**K-Nearest Neighbors Classifier**

**K-Nearest Neighbors (KNN)** is a classification algorithm. The central idea is that data points with similar attributes tend to fall into similar categories.

Consider the image to the right. This image is complicated, but for now, let's just focus on where the data points are being placed. Every data point — whether its color is red, green, or white — has an x value and a y value. As a result, it can be plotted on this two-dimensional graph.

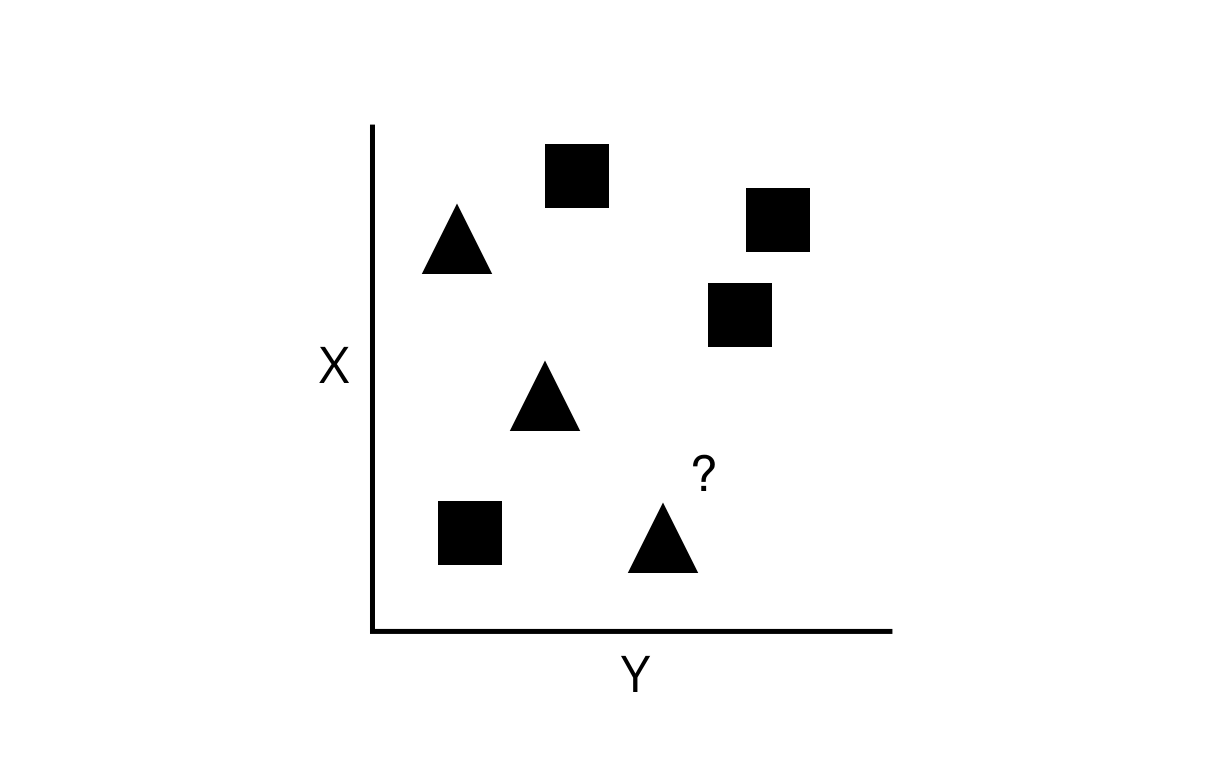
Next, let's consider the color of the data. The color represents the class that the K-Nearest Neighbor algorithm is trying to classify. In this image, data points can either have the class green or the class red. If a data point is white, this means that it doesn't have a class yet. The purpose of the algorithm is to classify these unknown points.

Finally, consider the expanding circle around the white point. This circle is finding the k nearest neighbors to the white point. When k = 3, the circle is fairly small. Two of the three nearest neighbors are green, and one is red. So in this case, the algorithm would classify the white point as green. However, when we increase k to 5, the circle expands, and the classification changes. Three of the nearest neighbors are red and two are green, so now the white point will be classified as red.

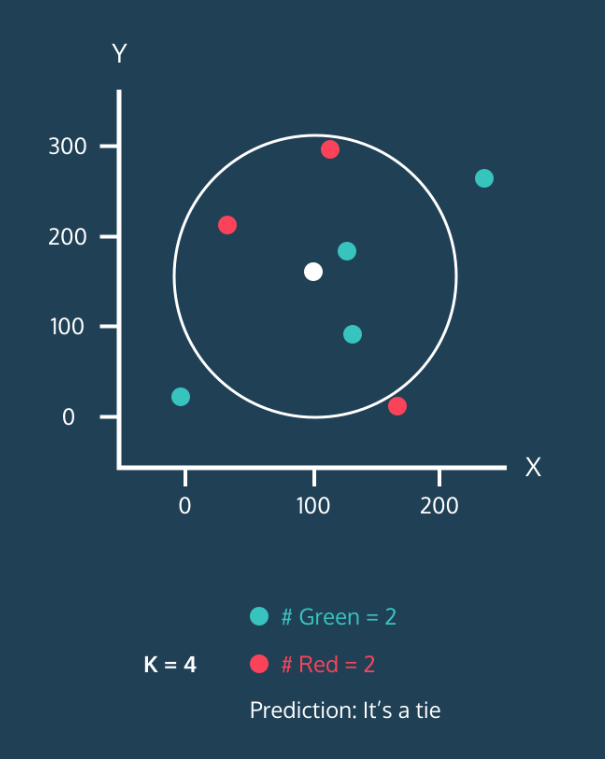
This is the central idea behind the K-Nearest Neighbor algorithm. If you have a dataset of points where the class of each point is known, you can take a new point with an unknown class, find it's nearest neighbors, and classify it.

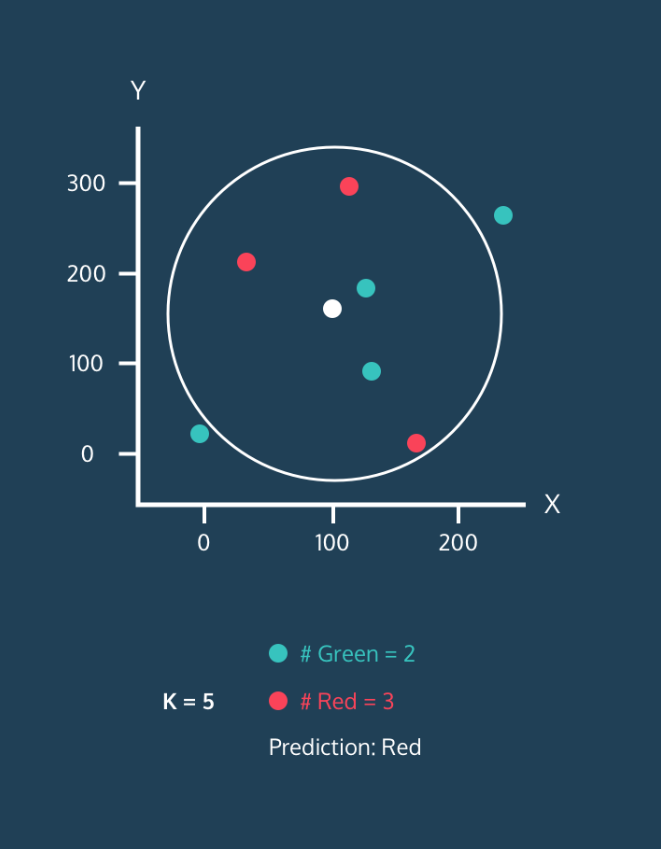
Before moving on to the next exercise, consider the image below:

* If k = 1, what would the class of the question mark be?
* If k = 5, what would it be?



Note that rather than using colors, in this image, the class is denoted by the shape of each point.





**Introduction**

Before diving into the K-Nearest Neighbors algorithm, let's first take a minute to think about a example.

Consider a dataset of movies. Let's brainstorm some features of a movie data point. A feature is a piece of information associated with a data point. Here are some potential features of movie data points:

* the *length* of the movie in minutes.
* the *budget* of a movie in dollars.

If you think back to the previous exercise, you could imagine movies being places in that two-dimensional space based on those numeric features. There could also be some boolean features: features that are either true or false. For example, here are some potential boolean features:

* *Black and white*. This feature would be True for black and white movies and False otherwise.
* *Directed by Stanley Kubrick*. This feature would be False for almost every movie, but for the few movies that were directed by Kubrick, it would be True.

Finally, let's think about how we might want to classify a movie. For the rest of this lesson, we're going to be classifying movies as either good or bad. In our dataset, we've classified a movie as good if it had an IMDb rating of 7.0 or greater. Every "good" movie will have a class of 1, while every bad movie will have a class of 0.

To the right, we've created some movie data points where the first item in the list is the length, the second is the budget, and the third is whether the movie was directed by Stanley Kubrick.

Instructions

**1.**

Add another movie to the code to your right. Add the variable gone\_with\_the\_wind. This movie is 238 minutes long (wow!), had a budget of $3,977,000, and was not directed by Stanley Kubrick.

**Distance Between Points - 2D**

In the first exercise, we were able to visualize the dataset and estimate the k nearest neighbors of an unknown point. But a computer isn’t going to be able to do that!

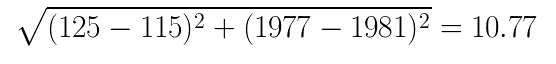
We need to define what it means for two points to be close together or far apart. To do this, we’re going to use the [Distance Formula](https://www.codecademy.com/content-items/8a61a8bd456c17af1e3a6922694c811f/exercises/points).

For this example, the data has dimensions:

* The length of the movie
* The movie’s release date

Consider *Star Wars* and *Raiders of the Lost Ark*. *Star Wars* is 125 minutes long and was released in 1977. *Raiders of the Lost Ark* is 115 minutes long and was released in 1981.

The distance between the movies is computed below:



Instructions

**1.**

Write a function named distance that takes two lists named movie1 and movie2 as parameters.

You can assume that each of these lists contains two numbers — the first number being the movie's runtime and the second number being the year the movie was released. The function should return the distance between the two lists.

Remember, in python, x \*\* 0.5 will give you the square root of x.

Similarly, x \*\* 2 will give you the square of x.

**2.**

Call the function on some of the movies we've given you.

Print the distance between *Star Wars* and *Raiders of the Lost Ark*.

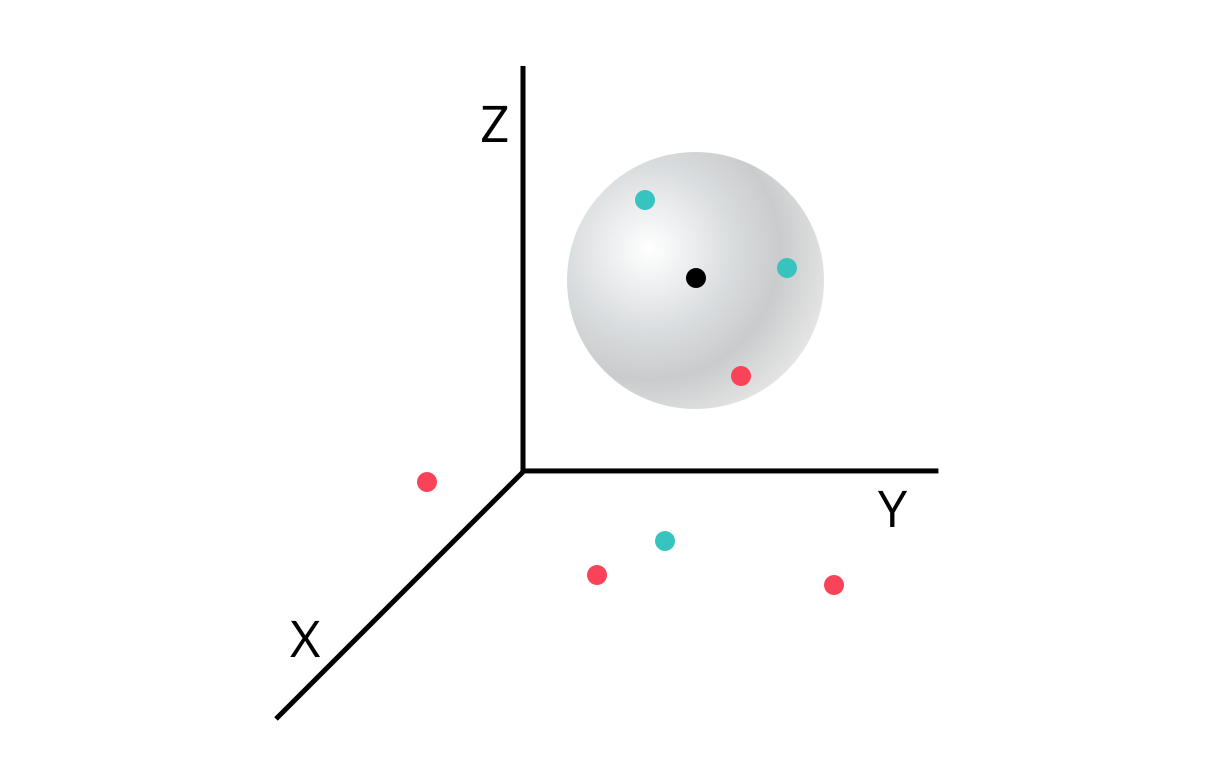
Print the distance between *Star Wars* and *Mean Girls*.

Which movie is *Star Wars* more similar to?

# Distance Between Points - 3D

Making a movie rating predictor based on just the length and release date of movies is pretty limited. There are so many more interesting pieces of data about movies that we could use! So let’s add another dimension.

Let’s say this third dimension is the movie's budget. We now have to find the distance between these two points in three dimensions.



What if we’re not happy with just three dimensions? Unfortunately, it becomes pretty difficult to visualize points in dimensions higher than 3. But that doesn’t mean we can’t find the distance between them.

The generalized distance formula between points A and B is as follows:

sqrt{(A\_1-B\_1)^2+(A\_2-B\_2)^2+ … +(A\_n-B\_n)^2}

​

Here, A1-B1 is the difference between the first feature of each point. An-Bn is the difference between the last feature of each point.

Using this formula, we can find the K-Nearest Neighbors of a point in N-dimensional space! We now can use as much information about our movies as we want.

We will eventually use these distances to find the nearest neighbors to an unlabeled point.

**1.**

Modify your distance function to work with any number of dimensions. Use a for loop to iterate through the dimensions of each movie.

Return the total distance between the two movies.

**2.**

We've added a third dimension to each of our movies.

Print the new distance between Star Wars and Raiders of the Lost Ark.

Print the new distance between Star Wars and Mean Girls.

Which movie is Star Wars closer to now?

**Data with Different Scales: Normalization**

In the next three lessons, we'll implement the three steps of the K-Nearest Neighbor Algorithm:

1. **Normalize the data**
2. Find the k nearest neighbors
3. Classify the new point based on those neighbors

When we added the dimension of budget, you might have realized there are some problems with the way our data currently looks.

Consider the two dimensions of release date and budget. The maximum difference between two movies' release dates is about 125 years (The Lumière Brothers were making movies in the 1890s). However, the difference between two movies' budget can be millions of dollars.

The problem is that the distance formula treats all dimensions equally, regardless of their scale. If two movies came out 70 years apart, that should be a pretty big deal. However, right now, that's exactly equivalent to two movies that have a difference in budget of 70 dollars. The difference in one year is exactly equal to the difference in one dollar of budget. That’s absurd!

Another way of thinking about this is that the budget completely outweighs the importance of all other dimensions because it is on such a huge scale. The fact that two movies were 70 years apart is essentially meaningless compared to the difference in millions in the other dimension.

The solution to this problem is to [normalize the data](https://www.codecademy.com/articles/normalization) so every value is between 0 and 1. In this lesson, we're going to be using min-max normalization.

**1.**

Write a function named min\_max\_normalize that takes a list of numbers named lst as a parameter (lst short for list).

Begin by storing the minimum and maximum values of the list in variables named minimum and maximum

**2.**

Create an empty list named normalized. Loop through each value in the original list.

Using min-max normalization, normalize the value and add the normalized value to the new list.

After adding every normalized value to normalized, return normalized.

**3.**

Call min\_max\_normalize using the given list release\_dates. Print the resulting list.

What does the date 1897 get normalized to? Why is it closer to 0 than 1?

**Finding the Nearest Neighbors**

The K-Nearest Neighbor Algorithm:

1. Normalize the data
2. **Find the k nearest neighbors**
3. Classify the new point based on those neighbors

Now that our data has been normalized and we know how to find the distance between two points, we can begin classifying unknown data!

To do this, we want to find the k nearest neighbors of the unclassified point. In a few exercises, we’ll learn how to properly choose k, but for now, let’s choose a number that seems somewhat reasonable. Let’s choose 5.

In order to find the 5 nearest neighbors, we need to compare this new unclassified movie to every other movie in the dataset. This means we’re going to be using the distance formula again and again. We ultimately want to end up with a sorted list of distances and the movies associated with those distances.

It might look something like this:

[

[0.30, 'Superman II'],

[0.31, 'Finding Nemo'],

...

...

[0.38, 'Blazing Saddles']

]

In this example, the unknown movie has a distance of 0.30 to Superman II.

In the next exercise, we'll use the labels associated with these movies to classify the unlabeled point.

**1.**

Begin by running the program. We've imported and normalized a movie dataset for you and printed the data for the movie Bruce Almighty. Each movie in the dataset has three features:

* the normalized budget (dollars)
* the normalized duration (minutes)
* the normalized release year.

We've also imported the labels associated with every movie in the dataset. The label associated with Bruce Almighty is a 0, indicating that it is a bad movie. Remember, a bad movie had a rating less than 7.0 on IMDb.

Comment out the two print lines after you have run the program.

**2.**

Create a function called classify that has three parameters: the data point you want to classify named unknown, the dataset you are using to classify it named dataset, and k, the number of neighbors you are interested in.

For now put pass inside your function.

**3.**

Inside the classify function remove pass. Create an empty list called distances.

Loop through every title in the dataset.

Access the data associated with every title by using dataset[title].

Find the distance between dataset[title] and unknown and store this value in a variable called distance\_to\_point.

Add the list [distance\_to\_point, title] to distances.

Outside of the loop, return distances.

**4.**

We now have a list of distances and points. We want to sort this list by the distance (from smallest to largest). Before returning distances, Use Python's built-in sort() function to sort distances.

**5.**

The k nearest neighbors are now the first k items in distances. Create a new variable named neighbors and set it equal to the first k items of distances. You can use Python's built-in slice function.

For example, lst[2:5] will give you a list of the items at indices 2, 3, and 4 of lst.

Return neighbors.

**6.**

Test the classify function and print the results. The three parameters you should use are:

* [.4, .2, .9]
* movie\_dataset
* 5

Take a look at the 5 nearest neighbors. In the next exercise, we'll check to see how many of those neighbors are good and how many are bad.

**Count Neighbors**

The K-Nearest Neighbor Algorithm:

1. Normalize the data
2. Find the k nearest neighbors
3. **Classify the new point based on those neighbors**

We've now found the k nearest neighbors, and have stored them in a list that looks like this:

[

[0.083, 'Lady Vengeance'],

[0.236, 'Steamboy'],

...

...

[0.331, 'Godzilla 2000']

]

Our goal now is to count the number of good movies and bad movies in the list of neighbors. If more of the neighbors were good, then the algorithm will classify the unknown movie as good. Otherwise, it will classify it as bad.

In order to find the class of each of the labels, we'll need to look at our movie\_labels dataset. For example, movie\_labels['Akira'] would give us 1 because Akira is classified as a good movie.

You may be wondering what happens if there’s a tie. What if k = 8 and four neighbors were good and four neighbors were bad? There are different strategies, but one way to break the tie would be to choose the class of the closest point.

**1.**

Our classify function now needs to have knowledge of the labels. Add a parameter named labels to classify. It should be the third parameter.

**2.**

Continue writing your classify function.

Create two variables named num\_good and num\_bad and set them each at 0. Use a for loop to loop through every movie in neighbors. Store their title in a variable called title.

Remember, every neighbor is a list of [distance, title] so the title can be found at index 1.

For now, return title at the end of your function (outside of the loop).

**3.**

Use labels and title to find the label of each movie:

* If that label is a 0, add one to num\_bad.
* If that label is a 1, add one to num\_good.

For now, return num\_good at the end of your function.

**4.**

We can finally classify our unknown movie:

* If num\_good is greater than num\_bad, return a 1.
* Otherwise, return a 0.

**5.**

Call classify using the following parameters and print the result.

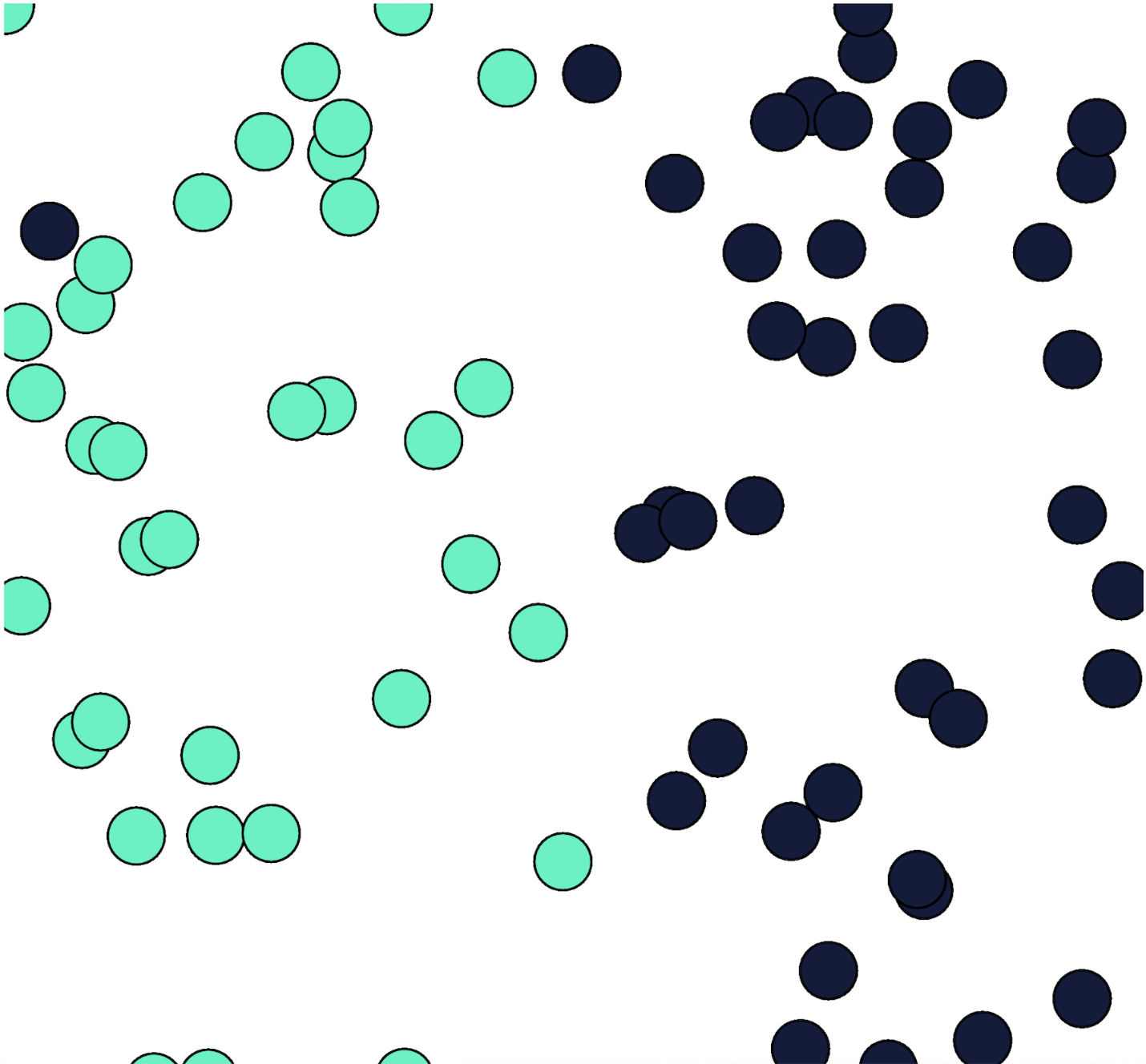
* [.4, .2, .9] as the movie you're looking to classify.
* movie\_dataset the training dataset.
* movie\_labels as the training labels.
* k = 5

Does the system predict this movie will be good or bad?

# Choosing K

In the previous exercise, we found that our classifier got one point in the training set correct. Now we can test every point to calculate the validation accuracy.

The validation accuracy changes as k changes. The first situation that will be useful to consider is when k is very small. Let's say k = 1. We would expect the validation accuracy to be fairly low due to overfitting. Overfitting is a concept that will appear almost any time you are writing a machine learning algorithm. Overfitting occurs when you rely too heavily on your training data; you assume that data in the real world will always behave exactly like your training data. In the case of K-Nearest Neighbors, overfitting happens when you don't consider enough neighbors. A single outlier could drastically determine the label of an unknown point. Consider the image below.



The dark blue point in the top left corner of the graph looks like a fairly significant outlier. When k = 1, all points in that general area will be classified as dark blue when it should probably be classified as green. Our classifier has relied too heavily on the small quirks in the training data.

On the other hand, if k is very large, our classifier will suffer from underfitting. Underfitting occurs when your classifier doesn't pay enough attention to the small quirks in the training set. Imagine you have 100 points in your training set and you set k = 100. Every single unknown point will be classified in the same exact way. The distances between the points don't matter at all! This is an extreme example, however, it demonstrates how the classifier can lose understanding of the training data if k is too big.

**1.**

Begin by creating a function called find\_validation\_accuracy that takes five parameters. The parameters should be training\_set, training\_labels, validation\_set, validation\_labels, and k.

**2.**

Create a variable called num\_correct and have it begin at 0.0. Loop through the movies of validation\_set, and call classify using each movie's data, the training\_set, the training\_labels, and k. Store the result in a variable called guess. For now, return guess outside of your loop.

Remember, the movie's data can be found by using validation\_set[title].

**3.**

Inside the for loop, compare guess to the corresponding label in validation\_labels. If they were equal, add 1 to num\_correct. For now, outside of the for loop, return num\_correct

**4.**

Outside the for loop return the validation error. This should be num\_correct divided by the total number of points in the validation set.

**5.**

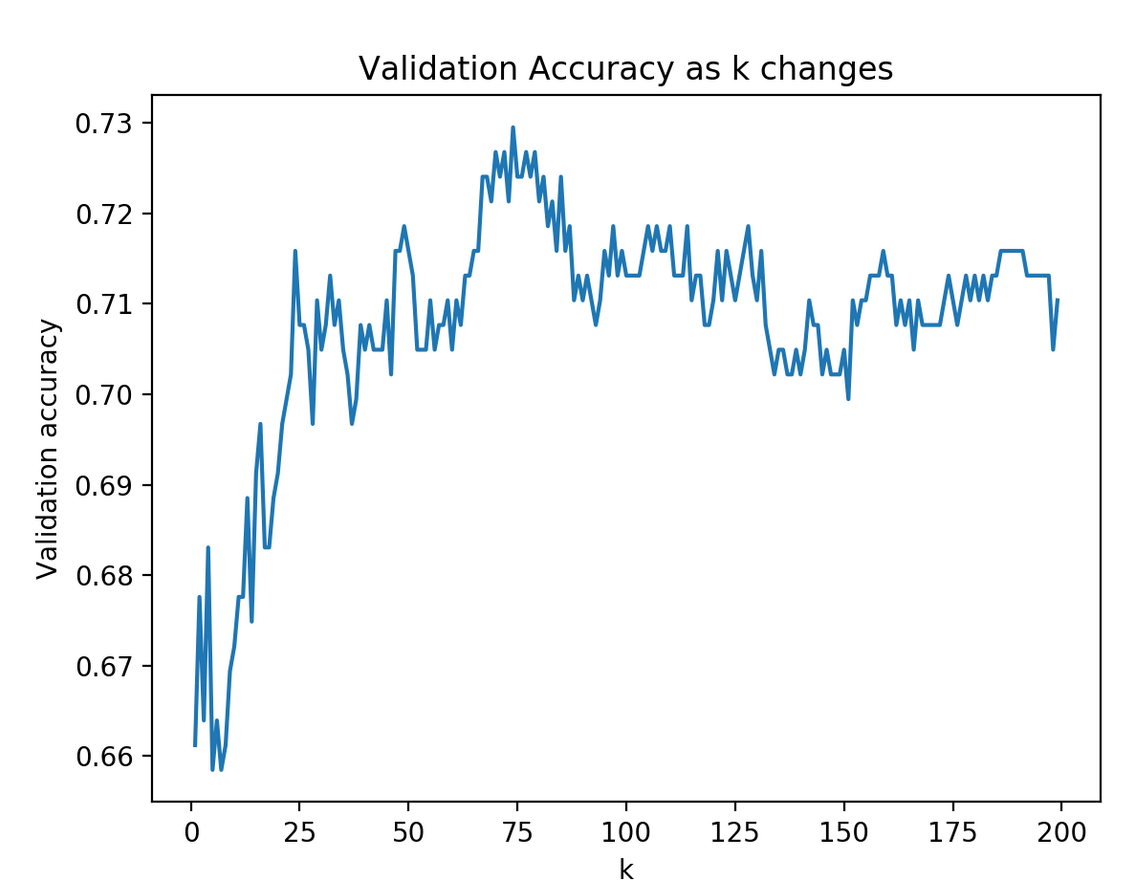
Call find\_validation\_accuracy with k = 3. Print the results The code should take a couple of seconds to run.

# Graph of K

The graph to the right shows the validation accuracy of our movie classifier as k increases. When k is small, overfitting occurs and the accuracy is relatively low. On the other hand, when k gets too large, underfitting occurs and accuracy starts to drop.

Instructions

What seems to be the best k for this dataset?



# Using sklearn

You've now written your own K-Nearest Neighbor classifier from scratch! However, rather than writing your own classifier every time, you can use Python's sklearn library. sklearn is a Python library specifically used for Machine Learning. It has an amazing number of features, but for now, we're only going to investigate its K-Nearest Neighbor classifier.

There are a couple of steps we'll need to go through in order to use the library. First, you need to create a KNeighborsClassifier object. This object takes one parameter - k. For example, the code below will create a classifier where k = 3

classifier = KNeighborsClassifier(n\_neighbors = 3)

Next, we'll need to train our classifier. The .fit() method takes two parameters. The first is a list of points, and the second is the labels associated with those points. So for our movie example, we might have something like this

training\_points = [

[0.5, 0.2, 0.1],

[0.9, 0.7, 0.3],

[0.4, 0.5, 0.7]

]

training\_labels = [0, 1, 1]

classifier.fit(training\_points, training\_labels)

Finally, after training the model, we can classify new points. The .predict() method takes a list of points that you want to classify. It returns a list of its guesses for those points.

unknown\_points = [

[0.2, 0.1, 0.7],

[0.4, 0.7, 0.6],

[0.5, 0.8, 0.1]

]

guesses = classifier.predict(unknown\_points)

**1.**

We've imported sklearn for you. Create a KNeighborsClassifier named classifier that uses k=5.

**2.**

We've also imported some movie data. Train your classifier using movie\_dataset as the training points and labels as the training labels.

**3.**

Let's classify some movies. Classify the following movies: [.45, .2, .5], [.25, .8, .9],[.1, .1, .9]. Print the classifications!

Which movies were classified as good movies and which were classified as bad movies?

Remember, those three numbers associated with a movie are the normalized budget, run time, and year of release.

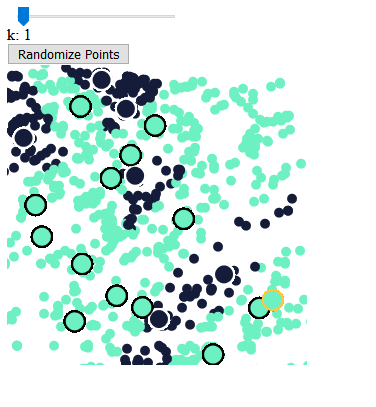
**Review**

Congratulations! You just implemented your very own classifier from scratch and used Python's sklearn library. In this lesson, you learned some techniques very specific to the K-Nearest Neighbor algorithm, but some general machine learning techniques as well. Some of the major takeaways from this lesson include:

* Data with n features can be conceptualized as points lying in n-dimensional space.
* Data points can be compared by using the distance formula. Data points that are similar will have a smaller distance between them.
* A point with an unknown class can be classified by finding the k nearest neighbors
* To verify the effectiveness of a classifier, data with known classes can be split into a training set and a validation set. Validation error can then be calculated.
* Classifiers have parameters that can be tuned to increase their effectiveness. In the case of K-Nearest Neighbors, k can be changed.
* A classifier can be trained improperly and suffer from overfitting or underfitting. In the case of K-Nearest Neighbors, a low k often leads to overfitting and a large k often leads to underfitting.
* Python's sklearn library can be used for many classification and machine learning algorithms.

To the right is an interactive visualization of K-Nearest Neighbors. If you move your mouse over the canvas, the location of your mouse will be classified as either green or blue. The nearest neighbors to your mouse are highlighted in yellow. Use the slider to change k to see how the boundaries of the classification change.

If you find any interesting patterns, share it with us on Twitter!



# Regression

# Regression

The K-Nearest Neighbors algorithm is a powerful supervised machine learning algorithm typically used for classification. However, it can also perform regression.

In this lesson, we will use the movie dataset that was used in the [K-Nearest Neighbors classifier lesson](https://www.codecademy.com/content-items/e6a14b06673aae14c8262dd5c3998401/exercises/knn). However, instead of classifying a new movie as either good or bad, we are now going to predict its IMDb rating as a real number.

This process is almost identical to classification, except for the final step. Once again, we are going to find the k nearest neighbors of the new movie by using the distance formula. However, instead of counting the number of good and bad neighbors, the regressor averages their IMDb ratings.

For example, if the three nearest neighbors to an unrated movie have ratings of 5.0, 9.2, and 6.8, then we could predict that this new movie will have a rating of 7.0.

**1.**

We've imported most of the K-Nearest Neighbor algorithm. Before we dive into finishing the regressor, let's refresh ourselves with the data.

At the bottom of your code, print movie\_dataset["Life of Pi"]. You should see a list of three values. These values are the [normalized](https://www.codecademy.com/articles/normalization) values for the movie's budget, runtime, and release year.

**2.**

Print the rating for "Life of Pi". This can be found in movie\_ratings.

**3.**

We've included the majority of the K-Nearest Neighbor algorithm in the predict() function. Right now, the variable neighbors stores a list of [distance, title] pairs.

Loop through every neighbor and find its rating in movie\_ratings. Add those ratings together and return that sum divided by the total number of neighbors.

**4.**

Call predict with the following parameters:

* [0.016, 0.300, 1.022]
* movie\_dataset
* movie\_ratings
* 5

Print the result.

Note that the list [0.016, 0.300, 1.022] is the normalized budget, runtime, and year of the movie Incredibles 2! The normalized year is larger than 1 because our training set only had movies that were released between 1927 and 2016 — Incredibles 2 was released in 2018.

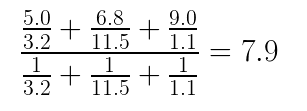
# Weighted Regression

We're off to a good start, but we can be even more clever in the way that we compute the average. We can compute a weighted average based on how close each neighbor is.

Let's say we're trying to predict the rating of movie X and we've found its three nearest neighbors. Consider the following table:

| **Movie** | **Rating** | **Distance to movie X** |
| --- | --- | --- |
| A | 5.0 | 3.2 |
| B | 6.8 | 11.5 |
| C | 9.0 | 1.1 |

If we find the mean, the predicted rating for X would be 6.93. However, movie X is most similar to movie C, so movie C's rating should be more important when computing the average. Using a weighted average, we can find movie X's rating:



The numerator is the sum of every rating divided by their respective distances. The denominator is the sum of one over every distance. Even though the ratings are the same as before, the weighted average has now gone up to 7.9.

**1.**

Let's redo our predict() function so it computes the weighted average.

Before you begin looping through the neighbors, create a variable named numerator and set it to 0. Loop through every neighbor and add the neighbor's rating (found in movie\_ratings) divided by the neighbor's distance to numerator.

For now, return numerator.

**2.**

Let's now calculate the denominator of the weighted average. Before your loop, create a variable named denominator and set it equal to 0.

Inside your for loop, add 1 divided by the neighbor's distance to denominator.

Outside the loop, return numerator/denominator.

**3.**

Once again call your predict function using Incredibles 2's features. Those features were [0.016, 0.300, 1.022]. Set k = 5. Print the results.

How did using a weighted average change the predicted rating? Remember, before calculating the weighted average the prediction was 6.86.

**Scikit-learn**

Now that you've written your own K-Nearest Neighbor regression model, let's take a look at scikit-learn's implementation. The KNeighborsRegressor class is very similar to KNeighborsClassifier.

We first need to create the regressor. We can use the parameter n\_neighbors to define our value for k.

We can also choose whether or not to use a weighted average using the parameter weights. If weights equals "uniform", all neighbors will be considered equally in the average. If weights equals "distance", then a weighted average is used.

classifier = KNeighborsRegressor(n\_neighbors = 3, weights = "distance")

Next, we need to fit the model to our training data using the .fit() method. .fit() takes two parameters. The first is a list of points, and the second is a list of values associated with those points.

training\_points = [

[0.5, 0.2, 0.1],

[0.9, 0.7, 0.3],

[0.4, 0.5, 0.7]

]

training\_labels = [5.0, 6.8, 9.0]

classifier.fit(training\_points, training\_labels)

Finally, we can make predictions on new data points using the .predict() method. .predict() takes a list of points and returns a list of predictions for those points.

unknown\_points = [

[0.2, 0.1, 0.7],

[0.4, 0.7, 0.6],

[0.5, 0.8, 0.1]

]

guesses = classifier.predict(unknown\_points)

**1.**

Create a KNeighborsRegressor named regressor where n\_neighbors = 5 and weights = "distance".

**2.**

We've also imported some movie data. Train your classifier using movie\_dataset as the training points and movie\_ratings as the training values.

**3.**

Let's predict some movie ratings. Predict the ratings for the following movies:

* [0.016, 0.300, 1.022],
* [0.0004092981, 0.283, 1.0112],
* [0.00687649, 0.235, 1.0112].

These three lists are the features for *Incredibles 2*, *The Big Sick*, and *The Greatest Showman*. Those three numbers associated with a movie are the normalized budget, runtime, and year of release.

Print the predictions!

**Review**

Great work! Here are some of the major takeaways from this lesson:

* The K-Nearest Neighbor algorithm can be used for regression. Rather than returning a classification, it returns a number.
* By using a weighted average, data points that are extremely similar to the input point will have more of a say in the final result.
* scikit-learn has an implementation of a K-Nearest Neighbor regressor named KNeighborsRegressor.

In the browser, you'll find an example of a K-Nearest Neighbor regressor in action. Instead of the training data coming from IMDb ratings, you can create the training data yourself! Rate the movies that you have seen. Once you've rated more than k movies, a K-Nearest Neighbor regressor will train on those ratings. It will then make predictions for every movie that you haven't seen.

As you add more and more ratings, the predictor should become more accurate. After all, the regressor needs information from the user in order to make personalized recommendations. As a result, the system is somewhat useless to brand new users — it takes some time for the system to "warm up" and get enough data about a user. This conundrum is an example of the *cold start problem*.