A Machine Learning Primer

Mihail Eric
@mihail_eric

Table of Contents

Fundamentals of Linear Regression

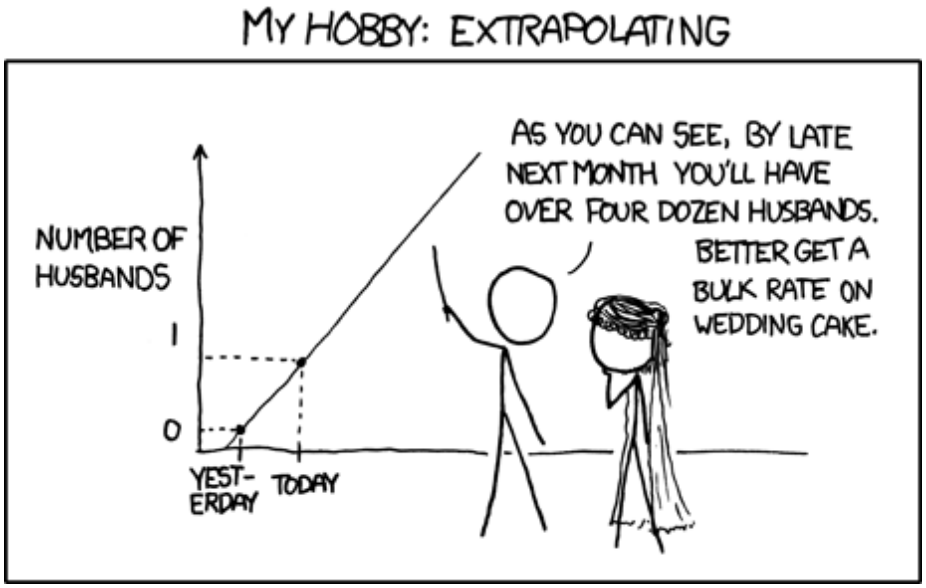

Figure 1: Courtesy of XKCD

In this section, we investigate one of the most common and widely used machine learning techniques: **linear regression**.

Linear regression is a very intuitive supervised learning algorithm and as its name suggests, it is a *regression* technique. This means it is used when we have labels that are continuous values such as car prices or the temperature in a room. Furthermore, as its name also suggests, linear regression seeks to find fits of data that are lines. What does this mean?

Motivations

Imagine that you received a data set consisting of cars, where for each car you had the number of miles a car had driven along with its price. In this case, let's assume that you are trying to train a machine learning system that takes in the information about each car, namely the number of miles driven along with its associated price.

Here for a given car, the miles driven is the input and the price is the output. This data could be represented as (X, Y) coordinates.

Plotting them on a 2-d coordinate system, this data could might look like Figure 2.

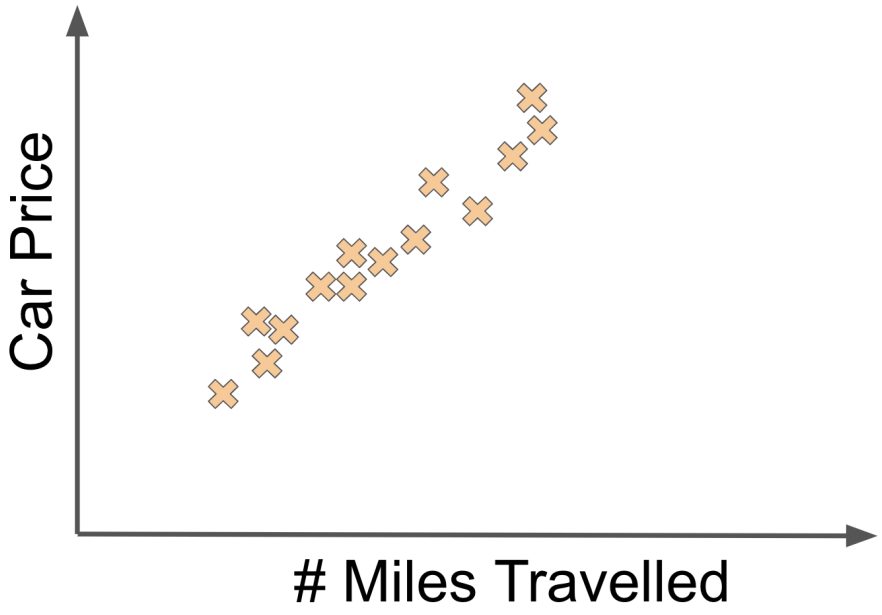

Figure 2: Linear regression car data plot

In this case, it seems that there is a linear relationship between the miles driven and the price. If we fit a reasonable looking line to the data, it could look like Figure 3.

We could describe the model for our car price dataset as a mathematical function of the form: $F(X) = A_1 \cdot X + A_0$

Here A_1 and A_0 are called **weights** and these are the values that determine how our linear function behaves on different inputs. All supervised learning algorithms have some set of weights that determine how the algorithm behaves on different inputs, and determining the right weights is really at the core of what we call *learning*.

Let's say that the linear fit above was associated with the weights $A_1 = 5$ and $A_0 = 0.5$. Now if we changed the A_0 value to something like $A_0 = -2$, we might get the linear fit in Figure 4.

Or imagine that we thought that there was a much steeper relationship between the number of miles driven and a car's price. In other words, we think the A_1 value (which here determines the slope of the line) should be a bigger value such as 8. Our linear fit would then look like Figure 5.

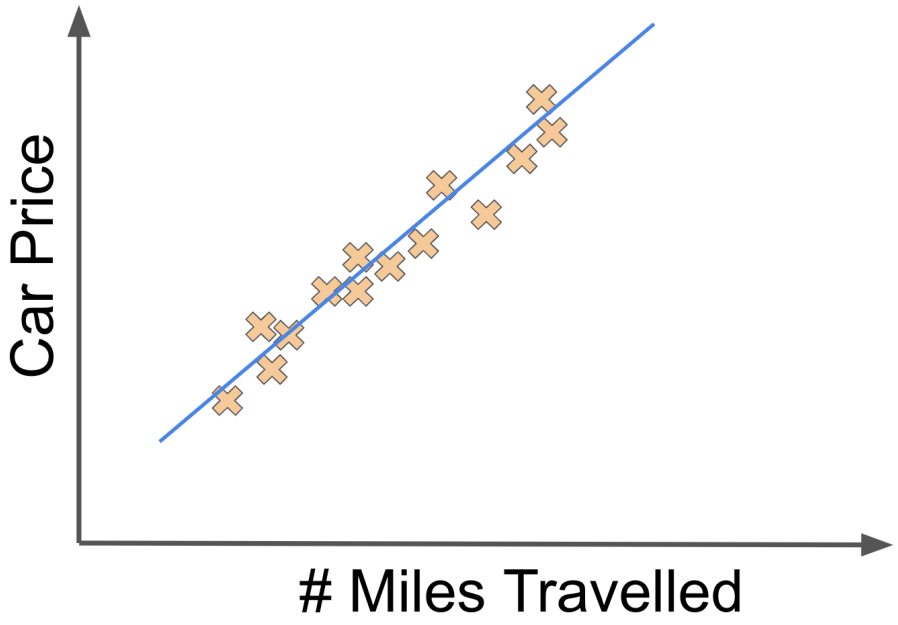

Figure 3: Linear regression with good linear fit

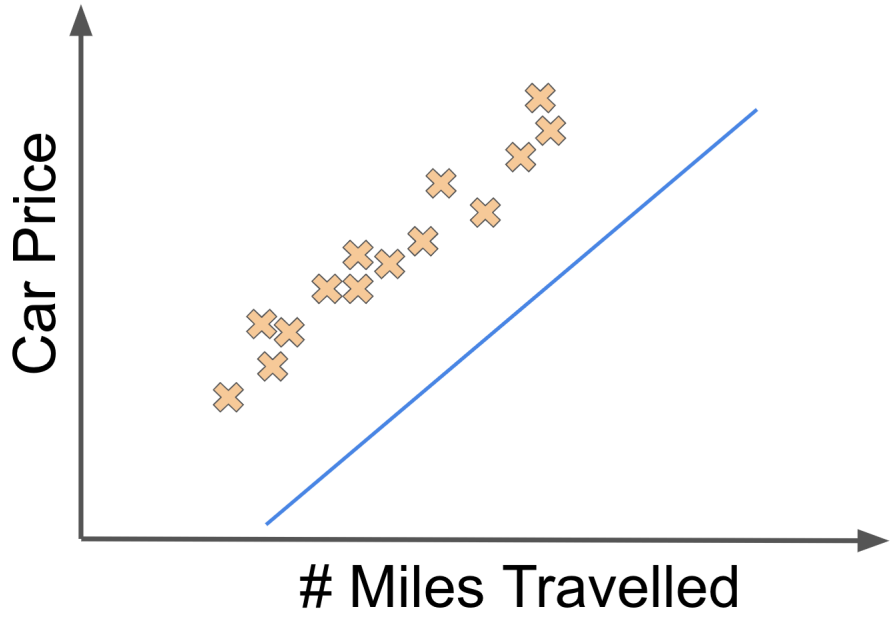

Figure 4: Linear regression shifted down

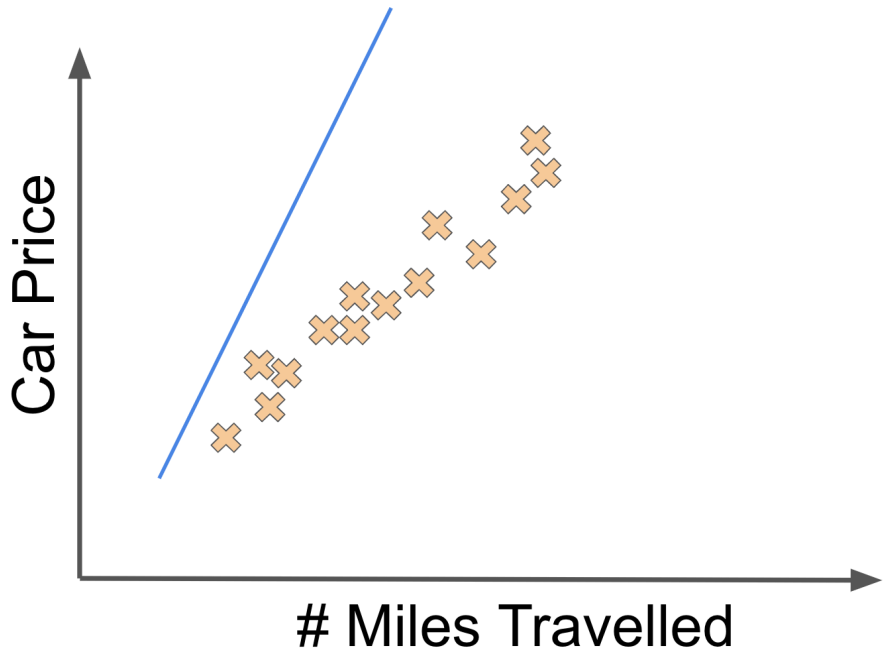

Figure 5: Linear regression higher slope

A Training Paradigm

These varieties of linear fits raise the question: how do we actually learn the weights of this model or any machine learning model in general? In particular, how can we leverage the fact that we have the correct labels for the cars in our dataset?

Training and evaluating a machine learning model involves using something called a **cost function**. In the case of supervised learning, a cost function is a measure of how much the *predicted* labels outputted by our model deviate from the *true* labels. Ideally we would like the deviation between the two to be small, and so we want to minimize the value of our cost function.

A common cost function used for evaluating linear regression models is called the **least-squares cost function**. Let's say that we have n datapoints in our data set.

This could look like $[(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)]$.

If we are learning a function $F(X)$, the least-squares regression model seeks to minimize:

$$C(X) = \frac{1}{2} \cdot \sum_{i=1}^{n} (F(X_i) - Y_i)^2$$

The deviation between our predicted output ($F(X)$) and the true output (Y) is defined as a **residual**. The least-squares cost is trying to minimize the sum of the squares of the residuals (multiplied by a constant in this case).

Here it is important to note that $F(X)$ is a function of the weights of our model. In our motivating example, $F(X)$ would be a function of A_1 and A_0. The values of A_1 and A_0 that produce our optimal model are the values which achieve the minimal value of $C(X)$.

How do we actually compute the weights that achieve the minimal values of our cost? Here, as with all machine learning models, we have two options: an **analytical** or a **numerical** solution. In an **analytical** solution, we seek to find an exact closed-form expression for the optimal value. In this particular case, that involves using standard calculus optimization. We would take the gradients (which are just fancy derivatives) of the cost function with respect to the weights, set those gradients to 0, and then solve for the weights that achieve the 0 gradient.

This technique is nice because once we have computed the closed-form expression for the gradients, we can get the optimal weight values for any new data. Here we are able to develop an analytical solution in the case of linear regression with a least-squares cost function.

However, not all models have nice well-formed gradient expressions that allow us to solve for the global optimum of a problem. For these problems, we must turn to **numerical** methods. Numerical methods typically involve a step-wise update procedure that iteratively brings weights closer to their optimal value. Here we again compute the gradients with respect to all the weights and then apply the following update for each weight A:

$$A_{new} = A_{old} - \alpha \cdot \frac{\partial C}{\partial A}$$

We continue applying these updates for some number of iterations until our weight values converge, by which I mean to say they don't change too much from one iteration to the next. This very important numerical optimization procedure is called **gradient descent**.

Note in our expression for gradient descent above, we have this magical alpha (α) value being multiplied to the gradient. Alpha is an example of what is called in machine learning a **hyperparameter**. The value of this hyperparameter alpha determines how quickly updates are made to our weights. We are free to adjust the value so that gradient descent converges more quickly.

Many machine learning algorithms have their own hyperparameters that we can adjust and fine-tune to achieve better performance in our algorithm. For some algorithms, such as in deep learning, hyperparameter tuning is a super important task that can drastically impact how good of a system we build.

In the case of linear regression with a least-squares cost we are guaranteed that gradient descent will eventually converge to the optimal weight values. However, certain optimization problems don't have that guarantee, and the best we can hope for is that gradient descent converges to something close to the global optimum. A prominent example of models with this behavior are deep learning models, which we will discuss in greater depth later.

An important point to keep in mind is that in the original linear model we proposed we only have two weights, A_1 and A_0. But what if we really believed that car prices were a function of two features, the number of miles driven and the size of the trunk space in cubic feet?

Now if we wanted to train a linear regression model, our dataset would have to include the number of miles driven, the size of the trunk space, and the price for every car. We would also now have three weights in our linear regression model: A_0, A_1, and A_2.

Furthermore, our data would now exist in a 3-d coordinate system, not a 2-d one. However, we could use the same least-squares cost function to optimize our model. As we increase the number of features, the same basic algorithmic considerations apply with a few caveats. We will discuss these caveats when we discuss the bias-variance tradeoff.

When Does a Linear Fit Fit?

When does linear regression work well as a modelling algorithm choice? In practice, it turns out that linear regression works best when there actually *is* a linear relationship between the inputs and the outputs of your data. For example, if our car price data looked like Figure 6.

then linear regression probably would not be a good modelling choice.

When we are building a machine learning system, there are a few factors that we have to determine. First off, we need to extract the correct features from our data. This step is crucial!

In fact, this step is so important that for decades the contributions of many artificial intelligence papers were just different set of features to use for a particular problem domain. This style of paper has become not as prevalent with the resurgence of deep learning .

After selecting features, we need to pick the right modelling algorithm to fit. For example, if we think there is a linear relationship between our inputs and outputs, a linear regression may make sense. However, if we don't think a line

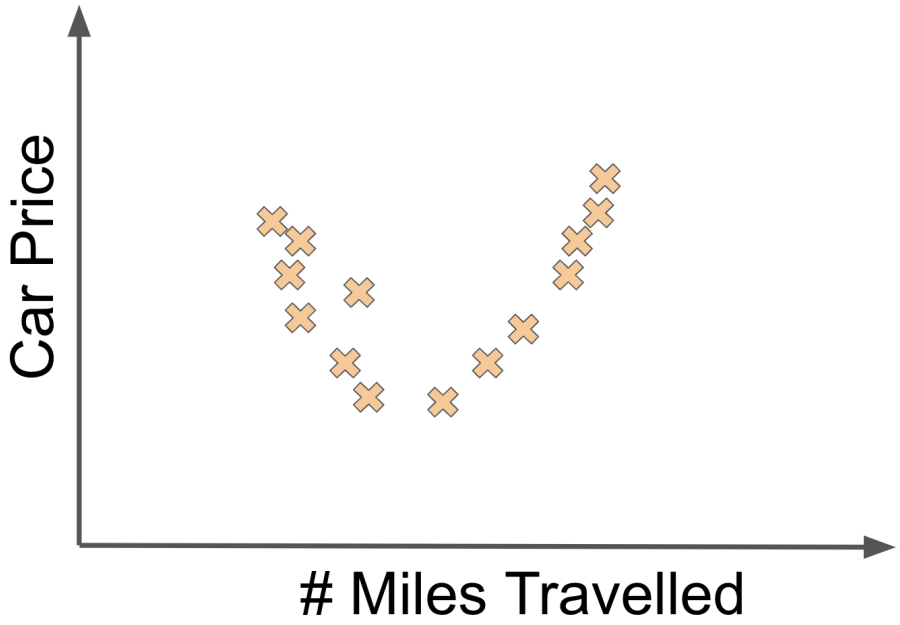

Figure 6: Non-linear data

makes sense, we would pick a different model that has some different assumptions about the underlying structure of the data. We will investigate many other classes of models with their assumptions about the data in later sections.

Test Your Knowledge

Definition

Cost Function

Training Techniques

Determine the Coefficients

Complete Implementation

Introduction to Logistic Regression

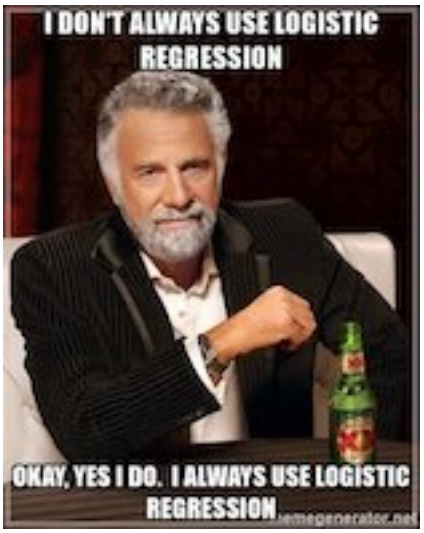

Figure 7: Source Twitter

In this section, we will continue with our study of supervised learning by exploring our first classification algorithm. A vast majority of problems tackled in the real world involve classification, from image labelling, to spam detection, to predicting whether it will be sunny tomorrow, so we are at an exciting milestone in our study of artificial intelligence!

Ironically, the first **classification** algorithm we will investigate is called *logistic regression* . Putting aside name confusion for now, the form of logistic regression we will look at is for binary classification tasks, where we only output two possible labels.

Model Definition

To motivate logistic regression, let us begin with a modified version of our running car example from the last section on linear regression.

Rather than build a system to predict the price of a car, we will build a system that is provided a set of features of a car and will determine whether the car is *expensive* or *cheap*. In particular, we will extract a few different features from the cars in our dataset such as:

- the size of the trunk,
- the number of miles driven
- who the car manufacturer is.

Let's call these features X_1, X_2, and X_3. We will consolidate these features into a single vector variable $X = (X1, X2, X3)$. These features will be fed into a mathematical function $F(X)$, to get a probability of whether or not the car is expensive.

In other words, we will compute $F(X)$ (where the function F is unspecified for now), and this will give us a probability between 0 and 1. We will then say that if our probability is greater than or equal to 0.5, we will label our prediction, *expensive*, otherwise it will be *cheap*. This can be expressed mathematically as follows:

$$\text{Prediction} = \begin{cases} \text{expensive} & \text{if } F(X) \geq 0.5 \\ \text{cheap} & \text{if } F(X) < 0.5 \end{cases}$$

Note, we could have reversed the labels and said a probability greater than 0.5 is *cheap* and it would not have made a difference. The only important thing is to be consistent once you've selected a labelling scheme!

So what exactly is going on in that $F(X)$ function? The logistic regression model describes the relationship between our input car features and the output probabilities through a very particular mathematical formulation. Given our input features X_1, X_2, and X_3 the formulation is as follows:

$$F(X) = \frac{1}{1 + e^{-(A_1 \cdot X_1 + A_2 \cdot X_2 + A_3 \cdot X_3)}}$$

where here the weights of our model that we have to learn are A_1, A_2, and $A3$. Ok, this gives us the mathematical form, but let's try to gain some visual intuition for the formulation. What does this function actually look like?

It turns out that this function of the inputs, which is called a *sigmoid*, has a very interesting form shown in Figure 8.

Notice how the mathematical form of the logistic regression function has a sort of elongated S shape. The probability returned is exactly 0.5 when the input is 0, and the probability plateaus at 1 as our input gets larger. It also plateaus at 0 as the inputs get much smaller.

Logistic regression is also interesting because we are taking our feature of inputs, transforming them via a *linear* combination of the weights (namely $A1 \cdot X1 + A2 \cdot X2 + A3 \cdot X3$), and then running them through a *nonlinear* function.

Training the Model

How do we train the weights of a logistic regression model? Let's assume that we have a dataset of **n** cars with their associated true labels: $[(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)]$. We won't dive into the mathematical details,

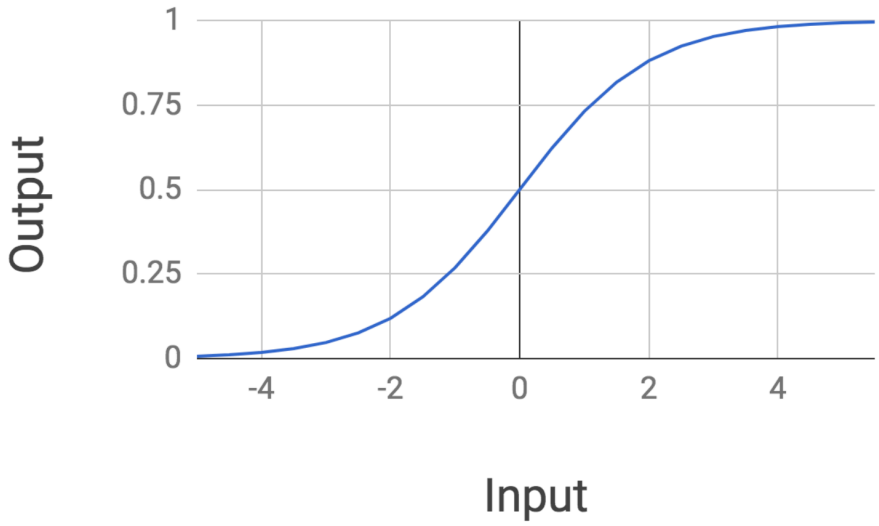

Figure 8: Logistic regression sigmoid graph

but it turns out we can write an expression for the total probability of our dataset which looks as follows:

$$\text{Probability} = \prod_{i=1}^{n} (F(X_i))^{Y_i} \cdot (1 - F(X_i))^{1-Y_i}$$

For our purposes, understand that our aim will be to maximize this probability. We can do by taking the derivative with respect to our weights and setting the derivative to 0. We can then run gradient descent using our computed gradient to get our optimal weights. This is analogous to the procedure used for numerically optimizing a linear regression model in the linear regression section.

Final Thoughts

Logistic regression can also be applied to problems with more than just binary outputs for a given set of inputs. In this case, the model is called **multinomial logistic regression**.

For this section we have restricted ourselves to binary outputs because it is a natural place to start. That being said, multinomial logistic regression is especially important for more sophisticated models used in deep learning.

When is logistic regression useful? In practice, logistic regression is a very nice off-the-shelf algorithm to begin with when you are doing any type of classification.

It has a fairly straightforward description, can be trained fairly quickly through techniques such as gradient descent because of its nice derivative, and often works well in practice. It is used frequently in biostatistical applications where there are many binary classification problems.

Test Your Knowledge

Implement the Algorithm

Why So Naive, Bayes?

Figure 9: Source Medium

I hope you're excited to learn about another fantastic class of machine learning models: **Naive Bayes**. Naive Bayes is wonderful because its core assumptions can be described in about a sentence, and yet it is *immensely* useful in many different problems.

But before we dive into the specifics of Naive Bayes, we should spend some time discussing the difference between two categories of machine learning models: **discriminative** and **generative** models.

Beginnings

Naive Bayes will be the first generative algorithm we look at, though other common examples include hidden markov models, probabilitistic context-free grammars, and the more hip generative adversarial networks.

Recall that in our running car example of the past few sections, we are given a dataset of cars along with labels indicating whether they are **cheap** or **expensive**. From each car, we have extracted a set of input features such as the size

of the trunk, the number of miles driven, and who the car manufacturer is.

We start from the distribution we are trying to learn $P(X_1, X_2, X_3, Y)$. We can expand the distribution using a few rules of probability along with Bayes' Rule:

$$P(X_1, X_2, X_3, Y) = P(Y) \cdot P(X_1|Y) \cdot P(X_2|X_1, Y) \cdot P(X_3|X_1, X_2, Y)$$

This formulation was derived from a few applications of the chain rule of probability. Now we get to the big underlying assumption of the Naive Bayes model.

We now assume the input features are **conditionally independent given** the outputs. In English, what that means is that for a given feature X_2, if we know the label Y, then knowing the value of an additional feature X_1 doesn't offer us any more information about X_2.

Mathematically, this is written as $P(X_2|X_1, Y) = P(X_2|Y)$. This allows us to simplify the right side of our probability expression substantially:

$$P(X_1, X_2, X_3, Y) = P(Y) \cdot P(X_1|Y) \cdot P(X_2|Y) \cdot P(X_3|Y)$$

And with that, we have the expression we need to train our model!

Naive Training

So, how do we actually train the model? In practice, to get the most likely label for a given input, we need to compute these values $P(X_1|Y)$, $P(X_2|Y)$, etc. Computing these values can be done through the very complicated process of counting!

Let's take a concrete example to illustrate the procedure. For our car example, let's say Y represents **cheap** and X_1 represents the feature of a car's manufacturer.

Let's say we have a new car manufactured by **Honda**. In order to compute $P(X_1 = \text{Honda}|Y = \text{cheap})$, we simply count all the times in our dataset we had a car manufactured by **Honda** that was **cheap**.

Assume our dataset had 10 cheap, Honda cars. We then normalize that value by the total number of cheap cars we have in our dataset. Let's say we had 25 cheap cars in total. We thus get $P(X_1 = \text{Honda}|Y = \text{cheap}) = 10/25 = 2/5$.

We can compute similar expressions (e.g. $P(X_2 = 40000 \text{ miles driven}|Y = \text{cheap})$) for all the features of our new car. We then compute an aggregated probability that the car is **cheap** by multiplying all these individual expressions together.

We can compute a similar expression for the probability that our car is **expensive**. We then assign the car the label with the higher probability. That

outlines how we both train our model by counting what are called *feature-label co-occurrences* and then use these values to compute labels for new cars.

Final Thoughts

Naive Bayes is a super useful algorithm because its extremely strong independence assumptions make it a fairly easy model to train. Moreover, in spite of these independence assumptions, it is still extremely powerful and has been used on problems such as spam filtering in some early version email messaging clients.

In addition, it is a widely used technique in a variety of natural language processing problems such as document classification (determining whether a book was written by Shakespeare or not) and also in medical analysis (determining if certain patient features are indicative of an illness or not).

However the same reason Naive Bayes is such an easy model to train (namely its strong independence assumptions) also makes it not a clear fit for certain other problems. For example, if we have a strong suspicion that certain features in a problem are highly correlated, then Naive Bayes may not be a good fit.

One example of this could be if we are using the language in an email message to label whether it has positive or negative sentiment, and we use features for whether or not a message contains certain words.

The presence of a given swear word would be highly correlated with the appearance of any other swear word, but Naive Bayes would disregard this correlation by making false independence assumptions. Our model could then severely underperform because it is ignoring information about the data. This is something to be careful about when using this model!

Test Your Knowledge

Naive Bayes Assumption

Smoothing

Basics of Support Vector Machines

Figure 10: Support vector meme

In this section, we will explore a family of classification algorithms that has a very different theoretical motivation from ones we've seen previously. We will be looking at **support vector machines**.

Back in the early 2000s before deep learning surged to the forefront of AI, support vector machines were the cool kid on the block. Even today, support vector machines are still one of the best go-to algorithms for a new classification task because of their powerful ability to represent very diverse types of statistical relationships in data as well as their ease of training.

Max-Margin Motivations

Let's begin by motivating support vector machines. Recall our ridiculously overused classification problem: identifying whether a car is **cheap** or **expensive**, based on a set of features of that car.

Imagine that we are plotting some of our data in 2-dimensional feature space. In other words, we are only extracting two features, X_1 and X_2, from each car for building our model. Furthermore, let's label each point BLUE if it represents a car that is **cheap** and RED if it represents a car that is labelled **expensive**.

We will try to learn a linear model of our features, parameterized by the weights W_1 and W_2. Our model will output BLUE if $W_1 \cdot X_1 + W_2 \cdot X_2 < 0$ and RED if $W_1 \cdot X_1 + W_2 \cdot X_2 >= 0$.

This model will describe a linear separator of a collection of data. That linear separator of our data could look as shown in Figure 11.

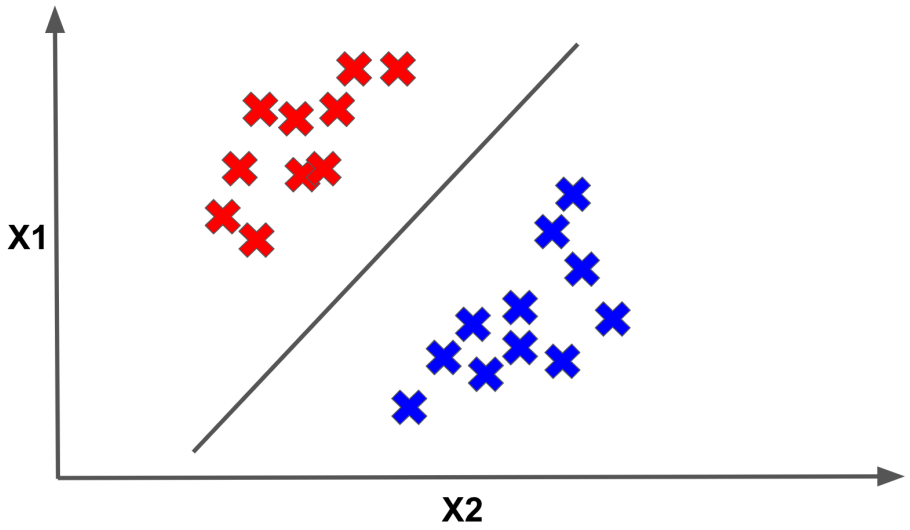

Figure 11: Linear separator of our data

Here the datapoints above the line are classified as RED, while those below are classified as BLUE. The important thing to see is that this is *only one* possible linear separator of the data. However, we could envision another separator that could separate the dataset into two colored halves just as well (shown in Figure 12).

In fact, there are an *infinite* number of possible separators that would split the data perfectly! How do we choose among them?

Consider how we decided to label a datapoint as RED or BLUE. We computed $W_1 \cdot X_1 + W_2 \cdot X_2$ and said if it was negative, we labelled the point BLUE. Otherwise we labelled it RED. Intuitively, it seems that if the quantity $W_1 \cdot X_1 + W_2 \cdot X_2$ for a given point is 15, we are more confident that it should be RED than one for which the quantity is 1.

Alternatively, the more negative the quantity is for a point the more confident we are that it should be BLUE. In fact, we can use this value to judge our confidence for every single point!

Geometrically, this confidence for a point's label can be represented as the perpendicular distance from the point to the separator. In Figure , we have designated the perpendicular distances to the separator for several sample points with the GREEN lines.

Now imagine if we took the minimum distance to the separator across all the

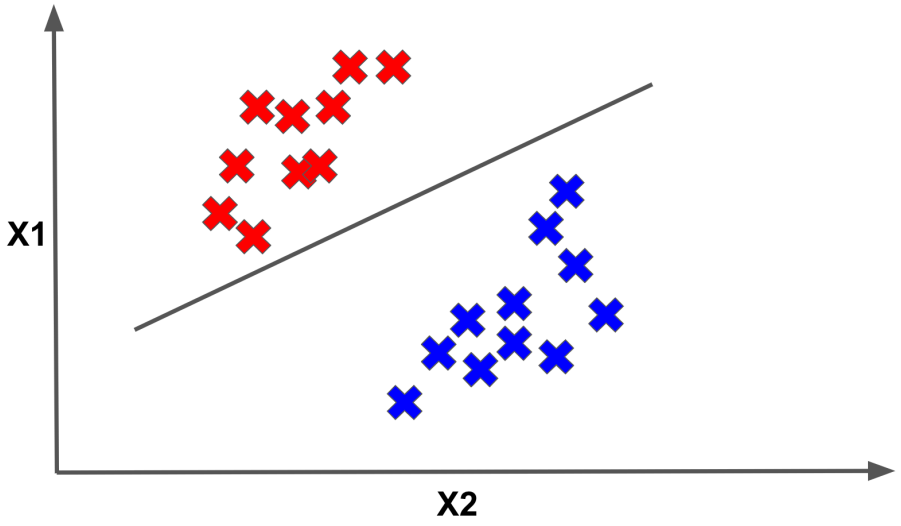

Figure 12: Alternative linear separator of our data

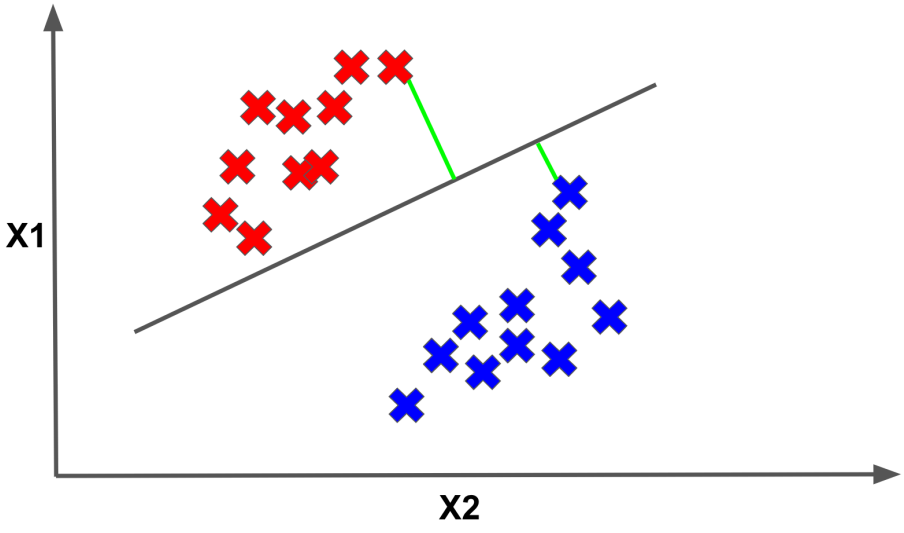

Figure 13: Linear separator of our data with distances shown

points in the dataset. This minimum value is called the **margin**. In Figure 14 we show the point that achieves the margin for the given separator.

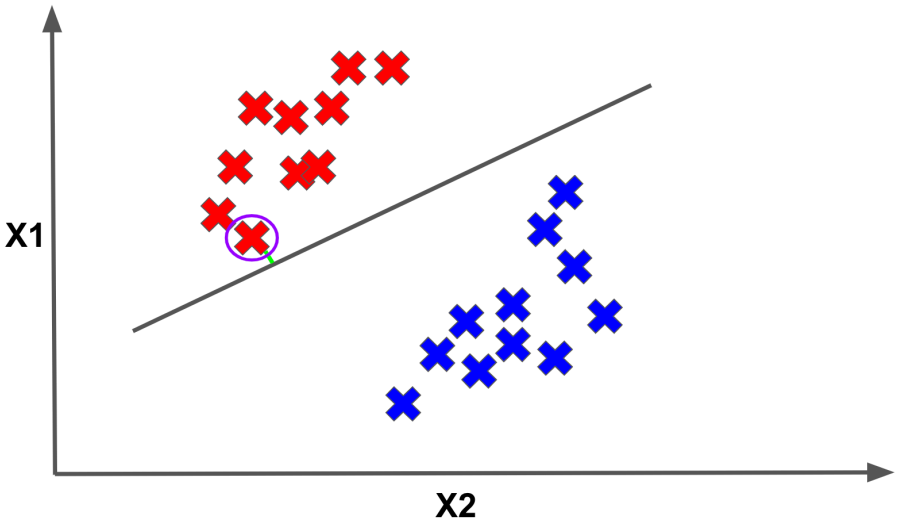

Figure 14: Margins shown

Optimal margins are at the core of support vector machine theory, and they inform how we will pick the "best" linear separator among the infinite choices. The way we will do this is by picking the linear separator that **maximizes the margin**.

Put another way, each separator defines a certain margin (i.e. an associated minimum perpendicular distance to the separator). Then, among the infinite possible separators, we will pick the separator that maximizes the margin.

Make sure you really understand that last statement! Support vector machines are often called **max-margin classifiers** for that exact reason. Given an optimal margin linear separator, the points with the smallest margins that are closest to the linear separator are called the **support vectors**.

Training the SVM

So, how do we train a support vector machine? The full mathematical details to describe the cost function we want to optimize are a bit beyond the scope of this section, but we will give a brief description of the training cost function.

Given a collection of datapoints $[(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)]$ (where X are the input features and Y are the correct output labels) and a vector of weights W for our input features, finding the optimal margin classifier amounts to solving

$$\min_W \frac{1}{2}||W||^2$$

$$\text{such that } Y_i(W^T X_i) \geq 1 \text{ for all } i = 1, ..., n$$

This cost function is basically a fancy way of saying that we want maximize the margin, while ensuring the margin of each datapoint is greater than or equal to 1.

It turns out finding the optimum for this cost function is a convex optimization problem, which means there exist standard algorithms for solving the problem. This is good news for us because it means optimally training a support vector machine is a tractable problem!

As a point of comparison, there exist some problems for which we cannot find the optimal value in a computationally reasonable time. It may even take exponential time to find the optimum!

The Kernel Trick

Thus far we have motivated support vector machines by only focusing on linear separators of data. It turns out that support vector machines can actually learn nonlinear separators, which is why they are such a powerful model class!

To motivate this idea, imagine that we have an input point with three features $X = (X_1, X_2, X_3)$. We can transform the 3-dimensional vector X into a higher-dimensional vector by applying a transformation function as follows: $T(X) = (X_1, X_2, X_3, X_1 \cdot X_1, X_2 \cdot X_2, X_3 \cdot X_3)$.

Our transformation function did two things:

1) it created a 6-dimensional vector from a 3-dimensional one
2) it added nonlinear interactions in our features by having certain new features be the squares of the original features.

Transforming features into a higher-dimensional space has some perks, namely that some data which is not linearly separable in a certain-dimensional space could become linearly separable when we project it into the higher-dimensional space.

As a motivating example, consider the data in Figure 15 which is not linearly separable in 2-d space.

It becomes separable in 3-dimensional space through an appropriate feature transformation Figure **??**.

Such input feature transformations are particularly useful in support vector machines. It turns out that support vector machines formalize the notion of these feature transformation function through something called a **kernel**.

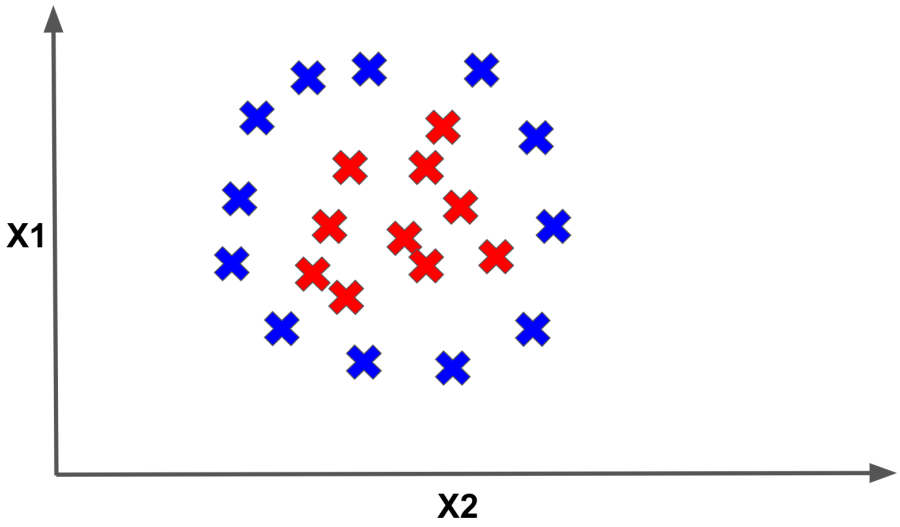

Figure 15: Non-separable data

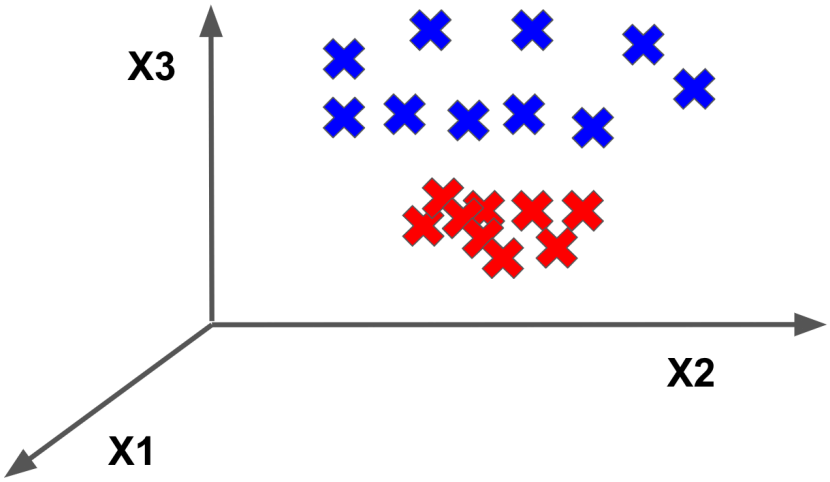

Figure 16: Separable data **??**

Without getting too much into the mathematical details, a kernel allows a support vector machine to learn using these transformed features in higher-dimensional space in a much more computationally efficient way. This is called the **kernel trick**.

Examples of kernels that are frequently used when learning a support vector machine model include the polynomial kernel and the radial basis function kernel.

A really crazy mathematical property of the radial basis function is that it can be interpreted as a mapping for an infinite-dimensional feature space! And yet we can still use it to do effective learning in our models. That's insane

Slacking Off

Thus far, we have assumed that the data we are working with is always linearly separable. And if it wasn't to begin with, we assumed we could project it into a higher-dimensional feature space where it would be linearly separable and the kernel trick would do all the heavy lifting.

However, our data may not always be linearly separable. We may find ourselves in a situation as shown in Figure 17 where it is impossible to linearly separate the data.

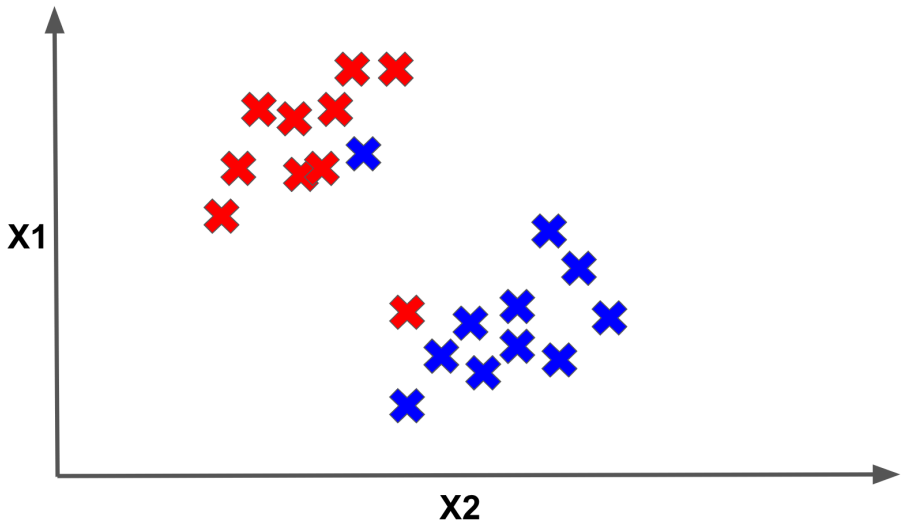

Figure 17: Non-linearly separable data

Moreover, recall that our original support vector machine cost function enforced the constraint that every point needs to have a margin greater than or equal to 1.

But perhaps this condition is too strict for all cases. For example, enforcing this condition for all points may make our model very susceptible to outliers as shown in Figure 18.

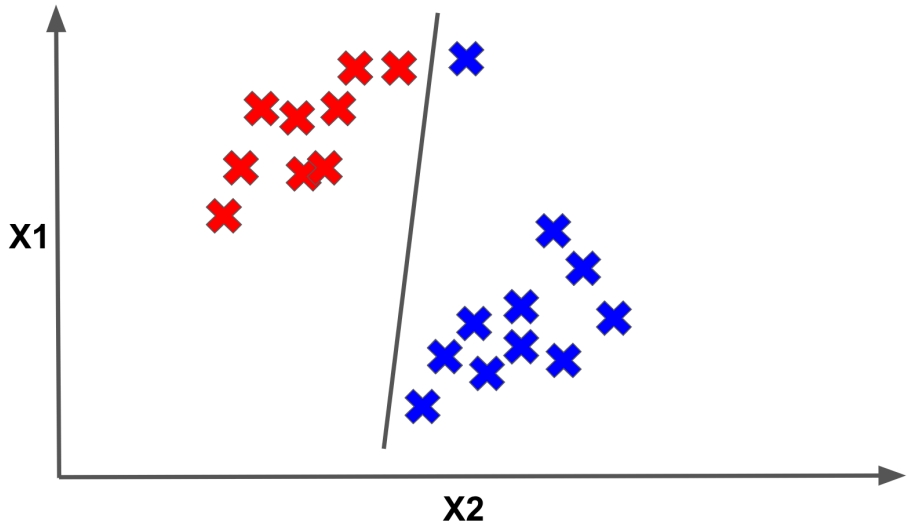

Figure 18: Support vector machine outlier diagram

To combat this problem, we can introduce what are called **slack penalties** into our support vector machine objective that allow certain points to have margins potentially smaller than 1. With these penalties, our new cost looks as follows:

$$\min_W \frac{1}{2}||W||^2 + K \cdot \sum_i s_i$$

$$\text{such that } Y_i(W^T X_i) \geq 1 - s_i \text{ for all } i = 1, ..., n$$

$$s_i \geq 0 \text{ for all } i = 1, ..., m$$

Here the s_i are our slack variables. Notice that we have changed the expression we are trying to minimize. The value **K** is what is called a *hyperparameter*.

In a nutshell, this means we can adjust **K** to determine how much we want our model to focus on minimizing the new term in our cost function or the old term. This is our first example of a very important concept in machine learning called **regularization**.

We will discuss regularization in a lot more detail in the next section. For now, just understand that this new addition makes the support vector machine more robust to nonlinearly-separable data.

Final Thoughts

Support vector machines are very powerful model. They are great for handling linearly separable data and with their various extensions can adequately handle nonlinearly separable data scenarios

As part of the machine learning toolkit, it is a great go-to model to try when starting on a new problem. However, one of its downsides is that using certain kernels, such as the radial basis function kernel, can sometimes make the model training slower, so be wary of that.

Test Your Knowledge

Support Vector Description

Slack Variables

Decision Trees

Figure 19: Decision tree meme

In this section we will discuss a wonderful model that is not only very powerful but has very convenient interpretability. In fact, the underlying structure of the model has very clear analogies to how humans actually make decisions.

The model we will be discussing is called **decision trees**. Decision trees are so robust that some machine learning practitioners believe they offer the best out-of-the-box performance on new problem domains. Sound exciting? Let's dive in!

Motivations

To motivate decision trees, let's start with a hypothetical problem: deciding whether or not we will pass our artificial intelligence exam this week. In other words we want to build a model that, given a dataset of past exam experiences, will output either a **YES** we will pass or **NO** we won't. This will be done by extracting some features from the dataset. So what features seem relevant for this problem?

Well, for starters it may be important how much sleep we get the night before the exam. In particular, if we get 8 or more hours of sleep, we have a much higher chance of doing well (no scientific evidence for this number but it sounds like it could be true). Now for an arbitrary exam that we want to classify, we could represent our logic as a flowchart shown in Figure 20.

Note, we could stop here if we wanted and let our entire model involve this

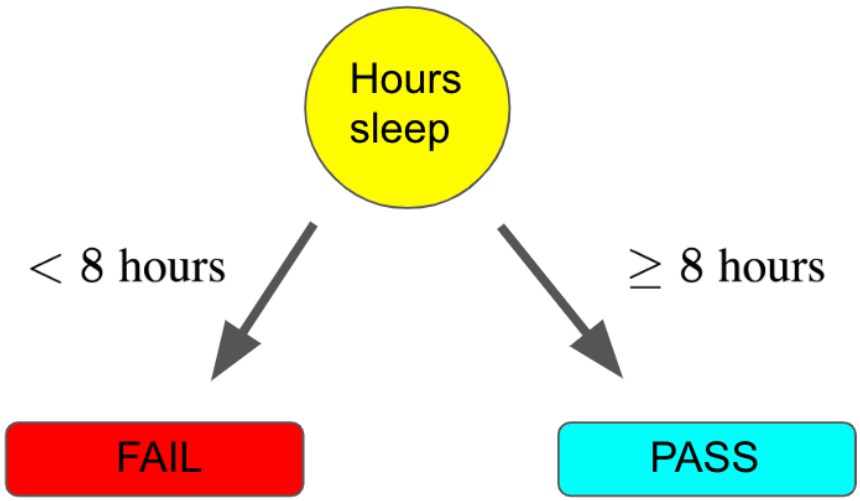

Figure 20: Depth 1 decision tree

single sleep question. But it seems like we could do better than this, since we are disregarding so many other possible indicators for success on our exam. Let's continue to make our model more powerful!

Another possibly useful feature is whether the exam is in the morning or evening. Perhaps we aren't really morning people and prefer to do things later in the day. We can continue to build out our flowchart shown in Figure 21.

Notice how with this tree, there is a flow where we get less than 8 hours of sleep but the exam is in the evening and so we still pass! The behavior of our model for different features is all predicated on the dataset we are using to train the model.

In this case, we must have had a collection of past exam datapoints where we got less than 8 hours of sleep and took the exam in the evening, and hence we passed the exam.

Now, we could continue to build out our model in this fashion. We may utilize other attributes of our data such as the day of the week the exam is on. It turns out this flow chart is our first **decision tree**!

Decision trees are defined by this hierarchical structure starting at some root. Note that here our tree is technically inverted, since it is growing downwards.

However, this is traditionally how decision trees are visualized. At each level, we select a feature to use for a binary split of our data. At the leaves of the tree, we predict one of the possible output labels of our problem.

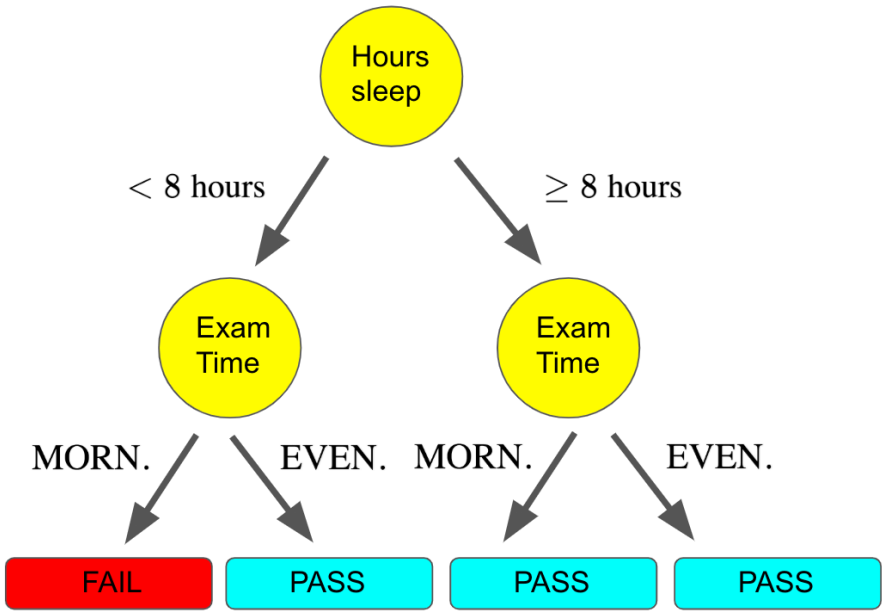

Figure 21: Depth 2 decision tree

Decision trees are a very nice model family because once we've built the tree, it's relatively straightforward to understand how a model makes a prediction for a given input. It literally does so in the same way we would use a flowchart, starting at the top and traversing the branches according to the features of our input. Simple, right?

Well, yes, assuming we already have a tree built out. But how do we choose the features to split at each respective level of the tree? There are a lot of options, and they could drastically impact our model's predictions. We will explore how to build a decision tree next.

Growing the Tree

It turns out that building an optimal tree is **NP-complete**, which is a fancy way of saying that given an arbitrary dataset, there is no *computationally efficient* way to determine an optimal tree. Because of this, building a tree always involves some sort of **greedy algorithm**.

What this means is that when we are building our tree, we will choose features for splits with metrics based on *locally optimal* considerations rather than *globally optimal* ones. As a consequence, we may not get the absolute best-performing tree, but we get the benefit of an optimization procedure that can be completed in a reasonable amount of time.

There are many metrics for picking these features to split, but we will focus on one commonly used metric called **information gain**.

Information gain is very related to the concept of **entropy** in information theory. In general, decision tree algorithms using information gain seek to build a tree top-down by selecting at each level the feature that results in the **largest information gain**. First we define the entropy of a tree T as follows:

$$\text{Entropy}(T) = -\sum_{i=1}^{n} p_i \log p_i$$

In our equation above, p_i denotes the fraction of a given label in a set of datapoints in the tree. What does this mean? To make this concrete, let's go back to our example of determining the outcome of our artificial intelligence exam.

Imagine that we had a training set of 20 examples (a lot of our friends have taken artificial intelligence courses). Recall that our example had two potential labels: **YES** we pass or **NO** we don't. Of those 20 samples, 12 friends passed, and 8 did not. Therefore, at the beginning of our tree building, before we have made any feature split, the entropy of our tree is:

$$\begin{aligned}\text{Entropy}(T_{\text{original}}) &= -\frac{12}{20}\log\frac{12}{20} - \frac{8}{20}\log\frac{8}{20} \\ &\approx 0.2922\end{aligned}$$

How does this change when we choose a feature to split for our first level? Let's assume that when we split on whether or not we slept for 8 hours, we get two sets of datapoints.

For the set with less than 8 hours of sleep, we have 10 samples, of which 7 did not pass and 3 passed (this is why you need to get sleep before tests). For the set with 8 or more hours of sleep, we have 20 - 10 = 10 samples, of which 9 passed and 1 did not. Hence the entropy for the tree's children with this split is

$$\begin{aligned}\text{Entropy}(\text{Split}_{<8 \text{ hours}}) &= -\frac{7}{10}\log\frac{7}{10} - \frac{3}{10}\log\frac{3}{10} \\ &\approx 0.265\end{aligned}$$

for the set with less than 8 hours and

$$\begin{aligned}\text{Entropy}(\text{Split}_{\geq 8 \text{ hours}}) &= -\frac{9}{10}\log\frac{9}{10} - \frac{1}{10}\log\frac{1}{10} \\ &\approx 0.1412\end{aligned}$$

for the set with 8 or more hours. Let T be our original tree and f be the feature we are considering splitting on. The information gain is defined as follows:

$$\text{Info Gain}(T, f) = \text{Entropy} - \text{Entropy}(T|f)$$

In plain English, this is saying that the information gain is equal to the entropy of the tree before the feature split minus the weighted sum of the entropy of the tree's children made by the split. Ok, not quite plain English.

Let's make this concrete. For our example, the information gain would be:

$$\begin{aligned}\text{Info Gain}(T, f) &= 0.2922 - \frac{10}{20} \cdot 0.265 - \frac{10}{20} \cdot 0.1412 \\ &\approx 0.0891\end{aligned}$$

Notice here that the $\frac{10}{20}$ fractions represent the weights on the entropies of the children, since there are 10 of the 20 possible datapoints in each child subtree.

Now when we are choosing a feature to split on, we compute the information gain for every single feature in a similar fashion and select the feature that produces the **largest information gain**.

We repeat this procedure at every single level of our tree, until some stopping criterion is met, such as when the information gain becomes 0. And that's how we build our decision tree!

Pruning the Tree

The extraordinary predictive power of decision trees comes with a cost. In practice, trees tend to have **high variance**. Don't worry too much about what that means exactly for now, as we will discuss it in greater detail when we talk about the bias-variance tradeoff.

For our purposes, this means that trees can learn a bit too much of our training data's specificity, which makes them less robust to new datapoints. This especially becomes a problem as we continue to grow out the depth of our tree, resulting in fewer and fewer datapoints in the subtrees that are formed. To deal with this issue, we typically employ some sort of **tree pruning** procedure.

There are a number of different ways of pruning our tree, though we will focus on a relatively simple one just to get a taste for what these pruning procedures look like. The procedure we will discuss is called **reduced error pruning**.

Reduced error pruning essentially starts with our fully-built tree and iteratively replaces each node with its most popular label. If the replacement does not affect the prediction accuracy, the change is kept, else it is undone. For example, assume we started with the tree in Figure 22.

reduced error pruning after one iteration, could leave us with the tree in Figure 23.

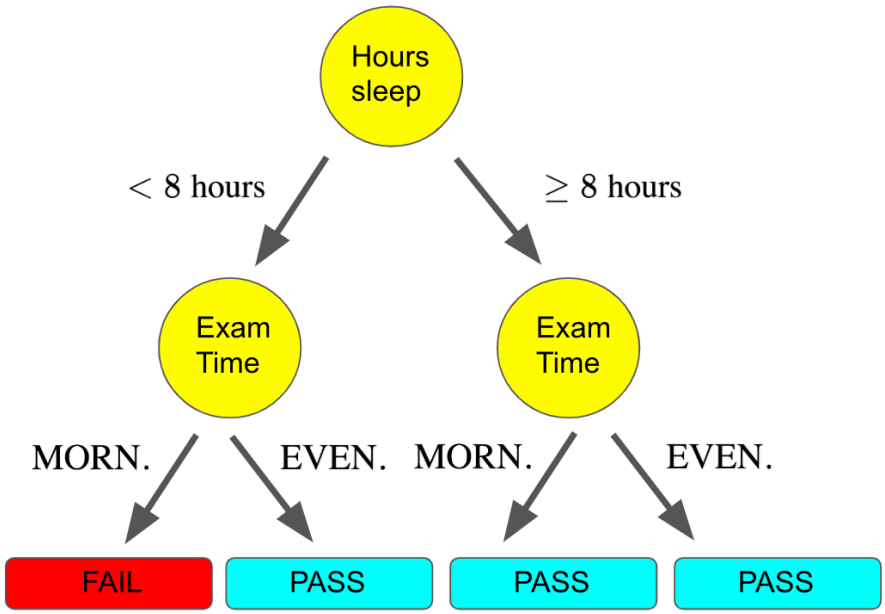

Figure 22: Depth 2 decision tree

This would happen if the number of FAIL labels on the two left branches exceeded the number of PASS labels. This could happen if there were 9 FAIL and 1 PASS points in the leftmost node, but only 1 FAIL and 3 PASS in the node next to it. Hence we would replace these two nodes with the majority label, which is FAIL.

Note, after we have done this pruning, we could certainly continue to prune other nodes in the tree. In fact, we *should* continue to prune as long as our resulting trees don't perform worse.

There are many more complex pruning techniques, but the important thing to remember is that all these techniques are a means of cutting down (pun intended) on the complexity of a given tree.

Final Thoughts

Now that we are wrapping up our tour of decision trees, we will end with a few thoughts. One observation is that for each split of our trees, we always used a binary split. You might feel that it seems very restrictive to only have binary splits at each level of the tree, and could understandably believe that a split with more than two children could be beneficial.

While your heart is in the right place, practically-speaking non-binary splits

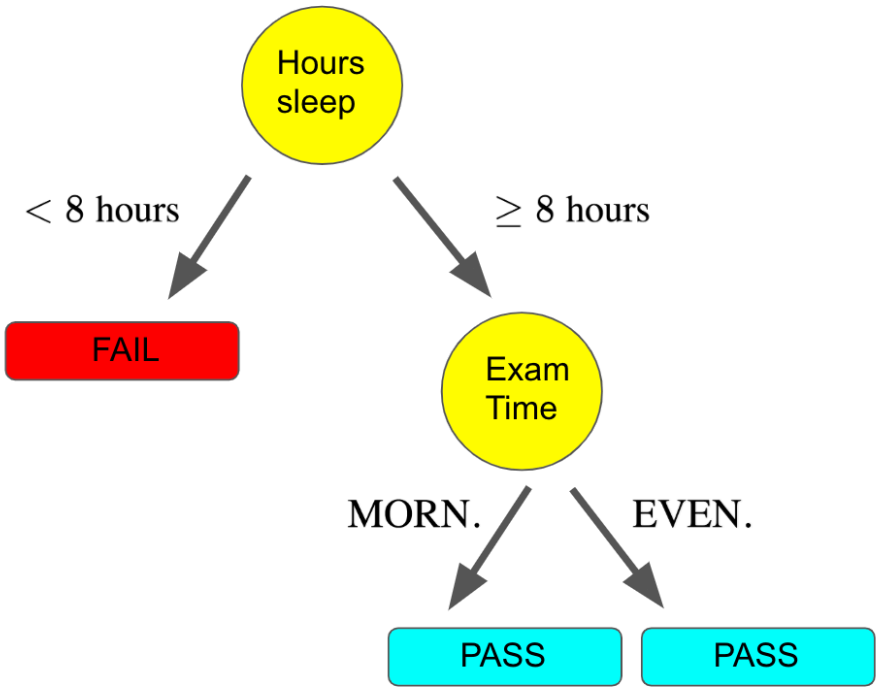

Figure 23: Pruned decision tree

don't always work very well, as they fragment the data too quickly leaving insufficient data for subsequent feature splits. Moreover, we can always also achieve a non-binary split with a series of consecutive binary splits, so it's not like non-binary splits give us any more expressive power.

Finally, while we only discussed a classification example, decision trees can also be applied to regression. This is yet another reason why they are so versatile!

Test Your Knowledge

Tree Labelling

Misclassification Rate

Tree Learning

Example Traversal

Decision Boundary

Tree Disadvantages

Node Impurity

Your Closest Neighbors

Figure 24: Knn meme

In this section, we are going to study an interesting flavor of supervised learning models known as **k-nearest neighbors**.

K-nearest neighbors is a bit of an outlier among models since it doesn't formally have a training procedure! Because of this, it is a relatively straightforward model to explain and implement.

The Model

K-nearest neighbors works as follows: assume we receive a new datapoint, P, that we are trying to predict an output for. To compute the output for this point, we find some number of datapoints in the training set that are closest to P, and we assign the majority label of its neighbors to P.

Woah, that was a mouthful! Let's break this down with a diagram. Assume we are dealing with a binary classification task, and we are trying to predict a label for an unknown P (shown in GREEN) as shown in Figure 25:

Let's say we use the **3 nearest neighbors** to classify our unknown point (Figure 26).

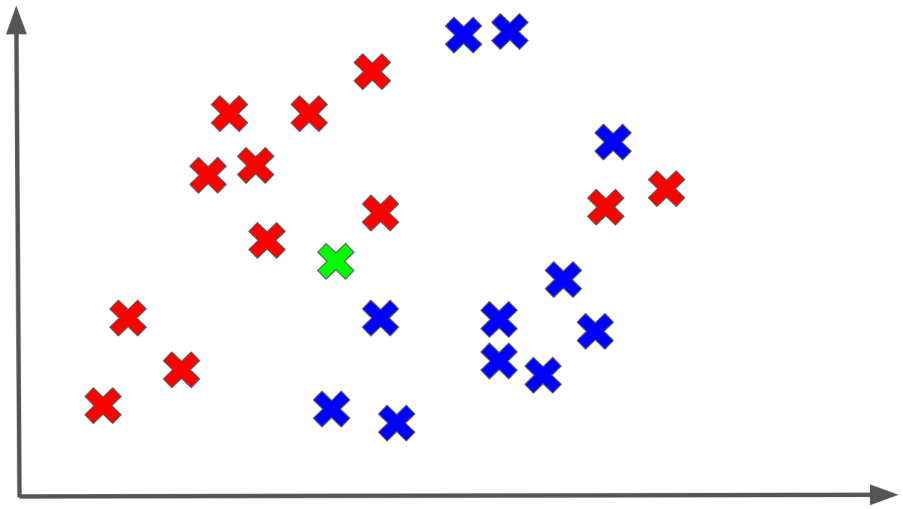

Figure 25: Unknown point

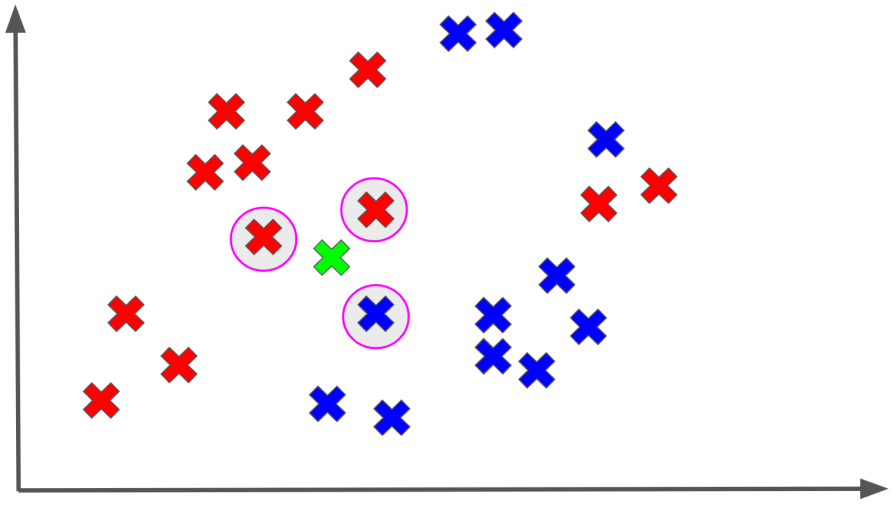

Figure 26: K nearest neighbors identified

Among the 3 nearest neighbors of P, the majority output is a red X, so we assign the RED label to P (Figure 27).

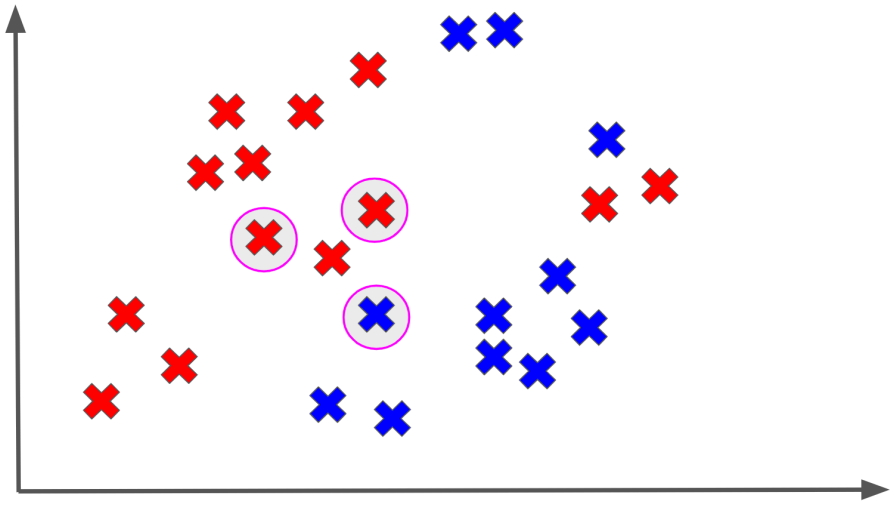

Figure 27: K nearest neighbors labelled

And that's basically k-nearest neighbors in a nutshell! The really important thing to realize is that we are not *really* fitting a model to our data.

What I mean is that because we need to find the labels of a point's nearest neighbors when we are trying to classify it, we actually need to *always* keep track of *all* of our training data.

Typically when we are training a supervised model, we understand that the training process may take a lot more time than the testing process, which is the point where we release a model to the wild to predict outputs for new points.

In fact, we hope that will be the case because training is always something that we expect will be done before deployment on our own time, and then the testing becomes important when we deploy the new model to the real world.

However, k-nearest neighbors turns that intuition upside down. Training time is **basically 0**, except perhaps for the cost of storing our training set somewhere. In exchange, testing time is actually quite substantial because for each new point, we need to find the nearest neighbors to it. This literally requires computing the distance from our point to every single point in the training set *every time*. This could take **a long time** if our training set is large.

Some Mechanics

While k-nearest neighbors is a relatively straightforward model, there are still a few details worth discussing with regards to how it is used in practice.

First off, we should discuss what distance metric we use for determining proximity of neighbors. In practice, when we are computing the distance between two points, we tend to use an L_2 (also known as *Euclidean*) distance, although other distance metrics such as L_1 (also known as *Manhattan*) can be used.

Given two points $U = (U_1, U_2, ..., U_n)$ and $V = (V_1, V_2, ..., V_n)$ these distance are defined as follows:

$$L_2(U, V) = \sqrt{(U_1 - V_1)^2 + ... + (U_n - V_n)^2}$$
$$L_1(U, V) = |(U_1 - V_1)| + ... + |(U_n - V_n)|$$

The distance metric chosen often depends on the nature of our data as well as our learning task, but L_2 is an often used default.

Another detail worth discussing is what the optimal choice of k, namely the number of neighbors we use, is. In practice, using a smaller number of neighbors, such as $k = 1$, makes for a model that can may *overfit* because it will be very sensitive to the labels of its closest neighbors.

We will discuss overfitting in greater detail in upcoming sections. As we increase the number of neighbors, we tend to have a smoother, more robust model on new data points. However, the ideal value of k really depends on the problem and the data, and this is typically a quantity we tune as necessary.

Final Thoughts

K-nearest neighbors is a unique model because it requires you to keep track of *all* your training data *all* the time. It is conceptually simple and data-intensive. As a result, it is often used as a baseline model in problems with small to moderate-sized datasets.

Furthermore, while we used a classification example to motivate the algorithm, in practice k-nearest neighbors can be applied to both classification *and* regression.

Test Your Knowledge

KNN Description

Lowering K

Implementing KNN

Controlling Your Model's Bias

Figure 28: Bias-variance meme

In this lesson, we are going to take a deeper dive into some of the theoretical guarantees behind building supervised learning models. We are going to discuss the **bias-variance tradeoff** which is one of the most important principles at the core of machine learning theory.

Besides being important from a theoretical standpoint, the bias-variance tradeoff has very significant implications for the performance of models in practice. Recall that when we are building a supervised model, we typically train on some collection of labelled data.

After the training is done, we really want to evaluate the model on data it never saw during training. The error incurred on *unseen* data tests a model's ability to generalize, and hence it is called the **generalization error**. The generalization error will be an important idea in the remainder of this section

and in our broader machine learning journey.

As we begin our discussion on the bias-variance tradeoff, we will be posing the following questions: how good can the performance of any machine learning model ever get on a problem? In other words, is it possible to reduce the generalization error of a model on unseen data to 0?

Motivations

Before we dive into these questions, let's motivate them by looking at some concrete examples of data. Imagine that we had a training set that looked like Figure 29.

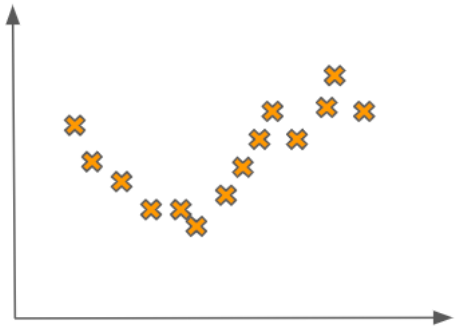

Figure 29: Quadratic data

If we are trying to learn an accurate supervised learning model on this data, it seems that something like a quadratic fit would be pretty good. Fitting a quadratic-like function to this data would look like Figure 30.

This seems reasonable. But why does this seem more reasonable than a linear fit like the one in Figure 31?

One thought we may have is that this linear fit seems to not really pick up on the behavior of the data we have. In other words it seems to be ignoring some of the statistical relationships between the inputs and the outputs.

We may even be so bold as to say that the fit is *overly simplistic*. This simplicity of the model would be especially pronounced if we received a new point in the test set, which could be from the same statistical distribution. This point could look as like Figure 32.

In this case, our linear fit would clearly do a poor job of predicting a value for the new input as compared to the true value of its output as in Figure 33.

In the machine learning lingo, we say that this linear fit is **underfitting** the

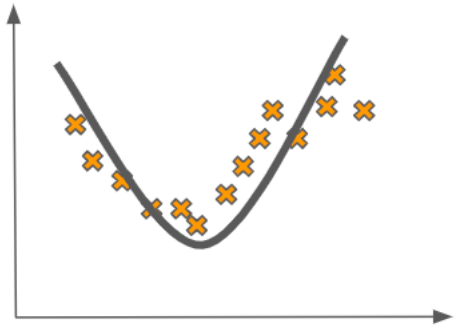

Figure 30: Quadratic data with quadratic fit

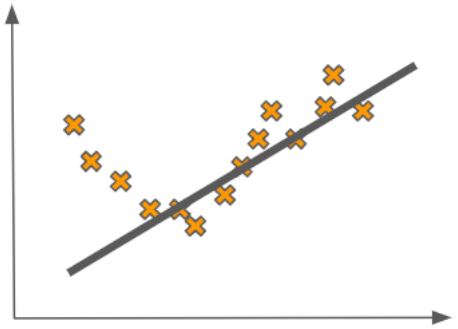

Figure 31: Quadratic data with linear fit

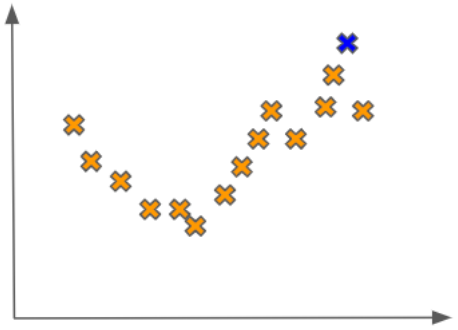

Figure 32: Quadratic data with test point

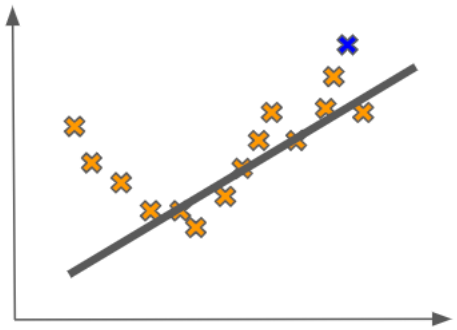

Figure 33: Quadratic data with linear test point

data. As a point of comparison, now imagine that we fit a more complex model to the data, like some higher-order polynomial (Figure 34).

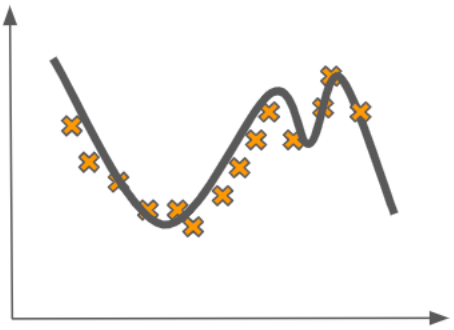

Figure 34: Quadratic data with polynomial

Here, we have the opposite problem. The model is fitting the data **too** well. It is picking up on statistical signal in the data that probably is not there. The data was probably sampled from something like a quadratic function with some noise, but here we are learning a far more complicated model.

We see that this model gives us poor generalization error when we see how far the test point is from the function (Figure 35).

In this case, we say that we are **overfitting** the data.

Compare the last fit's poor generalization to our quadratic fit on the data with the test point, shown in Figure 36.

This is clearly much better!

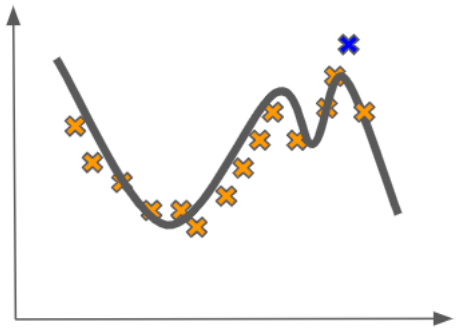

Figure 35: Quadratic data with test point polynomial

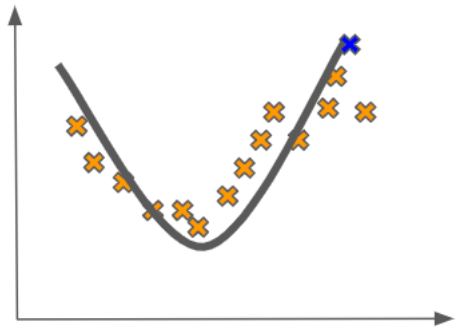

Figure 36: Quadratic data with test point quadratic

Formalizing Model Error

Now we are ready to formalize the notion of bias and variance as they pertain to a model's generalizability. It turns out that in supervised learning there are always three sources of error we have to deal with when we are trying to build the most general model.

One source of error is called the **irreducible error**, which is caused by the inherent noisiness of any dataset. This source of error is entirely out of our control and hence irreducible.

The two sources of error that we *can* control are **bias** and **variance**, and they are basically always competing.

The nature of this somewhat confrontational relationship is that **incurring a decrease in one of these sources of error is always coupled with incurring an increase in the other**. This can actually be demonstrated mathematically, though we will gloss over this derivation for now.

What do these sources of error actually mean? Bias is caused when we make some incorrect assumptions in our model. In this way, it is analogous to human bias.

Variance is caused when our algorithm is really sensitive to minor fluctuations in the training set of our data. In the examples we presented above, the complex polynomial fit is said to have a high variance (and hence a lower bias).

This is because it is really sensitive to the nature of the training data, capturing a lot of its perceived behavior. Too sensitive, in fact, because we know the behavior it is capturing is deceptive. This becomes clear when we are presented with a new test datapoint.

We can also get models with high variance when our model has more features than there are datapoints. In this case, such models tend to be overspecified, with too many sources of signal but not enough data to reinforce the true signal.

Meanwhile, the linear fit in our motivating example is said to have a high bias (and hence a lower variance). It captures very little of the behavior in the training data, making oversimplifying assumptions about the relationship between the features and the output labels. This is evidenced by the fact that the model believes the data was generated by a line when in fact we as omniscient bystanders know it has a quadratic relationship.

Final Thoughts

So what does all this mean for our attempts to get the best model? In practice, this means that there will always be these dual sources of error (bias and variance) that we will have to balance and calibrate.

We can never hope to get perfect generalization, but through empirical analyses we can try to reduce the bias by adding more complexity to our model or reduce

the variance by simplifying some of the assumptions of the model.

We will discuss exact techniques on how to do this in later lessons. The important thing for now is to be aware of the existence of these two sources of error and how they affect our model's generalizability.

Test Your Knowledge

Bias-Variance Explanation

Low Bias High Variance

Why Did You Choose This Model?

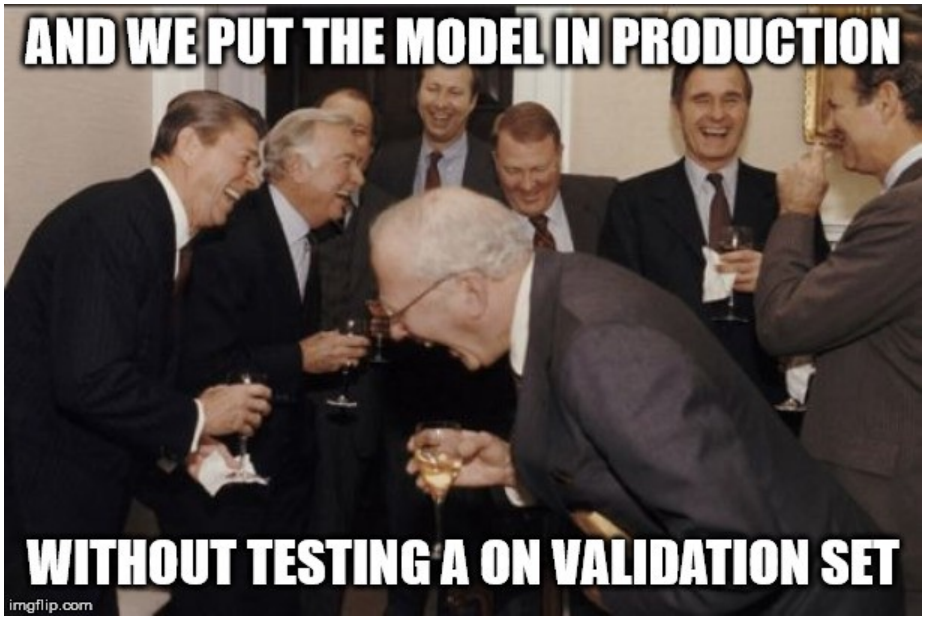

Figure 37: Source Toward Data Science

In this section, we are going to be continuing our discussion of practical machine learning principles, specifically as they pertain to the details of **model selection**. This selection is often done by assessing the generalization error of a model as compared to other models.

Note that in this lesson, we are making *no assumptions* about the particular model class we are dealing with, how many features, etc. In fact, the principles we will discuss could be applied to a collection comprised of diverse model classes such as logistic regression, support vector machines, neural networks, and anything else.

For this lesson, we are more concerned with how we can use the data we have and the models we have built to pick the best one, irrespective of model specifics. So let's get to it!

Motivations

Let's imagine that we have a classification dataset consisting of 100 datapoints and their associated labels. We want to pick the best model from among a support vector machine and a logistic regression to use.

One seemingly reasonable place to start is to take the support vector machine, train it on the entire dataset, and once it is trained see how many errors it makes on the original 100 datapoints it was trained on.

We could then do the same for the logistic regression, and then at the end compare the number of errors each model makes. Perhaps the support vector machine makes 10 incorrect classifications and the logistic regression makes 13, so we pick the support vector machine.

While this may seem like a reasonable way to model select, it is actually **very flawed**. Why?

It turns out when we are building a performant machine learning model, we are really most interested in the model's **generalization** ability. Recall that this refers to the model's error when tested on data it has **never seen** before.

We don't actually care how the model performs on data it has been trained on because that will just encourage selection of the model that *overfits on the training data* the most. As we saw in our discussion on the bias-variance tradeoff, such a model could completely misrepresent the true nature of the data we are analyzing in our problem. So what do we do to combat this issue?

Fixing Model Selection

Imagine that instead of using the full 100 datapoints we have for training our models, we randomly select 75 of them to be our training set and the remaining 25 to be our testing set.

Now, as expected, we only train each of our models using the training set, and then we evaluate each of them **just once** on the testing set, **once we have completed training**. The error that each model incurs on the testing set will then determine which model we pick.

This strategy avoids our problem from before and also gives us a much more reasonable means of assessing model generalization error. In practice, when we use this **held-out-test-set** strategy, we will typically hold out anywhere from $10 - 30\%$ of our data set for testing purposes, shown in Figure 38.

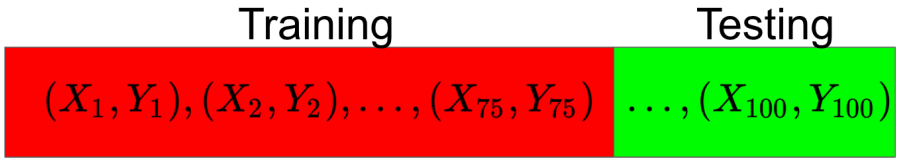

Figure 38: train test machine learning split

Another common strategy that is used is to split up your dataset into three sets: training set, validation set, and testing set.

Using this type of division, we would train on our training set and use the validation set to get some sense for the generalization error of our models. Then finally, once we selected the model that performed best on the validation set, we would evaluate it once on the testing set to get an absolute value for the model's performance.

This is often the strategy that is employed during machine learning competitions, where the training and validation sets of some data is released, and the testing set is not even given to the competitors.

Instead, once a competitor wants to submit their best performing model, they submit it and then the model's score is the error it incurs on the hidden testing set.

Dealing With Data Sparsity

What is the issue with these schemes for evaluating a model? Notice that when we are splitting up our dataset into a training/testing set for example, we are basically deciding that the testing set will not be used at all, except for evaluation at the end. In a sense, we are almost throwing away some of our data which we can't use to train our model.

But there are many domains where data collection is very expensive, such as medical imaging. Imagine if our entire dataset consisted of 10 examples! Could we really afford to train a model using 8 or fewer data points? To do so would negatively impact how good of a model we could train.

In this way, the schemes we have discussed are perfectly valid *when* we have a lot of data to spare. So is model selection doomed without sufficient data? Thankfully there is another very commonly used method called **k-fold cross-validation** that resolves our problem.

The way k-fold cross-validation works is by splitting up our dataset into some number of equally-sized partitions called **folds**. Let's make this concrete by assuming we are using 4 folds. This means we are going to train and evaluate each model 4 times.

During the first iteration, we are going to take the first fold as shown in Figure 39.

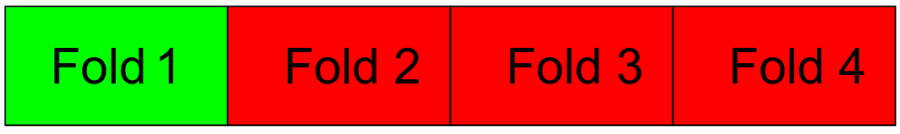

Figure 39: K-fold cross validation fold 1

That first fold will function as our testing set. In other words, we will train on all of the remaining data, **except** that first fold, and then at the end test our

model on that fold. This will give us some error that we will call E_1.

In the second iteration, we take the second fold of our data as shown in Figure 40.

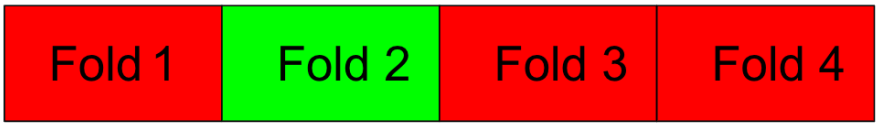

Figure 40: K-fold cross validation fold 2

and again train on all the data *except* that second fold. At the end of training, we then test on that second fold, giving us a new error: E_2. We then repeat this for the remaining folds:

Training on all but the third fold and then testing on the third fold gives Figure 41.

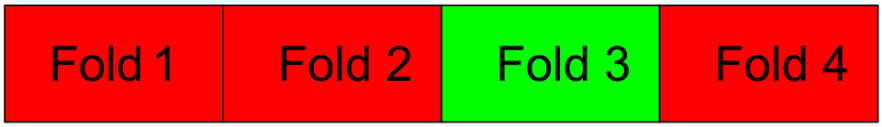

Figure 41: K-fold cross validation fold 3

This produces an error E_3. And finally repeating for the fourth fold (Figure 42)

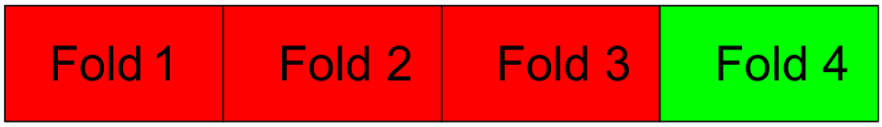

Figure 42: K-fold cross validation fold 4

give us an error E_4.

Once we have tested on all the 4 folds, we compute the final generalization error of our model as the average of E_1, E_2, E_3, and E_4.

One important note is that each fold is an independent run, where we train a new model *from scratch*. This means no model weights are transferred between folds.

In practice, we can use as many folds as we want, depending on the size of our

dataset. In situations of extreme data sparsity, we can even use what is called **leave-one-out cross validation**.

This is a special case of cross-validation, where each fold is literally a single datapoint! So during one iteration of training, we train on all but one datapoint, and then test on that datapoint. Notice that this strategy would use n folds where n is the number of points in our dataset.

However, for most use cases using anywhere from $5-10$ **folds** is often reasonable, and can give us a sufficient sense for the generalization error of a given model.

K-fold cross-validation is a nice solution to our original problem, but it also has its downsides. In particular, when we use k-fold cross-validation we have to actually train and evaluate our system using each of the folds.

Training and testing for each of the folds can be a very computationally expensive operation, which means that model selection can take very long. It then becomes very important to pick a reasonable number of folds (a good number but not too many) when you are performing cross-validaton.

Final Thoughts

As we finish up this lesson, keep in mind that all these model evaluation and selection algorithms are used extensively in practice. It is basically guaranteed that anytime you are building a new model, you will inevitably employ one of these techniques. Which technique you use will depend on your data and problem characteristics, so remember to be flexible to all the options.

Test Your Knowledge

Definition

Cross-validation Techniques

Wrong cross-validation

Cross-validation time series

What Features Do You Want?

Figure 43: Feature selection meme

We are now going to start discussing another very important bit of practical machine learning methodology: **feature selection**.

Feature selection is tightly related to model selection and some might even argue that it is, in fact, a type of model selection. We won't get too hung up on the semantics, and instead jump right into motivating why it's such an important technique.

Motivations

Consider the following scenario: you are dealing with a supervised problem domain where you have come up with a number of features that you deem could be important to building a powerful classifier.

To make the problem concrete, let's say you are trying to build a spam classifier and you've come up with 10,000 features, where each feature denotes whether a given email you are trying to classify contains (or doesn't) one of 10,000 possible words. So some of the features could be **CONTAINS("the")**, **CONTAINS("cat")**, or **CONTAINS("viagra")**.

Let's say further that you have only 500 emails in your entire dataset. What's one **big** problem you may encounter?

Well, in this case, you have far more features than you have data points. In such situations, your model will be overspecified. In other words, you have a very **high chance of overfitting** on your dataset.

In fact, it's quite likely that most of the features you've thought of won't actually help you build a more robust spam classifier. This is because you are adding a feature for each of 10,000 words, and it seems unlikely that a feature like **CONTAINS("the")** will help you determine a spam email.

Feature selection seeks to handle the problem of picking out some number of features that are the *most useful* for actually building your classifier. These are the features that give you the most powerful signal for your problem. So how do we do that?

A First Pass at Selection

In turns out there are quite a few techniques commonly used for feature selection. We will describe a few of them.

The first selection method we will begin with is called **best-subset selection**. This selection method involves trying out *every possible* subset of features from the entire collection.

Therefore if we had three possible features (A, B, C), we would consider the subsets: (A), (B), (C), (A, B), (A, C), (B, C), (A, B, C). For each of these feature subsets, we could run some sort of cross-validation technique to evaluate that feature set (for a review of cross-validation check out the previous lesson).

The feature set which achieved the **lowest generalization error** would be selected for our best-performing model. Here, lowest generalization error refers to the average error across the folds of our cross-validation.

What's one issue that comes up with this technique? Well, notice how even for a very small total feature set (3 in this case), we still had 7 feature subsets to evaluate.

In general, we can mathematically demonstrate that for a feature set of size N, we would have $2^N - 1$ possible feature subsets to try. Techniqually we have 2^N subsets, but we are disregarding the empty subset consisting of no features.

In other words, we have an **exponential number** of subsets to try! Not only is that a **HUGE** number of feature subsets even for a relatively small number, but on top of that, we have to run cross-validation on each feature subset.

This is prohibitively expensive from a computational standpoint. Imagine how intractable this technique would be with our 10,000 feature set for building our spam classifier! So, what do we do?

Improvements in Feature Selection

Notice that our best-subset feature selection technique guarantees finding the optimal feature set, since we are literally trying every possible combination. This optimality comes at a price, namely waiting an impossibly long time for finding the best subset on even small problems.

But what if we don't have to strictly achieve optimality? What if for our purposes, all we need is *good enough*? **Forward-stepwise selection** helps us do just that: get good enough (Yay, C's get degrees).

How does it work? Forward step-wise selection basically functions as follows: Assume we have k features in total to choose from. We will order these features as f_1, f_2, ..., f_k, and let S denote our best performing feature set so far. We execute the following procedure:

1) Initially start with S being the empty set (so no features).

2) Repeat the following process: for each of the k features we can choose from, let S_k be the feature set with feature f_k added to S. Run cross-validation using S_k. At the end assign S to the best performing S_k among all the S_k created during the loop.

3) Run the loop in 2) until our set contains a number of features equal to some threshold we choose.

Hence, at the beginning S is empty. Let's say we have three potential features f_1, f_2, and f_3. We will run cross-validation three times, adding a single feature to S each time. Hence we will have one run with the feature set $\{f_1\}$, one with $\{f_2\}$, and one with $\{f_3\}$.

The feature set with the best generalization error will be used for S. We then repeat this process, greedily adding a single feature to our new S each time, based on best generalization error.

And that's it! Notice that for this particular algorithm we can only run cross-validation up to order k^2 times (can you see why?) This is clearly **MUCH** better than running cross-validation an exponential number of times, which is what we get with best-subset selection.

There is another related feature selection technique called **backward-stepwise selection** where S is initialized to the set of all features, and then during each iteration of 2) above, we remove a feature instead of add a feature. The remainder of the algorithm is the same as forward-stepwise selecion.

Final Thoughts

Besides the aforementioned techniques, there are a number of more sophisticated feature selection approaches. These include simmulated annealing and genetic algorithms, just to name a few. For now, we will leave discussion of those techniques to another time! Regardless, feature selection is an important

technique to add to the toolbox, as it enables us to be more careful in how we build out our models.

Test Your Knowledge

Feature Extraction

Best Subset Features

Feature Selection Examples

Adding Features Example

Model Regularization

Figure 44: Regularization meme

In this section, we will continue with our odyssey through practical machine learning methodology by discussing **model regularization**. Regularization is an immensely important principle in machine learning and one of the most powerful ones in the practitioner's toolkit. Excited? Let's get started!

Regularization is another way to address the ever-present problem of model generalization. It is a technique we apply to deal with model overfitting, particularly when a model is overspecified for the problem we are tackling.

We have actually already seen regularization previously. Recall that when we were studying support vector machines, we introduced slack variables to make the model less susceptible to outliers and also make it so that the model could handle non-linearly separable data.

These slack variables represented a means of regularizing the model, through what we will show later is called L_1 **regularization**.

There are many types of regularization, some that can be applied to a broad range of models, and others that we will see are a bit more model-class specific

(such as dropout for neural networks).

L_2 Regularization

Traditionally when we are building a supervised model, we have some number of features we extract from our data. During training, we learn weights that dictate how important each feature is for our model.

These weights tune the strength of the features through interactions ranging from the simple linear ones (as in the case of linear regression) to more complex interactions like those we we will see with neural networks.

For the time being, let's assume that we are dealing with linear regression. Therefore, we have a weight vector $A = (A_1, A_2, ..., A_k)$ for our k features. L_2 **regularization** is the first type of regularization we will formally investigate, and it involves adding the square of the weights to our cost function.

What that looks like in mathematics is as follows: Recall that for linear regression we were trying to minimize the value of the least-squares cost:

$$C(X) = \frac{1}{2} \cdot \sum_{i=1}^{n} (F(X_i) - Y_i)^2$$

Adding an

$$L_2$$

penalty, modifies this cost function to the following:

$$C(X) = \frac{1}{2} \cdot \sum_{i=1}^{n} (F(X_i) - Y_i)^2 + L \cdot \sum_{i=1}^{k} A_i^2$$

So, this cost function involves optimizing this more complex sum of terms. Notice that now our model must ensure that the squared magnitude of its weights don't get too big, as that would lead to a larger overall value of our cost.

In practice, having smaller weight magnitudes serves the purpose of ensuring that any single feature is not weighted **too** heavily, effectively smoothing out our model fit. This is **exactly** what we want to do to prevent overfitting.

You may have noticed that we also have this extra term L that we multiply through in our L_2 penalty. L as you may remember is called a *hyperparameter* and is something that is typically tuned (i.e. a good value is chosen) during cross-validation or model training.

We can do some simple analysis to understand how L affects our cost. If L is really, really small (as in close to 0), it's as if we are not at all applying an L_2 penalty and our cost function degenerates to the original cost function we were optimizing before.

However, if L is really, really big, then our cost will focus solely on minimizing the value of our L_2 penalty. In practice, this amounts to sending all of our weights toward 0. Our model basically ends up learning nothing!

This makes sense because if we focus very hard on counteracting the effects of overfitting, we may effectively end up underfitting. In practice, there is a sweet spot for the L parameter which depends on our data and problem.

A quick note on terminology: you may also sometimes see L_2 regularization referred to as **ridge regression**, though for our purposes we will continue to call it L_2 regularization. While we focused on linear regression to introduce L_2 regularization, practically speaking this technique can be applied to many other model classes.

L_1 Regularization

We can now move on to discussing L_1 **regularization**. This technique is conceptually similar to L_2 regularization, except instead of adding the term

$$L \cdot \sum_{i=1}^{k} A_i^2$$

to our cost, we add the term

$$L \cdot \sum_{i=1}^{k} |A_i|$$

That's it! As mentioned previously, we've already seen L_1 regularization in our slack variables in the support vector machine cost. Notice how with our L_1 regularization term, we can use the same logic for tuning the L parameter as with L_2 regularization.

While L_1 regularization seems pretty similar mathematically, it has quite different implications for feature selection. It turns out that one of the consequences of using L_1 regularization is that many weights go to 0 or get really close to 0.

In that sense, L_1 regularization induces **stricter sparsity in our feature set**. This effectively means that many of the features aren't counted at all in our model. This makes it more like a traditional **feature selection** algorithm, as compared to L_2 regularization that achieves a smoother continuous set of weights for our feature set.

Final Thoughts

In addition to these regularization techniques, there are **many** more ways to regularize a model out there, which we won't cover. In practice, the type of

regularization you use very often depends on how you want to control your feature set. But regardless, it is a hugely important technique to keep under your belt as you venture into the machine learning jungle!

Test Your Knowledge

Reasons for Regularization

L1/L2 Differences

L1 over L2

Join the Ensemble

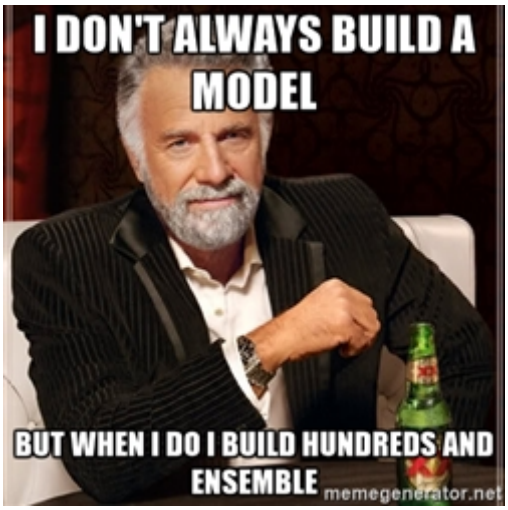

Figure 45: Source Analytics Vidhya

As its name suggests, the core idea of **ensembling** is about combining a collection of models to get a more performant model. This is analogous to the idea of combining individual musical instruments to get an orchestral ensemble of harmonious sounds. This lesson will be about how we can achieve that harmonious sound in machine learning .

Ensembling is an extremely powerful technique and is often a surefire way to squeeze out a few percentage points of performance on any task you tackle.

For example, the winning entries of the Netflix challenge in 2007 were all sophisticated ensembled systems. Ensembling often can either help us get a more performant model or help address issues of overfitting by reducing our model variance.

Put the Model in the Bag

The first technique for ensembling we will study is called **bagging** or **b**ootstrap **agg**regat**ing**. How does it actually work?

Say we have a dataset containing **N** datapoints. Traditionally, we would train a single model on this dataset, using some type of dataset splitting or cross-validation to assess the model's generalization error.

With bagging, we take our dataset and generate **K** bootstrapped smaller

datasets by randomly sampling some number **M** of datapoints with replacement from the original dataset. Phew that was a mouthful!

Let's take a concrete example to illustrate our point. Assume we have a dataset consisting of $N = 6$ points and we want to generate $K = 3$ smaller datasets with $M = 4$ points each. That would look as shown in Figure 46.

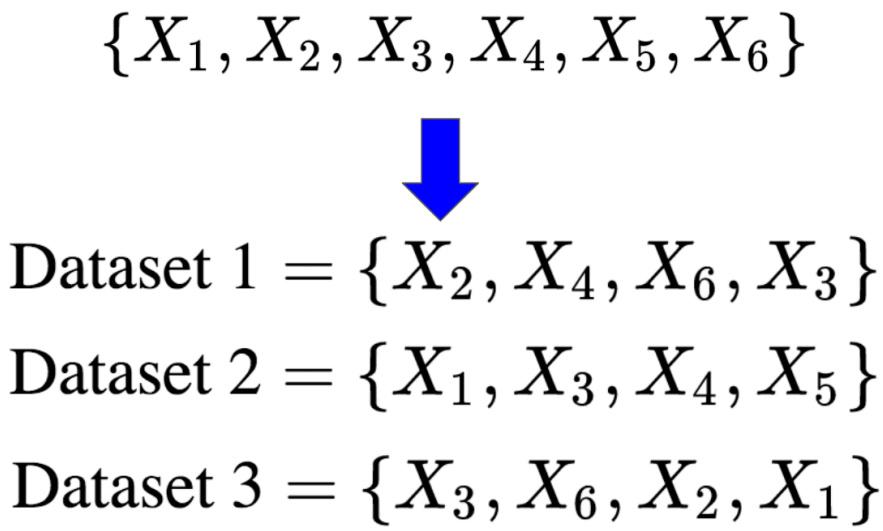

Figure 46: Bootstraped datasets

Note that because we are randomly sampling *with replacement*, points may be repeated in the smaller datasets. These smaller datasets are called **bootstrapped datasets**.

Now we train a separate model per bootstrapped dataset. When we are done we will have a mega model. When we want to label a new input, we run it through each of the models trained on the bootstrapped datasets, called **bootstrapped models**. We then average their outputs.

In the case of classification, we can simply take the majority label output by the bootstrapped models.

In the case of regression, we can numerically average the bootstrapped models' outputs. An ensemble of three bootstrapped classification models with their predictions being aggregated would look as shown in Figure 47.

For some high-level intuition about bagging, consider that by having each bootstrapped model learn using datapoints that are a subset of the total dataset, we allow for each model to learn some statistical regularities without overemphasizing any particular behavior.

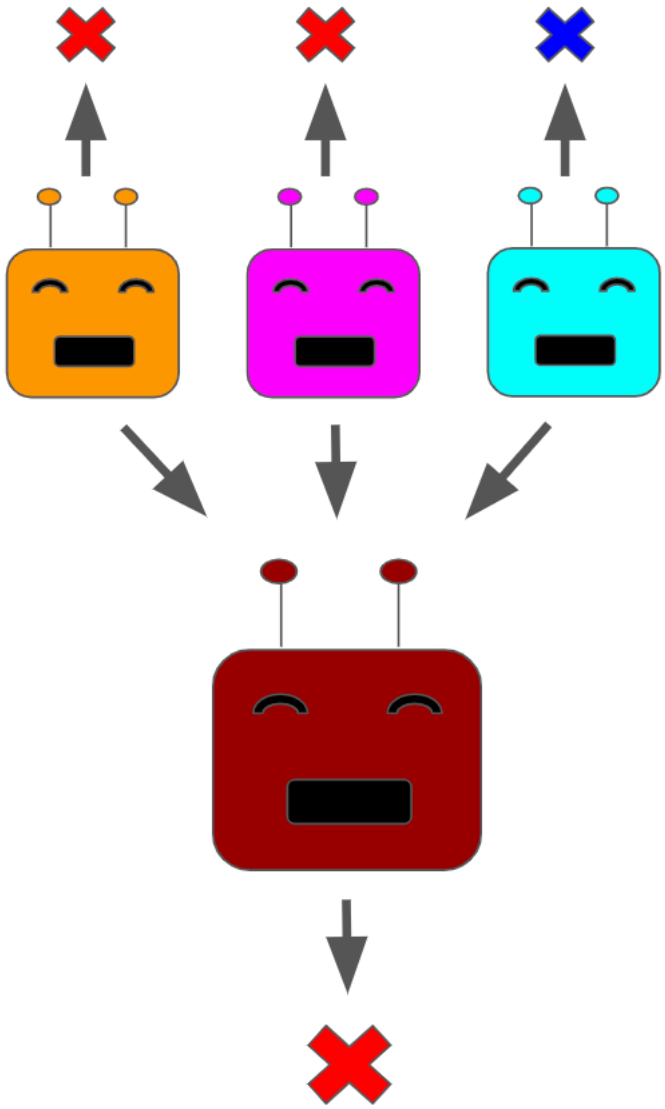

Figure 47: Ensembled model aggregate

In other words, by having many small models we cast a wider net in capturing the full dataset behavior. Bagging is often employed with decision trees, though it can be applied to any model class.

Getting a Boost

Boosting is a very different flavor of ensembling that is also extremely powerful. Many different boosting algorithms have been developed, though here we will focus on one of the more canonical techniques: **adaboost**. As with bagging, boosting involves training a collection of models on a dataset, though the exact procedure is very different.

Let's describe the procedure through a concrete example. Assume we have 4 points in our training dataset, and we are building a model for a binary classification task.

We start off by associating a numerical weight with each of the datapoints and use the weighted dataset to train a model. Let's call this trained model M_1. Note the weights begin with the same value, which we will just set to 1 for now. This looks like Figure 48.

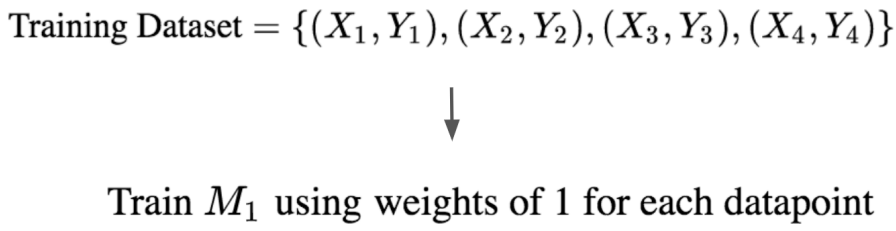

Figure 48: Model boosting Step 1

Next we compute how many of the datapoints our model miscalculated. This miscalculation error is used to associate a weight for the entire model M_1, which we will call W_1.

Furthermore, we now update all the weights on the dataset, upweighting those which were mispredicted and downweighting the others. This looks like Figure 49.

We now use this newly weighted dataset to train a fresh model M_2. This is then used to compute a new weight for M_2, which we will call W_2.

We continue this procedure of reweighting the dataset, training a new model, and then using the prediction error to associate a model weight.

Let's say we run this for 5 iterations. At the end, when we are predicting for a new input X, we take the predictions of our models M_1, ..., M_5 and form a majority vote as shown in Figure 50.

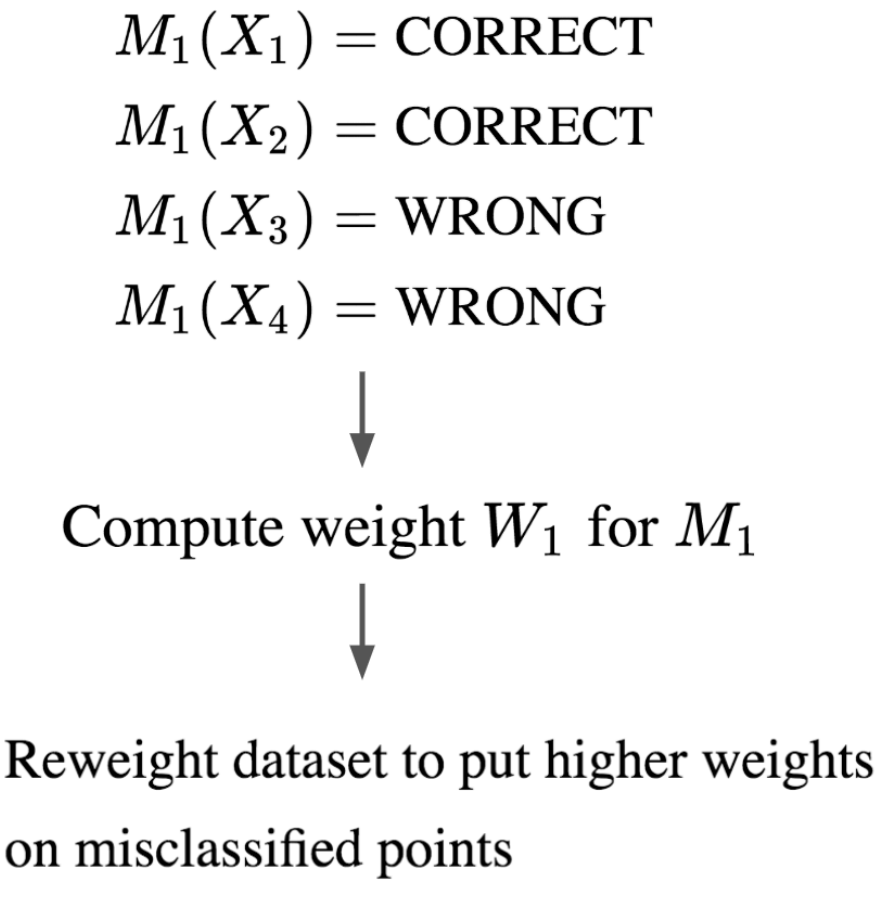

Figure 49: Model boosting Step 2

$$M_{\text{ensemble}}(X) = \sum_{i=1}^{5} W_i \cdot M_i(X)$$

Figure 50: Model boosting Step 3

In practice, the weights determined for the models tend to favor those models which were more accurate classifiers on the dataset. When we are building our initial "naive" models in the early stages of boosting, we are perfectly okay with training underperformant models.

In fact, that is what makes boosting such a beautiful and elegant technique: it demonstrates that you can take a collection of weak models and combine them to form a strong model.

Additionally, because we are training multiple models on effectively diverse datasets, we tend to reduce overfitting in our final boosted model.

Random Forests

We will now very briefly spend some time discussing a concrete ensembled model that is one of the best go-to models to employ for new tasks. This technique is called **random forests**, and it is an ensembling strategy for decision trees.

Random forests are often used to address overfitting that can happen in trees. It does this by performing a type of **feature bagging**. What this means is that during the procedure, we train individual trees on bootstrapped subsets of the data as in traditional bagging.

However, the trees that are created only use a random subset of the total feature set when they are being built. Recall that this is different from how trees are traditionally built, where during building we consider the full set of features for each split. Using a subset of the features at each split has the effect of even more strictly decorrelating the trees across the smaller datasets.

At the end, the new trees are combined as in normal bagging to form our mega model. Random forest are very nice models because they get the expressive power of decision trees but combat the high variance that trees are susceptible to.

Final Thoughts

We will end this lesson with a few additional notes. While ensembling is very powerful, it can be a very costly operation. This is because oftentimes in ensembling techniques we must train many models on a given dataset.

Because of this ensembling is often only used when you are trying to squeeze out a bit more performance on a certain problem. That being said, it is a fantastic technique for reducing overfitting in models.

Test Your Knowledge

Ensembling Uses

Model Evaluation

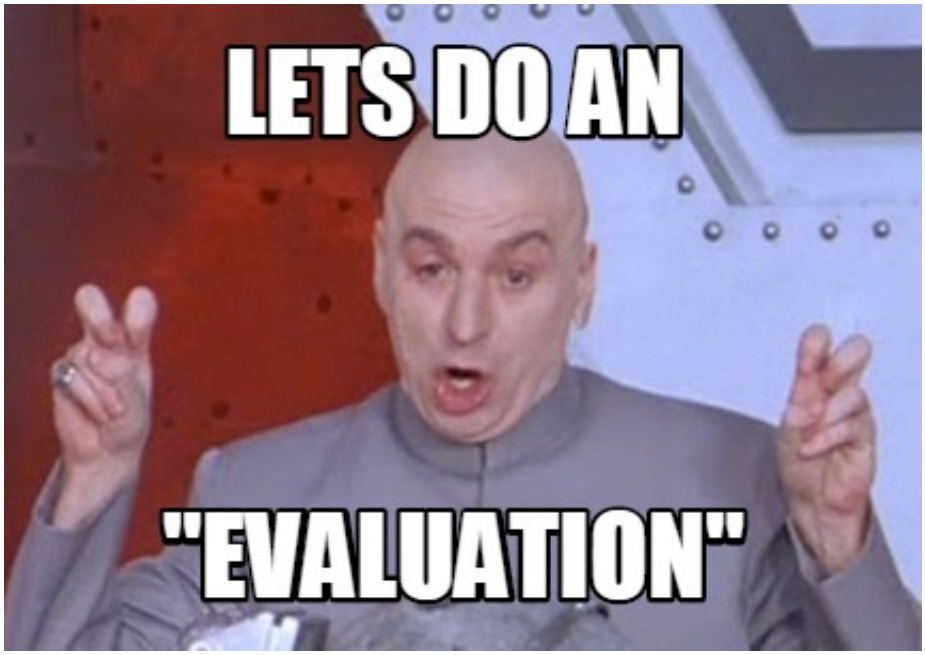

Figure 51: Evaluation meme

Thus far in our study of various machine learning models, we have often alluded to evaluating our models. Evaluation has been especially relevant when training and performing cross-validation.

Evaluation is also very important when we have successfully trained a model and are ready to say "Based on such-and-such measure, this is the quality of this model."

Metrics in other situations usually refer to evaluating some measure of success that allows us to quantitatively compare one value against another.

For example when we are talking about stock analysis, we may refer to metrics such as the price-to-earnings ratio of a company's stock, or we may refer to a company's market capitalization.

What do such metrics look like for machine learning models? It turns out there are a few commonly-used evaluation metrics both for classification and regression. This lesson will introduce a few of these metrics and describe what they are meant to capture.

Accuracy

Accuracy is one of the most fundamental metrics used in classification. In its most basic terms, accuracy computes the ratio of the number of correct predictions to the total number of predictions:

$$\text{Accuracy} = \frac{\#\text{Correct Predictions}}{\text{Total \# Predictions}}$$

For example, if we are doing a binary classification predicting the likelihood that a car is cheap or expensive, we may predict for a dataset of five cars **(CHEAP, CHEAP, EXPENSIVE, CHEAP, EXPENSIVE)**.

If in reality the cars are **(CHEAP, CHEAP, CHEAP, EXPENSIVE, EXPENSIVE)**, then our accuracy is $\frac{3}{5} = 60\%$. In this case our model is not doing a great job of solving the task.

Generally, accuracy is a good place to start with classification tasks. Sometimes, however, it is not a sufficient metric. When is that the case? To answer this question, we have to introduce a little more terminology.

In classification, a **true positive** is a positive label that our model predicts for a datapoint whose true label is also positive. For our running example, we can denote a **CHEAP** prediction as a positive one, so our model had 2 true positives.

A **true negative** is when our model accurately makes a negative prediction. In our running example, there are 2 **EXPENSIVE** cars of which our model labels 1 correctly, so the number of true negatives is 1.

A **false positive** is when our model predicts a positive label but the true label is negative. In the example, our model predicts **CHEAP** a single time when in fact the label was **EXPENSIVE**, so we have 1 false positive.

Similarly, a **false negative** is when our model predicts a negative label but the true label was positive. In our example, our model predicts **EXPENSIVE** once when in fact the label was **CHEAP**, so the number of false negatives is 1.

With these definitions in place, we can actually rewrite our definition of accuracy. Letting **TP = True Positive**, **TN = True Negative**, **FN = False Negative**, and **FP = False Positive** we have:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Given this definition, we see that our accuracy is $\frac{2+1}{2+1+1+1} = 60\%$ as we had before.

So why introduce all this terminology? All these new terms will help us to understand when accuracy is lacking as a metric.

Consider the example of car classification from several lessons ago, and imagine that we are classifying a dataset of 100 cars. Let's say that our model has 2 true positives, 1 false positive, 7 false negatives, and 90 true negatives.

In this case, our model's accuracy would be $\frac{2+90}{2+90+7+1} = 92\%$. That sounds great, right? Well, maybe at first.

But let's think about the label distribution of our car dataset. It turns out we have $90 + 1 = 91$ **EXPENSIVE** cars but only $2 + 7 = 9$ **CHEAP** cars. Hence our model identified 90 of the 91 **EXPENSIVE** cars as **EXPENSIVE**, but only 2 of the 9 **CHEAP** cars as **CHEAP**.

This is clearly a problem, where our accuracy metric is giving us an incorrect signal about the quality of our model.

It turns out that when we have a large disparity in our label distribution (91 **EXPENSIVE** cars vs. 9 **CHEAP** cars), accuracy is not a great metric to use. In fact if our system had literally only predicted **EXPENSIVE** for every new car it received as an input, it would still have had a **91% accuracy**.

But clearly not all cars are **EXPENSIVE**. It turns out we need a more fine-grained metric to deal with this issue.

F_1 Score

The F_1 **score** does a better job of handling the label-disparity issue we encountered with accuracy. It does this by leveraging two measures: **precision** and **recall**. These two quantities are defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

For our 100-car dataset above, the precision of our model would be $\frac{2}{2+1} = 66.6\%$, while the recall would be $\frac{2}{2+7} = 22.2\%$. Ouch!

The F_1 score is then defined as:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Our current model would receive an F_1 score of 33.3%. All of a sudden our model seems **MUCH** worse. That's a good thing because the model learned to completely disregard predicting one entire label in our dataset. This is really bad. Hence the F_1 score penalized it substantially.

In practice, when it comes to classification tasks, the F_1 score is more often used as a metric because it gives a more balanced measure of a model's performance.

Mean Absolute Error

Let's shift gears a bit to discuss a commonly-used metric for regression tasks: **mean absolute error**. If on a 3 point dataset, we have a model outputting the values Y_1, Y_2, Y_3 and the true values are G_1, G_2, G_3, then the mean absolute error is defined as:

$$\text{Mean Absolute Error} = \frac{\sum_{i=1}^{3} |y_i - g_i|}{3}$$

in other words, the average of the absolute errors. More concretely, if our model outputted $(0.1, -1.3, 0.8)$ and the true values are $(-0.4, -0.3, 1)$, then the mean absolute error would be:

$$\begin{aligned} \text{MAE} &= \frac{|0 - 1 - (-0.4)| + |-1.3 - (-0.3)| + |0.8 - 1|}{3} \\ &\approx 0.767 \end{aligned}$$

While relatively straightforward, mean absolute error is a standard way of assessing the quality of regression models.

Final Thoughts

The collection of metrics we discussed is only meant to provide a taste of the ways we can evaluate models. There are many, many more means of performing evaluation that are used by the scientific community.

While we introduced all these different metrics, we never discussed what a *good* score for a metric is. It turns out there is no one golden number for any metric. The score you should be aiming for is, of course, as close to perfect as possible. However, how reasonable perfection is depends mainly on your data, how complex it is, how learnable your problem is, etc.

Another important thing to remember is never to put all your eggs in one basket when evaluating a model and assume that excellent performance on a single metric definitely demonstrates the superiority of a model.

We'll end this lesson with a relevant note of caution from economist Charles Goodhart: *When a measure becomes a target, it ceases to be a good measure.*

Test Your Knowledge

Precision Example

F1 Definition

Calculate F1

Binary Confusion Matrix

Market Basket Analysis

Figure 52: Busy supermarket

In this section, we will be exploring another example of an unsupervised learning algorithm: **market basket analysis**. Market basket analysis deals with the problem of analyzing the relationship between sets of items and how often they appear in certain baskets.

To make this description more concrete, one common use case for market basket analysis is literally storing items in shopping baskets! Here we are concerned with studying customer's purchasing patterns.

We are particularly interested in rules for associating the presence of certain items in a basket with the appearance of another item. For example, it might be useful for store managers to know that when customers buy chips and salsa, they also tend to buy guacamole. This could inform how items are laid out in a store or allow for more targeted item recommendations.

A Flurry of Terminology

Now, how do we properly formalize these intuitions? Our study of market basket analysis will require introducing quite a bit of new terminology. The ideas themselves aren't super complicated, but it might be a bit tricky to keep all the new terms straight. Don't worry! Just refer back to the definitions as necessary.

To make our analysis concrete, let's assume we are analyzing an admittedly small sample of data from a grocery store, where we have 4 customers' baskets, containing the following items:

1) (cheese, pasta, carrots, potatoes, broccoli, chicken)
2) (chicken, pasta, tomatoes, carrots)
3) (chicken, broccoli, pasta, cheese)
4) (chocolate, water, carrots, sugar)

We define the **support** of an item (or a collection of items) to be the number of baskets in which it appears. For example, the support of (*chicken*) is 3, since

it appears in 3 baskets. Similarly, the support of $(chicken, pasta)$ is 3. A high support means an item or set of items appear in a large number of baskets.

We can specify a numerical **support threshold**, and define a **frequent** itemset to be one whose support is equal to or greater than our support threshold.

This is important because we will want to do an analysis on itemsets that actually appear in a reasonably large number of baskets, so that we can extract meaningful insights. What good is an analysis around something like *beets*, which probably only one customer bought all year? (Fine, I'm sure *at least* two people bought beets).

Our goal will then be to extract those frequent itemsets in our baskets. To do that, we define an **association rule** which is written as $M \to n$. Here M is a certain itemset, and n is a single item.

Therefore, an example of an association rule would be $(chicken, pasta) \to (cheese)$. These rules allow us to compute some likelihood that the item n is found in conjunction with those items in M.

To help quantify this notion, we also define the **confidence** of a rule as the ratio of the support of (M, n) to the support of M.

So for our running example, the confidence of $(chicken, pasta) \to (cheese)$ is $\frac{2}{3}$, since $(chicken, pasta, cheese)$ appear in 2 baskets whereas $(chicken, pasta)$ appear in 3.

For some qualitative analysis of this metric, note that a particularly small confidence (something close to 0) implies a very unlikely association rule. An example of that would be $(broccoli) \to (chocolate)$. The highest confidence we could have is 1, which would happen if item n appeared in every basket where M appeared.

We can make the relationship between M and n even more powerful from a causal standpoint by introducing the concept of **interest**.

Interest is defined as the confidence of the association rule minus the fraction of baskets in which n appears. In our running example, the interest of $(chicken, pasta) \to (cheese)$ would be $\frac{2}{3} - \frac{1}{2} = \frac{1}{6}$.

Notice that we want our interest to be as high as possible because that would suggest that having $(chicken, pasta)$ in your basket more directly implies you will have $(cheese)$ as well.

An interest of 0 would more closely suggest that every basket that has $(cheese)$ also happens to have $(chicken, pasta)$, which is not as meaningful from a causal standpoint.

Another *interesting* case is where we have a negative interest. A negative interest implies that the presence of $(chicken, pasta)$ discourages the presence of $(cheese)$.

Putting It All Together

With all this terminology in place, we can now state that the core of useful market basket analysis boils down to finding association rules that hold with high confidence on itemsets with high support. Wow that's a mouthful!

This leads us to the most sophisticated algorithm we will see in our studies: counting! Jokes aside, it turns out that finding association rules that have our desired properties effectively requires doing methodical counting of all possible combinations of itemsets over the set of all baskets.

The devil is in the details, however. Often the complexity in market basket analysis is not so much about any sophisticated math, as much as it is about dealing with analysis at scale.

Think about how many customer baskets there are in an average grocery store. It could be hundreds of thousands in a month, and a basket could contain anywhere from a few to upwards of 50 items.

That means we have **A LOT** of data to process and store. Much of the algorithmic complexity in market basket analysis comes in algorithms to do this processing efficiently. We will defer discussion of these clever algorithms to another time!

Final Thoughts

Before we finish this lesson, we will end with a few high-level notes about market basket analysis. It turns out this analysis technique is extremely useful and can be applied to a number of different problems.

One example includes doing large-scale processing of word documents where we are trying to extract words that can be lumped into concepts. In this case, our items are the words and the baskets are the documents. Think about what kinds of association rules could be useful there.

Market basket analysis can also be applied to fraud detection where we can analyze the spending patterns of users. Here users' credit card purchases over a given time period could be baskets, and the things they purchase could be items.

There are many other uses, and the most amazing thing is that we are deriving very meaningful analyses completely in an unsupervised setting. No ground truth but still a lot of useful deductions. Neat!

Test Your Knowledge

Frequent Itemset Example

Confidence Example

Support Example

Association Rule Example

What K-Means

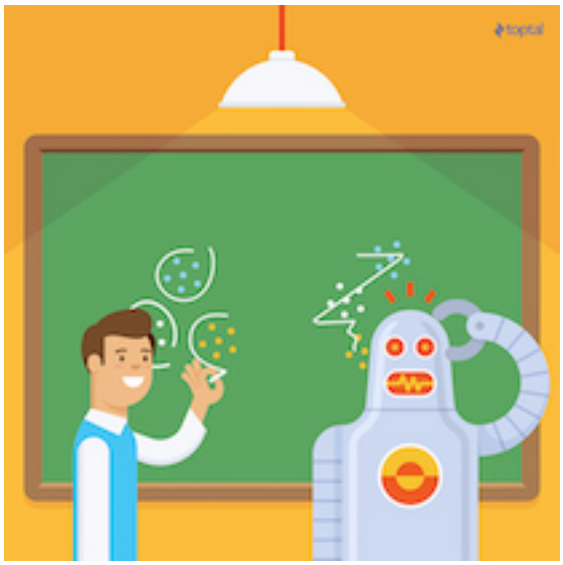

Figure 53: Source WhatMatrix

In this section, we are going to continue our study of unsupervised learning by taking our first look at data clustering. To do so, we will describe **k-means clustering**, which is one of the most popular clustering algorithms used by practitioners.

At a high-level, k-means clustering seeks to find a way to clump our dataset into some predefined number of clusters using a well-defined notion of similarity among the datapoints. In this case, the notion of similarity usually amounts to minimizing intra-cluster variation of the points across all the clusters. Let's investigate what this means in further detail.

The Algorithm

Let's say that we have n datapoints in our dataset: $(X_1, X_2, ..., X_n)$. Here we assume that each of the datapoints is already featurized and represented in its mathematical vector notation.

We begin by selecting some number, k, of what are called **cluster centroids**. Let's denote these centroids $(C_1, C_2, ..., C_k)$. Note that the centroids must have the same dimension as the points in our dataset. In other words, if each of our datapoints is a vector of dimension 5, then our centroid will also be a vector of dimension 5.

The k-means algorithm then runs the following steps repeatedly:

1) For every datapoint, find the centroid to which it is closest, according to a Euclidean distance metric.

2) Once we have assigned each datapoint to a centroid we recompute each centroid C_j by averaging the datapoints that have been assigned to C_j.

Let's give a bit more detail about these steps. For 1) if we start with X_1, we compute:

$$(L_2(X_1, C_j))^2 \text{ for every } C_j \text{ in } (C_1, C_2, ..., C_k)$$

Here $L_2(X_1, C_j)$ denotes the Euclidean (also known as L_2) distance we described in the lesson on regularization. After performing this calculation, we assign X_1 to the C_j which produced the minimum value for this squared Euclidean distance. We repeat that procedure for every single datapoint.

The calculation in step 2) has the effect of adjusting the centroid to more accurately reflect its constituent datapoints. In this way, the two steps constantly switch between assigning points to centroids and then adjusting the centroids.

That's all there is to it! We repeat these two steps until our algorithm converges. What does convergence mean here?

Convergence occurs when no datapoints get assigned to a new cluster during an execution of 1). In other words, the centroids have *stabilized*.

Below we include a visualization of the algorithm executing for some number of iterations on a dataset. Let's say we have some data and we are trying to cluster it into three clusters.

We start with all our points unassigned to any centroids. The centroids are indicated in red, blue, green in Figure 54.

Now we assign the points to their nearest centroid (Figure 55).

We then adjust the centroids accordingly (Figure 56)

We then repeat these steps until our centroids stabilize, at which point we are done. We should see three well-defined clusters as in Figure 57.

K-means clustering is elegant because one can demonstrate mathematically that the algorithm is guaranteed to converge! Now you may be thinking that something seems missing.

We seem to have avoided one huge detail: how do we actually pick the initial cluster centroids for the algorithm? In particular, though k-means is guaranteed to converge, how do we know it is converging to the absolute optimal clustering?

The short answer is we don't. It turns out that centroid selection can pretty drastically impact the clustering produced by the algorithm. One example of a

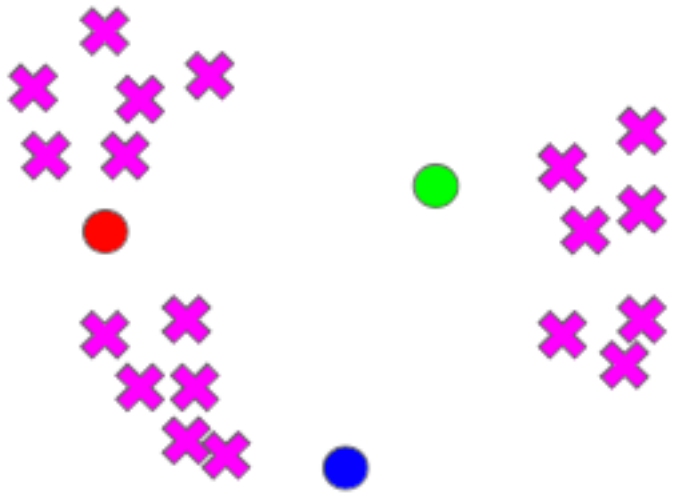

Figure 54: Unclustered data

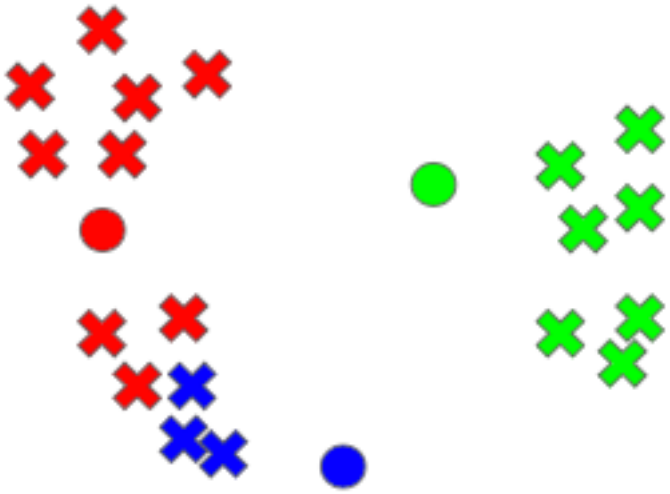

Figure 55: Nearest centroid assignment

Figure 56: Adjust centroid

Figure 57: Complete clustering

valid centroid selection algorithm that is used in practice is just to pick random centroids. Another one is to pick **k** points in our dataset that are spread far away from each other.

Because different centroid selection can impact the clustering, it is common to run k-means several times on a dataset using different selected centroids and picking the best clustering among them.

Final Thoughts

While k-means is not super complicated, it is a very useful method for unsupervised learning. As one simple use case, imagine we were given a collection of unidentified essays.

We could represent each document as a vector, based on the frequency of words appearing in the essay. We could then apply k-means clustering to find which essays are most similar, and those could perhaps give us some indication of how many authors' writing is represented in the collection.

Perhaps we could even extract the centroids and compare them to essays whose authors we know and that could help us actually identify the unknown essays in our dataset!

This is just a small example of how useful k-means can be in practice. There are many more scenarios where it can be *extremely useful* for analyzing datasets where we don't have labels.

Test Your Knowledge

K-means vs. KNN

Centroid Initialization

Principal Components Analysis

Figure 58: PCA meme

This section will introduce us to **principal components analysis**, which is going to be our first example of a data **dimensionality reduction** technique. While that might sound really fancy, at its core dimensionality reduction is a fairly intuitive idea.

Motivations

To motivate it, let's imagine we have a featurized dataset consisting of the points $(X_1, X_2, ..., X_n)$. So far in past sections, we have never spent too much time talking about the dimensionality of our datapoints, or in other words, the number of features per datapoint. We have largely imagined that our data consisted of a handful of manually-chosen features, maybe on the order of 5-20 or so.

But for many problems, our feature sets span upwards of *thousands of features*! As an example, consider a problem where we are analyzing some collection of text documents.

For such a problem, it is perfectly reasonable to featurize each document by the number of occurrences of each unique word. In other words, if **the** appears 3 times in the document and **cat** appears 5 times, we would have two separate features with values 3 and 5 respectively. Notice how in this case, our feature set could conceivably be equal in size to the number of words in the English

dictionary, which is **over 100,000 words!**

Once we are dealing with high-dimensional data, we face a number of new challenges related to the computational cost of training a model or memory costs related to storing data on disk. Another question that naturally comes up is: do we have any redundancy in our features?

We derive features from data because we believe that those features contain information that will be useful for learning the outputs of our models. But if we have multiple features that encode similar information, then those extra features don't help us learn a better model and may in fact just incur additional noise.

As an example, imagine that we are training a model to determine whether a given car is cheap or expensive, and we have one feature that is the size of the trunk in cubic feet and another feature that is the number of shopping bags that can fit in the trunk.

While these features aren't exactly the same, they are still **very related** and it certainly seems like we wouldn't have a less powerful model if we removed one of these features.

Welcoming PCA

Dimensionality reduction is a technique by which we seek to reduce each datapoint to a smaller feature set, where we preserve the information in each datapoint. For a little more visual intuition about what dimensionality reduction could look like, imagine a dataset with two primary features (and their associated axes) that looks as shown in Figure 59.

Notice how in this example, it seems that we really only need a single axis (rather than two) to represent our entire dataset, namely Z_1 as in Figure 60.

This means that we could project each datapoint on this new axis Z_1. Notice how except for a little bit of *wiggle* along the Z_2 direction that we don't capture, our dataset could be represented in terms of this new single *pseudofeature*. Now the big qustion is: how do we pick this Z_1 axis to use for representing our dataset?

And here we finally get to **principal components analysis** (or PCA for short). PCA is an algorithm whereby we find these axes along which the feature of a dataset can be more concisely represented. These axes are called the **principal components**.

How does the algorithm find these principal components?

Well, in order to gain some intuition for the algorithm, let's revisit our running example of the dataset above. While we deduced that Z_1 above was a good axis along which to project our data so as to reduce the dimensionality, imagine if we had picked Z_2 instead.

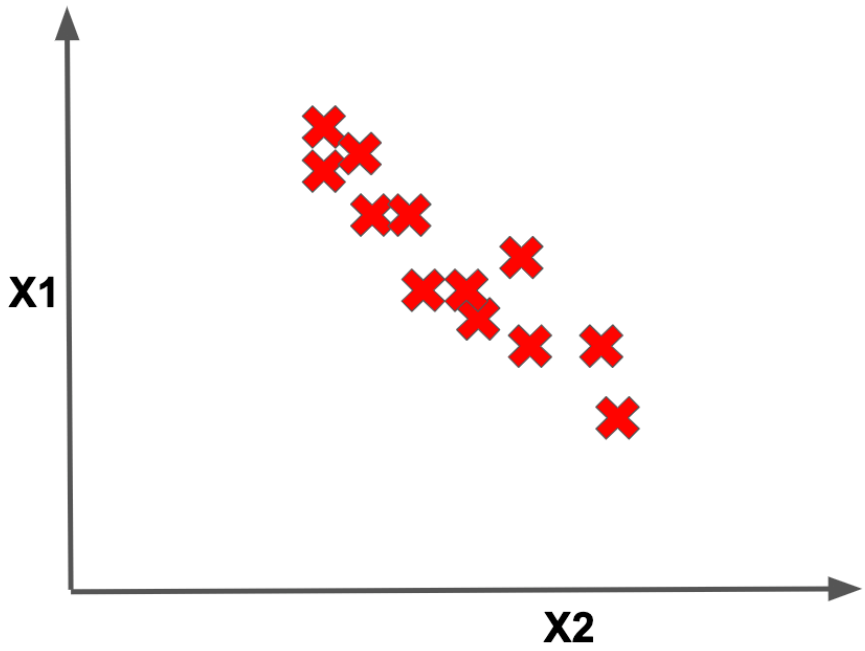

Figure 59: Sample dataset

It certainly feels like this is an inferior, fairly non-representative, axis along which to reduce our dataset. So what makes our original Z_1 a better axis than this alternate Z_2?

The PCA algorithm claims that we want to project our data onto an axis where the **variance of the datapoints is maximized**. The **variance** of a dataset roughly measures how far each point is from the dataset mean. Notice how the variance of our data projected onto Z_1 certainly is larger than that of the data projected onto Z_2.

In the second case, much of the data coalesces to a few very close points, which seems like this dimensionality reduction is making us lose some information.

Now that we know that PCA seeks to find the axis that maximizes the variance of our projected data, we're going to drop a bit of math to formalize this notion.

For the 1-dimensional case, imagine that we have a dataset $(X_1, X_2, ..., X_n)$. Maximizing the variance of the data is equivalent to finding the unit-vector, v, that maximizes:

$$\frac{1}{n}\sum_{i=1}^{n}(X_i^T v)^2$$

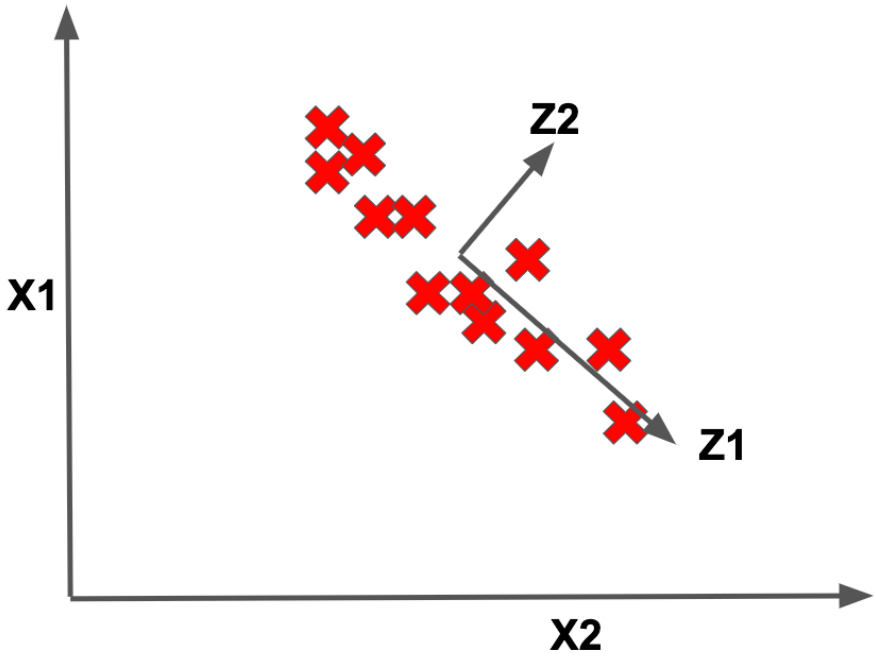

Figure 60: Axes fit on dataset

Finding this v can be done pretty efficiently using some linear algebra properties that we won't go into for now. Once we are done, this v is our first principal component!

The second principal component would be the axis that maximizes the variance on the data **after** we have removed the influence of the first principal component. If we extracted k principal components, these components would define a k-dimensional subspace that maximizes the variance of our data.

Final Thoughts

In practice, when we are reducing our dataset we will choose some reasonable number of principal components to extract. In this way we could take each datapoint consisting of maybe 1000 features and represent it as a compressed vector of maybe 20 values (if we chose to extract 20 principal components). That is pretty awesome!

Because of this, PCA is often a go-to technique for data preprocessing before any model training is done. Alternatively PCA can be used to visualize very high-dimensional data by projecting it into maybe 2 or 3 dimensions, to gain some insights into what the data represents.

One last practical note regarding PCA is that it is common to normalize the dataset before running the algorithm, by subtracting out the dataset mean and also ensuring that the features are using the same units (you don't want to have a feature that is a distance in meters and one in feet). After that normalization is done, we can run the PCA algorithm as described above.

Test Your Knowledge

PCA Definition

PCA Preprocessing

Best Axis

PCA Advantages

Deep Dive Into Neural Networks

Figure 61: Neural network meme

In this section, we will take our first deep dive into deep learning. Because deep learning is primarily the study of neural networks, we will spend the next few sections exploring the ins-and-outs of various classes of neural network models that each have different architectures and use-cases.

In this section, we will be studying the most vanilla flavor of neural networks: **feedforward neural networks**. Consider Figure 62, showing the process of running an image through some neural network-based classifier to determine it's an image of a dog:

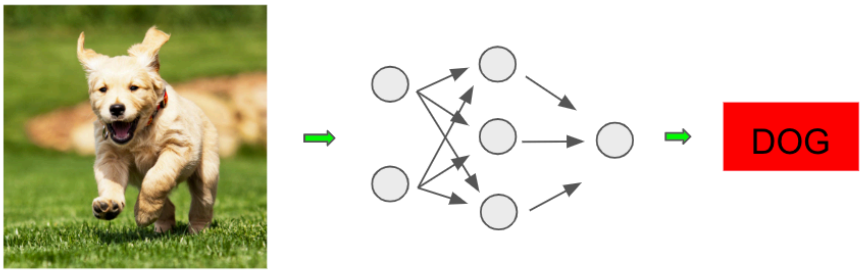

Figure 62: Basic neural network

The network in the middle is the main engine doing our classifying, and it turns

out this is our first example of a feedforward neural network. So what's going on here? Let's dive in and see!

Building Up the Network

Let's zoom in on just the model, shown in Figure 63:

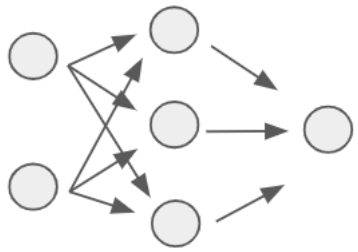

Figure 63: Neural network close-up

Figure 64 shows each component of this feedforward network labelled:

It turns out this network is performing certain layer-based computations. To understand what each layer is doing, we must first begin at the start of the model computation, which is where we provide the input data.

Recall that we can represent each point in a dataset through some featurized representation, and this is typically where we start when providing the point to a machine learning model. The exact features we extract often depend on the problem domain.

In the case of image classification, a common representation of an image is as a 3-d matrix (also known as a *tensor*) of red/green/blue pixel values.

Therefore if our image is 100x100 pixels, each pixel would consist of 3 values, one representing the strength of each of the red, green, and blue color channels. Our image represented as a 100x100x3 grid of values is shown in Figure 65:

We can further collapse this 3-d representation into a single vector of $100 \cdot 100 \cdot 3 = 30,000$ values. We do this merely for convenience, without modifying any of the underlying information. This is shown in Figure 66.

For the sake of our example, rather than operating on a $100,000$-d vector, imagine that we use a smaller representation of the picture consisting of a 3-d vector: $V = (1, 0.4, 2)$. Here we have picked these values arbitrarily.

Now let's take a 3-d weight vector $W = (-1, 0.2, 1)$ and compute the sum $W_1 \cdot V_1 + W_2 \cdot V_2 + W_3 \cdot V_3 = -1 \cdot 1 + 0.2 \cdot 0.4 + 1 \cdot 2 = 1.08$.

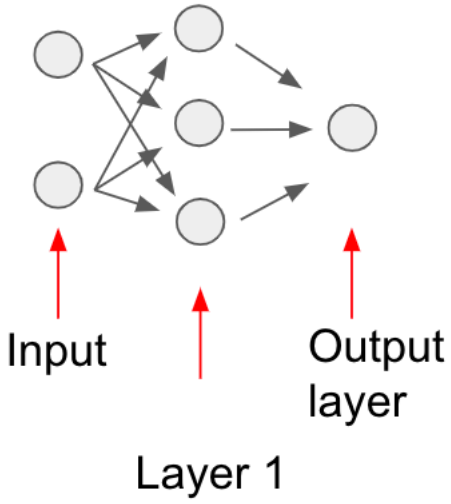

Figure 64: Neural network layer labelled

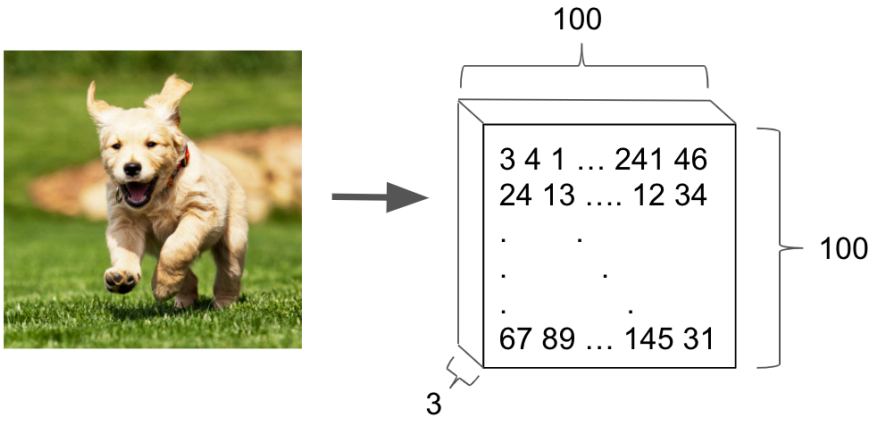

Figure 65: Image pixel value representation

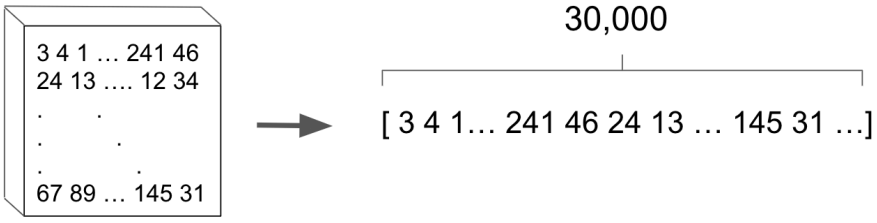

Figure 66: Collapsed image vector representation

Now let's further feed that sum through a sigmoid function (shown in Figure 67).

$$f(1.08) = \frac{1}{1+e^{-1 \cdot 1.08}} \approx 0.75$$

Figure 67: Feeding through sigmoid function

We then set that result as the value of the circle in the first layer (Figure 68).

This result is our first example of a **neuron** in the neural network terminology. Because the term **neuron** is a gross abuse of the biological term (and we don't want to annoy any of our neuroscience friends), let's just call it a **compute unit** for our purposes.

Now let's use a new weight vector $W_2 = (3, 0.3, -0.1)$ and perform the same computation with our feature vector, namely taking a linear sum and feeding it through a sigmoid function. If we repeat this computation with the new weight vector, we should get the result 0.81, which becomes the value of the second compute unit in our new layer (Figure 69).

We can repeat this computation for as many compute units as we want. The important thing is to use a different weight vector for each compute unit.

Let's say we repeat this computation for 3 units in total with some weight vectors getting the following values (Figure 70).

Congratulations! These computed values form the first layer of our neural network! These intermediate layers of a neural network after the layer where the inputs are fed in are called **hidden layers**.

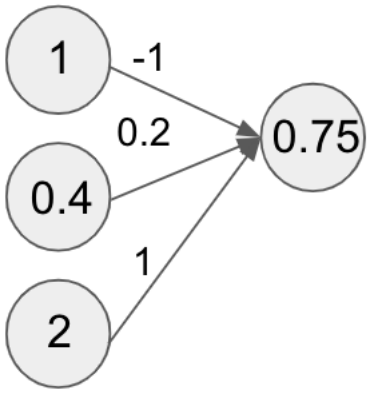

Figure 68: First unit neural network

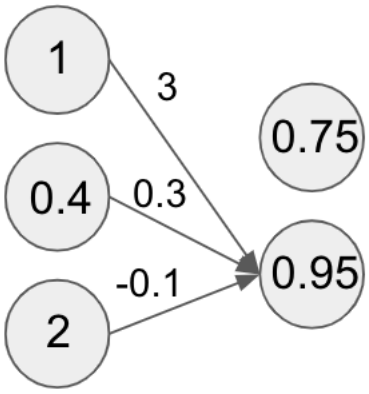

Figure 69: Second unit neural network

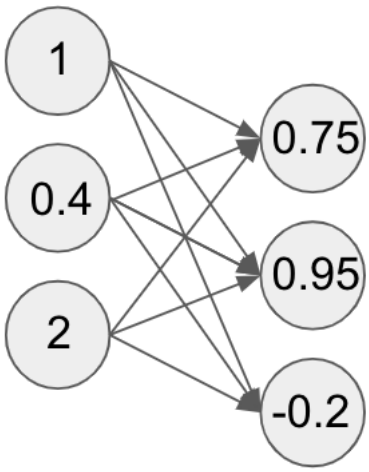

Figure 70: Third unit neural network diagram

This particular layer of computations also has a special name: a **fully connected layer**. Again we can have as many compute units as we want in the layer, and that number is something we can tune so as to maximize performance.

In addition, the sigmoid function we chose to run our linear sum through has a special name in the neural network community: an **activation function**. It turns out we can easily choose different activation functions to feed our linear sum through. We will explore a few commonly used variants in the next section.

One last note: we didn't really mention how we chose the values of our weight vectors. This seems like an important detail to mention because the weights determine how certain features get upweighted (or downweighted) at a given compute unit. The truth is we picked them randomly, and this is basically what is done in practice (with a few caveats which we will discuss later).

Now that we have this vector of 3 compute units at the first layer of our network, why not repeat the same set of operations as before?

So that's what we do. We pick new weights per compute unit, take a linear sum with the vector of values from the first layer, and run the result through a sigmoid function. This gives us Figure 71.

Nice! This is the second layer of our network! Let's assume that we are doing a 3-way classification task where we are trying to predict whether an image is of a **cat**, a **dog**, or **neither**.

Therefore we want to end up with three values that give us the probability for

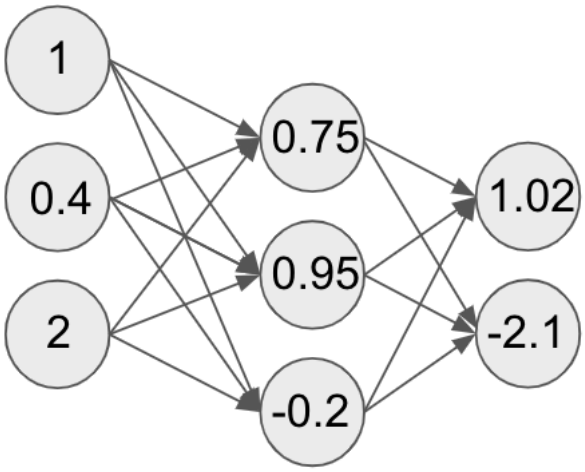

Figure 71: Second layer neural network

the values **cat**, **dog**, and **neither**. Let's run our second layer through another similar set of computations. This will give us a distribution over three values as shown in Figure 72

This set of computations through the layers of our network is called the forward pass. Now we finally see why these are called feedforward networks: we are **feeding** values **forward** through the network.

One important note on hidden layers and compute units per layer of a neural network. In general, the number of layers and the number of units are both hyperparameters that we can adjust as necessary to achieve the best generalization error in our network.

Backpropagating Errors

These computations are all nice and good if our network predicts the right value for a given input. However, it seems pretty unlikely that if our weights were randomly chosen that our model would know to predict the right value.

So what's missing? It turns out our feedforward pass is only half the story. We have said nothing about how to actually train the model using our data. Let's assume that during our forward pass our neural network made a mistake and instead predicted a **cat** (Figure 73)

It turns out that though our network made a mistake, we can still compute a cost for the values that our network computed. Intuitively it seems that if our

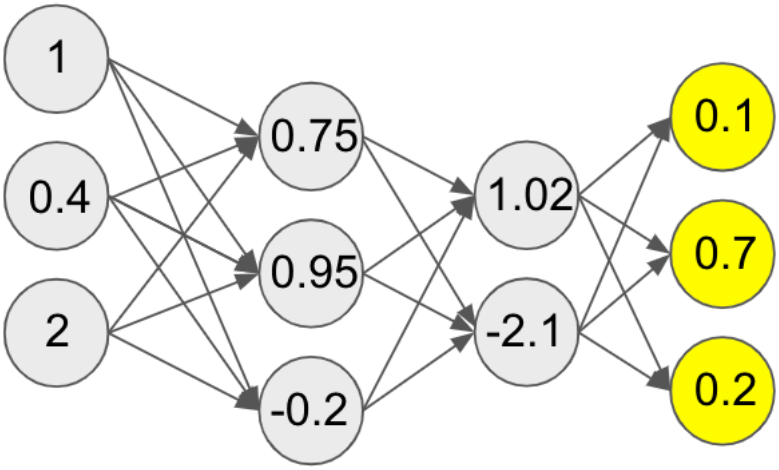

Figure 72: Third layer neural network

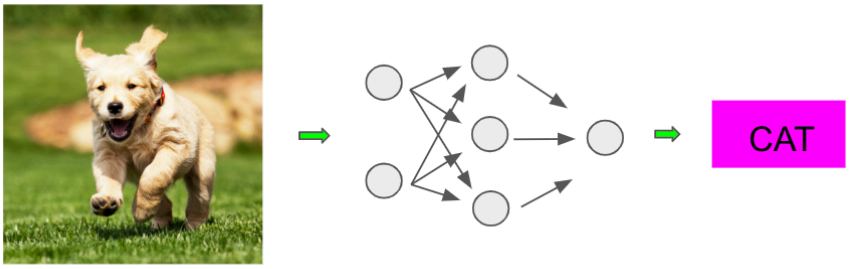

Figure 73: Neural network incorrect classification

network predicted $(0.35, 0.45, 0.2)$ for **(dog, cat, neither)** that is *less wrong* than if it had predicted $(0.1, 0.8, 0.1)$.

So we can formally associate a value for this level of *wrongness* using a cost function, similar to what we did previously with models such as linear regression. We will discuss some details about the possible cost functions we can use for different tasks in later sections.

For now let's assume the cost calculated is 5. Neural networks as we have defined have a special property called **differentiability**, which means we can compute derivatives of the cost function with respect to all the weights. Woah, that was a mouthful.

We won't go into the nitty-gritty details, but let's take the weights of the the first compute unit of the final layer (of the neural network in our running example). We can compute gradients indicating how much we should update these weights to get closer to predicting the right value (thereby lowering the value of the cost function).

In Figure 14, we see the gradient of the first unit with respect to the weights in the preceding layer flow backwards. This is indicated by the red arrows (Figure 74).

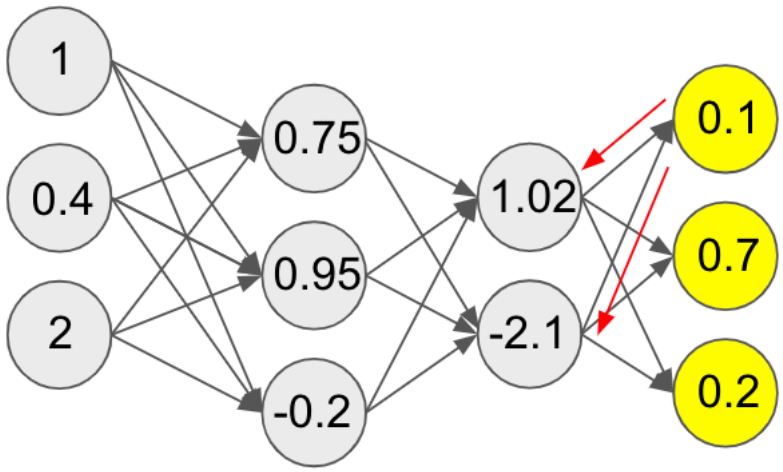

Figure 74: Back flow neural network gradient

Once we have these gradients, we update the weights as necessary. Notice how here the end result is being used to send information backward in the network. This backward flow of signal is called **backpropagation**.

It turns out we can compute these gradients for all the weights in a given layer.

Once we have computed the gradients and updated the weights in the last layer, we compute the derivatives and update the weights in the second layer.

We can continue this all the way to the first layer. These backpropagation updates are done using gradient descent. In practice, we update all the weights in the entire network for some number of iterations until our cost function decreases as much as we would like.

Final Thoughts

This concludes our (hopefully) gentle introduction to feedforward neural networks. While they may be the most vanilla networks, in practice feedforward networks can be successfully applied on many regression and classification tasks. Though we used the example of image classification, we can apply feedforward networks to many other problem domains.

In general, the compute units in neural network hidden layers learn many powerful representations of the data during training. In the next lessons we will be building out some of the details of neural network theory.

Test Your Knowledge

Matrix Practice I

Matrix Practice II

Neural Network Grab Bag

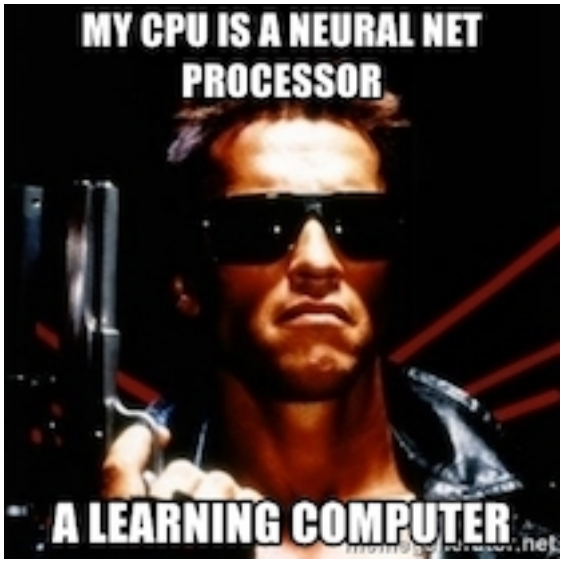

Figure 75: Neural network meme

In a previous section, we went through a motivating example of a feedforward neural network from start to finish. We introduced a lot of neural network concepts but also left out a lot of details related to activation functions, weight setting, and other aspects of neural network theory. In this section, we will make a grab bag of all the points we missed. Let's get started!

The Universal Approximator

Though we have introduced the neural network, we haven't really developed a sense for what makes these networks so great.

For starters, neural networks are a ridiculously powerful model class that can represent a diverse collection of functions. It has been mathematically shown that neural networks that have at least one layer are **universal approximators**.

What this means is that a neural network with at least a single layer *and* some nonlinear activation function can approximate any continuous function. That result is insane! You could literally draw any random function off the top of your head and a single-layer neural network could learn a representation of that function.

There's a big caveat, however. This is a **theoretical** result which means that

while a 1-layer network **could** learn any function, whether you are able to get it to do that **in practice** is a separate question. It turns out that it's quite hard to get a neural network to learn **any** function.

Our ability to do that is predicated on a number of other factors, including whether or not we can get our cost function to converge and all of the additional considerations that we will discuss below. But the fact that literally the simplest neural network can do so much is already an impressive property of this model class.

Weight Initialization

In our last section, we mentioned that at the beginning of neural training we initialized the weights of our model randomly. The truth is that is *literally* how the weights of our model are initialized (with a few bells and whistles).

An important point to understand is that we absolutely need the weights to be different for separate compute units of a given layer. If they were not, we would have identical compute units for a given layer, which would be like having a redundant feature. Randomizing the weights allows us to get the necessary variety in the weights we need.

Because neural networks are so powerful, they are also very susceptible to overfitting, so we want to *jitter* them a bit during learning. Random weight initialization helps us achieve this. Therefore in practice we often initialize each parameter to a random value within some small range like $(-1, 1)$.

Activation Functions

Recall that when we introduced our first neural network, an activation function was the function we applied to a value *after* computing the linear sum of a set of a weights with a certain layer.

In particular, we used a sigmoid activation function to transform the linear sum of each compute unit. We had previously encountered a sigmoid in our study of logistic regression. As a reminder, let's recall the form of a sigmoid shown in Figure 76.

It turns out that we can use other functions for our neural network activation function, and some are actually preferred! The only **really** important thing is that we use some function that performs a **nonlinear transform** on its input.

The way to see why this is important is imagine we had a 2-layer neural network that only applied a linear function between layers. In that case, we could write the full network computation as $Linear_2(Linear_1(X)) = Linear(X)$. This means applying a sequence of linear functions is equivalent to applying a single aggregated linear function. This would make it so that our network could only learn linear functions!

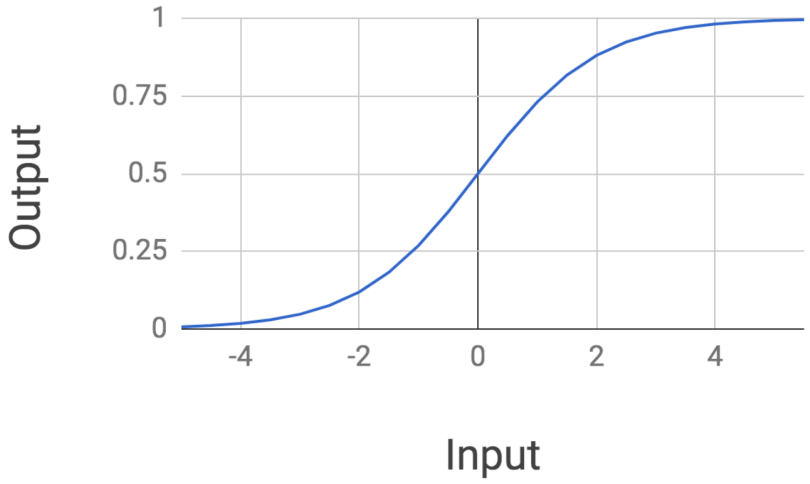

Figure 76: Sigmoid activation

In practice, sigmoids are not the go-to activation function to use because their outputs are not zero-centered and their gradients quickly drop to 0 for very large and very small values. This makes them very finicky to use during network training.

Two other common activation functions used in neural networks are the tanh and the **rectified linear (ReLu)** unit. The tanh function is shown in Figure 77:

Tanh looks very similar to the sigmoid except it is zero-centered. The ReLu activation is shown in Figure 78:

The ReLu is a nice well-behaved function that is easy to compute and allows for efficient training of networks. In general, both the ReLu and tanh are used in place of the sigmoid because they result in superior training of networks.

Regularization

Like any other supervised learning model, neural networks can be vulnerable to overfitting. In fact, because they are so powerful, they can be even more vulnerable than other model families.

It turns out that we can use many of the same regularization techniques we introduced previously to offset these effects. For example, we can modify the cost function we are optimizing during training by adding an L_1 or L_2 regularization term.

However, there is one neural network-specific regularization technique called

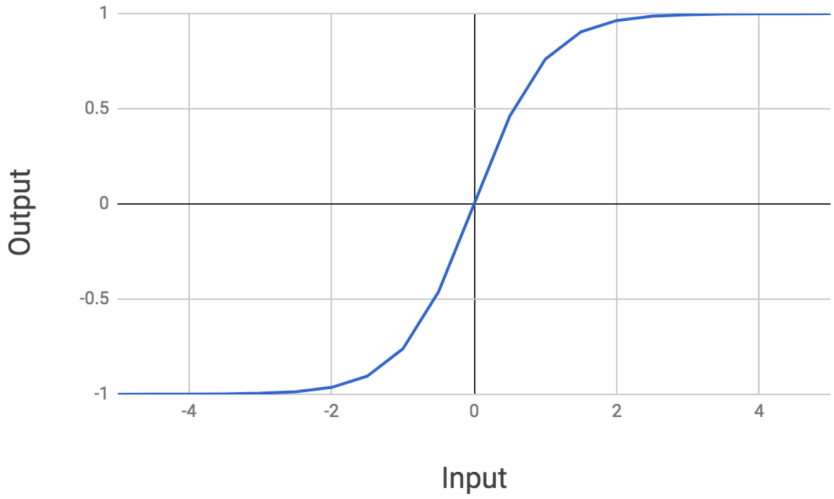

Figure 77: Tanh activation

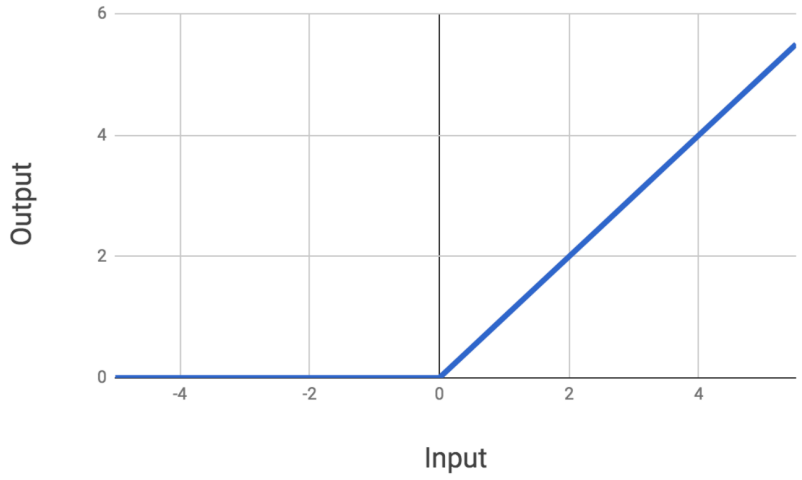

Figure 78: Relu activation

dropout. The idea behind dropout is relatively straightforward, so much so, that it's astounding how well it works in practice.

The basic idea behind dropout is that during the forward pass of our neural network, going from one layer to the next, we will randomly inactivate some number of units in the next layer. To make this concrete, assume we had the neural network in Figure 79.

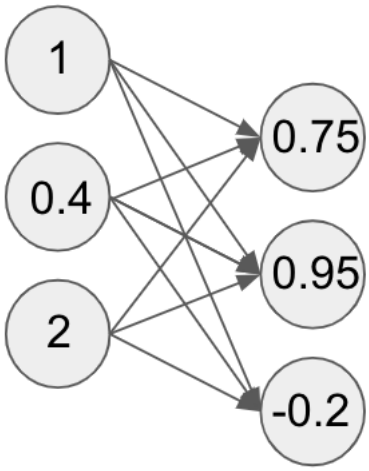

Figure 79: Neural network

In Figure 80, we see the result of applying dropout to the second layer of the neural network.

While in the diagram, it looks like all we've removed is some arrows to the compute units, this has the effect of actually *zeroing out* those compute units.

This is because we have essentially removed all the weights from the preceding input compute units to the output ones we are dropping. When we apply dropout, we drop each compute unit with some probability we choose, typically somewhere between $0.1 - 0.5$. This probability is a hyperparameter we can tune during training. Furthermore, we can apply dropout to hidden layers of a network as well as input layers.

Dropout has the effect of breaking symmetry during the training process by preventing the neural network from getting caught in local minima. It turns out this is an important effect because during training networks can get stuck in local optima, missing out on superior solutions.

Dropout is a bit of a strange idea because we are effectively forcing the network

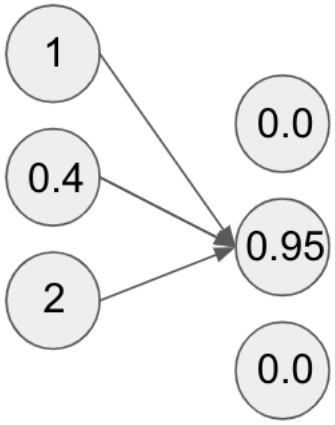

Figure 80: Neural network with dropout applied

to *forget a bit* of what it knows during training. It's like saying "Hey neural network you're getting a little too comfortable with this one feature. Forget about it for a bit." Note, here we are referring to compute units as features because they effectively function as features.

When we are done training a network and ready to feed new inputs through our model, we will not drop any units from our network. Instead, we will scale output unit values by the dropout probability. This ensures that at test time, our units have the same expected value as they had during training.

Strange though it may be, in practice dropout is arguably the most effective regularization technique used for neural networks.

The Power of Layers and Units

From the get-go, there are at least two degrees of freedom in determining the underlying architecture of a feedforward network: the number of hidden layers and the number of units per layer.

As we increase the number of layers in a neural network, we increase its representational power. The same is true for the number of compute units per layer. It turns out we can make our networks arbitrarily powerful by just bumping up its number of layers and compute units per layer.

However there is a downside to this that we must be aware of. The increase in power is offset by an increase in computational cost (i.e. it is more expensive

to train a model) as well as the fact that the models must be regularized more aggressively to prevent overfitting.

Loss Functions

In the past we have used the term cost function to refer to the mathematical function we are trying to optimize during training of a model. Here we will use the term **loss function** as it is more common in neural network descriptions, though recognize that we are treating the two terms as equivalent.

When we introduced our feedforward model, we were quite ambiguous about the details of the loss function we were using backpropagated during training. To make things concrete, we will discuss one commonly used loss function for classification: **the cross-entropy loss**.

Assume that we are outputting one of three labels $[1, 2, 3]$ for a dataset $[(X_1, Y_1), ..., (X_n, Y_n)]$, then our loss function takes the following form:

$$-\sum_{j=1}^{n}\sum_{c=1}^{3} I_{j,c}\log(P_{j,c})$$

Note that here $P_{j,c}$ refers to the probability of our model outputting label c for the datapoint j. In addition, $I_{j,c}$ is an indicator variable which is equal to 1 if datapoint j has a true label of c. For example, this means that if Y_3 has a label of 2, then $I_{3,2}$ is 1 while $I_{3,1}$ and $I_{3,3}$ are 0.

Neural networks can also be used for regression, and in those cases we can use a least squares loss as we did for linear regression.

Other Training Tidbits

There are a few other tidbits of neural network training that are worth mentioning. First off, when we are training a network with a given loss function it is crucial to ensure our gradient calculations are correct for backpropagation.

Because our gradients determine how much we should update certain weights so as to decrease our network's loss, if our computed gradients are off, our network training could be either subpar or completely wrong! Our network loss function may mistakenly never converge, so we really have to double check our math with these gradients.

In practice, today most libraries used for building neural networks such as Tensorflow, PyTorch, etc. perform **automatic differentiation**. This means that you don't actually have to mathematically derive gradients. Instead you just define your loss function and your model architecture and the libraries perform all the gradient updates during backpropagation implicitly. Super neat!

On a related note, neural network convergence during training is tricky to get right sometimes. Even if we don't make any errors in our gradients, networks have a tendency to get stuck in local optima during training. Because neural network training is a nonconvex optimization problem, we are never sure that we have achieved the absolute global optimum for a problem.

It may also take really long for a network to converge to any sort of an optimum. Neural network training for certain types of problems can take upwards of several weeks! Getting networks to train appropriately often requires us to tune various hyperparameters such as dropout probabilities, the number of layers, the number of units per layer, and other related factors.

Finally, it is worth commenting about different gradient descent techniques. When training neural networks and other machine learning models, two common gradient descent techniques we can use include **batch gradient descent** and **stochastic gradient descent**.

In batch descent, during an iteration of training we compute an aggregated gradient for a collection of datapoints that we use to update the weights. Batch descent generally leads to quite stable updates for each iteration of training, though each iteration takes a bit longer since we have to compute gradients for a collection of points.

By contrast, with stochastic descent we only perform weight updates with the gradient computed for a **single datapoint at a time**. Stochastic descent leads to a much more jittery descent during training, though each update is very fast since it only involves one point.

Final Thoughts

Phew! That was a lot of information spread across a wide variety of topics. At the end of the day, it is important to recognize that neural networks require quite a bit of details to get right and use effectively on new problems.

Sometimes building neural network models is more an art than a science. That being said, there do exist some systematic ways to build better models, many of which we have touched on in this section, so make sure to keep them in your machine learning toolkit!

Test Your Knowledge

Activation Practice I

Activation Practice II

Activation Practice III

Weight Initialization

Batch vs. Stochastic

Gradient Checking

Dance Dance Convolution

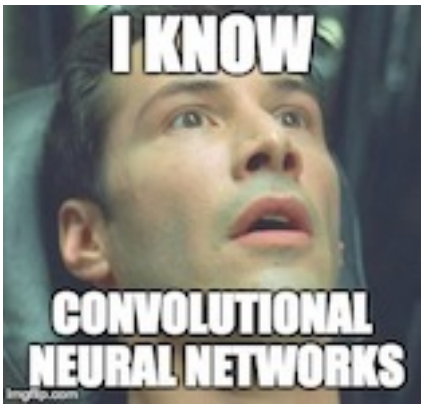

Figure 81: Convnets meme

About 9 years ago, the ImageNet challenge, an annual visual recognition competition was begun. In the challenge, teams were tasked with building the best object classifier on a collection of images from a set of 1000 possible object labels.

In the year 2011, a team from the University of Toronto trounced the competition by introducing the world's first successful image recognition system built entirely using neural networks.

This milestone helped to trigger the artificial intelligence wave that we are currently experiencing. Today we are going to investigate the neural network architecture that was at the heart of this revolutionary point in history. Excited? Let's get to it!

Motivations

The neural network class that was at the heart of the deep learning revolution is called **convolutional neural networks**. To motivate their design, we will start with the problem of image recognition. Consider the image of a dog in Figure 82.

Our brains are amazing because we are able to just about instantaneously recognize that this image contains a cute puppy. What allows us to recognize that this is a puppy?

Our brains are keen at picking up relevant features in an image, scanning efficiently for the things that are necessary for us to differentiate the image. Consider a more complicated example shown in Figure 83.

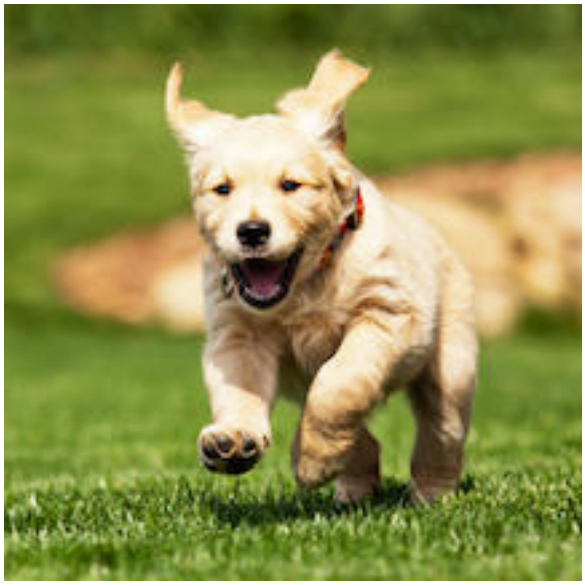

Figure 82: Dog

Figure 83: Jackson Pollack

Here it's not immediately clear what this is a picture of, so we must analyze the picture more deliberately. We may start at the upper left corner and scan horizontally, picking out details that seem relevant.

After we have scanned the entire image, our brain holistically tries to put together the pieces into some impression of the image subject. This intuition for how our own vision system seems to work by scanning and picking out features will be a useful analogy for convolutional networks.

The Convolutional Layer

Let's revisit how we represent images mathematically. Recall that images can be represented as a 3-d tensor of (red, green, blue) color channels. Previously when we were studying feedforward networks we collapsed this 3-d representation into a single dimensional vector. It turns out in practice this vector collapsing is not really done for images.

Instead imagine that we have some $4x4x3$ image and we keep it in its 3-dimensional format. We will aim to process our image by running it through a series of layerwise computations, similarly to how we did for feedforward networks. To do that we will take a $2x2x3$ tensor of weights, which we will call a **filter** (Figure 84):

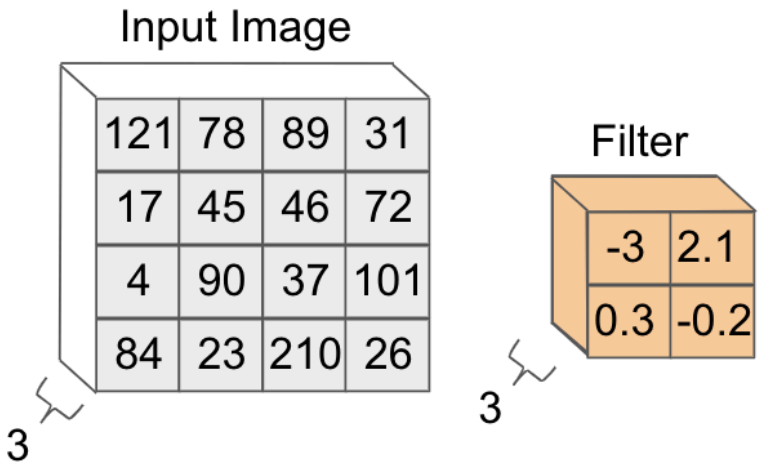

Figure 84: Convolutional network filter

Let's begin by applying our filter to the upper left $2x2x3$ corner of the image. The way this is done is we chop up our filter into 3 $2x2$ depth slices (Figure 85)

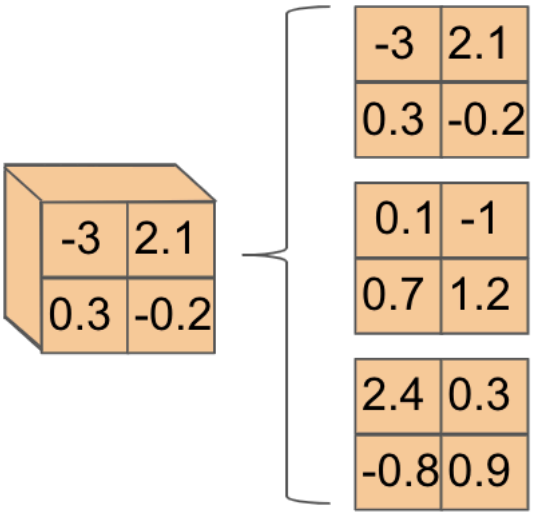

Figure 85: Convolutional neural network deconstructed into filters

We do the same for the corner of the image we are interested in looking (Figure 86).

Then we take the pairs of (depth-wise) matching slices and perform an element-wise multiplication. At the end, we take the 3 values we get from these depth-wise operations and we sum them together (Figure 87).

For those that are mathematically-inclined, we just took a 3-d dot product! This computed value will be the first element in the next layer of our neural network. Now, we slide our filter to the right to the next $2x2x3$ volume of the image and perform the same operation (Figure 88).

Note that we also could slide our filter over one pixel, rather than two as we have done. The exact number of pixels that we slide over is a hyperparameter we can choose. After this, we move our filter to the lower-left corner of the image (Figure 89).

and compute the same operation. We then do the same operation again, after sliding our filter horizontally. At the end, we have four values in this first slice of our next layer (Figure 90).

Exciting! This is our first example of a **convolution**! The results of applying this filter form one slice of this next layer in our network.

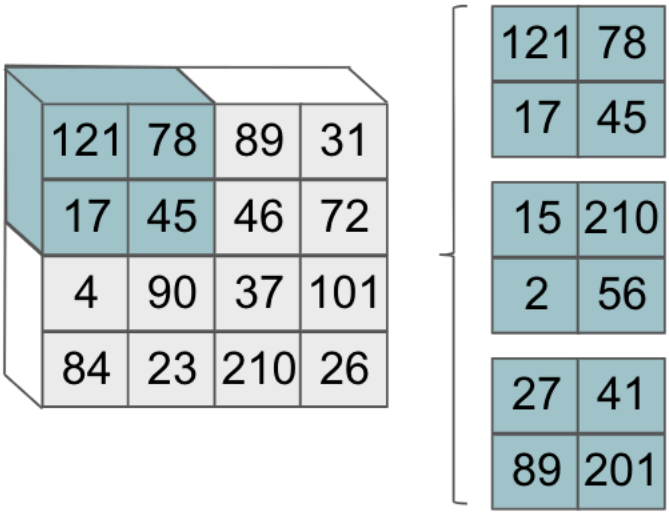

Figure 86: Convolutional neural network deconstructed corner

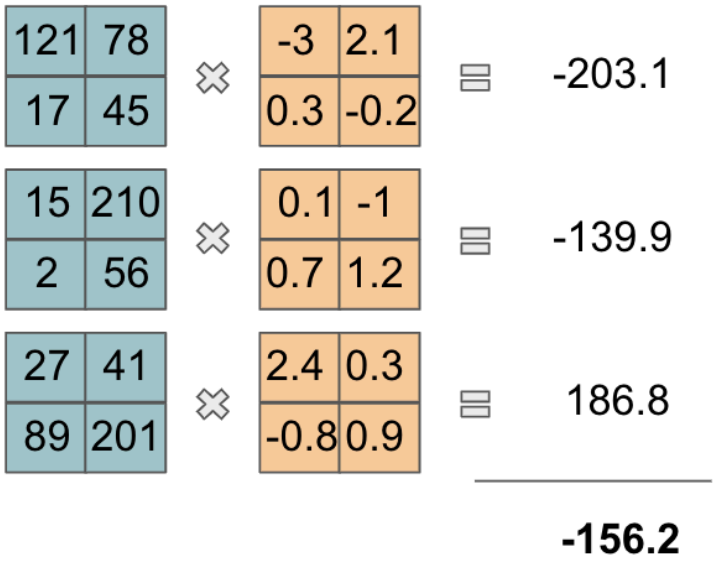

Figure 87: Convolutional neural network filter multiplied

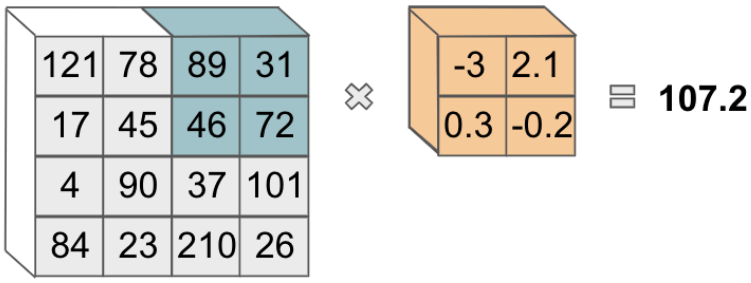

Figure 88: Top right filter multiplication in convolutional neural network

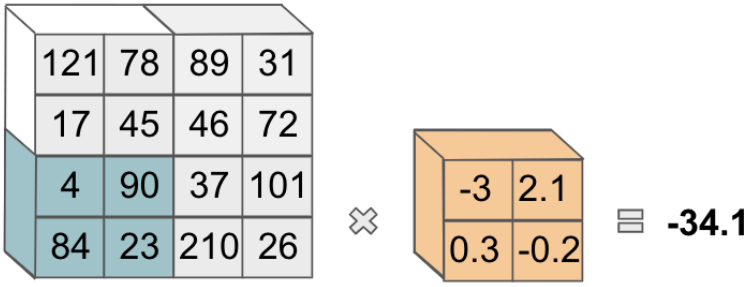

Figure 89: Lower left filter multiplication in convolutional neural network

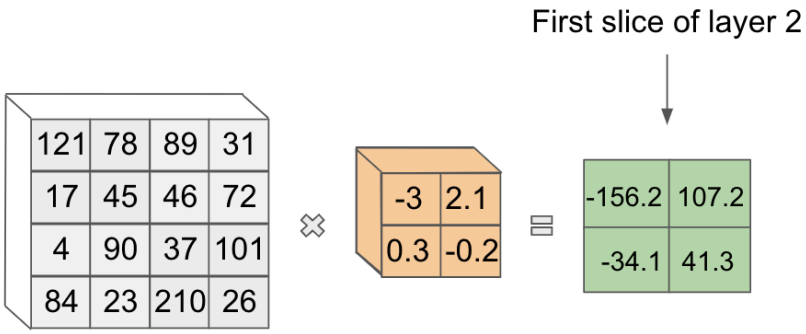

Figure 90: First filter multiplication in convolutional neural network

It turns out we can use another filter with new weights on this same image to get *another* slice in the next layer (Figure 91).

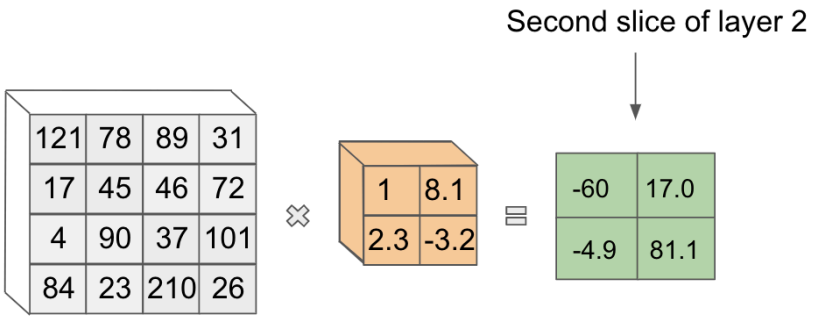

Figure 91: Second filter multiplication in convolutional neural network

In theory we can use as many unique filters as we want in our convolution, though the number to use becomes a hyperparameter we can tune.

When we are done applying filters, the output of these filter operations are stacked together to get our complete next layer. If we had used 10 filters in the first layer of our running example, our second layer would be a $2x2x10$ tensor of values. This transformation from input to output using filter convolutions is appropriately called a **convolutional layer**.

Note that once we have our second layer, we can start from scratch and apply another convolutional layer if we so choose. Notice also that as we apply our convolutions, the length and width dimensions of our tensors tend to get smaller.

The depth of our output layer is equal to the number of filters we use for our transformation, so depending on that number, we might also reduce the depth dimension. For this reason, convolutional layers can be viewed as a form of feature reduction.

One of the most elegant things about the convolution layer is that it has a very natural analogy to how humans compose their understanding of complex images.

When we see something like a puppy our visual understanding process may first pick out prominent points like eyes and the nose. It may then create a notion of edges that form the outline of the dog. We then may add textures to the outline and colors. This hierarchical process allows us to decompose images into smaller constituents that we use to build up our understanding.

In a similar fashion, convolutional networks can also learn hierarchical features as we go up the layers. Certain works have demonstrated that when you actually

visualize what is being learned at the layers of a convolutional network, we see visual entities of increasing complexity (points, lines, edges, etc.) as we go deeper into the network. This is really cool!

In addition to this nice intuitive design and interpretability, what else do we gain by using convolutions instead of our original feedforward fully connected layers? When we use convolutional layers, we also see a drastic reduction in the number of weights we need in a given layer.

Imagine if we collapsed a reasonably-sized image of dimensions $100x100x3$ into a 30,000-length vector. If we wanted to make a fully-connected layer, each compute unit in the first layer would be connected to every value in the input, creating 30,000 weights. If we had even a moderately-sized first layer of 100 units, we would have $30,000 \cdot 100 = 3,000,0000$ parameters just in the first layer! This clearly is not scalable.

On the other hand, if we didn't collapse the image and used a $5x5$ filter for a convolution, we would have $5 \cdot 5 \cdot 3 = 75$ weights per filter. Even if we used a relatively large number of filters like 200, we would have a total of $200 \cdot 75 = 15,000$ weights in the first layer which is a **huge** reduction! In this way, convolutional layers are more memory-efficient and compute-efficient when it comes to dealing with images.

Convolutional Network Mechanics

There are a number of design decisions in the convolutional layer that we can play with that influence the network's learning ability. For starters, we can certainly say a bit more about the dimensions of the filters applied.

In practice, we can see filter sizes anywhere from $1x1$ to upwards of $11x11$ in some research papers. Note also that when applying a $1x1$ filter, we wouldn't actually reduce the length and width dimensions of the layer we are convolving.

In addition, there's another design factor that we've used without formally defining. When we were sweeping our filter across the image in our motivating example, we were moving the filter over two pixels in a given dimension at a time (Figure 92).

However, what if we had shifted over our filter by one pixel instead? Notice that this would determine a different subset of the image that the second feature would focus on (Figure 93).

The value that determines how much we slide over our filter is called the **stride**. In practice, we often see that using smaller stride values produce better performance on trained convolutional networks.

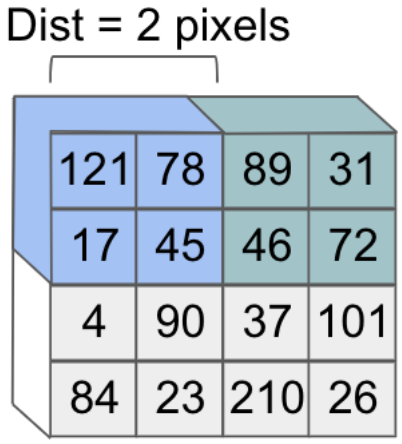

Figure 92: Convolutional neural network stride 2

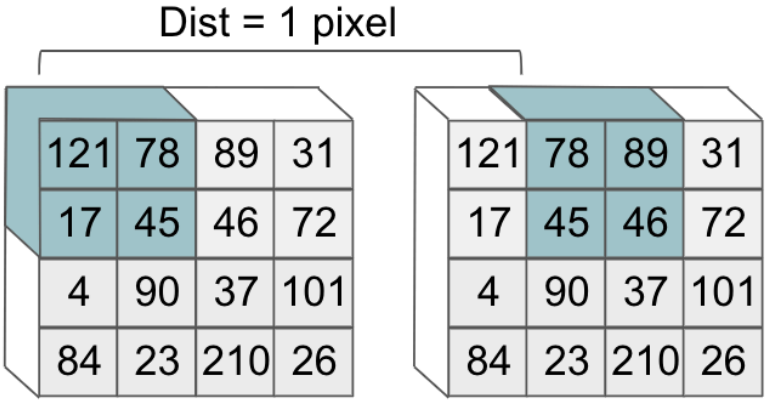

Figure 93: Convolutional neural network stride 1

Pooling Layers

In addition to convolutional layers, there is another commonly used layer in convolutional networks: the **pooling layer**. The purpose of pooling layers is often to more aggressively reduce the size of a given input layer, by effectively cutting down on the number of features.

One of the most common types of pooling is called **max pooling**. As its names suggests, the max pooling layer takes the max value over an area of an input layer (Figure 94).

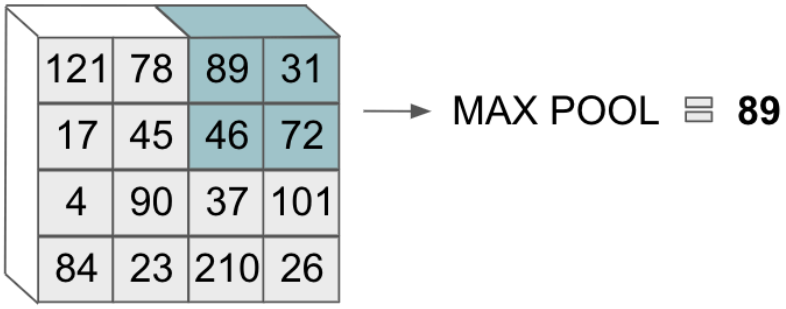

Figure 94: Convolutional neural network max pool

With pooling layers we also extend the operation depth-wise and apply it over a variably-sized area, as we did with convolutional filters. For example, we could use max pooling over a $2x2$ area that determines the segment we are applying the operation to. The depth of the output volume does not change.

Zooming out, we must be careful when choosing the filter and stride size in our convolutional networks and operations such as max-pooling tend to throw away a lot of information in an input layer.

The 2011 ImageNet Winner

With all that theory in place, we are now finally ready to revisit the 2011 ImageNet-winning entry. Historically, a lot of the architectures that have won past ImageNet competitions have been given catchy names because gloating rights are a thing in machine learning !

The architecture that won ImageNet 2011 was called **AlexNet** after its principal creator Alex Krizhevsky (Figure 95).

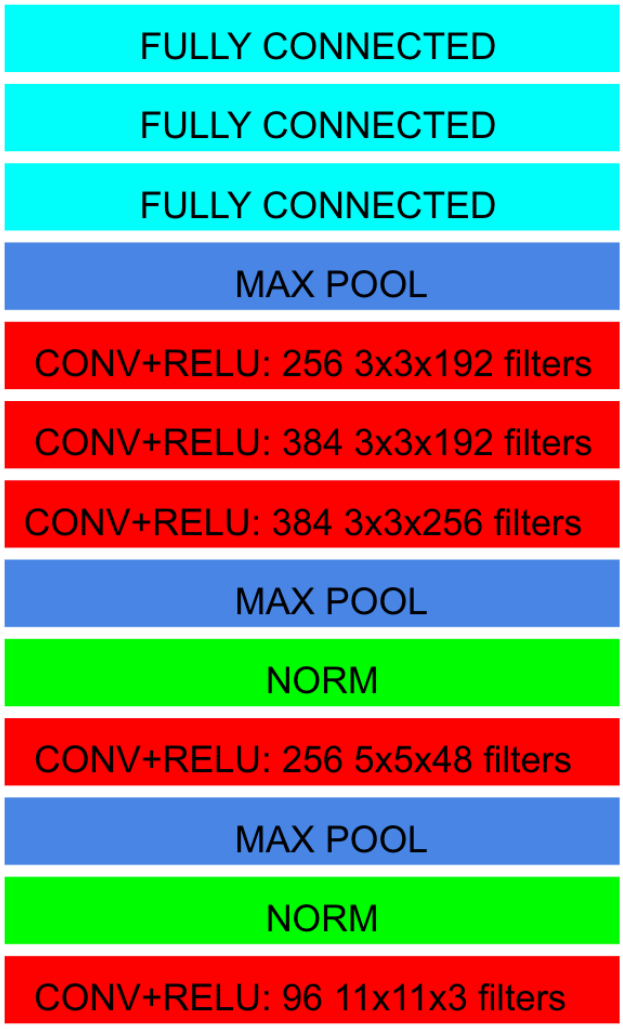

Figure 95: AlexNet convolutional neural network architecture

Here **norm** refers to a specific normalization operation that was introduced to help generalization. This particular operation has fallen out of favor in convolutional network architectures since then.

In addition, **fully connected** refers to a feedforward layer we saw when we looked at feedforward neural networks. At the end of the last fully connected layer, the architecture outputted a probability distribution over 1000 labels, which is how many labels there were in the challenge.

An important thing to note is many of the ideas behind the architecture of AlexNet had been introduced previously. What AlexNet added that was so revolutionary was increasing the depth of the neural network in terms of number of layers.

Typically as networks get more deep they get harder to train, but the University of Toronto team managed to effectively train a very deep network. Another novel contribution was the use of several stacked convolutional layers in a row, which had not been explored previously.

And with that, we've come full circle! From humble beginnings, we've successfully discussed the model that triggered the modern-day resurgence of artificial intelligence. What a phenomenal milestone in our AI journey!

Test Your Knowledge

Convolutional Application

Convolutional Layer Advantages

Recurrent Neural Networks

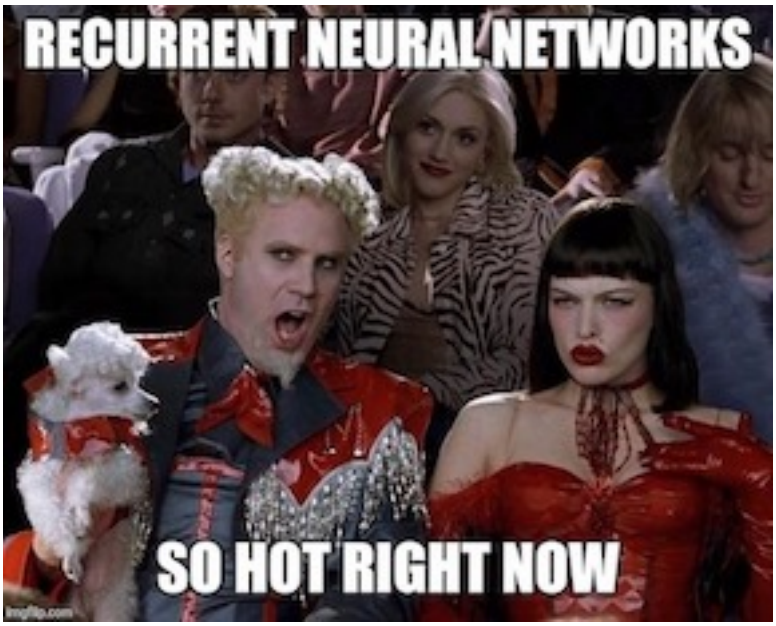

Figure 96: Source Chelsea

In this section, we continue our study of deep learning by introducing an exciting new family of neural networks called **recurrent networks**. Just as convolutional networks are the de facto architecture for any vision-related task, recurrent networks were once *the* standard for language-related problems.

In fact, for a long time there was a belief among natural language researchers that recurrent networks could achieve state-of-the-art results on just about *any* natural language problem. That is a tall order to fill for a single model class!

That being said, today on many natural language tasks, recurrent networks are still incredibly performant. So what's the big deal behind these recurrent networks? Let's take a look.

Motivations

Recurrent networks succeed in many natural language tasks because understanding natural language requires having some notion of memory, particularly memory related to modelling sequences.

For example, if I gave you the sentence **"I picked up the remote controller on the couch and turned on the...,"** and asked you to fill in the missing

word, you would probably have no problem filling in a word like **"television."**

In order to do that, you had to process all the given words in sequence, associate the action of **"picking up"** to the **"controller"**, interpret the situational context of a controller on a **"couch"**, and use that to inform what item could be **"turned on"** given this context. This level of processing is an absolutely amazing feat of the human mind!

Having a powerful sequential memory is essential here. Imagine if you were reading the sentence and by the time you got to the end, you had forgotten the beginning and all you remembered was **"turned on the"**.

It would be **MUCH** harder to fill in the right word with just that phrase. It seems like **"toaster"**, **"lawn mower"**, and **"drill"** could all be valid responses given only the phrase **"turned on the."** The full context informs and narrows the space of reasonable answers.

It turns out that neither feedforward neural networks nor convolutional networks are particularly good at representing sequential memory. Their architectures are not inherently designed to have sequential context intelligently inform their outputs. But let's say we wanted to build an architecture with that capability. What would such an architecture even look like?

The Recurrent Unit

Let's try to design an appropriate architecture. Imagine that we are feeding an input, X_1, into some neural network unit, which we will leave as a black box for now. That unit will do some type of computation (similar to our feedforward or convolution layers from our previous sections) and produce an output value O_1 (Figure 97).

To make this even more concrete, our input could be the start of our phrase from above, namely the word **"I"**, and the output (we hope) would be **"picked"** (Figure 98).

Now that we have output **"picked"**, we would like to have that output inform the next step of computation of our network. This is analogous to when we are processing a sentence, our minds decide what word best follows from the previous words we've seen.

So we will feed in **"picked"** as the input of the next step. But that's not sufficient. When we process the word **"picked"** and are deciding what comes next, we also use the fact that the previous two words were **"I picked."**

We are incorporating the full history of previous words. In an analogous fashion, we will use the computation generated by our black box in the first step to also inform the next step of computation.

This is done by integrating what is called a **hidden state** from the first compute unit into the second one (in addition to the token **"picked"**). This is shown in

Figure 97: General compute for recurrent neural network

Figure 98: Recurrent unit with word

Figure 99.

We can repeat this procedure for the third step of computation (Figure 100).

In fact, we can do the same process all the way to the end of the phrase, until our model outputs the desired word **"television."** This is shown in Figure 101.

Notice that we are using the same black box unit for the computations of all the timesteps. And with that we have constructed the hallmark architecture of a recurrent network!

A recurrent network is said to **unroll** a computation for a certain number of timesteps, as we did for each word of the input phrase. The same compute unit is being used on all computations, and the most important detail is that the computation at a particular timestep is informed not only by the input at the given timestep but by the context of the previous set of computations.

This context being fed in serves as an aggregated sequential memory that the recurrent network is building up. We haven't said much about what actually goes on within the compute unit. There are a number of different computations that can be done within the compute unit of a recurrent network, but we can describe the vanilla case pretty succinctly.

Let S denote the sequential memory our network has built up, which is our **hidden state.** Our recurrent unit can then be described as follows:

$$S_t = F(UX_t + WS_{t-1})$$
$$O_t = \text{Softmax}(V^T S_t)$$

Here the function, F, would apply some sort of a nonlinearity such as tanh. U and W are often two-dimensional matrices that are multiplied through the input and hidden state respectively. V is also a two-dimensional matrix multiplied through the hidden state.

Notice also that the computation at a given timestep uses the hidden state generated from the previous timestep. This is the recurrent unit making use of the past context, as we want it to.

Mathematically, the weight matrix U has the effect of selecting how much of the current input we want to incorporate, while the matrix W selects how much of the former hidden state we want to use. The sum of their contributions determines the current hidden state.

When we say we use the same recurrent unit per computation at each timestep, this means that we use the exact same matrices U, W, and V for our computations. These are the weights of our network.

As mentioned, the role of these weights is to modulate the importance of the input and the hidden state toward the output during the computations. These

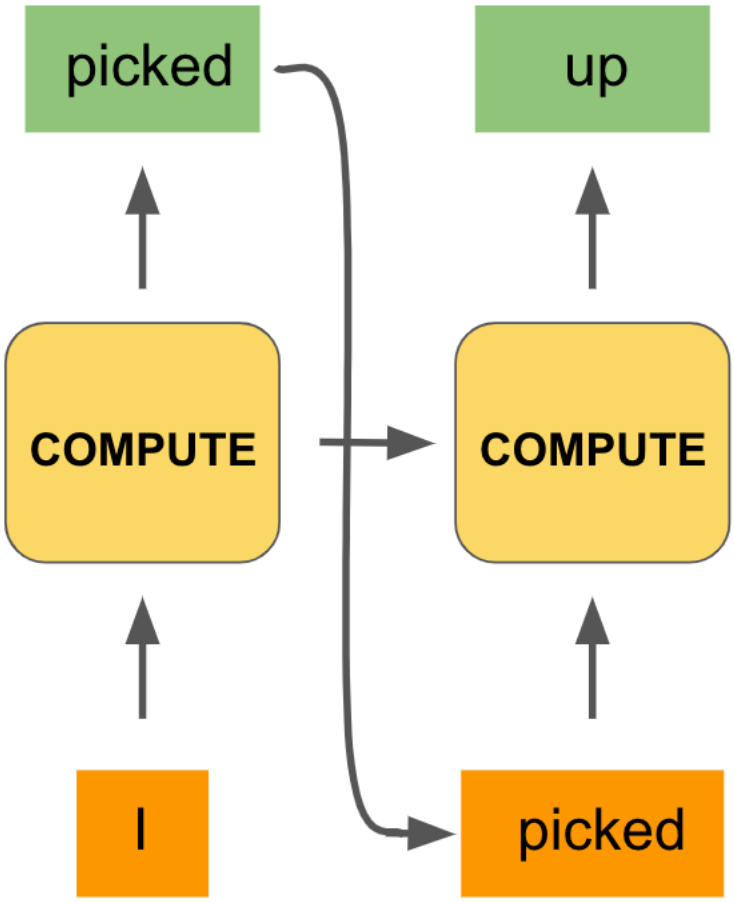

Figure 99: Two recurrent unit states in recurrent neural network

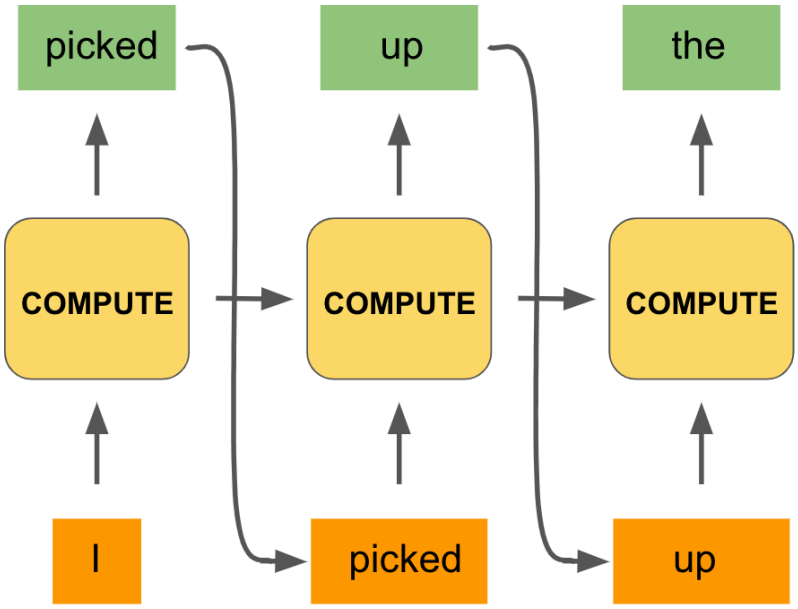

Figure 100: Three recurrent unit states in recurrent neural network

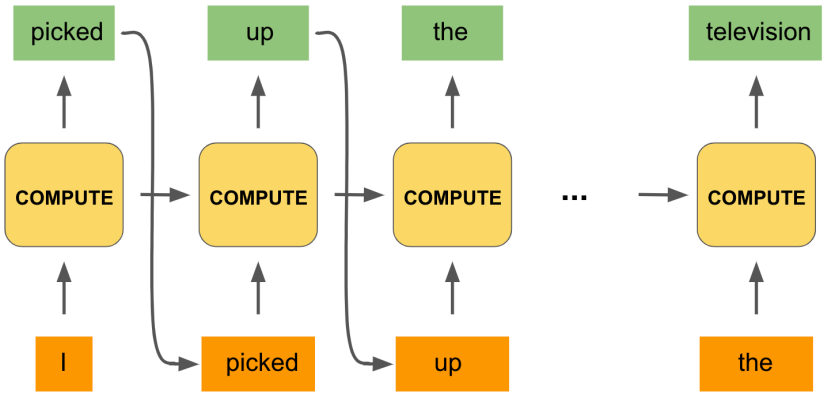

Figure 101: Recurrent neural network unrolled

weights are updated during training via a loss function, as we did with the previous neural network families we studied.

And with that, we have endowed a neural network with sequential memory!

Recurrent Unit 2.0

Well, almost. It turns out that in practice our vanilla recurrent networks suffer from a few pretty big problems. When we are training our recurrent network, we have to use some variant of a backpropagation algorithm as we did for our previous neural network architectures.

That involves calculating a loss function for our model outputs and then computing a gradient of that loss function with respect to our weights. It turns out that when we compute our gradients through the timesteps, our network may suffer from this big problem called **vanishing gradients**.

As an intuition for this problem, imagine the situation of applying a tanh nonlinearity to an input several times (Figure 102).

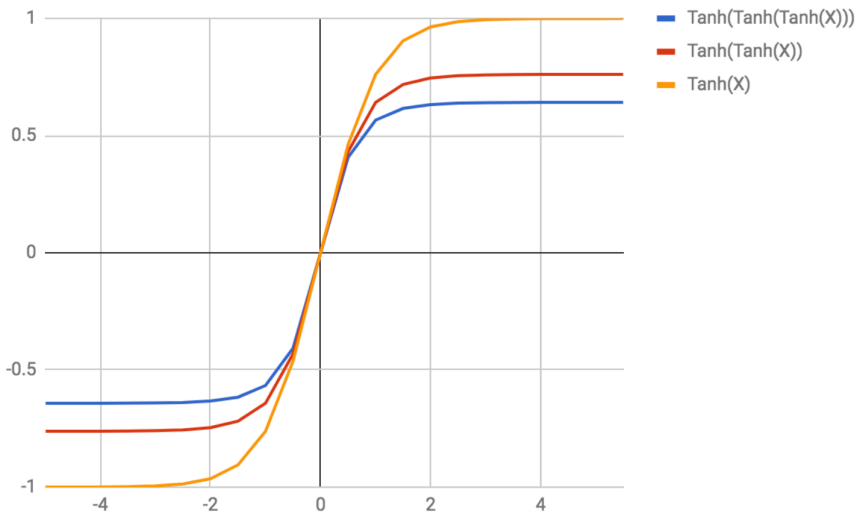

Figure 102: Vanishing gradient problem in recurrent neural network

Notice that the more times we apply the tanh function, the more flat the gradient of the function gets for a given input. Applying a tanh repeatedly is the analogue of performing a computation in our recurrent network for some number of timesteps.

As we continue to apply the tanh, our gradient is quickly going to 0. If our gradient is 0, then our network weights won't be updated during backpropagation,

and our learning process will be ineffective!

The way this manifests itself is that our network may not be able to have a memory of words it was inputted several steps back. This is clearly a problem.

To combat this issue, researchers have developed a number of alternative compute units that perform more sophisticated operations at each timestep. We won't go into their mathematical details, but a few well-known and famous examples include the **long short-term memory (LSTM) unit** and the **gated recurrent unit (GRU)**. These units are specifically better at avoiding the vanishing gradient problem and therefore allowing for more efficient learning of longer term dependencies.

Final Thoughts

To conclude our whirlwind tour of recurrent networks, let's leave with a few examples of problem spaces where they have been applied very successfully.

Recurrent networks have been applied to a variety of tasks including speech recognition (given an audio input, generate the associated text), generating image descriptions, and machine translation (developing algorithms for translating from one language to another). These are just a few examples, but in all cases, the recurrent network has enabled unprecedented performance on these problems.

Test Your Knowledge

Recurrent Network Advantages

Alternative Recurrent Units