Stat 442 Lecture Notes

Spring Semester, 2025

Last Updated: March 12, 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.

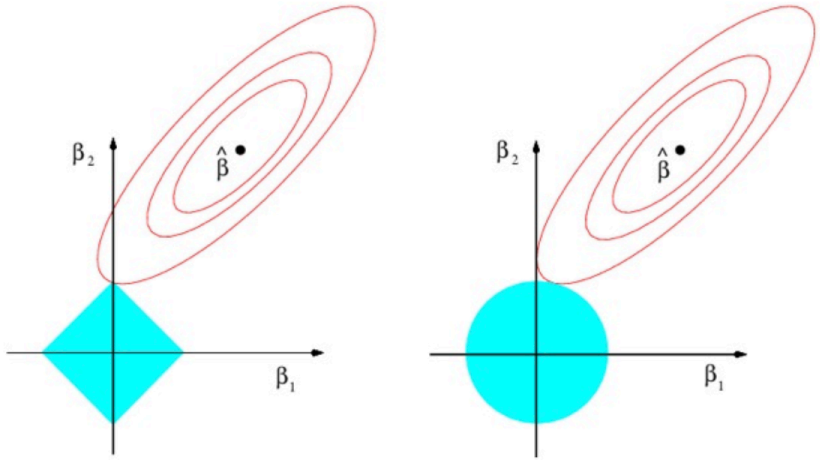


Figure 1: A figure taken from ISL that illustrates why the Lasso prefers sparse solutions while Ridge does not. If we are optimizing the least squares objective (oval) subject to the constraint that either the L1 or L2 norm of β is not too large, the our solution will lie in the place where the contours 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. For more intuition on why, you should see the .Rmd document that I posted on GLOW, which will let you draw this picture for yourself for a particular dataset.

- **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

This was a whirlwind intro to regularization, ridge, and lasso, which are REALLY important topics. We will do some R demos next class. We will also discuss Figure 1 next class: we really have not yet done it justice.

In general, even if Ridge and Lasso feel rushed 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: More on regularization

We did not end up getting to classification at all! Because there is a LOT to say about Ridge and Lasso, so we spent a lot of time on them!

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:**

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

- **Ridge:**

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

- **Lasso:**

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

Ff we had to summarize an overall comparison of pros and cons very quickly, I think we would want to say:

- 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 in the coefficients.

We visualized the bias and variance of Ridge and Lasso as a function of λ in an R demo that is posted on GLOW. We compared to linear model and stepwise regression. We went through the document kind of fast, but there is a lot of good stuff in the document: please read and make sure you understand!

There are a few more things that we need to discuss.

- What the heck is going on in Figure 1? More specifically, what is the connection between regularization and constrained optimization?
- Can we say precisely whether best subset or ridge or lasso is more or less wiggly?

These are covered below.

5.2 Connection between regularization and constrained optimization

To understand Figure 1, it is important to know (14), (12), and (13) can all be written as constrained optimization problems. This is discussed on Page 243 of ISL.

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

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

For some s, which depends on λ and also the values of $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$.

This tells us that our original idea of penalized regression is equivalent to a constrained optimization problem: the problem of solving the least squares problem, but subject to the constraint that $||b||_1 < s$ for some budget s. This formulation is useful because there are smart people who know a lot about constrained optimization!

In class, people asked really good questions about WHY we can write a loss+penalty problem as a constrained optimization problem, and also about the connection between s and λ. This has to do with the theory of the Lagrangian and Lagrange multipliers for constrained optimization! I put a document on GLOW with a lot of notes about this. Please take a look!

How does this idea relate to Figure 1?

- Figure 1 draws the contours of the least squares objective function for a given dataset. The minimum of the least squares objective function is the OLS solution, which is marked as $\hat{\beta}$ in the figure. The contours are drawn as a function of b_1 and b_2, which we are optimizing over.
- Since we can see ridge and lasso as solving this objective subject to a budget constraint, the diamond and the circle draw the Lasso and Ridge budget constraints for a particular s.

- The theory of Lagrange multipliers and constrained optimization tells us that our constrained solution occurs where the contours of the OLS function are tangent to the constraint set. This I think you should be able to understand.
- It turns out that, because the lasso constraint is "pointy", solutions are more likely to occur on the axis: i.e. at sparse solutions where one element of β is exactly 0. This I do not think we really have quite enough theory to understand: we would need to study convex optimization and the KKT conditions and such. But I hope the picture gives you a little bit of intuition!

The .Rmd document that I posted on GLOW gives you code to actually make Figure 1 for any dataset that you generate, and plot the Ridge and Lasso solutions. I hope that you will play around with this code until you understand what is being plotted, and believe me that Lasso tends to give sparse solutions!

5.3 Other

Some random thoughts and notes.

5.3.1 Modern comparison!

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!

5.3.2 Using the lasso for variable selection in practice, and inference after lasso

I was not planning to cover this in class, but then people asked a lot of questions related to it!

Suppose that you are using lasso for variable selection.

You can run lasso (with CV to choose λ on your data). This will return a p-dimensional coefficient vector with a lot of 0s.

You could just start interpreting the non-zero regression coefficients in the way you would interpret multiple regression coefficients. But the slightly strange thing is that, not only did lasso set a bunch of coefficients to 0, but it also shrunk the non-zero coefficients! While it turns out that this shrinking is mathematically good for the bias/variance tradeoff, it isn't good for interpretation!

So, a common thing to do is to re-fit your model using `lm()` (unpenalized regression) after running lasso, but using only the variables that lasso selected.

This is nice! Under the assumption that lasso selected the "true" set of non-zero variables, these coefficients are unbiased for the "true" coefficients and you can interpret them and do inference. This is a very strong assumption. Among variables that lasso selected that turn out to not truly be important, if you trust the `lm()` p-values you will get a hugely inflated Type 1 error rate. This is what I mean when I say that "inference after lasso is invalid".

This is my passion and my research area so if you want to know more about this come ask me! The stepwise regression problem on HW3 also gets at this. But mainly I did not spend much class time on this because inference isn't necessarily the focus of this class.

Also, it's good to know about this "re-fit the model without a penalty" idea, because it seems that they use this technique in the "modern comparison" paper above when they do the "relaxed lasso", which they find works really well!

5.3.3 ElasticNet

We have a method that combines the strength and weaknesses of Ridge and Lasso.

$$\hat{\beta}_{elasticnet,\lambda,\alpha} = \underset{b\in\mathbb{R}^p}{\arg\min} \sum_{i=1}^{n}(y_i - b^T x_i)^2 + \lambda\left(\alpha||b||_1 + \frac{(1-\alpha)}{2}||b||_2^2\right) \tag{14}$$

You could go explore and see if you think this actually works better than Lasso or Ridge individually! And explore its properties! You can also see the shape of the constraint function in ESL Page 73. This section of ESL also talks more about the connection between regularization and Bayesian regression with different priors. How cool and neat!!

6 Monday, 2/24: Introduction to 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.

6.1 What is 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 but known distribution for $Pr(Y = g_k \mid X)$ that we want to model.

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 $f(X)$. 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! For example, the following aspects will all still be important when we evaluate models:

- Bias, which will improve when models are more complex.
- Variance, which will get worse when models are more complex.
- Interpretability, which often gets worse when models are more complex,
- Computational efficiency
- How easy would it be for a non-expert to use it?

6.2 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

[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!

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! Today we will cover three: KNN, logistic regression, and LDA.

6.3 KNN classification

For a test point x^{test}, we find the k nearest neighbors in the training set. We let our prediction for x^{test} be the majority class of these neighbors. The majority class among the K-nearest neighbors is a direct estimate of $\arg\max_{k \in 1,\ldots,G} Pr\left(Y = g_k \mid X = x^{\text{test}}\right)$, if we assume that $Pr\left(Y = g_k \mid X = x^{\text{test}}\right)$ is well-approximated by $Pr\left(Y = g_k \mid X \text{ is a neighbor of } x^{\text{test}}\right)$.

This makes very few assumptions on the function form of $Pr\left(Y = g_k \mid X = x\right)$: it only assumes that neighbors are similar. But we will have a curse of dimensionality problem again. And a bias/variance tradeoff as we change k. The story is basically the same as it was for regression!

6.4 Logistic regression

Suppose for now that there are only two classes! So Y can take on values 0 or 1. The notion of the 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! We still use cv.glmnet! We can make our model less wiggly with regularization.
- It is also still true that we could have bias if our log odds are not truly linear. So we could add polynomial terms or something to make our model more wiggly.

6.5 LDA: discussion lead by Alessa!

The Bayes classifier tells us that we should model $Pr(Y = g_k \mid X)$. Logistic regression tried to model this directly. But, what if we first apply Bayes rule to note that:

$$Pr(Y = g_k \mid X) \propto Pr(X \mid Y = g_k) pr(Y = g_k).$$

This is an interesting idea! If we imagine that our predictors X are *generated* from one distribution when $Y = 1$ and another distribution when $Y = 0$, then using this formulation to model the distribution of X separately for the two classes kind of makes sense!

LDA, or linear discriminant analysis, decides that we should model $Pr(X \mid Y = g_k) pr(Y = g_k)$ directly. It does this under the assumption that

$$Pr(X \mid Y = g_k) \sim N_p(\mu_k, \Sigma),$$

where the mean vectors μ_k for each class and the common covariance matrix Σ are unknown and must be estimated from the training data. LDA estimates these parameters, and then assigns $\hat{Y}(X)$ to be the class that maximizes $\widehat{Pr}(X \mid Y = g_k)\widehat{Pr}(Y = g_k)$.

Why would a method like this be called "linear" discriminant analysis?

Well, under the assumption of multivariate normality, the *decision boundary* between predicting $\hat{Y}(X) = g_1$ and $\hat{Y}(X) = g_2$ is the set of points where $\widehat{Pr}(X \mid Y = g_1)\widehat{Pr}(Y = g_1) = \widehat{Pr}(X \mid Y = g_2)\widehat{Pr}(Y = g_2)$. Let's solve for this decision boundary under this normality assumption.

Let $\hat{\mu}_1$ be our estimated mean for class g_1. Let $\hat{miu}_2$ be our estimated mean for class g_2. Let $\hat{\Sigma}$ be our shared estimated covariance. Let $\hat{\pi}_1$ and $\hat{\pi}_2$ denote the estimated probabilities of falling in class 1 or class 2. These do not depend on X.

Let's solve for x such that:

$$(2\pi)^{-k/2}|\hat{\Sigma}|^{-1/2} \exp\left(-\frac{1}{2}(x-\hat{\mu}_1)^T\hat{\Sigma}^{-1}(x-\hat{\mu}_1)\right)\hat{\pi}_1 = (2\pi)^{-k/2}|\hat{\Sigma}|^{-1/2} \exp\left(-\frac{1}{2}(x-\hat{\mu}_2)^T\hat{\Sigma}^{-1}(x-\hat{\mu}_2)\right)\hat{\pi}_2$$

This means solving for x such that

$$-\frac{1}{2}(x-\hat{\mu}_1)^T\hat{\Sigma}^{-1}(x-\hat{\mu}_1) + log(\hat{\pi}_1) = -\frac{1}{2}(x-\hat{\mu}_2)^T\hat{\Sigma}^{-1}(x-\hat{\mu}_2) + log(\hat{\pi}_2).$$

Simplifying further,

$$-x^T\hat{\Sigma}^{-1}(\hat{\mu}_1 + \hat{\mu}_2) + \frac{1}{2}\hat{\mu}_1^T\hat{\Sigma}^{-1}\hat{\mu}_1 + \log(\frac{\hat{\pi}_1}{\hat{\pi}_2}) = \frac{1}{2}\hat{\mu}_2^T\hat{\Sigma}^{-1}\hat{\mu}_2.$$

Note that this is a linear function of the vector x: the quadratic terms cancelled out. This tells is that the boundary between the regions where we predict class 1 and the region where we predict class 2 is a straight line or a plane in X-space. So, LDA tries to draw straight lines or planes in X-space to separate classes. It turns out that this will work well if the classes are truly linearly separable, and will not work well otherwise.

I am sure that Alessa will tell us more about the pros and cons of LDA and how it compares to other methods. These are just initial notes. I bet you can already start making some guesses about bias, variance, interpretability, etc.

On your own time, convince yourself that if you drop the assumption that Σ is shared across classes, you find that the decision boundary is quadratic! This is QDA instead of LDA!

6.6 Asymmetric Loss and ROC Curves

The premise of today's class was that we want to minimize the expected 0/1 loss. But you all know that, in real life, sometimes false positives are more severe than false negatives, or vis versa.

In such a setting, we might not want to pick the classifier that minimizes the expected 0/1 loss. We might want to pick the classifier that minimizes the false positive rate while maintaining a reasonable false negative rate, or something like that!

This leads us to the notion of an ROC curve, which you will see on your homework 3!

7 Thursday, Feb 27: making models more complex

7.1 Where are we?

We have been talking about ways to predict Y from predictors $X_1, \ldots, X_p$. We have now covered cases where Y is numerical and cases where Y is categorical: and it turns out that many of the common themes hold in both cases.

In basically all cases, we are trying to predict Y using X. We think there is some true function $f()$ that would help us make predictions. But, since we don't know $f()$, we will use our training data to come up with an approximation $\hat{f}(X)$. We usually pick the $\hat{f}(X)$ that does the "best job" on our training data (i.e. $\hat{f}(X) \approx y$), but then we use a test set or cross-validation to make sure that we are not *overfitting*.

It is up to us to decide how complex we want to allow $\hat{f}(X)$ to be. Do we want to restrict ourselves to something simple like a linear model? Or do we want to allow $\hat{f}(X)$ to be very wiggly? This choice always involves a tradeoff between bias and variance.

Last week, we talked about a setting where p is large, and where even a simple linear model has variance that is too high. Thus, we talked about feature selection, dimension reduction, and regularization. These are all ways to take the linear model:

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \ldots + \hat{\beta}_p X_p$$

and make this linear model LESS wiggly, so as to reduce the variance.

Today, we are going to talk about ways to take this simple linear model and make it MORE wiggly, so as to reduce the bias. For simplicity, to start, we will return to the case where $p = 1$. The model:

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1$$

is SO simple. We might be able to see just from looking at a plot that it is not expressive enough to model our data. What do we do about this? This is the topic of Divij's talk. But first, let's spend 5 minutes on Degrees of Freedom, since we skipped it a few classes ago.

7.2 Degrees of Freedom and Effective parameters!

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.

Take a few moments to let this definition sink in. What is it telling us? What is the degrees of freedom of a procedure that memorizes the training set (and therefore gets 0 training error)?

$$df(\text{memorizer}) = \frac{1}{\sigma^2} \sum_{i=1}^{n} Cov(Y_i, Y_i) = \frac{1}{\sigma^2} \sum_{i=1}^{n} \sigma^2 = \frac{1}{\sigma^2} n\sigma^2 = n.$$

This tells us that the degrees of freedom of 1NN, for example, which we know to be extremely wiggly, is the size of the training set n!

Calculating degrees of freedom is 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.

The concept of degrees of freedom is very related to the concept of "optimism". These concepts are covered in Chapter 7 of **ESL** (not ISL). I have been sprinkling these concepts in throughout the course, but next week I will be sure to

put a question specifically about model complexity on the HW!

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 with p predictors. They have fewer than p "effective parameters" because of the regularization.

Note that, in this group, KNN is the only model whose degrees of freedom grows with the size of the training set n. This is in fact what it means for KNN to be *nonparametric.*

Returning to our example for today, suppose that we only have one predictor X and we are going to fit the model:

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1.$$

This is sort of a bummer because it only has two degrees of freedom! This might not be enough! We all already know that we could increase the complexity (df) of this model by instead fitting something like

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_1^2 + \hat{\beta}_3 X_1^3.$$

Today we will discuss even better ways to do this!

7.3 Splines

There are actually two distinct topics here: regression splines and smoothing splines. At first glance, they do not seem that similar! But a beautiful theorem tells us why they are related, and therefore why they both get the name "spline".

7.3.1 Regression splines

These start from a pretty straightforward motivation.

We already know about polynomial regression, We can fit the model

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_1^2 + \hat{\beta}_3 X_1^3$$

using "lm()" in R, by just passing in our squared terms and cubic terms as new predictors.

But we don't need to limit ourselves to polynomials. Many of you have likely also used models of the form

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 1(X_1 < c),$$

which produces a step function. This is what we do when we decide that maybe our model is better if we convert a numerical predictor variable X_1 to a categorical predictor variable. Once again, we can just fit this model with "lm".

More generally, we can use any set of fixed/known basis function to turn X_1 into an arbitrarily wiggly and complicated function. As long as we use less than n basis functions, we can still fit the model with 'lm() in R.

People have proposed regression splines as a really nice way to make basis functions. These are piecewise polynomials. The idea is that, to make the model $\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_1^2 + \hat{\beta}_3 X_1^3$ more complex, we could add a term of degree 4

or degree 5. But high degree polynomials tend to have really crazy behavior. What if, instead, we limited ourselves to cubic, but we added a "knot", and we let the coefficients be different on either side of the knot. This could be nice, but this could also lead to discontinuous functions which we don't want! The spline basis representation gives a way to make continuous piecewise-constant polynomials with smooth derivatives. A *natural spline* is additionally linear outside of the range of the training Xs, which keeps predictions from blowing up to crazy values if we extrapolate beyond our training data.

I don't really care if you know the formulas for a spline basis representation: I just care that you understand how the concept a spline fits into our framework of the bias/variance tradeoff. We could decide how many knots to give our spline using cross validation! I also care that you understand that fitting a spline just means using "lm()" on a new set of features. You will do this on your homework!

7.3.2 Smoothing splines

These start from a totally different, non-parametric motivation. Still assume that we only have one predictor.

Suppose that we wanted to find the function $g()$ that minimizes

$$\sum_{i=1}^{n}(y_i - g(x_i))^2$$

on the training set, but we did not want to specify a particular form for $g()$. For example, we don't want to choose in advance to make it linear, or a cubic spline, or a sin curve, etc. What we do know is that a function that is too wiggly will have high variance and will not generalize well. So, we decide that we will add a penalty term for "non-smooth" functions $g()$.

This sounds like regularization! Indeed, we will try to minimize:

$$\sum_{i=1}^{n}(y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt.$$

Integrating over the second derivative just means summarizing how much $g'(t)$ changes over the entire range of t. If we fit a linear model, then $g''(t)$ is 0, and so there is no penalty on a linear model. The parameter λ is a tuning parameter that tells us how much we should penalize non-linear or wiggly functions.

The function $g()$ that minimizes this equation is called a smoothing spline. Why is it called a spline? See the beautiful theorem.

7.3.3 Beautiful theorem

It turns out that, out of all functions $g()$, the one that minimizes

$$\sum_{i=1}^{n}(y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt.$$

is actually a natural cubic spline with knots at every training data point $x_1, \ldots, x_n$. Woah!!!

Does this mean that, to get an optimal smoothing spline, we can just use "lm()" in R and regress y on the natural spline basis function with $x_1, \ldots, x_n$ knots?

- No
- The first reason we cannot do this is that the natural spline basis function with $x_1, \ldots, x_n$ knots has more than n terms. This means that we would be fitting a linear model with n datapoints onto more than n predictors. R cannot solve this for us, since $(X^T X)$ cannot be inverted.
- Also, if we COULD solve the least squares solution for regressing y on the natural spline basis function with $x_1, \ldots, x_n$ knots, it would get 0 training error and would memorize the training set. This is not actually what we want!
- The penalty term λ ends up enforcing that we also shrink all of our coefficients towards 0: as we would with ridge regression.
- So, we fit this using a different R package. I think it might be the default in "geom_smooth()"! But it is really cool to know about the connection with a natural cubic spline.
- QUESTION: from the penalty perspective, why does it make perfect sense that $g()$ is a NATURAL cubic spline, meaning that it is linear outside of the range of the training points?

7.4 GAMs

7.4.1 What is a GAM?

Let's build on the spline idea, but now let's suppose that we actually have more than one predictor. We have $X_1, \ldots, X_p$, and we want to let the function $\hat{f}(X)$ be wiggly in all of the predictors (if needed).

We could try to do a spline basis expansion for every single variable. But this would start to get kind of crazy! We don't necessarily have a larger n than we did before. If we add 12 coefficients for every predictor variable, we are going to have the number of coefficients exceed the number of variables pretty quickly: and then we can no longer fit with lm!

Let's restrict our attention to models where we don't allow the predictors to interact with each-other. This is limiting in some ways. But, if we want to start letting variables interact, the complexity of our problem blows up even more quickly.

So, lets model:

$$Y = \beta_0 + \beta_1 f_1(X_1) + \beta_2 f_2(X_2) + \beta_3 f_3(X_3) + \ldots.$$

Each $f_j()$ could be any basis function representation of X_j. If could simply be a linear model if we think that Y is linear in X_j, but it could also be a full on smoothing spline for a different predictor. Note that, because f_j can be whatever we want, it can also absorb the coefficient β_j. So we will actually just write this as:

$$Y = \beta_0 + f_1(X_1) + f_2(X_2) + f_3(X_3) + \ldots.$$

A nice thing about the additivity is that I can visualize one variable at a time, and decide how wiggly I think Y needs to be in that particular variable. This does not take into account the idea of "holding all other variables constant", but can be a nice first approximation.

GAMS are really powerful. Suppose that we really want to be able to *interpret* the effect of X_1 on Y, but we want to make sure we have already sucked up all possible variation that could be explained by other components $X_2, \ldots, X_p$. We could let $f_j()$ be a smoothing spline for all $j > 1$, but force $f_1()$ to be a linear function. This would let us interpret the linear effect of X_1 on the part of Y that would still be noise even if we fit a many-dimensional geom_smooth() to all other predictors. That is cool!

7.4.2 How do we fit a GAM?

Unless all components f_j are pre-specified with a parametric basis function, we cannot fit a GAM using least squares.

Instead, we take advantage of the additive structure to do an iterative back-fitting procedure. The function GAM in R implements this for you, and by default it uses a natural spline for each f_j.

To learn about the back-fitting algorithm, see page 297 of ESL. The idea is that, since everything is additive, we can iteratively fit a model that predicts $Y - \hat{\beta}_0 - \sum_{j \neq j^*} \hat{f}_j(X_j)$ using X_{j^*}. This will let us see the function that "right now" best predicts Y using X_{j^*}, after taking into account all variation currently explained by other predictors. We start with random guesses for all of the $\hat{f}_j$ terms, and we iterate through $j = 1, \ldots, p$ many times until convergence.

What I want you to know about back-fitting is: the fact that we can do one variable at a time makes this relatively easy! With neural networks next week, we will not be able to go one variable at a time, because we will have interactions! This will necessitate a more advanced algorithm.

7.5 What do we think about splines and GAMs?

- Bias: they have less bias than linear regression if the true data generating mechanism is not linear.
- Variance: they can have high variance if we let them be too wiggly.
- Interpretability: they are not black boxes. they are parametric, and we can examine them to figure out which predictors impact our predictions. but they can get kind of complex and kind of hard to interpret! Especially without drawing a graph.
- Computational efficiency: regression splines can be fit very efficiently. GAMs require an iterative algorithm, but the fact that they are additive still makes this algorithm reasonably efficient.
- Anything else I am missing?

You could add a spline basis function to a logistic regression. You can also definitely use GAMs for classification. So ... none of this was specific to regression! The concepts of wiggly and less wiggly apply to classification too!

The main idea of class next Monday will be: what happens when we drop the additivity requirement?! It is going to turn out that, when we let models be non-linear AND we allow for interactions, we get crazy complex models! Such

as neural networks!

8 Monday, March 3: Introduction to a simple neural network

Main references: ISL 10.1, 10.2, 10.6, 10.7, and 10.8. I also think that ESL Chapter 11 has good context, but some of it is beyond the scope of this class.

8.1 A statistical motivation

8.1.1 GAMs, but with feature engineering?

Last week, we discussed GAMs. These are functions where we let

$$\hat{Y} = \hat{\beta}_0 + \hat{f}_1(X_1) + \hat{f}_2(X_2) + \ldots + \hat{f}_p(X_p).$$

We let the functions $\hat{f}_j()$ be extremely non-linear functions, such as smoothing splines. Even if we let $\hat{f}_j()$ be completely non-parametric and wiggly, fitting a GAM is not too bad. The additive structure means that we can iterate through one variable at a time, and just fit a one-dimensional model of $Y - \sum_{j \neq j^*} \hat{f}_j(X)$ on X_{j^*}.

But, the additive structure of the model will always introduce some bias if the true data generating mechanism is not additive! Consider image classification. Why would the image's class be determined by a sum of functions of every individual pixel? It is really the patterns between the pixels that matters. And we don't need the ability to interpret the effect of each individual pixel: this is going to be the type of data that neural networks are really good for!

We know that we could fit a GAM to the principal components of X. We could also manually add interaction terms. All of that would sort of help our GAM problem. But let's try to do this in an automated way.

The idea for today is that maybe we want to let:

$$\hat{Y} = \beta_0 + \sum_{k=1}^{K} g_k(w^T X).$$

I am getting this particular form of this equation from ESL section 11.1. This is a GAM, but it is a GAM applied to K features, each of which is a linear combination of ALL of the Xs. And we haven't yet determined the weights w that will make up the linear combinations. In projection pursuit regression (ESL, 11.1), the $g_k()$ are smoothers, as in additive models. Today, we will have them be something simpler.

As noted in ESL, project pursuit regression never became widely used. This is because, at the time of its introduction, you would need really powerful computers to actually fit the model. However, the reason that it is exciting for us is that it was a completely statistical motivation for a topic that was simultaneously becoming popular in the field of AI. We will discuss the AI motivation later.

8.1.2 The neural network model

ISL introduces its first neural network model on page 404. The idea is that we will let $\hat{Y} = \hat{f}(X)$, where $\hat{f}(X)$ must come from a class of functions that can be written as:

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right). \end{aligned} \tag{15}$$

The function $g()$ is a non-linear activation function. The idea is that we model Y as a linear combination of K "hidden units" or "activations". K is a parameter that we get to pick. Each of these is a non-linear transformation of a simple linear combination of the X variables. Note that if we chose $g()$ to be a linear function, this would just collapse down into a linear model with no interaction terms.

So ... this is our neural network! We need to use our training data to come up with estimates for $\beta_0, \beta_1, ..., \beta_K$ and $w_{k0}, \ldots, w_{kp}$ for $k = 1, \ldots, K$. This is a total of $(K+1) + (p+1) * K = (p+2) \times K + 1$ parameters. We want to come up with guesses such that $Y \approx \hat{f}(X)$, hopefully on both out training data but also on unseen test data.

Because of the fact that there are so many parameters in (15), we often draw this model as a picture so as to keep track of everything. The picture is shown in Figure 2. The idea is that every line in the picture corresponds to a parameter that we learn. Once we have drawn this as a picture, we can call the setup of our model the "architecture". The architecture is up to us! If we wanted to have hidden node A_1 not connect to X_1, that would be totally fine: we would just have one less arrow in our picture and one less parameter to estimate.

There is one note in Figure 2 that is not reflected in (15). In (15), I was imagining a regression problem: so once we take our linear combination of linear combinations, we have $\hat{Y} = \hat{f}(X)$. If we are doing classification, we might want a final step where we convert the output of our network into a probability (using a sigmoid or softmax function), and then maybe convert it to a discrete prediction. Multi-class classification is seamless: we just have one output node per class, and we later convert to probabilities with a softmax function.

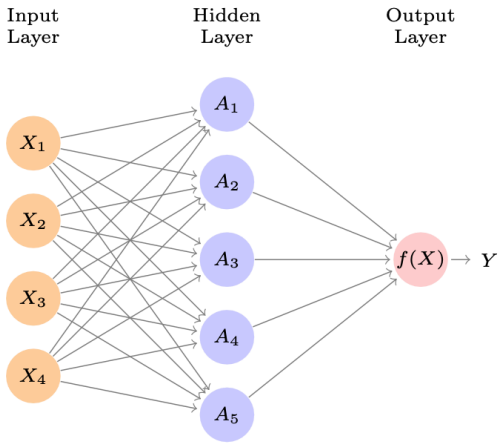


Figure 2: A visualization of (15). The arrows between the inputs X and the hidden later each represent a parameter w_{kj}, where the k tells us which hidden layer we connect to and the j tells us which input variable to connect to. Sometimes we draw an extra input layer node to represent the intercept. The arrows from the hidden layer to the prediction layer are the parameters β_k. All of these parameters must get estimated.

While we won't be able to fit models like the one shown in Figure 2 directly with least squares, we will be able to fit it by optimizing over a loss function: something that you all have already seen many times! So ... a neural network really isn'y all that different from a linear model! This is a very statistical perspective on neural networks.

8.1.3 Deep learning

The really key idea in Figure 2 is that: **linear combinations of non-linear transformations of linear combinations** can be arbitrarily complicated and wiggly! In fact, there is a theorem that says that any continuous function $f()$ can be expressed in the form (15) as long as $g()$ is non-polynomial and K is big enough. Another idea, if we don't want to make K huge, is to add more layers! And more levels of non-linearity! This is shown in Figure 3, which is also taken from ISL.

The idea of approximating really complicated functions with these networks has really taken off in the field of deep learning! Models like the one in Figure 3 have become very state of the art in image classification, computer vision, LLMs, and basically any other big-tech application that you can possibly think of. Whatever big data you see out there in the real world- it is probably making use of deep neural networks! It almost makes me feel like there is no need to teach the other methods that we have been discussing or will be discussing in this class. But only almost! Deep neural networks are not appropriate for every circumstance, and they certainly have a lot of downsides!

8.2 Historical context and a less statistical motivation

When neural networks were first developed in the 1940s, there was a biological interpretation. Scientists wanted to model the living brain! An early reference is McCulloch and Pitts (1943).

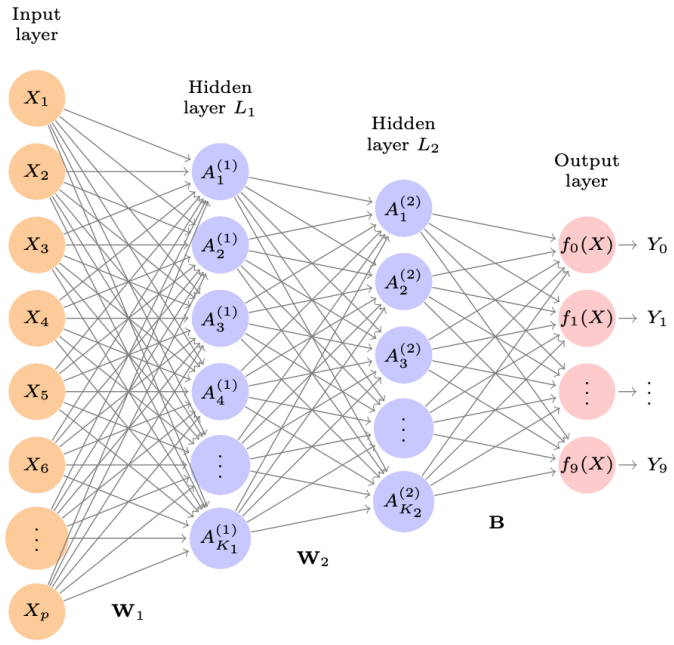


FIGURE 10.4. *Neural network diagram with two hidden layers and multiple outputs, suitable for the* `MNIST` *handwritten-digit problem. The input layer has* $p = 784$ *units, the two hidden layers* $K_1 = 256$ *and* $K_2 = 128$ *units respectively, and the output layer* 10 *units. Along with intercepts (referred to as* biases *in the deep-learning community) this network has 235,146 parameters (referred to as weights).*

Figure 3: A multilayer neural net. Note that here we have multiple outputs Y because this is for a categorical output with 10 categories. We represent this as a vector of length 10, where $Y_k = 1$ if $Y = k$ and $Y_k = 0$ otherwise.

The nodes in the hidden layer are supposed to represent neurons. A neuron only fires if the amount of signal being passed into it exceeds a certain threshold. So, the activation function $g()$ was typically a step function: either the node (neuron) is turned on, or turned off. However, the step function was later replaced with differentiable alternatives for the purposes of actually fitting the model. But, the idea that the activation function should often be near 0 until it eventually gets "activated" or "fires" remained.

Do you think that neural networks work well because they model the human brain (which works well), or do you think they work well because they are very expressive linear models that do automatic feature engineering? It is up to you to decide on your interpretation! I think the latter, but I also do not know much about the real human brain!

A few more historical notes:

- The ideas of neural networks are so old and were developed independently by different fields!
- But ... when did they really take off?
- Up until 2010 or so, they were not popular in computer vision. Only Yann LeCun and Geoff Hinton were really pursuing these. After some students of Geoff Hinton won an ImageNet competition (used a neural network for an image classification task and got the best performance), then people started to pay attention.
- By the way, my history knowledge was really rusty and anecdotal. I found this slide deck from a course at Univ. Wisconsin `https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L02_dl-history_slides.pdf` and I liked the way the material was presented: he has screenshots from a lot of cool books. Check these out and check out the related sources if you want to know more!
- Things I learned: the ReLU activation function was an idea of Hinton+coauthor, and it was in 2010. And this change in activation function must have been a huge difference maker!
- You can also check out Section 1.2 of the free deep learning book: `https://www.deeplearningbook.org`.

8.3 What do I pick for my activation function $g()$?

- In projection pursuit regression, the idea was that this could be a smoothing spline or something! But projection pursuit regression was always going to have like $K = 5$ and one hidden layer. If we want to do BIG neural networks, we need something simple. And hopefully differentiable, because we will fit with gradient descent.
- From the "neurons in the brain" view of neural networks, a 0/1 step function made sense. A neuron is either turned on or turned off. But, since a step function is not differentiable, the sigmoid/logistic: $g(z) = \frac{1}{1+e^{-z}}$ was a nice alternative. Or tanh.
- But, nowadays, a really common one is: $g(z) = max(0, z)$. This is called the ReLU. This keeps the motivation of a neuron being "turned on" vs. "turned off," but now allows really strong signals to be carried forward by having larger magnitude. One issue with this activation function is the "dead ReLU" problem, which we will discuss.

8.4 Themes

People have the tendency to treat neural networks and deep learning as magical! I hope to convince you that these are not magical, and that they in fact relate to many things that you have already seen in this class: e.g. feature engineering, basis expansion. And I hope you can start to see how neural networks relate a bit to what I talked about on the very first day of class: the biggest difference between modern machine learning and classical statistics has to do with "how much is pre-specified". Neural network relates to feature engineering or basis expansions, but the form of the new features or the basis functions was not pre-specified.

Because NNs are not so different than everything else we can see in this class, we can discuss our favorite themes.

- Bias:
  - We can approximate ANY continuous function f with a neural network with one hidden layer and a non-polynomial activation function. We might need a really huge (but finite) value of K. But still! This is so cool. It means that we are not limited by modeling assumptions for a neural network: we should be able to model any true f really well.
  - One note: this theorem says that any continuous function f can be expressed using a picture like Figure 2 and certain values of the weights. It does not say that we will be able to estimate the weights from our training data!
  - But still. Neural networks have really low bias! No bias if we use enough layers or nodes.
  - They work really well!
- Variance
  - We know that variance depends on degrees of freedom, which is the effective number of parameters that we are estimating. If we make a lot of hidden layers, or if we make K really big, then surely we will have more parameters than we do training observations n. Isn't this really bad for variance? Won't we memorize our training set and overfit?
  - Short answer: statisticians were very skeptical of deep learning for these reasons for many years. Over-parameterized models should score poorly for variance! And they definitely do when you don't have a really big value of n.
  - We should regularize to help keep the variance under control. More on this when we discuss double descent! We should use a validation set to stop training before we overfit.
  - But sometimes, when you have enough data, NNs perform really well on test sets, even though they are over-parameterized. This also isn't magic: we need to distinguish between number of parameters and number of effective parameters
- Interpretability
  - This is our first true black-box model. It is really hard to understand where the predictions of a neural network model are coming from!
  - Do not pick deep learning if you have a scientific application where you need to interpret and explain your results!
  - Since the model itself is a black-box, we will need to use clever explainability techniques on top of the model to try to understand what is going on: this is a hot area of research that we will discuss after spring break.
- Usability (is the method "off the shelf")?
  - Neural networks tend to require a lot more tuning than something like a random forest. They are NOT very "off-the-shelf". This is a reason why they took a while to become popular. They were hiding in the background for many years before deep learning really took off.

  - Evidence: I will not make you do a HW problem where you actually use neural nets, because the R packages are finicky!
  - There is flexibility, which is nice: you can modify the architecture really to your liking. If you know what you are doing, this is great! But this flexibility does make it harder to make all-purpose, useable software.
- Computational efficiency
  - We are going to actually go over gradient descent and backpropogation on Thursday! So you will learn more about fitting.
  - We need to use a slow iterative algorithm to fit: and we always need to be worried that maybe we did not converge in our fitting!
  - However, there is also something very parallel and distributed about our computations, which makes huge deep neural networks actually feasible to fit.

9 Thursday, March 6: Fitting a neural network: backpropagation and gradient descent

Recall from Monday that our neural network model with one hidden later is a model that makes predictions of the form:

$$\hat{y} = f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_j\right). \tag{16}$$

Because this is a lot of parameters to keep track of, we sometimes draw the model as a picture. But ultimately, given our neural network architecture (our picture, which should tell us K and should tell us if the network is "fully connected"), we need to use our training data to FIT the network.

This just means coming up with values $\hat{\theta} = (\hat{w}_{01}, \ldots, \hat{w}_{Kp}, \hat{\beta}_0, \ldots, \hat{\beta}_K)$ that minimize the error over our training set:

$$\sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2.$$

Of course eventually we will also care about overfitting and test error. But to start, let's just figure out how we might even minimize the loss when $\hat{Y}_i$ has the form in (16). It is not going to be simple! We are going to need to use an optimization technique called gradient descent. The key insight for neural networks is that we can use the chain rule to do gradient descent in a very organized way, called backpropagation.

9.1 Gradient descent

Let θ be a long vector that stores all of our parameters in our network. So our goal is to minimize, with respect to θ,

$$L(\theta) = \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2.$$

Unfortunately, for neural networks, this is too complicated to just take the derivative with respect to every single parameter and set them equal to 0. This would be a massive system of equations.

So instead of solving for a minimum directly, let's try to do it iteratively.

Let's start with θ^0: a set of random guesses for every parameter in the network. These values will produce estimates $|hatY_i$ for every element in our training set: they probably will just not be very good estimates!

Then, let's take steps along the negative gradient (with respect to θ) of the loss function. The gradient of $L(\theta)$ with respect to θ is denoted $\nabla L(\cdot)$. The gradient is simply the vector of partial derivatives with respect to each element of θ. Once we have taken this step, we will update our values of all of our parameters, so θ^0 becomes θ^1. This new vector then yields new predictions $\hat{Y}$ for every training point.

The idea is that, if we take steps in the direction of the negative gradient of the loss function, we always move a current guess θ^t to a new guess θ^{t+1} in a way that will make the loss smaller the next time we compute it. What a nice idea!

While we take a step in the "direction" of the gradient, we don't want to take a step that is the SIZE of the gradient.

We want to take really small steps so that we don't "overshoot" a minimum. So we let:

$$\theta^{t+1} = \theta^t - \rho \nabla L(\theta^t),$$

where ρ is a small learning rate that we get to pick. Picking it can actually be really finicky: if it is TOO small you will never find a good value of θ. But if it is too big you can "overshoot" a minimum.

Note: if our step size ρ is going to be approximately the same for all elements of θ, we probably want the elements of θ to all be in the same "units". This means that we probably want to normalize our input variables X! This is not changed from other algorithms.

We keep taking small steps in the direction of the negative gradient until our "guesses" stop changing. At this point, we have landed in a region of θ-space where the derivative is 0! This hopefully means that we found a minimum of the loss function!

Unfortunately, there is no guarantee that it is a global minimum. For really complicated non-convex loss functions, we can get stuck in suboptimal local minima. We will discuss strategies for avoid it later! For now, see Figure 4 for an illustration of gradient descent in a setting where θ is one-dimensional.

Gradient descent is a really general optimization technique! It is used in contexts that have nothing to do with fitting neural networks!

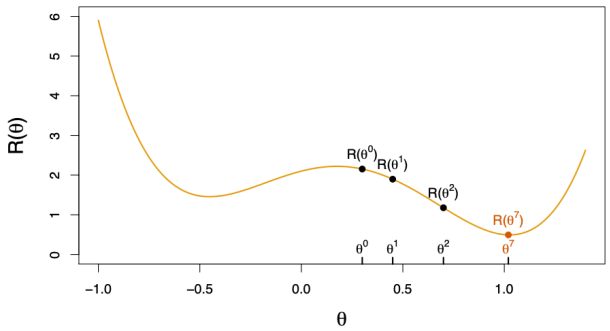
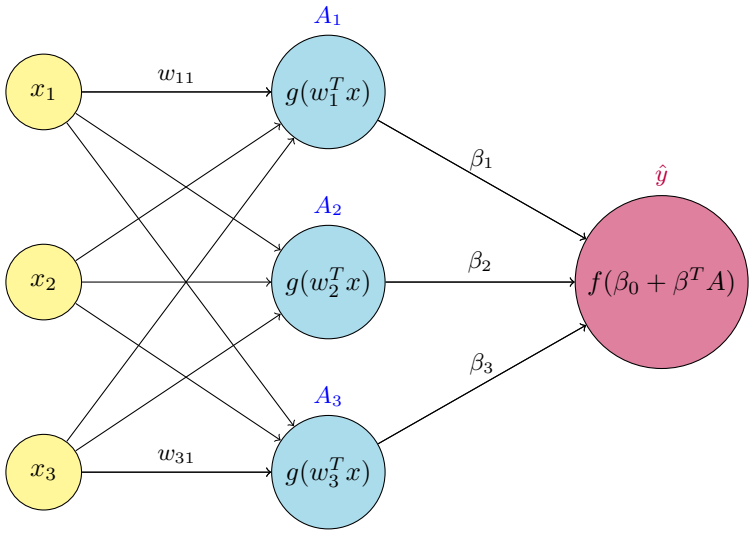


FIGURE 10.17. *Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one at $\theta = -0.46$ (local), the other at $\theta = 1.02$ (global). Starting at some value θ^0 (typically randomly chosen), each step in θ moves downhill — against the gradient — until it cannot go down any further. Here gradient descent reached the global minimum in 7 steps.*

Figure 4: An illustration from ISL of gradient descent for a very simple one-dimensional θ. In this case, we found the global minimum. But that is not guaranteed!

9.2 Backpropagation and the Chain Rule

Let's study gradient descent for a one-layer feed-forward neural network in the following very simple case. This is a network with one hidden later and three hidden nodes ($K = 3$). There are also only three inputs ($p = 3$). For simplicity, I did not draw the intercept (bias) terms only the diagram.



In this case, suppose that we are doing regression, and so the function $f()$ at the very end is just the identity function. Recall that when we are doing classification, we need to turn our final output into a probability or a prediction using a link function. But for regression we are all set on our original scale.

For given values of $w_{10}, \ldots, w_{pK}$ and $\beta_0, \ldots, \beta_K$, each input example x_i gets turned into a $\hat{y}_i$. Our squared error loss for our training set is:

$$\begin{aligned} L(\theta) &= \sum_{i=1}^{n} L_i(\theta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \\ &= \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{k=1}^{3} \beta_k A_k \right)^2 \\ &= \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{k=1}^{3} \beta_k g \left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij} \right) \right)^2 . \end{aligned}$$

I wrote this at multiple levels of granularity. Sometimes it is nice to explicitly see that the loss function depends on every w and every β. Other times, it is kind of nice to hide this detail.

Suppose that we begin with random guesses for every β and every w. Then, we send our entire training set "forward" through the neural network and compute this loss function given the current predictions. Then, we compute the gradient. We step along the gradient, and we update our guesses of w and β accordingly.

For our neural network loss function, let's figure out the steps along the gradient. The key idea is that we can chain rule!

In this example, since $f()$ is just the identity function, we have that:

$$\begin{aligned} \frac{dL}{d\beta_k} &= -2 \sum_{i=1}^{n} (y_i - \hat{y}_i) \frac{d\hat{y}_i}{d\beta_k} \\ &= -2 \sum_{i=1}^{n} (y_i - \hat{y}_i) A_k \\ &= -2 \sum_{i=1}^{n} (y_i - \hat{y}_i) g \left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij} \right) \end{aligned}$$

And then:

$$\begin{aligned}\frac{dL}{dw_{kj}} &= -2\sum_{i=1}^{n}(y_i - \hat{y}_i)\frac{d\hat{y}_i}{dw_{kj}} \\ &= -2\sum_{i=1}^{n}(y_i - \hat{y}_i)\beta_k g'\left(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\right)x_{ij},\end{aligned}$$

where the specific form depends on the activation function $g()$. But hopefully we choose a differentiable $g()$ where this isn't too complicated.

Note that both of these partial derivatives imply that the direction of our step depends on our residuals $y_i - \hat{y}_i$: this is a good thing! We want to make the residuals smaller!

The reason that we call this special version of gradient descent "backpropagation" is that we can think of these applications of the chain rule as stepping backward through our network, and passing our residuals/errors along as we go. As we add more layers to the network, we add more chain rules! But nothing gets too difficult.

Recall that gradient descent lets:

$$\theta^{t+1} = \theta^t - \rho\nabla L(\theta^t).$$

So, the size of the step depends on the learning rate ρ, the size of the residuals $(y_i - \hat{y}_i)$, the magnitude of the gradient evaluated at θ^t, and the actual inputs x_{ij}. This is a lot of things! Making sure that we take the right sized steps can be very finicky.

9.3 Stochastic Gradient Descent

Instead of computing the loss function and the gradient over all n training instances at once, we typically go instance-by-instance or batch-by-batch. This is related to *online learning*, which you read about for HW2. A batch is just a subset of the n training observations. The motivation is that we will speed up our learning about θ by updating θ^t more often. If our training set is huge, we don't want to bother processing a massive training set with a bad guess θ^t. We should take advantage of the fact that we see a lot of errors in the first few training examples, and we should update θ^t to θ^{t+1} right away.

If we are using SGD and we end up iterating through the entire training set 10 times, this means that we used 10 epochs. But, this might mean that we used $T = 100$ updates of θ, if our batch size was 1/10 of the training set. We need to decide how many epochs we want, etc.

This is stochastic because our gradient computed on a small number of training observations should be a good approximation of our overall gradient. But the exact step we take depends on the specific training observation! It also turns out that the extra noise added with SGD can help us avoid local minima! That is cool too! We bounce around θ-space and explore a bit more.

Very strangely: it turns out that stochastic gradient descent enforces its own form of approximately quadratic regularization. I don't think that this is at all obvious. I learned from preparing for this class that the reason is because stochastic gradient descent gives updates for θ that remain in the row space of the inputs. Out of all solutions that would give 0 training error (in a setting where this is possible), stochastic gradient descent is supposed to find the one with the smallest L2 norm. "Supposed to" because I think it still might not due to initial conditions and step size. But anyway, I don't think this was the motivation for people coming up with SGD, but it is a recently studied property, and it explains double descent!! So cool.

9.4 Complications

While (stochastic) gradient descent is a very general idea that can be used in many contexts, it is ultimately only a way to find an approximate minimizer of a loss function. There are a lot of complications.

- We need to choose good initial guesses for the parameters. If we choose bad ones, we can get stuck at local minima.
- We need to choose a good step size! To converge in a reasonable about of time without jumping over minima.
- We have to worry about "vanishing gradients" / the dead ReLU: where once a ReLU unit is outputting a 0 for every training example, its weights never get updated again.
- We need to choose batch size for SGD.
- We need to choose how many iterations or epochs to train for. In theory, I guess we should go until the θ^t stop changing (convergence). But ... what if we have used up all of our computational resources and we have not yet

converged?

- We also need to pick our model architecture! How many hidden nodes do we want? How many hidden layers? What activation function should we use?

Previously, to pick a tuning parameter, you used cross validation. Doing cross validation for ALL of these choices and coming up with the best possible parameter sets is a full time job. And, literally, this is a full time job that you could all go get! TLDR: you basically need to be an expert to fit a neural network. It is not an "off-the-shelf" algorithm.

9.5 Overfitting and model complexity

We know that a neural network model with K hidden units in one layer and p inputs needs to learn around $(p+1)*(K+1)$ parameters. If we add more hidden laters, this just gets bigger.

We know that, if we have n training datapoints, a model with more than n parameters should be able to memorize the training set. And we know that this is bad, because we will overfit, and will have poor generalization error.

In modern deep neural networks, we often have more parameters than datapoints! So, how can we prevent overfitting?

There are several strategies that people were already doing:

- Add explicit regularization to our loss function! Like λ times the L1 or L2 norm of our ws and our βs. This affects every step of our gradient descent calculation, and could make it more complicated. But it will keep the variance of our model down, as we know.
- Random dropout: A possibly simpler version of regularization. We can just randomly turn off some hidden nodes when we pass some training observations through the network. This forces other nodes to pick up the slack of these turned-off nodes. This means that a single node cannot be used to memorize a single training point—- because it has to perform multiple tasks. What a super cool idea!
- Early stopping via a validation set. Suppose that we want to encourage our weights to be small. We can start with small initial values. Then, every time we finish a batch or an epoch, we can check our error on a validation set. Presumably at first, this will be decreasing as we update our weights. If, at some point it starts increasing, we can assume that we started overfitting, and we can just stop training (even if we haven't converged). This is cool and should also work well!

Note that any type of regularization adds to our training complications. How to we pick our penalty parameter? How do we pick how much dropout to have, or how MUCH the validation error needs to increase in order for us to think we should stop? There is already so much going on! This is not automated!

Ok, but even with all of these strategies, don't we think that these massive models will overfit? And perform badly in terms of the bias-variance tradeoff? Especially when we don't have that many training observations?

- Mostly, yes I do think this!
- But empirically, we have seen a strange phenomenon called double descent.
- It turns out it is not actually that strange or mysterious! Let's explore.

9.6 Double descent, and its relation to stochastic gradient descent

This is one of my favorite topics! See ISL 10.8 for a beautiful explanation. You will also explore this concept on your homework, and I may fill in these notes in more detail after Thursday's class.

The story of double descent is shown in Figure 5. We know that as the degrees of freedom in a model approaches n, the training set size, the training error goes to 0. This is the interpolation threshold; or the point at which our model is simply memorizing the training data.

Classical statistical theory says that the test set should be really bad at this point.

But, in practice, people noticed that for deep learning, you could keep making the model MORE complex (more layers, more nodes) AFTER the interpolation threshold, and while the training error remains 0, the test error starts to decrease again. This is called double descent: and some people thought it was magic! They thought it totally changed our interpretation of ALL of statistics!! What happened to the bias-variance tradeoff?

It turns out that the issue with this plot is what we draw on the X-axis. If we draw the X-axis as "number of parameters", double descent does indeed happen. But if we correctly make the axis the degrees of freedom, we will not see the double descent phenomenon. See this awesome recent paper: `https://proceedings.neurips.cc/paper_files/paper/2023/hash/aec5e2847c5ae90f939ab786774856cc-Abstract-Conference.html`.

On your homework, you will see how this plays out for linear regression onto natural splines. It turns out that, if we continue to increase the degrees of freedom of our spline basis past the interpolation threshold, we end up doing some automatic regularization that we didn't (necessarily) realize we were doing. This means that we aren't actually just

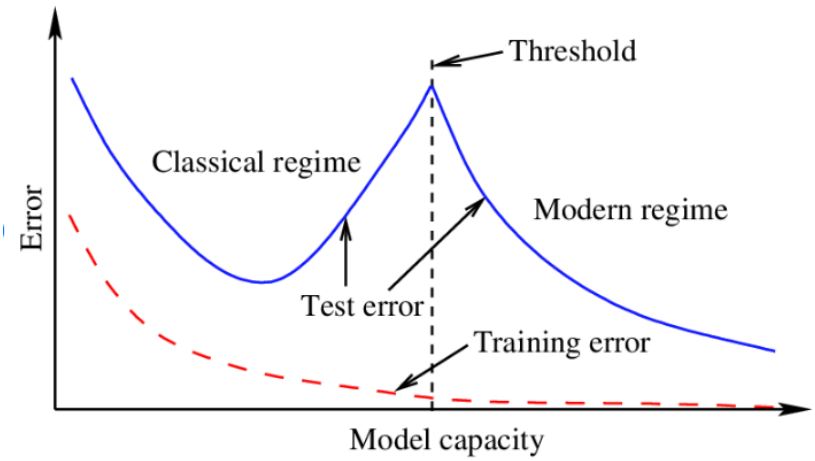


Figure 5: A classic illustration of double descent.

increasing model complexity without bound! We are fitting a regularized model, and so number of parameters is no longer equal to model complexity.

For deep learning, it turns out that stochastic gradient descent actually does implicit regularization. If there are many possible parameter vectors that would all give the same interpolating solution, SGD finds the minimum-norm solution. This is regularization! So ... there is no magic. Deep learning performs well because it has no bias and is ALSO regularized to have low variance.

9.7 Extensions of neural networks

We have barely scratched the surface. There are so many cool topics we could cover. Such as multitask learning, auto-encoders, convolutional neural networks, or recurrent neural networks. Auto-encoders also relate to semi-supervised learning. We are not going to cover these! But hopefully you now have a little bit of foundational knowledge to understand these better in the future. And you could always do one of these for your final project!

10 Monday, March 10: Classification and Regression Trees

This is one of my favorite topics! At first, this algorithm might seem like a totally "new idea". But later, I hope you will see a lot of connections with algorithms we have already studied!

One note before we start: there are actually a lot of algorithms out there for building classification and regression trees! I will get to this in "historical context." Whenever I don't say otherwise, if I am talking about a tree algorithm, assume that I am talking about the CART framework, which was popularized by Breiman et al. in 1984.

10.1 The main idea of the algorithm

Let's start with some really basic motivation for a regression tree. At first glance, it might seem a bit different than the other methods we have seen in this class.

The motivation begins with the left panel of Figure 6. We have two covariates, X_1 and X_2. And then we have a numerical response variable Y. We begin the algorithm with the simplest possible model. This model just ignores the covariates and predicts $\hat{y} = \bar{y}$ for all observations. In this case, $\bar{y} = -2.6$. This is the "intercept only" model. It has MSE given by: $\sum_{i=1}^{n}(y_i - \bar{y})^2$.

The algorithm then proceeds in a way that might remind you of forward stepwise regression. The algorithm says: at this moment in time, what is the binary split in my covariate space that will most improve the MSE, if I now let each sub-region be summarized by its own sample mean. This is illustrated in the second panel of Figure 6. In this particular dataset, it turns out that the best way to chop our space into two is to draw a vertical line at $X_1 = -0.89$. Once we do this, any observation to the left of the split gets $\hat{y} = 10$, and every observation to the right of the split gets $\hat{y} = -7$. This cutoff of $X_1 = -0.89$ was chosen greedily after considering all possible splits.

Let's write this down more rigorously. As of step 1 in the algorithm, our model has a single "region" in it. This region is $R_0 = \mathbb{R}^p$: the whole covariate space. The set of possible splits are indexed by $j \in 1, \ldots, p$ and $s \in 1, \ldots, n$, where

$x_{j,(s)}$ denotes the sth order statistic of the jth covariate[4]. At the first level of the tree, we search exhaustively for:

$$j^*, s^* = \underset{j \in 1,\ldots,p, s \in 1,\ldots n}{\arg\max} \; Gain(R_0, j, s) = \sum_{i \in R_0} (y_i - \bar{y}_{R_0})^2 - \left(\sum_{i \in R_{L(j,s)}} (y_i - \bar{y}_{R_{L(j,s)}})^2 + \sum_{i \in R_{R(j,s)}} (y_i - \bar{y}_{R_{R(j,s)}})^2 \right), \quad (17)$$

where $R_{L(j,s)} = \{i \in R_0 : x_{j,i} \leq x_{j,(s)}\}$ and $R_{R(j,s)} = \{i \in R_0 : x_{j,i} > x_{j,(s)}\}$. We refer to these as the regions to the left and the right of the split. Note also that $\bar{y}_R = \frac{1}{|i \in R|} \sum_{i \in R} y_i$ just denotes the sample mean of y within a certain region. This was a lot of notation, but it can be worth it to make sure that you really understand what the algorithm is doing! But the idea is simple: we choose the split that most improves the MSE of our model right now. Alternate Gain functions are possible, but this MSE one is the one used by Breiman et al. (1984) in their very widely used CART algorithm.

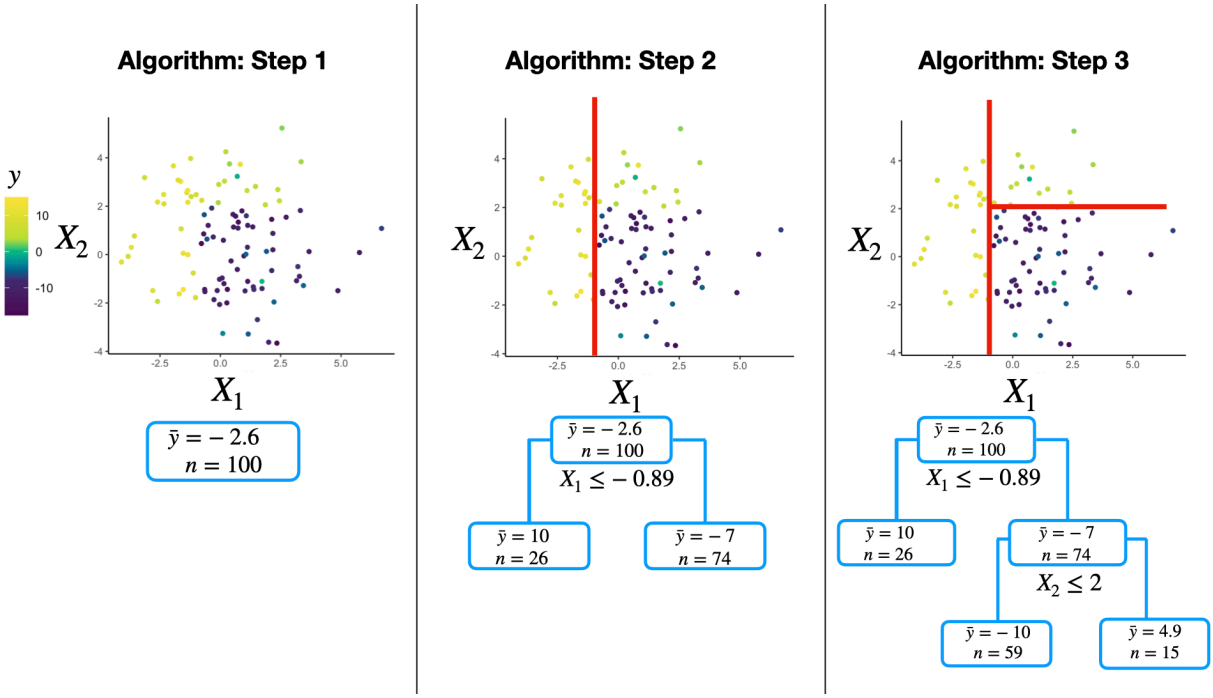


Figure 6: A little animation of the first three steps of a regression tree algorithm. Here, we have two covariates: X_1 and X_2. And then we have a numerical response variable Y. In this case, it seems like the data is generated from a "true tree" structure, where there are rectangles in covariance space that define the average value of Y. The CART algorithm uses binary recursive partitioning to greedily search for the best possible rectangles, according to MSE!

Finally, the right panel of Figure 6 shows what happens next. Regression trees are *recursive partitioning algorithms*. So, the greedy search for the best possible splits, but now R_0 is replaced by the already-selected regions $R_{L(j^*,s^*)}$ and $R_{R(j^*,s^*)}$. We proceed until a stopping criteria is met.

The final model is a set of nested rectangles in covariate space. The prediction in each rectangle is just the sample mean of the training observations in that rectangle. We draw the model as a tree; as shown in Figure 6. We write this model as:

$$\hat{y} = \hat{f}(X) = \sum_{R \in \text{TREE}} \bar{y}_R \mathbf{1}\{X \in R\}.$$

Sometimes the final regions in the tree are called leafs, and the series of splits that lead to them are called branches. This really leans into the tree analogy. I will just as often refer to them as terminal regions or terminal nodes, etc.

We really have now gone over the main idea of trees! Now we will talk about some considerations and extensions.

[4]In the case of non-ordinal categorical X_j, we assign orders to the categories by sorting them by their average value of Y in the training set. Thus, the order statistics are still defined; we just have a lot of ties in the order statistics: meaning that $x_{j,(1)} = x_{j,(2)} = x_{j,(3)}$ if all three observations belong to the same category.

10.2 Considerations, extensions, and historical context

10.2.1 Stopping criteria

How do we know how big to build our tree? As you may have guessed, this is the main knob that controls our bias-variance tradeoff for trees.

The biggest possible tree would keep splitting until we have one training observation per leaf region. This tree would have worst-case depth n, but likely depth closer to $log(n)$ if the tree remains more balanced. I think you all know that a tree like this will overfit severely to the training data, and we probably do not want to build this tree! We should stop before this!

Some simple ideas for when to stop are to pre-specify the maximum depth of our tree, or the minimum node size in our tree (i.e. stop splitting when the region has less than 10 training observations in it). However, these are stopping criteria that do not adapt to the amount of signal in the data. The true "right-sized" tree probably depends on the signal in our data! So, there are some more popular choices.

Consider (17). What if we decided to not choose ANY split if this gain does not exceed some threshold? For example, if the MSE does not improve by more than 5% when pick the best possible way to split this region, do not split this region at all. Such an idea is very similar to doing forward stepwise selection with an AIC or BIC stopping criteria. We recognize that ANY split will improve the training MSE *some*, but we want to make sure that the split seems "worth it". In the `rpart` R package, this is controlled by the parameter `cp`. From the `rpart` documentation: "any split that does not decrease the overall lack of fit by a factor of `cp` is not attempted. For instance, with anova splitting, this means that the overall R-squared must increase by `cp` at each step. The main role of this parameter is to save computing time by avoiding splits that are obviously not worthwhile." The default in `rpart` is 0.01, which is pretty small. This is an example of an *adaptive stopping criteria*.

Your textbook (ISL) calls the adaptive stopping approach above "short-sided". The idea is that, due to the greedy nature of CART, it could be the case that no split will exceed the MSE threshold "right now", but it could help us uncover an important interaction in the next level of the tree. So, it would be good to avoid stopping too early!

Another idea is to grow a tree that is purposely too large, and then to `prune` splits away from the tree. More formally, let:

$$L_\lambda(T) = \sum_{R \in T} \sum_{i \in R} (y_i - \bar{y}_R)^2 + \lambda |T|.$$

This is a loss function that incorporates both tree MSE as well as a penalty term that penalizes larger trees: $|T|$ denotes the number of leaves in a tree. It turns out that nice properties of trees tell us that, if we start by building a large tree T^{big}, then there is a simple nested sequence of sub-trees that corresponds to the best tree for different values of λ. And we can find this sequence of trees easily: starting from T^{big} we prune the "weakest link" in the tree one at a time. This gives us the nested sequence of trees. This is a "solution path"- like we had for Ridge or Lasso.

To be really complete, we would want to use cross-validation to pick the best value of λ. And then we would want to re-fit a tree to the full training set using this value of λ. If you are not yet comfortable with the idea of cross validation + refitting to entire training set, please ask questions before the midterm! It is an important idea!

10.2.2 Categorical predictors

CART can really seamlessly handle categorical predictors. We don't need to turn them into dummy variables!

The Breiman et al. CART algorithm always makes BINARY splits; even for categorical variables. You might encounter other decision tree algorithms some day that would make a three-legged-split for a categorical variable with three categories.

Ordinal categorical variables still have order statistics, so we split in the same manner than we did above in (17). For unordered categorical variables with k categories, we do not need to consider 2^k possible ways to split the variable into two groups. We just order the categories by $\bar{y}$ on the training set, and then only let ourselves make a binary split that respects the ordering of these categories.

NOTE: some critics of CART do not like the following fact. A numerical covariate has $O(n)$ chances to be chosen as the winning split. A binary categorical covariate has only 1 chance to be chosen as the winning split! If the binary covariate is truly important, but is associated with the numerical one, the numerical one will often appear in our tree! Just due to the extra random chance that it gets to be selected as a winner. This is too bad! There are modifications to the algorithm that try to get around this bias.

10.2.3 Categorical response

CART is really easily used for either regression or classification.

For classification trees, all we do is modify our gain function. Instead of using MSE, we split based on either Gini Index or Entropy. With either of these, we are choosing a split that makes the resulting child notes as pure as possible: meaning homogenous with respect to y.

While we use Gini Index or Entropy to choose our splits greedily, we might still do cross validation using simple 0/1 classification error loss.

10.2.4 History

I got some nice historical notes from this paper: `https://pages.stat.wisc.edu/~loh/treeprogs/guide/LohISI14.pdf`.

The first classification tree algorithm was in 1963, and it was published under the name "automatic interaction detection (AID)". This should already tell you something about why trees have been so popular- people absolutely love this feature that they can identify interactions without those interactions being pre-specified. At first, AID did not attract much attention from statisticians. People were worried about overfitting. People were also worried that, in the presence of correlated predictors, the conclusions could be spurious. Probably only one of the pair of correlated predictors will be selected for the tree, and the other will be totally left out. It might be dangerous to over-interpret the tree as signaling variable importance in that case[5]. At the same time, however, computer scientists were making their own decision trees for "concept learning." It seems that an early reference is Hunt, Marin, and Stone (1966). The algorithm that I learned about in CS class is ID3, which was published by Quinlan in a series of work in the 1980s. This is again a setting where great ideas were coming out concurrently in multiple disciplines!

The credit for "popularizing" trees, at least in statistics, goes to Breiman et al. in 1984. I am pretty sure that one of the main innovations by Breiman et al. that really helped people take up trees was the introduction of efficient cost-complexity pruning.

Since trees first became popular, there has been a lot more work on them! Some statisticians (including me) do not like that the splits in a CART tree lack a notion of statistical significance[6] One family of competing algorithms is called CTree; splits are chosen based on statistical significance, rather than based on an improvement in MSE. There are also models that make splits on linear combinations of variables, instead of single variables. Or models that fit an entire regression model to each leaf node, rather than choosing a piecewise constant model. And I am sure I am missing a lot of innovations!

10.3 What do we think about trees?

10.3.1 Bias

An extremely large tree has almost no bias. We can approximate basically any function with a big combination of step functions. So if n is big and our tree is big, trees are flexible. It is nice that trees can capture interactions between variables and other sorts of non-linear relationships without us needing to pre-specify.

However, a tree of reasonable size is very limited. Consider the case where $Y = 5 * X_1$. To approximate this well by a sequence of binary splits on the variable X_1, we will need a lot of splits! As we add more and more, our approximation will get better and better! But this function will obviously be much easier to approximate with a linear model!

In general, if the true data generating mechanism is not made up of rectangles in covariate space, a simple tree might struggle. But, if you are worried that rectangles in covariate space are REALLY biased, not that a really big tree works a lot like 1-NN, which we know does not have bias. See below for more about the connection to KNN.

10.3.2 Variance

An extremely large tree has a lot of variance because it overfits. But even a small tree can have a lot of variance due to the greedy nature of trees. A small change to the input dataset can totally chance our first split, which could then affect our whole tree- especially if we only plan to make a small tree. So trees do not score super well for variance.

10.3.3 Interpretability

Trees are a dream! We can explain our predictions so easily to non-experts! Even things like interactions do not seem scary when they are presented as a tree! However, I have some notes below about a fundamental issue. If we

[5] I agree!

[6] This is very related to my claim, which we never finished going over, that you cannot easily do inference after stepwise regression. Greedily searching for optimal things makes inference hard.

know that our tree is unstable (high variance), what are we really interpreting? This makes formal inference really important. Which is something I worked on in my PhD!

10.3.4 Use-ability

Trees are a dream! They are "off the shelf". The only tuning parameter is tree size, and tree size is something that we understand. And the nested-tree property of the pruning algorithm makes it really nice.

We don't need to scale our predictors or worry about whether or not we are including an intercept. Categorical predictors need not be converted to binary. Missing data is also seamless to incorporate using the concept of surrogate splits. You don't need to start with preprocessing of variable selection. Trees do built in variable selection, and are not TOO impacted by curse of dimensionality.

10.3.5 Computational Efficiency

When I first learned about trees, I thought that the algorithm sounded slow. Search exhaustively for best possible split? But now that I know about things like neural nets, I'm less concerned haha. Trees are pretty easy to implement. The greediness saves us. $O(np)$ things to compute at each level to choose a split. At least no squared terms, right? And the tree depth will be at MOST n, usually more like $log(n)$ if balanced. So maybe $O(nplog(n))$. This really isn't bad. The pruning algorithm is also efficient.

10.4 Drawing connections!

Right now, it might seem like decision trees are a random topic we have thrown in that are totally different from the other algorithms that we have seen. This is not true! Let's draw some connections.

10.4.1 Write as a regression model or optimization problem

Really, a regression tree defines a model class of piecewise constant models on rectangles in covariate space. So we could just abandon the whole tree idea and say that we are looking for a set of regions $R_1, \ldots, R_T$ such that we minimize

$$L_\lambda(T) = \sum_{r=1}^{T} \sum_{i \in R_r} (y_i - \bar{y}_{R_r})^2 + \lambda |T|,$$

and where $\hat{y} = \sum_{r=1}^{T} \mathbf{1}(X \in R_r) \bar{y}_{R_r}$.

This looks a bit more like an optimization problem or a regression problem. We are regressing onto lots and lots of possible step-function indicator variables, but we are doing variable selection first to decide which ones to include. We can think of trees in this way! And then the innovation is just that this loss function will be really really difficult to minimize exactly. So, we give up on searching fully for the optimal tree. We instead use our greedy, top-down approach. Which finds a pretty good tree, but not necessarily the very best optimal tree. This reminds us of the difference between best-subset regression, which is infeasible, and stepwise regression, which is a greedy approximation.

10.4.2 How is a regression tree like KNN?

At the end of the day, the prediction $\hat{y} = \hat{f}(x^{\text{test}})$ for a new datapoint x^{test} is the sample mean of some training points y that are near x^{test} in covariate space. This sounds a lot like KNN! How is it different?

- In finding the points that are "near" x^{test}, we do not consider all covariates $X_1, \ldots, X_p$. We only consider the ones that were selected for the tree. If we did a good job selecting splits for the tree, this should really help with the curse of dimensionality problem that KNN encountered. Irrelevant variables do not contribute!
- And we selected rectangles that lead to good predictions! So we should have retained important directions while ignoring irrelevant ones.
- We still have the "prototype" interpretation: "you got these predictions because other previous points in your rectangle had this average response." That is a really nice prediction!
- Overall, regression trees can maybe be seen as something that improves on KNN.

10.4.3 Are regression trees parametric or non-parametric

We just said that regression trees are kind of like stepwise regression (which is very parametric), but we also said that they are kind of like KNN (which is non-parametric). Which is it?

Remember that the definition of non-parametric is that our model complexity grows with our sample size n. As we said, the absolute biggest tree we can grow has n leaves: one per training datapoint.

So, if we are building trees to unconstrained depth, then regression trees are non-parametric and are mode like KNN. Or, if we build trees with the restriction that the minimum node size is 10 but do no other pruning, then our model really is quite a bit like 10-NN. If we add more training datapoints, we can make more nodes, and so the complexity grows. Unless of course we run out of possible splits, which could happen if our variables are all categorical with not that many categories.

However, if we build trees to a maximum depth of 3, then this is parametric and is a lot more like stepwise regression. We could enumerate the set of all possible models based on the size of our covariate space, and the models would not get more complex as we increase n; we would just get more observations per terminal node which reduces variance. Actually, we technically have more possible splits to choose from at each point when n increases (order statistics), so if you want to be really precise about this being parametric imagine that your Xs are discrete and can take on only a set number of values.

Overall, I think that the difference between fixed-depth trees and fixed-node-size trees as parametric vs. non-parametric models is interesting! And of course, adding in pruning changes the complexity again. You could study this yourself on a final project!

10.5 Philosophically, can we really call an unstable model interpretable?

I think there is a really important point when we talk about the pros and cons of decision trees. I will try to illustrate this point with my R demo.

Regression trees are supposedly so great because they are interpretable. But ... how do we know when we are seeing the truly important variables vs. seeing noise? We could also perturb our training dataset slightly and get a totally different tree, in the context of correlated predictors or similarly important predictors, etc.

Isn't this really bad? What does it mean to interpret something that is unstable? We should probably add some notion of statistical significance to CART. Or work on making it more stable! This is the topic of a lot of research, including my own. Let me know if you want references!

11 Thursday, March 13: Support Vector Machines

Like Monday's class, today's class might at first seem like we are just randomly jumping to a new algorithm. But I hope that by the end of class, you are once again motivated to think of this instead as a new lens through which to study our important themes from the class. Today's theme is all about feature engineering in a really clever way, and what that gets us. SVMs also have some historical importance!

11.1 Motivation for SVMs

To motivate today's class, we are going to be thinking about a picture that we haven't thought about since the day that we covered LDA. The picture is of two quantitative predictors X_1 and X_2, and a categorical response y (shown by the colors). See Figure 7.

Consider the left panel of Figure 7. Our goal is to predict Y using X_1 and X_2. In this case, the tasks looks almost unbelievably easy. The classes are linearly separable!

We learned during our LDA lecture that logistic regression, surprisingly, does quite badly in this perfectly separable case. If you try to fit a logistic regression in R to this data you will get warnings about convergence and "fitted probabilities of 0 or 1" occurring. Logistic regression is all about modeling probabilities of $Y \mid X$, and there the estimates for certain regions all end up being 0 or 1, which makes the exact coefficients really uncertain.

You can envision this problem a little bit in the middle panel of Figure 7. All three lines perfectly separate the two classes, but they all have totally different slopes. How do we know which line to choose? You know based on last class that a classification tree will choose the vertical straight up and down line. But ... is that really going to be the line that generalizes best to new examples?

LDA and QDA chose between these different lines by making a very strong assumption. They assume that $X \mid Y$ is Gaussian. See the right panel of Figure 7. Then, you can draw estimated Gaussian contour lines for each class. The decision boundary chosen by LDA or QDA has to do with when these contours are set equal to one another: when is the red class or the blue class more likely, given X, based on the estimated Gaussian densities?

What if we want a way to pick between all of the lines in the center panel of Figure 7? And we want a way that does not make a Gaussian assumption, or arbitrarily restrict itself to straight lines defined by a single variable? This is the

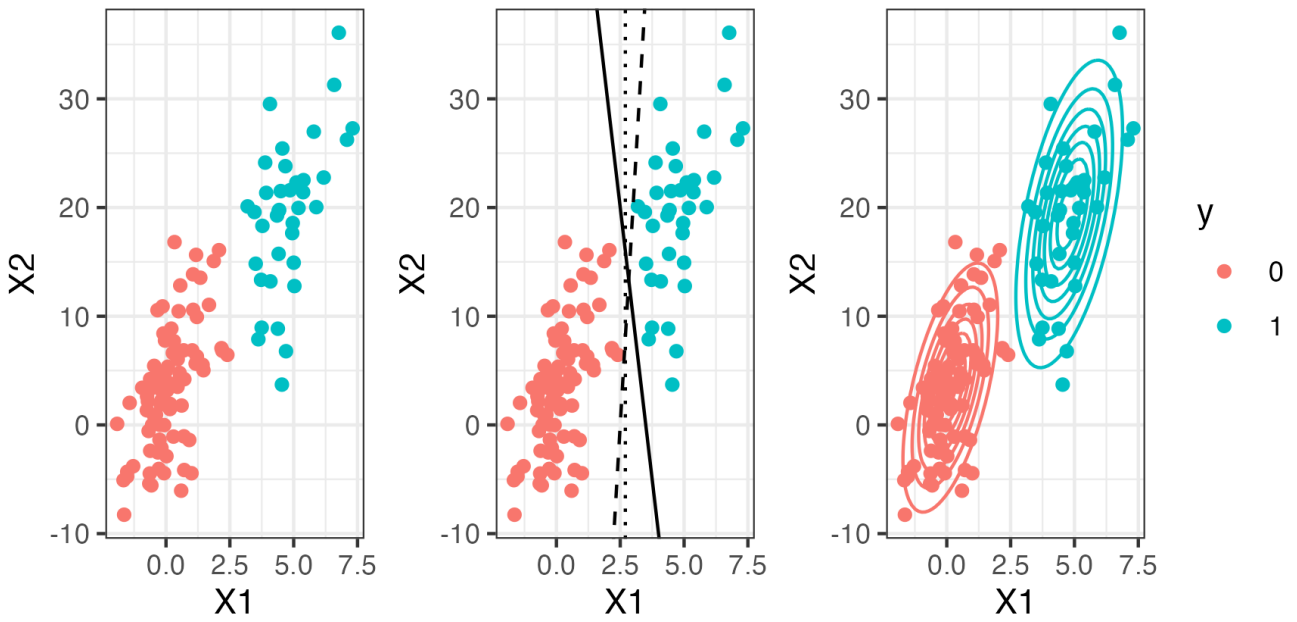


Figure 7: A figure to motivate SVMs, and their differences with logistic regression, LDA, or QDA.

idea of the maximum margin classifier, which is summarized in Figure 8, which is taken from ISL. The idea is quite simple: let's pick between all of the possible separating lines by choosing the one that is as far as possible from all of the training observations.

11.2 Maximum margin classifier using constrained optimization

How do we actually fit the line in Figure 8? There is some math that relates to how we actually draw these lines onto plots. Also, note that we could have more than 2 X variables, and then we would be using a plane and not a line to separate our classes.

In general, we will write our linear boundary as the line:

$$\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p = 0.$$

The idea is to pick a line so that $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p < 0$ whenever $y = -1$ and $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p > 0$ whenever $y = 1$. For today, we are doing binary classification and we are writing our two classes as -1 and 1, for simplicity.

Any of the lines in the center panel of Figure 7 actually have this property. So now the idea is to do even better. Let's both have it be the case that $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p < 0$ whenever $y = -1$ and $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p > 0$ whenever $y = 1$, but also have it be the case that $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$ is basically never TOO close to 0. Because when it is near 0, it means that points are close to the line and we have uncertainty. If all of our points are far from the boundary, we have less uncertainty.

One quick note about these hyperplanes: if our class boundary is the line $\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p = 0$, then $c(\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p) = 0$ defines the same class boundary for any constant c. Thus, in defining planes, we restrict our attention to β vectors where $||\beta||_2^2 = 1$: to make sure that we have a unique solution.

So, the maximum margin classifier says:

$$\begin{aligned}
&\underset{\beta_0,\beta_1,\ldots,\beta_p,M}{\text{maximize}} \; M \\
&\text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1 \\
&\text{and } y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M.
\end{aligned}$$

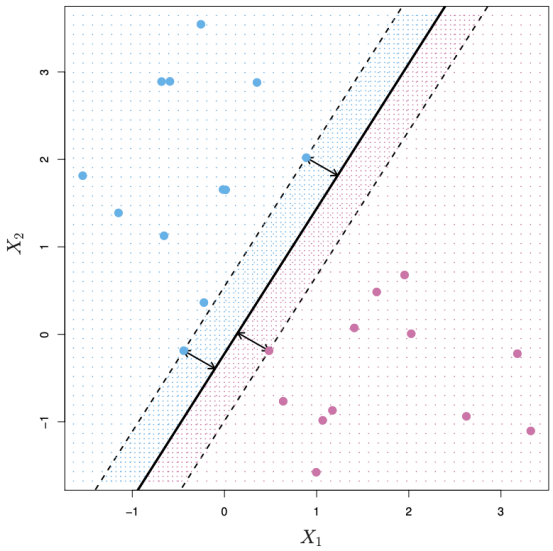


FIGURE 9.3. *There are two classes of observations, shown in blue and in purple. The maximal margin hyperplane is shown as a solid line. The margin is the distance from the solid line to either of the dashed lines. The two blue points and the purple point that lie on the dashed lines are the support vectors, and the distance from those points to the hyperplane is indicated by arrows. The purple and blue grid indicates the decision rule made by a classifier based on this separating hyperplane.*

Figure 8: The maximum margin classifier picks the line that is far as possible from all observations.

The first constraint is just for uniqueness. The second constraint says that every training observation is correctly classified (if M is positive), and the fact that we are maximizing M says that we are trying to make all of our prediction values far from 0 (recall that y_i is just -1 or 1).

For your purposes: know that people are good at convex constrained optimization. So this can be solved if the classes are linearly separable: i.e. if a separating hyperplane exists!

There are actually a few interesting properties of this solution. One interesting property is that the optimal hyperplane actually depends on the data ONLY through the points that lie ON the margin M: the other points do not contribute to the βs at all. In Figure 8, there are only 3 of these "support vectors" that actually impact our classifier. That is interesting— we could add 1,000 blue points to Figure 8, and as long as we add them on the "blue side" of the current hyperplane, our solution does not change at all. This is interesting!! And is certainly different than LDA– in LDA, the overall class proportions affect our decision boundary (via the prior).

You might be worried that this property of SVMs- that they only depend on a few observations- could lead to overfitting. I am certainly worried about that! We usually do not want ONE datapoint to impact our entire classifier that much!

The question you should definitely be asking yourself right now is: what if our classes overlap? Real data never looks like Figure 7. How do we use an SVM in this case? To answer this question, we will discuss two concepts.

- Concept 1: we can rephrase our optimization problem to allow a small number of mistakes. (this will actually also help with the overfitting concern, even when our classes are technically separable).
- Concept 2: if we transform our feature space enough, we can probably make the class linearly separable in new feature space.

We will discuss both of these!

11.3 Support vector classifier

The support vector classifier just takes the maximum margin classifier and says "let's be okay with a few mistakes" (concept 1).

We once again write a constrained optimization problem, and once again smart people know how to solve it efficiently because it is convex.

$$\begin{aligned}
&\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n,M}{\text{maximize}} M \\
&\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1 \\
&\text{and } y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq M(1-\epsilon_i) \\
&\text{and } \epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C.
\end{aligned}$$

The only thing that we added here is that a training datapoint is allowed to be on the "inside of the margin" ($0 < \epsilon_i < 1$), or even on the wrong side of the hyperplane ($\epsilon_i > 1$). But, we limit how many of these points are allowed using the total cost C, which is something that we pick. We still make predictions based on whether or not $\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$ is greater than or less than 0: we just know that there are some mistakes in our training set.

We usually pick C with cross validation. A large C leads to a cross validation with more bias but less variance. Because, when C is big, we let lots of individual datapoints have non-zero ϵ, which means we let them "break our rules". This lessens the dependence of the classifier on the individual observations, which means less overfitting and less variance.

Once again, it turns out that ONLY datapoints with non-zero ϵ affect the final classifier. If you remove other points, or add other points that fall outside of the margin on the correct side, you do not change the classifier! The observations that DO affect the classifier are called the support vectors. When C is big: we have a LOT of support vectors. So we depend on a LOT of the data– low variance! When C is small: there are only a few support vectors — we fit the data really well but we have high variance!

Overall though, we are still only focusing on observations that are near the boundary. Points that are very clearly members of one class or the other do not affect our classifier rule! This makes it more like logistic regression than LDA.

11.4 Support vector machine

If our data are not linearly separable, we could just make our cost C really big until we end up with a valid classifier. But ... sometimes a hyperplane in our original feature space is simply not going to give us a good rule! We now turn to Concept 2.

If the relationship between our predictors and our classes is not linear, we should not try to use a separating hyperplane! But the key insight of a support vector machine is that maybe we can use a separating hyperplane in a new feature space. This is exactly the same idea as just adding polynomial terms to linear regression when we think our function is not linear!

If you just put in X_2^2 as a "new covariate", then you can still fit a support vector classifier. It will give you an equation that has a $\beta_k X_2^2$ term in it. This is not a hyperplane in the original feature space. But if you drew a new feature space that had an axis for X_2^2, this would be a hyperplane.

So, we can add as many features as we want. We can actually just keep adding features until our classes are perfectly linearly separable, and then we wouldn't even need a budget C! Although this is probably a bad idea from an overfitting perspective. We should probably keep C, but also add dimensions if we think we have non-linearity.

You could just add a lot of features and then directly try to fit a support vector classifier. But, you could quickly get overwhelmed by a huge number of features, and the computations would actually get really hard (I told you that smart people know how to do these optimization problems, but that doesn't mean they are trivial).

So, running with this idea, we will learn a little but more about optimization so that we can understand a really magical thing called the kernel trick.

11.5 Optimization

We are not covering optimization very much in this class. We are sprinkling in a few concepts here and there (such as gradient descent), but really optimization is a topic of an entire course! So I will not pretend to do justice to these ideas.

BUT, in order to understand the kernel trick, which is VERY important, we need to understand a little bit about optimization and how we actually solve for our support classifier.

We first note that the support vector optimization problem can actually be rewritten as:

$$\begin{aligned}&\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n}{\text{minimize}} \frac{1}{2}||\beta||_2^2 + \lambda\sum_{i=1}^{n}\epsilon_i\\ &\text{subject to: } y_i(\beta_0+\beta_1 x_{i1}+\ldots+\beta_p x_{ip}) \geq (1-\epsilon_i)\\ &\text{and } \epsilon_i \geq 0.\end{aligned}$$

There are two differences between this and what we wrote before. First, recall that we were restricting ourselves to $||\beta||_2^2 = 1$ because any scalar multiple of a β vector gives us the same separating hyperplane. But ... that same scalar multiple also changes the meaning of a margin M. So, we might as well look for the SMALLEST $||\beta||_2^2$ that gives us a margin of 1. And it removes one thing for us to worry about - we don't actually need M. Second, we just moved the penalty on the size of the ϵ to the objective instead of setting it as a hard budget constraint: we already know from Ridge/Lasso that we can do this. We now have a penalty parameter (chosen via cross validation!) instead of a budget constraint.

Ok. Now that it is in this form, how do we solve this?

Well, remember Lagrange multipliers? Kind of? From Math 150/151, or from Econ 251, or from Amina/Bekah's colloquium, or from the Ridge/Lasso document that I put on GLOW? It's okay if you don't remember the details. But the idea is that we can solve a constrained optimization problem by first writing it in its Lagrangian form. This form introduces many free variables. In this case, the Lagrangian primal is:

$$L_P = \frac{1}{2}||\beta||_2^2 + \lambda\sum_{i=1}^{n}\epsilon_i - \sum_{i=1}^{n}\alpha_i\left(y_i(\beta_0+\beta_1 x_{i1}+\ldots+\beta_p x_{ip}) - 1 + \epsilon_i\right) - \sum_{i=1}^{n}\mu_i\epsilon_i.$$

We solve this by taking derivatives with respect to β and ϵ and setting them to 0. These derivatives then set up big system of equations. There is theory from convex optimization that says that, instead of solving the Lagrangian directly, we can instead solve the dual function. Go read about this in the Boyd book (chapter 5) that I linked to the Ridge/Lasso notes! `https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf`.

In this case, the Lagrangian dual is:

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j.$$

Convex optimization theory says that we can solve our optimization problem by maximizing this with respect to α.

I still haven't told you how to solve this. Go read the Boyd book: Chapter 5! But the magical fact that you should notice is that this dual depends on the training data covariate vectors x_i and x_j only through the dot product $x_i^T x_j$, which is a scalar. For our n datapoints, we can store all of the values $x_i^T x_j$ in an $n \times n$ matrix: p, which is the dimension of our (possibly expanded) feature space, actually does not matter here, once we have written down the inner products.

11.6 The kernel trick

To make this more concrete: if we start with original features vectors x_i for all n individuals, and then transform them into a higher-dimensional space, we could denote this with $h(x_i)$. This might just be a function that takes a p-dimensional vector $(x_1, \ldots, x_p)$ and returns the $2p$-dimensional vector $h(x) = (x_1, \ldots, x_p, x_1^2, \ldots, x_p^2)$.

The support vector optimization problem, in dual form, turns out to only depend on $h(x_i)^T h(x_j)$. Let's go one step further. I wrote this as a dot-product, but less expand this to any inner product, which we will denote $K(h(x_i), h(x_j))$. This is just a generalization of the dot-product to non-Euclidian spaces. My main reference is the wikipedia page!

What this means for us: we could choose a high-dimensional transformation $h(x)$, and if we happen to have a convenient way to write down $K(h(x_i), h(x_j))$ from a formula, we might need to never actually explicitly write down the high-dimensional vectors $h(x_i)$. We call $K(h(x_i), h(x_j))$ the kernel function. We can work directly with the $n \times n$ matrix of inner-products, and can actually solve for the hyperplane without even needing our high dimensions!

Common choices of the kernel function k are given on page 424 of ESL. The really cool thing is that the actual classifier can also just be written in term of the kernel function: we never need the p-dimensional $\hat{\beta}$.

We can write our classification for a new input x as:

$$\hat{f}(x) = \sum_{i=1}^{N} \hat{\alpha}_i K(x, x_i) + \hat{\beta}_0.$$

All we need to do is figure out which vectors are the support vectors. These will have $\hat{\alpha}_i \neq 0$. Then, once we have this, we estimate an intercept, and write our classification rule! So cool! We predict class 1 if $\hat{f}(x) > 0$ and predict -1 otherwise.

Kernels have now been used in a lot of contexts beyond SVMs! You already know from polynomial regression and splines that expanding our feature space is nice. And this is a cool extension about when this is really efficient.

11.7 Reconciling Concept 1 and Concept 2

We started with the maximal margin classifier, which was linear and allowed no mistakes.

We then started allowing mistakes, using a budget C, in case our classes are not linearly separable. But then we enlarged our feature space a lot. In a high-dimensional enough feature space, all datasets become linearly separable. So ... do we throw out the idea of making mistakes?

No! See the figures on page 425 on ESL. Big warning: the penalty C used in ESL is the OPPOSITE of the budget/cost C used in ISL. How confusing! That's why I made our penalty λ in these notes. In the way that I wrote it in these notes for the kernel trick version. The penalty works as we would expect.

A big value of λ places a big penalty on mistakes. This means that the boundary will be VERY wiggly and will overfit the training points: so as to avoid any mistakes. This will also mean very few support vectors, so the boundary is totally determined by a small number of datapoints. A small value of λ encourages small $||\beta||_2^2$, which makes our boundary smoother.

11.8 Why are we learning about SVMs: do they work well and are they important?

Check out ISL 9.5, or ESL 12.3.4.

When SVMs came out in the 1990s, they were splashy and new and exciting. They worked really well in terms of accuracy, and seemed sort of mysterious. People actually thought that the kernel trick would help us avoid the curse

of dimensionality. Unfortunately(?), this isn't true. It turns out that SVMs are not so mysterious after all: they relate to all of our themes that we have seen in this course so far. And studying these connections is kind of cool and beautiful!

A few takeaways:

- Chapter 9.5 of ISL explains the close connection between SVMs and logistic regression. SMVs really do just have a loss+penalty form, which we are all used to.
- We still have a curse of dimensionality. If our true classification boundary depends linearly only on X_1 and X_2, but we use a massive non-linear kernel, our "machine" will struggle to learn the correct rule. Bummer!
- Choosing the cost parameter C or the penalty λ is still SO important. We don't want to overfit! We need cross-validation, which sounds slow. Luckily, like for lasso, it turns out that we can solve efficiently for many values of λ at once.

Also, we should mention that we can use SVMs for multi-class classification and regression: we just only went over binary regression. Second, note that the whole idea of a plane means that the Xs cannot be categorical: we need to represent categorical variables as dummy variables, and for dummy variables polynomial transformations don't help us :(.

Some overall takeaways: with a big kernel, SVMs have low bias. But we need to control the bias/variance tradeoff with our penalty parameter. They are pretty computationally efficient, and only a little bit interpretable. If we have few support vectors, we can sort of interpret them which is nice. If we have a lot of support vectors and a big kernel, they are basically a black box. So, don't choose them for their interpretability. For useability, we just need to choose a penalty parameter and kernel, which is not so bad.

I almost took SVMs out of this class! Because, now that neural networks exist, IDK if people actually use SVMs. But, seeing a big variety of different types of algorithms is nice. To help you draw connections. Also, seeing a variety of different motivations for algorithms might help you understand how new algorithms are developed! Which is important!

Also: the name "support vector machine" sounds so scary and you might encounter it someday! It's not actually scary: now you know it is a penalized hyperplane classifier!