

# Machine Learning for Human Beings

Build Machine Learning Algorithms with Python

**Mohit Deshpande**

# Table of Contents

## [Table of Contents](#)

<a href="#">Overview of Machine Learning</a>	4
<a href="#">Good Old-Fashioned Artificial Intelligence (GOFAI)</a>	4
<a href="#">The Problem with GOFAI</a>	6
<a href="#">Supervised vs Unsupervised Learning</a>	8
<a href="#">BONUS: Neural Networks</a>	9
<a href="#">Building Blocks - Data Science and Linear Regression</a>	12
<a href="#">Data Extraction and Exploration</a>	12
<a href="#">Linear Regression</a>	14
<a href="#">Linear Regression Code</a>	16
<a href="#">Correlation Coefficient</a>	18
<a href="#">Text Classification Tutorial with Naive Bayes</a>	21
<a href="#">Bayes Theorem</a>	21
<a href="#">Deriving Naive Bayes</a>	23
<a href="#">Naive Bayes Assumption</a>	24
<a href="#">Numerical Stability</a>	25
<a href="#">Dataset</a>	25
<a href="#">Naive Bayes Code</a>	25
<a href="#">Data Clustering with K-Means</a>	30
<a href="#">Clustering</a>	30
<a href="#">Choosing the Number of Clusters</a>	31
<a href="#">K-Means Clustering</a>	31
<a href="#">K-Means Clustering Code</a>	31
<a href="#">Poor Initialization of Cluster Centers</a>	36
<a href="#">K-Means Convergence</a>	37
<a href="#">Clustering with Gaussian Mixture Models</a>	38
<a href="#">Gaussian Distribution</a>	38
<a href="#">Multivariate Gaussians</a>	39
<a href="#">Gaussian Mixture Models</a>	41
<a href="#">Expectation-Maximization</a>	44
<a href="#">Applying GMMs</a>	45
<a href="#">Face Recognition with Eigenfaces</a>	49
<a href="#">Face Recognition</a>	49

<a href="#">Dimensionality Reduction</a>	50
<a href="#">Principal Component Analysis</a>	51
<a href="#">Aside on Face Detection</a>	52
<a href="#">Eigenfaces Code</a>	52
<a href="#">Decision Trees</a>	58
<a href="#">Trees and Binary Trees</a>	58
<a href="#">Decision Trees</a>	60
<a href="#">Classification and Regression Trees (CART) Framework</a>	62
<a href="#">Decision Tree Code</a>	64
<a href="#">Dimensionality Reduction</a>	69
<a href="#">Handwritten Digits: The MNIST Dataset</a>	69
<a href="#">Dimensionality Reduction</a>	70
<a href="#">Principal Component Analysis</a>	71
<a href="#">Linear Discriminant Analysis</a>	72
<a href="#">t-Distributed Stochastic Neighbor Embedding (t-SNE)</a>	73
<a href="#">Dimensionality Reduction Visualizations</a>	74
<a href="#">Classification with Support Vector Machines</a>	81
<a href="#">Perceptron Review</a>	81
<a href="#">Support Vector Machines</a>	84
<a href="#">SVMs for Logic Gates</a>	87
<a href="#">SVMs for Linearly Inseparable Problems</a>	90
<a href="#">The Kernel Trick</a>	91
<a href="#">SVM for The Iris Dataset</a>	92
<a href="#">Reinforcement Learning: Balancing Act</a>	95
<a href="#">Games</a>	95
<a href="#">Value Iteration</a>	96
<a href="#">Q-Learning</a>	100
<a href="#">Q-Learning Agent for CartPole</a>	102

## Overview of Machine Learning

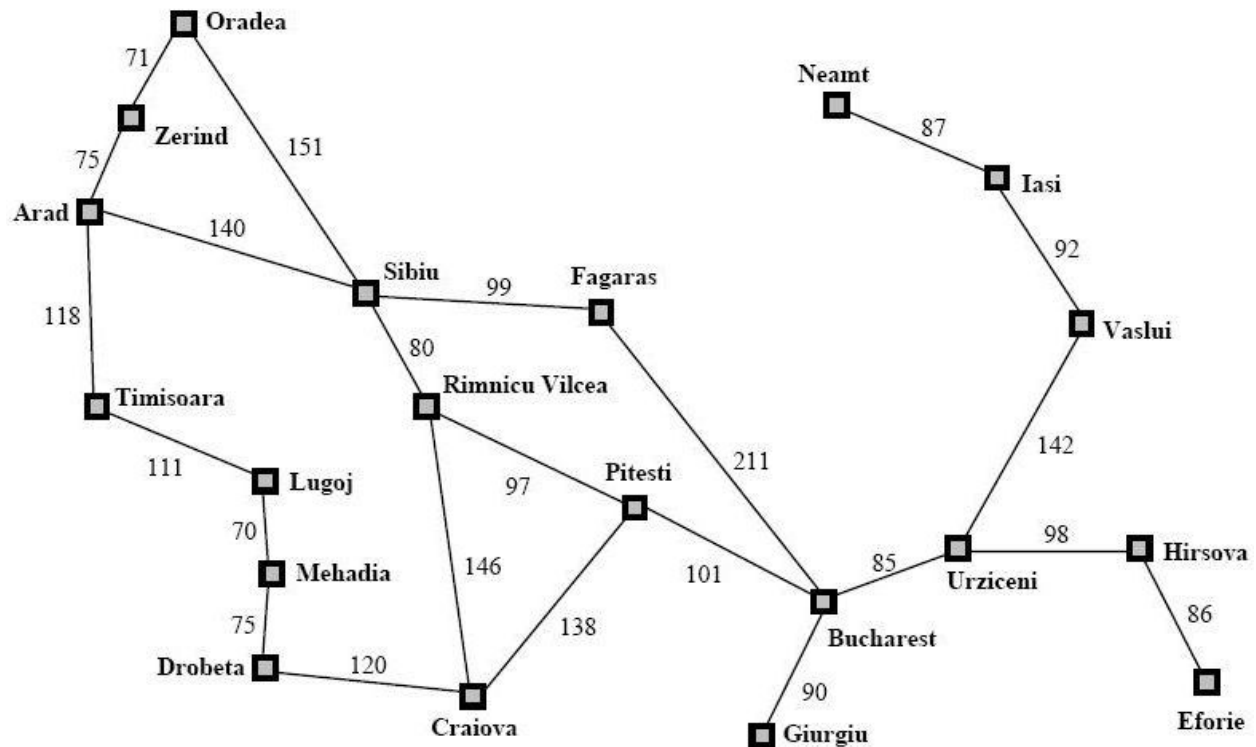
Computers are incredibly dumb. They have to be told explicitly what to do in the form of programs. Programs have to account for every possible branch of logic and are specific to the task at hand. If there are any anomalies in the set of inputs, a program might not produce the right output or just crash outright! In many cases, this behavior is acceptable since programs are quite specialized for a particular purpose. However, there many more cases where we want our program to be more general and *learn* how to accomplish the task after we have given it many examples of how to do the task. This is only a slice of what the field of artificial intelligence is trying to solve.

**Artificial intelligence (AI)** is a very broad field in computer science whose goal is to build an entity that can perceive and reason about the world as well as, or better than, humans can. **Machine Learning** is the largest subfield in AI and tries to move away from this explicit programming of machines. Instead of hard-coding all of our computer's actions, we provide our computers with many examples of what we want, and the computer will *learn* what to do when we give it new examples it has never seen before.

To introduce the field, we'll briefly touch on a wide variety of different topics to get a better feel for what they are and how they work. Later, we'll delve into more detail to discuss really *how and why* these techniques, algorithms, and architectures work. First we will discuss, in moderate detail, how AI solved problems before the advent of machine learning. Although those algorithms are still used today for other tasks, they have been replaced by more powerful algorithms, and it will be the only time we discuss classical AI. This should set the stage for moving on to more modern approaches of machine learning. We'll briefly discuss two of the most common types of machine learning: **supervised** and **unsupervised** learning. Finally, there will be a bonus section toward the end briefly introducing **neural networks**, since they are such a hot topic. Later, we will pick apart a neural network to see *exactly* how they work and write the code to construct, train, and test it from scratch.

### Good Old-Fashioned Artificial Intelligence (GOFAI)

Artificial intelligence is certainly not a new development in computer science. The field has actually existed for many decades (since the 1950s!), but, back then, people in that field were solving problems in a different way than the machine learning algorithms used today: their approaches centered around building **search algorithms**. And this is what we call **good old-fashioned artificial intelligence (GOFAI)**.



(credit <http://athena.ecs.csus.edu/~gordonvs/215/215homework.html>)

To illustrate this approach, let's consider a real-world problem: *what's the best route to go Bucharest from Arad?* This is *the* classic problem that is used to illustrate search algorithms in almost every university course on AI so we'll also discuss it here. Take a look at the chart above, more formally called a discrete mathematics **graph**. Each city (or **node**) is connected to others with an **edge**. On any edge, we see a number that represents the **cost** of traveling to a city from a particular city. We can think of this as the distance between them, though the cost can be more generic than that. For our purposes, how this cost value is calculated isn't important. The goal is to find a list of cities that we can traverse through to find the best, i.e., lowest cost, route to Bucharest.

Suppose we were transported back before modern approaches existed, and we were tasked with writing a program to do this. We won't cover the variety of search algorithms that exist, but let's see how we can frame the problem so that we can use search.

(In our case of path planning, we can see how this can be considered a search problem, but it won't be this obvious for other kinds of problems. If we frame the problem using the components that we'll discuss, then we can use an off-the-shelf search algorithm to solve our problem.)

From the phrasing of the problem and the map above, we know all of the pieces of information we need to frame our search problem. First of all, the problem statement says that we're starting in Arad and ending up in Bucharest. These two pieces of information are the **start state and goal state**. In reality, we may have multiple goal states and thus generalize the notion of goal states to a **goal test function**, whose job is to test if we're in a goal state or not. Suppose our goal was to arrive at an airport, and Bucharest, Zerind, and Iasi all had airports. Then arriving at any of these cities would solve our problem!

The other two pieces of information we need to completely frame our search problem are taken from the graph above. Notice that we have a finite, enumerable number of cities. This is called our **state space**. In our case, the city represents the state, but, consider the example of traveling between two cities of different countries. Then perhaps our state would consist of a tuple of city and country. The **state space** is simply all states that we're considering. The other is the **successor function**. The purpose of this function is to figure out, given one state, the other states can I get to and how much will it cost. This information is encoded in the edges of the graph above. For example, starting in Arad, I can go to Sibiu with a cost of 140.

Now that we have our search problem, we can apply a search algorithm to our problem and get a solution! Typically, we use a data structure to store the states that we're considering to visit (called the **fringe**). The data structure, usually a stack, queue, or priority queue, depends on the specific search algorithm. However, all search algorithms follow the same basic steps:

1. Start at the start state and put it into the data structure
2. Remove a state from the data structure
3. Check if it is the goal state. If so, then return!
4. Figure out what other states we can get to from this state using the successor function.
5. Put all of those successor states into the data structure
6. Go to 2

The details of the search algorithm determine how we remove the state from the data structure. For example, if we were doing depth-first search, we would use a stack and pop off the top of the stack. Besides the details of the search algorithm, we can use this framework to return solutions.

To sum up, GOF AI is centered around this notion that we frame a problem using four elements: start state, state space, successor function, and goal states/goal test function, and apply a search algorithm to return the solution.

## The Problem with GOF AI

One of the big issues that people had with GOFAI is that these systems weren't really *learning* anything. They were *searching* through a potentially huge state space for the right answer. If we gave a different problem to the same system, it would treat that problem as independent of all of the other problems that system has solved. This isn't *learning*!



(credit <https://www.pexels.com/search/apple/>)

To illustrate this point, think about a child learning what an apple is for the first time. Initially, a child has no idea what this fruit is, but someone tells the child that this is an apple. If you gave a different apple to the child, the child would still tell you that this fruit is an apple. An amazing thing happens: the child *generalizes* the concept of an apple. In other words, you don't have to show the child every single apple in the world for the child to learn what an apple is; you give the child a few examples and he/she will learn what the abstract concept of an apple is. *This is exactly what GOFAI is lacking.*

Along comes machine learning, backed by statistics, calculus, linear algebra, and mathematical optimization to save the day! Instead of *searching* through a state space, we give machine learning algorithms many examples, and they can learn to generalize. Machine learning algorithms and approaches are characterized by their **parameters** or **weights**. These parameters affect the result of the machine learning algorithm. As we give the algorithm more and more examples, the algorithms tune these parameters based on the previous value of the parameters and the examples they're currently looking at. This is an iterative

process where we keep feeding in examples and the algorithm keeps tuning parameters until we achieve good results.

## Supervised vs Unsupervised Learning

The two largest categories of machine learning are **supervised** and **unsupervised** learning. Think back to the example of teaching the child what an apple is. This is a prime example of supervised learning. When teaching the child, we give an example of an apple along with the correct answer: apple. When using supervised learning, we feed the machine learning algorithm an example and the correct answer. This is why we call it supervised learning. Here are just some of the following tasks that supervised learning can help us with.

- Image classification: what is in an image, e.g., a cat.
- Image captioning: given an image, generate a novel caption that describes it.
- Image retrieval: given an image sentence description, fetch the image that most closely represents that description.
- Sentiment classification: what is the sentiment behind this phrase?
- Machine translation: translating between languages, e.g., English and French.
- Curve fitting or predicting trends in business
- ...

From the list above, supervised learning has many different applications. However, it can be difficult producing labeled datasets. If our problem is very domain-specific, we usually have to hire people to annotate our dataset, which may be costly or time-consuming. In unsupervised learning, we don't give the correct answer (often we don't know it!). Instead, we rely heavily on statistics and the distribution of the input data. Here are some examples of unsupervised learning.

- Clustering: given a set of data, we want to find grouping of similar data points.
- New example generation: given a set of training data, create a new data point *similar* to the training data but not actually in the training data
- Anomaly detection: find the anomalous data point in a set of data

Compared to supervised learning, we don't need to spend time and money annotating a dataset. However, the objective of unsupervised learning seems a bit more difficult than supervised learning: we want insights from our data without giving examples of "correct" results.

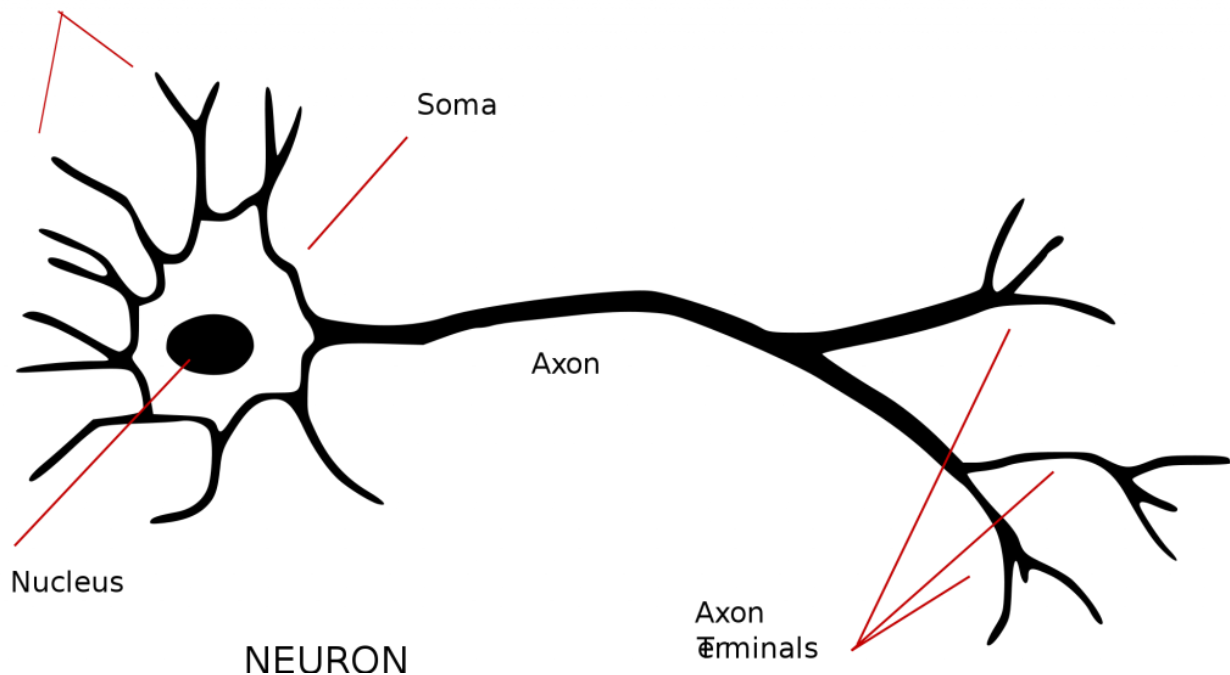
To summarize, supervised learning requires each example to have a correct answer while unsupervised learning can take a set of unlabeled examples and learn from it. In the future, we'll learn about various supervised and unsupervised learning algorithms and techniques.



From the examples of supervised and unsupervised classification, we can grasp the essence of machine learning: *learning by example*. Instead of having to hard-code our programs, we give these algorithms examples of what we want, and they generalize. This is an amazing thing for a computer to do! And we're going to see exactly how this generalization works across different types of machine learning, modalities (like images, text, audio, etc), and architectures!

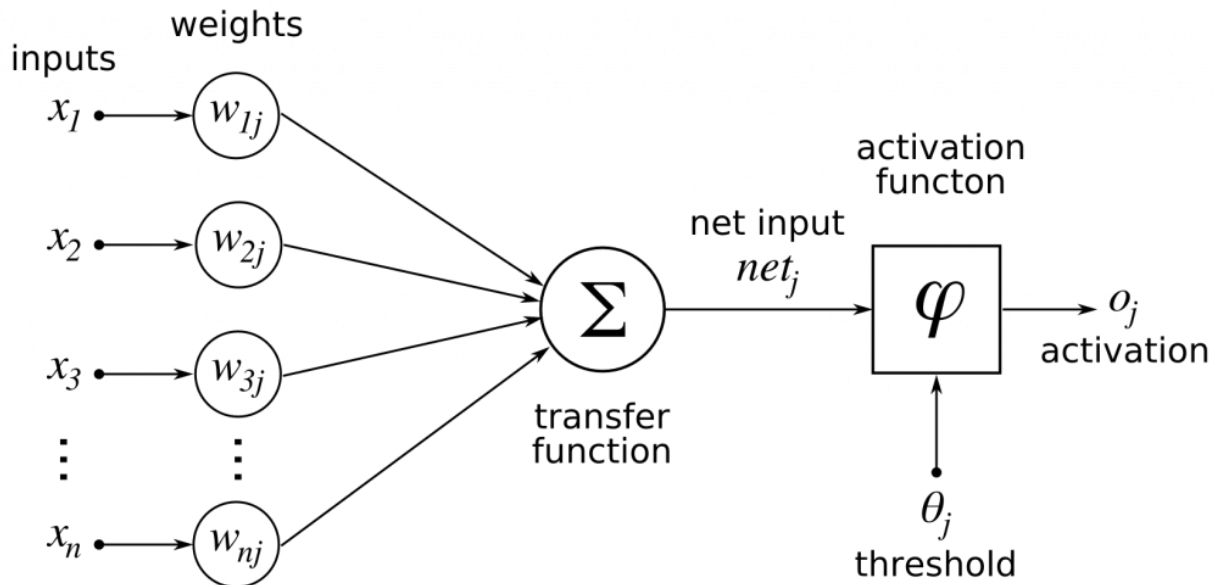
## BONUS: Neural Networks

As a bonus section, let's briefly take a look at neural networks. They were invented many decades ago but have recently come to the forefront of machine learning since we have developed new techniques (and even specialized hardware!) to train massive networks. Dendrites



(credit [https://commons.wikimedia.org/wiki/File:Neuron\\_-\\_annotated.svg](https://commons.wikimedia.org/wiki/File:Neuron_-_annotated.svg))

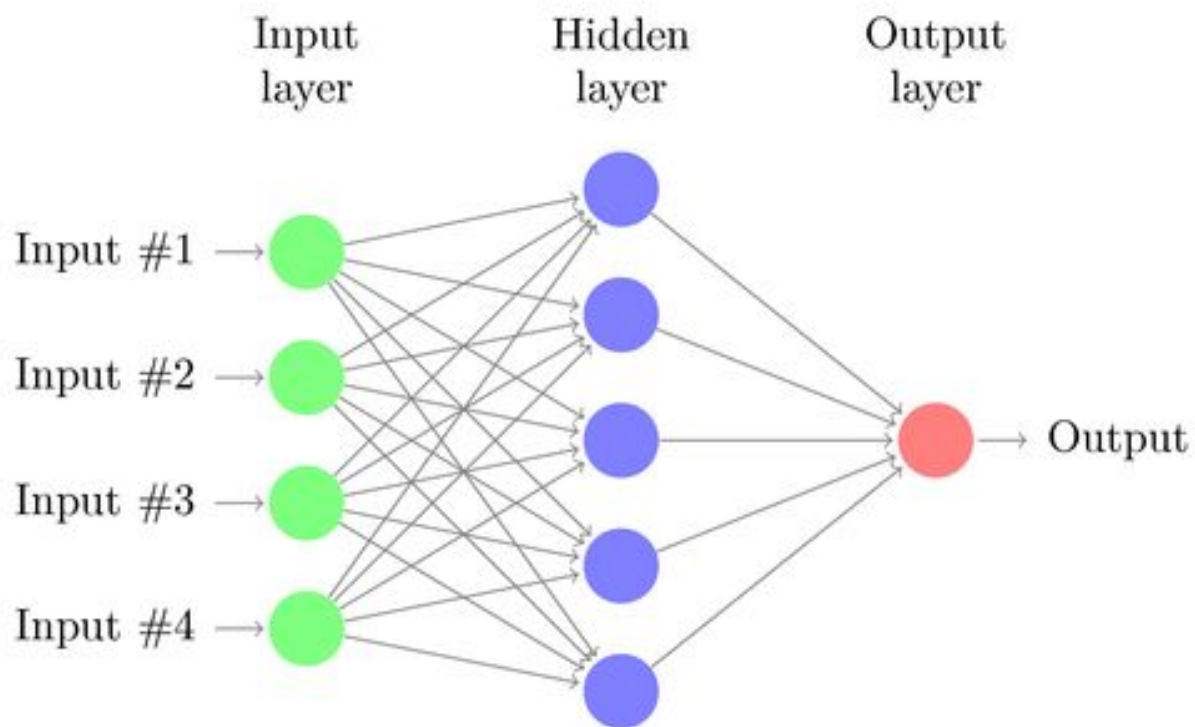
The inspiration for neural networks comes from biological neurons in our brains. Above is a very simplified version of one. Each neuron takes some number of inputs and produces a single output that may become inputs to other neurons. A biological neuron receives inputs from the **dendrites**. They touch the cell body, or **soma**, but, they are connected to other neurons by a little gap called the **synapse** that weights how strongly this input should be considered. The neuron weights each of its inputs and processes them all together. The actual processing is done in the nucleus. After that processing, the output is sent through the axon so that it may become input to other neurons, and the process repeats.



(credit <https://commons.wikimedia.org/wiki/File:Rosenblattperceptron.png>)

We can take this biological model and create an approximate, simplified mathematical model from it. The dendrites are the inputs  $x_1, \dots, x_n$ . We model the synapse as the weights  $w_{1j}, \dots, w_{nj}$ . Conventionally, the first index refers to the input neuron and the second index refers to the output neuron. In this example, just ignore  $j$  (it means the output neuron is the  $j$ th neuron in the next layer, but we'll discuss architecture very soon). We take each input and multiply it with its corresponding weight and add them all up. We're taking a weighted sum, in other words. Then we apply a non-linear activation function to produce an output. (Later we will see why this function has to be non-linear.) This output  $o_j$ , then becomes the input to any number of subsequent neurons.

This is the fundamental model of a single neuron. Surprisingly, a single neuron can still solve a number of different problems. But the recent success of neural networks has come from organizing many of these neurons in connected layers.



(credit <http://www.texample.net/tikz/examples/neural-network/>)

In the above figure, we have a 2-layer neural network (conventionally, we don't count the input layer). Each input contributes to each neuron in the **hidden layer**. Finally, each neuron of the hidden layer goes into the final **output layer**. This is a very small neural network by today's standards: many modern architectures are many layers deep and have thousands of neurons per layer.

We've seen what a neural network is, but what can they do? Turns out, they can do just about anything! We can use them for both unsupervised and supervised learning. There are special variants of these that work really well for different kinds of inputs like images, text, audio, etc.

We covered a very top-level overview of what neural networks are, but we will discuss them in much more detail. We will discuss different architectures, domain-specific variants, and training algorithms later.

# Building Blocks - Data Science and Linear Regression

"Data science" or "Big data analyst" is a phrase that has been tossed around since the advent of Big Data. But what is it, really? Well imagine working for a retail company. One of the questions you may be asked to answer is "how many chips should we stock up for this month?" It seems like a simple question that makes sense to ask. But there are a lot of steps involved in answering that single question. We need to collect the data, clean/prep the data, figure out which features we need to answer the question, determine the appropriate machine learning algorithm to use, get the results, and, finally, write a report that outlines the results of the analysis. And that's still a fairly top-level understanding of what might be needed to complete this task! Some portion of that might take a long time, e.g., we may spend a few weeks collecting and cleaning/prepping data!

We're going to go through an example question similar to what a data scientist may be expected to answer. We'll also cover the topic of linear regression, a fundamental machine learning algorithm.

Download the dataset and code in the ZIP file [here](#).

Data science hinges on questions whose answer lie in the data itself. We're going to go through an example of answering one of these questions using data from Facebook. The University of California Irvine (UCI) has a ton of free-to-use machine learning datasets [here](#), and we're going to use their Facebook metrics dataset to answer some questions.

Let's start off with a question whose answer you can probably intuit: "When a post is shared more often, do more people tend to like that post as well?" You can probably use your intuition to figure out that the more a post is shared, the more likely it is to be liked or commented on. But intuition doesn't always pan out! The key word in "data science" is "data"! We need evidence to support our intuition or our hypothesis.

## Data Extraction and Exploration

The first thing we should do is spend some time to explore our dataset. This includes writing code to read in our data and to create a visualization. Since the data are in a CSV file, we can use the built-in csv module to extract them. If we're dealing with large datasets, it's quicker to pre-allocate our numpy arrays whenever possible. This is why we count the number of rows in our csv file, which should be exactly 500. (We subtract one because of the header row.) Then we can pre-allocate our X and Y arrays. We use Python's csv module to open the file, skip the header row, and load it into the numpy arrays.

A few minor points to note: the enumerate function will produce a tuple with the index and value at that index if given a list or generator. This is particularly useful for loading into pre-allocated arrays. Another note is that some of data might not exist. In this case, we have to decide what we should do. There are many valid options: forget/remove the nonexistent data, replace it with a default value, interpolate (for time-series data), etc. In our case, we're just substituting a reasonable value (zero).

```
import numpy as np
import matplotlib.pyplot as plt
import csv

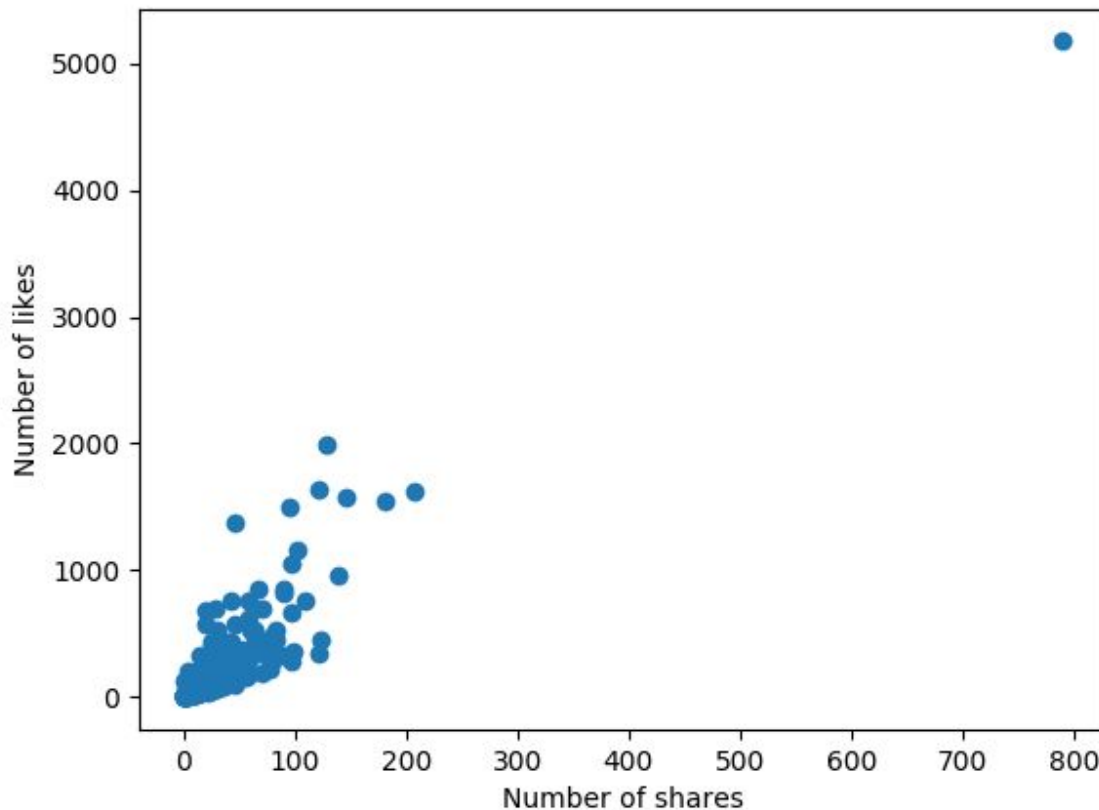
def load_dataset():
    num_rows = sum(1 for line in open('dataset_Facebook.csv')) - 1
    X = np.zeros((num_rows, 1))
    y = np.zeros((num_rows, 1))
    with open('dataset_Facebook.csv') as f:
        reader = csv.DictReader(f, delimiter=';')
        next(reader, None)
        for i, row in enumerate(reader):
            X[i] = int(row['share']) if len(row['share']) > 0 else 0
            y[i] = int(row['like']) if len(row['like']) > 0 else 0
    return X, y
```

Now that our dataset is loaded, we can use matplotlib to create a scatter plot. Remember to label the axes!

```
def visualize_dataset(X, y):
    plt.xlabel('Number of shares')
    plt.ylabel('Number of likes')
    plt.scatter(X, y)
    plt.show()

if __name__ == '__main__':
    X, y = load_dataset()
    visualize_dataset(X, y)
```

Here is the resulting plot:



From this plot alone, we notice a few things. There seems to be an outlier with around 800 shares and a little over 5000 likes. This might be an interesting post to investigate. A second point to notice is that most of our data are between 0 - 200 shares and 0 - 2000 likes, quite densely actually. Remember that 500 data points are being shown! Another interesting point we notice is the right-upwards trend of our data. This seems to fit our intuition: the more *shares* a post receives, the most *likes* it tends to have!

Normally, we would spend more time exploring our data, but our question can be answered using these data. Now that we see our intuition fits our data, we need to provide quantitative numbers that show this relationship between the number of shares and number of likes. The best way to do this, in our case, is using linear regression.

## Linear Regression

**Linear regression** is a machine learning algorithm used find *linear* relationships between two sets of data. The crux of linear regression is that it only works when our data is somewhat linear, which fits our data. There are metrics that we'll use to see exactly how linear our data are.

With linear regression, we're trying to estimate the parameters of a line:

$$[\hat{y} = wx + b]$$

Following suit with neural networks, we use  $w$  instead of  $m$ . Another reason for this is because we may have multivariate data where the input is a vector. Then we generalize to a dot product. Based on this equation, we have two parameters:  $w$  and  $b$ .

We want to compute the line-of-best-fit, but what does "best" mean? Well we need to define a metric that we use to measure "best" because we need a way to determine if one set of parameter values is better than another. To do this, we define a **cost function**. Intuitively, this is just a measure of how good our parameters are, given our data. When our cost function is minimal, that means that our parameters are optimal!

$$\begin{aligned} C(w, b) &= \frac{1}{2} \sum_{i=0}^N (y_i - \hat{y}_i)^2 \\ &= \frac{1}{2} \sum_{i=0}^N (y_i - (wx_i + b))^2 \\ &= \frac{1}{2} \sum_{i=0}^N (y_i - wx_i - b)^2 \end{aligned}$$

(where  $N$  is the number of training examples and  $y$  is the true value)

Let's take a closer look at this cost function intuitively. It is measuring the sum of the squared error of all of our training examples. The reason we square is because we don't really care if the error is positive or negative: an error is an error! (More precisely, the reason we use a square and not an absolute value is because it is differentiable: the derivative of a squared function is nicer than the derivative of an absolute value function.)

Now that we know a bit about the cost function. How do we use it? The optimal parameters are found when the cost is at a minimum. So we need to perform some optimization! To find the optimal value of  $w$ , we take the partial derivative of  $C$  with respect to  $w$ , set it equal to zero, and solve for  $w$ ! We do the same thing for  $b$ .

We'll skip the partial derivatives and derivation for now, but we can use some statistics to rearrange the solutions into a closed-form:

$$\begin{aligned} w &= \frac{\sum_{i=0}^N (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=0}^N (x_i - \bar{x})^2} \\ b &= \bar{y} - w\bar{x} \end{aligned}$$

(where the bars over the variables represent average/mean)

Now we can figure out the slope and y-intercept from the data itself!

## Linear Regression Code

Now that we have the math figured out, we can implement this in code. Let's create a new class with parameters  $w$  and  $b$ .

```
class LinearRegression(object):  
    """Implements linear regression"""  
    def __init__(self):  
        self.w = 0  
        self.b = 0
```

Now we can write a fit function that computes both of these using the closed-forms we discussed.

```
def fit(self, X, y):  
    mean_x = X.mean()  
    mean_y = y.mean()  
    errors_x = X - mean_x  
    errors_y = y - mean_y  
    errors_product_xy = np.sum(np.multiply(errors_x, errors_y))  
    squared_errors_x = np.sum(errors_x ** 2)  
  
    self.w = errors_product_xy / squared_errors_x  
    self.b = mean_y - self.w * mean_x
```

We are making use of numpy's vectorized operations to speed up our computation. We compute the means of our inputs and outputs using a quick numpy function. We can compute the errors in a vectorized fashion as well. In numpy, when we perform addition, subtraction, multiplication, or division of an array by a scalar, numpy will apply that operation to all values in the array. For example, when we subtract the scalar `mean_x` from the vector `X`, we're actually taking each element of `X` and subtracting `mean_x` from it. Same goes for the vector `y`. When we compute the errors, we have to tell numpy to do an *element-wise* multiplication of the two vectors, not a vector-vector multiplication. (Mathematically, this is called the **Hadamard product**). This takes the first element of `errors_x` and multiplies it by the first element of `errors_y` and so on to produce a new vector. Then we take the sum of all of the elements in that vector. This produces the numerator of the expression to compute the slope.

To compute the denominator, we can simply take the square of the errors and take the sum. Numpy can also apply exponents to each element of an array. Finally, we can compute the slope and y-intercept!



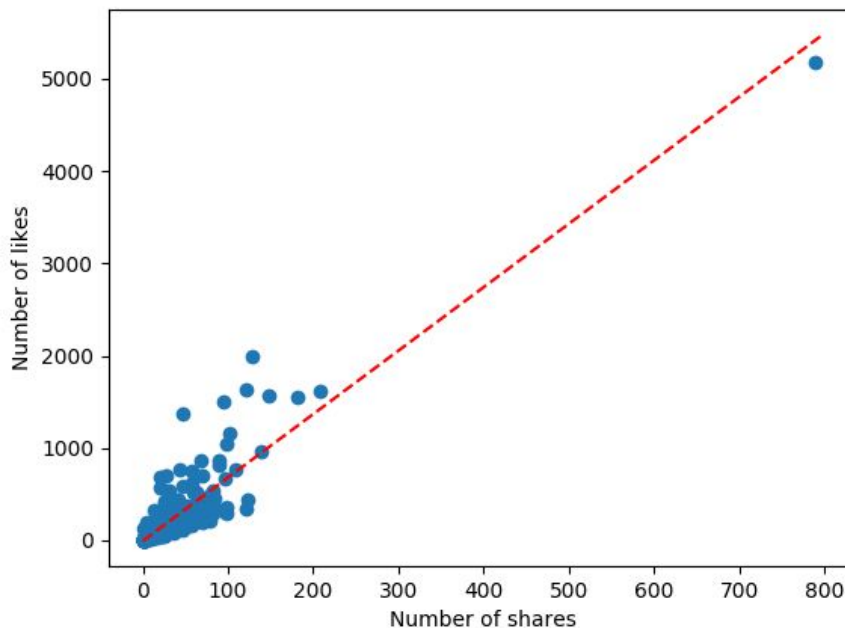
Before we visualize our answer, let's make another convenience function that takes in inputs and applies our equation to return predictions.

```
def predict(self, X):  
    return self.w * X + self.b
```

Now that we have the slope and y-intercept, we can visualize this along with the scatter plot of our data to see if our solution is correct.

```
def visualize_solution(X, y, lin_reg):  
    plt.xlabel('Number of shares')  
    plt.ylabel('Number of likes')  
    plt.scatter(X, y)  
  
    x = np.arange(0, 800)  
    y = lin_reg.predict(x)  
    plt.plot(x, y, 'r--')  
  
    plt.show()  
  
if __name__ == '__main__':  
    X, y = load_dataset()  
    lin_reg = LinearRegression()  
    lin_reg.fit(X, y)  
  
    visualize_solution(X, y, lin_reg)
```

Here, we simply create some x values and apply our line equation to them to produce the red, dashed line. The result is shown below:



We can see that our red, prediction line-of-best-fit fits most of our data! However, there's another statistical measure that we can compute to give us more information about our line. In particular, it will tell us how linear our data are and the correlation.

## Correlation Coefficient

The **Pearson correlation coefficient** is a number that represents how linear our data are *and* the relationship between the input and output, namely between the number of shares and the number of likes.

We won't cover the exact derivation of it, but we can represent it quite simply in statistical terms:

$$\rho = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

(where  $\sigma_x$  and  $\sigma_y$  are the standard deviations of X and Y and  $\sigma_{xy}$  is the covariance of X and Y)

We can compute the covariance  $\sigma_{xy}$  using the following equation

$$\sigma_{xy} = \frac{1}{N} \sum_{i=0}^N (x_i - \bar{x})(y_i - \bar{y})$$

The correlation coefficient is a value from -1 to 1 that represents two things: the linearity of our data and the correlation between X and Y. A value close to -1 or 1 means that our data is very linear. A value close to 0 means that our data is not linear at all. If the value is positive, then *high values* of X tend to produce *high values* of Y and vice-versa. If the value is negative, *high values* of X tend to produce *low values* of Y and vice-versa. This is a very powerful metric

and essential to answering our question: "are posts that are shared more often receive more likes?"

To compute the value, let's add another property.

```
class LinearRegression(object):
    """Implements linear regression"""
    def __init__(self):
        self.w = 0
        self.b = 0
        self.rho = 0
```

We can compute this value in the fit function when we have the inputs and outputs.

```
def fit(self, X, y):
    mean_x = X.mean()
    mean_y = y.mean()
    errors_x = X - mean_x
    errors_y = y - mean_y
    errors_product_xy = np.sum(np.multiply(errors_x, errors_y))
    squared_errors_x = np.sum(errors_x ** 2)

    self.w = errors_product_xy / squared_errors_x
    self.b = mean_y - self.w * mean_x

    N = len(X)
    std_x = X.std()
    std_y = y.std()
    cov = errors_product_xy / N
    self.rho = cov / (std_x * std_y)
```

Now we can add some code in our visualization to show this value in the legend of the plot.

```
def visualize_solution(X, y, lin_reg):
    plt.xlabel('Number of shares')
    plt.ylabel('Number of likes')
    plt.scatter(X, y)

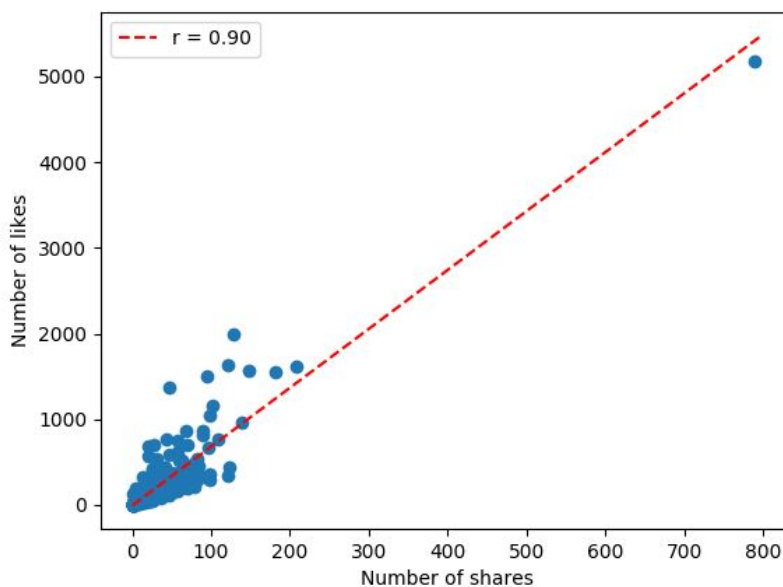
    x = np.arange(0, 800)
    y = lin_reg.predict(x)
    plt.plot(x, y, 'r--', label='r = %.2f' % lin_reg.rho)
    plt.legend()
```

```
plt.show()

if __name__ == '__main__':
    X, y = load_dataset()
    lin_reg = LinearRegression()
    lin_reg.fit(X, y)

    visualize_solution(X, y, lin_reg)
```

Notice the label parameter to the plot function. Finally, we can show our plot:



We notice that our data has a correlation coefficient of +0.9, which is close to 1 and positive. This means our data is really linear and *the more shares a post gets, the more likes it tends to get too!* We've answered our question!

There are many more columns to this data set, and I encourage you to explore all of them using scatter plots, histograms, etc. Try to find hidden correlations! (But remember that *correlation does not imply causation!*)

To summarize, we learned how to answer a data science question using linear regression, which gives us the line-of-best-fit for our data. We use a cost function to mathematically represent what the "best" line is, and we can use optimization to directly solve for the slope and intercept of the line. However, this equation might not be enough. Additionally, we can compute the correlation coefficient to tell us the linearity of our data and the correlation between the inputs and outputs.

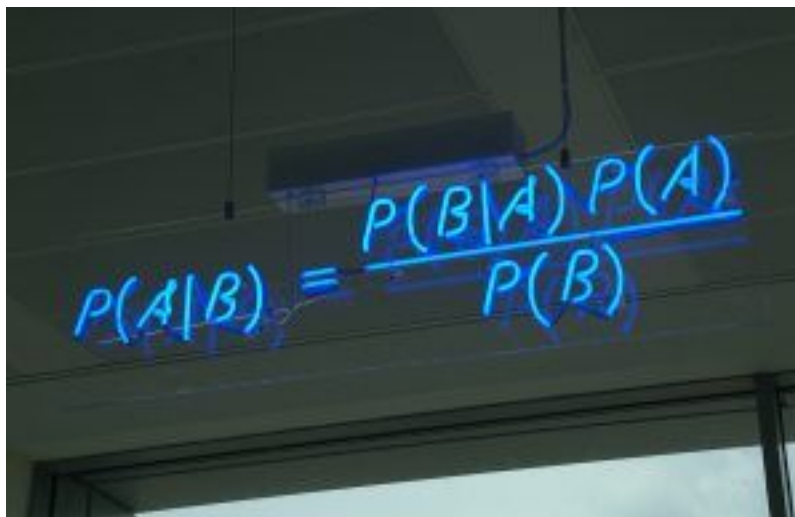
Linear regression is a fundamental machine learning algorithm and essential to data science!

# Text Classification Tutorial with Naive Bayes

The challenge of text classification is to attach labels to bodies of text, e.g., tax document, medical form, etc. based on the text itself. For example, think of your spam folder in your email. How does your email provider know that a particular message is spam or "ham" (not spam)? We'll take a look at one natural language processing technique for text classification called **Naive Bayes**.

Download the full code [here](#).

## Bayes Theorem



(credit:

[https://en.wikipedia.org/wiki/Bayes%27\\_theorem#/media/File:Bayes%27\\_Theorem\\_MMB\\_01.jpg](https://en.wikipedia.org/wiki/Bayes%27_theorem#/media/File:Bayes%27_Theorem_MMB_01.jpg))

Before we derive the algorithm, we need to discuss the fundamental rule that Naive Bayes uses: Bayes Theorem:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

where  $A$  and  $B$  are events and  $p(\cdot)$  is a probability.

Let's take a second to break this down. On the left, we have the probability of an event  $A$  happening given that event  $B$  happens. We say this is equal to the probability of event  $B$  happening given event  $A$  times the probability that event  $A$  happens overall. All of that is divided by the probability that event  $B$  happens overall. An example of this might help shed some light on why this is an ingenious theorem.

The classic example used to illustrate Bayes Theorem involves medical testing. Let's suppose that we were getting tested for the flu. When we get a medical test, there are really 4 cases to consider when we get the results back:

- **True Positive:** The test says we have the flu and we actually have the flu
- **True Negative:** The test says we don't have the flu and we actually don't have the flu
- **False Positive:** The test says we have the flu and we actually *don't* have the flu
- **False Negative:** The test says we *don't* have the flu and we actually *do* have the flu

Suppose we also know some information about the flu and our testing methodology: we know our test can correctly detect that a person has the flu 99.5% of the time (i.e.,  $p(+|Flu) = 0.995$ ) and correctly detect that a person does not have the flu 99.5% of the time (i.e.,  $p(-|No\ Flu) = 0.995$ ). These correspond to the **true positive rate** and **true negative rate**. We also know that this specific type of flu is rare and only affects 1% of people. Given this information, we can compute the probability that any randomly selected person will have this specific type of the flu. Specifically, we want to compute the probability that the person has the specific type of flu, *given* that the person tested positive for it, i.e., event  $A = Flu$  and  $B = +$ . Let's just substitute the problem specifics into Bayes Theorem.

$$p(Flu|+) = \frac{p(+|Flu)p(Flu)}{p(+)}$$

Now let's try to figure out specific values for the quantities on the right-hand side. The first quantity is  $p(+|Flu)$ . This is the probability that someone tests positive given that they have the flu. In other words, this is the true positive rate: the probability that our test can correctly detect that a person has the flu! This number is 99.5% or 0.995. The next quantity in the numerator is  $p(Flu)$ . This is called the **prior probability**. In other words, it is the probability that any random person has the flu. We know from our problem that this number is 1%, or 0.01. Let's substitute in those values in the numerator.

$$p(Flu|+) = \frac{0.995 \cdot 0.01}{p(+)}$$

Now we have to deal with the denominator:  $p(+)$ . This is the probability that our test returns positive overall. We can't quite use the information given in the problem as directly as before however. But first, why do we even need  $p(+)$ ? Recall that probabilities have to be between 0 and 1. Based on the above equation, if we left out the denominator, then we wouldn't have a valid probability!

Anyways, when can our test return positive? Well there are two cases: either our test returns positive and the person actually has the flu (true positive) or our test returns positive and our person does not have the flu (false positive). We can't quite simply sum both of these cases to be the denominator. We have to weight them by their respective probabilities, i.e., the

probability that any person has the flu overall and the probability that any person *does not* have the flu overall. Let's expand the denominator.

$$p(\text{Flu}|+) = \frac{0.995 \cdot 0.01}{p(+|\text{Flu})p(\text{Flu}) + p(+|\text{No Flu})p(\text{No Flu})}$$

Now let's reason about these values.  $p(+|\text{Flu})p(\text{Flu})$  is something we've seen before: it's the numerator! Now let's look at the next quantity:  $p(+|\text{No Flu})p(\text{No Flu})$ . We can compute the first term by taking the complement of the true negative:

$p(+|\text{No Flu}) = 1 - p(-|\text{No Flu}) = 0.005$ . And  $p(\text{No Flu}) = 1 - p(\text{Flu}) = 0.99$  since they are complimentary events. So now we can plug in all of our values and get a result.

$$p(\text{Flu}|+) = \frac{0.995 \cdot 0.01}{0.995 \cdot 0.01 + 0.005 \cdot 0.99} = 0.6678$$

This result is a little surprising! This is saying, despite our test's accuracy, knowing someone tested positive means that there's only a 67% chance that they actually have the flu! Hopefully, this example illustrated how to use Bayes Theorem.

## Deriving Naive Bayes

Now let's convert the Bayes Theorem notation into something slightly more machine learning-oriented.

$$p(H|E) = \frac{p(E|H)p(H)}{p(E)}$$

where  $H$  is the hypothesis and  $E$  is the evidence. Now this might make more sense in the context of text classification: the probability that our hypothesis is correct given the evidence to support it is equal to the probability of observing that evidence given our hypothesis times the prior probability of the hypothesis divided by the probability of observing that evidence overall. Let's break this down again like we did for the original Bayes Theorem, except we'll use the context of the text classification problem we're trying to solve: spam detection. Our hypothesis  $H$  is something like "this text is spam" and the evidence  $E$  is the text of the email. So to restate, we're trying to find the probability that our email is spam given the text in the email. The numerator is then the probability that that we find these words in a spam email times the probability that any email is spam. The denominator is a bit tricky: it's the probability that we observe those words overall.

There's something a bit off with this formulation though: the evidence needs to be represented as multiple pieces of evidence: the words  $w_1, \dots, w_n$ . No problem! We can do that and Bayes Theorem still holds. We can also change hypothesis  $H$  to a class Spam.

$$p(\text{Spam}|w_1, \dots, w_n) = \frac{p(w_1, \dots, w_n|\text{Spam})p(\text{Spam})}{p(w_1, \dots, w_n)}$$

Excellent! We can use a conditional probability formula to expand out the numerator.

$$p(\text{Spam}|w_1, \dots, w_n) = \frac{p(w_1|w_2, \dots, w_n, \text{Spam})p(w_2|w_3, \dots, w_n, \text{Spam}) \dots p(w_{n-1}|w_n, \text{Spam})p(\text{Spam})}{p(w_1, \dots, w_n)}$$

Not only does this look messy, it's also quite messy to compute! Let's think about the first term:  $p(w_1|w_2, \dots, w_n, \text{Spam})$ . This is the probability of finding the first word, given all of the other words and given that the email is spam. This is really difficult to compute if we have a lot of words!

## Naive Bayes Assumption

To help us with that equation, we can make an assumption called the **Naive Bayes assumption** to help us with the math, and eventually the code. **The assumption is that each word is independent of all other words.** *In reality, this is not true!* Knowing what words come before/after do influence the next/previous word! However, making this assumption greatly simplifies the math and, in practice, works well! This assumption is why this technique is called *Naive Bayes*. So after making that assumption, we can break down the numerator into the following.

$$p(\text{Spam}|w_1, \dots, w_n) = \frac{p(w_1|\text{Spam})p(w_2|\text{Spam}) \dots p(w_n|\text{Spam})p(\text{Spam})}{p(w_1, \dots, w_n)}$$

This looks better! Now we can interpret a term  $p(w_1|\text{Spam})$  to mean the probability of finding word  $w_1$  in a spam email. We can use a notational shorthand to symbolize product ( $\prod$ ).

$$p(\text{Spam}|w_1, \dots, w_n) = \frac{p(\text{Spam}) \prod_{i=1}^n p(w_i|\text{Spam})}{p(w_1, \dots, w_n)}$$

This is the Naive Bayes formulation! This returns the probability that an email message is spam given the words in that email. For text classification, however, we need an actual label, not a probability, so we simply say that an email is spam if  $p(\text{Spam}|w_1, \dots, w_n)$  is greater than 50%. If not, then it is not spam. In other words, we choose "spam" or "ham" based on which one of these two classes has the higher probability! Actually, we don't need probabilities at all. We can forget about the denominator since its only purpose is to scale the numerator.

$$p(\text{Spam}|w_1, \dots, w_n) \propto p(\text{Spam}) \prod_{i=1}^n p(w_i|\text{Spam})$$

(where  $\propto$  signifies proportional to) That's one extra thing we don't have to compute! In this instance, we pick whichever class has the higher **score** since this is not a true probability anymore.



## Numerical Stability

There's one extra thing we're going to do to help us with **numerical stability**. If we look at the numerator, we see we're multiplying many probabilities together. If we do that, we could end up with *really* small numbers, and our computer might round down to zero! To prevent this, we're going to look at the **log probability** by taking the log of each side. Using some properties of logarithms, we can manipulate our Naive Bayes formulation.

$$\begin{aligned}\log p(\text{Spam}|w_1, \dots, w_n) &\propto \log p(\text{Spam}) \prod_{i=1}^n p(w_i|\text{Spam}) \\ \log p(\text{Spam}|w_1, \dots, w_n) &\propto \log p(\text{Spam}) + \log \prod_{i=1}^n p(w_i|\text{Spam}) \\ \log p(\text{Spam}|w_1, \dots, w_n) &\propto \log p(\text{Spam}) + \sum_{i=1}^n \log p(w_i|\text{Spam})\end{aligned}$$

Now we're dealing with *additions* of log probabilities instead of *multiplying* many probabilities together! Since log has really nice properties (monotonicity being the key one), we can still take the highest score to be our prediction, i.e., we don't have to "undo" the log!

## Dataset

We'll be using the Enron email dataset for our training data. This is *real* email data from the Enron Corporation after the company collapsed. Before starting, download all of the numbered folders, i.e., enron1, enron2, etc., [here](#) and put all of them in a top-level folder simply called enron. Put this enron folder in the same directory as your source code so we can find the dataset!

**A WORD OF WARNING!** Since this dataset is a real dataset of emails, it contains real spam messages. Your anti-virus may prune some these emails because they are spam. Let your anti-virus prune as many as it wants. This will not affect our code as long as there are some spam and ham messages still there!

## Naive Bayes Code

Here is the dataset-loading code:

```
import os
import re
import string
import math

DATA_DIR = 'enron'
target_names = ['ham', 'spam']

def get_data(DATA_DIR):
```

```

subfolders = ['enron%d' % i for i in range(1,7)]

data = []
target = []
for subfolder in subfolders:
    # spam
    spam_files = os.listdir(os.path.join(DATA_DIR, subfolder,
'spam'))
    for spam_file in spam_files:
        with open(os.path.join(DATA_DIR, subfolder, 'spam',
spam_file), encoding="latin-1") as f:
            data.append(f.read())
            target.append(1)

    # ham
    ham_files = os.listdir(os.path.join(DATA_DIR, subfolder,
'ham'))
    for ham_file in ham_files:
        with open(os.path.join(DATA_DIR, subfolder, 'ham',
ham_file), encoding="latin-1") as f:
            data.append(f.read())
            target.append(0)

return data, target

```

This will produce two lists: the data list, where each element is the text of an email, and the target list, which is simply binary (1 meaning spam and 0 meaning ham). Now let's create a class and add some helper functions for string manipulation.

```

class SpamDetector(object):
    """Implementation of Naive Bayes for binary classification"""
    def clean(self, s):
        translator = str.maketrans("", "", string.punctuation)
        return s.translate(translator)

    def tokenize(self, text):
        text = self.clean(text).lower()
        return re.split("\W+", text)

    def get_word_counts(self, words):
        word_counts = {}
        for word in words:

```

```
word_counts[word] = word_counts.get(word, 0.0) + 1.0
return word_counts
```

We have a function to clean up our string by removing punctuation, one to tokenize our string into words, and another to count up how many of each word appears in a list of words. Before we start the actual algorithm, let's first understand the algorithm. For training, we need three things: the (log) class priors, i.e., the probability that any given message is spam/ham; a vocabulary of words; and words frequency for spam and ham separately, i.e., the number of times a given word appears in a spam and ham message. Given a list of input documents, we can write this algorithm.

1. Compute log class priors by counting how many messages are spam/ham, dividing by the total number of messages, and taking the log.
2. For each (document, label) pair, tokenize the document into words.
3. For each word, either add it to the vocabulary for spam/ham, if it isn't already there, and update the number of counts. Also add that word to the global vocabulary.

```
def fit(self, X, Y):
    self.log_class_priors = {}
    self.word_counts = {}
    self.vocab = set()

    n = len(X)
    self.log_class_priors['spam'] = math.log(sum(1 for label in Y if label
== 1) / n)
    self.log_class_priors['ham'] = math.log(sum(1 for label in Y if label
== 0) / n)
    self.word_counts['spam'] = {}
    self.word_counts['ham'] = {}

    for x, y in zip(X, Y):
        c = 'spam' if y == 1 else 'ham'
        counts = self.get_word_counts(self.tokenize(x))
        for word, count in counts.items():
            if word not in self.vocab:
                self.vocab.add(word)
            if word not in self.word_counts[c]:
                self.word_counts[c][word] = 0.0

            self.word_counts[c][word] += count
```

First, we can compute the log class priors by counting up how many spam/ham messages are in our dataset and dividing by the total number. Finally, we take the log. Then we can iterate through our dataset. For each input, we get the word counts and iterate through each (word, frequency) pair. If the word isn't in our global vocabulary, we add it. If it isn't in the vocabulary for that particular class label, we also add it along with the frequency.

For example, suppose we had a "spam" message. We count up how many times each unique word appears in that spam message and add that count to the "spam" vocabulary. Suppose the word "free" appears 4 times. Then we add the word "free" to our global vocabulary and add it to the "spam" vocabulary with a count of 4.

We're keeping track of the frequency of each word as it appears in either a spam or ham message. For example, we expect the word "free" to appear in both messages, but we expect it to be more frequent in the "spam" vocabulary than the "ham" vocabulary. Now that we've extracted all of the data we need from the training data, we can write another function to actually output the class label for new data. To do this classification, we apply Naive Bayes directly. For example, given a document, we need to iterate each of the words and compute  $\log p(w_i|\text{Spam})$  and sum them all up, and we also compute  $\log p(w_i|\text{Ham})$  and sum *them* all up. Then we add the log class priors and check to see which score is bigger for that document. Whichever is larger, that is the predicted label!

To compute  $\log p(w_i|\text{Spam})$ , the numerator is how many times we've seen  $w_i$  in a "spam" message divided by the total count of all words in every "spam" message.

On additional note: remember that the log of 0 is undefined! What if we encounter a word that is in the "spam" vocabulary, but not the "ham" vocabulary? Then  $p(w_i|\text{Ham})$  will be 0! One way around this is to use **Laplace Smoothing**. We simply add 1 to the numerator, but we also have to add the size of the vocabulary to the denominator to balance it.

```
def predict(self, X):
    result = []
    for x in X:
        counts = self.get_word_counts(self.tokenize(x))
        spam_score = 0
        ham_score = 0
        for word, _ in counts.items():
            if word not in self.vocab: continue

            # add Laplace smoothing
            log_w_given_spam = math.log(
                (self.word_counts['spam'].get(word, 0.0) + 1) /
                (sum(self.word_counts['spam'].values()) + len(self.vocab)) )
```

```

        log_w_given_ham = math.log( (self.word_counts['ham'].get(word,
0.0) + 1) / (sum(self.word_counts['ham'].values()) + len(self.vocab)) )

        spam_score += log_w_given_spam
        ham_score += log_w_given_ham

    spam_score += self.log_class_priors['spam']
    ham_score += self.log_class_priors['ham']

    if spam_score > ham_score:
        result.append(1)
    else:
        result.append(0)
    return result

```

In our case, the input can be a list of document texts; we return a list of predictions. Finally, we can use the class like this.

```

if __name__ == '__main__':
    X, y = get_data(DATA_DIR)
    MNB = SpamDetector()
    MNB.fit(X[100:], y[100:])

    pred = MNB.predict(X[:100])
    true = y[:100]

    accuracy = sum(1 for i in range(len(pred)) if pred[i] == true[i]) /
float(len(pred))
    print("{0:.4f}".format(accuracy))

```

We're reserving the first 100 for the testing set, "train" our Naive Bayes classifier, then compute the accuracy.

To recap, we reviewed Bayes Theorem and demonstrated how to use it with an example. Then we re-worked it using hypotheses and evidence instead of just events  $A$  and  $B$  to make it more specific to our task of spam detection. From there, we derived Naive Bayes by making the Naive Bayes Assumption that each word appears independently of all other words. Then we formulated a prediction equation/rule. Using the Enron dataset, we created a binary Naive Bayes classifier for detecting spam emails.

Naive Bayes is a simple text classification algorithm that uses basic probability laws and works quite well in practice!

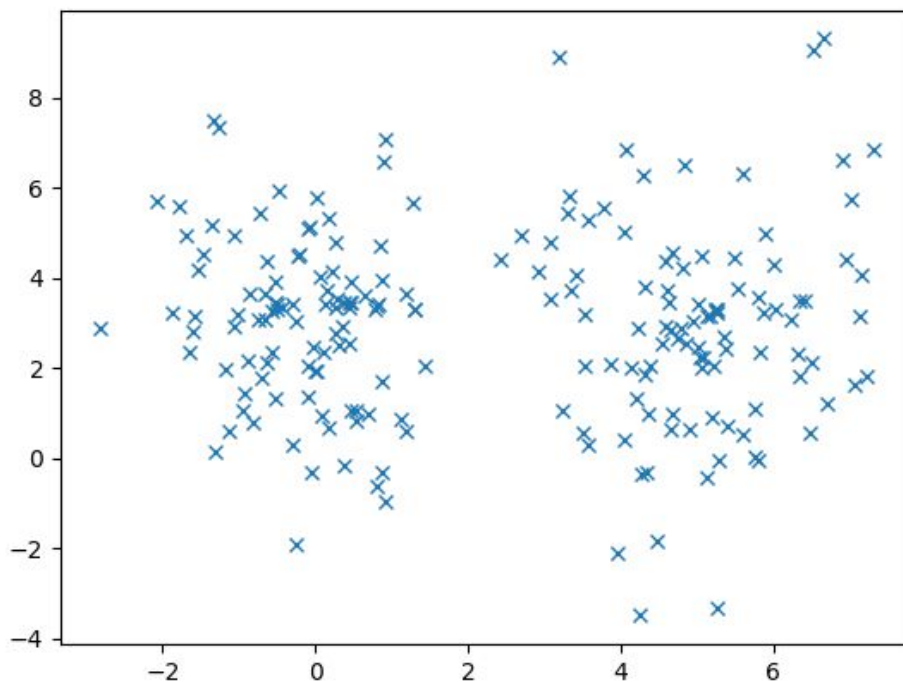
# Data Clustering with K-Means

Determining data clusters is an essential task to any data analysis and can be a very tedious task to do manually! This task is nearly impossible to do by hand in higher-dimensional spaces! Along comes machine learning to save the day! We will be discussing the K-Means clustering algorithm, the most popular flavor of clustering algorithms. Download the full code [here](#).

## Clustering

Before diving right into the algorithms, code, and math, let's take a second to define our problem space. Clustering is a type of **unsupervised learning**; our data do not have any ground-truth labels associated with them. With clustering, we have a set of **unlabeled** data  $D = \{x_1, \dots, x_n\}$  where each  $x_i \in \mathbb{R}^m$ . In other words, we have a set of vectors of an arbitrary dimension. We'll only be dealing with vectors in 2D, but we can easily extend our task and approach to any dimensionality of data.

Given our dataset and  $k$ , the number of clusters, we want to automatically find where to position the clusters. To make this example a bit more concrete, let's consider a set of 200 data points.



The problem of clustering is to figure out where those  $k$  clusters should be. From these data, we can figure out that there should be 2 clusters. Maybe the left cluster should be about  $(-0.25, 2)$  and the right cluster should be about  $(5.75, 2)$ . In fact, I've generated this data according to a specific algorithm, that I won't reveal until later, so we can see how close our clustering algorithms get!

## Choosing the Number of Clusters

One question that always comes up when discussing clustering algorithms is "how many clusters should I use?" From the clustering algorithm, we at least know that it should be strictly less than the number of data points! Naturally, there's no *right* answer to this question, although there are many different heuristics that we could use to select this.

One such heuristic is simply to try many different values of  $k$  and see which one works the best. In reality, there's no one heuristic or algorithm that we can use to automatically determine the number of clusters each time. Sometimes, we just have to eyeball it! With 2D data, we can usually determine at least a good range of  $k$  to try. In our above example,  $k = 2$  would be a good choice, but so might  $k = 4$ . We're going to assume  $k = 2$  for the data that we're dealing with.

## K-Means Clustering

Let's discuss **k-means clustering**. The algorithm itself is fairly intuitive so we'll look at that first. Then, we'll take a closer look at some nice properties of k-means clustering. The k-means algorithm starts by randomly initializing the cluster centers. This random initialization step is *very* important! We'll see the effects of poor initialization later. Then, we take each point in our dataset and assign it to the closest cluster center using the Euclidean distance from the cluster center to the point. At the end of this step, each point should be assigned to its closest cluster, i.e., there will never be an unassigned point (although there may be clusters with no assigned points). Then we update the position of the cluster by taking the mean of the points that are assigned to it. Then we assign each point to its closest cluster center again and repeat! How long do we repeat for? We repeat until convergence, and k-means will always converge (we'll discuss the reason for this later).

Here is the k-means algorithm.

1. Randomly initialize cluster centers  $\mu_1, \dots, \mu_k$
2. While not converged:
3. Assign each point to its closest cluster center
4. Update each  $\mu_i$  to be the mean of all points assigned to that cluster

Now that we've seen the algorithm, let's get to the code!

## K-Means Clustering Code

First, download the ZIP file (link is at the beginning of this post). Inside, there is a file called data.pkl that has all of our data points. We can use Python's pickle library to load data from this file and plot it using the following code snippet. We should see the same plot as above.

```

if __name__ == '__main__':
    with open('data.pkl', 'rb') as f:
        data = pickle.load(f)

    X = data['x']

    plt.plot(X[:,0], X[:,1], 'x')
    plt.show()

```

Remember that the k-means algorithm is an **unsupervised** algorithm so there are no ground-truth labels. Here, the matrix  $X$  holds all of our data, where each row is a data point and each column is a dimension, e.g., the dimensions of  $X$  should be  $200 \times 2$  in our example above. Now we can get started creating our class.

```

class KMeans(object):
    """Implementation of KMeans"""
    def __init__(self, k):
        self.k = k

```

The only parameter that we need is the number of clusters  $k$ . Now we can start writing our function to initialize and move the cluster centers.

```

def fit(self, X):
    self.centers = X[np.random.randint(X.shape[0], size=self.k)]

    plt.plot(X[:,0], X[:,1], 'bx')
    plt.plot(self.centers[:,0], self.centers[:,1], 'ro')
    plt.show()

```

How do we randomly initialize our clusters? Let's just randomly select  $k$  points from our dataset to act as the initial cluster center. There are certainly more sophisticated ways of initializing the cluster centers, but this is the simplest and works well.

We will also be learning some new numpy tricks along the way. For example, inside of the brackets when we initialize the cluster centers, we're actually giving numpy a list of indices. And this does exactly what you think! Numpy will give us a matrix back with the rows that correspond to those indices.

Now we can write the main loop.

```

while True:
    distances = np.sqrt(np.sum((X - self.centers[:, np.newaxis]) ** 2,
axis=2))
    closestClusters = np.argmin(distances, axis=0)

```



First, we're using an infinite loop, which would usually be a bit dangerous, but k-means has some nice properties about convergence that we'll discuss later. The two lines after, we compute the Euclidean distance of each point to each cluster center and determine the index of the cluster. There is a lot going on in this first line, and we use another numpy trick.

The square root, sum, and square is just part of computing the Euclidean distance. The strange part of the code seems to be the following line.

```
X - self.centers[:, np.newaxis]
```

If we look at the resulting shape of the array, we see  $2 \times 200 \times 2$ , but, in general, we would get  $k \times n \times 2$ , where the shape is the number of cluster centers, number of data points, and the dimensionality of the data points, in that order. But wait, didn't we only have 2 dimensions before? The extra dimension is from using `np.newaxis`. The extra dimension is only temporary since we need it to compute the Euclidean distance, the square root of the sum of the square of each input dimension (2 in our case).

With the last line above, we're just selecting the index of the cluster that is closest to a given point. At the end of these lines of code, we know the index of the closest cluster for each point.

Next, we need to compute the new cluster centers by taking the mean of the set of data points that belongs to a cluster. We can do this efficiently in numpy in a single line of code.

```
newCenters = np.array([np.mean(X[closestClusters == c], axis=0) for c in range(self.k)])
```

In the innermost statement, `X[closestClusters == c]`, we're asking numpy to select all of the points who have `c` as their closest cluster. Then we can take the mean point of that list of points.

Why don't we update the cluster centers? Why do we create a new variable? This is because we need a way to break out of the infinite loop. When k-means converges, our cluster centers have settled and *do not move*! We can write code to check for this and break out of the loop if the new cluster centers haven't changed.

```
if np.all(self.centers - newCenters < 1e-5):  
    break  
self.centers = newCenters
```

Since we're dealing with floating-point numbers, we actually have a small margin of  $10^{-5}$ . If we have moved more than that, then we update the cluster centers.

Here's the entire code for the main k-means function.

```
def fit(self, X):
    self.centers = X[np.random.randint(X.shape[0], size=self.k)]

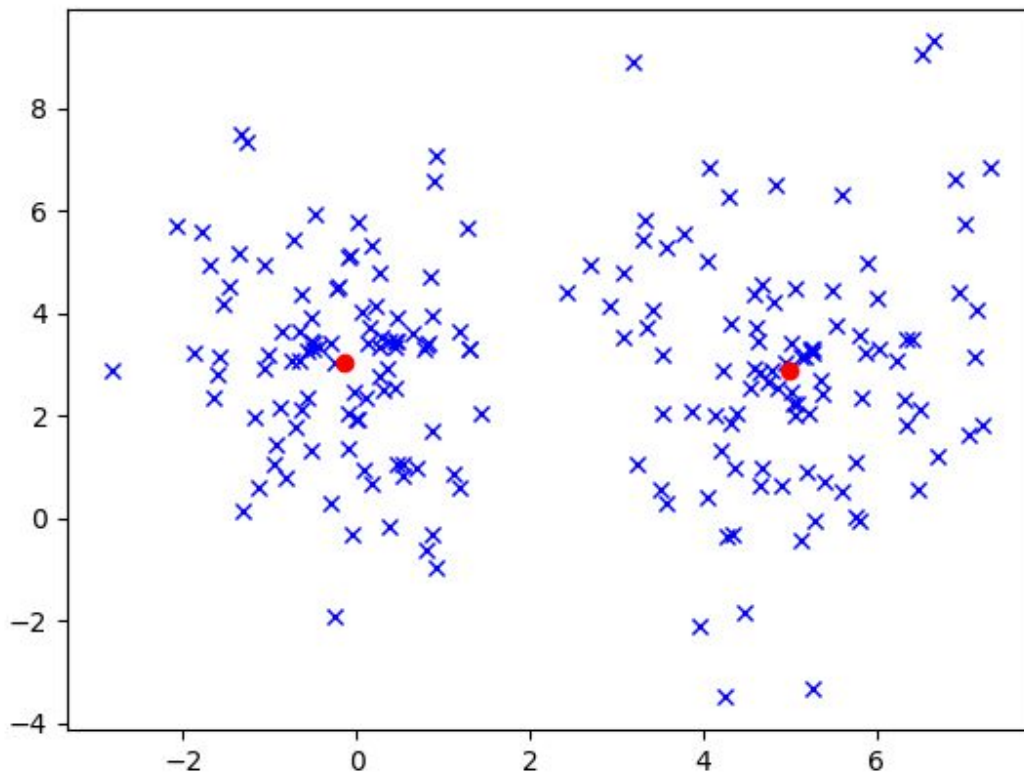
    plt.plot(X[:,0], X[:,1], 'bx')
    plt.plot(self.centers[:,0], self.centers[:,1], 'ro')
    plt.show()

    while True:
        distances = np.sqrt(np.sum((X - self.centers[:, np.newaxis]) ** 2,
axis=2))
        closestClusters = np.argmin(distances, axis=0)

        newCenters = np.array([np.mean(X[closestClusters == c], axis=0) for
c in range(self.k)])
        if np.all(self.centers - newCenters < 1e-5):
            break
        self.centers = newCenters

    plt.plot(X[:,0], X[:,1], 'bx')
    plt.plot(self.centers[:,0], self.centers[:,1], 'ro')
    plt.show()
```

There's also code that displays the initial and final cluster positions.

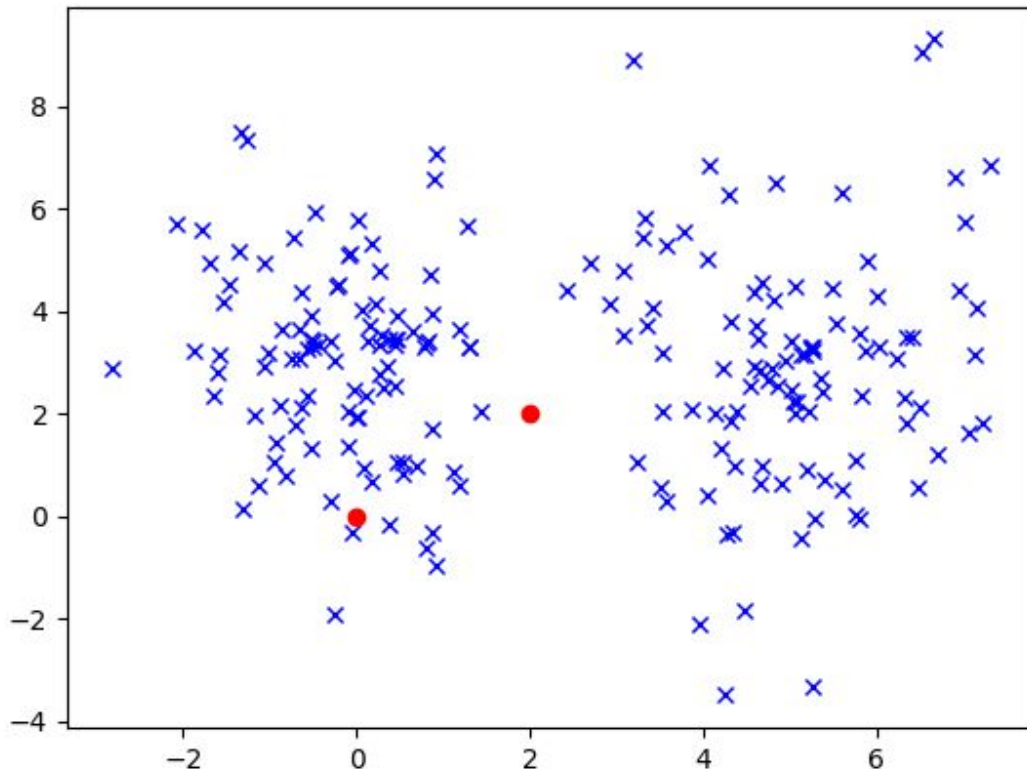


This figure shows an example of what we might expect to see with K-Means. In fact, the final cluster positions are *very close to the right answer!* I generated this data using two multivariate Gaussian distributions centered at  $(0, 3)$  and  $(5, 3)$ .

Due to random initialization, it might be the case that your cluster centers converged to a different location. Usually, the cluster centers will be pretty close to the true centers. (Obviously, when dealing with real-world data, we can't possibly know where the *true* centers are!) But there are some instances where poor initialization can lead to bad results.

## Poor Initialization of Cluster Centers

Good initialization of the cluster centers is critical to good results! Consider the following figure.



This is an example of poor initialization! The cluster centers are not near the true values at all. (To generate this, I manually initialized the cluster centers to bad values.) This is certainly a possibility when using clustering algorithms that randomly initialize data. How can we combat this? A simple, but effective, technique to prevent this is to run the k-means algorithm for multiple trials and take the average of the final cluster centers. This way, if we have poor initialization for a few trials, they won't affect our final results much. There are more sophisticated ways than taking an average, e.g., build a distribution over the cluster centers from all trials and get rid of any final cluster positions that are some distance away from the mean.

## K-Means Convergence

Now that we've discussed the algorithm, there's still a question we haven't answered yet: How do we know the k-means algorithm will converge? To answer this question, we have to look at its cost function.

$$J = \sum_{j=1}^k \sum_{i \in C_j} ||x_i - \mu_j||^2$$

At a high level, the cost is measured by computing the sum of all points from their assigned cluster for all clusters. The cost function is at a minimum when, for each cluster, it is positioned so that the cluster center is a minimal distance from each point assigned to it. The inner sum is over a particular cluster  $j$ . It takes the sum of the distance from all points that belong to cluster  $j$  to the cluster center of cluster  $j$ . Then the outer sum simply sums over all clusters.

We won't really *prove* this, but I'll provide an intuitive understanding as to why k-means converges and doesn't keep iterating to infinity. The idea hinges on the fact that, at each step of the loop, the cost value at the new cluster center must be less than or equal to than the previous cluster center location. In other words, the cost is *monotonically decreasing* since we're updating the centers based only on the data. Notice that the cluster centers update *does not* depend on the previous location! (There is a formal convergence proof that proves k-means converges in a finite number of steps, but I've skipped over that since the intuition is more useful than the proof.) Hence, k-means will keep iterating until the new cost value is the same as the old one. We have a conditional check for this in our code, and that's where we break out of the loop. There may be cases where k-means takes a long time; in those cases, we could replace the infinite while loop with a finite loop that iterates until the maximum number of allowable iterations is met.

One important thing to note is that *there's no guarantee* that it will converge to the right answer or even an answer that makes sense! (Theoretically, computing the global minimum, i.e., the optimal answer, is an NP-hard problem.) The only guarantee is that k-means will stop, i.e., the cluster centers will stop moving, in a finite amount of steps.

To summarize, we discussed the most popular clustering algorithm: k-means clustering. This is an intuitive algorithm that, given the number of clusters, can automatically find out where the clusters should be. First, we randomly initialize the cluster centers. Then we assign each point to its closest cluster and update the cluster center to be the mean of the set of all points assigned to it. After we perform this update, we simply re-assign each point and repeat until the algorithm converges. We discussed some critical aspects of this algorithm such as initialization and convergence. In addition, we learned some numpy tricks along the way.

In practice, the k-means algorithm is very popular and the first-used algorithm when given a clustering task.

# Clustering with Gaussian Mixture Models

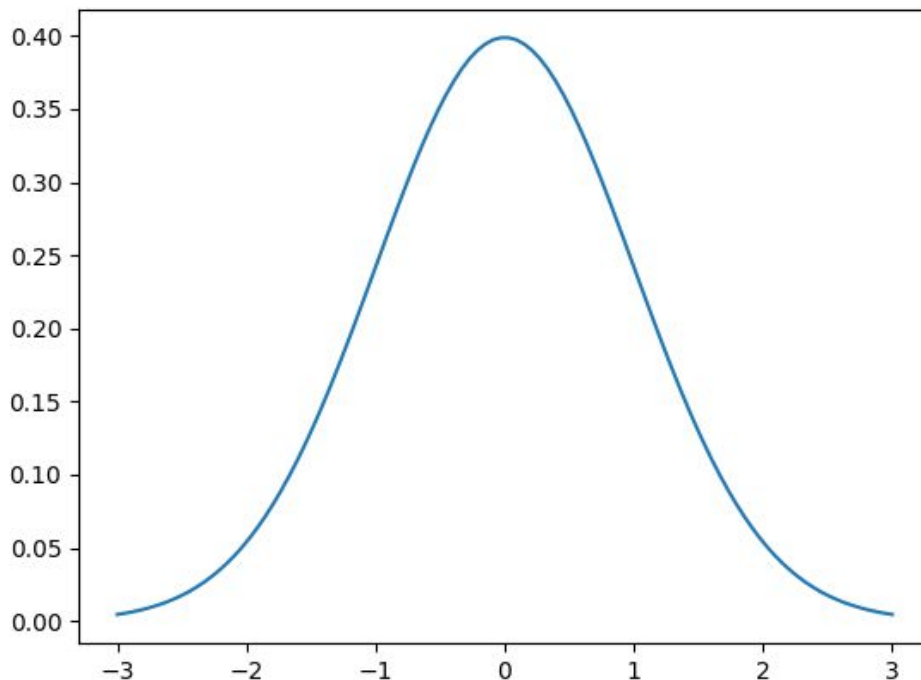
Clustering is an essential part of any data analysis. Using an algorithm such as K-Means leads to **hard assignments**, meaning that each point is definitively assigned a cluster center. This leads to some interesting problems: what if the true clusters actually overlap? What about data that is more spread out; how do we assign clusters then?

**Gaussian Mixture Models** save the day! We will review the Gaussian or normal distribution method and the problem of clustering. Then we will discuss the overall approach of Gaussian Mixture Models. Training them requires using a very famous algorithm called the **Expectation-Maximization Algorithm** that we will discuss. Download the full code [here](#).

If you are not familiar with the K-Means algorithm or clustering, read about it [here](#).

## Gaussian Distribution

The first question you may have is "what is a Gaussian?". It's the most famous and important of all statistical distributions. A picture is worth a thousand words so here's an example of a Gaussian centered at 0 with a standard deviation of 1.



This is the **Gaussian** or **normal distribution**! It is also called a **bell curve** sometimes. The function that describes the normal distribution is the following

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x - \mu)^2}{2\sigma^2}}$$

That looks like a really messy equation! And it is, so we'll use  $\mathcal{N}(x; \mu, \sigma^2)$  to represent that equation. If we look at it, we notice there are one input and two parameters. First, let's discuss the parameters and how they change the Gaussian. Then we can discuss what the input means.

The two parameters are called the **mean**  $\mu$  and **standard deviation**  $\sigma$ . In some cases, the standard deviation is replaced with the **variance**  $\sigma^2$ , which is just the square of the standard deviation. The mean of the Gaussian simply shifts the center of the Gaussian, i.e., the "bump" or top of the bell. In the image above,  $\mu = 0$ , so the largest value is at  $x = 0$ .

The standard deviation is a measure of the *spread* of the Gaussian. It affects the "wideness" of the bell. Using a larger standard deviation means that the data are more spread out, rather than closer to the mean.

What about the input? More specifically, the above function is called the **probability density function (pdf)** and it tells us the probability of observing an input  $x$ , given that specific normal distribution. Given the graph above, we see that observing an input value of 0 gives us a probability of about 40%. As we move away in either direction from the center, we are decreasing the probability. This is one key property of the normal distribution: the highest probability is located at mean while the probabilities approach zero as we move away from the mean. (Since this is a probability distribution, the sum of all of the values under the bell curve, i.e., the integral, is equal to 1; we also have no negative values.)

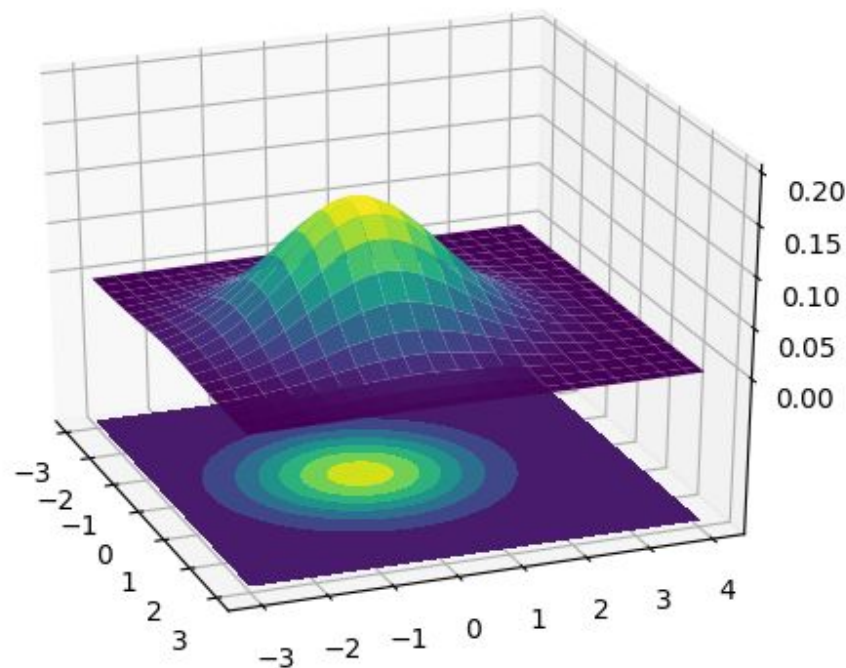
Why are we using the Gaussian distribution? The Expectation-Maximization algorithm is actually more broad than just the normal distribution, but what makes Gaussians so special? It turns out that *many* dataset distributions are actually Gaussian! We find these Gaussians in nature, mathematics, physics, biology, and just about every other field! They are ubiquitous! There is a famous theorem in statistics called the **Central Limit Theorem** that states that as we collect more and more samples from a dataset, they tend to resemble a Gaussian, *even if the original dataset distribution is not Gaussian!* This makes Gaussian very powerful and versatile!

## Multivariate Gaussians

We've only discussed Gaussians in 1D, i.e., with a single input. But they can easily be extended to any number of dimensions. For Gaussian Mixture Models, in particular, we'll use 2D Gaussians, meaning that our input is now a vector instead of a scalar. This also changes our parameters: the mean is now a vector as well! The mean represents the center of our data so it must have the same dimensionality as the input.

The variance changes less intuitively into a **covariance matrix**  $\Sigma$ . The covariance matrix, in addition to telling us the variance of each dimension, also tells us the relationship *between* the inputs, i.e., if we change x, how does y tend to change?

We won't discuss the details of the multivariate Gaussian or the equation that generates it, but knowing what it looks like is essential to Gaussian Mixture Models since we'll be using these.

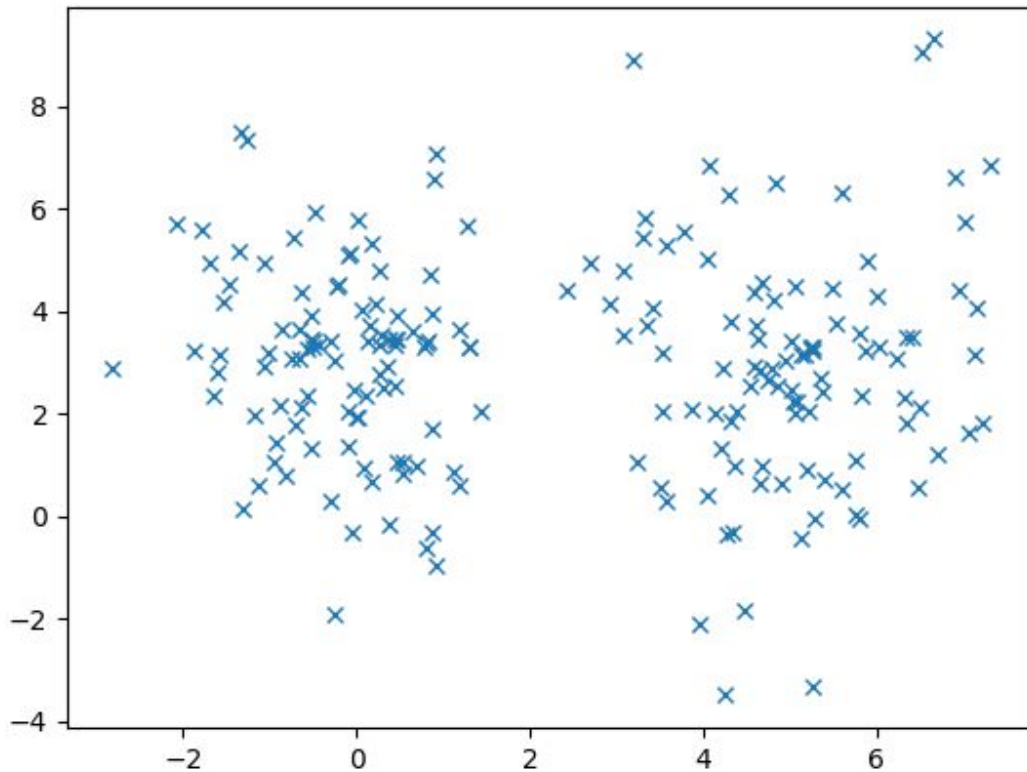


The above chart has two different ways to represent the 2D Gaussian. The upper plot is a **surface plot** that shows this our 2D Gaussian in 3D. The X and Y axes are the two inputs and the Z axis represents the probability. The lower plot is a **contour plot**. I've plotted these on top of each other to show how the contour plot is just a flattened surface plot where color is used to determine the height. The lighter the color, the larger the probability. The Gaussian contours resemble ellipses so our Gaussian Mixture Model will look like it's fitting ellipses around our data. Since the surface plot can get a little difficult to visualize on top of data, we'll be sticking to the contour plots.

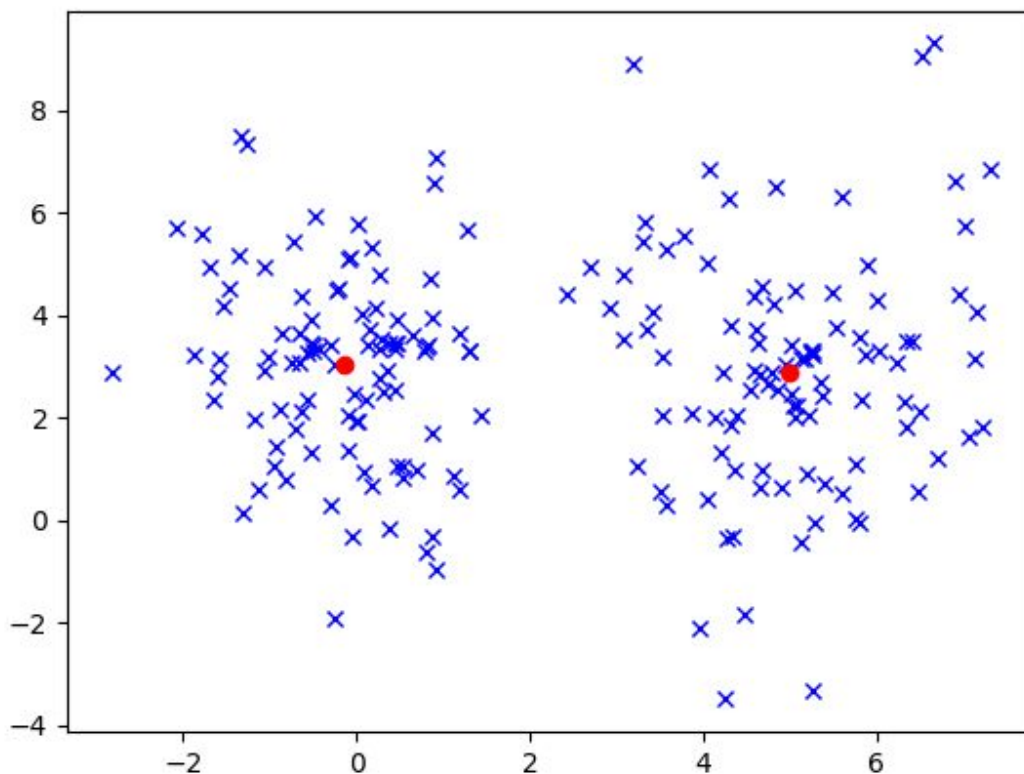


## Gaussian Mixture Models

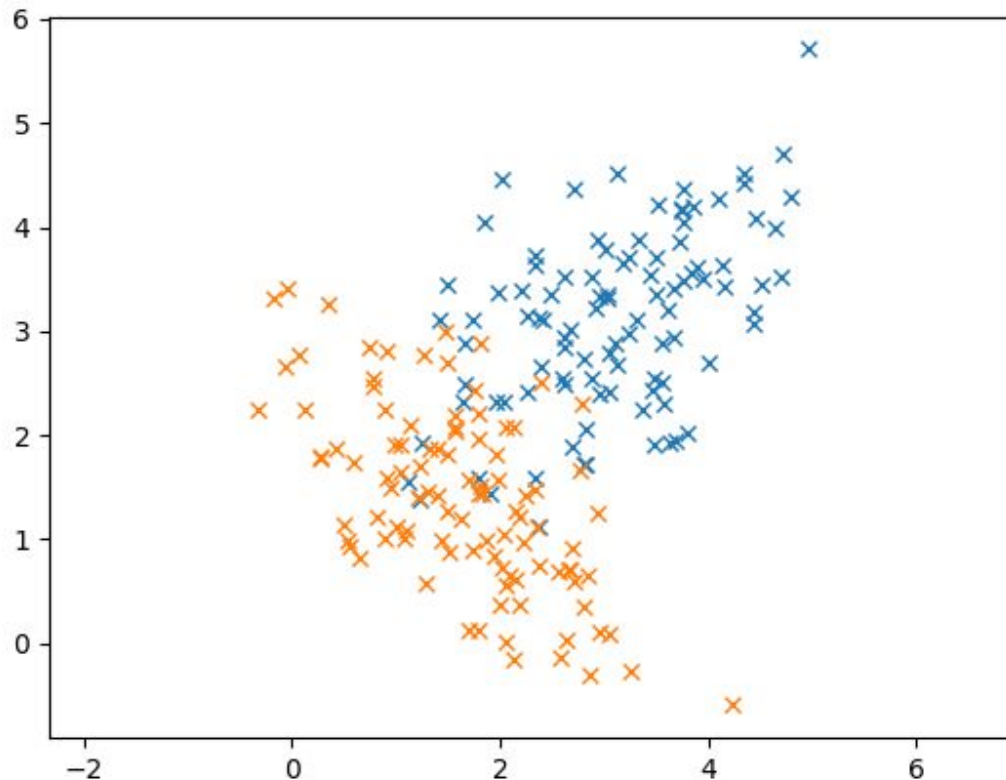
Now that we understand Gaussians, let's discuss to Gaussian Mixture Models (GMMs)! To motivate our discussion, let's see some example data that we want to cluster.



We could certainly cluster these data using an algorithm like K-Means to get the following results.



In this case, K-Means works out pretty well. But let's consider another case where we have overlap in our data. (The two Gaussians are colored differently)



In this case, it's pretty clear that these data are generated from Gaussians from the elliptical shape of the 2D Gaussian. In fact, we know that these data follow the normal distribution so using K-Means doesn't seem to take advantage of that fact. Even though I didn't tell you our data were normally distributed, remember that the **Central Limit Theorem** says that enough random samples from any distribution will look like the normal distribution.

Additionally, K-Means doesn't take into account the covariance of our data. For example, the blue points seem to have a relationship between X and Y: larger X values tend to produce larger Y values. If we had two points that were equidistant from the center of the cluster, but one followed the trend and the other didn't, K-Means would regard them as being equal, since it uses Euclidean distance. But it seems certainly more likely that the point that follows the trend should match closer to the Gaussian than the point that doesn't.

Since we know these data are Gaussian, why not try to fit Gaussians to them instead of a single cluster center? The idea behind **Gaussian Mixture Models** is to find the parameters of the Gaussians that best explain our data.

This is what we call **generative modeling**. We are assuming that these data are Gaussian and we want to find parameters that maximize the likelihood of observing these data. In other words, we regard each point as being generated by a *mixture of Gaussians* and can compute that probability.

$$p(x) = \sum_{j=1}^k \phi_j \mathcal{N}(x; \mu_j, \Sigma_j)$$

$$\sum_{j=1}^k \phi_j = 1$$

The first equation tells us that a particular data point  $x$  is a *linear combination* of the  $k$  Gaussians. We weight each Gaussian with  $\phi_j$ , which represents the strength of that Gaussian. The second equation is a constraint on the weights: they all have to sum up to 1. We have three different parameters that we need to write update: the weights for each Gaussian  $\phi_j$ , the means of the Gaussians  $\mu_j$ , and the covariances of each Gaussian  $\Sigma_j$ .

If we try to directly solve for these, it turns out that we can actually find closed-forms! But there is one huge catch: we have to know the  $\phi_j$ 's! In other words, if we knew exactly which combination of Gaussians a particular point was taken from, then we could easily figure out the means and covariances! But this one critical flaw prevents us from solving GMMs using this direct technique. Instead, we have to come up with a better approach to estimate the weights, means, covariances.

## Expectation-Maximization

How do we learn the parameters? There's a very famous algorithm called the **Expectation-Maximization Algorithm**, also called the **EM algorithm** for short, (written in 1977 with over 50,000 paper citations!) that we'll use for updating these parameters. There are two steps in this algorithm as you might think: *expectation* and *maximization*. To explain these steps, I'm going to cover how the algorithm works at a high level.

The first part is the *expectation* step. In this step, we have to compute the probability that each data point was generated by each of the  $k$  Gaussians. In contrast to the K-Means **hard assignments**, these are called **soft assignments** since we're using probabilities. Note that we're not assigning each point to a Gaussian, we're simply determining the probability of a particular Gaussian generating a particular point. We compute this probability for a given

Gaussian by computing  $\phi_j \mathcal{N}(x; \mu_j, \Sigma_j)$  and normalizing by dividing by  $\sum_{q=1}^k \phi_q \mathcal{N}(x; \mu_q, \Sigma_q)$ .

We're directly applying the Gaussian equation, but multiplying it by its weight  $\phi_j$ . Then, to make it a probability, we normalize. In K-Means, the expectation step is analogous to assigning each point to a cluster.

The second part is the *maximization* step. In this step, we need to update our weights, means, and covariances. Recall in K-Means, we simply took the mean of the set of points assigned to a cluster to be the new mean. We're going to do something similar here, except apply our expectations that we computed in the previous step. To update a weight  $\phi_j$ , we simply sum up the probability that each point was generated by Gaussian  $j$  and divide by the total number of points. For a mean  $\mu_j$ , we compute the *mean* of all points weighted by the probability of that point being generated by Gaussian  $j$ . For a covariance  $\Sigma_j$ , we compute the *covariance* of all points weighted by the probability of that point being generated by Gaussian  $j$ . We do each of these for each Gaussian  $j$ . Now we've updated the weights, means, and covariances! In K-Means, the maximization step is analogous to moving the cluster centers.

Mathematically, at the expectation step, we're effectively computing a matrix where the rows are the data point and the columns are the Gaussians. An element at row  $i$ , column  $j$  is the probability that  $x^{(i)}$  was generated by Gaussian  $j$ .

$$W_j^{(i)} = \frac{\phi_j \mathcal{N}(x^{(i)}; \mu_j, \Sigma_j)}{\sum_{q=1}^K \phi_q \mathcal{N}(x^{(i)}; \mu_q, \Sigma_q)}$$

The denominator just sums over all values to make each entry in  $W$  a probability. Now, we can apply the update rules.

$$\begin{aligned}\phi_j &= \frac{1}{N} \sum_{i=1}^N W_j^{(i)} \\ \mu_j &= \frac{\sum_{i=1}^N W_j^{(i)} x^{(i)}}{\sum_{i=1}^N W_j^{(i)}} \\ \Sigma_j &= \frac{\sum_{i=1}^N W_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^N W_j^{(i)}}\end{aligned}$$

The first equation is just the sum of the probabilities of a particular Gaussian  $j$  divided by the number of points. In the second equation, we're just computing the mean, except we multiply by the probabilities for that cluster. Similarly, in the last equation, we're just computing the covariance, except we multiply by the probabilities for that cluster.

## Applying GMMs

Let's apply what we learned about GMMs to our dataset. We'll be using scikit-learn to run a GMM for us. In the ZIP file, I've saved some data in a numpy array. We're going to extract it, create a GMM, run the EM algorithm, and plot the results! First, we need to load the data.

```
import numpy as np
```

```
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture

X_train = np.load('data.npy')
Additionally, we can generate a plot of our data using the following code.
plt.plot(X[:,0], X[:,1], 'bx')
plt.axis('equal')
plt.show()
```

Remember that clustering is unsupervised, so our input is only a 2D point without any labels. We should get the same plot of the 2 Gaussians overlapping.

Using the GaussianMixture class of scikit-learn, we can easily create a GMM and run the EM algorithm in a few lines of code!

```
gmm = GaussianMixture(n_components=2)
gmm.fit(X_train)
After our model has converged, the weights, means, and covariances should
be solved! We can print them out.
print(gmm.means_)
print('\n')
print(gmm.covariances_)
```

For comparison, I generate the original data data according to the following Gaussians.

$$\mu_1 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\Sigma_1 = \begin{bmatrix} 1 & 0.6 \\ 0.6 & 1 \end{bmatrix}$$

$$\mu_2 = \begin{bmatrix} 1.5 \\ 1.5 \end{bmatrix}$$

$$\Sigma_2 = \begin{bmatrix} 1 & -0.7 \\ -0.7 & 1 \end{bmatrix}$$

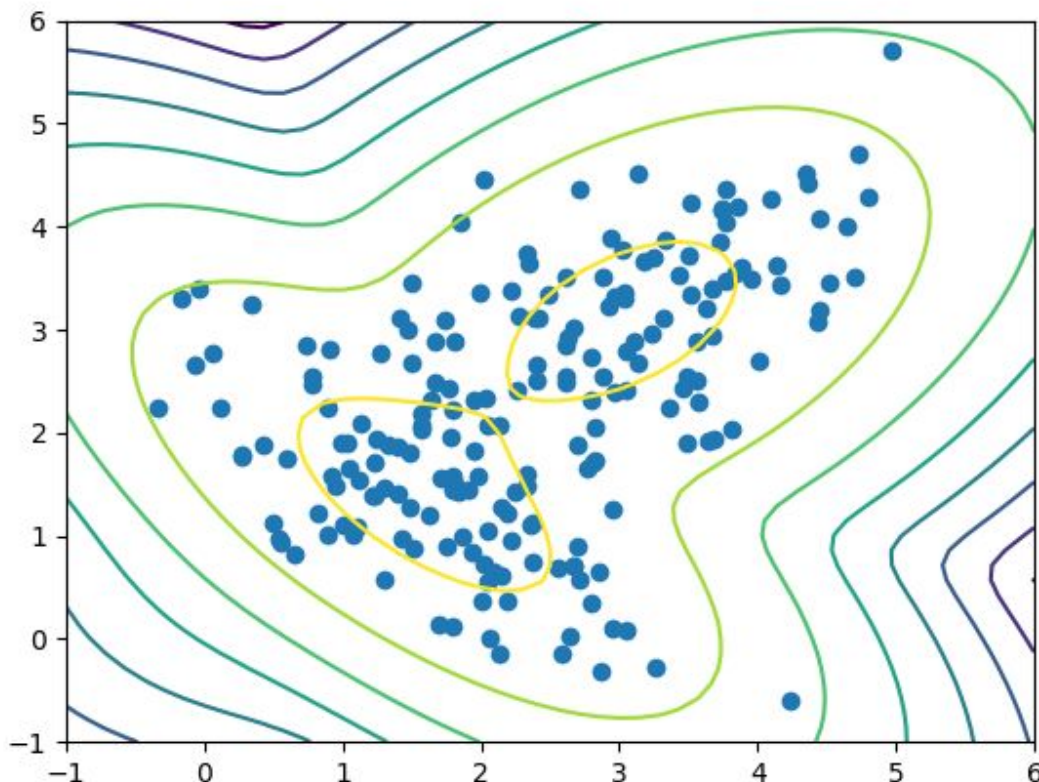
Our means should be pretty close to the actual means! (Our covariances might be a bit off due to the weights.) Now we can write some code to plot our mixture of Gaussians.

```
X, Y = np.meshgrid(np.linspace(-1, 6), np.linspace(-1,6))
XX = np.array([X.ravel(), Y.ravel()]).T
Z = gmm.score_samples(XX)
Z = Z.reshape((50,50))
```

```
plt.contour(X, Y, Z)
plt.scatter(X_train[:, 0], X_train[:, 1])

plt.show()
```

This code simply creates a grid of all X and Y coordinates between -1 and 6 (for both) and evaluates our GMM. Then we can plot our GMM as contours over our original data.



The plot looks just as we expected! Recall that with the normal distribution, we expect to see most of the data points around the mean and less as we move away. In the plot, the first few ellipses have most of the data, with only a few data points towards the outer ellipses. The darker the contour, the lower the score.

(The score isn't quite a probability: it's actually a weighted log probability. Remember that each point is generated by a weighted sum of Gaussians, and, in practice, we apply a logarithm for numerical stability, thus prevent underflow.)

To summarize, Gaussian Mixture Models are a clustering technique that allows us to fit multivariate Gaussian distributions to our data. These GMMs work well when our data is actually

Gaussian or we suspect it to be. We also discussed the famous expectation-maximization algorithm, at a high level, to see how we can iteratively solve for the parameters of the Gaussians. Finally, we wrote code to train a GMM and plot the resulting Gaussian ellipses. Gaussian Mixture Models are an essential part of data analysis and anomaly detection!



# Face Recognition with Eigenfaces

Face recognition is ubiquitous in science fiction: the protagonist looks at a camera, and the camera scans his or her face to recognize the person. More formally, we can formulate face recognition as a classification task, where the inputs are images and the outputs are people's names. We're going to discuss a popular technique for face recognition called **eigenfaces**. And at the heart of eigenfaces is an unsupervised dimensionality reduction technique called **principal component analysis (PCA)**, and we will see how we can apply this general technique to our specific task of face recognition. Download the full code [here](#).

## Face Recognition

Before discussing principal component analysis, we should first define our problem. **Face recognition** is the challenge of classifying whose face is in an input image. This is different than **face detection** where the challenge is determining if there is a face in the input image. With face recognition, we need an existing database of faces. Given a new image of a face, we need to report the person's name.

A naïve way of accomplishing this is to take the new image, flatten it into a vector, and compute the Euclidean distance between it and all of the other flattened images in our database.

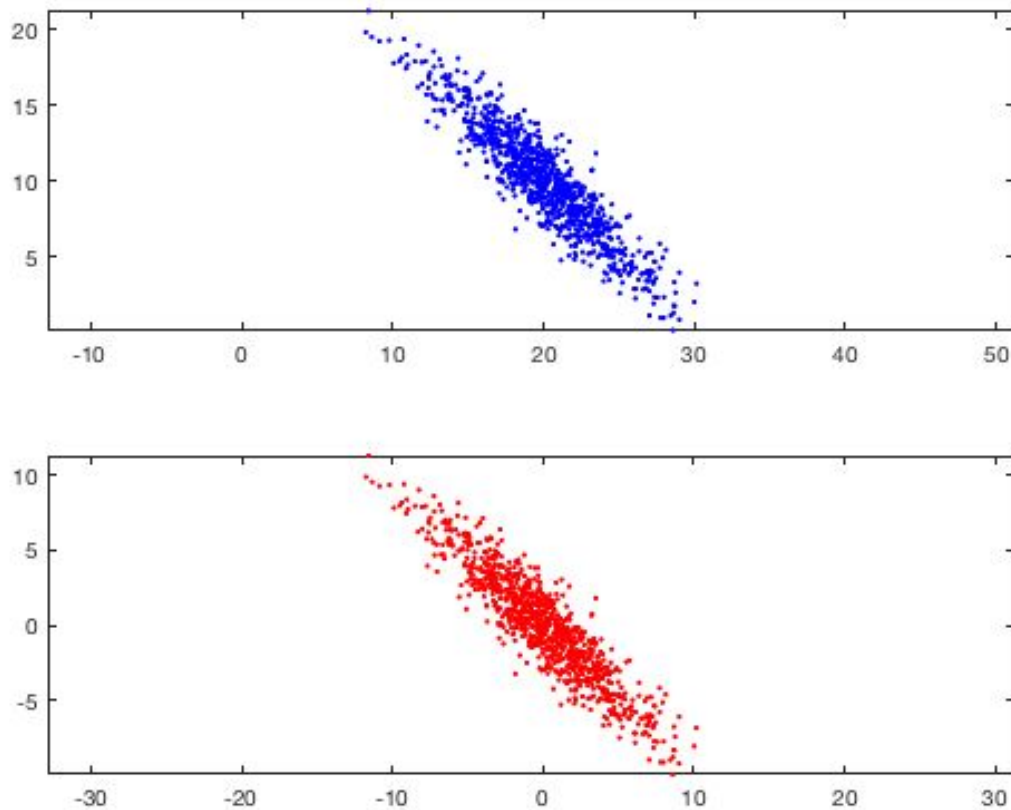
There are several downsides to this approach. First of all, if we have a large database of faces, then doing this comparison for each face will take a while! Imagine that we're building a face recognition system for real-time use! The larger our dataset, the slower our algorithm. But more faces will also produce better results! We want a system that is both fast and accurate. For this, we'll use a neural network! We can train our network on our dataset and use it for our face recognition task.

There's an issue with directly using a neural network: images can be large! If we had a single  $m \times n$  image, we would have to flatten it out into a single  $mn \times 1$  vector to feed into our neural network as input. For large image sizes, this might hurt speed! This is related to the second problem with using images as-is in our naïve approach: they are high-dimensional! (An  $m \times n$  image is really a  $mn \times 1$  vector) A new input might have a ton of noise and comparing each and every pixel using matrix subtraction and Euclidean distance might give us a high error and misclassifications!

These issues are why we don't use the naïve method. Instead, we'd like to take our high-dimensional images and boil them down to a smaller dimensionality while retaining the *essence* or *important parts* of the image.

## Dimensionality Reduction

The previous section motivates our reason for using a dimensionality reduction technique. Dimensionality reduction is a type of unsupervised learning where we want to take higher-dimensional data, like images, and represent them in a lower-dimensional space. Let's use the following image as an example.



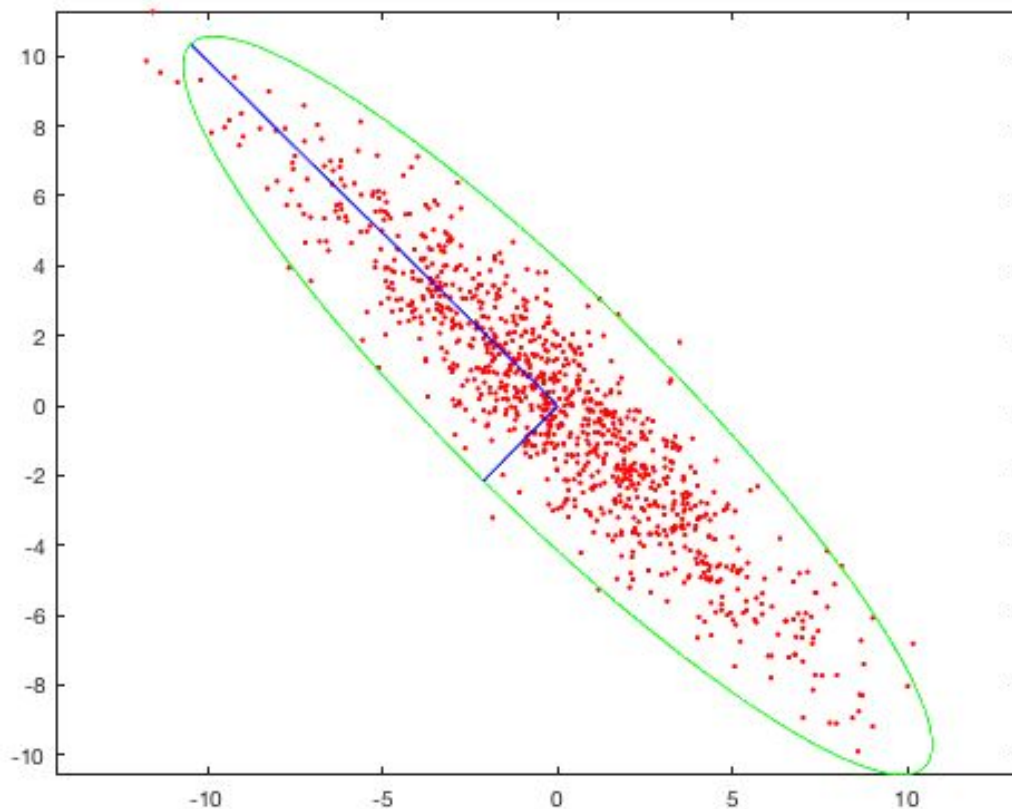
These plots show the same data, except the bottom chart zero-centers it. Notice that our data do not have any labels associated with them because this is *unsupervised learning*! In our simple case, dimensionality reduction will reduce these data from a 2D plane to a 1D line. If we had 3D data, we could reduce it down to a 2D plane or even a 1D line.

All dimensionality reduction techniques aim to find some **hyperplane**, a higher-dimensional line, to *project* the points onto. We can imagine a *projection* as taking a flashlight perpendicular to the hyperplane we're project onto and plotting where the shadows fall on that hyperplane. For example, in our above data, if we wanted to project our points onto the x-axis, then we pretend each point is a ball and our flashlight would point directly down or up (perpendicular to the x-axis) and the shadows of the points would fall on the x-axis. This is a *projection*. We won't worry about the exact math behind this since scikit-learn can apply this projection for us.

In our simple 2D case, we want to find a line to project our points onto. After we project the points, then we have data in 1D instead of 2D! Similarly, if we had 3D data, we want to find a plane to project the points down onto to reduce the dimensionality of our data from 3D to 2D. The different types of dimensionality reduction are all about figuring out which of these hyperplanes to select: there are an infinite number of them!

## Principal Component Analysis

One technique of dimensionality reduction is called **principal component analysis (PCA)**. The idea behind PCA is that we want to select the hyperplane such that when all the points are projected onto it, they are maximally spread out. In other words, we want the *axis of maximal variance*! Let's consider our example plot above. A potential axis is the x-axis or y-axis, but, in both cases, that's not the best axis. However, if we pick a line that cuts through our data diagonally, that is the axis where the data would be most spread!



The longer blue axis is the correct axis! If we were to project our points onto this axis, they would be maximally spread! But how do we figure out this axis? We can borrow a term from linear algebra called **eigenvectors**! This is where **eigenfaces** gets its name! Essentially, we compute the covariance matrix of our data and consider that covariance matrix's largest **eigenvectors**. Those are our *principal axes* and the axes that we project our data onto to reduce dimensions. Using this approach, we can take high-dimensional data and reduce it down

to a lower dimension by selecting the largest eigenvectors of the covariance matrix and projecting onto those eigenvectors.

Since we're computing the axes of maximum spread, we're retaining the most important aspects of our data. It's easier for our classifier to separate faces when our data are spread out as opposed to bunched together.

(There are other dimensionality techniques, such as Linear Discriminant Analysis, that use supervised learning and are also used in face recognition, but PCA works really well!)

How does this relate to our challenge of face recognition? We can conceptualize our  $m \times n$  images as points in  $mn$ -dimensional space. Then, we can use PCA to reduce our space from  $mn$  into something much smaller. This will help speed up our computations and be robust to noise and variation.

### Aside on Face Detection

So far, we've assumed that the input image is only that of a face, but, in practice, we shouldn't require the camera images to have a perfectly centered face. This is why we run an out-of-the-box face detection algorithm, such as a cascade classifier trained on faces, to figure out what portion of the input image has a face in it. When we have that bounding box, we can easily slice out that portion of the input image and use eigenfaces on that slice. (Usually, we smooth that slice and perform an affine transform to de-warp the face if it appears at an angle.) For our purposes, we'll assume that we have images of faces already.

### Eigenfaces Code

Now that we've discussed PCA and eigenfaces, let's code a face recognition algorithm using scikit-learn! First, we'll need a dataset. For our purposes, we'll use an out-of-the-box dataset by the University of Massachusetts called Labeled Faces in the Wild (LFW). Feel free to substitute your own dataset! If you want to create your own face dataset, you'll need several pictures of each person's face (at different angles and lighting), along with the ground-truth labels. The wider variety of faces you use, the better the recognizer will do. The easiest way to create a dataset for face recognition is to create a folder for each person and put the face images in there. Make sure each are the same size and resize them so they aren't large images! Remember that PCA will reduce the image's dimensionality when we project onto that space anyways so using large, high-definition images won't help and will slow down our algorithm. A good size is  $\sim 512 \times 512$  for each image. The images should all be the same size so you can store them in one numpy array with dimensions (num\_examples, height, width) . (We're assuming grayscale images). Then use the folder names to disambiguate classes. Using this approach, you can use your own images.

However, we'll be using the LFW dataset. Luckily, scikit-learn can automatically load our dataset for us in the correct format. We can call a function to load our data. If the data aren't available on disk, scikit-learn will automatically download them for us from the University of Massachusetts' website.

```

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_lfw_people
from sklearn.metrics import classification_report
from sklearn.decomposition import PCA
from sklearn.neural_network import MLPClassifier

# Load data
lfw_dataset = fetch_lfw_people(min_faces_per_person=100)

_, h, w = lfw_dataset.images.shape
X = lfw_dataset.data
y = lfw_dataset.target
target_names = lfw_dataset.target_names

# split into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

```

The argument to our function just prunes all people without at least 100 faces, thus reducing the number of classes. Then we can extract our dataset and other auxiliary information. Finally, we split our dataset into training and testing sets.

Now we can simply use scikit-learn's PCA class to perform the dimensionality reduction for us! We have to select the number of components, i.e., the output dimensionality (the number of eigenvectors to project onto), that we want to reduce down to, and feel free to tweak this parameter to try to get the best result! We'll use 100 components. Additionally, we'll whiten our data, which is easy to do with a simple boolean flag! (Whitening just makes our resulting data have a unit variance, which has been shown to produce better results)

```

# Compute a PCA
n_components = 100
pca = PCA(n_components=n_components, whiten=True).fit(X_train)

# apply PCA transformation
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

```

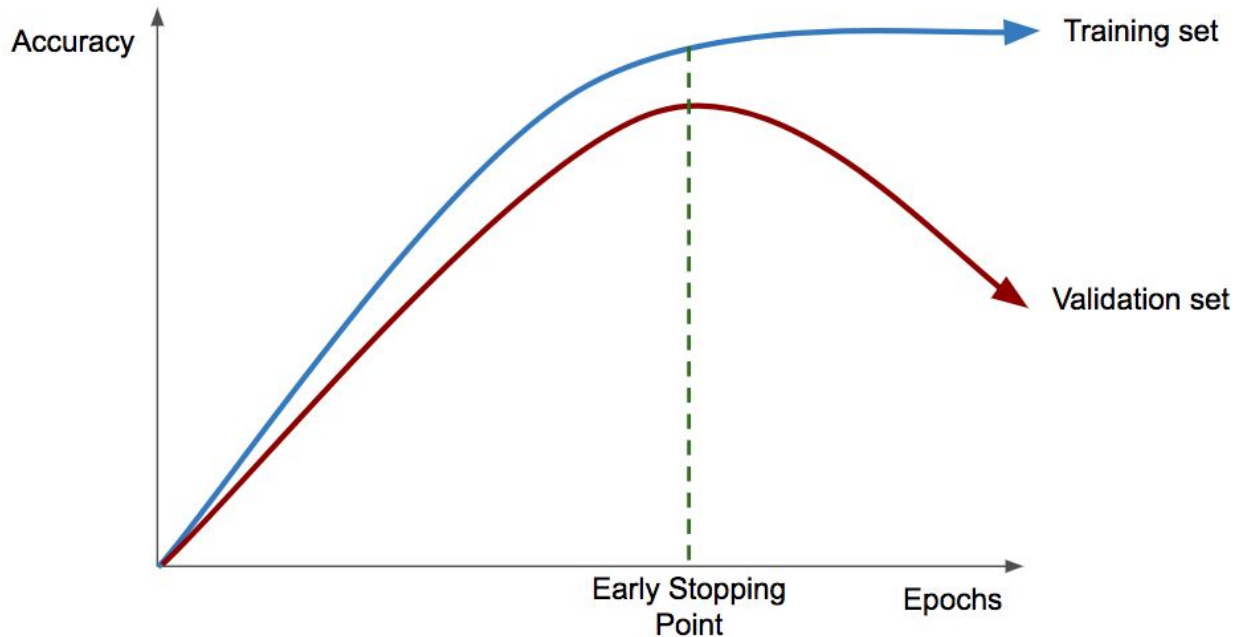
We can apply the transform to bring our images down to a 100-dimensional space. Notice we're not performing PCA on the entire dataset, only the training data. This is so we can better generalize to unseen data.

Now that we have a reduced-dimensionality vector, we can train our neural network!

```
# train a neural network
print("Fitting the classifier to the training set")
clf = MLPClassifier(hidden_layer_sizes=(1024,), batch_size=256,
verbose=True, early_stopping=True).fit(X_train_pca, y_train)
```

To see how our network is training, we can set the verbose flag. Additionally, we use **early stopping**.

Let's discuss early stopping as a brief aside. Essentially, our optimizer will monitor the average accuracy for the validation set for each epoch. If it notices that our validation accuracy hasn't increased significantly for a certain number of epochs, then we stop training. This is a **regularization technique** that prevents our model from overfitting!



Consider the above chart. We notice overfitting when our validation set accuracy starts to decline. At that point, we immediately stop training to prevent overfitting.

Finally, we can make a prediction and use a function to print out an entire report of quality for each class.

```
y_pred = clf.predict(X_test_pca)
print(classification_report(y_test, y_pred, target_names=target_names))
Here's an example of a classification report.
```

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

Colin Powell	0.86	0.89	0.87	66
Donald Rumsfeld	0.85	0.61	0.71	38
George W Bush	0.88	0.94	0.91	177
Gerhard Schroeder	0.67	0.69	0.68	26
Tony Blair	0.86	0.71	0.78	35
avg / total	0.85	0.85	0.85	342

Notice there is no accuracy metric. Accuracy isn't the most specific kind of metric to begin. Instead, we see precision, recall, f1-score, and support. The support is simply the number of times this ground truth label occurred in our test set, e.g., in our test set, there were actually 35 images of Tony Blair. The F1-Score is actually just computed from the precision and recall scores. Precision and recall are more specific measures than a single accuracy score. A higher value for both is better.

After training our classifier, we can give it a few images to classify.

```
# Visualization
def plot_gallery(images, titles, h, w, rows=3, cols=4):
    plt.figure()
    for i in range(rows * cols):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i])
        plt.xticks(())
        plt.yticks(())

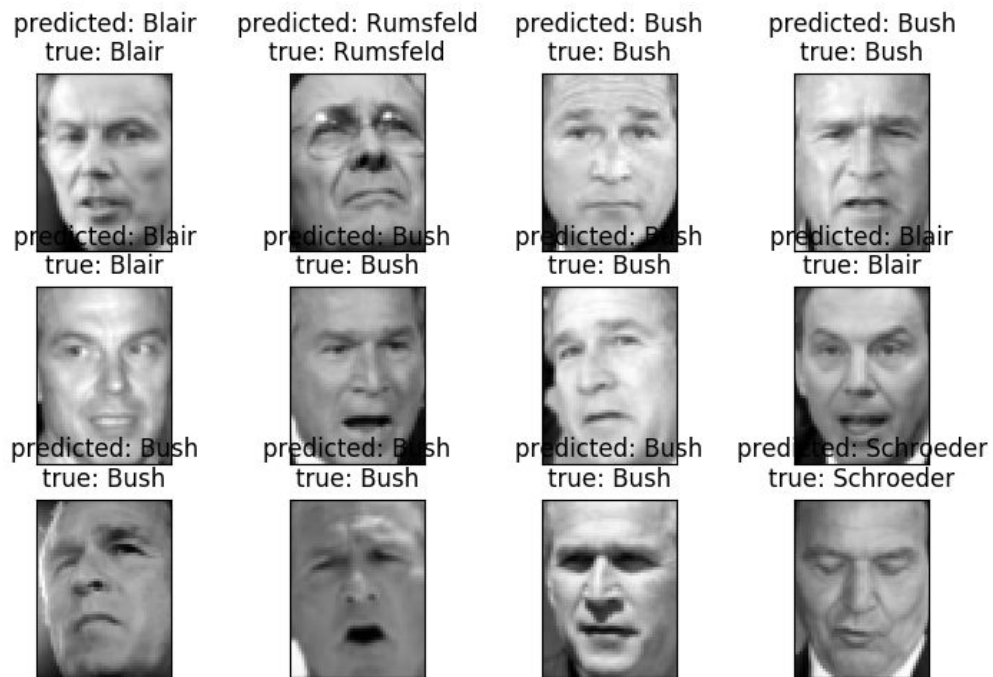
def titles(y_pred, y_test, target_names):
    for i in range(y_pred.shape[0]):
        pred_name = target_names[y_pred[i]].split(' ')[-1]
        true_name = target_names[y_test[i]].split(' ')[-1]
        yield 'predicted: {0}\ntrue: {1}'.format(pred_name, true_name)

prediction_titles = list(titles(y_pred, y_test, target_names))
plot_gallery(X_test, prediction_titles, h, w)
```

(plot\_gallery and titles functions modified from the scikit-learn documentation)

We can see our network's predictions and the ground truth value for each image.





Another thing interesting thing to visualize is are the eigenfaces themselves. Remember that PCA produces eigenvectors. We can reshape those eigenvectors into images and visualize the eigenfaces.





These represent the "generic" faces of our dataset. Intuitively, these are vectors that represent directions in "face space" and are what our neural network uses to help with classification. Now that we've discussed the eigenfaces approach, you can build applications that use this face recognition algorithm!

We discussed a popular approach to face recognition called **eigenfaces**. The essence of eigenfaces is an unsupervised dimensionality reduction algorithm called **Principal Components Analysis (PCA)** that we use to reduce the dimensionality of images into something smaller. Now that we have a smaller representation of our faces, we apply a classifier that takes the reduced-dimension input and produces a class label. For our classifier, we used a single-layer neural network.

Face recognition is a fascinating example of merging computer vision and machine learning and many researchers are still working on this challenging problem today! Nowadays, deep convolutional neural networks are used for face recognition. Try one out on this dataset!

## Decision Trees

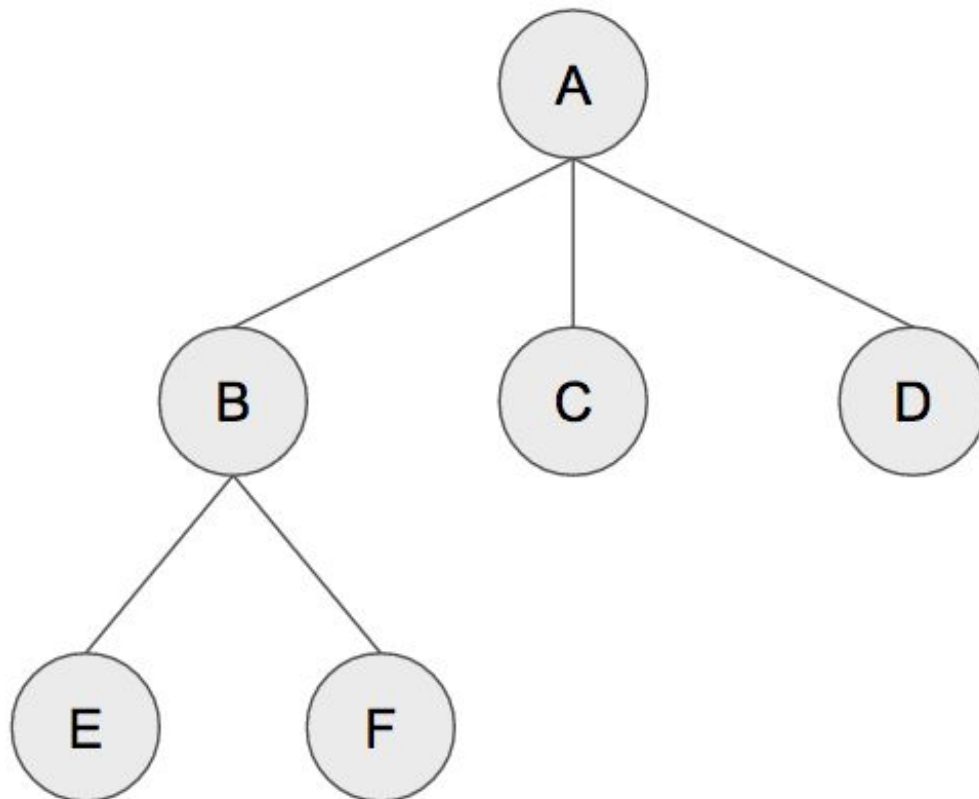
One challenge of neural or deep architectures is that it is difficult to determine what exactly is going on in the machine learning algorithm that makes a classifier decide how to classify inputs. This is a huge problem in deep learning: we can get fantastic classification accuracies, but we don't really know what criteria a classifier uses to make its classification decision. However, **decision trees** can present us with a graphical representation of how the classifier reaches its decision.

We'll be discussing the CART (Classification and Regression Trees) framework, which creates decision trees. First, we'll introduce the concept of decision trees, then we'll discuss each component of the CART framework to better understand how decision trees are generated.

Download the full code [here](#).

### Trees and Binary Trees

Before discussing decision trees, we should first get comfortable with trees, specifically binary trees. A **tree** is just a bunch of nodes connected through edges that satisfies one property: no loops!

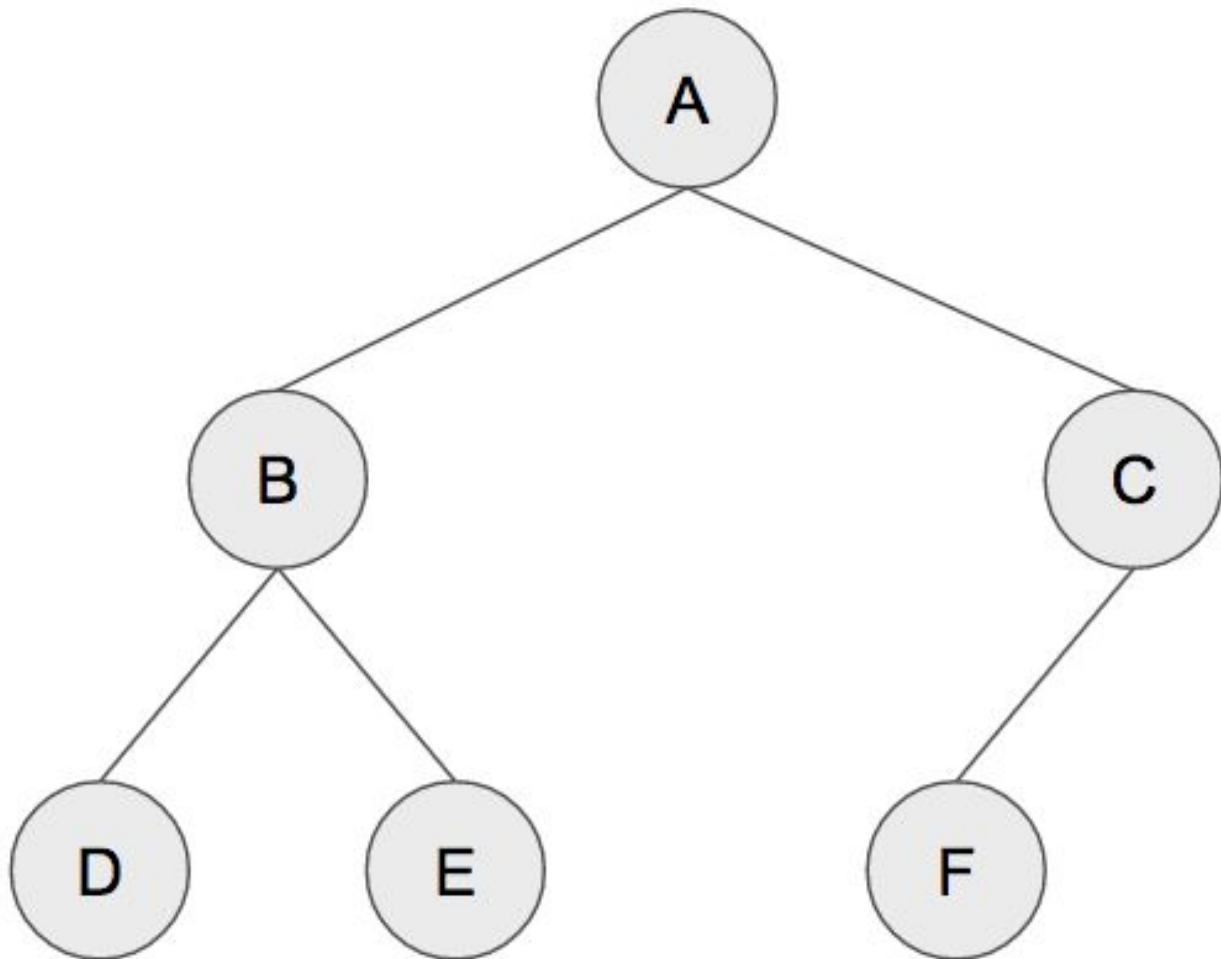


The above is an example of a tree. The nodes are A, B, C, D, E, and F. The edges are the lines that connect the nodes. The only rule we have to follow for this to be a valid tree is that it cannot have any loops or circuits.

Speaking of Node A, we consider it to be the **root node**, or our starting point, in other words. We conventionally pick the root node to be the node at the top of the tree. (Technically, any node in a tree can be the root). Node A has three **children**: Node B, Node C, and Node D. Child nodes are connected to the **parent nodes** through an edge. The only node without a parent is the root node. One other bit of terminology: Node E, Node F, Node C, and Node D are called **leaf nodes** because they are at the very bottom of the tree and have no children.

The **depth** of a tree is defined to be the number of levels, not including the root node. The tree above has a depth of 2 since Node B, Node C, and Node D are all on one level and Node E and Node F are on another level. In other words, it is the number of edges we have to traverse from the root to the farthest leaf node. Although Node A, the root, is on its own level, we usually do not include it when counting the depth of a tree. (If we have to consider it, we call it level 0.)

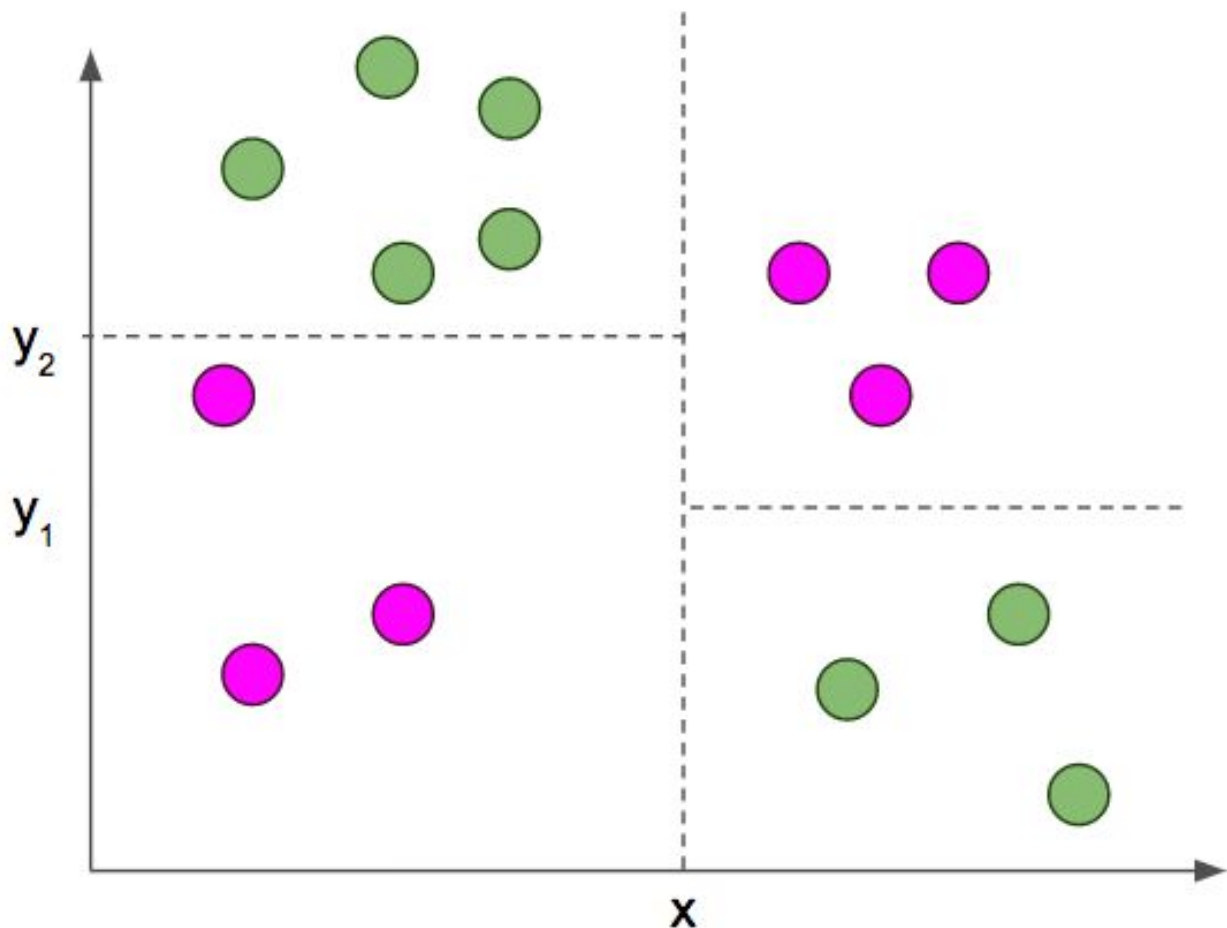
The tree above is a general tree, but it is not a **binary tree**. Decision trees are binary trees. Below is an example of a binary tree.



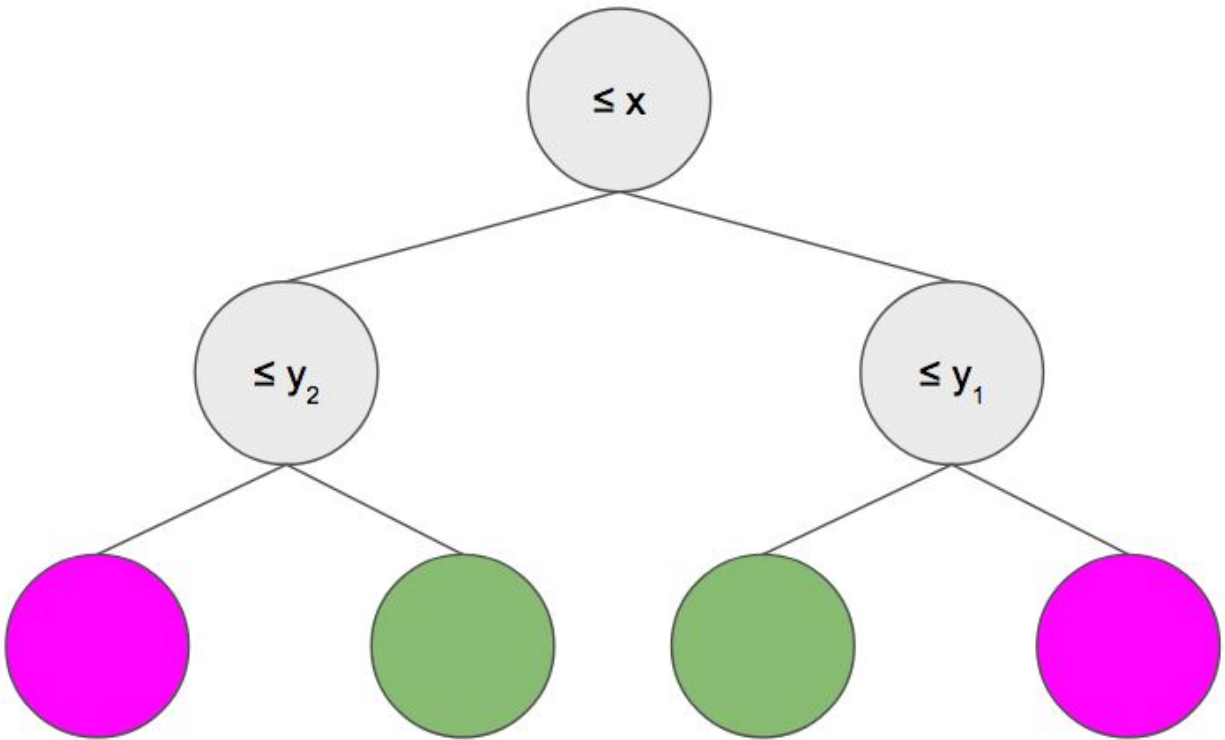
For a tree to be considered a **binary tree**, each parent node must have *at most* 2 child nodes. The above satisfies that condition. Parent nodes can have 0, 1, or 2 child nodes, but no greater than 2!

## Decision Trees

Now that we have a basic understanding of binary trees, we can discuss decision trees. A **decision tree** is a kind of machine learning algorithm that can be used for classification or regression. We'll be discussing it for classification, but it can certainly be used for regression. A decision tree classifies inputs by segmenting the input space into regions. Let's consider the following data.



We can partition the 2D plane into regions where the points in each region belong to the same class. The splits or partitions are denoted with dashed lines: there's one at  $x$ ,  $y_1$ , and  $y_2$ . This is an example segmentation that the decision tree might make. Under the hood, the decision tree represents this as a binary tree! Here's the decision tree that corresponds to the above segmentation.



The structure, number of nodes, and positioning of the edges of our decision tree is not known a-priori but is built from our training data. We'll soon discuss how we can create the tree from scratch using the CART framework. For now, let's suppose that our decision tree is already created.

To classify a new input point, we simply traverse down the tree. At each node in the decision tree, we ask a question about our data point. For example, at the root node, we ask "is the x coordinate of our data point less than  $x$ "? If it is, then we branch left. If it isn't, we branch right. In general, if the condition at the node is met, we go left; if it is not true, then we go right. Suppose that the x coordinate of our node is indeed less than  $x$ , so we branch left. Now we ask "is the y coordinate of our node less than  $y_2$ ? Let's suppose that it isn't. In this case, we go right and end up at a leaf node. The leaf nodes of a decision tree represent which class we assign to an input point. In our example, we know that our test point is in the green class. In our 2D plot, this particular test point is in the top-left region.

There are some things to keep in mind about this demonstration of a decision tree. In our simple example, we alternated between considering the x coordinate and the y coordinate, but there's no rule that says we have to do this. It is possible to have trees where we only consider a single dimension/feature for a few levels in the tree. Actually, decision trees work for data of any dimension, not just 2D data! Our simple example creates a segmentation where each region has 100% accuracy. While this is technically possible to do with a decision tree, it means we have overfit! We'll discuss how to fix this when we talk about the CART framework.

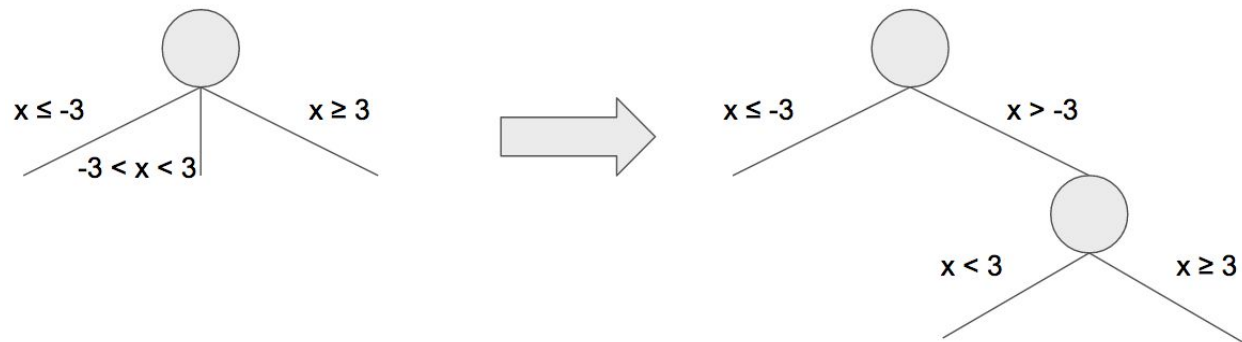
## Classification and Regression Trees (CART) Framework

Now that we have an intuitive understanding of how we use decision trees to classify points, let's discuss how we build the tree in the first place! There are four points we have to consider:

1. How many splits per node?
2. Which dimension do we test?
3. When do we stop?
4. How do we assign class labels to the leaf nodes?

The first point is easy to address: we always use two splits at each node. Recall that a decision tree is a binary tree, so we need to make sure that there are no more than two splits per node.

But suppose we had several cases for a node, such as on the left. We have three regions:  $(-\infty, -3]$ ,  $(-3, 3)$ , and  $[3, \infty)$ .

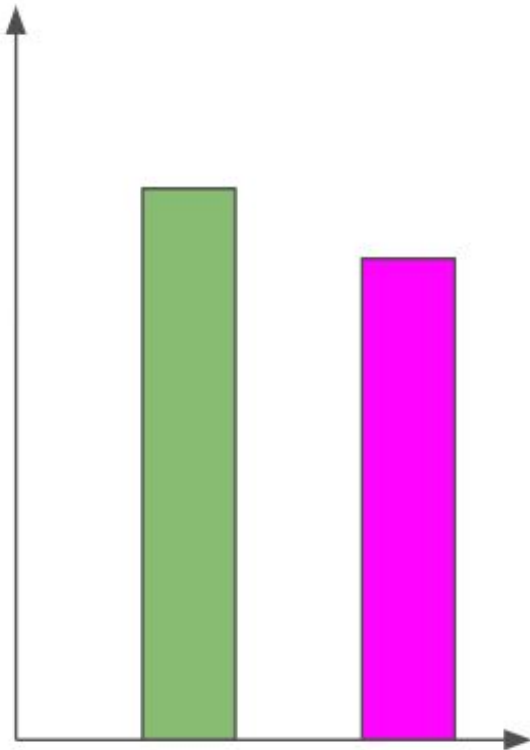


We can still split it into two binary decisions, like on the right. First, split into  $(-\infty, -3]$  and  $(-3, \infty)$ . Then we can split  $(-3, \infty)$  into two regions:  $(-3, 3)$  and  $[3, \infty)$ . We can do this for any decision with any number of cases: split it into a sequence of binary decisions.

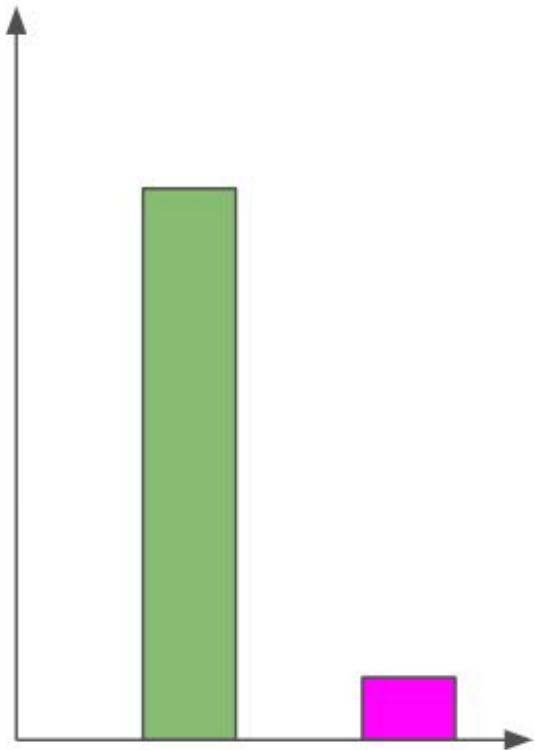
But how do we even know which dimensions to check? Ideally, we want to choose values so that the resulting split causes one class to be much more present than the others. When we only have training data, at each node, we'll be keeping track of how many examples of each class are present at a node as a vector  $[g, m]$  where  $g$  is the number of green examples and  $m$  is the number of magenta examples. We want to make splits so that we have many examples of only one particular class and few examples of the other.

In other words, we want the split that decreases the **entropy**. High entropy means that we have a mix of different classes; low entropy means that we have predominantly one class.

## High Entropy



## Low Entropy



We want to make the split so that we decrease the entropy: the resulting split causes us to have many of one class and few of the other. Ideally, we want our nodes to have no entropy, i.e., all examples at this node are definitely of one class. This low entropy is desirable at the leaf nodes since, when we classify an example, we can be very sure of its class in a low entropy leaf node.

Mathematically, there are several ways to measure this. One way is using the definition of entropy.

$$S(N) = - \sum_i p_i \log_2 p_i$$

This is computing the entropy at node  $N$  by summing over all classes  $i$  and computing  $p_i \log_2 p_i$  where  $p_i$  is the proportion of examples that belong to class  $i$  at node  $N$ .

There is another, more commonly used metric called Gini impurity. Impurity and entropy mean the same thing: we want lower Gini impurity. We can compute the Gini impurity using the following.

$$G(N) = \sum_i p_i (1 - p_i)$$

We can check each one in order to make our split with categorical dimensions. This is trickier if we have continuous features. Specific algorithms use different techniques, such as considering only values that the features take or sampling along a dimension.

Now let's address the third point: when do we stop splitting? Using a decision tree, we can keep splitting until each leaf node only has a single training example: that would be zero entropy/impurity! However, think of what this would look like graphically: we would have many small regions. In other words, we would overfit very much! On the other hand, we don't want to stop splitting too early!

The solution to this is to fully build out the decision tree so that we overfit. Then, we can prune nodes, starting from the leaves. We compute the increase in entropy/impurity at the parent if we were to prune two child nodes, and perform the pruning if that entropy/impurity increase is below some constant or threshold. Varying this constant affects the depth of our decision tree: a small value won't prune that much, but a large value will more aggressively prune the decision tree.

Finally, we have to address how we assign class labels to a leaf node. Ideally, at the leaf node, we'll have zero entropy/impurity, and we simply select the only available class. However, in many cases, we won't be that lucky. We might have a few examples from each class. There are several ways of handling this. A common thing to do is to randomly sample from the resulting distribution at that leaf node. We can also consider the total number of each class, i.e., we may have more green training examples than magenta, when assigning class labels.

Using the CART framework, we can build our human-readable decision tree!

## Decision Tree Code

We'll build and visualize a decision tree on the iris dataset. With scikit-learn, this is very easy to do!

```
from sklearn.datasets import load_iris
from sklearn import tree
from sklearn.metrics import classification_report, accuracy_score
from sklearn.model_selection import train_test_split
import graphviz

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.4, random_state=17)

clf = tree.DecisionTreeClassifier(random_state=17)
clf = clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred,
```



```
target_names=iris.target_names))  
print('\nAccuracy: {0:.4f}'.format(accuracy_score(y_test, y_pred)))
```

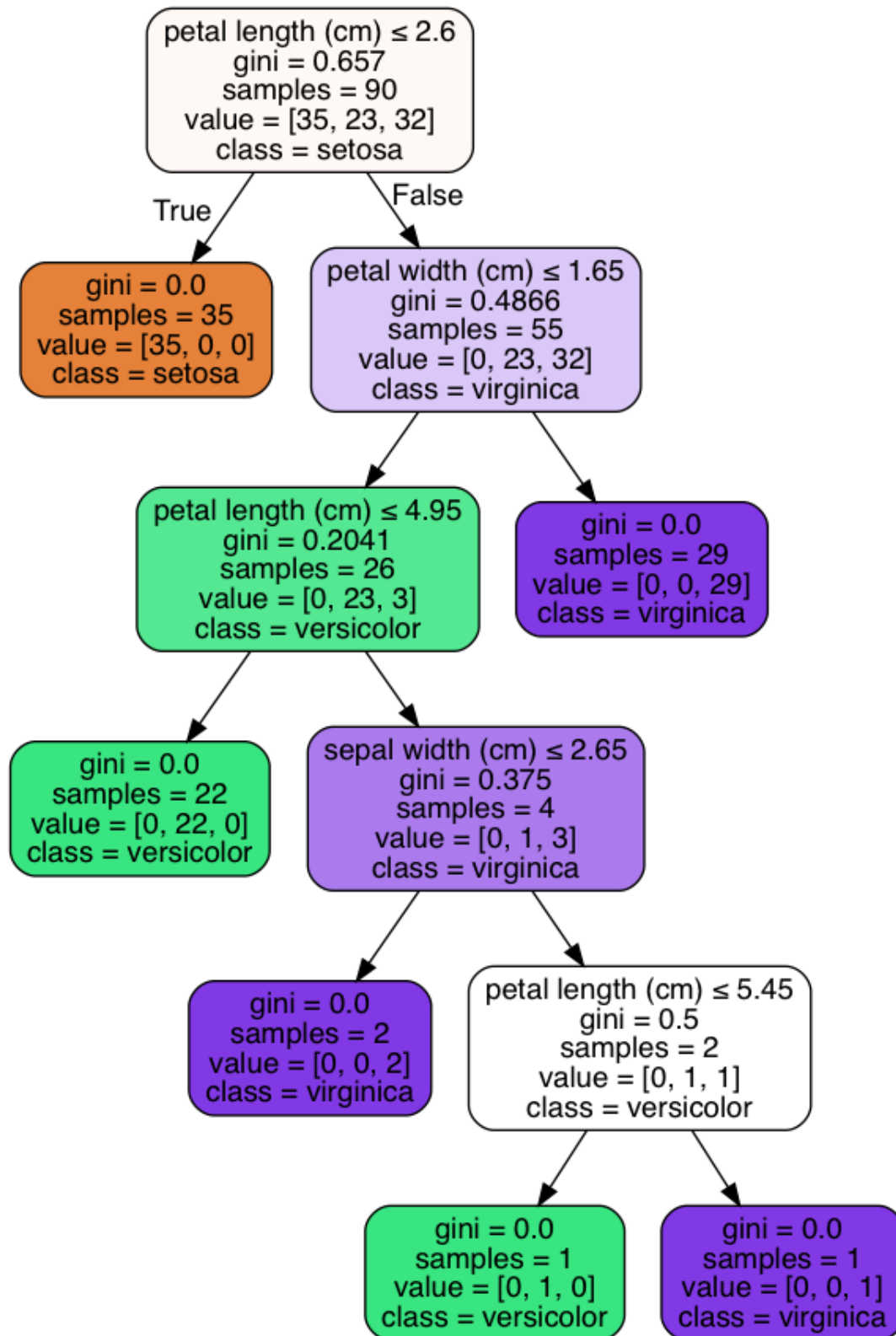
We load the iris dataset and split it into training and testing data. We can then construct a decision tree classifier and train! We evaluate on the test data and print the detailed precision, recall, and F1 score. We also compute the accuracy (around 95%).

To visualize the decision tree, we'll need to install graphviz (**brew install graphviz** or **sudo apt-get install graphviz** ).

```
dot = tree.export_graphviz(clf, out_file=None,  
feature_names=iris.feature_names, class_names=iris.target_names,  
filled=True, rounded=True, special_characters=True)  
  
graph = graphviz.Source(dot)  
graph.format = 'png'  
graph.render('iris', view=True)
```

We're using the feature names of the dataset so that we don't reference the dimension number (remember that the iris dataset has 4 features: petal width, petal length, sepal width, and sepal length).

We should see the following image in the same directory as the Python file.

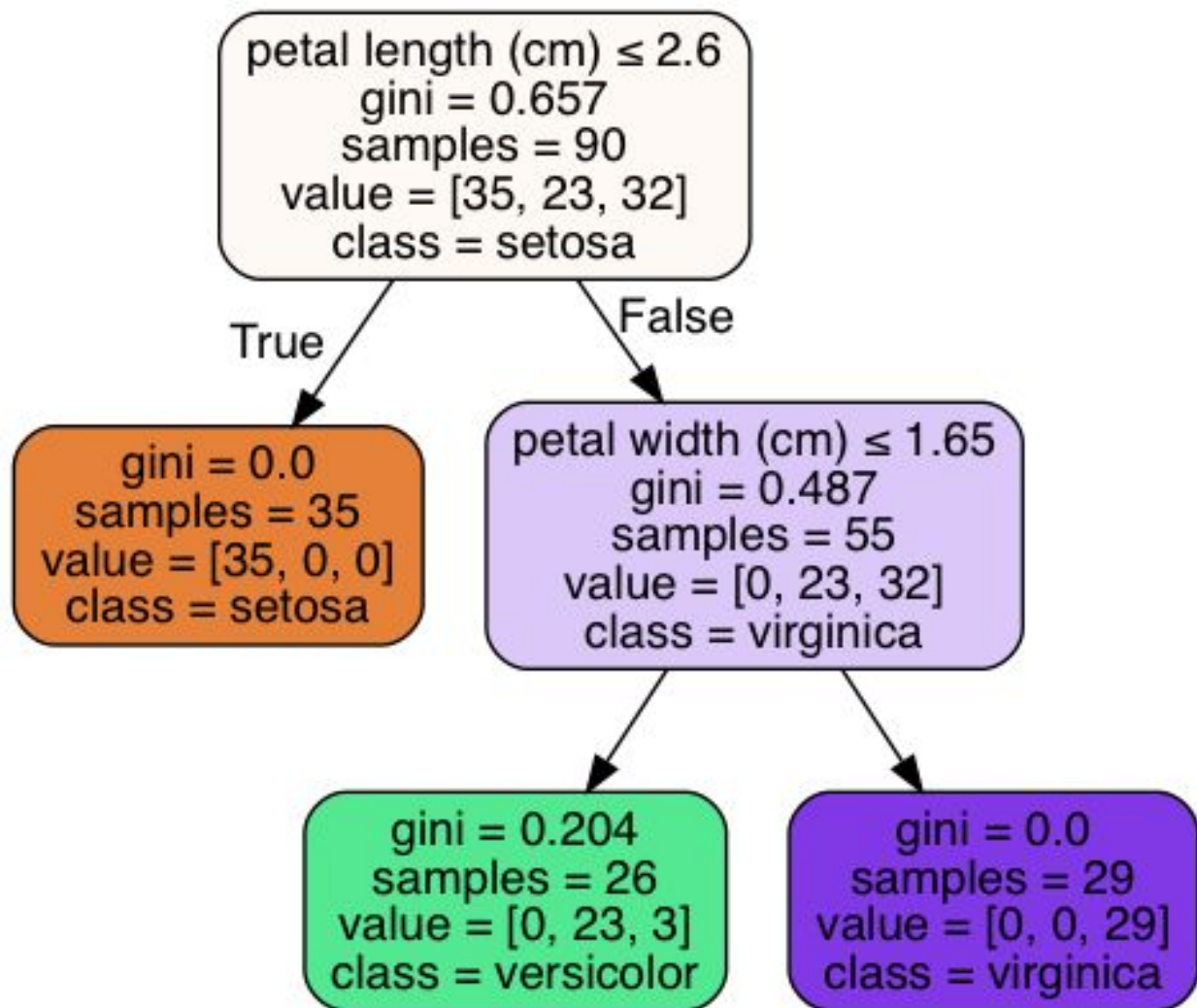


At each node, we ask a question about the features. If the answer to the question is yes, we go left, if not, we go right. Samples tells us how many examples are at that node, and the

value is that vector of samples for each class. We're using the Gini impurity as our metric, and notice how it is zero at each leaf node. Given test data, we can simply follow this tree to arrive at the class label!

Now let's mess with the minimum entropy threshold parameter to see if we can prune this tree and how that affects the accuracy. Let's set it to 0.1. This will tell our classifier that it's okay to prune children such that the entropy/impurity at the parent after pruning is less than 0.1.  
`clf = tree.DecisionTreeClassifier(random_state=17, min_impurity_decrease=0.1)`

When we do this, we get about the same accuracy, but our tree is much smaller!



If we set this parameter to be too large, then everything will collapse into a single node with a poor accuracy. Remember that the larger the value, the more aggressive pruning! To summarize, we discussed how to use build and use decision trees. They give us a very human-friendly way of interpreting how they make classification decisions. We first discussed trees and binary trees, then built the intuition behind how to use decision trees. We discussed how to build a decision tree using the Classification and Regression Tree (CART) framework. Finally, we used a decision tree on the iris dataset.

**Decision Trees** are one of the few machine learning algorithms that produces a comprehensible understanding of how the algorithm makes decisions under the hood.

# Dimensionality Reduction

Dimensionality Reduction is a powerful technique that is widely used in data analytics and data science to help visualize data, select good features, and to train models efficiently. We use **dimensionality reduction** to take higher-dimensional data and represent it in a lower dimension. We'll discuss some of the most popular types of dimensionality reduction, such as principal components analysis, linear discriminant analysis, and t-distributed stochastic neighbor embedding. We'll use these techniques to project the MNIST handwritten digits dataset of images into 2D and compare the resulting visualizations.

Download the full code [here](#).

## Handwritten Digits: The MNIST Dataset

Before discussing the motivation behind dimensionality reduction, let's take a look at the MNIST dataset. We'll be using it as a running example.



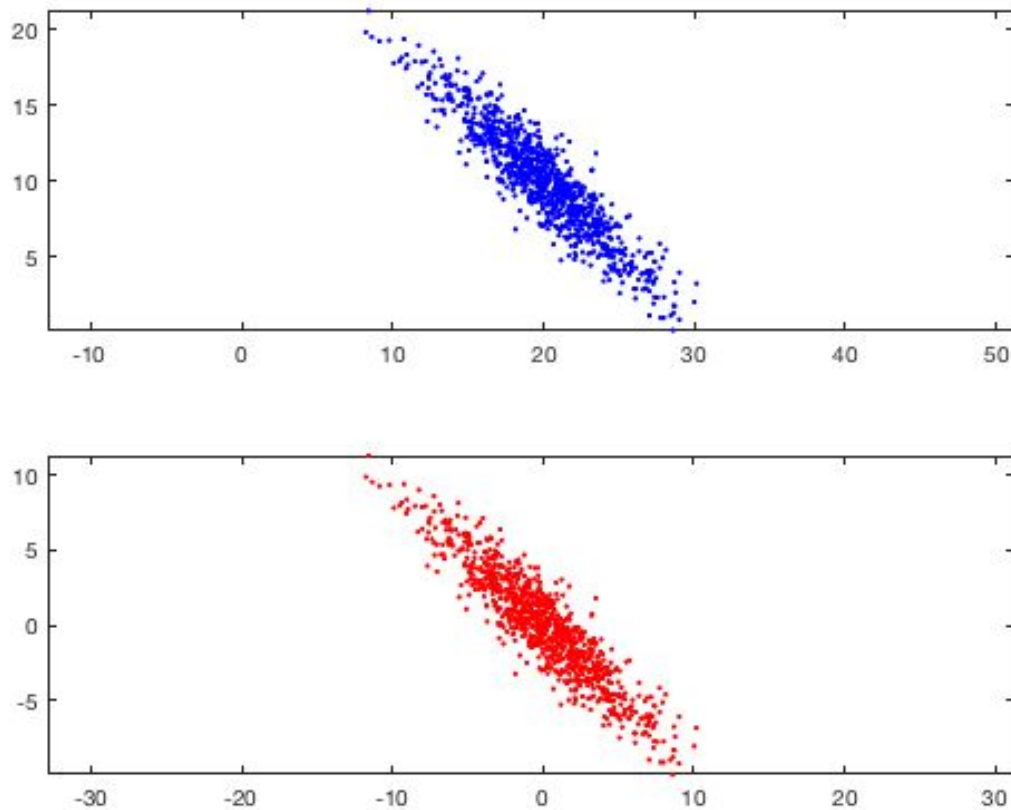
The MNIST handwritten digits dataset consists of binary images of a single handwritten digit (0-9) of size  $28 \times 28$ . The provided training set has 60,000 images, and the testing set has 10,000 images.

We can think of each digit as a point in a higher-dimensional space. If we take an image from this dataset and rasterize it into a  $784 \times 1$  vector, then it becomes a point in 784-dimensional space. That's impossible to visualize in that higher space!

These kinds of higher dimensions are quite common in data science. Each dimension represents a feature. For example, suppose we wanted to build a naive dog breed classifier. Our features may be something like height, weight, length, fur color, and so on. Each one of these becomes a dimension in the vector that represents a single dog. That vector is then a point in a higher-dimensional space, just like our MNIST dataset!

## Dimensionality Reduction

Dimensionality reduction is a type of learning where we want to take higher-dimensional data, like images, and represent them in a lower-dimensional space. Let's use the following data as an example.



(These plots show the same data, except the bottom chart zero-centers it.)

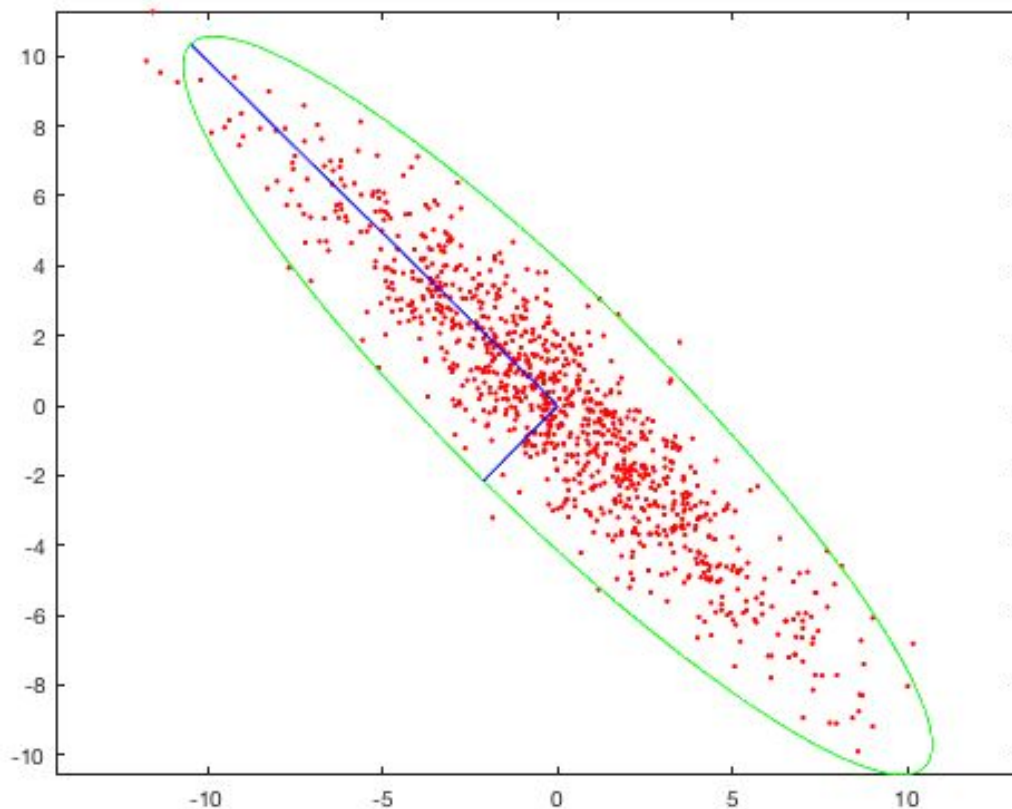
With these data, we can use a dimensionality reduction to reduce them from a 2D plane to a 1D line. If we had 3D data, we could reduce them down to a 2D plane, and then to a 1D line.

Most dimensionality reduction techniques aim to find some **hyperplane**, which is just a higher-dimensional version of a line, to *project* the points onto. We can imagine a *projection* as taking a flashlight perpendicular to the hyperplane we're projecting onto and plotting where the shadows fall on that hyperplane. For example, in our above data, if we wanted to project our points onto the x-axis, then we pretend each point is a ball and our flashlight would point directly down or up (perpendicular to the x-axis) and the shadows of the points would fall on the x-axis. This is what we call a **projection**. We won't worry about the exact math behind this since scikit-learn can apply this projection for us.

In our simple 2D case, we want to find a line to project our points onto. After we project the points, then we have data in 1D instead of 2D! Similarly, if we had 3D data, we would want to find a plane to project the points down onto to reduce the dimensionality of our data from 3D to 2D. The different types of dimensionality reduction are all about figuring out which of these hyperplanes to select: there are an infinite number of them!

## Principal Component Analysis

One technique of dimensionality reduction is called **principal component analysis (PCA)**. The idea behind PCA is that we want to select the hyperplane such that, when all the points are projected onto it, they are maximally spread out. In other words, we want the *axis of maximal variance*! Let's consider our example plot above. A potential axis is the x-axis or y-axis, but, in both cases, that's not the best axis. However, if we pick a line that has the same diagonal orientation as our data, that is the axis where the data would be most spread!



The longer blue axis is the correct axis! (The shorter blue axis is for visualization only and is perpendicular to the longer one.) If we were to project our points onto this axis, they would be maximally spread! But how do we figure out this axis? We can use a linear algebra concept called **eigenvectors**! Essentially, we compute the covariance matrix of our data and consider that covariance matrix's largest **eigenvectors**. Those are our *principal axes* and the axes that we project our data onto to reduce dimensions.

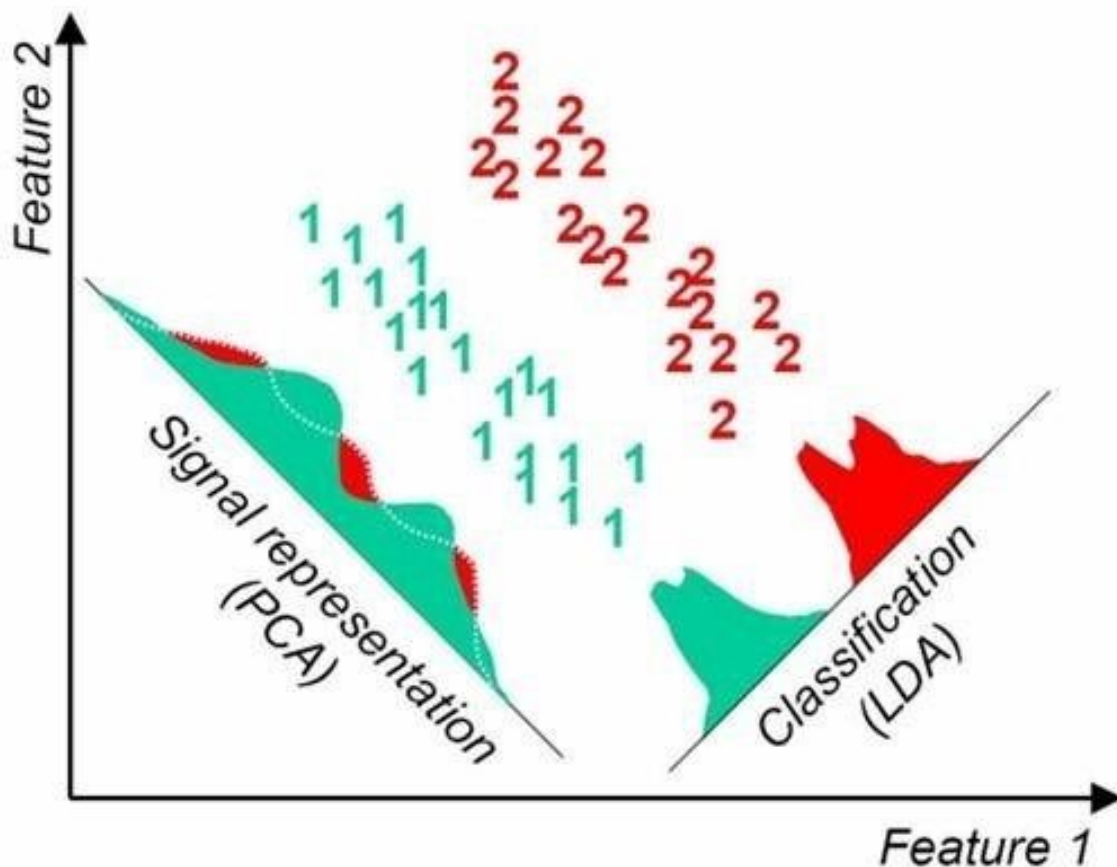


Using this approach, we can take high-dimensional data and reduce it down to a lower dimension by selecting the largest eigenvectors of the covariance matrix and projecting onto those eigenvectors.

### Linear Discriminant Analysis

Another type of dimensionality reduction technique is called **linear discriminant analysis (LDA)**. Similar to PCA, we want to find the best hyperplane and project our data onto it. However, there is one big distinction: *LDA is supervised!* With PCA, we were using eigenvectors from our data to figure out the axis of maximum variance. However, with LDA, we want the *axis of maximum class separation*! In other words, we want the axis that separates the classes with the maximum margin of separation.

The following figure shows the difference between PCA and LDA.



(Source: <https://algorithmsdatascience.quora.com/PCA-4-LDA-Linear-Discriminant-Analysis>)

With LDA, we choose the axis so that Class 1 and Class 2 are maximally separated, i.e., the distance between their means is maximal. We must have class labels for LDA because we need to compute the mean of each class to figure out the optimal plane.



It's important to note that LDA does make some assumptions about our data. In particular, it assumes that the data for our classes are normally distributed (Gaussian distribution). We can still use LDA on data that isn't normally distributed, but we may not find an optimal hyperplane. Another assumption is that the covariances of each class are the same. In reality, this also might not be the case, but LDA will still work fairly well. We should keep these assumptions in mind when using LDA on any set of data.

### **t-Distributed Stochastic Neighbor Embedding (t-SNE)**

A more recent dimensionality reduction technique that's been widely adopted is **t-Distributed Stochastic Neighbor Embedding (t-SNE)** by Laurens Van Der Maaten (2008). t-SNE fundamentally differs from PCA and LDA because it is *probabilistic*! Both PCA and LDA are deterministic, but t-SNE is stochastic, or probabilistic.

At a high level, t-SNE aims to minimize the divergence between two distributions: the pairwise similarity of the points in the higher-dimensional space and the pairwise similarity of the points in the lower-dimensional space.

To measure similarity, we use the **Student's t-distribution or Cauchy Distribution**! This is a distribution that looks very similar to a Gaussian, *but it is not the Gaussian distribution*! To compute the similarity between two points  $x_i$  and  $x_j$ , we use *probabilities*:

$$p_{j|i} = \frac{\exp(-||x_i - x_j||^2 / 2\sigma_i^2)}{\sum_{k=1}^N \exp(-||x_i - x_k||^2 / 2\sigma_i^2)}$$

where N is the number of data points and  $k \neq i$ . In other words, this equation is telling us the likelihood that  $x_i$  would choose  $x_j$  as its neighbor. Notice the t-distribution is centered around  $x_i$ . Intuitively, the farther  $x_j$  is from  $x_i$ , the smaller the probability becomes.

Similarly, we can compute the same quantity for the points in the lower-dimensional space.

$$q_{j|i} = \frac{\exp(-||y_i - y_j||^2 / 2\sigma_i^2)}{\sum_{k=1}^N \exp(-||y_i - y_k||^2 / 2\sigma_i^2)}$$

Now how do we measure the divergence between two distributions? We simply use the

### **Kullback-Leibler divergence (KLD).**

$$C = \sum_i KL(P_i || Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

This is our cost function! Now we can use a technique like gradient descent to train our model.

There's just one last thing to figure out:  $\sigma_i$ . We can't use the same  $\sigma$  for all points! Denser regions should have a smaller  $\sigma$ , and sparser regions should have a larger  $\sigma$ . We

solidify this intuition into a mathematical term called **perplexity**. Think of it as a measure of the effective number of neighbors, similar to the  $k$  of k-nearest neighbors.

$$\text{Perplexity}(P_i) = 2^{H(P_i)}$$

$$H(P_i) = - \sum_j p_{j|i} \log_2 p_{j|i}$$

t-SNE performs a binary search for the  $\sigma_i$  that produces a distribution  $P_i$  with the perplexity specified by the user: perplexity is a hyperparameter. Values between 5 and 50 tend to work the best.

In practice, t-SNE is very resource-intensive so we usually use another dimensionality reduction technique, like PCA, to reduce the input space into a smaller dimensionality (like maybe 50 dimensions), and *then* use t-SNE.

t-SNE, as we'll see, produces the best results out of all of the dimensionality reduction techniques because of the KLD cost function.

## Dimensionality Reduction Visualizations

Now that we've discussed a few popular dimensionality reduction techniques, let's apply them to our MNIST dataset and project our digits onto a 2D plane.

First, we need to import numpy, matplotlib, and scikit-learn and get the MNIST data. Scikit-learn already comes with the MNIST data (or will automatically download it for you) so we don't have to deal with uncompressing it ourselves! Additionally, I've provided a function that will produce a nice visualization of our data.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import offsetbox
from sklearn import manifold, datasets, decomposition,
discriminant_analysis

digits = datasets.load_digits()
X = digits.data
y = digits.target
n_samples, n_features = X.shape

def embedding_plot(X, title):
    x_min, x_max = np.min(X, axis=0), np.max(X, axis=0)
    X = (X - x_min) / (x_max - x_min)

    plt.figure()
    ax = plt.subplot(aspect='equal')
```

```

sc = ax.scatter(X[:,0], X[:,1], lw=0, s=40, c=y/10.)

shown_images = np.array([[1., 1.]])
for i in range(X.shape[0]):
    if np.min(np.sum((X[i] - shown_images) ** 2, axis=1)) < 1e-2:
        continue
    shown_images = np.r_[shown_images, [X[i]]]

ax.add_artist(offsetbox.AnnotationBbox(offsetbox.OffsetImage(digits.images[
i], cmap=plt.cm.gray_r), X[i]))

plt.xticks([], plt.yticks([]))
plt.title(title)

```

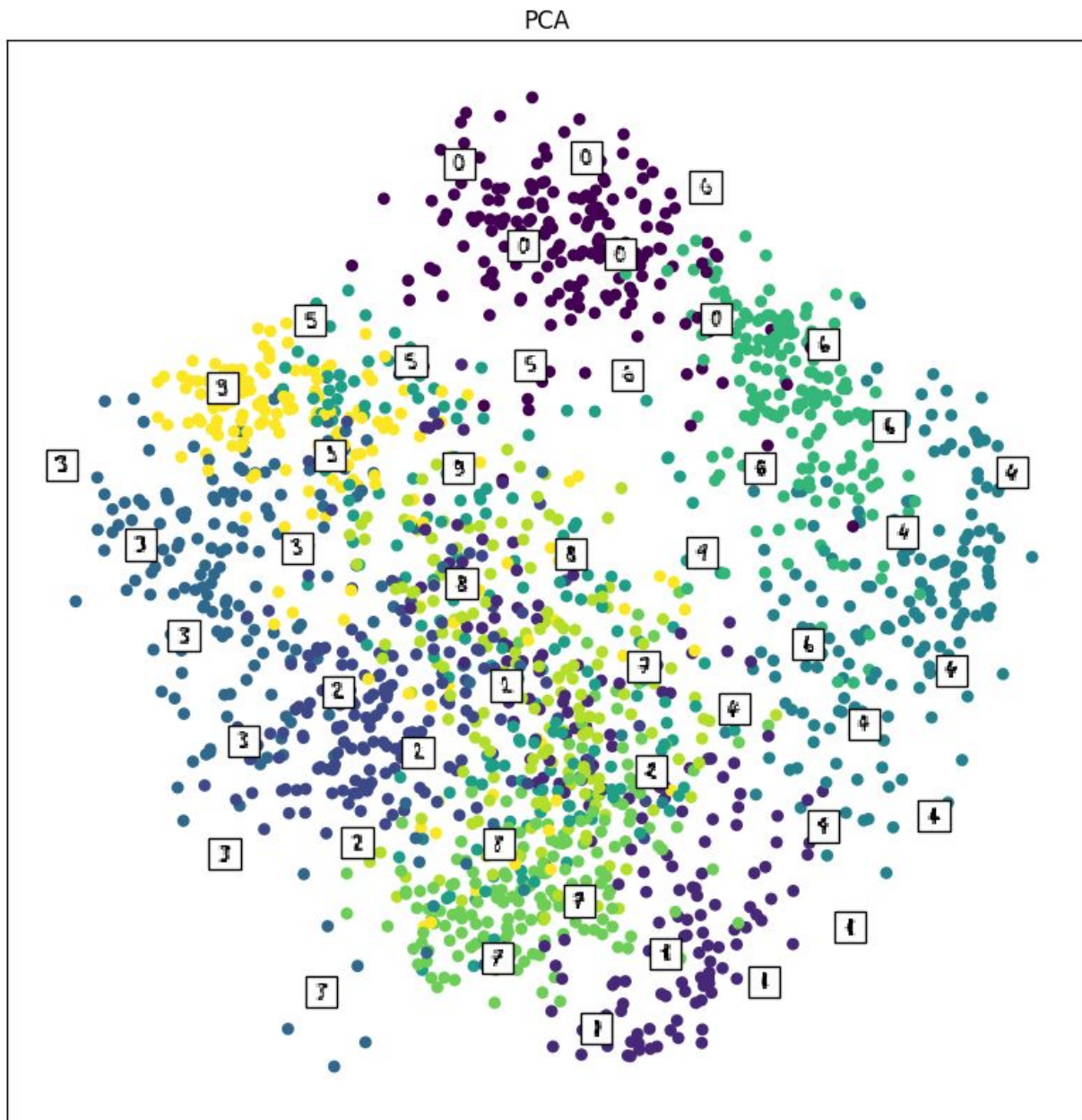
Using any of the dimensionality reduction techniques that we've discussed in scikit-learn is trivial! We can get PCA working in just a few lines of code!

```

X_pca = decomposition.PCA(n_components=2).fit_transform(X)
embedding_plot(X_pca, "PCA")
plt.show()

```

Below is the resulting plot.

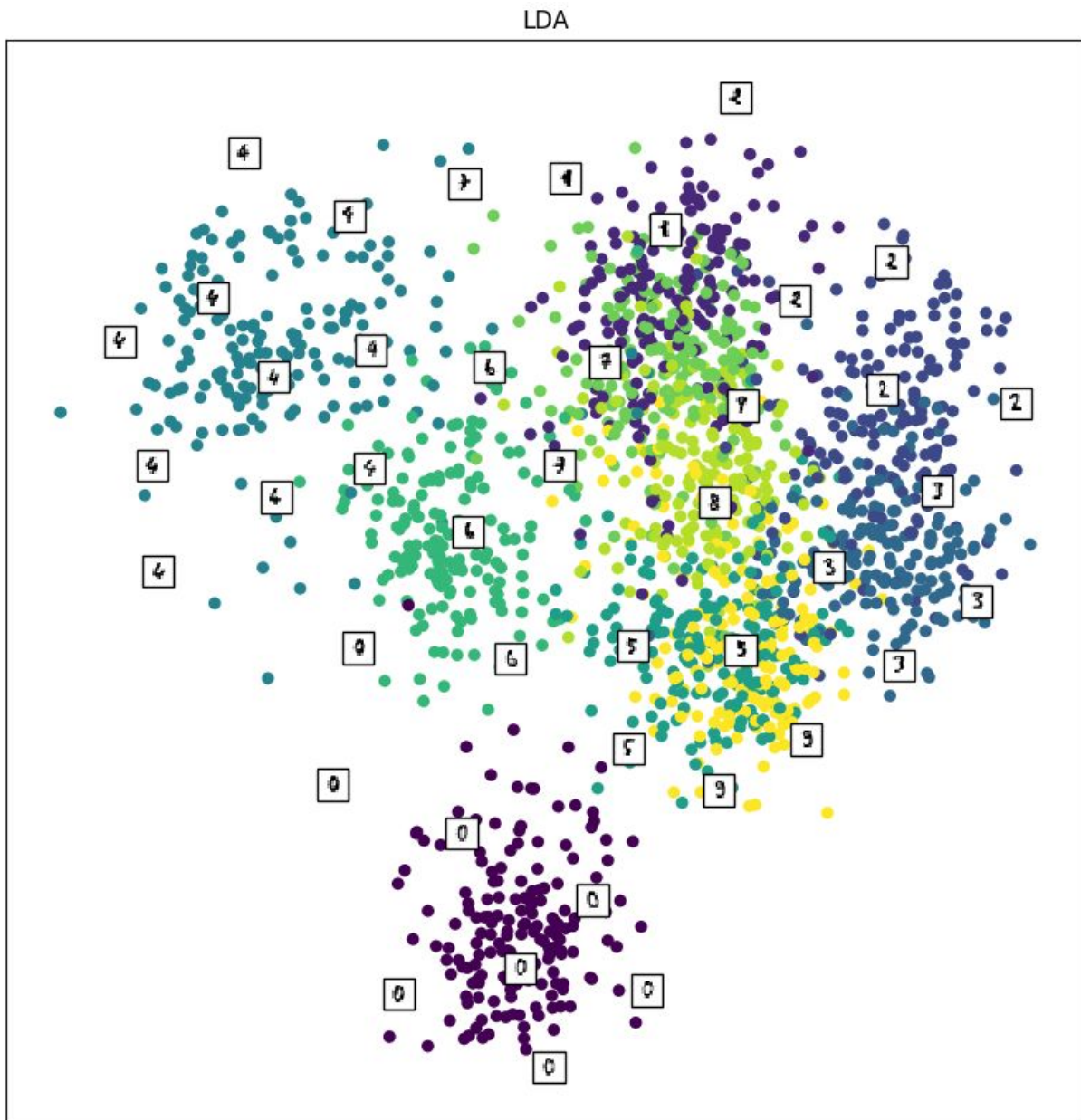


After taking a closer look at this plot, we notice something spectacular: similar digits are grouped together! If we think about it, this result makes sense. If the digits looked similar, when we rasterized them into a vector, the points must have been relatively close to each other. So when we project them down to the lower-dimensional space, we also expect them to be somewhat close together as well. However, PCA doesn't know anything about the class labels: it's going off of just the axis of maximal variance. Maybe LDA will work better since we can separate the classes in the lower-dimensional space better.

Now let's use LDA to visualize the same data. Just like PCA, using LDA in scikit-learn is very easy! Notice that we have to also give the class labels since LDA is *supervised*!

```
X_lda =  
discriminant_analysis.LinearDiscriminantAnalysis(n_components=2).fit_transform(X, y)  
embedding_plot(X_lda, "LDA")  
  
plt.show()
```

Below is the resulting plot from LDA.



This looks slightly better! We notice that the clusters are a bit farther apart. Consider the 0's: they're almost entirely separated from the rest! We also have clusters of 4's at the top left, 2's and 3's at the right, and 6's in the center. This is doing a better job at separating the digits in the lower-dimensional space. We can attribute this improvement to having information about the classes: remember that LDA is supervised! By knowing the correct labels, we can choose hyperplanes that better separate the classes. Let's see how t-SNE compares to LDA. (We may get a warning about collinear points. The reason for the warning is because we have to take a matrix inverse. Don't worry too much about it since we're just using this for visualization.)

Finally, let's use t-SNE to visualize the MNIST data. We're initializing the embedding to use PCA (in accordance with Laurens Van Der Maaten's recommendations). Unlike LDA, t-SNE is completely unsupervised.

```
tsne = manifold.TSNE(n_components=2, init='pca').fit_transform(X)
embedding_plot(X_tsne, "t-SNE")

plt.show()
```



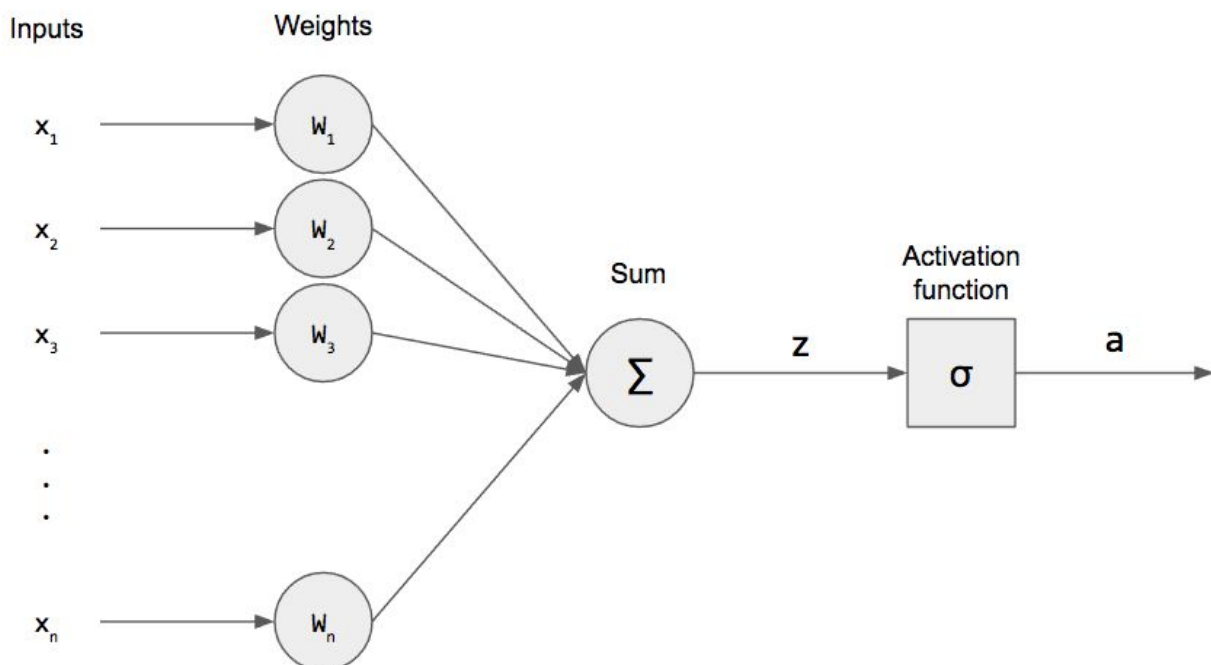
# Classification with Support Vector Machines

One of the most widely-used and robust classifiers is the **support vector machine**. Not only can it efficiently classify linear decision boundaries, but it can also classify non-linear boundaries and solve linearly inseparable problems. We'll be discussing the inner workings of this classification jack-of-all-trades. We first have to review the perceptron so we can talk about support vector machines. Then we'll derive the support vector machine problem for both linearly separable and inseparable problems. We'll discuss the **kernel trick**, and, finally, we'll see how varying parameters affects the decision boundary on the most popular classification dataset: the iris dataset.

Download the full code [here](#).

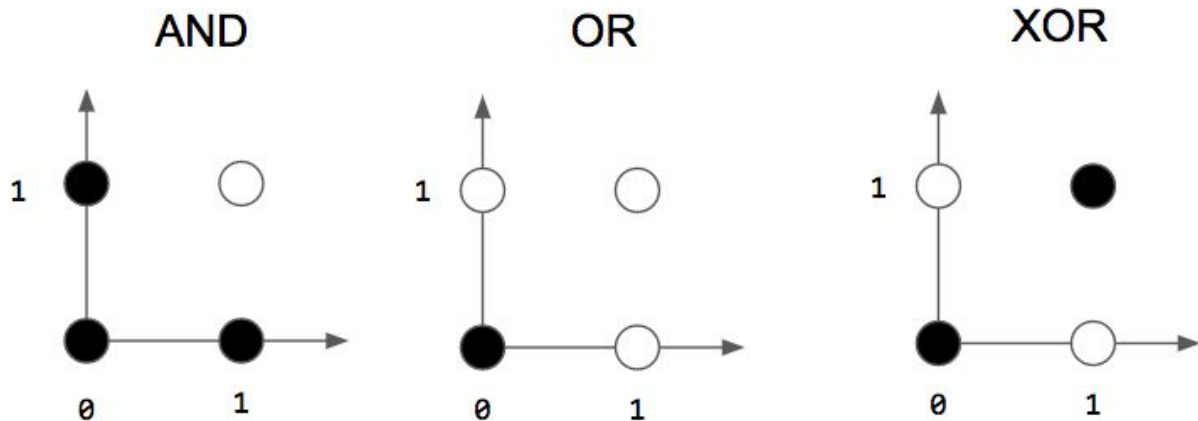
## Perceptron Review

Before continuing on to discuss support vector machines, let's take a moment to recap the perceptron.



The perceptron takes a weighted sum of its inputs and applies an activation function. To train a perceptron, we adjust the weights of the weighted sum. The activation function can be any number of things, such as the sigmoid, hyperbolic tangent (tanh), or rectified linear unit (ReLU). After applying the activation function, we get an activation out, and that activation is compared to the actual output to measure how well our perceptron is doing. If it didn't correctly classify our data, then we adjust the weights. We keep iterating over our training data until the

perceptron can correctly classify each of our examples (or we hit the maximum number of epochs).

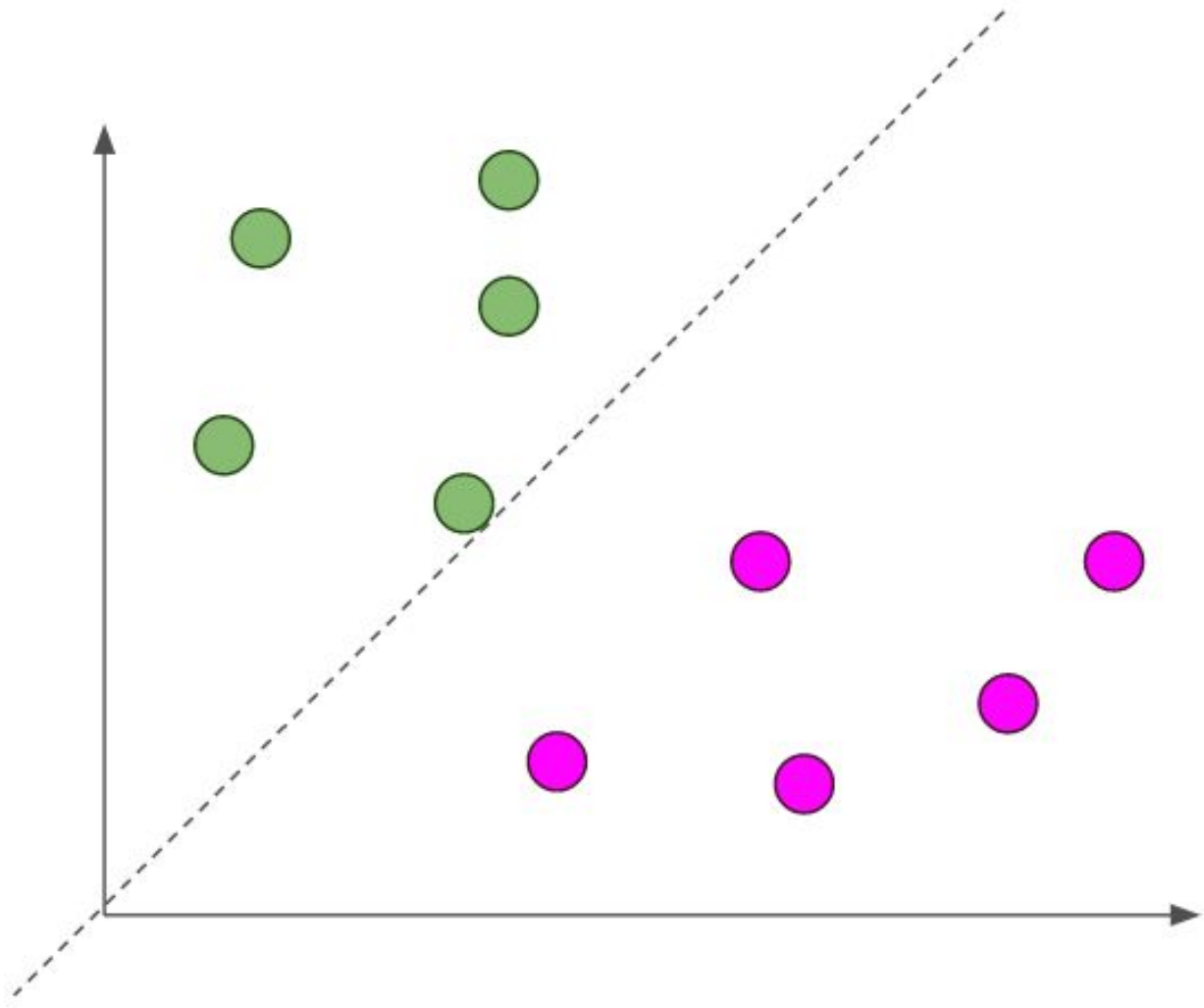


We trained our perceptron to solve logic gates but came to an important realization: the perceptron can only solve *linear* problems! In other words, the perceptron's weights create a line (or hyperplane)! This is the reason we can't use a single perceptron to solve the XOR problem. Let's discuss just linear problems for now. One of the most useful properties of the perceptron is the **perceptron convergence theorem**: for a linearly separable problem, the perceptron is *guaranteed* to find an answer in a finite amount of time.

However, there is one big catch: it finds the *first* line that correctly classifies all examples, not the *best* line. For any problem, if there is a single line that can correctly classify all training examples, there are an infinite number of lines that can separate the classes! These separating lines are also called **decision boundaries** because they determine the class based on which side of the boundary an example falls on.

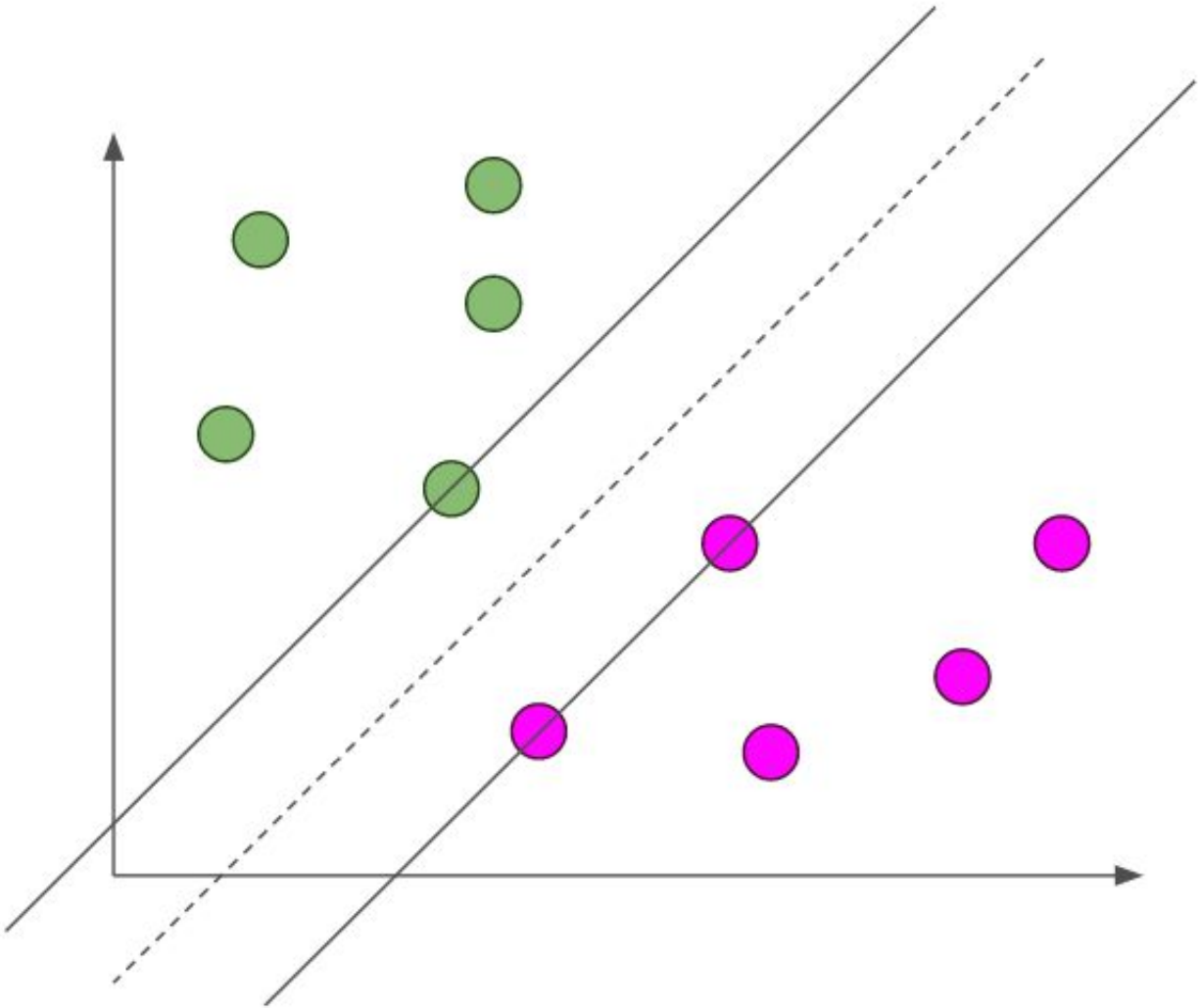
Let's see an example to make this more concrete. Suppose we had the given data for a binary classification problem. If we used a perceptron, we might get a decision boundary that looks like this.





This isn't the best decision boundary! The line is really close to all of our green examples and far from our magenta examples. If we get new examples, then we might have an example that's really close to the decision boundary, but on the magenta side. If I didn't draw that line, we would certainly think that the new point would be a green point. But, since it is on the other side of the decision boundary, even though it is closer to the green examples, our perceptron would classify it as a magenta point. This is not good!

If this decision boundary is bad, then where, among the infinite number of decision boundaries, is the *best* one? Our intuition tell us that the best decision boundary should probably be oriented in the exact middle of the two classes of data.



The dashed line is the decision boundary. This seems like a better fit! Now, if we have a new example that's really close to this decision boundary, we still can classify it correctly! But how do we find this best decision boundary?

## Support Vector Machines

The goal of support vector machines (SVMs) is to find the optimal line (or hyperplane) that *maximally separates the two classes*! (SVMs are used for binary classification, but can be extended to support multi-class classification). Mathematically, we can write the equation of that decision boundary as a line.

$$[g(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b = 0]$$

Note that we set this equal to zero because it is an *equation*. Depending on the value of  $g$  for a particular point  $\mathbf{x}$ , we can classify into the two classes. We're using vector notation to be as general as possible, but this works for a simple 2D (one input) case as well.

If we do some geometry, we can figure out that the distance from any point to the decision boundary is the following

$$r = \frac{g(\mathbf{x})}{\|\mathbf{w}\|}$$

Our goal is to maximize  $r$  for the points closest to the optimal decision boundary. These points are so important that they have a special name: **support vectors!**

We can actually simplify this goal a little bit by considering only the support vectors. Notice that the numerator just tells us which class (we're assuming the two classes are 1 and -1), but the denominator doesn't change. We can take the absolute value of each side to get rid of the numerator.

$$|r| = \frac{1}{\|\mathbf{w}\|}$$

where  $\mathbf{w}$  is the optimal decision boundary (later we'll show that the bias is easy to solve for if

we know  $\mathbf{w}$ ) We can simplify even further! Maximizing  $\frac{1}{\|\mathbf{w}\|}$  is equivalent to minimizing  $\|\mathbf{w}\|$ .

This is a bit tricky to do mathematically, so we can just square this to get  $\min \frac{1}{2} \mathbf{w}^T \mathbf{w}$ . (The constant out front is there so it can nicely cancel out later!)

However, we need more constraints, else we could just make  $\|\mathbf{w}\| = 0$ ! That wouldn't solve anything! The other constraints come from our need to correctly classify the examples!

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 \quad \forall i$$

where  $y_i$  is the ground truth and we iterate over our training set. To see why this is correct, let's split it into the two classes 1 and -1:

$$\mathbf{w}^T \mathbf{x}_i + b \geq 1 \quad \forall y_i = 1$$

$$\mathbf{w}^T \mathbf{x}_i + b \leq -1 \quad \forall y_i = -1$$

We can compress the two into the single equation above. After we've considered all of this, we can formally state our optimization problem! (In the constraints, the 1 was moved over to the other side of the inequality.)

$$\text{minimize} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$\text{subject to} \quad y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 \geq 0 \quad \forall i$$

This is called the **primal problem**. This is a run-of-the-mill optimization problem, so we can use the technique of Lagrange Multipliers to solve this problem.

$$\mathcal{L} = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^N \alpha_i [y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1]$$

where the  $\alpha_i$ 's are the Lagrange multipliers. To solve this, we have to compute the partial derivatives with respect to our weights and bias, set them to zero, and solve! I'll skip over the derivation and just give the solutions.

$$\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$$

$$\sum_{i=1}^N \alpha_i y_i = 0$$

The first equation is  $\frac{\partial \mathcal{L}}{\partial \mathbf{w}}$  and the second equation is  $\frac{\partial \mathcal{L}}{\partial b}$ . These solutions tell us some useful things about the weights and Lagrange multipliers. In particular, they give some constraints on the Lagrange multipliers. These  $\alpha_i$ 's also tell us something very important about our SVM: they indicate the support vectors! If a particular point  $\mathbf{x}_i$  is a support vector, then its corresponding Lagrange multiplier  $\alpha_i$  will be greater than 0! If it is not a support vector, then it will be equal to 0!

However, we still don't have enough information to solve our problem. As it turns out, there is a corresponding problem called the **dual problem** that we can solve instead.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \quad \forall i \\ & \alpha_i \geq 0 \quad \forall i \end{aligned}$$

This is something that we can solve! Notice that it's only in terms of the Lagrange multipliers! Everything else is known! We usually use a quadratic programming solver to do this for us because it is infeasible to solve by-hand for large numbers of points. But we would solve

for this by setting each  $\frac{\partial \mathcal{L}}{\partial \alpha_i} = 0$  and solving.

After we've solved for the  $\alpha_i$ 's, we can find the optimal line using the following equations.

$$\begin{aligned} \mathbf{w} &= \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \\ b &= \text{average}_{i \in N} \frac{1}{y_i} - \mathbf{w}^T \mathbf{x}_i \end{aligned}$$

The first is from the primal problem, and the second is just solving for the bias from the decision boundary equation.

## SVMs for Logic Gates

Let's take a break from the math and apply support vector machines to a simple logic gate, like what we did for perceptrons. In particular, let's train an SVM to solve the logic AND gate.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets

X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
y = np.array([0, 0, 0, 1])

clf = svm.SVC(kernel='linear', C=1e6)
clf.fit(X, y)
```

We're building a linear decision boundary. Ignore the other parameter  $C$ ; we'll discuss that later.

Now we can use some plotting code ([source](#)) to show the decision boundary and support vectors.

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

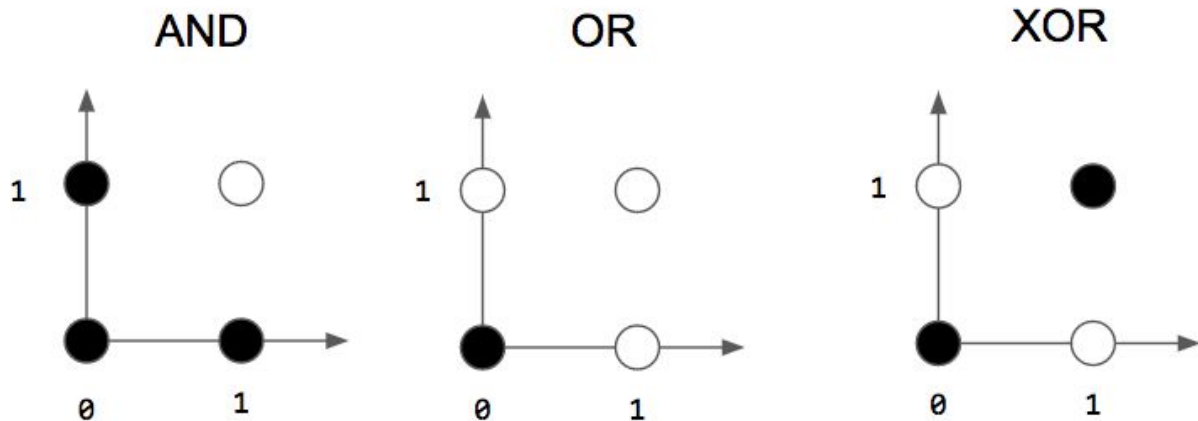
# plot the decision function
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# create grid to evaluate model
xx = np.linspace(xlim[0], xlim[1], 30)
yy = np.linspace(ylim[0], ylim[1], 30)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = clf.decision_function(xy).reshape(XX.shape)

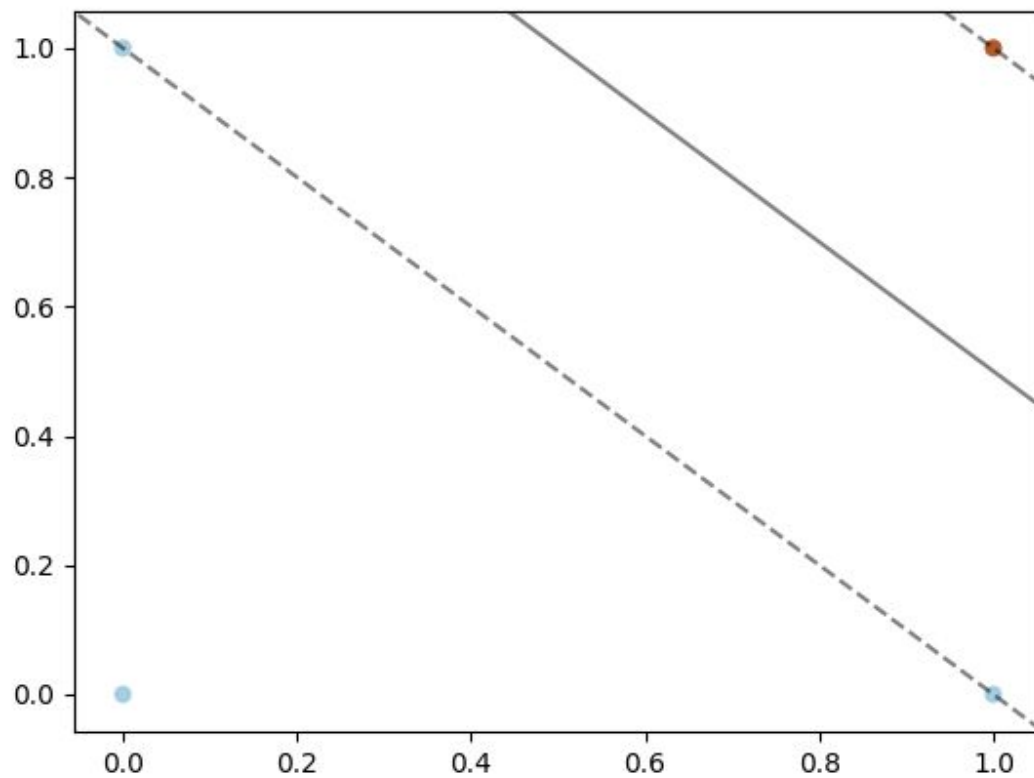
# plot decision boundary and margins
```

```
ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
linestyles=['--', '-', '--'])
# plot support vectors
ax.scatter(clf.support_vectors_[0], clf.support_vectors_[1], s=100,
linewidth=1, facecolors='none')
plt.show()
```

Before we plot this, let's try to predict what our decision boundary and surface will look like. Here's the picture of the logic gates again.



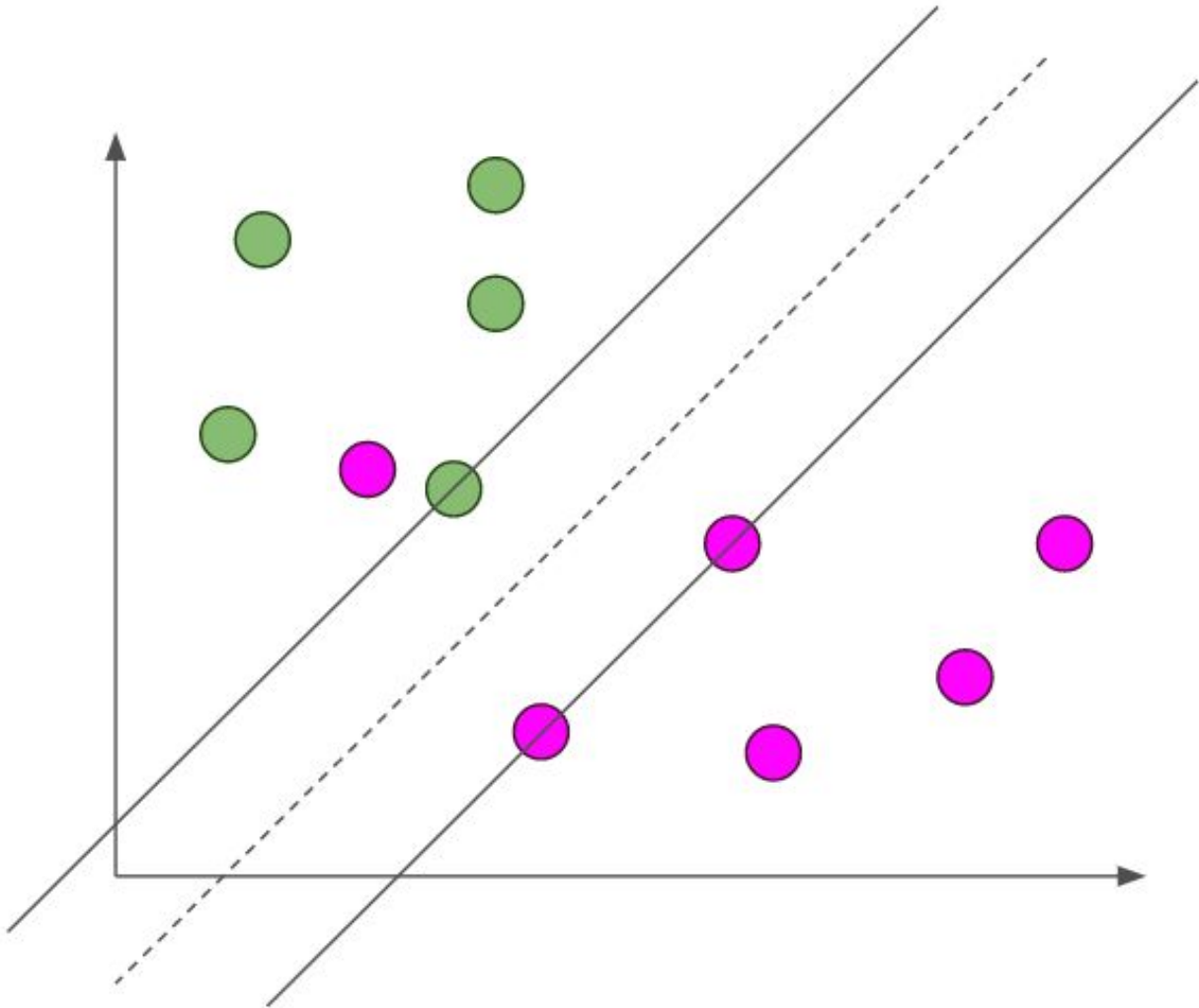
Where will the decision boundary be? Which points will be the support vectors? The decision boundary will be a diagonal line between the two classes. The support vectors will be (1,1), (0,1), and (1,0) since they are closest to that boundary.



This matches our intuition! So SVMs can certainly solve linear separable problems, but what about non-linearly separable problems?

## SVMs for Linearly Inseparable Problems

Suppose we had the following linearly inseparable data.



There is no line that can correctly classify each point! Can we still use our SVM? We can, but with a modification. We have to add **slack variables**  $\xi_i$ . These measure how many misclassifications there are. We also want to minimize the sum of all of the slack variables. Intuitively, this corresponds to minimizing the number of incorrect classifications. We can reformulate our primal problem.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=0}^N \xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 + \xi_i \geq 0 \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned}$$

where we introduce a new hyperparameter  $C$  that measures the tradeoff between the two objectives: largest margin of separation and smallest number of incorrect classifications. And, from there, go to our corresponding dual problem.

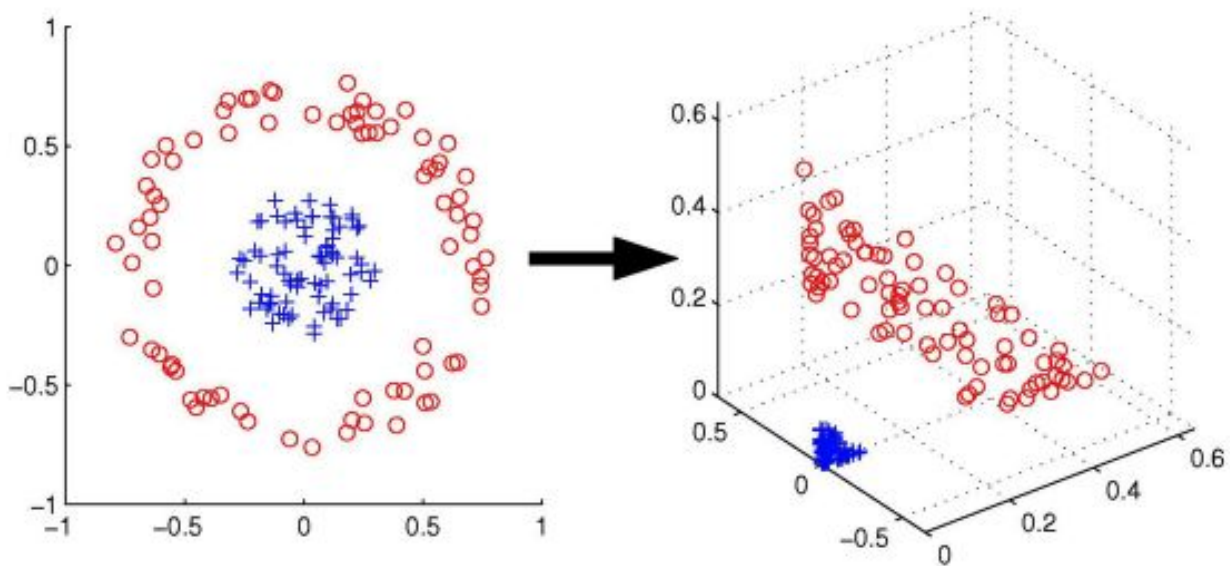


$$\begin{aligned}
& \text{maximize} && \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\
& \text{subject to} && \sum_{i=1}^N \alpha_i y_i = 0 \quad \forall i \\
& && 0 \leq \alpha_i \leq C \quad \forall i
\end{aligned}$$

This looks almost the same as before! The change is that our  $\alpha_i$ 's are also bounded above by  $C$ . After solving for our  $\alpha_i$ 's, we can solve for our weights and bias exactly the same as in our linearly separable case!

### The Kernel Trick

One last topic to discuss is the **kernel trick**. Instead of having a linear decision boundary, we can have a nonlinear decision boundary. The idea behind the kernel trick is to apply a nonlinear kernel to our inputs  $\mathbf{x}_i$  to transform them into a *higher-dimensional* space where we can find a linear decision boundary.



Consider the above figure. The left is our 2D dataset that can't be separated using a line. However, if we use some kernel function  $\varphi(\mathbf{x}_i)$  to project all of our points into a 3D space, then we can find a plane that separates our examples. The intuition behind this is that higher dimensional spaces have extra degrees of freedom that we can use to find a linear plane! There are many different choices of kernel functions: radial basis functions, polynomial functions, and others.

## SVM for The Iris Dataset

One of the most famous datasets in all of machine learning is the **iris dataset**. It has 150 data points across 3 different types of flowers. The features that were collected were sepal length/width and petal length/width. Our goal is to use an SVM to correctly classify an input into the correct flower and to draw the decision boundary.

Since the iris dataset has 4 features, let's consider only the first two features so we can plot our decision regions on a 2D plane. First, let's load the iris dataset, create our training and testing data, and fit our SVM. We'll change some parameters later, but let's use a linear SVM.

```
iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target

C = 1.0
clf = svm.SVC(kernel='linear', C=C)
clf.fit(X, y)

Now we can use some auxiliary functions (source) to plot our decision
regions.
ax = plt.gca()
def make_meshgrid(x, y, h=.02):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min,
y_max, h))
    return xx, yy

def plot_contours(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out

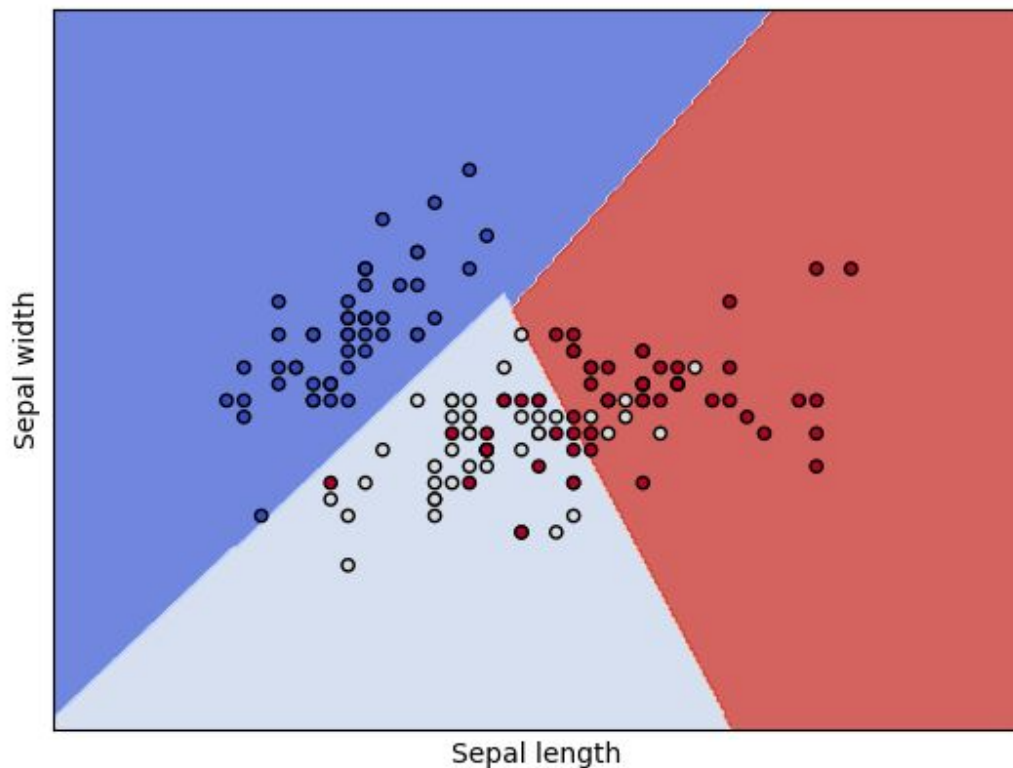
plot_contours(ax, clf, xx, yy, cmap=plt.cm.coolwarm, alpha=0.8)
ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xlabel('Sepal length')
ax.set_ylabel('Sepal width')
ax.set_xticks(())
ax.set_yticks(())
```

```
plt.show()
```

Additionally, we're going to print the classification report to see how well our SVM performed.

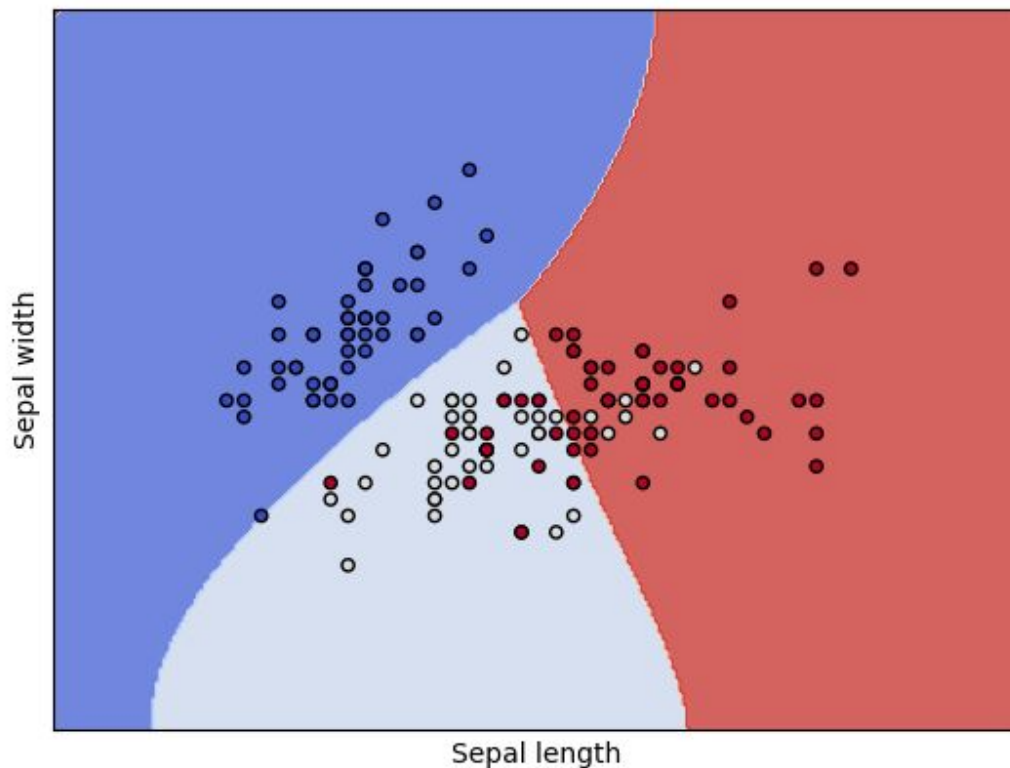
```
from sklearn.metrics import classification_report
print(classification_report(y, clf.predict(X),
target_names=iris.target_names))
```

Now let's run our code to see a plot and classification metrics!



Additionally, we can try using an RBF kernel and changing our  $C$  value. Recall that  $C$  controls the tradeoff between large margin of separation and a lower incorrect classification rate.

```
C = 1.0
clf = svm.SVC(kernel='rbf', C=C)
```



Try varying different parameters to get the best classification score!

To summarize, Support Vector Machines are very powerful classification models that aim to find a maximal margin of separation between classes. We saw how to formulate SVMs using the primal/dual problems and Lagrange multipliers. We also saw how to account for incorrect classifications and incorporate that into the primal/dual problems. Finally, we trained an SVM on the iris dataset.

Support Vector Machines are one of the most flexible non-neural models for classification; they're able to model linear and nonlinear decision boundaries for linearly separable and inseparable problems.

## Reinforcement Learning: Balancing Act

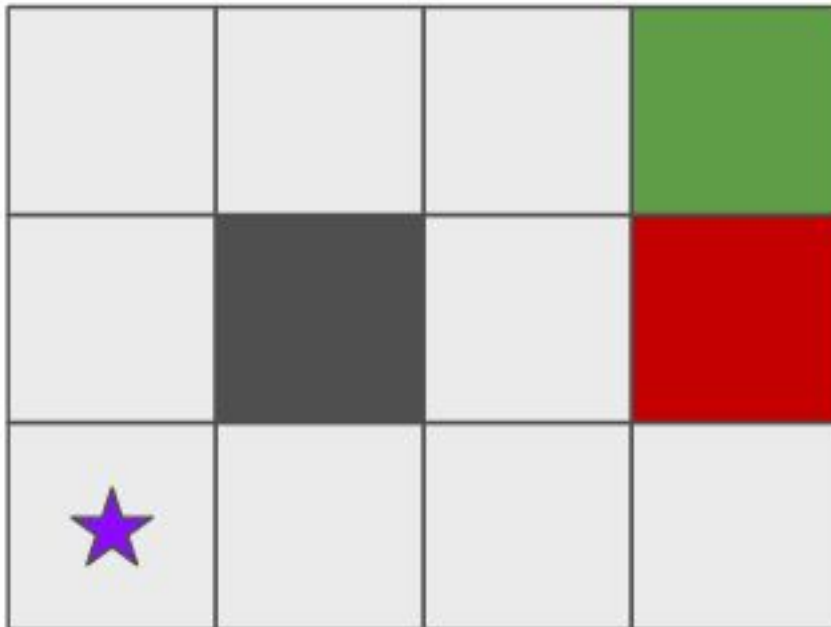
Think back to the time you first learned a skill: driving a car, playing an instrument, cooking a recipe. Let's consider the example of playing chess. Initially, it might have seemed difficult, but, as you played more and more, it became easier to understand the game. After playing many games of chess, you are much better than when you first started and "get" the game. You know which moves you should avoid and which moves you should play. In other words, you've developed *experience* by playing the game.

This is the same idea behind **reinforcement learning**. First, we'll discuss what we need to define a game through a Markov Decision Process. Then we'll discuss how we solve these using the Value Iteration Algorithm. We'll discuss the Q-Learning algorithm for teaching a machine to play a game. Finally, we'll implement Q-Learning in an OpenAI gym environment to teach a machine to play CartPole!

Download the full code [here](#).

### Games

Before discussing reinforcement learning, we have to discuss the setup of a game. Before delving into the mathematics and definitions, let's first try to use our reasoning. Suppose we have a robot (purple star) in an environment like the following picture.



The dark gray squares are walls. The green square yields the highest reward, e.g., the location of the buried treasure. The red square yields the lowest reward, e.g., a fire pit. We want to give commands to our robot to make it go to the green square and avoid the red square in the

most efficient way possible. We have a finite number of actions: move left, up, right, and down. But, since we're in the real world, the robot may not always go up. Robots don't always act perfectly in the real world! Instead, our robot has a very high *probability* of doing the intended action, e.g., 90% chance of going up when we tell it to up. However, there are non-zero probabilities that the robot will go in a different direction, e.g., 5% chance of going left or right when we tell it go to up. We have to keep these in mind when deciding which action to take.

These actions have to be taken within the confines of our environment, the most important characteristic of our game. Given our finite environment, there are only so many possibilities where our robot can be: the number of light gray squares. When we take an action, we go from one square to another, changing our board configuration. If we take the right actions, we can get to the green square and get a large reward!

Using the framework we described, we can define our game. Specifically, we use a **Markov Decision Process** (MDP) to define our game. An MDP is defined by the following.

1. Set of states  $S$
2. Set of actions  $A$  that we can take at a state  $s$
3. Transition function  $T(s, a, s')$  where  $s \in S$ ,  $a \in A$ , and  $s' \in S$  that is *nondeterministic* because of real-world situations where we may not always follow the given action always.
4. Reward function  $R(s, a, s')$  that tells us what reward we get by taking action  $a$  in state  $s$  and ending up in state  $s'$

Using these four things, we can define an MDP. The goal is to learn an **optimal policy**  $\pi^*(s)$  that tells us the optimal action to take at every state to maximize our reward.

We also introduce the concept of **discounting**: we prefer to have rewards sooner rather than later. We *discount* rewards at each time step so that we place an emphasis on having rewards sooner. This not only has a semantic meaning, but it is also mathematically useful: it helps our algorithms actually converge! Mathematically, we denote the discounting factor as  $\gamma$ . This will manifest itself in terms of exponential decay, but we'll discuss that soon.

We can define these in terms of our robot in the environment. The set of states is all possible position of our robot. The set of actions is moving left, up, right, and down. We can model the transition function as being a probabilistic function that returns the probability of going into state  $s'$  after taking action  $a$  in state  $s$ . And the reward function can simply be a positive value if we go into the green square and a negative value for going into the red square.

## Value Iteration

Knowing the transition function and reward function, the goal is to figure out the optimal policy  $\pi^*(s)$ . **Value Iteration** is technique we can use to solve for this. The idea is that we

assign a value  $V(s)$  to each state that represents the expected reward of starting in  $s$  and acting optimally. We have to compute an expected value because we're not deterministic! We can compute this using one of the **Bellman Equations**.

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

Let's take a second to break this down. The sum is over all states  $s' \in S$  and computes the expected reward of taking action  $a$  in state  $s$ . The  $\max$  in the front tells us to consider all actions  $a \in A$  and pick the expected reward that is the largest. Notice that we're considering the discounting factor  $\gamma$  in the expectation. Also note that this definition is recursive: we have to know  $V^*(s')$  to compute  $V^*(s)$ .

Values near the high-reward states will have high  $V^*(s)$  values since there is a high expected reward because we're close to a high-reward state and the discounting factor isn't large. On the other hand, values far away from the high-reward state will have smaller expected reward because the discounting factor will be larger. Think of it this way: the farther you are, there are more chances for malfunctioning circuits in that distance as opposed to being right next to a high-reward state.

To compute these optimal expected rewards, we can use the **Value Iteration** algorithm. We initialize all  $V_0(s) = 0$  for all states. And we compute the  $V_1(s)$  given all of the values of  $V_0(s)$ . Then, we compute  $V_2(s)$  using  $V_1(s)$  and so on using this equation.

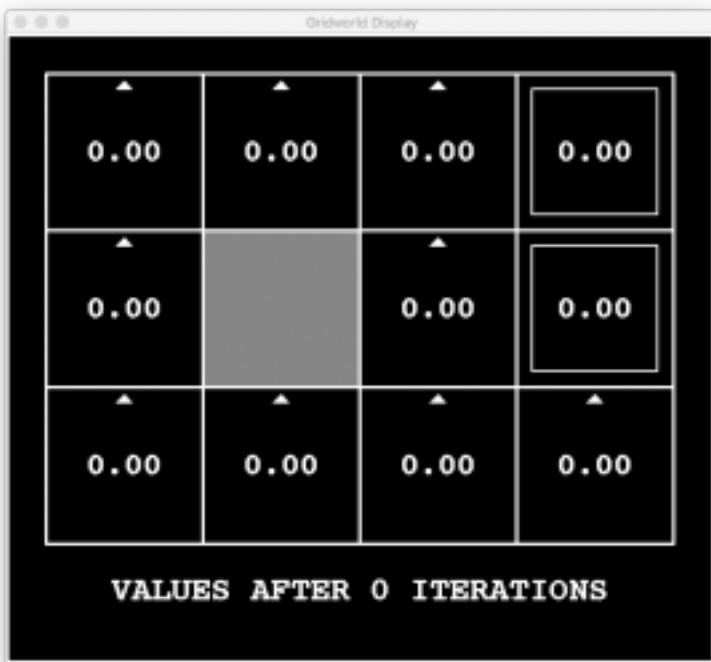
$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

We use this iterative Bellman equation to compute the next iteration of all of the values for all of the states given the previous values. Eventually, the values won't change, and we say that the algorithm has converged.

Algorithmically, we can write the following.

1. Start with  $V_0(s) = 0$  for all states
2. Compute  $V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$
3. Go to Step 2 and repeat until convergence

The amazing thing is that value iteration will certainly converge to the optimal values (I've omitted the exact proof). Here are some images that show value iteration for our robotic environment.







But remember our goal: find the optimal policy  $\pi^*(s)$ . How can we use these values to compute the optimal policy?

Suppose that we've run value iteration for a long enough amount of time, and our values have converged. In other words, we know  $V^*(s)$  for all states. To determine the policy, i.e., how to act at a state, we do a one-step lookahead to see which action will produce the maximum expected reward. This technique is called **policy extraction**.

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

Using value iteration and policy extraction, we can solve MDPs! However, we stated that we knew the transition and reward functions. In many real-world scenarios, we don't know these explicitly! In other words, we don't know which states are good/bad or what the actions actually do. Therefore, we can't use value iteration and policy extraction anymore! We actually have to try to take actions in the environment and observe their results to learn. This is the heart of reinforcement learning: learn from experience!

## Q-Learning

If we knew the transition and reward functions, we could easily use value iteration and policy extraction to solve our problem. However, in reinforcement learning we don't know these! Q-Learning is a simple modification of value iteration that allows us to train with the policy in mind. Instead of using and storing just expected reward, we consider actions as well in a pair called a **q-value**  $Q(s, a)$ ! Then we can iterate over these q-values and computing the optimal policy is simply selecting the action with the largest q-value.

$$\pi^*(s) = \operatorname{argmax}_a Q(s, a)$$

The q-value update looks very similar to value iteration.

$$Q_{k+1}(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$$

We consider the expected reward, but we only look at the action that produces the largest q-value ( $\max_{a'} Q_k(s', a')$ ). We can iterate over the q-values: the idea is to learn from each experience!

Here is the Q-Learning algorithm:

1. Perform an action  $a$  in a state  $s$  to end up in  $s'$  with reward  $r$ , i.e., consider the tuple  $(s, a, s', r)$ .

2. Compute the intermediate q-value  $\hat{Q}(s, a) = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$

3. Incorporate that new evidence into the existing value using a running average

$Q_{k+1}(s, a) = (1 - \alpha)Q_k(s, a) + (\alpha)\hat{Q}(s, a)$  (where  $\alpha$  is the learning rate). This can be re-written in an gradient-descent update rule fashion as

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha(\hat{Q}(s, a) - Q_k(s, a))$$

Q-Learning, like value iteration, also converges to the optimal values, even if we don't act optimally! The downsides of q-learning is that we have to make sure we explore enough and decay the learning rate.

Why do we want our agent to explore? If we find one sequence of actions that leads to a high reward, we want to repeat those actions. But there's a chance that there exist *another* sequence of actions that could produce an *even higher* expected reward. In practice, we sometimes pick a random action instead of the optimal one a fraction of the time. This fraction is denoted by  $\epsilon$ . The value of this encourages our agent to try doing some random values, i.e., to explore the state space more! This is called the  $\epsilon$ -**Greedy Approach**.

Initially, our agent doesn't know what to do or what actions lead to good rewards, so we start with a high value of  $\epsilon$  (relatively) and decay it over time (because good actions become clearer as we play, and we should stop taking random actions). We may decay to zero, which means we should always take the optimal action after convergence. Or we can decay to a very small value, encouraging the agent to try random actions every once-in-a-while. The other parameter we decay is the learning rate, which will help our network converge to the optimal values.

Here an example of running Q-Learning for our robotic environment.



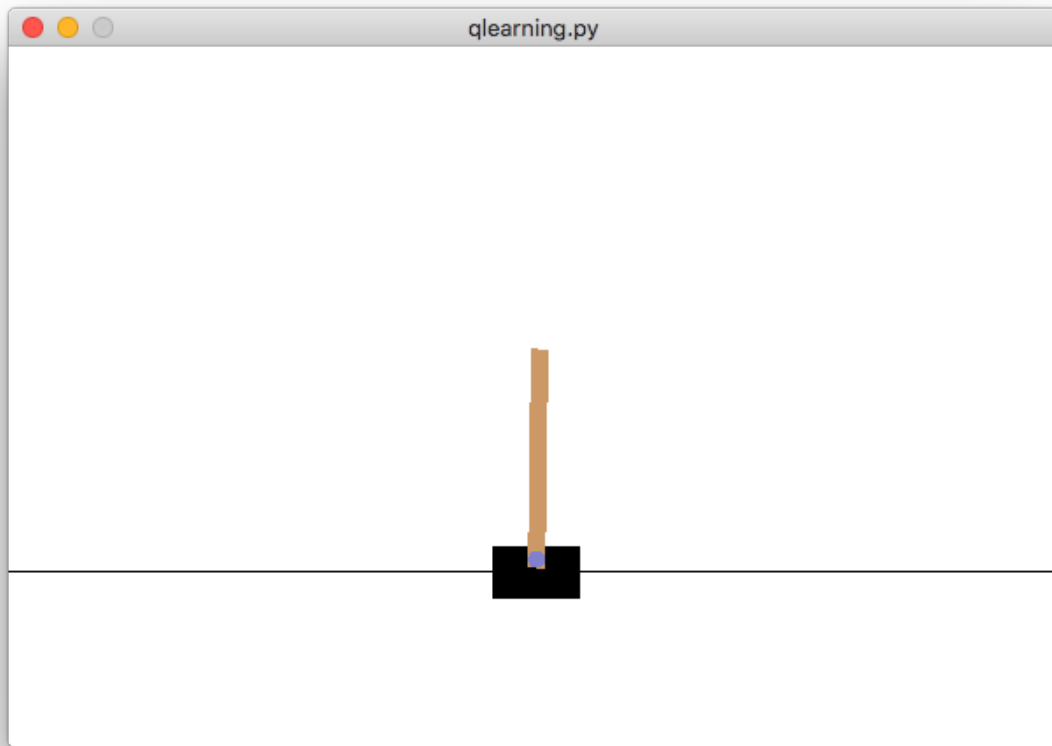


Recall that the optimal policy is to take the action with the largest q-value at any state. If we did this for the above q-values, we would tend to go towards the higher reward and avoid the lower one.

Notice that there are some q-values that are still zero. In other words, we have arrived at a particular state where we haven't taken a specific action. For example, consider the bottom-right square. We haven't taken the down or right action in that state so we don't know what value should be assigned to it.

### Q-Learning Agent for CartPole

Now that we have an understanding of Q-Learning, let's code! Before starting, we'll need to install [OpenAI Gym](#) (`pip3 install gym`) and `ffmpeg` (`brew install ffmpeg`). The OpenAI Gym provides us with a ton of different reinforcement learning scenarios with visuals, transition functions, and reward functions already programmed. Now we'll implement Q-Learning for the simplest game in the OpenAI Gym: CartPole! The objective of the game is simply to balance a stick on a cart.



This is a simple game that we can understand well. You can take the Q-Learning code we implement and generalize it to any of the games in the OpenAI Gym.

The state of this game is characterized by 4 quantities: position of the cart, velocity of the cart, angle of the pole, and angular velocity of the pole  $(x, \dot{x}, \theta, \dot{\theta})$ , respectively. But all of these quantities are continuous variables! We have to *discretize* the values by putting them into buckets, e.g., values between -0.5 and -0.4 will be in one bucket, etc. As for the actions, there are only two: move left or move right!

Let's create a class for our Q-Learning agent.

```
import gym
import numpy as np
import math

class CartPoleQAgent():
    def __init__(self, buckets=(1, 1, 6, 12), num_episodes=1000,
min_lr=0.1, min_explore=0.1, discount=1.0, decay=25):
```

```

self.buckets = buckets
self.num_episodes = num_episodes
self.min_lr = min_lr
self.min_explore = min_explore
self.discount = discount
self.decay = decay

self.env = gym.make('CartPole-v0')

self.upper_bounds = [self.env.observation_space.high[0], 0.5,
self.env.observation_space.high[2], math.radians(50) / 1.]
self.lower_bounds = [self.env.observation_space.low[0], -0.5,
self.env.observation_space.low[2], -math.radians(50) / 1.]

self.Q_table = np.zeros(self.buckets + (self.env.action_space.n,))

```

In the constructor, we use buckets to discretize our state space. In reinforcement learning, we train for a number of episodes, kind of like the number of epochs for supervised/unsupervised learning. We also have the minimum learning rate, exploration rate, and discount factor. Finally we have the decay factor that will be used for the learning and exploration rate decay.

We also bound the position and angle to be the same as the low and high of the environment. We manually bound velocity ( $\pm 0.5$  m/s) and angular velocity ( $\pm 50$  deg / s). Finally, we create the table of q-values.

In practice, we store all of the q-values in a giant lookup table. The rows are the state space and the columns are the actions. However, we characterized our state space as being a 4-tuple. So the resulting table will have 5 dimensions: the first four correspond to the state and the last one is the action index:  $(x, \dot{x}, \theta, \dot{\theta}, a)$ . Given this tuple, we can access a scalar value in the table.

Speaking of the q-values, let's write the code that will update a value in the table. Recall that the update formula is the following (substituting for  $\hat{Q}(s, a)$ ).

$$Q_{k+1}(s, a) \leftarrow Q_k(s, a) + \alpha [R(s, a, s') + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a)]$$

We can translate this directly into Python code.

```

def update_q(self, state, action, reward, new_state):
    self.Q_table[state][action] += self.lr * (reward + self.discount *
np.max(self.Q_table[new_state]) - self.Q_table[state][action])

```

Additionally, we'll write update rules for the exploration and learning rates.

```
def get_explore_rate(self, t):
    return max(self.min_explore, min(1., 1. - math.log10((t + 1) /
self.decay)))

def get_lr(self, t):
    return max(self.min_lr, min(1., 1. - math.log10((t + 1) / self.decay)))
```

This is just a form of decay. Essentially, we decay our exploration and learning rates until it is the minimum exploration and learning rate that we specified in the constructor.

We also need a function to choose an action using the  $\epsilon$ -Greedy approach.  $\epsilon$  percent of the time, we take a random action, but the rest of the time we take the optimal action. Recall that the optimal action is the following.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q(s, a)$$

We can translate that into code.

```
def choose_action(self, state):
    if (np.random.random() < self.explore_rate):
        return self.env.action_space.sample()
    else:
        return np.argmax(self.Q_table[state])
```

Now we can write some code to discretize our search space and train our agent! The first function simply does some math to figure out which buckets our observation should be in.

```
def discretize_state(self, obs):
    discretized = list()
    for i in range(len(obs)):
        scaling = (obs[i] + abs(self.lower_bounds[i])) /
(self.upper_bounds[i] - self.lower_bounds[i])
        new_obs = int(round((self.buckets[i] - 1) * scaling))
        new_obs = min(self.buckets[i] - 1, max(0, new_obs))
        discretized.append(new_obs)
    return tuple(discretized)

def train(self):
    for e in range(self.num_episodes):
        current_state = self.discretize_state(self.env.reset())

        self.lr = self.get_lr(e)
        self.explore_rate = self.get_explore_rate(e)
```

```

done = False

while not done:
    action = self.choose_action(current_state)
    obs, reward, done, _ = self.env.step(action)
    new_state = self.discretize_state(obs)
    self.update_q(current_state, action, reward, new_state)
    current_state = new_state

print('Finished training!')

```

The training function runs for a number of episodes. We reset the CartPole environment and discretize the state  $(x, \dot{x}, \theta, \dot{\theta})$ . Then we fetch the exploration and learning rates at that episode. The inner loop iterates until the episode is finished, which is determined by the CartPole environment. We can figure out the exact conditions if we look inside the code of CartPole. The episode is done when the position and angle are within a threshold. Until then, we pick an action and perform that action. We get back a new state, reward, and other parameters. We use that discretized state and reward to update the q-value and update the current state.

After we finish training, we can render the CartPole environment in a window and see the cart balance the pole.

```

def run(self):
    self.env = gym.wrappers.Monitor(self.env, directory='cartpole',
    force=True)

    while True:
        current_state = self.discretize_state(self.env.reset())
        done = False

        while not done:
            self.env.render()
            action = self.choose_action(current_state)
            obs, reward, done, _ = self.env.step(action)
            new_state = self.discretize_state(obs)
            current_state = new_state

if __name__ == "__main__":
    agent = CartPoleQAgent()
    agent.train()
    agent.run()

```



Since we're not training, I decided not to update the q-values, but you may choose to do so! By configuring a monitor, OpenAI Gym will create short clips like the following.

To summarize, we discussed the setup of a game using Markov Decision Processes (MDPs) and value iteration as an algorithm to solve them when the transition and reward functions are known. Then we moved on to reinforcement learning and Q-Learning. Finally, we implemented Q-Learning to teach a cart how to balance a pole. This is just the first step into Deep Q-Networks where the q-value table can be replaced with a neural network. Reinforcement Learning, such as Deep Q-Networks, is currently in use everywhere: to teach computers to play games like Pacman, Chess, and Go to driving cars autonomously!