USC CSCI 467: Introduction to Machine Learning
Lecture Notes

Robin Jia

Fall 2023

Chapter 1

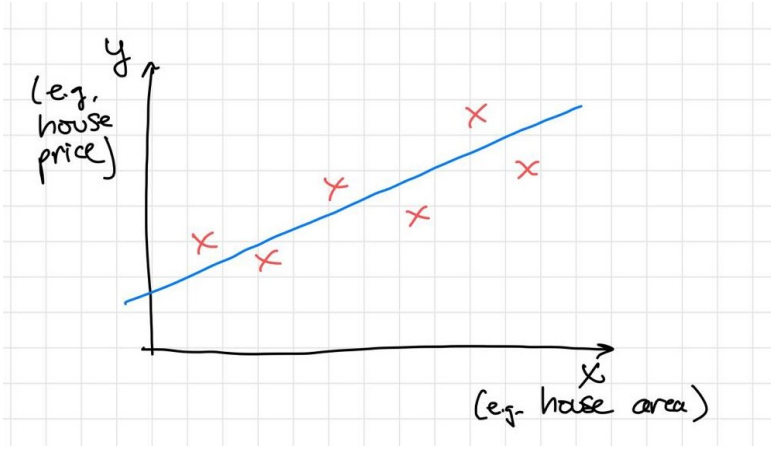
Linear Regression

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.



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 $w^{(T)}$