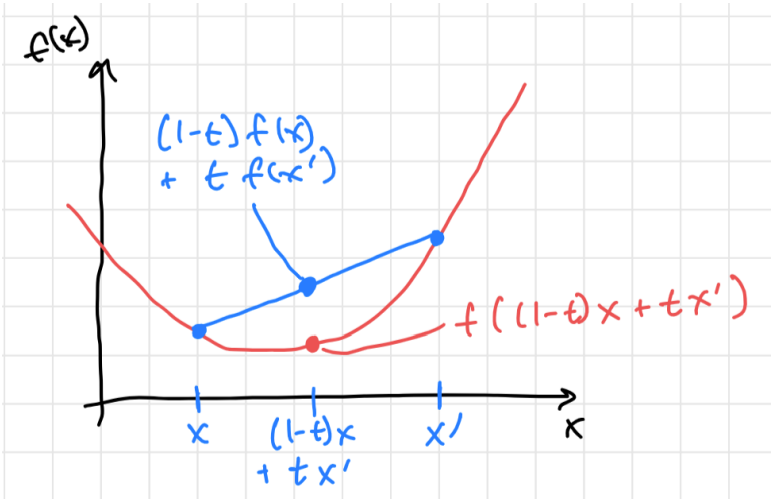

Figure 1.4: A function is convex if and only if every line segment you draw connecting two points on the function lies above the function itself.

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 $\|x' - x\| < \epsilon$.*

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

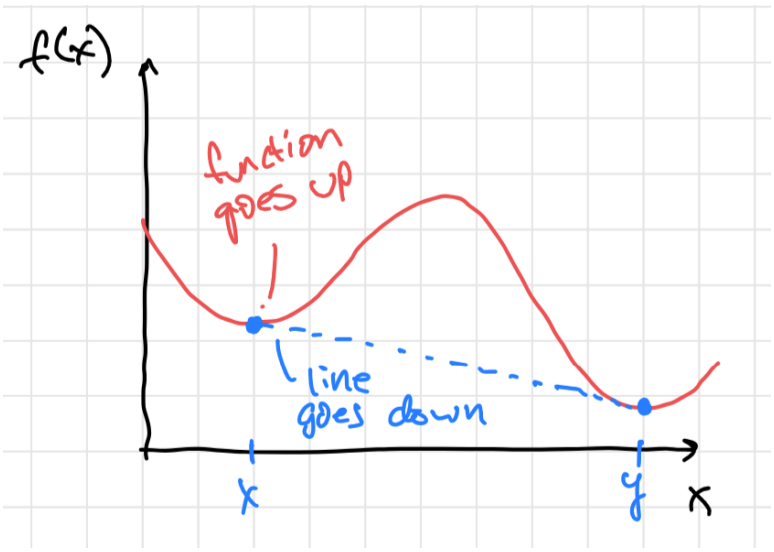

Figure 1.5: A sketch of why a convex function cannot have two distinct local minima (unless they are both equally good). If we draw a line connecting the worse local minimum (x) to the better one (y), that line must be going down, but the function around the worse local minimum must be going up. This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

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

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

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.

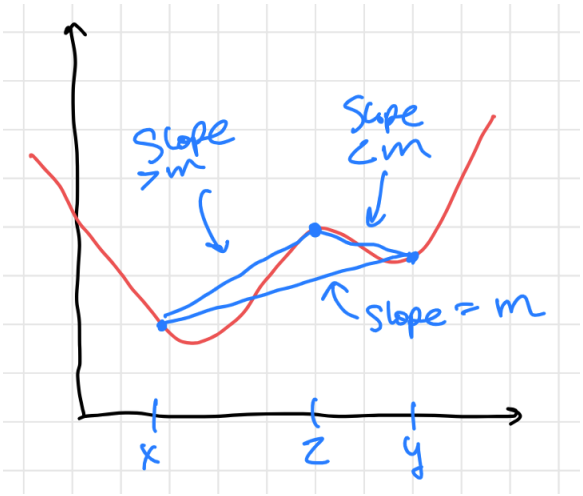

Figure 1.6: An illustration for the mean value theorem argument showing that a function with $f''(x) \geq 0$ everywhere must be convex.

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 Normal Equations: A Closed-form Solution

I should mention that specifically for linear regression, there is actually a way to arrive at a closed-form solution for w. I don't think it's super important to remember the details, but I want to sketch how it works.

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.

$$\nabla_w L(w) = \frac{1}{\cancel{n}} \sum_{i=1}^{n} \cancel{2} \cdot \left(w^\top x^{(i)} - y^{(i)} \right) \cdot x^{(i)} = 0 \tag{1.6}$$

$$\sum_{i=1}^{n} w^\top x^{(i)} \cdot x^{(i)} = \sum_{i=1}^{n} y^{(i)} \cdot x^{(i)} \tag{1.7}$$

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

$$\sum_{i=1}^{n} w^\top x^{(i)} \cdot x^{(i)} = \sum_{i=1}^{n} (x^{(i)\top} w) \cdot x^{(i)} \qquad \text{Dot product is symmetric} \tag{1.8}$$

$$= \sum_{i=1}^{n} x^{(i)} \cdot (x^{(i)\top} w) \qquad \text{Multiply scalar on right instead of left} \tag{1.9}$$

$$= \sum_{i=1}^{n} (x^{(i)} x^{(i)\top}) w \qquad \text{Matrix multiplication is associative} \tag{1.10}$$

$$= \left(\sum_{i=1}^{n} x^{(i)} x^{(i)\top} \right) w \qquad \text{Factor out } w \tag{1.11}$$

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

$$\sum_{i=1}^{n} x^{(i)} x^{(i)\top} = X^\top X. \tag{1.12}$$

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

$$(X^\top X)\theta = X^\top y. \tag{1.13}$$

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:

$$w = (X^\top X)^{-1} X^\top y. \tag{1.14}$$

Non-invertibility. Besides just the fact that we can do linear regression by solving a linear equation, what have we learned? One thing worth noting is that we run into issues when $X^\top X$ is not invertible. Well, when would it be non-invertible? 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 θ. Essentially, when there are redundant features, there are many equally good values of θ (e.g., you could learn a very positive weight for θ_i and very negative weight for θ_j, and they would cancel out). This makes things potentially unstable!

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. The solution is only unique when $X^\top X$ is invertible. Numpy has a special function (`numpy.linalg.pinv`) that will compute the pseudoinverse for you.

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.