Stat 442 Lecture Notes

Spring Semester, 2025

Last Updated: February 13, 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 the 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.
4. Two methods of preprocessing:
    - Feature selection.
    - Feature extraction.

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

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

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

[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.

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 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.

3.4 Avoiding the issues of high-dimensional data through preprocessing

The comparison above assumes that we are going to directly fit KNN or linear regression using all X original predictors. This is not always what we do in practice! We can actually improve our performance of either method a lot through preprocessing. There are two main types of preprocessing: feature selection and feature extraction.

3.4.1 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.

Let's first talk about feature selection for linear regression. This is easy because you have all done it before!

In linear regression, we have some methods to do this in an interactive way! You all did this in Stat 202 or 346. Look at p-values, remove variables that don't seem significant, etc. More formally, to help with variance, we want to let:

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

where

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

We are just saying that we want to fit a linear regression with a *sparse* solution. Because sparse solutions are interpretable, and also because we know they will have lower variance!

Can we actually solve (7)?

If p is small, we could just try out out fitting least squares models to all of the different subsets $S \subset \{1, \ldots, p\}$. We could then directly compare the value of (7) 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!

Because (7) will essentially be infeasible to solve exactly, we can come up with ways to approximate it. Today we will talk about using forward stepwise regression to approximate it; next week we will talk about regularization methods.

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 do this on your homework.

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.

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.

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 might need to tune a hyper-parameter such as "how many variables to save after stepwise regression". 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; we can see this in an R demo.
- **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 I suppose.

3.4.2 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.

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 $\mathbf{X}$.
  - An issue is that we still need to choose S. Is this now another tuning parameter, in the way that K for KNN is a tuning parameter??
- It turns out that you can randomly project your original 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.

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.

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 explore your entire high dimensional dataset anymore.

3.4.3 Feature selection vs. feature extraction

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.

3.4.4 Sidebar: how do we compute principal components and what are they supposed to do?

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.

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 2-norm) rank-r approximation to X.

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.