Stat 442 Lecture Notes

Spring Semester, 2025

Last Updated: February 19, 2025

1 Wednesday 2/5: Welcome, what is statistical learning, basics of supervised learning

1.1 What is statistical learning?

Classical statistics is built on the scientific method. It assumes that we start with a hypothesis, and then we go out and we collect data that can be used to test this hypothesis. Importantly, the statistical methods that we will use to test this hypothesis are pre-specified; we choose them before we ever look at our data.

For example, maybe we come up with the hypothesis that college students who sleep more than 7 hours per night have higher average GPAs than those who sleep less than 7 hours per night. We then go out and take a random sample of students, and ask them all if they sleep more than 7 hours per night, and also ask for their GPA. We then use a t-test to test if there is a statistically significant difference in GPA between students who sleep more or less than 7 hours per night.

Statistical learning is different. While there is no single, satisfying definition of statistical learning, it refers generally to a collection of tools used to make sense of complex data. Unlike classical statistics, where a hypothesis is defined before data collection, statistical learning often involves discovering patterns in data without an initial hypothesis.

In the context of our motivating example, suppose that we collect a large survey of students at a college. We are generally interested in what factors impact the GPA of a college student. We search through the data, try different models, and end up realizing that "hours of sleep per night" seems to be an important factor in determining GPA: more important than, say, major, whether or not you are a varsity athlete, etc. However, we also realize that the relationship between hours of sleep per night and GPA does not look linear. We decide to turn "hours of sleep per night" into a binary variable: yes/no do you sleep more than 7 hours per night. We have ended up with the same "model" as before, but this was a much more exploratory process. It started with a general task of *prediction* (as opposed to a specific model or hypothesis), and used *variable selection* and *feature engineering* to settle on a final model. I would consider this to be an example of statistical learning.

In recent decades, as datasets have become larger and computers have become more powerful, there have been a proliferation of complex methods for analyzing data that are often labeled as machine learning methods. In general, however, there is no clear dividing line between statistics and machine learning. Linear regression and logistic regression, which you all studied in detail in Stat 346, are certainly examples of machine learning methods: they help us learn from data. In fact, they are both methods for *supervised learning*, which is one of two primary types of statistical learning.[1]

- In *supervised learning*, we start with a response variable of interest and a set of potential predictor variables (also known as explanatory variables). Our general goal is to use the predictor variables to explain or predict the response variable.
- In *unsupervised learning*, we just start with a collection of variables: there is no specific response variable. Our goal is to find structure or patterns in the data.

In this course, we will spend around 9 weeks on supervised learning, and only 2 weeks at the end on unsupervised learning. However, I do not want to give you the impression that unsupervised learning is less important! Unsupervised learning is extremely important, and is in fact the back-bone of recent generative AI technology. We start with supervised learning because it has a more natural connection to classical statistics and builds more naturally on your previous work. Hopefully, some of the concepts that you learn during our supervised learning unit will help you be equipped to delve more deeply into unsupervised learning in the future if you choose.

There is one more type of learning that has become really important in recent years, partly due to the proliferation of generative AI. This is the field of *semi-supervised learning*, in which we have a (potentially small) set of *labeled data* (data with predictors + response and a (potentially much larger) set of *unlabeled data* (data that is missing the response). While we will not cover this in lecture, it would make a great *final project topic*. Throughout the semester, I will try to flag these topics as they come up!

[1] I would be remiss if I did not mention *reinforcement learning*, which is a very important topic in machine learning but does not relate as directly to drawing insights from a given dataset.

1.2 Supervised learning

1.2.1 Introduction

As mentioned, you all already know at least two methods for supervised learning: linear regression and logistic regression. In general, these represent one from each of two classes of supervised learning methods.

- Regression: we use this term for any method that is used to predict a quantitative response variable.
- Classification: we use this term for any method that is used to predict a categorical response variable.

As a reminder, a quantitative variable can be either continuous or discrete, but it must be a number. A categorical variable can be either ordinal (the categories have orders) or unordered. In this class, we will not cover any methods for considering ordinal response variables: if this topic interests you, it is another *potential final project topic*.

Linear regression and logistic regression are basic examples of supervised learning methods. In the past few decades, much more complex algorithms have been developed, and have exploded in popularity. There are a few reasons for this proliferation in complex methods.

- Datasets have gotten bigger. As we will learn in this class, complex models are often only appropriate when applied to enough data. Thus, it is natural that the "big data" era has gone hand-in-hand with the development of complex statistical learning methods.
- Computers have gotten better, faster, and cheaper, which has enabled the fitting of complex models.
- Free, open-source software programs like R and python have allowed researchers to develop algorithms and then easily share them with non-experts. This has allowed many methods to become popular.

Given the huge number of statistical learning methods, we could spend each class this semester learning a new method, such that at the end you are left with a huge cookbook of methods to choose from for your future data analysis needs. In fact, this class will be a little bit like this. But, there is a problem with this approach of treating statistical learning like a cookbook. New machine learning methods get published every day, and the state-of-the-art is constantly changing. If your goal is to learn a list of methods or algorithms, you will find yourself continually behind and out-of-date! Thus, the goal of this class is not really to teach you a list of methods. Instead, the goal is to teach you the fundamental concepts and overarching themes that go into the development of the methods, such that you can compare methods, recognize their limitations, and learn new methods on your own in the future. This goal helps explain two features of this course:

- We will spend a fair amount of time on older, less state-of-the-art methods. We will use these methods to illustrate general principles and themes, even if they may not be the methods that you will use in practice in your future.
- You will all be doing a *teaching presentation* this semester. You will be responsible for doing independent reading to learn about a method or a concept. You will need to synthesize what you learn, and give a 20 minute presentation to the class explaining the method. This skill mirrors the way you will all interact with machine learning in the future: you will be continually learning and synthesizing new methods.

1.2.2 Notation for datasets and random variables

The backbone of statistical learning is a matrix of predictors $\mathbf{X} \in \mathbb{R}^{n \times p}$ and a vector of length n that stores the responses $\mathbf{y}$. We let n denote the number of *observations* (rows of our dataset) and let p be the number of *variables* (columns of our dataset, usually not including the response): this notation is quite standard in statistics, but isn't always particularly precise.

Your textbook (ISL) denotes pieces of the dataset $\mathbf{X}$ and $\mathbf{y}$ as follows. We observe x_i for $i = 1, \ldots, n$, where $x_i = (x_{i1}, \ldots, x_{ip})^\top$ is a p-vector for each observation. We use $\mathbf{x}_1, \ldots, \mathbf{x}_p$ to denote the columns of $\mathbf{X}$; each is an n-vector representing its own variable. Your textbook says that it will use bold lowercase letters anytime the vector has length n and use unbold lowercase letters otherwise: I am sure that I will start messing this up soon, but I will try to be consistent.

The other important distinction is between a random variable and its observed realization. We will use capital, unbold letters to denote random variables. In our example from Section 1.1, we might let X_1 denote the number of hours that a randomly selected student sleeps per night and let Y denote their GPA. Once we observe a particular n students for our dataset, we let their sleep values be $\mathbf{x}_1 = (x_{11}, x_{12}, \ldots, x_{1n})^\top$ and we let their GPA values be $\mathbf{y} = (y_1, \ldots, y_n)^\top$.

Consider a dataset that contains information on n pets. The response variable is the weight of the pet $\mathbf{y}$, and we have one single predictor variable: the type of pet. This is a categorical variable that takes on values {cat, dog, hamster, rabbit}. I could represent $\mathbf{X}$ in this case as a matrix with one column; where the column stores the actual values {dog, dog,

cat, dog, hamster, cat, dog, ...}. But, as you all know from Stat 346, it is likely going to be convenient for fitting to instead let $\mathbf{X}$ be a matrix with four columns, that store binary variables indicating "is this a dog", "is this a cat", etc. This simple example shows us how notation can get complicated: do we say that $p = 4$ or $p = 1$ for this dataset? Practically speaking, the $p = 4$ is likely more relevant! But it can depend!

We can also get into problems when counting n sometimes. Suppose we take measurements on 920 students, who are spread out between 8 schools. The schools are randomly assigned to either have required yoga PE class, or to have traditional PE class. While we have 920 students, the students within a given school are not *independent* of one another: there may be factors other than the yoga class that are making their test scores more similar to one another. On the other hand, the schools underwent random assignment and can be safely assumed to be independent of one another. Thus, is the sample size n the 920 students or the 8 schools? It turns out that for this *clustered* or *hierarchical* data, we have a notion of *effective sample size* that is somewhere between 920 and 8, and depends on how correlated the students are within the schools.

We should also mention that data does not always come to you in a form where it looks like $\mathbf{X} \in \mathbb{R}^{n \times p}$ and $\mathbf{y} \in \mathbb{R}^n$. Consider the task of image classification. You get a set of 500 images, which are stored as 400×400 pixel images, where each pixel records a numerical value between 0 and 255 for red, green, and blue. Each image is associated with a label in $\{\text{cat}, \text{dog}\}$, and you observe $y_1, \ldots, y_{500}$. In this case, the RGB values in the pixels in the images are your predictor values, but they don't come in a simple $n \times p$ matrix. We need to reshape the data so that each image i has an associated vector $x_i \in \mathbb{R}^{400 \times 400 \times 3}$ that stores all three color values for all 400×400 pixels. It will also likely be convenient to code y_i such that a 1 denotes a dog and a 0 denotes a cat, such that y becomes numerical. When there are more than 2 classes, such as $\{\text{cat}, \text{dog}, \text{fish}\}$, we will need more than one column to store our response variables numerically.

Who knew that counting n and p could be so complicated! In this class, just think of it as the number of rows and columns (minus the response columns) of the data, and remember these subtleties in case they ever come up.

In our supervised learning unit, we will start with *regression*. Thus, for the rest of today, we assume that Y is numerical.

1.3 Regression

In a regression problem, we assume that our response variable Y is related to a set of predictors $X = (X_1, \ldots, X_p)$ via some model that can be written as

$$Y = f(X) + \epsilon. \tag{1}$$

The function $f()$ is a fixed but unknown function that represents the systematic information that X provides about Y, and ϵ is a random error term, which should satisfy the condition that $E[\epsilon] = 0$ and that ϵ is independent of X. These conditions ensure that $f()$ is capturing all parts of Y that can be explained by X, and ϵ captures the parts of Y that cannot be explained by X. The task in regression is to come up with a good approximation for the unknown function $f()$. More specifically, we want to find a function $\hat{f}()$ such that, $Y \approx \hat{f}(X)$.

We talked really briefly about two methods that you might use to find a good $\hat{f}$, that sort of represent opposite ends of the spectrum in terms of how "wiggly" they are. We talked about:

- Linear regression, which you have all seen before.
- KNN, which many of you have not seen before. We will go into a lot more detail on KNN next class.

1.3.1 Two possible goals

There are two reasons why we might want to come up with a function $\hat{f}()$ such that, $Y \approx \hat{f}(X)$.

1. We might want to be able to come up with predictions $\hat{Y} = \hat{f}(X)$ for future realizations of data where Y itself is not observed.
2. We might want to understand which predictors in X are most associated with Y in order to draw a scientific conclusion, or at least take a step towards drawing a scientific conclusion.

When beginning a regression problem, it is important to clearly define which of these two reasons is more important to you. This will aid you in choosing the best strategy for approaching the problem!

As we will see throughout the semester, these two goals can sometimes be at odds with one another! If we only care about prediction, we do not need our function $\hat{f}()$ to be interpretable: we can use an extremely complex model (known as a "black box"), and we will be happy as long as $\hat{Y} \approx Y$. On the other hand, if we want to draw scientific

conclusions, then we might wish to use a simple, interpretable model for $\hat{f}()$ that lends itself to rigorous inference. We should also note that it is not always an either-or situation. Sometimes, we want to make good predictions and also understand which variables are contributing significantly to the predictions. Managing the tradeoff, or lack thereof, between these two goals will be one of our fundamental themes for the semester.

We will pick up with everything else next class!

2 Monday, Feb 10: The bias variance tradeoff, as illustrated with KNN and linear regression

Like last class, we will focus on a really simple regression setting. We observe a single response $\mathbf{y} \in \mathbb{R}^n$ and a single predictor $\mathbf{x} \in \mathbb{R}^n$. We assume that our observations are i.i.d. realizations of random variables (X, Y), where

$$Y = f(X) + \epsilon.$$

We assume that $E[\epsilon] = 0$ and $\epsilon \perp\!\!\!\perp X$, but the function $f()$ is unknown. Our goal is to find a function $\hat{f}$ such that $Y \approx \hat{f}(X)$. In this unit, we will talk about different algorithms for finding such a function $\hat{f}$!

We will assume that we are looking for $\hat{f}$ that makes the squared error loss:

$$\left(Y - \hat{f}(X)\right)^2$$

small on average. Today we are going to talk a lot about this goal. And we are going to talk about how well KNN and linear regression each achieve this goal.

For today, we are using squared error loss everywhere. But please remember that this is not the only way to measure model accuracy! The exact math of the bias-variance decomposition that we will discuss today holds only for squared error loss, but similar concepts apply for other loss functions.

Here is the agenda for today:

1. Training error vs. expected test error.
2. The bias-variance decomposition of the expected test error.
3. What is the ideal function for minimizing the expected test error?
4. How do KNN and Linear Regression each approximate this ideal function?
5. R Demo of KNN and Linear regression and the bias variance tradeoff.

We might move a little bit fast, but HW1 is going to give you a chance to practice all of these concepts.

2.1 Training error vs. test error

If we use our dataset $\mathbf{y} \in \mathbb{R}^n$ and $\mathbf{x} \in \mathbb{R}^n$ to *train* a function $\hat{f}$, then our *training error* is:

$$MSE_{\text{train}} = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \hat{f}(x_i)\right)^2.$$

For a fixed function $\hat{f}$ that we already trained, if we observe some test points $x_i^{\text{test}}, y_i^{\text{test}}$ for $i = 1, \ldots, n_{\text{test}}$, we can compute our test error as

$$MSE_{\text{test}} = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n^{\text{test}}} \left(y_i^{\text{test}} - \hat{f}(x_i^{\text{test}})\right)^2,$$

where the important thing is that the points $x_i^{\text{test}}, y_i^{\text{test}}$ were not used to compute the function $\hat{f}$.

While in practice we usually do compute MSE_{test} over some fixed test set that has size n_{test}, we don't want our notion of model performance to be specific to the test set that we happened to see. We are likely studying MSE_{test} to estimate how big our error will be on a random new datapoint. In this case, we might be computing this for the particular function $\hat{f}$ that we already computed. In this case, our expected prediction error conditional on our particular function $\hat{f}$ that we already computed is:

$$E_{X^{\text{test}}, Y^{\text{test}}} \left[\left(Y^{\text{test}} - \hat{f}(X^{\text{test}})\right)^2 \right],$$

where the function $\hat{f}$ is not treated as random.

If we want to evaluate the potential of an algorithm or a procedure to make good predictions, then we want to take into account the randomness in our training set. We want to acknowledge that we could have seen a different training set that would have given us a different function $\hat{f}$. With this in mind, in the next section we will define a more complex notion of expected prediction error. This more complex notion will let us come up with a very beautiful decomposition!

2.2 The bias-variance tradeoff

Suppose that we train our model $\hat{f}$ on a dataset $\mathbf{X}^{\text{train}}, \mathbf{y}^{\text{train}}$. Since this dataset is random, the function $\hat{f}()$ is also random. Now, suppose that we would like to know about the expected prediction error for a new datapoint with $X = x^{\text{test}}$ and unknown $Y = y^{\text{test}}$. The prediction error itself is $\left(y^{\text{test}} - \hat{f}(x^{\text{test}})\right)^2$, but to find the *expected* prediction error we need to take the expected value over multiple sources of randomness.

For our purposes, let's treat x^{test} as fixed. We want to find the average prediction error that we would obtain if we repeatedly (1) sampled training sets of size n from the population, (2) refit the function $\hat{f}$ using this training data, and (3) evaluated the squared prediction error for a datapoint drawn with $X = x^{\text{test}}$. So, we want to evaluate:

$$E_{\mathbf{X}^{\text{train}},\mathbf{y}^{\text{train}},Y|X=x^{\text{test}}}\left[\left(Y - \hat{f}(X)\right)^2 \mid X = x^{\text{test}}\right] = E_{\mathbf{X}^{\text{train}},\mathbf{y}^{\text{train}},Y|X=x^{\text{test}}}\left[\left(Y - \hat{f}(x^{\text{test}})\right)^2 \mid X = x^{\text{test}}\right].$$

For brevity, I am not going to keep writing the subscripts in the expected values. I am going to assume that we are treating x^{test} as fixed but taking the expected value over all other randomness. But remember that it is always good to know what exactly you are taking the expected value over!

To break this quantity down, we are going to do a clever trick that involves adding and subtracting 0 twice. This is a very common proof technique! After we do this, we expand our big squared quantity by thinking of it as $(a+b+c)^2$, and we apply linearity of expectation to put each term in its own expected value.

$$\begin{aligned}
E\left[\left(Y - \hat{f}(x^{\text{test}})\right)^2\right] &= E\left[\left(Y - f(x^{\text{test}}) + f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})] + E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)^2\right] \\
&= E\left[\left(Y - f(x^{\text{test}})\right)^2\right] + E\left[\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)^2\right] + E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)^2\right] \\
&+ 2E\left[\left(Y - f(x^{\text{test}})\right)\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\right] \\
&+ 2E\left[\left(Y - f(x^{\text{test}})\right)\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right] \\
&+ 2E\left[\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right].
\end{aligned}$$

We will now argue that each of the cross terms is 0. We start with the orange term, and note that:

$$\begin{aligned}
E\left[\left(Y - f(x^{\text{test}})\right)\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\right] &= \left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right) E\left[Y - f(x^{\text{test}})\right] \\
&= \left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\left(E[Y] - f(x^{\text{test}})\right) \\
&= \left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\left(f(x^{\text{test}}) - f(x^{\text{test}})\right) = 0.
\end{aligned}$$

The first two equalities follow because the quantities $f(x^{\text{test}})$ and $E[\hat{f}(x^{\text{test}})]$ are not random, since x^{test} is a constant. The last equality follows since, because we are conditioning on $X = x^{\text{test}}$, $Y = f(x^{\text{test}}) + \epsilon$, and so by our assumption that $E[\epsilon] = 0$, $E[Y] = f(x^{\text{test}})$.

We now consider the purple term. We note that $Y - f(x^{\text{test}})$ is independent of the training dataset, since $f()$ is non-random and Y is a new realizations. On the other hand, $\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)$ is a function only of the training data, since x^{test} is non-random. Thus, this term has the form $E[ab]$ where a and b are independent. So we can write

it as:

$$E\left[\left(Y - f(x^{\text{test}})\right)\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right] = E\left[Y - f(x^{\text{test}})\right] E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right]$$
$$= E\left[Y - f(x^{\text{test}})\right]\left(E[\hat{f}(x^{\text{test}})] - E\left[\hat{f}(x^{\text{test}})\right]\right) = 0.$$

Finally, we consider the green term.

$$E\left[\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right] = \left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right) E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right]$$
$$= \left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)\left(E[\hat{f}(x^{\text{test}})] - E\left[\hat{f}(x^{\text{test}})\right]\right) = 0.$$

Now that we know that all three cross terms are 0, we know that

$$E\left[\left(Y - \hat{f}(x^{\text{test}})\right)^2\right] = E\left[\left(Y - f(x^{\text{test}})\right)^2\right] + E\left[\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)^2\right] + E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)^2\right].$$

Our next goal is to understand each of these terms!

1. The first term can be written as:

$$E\left[\left(Y - f(x^{\text{test}})\right)^2\right] = E\left[\left(f(x^{\text{test}}) + \epsilon - f(x^{\text{test}})\right)^2\right] = E\left[\epsilon^2\right] = Var(\epsilon).$$

   This is our irreducible error term! It comes from the inherent noise in our data that cannot be explained by X.
2. In the second term,$f(x^{\text{test}})$ and $E[\hat{f}(x^{\text{test}})]$ are not random, so we do not need the expected value! This just becomes

$$\left(f(x^{\text{test}}) - E[\hat{f}(x^{\text{test}})]\right)^2,$$

   which we call the squared bias. Is $\hat{f}(x^{\text{test}})$ equal to $f(x^{\text{test}})$ on average? If so, then $\hat{f}$ is unbiased and this term will disappear.
3. Finally, in the third term:

$$E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)^2\right] = Var\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right] + E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right]^2.$$

   First, note that $Var\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right] = Var\left[\hat{f}(x^{\text{test}})\right]$, because $E[\hat{f}(x^{\text{test}})]$ is not random. Then, note that $E\left[\left(E[\hat{f}(x^{\text{test}})] - \hat{f}(x^{\text{test}})\right)\right] = E[\hat{f}(x^{\text{test}})] - E\left[\hat{f}(x^{\text{test}})\right] = 0$. So, this third term is simply the variance (with respect to the training set) of $\hat{f}(x^{\text{test}})$.

This was a lot of work, but we can finally say that, at a given point x^{test},

$$E\left[\left(Y - \hat{f}(x^{\text{test}})\right)^2\right] = Var(\epsilon) + Bias(\hat{f}(x^{\text{test}}))^2 + Var(\hat{f}(x^{\text{test}})).$$

This is the bias variance decomposition, which will be important throughout the semester!

The $Var(\epsilon)$ term is our irreducible error. No matter how good we are at estimating f, this is just the amount of noise in our data, so we will always have this error. On the other hand, we can reduce the bias and the variance terms if we pick a better statistical learning algorithm for estimating f.

Without giving too much away: very simple models tend to have high bias but low variance. Very complex (wiggy) models have low bias (they are never constrained!) but very high variance (they overfit to the training data). To minimize our expected prediction error, we are always looking for functions that hit the sweet spot of complexity! This sweet-spot is usually at the minimum of a U-shaped curve for test-set error! You will make U-shaped curves on your homework this week!

2.3 The ideal function $\hat{f}$

Let's briefly forget the training data! If our goal was to minimize

$$E_{X,Y}\left[\left(Y-\hat{f}(X)\right)^2\right]$$

and we had all of the resources in the world, what would we choose for $\hat{f}$?
Well, under the law of total expectation, we can write this as:

$$E_X\left[E_{Y|X}\left[(Y-\hat{f}(X))^2 \mid X\right]\right].$$

In the inner expected value, X is no longer random, and solving for the point-wise minimum is easy. We have that:

$$\underset{c}{\text{argmin}}\ \mathrm{E}_{Y|X=x}\left[(Y-c)^2 \mid X=x\right] = E[Y \mid X=x].$$

This comes from the fact that an expected squared error is always minimized at a mean. You will prove this on HW1, and you have likely seen it before.
If $E[Y \mid X=x]$ minimizes the point-wise expected prediction error at every x, then the function $\hat{f}(x) = E[Y \mid X=x]$ is the function that minimizes the overall expected prediction error. Of course, we do not know the joint distribution of X and Y, and so we cannot simply set $\hat{f}(x)$ to be this conditional expectation. But we can develop methods that attempt to approximate $\hat{f}(x) = E[Y \mid X=x]$ under various sets of assumptions! As we will see today, both KNN and linear regression try to approximate $E[Y \mid X]$.

2.4 How close do KNN and Linear Regression get to this ideal function?

Let's review a really simple case. We have a single numerical response variable Y and a single numerical predictor variable X. We observe 100 datapoints, so $n = 100$ and $p = 1$. Our goal is to use our dataset $\mathbf{x} = (x_1, \ldots, x_n)$, $\mathbf{y} = (y_1, \ldots, y_n)$ to come up with a function $\hat{f}$ such that, on new, unseen data, $Y \approx \hat{f}(X)$. Today, we will consider two methods for coming up with $\hat{f}$.

2.4.1 Simple linear regression

We restrict our attention to $\hat{f}$ that have the form $b_0 + b_1 x$ for $b_0, b_1 \in \mathbb{R}, b_1$. This makes our problem of finding the best $\hat{f}$ easier, because we limited the complexity of the model class that we are considering.
Based on our training data, we let:

$$\hat{\beta}_0, \hat{\beta}_1 = \underset{b_0,b_1}{\text{argmin}} \sum_{i=1}^{n} (y_i - b_0 - b_1 x_i)^2. \tag{2}$$

We then let $\hat{f}(X) = \hat{\beta}_0 + \hat{\beta}_1 X$. Under the assumption that $E[Y] = \beta_0 + \beta_1 X$ for some true, unknown β_0 and β_1, this solution directly approximates the best possible function $E[Y \mid X]$.
However, if this assumption does *not* hold, then the linear model is too limiting. The result of limiting our model class so much is that we might have a lot of bias. If our linearity assumption does not hold, then no matter how much data we have $E[\hat{f}(x)]$ just cannot be that close to f. This is why simple models can have a lot of bias– we did not give them enough complexity to be able to capture the true function!
You are all experts in linear regression! So we will not spend too much more space in the notes on it.

2.4.2 K-nearest neighbors

KNN is at the opposite end of the spectrum from linear regression. The premise is quite simple. KNN lets

$$\hat{f}(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i, \tag{3}$$

where $N_k(x)$ is a function that returns the k points in the training set that are closest to the input point x according to some distance metric (for us, Euclidean distance). So, avoiding equations: KNN makes predictions by finding the k training observations with x_i closest to x, and predicts that the response for x will be the average of the responses for those k points.

The hyper-parameter k is chosen by the user. When $k = n$, KNN just predicts $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ for all observations, regardless of x. As k decreases, the functions returned by KNN get increasingly wiggly and flexible. When $k = 1$, the KNN function is a step function that can be arbitrarily wiggly, and that always has 0 training error.

Note that KNN directly approximates $E[Y \mid X = x]$ *locally*, making no assumptions on the structure of $E[Y \mid X]$. When k is small, the estimation of $E[Y \mid X = x]$ uses only data that are really close to x. This lets our prediction function get arbitrarily wiggly– we will not run into an issue of bias. However, with small k, there is a lot of variance in each individual prediction, since it is made using very few datapoints. When k is large, there is less variance in the predictions, but we do not let our predictions be as wiggly, and so we introduce more bias. Thus, KNN is a great place to see the bias-variance tradeoff at work.

A few notes on KNN:

- KNN is non-parametric; we cannot write down the function $\hat{f}$ without storing basically the entire dataset. The number of effective parameters grows with the sample size n.
- Linear regression in theory takes some time to train, but it is nearly instantaneous to make new predictions. KNN involves no training step, but it could be computationally expensive to obtain new predictions.

2.5 Time for an R demo!

The RMarkdown document that we go over in class will be posted on GLOW!

3 Thursday, Feb 13: adding more predictor variables!

Recall our typical regression setting. We assume that our data are i.i.d. realizations of random variables (X, Y), where

$$Y = f(X) + \epsilon.$$

We assume that $E[\epsilon] = 0$ and $\epsilon \perp\!\!\!\perp X$, but the function $f()$ is unknown. Our goal is to find a function $\hat{f}$ such that $Y \approx \hat{f}(X)$.

More specifically, we would like to find $\hat{f}$ that minimizes the expected squared error loss for a new, unseen datapoint (X, Y). So ideally, we are looking for

$$\text{argmin}_{\hat{f} \in \mathcal{F}} \;\; E_{X,Y}\left[\left(Y - \hat{f}(X)\right)^2\right], \tag{4}$$

where $\mathcal{F}$ is some class of functions. On your homework, you will argue that if $\mathcal{F}$ were totally unconstrained, we would want to set $\hat{f}(x) = \mathrm{E}[Y \mid X = x]$. This argument was also in the Lecture 2 notes, but we skipped it.

This is where we hit practical issues. We do not know the joint distribution of X and Y, and so we cannot pick $\hat{f}$ to be this ideal function $\mathrm{E}[Y \mid X = x]$. And we really cannot search efficiently over all possible real-valued functions $\mathcal{F}$ to find a good choice for $\hat{f}$. So, a statistical learning algorithm typically restricts the class $\mathcal{F}$ to make this task doable. So far in this class, we have discussed two possible methods for picking $\hat{f}$. It turns out that both of these can be viewed as approximating $E[Y \mid X = x]$; they just do this using different sets of assumptions.

Up until now, we have been assuming that we just have one predictor variable, and so X is a scalar. Today, we will let X be a vector, meaning that we have p predictor variables $X_1, \ldots, X_p$. This is going to introduce a lot more nuance to the comparison between linear regression and KNN!

Agenda :

1. Linear regression in high dimensions.
2. KNN in high dimensions.
3. R demo, and overall comparison of linear regression vs. KNN, without preprocessing.

3.1 Linear regression in high dimensions

We restrict our model class $\mathcal{F}$ to

$$\mathcal{F} = \{f : f(x) = b_0 + b^T x, b_0 \in \mathbb{R}, b \in \mathbb{R}^p\},$$

such that (4) becomes

$$\text{argmin}_{b_0 \in \mathbb{R}, b \in \mathbb{R}^p} \;\; E_{X,Y}\left[\left(Y - b_0 - b^T X\right)^2\right]. \tag{5}$$

As you know from Stat 346, it is convenient to append a column of 1s to our predictor matrix $\mathbf{X}$ so that we are just optimizing over a single $b \in \mathbb{R}^{p+1}$. That way, we don't need to keep writing the intercept separately. Adopting this convention, with linear regression we approximate (5) by letting

$$\hat{\beta} = \underset{b \in \mathbb{R}^{p+1}}{\arg\min} \quad \sum_{i=1}^{n} \left(y_i - b^T x_i\right)^2. \tag{6}$$

You know 1,000 things about this estimator from Stat 346. For example, as long as $n > p$, the solution to (6) has a closed-form: $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y$. Furthermore, under the assumption that $E[Y \mid X] = \beta^T X$ for some true unknown β (i.e. under the assumption that the true model is linear), this estimator is BLUE (best unbiased linear estimator), where "best" here means lowest variance.[2] This is the *Gauss-Markov Theorem*, and I am assuming that you saw it in Stat 346.

Here are a few pros and cons of linear regression to keep in mind. We will discuss these a lot, and also revisit them in our R demo.

- **Pro: efficiency:** If $n > p$, we have a closed-form solution. This means that the function is efficient to train. It is also really efficient to generate new predictions.
- **Con: identifiability:** If $n < p$, we cannot solve for $\hat{\beta}$ because there is no unique solution; the model is not identifiable.
- **Con: bias:** If the true model is not linear, we will have bias! Because the linear model might simply not be flexible enough to model the true relationship between the predictors and the response.
- **Pro/Con: variance:** Because it is not super flexible, linear regression should be a low-variance method. Unfortunately, when we start adding a lot of irrelevant or redundant predictors to the model, the variance can get very high.
- **Pro: interpretability:** Linear regression is interpretable! We can look at the estimated function $\hat{f}$ and say what variables are important, and in what direction they are contributing to our predictions.
- **Pro: usability:** Linear regression is easy to use "out-of-the-box" for a non-expert. There isn't anything to *tune* if we just regress y on all the variables (which we can do as long as $p < n$. Any refinement that the user wants to do after that is pretty easy to explain/understand.

3.2 KNN in high dimensions

We know that the best solution to (4) is $E[Y \mid X = x]$. So, we assume only that $E[Y \mid X = x]$ is smooth enough that:

$$E[Y \mid X = x] \approx E[Y \mid X \text{ is in a neighborhood of } x].$$

We then approximate this function directly by letting:

$$\hat{f}(x) = \frac{1}{k} \sum_{i \in N_k(x)} y_i,$$

where $N_k(x)$ is the k training set datapoints that are closest to the desired test point x.

We didn't emphasize this point when we only had a single predictor variable, but the notion of "closest" datapoints requires a distance metric! Our distances are now measured in p-dimensional predictor space, and we actually have many different distance metrics that we could use. Unless otherwise specified, assume that we are using Euclidean distance. This means that:

$$N_1(x^{\text{test}}) = \underset{i=1,\ldots,n}{\arg\min} \sqrt{\sum_{j=1}^{p} (x_j^{\text{test}} - x_{ij})^2} = \underset{i=1,\ldots,n}{\arg\min} ||x^{\text{test}} - x_i||_2.$$

Importantly, if your different predictors x are measured with different units, this neighbor function might not even make sense! This neighbor function treats all features the same. If one of our features is "price of house" and another feature is "number of bedrooms in house", one of these has values in the hundreds of thousands and the other has values that are likely below 10. In this setting, the euclidean distance between points will be totally dominated by price, and bedrooms will be basically ignored. Because of this, you should almost certainly scale your features before applying KNN: usually we would do this by dividing every feature by its standard deviation. This creates a distance

[2] Do you remember what it means to call this a *linear* estimator? A hint is that it would still be linear even if we used polynomial features like X^2.

function where all features contribute equally. Below, we will talk about how this can lead to its own issues.

There are some pros and cons of KNN that are relevant regardless of the dimension p.

- **Con: usability:** we need to choose k. This means that KNN cannot be used directly "out of the box"- we probably want to choose k using cross-validation.
- **Pro/Con: bias/variance tradeoff:**
  - With small k, we can approximate functions that are arbitrarily wiggly. So we don't need to worry that our function class $\mathcal{F}$ is not sufficiently flexible.
  - When k is small, each prediction is generated with very few datapoints, which means that we have high variance.
  - When k is large, a prediction might be generated with points that are actually far away from the test point x^{test}, which can introduce bias.
  - If we have enough data points n such that our predictor space is densely populated with training points, we can pick a pretty large k and still have it be the case that every training point in $N_k(x)$ is actually close to x. This means that we won't have much bias, but we will also have low variance since k is large.
  - For a given problem, we may or may not find a "sweet-spot" k that makes KNN work really well!
- **Pro / Con: efficiency:**
  - Computational cost of training is essentially free. Just store the training dataset.
  - At testing, we need to compute distances between the new test point and all points in the training set. This could be slow if you implement it naively, but people have cool algorithms for finding nearest neighbors efficiently.

There are some additional cons of KNN that come up when the dimension p is large.

- **Impact of irrelevant features:** Suppose that we have p features in our dataset, but only a small subset of these features actually impact the response Y. All of these features are used in computing the neighbors of x! So, we are letting totally irrelevant features contribute to our distance metric. This could introduce either bias or variance. The bias would be because I am doing a bad job selecting the meaningful neighbors. The variance would be because my prediction can vary based on extra random noise in some random irrelevant feature.
- **Lack of interpretability:** Unlike linear regression, running KNN on a high dimensional dataset tells you nothing at all about what features are most associated with Y. If you wanted to remove irrelevant features to improve performance, KNN provides no built in guidance for doing this. This is in contrast to linear regression.
- **Computational cost:** Storing the training set is actually quite expensive when n and p are large. So even though "training the model" just involves "storing the training set"- this is not free! Neither is computing lots of pairwise distances in high dimensions, or searching through high dimensional space for neighbors! We will need to come up with more clever algorithms for KNN to get around this.
- **Curse of dimensionality:** It turns out that, in high dimensions, points just become far from one another! So selecting neighbors in high dimensions is just a really hard task! The entire notion of neighbors hardly makes sense anymore, because our high-dimensional feature space will not be densely populated unless n is really large compared to p. So even though we can theoretically do KNN when $p > n$ (unlike for linear regression), it is going to work very poorly in this setting.

3.3 Comparing linear regression and KNN, without preprocessing

See the R Demo from class.

We ran out of time before we could talk about what to do about this! We will talk about that next time!

3.4 Miscellaneous

People asked good questions in class, so here are some extra points that I wanted to add!

- Another big advantage of linear regression over KNN is that we have theory that lets us do inference (e.g. prediction intervals!) to rigorously quantify our uncertainty for linear regression.
  - This inference depends on some number of assumptions about the underlying data generating mechanism. For example, the typical inference that we do assumes homoskedasticity (equal variance of our errors), but robust/sandwich standard errors let us relax this assumption (at the possible cost of statistical power).
  - We like that KNN requires no assumptions whatsoever: we don't need to believe that our model is truly linear, etc. But a downside of this lack of assumptions is that we don't have many tools available to make

prediction intervals! This is a typical tradeoff: we would love to avoid assumptions, but usually avoiding assumptions also means we can't say much about our final answer!

- Off the top of my head, I don't know much about inference for KNN. But I included some random thoughts below!
- For KNN with $K = 3$, each prediction is based on three datapoints! Without additional assumptions, there is going to be a LOT of uncertainty associated with this prediction.
- Later in the semester, we will talk about conformal inference, which lets us make prediction intervals for any machine learning model. This could be applied to KNN!
- To make a prediction interval, we need both an estimate but also a measure of sampling uncertainty associated with this estimate. Maybe we could use bootstrapping to estimate the SE of a KNN prediction and make an interval based on that? When I googled "inference for KNN", I found a bunch of suggestions about bootstrapping! It would be interesting to try this out and see if it works!
  - A sketch of the idea: Get B different KNN predictions for a point x^{test} by using different random bootstrap subsamples of the training dataset each time. Then, compute the standard deviation of these B predictions. Then, get your main prediction $\hat{y}^{test}$ using all of the data. Let $\hat{y}^{test} \pm 2\hat{SD}(\hat{y}^{test})$ be your prediction interval. Would this cover the true y^{test} 95% of the time? I am not sure because KNN is strange and bootstrapping does have theory/requirements. Let's try it out in a future R demo during lecture.

- We mentioned that KNN requires storing the entire dataset to make test predictions, and that it is inefficient. This is not strictly speaking true, because people have made smart algorithms for getting around this problem. If you google "KNN with KD tree or Ball tree" you will find a lot of resources on fast algorithms. These are the ones that I learned about in my ML class, but I think a lot of others exist too!
- Someone asked if you need to store the entire training dataset to fit a linear regression model. It turns out that you don't! I will put a reading on your HW about "online learning" for linear regression. The idea is that, instead of storing our training dataset X, y all at once and computing $\hat{\beta} = (X^T X)^{-1} X^T y$, we can instead write $\hat{\beta}$ as:

$$\hat{\beta} = \arg\min_b \sum_{i=1}^{n} (y_i - x^T b)^2$$

  and assume that we only have access to a single example x_i, y_i at a time. We have algorithms that can step along the gradient of this loss function according to one example x_i, y_i at a time in order to find the minimum. So the full data never needs to be stored!
- I'm sure there are things I forgot. Keep asking good questions!

4 Monday, 2/17: Feature selection, feature engineering, and regularization!

We ended last class by talking about how both KNN and Linear Regression can do badly when the number of variables p is very large! There are two distinct problems.

- For linear regression with no preprocessing, increasing p just increases the model complexity of a linear regression model. This allows us to overfit (or memorize) our training set, which leads to very high variance and poor test error.
- For KNN, p doesn't really have anything to do with model complexity, but we do poorly because KNN works best when we have training points that are dense in our feature space (so that the k nearest neighbors to a test point x are actually close to x). As p grows, it is really really hard for points to be dense (n would need to be HUGE), and so we are making predictions with neighbors that aren't really that close (bias) and who our neighbors actually are is affected by noise in so many different predictors (variance).

Real data is high dimensional! So we really need a way around these issues! Today, we will talk about two strategies for avoiding issues with high dimensions.

- **Strategy 1:** Preprocess the data to reduce the dimension before we apply the algorithm. Within strategy 1, we have two sub-strategies.
  - **Feature selection**: You have all used this for linear regression; it's a really natural fit. It definitely *could* be used for KNN, but you would need to pick some sort of selection algorithm that might not have anything to do with KNN.
  - **Feature extraction**: Can do this before KNN, linear regression, or any other algorithm.
- **Strategy 2:** Modify the algorithm using **regularization** to automatically reduce the variance.

- We are going to talk about this in the context of linear regression, not KNN.
- The concept (Regularization!!) will be relevant to any algorithm we use this semester that gets fit by minimizing a loss function. KNN is one of the only algorithms that we will use this semester that doesn't have this loss-function form, and so regularization is not directly applicable.
- It will turn out that Lasso regularization ends up looking a lot like variable selection! But they arise from slightly different concepts!

Let's talk about each of these! I am going to zoom through material today. Chapter 6 of ISL has a really nice treatment of feature selection, feature extraction, and regularization for linear regression. Please read this on your own!

4.1 Feature selection

The idea is that if we have p predictors $\{X_j : j \in \{1, \ldots, p\}\}$ but we don't think that all of them are relevant, we should select a subset $\mathcal{S} \in \{1, \ldots, p\}$. We should then do our statistical learning algorithm using only $\{X_j : j \in \mathcal{S}\}$. If we do a good job with selection, we should improve our performance, because our variance will be smaller (for linear regression) and we will have less curse of dimensionality (for KNN).

4.1.1 Guess and check

You have all done variable selection for linear regression via what I might call "guess and check" before. You have all fit a model with a bunch of variables, and then looked at the p-values and decided only to include variables that seem significant! You have also checked for things like multicollinearity, and removed variables that you are worried are redundant. All of these strategies are great, and show a big advantage of linear regression: we have interpretable tools for built-in variable selection.

4.1.2 Best subset regression

A more automated way to do variable selection would be to try every single subset $\mathcal{S} \in \{1, \ldots, p\}$ and choose the subset that leads to the lowest test set MSE, the lowest AIC or BIC, or is the best according to some other metric.

If p is small, we could just try out fitting least squares models to all of the different subsets $S \subset \{1, \ldots, p\}$. We could then directly compare our metrics for every possible subset, and if we pick the subset that leads to the lowest value we have our solution! This is best-subset regression, and I think you should have heard about it in Stat 346! Best-subset regression is kind of silly, because it is computationally infeasible when p is big, which is exactly the situation in which we need it!

4.1.3 Forward stepwise regression

Since best subset regression is computationally infeasible, one idea is to try to approximate the solution to best subset regression with a more reasonable search strategy. With best subset regression, we would need to try out 2^p different models. With the forward stepwise algorithm suggested below, we need to try out at most $p + (p-1) + (p-2) + ...$ models.

The idea of forward stepwise regression is simple:

- Start with an empty model that only includes an intercept
- Until a stopping criteria is met:
  - Look through all possible predictors that are not yet in the model. Add the predictor that most improves the model at this moment!

We get to decide what we mean by "most improves the model". We could, at each step, add the most significant predictor, or the one that most improves R^2, etc. We could use a stopping criteria such as: "until the BIC stops improving" or "until no variable that could be added has a p-value less than 0.05". If we do this, then the size of our final model is determined for us, using only the training set.

We could also use no stopping criteria, and just go until we run out of predictors, or until the number of predictors is equal to the number of datapoints. If we do this, we get a sequence of nested models, where the variables were added in a greedy order. We could select our final model size by seeing which of these nested models minimizes the test set error, for example. You did this on your homework. This idea of getting a whole sequence of models, and then selecting the one that minimizes test set error, will look a bit like regularization!

Note that backwards stepwise regression is also a thing that you may have learned about in Stat 346. I think it is unsatisfying when we are discussing high dimensional regression, since it cannot be used for $p > n$ (since you cannot fit the initial model to step backwards from).

I think you all know a bit about feature selection for linear regression from Stat 346! So I will not talk too much more about it.

4.1.4 Feature selection for KNN??

KNN does not lend itself to a built-in way to do feature selection, as far as I know. We could "try out fitting KNN with different features included or removed", and compare test MSE for different options. This is a lot like best-subset selection; it is computationally infeasible to try out all possible options. We could also run something like stepwise linear regression as a preprocessing step to KNN. This would be a strange thing to do, but the idea would be that we think we need wiggly functions (hence KNN), but we first want to have some efficient way to select variables that seem associated with Y. People also use a preprocessing method called marginal screening, which just computes the marginal correlation between Y and X_j for $j = 1, \ldots, p$ and only keeps variables j that seem correlated with Y. You could try this out with KNN! A fear with any of these methods is that removing some Xs can alter who is neighbors with who in your data: which might be good but it also might be bad!

4.1.5 Overview:

In general, what does feature selection get us, and what are the risks?

- **Interpretability:** selecting a small number of features makes our final model more interpretable.
- **Usability:** A user needs to tune a hyper-parameter that controls "how many variables to keep." This makes everything a bit more complicated, but at least the tuning parameter is interpretable, and automated procedures exist.
- **Bias:** If we fail to include an important variable, we could introduce bias. In other words, we could accidentally select a model that is too simple.
- **Variance:** The whole point of feature selection is to reduce variance! It definitely does this.
- **Computational efficiency:** Some methods like best subset selection and stepwise regression could be slow. So the actual selection step can be slow. But in general, if we select only some features from our data and use these for the rest of our analyses, we have less data to store and we will make downstream tasks faster.
- **Inference:** If you were hoping that, after variable selection, you could get nice p-values from `lm()` for each of your selected variables, you are wrong! This is the problem of *post-selection inference*, which is my research area! We cannot use the same data more model selection and model inference. If we want to do inference after model selection, we need to refit the selected model on a totally held-out test set. Ask me about this sometime, or do this topic for your final project! There are papers that do fancy math to do inference after stepwise regression!

4.2 Feature extraction

The idea is that if we have p predictors $\{X_j : j \in \{1, \ldots, p\}\}$ but we think that a lot of them are redundant, maybe we can compress the information from the p features into a smaller number of features $\left\{\tilde{X}_k = f_k(\mathbf{X}) : k \in \{1, \ldots, S\}\right\}$. Each of the S new features can contain information from all of the old features.

Consider image classification. Our high-dimensional set of predictors is every pixel in an image. Feature selection will work terribly here: we can't just select some of the pixels for every image and expect to do a good job. But there are probably low-dimensional concepts hidden in the images that can capture all of the information that we need, without storing every single pixel. Thus, image classification is a setting where feature extraction is really helpful but where feature selection makes no sense.

Here are a few examples:

- The new features $\tilde{X}_k$ for $k \in \{1, \ldots, S\}$ could be the first S principal components of the original feature matrix $\mathbf{X}$. We still need to choose S: this is now a tuning parameter.
- It turns out that you can randomly project your original p-dimensional feature matrix $\mathbf{X}$ into a lower dimensional subspace to get $\tilde{X}$. Magical theorems say that the distances between observations are approximately preserved under random projections, and so this can work pretty well as a preprocessing step to KNN.
- PCA and random projections are linear embeddings. You could use something like UMAP or tSNE as a non-linear embedding. These are popular in genomics! They put your points onto a low-dimensional manifold.
- Autoencoders are a really cool way to learn a low dimensional representation of a high dimensional object!

PCA is a really classic example, and I think that you are all familiar with PCA from Stat 346. I included a little sidebar on PCA below, just in case we have time to go over it or just in case you are curious. But how PCA actually works is not really our topic for today.

The general idea of any of these is that we can avoid the curse of dimensionality if we can capture most of the information about our $\mathbf{X}$ variables in fewer dimensions. This is going to work best when we have a lot of redundant predictors.

4.2.1 PCA

In the simplest case, if we assume that our feature matrix X has been centered and scaled so that the mean of each variable is 0 and the standard deviation of each variable is 1, and we also assume that $n > p$, then we can take the singular value decomposition of X, and write it as

$$X = UDV^T,$$

where $U \in \mathbb{R}^{n \times p}$ is a matrix whose columns are orthogonal unit vectors, $D \in \mathbb{R}^{p \times p}$ is a diagonal matrix, and $V \in \mathbb{R}^{p \times p}$ is an orthonormal matrix.

The columns of V define a new set of axes in our predictor space. V_1 represents the direction that contains as much of the variance in X as possible. V_2 represents the direction orthogonal to V_1 that explains as much of the leftover variance as possible, and so on. In PCA-speak, these are called the loadings. They correspond to the eigenvectors of the correlation matrix of the data X, and are ordered in such a way that V_1 corresponds to the biggest eigenvalue, V_2 to the second biggest eigenvalue, etc.

If the columns of V are the axes, then the columns of UD store the position of each datapoint along these axes. We call these the scores.

Let U_r denote the first k columns of u, let D_r denote the first r rows and columns of D, and let V_r denote the first r columns of V. Then, the matrix

$$U_r D_r V_r^T,$$

is the "best" (according to mean-squared-error or L2-norm) rank-r approximation to X.

What this means is that if we let $\tilde{X} = U_r D_r \in \mathbb{R}^{n \times r}$, we have made ourselves r new variables that store "as much of the variation in X as possible" (from a MSE perspective, and among linear embeddings).

The new variables are also independent of one another, so there is no multicollinearity left. We can use this $\tilde{X}$ in our downstream task. We probably only want to do this if it stores a large proportion of the total variance in X: we can plot this proportion of variance vs. r to help decide how many principal components to keep!

4.2.2 Overview

In general, what does feature extraction get us, and what are the risks?

- **Interpretability:** Often, feature extraction can make a final model less interpretable. Occasionally, you can get lucky, and identify your top few principal components as recognizable concepts.
- **Usability:** You need to figure out how to extract features, how many PCs to keep, etc. This adds a tuning task that I think is less straightforward than the selection task.
- **Bias:** If you don't retain enough good information about $\mathbf{X}$, you could introduce bias.
- **Variance:** The point is to reduce variance!
- **Computational efficiency:** Story is the same as for feature selection. Something like UMAP is computationally hard to run, but it saves you time and space down the road because you don't need to store your entire high dimensional dataset anymore.

One more nice thing is that something like PCA has nothing to do with linear regression. We can use PCA before KNN and it works really well: you will do this on your homework! So the techniques we use for feature extraction tend to be very general; even if Chapter 6 of your ISL textbook talks about PCA in the context of linear regression only.

4.3 Regularization

The idea of regularization is to reduce our variance by adding a penalty on model complexity directly into our loss function!

4.3.1 Best subset regression through L0-regularization!

We can revisit the idea of best subset regression, but case it as a regularization problem. We can do linear regression, but we can let

$$\hat{\beta} = \text{argmin}_{b \in \mathbb{R}^p} \quad \sum_{i=1}^{n} \left(y_i - x_i^T b\right)^2 + \lambda ||b||_0, \tag{7}$$

where

$$||b||_0 = \{\#i : b_i \neq 0\}.$$

This is just saying that we want to fit a least squares regression, but we want to prefer a *sparse* solution.

The cool thing about writing the objective function in this way is that the parameter λ directly controls the bias-variance tradeoff. If we set λ to be very large, we will select a model with very few non-zero coefficients. This model will have a lot of bias (even if the *true* model is linear! because we might have accidentally left out important variables). However, the model will have low variance because it is so simple. On the other hand, as λ approaches 0, we approach our un-regularized high-dimensional regression, whose variance we know is very high.

This is a nice idea! But there is one huge problem! Can we actually solve (7)?

Unfortunately, the answer is no! Not efficiently! The only way to solve (7) exactly is to try out all 2^p possible models, which is infeasible. While this first example of regularization is not actually something that we do in practice, the idea turns out to be really useful!

4.3.2 Ridge regression through L2-regularization!

One reason that we cannot solve (7) is that our favorite way to find minimums is to take a derivative, and $||b||_0$ is not differentiable. So, what if we just modified (7) to make it differentiable? Ridge regression solves the following optimization problem:

$$\hat{\beta} = \text{argmin}_{b \in \mathbb{R}^p} \quad \sum_{i=1}^{n} \left(y_i - x_i^T b\right)^2 + \lambda ||b||_2^2, \tag{8}$$

where

$$||b||_2^2 = \sum_{i=1}^{p} b_i^2.$$

Unlike (7), we can solve (12) in closed-form. You will do this on HW2! But ... what does it get us?

If, in truth, $y_i = x_i^T \beta + \epsilon$, then the solution to (12) is biased for β. The amount of bias will grow as λ grows. However, we can prove that, as λ grows, the variance of the solution to (12) always goes down. So, while the least squares solution ($\lambda = 0$) is known to be the Best Linear Unbiased Estimator, it turns out that some solutions to (12) (i.e. solutions for some other values of λ) could have lower expected prediction error than the least squares solution—even though they are biased! This is really cool- but how do we find these λs?

We have not officially talked about K-fold cross-validation yet in this class. While it is a really general technique that has nothing to do specifically with ridge regression, I am going to talk about it right now! Because it is the way we will choose a value of λ for ridge regression.

Here is how we would use K-fold cross-validation in this context!

- Divide our n datapoints into K non-overlapping folds of data.
- For k in $1, \ldots, K$:
  - Let the kth fold be the test set. Let the other $K-1$ folds be the training set.
  - For λ' in a big grid of possible values of λ:
    - Solve (12) with $\lambda = \lambda'$, using the training set.
    - Compute the prediction error on the test set. Save this error for this fold and this value λ'.
- For each λ', add up the test set errors across all K folds.
- Select the λ^* that minimizes the total test set error. *(Or use the 1-SE rule to select a slightly different λ^*. We will come back to this at some point on a HW.)*
- Refit (12), using all n datapoints and λ^*. Let this be your final model.

Ok, so we know that we can solve (12). And we also know that we can use cross validation to pick a λ^* that hopefully hits a sweet spot of bias and variance. There are a few practical notes to talk about.

- It is good to center and scale your X variables before you apply ridge. The reason you should center them is that we want our model to spiritually include an intercept, but we don't want to put the penalty λ on the intercept. It is good to scale them because we don't want a variable with different units to get unfairly penalized by the penalty term- we want all of the coordinates of b_i to be on the same scale.
- The solution to (12) is not sparse, so is not necessarily interpretable. Sometimes, people look at the output of (12) and look at which variables got their coefficients shrunken very small. They then decide to remove those, and refit a final model using least squares to only the remaining variables. This makes the final model more interpretable. We should note that standard "lm()" based inference is invalid after doing this (it's the same issue as doing inference after variable selection).
- The amount by which the ridge solution shrinks each element of $\hat{\beta}$ actually depends on the singular vectors of X! Which is cool! More redundancy in the columns of X will lead to more shrinkage. This makes sense, because we also know that more redundancy in the columns of X means more variance in a least squares solution, so more shrinkage is needed!
- There are a lot of cool things we could study for ridge. It's mathematically beautiful! For example, there is a way to view ridge regression as a continuous analog of regression in principal components! That is so cool. A few cool things about ridge will be on your HW or in the textbook. For now, we will move on!

4.3.3 Lasso regression through L1 regularization

It is sort of a bummer that ridge regression does not give us sparse solutions. It does reduce the variance of the fitted model, but just by looking at the fitted model it is hard to tell that we have reduced its complexity. That is where lasso comes in. Magically, it turns out that the solution to

$$\hat{\beta} = \text{argmin}_{b \in \mathbb{R}^p} \quad \sum_{i=1}^{n} \left(y_i - x_i^T b\right)^2 + \lambda ||b||_1, \tag{9}$$

where

$$||b||_1 = \sum_{i=1}^{p} |b_i|,$$

tends to be sparse! We call this Lasso regression!

A lot of things are the same as with ridge. The parameter λ governs a bias-variance tradeoff. You should be sure to center and scale your variables before you use Lasso. You should use cross validation to select the best value of λ.

By sparse, we mean that, for large enough λ, all of the coefficients will be shrunk towards 0, but several will be exactly 0. This means that the Lasso actually performs variable selection! This is such a simple change compared to (12). Why would the solution suddenly be sparse? This is a fun optimization fact that is often explained using Figure 1.

Of course, an issue is that $||b||_1$ is no longer differentiable. In fact, this is a non-convex optimization problem. So, while Ridge had a closed-form solution, Lasso does not. We need to solve (13) with an iterative algorithm called coordinate descent.

In a setting where all variables are actually related to the response variable, Ridge regression tends to have slightly better test set performance than Lasso (at their respective optimal values of λ, which do not need to be the same). However, in a setting where the true solution is sparse, Lasso can outperform ridge. So, in practice, both are useful tools, and you might want to compare them for a given problem using cross validation. Personally, I love the Lasso because I love the sparse property for interpretability. But some problems are not sparse! (Remember our image example!).

Note that, if you are using Lasso for variable selection, you might want to handle categorical dummy variables in a special way. You might want to make sure that groups of βs representing categories of the same variable are either selected or not selected together. You can do this with the group-lasso.

4.3.4 Overview

In general, what does regularization get us, and what are the risks?

- We will work with so many algorithms this semester that can be written as: "minimize this loss function over a training set". Any algorithm that has this form has the risk of overfitting to the training set! But adding regularization on model complexity can always help us avoid overfitting! This is beautiful and general!
- **Usability:** We need to do cross validation to choose λ. Luckily, for things like Ridge and Lasso, this is really easy and is just built into the software.

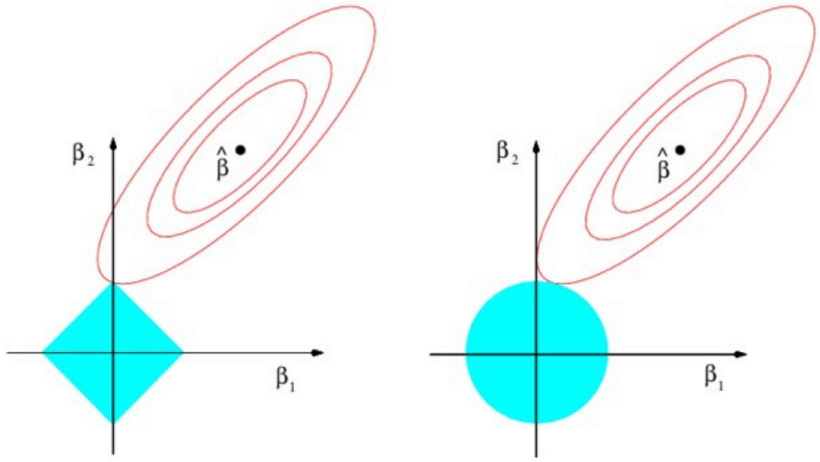

Figure 1: A figure taken from ISL that illustrates why the Lasso prefers sparse solutions while Ridge does not. If we are optimizing the least squares objective (oval) subject to the constraint that either the L1 or L2 norm of β is not too large, the our solution will lie in the place where the contours of our least squares objective first hit our constraint set. For the L2 constraint (circle), this can happen anywhere. For the L1 solution (diamond), this is more likely to happen on one of our 4 pointy corners.

- **Bias:** Regularization introduces bias.
- **Variance:** Regularization reduces variance. That is the whole point!
- **Computational efficiency vs. interpretability:** From an interpretability standpoint, we most would have wanted to do L0 regularization. This encourages sparse solutions, but doesn't actually shrink the values of coefficients that are in our model. However, L0 regularization is computationally infeasible. On the other hand, L2 regularization is really fast and efficient, but the solution is not sparse and so isn't as interpretable. L1 regularization is a nice happy medium! It does not have a closed form solution but we can still solve it, and we get a sparse solution (with coefficients that also got shrunk).

4.4 Wrap up

I am sure I forgot some key points and I am sure we will be rushed! We will do our Lasso/Ridge R demo next class, and use that as a way to revisit some points!

In general, even if Ridge and Lasso got smushed into one lecture in this class, I hope that you will remember them forever! They are really fundamental concepts! We have ridge/lasso logistic regression too- it's not only a linear regression thing! It's a really beautiful example of the bias-variance tradeoff and the interplay between optimization and statistics!

5 Thursday, 2/20: Wrap up regularization, introduction to classification

5.1 Recap from last time on Regularization

If we have n training datapoints $(x_1, y_1), \ldots, (x_n, y_n)$, then ordinary least squares solves the following objective function on the training set:

$$\hat{\beta}_{OLS} = \arg\min_{b \in \mathbb{R}^p} \sum_{i=1}^{n} (y_i - b^T x_i)^2 \tag{10}$$

When our true model is linear, this is unbiased for the "true β". But, it can have high variance, especially if the number of predictors p is large and many of those predictors are irrelevant.

Last class, we mentioned three regularization methods. The point of all three is to reduce the variance of (10), but all three do this at the expense of possibly introducing bias. The three methods were:

- **Best subset:**

$$\hat{\beta}_{subset,\lambda} = \underset{b\in\mathbb{R}^p}{\arg\min} \sum_{i=1}^{n}(y_i - b^T x_i)^2 + \lambda||b||_0 \tag{11}$$

- **Ridge:**

$$\hat{\beta}_{ridge,\lambda} = \underset{b\in\mathbb{R}^p}{\arg\min} \sum_{i=1}^{n}(y_i - b^T x_i)^2 + \lambda||b||_2^2 \tag{12}$$

- **Lasso:**

$$\hat{\beta}_{lasso,\lambda} = \underset{b\in\mathbb{R}^p}{\arg\min} \sum_{i=1}^{n}(y_i - b^T x_i)^2 + \lambda||b||_1 \tag{13}$$

We didn't finish the lecture notes from last class, so we will start by going through those! But there are also a few points that I want to add.

5.1.1 Connection between a penalty and a "budget"

To understand Figure 1, I think it is important to know (11), (12), and (13) can all be written as constrained optimization problems. See page 243 of ISL for more info.

Let's focus on Lasso for simplicity. It turns out that (13) is equivalent to:

$$\hat{\beta}_{lasso,\lambda} = \underset{b\in\mathbb{R}^p}{\arg\min} \sum_{i=1}^{n}(y_i - b^T x_i)^2 \qquad \text{subject to: } ||b||_1 < s_\lambda.$$

This just says to solve the least squares problem, but subject to the constraint that $||b||_1 < s$ for some budget s. It turns out that every λ you could use for Lasso has a corresponding s that gives you the same solution. Sometimes this formulation can be useful, because there are smart people who know a lot about constrained optimization!

Some very important overall pros and cons:

- Best subset regression is interpretable! And, if we select the "true" subset of important variables, we have unbiased coefficients for the "true" non-zero elements of β! Unfortunately, it is really computationally expensive!
- Ridge regression has a closed form solution so it is really efficient! But it doesn't lead to an interpretable, low-dimensional model. It just reduces variance.
- Lasso regression is interpretable and can be solved more efficiently than best subset! We do still shrink the non-zero coefficients, so even if we select the "true" subset of important variables, we will have bias!

5.1.2 Modern comparisons!

This paper was published in 2020, which is crazy recent for something about such fundamental topics: `https://www.jstor.org/stable/pdf/26997932.pdf`.

The big idea is that recent computational advances have actually made best-subset regression more doable! So ... should we just do best-subset? Is it the gold-standard over lasso? Apparently not! Best subset and lasso can each outperform each-other in certain settings, and the overall winner is actually the "relaxed lasso". I recommend you read the paper!

5.1.3 R demo

I want to spend a bunch of time going over an R demo. See the link on GLOW!

5.2 A quick aside about degrees of freedom

We have been talking about how the bias/variance tradeoff is a function of model complexity. I have been calling this "how wiggly a function is"- or how capable the model is of overfitting to the training set.

One way to formally measure the complexity of a mode is the *degrees of freedom*. I am sure that you have all heard of degrees of freedom before, but I am not sure that you have ever seen it defined formally. As far as I can tell, ISL avoids defining this formally: they just talk about it being the effective number of parameters in a model.

I feel like we should mention the official definition at least once. Assume that $x_1, \ldots, x_n$ are fixed. Let $g(y) = \hat{y}$, where g is a "model fitting procedure", that takes in the observed values of y as input and trains a prediction model and

returns the training set predictions. We are not writing this as $\hat{f}$ because it does not refer to a specific fitted model: it refers to the procedure. And we are not writing it as a function of X because the Xs are just hiding in the background since they are not random. According to this paper, `https://www.stat.berkeley.edu/~ryantibs/papers/sdf.pdf`,

$$df(g) = \frac{1}{\sigma^2} \sum_{i=1}^{n} Cov(Y_i, g(Y)_i),$$

where σ is $Var(Y_i)$, which is assumed to be constant across i.

Calculating degrees of freedom is really straightforward for simple procedures like linear regression. For more complicated procedures, actually computing this is really complicated! So we might need to estimate or approximate some degrees of freedom.

I don't think we are going to spend too much more time on this. ISL says it is "outside the scope of the book". But I think we should mention DF or approximate DF for some models we have seen so far.

- Linear regression with p coefficients: p degrees of freedom.
- KNN: we are making around n/k unique predictions, so we have around n/k degrees of freedom. In fact, n/k is an unbiased estimator for the degrees of freedom apparently. But the true DF depends on the distribution of the Xs: how many of your "regions" overlap affects the number of effective parameters.
- Ridge: You can prove directly from the definition that ridge regression with p coefficients has $< p$ degrees of freedom, and the DF shrinks as λ grows. This is because the penalty makes it strictly less wiggly than unpenalized regression on all p predictors.
- Lasso: If you apply Lasso with p predictors but end up with p_0 predictors, it turns out that p_0 is an unbiased estimate for the degrees of freedom of Lasso! This is magic! We are considering many more than p_0 possible models, which would make you think it is bigger than p_0. But, we are also ending up with a p_0-dimensional linear regression model and applying shrinkage to this model, which would make you think it is smaller than p_0. These two factors apparently average out to p_0!
- Stepwise regression or subset regression: If you start with p predictors but end up with p_0 predictors in the final model, your DF should be larger than p_0. Because, even though your final model uses p_0, you also gave yourself a bunch of options for models to try out, which allows for a bit more potential for overfitting compared to just lm on a fixed set of p_0 predictors.

TLDR: Ridge and Lasso are both LESS WIGGLY than linear regression. Next week, we will see some models that are MORE WIGGLY than linear regression, such as splines or GAMS.

5.3 What is classification

If we rush through this material in class, I recommend that you go revisit Chapter 2 of ISL. It introduces the basics of classification at the same time as the basics of regression. After reviewing this, we will move onto Chapter 4, which discusses methods for classification.

We are switching gears! We assume that we still have a setup with n observations, p predictors in X, and one response variable y. But now y is a categorical variable that can take on one of G possible values[3]

We still believe that there is some *true* data generating process that generates y from X, with some noise. However, it may not make sense anymore to write $y = f(X) + \epsilon$. Because, if y is a category, what form does the noise ϵ take on? Instead, we assume that y can take on values $g_1, \ldots, G_g$. And that there is some true distribution where

$$Pr(Y = g_k \mid X) = p(X).$$

In the special case that Y is binary (there are only two classes), then it must be the case that:

$$Y \mid X \sim \text{Bernoulli}(f(X)),$$

and so we just need to model the probabilities θ. This is an easier case, so we will focus on this a lot. In the case where we have multiple categories, we just replace Bernoulli with Multinomial: we are still going to model probabilities of belonging to certain classes.

But in either case, our goal is to make predictions $\hat{Y}$ using the predictors X! And we want our predictions to work well for unseen data. And we do not know the true function $f()$, so we need to estimate it.

It is going to turn out that a lot of things are the same as what we discussed in the regression case!

[3] A lot of people use K for number of classes. I think this is sort of horrible, because we also have KNN and K-fold cross-validation. We need a new letter!

- We are still going to have a bias-variance tradeoff! We want to use models that are medium-wiggy, so that they can capture structure in the data but so that they cannot just memorize the training set.
- This means that we can't just rely on training set error to measure how good a model is! We need test error, or cross validation.
- We can still use KNN! I am sure you can all figure out how it works: we predict $\hat{Y}(x)$ to be the majority class out of the K nearest neighbors in the training set of x. And we still have the curse of dimensionality!

5.3.1 What is our new goal, and what is the ideal model?

Instead of squared error loss, we will use 0/1 loss. We let:

$$L(Y, \hat{f}(X)) = \begin{cases} 0 & \text{if } Y = \hat{Y}(X) \\ 1 & \text{if } Y \neq \hat{Y}(X) \end{cases}.$$

We want to minimize the expected value of this 0/1 loss over possible new datapoints that we might see. Using the law of total expectation twice, if the possible values of Y are $\{g_1, \ldots, g_G\}$, this means minimizing:

$$\begin{aligned} E_{X,Y}\left[L(Y, \hat{Y}(X))\right] &= E_X\left[E_{Y|X}\left[L(Y, \hat{Y}(X)) \mid X\right]\right] \\ &= E_X\left[\sum_{k=1}^{G} L(Y, \hat{Y}(X)) Pr(Y = g_k \mid X)\right] \\ &= E_X\left[\sum_{k=1}^{G} 1\{\hat{Y}(X) \neq g_k\} Pr(Y = g_k \mid X)\right] \end{aligned}$$

Think about the sum over the G categories. This sum will always have one term that is 0, and the rest of the $G-1$ terms will have value $Pr(Y = g_k \mid X)$. To minimize this quantity, we should always let $\hat{Y}(X)$ be the category g_k that maximizes $Pr(Y = g_k \mid X)$. That way, we are zero-ing out the largest of the G terms $Pr(Y = g_k \mid X)$ in our sum, and are thus making the sum as small as possible.

So, for regression, the very best possible predictor $\hat{Y}$ if we knew the data generating mechanism was to let $\hat{Y}(x) = E[Y \mid X = x]$. For classification, the very best possible prediction $\hat{Y}$ if we knew the data generating mechanism is to let

$$\hat{Y}(x) = \underset{k \in 1,\ldots,G}{\arg\max}\, Pr\left(Y = g_k \mid X = x\right).$$

This is called the **Bayes classifier**. The error rate of this classifier is called the **Bayes error rate**. The Bayes error rate is like the irreducible error that we had before for linear regression, where the irreducible error was $Var(\epsilon)$ and it related to variation in Y that could not be explained by X. The Bayes error rate is the same: no matter how good our statistical learning model is, we can never beat the Bayes error rate, and we can never beat the model that knows $Pr\left(Y = g_k \mid X = x\right)$ and always predicts the Y that maximizes this probability.

Instead, we will think about methods that try to approximate the Bayes classifier using various techniques!

- Warm up 1: Think about KNN for classification! How does it directly approximate the Bayes classifier? When will it do well and when will it do poorly?
- Warm up 2: What do you remember about logistic regression, and how does it relate to the Bayes classifier?

5.3.2 Review of logistic regression

Assume for now that there are only two classes! So Y can take on values 0 or 1.

The notion of Bayes classifier tells us that we should try to model $Pr(Y = 1 \mid X = x)$. Since we are doing binary classification, this is all we need: we will predict $\hat{Y} = 1$ if our estimated probability is greater than 0.5, and we will predict $\hat{Y} = 0$ otherwise.

If we would like to fit a parametric model for $Pr(Y = 1 \mid X = x)$, a simple first idea would be to assume that

$$Pr(Y = 1 \mid X = x) = \beta^T X.$$

We could estimate this with linear regression! But the issue is that we will end up getting predicted probabilities outside of 0 and 1.

The solution, which you have all seen before, is to use a *link function*. The link function will let us use the basic idea of a linear model, but constrain our predictions to be between 0 and 1 so that they can be interpreted as probabilities.

This is logistic regression: confusingly named, because it is a classification method, but it also has deep ties to linear regression and other generalized linear models. We assume that:

$$Pr(Y = 1 \mid X = x) = \frac{e^{\beta^T X}}{1 + e^{\beta^T x}}.$$

To solve this, we find the vector $\hat{\beta}$ that minimizes the negative log likelihood of the training data.

This model ends up being pretty interpretable (recall from Stat 346 that we can interpret our coefficients on a log-odds scale). Furthermore, if our model assumption holds (the log-odds truly are linear in the Xs), then theorems from Stat 360 tell us that the maximum likelihood estimator is asymptotically unbiased and achieves the asymptotically smallest possible variance among all unbiased estimators. This is just like our BLUE fact for linear models. Basically, logistic regression should perform well if our assumptions are met.

However, if we have a ton of predictors p, then our variance can be quite high, and we might actually do better with a biased estimate. Since this optimization problem fits the general framework of looking for a $\hat{\beta}$ that minimizes a training dataset *loss function*, we can reduce our variance with by adding a ridge or lasso penalty to our loss function! And everything will work like it did for linear models! Much of what we have been learning is broader than a single algorithm!

I don't have too much else to say for today! I am sure we will return to logistic regression next class, and I am also sure that you have seen it before! Please read textbook chapter 4 for more information.

Next class we will talk about ROC curves and asymmetric loss functions. AKA why you might decide NOT to predict $\hat{Y} = 1$ whenever your predicted probability exceeds 0.5, but instead might pick a different threshold.