

Contents

1. Introduction / Regression	1-2
2. Assessing Performance; Bias + Variance Tradeoff	2-9
3. Cross Validation / Regularization	3-14
4. Feature Selection + LASSO	4-20
5. Classification	5-25
6. Logistic Regression	6-31
7. Naive Bayes and Decision Trees	7-37
8. Ensemble Methods	8-47
10. Fairness in ML	10-53
11. Clustering	11-59
12. More Clustering Methods	12-65

Chapter 1

Introduction / Regression

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

1.1 What is Machine Learning?

Definition 1.1: Machine Learning

Machine learning is the study of algorithms that can improve their performance at some task with experience.

From the Netflix series recommended to you, to Siri's assistance, to the GPS system on your phone, Machine learning is overwhelmingly prevalent in the majority of the services presented by the modern technological world. This is because with machine learning, we can build models that analyze and automate processes in the real world. The key idea behind a machine learning algorithm is to intake data from the real world and improve its model of the real world accordingly.

By modeling the real world as accurately as possible, machine learning aims to predict or categorize the unseen. It does so by exploiting relationships in data as we will explore later on in this chapter.

Example(s)

A common example of the use of machine learning is Netflix's series recommendations. If your Netflix account has been designated as a consistent watcher of a specific show, you will be recommended shows that Netflix thinks you will enjoy in the future. How does this happen?



Figure 1.1: With enough data, a machine learning algorithm can make decisions that seem intelligent.

Sometimes, such relationships in real world data are not easily conceivable to the human mind, which is what gives machine learning algorithms the illusion of being “intelligent”, especially to those only familiar with AI in the context of science fiction. However, we invite you to entertain the thought that perhaps more exciting than a science-fiction cyborg is the idea that massive data computations are enough to present a sense of genuine intelligence like humans. Whether it is generating speech or classifying animals, such tasks humans for so long thought were only possible by their own brains became a matter of mathematical modeling.

Let us consider the Netflix example above. If a friend really enjoys “The Office”, you might be inclined to recommend that they also watch “Parks and Rec” due to the similar style of comedy in the shows. This is a simple prediction analysis you could have done in your own mind—no machine learning model required

here. But what if you wanted to do the same for lots of people who have different show preferences than you? In our current digital era we now have massive amounts of data with thousands of variables—some of those variables interact with each other, and we don’t exactly know which ones, or how, or why. This is where machine learning comes in. The applications of this technology have had profound impact on various arenas like medicine, language processing, robotics, and most importantly, predicting **the winning contestant of The Bachelor!**

1.2 Types of Machine Learning

There are different types of machine learning, defined by how the “learning” happens. The main three are:

Definition 1.2: Supervised Learning

Supervised Learning is the most basic type of machine learning. A model is presented with a dataset that has labeled input and output pairs.

This learning is “supervised” by humans with guidance on what the input is exactly and what the output is exactly. The model will improve its performance by training on these explicitly labeled pairs. A classic example is image classification, like sorting animals, where a bunch of labeled images are fed into a model so the model has lots of pre-labeled examples of “cats” and “dogs”.

Definition 1.3: Unsupervised Learning

Unsupervised Learning analyzes an unlabeled dataset. There is no specified input or output, just a big mesh of data. Thus, it is up to the model to categorize and find patterns within the data.

Clustering things that are similar with each other is a popular technique in unsupervised machine learning. Image classification can be unsupervised too: The model is fed a bunch of pictures of cats and dogs but none of these images are labeled. Because all cats kind of look like each other (they are hence structurally similar in data), and all dogs kind of look like each other, but cats and dogs look different from each other relatively in the dataset, a model can assert that these are probably two distinct categories.

Definition 1.4: Reinforcement Learning

Reinforcement Learning is a type of machine learning that devises a method to maximize desired behavior in a model by using a reward system, and penalizes undesired behavior.

What makes reinforcement learning unique is that its reward system behaves as a separate agent so that short-term successes aren’t stalled upon, and the model prioritizes long-term success. An example of reinforcement learning is in Pac-Man, where long-term planning coded into the algorithm can achieve superhuman performance. This type of learning is very common in gaming, though we will not expand upon it further in this text as its usage is limited.

For now, we will focus on Supervised learning, so don’t worry if the other two don’t make sense right now.

1.3 Main Applications of Machine Learning

The rest of this text will be organized in a way that follows the 5 main kinds of machine learning application:

1. Regression
2. Classification

3. Document Retrieval & Clustering
4. Recommender Systems
5. Deep Learning

1.4 Case Study 1: Regression

1.4.1 What is Regression?

Definition 1.5: Regression

Regression is a supervised learning technique used to predict a dependent variable based on one or more independent variables by estimating a relationship between them.

Regression is the first application of machine learning that we will discuss here. Regression is applicable when we want to know the answer to one thing, but we have clues coming from multiple sources. An example of this is house prices. What can you use to determine the price of a house? Its square footage? The neighborhood it's in? What about the view? What about when it was built? There are many independent variables here, and they may all be important for predicting our dependent variable, the price.

1.4.2 How Does Regression Work?

Let's first go over the general premise of how regression works. Consider our dataset of x and y pairs. The x 's are training examples, or the input. If we are going off the house prices example, you can think of each x as a list of all the information about one specific house that helps determine its price, like the square footage and the year it was built. The y 's are the values of our output. In this case each y is a price for one specific house. We will say that we have n number of x 's and y 's like so:

$$(x, y)^{(1)}, (x, y)^{(2)}, \dots, (x, y)^{(n)} \quad (1.1)$$

So, $x^{(1)}$ is a list of facts about house 1, and $y^{(1)}$ is its price. If we let i be equal to some index number, then what we mean for any house i is that it has information $x^{(i)}$ and price $y^{(i)}$. Given all our x and y pairs, we want to build a predictor function that learns how to map $x^{(i)}$ to $y^{(i)}$. We would ideally like to be able to predict a house price (output) from a house's information (input).

It is important to note that a single x is a list. It includes various pieces of information about a house like its square footage and its year built. We will call these pieces of information "features".

Definition 1.6: Feature

A **feature** in machine learning is a single variable that is part of the input. Our input "house information", has features "square footage" and "year built".

Notation-wise, we will let capital X denote a single feature and d be the number of features we have in a single input $x^{(i)}$:

$$x^{(i)} = X_1^{(i)}, X_2^{(i)}, \dots, X_d^{(i)} \quad (1.2)$$

These features are important to label distinctly because some features in an input may be more important for predicting the output than others.

For simplicity's sake, we will consider an input of one feature, square footage. So, let $x^{(i)}$ be a list of size $d = 1$ containing only square footage as input. Observe the pairings below:

$$\begin{aligned}(x_1, y_1) &= (2318 \text{ sq.ft.}, \$315k) \\(x_2, y_2) &= (1985 \text{ sq.ft.}, \$295k) \\(x_3, y_3) &= (2861 \text{ sq.ft.}, \$370k) \\&\vdots \\(x_n, y_n) &= (2055 \text{ sq.ft.}, \$320k)\end{aligned}$$

Figure 1.2: There are n number of x and y pairs, with x being the square footage and y being the price of each house.

To use regression, we have to assume that there exists a relationship between x and y . This means that we have to assume that somehow, for some reason, the square footage has an effect on the house price. Thus, we are making the assumption that:

$$y \approx f(x) \tag{1.3}$$

where $y \in \mathbb{R}$ and $x \in \mathbb{R}^d$. Remember that \mathbb{R} is the set of all real numbers. So all this notation means is that y is a real number, and x is d number of real numbers (since our input can be made up of multiple features). Since we assumed that y has a relationship with x , we have asserted that some function applied to x should give us y . We call this the true function.

Definition 1.7: True Function

The **true function** in regression is the function we assume exists in the real world that maps out the input to the output. It is denoted as f .

Finally, remember that in this equation, x will always be an independent variable, or an input, and y will always be a dependent variable, the output, or the final thing we are trying to predict.

1.4.3 Defining the Model

After assuming that true function f exists, we now try to define a model that can be as close as possible to f since f is unknown. The model that we define ourselves to match close enough with f is denoted as \hat{f} .

Consider the plot in Figure 1.1.3.

Suppose the true function f is the polynomial function plotted above, because for whatever reason, house prices correlate with square footage in that exact pattern in the real world. We can eye this plot and intuitively draw a curve ourselves that could summarize the data pretty well. However, if our input was more than one dimension, this could be very difficult to do and a regression algorithm can determine a better fitting function in any case. A regression algorithm will thus conduct a series of computations to determine which \hat{f} best fits the data. We will discuss how regression algorithms determine what is "best" in the next section.

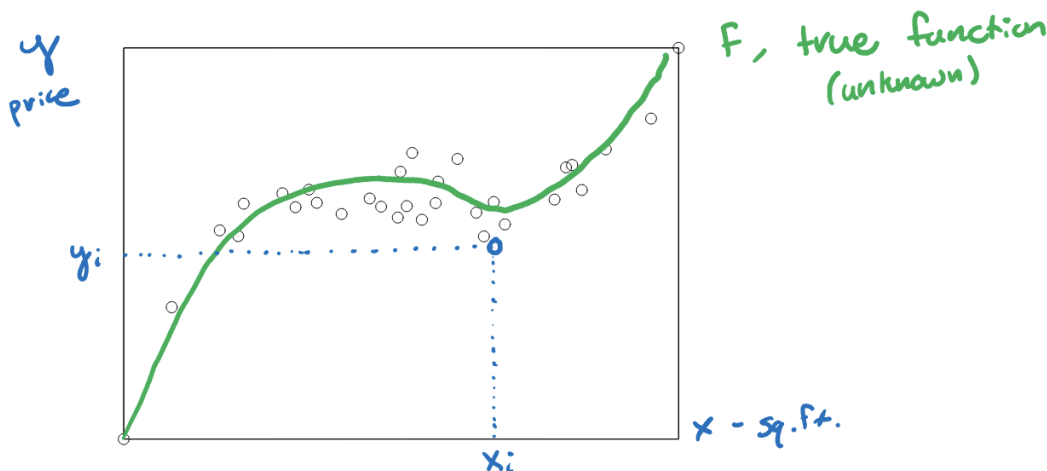


Figure 1.3: The dataset is plotted on a graph where the x-axis is square footages and the y-axis is house prices.

1.5 Linear Regression

Though the learning process for all types of regression is more or less the same, we will start with linear regression for simplicity's sake. If we are doing linear regression, we assert that the relationship between x and y is linear:

$$y^i \approx f(x^i) = wx^i + b \quad (1.4)$$

Hence our \hat{f} will also be linear:

$$\hat{y}^i = f(\hat{x}^i) = \hat{w}x^i + \hat{b} \quad (1.5)$$

Note here the variables that have hats above them. Only x^i doesn't have a hat above it because it is the true input. But the other components of our model \hat{w} and \hat{b} are mere estimates of the true w and b , and the final output \hat{y}^i is a prediction, not the given y -value from our dataset.

Now, our goal is to find the most accurate values for \hat{w} and \hat{b} .

The big picture outline for the regression algorithm is as follows:

1. Start with a random line to fit the data. This means try out a random value for \hat{w} and \hat{b} .
2. Calculate the error from this line to know how bad it is.
3. Try another random line.
4. Calculate the error again. If this line is worse than the one we just tried, try to go back closer to your older values for \hat{w} and \hat{b} . If it's better, just keep going in this direction!
5. Repeat until your error can't get any smaller.

There are many ways to calculate error but the most popular metric and the one we will use for regression is Mean-Squared Error (MSE).

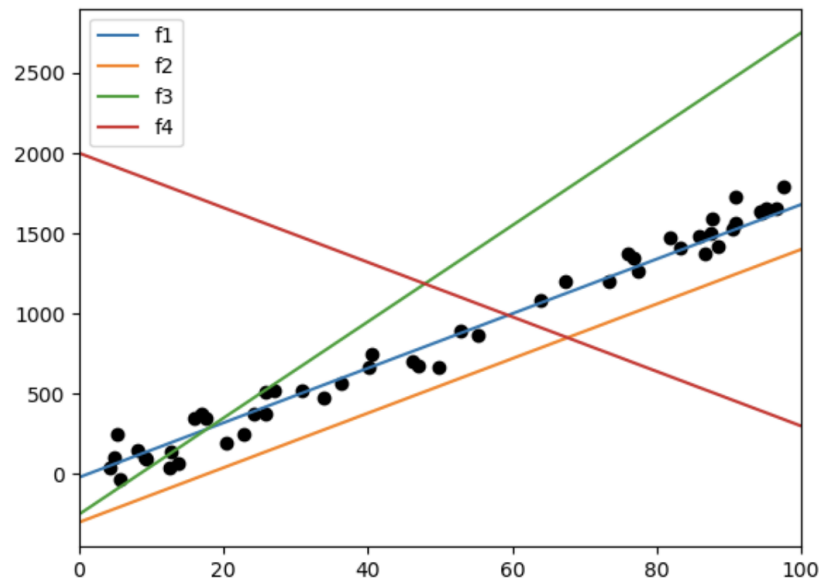


Figure 1.4: Regression starts with a random line and then tries to gradually make it better and better based on error calculation.

Definition 1.8: Mean-Squared Error

Mean-Squared Error (MSE) is a type of error calculation. To calculate MSE you sum up all the differences between your true y values from each x and your predicted \hat{y} values from each x and square them. Then, you divide this number by your dataset size n :

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^i - \hat{y}^i)^2 \quad (1.6)$$

A smaller MSE indicates a smaller loss which indicates a better model \hat{f} . So, our goal is to *minimize* MSE. Since in the MSE equation the changing values are w and b , we can define MSE as a function with two parameters. Mathematically, we solve for:

$$\hat{w}, \hat{b} = \operatorname{argmin} MSE(w, b) \quad (1.7)$$

We minimize MSE to get the best \hat{f} through a process called Gradient Descent.

Definition 1.9: Gradient Descent

Gradient Descent is an optimization algorithm aimed at finding a local minimum in an error graph. It uses calculus to utilize the gradient (slope) of a given error.

Gradient Descent uses the slope of an error graph to determine which direction signals a higher error versus a lower error. You can think of it like rolling a ball down a hill.

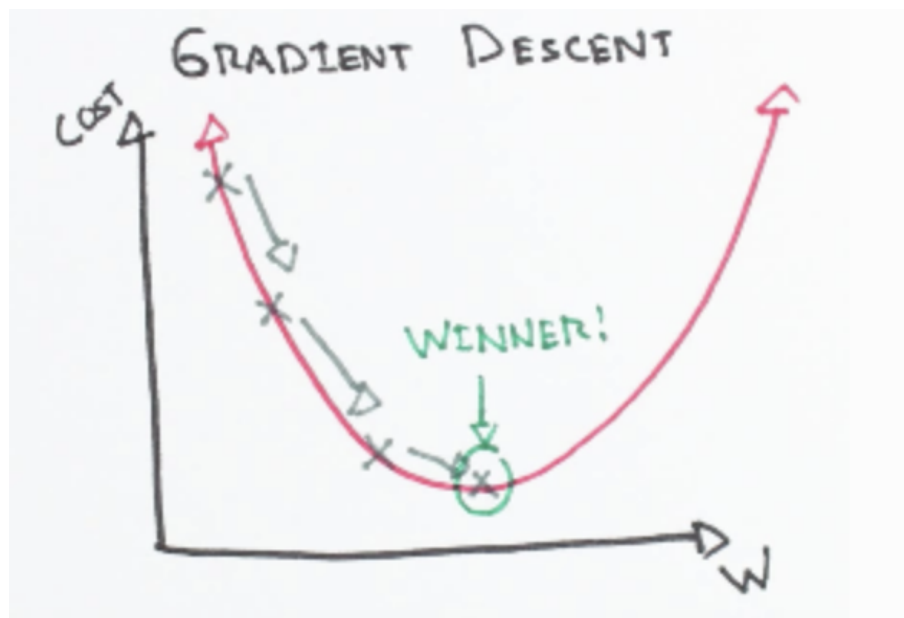


Figure 1.5: You can think of Gradient Descent as rolling a ball down a hill.

Chapter 2

Assessing Performance; Bias + Variance Tradeoff

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

2.1 Linear Regression Model

Let us recap what we have learned from Chapter 1 about linear regression. With linear regression we have many these many parts at play:

1. **Data set:** $\{(X_i, y_i)\}_{i=1}^n$ where $X_i \in R^d, y \in R$
2. **Feature Extraction:** $h(X_i) = (h_0(X_i), h_1(X_i), \dots, h_D(X_i))$
3. **Linear Regression Model:** $y = w^T h(X_i)$
4. **Quality Metric / Loss function:** $\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
5. **Predictor:** $\hat{w} = \underset{w}{\text{argmin}} \text{MSE}(w)$
6. **Optimization Algorithm:** Optimized using Gradient Descent
7. **Prediction:** $\hat{y}_i = \hat{w}^T h(X_i)$

From our examples we have seen that a linear regression can be quite useful for predicting linear data with high accuracy however, a linear regression has its limitations in that it can only predict linear data well. For instance, take a look at some sample data below.

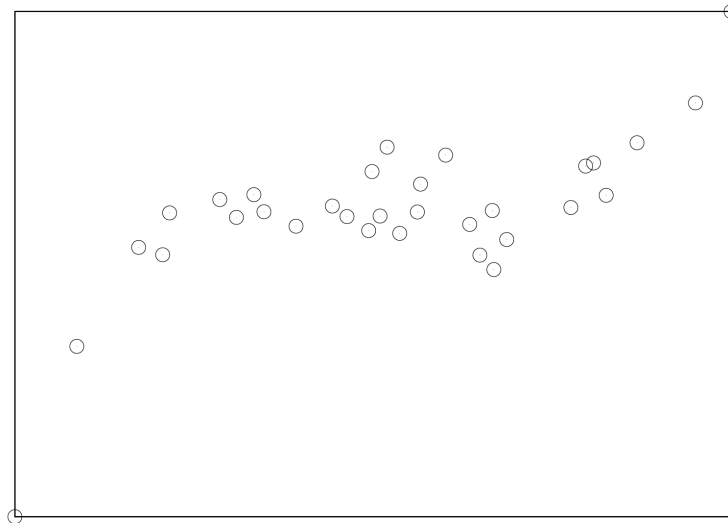


Figure 2.6: Nonlinear sample data.

We could attempt to fit a line to this data using the methods we have discussed but such a model would never truly be able to represent the data. To better predict data like this we will use a similar model using more features called polynomial regression.

2.2 Polynomial Regression

From the sample data shown earlier, we see that a linear model would not fit the data well. Polynomial regression is extremely similar to linear regression, which is represented using one input and a trained weight value:

$$y = wx + b \quad (2.8)$$

Now, with polynomial regression we take more input features (weights) to approximate any degree polynomial of our choosing to better fit polynomial like data.

Definition 2.1: Polynomial Regression Model

$$y = w_0 + w_1x + w_2x^2 + \dots + w_dx^d$$

Now that we have the option to pick the degree of our model we have encountered our first hyperparameter.

Definition 2.2: Hyperparameter

A **hyperparameter** is a parameter whose value is used to control the learning process, different from the values of other parameters or weights which are derived from training.

Looking again at the sample data from before, the data looks cubic so we would likely choose the model's degree hyperparameter to be 3. However, data does not often look this clean or is easy to visualize, so choosing a polynomial complexity can be difficult. Let's imagine we pick different model complexities for new data and see how they fit.

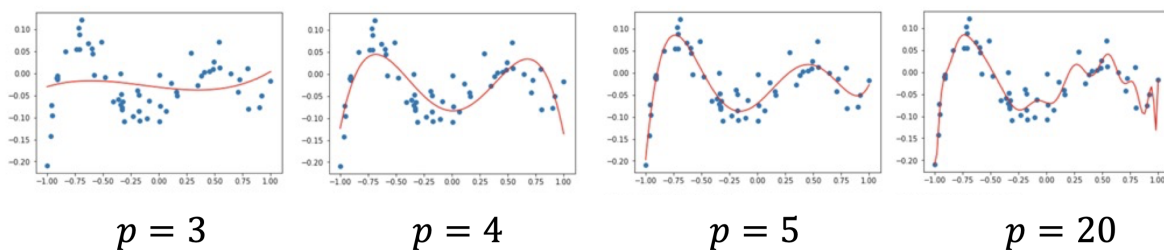


Figure 2.7: Fitting different complexity models to data.

If we choose a linear regression model (degree = 1) or any other model complexity that is too low, our model will perform poorly no matter how well our model is trained (see above). This idea can be referred to as having a model that **underfits** the data. However, it is also possible to choose a model complexity that is too high (see above $p = 5, 20$) which we call **overfitting**. This introduces the balancing act of machine learning: variance and bias.

2.3 Bias and Variance

Definition 2.3: Bias

Bias is the difference between the average prediction of our model and the expected value which we are trying to predict.

Definition 2.4: Variance

Variance is the variability in the model prediction, meaning how much the predictions will change if a different training data set is used.

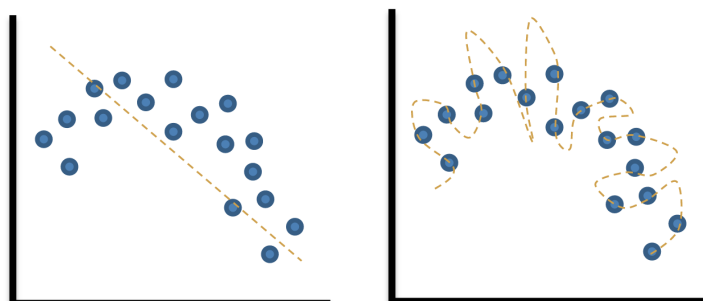


Figure 2.8: Often low complexity or simple (on the left) models tend to have high bias and high complexity models (on the right) tend to have high variance.

Variance and bias form the two main sources of error in a model that we work to control, but there is also irreducible error, which is the error we can't avoid or possibly eliminate.

Definition 2.5: Irreducible Error

Irreducible error is unavoidable error caused by elements outside of our control, such as noise from observations.

In general for ML models, simple models have high bias while overly complex models have low bias but high variance. We can identify our total error as this:

$$\text{Error} = \text{Biased squared} + \text{Variance} + \text{Irreducible Error} \quad (2.9)$$

Or more formally as:

$$E[(y - \hat{f}(x))^2] = \text{bias}[\hat{f}(x)]^2 + \text{var}[\hat{f}(x)] + \sigma_\epsilon^2 \quad (2.10)$$

When it comes to best fitting a model, choosing hyperparameters like the degree in a polynomial regression can have a major impact on the results of the model. In almost all real world data there is some noise. The goal for creating a model is to be able to make a prediction about the given training data while also generalizing well enough to predict the test data accurately. In Figure 1.1.3, we see that the simple linear regression on the left is not fitting the data well and will then have poor training and test set performance. The model on the right fits the training set with zero training error...but do we have a perfect model? The

answer is no, this model is overfitting our training data, meaning that once we give the model more data it will likely predict the new data with low accuracy as the model snakes around to hit every training point perfectly. While it is our goal in ML to reduce error from the model, 0 training error is often not ideal. The high complexity model in Figure 1.1.3 has high variance, relating to its definition, because it fits the training data well but will perform poorly on new test set data, showing the differing results from one set to another.

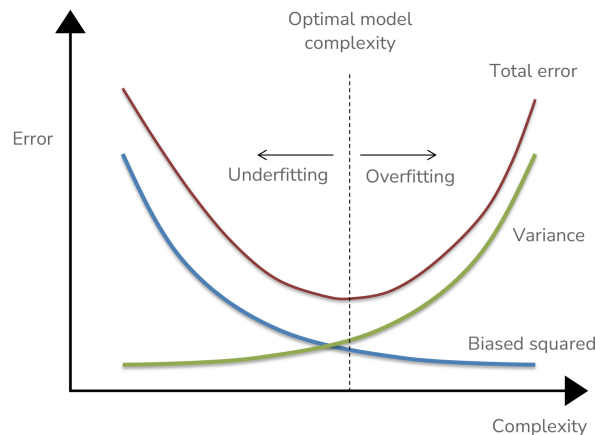


Figure 2.9: Bias and variance tradeoff

Our goal is to find a model that fits the data most accurately. This middle ground often lies where there is a good balance between bias and variance, as shown in Figure 1.1.4. This means that we are truly understanding the main function at play and not the noise within the data set.

Another factor to consider is the amount of data. With a low volume of training data, being able to generalize well is difficult. Imagine trying to fit a potentially highly complex model to only 10 data points. This would make understanding the true relationship difficult. With more data, our model will inherently begin to generalize better as outliers in the data have a decreasing impact on the overall model. This is because increasing the number of data points to consider decreases the values placed on each data point individually.

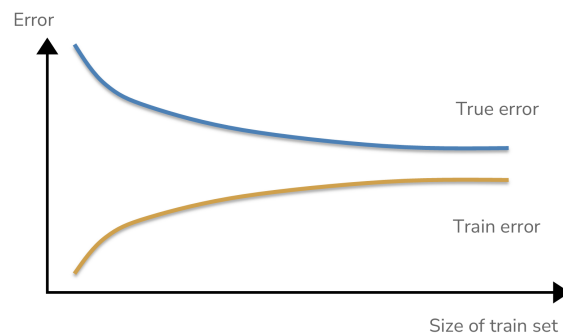


Figure 2.10: Error versus the size of the training

The figure above represents how the decreased weight on individual points can help us generalize better and decrease the model's true error. Because the model is generalizing better (reducing variance), train error slightly increases as it converges towards the decreasing true error.

In summary for choosing complexity we have learned that:

1. Choosing hyperparameters are essential for a well fitting model.
2. Overfitting and Underfitting
3. Simple models tend to have high bias and low variance and complex models tend to have low bias and high variance.
4. Choosing a model which has 0 training error is often not a good idea due to its high variance.
5. Error as a function of train set size

From these ideas it is easy to see that picking the model which has the best performance for unseen test data set is the best indicator of true error. By choosing a model based on test error we will minimize variance as we compare different model complexities. However, this situation would create a biased approach. Our goal for a test set is to have a pure, "untouched" representation for our model. If we chose the best model based solely on test data performance we are choosing a model for that set. The test set would no longer represent "the unknown" since we probed it many times to figure out which model would be best.

When choosing what type of model will perform the best, we must perform another split in our data. Between a training and test set there is usually an 80/20 split, however this can vary depending on data set sizes. To now choose a model's hyperparameters without biasing towards the test set, we must create a third split of the data called a **validation set**. This will allow us to make decisions about the model architecture and complexity without ever touching the final testing set. We will learn more about validation sets in the following chapter.

Chapter 3. Cross Validation / Regularization

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

As we have discussed in the previous chapter, we must assess different model complexities. The metric used to determine the best model complexity cannot be the training error, as it will favor the model that overfits the training data, not the model that will perform the best in the future. However, we cannot use the test error either, because we cannot tamper with the test set until the very end. Otherwise, our test set no longer represents the "unknown".

3.1 The Validation Set

Since we cannot use the data that we trained on, nor the data that should be an unbiased representation of the future, we must designate a new set to determine the best model complexity that is neither trained on, nor promised to be an unbiased representation of future data. This will be the **validation set**.

Definition 3.1: Validation Set

The **validation set** is the set of the data used to optimize and find the best model for a task.

Validation data is not used to train on directly, but it is used at the end of each training round to calculate an error based on unseen data. This is different from the test set because the validation error is what we use to decide if training is worth continuing for a particular model, and how successful this model complexity is compared to the others.

Observe the process captured in pseudocode below:

```
train, validation, test = random_split(dataset)
for each model complexity p:
    model = train_model(model_p, train)
    val_err = error(model_p, validation)
    keep track of p with smallest val_err
return best p + error(model_best_p, test)
```

Figure 3.11: Pseudocode for validation set usage.

The total dataset is split into train, validation, and test sets. For each model complexity that we want to assess, we train a model of that complexity and calculate the validation error at the end of training. We only keep track of the lowest validation error because the model with the lowest validation error is the one that is most likely to be successful on future unseen data. This is how we define "best".

The pros of having a validation set are that it is easy to implement and it is relatively fast. Validation sets are used in Deep Learning for this reason. The cons of having a validation set are even though you don't

train on validation data, you can still overfit to validation data since you keep tailoring your model to be successful on the validation set. Also, you have to sacrifice even more of your training data:

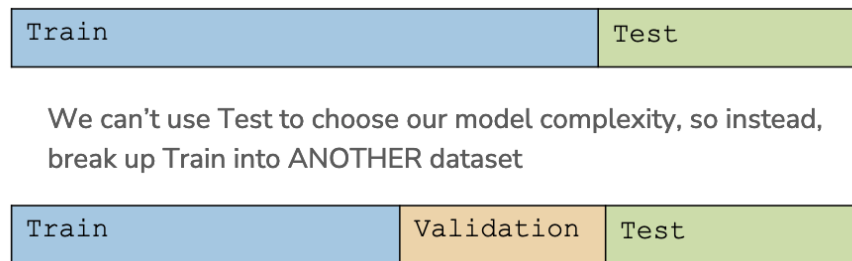


Figure 3.12: The validation set is taken from the training set, reducing the size of the training set.

3.2 Cross Validation

A clever idea to overcome the cons listed above is a technique called **cross validation**. Cross validation shuffles different parts of the training data for training versus validating.

Definition 3.2: Cross Validation

Cross Validation is a technique of resampling different portions of training data for validation on different iterations.

Here is the breakdown of the Cross Validation process:

1. First, the training set must be split into k number of chunks.

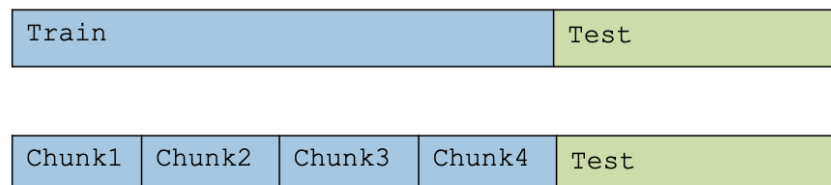


Figure 3.13: The training set is divided into a number of chunks.

2. One of these chunks is designated as a validation set for one iteration of training.
3. After one iteration of training, validation error is calculated on the single validation set chunk.
4. This is done k times, once for each chunk being the validation set.
5. All of the errors from each iteration are averaged. This is the final validation error for a model complexity.
6. Repeat steps 1-5 for every model complexity you are assessing. The model complexity with the lowest average validation error is the best model.

Below we have provided pseudocode for the steps above:



14

Figure 3.14: The validation error is calculated by averaging the error from k different samples of validation data.

```

train_set, test_set = random_split(dataset)
randomly shuffle train_set
choose a specific k and split it into k groups
for each model complexity p:
    for i in [1, k]:
        model = train_model(model_p, chunks - i)
        val_err = error(model, chunk_i)
        avg_val_err = average val_err over chunks
    keep track of p with smallest avg_val_err
retrain the model with best complexity p on the
full training set
return the error of that model on the test set

```

Figure 3.15: Cross Validation pseudocode.

The pros of this method is that you don't get rid of any of your training data. Also, you prevent overfitting because you train on a unique subset of the train set for each iteration, instead of the same training set for all iterations. The cons of this method is that it is very slow, because you have to train each model k many times. This makes training extremely computationally expensive. Your choice of k , how many chunks you split your train set into, also impacts your bias-variance trade-off: A lower k means you will have high bias while a higher k means you will have high variance. In practice, developers will use $k = 10$ because it

achieves a fine medium balance.

Now, we will look at other ways to prevent overfitting.

There are two other main culprits behind overfitting: using too many features, and placing too much importance on specific features that are not that important. In other words, if the weight of a feature is very large, this could be a sign of overfitting. To overcome this, the model must "self-regulate" when its weights become too big. This process is called **regularization**.

3.3 Regularization

Definition 3.3: Regularization

Regularization is a technique used by a model to make sure it maintains balanced weights for its features to prevent overfitting.

Up until this point, we have always estimated our weights \hat{w} by using a quality metric that minimizes the loss:

$$\hat{w} = \underset{w}{\operatorname{argmin}} L(w)$$

However, we have now learned that minimizing the loss for the training set might make our weights overfit the training set, and not be as ideal for future sets. So, we introduce a new term to our quality metric:

$$\hat{w} = \underset{w}{\operatorname{argmin}} L(w) + \lambda R(w)$$

where $R(w)$ is the magnitude of the weights and λ is our **regularization parameter**. We will thus account for our weights inside the quality metric. To account for the magnitude of all weights both negative and positive, we can either take the sum of all absolute values, or the sum of squares.

3.3.1 Ridge Regression

Definition 3.4: Ridge Regression

Ridge Regression is a regularization method that uses the sum of the squares of the weights in a model like so:

$$R(w) = |w_0|^2 + |w_1|^2 + \dots + |w_d|^2 = \|w\|_2^2 \quad (3.11)$$

The notation for Ridge Regression is $\|w\|_2^2$ because it uses an L2-norm, which is a type of norm. The notation for any norm uses the double bars and the subscript is the degree we raise each term to. Unlike a 2-norm we aren't raising the final sum to the 1/2 power, so the norm is squared, hence it has the power of 2.

As Ridge Regression utilizes the sum of the square of our weights, our new quality metric becomes:

$$\hat{w} = \underset{w}{\operatorname{argmin}} L(w) + \lambda \|w\|_2^2$$

The regularization parameter, λ is a hyperparameter that determines how important the regularization term is in our quality metric.

if $\lambda = 0$:

$$\Rightarrow \hat{w} = \underset{w}{\operatorname{argmin}} L(w)$$

if $\lambda = \infty$:

\Rightarrow Case 1: if $\|w\|_2^2 = 0$ then $\hat{w} = \text{argmin } L(w) + 0$

\Rightarrow Case 2: if $\|w\|_2^2 > 0$ then $\hat{w} = \text{argmin } L(w) + \lambda\|w\|_2^2 \geq \lambda\|w\|_2^2 = \infty$. Since We are trying to minimize $L(w)$, $\min L(w) = 0$ so $\hat{w} = 0$.

if λ is in between 0 and ∞ :

$\Rightarrow 0 \leq \|w\|_2^2 \leq \text{argmin } L(w)$

3.3.2 Coefficient Paths

Observe the change in size of the coefficients of the weights in our model as we increase λ . As λ increases, all weights get closer and closer to 0. This is what we mean by "regularizing" the weights.

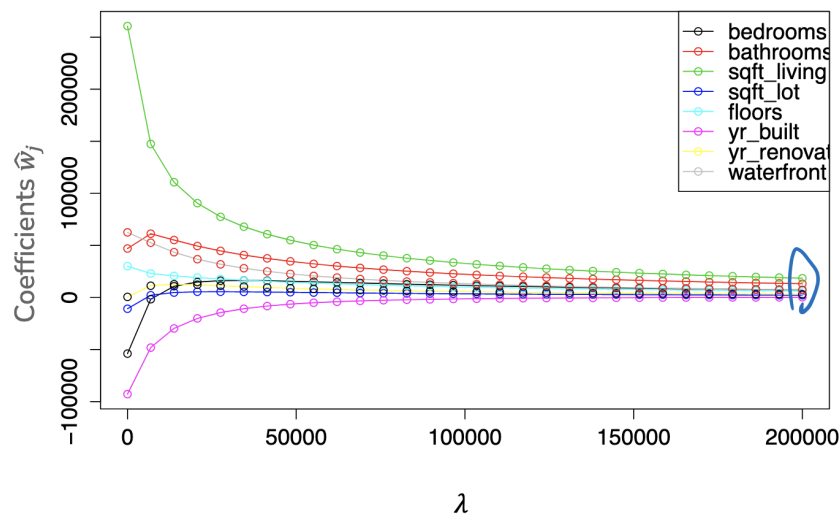


Figure 3.16: The effect of λ on the size of the weights.

3.3.3 The Intercept

When regularizing all the weights, we also happen to regularize w_0 , the intercept term. This doesn't make sense because we shouldn't penalize a model for having a higher intercept since that may mean that the y value units are just really high. The intercept also doesn't affect the curvature of the a loss function. We deal with this by changing the measure of overfitting to not include the intercept:

$$\hat{w} =_{(w_0, w_{rest})} L(w_0, w_{rest}) + \lambda \|w_{rest}\|_2^2$$

3.3.4 Scaling Features

Suppose we have a weight \hat{w}_1 for a feature, *house square footage*. What would happen if we changed the units of that feature to miles? Would \hat{w}_1 need to change? Since we change the input to be a of a larger scaled unit, the input values for this feature would be smaller. However, if the values are now smaller this means that we are giving the feature less importance in our model. The solution to this problem is **normalizing** all the features such that they are all on the same scale:

$$\tilde{h}_j(x^{(i)}) = \frac{h_j(x^{(i)}) - \mu_j(x^{(1)}, \dots, x^{(N)})}{\sigma_j(x^{(1)}, \dots, x^{(N)})}$$

Where

The mean of feature j :

$$\mu_j(x^{(1)}, \dots, x^{(N)}) = \frac{1}{N} \sum_{i=1}^N h_j(x^{(i)})$$

The standard deviation of feature j :

$$\sigma_j(x^{(1)}, \dots, x^{(N)}) = \sqrt{\frac{1}{N} \sum_{i=1}^N (h_j(x^{(i)}) - \mu_j(x^{(1)}, \dots, x^{(N)}))^2}$$

Figure 3.17: We must apply this scaling to the test data and all future data using the means and standard deviations of the training set, otherwise the units of the model and the units of the data are not compatible.

Chapter 4

Feature Selection and LASSO

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

4.1 Ridge Regression Recap

For ridge regression we use a standard MSE loss with an L2 norm regularizer.

$$\hat{w} = \underset{w}{\operatorname{argmin}} \operatorname{MSE}(W) + \lambda \|w\|_2^2 \quad (4.12)$$

The hyperparameter λ can play a large role in how a model behaves. For instance, if $\lambda = 0$ we would then have a standard regression model with no regularization. On the opposite side of the spectrum, if $\lambda = \infty$ then any feature usage at all (any weights greater than 0) would penalize the model completely, leaving us with a model that can only output 0. Clearly, if the goal is to create a useful model that uses regularization neither of those two options are ideal. Instead, we will use something in between.

When choosing λ if it is too small then the model will overfit to the training data. As in previous chapters, we must evaluate our hyperparameter on the validation set. We should typically pick the model that best performs on the validation set (lowest error) using MSE only. Why MSE only when we have a regularized objective function already? The reason is that different λ , or any hyperparameter for that matter, mean we are evaluating the model's performance in different contexts or "units". We could compare the results of any number of models using the same objective function, however when the hyperparameters are different we cannot compare them.

For reference, here is an example process for choosing λ for ridge regression.

Algorithm 1 Process for selecting λ for ridge regression

for λ in λ s **do** Train a model using Gradient Descent

– $\hat{w}_{\text{ridge}(\lambda)} = \underset{w}{\operatorname{argmin}} \operatorname{MSE}(W) + \lambda \|w\|_2^2$

Compute validation error

– $\text{validation_error} = \operatorname{MSE}_{\text{val}}(\hat{w}_{\text{ridge}(\lambda)})$

Track λ with smallest *validation_error* **return** λ^* and estimated future error $\operatorname{MSE}_{\text{val}}(\hat{w}_{\text{ridge}(\lambda^*)})$

We have seen how regularization can be useful with ridge regression to avoid overfitting by adding a level of smoothness to our model. However there are many different ways to regularize and one way we will investigate is called lasso, useful for selecting features.

4.2 Feature Selection

Feature selection sounds fun but why should we care? There are three major reasons why feature selection is useful in machine learning.

1. **Complexity** - With too many features our model could be overly complex and overfit.
2. **Interpretability** - With less features, or selected feature, we can better understand our model and see which features carry more information
3. **Efficiency** - With less features to consider, our model could be trained much easier and faster

Let's introduce a scenario where we want to create a good model for predicting the price of a house. We have the opportunity to collect a wide variety of features, however we are unsure of what features are best to be used.

Example(s)

3

- Lot size
- Single Family
- Year built
- Last sold price
- Last sale price/sqft
- Finished sqft
- Unfinished sqft
- Finished basement sqft
- Number of floors
- Flooring types
- Parking type
- Parking amount
- Cooling
- Heating
- Exterior materials
- Roof type
- Structure style
- Dishwasher
- Garbage disposal
- Microwave
- Range / Oven
- Refrigerator
- Washer
- Dryer
- Laundry location
- Heating type
- Jetted Tub
- Deck
- Fenced Yard
- Lawn
- Garden
- Sprinkler System

You or I could pick out that the features "lot size" and "last sold price" (among others) are likely good indicators of a house's sale price and that the features "microwave" and "garbage disposal" could have an effect, but are not likely to be key factors for predicting a house sale price. This is a key example of feature selection.

By selecting a smaller number of features, this model will be more interpretable as it is easier to determine that "lot size" is one of the most important features for predicting house price. With less features, our model can be trained more easily. In addition, if the data must be gathered ourselves, gathering less data points is simply easier to do.

Although it could be helpful to use all the information available to make a model that can predict well, a large amount of features can be prohibitively expensive to train. Also, if features are used which are

unimportant it could lead to the model overfitting the data.

Example(s)

Let us imagine it takes us 5 seconds to train a model with 8 features:

- Training 16 features would take 21 minutes.
- Training 32 would take about 3 years.
- Training 100 features would take $7.5 * 10^{20}$ or about 50,000,000,000 times longer than the age of the universe.

Seems like it might not be worth considering if the "sprinkler system" feature is included with the house in our model. One method to feature select would be to try many small sets of features and see how the model performs manually. However, this can be tedious and without trying every set of hyperparameters we would not be certain that we have chosen the optimal features. To aid in feature selection we can use another linear regression model with a new type of regularizer called lasso.

4.3 LASSO

We can create a regression using all of these features and add a regularizer in the hope that we achieve a sparse weight vector which still performs well on the data. A sparse matrix / vector is one that is filled with mostly 0. This would help us in feature selection as the unimportant features would have weight 0, meaning they are not being considered.

Our format for a model with regularization is as such where $L(w)$ is the measure of fit and $R(w)$ measures the magnitude of coefficients.

$$\hat{w} = \underset{w}{\operatorname{argmin}} L(w) + R(w) \quad (4.13)$$

There are many different ways to create a regularizer. One that works well is the sum of squares which is what we have seen in ridge regression (L2 norm).

Another possibility is simply the sum of the weights. This however would not be ideal as negative and positive weights could cancel one another out. To fix this we could sum the absolute value of the weights, which is exactly what the lasso model does. This is also known as the L1 norm.

Definition 4.1: Lasso Regression Model

$$\hat{w} = \underset{w}{\operatorname{argmin}} \operatorname{MSE}(W) + \lambda ||w||_1$$

Remember the definition of a p-norm.

Definition 4.2: p-norm

$$||w||_p^p = |w_0|^p + |w_1|^p + \dots + |w_d|^p$$

Without lasso, we could have tried different features and computed the resulting loss, but now we start with a full model and then shrink coefficients towards 0. With ridge regression many of our coefficients as the model is trained will descend towards 0, but not reach 0. Why? It has to do with the shape of the norm. Ridge introduces a smoothness to prevent overfit by limiting the amount that one single feature can use.

In general, the coefficients are pushed towards 0, but never actually reach it despite training for thousands of epochs. In lasso, the L1 norm favors sparsity and produces a coefficient path where coefficients go towards and actually remain at 0. Take a look at some coefficient paths for ridge regression below.

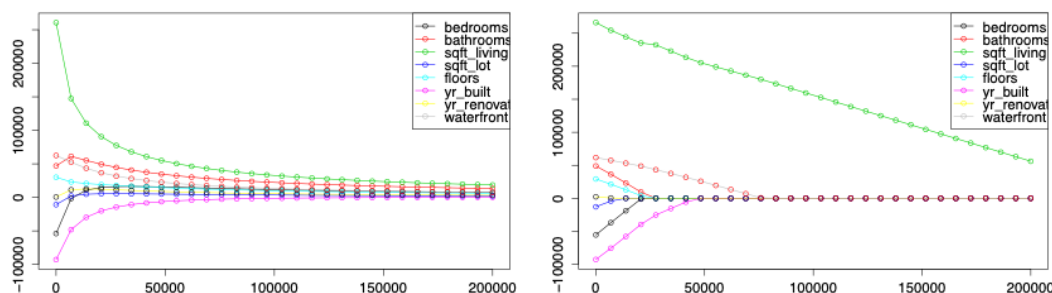


Figure 4.18: Ridge (right) and lasso (left) coefficient paths.

One way to think about this geometrically is that the L1 norm has a "spikey" solution. In a L1 norm, as coefficients are close to 0, the only way to get close to 0 is actually being 0. In a L2 norm, coefficients can be very small and not quite reach 0.

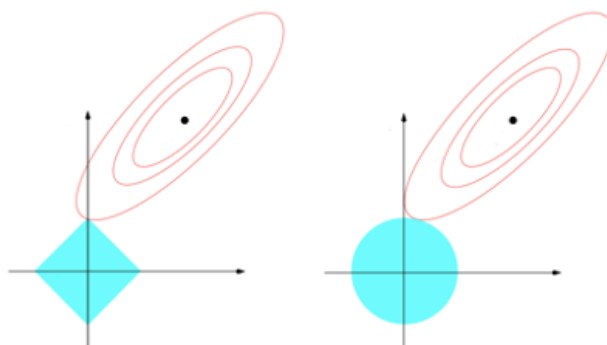


Figure 4.19: Visualizing sparse solutions with L1 on left and L2 on right.

In Figure 2.0.2, L1 has spikes representing the places where the coefficients are 0. When the unregularized coefficient paths (represented in red) touch the regularized solution in either graph we see that the diamond shape will likely touch at one of its corners where a coefficient is 0 whereas L2 would be much more likely to touch equally and at any location on the circle.

When it comes to choosing a λ for lasso, the choice is done exactly the same as what we have learned for ridge regression.

However, there are caveats of using lasso. As with any regularizer, we are adding bias to our least squares solution (remember bias + variance trade off). One way to remove some bias, but still benefit from having a sparse solution would be to first run lasso, then extract the non-zero features and run those features on a unregularized least squares so that coefficients are no longer shrunk from their possible "truth" values as the model is trained.

As with all types of machine learning it is important to remember that correlation does not equal causation. Lasso will do exactly what you ask of it and nothing more. Lasso will find the features that are important for predicting some output well. However, it is important to always think before and after whether the solutions are representative of the type of data you are training on.

Example(s)

There is a publicly available Communities and Crime Data Set which has a large number of features which could be trained on to predict where crime will take place based on the demographics of that location like its population density, the age of people living there, etc.

If you were to train a lasso model on this data set you would likely see that areas where there is a high proportion of people aged 65 and up usually have low crime and lasso will likely select this as a good predictor (non-zero coefficient). This is an example of correlation and not causation. We asked our model to find good predictors of crime rate, however clearly it would not be logical to attempt to lower high crime areas by moving lots of senior citizens there. In complex social issues, among other data sets, remembering to think about what your model is doing vs what you would like it to do is important.

In practice, lasso can run into some issues. Lasso tends to pick arbitrarily from correlated features, like number of bathrooms and number of showers from our house price example. In this case it might make more sense to choose bathrooms or to select them together, however lasso does not offer this guarantee. In addition, lasso tends to have worse performance in practice compare to ridge regression because of the bias introduced by pushing feature coefficients towards 0.

A solution to this problem where we want feature selection and the performance from ridge is a model called elastic net which is simply the combination of a L1 norm and a L2 norm regularizer.

Definition 4.3: Elastic Net

$$\hat{w}_{\text{Elastic Net}} = \underset{w}{\operatorname{argmin}} \operatorname{MSE}(W) + \lambda_1 \|w\|_1 + \lambda_2 \|w\|_2^2$$

A combination of both L1 (Lasso) and L2 (Ridge) regularizers.

To review we learned that lasso:

- Introduces more sparsity to the model.
- Is helpful for feature selection.
- Is less sensitive to outliers.
- Is more computationally efficient as a model due to the sparse solutions.

And ridge regression:

- Pushes weights towards 0, but not actually 0.
- Is more sensitive to outliers (due to the squared terms).
- In practice, usually performs better.

This concludes the chapter's focus on regression and next we will learn about different models for different uses like classification.

Chapter 5

Classification

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

So far, we have talked about Regression analysis for continuous output values such as house prices, where a house price can be any number in a range. We will now transition to the next application of machine learning, Classification. Now, our output values are no longer continuous, rather they are distinct *classes*.

Definition 5.1: Classification

Classification is an application of machine learning where a class label is predicted based on some input data.

Examples of classification are classifying emails as spam or not, classifying animals as cats or dogs, classifying product reviews as positive or negative. Notice that in all of these scenarios, our outputs are not numerical values along a range, so regression no longer applies here. This means that for our quality metric, Mean-Squared Error is also no longer a viable option because we do not have predicted numerical values.

We will examine classification through a restaurant review example. Suppose you have a collection of different restaurant reviews and you want to classify them as positive or negative. You would like to extract the sentences from the review, feed them as input to a classifier model, and let the model tell you if the review is good or bad. How would you go about this?

5.1 Simple Threshold Classifier - Implementation 1

The first way we could solve this problem is by using a Simple Threshold Classifier. In this type of classifier, we do **Sentiment Analysis**: We will curate a list of positive and negative words, and then classify each of the reviews by the frequency of positive words and the frequency of negative words. Our lists may look something like this:

Positive: "great", "awesome", "good", "amazing"

Negative: "bad", "terrible", "disgusting", "sucks"

If a sentence has more positive words than negative words, we will assign it the output value $y = +1$, to indicate that this is a *positive review*. Otherwise, we will assign it the output value $y = -1$, to indicate that this is a *negative review*. For example, if our review was "Sushi was great, food was awesome, but the service was terrible", we count two positive words "great" and "awesome", and one negative word "terrible". Since there are more positive words than negative words, we assign this review the output $+1$, and classify it as a positive review.

There are unfortunately a number of limitations with this approach. First off, how do we get the list of positive and negative words? If we want a robust list covering the entire dataset, do we have to go through each review ourselves and painstakingly write this list up from scratch? And also, what about words that have varying degrees of sentiment? Would you say "awesome" has a higher sentiment than "good"? And what about the adverb "not"? If we say "not good" we mean negative sentiment, but "good" was on the list of positive words. As you can see, the Simple Threshold Classifier may not suffice for our problem, which calls the need for a better approach.

5.2 Linear Classifier - Implementation 2

We will try to account for the issue of varying sentiment in a Linear Classifier approach. Unlike the Simple Threshold Classifier, the Linear Classifier does not require us to write a list of positive and negative words ourselves. Instead, it will *learn*, using labeled training data, the weights of different words. These weights will be used to score a sentence. Suppose we have trained a model that takes in reviews x and their ratings y , and uses the words in each as features to predict the review's sentiment \hat{y} . Let the following be the learned weights for each word:

Word	Weight
good	1.0
great	1.5
awesome	2.7
bad	-1.0
terrible	-2.1
awful	-3.3
restaurant, the, we, where, ...	0.0
...	...

Figure 5.20: Learned weights representing sentiments for words found in the restaurant reviews.

Now let's see how these weights play out on our example:

"Sushi was great, food was awesome, but the service was terrible"

We will calculate the sentiment score of this input using the frequencies of each word:

$$\begin{aligned}
 \text{Score}(x) &= \text{"awesome"} * 1 + \text{"great"} * 1 + \text{"terrible"} * 1 \\
 \text{Score}(x) &= 2.7 * 1 + 1.5 * 1 + -2.1 * 1 \\
 \text{Score}(x) &= +2.1
 \end{aligned}$$

As the score here is positive, we decide this review is positive. We call this classification method "Linear" because our output is a linear weighted sum of inputs. Note here that our threshold is 0, and in the unlikely event we achieved a score of exactly 0, we would choose an arbitrary class for this review.

Can you think of some limitations for this approach? We still have not really resolved the case "not good" meaning not good, and "not not good" meaning good. Also, sometimes a word's sentiment relies on surrounding words. For example, saying "a bit sad" is less negative than just "sad". Thus, a linear classifier cannot learn complex models very well. Though we will not go into it much in this textbook, **Natural Language Processing (NLP)** is a field of machine learning that is used for more complex analyses of language.

$$\text{Model: } \hat{y}^{(i)} = \text{sign}(\text{Score}(x^{(i)}))$$

$$\begin{aligned} \text{Score}(x_i) &= w_0 h_0(x^{(i)}) + w_1 h_1(x^{(i)}) + \dots + w_D h_D(x^{(i)}) \\ &= \sum_{j=0}^D w_j h_j(x^{(i)}) \\ &= w^T h(x^{(i)}) \end{aligned}$$

We will also use the notation

$$\begin{aligned} \hat{s}^{(i)} &= \text{Score}(x^{(i)}) = w^T h(x^{(i)}) \\ \hat{y}^{(i)} &= \text{sign}(\hat{s}^{(i)}) \end{aligned}$$

Figure 5.21: Notation for linear classifiers.

5.3 Decision Boundaries

We can visualize the effect of the weighted words on determining if a review is positive or negative via illustrating the decision boundary. For simplicity's sake, suppose we only had two nonzero-weighted features, "awful" and "awesome". Our decision boundary would look like so:

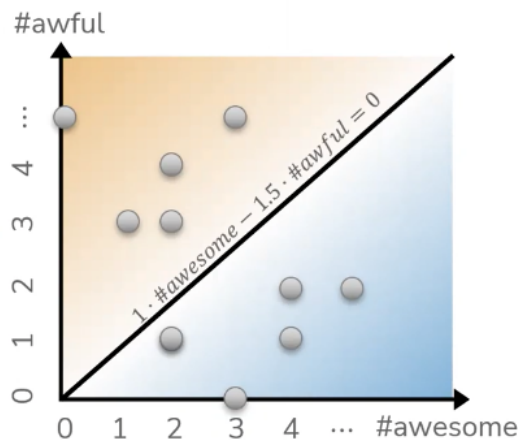


Figure 5.22: A linear decision boundary between two features.

Since we have more than two nonzero-weighted features, the decision boundary for the example would be a higher-dimensional graph with linear boundaries. If we needed a more complex, nonlinear decision boundary, then we would need a more complex model. These are examples of more complex decision boundaries:

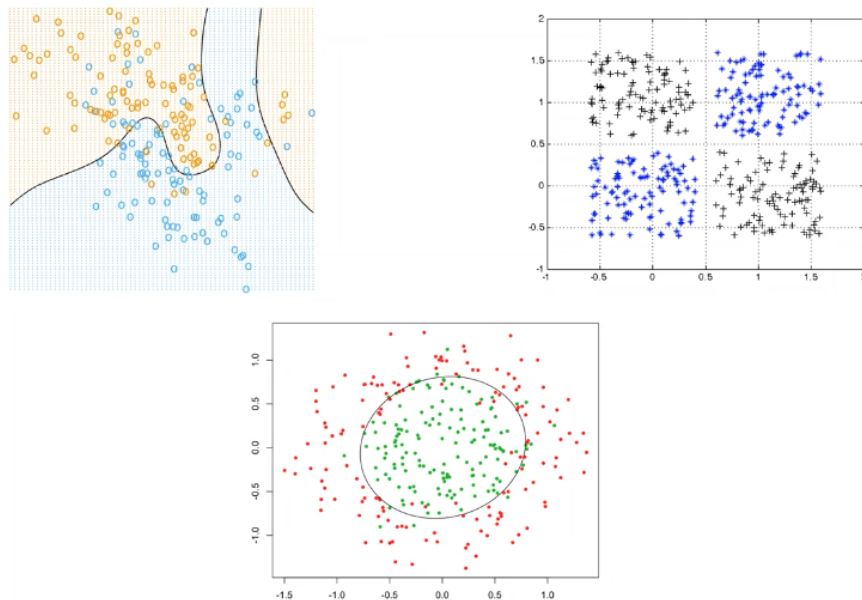


Figure 5.23: Visualization of nonlinear decision boundaries produced by more complex classification models.

5.4 Classification Error

As we mentioned earlier, Mean-Squared Error is no longer applicable to Classification, because we aren't trying to fit a predictor to some datapoints by reducing each point's error from the predictor. This is because our output values aren't continuous numerical values, they are classes. So, we calculate error in a much simpler way:

$$Error = (\text{wrong classifications})/(\text{total classifications})$$

$$Error = \frac{1}{n} \sum_{i=1}^n (y_i \neq \hat{y}_i) \quad (5.14)$$

where n is the total number of classifications.

A "wrong" classification would be if the true label y of a review was positive, but we predicted \hat{y} to be negative, or if the true label y of a review was negative, but we predicted \hat{y} to be positive. Thus, for our accuract calculation we have:

$$Accuracy = 1 - Error = (\text{correct classifications})/(\text{total classifications})$$

5.5 Classification Accuracy

When calculating the accuracy of a classification model, it is important to note that since we are outputting distinct classes, not all correct classifications are due to the model's own learning. Some are due to random chance. For example if you had two output classes that were equally likely and your model was simply randomly guessing, it could achieve 0.5 accuracy. If all classes are equally likely to be chosen, then random-chance accuracy is $1/k$ classes.

Accuracy as a metric should be taken with some precaution. If you were trying to classify emails as spam or not spam, and you made a Dummy Classifier that always outputted "spam" no matter what, you could

get 90% accuracy, simply because you might have 90% of your inbox filled with spam! We call this Dummy Classifier a Majority Class Classifier, because it classifies everything as whatever the majority is. At the bare minimum, your classification model should be higher than a Majority Class Classifier.

Evidently, if there is a class imbalance we might get a higher accuracy than random chance when this is not actually a reliable baseline to compare our model performance to. This is where a **confusion matrix** is helpful.

Definition 5.2: Confusion Matrix

A **Confusion Matrix** is a table comparing all the true positive, false positive, false negative, and true negative values from the output of a model.

		Predicted Label	
		+	-
True Label	+	True Positive (TP)	False Negative (FN)
	-	False Positive (FP)	True Negative (TN)

Figure 5.24: A confusion matrix for binary classification.

Is a false negative or a false positive worse? The answer relies on your application. If we are building a model to detect spam email, a false negative would result in an annoying spam email in your inbox, but a false positive would mean that an important email is now lost. If we are building a model to detect a disease, a false negative means that the disease goes untreated and a patient could suffer greatly while a false positive means that a healthy patient wasted some money on a treatment. Depending on the consequences of false predictions of our models, we will have to make decisions on whether or not the model is ethical. In later chapters we will discuss the ethical consequences of erroneous or discriminatory models on different demographic groups further in depth.

5.6 Learning Theory

How does adding more data affect the error rate of our models? Previously, we discussed a limitation of the phrase "not good". While this has a positive word "good" in it, ultimately it should not count positively towards the score. So for our language analysis we might look at two words at a time instead of one. We refer to one word as a "unigram" and two words as a "bigram". In general, the more data the better. However, it is interesting to note that if we parse our sentences differently, the advantage of more data changes drastically. A model that uses bigrams instead of unigrams is a more complex model, so it will do worse than a unigram model with less data (due to overfitting to the limited training set), but after a sufficient amount of data, it will start doing much better.

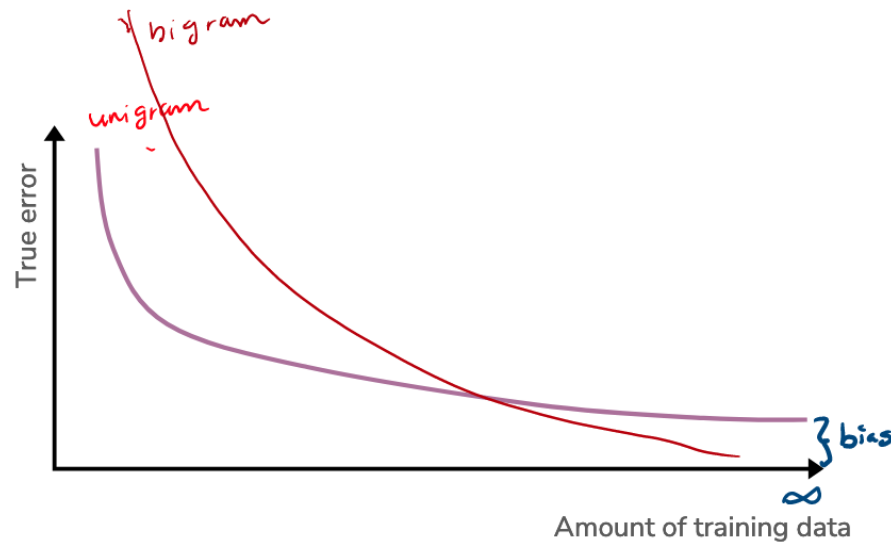


Figure 5.25: The learning curve for different data parsing types.

5.7 Changing the Threshold

If some types of error, such as false negatives or false positives are more unacceptable than the other in our context, it might be a good idea to change our scoring threshold in the decision boundary. If a doctor wants to lower the risk of making a false negative prediction for the safety of her patients, she could tune her model's threshold to be closer to negative ∞ . This way, a score less than 0 is required to output "negative", so while we increase the likelihood of false positives, at least we decrease the likelihood of false negatives, which are more dangerous in the medical world.

Conversely, if I want to lower my chances of missing out on Aritzia's latest Winter sale in my inbox, I would tune my spam-detector model's threshold to be closer to positive ∞ . I want there to be less instances of detecting spam when the email is actually important, so by increasing the score threshold, the score must be higher to be marked as spam.

Chapter 6

Logistic Regression

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

6.1 MSE / Convexity

In previous chapters we have discussed how mean squared error (MSE) can be used to create machine learning regression models which can perform quite well. However, in classification MSE is not possible because the task is not convex, continuous or differentiable. In addition, there is not a closed form solution, like in regression. This means that algorithms like gradient descent won't be able to find an optimal set of weights for the discrete results.

Definition 6.1: Convexity

A function is considered **convex** if a line segment between any two points of the function does not lie below the graph.

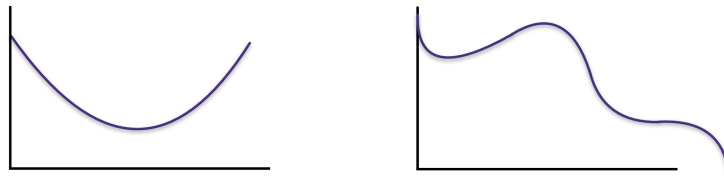


Figure 6.26: Convex (right) and non convex (left) functions.

Without a convex function, gradient descent might only be able to find a local minimum for the function or be unable to find any minimum whatsoever. Imagine taking gradient descent on the left non convex function shown in Figure 6.1. In this scenario, we would never reach a global minimum.

In classification with MSE, the three requirements of a convex, continuous and differentiable function are not met. The solutions are discrete values without any ordinal nature so we need to transform our problem into something where we can measure how close a prediction is to some category, rather than directly outputting that category from our model. One way we can do this is by measuring error using probabilities. Probabilities are continuous and can help measure the level of certainty we might have for a prediction.

Example(s)

For example, let us imagine we are creating a model to predict the sentiment of a restaurant review where a positive review maps to +1 and a negative review maps to -1. Ideally the trained model would be able to predict a review like "The sushi everything else were awesome!" positively.

$$P(y = +1 | x) = \text{"The sushi and everything else were awesome!"} = 0.99$$

Then, using probabilities for reviews that have a higher degree of uncertainty regarding the sentiment, our model could recognize that.

$$P(y = -1 | x) = \text{"The sushi was alright and service was okay."} = 0.50$$

With probabilities the certainty of an event is now known but the results must be mapped back to the classifier to make a choice. We can assign values based on if the resulting probability is greater than 0.50 to be the output. With probabilities it is easy to recognize when a model has a high degree of certainty (high probabilities). In addition, we can compare what types of inputs lead to uncertainty for the model (mid-range probabilities).

6.2 Logistic Regression Model

The **sigmoid function** takes arbitrarily large and small numbers then maps them between 0 and 1. We can input a score to this function and receive a probability so that we will be able to take gradient descent to train the model. The use of the sigmoid function in this way is called the logistic regression model.

Definition 6.2: Logistic Regression Classifier

$$P(y = +1 | x_i, w) = \text{sigmoid}(\text{Score}(x_i)) = \frac{1}{1 + e^{-w^T h(x_i)}}$$

This model allows its objective function to be trained such that $P(y = +1 | x_i, w)$ can be maximized for some input x and weights w . The goal is to maximize correct behavior / classification and minimize incorrect behavior of the model. Also, it is important to remember that in a binary classification, or for any number of classes, the probabilities must sum to 1. For binary classification $P(y = -1 | x_i, w) = 1 - P(y = +1 | x_i, w)$. We can combine these terms and use the logistic regression definition to show this:

$$P(y = -1 | x_i, w) = 1 - P(y = +1 | x_i, w) \quad (6.15)$$

$$= 1 - \frac{1}{1 + e^{-w^T h(x_i)}} \quad (6.16)$$

$$= \frac{e^{-w^T h(x_i)}}{1 + e^{-w^T h(x_i)}} \quad (6.17)$$

In order to train our model we want to maximize the likelihood function. The likelihood function in this case is:

Definition 6.3: Likelihood Function for Logistic Regression

$$\ell(w) = \prod_i^n P(y^{(i)} | x^{(i)}, w)$$

This seems different from regression where previously we wanted to minimize our error from the objective function by taking the argmin respective to the weights. However, in this case we are maximizing (argmax respective to weights) the likelihood function (maximizing the likelihood of a correct decision) is essentially the same as minimizing error.

$$\hat{w} = \operatorname{argmax}_w \ell(w) \quad (6.18)$$

$$= \operatorname{argmax}_w \frac{1}{n} \prod_{i=1}^n P(y^{(i)}|x^{(i)}, w) \quad (6.19)$$

$$= \operatorname{argmax}_w \frac{1}{n} \sum_{i=1}^n \log(P(y^{(i)}|x^{(i)}, w)) \quad (\text{From product to sum of logs}) \quad (6.20)$$

$$= \operatorname{argmin}_w \frac{1}{n} \sum_{i=1}^n -\log(P(y^{(i)}|x^{(i)}, w)) \quad (\text{Negative Log-Likelihood Loss}) \quad (6.21)$$

When we adjust our weights to find this point it is called the Maximum Likelihood Estimate (MLE). Remember that maximizing our likelihood is the same as minimizing our error for our model so that we have a high probability for a correct prediction and a low probability for incorrect (Definition 6.3).

Definition 6.4: Negative Log-Likelihood Loss Function

$$L(w) = \sum_{i=1}^n -\log(P(y^{(i)}|x^{(i)}, w))$$

Thus far, everything discussed for classification has been a binary +1 or -1 classification but we can also represent multi-class classification problems. Previously we used a sigmoid function to turn the score of some input into a probability, which then uses a Negative Log-Likelihood to measure the loss for the binary classification. For multi-class classification we will use the **softmax** function instead of the sigmoid function. We will still use the Negative Log-Likelihood to measure the loss. The combination of these two is called **cross-entropy loss** which we will cover in more depth later in deep learning.

Just like in regression, the logistic regression model can predict simple problems quite well if there is an easy linear separation between points. But like regression again we can model more complicated problems by including more features and/or more complex features.

Example(s)

For instance, instead of only using two words as features for our sentiment review classifier:

$$h_1(x) = \text{number of "awesome" words}$$

$$h_2(x) = \text{number of "awful" words}$$

We could use more features with 2 being more complex, like this:

$$h_1(x) = \text{number of "awesome" words}$$

$$h_2(x) = \text{number of "awful" words}$$

$$h_3(x) = (\text{number of "awesome" words})^2$$

$$h_4(x) = (\text{number of "awful" words})^2$$

However, it is important to remember that just like regression a logistic regression model can overfit just like in regression. To avoid this, just as before, we can use regularization techniques including the L1 and L2 norms which can change our quality metric to avoid overfitting, and in the case of L1, aid in feature selection. The coefficient paths will follow similar trajectories so the same ideas of comparing models with different hyperparameters and regularization techniques using only MSE on validation data still applies.

It is important to remember that while the plots from above show the decision boundaries, we are still representing our result as a probability / certainty of the model's output. When using regularization, smoothness

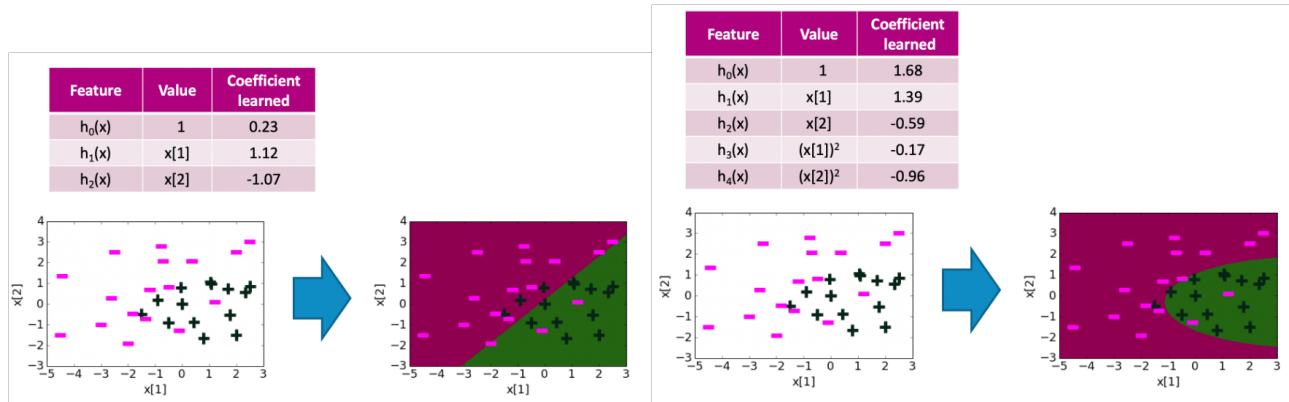


Figure 6.27: Simple and quadratic classification boundaries

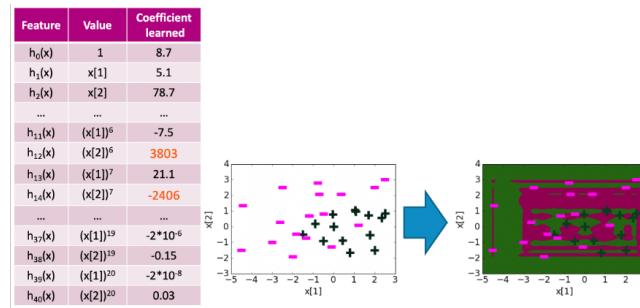


Figure 6.28: Overfit classification boundaries

can be introduced to better represent areas of certainty and uncertainty. Increasing λ corresponding to increasing the amount of regularization, for L2 especially, will increase the amount of this white / uncertain area in an attempt to resist overfitting.

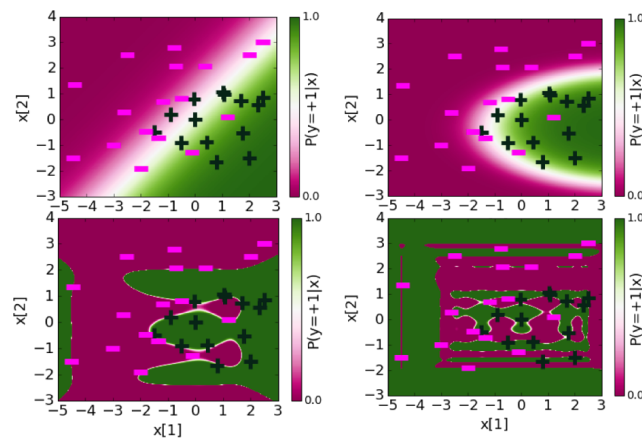


Figure 6.29: Regularized plotted probabilities

6.3 Gradient Descent

As previously stated, unlike a linear regression, we do not have a closed form solution for this problem. Instead, we must use an iterative method to find the global minimum for the function (equation 6.7) like gradient descent. In gradient descent, since we cannot compute the solution with a closed form, we must move towards the minimum step by step.

Algorithm 2 Gradient Descent

Start with random weights w

while w has not converged **do** $w = w - \alpha \Delta L(w)$ - α = Learning Rate
 - $\Delta L(w)$ = Gradient of loss function on weights w

Remember that our learning rate is important as it controls how large of a step we take during gradient descent where a learning rate that is too small could take too long to converge but a large learning rate may result in gradient descent never reaching convergence. This process for choosing the learning rate as a hyperparameter often requires lots of trial and error but is necessary for a fast, yet reliable, gradient descent. Often we use values that are exponentially spaced out.

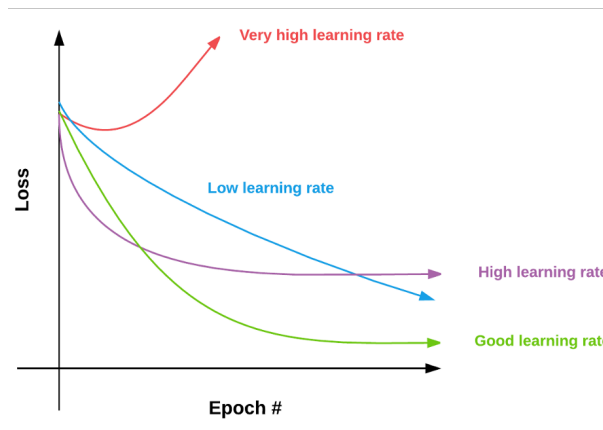


Figure 6.30: Effect of different learning rates.

Previously we discussed how different hyperparameters can affect our model's performance and what we should be looking for in hyperparameters. For fine-tuning hyperparameters, two methods are commonly used for finding the optimal values: **grid search** and **random search**

Definition 6.5: Grid Search

Start by defining a "grid" of hyperparameter values and evaluate the model at every position on the grid returning the argmin of the grid.

Definition 6.6: Random Search

Define a searching domain for possible hyperparameter values and randomly sample points in that domain evaluating the model each time.

Using these methods we can search for the hyperparameter or set of hyperparameters which results in the lowest error for the validation set. When we have multiple sets of hyperparameters to choose from in either

grid or random search, the process is repeated so that every combination is tried together, usually involving nested for loops.

The learning rate in gradient descent does not need to be fixed, though. Ideally, gradient descent could descend quickly while still being reliable enough to reach some loss / weight value convergence. Note that we can either assess convergence by the lack of changing error or more ideally by the lack of changing weight values. We could start with large step sizes, then progressively lower the learning rate each epoch / iteration of gradient descent where t is the number of iterations.

$$\alpha_t = \frac{\alpha_0}{t} \quad (6.22)$$

To compute the gradient at each iteration for gradient descent (indicating which direction we should go) we have to calculate the loss from each data point to calculate the gradient of an update. This means that for large data sets especially, gradient descent can be very computationally expensive, even with optimal learning rates. One idea for avoiding this is to simply not calculate the loss for each data point at each iteration. Less work means our process can speed up! Instead, we can sample a small number of points from the training set, rather than the entire set, to calculate the gradient at each epoch to decrease computational time. This idea is called stochastic gradient descent. In practice, this is used widely as most often each step in gradient descent does not require the entire data set and by randomly sampling the model will still be able to learn well.

The number of points used at each step can vary widely from half of the data set to only 1 single data point. Clearly, 1 single data point is computationally faster, but might require more iterations to converge, or not converge at all if the data is particularly noisy. These batch sizes can change in size widely, largely depending on the noise and size of the data set being trained on and can be thought of as another hyperparameter similar to the learning rate.

In review of gradient descent and the logistic regression model, we have learned that

- Minimizing the error is the same as maximizing the likelihood in a logistic regression model.
- Predictions are made using probabilities.
- We can use the sigmoid function to turn a score into a probability.
- To prevent overfitting regularization should still be used.
- Gradient descent must be used as there is no closed form solution to our objective function.
- Learning rate in gradient descent are important for gradient descent convergence
- Stochastic gradient descent is a good option when working with large data sets.

In the following chapter other classification models will be discussed.

Chapter 7

Naïve Bayes and Decision Trees

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

In the previous chapter we dove into the workings of the Logistic Regression Model. As you might remember, the Logistic Regression model utilizes the sigmoid function to map out arbitrary input values to output values between 0 and 1. How close an output was to either 0 or 1 gave us a certainty about classifying it, while an output closer to 0.5 meant higher ambiguity. In this chapter we will introduce another way to explore these probabilities.

7.1 Naïve Bayes

Naïve Bayes is an alternative way to compute a class probability for a given input. The formula for a given input x and its output class y is as follows:

Definition 7.1: Bayes Rule

Bayes Theorem computes the probability of a class using the formula below. If we want to compute the probability that y is a positive review given review x , we multiply the probability of x given that y is positive with the probability that y is positive, and divide it with the probability of x alone.

$$P(y = +1|x) = \frac{P(x|y=+1)P(y=+1)}{P(x)}$$

We use this equation to compute how likely a review is positive or negative:

$$\frac{P(\text{"The sushi & everything else was awesome!"} | y = +1) P(y = +1)}{P(\text{"The sushi & everything else was awesome!"})}$$

We may discard the divisor, as we are only comparing which class's probability is greater. Now, think about how you would approach this problem. We would need to plug in a probability for each part of the Bayes Theorem. Do we know $P(\text{"The sushi everything else was awesome!"} | y=+1)$, the probability of this specific review, given that we are told the review is positive? Most likely not, as the sentence "The sushi everything else was awesome!" is unique, and it only appears in one review.

Thus, we make the naïve assumption that every word's probability is independent from each other: Instead of computing this entire sentence's probability, we compute the probability of every word occurring alongside each other. So we compute the probability of "The" given the review is positive AND the probability of "sushi" given the review is positive, and so forth:

$$P(\text{"The sushi everything else was awesome!"} | y=+1) = P(\text{The} | y=+1) * P(\text{sushi} | y=+1) * P(\text{ | } | y=+1) * P(\text{everything} | y=+1) * P(\text{else} | y=+1) * P(\text{was} | y=+1) * P(\text{awesome} | y=+1)$$

Our final model is thus:

$$P(y|x_1, x_2, x_3, \dots, x_d) = \prod_{j=1}^d P(x_j|y)P(y)$$

There are a number of issues with this approach. First, since we are multiplying so many probabilities, our final product will be a very long decimal number. Since this decimal number is so long, we might end up with floating point overflow. We can overcome this by taking the log of each probabilities, such that we compute a sum instead of a product. Another issue that we encounter with using products is that if

we encounter a word we haven't seen before, its probability will be 0 and so the entire product will be 0. Laplacian Smoothing, adding a constant to each term to avoid multiplying by 0, can be used in this case.

Let us now compare the two models we have learned for computing classification probabilities thus far:

Logistic Regression:

$$P(y = +1|x, w) = \frac{1}{1 + e^{-w^T h(x_i)}}$$

Naïve Bayes:

$$P(y = +1|x_1, x_2, x_3, \dots, x_d) = \prod_{j=1}^d P(x_j|y = +1)P(y = +1)$$

While the Logistic Regression model is discriminative, the Naïve Bayes model is generative.

Definition 7.2: Discriminative Model

A model is discriminative when it only cares about finding and optimizing a decision boundary.

Definition 7.3: Generative Model

A model is generative when it defines a distribution for generating x .

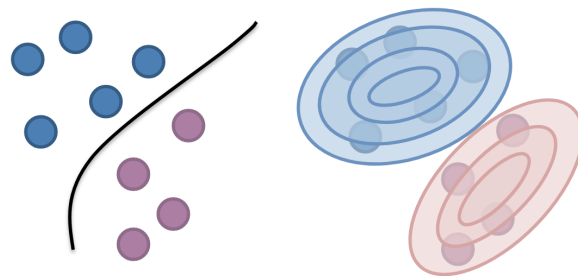


Figure 7.31: On the left visualize a discriminative model, where we only care about optimizing a decision boundary. On the right, we have a generative mode, where the likelihood of each class is also captured.

The Naïve Bayes model is considered a generative model because it computes each input's probability given the output $P(x_j|y = +1)$, so it can be used to map out a probability distribution for each x . This is unlike the Logistic Regression model, that does not compute the probability of each input, but rather solely the probability of the output given the input. Usually a discriminative model like Logistic Regression will do better, but generative models are useful for modeling distributions.

7.2 Decision Trees

The next application of machine learning we will explore will be decision trees. Take a look at the decision tree below:

Definition 7.4: Decision Tree

A decision tree is a series of questions that lead to multiple outcomes such that we can explore each question further.

Decision trees are most applicable for nonlinear decision boundaries that occur along a set of distinct criteria. Let's consider a Loan Application for buying a house as an example. To make an informed decision about a

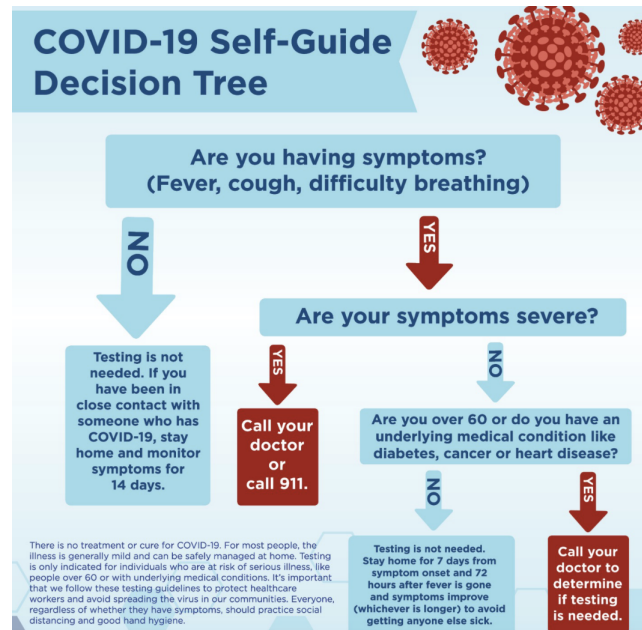


Figure 7.32: A decision tree on COVID-19 safety guidelines based on a series of questions. Source: [Holzer](#)

candidate, a loan application may ask about credit history, income, the lease term, and personal information. So, an example dataset might look like this:

Credit	Term	Income	y
excellent	3 yrs	high	safe
fair	5 yrs	low	risky
fair	3 yrs	high	safe
poor	5 yrs	high	risky
excellent	3 yrs	low	safe
fair	5 yrs	low	safe
poor	3 yrs	high	risky
poor	5 yrs	low	safe
fair	3 yrs	high	safe

Figure 7.33: Let our training dataset be n number of inputs and output pairs. Each input has 3 features: credit, term and income and one output, y .

After training on the train dataset, we aim to reach an accurate prediction \hat{y} of whether a new candidate is safe or risky based on their loan application:

A decision tree for this example can be set up like so:

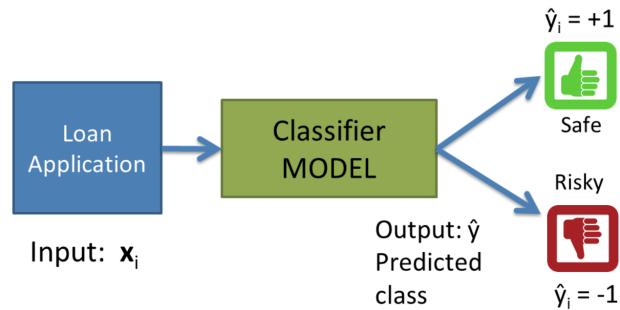


Figure 7.34: A loan application is fed into the model as input x_i . The output will be either $\hat{y}_i = +1$ (safe) or $\hat{y}_i = -1$ (risky).

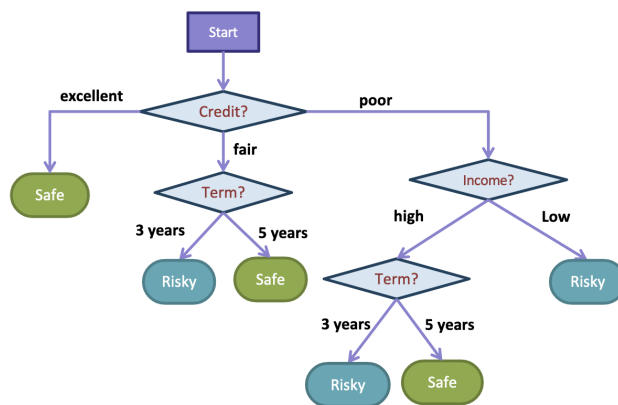


Figure 7.35: A decision tree for determining whether a candidate is safe or risky to lend a loan to. The branch/internal nodes split into possible values of a feature, while the leaf nodes are the final decision, the output class.

Definition 7.5: Tree Terminology

A **Tree** in computer science is a datastructure characterized by an interconnected set of nodes, where no connection between nodes creates a cycle.

Node: any datapoint in the tree.

Root: the first datapoint in the tree.

Leaf: one of the last datapoints in the tree.

In this example, the model first splits up loan applications based on credit history. If the credit history is excellent, a candidate's application is considered safe. However, if their credit history is poor, the tree splits into a more complex structure as more questions are required to determine if the candidate should receive a loan.

At this point, you may be wondering why we use credit history as the first criteria to judge an applicant, why we split into further questions on certain nodes, and how a lease term can determine an application's safety. These are all choices our model made to optimize its tree such that the tree would output the most probable prediction \hat{y} based on its training on the train data. We will now examine how these choices are

made in the building of a decision tree.

7.2.1 Building the Decision Tree

How does a model decide on the optimal decision tree structure? Recall that our training data is what our machine learning model uses to learn on. Given a set of x and y training pairs, we will build a decision tree that aims to lead as many train inputs x to their correct train outputs y as would perform the best in the real world. Since we want to avoid overfitting on the train set, this doesn't necessarily mean that we will build our tree such that every training input x will always lead to its exact correct training output y , so we will also have to decide a good stopping point for growing our tree. The end goal is that when new, unseen test data x is fed into the tree, we can achieve a decently accurate prediction \hat{y} .

Let's first start with the very first node in our tree, the root node:

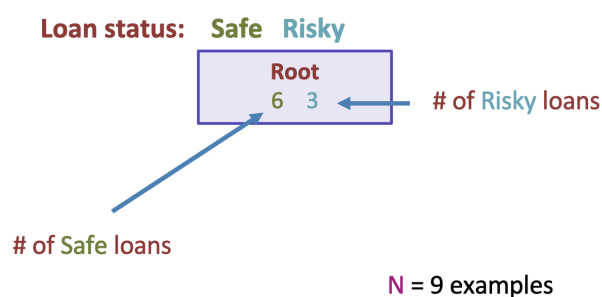


Figure 7.36: The root node of the decision tree stores information about the safe loans and the risky loans. In total, there are 9 loan applications in our dataset.

Since we have just one node, this one node is the root node and it is a leaf node at the same time. In a decision tree, the decisions are made at the leaf nodes. The outputted decision is based on the majority class of the training set inside that node. So if our tree is just a root, then our tree would always output "Safe" since there are 6 safe loans in the training set and 3 risky loans. So, on future unseen test data, we will always output "Safe".

On the train data, this gives us a $6/9 = 66.66\%$ accuracy. If we decide this isn't good enough, we will add a second layer to the tree. This second layer is a **decision stump**

Definition 7.6: Decision Stump

In a Decision Tree, a **stump** is a point at which a node splits into further decision categories.

Suppose then, that we decide to split on a candidate's credit history. Credit history can be either "excellent", "fair" or "poor".

Since we split on credit history, we add a node to our tree for each type of credit history. For each of these nodes, we now store how many of the loans are safe and how many are risky. Recall in our original dataset in Figure 8.48 there are 2 safe loans and 0 risky loans that have excellent credit history, 3 safe loans and 1 risky loan with fair credit history, and 1 safe loans and 2 risky loans with poor credit history.

If we end our tree here, we must finish it off with leaf nodes that have final output classes in them as the final decisions. Remember that these are the majorities of each node.

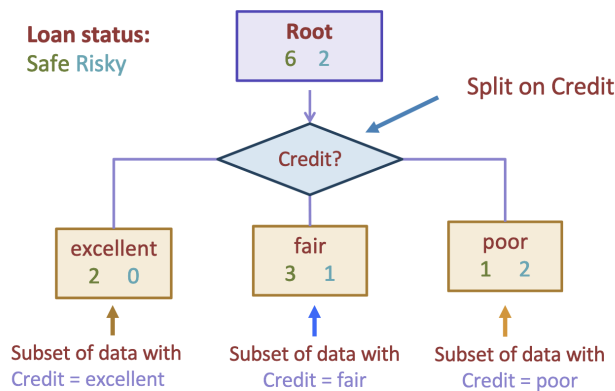


Figure 7.37: The tree splits the training data into different levels of credit history.

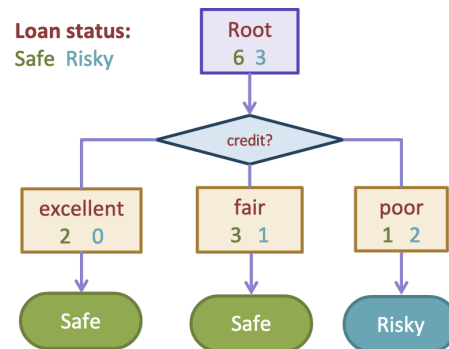


Figure 7.38: The majority class of each node determines the final output.

Now, our training accuracy is $7/9 = 77.77\%$ since 7 out of the 9 loan applications are correctly classified, and 2 are not. What if we were to split on Term first instead of Credit? Let's compare the two choices:

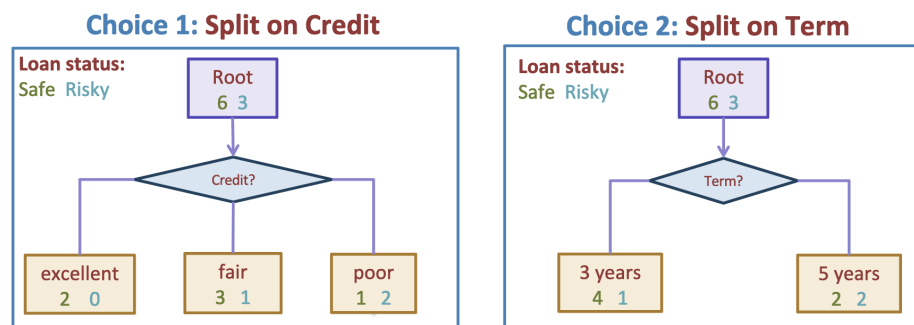


Figure 7.39: A split on Credit versus a split on Term in the decision tree.

In Choice 1 when we split on credit, we have a $7/9 = 77.77\%$ accuracy. Let's calculate the accuracy for Choice 2. For a term of 3 years, we have 4 safe loans and 1 risky loan. Safe loans make up the majority here,

so anything with a term of 3 years will be classified as "Safe". For a term of 5 years, there are 2 safe loans and 2 risky loans. Since there is a tie, we can choose either class. Let's designate the output of a 5 year term as a "Risky" loan. Since 4 loan applications for 3 year terms are classified as "Safe" when they actually are and 2 loan applications for 5 year terms are classified as "Risky" when they actually are, $4 + 2 = 6$ loan applications are correctly classified. $6/9 = 66.66\%$ of loan applications are classified correctly. Likewise, we can also use an error metric here: 33.33% of loan applications are classified incorrectly if we split on the Term feature, and 22.22% of loan applications are classified incorrectly if we split on the Credit feature. Since the Credit feature split yields a lower error rate, our tree will split on Credit first. This decision making is how the model splits a node in a tree and it can be summarized by the following pseudocode:

Split(node)

- Given a subset of data M in node
- For each feature h_i :
 - Compute classification error for a split of M according to feature h_i :
- Chose feature $h^*(x)$ with lowest classification error and expand the tree to include the children of current node after the split

Figure 7.40: The algorithm for selecting the best feature a node should be split on.

Our model will choose between features to split on for every node in the tree, until the number of datapoints in each node or the error rate reaches a certain threshold to stop building the tree. This algorithm can be summarized as:

BuildTree(node)

- If the number of datapoints at the current node or the classification error is within a certain threshold:
 - Stop
- Else:
 - Split(node)
 - For child in node:
 - BuildTree(child)

Figure 7.41: The algorithm for building the tree.

Notice that the Decision Tree algorithm is greedy: It aims to optimize the classification error at each node. As a result, the final result won't be globally optimal, but it guarantees computational efficiency. Also take note that the Decision Tree algorithm is recursive: From the current node, if we decide to further expand the tree, we repeat the same operations in the child node.

How do we decide when to stop? Going back to Figure 12.71, we observe that for applicants with excellent credit history, all two of them would be safe loans and none of them would be risky. So, the majority class classifier would suffice here, because it would yield no error. But when the Credit is fair or poor, we had some safe loans and some risky loans. As there is no absolute majority for either of these nodes, we could recursively treat them as roots of their own tree which we continue to build below. Thus by further splitting these nodes, we grow a more complex and precise path in our tree with the hopes of lowering the error rate.

We can set a certain error threshold and stop growing our tree until every leaf yields that error or less.

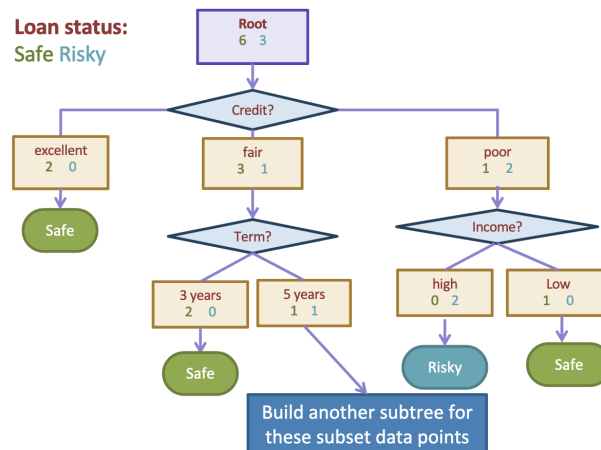


Figure 7.42: Every subtree build from each node will be a subset of the original trainset where the condition of the node is true for the remainder of the tree.

7.2.2 Numeric v.s. Categorical Features

In our Loan Application example, we have been looking at data without numeric value. For example, a candidate's credit history can be of the values "good", "fair", or "poor". There are 3 main datatypes a machine learning algorithm can learn from.

Definition 7.7: Numeric

Numeric data is data that has numerical value, such as square footage of a house.

Definition 7.8: Ordinal

Ordinal data is data that has ordered categories, even if it is not numerical. Credit score is ordinal, because you can rank "good", "fair", and "bad".

Definition 7.9: Nominal

Nominal data is data that has no numeric value and cannot be ordered. Colors such as "red", "blue", "green" cannot be ranked.

Datatypes that are not numeric like ordinal and nominal data are **categorical**.

Some decision trees may or may not require all numerical inputs depending on their implementation. However, in models that use differentiable loss functions (like Linear Regression / Logistic Regression, some forms of decision trees), you need to transform categorical data.

Transforming Ordinal Data: Rank the values (bad = 0, fair = 1, good = 2).

Transforming Nominal Data: Use one-hot encoding.

Definition 7.10: One-Hot Encoding

One-hot encoding is a transformation technique of nominal data. For a feature, it assigns a digit for every possible category that feature can be. For example, for a Color feature, we can assign "red" to the first digit, "blue" to the second digit, and "green" to the third. So, red would be 100, blue would be 010, and green would be 001.

It is important to note that many nominal feature do come in the form of a number, yet they cannot be ordered. An example of this is zip code when estimating house prices. 10018 is a zip code in Manhattan, NY while 98105 is a zip code in Seattle, WA. If we use zip code as a feature of a linear regression model to predict house prices, this does not mean that houses in Seattle are more expensive than those in Manhattan.

7.2.3 Threshold Split for Numeric Features

For numeric features, it doesn't make sense to have individual branches in our tree for every single numerical value that feature could hold. So, we use an inequality instead. One branch will be less than a certain number v , and the other branch can be greater than or equal to. Given a range of numerical values from a single feature, we can choose the best threshold to make this split by calculating the classification error for every possible split that occurs exactly in between each of the possible numerical values v_1, v_2, \dots, v_n and select the threshold yielding the lowest error. For more precision, we can split these inequalities further. There is a limitation to using classification error: Two different splits can give us the same error, so continuous loss functions like entropy loss or Gini impurity loss (which we will not talk about in this text) are used.

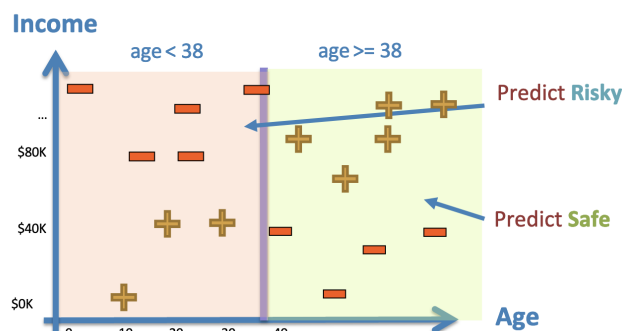


Figure 7.43: For a decision tree, a splitting of a numeric feature will always be along the axis.

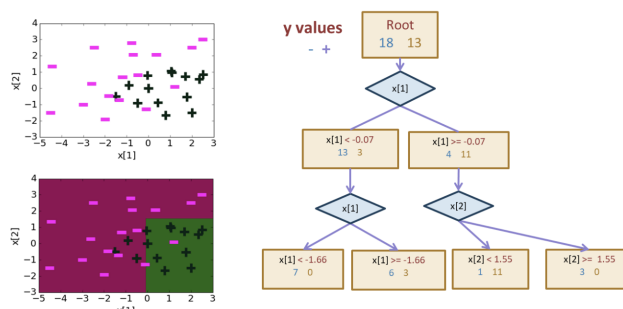


Figure 7.44: If we decide to split further, we continue to split along the axes.

7.2.4 Advantages of Decision Trees

1. Decision Trees are easy to interpret
2. They can handle both continuous and categorical variables without preprocessing
3. They do not require normalization
4. They can create non-linear decision boundaries
5. They can handle missing values

7.2.5 Disadvantages of Decision Trees

1. Deep Decision trees are prone to overfitting. They are not suitable for large datasets for this reason.
2. The decision boundaries are unstable because adding a new datapoint to our trainset can cause the entire tree to regenerate.
3. The decision boundaries must be axis-parallel.

To overcome overfitting, we can implement conditions for early stopping such as establishing a fixed depth length, setting a maximum number of nodes, or halting growth if error does not considerably decrease. We can also **prune** our trees by cutting off some nodes.

Chapter 8

Ensemble Methods

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

So far in our exploration of Classification, we have covered Logistic Regression, Bayes Theorem, and Decision Trees as candidates for modeling data that is intended to be separated into classes. With the Decision Tree approach, we can model our data in a much more interpretable way, but Decision Trees are not known for reaching stellar accuracy. In this chapter we will discuss the modifications done unto Decision Trees to improve their performance. Specifically, we will discuss the Ensemble Method.

8.1 Ensemble Method

The basic idea behind the Ensemble Method is to combine a collection of models in hopes that their unity will achieve better performance rather than designing a new model from scratch. This collection of models is referred to as a **model ensemble**.

Definition 8.1: Model Ensemble

A **model ensemble** is a collection of (generally weak) models that are combined in such a way to create a more powerful model.

With trees, this is done in via either the **Random Forest (Bagging)** method, or **AdaBoost (Boosting)**.

8.1.1 Random Forest

The Random Forest method, also called Bagging, utilizes a collection of Decision Trees. Each decision tree casts a "vote" for a prediction based on an input and the ensemble predicts the majority vote of all of its trees. Each tree is built from a subset of the training dataset by random sampling the training dataset with replacement. This technique is also called **bootstrapping**.

When training the decision trees on the bootstrapped samples, the goal is to build very deep overfitting trees. Each tree thus has very high variance. As multiple models are used for the final prediction in the ensemble, the overall outcome is a model with low bias. This is because if many overfitting models are averaged out, the result is a model that does not overfit to any one sample so the ensemble has low bias and lower variance. So, while each tree is a robust modeling of a data sample, the united model ensemble is more powerful.

Example(s)

Random Forest has many applications in image processing and object detection. Microsoft [used the Random Forest modeling technique in their Kinect system](#) to identify the "pose" of a person from the depth camera.

The advantage of using a Random Forest is that it is versatile, and has many applications in classification, regression, and clustering. It is also low-maintenance and hence easily interpretable: There are few to none hyper-parameters in most cases, and generally more trees gives us better performance. Random Forest is also efficient, because the trees, as separate components, may be trained in parallel. So instead of waiting for a single complex model to train one step at a time as we may do with other models, we can assign each tree of the ensemble to a different machine so that multiple trees are trained simultaneously.

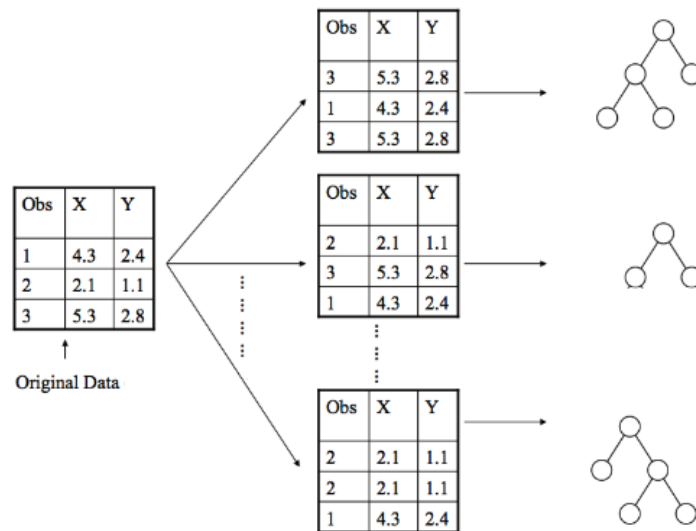


Figure 8.45: A subset of the original dataset is randomly selected with replacement for every tree we intend to build.

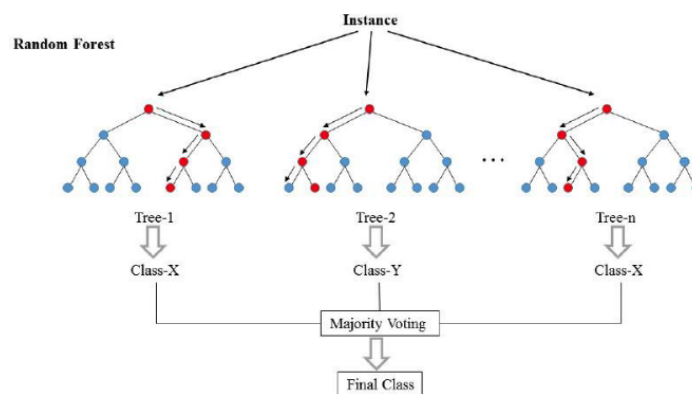


Figure 8.46: Each tree predicts its own final decision based on the input and the majority prediction is selected by the ensemble.

8.1.2 AdaBoost

Recall that a decision stump in a tree is a node that splits into multiple decisions. When we say "stump" we will refer to a tree of one-level. As opposed to growing very deep and overfitting trees in Random Forest, Adaboost utilizes stumps. So, the model ensemble for Adaboost is a collection of one-level trees. Each of these trees are assigned a weight. A weighted majority of votes from these decision stumps is what determines the ensemble's final prediction.

A key difference between Random Forest and AdaBoost is that AdaBoost is sequential, while Random Forest can train each of its trees in parallel. This is because in AdaBoost, each stump belongs to a certain order, and the first stump's decision impacts the second, which impacts the third, and so forth.

Training AdaBoost

When training AdaBoost, the error of the previous stump affects the learning of the next stump. To do this, two types of weights are stored:

1. \hat{w}_t : the weight of each stump t .
2. α_i : the weight of each datapoint i .

The training process of AdaBoost is as follows:

1. Decide on our stumps: the more features our data has, the more stumps we will need in AdaBoost since we will have to make at least one decision from each feature. We will call the total set of stumps T , and each stump in T is t .
2. Initialize the weights for all the datapoints in our training set α_i to be equal to each other.
3. For every stump t in T :
 - (a) Learn a final prediction $\hat{f}_t(x)$ based on α , the weights of the datapoints used in t .
 - (b) Compute the error of t 's prediction $\hat{f}_t(x)$. As AdaBoost introduces weights to datapoints, we must compute a **weighted** error:

$$\text{WeightedError}(f_t) = \frac{\sum_{i=1}^n \alpha_i \mathbb{I}\{\hat{f}_t(x_i) \neq y_i\}}{\sum_{i=1}^n \alpha_i} \quad (8.23)$$

This equation is simply taking the weighted sum of all the misclassified datapoints and then dividing it by the total weight of all the datapoints. In the weighted error, every datapoint's unique weight is used to compute the total error.

- (c) Based on the error rate of prediction $\hat{f}_t(x)$, update model t 's weight \hat{w}_t . If t yields a low error, then we want \hat{w}_t to be greater, but if the error is high, then the weight should be low. This is because we want more better performing models to have a higher influence in the final decision of the ensemble. The formula for computing the model weight is:

$$\hat{w}_t = \frac{1}{2} \ln\left(\frac{1 - \text{WeightedError}(\hat{f}_t)}{\text{WeightedError}(\hat{f}_t)}\right) \quad (8.24)$$

- (d) Recompute the datapoint weights α . AdaBoost will *increase* the weight of those datapoints it misclassified, and *decrease* the weights of those it classified correctly. This way, misclassified datapoints are more sensitively accounted for in the training of the next stump. This is the notation for updating the weights:

$$\alpha_i \leftarrow \begin{cases} \alpha_i e^{-\hat{w}_t} & \text{if } \hat{f}_t(x_i) = y_i \\ \alpha_i e^{\hat{w}_t} & \text{if } \hat{f}_t(x_i) \neq y_i \end{cases}$$

4. After we have computed $\hat{f}_t(x)$ for every stump t , we take the sign of the model outputs' weighted sum like so:

$$\hat{y} = \hat{F}(x) = \text{sign}\left(\sum_{t=1}^T \hat{w}_t \hat{f}_t(x)\right) \quad (8.25)$$

Real-Valued Features

If we are splitting real-valued numeric features in our AdaBoost method, the algorithm is more or less the same but the splits must happen dependent on the weights of each numerical value.

Normalizing Weights

Generally, the weights for some datapoints can get very large or very small in magnitude due to how the data is laid out. If our numbers are on wildly different scales, we may run in to problems when running AdaBoost on our machine due to the finite precision of computers when it comes to real numbers. To resolve this, we normalize all the weights to sum to 1:

$$\alpha_i \leftarrow \frac{\alpha_i}{\sum_{j=1}^n \alpha_j} \quad (8.26)$$

Visualizing AdaBoost

Recall that all the weights for the datapoints are initialized to be equal to each other. After training on one stump, we compute new weights α_i based on the errors of \hat{f}_1 .

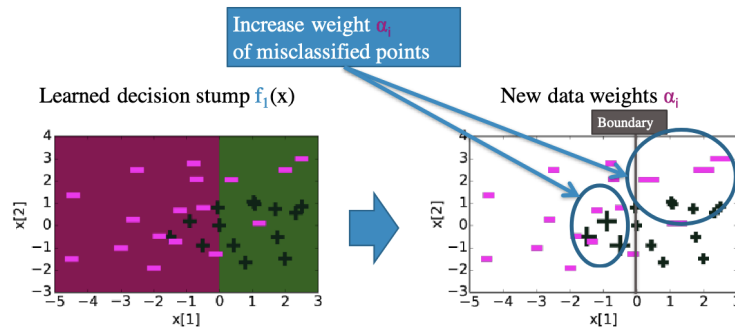


Figure 8.47: After training on the first stump, misclassified datapoints are given a greater weight (drawn larger).

The new weights α_i are used to learn the best stump that minimizes the weighted classification error. Then, update the weights again based on the error from \hat{f}_2 .

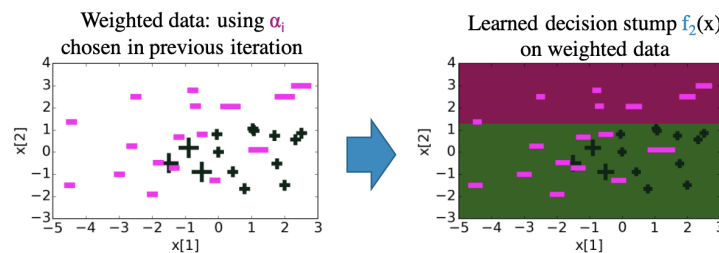


Figure 8.48: \hat{f}_2 is predicted based on the weights computed in the previous iteration.

If we plot what the predictions would be for each point, we get something that looks like this:

Eventually, we can achieve a training error of 0 with a large enough set of weak learners.

The example above illustrates that AdaBoost is still capable of overfitting!

AdaBoost Theorem

The core idea behind the Ensemble Method is that a collection of weaker models combined will yield a more powerful model. This is the case with AdaBoost, where a series of weak single-stump trees can gradually

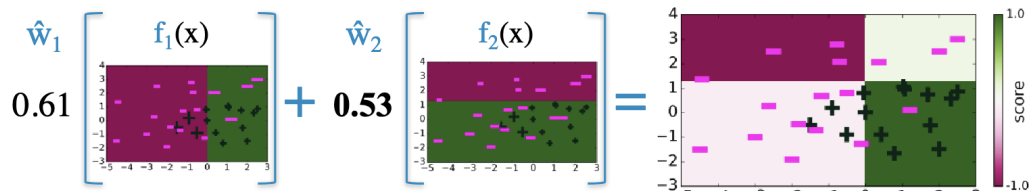


Figure 8.49: The weighted sum of the models $t = 1$ and $t = 2$ yields the weighted decision boundary on the right.

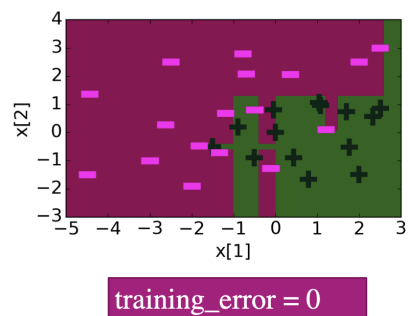


Figure 8.50: Decision boundary for $t = 30$.

reduce the training error of the overall model. Let's compare AdaBoost to a standard Decision Tree.

While a Decision Tree can significantly reduce the error of its training set, we see that this is at the expense of severe overfitting. So a Decision Tree lacks the ability to generalize to unseen test data. However, due to the internal mechanisms of AdaBoost, we avoid this severe overfitting and are more likely to achieve a lower test error. AdaBoost is also capable of overfitting as we have seen in the previous example, but in general the test error will stabilize. To determine the best T to avoid overfitting and underfitting, we must treat T as a hyper-parameter and compare different T 's with the intent of minimizing the validation error.

Applications of AdaBoost

Boosting, AdaBoost, and other variants like Gradient Boost are some of the most successful models to date. They are extremely useful in computer vision, as they are the standard for face detection. Most industry Machine Learning systems use model ensembles.

To conclude, AdaBoost is a powerful model ensemble technique and it typically does better than Random Forest with the same number of trees. However, while you do not have to tune parameters for AdaBoost (besides selecting T), boosting is sequential, so trees cannot be trained in parallel.

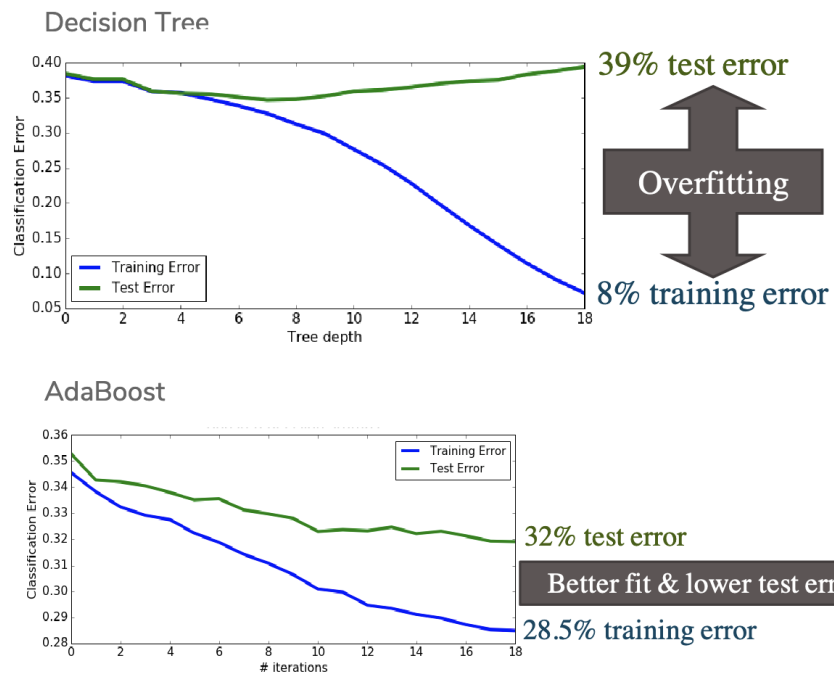


Figure 8.51: The train and test error curves for a standard Decision Tree and AdaBoost.

Chapter 10

Fairness in ML

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

In this chapter, we will look at more examples of bias and fairness in Machine Learning. We will dive deeper into the motivating examples for why one should care about bias, multiple case studies of bias sources, the pillars of trustworthy machine learning, how to tackle bias in machine learning, and finally real world examples of tackling bias.

10.1 Examples of Bias

10.1.1 COMPAS Recidivism

COMPAS is a tool used to predict **recidivism**, the tendency of a convicted criminal to reoffend and it has been used in almost all 50 states. In 2016, [ProPublica investigated the bias COMPAS exhibited against black people](#) and concluded a statistically significant difference in prediction scores for people of different races committing the same crimes.

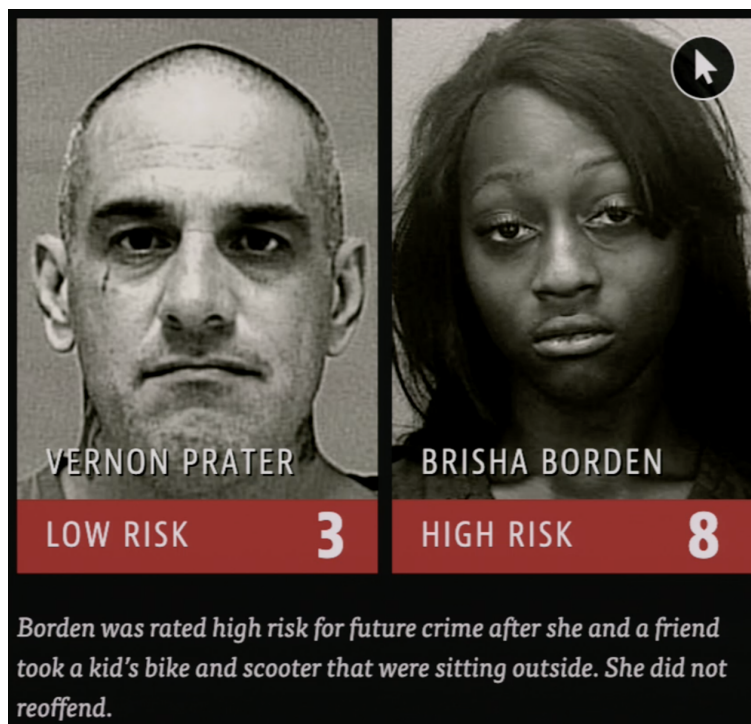


Figure 10.52: Shown above is one example out of the thousands investigated by ProPublica of criminals of the same offense assigned different recidivism scores. Both Prater and Borden were convicted of stealing a bike. While Prater was given a recidivism score of "3", Borden received an "8".

10.1.2 Amazon Recruiting Algorithm

Amazon discovered in 2015 that their **recruiting algorithm was biased against women**. Their model was trained on the résumés of previously hired employees, so whether a new candidate passed the primary screening was dependent on how the model rated their résumé. The company couldn't find a way to fix the bias in their model so they got rid of its use completely.

10.1.3 Healthcare Risk Management

In 2019 the Scientific American published an article on an **unnamed software deployed by American healthcare companies to determine which patients qualified for "high-risk care management"**. The software was used on 200 million patients in the US. Frequency of health care usage and medical spending were factors the model used to determine qualifying patients, and as a result black patients, while suffering from chronic illness at higher rates, would qualify less.

10.1.4 Predictive Policing

PredPol, a software company specialized in **predicting policing, made predictions on future crime based on existing crime data**. Criticized by several academics, PredPol was found to be target black and latinx communities. As more police were deployed in certain neighborhoods, members of those neighborhoods were watched more frequently and thus arrested for more crimes, even if these crimes were minor. This data was used to further train the model, creating a feedback loop, thereby further biasing the model.

10.1.5 Amazon Prime

Amazon Prime's one day delivery service was found to exclude predominantly black neighborhoods across the United States. Neighborhood income was used to determine where Amazon warehouses should be built. As Amazon is a large company, it quickly came under fire by major news outlets. To avoid a bad public image, the company **quickly vowed to fill racial gaps in its delivery services**.

10.1.6 Facebook Ads

In 2019 MIT published a study showcasing Facebook's biased algorithm for ads. While men were more likely to be shown ads for careers like being a doctor or an engineer, women were shown ads for nursing and becoming secretaries. While white people were shown ads for home sales minorities were shown rentals.

10.1.7 Google Ads

In 2013 Latanya Sweeny, a Harvard Professor, found out that **Googling "black-sounding" names were 25% more likely to trigger ads for finding criminal records** in someone's background.

10.1.8 Face Detection

Computer scientist and activist Joy Buolamwini began a project that became her MIT Thesis: *Gender Shades: Intersectional Phenotypic and Demographic Evaluation of Face Datasets and Gender Classifiers*. The research found that many of the leading facial classification technologies used in a variety of applications today classify black women's faces nearly 30% less accurately than their white male counterparts.

10.1.9 Nikon Camera Blink Detection

Nikon's blink detection was found to falsely classify Asian people as blinking.



Figure 10.53: Harvard Professor Dr. Latanya Sweeney.



Figure 10.54: Activist and computer scientist Joy Buolamwini. You can watch Buolamwini's interview [here](#).

10.2 Source of Bias: Case Studies

Now that we have gone over dire examples of why fairness in ML matters, we will examine some of the **sources** of bias. First, let's review some different causes that can lead to bias. Recall that there is **historical bias**, **measurement bias** **representation bias**, **deployment bias**, **evaluation bias**.

We will look at examples of case studies where such biases are found.

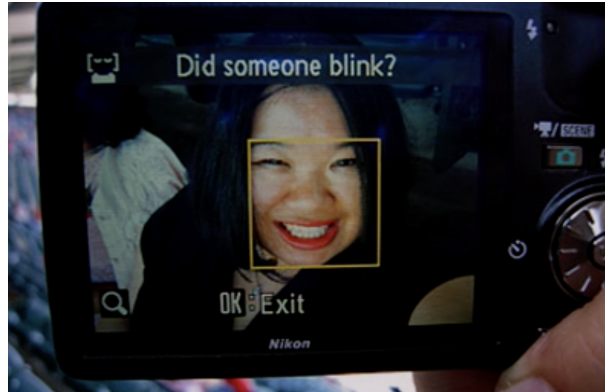


Figure 10.55: Nikon's failure to correctly detect Asian faces.

Example(s)**Case Study: Distaster Relief**

Goal: Accurately assess damage and send appropriate relief resources.

Data: Twitter posts, facebook posts geocoded with coordinates within disaster area, keywords and hashtags related to the storm.

Analysis: Intensity and type of damage by neighborhood.

Actions: Assessment and allocate relief effort (type and amount).

Constraints: Limited resources for relief efforts.

Can you find some problems with this method of resource allocation?

1. **Representation bias:** Not everyone has access to social media.
2. **Historical bias:** Wealthier neighborhoods, with more infrastructure, will report more damage.

Example(s)**Case Study: Personality Detection**

This case study is based on a real example called Faceception, an Israeli startup is selling a software that can classify personalities with 80% accuracy.

Goal: Detect if someone is a terrorist or not.

Data: People's faces.

Analysis: IQ, profession, skills, personality, terrorist activity.

Actions: US Homeland Security body purchased the software in 2016.

Constraints: Limited pool of faces and personality for training data.

Can you find some problems with this method?

1. **Representation bias:** One country is not representative of the faces and personalities around the world.
2. **Historical bias:** In media headlines middle eastern people are labeled as "terrorists" while white terrorists are always labeled as "troubled young man with mental health problems". If this is the labeling used for the dataset, then the model is trained on skewed historical bias.

Example(s)

Case Study: Sexuality Detection

This case study is also based on a real example. A Stanford professor developed a facial recognition system using 75000 photos from online dating profiles of all white people to **label people's sexualities**. The model reaches 71% accuracy for women and 81% accuracy for men.

Goal: Detect if someone is gay or straight.

Data: People's faces.

Analysis: Sexuality based on facial structure.

Actions: The professor claims that the research would help people, but he said in an interview that he was "flattered" that the Russian Prime Minister and the Russian cabinet convened with him to discuss his technology. In 2018 Christopher Wylie, a whistleblower, revealed that Cambridge Analytica was using this technology as an "instrument of psychological warfare".

The problems and constraints with this application of ML are obvious.

10.3 Pillars of Trustworthy ML

As you can see, machine learning is a powerful double-edged sword. While the technology has the potential to do good, it can also have disastrous consequences if handled by the wrong people. As an ML developer in the making, it is your responsibility to employ the Pillars of Trustworthy ML so that we can transform this space into one that honors an intersectionally empowering future.

10.3.1 Pillar 1: Privacy

Cloud photo storage company Ever AI sold the contents of thousands of people's private camera rolls to surveillance agencies to train facial recognition software. Upon accusations, Ever AI renamed itself to Paravision to claim that they had never done what Ever AI did. Before 2020 they made \$29 million in profits before finally busted by the ACLU. Thus, privacy is imperative for an ethical ML system.

10.3.2 Pillar 2: Ethics

In 2018, CEO Matthew Zeiler disclosed that computer vision software company Clarifai was selling software to the Pentagon for autonomous weapons. Clarifai was aiming to use computer vision technologies to auto-detect enemies on a battlefield. Zeiler was forced to disclose the news thanks to company whistleblower Liz O'Sullivan. By whistleblowing the usages of Clarifai, O'Sullivan exemplified ethics in ML.

10.3.3 Pillar 3: Interpretability

Interpretability is the ability to explain *why* a model makes a certain decision. An example of an interpretability problem is when a bank uses an ML model to decide which customers' loan request should be approved. A customer who earnestly needed money asks why their loan request did not get approved and what they can do to get it approved. To answer to this customer, the bank would have to dissect and understand how its model made its decision. This pillar is especially relevant to the healthcare management example we previously showed.

10.3.4 Pillar 4: Robustness

Robustness will test how consistently reliable a model is. While an image of a pig is classified correctly at first, adding random noise to the image confuses the model and the model classifies the pig as an airplane. This is an especially large problem for facial detection if a model is not exposed to a diverse enough set of face types.

10.4 Tackling Bias

”Fairness” can take on a number of definitions depending on the context. In order to conduct a context-aware bias mitigation, we can follow the practices of each stages of processing:

1. Pre-processing: Resampling of rows of the data, reweighing rows of the data, flipping the class labels across groups, and omitting sensitive variables or proxies.
2. : In-processing: Modifying the loss function to account for fairness constraints with respect to an analysis of false positives and false negatives.
3. Post-processing: Adjust the outputs of the model to abide by a fairness criteria.

10.5 Real World Examples of Tackling Bias

Chapter 11

Clustering

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

11.1 Introduction to clustering

So far we have discussed many different supervised learning algorithms, from linear regression to decision trees. Remember, supervised learning is where the data inputs map to a known output. In this chapter, we will learn about our first unsupervised learning algorithm, meaning it uses an unlabeled data set to form predictions.

Definition 11.1: Clustering

Clustering is an unsupervised and automatic process of trying to identify and group similar data points within a larger data set.

Although the primary use of clustering is for unlabeled data, we can still use clustering algorithms to fit to known labels, like "world events" or "sports". This supervised learning version of clustering, as we have learned previously, is **classification**.

Example(s)

Clustering has a number of uses, varying from simple to complex:

- Classifying news article topics, for example between "world events" and "sports".
- Clustering groups of people according to their genome.
- Recommending entertainment for consumers using user preferences. However, topics and preferences are often not well defined, making this problem challenging.

Clustering uses the given inputs, for example x_1, x_2, \dots, x_n , and attempts to learn the structure from the unlabeled data X and assign them to groups. If we decide that there are 3 groups we wish to cluster, the outputs would be y_1, y_2, \dots, y_n where y_i is either one of those three classes, for example 1, 2, or 3. More formally, clustering uses a given x_i and assigns $y_i \in [1, 2, \dots, k]$ where k is the number of clusters to be identified. A cluster is usually based on the closest centroid, which is the average for some group of data.

As usual, we should define some objective which determines how good each of the assignments are. One example could be euclidean distance from the centroid, such that points far away from a center would have high error, while those very close to a center have low error. In this case a close distance would represent a strong similarity between points. However, it is important to remember that clustering is not always straight forward and distance alone might not determine specific clusters.

11.2 K-Means

11.2.1 Overview of K-Means algorithm

K-means is the first clustering algorithm we will discuss. The main objective for k-means is to assign points to a cluster based off the point's distance from a centroid. This means that a shorter distance will create better clustering when using this algorithm.

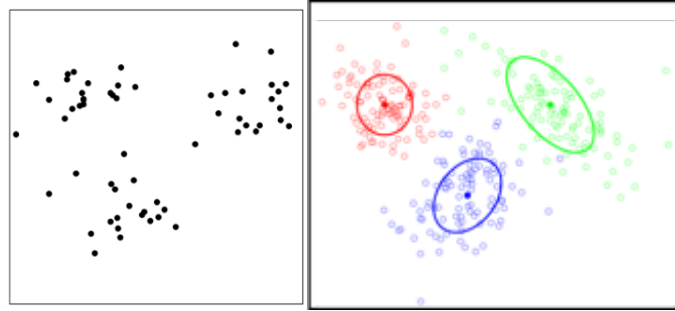


Figure 11.56: Unlabeled input data on the left with 3 defined clusters on the right.

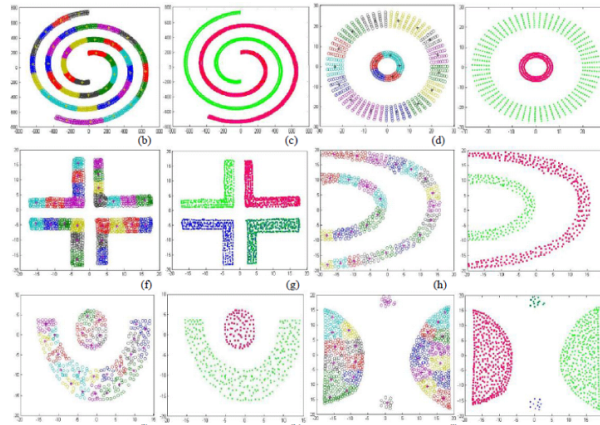


Figure 11.57: Many different examples of more complicated clusters.

Algorithm 3 K-Means Clustering

Given a data set of n data points and a particular choice of k

Step 0: Initialize k centroids randomly to points in the data set

while not converged **do** **Step 1:** Assign each example to its closest cluster centroid

Step 2: Update the centroids to be the average of all the points assigned to that cluster **return** cluster assignments as closest centroid to each data point

Above is the basic outline for K-Means. However, there are many different ways to approach each of the steps, depending on the use case.

- **Step 0:** Usually we initialize the centroids $\mu_1, \mu_2, \dots, \mu_k$ randomly, however there are smarter initialization techniques which will be discussed below.
- **Step 1:** We assign each example to its closest centroid such that:

$$y_i = \underset{j}{\operatorname{argmin}} ||\mu_i - x_i||_2^2$$

- **Step 2:** Now we update the centroids to be the mean of all the point assignments from the previous step (i.e. the center of mass for the cluster).

$$\mu_j = \frac{\sum_{i=1}^n 1(y_i = j)x_i}{\sum_{i=1}^n 1(y_i = j)}$$

- **Convergence:** There are multiple different stopping conditions. Some include:
 - When the cluster assignments haven't changed.
 - Centroids haven't changed locations.
 - Some number of max iterations have passed.

It is also important to remember that K-Means is heavily dependent on its starting assignments so the algorithm may only converge to a local optima rather than a global one. To get around this, we can either use a smarter initialization compared to random and/or by running K-Means multiple times to ensure we arrive at similar results across trials.

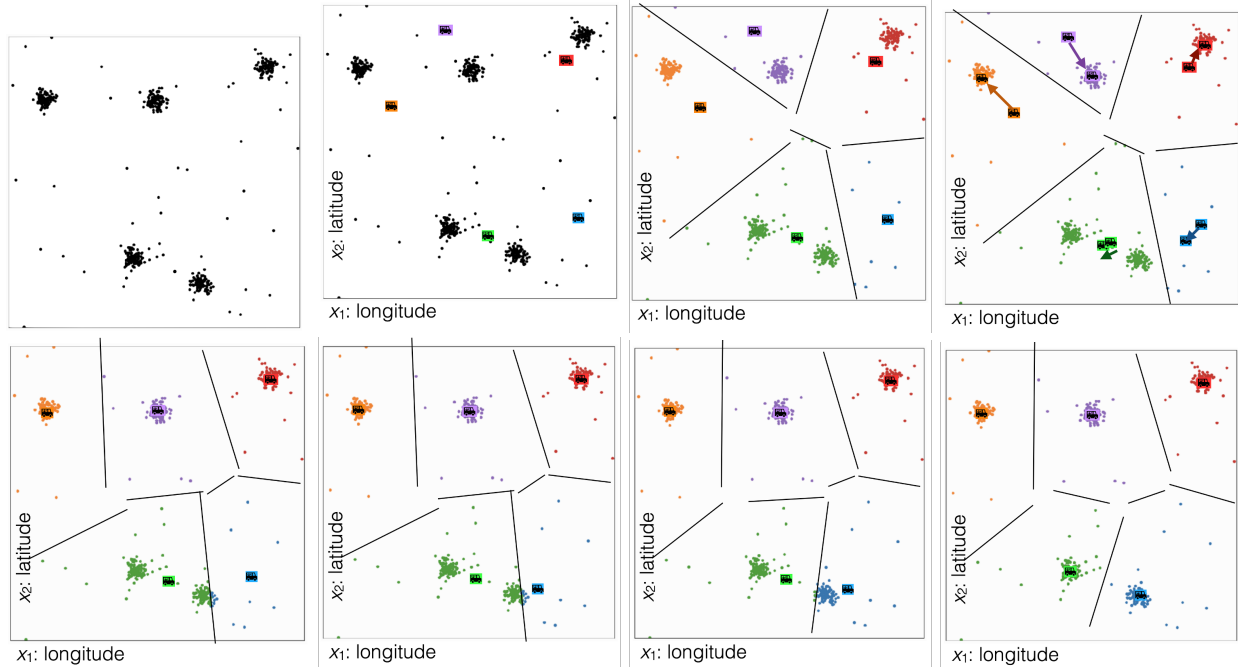


Figure 11.58: K-Means algorithm visualized.

11.2.2 Choosing k in K-Means

It is also important to remember that k in K-Means is a hyperparameter which defines the number of clusters to find so that results will be very different for different values of k . Choosing the correct k is very important as a k that is too low, for example, could be grouping things together when they really are distinct groups. Illustrating this, the green and blue groups from the example above would likely get grouped together if $k = 4$, instead of the correct choice of $k = 5$. Following that same logic, a k that is too high will put 2 (or more) groups together when they should be one.

Definition 11.2: Heterogeneity Objective

$$\operatorname{argmin}_{y, \mu} \sum_{j=1}^k \sum_i 1(y_i = j) \|\mu_j - x_i\|_2^2$$

This is the objective that K-Means uses, which is equivalent to minimizing the distance from each centroid to its group of examples.

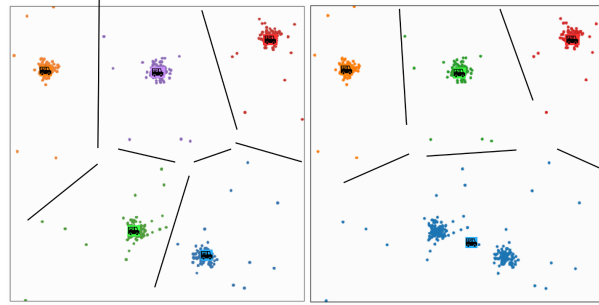


Figure 11.59: Example of choosing different values of k . ($k = 5$ on right and $k = 4$ on left)

However, the choice of k is often not easy to see, either because the dimension of the data is high or it is not easy to visualize the data. One way to approach this problem is to use a validation set / k -fold approach like discussed in previous chapters to see which k is the best choice for clustering. This is a good idea but slightly different when put into practice. Clustering is different than trying to solve for the quality metric's minimum. If that was our goal, the optimum choice of k would be n , the number of data points, such that the total distance from each point to its centroid is 0. Clearly, this is not helpful in terms of clustering.

Classification quality metrics do not apply to clustering. Classification learns from minimizing the error between a prediction and an actual label while in clustering the "labels" don't have meaning, so it is up to human input to determine what good clustering is and isn't.

Our goal of minimizing error, similar to the idea of overfitting from previous chapters, has caveats. One way we can approach using a validation set with hard-to-visualize data is the elbow method. We plot the quality metric across different values of k (multiple times for each k due to reasons stated above) and see where there is an "elbow" in the graph. This corresponds to reaching the optimal k value, as the error will drop sharply, but after adding more points the error should only reduce slightly run after run.



Figure 11.60: Choosing the best k using the elbow method.

Example(s)

Movies: choosing k when clustering movies should most likely be used with a pre-determined number categories as a human choice, rather than chasing the k which has the lowest validation set loss.

There are a number of other methods to choose k also: Bayesian Information Criterion, Silhouette, etc.

11.2.3 Effects of Initialization in K-Means

As previously mentioned, the initialization of the centroids in K-Means can have a large impact on the outcome of the clusters. First, it is important to remember that the label assigned in K-Means does not have any meaning. We know that runs of K-Means work well if the same groups of points are assigned together, but not necessarily if a certain centroid is always assigned to the same group of points.

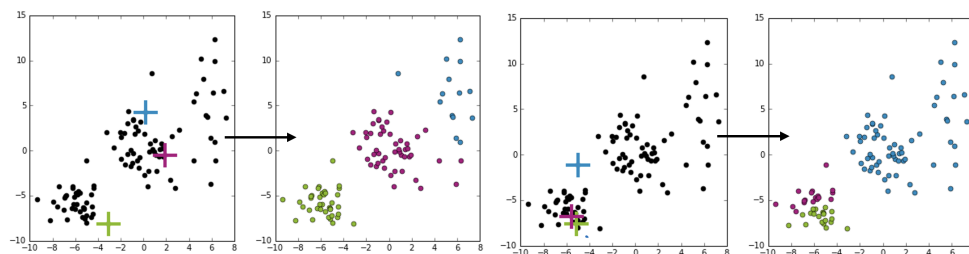


Figure 11.61: Initialization can affect cluster outcomes.

Given that K-Means is quite simple and only considers centralizing the points which are closest to a given centroid, K-Means might get stuck in a local optima rather than a global. Ideally we would like to initialize such that centroids start far away from one another, rather than simply at random, so that two centroids don't converge to one group, leaving 2 "real" groups potentially clustered as 1.

11.3 K-Means++

To initialize points with a wide spread we could factor in whether we choose a point to be an initialization based on how far away that point is from other centroids we have already chosen during the initialization. Using this we can initialize to a point we are more likely to be closer to the optimal solution than purely at random. This can lead to faster and more accurate/reliable convergence of centroids. This type of initialization will also likely reach a global minima rather than only a local. However, note that this initialization is more computationally expensive at the beginning compared to random initialization as we have to keep track of each points distance to its nearest centroid and update those values each time we select of new one.

Algorithm 4 K-Means++ Initialization

Step 1: Choose first centroid μ_1 uniformly at random from dataset

for j from 1 to k **do** **Step 2:** For current set of centroids compute the distance between each datapoint and its closest centroid

Step 3: Choose new centroid μ_j from the remaining data points with probability of x_i being chosen proportional to distance between x_i and its closest centroid (it's current assigned center) **Step 4:** run K-Means from intialized centroids as normal

11.4 Caveats of K-Means

K-Means is a powerful tool for clustering however, like most ML algorithms, it is not perfect for every scenario. K-Means works well for **hyper-spherical** clusters of the same size but when the true clusters don't look like that, it can produce less than ideal results.

To wrap up our introduction to clustering, there are a couple important points to remember:

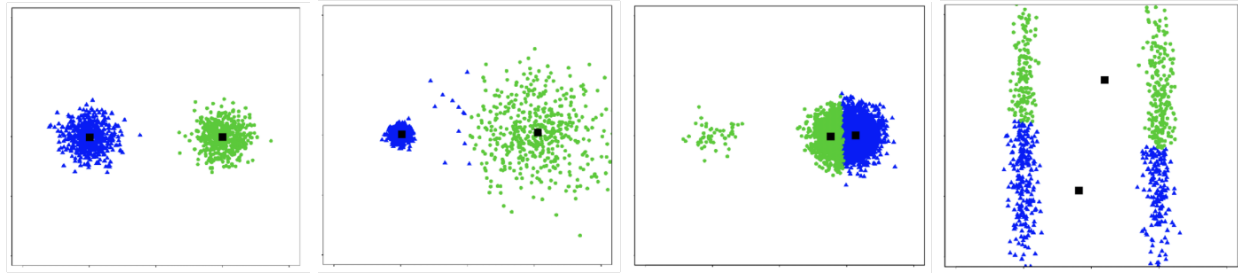


Figure 11.62: Some examples where K-Means works well and some where it fails.

- Usually, we don't split train / test set for clustering problems, because there are no labels to measure how good a set of clusters are.
- Similarly, definitions like bias, variance, complexity bias-variance trade off might not work for unsupervised learning problems.
- The heterogeneity objective in clustering does not have the same meaning as training error in supervised learning tasks. There is no definition of accuracy for clustering either.

Chapter 12

More Clustering Methods

[Slides \(pdf\)](#)

[Video \(Panopto\)](#)

Recall in the last chapter that we introduced the concept of a **cluster**, a centroid with a spread of a specific shape and size. We also went over some limitations of the k-means algorithm. In this chapter we will look at a more diverse set of clustering techniques that allow us to overcome some of the limitations of k-means.

12.1 Mixture Models

One example of a mode of clustering that can generalize better than the k-means algorithm is a mixture model. Each cluster can have its own probability distribution. One example of a probability distribution for clusters is **Gaussian Mixtures**. The best shape and size for each cluster is learned via a technique called **Expectation Maximization**, though we will not go in depth into it. Since Gaussian Mixtures allow for different shapes and sizes of clusters, we can allow **soft assignments** to clusters. This means that every datapoint has a probability associated with how much it belongs to a cluster.

Example(s)

A **soft assignment** may be useful for categorizing news articles. A news article could have a 54% chance it is about world news, 45% chance it is about science, and 1% chance it is a conspiracy theory.

Therefore, this leaves us with two types of clustering.

12.1.1 Hard Clustering

Hard clustering is clustering that has no overlaps. An example of hard clustering is K-means, because every datapoint belongs to at most one cluster. Hard clustering is a subset of soft clustering.

12.1.2 Soft Clustering

Soft clustering does allow for overlap, though clusters do not necessarily have to overlap. As Gaussian Mixtures use probabilities to cluster data, clusters may overlap.

When using a Gaussian Distribution to model our clusters, each cluster is defined by a mean μ and a covariance σ . Hence, clusters can be unique in shape and size.

12.2 Hierarchical Clustering

Hierarchical Clustering takes advantage data that might have a hierarchical structure.

Definition 12.1: Dendrogram

A **dendrogram** is a type of tree diagram that is used for a hierarchical ordering of things.

We can organize data into a hierarchical structure using one of the two approaches:

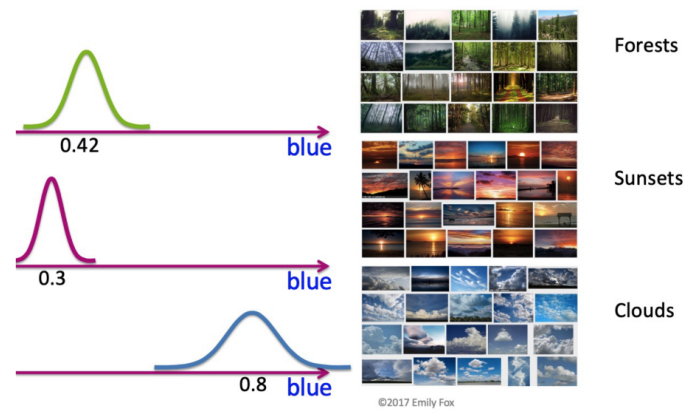


Figure 12.63: Taking a look at how "blue" each pixel of an image is, we have soft clusters resulting from a Gaussian distribution.

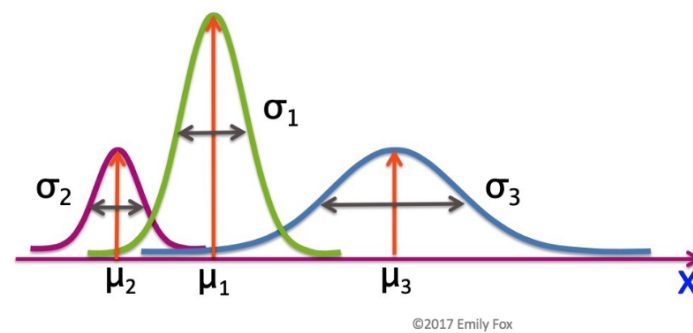


Figure 12.64: Gaussian mixture components represent unique clusters specified by mean and covariance.

Definition 12.2: Divisive

The **divisive** algorithm, a.k.a. **top-down** starts with all the data in one big cluster and then recursively splits the data into smaller clusters.

Definition 12.3: Agglomerative

The **agglomerative** algorithm, a.k.a. **bottom-up** starts with each datapoint in its own cluster and merges clusters until all points are in one big cluster.

12.2.1 Divisive Clustering

To structure our data into a dendrogram, we might adapt the k-means algorithm to do divisive clustering. We could run the k-means algorithm on one big cluster, and then running it on each of those smaller clusters, and keeping going until we have the desired number of sub-clusters. This is one option out of many divisive

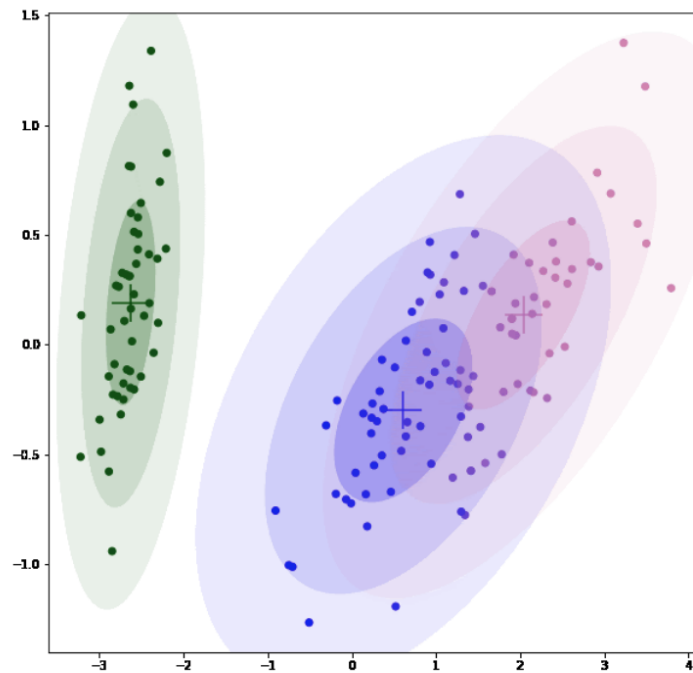


Figure 12.65: The distribution of each cluster gives it a unique shape.

clustering techniques as there are a number of choices we must consider:

- **Algorithm:** K-means? Or something else?
- **Clusters per split:** How many subclusters result from splitting one cluster?
- **Stopping:** When do we stop splitting? It could be when we reach a **max cluster size**, or a **max cluster radius**, or a **specified number of clusters**.

12.2.2 Agglomerative Clustering

Agglomerative clustering can be summarized by the following algorithm:

1. Initialize each point in its own cluster.
2. Define a distance metric between clusters.
3. While there is more than one cluster present, merge the two closest clusters.

With agglomerative clustering, we have to make the choice of which distance metric to use. As an example, we can use the **single linkage** metric, which determines the distance between two clusters as the distance between the closest two points of those clusters:

$$distance(C_1, C_2) = \min_{x_i \in C_1, x_j \in C_2} d(x_i, x_j) \quad (12.27)$$

The above equation states that the distance between Cluster 1 and Cluster 2 is some x_i in Cluster 1 and some x_j in Cluster 2 where those two points are the closest possible pair between the two clusters.

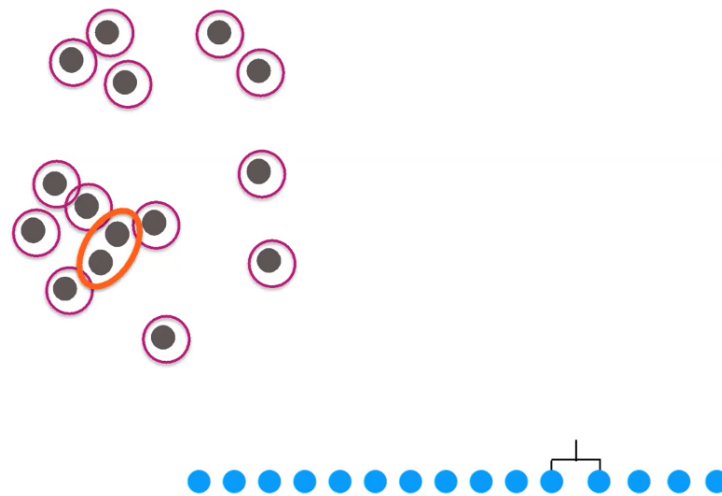


Figure 12.66: In the above figure, we have a set of datapoints in the top left. We want to organize them into a dendrogram (in blue) using agglomerative clustering, so we will group datapoints together in the dendrogram (bottom right) as we merge clusters (top left). To start off, every cluster has one single datapoint. After merging two of the closest clusters together, notice how one cluster (in red) now has two datapoints. A branch on the dendrogram connecting these datapoints has been drawn to reflect this change.

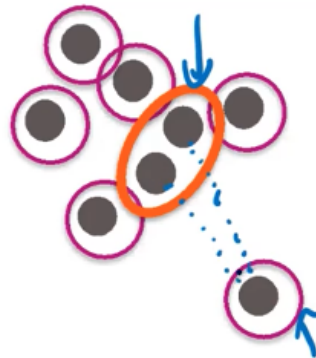


Figure 12.67: Now that one of our clusters has more than one datapoint in it, we can utilize the single linkage distance metric. If we want to measure the distance between the red cluster A and the cluster at the bottom-right B, we have two points to compare. Clearly, the datapoint at the bottom of the red cluster A is closer to cluster B.

Since the single linkage distance metric compares the two closest points between two clusters, we are able to draw complex cluster shapes.

Now, to figure out which two clusters to merge, we have to compare the closest datapoint pairs between every combination of clusters. Once we find the closest datapoint pairs between the two closest clusters, we merge these clusters and repeat until there is only one cluster. As we keep merging until we get one cluster, we complete our dendrogram. One thing to notice is that the higher the connecting line between two point is in the dendrogram, the further away they are in the cluster.

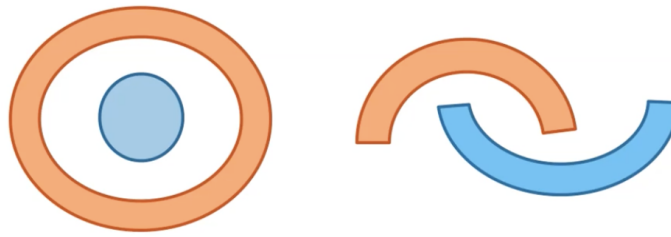
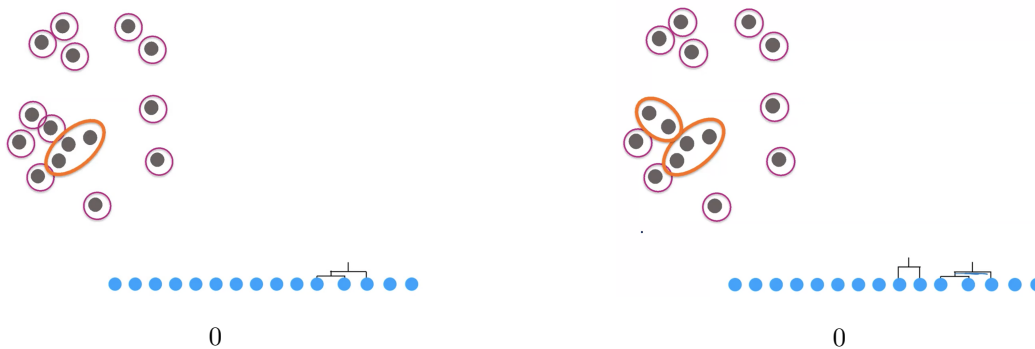


Figure 12.68: A doughnut and a crescent clustering achieved by the single-linkage distance metric in agglomerative clustering. Due to single linkage, before every clustering, we look at the two closest points between the two closest clusters. Because of this, the ring that is the doughnut shape can be formed distinctly from the doughnut's center, as every time we add a point to the ring, it happens to lie on the circumference of the ring and is closer to the ring than the center of the doughnut is to any of the ring points.



This distance measure is important because it allows us to mark specific heights within our dendrogram that specify cluster separations, if we want to preserve some clusters.

Final Notes on Agglomerative Clustering

There are a number of choices we must make when considering the practical applications of agglomerative clustering. Firstly, the single linkage distance metric isn't the only metric there is. We could also use a **complete linkage** or a **centroid linkage**. The complete linkage computes the distances of clusters via the *farthest* points of each cluster away from each other, and the centroid linkage uses the center point of each cluster. We also have to think about how to define our D^* , or where to cut the dendrogram, i.e. how many clusters to preserve. In general, fewer clusters are more interpretable, but if we only have one cluster then we might as well not have clustered anything at all. This is why we cut the dendrogram, so that we can highlight key categories of the data. Finally, we cannot forget about outliers. We might set a distance threshold to pick out outliers, or come up with a more complex algorithm all together. The key takeaway from all of these choices we must make is that in the end there is no one correct answer, but rather what yields the most practical and interpretable results given our data and purposes.

The runtime of the algorithm we described above is $O(n^2 \log(n))$. In this class you do not need to fully understand runtime analysis, but this runtime comes from the fact that this algorithm compares every single point to every single other point, and sorts each of those comparisons. The **triangle inequality**, a theorem that states that "for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side", allows us to bypass some of these comparisons, making the best known

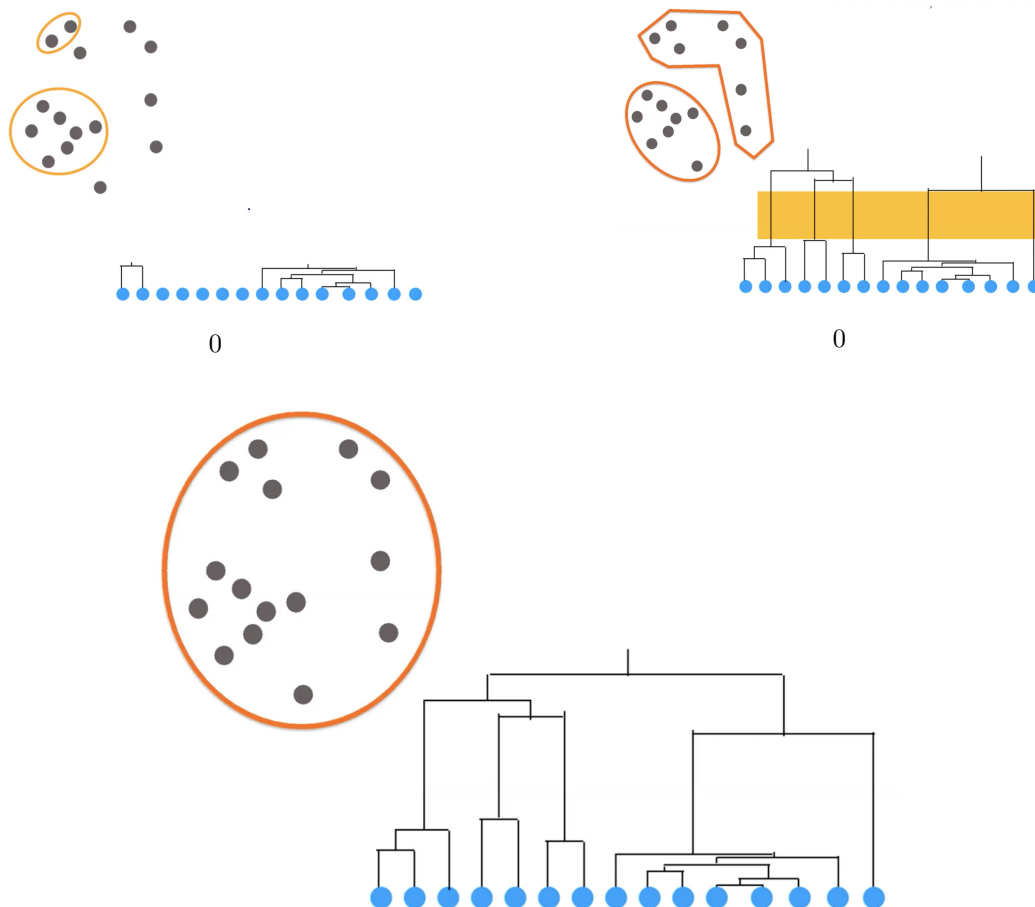
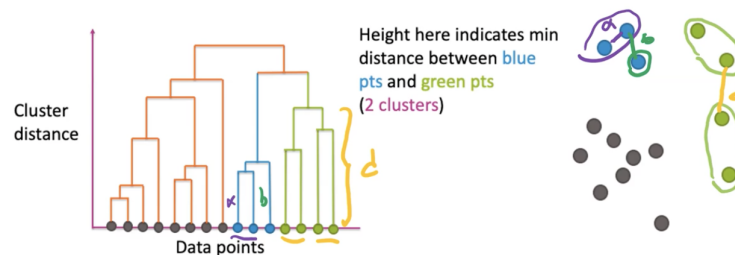


Figure 12.69: The final cluster, with the completed dendrogram.

Figure 12.70: The height of the branches that connect datapoints gives is proportional to the distance between clusters. See distances of α, b, d .

optimal runtime for agglomerative clustering $O(n^2)$.

12.3 Missing Data

In all types of data analysis, the problem of missing data is extremely common. In this section we will discuss our options in the face of missing data.

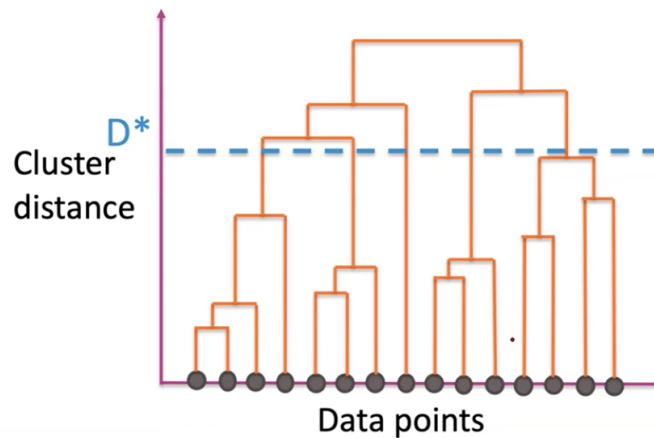


Figure 12.71: If we want to preserve some clusters, we can "cut" the dendrogram at a specific height D^* depending on how many clusters we want. The above example would preserve 5 unique clusters.

12.3.1 Strategy 1: Skip

We can simply ignore missing data by discarding data with missing elements in it.

Pros:

- Very easy to understand.
- Can be applied to any model.

Cons:

- We could be removing useful information.
- We don't know if we should remove one datapoint with a missing feature, or remove that feature from the dataset entirely to keep the datapoint.

12.3.2 Strategy 2: Sentinel Values

Instead of cutting out missing data, we can replace it with a value that indicates it is missing.

Pros:

- Very easy to understand.
- Works well with categorical data, as "missing" becomes its own category.

Cons:

- Does not work well with numeric features.

12.3.3 Strategy 3: Imputation

With imputation, we utilize some heuristic to "predict" what a missing value may be. A simple approach for categorical data could be simply just using the most popular category as the value for the missing data, and for numeric data it could be using the median or the mode. A more complex approach could be using machine learning to estimate a prediction of what that missing value should be based on the relationship

between other features in the data.

Pros:

- Simple to implement if using the simple approach.
- Can be applied to any model.

Cons:

- May result in systematic errors. Maybe there is a deeper-rooted issue for why the values are missing, and we shouldn't just fill them in as if nothing happened.

12.3.4 Strategy 4: Modify Algorithm

We can completely redesign the model so that it accounts for missing values. For example, we could recreate a tree such that it has a special branch for missing values.

Pros:

- This is very similar to the sentinel value approach so it is likewise simple.
- It works for numeric features.
- It can yield more accurate predictions.

Cons:

- You have to implement an entirely new model. This could be pretty difficult for some model types.