Contents

Chapter 1
Introduction / Regression

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?

Data
ML Algorithm
Intelligence

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 5 important areas of machine learning:

1. Regression
2. Classification
3. 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:

$$(x, y)^{(1)}, (x, y)^{(2)}, ..., (x, y)^{(n)} \tag{1.1}$$

So, $x^{(1)}$ is a list (or *vector*) 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 observation and d be the number of features we have in a single observation $x^{(i)}$:

$$x^{(i)} = X_1^{(i)}, X_2^{(i)}, ..., X_d^{(i)} \tag{1.2}$$

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

For simplicity's sake, we will consider an observation 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:

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

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.

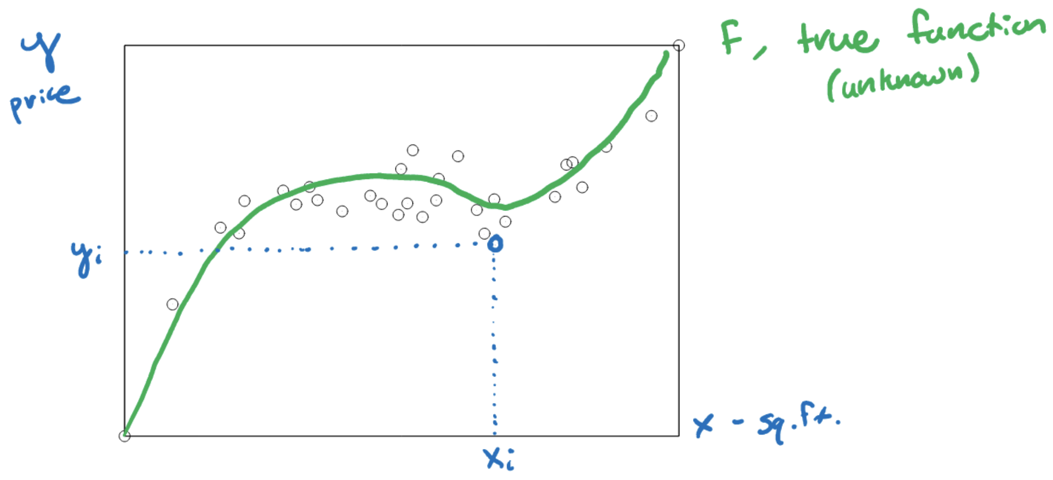


Figure 1.3: The dataset is plotted on a graph where the x-axis is square footages and the y-axis is house prices.

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 assume that the relationship between x and y is linear:

$$y^i \approx f(x^i) = wx^i + b \tag{1.4}$$

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

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

Note here the variables that have hats ˆ 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 $\hat{b}$.

While there also exists a closed-form estimate of the parameters of linear regression (which means we can directly compute our parameter estimates), for this course, we'll be using a different algorithm to estimate our parameters. 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 stops decreasing.

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).

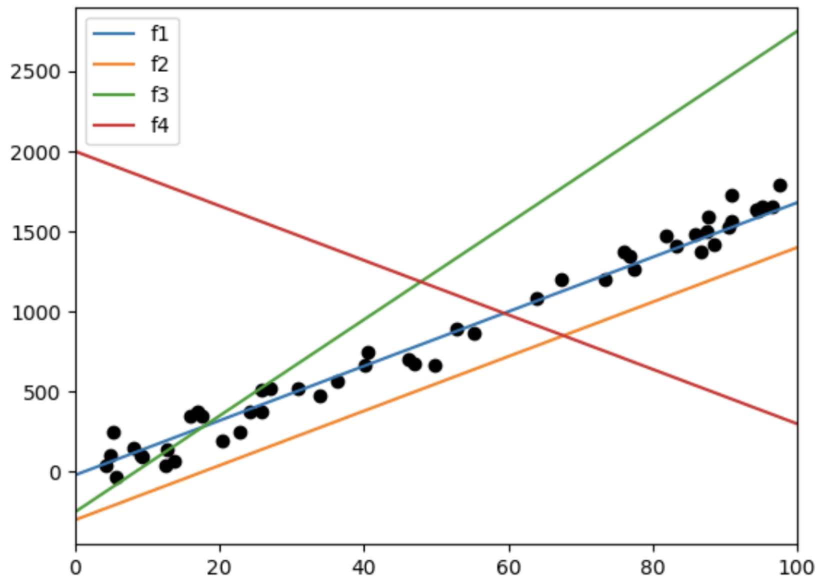


Figure 1.4: Regression starts with a random line and then tries to gradually make it better and better based on error calculation.

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:

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

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:

$$\hat{w}, \hat{b} =_{w,b} MSE(w, b) \tag{1.7}$$

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.

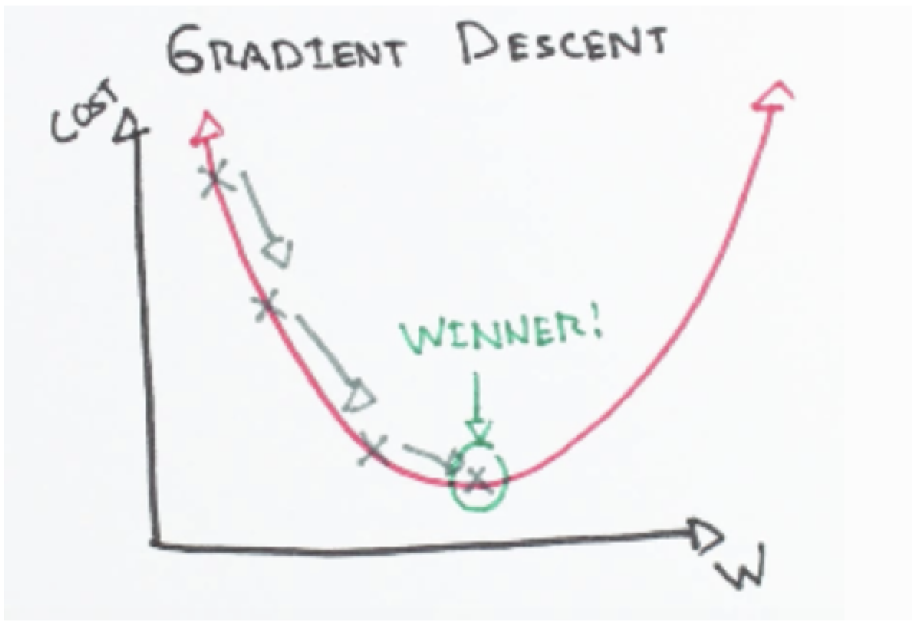


Figure 1.5: You can think of Gradient Descent as rolling a ball down a hill.

Chapter 2
Assessing Performance; Bias + Variance Tradeoff

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**: $h(X_i) = (h_0(X_i), h_1(X_i), ..., h_D(X_i))$
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**: $\hat{w} = \underset{w}{\operatorname{argmin}} \text{MSE}(w)$
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.

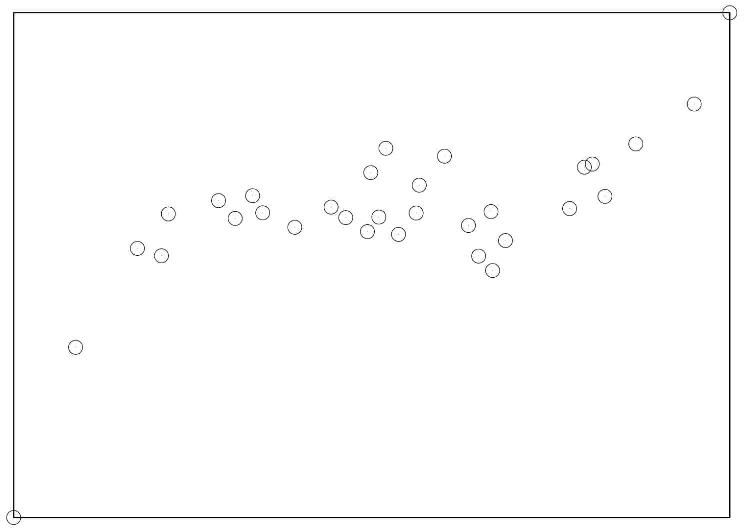


Figure 2.6: Sample data. Note that the true relationship between x and y is nonlinear.

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:

$$y = wx + b \tag{2.8}$$

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

$$y = w_0 + w_1 x + w_2 x^2 + ... + w_d x^d$$

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.

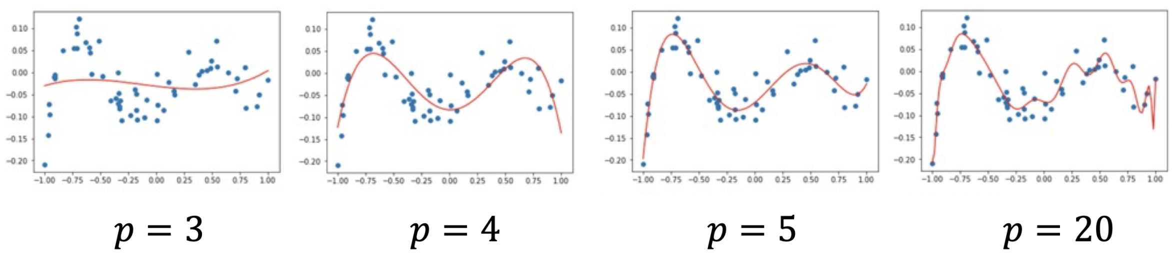


Figure 2.7: Fitting different complexity models to data.

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.

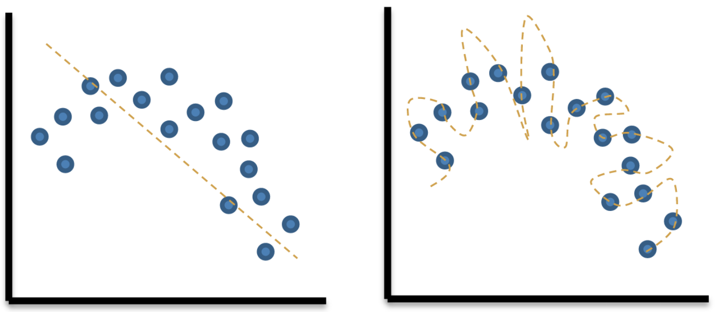


Figure 2.8: Often low complexity or simple (on the left) models tend to have high bias and high complexity models (on the right) tend to have high variance.

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:

$$\text{Error} = \text{Biased squared} + \text{Variance} + \text{Irreducible Error} \tag{2.9}$$

Or more formally as:

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

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.

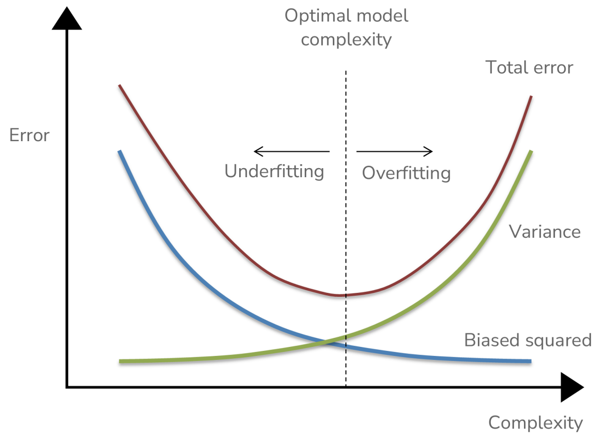


Figure 2.9: Bias and variance tradeoff

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.

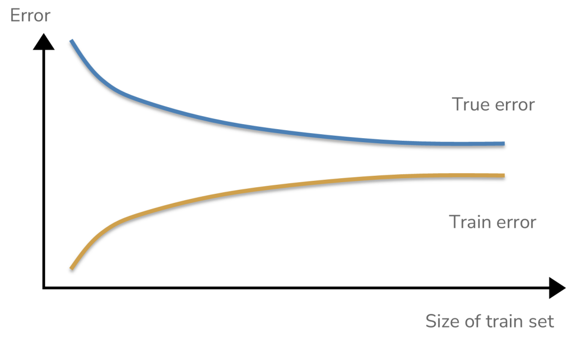


Figure 2.10: Error versus the size of the training

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 ten 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

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:

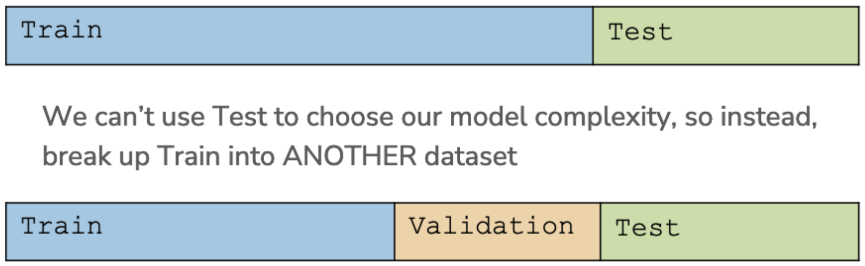

Train
Test
We can't use Test to choose our model complexity, so instead, break up Train into ANOTHER dataset
Train
Validation
Test

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.

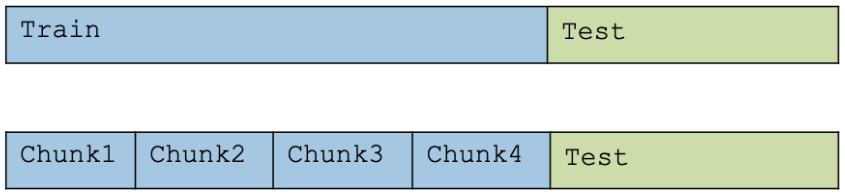


Figure 3.13: The training set is divided into a number of chunks.

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:

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

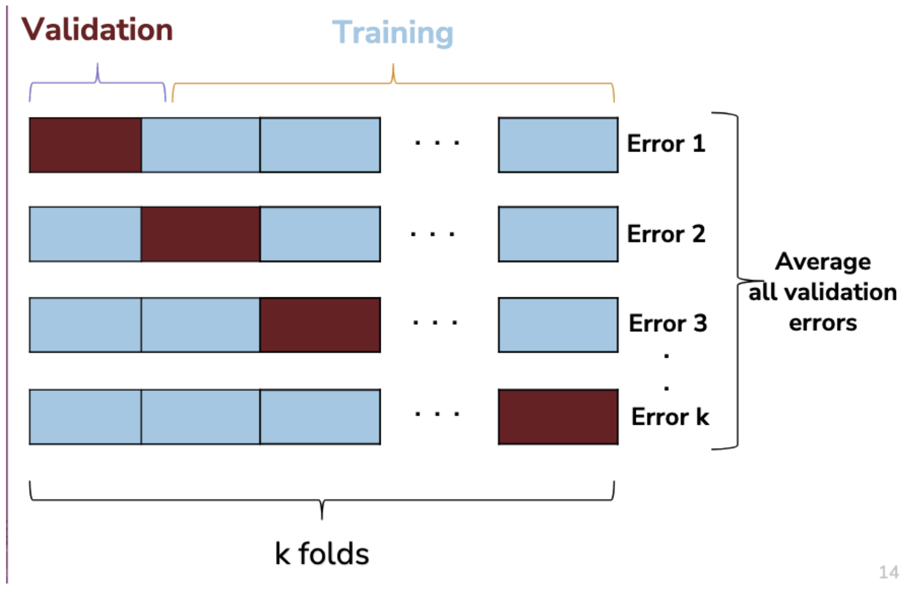


Figure 3.14: The validation error is calculated by averaging the error from k different samples of validation data.

```
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.

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 out weights $\hat{w}$ by using a quality metric that minimizes the loss:

$$\hat{w} =_w L(w)$$

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:

$$\hat{w} =_w L(w) + \lambda R(w)$$

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:

$$R(w) = |w_0|^2 + |w_1|^2 + ... + |w_d|^2 = ||w||_2^2 \tag{3.11}$$

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:

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

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

if $\lambda = 0$:

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

if $\lambda = \infty$:

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

⇒ 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 ∞:

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

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.

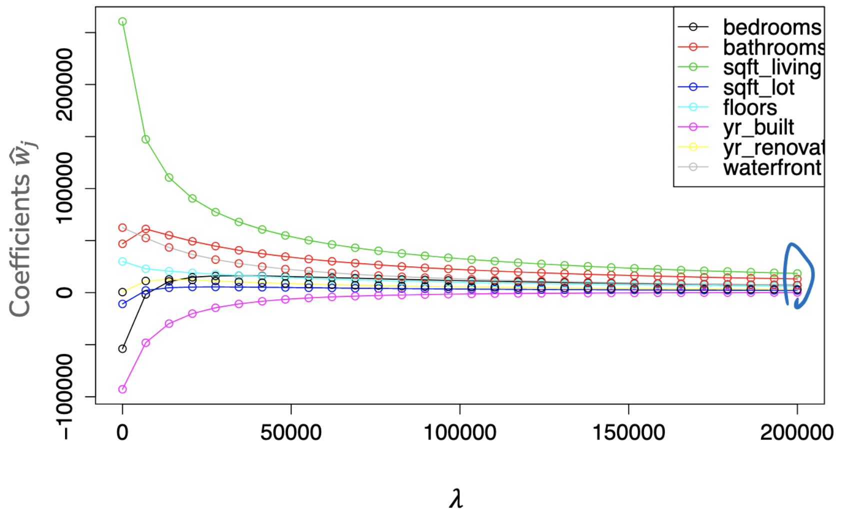


Figure 3.16: The effect of λ on the size of the weights.

3.3.3 The Intercept

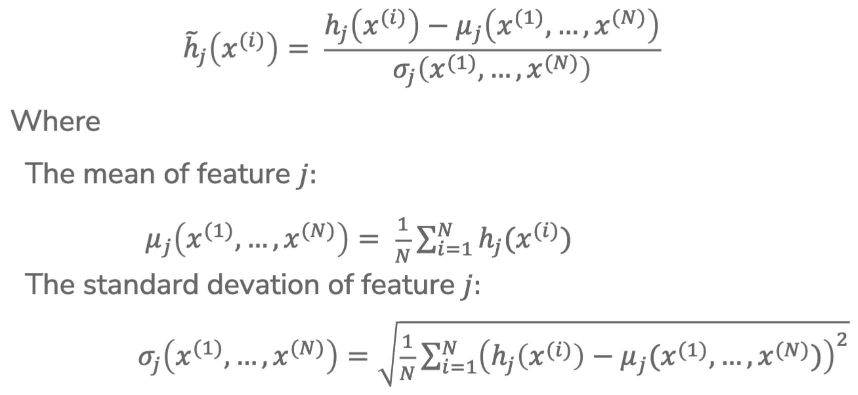
When regularizing all the weights, we also happen to regularize w_0, the intercept term. This doesn't make sense because we shouldn't penalize a model for having a higher intercept since that may mean that the y value units are just really high. The intercept also doesn't affect the curvature of the 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:

In particular, normalization ensures that all features have mean zero and standard deviation 1.



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

Where

The mean of feature j:

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

The standard devation of feature j:

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

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

4.1 Ridge Regression Recap

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

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

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 validations 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 hyperparemeter 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 $\lambda$ in $\lambda$s do  Train a model using using Gradient Descent
– $\hat{w}_{\text{ridge}(\lambda)} = \underset{w}{\operatorname{argmin}}\, \text{MSE}(W) + \lambda||w||_2^2$
Compute validation error
– $validation\_error = \text{MSE}_{val}(\hat{w}_{\text{ridge}(\lambda)})$
Track $\lambda$ with smallest $validation\_error$  return $\lambda^*$ and estimated future error $\text{MSE}_{val}(\hat{w}_{\text{ridge}(\lambda^*)})$
```

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 Scaling Features Recap

Features in dataset can have different units, and if we change our features to different units, for example, change meter to kilometer, the corresponding parameter for this feature is also changed. Feature scaling is important before applying regularization, because regularization penalizes large parameters. Therefore, if the parameter of a feature gets larger because of unit change, its parameter would be penalized more by regularization. This is not ideal, since we want regularization to work the same regardless of the units. The solution to solve this problem is using feature normalization.

Definition 4.1: Feature Normalization

$$\tilde{h_j}(x) = \frac{h_j(x) - \mu_j}{\sigma_j}$$

where μ_j is mean of feature j in train set and σ_j is standard deviation of feature j in train set.

Scaling features is always a good practice for data preprocessing. The only downside is that the features you have after scaling are no longer as interpretable as before.

To normalize data, we calcuate mean and standard deviation only using train set.

4.3 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 select features would be to try every possible combination of features and see how to 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.

4.4 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.

$$\hat{w} = \operatorname*{argmin}_w L(w) + R(w) \tag{4.13}$$

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 4.2: Lasso Regression Model

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

Remember the definition of a p-norm.

Definition 4.3: p-norm

$$||w||_p^p = |w_0|^p + |w_1|^p + ... + |w_d|^p$$

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.

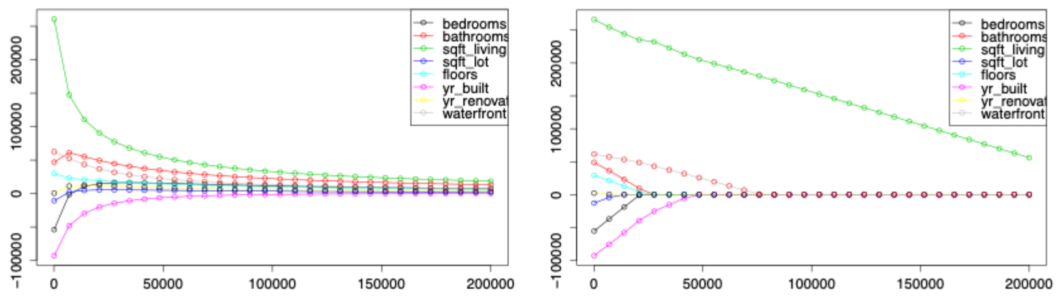


Figure 4.18: Ridge (right) and lasso (left) coefficient paths.

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.

In Figure 4.2, 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

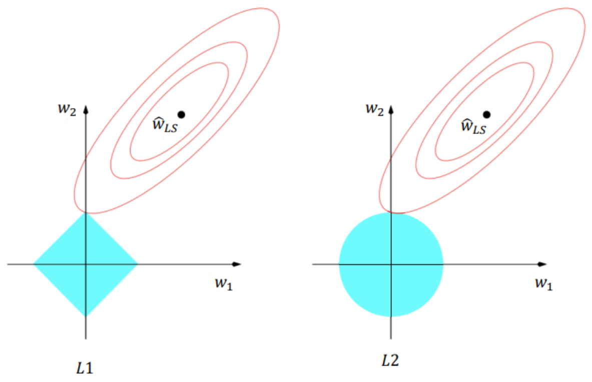


Figure 4.19: Visualizing sparse solutions with L1 on left and L2 on right.

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 4.4: 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 betters.

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

Chapter 5
Classification

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.

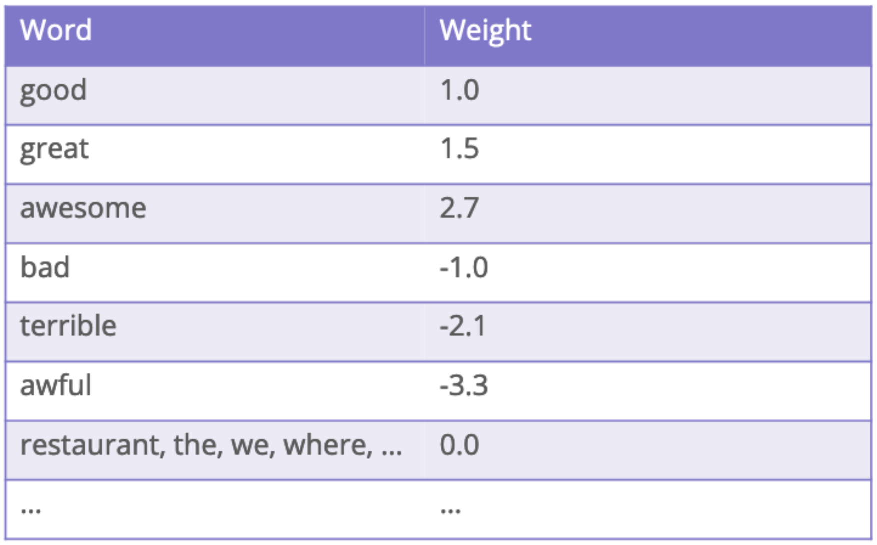
There are unfortunately a number of limitations with this approach. First off, how do we get the list of positive and negative words? If we want a robust list covering the entire dataset, do we have to go through each review ourselves and painstakingly write this list up from scratch? And also, what 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 Regression - Implementation 2

We will try to account for the issue of varying sentiment in a Linear Regression approach. Unlike the Simple Threshold Classifier, the Linear Regression 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.

$$\text{predicted sentiment} = \hat{y} = \sum_{i=0}^{D} w_i h_i(x)$$

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:

$$\hat{y} = \text{"awesome"} * 1 + \text{"great"} * 1 + \text{"terrible"} x 1$$
$$\hat{y} = 2.7 * 1 + 1.5 * 1 + -2.1 * 1$$
$$\hat{y} = +2.1$$

Issue: Recall that the labels for sentiment analysis are binary: positive/negative, ie. +1 or -1. However, linear regression predict continuous values.

5.3 Linear Classifier - Implementation 3

To solve the continuous-value output problem in Linear Regression, we introduce Linear Classifier. The idea is to only predict the **sign of the output**. This means that if the output of Linear Regression is positive, then we predict positive; otherwise, we predict negative. We define a score function $Score(x)$ to be the output from Linear Regression.

$$Score(x) = \sum_{i=0}^{D} w_i h_i(x) = \mathbf{w}^T \mathbf{h}(x)$$

Then, by using Linear Classifier,

$$\text{predicted sentiment} = \hat{y} = sign(Score(x))$$

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.

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

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

We will also use the notation

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

Figure 5.21: Notation for linear classifiers.

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.

Beside the limitation discussed above, it is hard to train Linear Classifier to get the optimal weights. If we use MSE as loss function, the derivative of the sign function is 0, so Gradient Descent will no longer work. Therefore, we need Logistic Regression (next chapter) to solve this problem so that we are able to learn weights.

5.4 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:

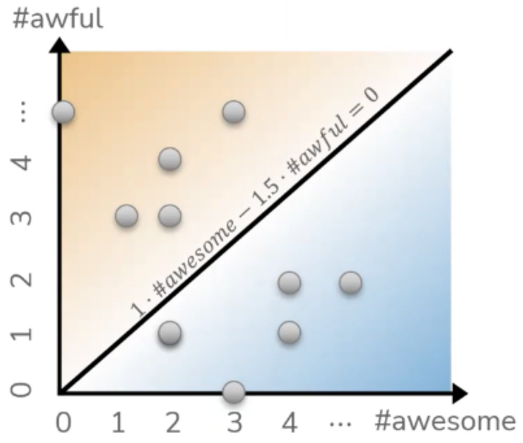


Figure 5.22: A linear decision boundary between two features.

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:

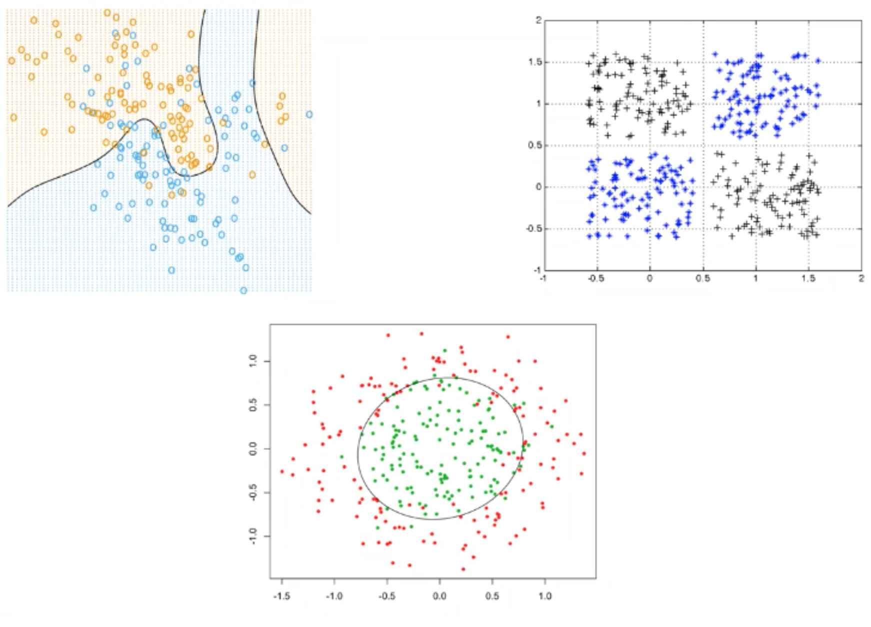


Figure 5.23: Visualization of nonlinear decision boundaries produced by more complex classification models.

5.5 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:

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

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

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:

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

5.6 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.

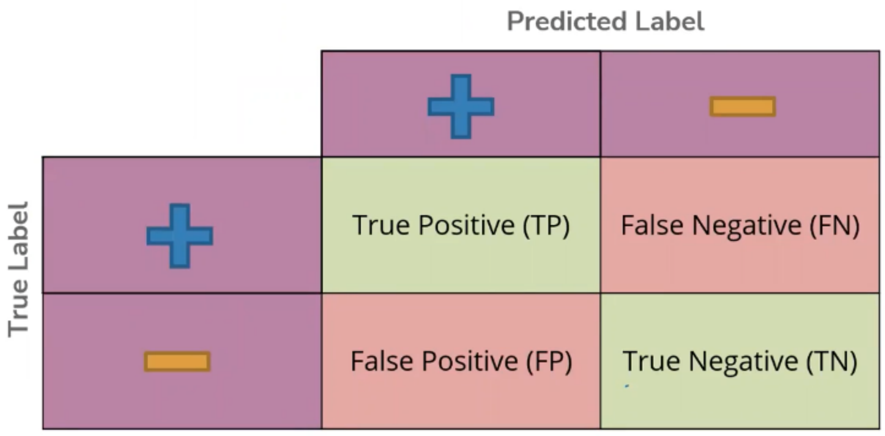


Figure 5.24: A confusion matrix for binary classification.

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.

5.7 Learning Theory

How does adding more data affect the error rate of our models? Previously, we discussed a limitation of the

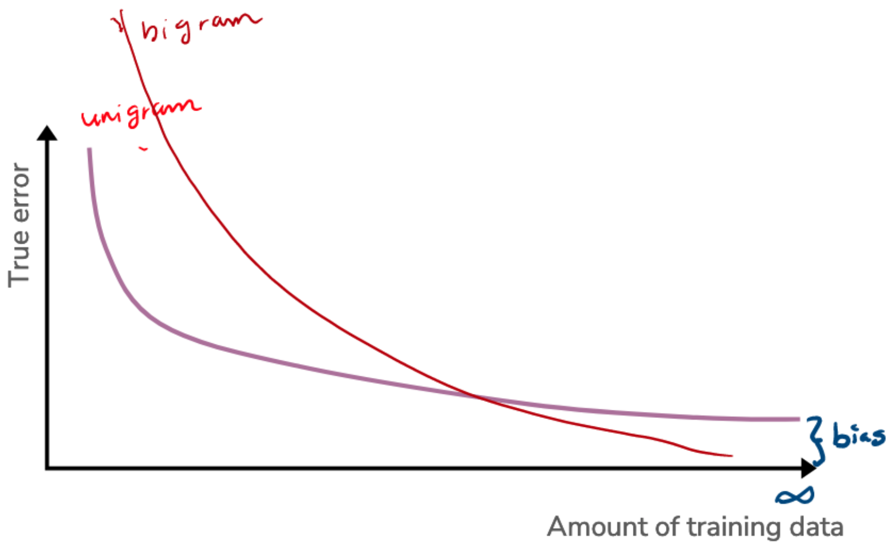


Figure 5.25: The learning curve for different data parsing types.

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.

5.8 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

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.

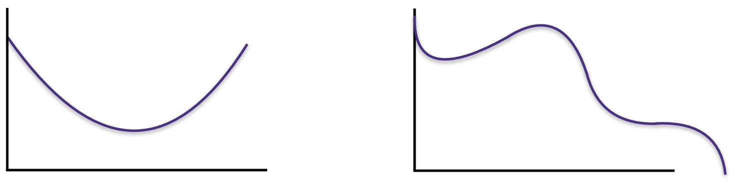


Figure 6.26: Convex (right) and non convex (left) functions.

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.

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

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

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

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

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

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:

$$\begin{aligned} P(y=-1 \mid x_i, w) &= 1 - P(y=+1 \mid x_i, w) && (6.15)\\ &= 1 - \frac{1}{1+e^{-w^T h(x_i)}} && (6.16)\\ &= \frac{e^{-w^T h(x_i)}}{1+e^{-w^T h(x_i)}} && (6.17) \end{aligned}$$

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).

$$\hat{w} = \underset{w}{\operatorname{argmax}}\, \ell(w) \tag{6.18}$$

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

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

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: 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 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 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:

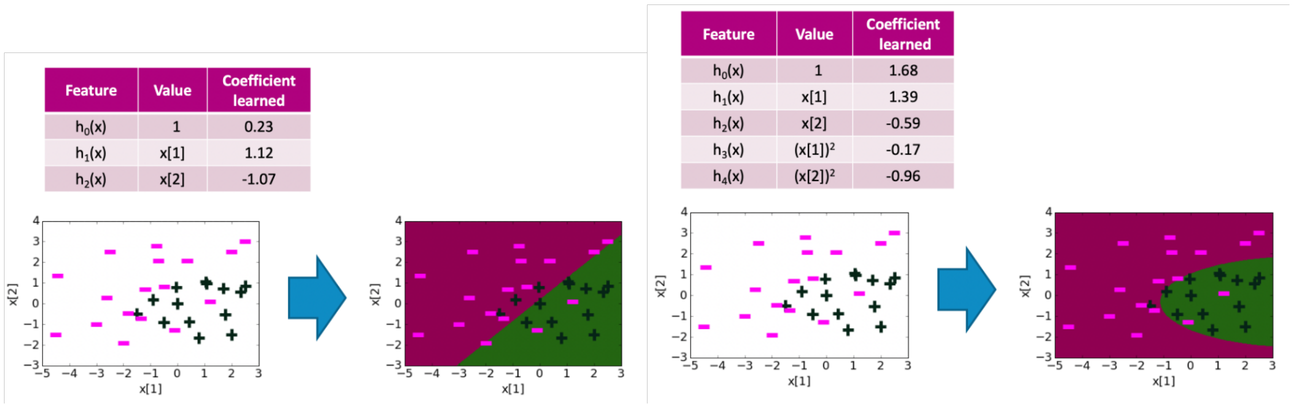
$h_1(x)$ = number of "awesome" words
$h_2(x)$ = number of "awful" words

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

$h_1(x)$ = number of "awesome" words
$h_2(x)$ = number of "awful" words
$h_3(x)$ = (number of "awesome" words)^2
$h_4(x)$ = (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.

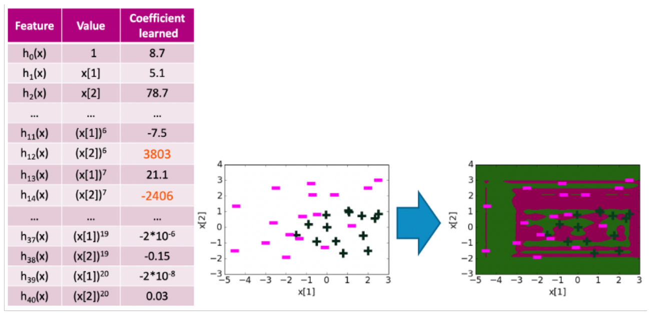
It is important to remember that while the plots from above show the decision boundaries, we are still representing our result as a probability / certainty of the model's output. When using regularization, smoothness can be introduced to better represent areas of certainty and uncertainty. Increasing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



Feature	Value	Coefficient learned
$h_0(x)$	1	0.23
$h_1(x)$	x[1]	1.12
$h_2(x)$	x[2]	-1.07

Feature	Value	Coefficient learned
$h_0(x)$	1	1.68
$h_1(x)$	x[1]	1.39
$h_2(x)$	x[2]	-0.59
$h_3(x)$	$(x[1])^2$	-0.17
$h_4(x)$	$(x[2])^2$	-0.96

Figure 6.27: Simple and quadratic classification boundaries



Feature	Value	Coefficient learned
$h_0(x)$	1	8.7
$h_1(x)$	x[1]	5.1
$h_2(x)$	x[2]	78.7
…	…	…
$h_{11}(x)$	$(x[1])^6$	-7.5
$h_{12}(x)$	$(x[2])^6$	3803
$h_{13}(x)$	$(x[1])^7$	21.1
$h_{14}(x)$	$(x[2])^7$	-2406
…	…	…
$h_{37}(x)$	$(x[1])^{19}$	$-2*10^{-6}$
$h_{38}(x)$	$(x[2])^{19}$	-0.15
$h_{39}(x)$	$(x[1])^{20}$	$-2*10^{-8}$
$h_{40}(x)$	$(x[2])^{20}$	0.03

Figure 6.28: Overfit classification boundaries

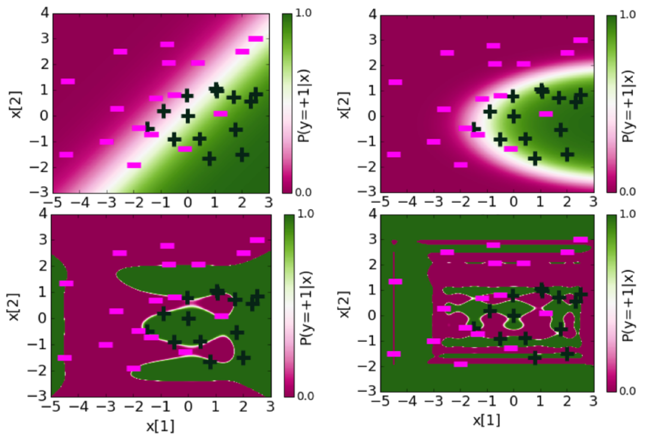


Figure 6.29: Regularized plotted probabilities

6.3 Gradient Ascent

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 maximum for the function (equation 6.6) like gradient ascent. In gradient ascent, since we cannot compute the solution with a closed form, we must move towards the maximum step by step.

Algorithm 2 Gradient Ascent

```
Start with random weights $w$

while $w$ has not converged do  $w = w + \alpha \Delta L(w)$  - $\alpha$ = 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 ascent where a learning rate that is too small could take too long to converge but a large learning rate may result in gradient ascent 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 ascent. Often we use values that are exponentially spaced out.

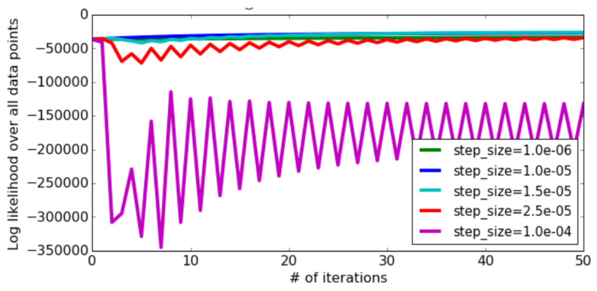


Figure 6.30: Effect of different learning rates.

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 ascent does not need to be fixed, though. Ideally, gradient ascent could ascend 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 ascent where t is the number of iterations.

$$\alpha_t = \frac{\alpha_0}{t} \tag{6.21}$$

To compute the gradient at each iteration for gradient ascent (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 ascent 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 ascent/descent. In practice, this is used widely as most often each step in gradient ascent/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 ascent and the logistic regression model, we have learned that

- 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 ascent/descent must be used as there is no closed form solution to our objective function.
- Learning rate in gradient ascent/descent are important for gradient ascent/descent convergence
- Stochastic gradient ascent/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

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 a another way to explore these probabilities.

7.1 Naïve Bayes [Optional content for Summer 2022]

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:

$$\frac{P(\text{“}\textit{The sushi \&everything else was awesome!}\text{”}\,|\, y=+1)\; P(y=+1)}{P(\text{“}\textit{The sushi \& everything else was awesome!}\text{”})}$$

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(”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 occuring 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(”The sushi & everything else was awesome!” | y=+1) = P(The | y=+1) * P(sushi | y=+1) * P(& | y=+1) * P(everything | y=+1) * P(else | y=+1) * P(was | y=+1) * P(awesome | y=+1)

Our final model is thus:

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

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

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.

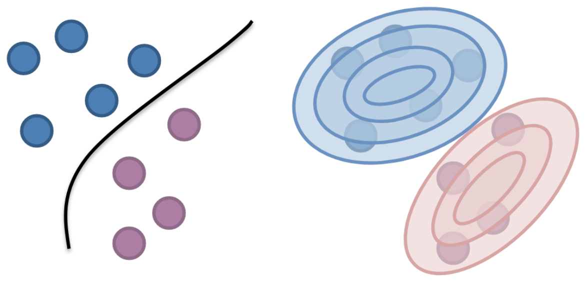


Figure 7.31: On the left visualize a discriminative model, where we only care about optimizing a decision boundary. On the right, we have a generative mode, where the likelihood of each class is also captured.

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.

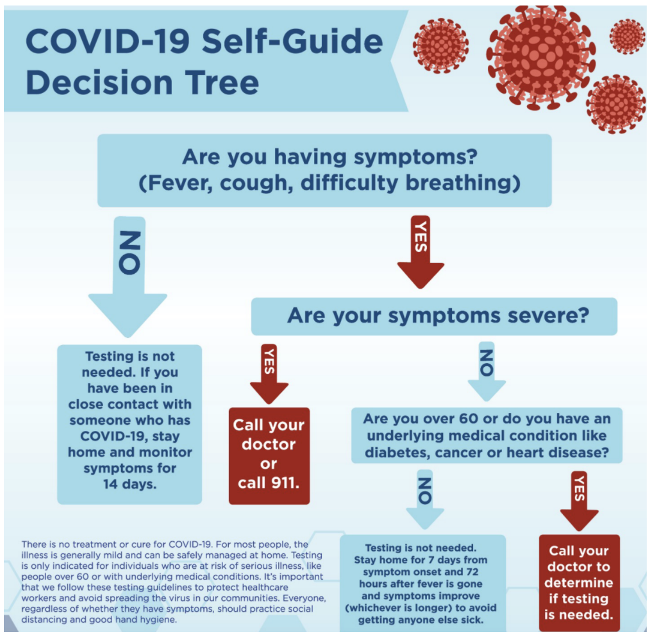
7.2 Decision Trees

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

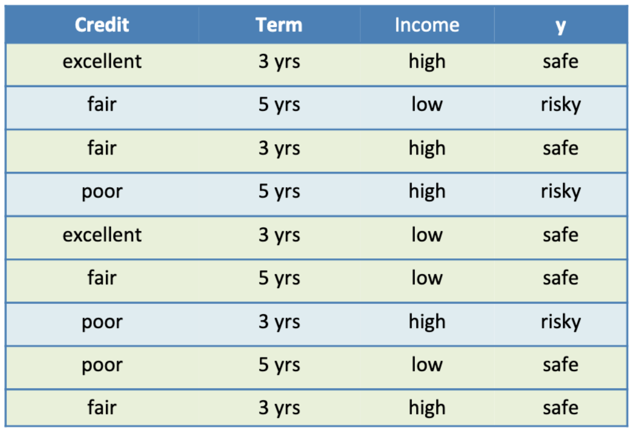
A decision tree is a series of questions that lead to multiple outcomes such that we can explore each question further.

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


COVID-19 Self-Guide Decision Tree
Are you having symptoms? (Fever, cough, difficulty breathing)
NO
YES
Are your symptoms severe?
Testing is not needed. If you have been in close contact with someone who has COVID-19, stay home and monitor symptoms for 14 days.
YES
Call your doctor or call 911.
NO
Are you over 60 or do you have an underlying medical condition like diabetes, cancer or heart disease?
NO
YES
Testing is not needed. Stay home for 7 days from symptom onset and 72 hours after fever is gone and symptoms improve (whichever is longer) to avoid getting anyone else sick.
Call your doctor to determine if testing is needed.

There is no treatment or cure for COVID-19. For most people, the illness is generally mild and can be safely managed at home. Testing is only indicated for individuals who are at risk of serious illness, like people over 60 or with underlying medical conditions. It's important that we follow these testing guidelines to protect healthcare workers and avoid spreading the virus in our communities. Everyone, regardless of whether they have symptoms, should practice social distancing and good hand hygiene.

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



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:

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.

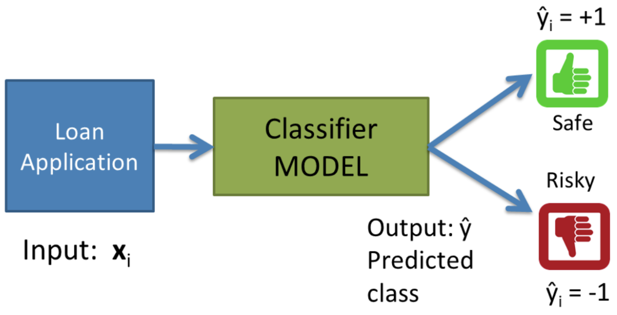


Figure 7.34: A loan application is fed into the model as input x_i. The output will be either $\hat{y}_i = +1$ (safe) or $\hat{y}_i = -1$ (risky).

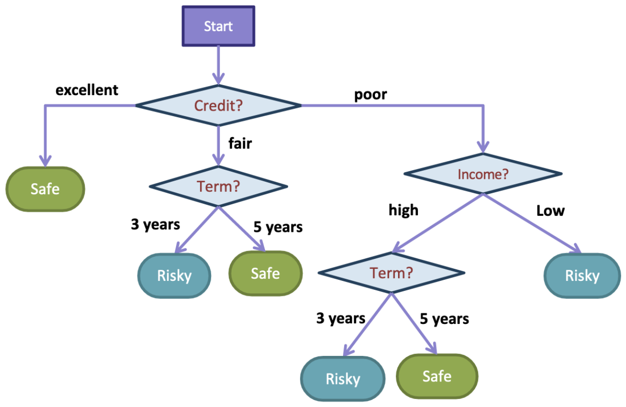


Figure 7.35: A decision tree for determining whether a candidate is safe or risky to lend a loan to. The branch/internal nodes split into possible values of a feature, while the leaf nodes are the final decision, the output class.

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:

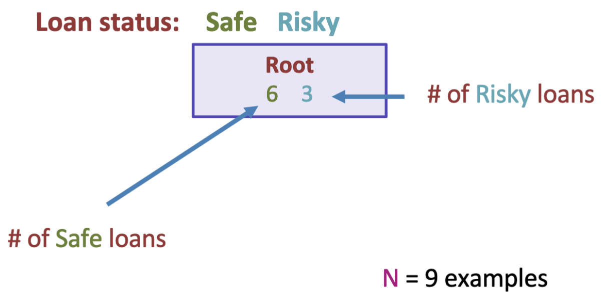


Figure 7.36: The root node of the decision tree stores information about the safe loans and the risky loans. In total, there are 9 loan applications in our dataset.

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 16.83 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.

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:

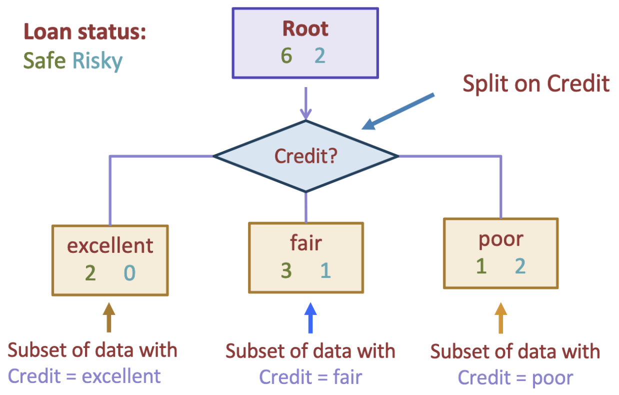


Figure 7.37: The tree splits the training data into different levels of credit history.

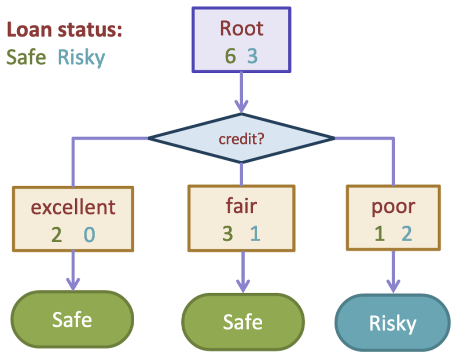


Figure 7.38: The majority class of each node determines the final output.

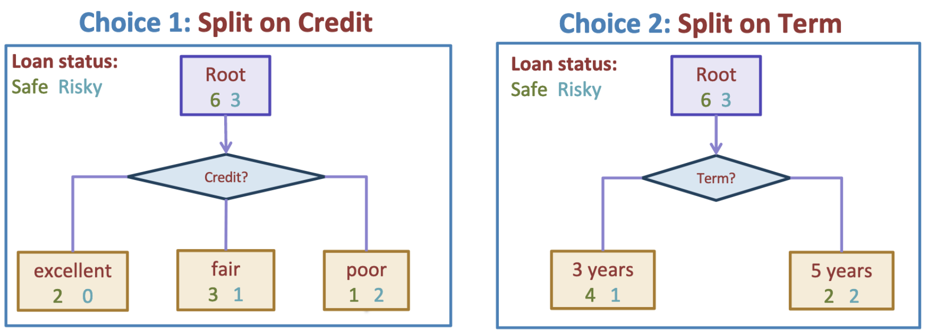


Figure 7.39: A split on Credit versus a split on Term in the decision tree.

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)
    o Given a subset of data M in node
    o For each feature $h_i$ :
        o Compute classification error for a split of M
          according to feature $h_i$ :
    o 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 beuilding the tree. This algorithm can be summarized as:

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

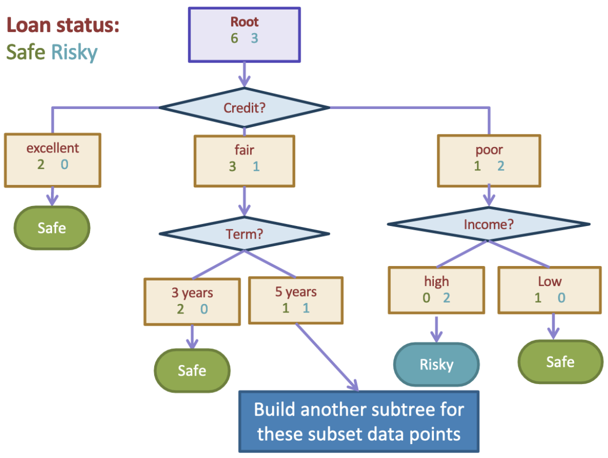


Figure 7.42: Every subtree build from each node will be a subset of the original trainset where the condition of the node is true for the remainder of the tree.

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 $v_1, v_2, ..., v_n$ 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.

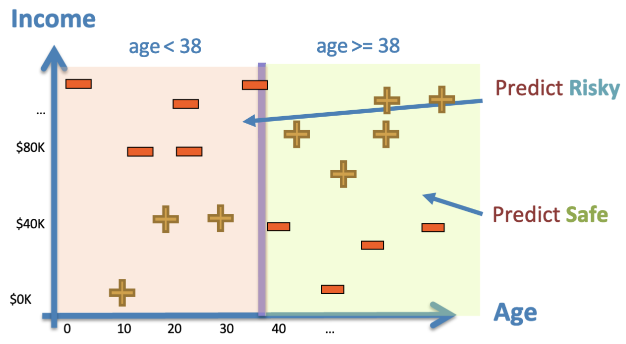


Figure 7.43: For a decision tree, a splitting of a numeric feature will always be along the axis.

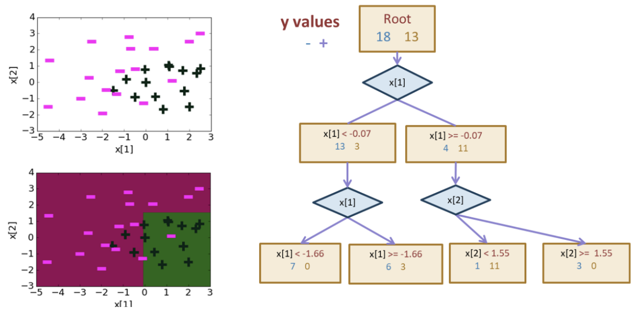


Figure 7.44: If we decide to split further, we continue to split along the axes.

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

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

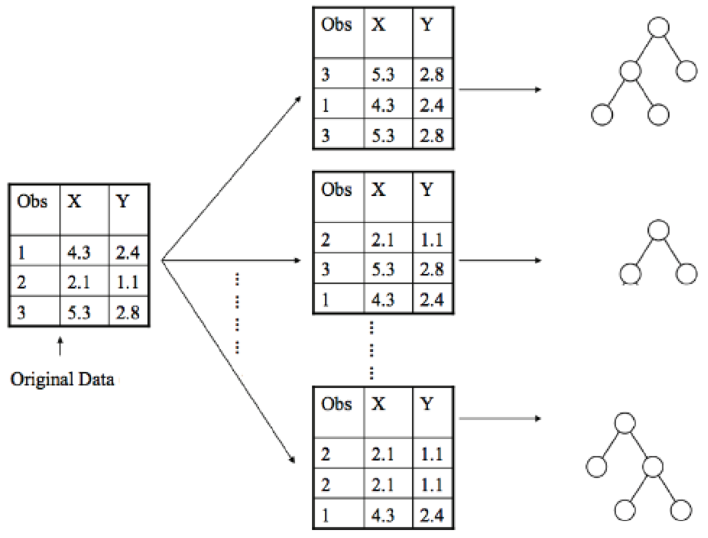
The Random Forest method, also called Bagging, utilizes a collection of Decision Trees. Each decision tree casts a ”vote” for a prediction based on an input and the ensemble predicts the majority vote of all of its trees. Each tree is built from a subset of the training dataset by random sampling the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



Obs	X	Y
1	4.3	2.4
2	2.1	1.1
3	5.3	2.8

Obs	X	Y
3	5.3	2.8
1	4.3	2.4
3	5.3	2.8

Obs	X	Y
2	2.1	1.1
3	5.3	2.8
1	4.3	2.4

Obs	X	Y
2	2.1	1.1
2	2.1	1.1
1	4.3	2.4

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

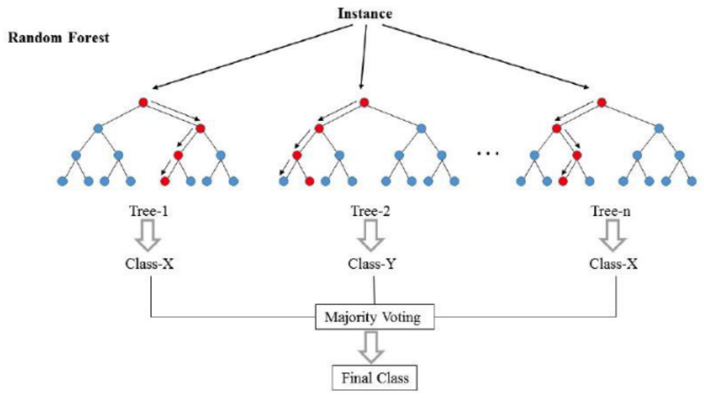


Figure 8.46: Each tree predicts its own final decision based on the input and the majority prediction is selected by the ensemble.

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:

$$WeightedError(f_t) = \frac{\sum_{i=1}^{n} \alpha_i || \{\hat{f}_t(x_i) \neq y_i\}}{\sum_{i=1}^{n} \alpha_i} \tag{8.22}$$

   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:

$$\hat{w}_t = \frac{1}{2} ln(\frac{1 - WeightedError(\hat{f}_t)}{WeightedError(\hat{f}_t)}) \tag{8.23}$$

   (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:

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

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:

$$\hat{y} = \hat{F}(x) = sign(\sum_{t=1}^{T} \hat{w}_t \hat{f}_t(x)) \tag{8.24}$$

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:

$$\alpha_i \leftarrow \frac{\alpha_i}{\sum_{j=1}^{n} \alpha_j} \tag{8.25}$$

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 $\hat{f}_1$.

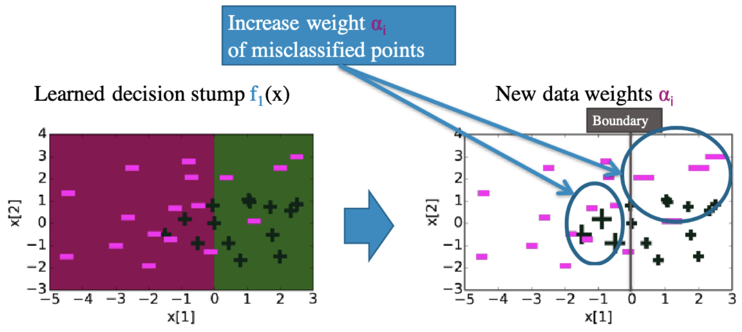


Figure 8.47: After training on the first stump, misclassified datapoints are given a greater weight (drawn larger).

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 $\hat{f}_2$.

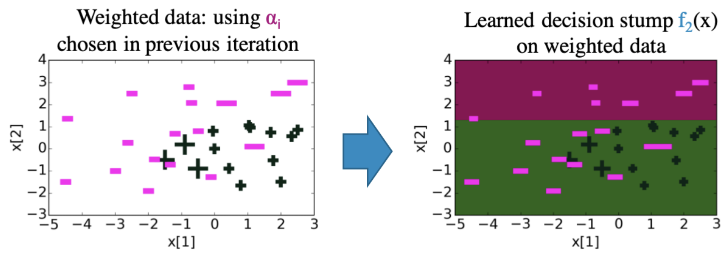


Figure 8.48: $\hat{f}_2$ is predicted based on the weights computed in the previous iteration.

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

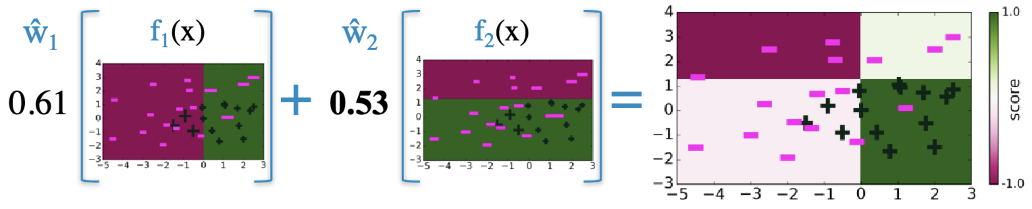


Figure 8.49: The weighted sum of the models $t = 1$ and $t = 2$ yields the weighted decision boundary on the right.

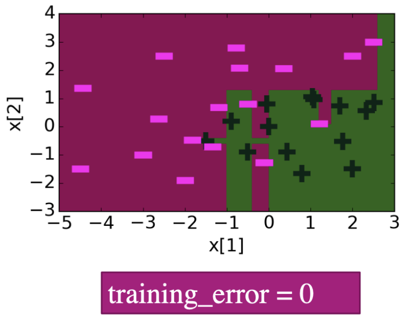


Figure 8.50: Decision boundary for $t = 30$.

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 extrmeley 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.

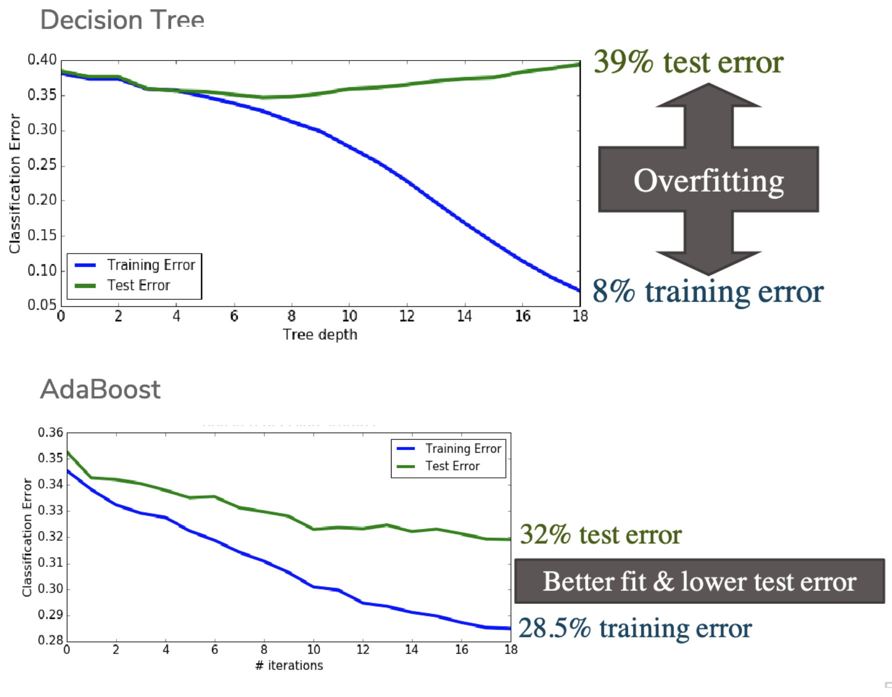


Figure 8.51: The train and test error curves for a standard Decision Tree and AdaBoost.

Chapter 10
Fairness in ML

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 signficant difference in prediction scores for people of different races committing the same crimes.

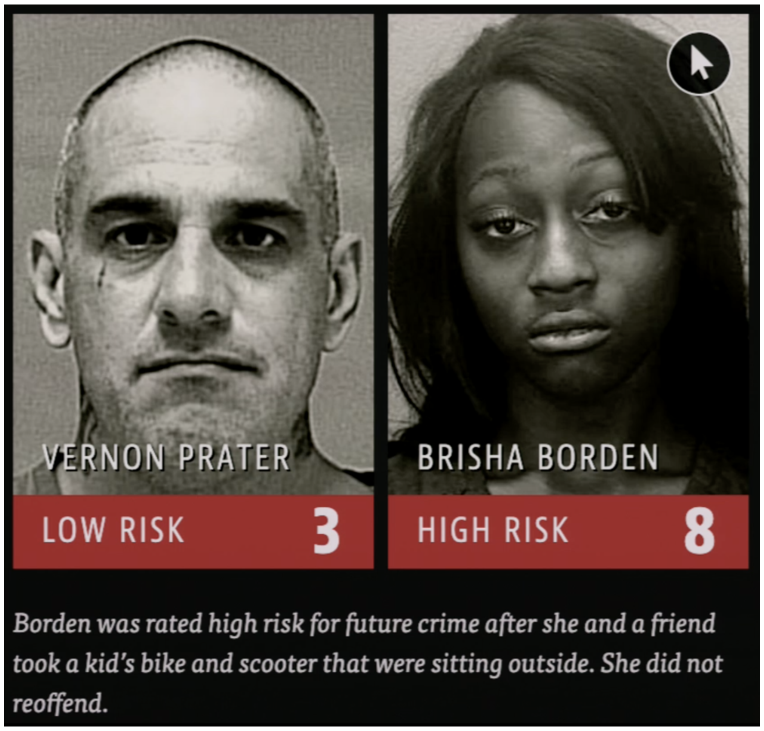

VERNON PRATER
LOW RISK 3
BRISHA BORDEN
HIGH RISK 8
Borden was rated high risk for future crime after she and a friend took a kid's bike and scooter that were sitting outside. She did not reoffend.

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.

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**,

Figure 10.53: Harvard Professor Dr. Latanya Sweeney.

Figure 10.54: Activist and computer scientist Joy Buolamwini. You can watch Buolwamwini's interview here.

measurement bias representation bias, **deployment bias**, **evaluation bias**.

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

Example(s)

Case Study: Distaster Relief
Goal: Accurately assess damage and send appropriate relief resources.
Data: Twitter posts, facebook posts goecoded with coordinates within disaster area, keywords and hashtags related to the storm.
Analysis: Intensity and type of damage by neighborhood.

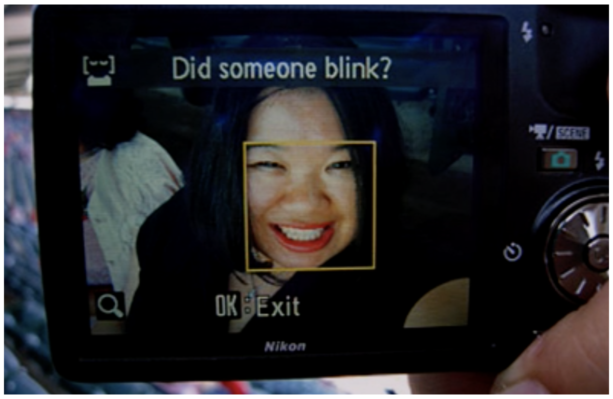


Figure 10.55: Nikon's failure to correctly detect Asian faces.

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

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 $x_1, x_2, ..., x_n$, 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 $z_1, z_2, ..., z_n$ where z_i is either one of those three classes, for example 1, 2, or 3. More formally, clustering uses a given x_i and assigns $z_i \in [1, 2, , ..., k]$ where k is the number of clusters to be identified. Note that z has been used here instead of the more typical y. As a reminder, clustering is an unsupervised learning approach so to reduce confusion between some ground truth label for an observation y_i, we have used z_i to denote the assigned cluster. 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

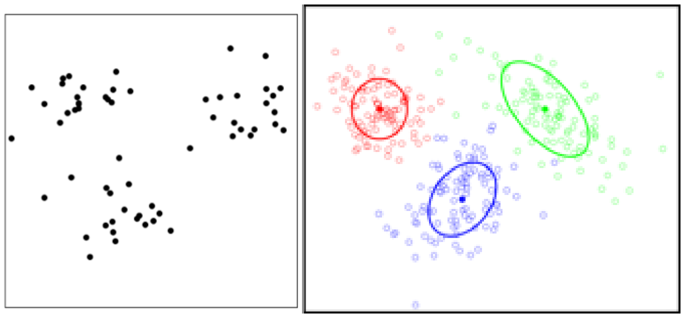


Figure 11.56: Unlabeled input data on the left with 3 defined clusters on the right.

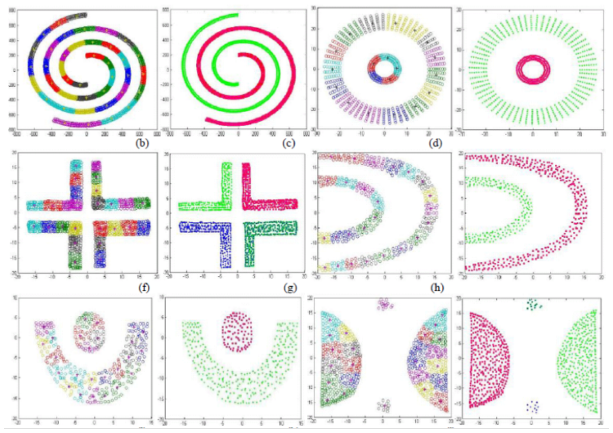


Figure 11.57: Many different examples of more complicated clusters.

better clustering when using this algorithm.

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

$$z_i = \underset{j}{\text{argmin}} ||\mu_i - x_i||_2^2$$

- **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).

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

- **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.

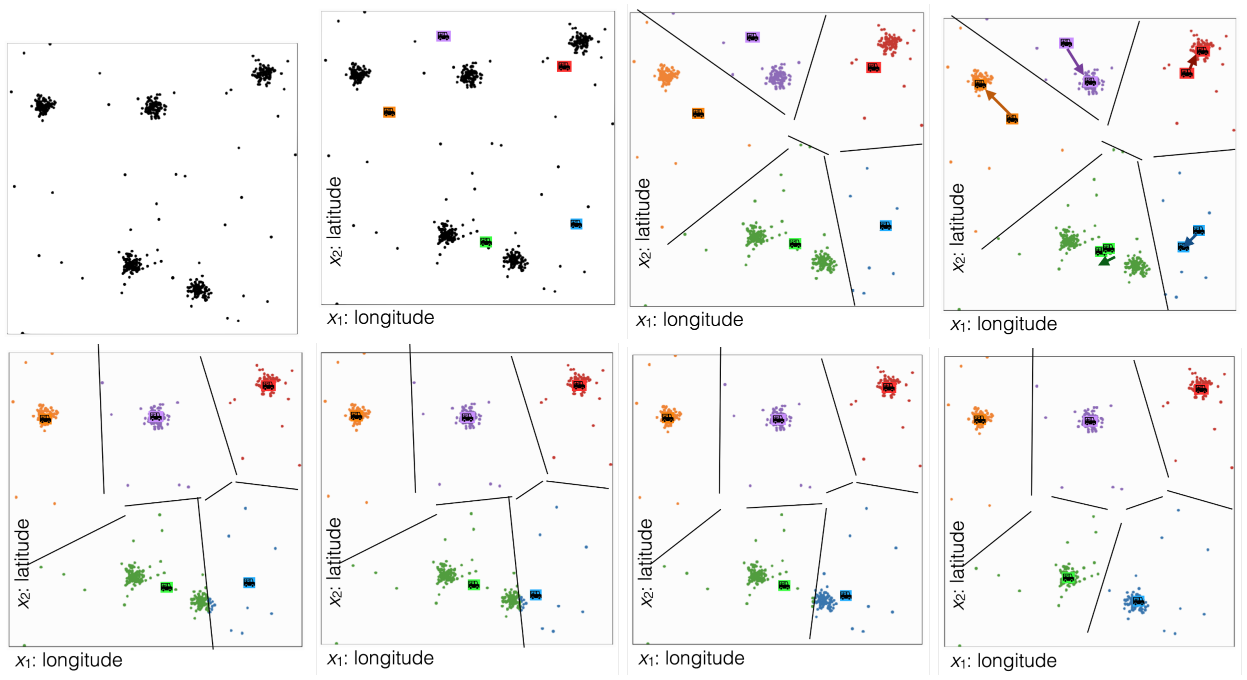


Figure 11.58: K-Means algorithm visualized.

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

$$\underset{z,\mu}{\text{argmin}} \sum_{j=1}^{k} \sum_{i=1}^{n} 1\{z_i = j\} ||\mu_j - x_i||$$

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

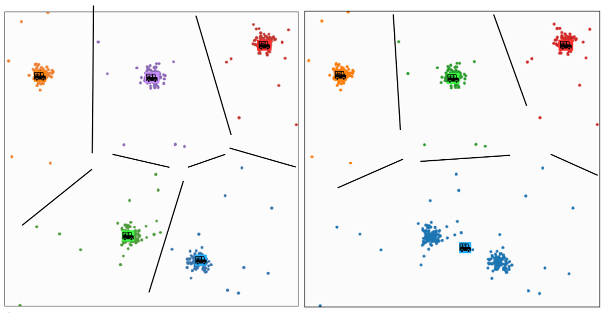


Figure 11.59: Example of choosing different values of k. ($k = 5$ on right and $k = 4$ on left)

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.

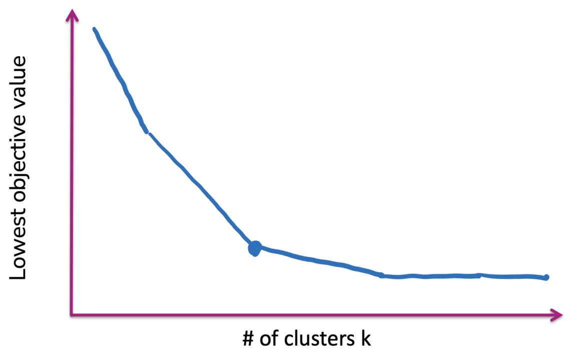


Figure 11.60: Choosing the best k using the elbow method.

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.

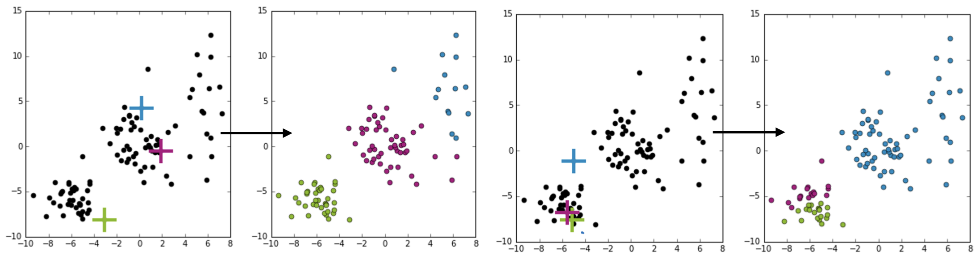


Figure 11.61: Initialization can affect cluster outcomes.

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 were 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 $\mu_1$ 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 $\mu_j$ from the remaining data points with probability of $x_i$ being chosen proportional to the squared 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.

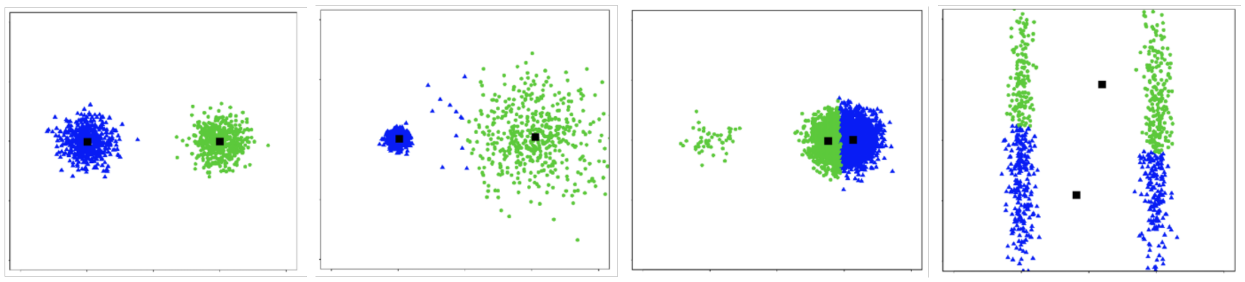


Figure 11.62: Some examples where K-Means works well and some where it fails.

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

- 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.

11.5 Text Embedding: TF-IDF

TF-IDF (term frequency-inverse document frequency) is used to encode the relevance of a word in a document in relation to other text documents in a corpus of text. This is relevant to k-means, as it allows us to in essence, "translate" text documents into something that the algorithm can interpret, and thus perform clustering on. TF-IDF specifically places more emphasis on important words within a corpus, as can be seen in the following section.

The following outlines the process by which TF-IDF is calculated:

- let each document within a corpus be represented by an array, the length of which represent the number of unique words (v) in the entire corpus
- calculating TF: record the number of times a word appears in a document divided by the total number of words in said document, and store the results in an array of length v
- calcualting IDF: record the logarithm of the fraction of the total number of documents divided by (1 + the number of documents using the word)
- fill the original array representing the document with the element-wise (Hadamard) product of the TF and IDF arrays

`Chapter 12/Images/tf-idf.png`

As shown in the image, words that appear very frequently in the corpus will have smaller IDF, and thus, smaller TF-IDF overall, thus placing less emphasis on them. Words that do not appear as frequently will have a larger IDF, and thus, larger TF-IDF. This is a countermeasure to overemphasizing common words like ”the” (at least for the English language) in other embedding methods like bag of words.

Chapter 12
More Clustering Methods

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 Guassian 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:

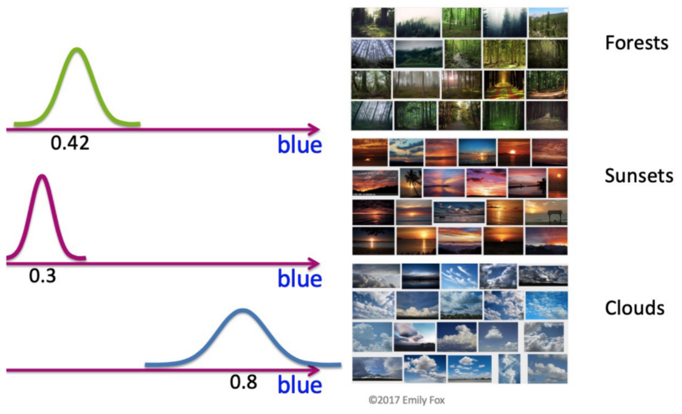


Figure 12.63: Taking a look at how "blue" each pixel of an image is, we have soft clusters resulting from a Guassian distribution.

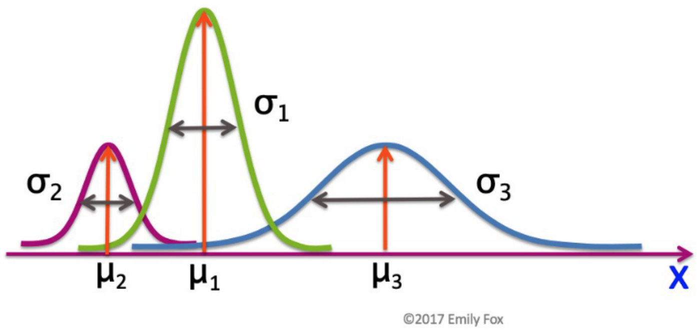


Figure 12.64: Gaussian mixture components represent unique clusters specified by mean and covariance.

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

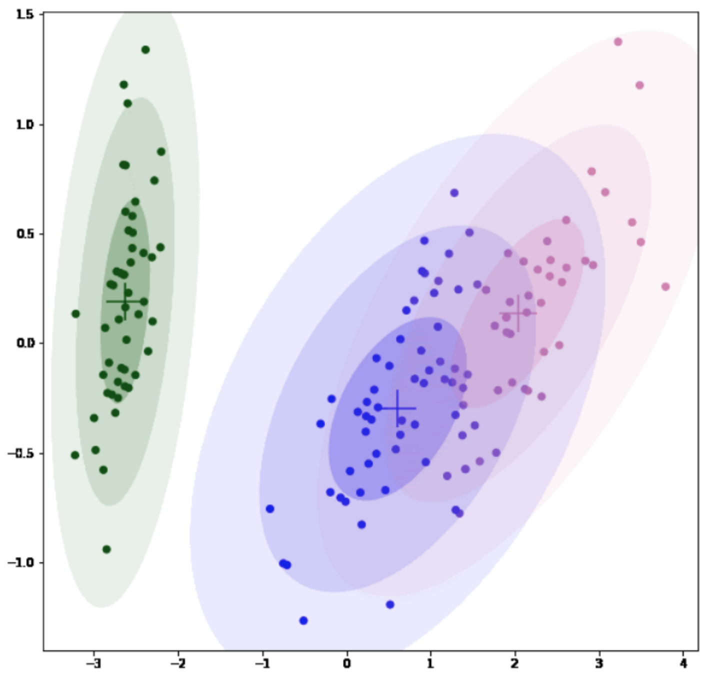


Figure 12.65: The distribution of each cluster gives it a unique shape.

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:

$$distance(C_1, C_2) = min_{x_i \in C_1, x_j \in C_2} d(x_i, x_j) \tag{12.26}$$

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.

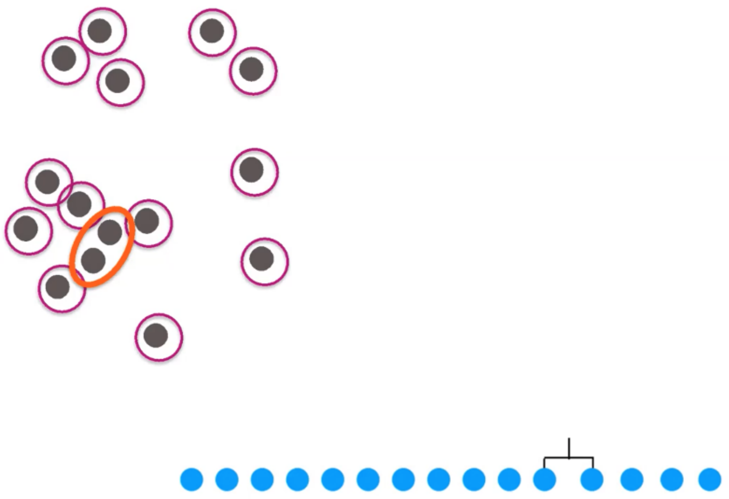


Figure 12.66: In the above figure, we have a set of datapoints in the top left. We want to organize them into a dendrogram (in blue) using agglomerative clustering, so we will group datapoints together in the dendrogram (bottom right) as we merge clusters (top left). To start off, every cluster has one 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.

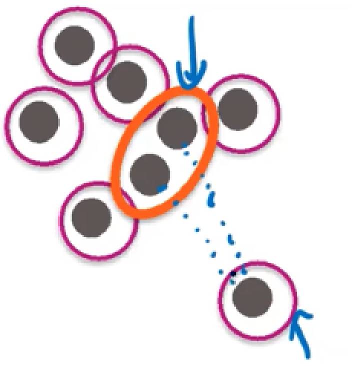


Figure 12.67: Now that one of our clusters has more than one datapoint in it, we can utilize the single linkage distance metric. If we want to measure the distance between the red cluster A and the cluster at the bottom-right B, we have two points to compare. Clearly, the datapoint at the bottom of the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

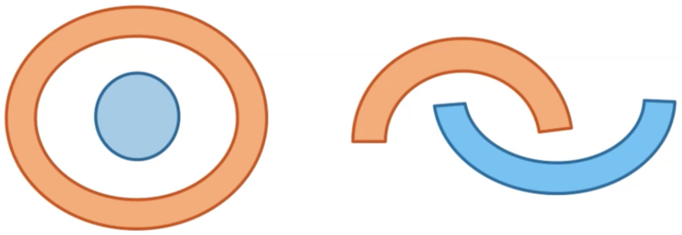
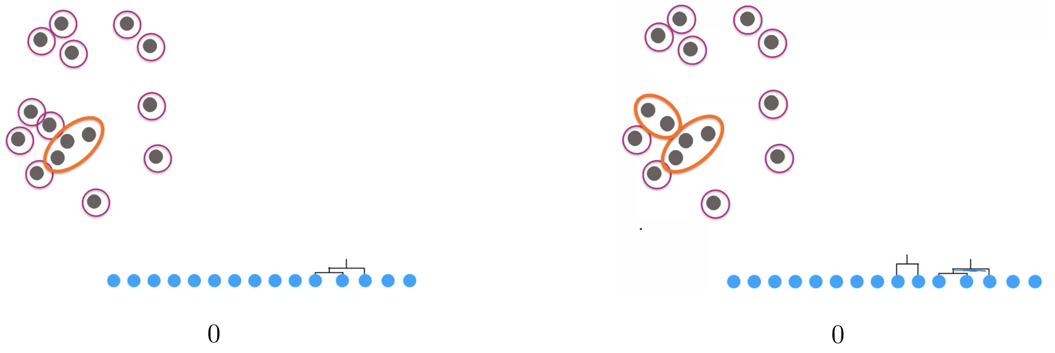


Figure 12.68: A doughnut and a crescent clustering achieved by the single-linkage distance metric in agglomerative clustering. Due to single linkage, before every clustering, we look at the two closest points between the two closest clusters. Because of this, the ring that is the doughnut shape can be 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.

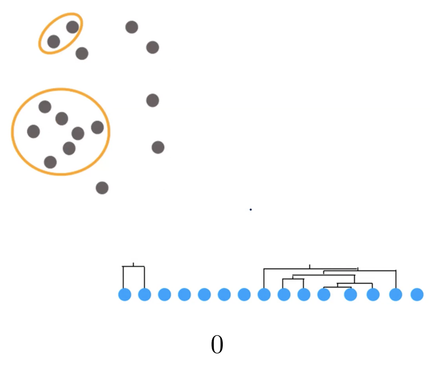


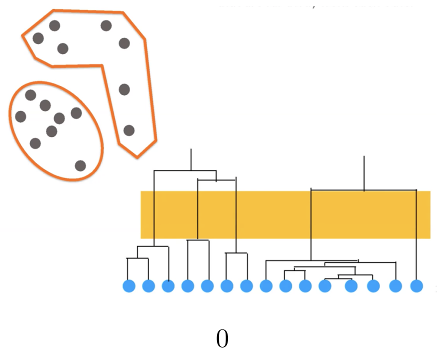
This distance measure is important because it allows us to mark specific heights within our dendrogram that specify cluster separations, if we want to preserve some clusters.

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**, a **centroid linkage**, or an **average linkage**. The complete linkage computes the distances of clusters via the *farthest* points of each cluster away from each other, the centroid linkage uses the center point of each cluster, and the average linkage computes the distance of clusters via the average coordinate of all the points within a 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





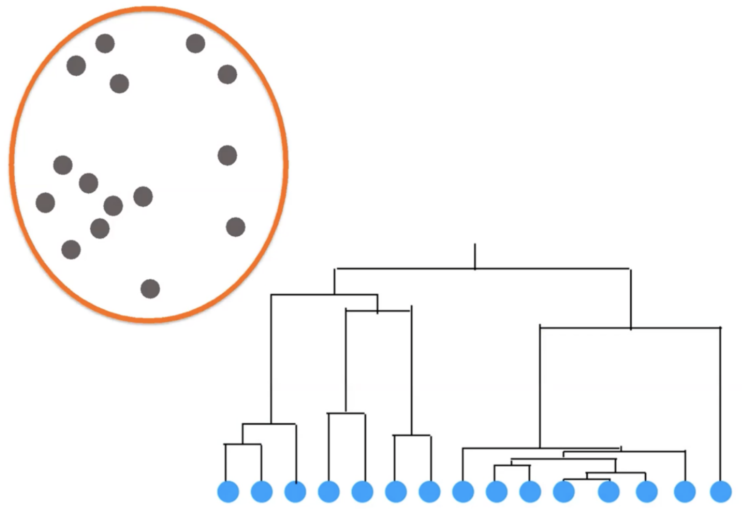


Figure 12.69: The final cluster, with the completed dendrogram.

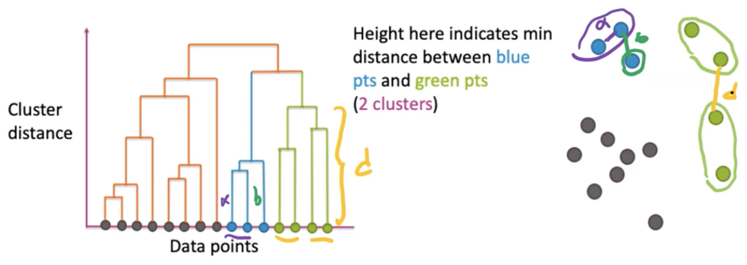

Cluster distance
Data points
Height here indicates min distance between blue pts and green pts (2 clusters)
d

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.

the length of the remaining side", allows us to bypass some of these comparisons, making the best known optimal runtime for agglomerative clustering $O(n^2)$.

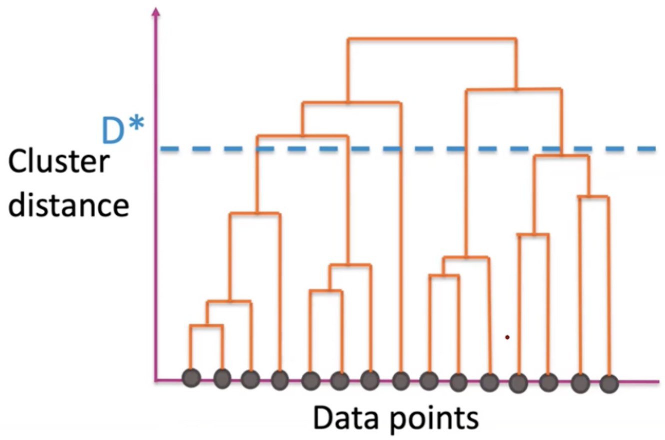


Figure 12.71: If we want to preserve some clusters, we can "cut" the dendrogram at a specific height D^* depending on how many clusters we want. The above example would preserve 5 unique clusters.

12.3 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.

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.

12.4 Curse of Dimensionality (Optional)

One additional thing to keep in mind when utilizing clustering algorithms, especially those that rely on the computation of distances between clusters (e.g. k-means), is the curse of dimensionality. Intuitively speaking, as more dimensions get added, the space gets more "dense" in the corners. As a result, as the dimensions of the data increase, so does the sparsity of the data and their overall distance from the center of the space. This sparsity, especially in very high dimensions, or even in extreme cases where the number of dimensions is higher than the number of observations, can lead to undesirable or strange results, motivating the case for some kind of dimensionality reduction, discussed in future chapters.

Chapter 13
Dimensionality Reduction / PCA

13.1 The Curse of Dimensionality

In your learning up until now, you have likely encountered the **curse of dimensionality**. The curse of dimensionality is when a data set has many features, making it difficult to create and train models, interpret them, and visualize the data. In addition, high dimensional data requires a greater number of samples to be able to accurately fit a model to it.

The number of samples that are required increases exponentially in relation to the dimension. This is similar to filling the volume of the dimensional space. For instance, a 2x2 plot has area 4, while a cube has a volume of 8, and in the 4th dimension 16, etc. This illustrates how in a higher dimension the amount of samples required to fit well in a low dimension will scale exponentially.

Example(s)

Some examples of high dimensional data occur in computer vision, where each pixel of an image represents a feature. For color images, each pixel could be three features for the corresponding red, green, and blue channels.
Another example is DNA analysis, which for different types of models can vary in size immensely, going from only a few base pairs to the size of the human genome which is 6.4 billion base pairs long.

Despite all having a huge amount of data acquired, being able to understand and make predictions on such high dimensional data is not an easy task. Many features correlates to slower training and more complex models.

13.2 Dimensionality Reduction

However, hope is not completely lost for interpreting high dimensional data. A key tool for understanding and visualizing such data is dimensionality reduction. The key idea here is that low dimensional data is easy to work with, so if there was some way to convert our high dimensional data into a lower one, without throwing away too much information, it would be much better to analyze that data in a reduced dimension.

Definition 13.1: Dimensionality Reduction

Dimensionality Reduction is the the task of representing the data with fewer dimensions, while keeping meaningful relations between data.

It is not always the case that data has some low dimensional representation that is accurate, however, in many cases dimensionality reduction is incredibly applicable.

Example(s)

For example, in natural language processing, we would like to understand relations between words (which ones are similar and which ones are not). We can create word embeddings which can group certain types of words together.

We previously learned about the flaws of bag of words model for predicting the sentiment of a restaurant review. Ideally, words like "bad", "angry", "hate", etc. should be grouped together, however a bag of words format would ignore these similarities. By using some type of dimensionality reduction, words as features could be better represented by the types of sentiment they are associated with which can better help train some classifier.

Dimensionality reduction is also incredibly useful for visualization and clustering. Imagine we would like to group some 2,000 cells into specific types using gene expression levels for 2 genes in each cell. This sounds pretty easy. We can simply plot each gene expression on the x and y axis and see if certain cells form clusters. In the real world though, analyzing cells only using 2 gene expression levels is trivial, usually we might analyze some 10,000 gene expression levels. All of a sudden, visualizing the differences between cells is not so easy.

We would like to see if we could convert this 10,000 dimension data into 2 or 3 dimension so that we could plot the cells to identify differences as we could have done if we only analyzed 2 gene expression levels. Using dimensionality reduction, there are ways to convert such high dimensional data down to 2 dimensions to show differences and similarities between data points by capturing some axis as a combination of features.

To conclude, here are some benefits dimensionality reduction can provide:

- **Easier learning.** Fewer parameters, no curse of dimensionality
- **Visualization.** Beyond the 3rd dimension, visualization becomes difficult.
- **"Discovering intrinsic dimensionality".** Often high dimensional data is truly better represented in a lower dimension, as there can be a lot of redundant information.

13.3 PCA

Definition 13.2: Principal Component Analysis

PCA is a common dimensionality reduction algorithm which linearly projects d-dimensionality data to k-dimension data where $k \leq d$.

Our goal in PCA is to find some linear projection which minimizes reconstruction error for the projected data. This reconstruction error can be thought of as the information that is lost when you attempt to recreate the original data from the projection ("undo" the projection).

A linear projection simply uses a linear combination of features, each feature weighted by some value, similar to a linear regression, which can be used to convert a high dimensional data set to a lower one by using some lower number of these linear combination vectors.

In essence, this creates a new feature which comprises many of the original features. Although this seems naive, often this can be useful as a certain combination of features might contain meaningful information.

Example(s)

Here are k new features z created from the original d features x where the coefficients are something random (real projections from PCA will not have these values):

$$\begin{aligned} z_i[1] &= 2x_i[1] + 3x_i[2] - 7x_i[3] + ... + 5x_i[d] \\ z_i[2] &= 5x_i[1] + 0x_i[2] + 4x_i[3] + ... - 2x_i[d] \\ &... \\ z_i[k] &= ... \end{aligned}$$

The most simple projection that PCA can accomplish is projecting 2 dimensions of data down to one dimension. Remember that our goal is to fit some line (1 dimension) to the 2 dimensional data which minimizes reconstruction error, which is our standard sum of squared differences.

$$\text{PCA reconstruction error} = ||x_i - \hat{x}_i||_2^2 \text{ where } \hat{x}_i = z_i u_1 \tag{13.27}$$

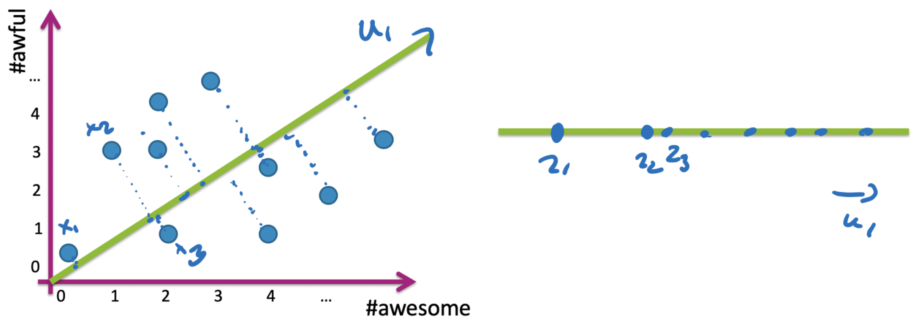


Figure 13.72: Fitting a line u_1 which minimizes reconstruction error.

In PCA, the first feature we project onto is the one that offers the lowest reconstruction error out of any single linear projection (onto 1 dimension), or similarly captures the most variance of the original data in that projected space. In the figure above we see that best line as u_1. This is called our 1st **principal component**.

If we wish to project data to some dimension greater than 1, then we can use more principal components. Each principal component is orthogonal to the previous. For instance, the next principal component we would select would look like an orthogonal line on our data from before.

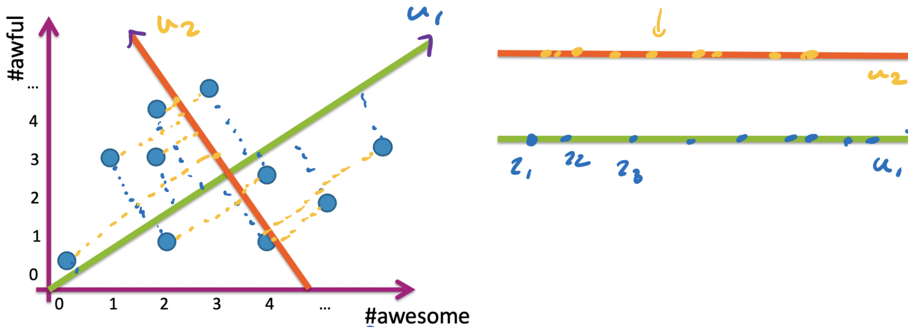


Figure 13.73: Fitting the second principal component.

As you may have guessed, with $k = d$ our reconstruction error goes to 0 and we have captured 100 percent of the variance from the original data.

To compute some of the data's principal components we must run the PCA algorithm.

Algorithm 5 PCA Algorithm

```
**Step 1:** Recenter the data by subtracting the mean from every row.
    • $X_c = X - \bar{X}[1:d]$
**Step 2:** Compute spread/orientation (covariance matrix $\Sigma$)
    • $\Sigma[t,s] = \frac{1}{n}\sum_{i=1}^{n} x_{c,i}[t]\, x_{c,i}[s]$
**Step 3:** Find basis for orientation (computer eigenvectors of $\Sigma$)
    • Select $k$ eigenvectors $u_1, u_2, ..., u_k$ with the largest corresponding eigenvalues ($u_1$ will have the largest eigenvalue and $u_2$ the second and so on).
**Step 4:** Project Data onto principal components
    • $z_i[1] = u_1^T x_{c,i} = u_1[1]\, x_{c,i}[1] + ... + u_1[d]\, x_{c,i}[d]$
      ...
      $z_i[1] = u_k^T x_{c,i} = u_k[1]\, x_{c,i}[1] + ... + u_k[d]\, x_{c,i}[d]$
=0
```

When PCA is applied to different high dimension data sets, it can provide interesting and meaningful principal components.

Example(s)

In one data set of genes–500,568 base pairs each, for 3,191 people, labeled with their country of origin–PCA shows a low dimensional unexpected meaning. The first two principal components represent latitude and longitude, which can reconstruct the map of Europe using the original data colored in by their country of origin. This showcases the power of PCA as despite having a huge number of features, there is a valid and meaningful, low dimensional subspace for the data. This makes sense as genotypes are likely similar for people from the same areas around the world.
This is a unique and interesting example of how high dimension and complex data can have a meaningful low dimensional representation.

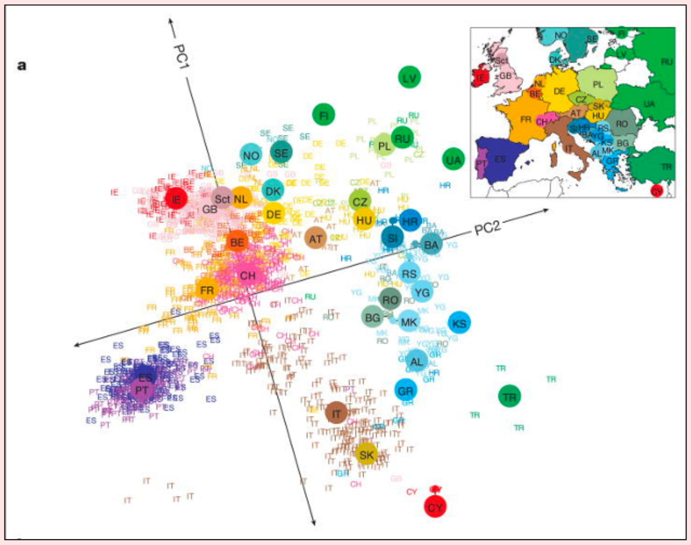


Figure 13.74: Genes for 3191 people projected using two principal components.

13.4 PCA Reflection

It is important to note that there are some major caveats of PCA. First, an integral part of being able to run PCA is computing the covariance matrix Σ which is a $d \times d$ matrix. For something with 10,000 features, this can be quite large and take a long time to compute.

In practice, singular value decomposition is used (SVD) to find k eigenvectors with the largest k eigenvalues, which can be done more quickly than computing the entire covariance matrix. While SVD will not be covered in this course, it is important to understand it is likely to be used in practice when using PCA.

Another major caveat of PCA is that it is simply a linear combination of features for each principal component, this means that PCA assumes there is a lower dimensional linear subspace that represents data well, but this is not always the case.

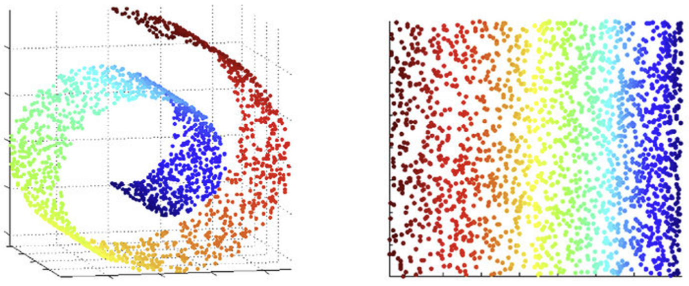


Figure 13.75: 3D to 2D to PCA example where there is no linear low dimension subspace.

In conclusion, PCA is an important algorithm for dimensional reduction which can help avoid the issues that come with the curse of dimensionality. We learned that the goal of PCA is to minimize reconstruction error / maximize captured variance for some k number of dimensions. While it has its drawbacks in that it assumes a linear subspace, it is usually worth exploring what PCA can produce for high dimensional data sets as interesting and important meaning is often found. Note: for the following sections, readers can think of the word ”consumed” being analogous to ”bought” for storefronts like Amazon or ”watched” for streaming services like Netflix.

13.5 Recommender Systems

Chapter 14
Recommender Systems

In this chapter we will be exploring **recommender systems**. We will take a look at how to utilize **matrix factorization** for the recommendation process, limitations of the approach, and solutions.

A recommender system is, in intuitive terms, an algorithm that takes n users and m items, and recommends users items that they will consume (i.e. products users will buy on Amazon, videos users will watch on YouTube). Typically, $n \gg m$, and there are many different ways that such a system can be implemented, a few of which we will briefly explore below.

14.1 Popularity

In this type of recommender system, the items recommendeed are purely what are the most popular at the moment. For example, if a vast majority of Netflix users have been watching the show "Squid Game" due to it's popularity, it will be one of the top shows recommended to other users. Though a system like this is easy to implement, it is prone to positive feedback loops and lacks personalization for individual users, whose interests may not always perfectly align with the most popular items.

14.2 User-User

We can typically do better than a simple popularity-based recommender system, and one of those such methods is with one based on nearest users. In other words, given some user u_i, compute some k nearest neighbors and recommend the items that the users nearest to them. The users and the features for each user are stored in an $n \times m$ matrix, with m representing some number of items that each user has interacted with. The interaction (i.e. liking a Youtube video, leaving a 1 star Amazon review) is recorded and then used as each user's features to calculate the distance between them. Similar users are then recommended products that users around them interacted positively with.

14.3 Item-Item

Another method is to recommend items that are commonly consumed in tandem. For example, if users who buy baby formula also often buy diapers on Amazon, then a user who is bought baby formula will be recommended to buy diapers.

14.3.1 Co-occurence Matrix

Definition 14.1: Co-occurence Matrix

A **co-occurence matrix** is a popular type of recommender system used to recommend data based on similarity.

Example(s)

Since a co-occurence matrix can recommend things based on similarity, a good application is product recommendation. If we have m products in our store, our co-occurence matrix C will be of size $m*m$, and C_{ij} is the number of people who bought products both i and j, where i is along the length of our matrix and j is along the width.

Note that we do have to normalize our data when using a co-occurence matrix, because we don't want an item to be falsely associated with all the other items in the matrix just because it is a popular item. We can normalize by using the $JaccardSimilarity$:

$$\text{Similarity between items } i \text{ and } j \ = \frac{\#\text{ purchases } i \textbf{ and } j}{\#\text{ purchases } i \textbf{ or } j} \tag{14.28}$$

14.3.2 Limitations of the Co-occurence Matrix

While the co-occurence matrix is good at personalizing to a user based on their purchase history, it leaves out some information that could be equally useful in predicting a user's purchasing behavior. For example, context (such as the time of day), the demographics of the users, and the product features are all data that are not captured by the co-occurence matrix but could have helped recommendation. Finally, a co-occurence matrix is also not scalable. As we start off with a fixed size for the matrix, once we add a new item, we come across a **cold start problem**.

Definition 14.2: Cold Start Problem

The **Cold Start problem** is an issue that we will repeatedly visit when discussing the limitations of recommender systems. In essence, this is the issue that occurs when a new item is added to our dataset, and the fact that this new item has no data on it to begin with yields the false computation that this item is not compatible with any other item in the dataset.

14.4 Feature Based

Feature based recommender systems are a possible solution to the cold start problem. They rely on features of the product (i.e. the genre/release year of a movie) and can be additionally enhanced with user specific features (i.e. age, gender identity) in order to provide recommendations. Now, when new users join or new products are listed, the system already has some information about the user or item intrinsically (i.e. user's age, movie's genre), thereby allowing us to solve the cold start problem.

Some weights $w_G \in \mathbb{R}^d$ are first defined for all users.

Then, the following linear model can be fitted where R is the total number of ratings. Note that we are trying to solve for $\hat{r}_v$, or in other words, the predicted rating that will be assigned to a product based on its features.

$$\hat{r}_v = w_G^T h(v) = \sum_{i=0}^{d} w_{G,i} h_i(v)$$

$$\hat{w}_G = \underset{w}{\text{argmin}} \frac{1}{R} \sum_{v} (w_G^T h(v) - r_v)^2 + \lambda ||w_G||$$

In order to personalize these results and instead solve for $\hat{r}_{u,v}$, the rating for a product personalized to the user, we can take the following two approaches.

14.4.1 Add User-Specific Features

Intuitively, we are just appending to our existing item-specific feature matrix, denoted $h(v)$ the user-specific features denoted $h(u)$. We can simply rewrite the equations above taking into consideration these features. Note that d now represents the total number of features across the item and user specific features, instead of just the item-specific features.

$$\hat{r}_{u,v} = w_G^T h(u,v) = \sum_{i=0}^{d} w_{G,i} h_i(u,v)$$
$$\hat{w}_G = \underset{w}{\text{argmin}} \frac{1}{R} \sum_{u,v} (w_G^T h(u,v) - r_{u,v})^2 + \lambda ||w_G||$$

14.4.2 Linear Mixed Models

The Linear Mixed Model approach relies on the inclusion of a user-specified deviation from the global model. The intuition for this approach is that the global weights will be modified by a set of weights specific to each user denoted $\hat{w}_u$ to instead predict $\hat{r}_{u,v}$

$$\hat{r}_{u,v} = (\hat{w}_G + \hat{w}_u)^T h(v)$$

New users have their $\hat{w}_u$ initialized to the zero vector, but update over time based on the residuals of the global model or with Bayesian Update. While the latter is out of the scope of this class, the general process is that the vector is initialized with a probability distribution over user-specific deviations, then gets updated as more data is acquired.

14.5 Matrix Factorization

Definition 14.3: Matrix Factorization

Matrix factorization is another type of recommendation system. It utilizes ratings instead of similarities.

Example(s)

As matrix factorization utilizes ratings, a common application of this recommender system is movie recommendations.

Let's consider the movie recommendations example to dissect matrix factorization. First and foremost, note that for our dataset, we need a collection of ratings from a multitude of users for every movie. Not every user will rate every movie they watch, so our dataset is very **sparse.** In matrix factorization, we account for this by trying to **predict** what a given user would rate a movie based on other patterns in the data. This is virtually impossible to do with complete accuracy, because people can be unpredictable. So, we can only try our best by making assumptions about the dataset.
First, let's assume that there are k types of movies, and every movie belongs to at least one of those k types and every user enjoys at least one of those k types. Thus, we describe each movie in our dataset v with a feature vector R_v of length k, such that every value inside of R_v is how much v belongs into one of the k movie types. We can describe each user in our dataset u with a feature vector L_u of length k, such

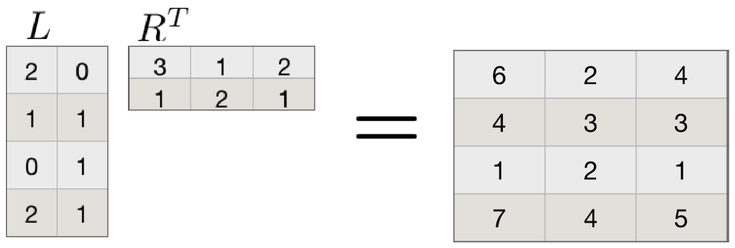


Figure 14.76: An example of a dot-product calculation of a user feature vector L and a movie feature vector R.

that every value inside of L_u is how much u enjoys each of the k movie types. Thus, our predicted rating $\hat{Rating}(u, v) = L_u \cdot R_v$, the dot product of movie v's features and user u's features.

Hence, the problem reduces down to a regression: We must find L and R such that when multipled, achieve predicted ratings that are close to the values that we have data for. Our quality metric is thus:

$$\hat{L}, \hat{R} = argmin_{L,R} \sum_{u,v:r?} (L_u \cdot R_v - r_{uv})^2 \tag{14.29}$$

$u, v : r?$ are the entries with known ratings.

Recall that in a regression problem, we use gradient descent to update all of our parameters at once. In this problem, since we have two unknowns $\hat{L}$ and $\hat{R}$, we will instead use **coordinate descent**, which is a similar optimizing technique to gradient descent except it updates one coordinate in our optimization at a time. We alternate between updating coordinates to simplify the computation for each round of optimization.

14.5.1 Coordinate Descent for Matrix Factorization

We will first begin by optimizing for L. One key insight is that **what one user prefers should have no influence on another user** so not only can we optimize for L by fixing R, we can also optimize for each user's feature vector L_u one at a time. This drasticallu simplifies the computation we have to do:

$$\text{for each user } \hat{L_u} = argmin_{L_u} \sum_{v \in V_u} (L_u \cdot R_v - r_{uv})^2 \tag{14.30}$$

where V_u are all the movies user u has rated, and $R_v - r_{uv}$ is fixed. Now, since everything in this formula except for L_u is fixed, we can treat this as a linear regression problem where $R_v - r_{uv}$ is an input and L_u is a coefficient we are intending to learn. Since this is now just linear regression, we can easily use gradient descent. The same logic goes for movies, so when we alternate to predicting $\hat{R}$, we can compute each R_v separately.

14.5.2 Using Results

Via matrix factorization, we can effectively use the movie features R to discover "topics" of a movie v. This is useful for categorizing movies into genres, or adding keywords to movies so that they show up with specific search queries. We can also use user features to discover "topic preferences" of a user u to recommend relevant movies to that user.

14.5.3 Limitations of Matrix Factorization

Although matrix factorization performs a more personalized computation for each user, it still does not capture context and fails to solve the cold start problem.

14.6 Blending Models: Featured Matrix Factorization

With matrix factorization, consider the scenario where a movie database adds a new user. As this user has no data yet, all of their preferences for every movie category starts off as 0. So, when trying to update the future preferences of this user, we fail to predict reasonably. The solution to this is to supplement matrix factorization with another machine learning model. We will define a feature vector for each movie that takes context into consideration, like the movie's genre, the year it was made, and maybe the director. Then, we will define a model that learns these features for *all* the users in the database. So, if we add a new user, instead of starting their preference for everything at 0, we just set it to be what movies seem to be popular with the entire population as a whole.

14.7 Evaluating Recommendations

Classification accuracy will not serve us well with recommendation systems since we do not care too much about what a user does not like, so it is not very practical to go through every single movie in the entire database and label it with a user's liking or disliking. Instead, we can measure **precision and recall** by assessing how many of our recommendations did the user actually like, and how many of the items that the user liked did we recommend.

Chapter 15
Nearest Neighbors / Distance Metrics

15.1 Assessing Accuracy

Once we have a trained model in machine learning it is important to be able to assess the performance of that model. We also want to see which parts of the model does well and which parts need improvement beyond simply looking and training, validation, and test errors.

Example(s)

Let us create a "dummy classifier" which will predict whether emails are spam or not.
My ingenious classifier will simply always predict an email is spam. I run my model and my test error results in a 90 percent accuracy.
Why? There is a **class imbalance** where 90 percent of my emails are spam, maybe because I signed up for too many email lists, or maybe somehow my email keeps getting passed out.

The model idea reference from above is called a **majority class classifier**. This type of simple model will achieve high accuracy when there is a large class imbalance, when one class appears much more frequently than another in the dataset.

From this we can deduce that just accuracy isn't the best way of always measuring performance for some model. We need to consider these ideas of if there is a class imbalance, how the implemented model compares to a baseline approach like the majority class classifier or randomly guessing. It should also be considered what the purpose of the model will be where accuracy might be the upmost importance, or we might prioritize other metrics of our model which will be discussed below.

One way to visualize mistakes a model might make is called the **confusion matrix**. In this matrix we compare the predicted labels to the actual labels to showcase where the classifier makes mistakes. We represent each cell in the matrix as a count or a score where the diagonal marks a correct predictions and the other cells show misclassifications.

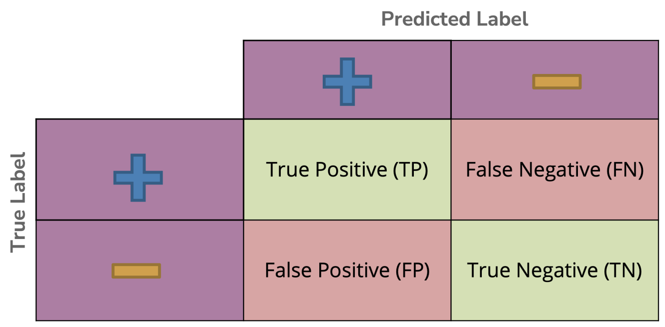


Figure 15.77: Confusion matrix for binary classification.

There are many ways to analyze the model's results, here are a few whre C_{TP} represents the number of true positives, C_{FN} for false negatives and so on:

$$\text{Error rate:} \quad \frac{C_{FP} + C_{FN}}{N} \tag{15.31}$$

$$\text{Accuracy rate:} \quad \frac{C_{TP} + C_{TN}}{N} \tag{15.32}$$

$$\text{False positive rate (FPR):} \quad \frac{C_{FP}}{N_N} \tag{15.33}$$

$$\text{False negative rate (FNR):} \quad \frac{C_{FN}}{N_P} \tag{15.34}$$

$$\text{True positive rate (TPR) or recall:} \quad \frac{T_P}{N_P} \tag{15.35}$$

$$\text{Precision:} \quad \frac{T_P}{C_{TP} + C_{FP}} \tag{15.36}$$

$$\text{F1-score:} \quad 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{15.37}$$

$$\tag{15.38}$$

One way performance is measured is by plotting the the true positive rate (TPR) against the false positive rate (FPR) to reveal a graph which is called the **ROC curve**. In this plot, we want to maximum the area under the curve (AUC). With a maximized AUC thta means that our TPR could be 100 percent with our FPR being 0 percent. However, in practice this is usually not possible as increasing the TPR usually increases the FPR also. These roc curves can be plotted for specific classes in a classifier which can help show what classes are doing well and which are not. In addition this a quick way to show some model's performance.

Similar to the tradeoff between TPR and FPR, we can plot the **precision** and **recall** of a model, we can plot those values against one another hoping to maximuze the AUC also. Depending on the situation we may choose to get the best precision meaning for the ones predicted positive, how many of them were actually positive. Or we may choose to prioritize some model's recall meaning for the things that are truly positive, how many of them were correctly predicted as positive.

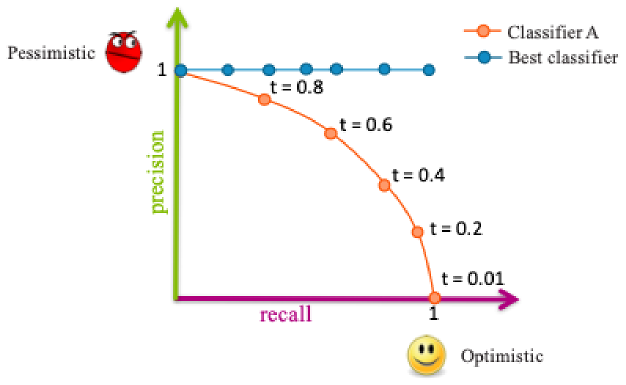


Figure 15.78: Sample precision recall curves.

An optimistic model will predict almost everything as positive meaning it has a high recall, but low precision. A pessimistic mdoel would predcit most things as negative meaning it has a higher precision, but a low recall.

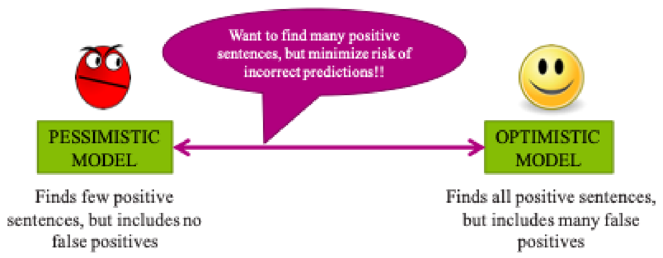

Want to find many positive sentences, but minimize risk of incorrect predictions!!
PESSIMISTIC MODEL
OPTIMISTIC MODEL
Finds few positive sentences, but includes no false positives
Finds all positive sentences, but includes many false positives

Figure 15.79: Pessimistic model vs optimistic model.

Example(s)

Example 1. UW Medicine is training a classifier to predict if a patient has cancer (positive label), from a x-ray image. Their goal is that for input images from people who have cancer, the classifier should accurately predict they have cancer. They are okay with their classifier predicting positive for some people who don't have cancer, as long as it correctly predicts people who have cancer.

In this case, they want to prioritize recall. When a person actually has cancer, UW Medicine wants there to be a high probability that their model will predict they do.

Example 2. YouTube is training a classifier to determine whether a video is appropriate for kids (positive label) or not. Their goal is to not show kids any adult content; it is okay if their classifier predicts less positives, as long as it doesn't misclassify a negative as a positive.

In this case, they are prioritizing precision. When the models predicts a video is appropriate for kids, there should be a very high probability that it actually is.

In addition to showing one model's performance, using these metrics we can compare precision values for different levels of k in a clustering algorithm for example. This can assist in hyperparameter tuning to see what values achieve the desired goal for the model.

15.2 k-Nearest Neighbors

As the name suggests, KNN allows us to find the closest k data points to some point. We can then make a decision based off the neighbors which are found. This would allow us to fin the k closest books to a some book we have read according to some distance metric. When $k = 1$ the algorithm is relatively simple. It searches through each of the data points and computes which item is the closest according to some distance metric.

Formally for KNN when $k = 1$ the output is as follows:

$$x^{NN} = \underset{x_i \in [x_1, \ldots, x_n]}{\text{argmin}} \; distance(x_q, x_n) \tag{15.39}$$

There is no one correct distance metric as it largely depends on the type of problem being worked with. For simple plots likely we would use the euclidean distance (L2 norm) where all features are numerical within some coordinate grid.

The big idea is that we can use KNN to define an **embedding** and similarity metric to find some "nearest neighbors" to a query.

Definition 15.1: k-Nearest Neighbors

k-nearest neighbors (KNN) is a supervised regression and classification (usually classification) algorithm which uses proximity to make classifications or predictions by grouping a single data point which is k closest other data points. KNN assumes that points can similar points are found close to one another.

The input for KNN is x_q which is a query data point and a corpus of data $x_1, ..., x_n$, documents for example. The output is then the data points that are the closes to that query. When $k = 1$ it will simply output the single closest neighbor.

Algorithm 6 k-NN Algorithm

```
**Input:** $x_q$
$X^{k-NN} = [x_1, ..., x_n]$
$nn\_dists = [\text{dist}(x_1, x_q), \text{dist}(x_2, x_q), ..., \text{dist}(x_k, x_q)]$
**for** $x_i \in [x_{k+1}, ..., x_n]$ **do**
    $dist = \text{dist}(x_i, x_q)$
    **if** $dist < \max(nn\_dists)$ **then**
        remove largest $dist$ from $X^{k-NN}$ and $nn\_dists$
        add $x_i$ to $X^{k-NN}$ and $\text{dist}(x_i, x_q)$ to $nn\_dists$
**Output:** $X^{k-NN}$
```

As previously stated in the KNN definition can be used for a variety of tasks.

- **Retrieval.** Return X^{k-NN}
- **Regression.** $\hat{y}_i = \frac{1}{k} \sum_{j=1}^{k} x^{NN_j}$
- **Classification.** Majority class of X^{k-NN}

However, it is important to note that KNN relies on two important questions:

- How do we represent the data points x_i?
- How do we measure the distance $distance(x_q, x_i)$?

15.3 Embeddings

Embeddings answer the question for how we can represent different types of data. Again, continuous numerical data is easy to represent but documents for instance can be represented in a variety of ways. We will show a few here.

- **Bag of words.** The simplest way to represent a document is the bag of words. Each document will become a W dimension vector where W is the number of words in the entire corpus of documents. Each value $x_i[j]$ will be the number of times word j appears in document i. We simply store the counts of each word for its document with some values possibly having 0 because they are not in some document, but in another one in the same corpus. While this is an easy answer to embed a document this representation ignores the order of words only keeping track of the counts.
- **TF-IDF.** TF-IDF (term frequency-inverse document frequency) is another way to representation. The goal of TF-IDF is to emphasize important words to a document. Again each document will be represented as a W long vector. The values in the vector are the **term frequency** multiplied by the **inverse document frequency** (TF $\cdot$ IDF).

$$\text{Term frequency} = \text{word counts} \tag{15.40}$$

$$\text{Inverse document frequency} = \log \frac{\#\text{ of docs}}{1 + \#\text{ of docs using word}} \tag{15.41}$$

This means that for words which appear in every document, they will have a IDF resulting in a smaller TF-IDF.

These are two examples of embeddings for documents however there are more complex ones which can be useful depending on the goal for embeddings. Clearly, both embedding examples above do not take into account the order which is extremely if semantic meaning is trying to represented.

15.4 Distance

The second major reliance for KNN is calculating distances between points. Here we will define a few distance metrics but again similar to embeddings, different metrics are better for different types of data. Note that $distance = 1 - similarity$.

- **Euclidian Distance.**

$$distance(x_q, x_i) = ||x_i - x_q||_2 = \sqrt{\sum_{j=1}^{D} (x_i[j] - x_q[j])^2} \tag{15.42}$$

- **Manhattan Distance.**

$$distance(x_q, x_i) = ||x_i - x_q||_1 = \sum_{j=1}^{D} |x_i[j] - x_q[j]| \tag{15.43}$$

- **Weighted Distances.** Some features vary more than others or are measured in different units. We can weight different dimensions differently to make the distance metric more reasonable. We can define some a to correspond for when some feature or metric might be more/less important. For example a could be a function of a feature's spread: $a_j = \frac{1}{\max_i(x_i[j]) - \min_i(x_i[j])}$

$$distance(x_q, x_i) = ||x_i - x_q||_2 = \sqrt{\sum_{j=1}^{D} a_j^2 (x_i[j] - x_q[j])^2} \tag{15.44}$$

- **Similarity.** This is not a measure of distance so a larger number means more similar.

$$similarity = x_i^T x_q = \sum_{j=1}^{D} x_i[j] - x_q[j] \tag{15.45}$$

- **Cosine Similarity.** Technically this is not a true distance metric as this normalizes the vectors before finding the similarity. This is also very efficient for sparse vectors. In general $-1 \leq cosine\ similarity \leq 1$ but for positive features only (like TF-IDF) $0 \leq cosine\ similarity \leq 1$.

$$cosine\ similarity = \frac{x_i^T x_q}{||x_i||_2 ||x_q||_2} = \cos(\theta) \tag{15.46}$$

It is also important to consider situations when normalization is important and when it actually would hurt a model's performance. For example, when comparing a long document to a short tweet normalization is likely not preffered as it will make dissimilar objects appear more similar than they actually are, but again this is largely dependent on the type of situation. In practice, we can use multiple distance metrics and combine them using some defined weights.

To recap we have learned many different ways to evaluate performance of a model and in which situations some metrics may be preferred over others. We learned a nearest neighbor algorithm which can be used in a variety of different ways with different distance metrics. It is important to know that there is not one-size-fits-all approach to finding good evaluation and distance metrics so it is important to consider the type of problem being worked with and what items are important/less important to consider.

Chapter 16
Kernel Methods and Locality Sensitive Hashing

Suppose you wanted to have a book recommendation system that was capable of quantifying how similar different books are to each other and can look up books based on that similarity. The Document Retrieval technique defines an **embedding** and a similarity metric for those books, to find the ”nearest neighbor” of a book.

Definition 16.1: Embedding

In machine learning, a text **embedding** is a numerical space by which we can define that text. So, we can represent the text as a feature vector.

In this chapter we will dissect the mechanisms behind the similarity metrics we can use for Document Retrieval.

16.1 Nearest Neighbors

This notion of a ”nearest neighbor” seeks to find a local modeling instead of a global modeling, because we do not care to model the entire dataset, but simply what is closer to our input. We can extend this modeling to output the k nearest neighbors of our input, such that we take the average of these neighbors as the most similar candidate to our input. This gives us a smoother prediction function like so:

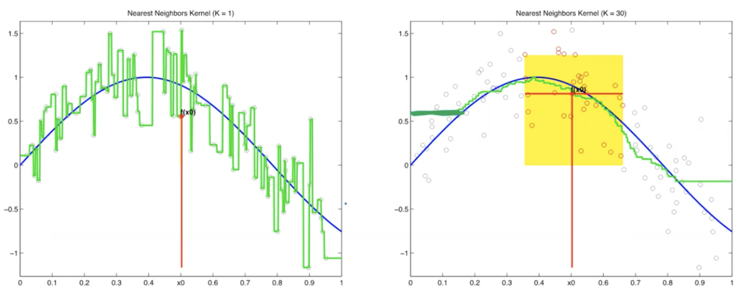


Figure 16.80: Comparing outputting the first nearest neighbor of an input datapoint versus 30 of them.

If we increase our k, we have a higher bias, whereas if we decrease our k we have a higher variance. As this is a difficult balance to maintain, a solution to reliably smoothening out our modeling system is to put more weight on the closer neighbors. Thus we predict:

$$\hat{y}_q = \frac{sum_{j=1}^{k} c_{q,NNj} y^{NNj}}{sum_{j=1}^{k} c_{q,NNj}} \tag{16.47}$$

where $c_{q,NNj}$ is the weight of each neighbor j. If this neighbor is far away, $c_{q,NNj}$ should be smaller, and if the neighbor is closer to our input, it should be weighted more. To turn a distance into a weight, we use a **kernel.**

Definition 16.2: Kernel

A **kernel** is a function that inputs a distance and outputs a weight. There are several function examples that can do this, such as Uniform, Guassian, or Epanechnikov.

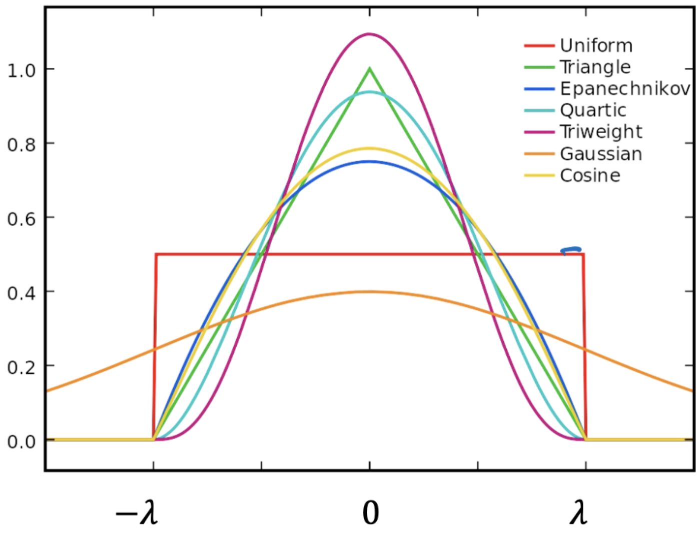


Figure 16.81: Different types of kernels.

16.2 Kernel Regression

Kernel regression weighs all of the training points in a dataset. We can select a bandwidth to to determine how many of the datapoints are relevant to us.

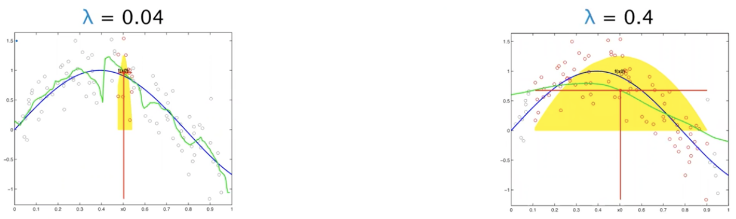


Figure 16.82: Often, which kernel you use matters much less than which value you use for your bandwidth λ.

A validation set can be used to determine which value for our bandwidth is best.

16.3 Efficient Nearest Neighbors

We will now switch back to k-nearest neighbors again, but this time examine more efficient approaches. To begin with, nearest-neighbor methods are already very efficient because we do not have to train any data for them. We simply grab what is closest to our input in our dataset. However, actually determining what *closest* can take up quite some time. We have to visit every single point in our dataset to make a comparison.

If our dataset is of size n =hundreds of billions, this could take an extremely long time. Unfortunately, there is no faster way to find the closest neighbor to an input than comparing the entire dataset, so if we want to acquire higher speed, we will have to sacrifice some accuracy.

16.3.1 Approximate Nearest Neighbor

In the Approximate Nearest Neighbor approach, for a faster computation time we return an approximate instead of a definitive closest neighbor. Going back to our book example, this is fine to do in the real world because a client of your book recommendation system probably does not know what is the *most* similar book to their book, versus what is a *very* similar book. We will find an approximate nearest neighbor via **locality sensitive hashing** (LSH). LSH yields an approximate neighbor within a specific probability.

16.4 Locality Sensitive Hashing

The big idea of LSH is as follows: Data is broken down into smaller bins based on how close they are to each other. When you want to find a nearest neighbor, choose an appropriate bin and do an exact nearest neighbors search in that bin. Thus, the nearest-neighbors algorithm is done on a smaller subset of the data. If we have more bins, our search will be faster as each bin will have fewer data points. However, we also risk making more errors since we are overlooking more of the data.

Let's start off by considering the following dataset, split into 2 bins:

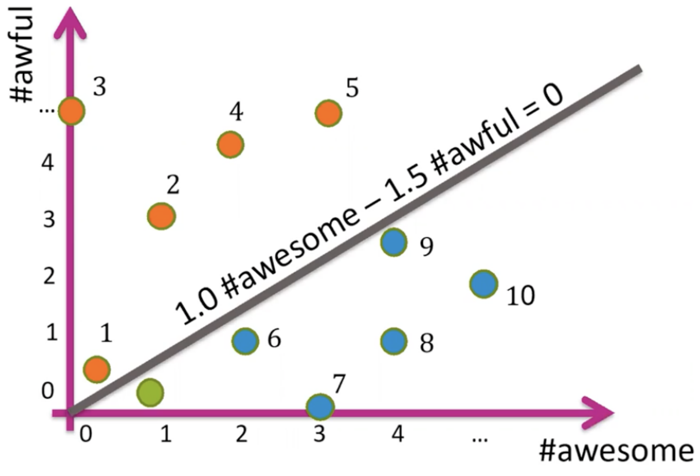


Figure 16.83: Splitting up the data into 2 bins.

We decide that if a datapoint belong to the top of the line, we will only consider the orange datapoints, but if it belong below the line, we will only consider the blue datapoints. The green datapoint in this example is plotted underneath the line, so it will only be considered with the blue datapoints. Based on this binning, the closest datapoint to the green point is datapoint # 6.

You might be wondering first, how do we choose a line, and second, how many lines do we choose? After all, two bins doesn't reduce our computation time that significantly especially if we were the have a much larger dataset. For the first question, LSH selects the split **randomly**. This is because the probability of two close points being binned up separately is quite low, simply because they are close to each other in the plot. As for the second question, it is true that depending on our purposes 2 bins might not be enough. We can simply keep splitting the dataset with more random lines until we get to our desired computation speed (at the expense of accuracy of course).

Bins are indexed in binary as follows:

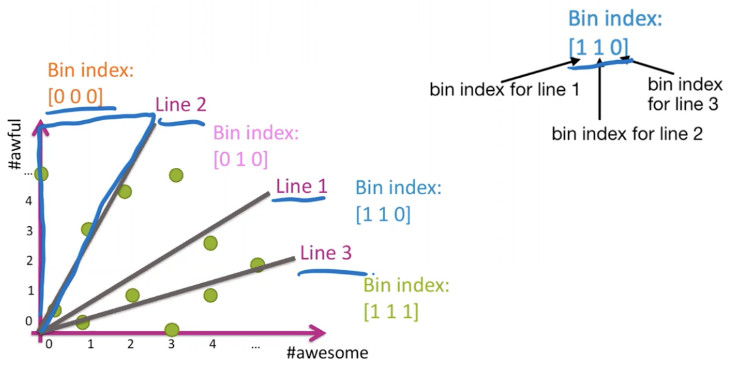
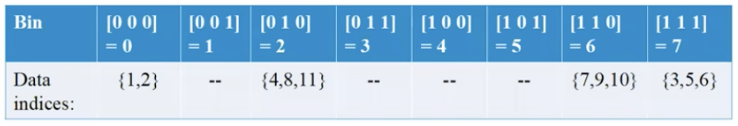


Figure 16.84: Bins are indexed as binary code, with the number of digits equal to the number of line. Each digit then represents if the bin is above or below the line of that digit.

We can then store each bin's index with its corresponding dataset into a hashtable for a quick lookup:



Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
Data indices:	{1,2}	--	{4,8,11}	--	--	--	{7,9,10}	{3,5,6}

Figure 16.85: Bin values stored in a hashtable.

The reason why we index the bins in binary this way is because it makes it extremely quick to determine which bin a dataset belongs to. You simply have to compare a datapoint's output value to the output value of each of the lines at the same input coordinate as the datapoint. Then, depending if these output values are less than or greater than the datapoint's output, you build your binary code. This binary index is then used to quickly look up the correct subset of data in the hashtable. Note that this is not as direct of a computation if we index our bins with decimal values. In addition, with LSH even if we split our data into many bins, depending on our search time budget, we can decide to search more bins later on.

16.4.1 Higher Dimensions

What if we want to implement LSH into higher dimensions? If our dataset was 3-dimensional, instead of lines we would be selecting *hyperplanes*. Our "split" will be one less dimension than the dimensionality of our dataset. As for how many times we split the data, a good rule of thumb is $log_2(d)$ splits for d dimensions.

Chapter 17
Neural Networks & Deep Learning

17.1 What is Deep Learning?

When people talk about **deep learning** they are generally talking about a class of models called **neural networks** that are a loose approximation of how our brains work. Although deep learning has had a lot recent activity in the news, these types of models are not "new" as they have been used for around 50 years. Previously though, they fell in disfavor when simpler models, like the ones that we have already learned in this class, were already performing quite well. Recently, there has been a huge resurgence mainly due to the impressive accuracy these types of models could achieve on benchmark problems. In addition, with the increased amount of data and compute resources (GPUs for example) currently available, training neural networks for complex and large problems has become feasible.

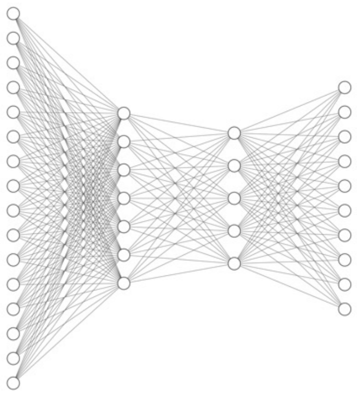


Figure 17.86: Graphical Representation of a Neural Network

17.2 One Unit of a Neural Network

Definition 17.1: Perceptron

A **perceptron** is a single layer neural network that consists of input values also known as the input layer, weights, biases, a summation, and finally an activation function. **Perceptron** used together in multiple layers are called neural networks.

Essentially, a perceptron is a graphical representation of a simple linear classifier. To review, a linear classifier has vector of weights and a bias.

Note that sometimes the bias term is written as part of the weights as w_0 but can also be represented using b for bias. The bias is often referred to as the y intercept in 2D space. This is purely notation and there is no difference in meaning between b and w_0 in this context.

We can more formally define a linear classifier's score as follows:

$$\text{Score}(x) = \sum_{j=1}^{d} w_j x[j] = w_0 + w_1 x[1] + w_q x[2] + ... + w_d x[d] \tag{17.48}$$

This is simply a linear equation which no different than the equation used in a linear regression. This "score", a weighted sum, can be turned into a classifier by using an **activation function**. For example, we could use the *sign* function that outputs 1 if the score is greater than 0 and 0 otherwise. We can see this represented below:

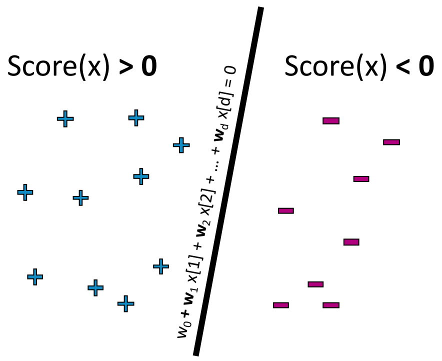


Figure 17.87: Linear Classifier

Definition 17.2: Activation Function

An **activation function** is a function that defines the output of a neuron from the score/output of a weighted sum. There are many different types of activation functions that can help a neural network's training and performance.

In practice, the *sign* function is usually not used as for two main reasons: it is not differentiable and it has no notion of confidence in its output. As the *sign* function only maps to 0 or 1 it is non-differentiable. Then for most classification outputs it would be ideal to have a measure of certainty. It is useful for both training and using a model to know if it has made a sure decision or is uncertain. For those reasons we usually look for activation functions which don't compromise on both of these. However, note that choice of an activation function is highly dependent on its use case. Below are listed a few commonly used activation functions in neural networks:

- **Sigmoid.** There are many different types of sigmoid functions. One for instance is the logistic function as shown above. This was historically extremely popular but has since been used less because the neuron's activation saturates, meaning the weights become increasingly large as the gradient gets smaller. In addition, it is not 0 centered, which can create issues in the gradient steps. Note that when this is applied on the output layer of a neural network this is called **softmax** as it can be interpreted as a class probability (a soft assignment).

$$g(x) = \frac{1}{1 + e^{-\text{Score}(x)}} \tag{17.49}$$

- **Hyperbolic Tangent.** This is also considered a sigmoid function and saturates similarily but has the benefit of being centered at 0.

$$g(x) = \tanh(x) \tag{17.50}$$

- **Rectified Linear Unit (ReLU).** This is the most popular activation function uses in neural networks as it is easy to implement, fast to compute, and has a true 0 value. However, neurons can “die off” as their outputs are 0. Note that there are variants of this which are “leaky” and “noisy” which can be beneficial depending on the use case.

$$g(x) = \max\{x, 0\} \tag{17.51}$$

- **Softplus.** Softplus is a smooth approximation of ReLU.

$$g(x) = \log(1 + \exp(x)) \tag{17.52}$$

The two components of a linear weighted sum and an activation function make up what we call a single **neuron**. Then combined with the input layer with a single neuron into the linear model we get a perceptron.

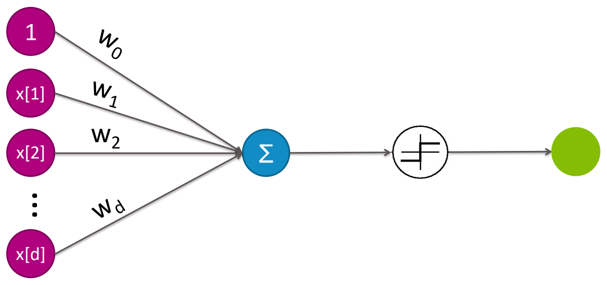


Figure 17.88: A perceptron.

17.3 Perceptron Learning

As we have learned, a perceptron is essential a linear model. We will take a look at two functions that a perception can learn. A perceptron can find weights which can solve these two problems with the 3 trained weights (bias, x_1 and x_2) and the *sign* activation function.

- **The OR Function.**

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

Here is one solution: $w_0 = -0.5$, $w_1 = 1$, and $w_2 = 1$.

To solve this we simply find the weighted sum:

$$sign(\sum_{j=1}^{d} w_j x[j]) = sign(w_0 + w_1 * x_1 + w_2 * x_2) \tag{17.53}$$

Then let’s check if this works for the second row of the table above using $x_1 = 0$ and $x_2 = 1$:

$$sign(-0.5 + 1 * 0 + 1 * 1) = sign(0.5) = 1 \tag{17.54}$$

The rest of these are also valid and can be checked if you wish. Clearly there are multiple solutions to this problem as there are only 4 data points to fit here but this is one that separates the data correctly.

- **The AND Function.**

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

  Here is one solution: $w_0 = -1.5, w_1 = 1$, and $w_2 = 1$. Again, similar to the previous problem there are multiple solutions.

However, as you likely have predicted, a single perception has its limitations in that it can only fit linear classification problems. For instance, a single perceptron cannot fit to the XOR function because it is not linearly separable:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

In order to create a neural network complex enough for nonlinear problems we need to combine multiple perceptrons in layers.

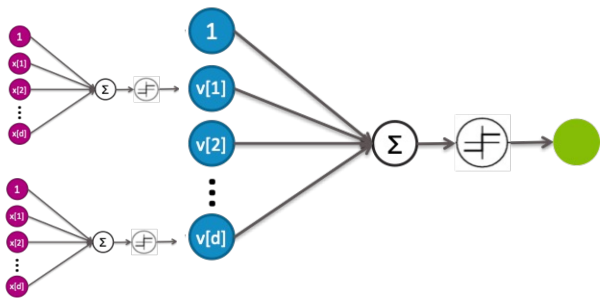


Figure 17.89: Perceptrons in Layers (a Neural Network)

Definition 17.3: Hidden Layer

A **hidden layer** is any layer in a neural network that is between the input and output layers where the neuron takes weighted inputs and produces an output using an activation function.

Let's design a two layer (one hidden layer) neural network that can fit the XOR function. We will start by breaking down the XOR function:

$$x_1 \text{ XOR } x_2 = (x_1 \text{ AND } !x_2) \text{ OR } (!x_1 \text{ AND } x_2) \tag{17.55}$$

Using this we can design a neural network to fit XOR. We will use the first half of the OR statement as the first layer and the second half as the second layer. We can also refer to the first layer as the **input layer** and the second layer as the **hidden layer**.

We will define $v_1 = (x_1 \text{ AND } !x_2)$ and $v_2 = (!x_1 \text{ AND } x_2)$. We can now create a neural network which in the second layer uses the same OR perceptron we fit earlier. Then we define the first layer's weights to apply the correct AND for both v_1 and v_2. We are also still using the *sign* activation function for all neurons. Combined together we have a valid XOR fit neural network.

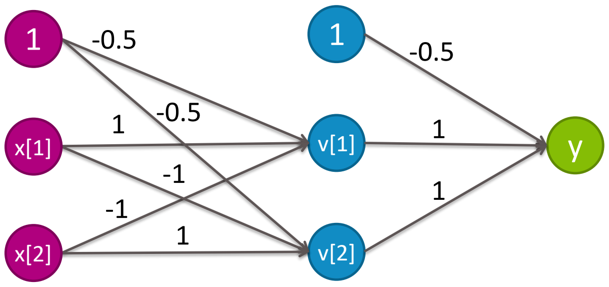


Figure 17.90: Two Layer XOR Solution

We have now defined our first two layer (one hidden layer) neural network. Keep in mind normally this neural network will be trained using data instead of assigned like above. Training a neural network will be discussed below.

We will now try the neural network's prediction by running a forward pass using the sample input $x_1 = 1$ and $x_2 = 0$.

Following the model as node v_1 we get $-0.5 + 1 * 1 + -1 * 0 = 0.5$ which is positive so node $v_1 = 1$. For node v_2 we get $-0.5 + -1 * 1 + 1 * 0 = -1.5$ which is negative so we $v_2 = 0$. In the hidden layer we sum to get $b + v_1 * 1 + v_2 * 1 = -0.5 + 1 * 1 + 0 * 1 = 0.5$ which again is positive so the output is 1 which matches XOR.

Surprisingly any two layer neural network can represent any function if enough nodes are allowed in the hidden layer. Although this is ideal in that we can fit complex functions well it has the consequence that neural networks are likely to overfit however there are some methods to try to avoid overfitting similar to other models. Having lots of training data is important along with regularization and/or dropout. Also, keeping the network shallower and with fewer nodes can help to avoid overfitting, going deeper only helps if you are very careful.

17.4 Applications

17.4.1 Regression or Classification?

We have showed that using a binary final output function (0 or 1) in neural network can be used for classification tasks like AND, OR, and XOR. For more complex classification tasks, like previously stated, we can use softmax to transform scores from the neural network into a probability output. This can be used in a variety of settings, such as in computer vision.

For instance, let us decide that we wish to classify images into one of 5 different classes with our newfound understanding of neural networks. We should design our neural network to have 5 output nodes corresponding to the 5 different classes. Then when using the softmax activation function on the output scores we can turn those 5 scores into 5 probabilities. Then as in this scenario we wish to just decide one classification, we can choose the class with the highest probability and assign the image to some class that way.

Example(s)

One example of where classification is used in neural networks is for the MNIST dataset. These are hand-written digit images that are labelled with the correct value. In a neural network design for this there will be 10 nodes in the output layer corresponding to the probability of some image (the input) being classified as some digit (0-9). That means that number of classes should be equal to the number of outputs.

Neural networks are not limited to classification tasks though. For instance, in a regression task for a neural net, they will have one output node as the single number prediction.

17.4.2 Learning Features

As we have learned through using LASSO, it can be immensely helpful to decide which features are important to take use. Sometimes we don't know if the best features are even from the set of the current ones or if it would be better to use combinations of multiple of them. This is a key feature of neural networks that makes complex tasks much easier.

Previously, computer vision methods used hand crafted features to make decisions about images. For example, if trying to classify images of people it might look for the face, nose, mouth, etc. These are major generalizations though and not very easy to simply plug an image in and know these features directly. In addition this relies on coming up with these hand picked features which often we might not know what will be the best indicators for complex classification problems

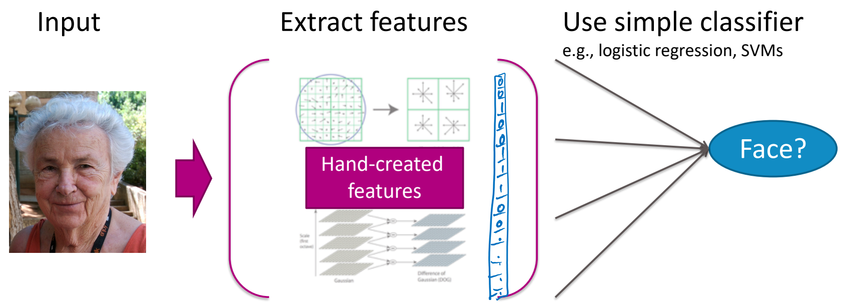


Figure 17.91: Manual Feature Selection in Computer Vision

Using neural networks fixes this problem. We can directly input images as multi-dimension arrays (you will see how to do this in the next chapter) and the neural network can learn features across the multiple layers of the network. Each layer will learn subsequently more complex features starting from the pixel level as the input to understanding features in later layers that we might have hand picked or more likely features that we might not understand, but help to make an accurate classifier.

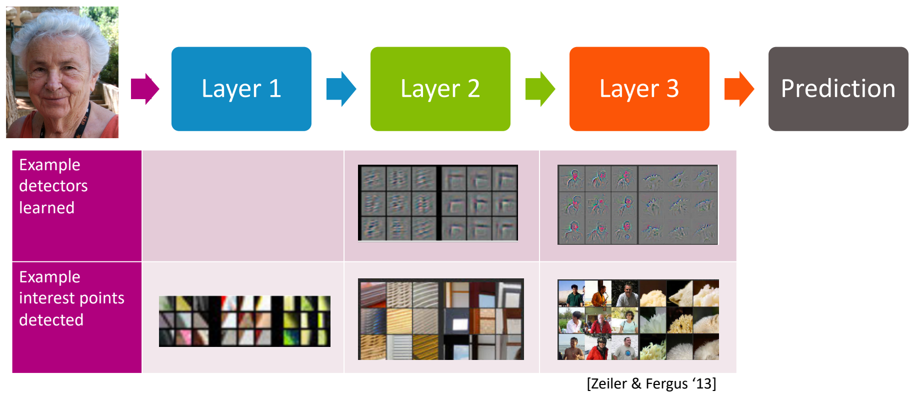


Figure 17.92: Manual Feature Selection in Computer Vision

17.5 Training

Now that we understand the basics of what make of neural networks are, what they can learn, and what they can be used for we should learn about how to actually train one of these models. The first step, similar to the other models we have learned, is that we must define a cost function.

- For regression tasks, typically the residual sum of squares (RSS) or Root Mean Square Error (RMSE) is used
- For classification, typically Cross Entropy loss is used.

With a cost function defined, we can train a neural network, similar to other models, using gradient descent in three main steps:

1. Do a forward pass of the data through the network to get predictions.
2. Compare predictions to true values using the cost function.
3. Backpropagate errors by pushing the calculation of the gradient through the back of the network so the weights make better predictions.

To train neural networks well, usually there is a large amount of data. To speed training up instead of following the three steps from above for the entire dataset at once, the data is broken up in to **batches**. Generally, going through the entire data set once is not enough to train the network so multiple passes through the same data is required. We call an iteration that goes over every batch (the entire training set) once an **epoch**.

Here is how the training of the neural network would look like in code across a set number of epochs and batches:

```
for i in range(num_epochs):
  for batch in batches(training_data):
    preds = model.predict(batch.data) # Forward pass
    diffs = compare(preds, batch.labels)  # Compare
    model.backprop(diffs)          # Backpropagation
```

Figure 17.93: Training a Neural Network

One major challenge of training neural networks is that there are many hyperparameters. We must consider at least the shape of the network (how many hidden layers and hidden neurons), the activation functions, the learning rate for gradient descent, the batch size, and the number of epochs. Unfortunately, there is no best solution to determining these values and often is lots of trial and error. Grid search and random search can be useful and Bayesian Optimization can also help with this.

In general, loss functions with neural networks are not convex meaning that the backpropagation algorithm for gradient descent will only converge to a local optima. Similar to k-means the initialization is important and could affect the final result. However, there is not one known way and usually random initialization is used. The same applies for the learning rate if the step sizes are too large when running gradient descent.

There is still ongoing research about understanding neural networks and why they work so well on certain problems, the best ways to train, and the caveats of using them. It is important to know that while neural networks are useful for fitting complex problems, they are not the one and done solution as they are often quick to overfit and overly complex for many tasks.

Chapter 18
Convolutional Neural Networks

Advances in deep learning were primarily catapulted by image data analysis. For this reason, *Convolutional Neural Networks* play a dominant role in the developments of machine learning research.

Recall that a Neural Network has its **input, hidden, and output** layers. This is because learning in layers promotes the learning of individual **feature representations.** How well different features within a data are learned relies on our ability as ML developers to optimize hyperparameters. In this chapter we will transition into a more specific architecture type that optimizes learning on image data.

18.1 Images as Data

We know that input data for our machine learning models use vectors. How can we represent image data as vectors? If we were to flatten a two-dimensional image into a single stream of numbers such that we could capture it as a vector, we lose some key relationships between the pixels, such as the distance between pixels representing the same object. In addition to that, if we are working with colored images, a pixel in an image is generally defined by three values: Red, Green, and Blue. So, if a 200 by 200 colored image has 3 colored channels, we are dealing with a vector of 200*200*3=120 thousand features. That is quite a hefty bite to chew on for our model especially for such a small image, so we introduce the concept of **convolutions**.

18.2 Convolutions

Definition 18.1: Convolution

A **convolution** in neural network training is a reduction of the number of weights that the model needs to learn by summarizing the image into fewer pixels.

A convolution combines information about local pixels such that pixels close to each other in an image are ”summarized” by a smaller set of pixels.

This ”summarization” is done by ”sliding” a **kernel** across an image to output a final product for each position of the kernel.

Definition 18.2: Kernel

A **kernel** in a convolution is an n x n matrix of numbers. In Figure 18.1, the kernal is the dark blue shaded region. Since the image is a 4x4 image and the kernel is a 3x3 matrix, the kernel can have 4 unique positions on the image. As a result, the output is a 4-pixel, or 2x2 image.

Each number inside the kernel matrix is multiplied with the value for each pixel it lines up with, and then all the elements of the kernel are summed to output one single value. Then, the kernel slides over to a new position in the image and the process is repeated.

The optimal numbers for a kernel matrix are learned by the model. To have the highest success rate, a model will learn values for a kernel that capture some sort of key information within an image, though that information is not directly interpretable by humans. Some examples of things that CNN developers have discovered kernels can extract are specific objects within an image, structural patterns, or dominant outlines in the image.

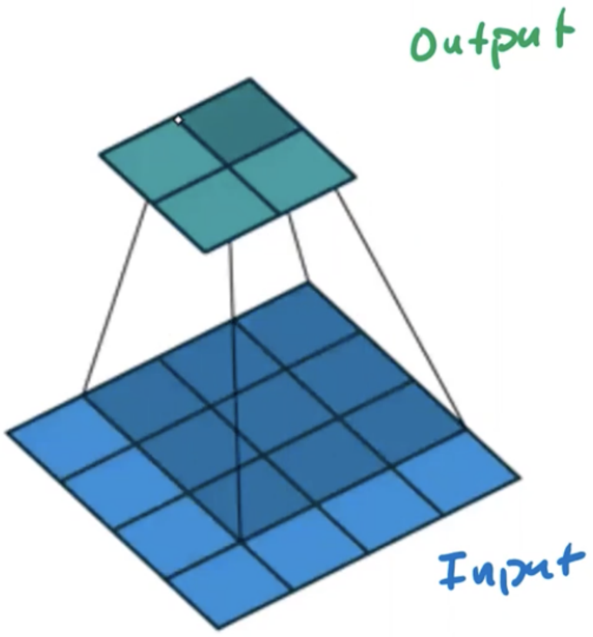
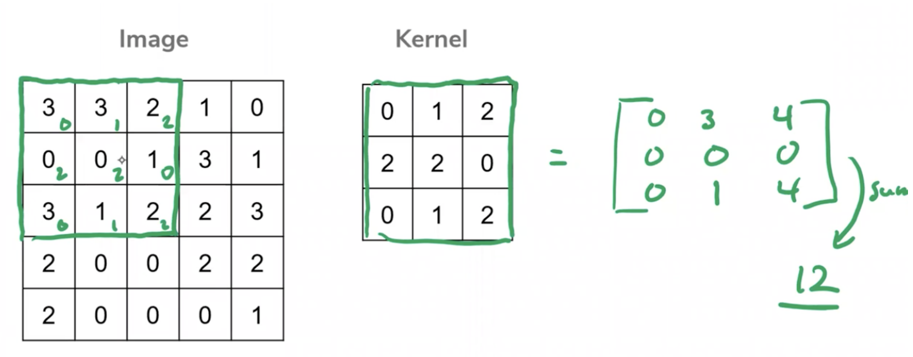


Figure 18.94: In the convolution above, the 4x4 image is summarized into a 2x2 image.

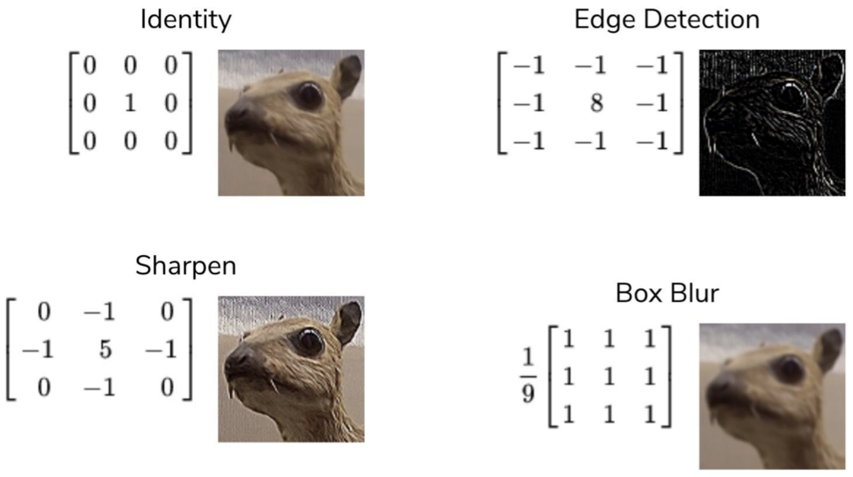


3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

0	1	2
2	2	0
0	1	2

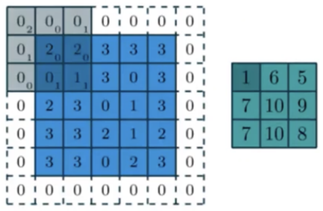
$$= \begin{bmatrix} 0 & 3 & 4 \\ 0 & 0 & 0 \\ 0 & 1 & 4 \end{bmatrix} \xrightarrow{\text{sum}} \underline{12}$$

Figure 18.95: Calculations for applying a kernel to an image for one position.



$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 18.96: Special kernels which have specific properties such as maintaining the same image, detecting edges, sharpening the image, and applying a box blur


[First iteration]

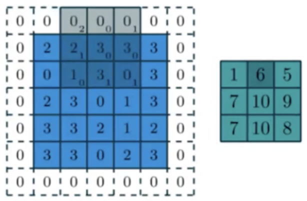

[Second iteration]

Figure 18.97: The above shows a kernel of size 3x3 applied to a 5x5 image. The image has 1x1 padding. Notice that the second number in the output, 6, is the result of the kernel slidden over 2 pixels to the right across the image, so we know the stride is 2x2.

We can tune some specifications for our kernel such as **padding**: a layer of pixels of some static (unchanging) value wrapped around the image so that the kernel can align with the image more conveniently, **dimension**: how large the kernel should be, **stride**: how far the kernel should slide across the image each iteration.

18.3 Pooling

Pooling is another similar operation to convolutions done on an image. Sometimes, applying a convolution to an image doesn't reduce an image enough, so we can further utilizing pooling operations. Pooling is a lot more simple than applying a convolution in that we do not have a kernel full of elements. Instead, we have a filter that we slide across the image, and for every position of this filter, we simply take the minimum, maximum, mean, or median, of the set of pixels inside the image that fall underneath this filter. It is most typical to use a **max pool** with a 2x2 filter and a 2x2 stride, such that there is no overlap between filters. Using max pool has seemed to work better than the average pool.

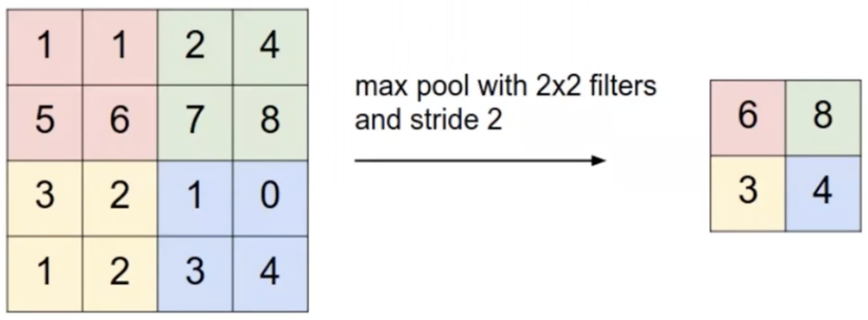


Figure 18.98: A maxpooling in action. You can think of each colored region as a position of the 2x2 filter.

18.4 Summary

All in all, we can think of a Convolutional Neural Network as a layered combination of these operations we have just learned. Take a look at this example CNN diagram below. Notice how after each operation, we end up with some dimensionality reduction. This dimensionality reduction helps us learn fewer weights since we have less data now, and the data is a holistic "summary" of the original input. Why? Because with convolutions and pools, we apply kernels and filters that slide across the entire image. So even though we have less data, the data is extracted from dispersed pieces of the image. So, the data, 'represents' the image pretty well (even if we don't know exactly how it has chosen to represent it)–hence why machine learning developers will often use the terminology "representations".

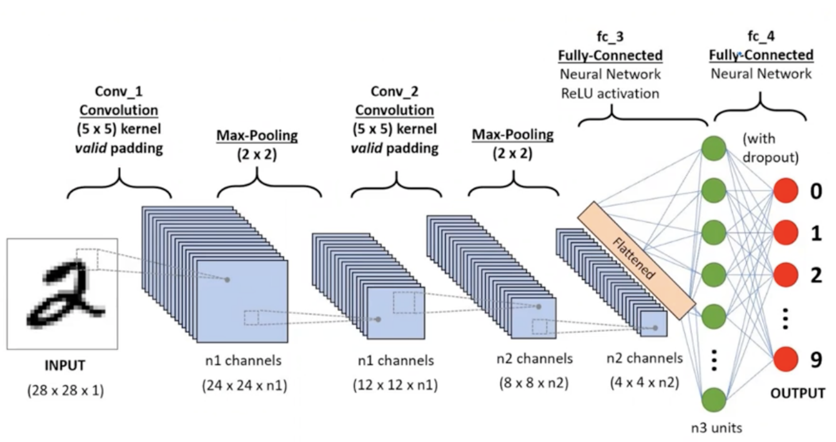


Figure 18.99: A high-level overview of a sample CNN.

This method is an extremely clever tactic that not only optimizes our deep learning algorithms **(efficiency)**, but also teaches the model fundamental groupings within a a two-dimensional image **(shift invariance)**. Outlines within an image are learned through these layers. Or, perhaps, the presence of two eyeballs of some animal. Or, it could be learning how to associate a bright yellow ball in the sky with the behavior of casting shadows on other objects. These are all relationships within a two-dimensional space that can be captured by convolutions since convolutions make use of distance within 2D space, but those same relationships might not be captured if we were to flatten the image into a vector. I mention these examples so that you may understand how effective convolutions and pools are at extracting **syntactical** meaning from an image. The sun, as an object, and the way it affects other objects in an image, should be learned. In other words, whether the sun is on the left side or the right side of the image shouldn't make our model think it's a totally different input. It should be able to recognize objects within a 2-dimensional space in a consistent manner even if those objects shift around spatially. This is what we mean by shift-invariance.

Due to the multitude of unknowns in our CNNs (and really, for all Neural Networks for that matter) deep learning is a very experimental field. We just can't really know how one model architecture will perform against another, so researchers test out many different architectures and combinations of hyperparameters when desigining their models.

18.4.1 A Kernel as a Neuron

An alternative way to think about convolutions in a CNN is by treating each kernel as a neuron. We can think of each kernel or each neuron responsible for learning some thing about the image. Depending on our task, we will have a specific amount of output neurons. Our ultimate goal is the optimize the weights for these output neurons. So, each of these output neurons will have learned a specific set weights pertaining to a specific attribute about the image (sharp edges, the sun, an eyeball) by having scanned the entire image.

18.4.2 General CNN Architecture

In general, the start of the CNN contains a series of Convolutions, Activation Functions, and Pooling layers. It's very common to do a pool after each convolution. After this series, the lower-dimension representation of the image is flattened to work with the final neural network.

18.4.3 Impacts

CNNs have proven to be extremely successful when it comes to image classification. The ImageNet challenge, an image classification competition, has seen slow and steady improvements in the task with standard neural networks up until 2012, when SuperVision, a convolutional network, superceded its competitors with a historical 17% error rate. Google's GoogLeNet achieved a 6.66% error rate in 2014. Since then, convolutional networks have been paving the way for unprecedented successes in image data analysis.

CNNs are important for many different industries in our society. For example, recognizing roads and buildings are important for self-driving car vision. CNN's aren't limited to image data. They have also proven to be groundbreaking tools in the fields of Natural Language Processing and Speech Recognition.

Of course, with all Machine Learning algorithms, CNNs have their limitations too. For one, they require an overwhelming amount of data. The ImageNet challenge provides competitors with 1.2 million images, but this is a relatively small dataset for neural network standards. They are also computationally expensive and even more expensive environmentally. It is hard to tune hyper-parameters as there are numerous choices to make such as the size of kernels, stride, 0 padding, number of convolutional layers, and the depth of outputs of convolutional layers. Finally, neural networks are extremely uninterpretable, which makes them an ethically unpopular choice for tools that directly impact humanity.

18.4.4 Transfer Learning

Transfer learning is a useful technique for neural networks. It is the act of using a pre-trained model for a new task. Since the early stages of a neural network learns high-level features that is usually not sensitive to the task at hand, one can chop off the first part of a neural network and tack on a new end to it such that those learned high level features are preserved.

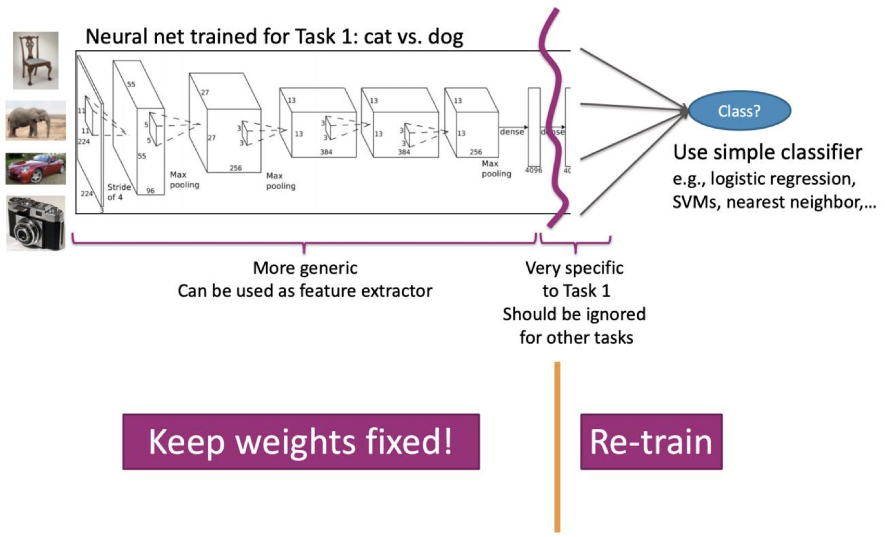

Neural net trained for Task 1: cat vs. dog
Stride of 4
Max pooling
Max pooling
Max pooling
dense
dense
Class?
Use simple classifier
e.g., logistic regression, SVMs, nearest neighbor,...
More generic
Can be used as feature extractor
Very specific to Task 1
Should be ignored for other tasks
Keep weights fixed!
Re-train

Figure 18.100: A transfer learning example using a neural network trained for differentiating between cats and dogs to help classify numerous other animals

18.4.5 Limitations

Neural Networks are easily decievable due to dataset bias. Previously, we discussed the idea of "shift invariance". Unfortunately for many neural networks, this is still a challenge. Simply moving an object's position can easily confused a neural network that trained on recognizing an object in a specific position.