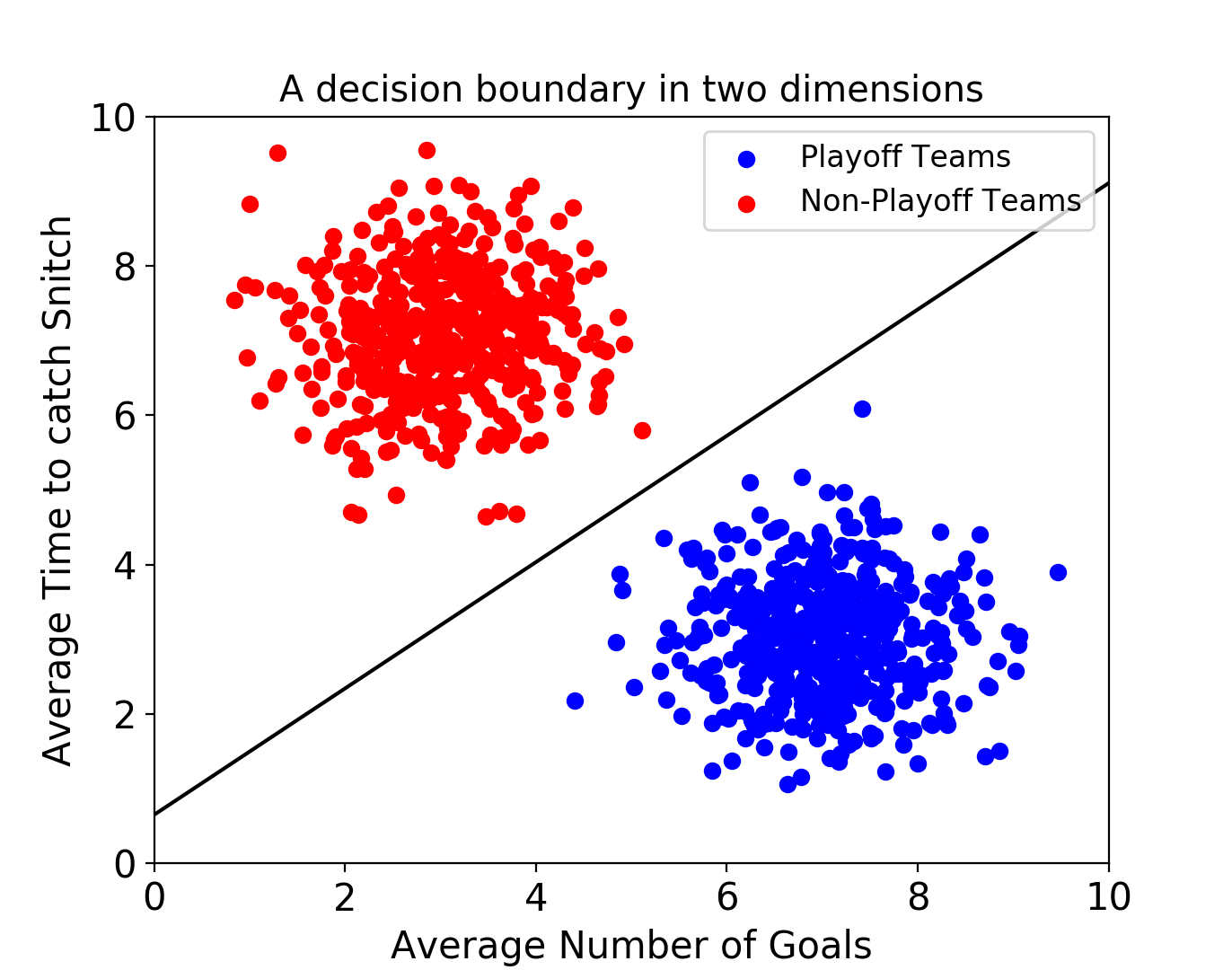
**Support Vector Machines**

A **Support Vector Machine** (SVM) is a powerful supervised machine learning model used for classification. An SVM makes classifications by defining a decision boundary and then seeing what side of the boundary an unclassified point falls on. In the next few exercises, we’ll learn how these decision boundaries get defined, but for now, know that they’re defined by using a training set of classified points. That’s why SVMs are *supervised* machine learning models.

Decision boundaries are easiest to wrap your head around when the data has two features. In this case, the decision boundary is a line. Take a look at the example below.

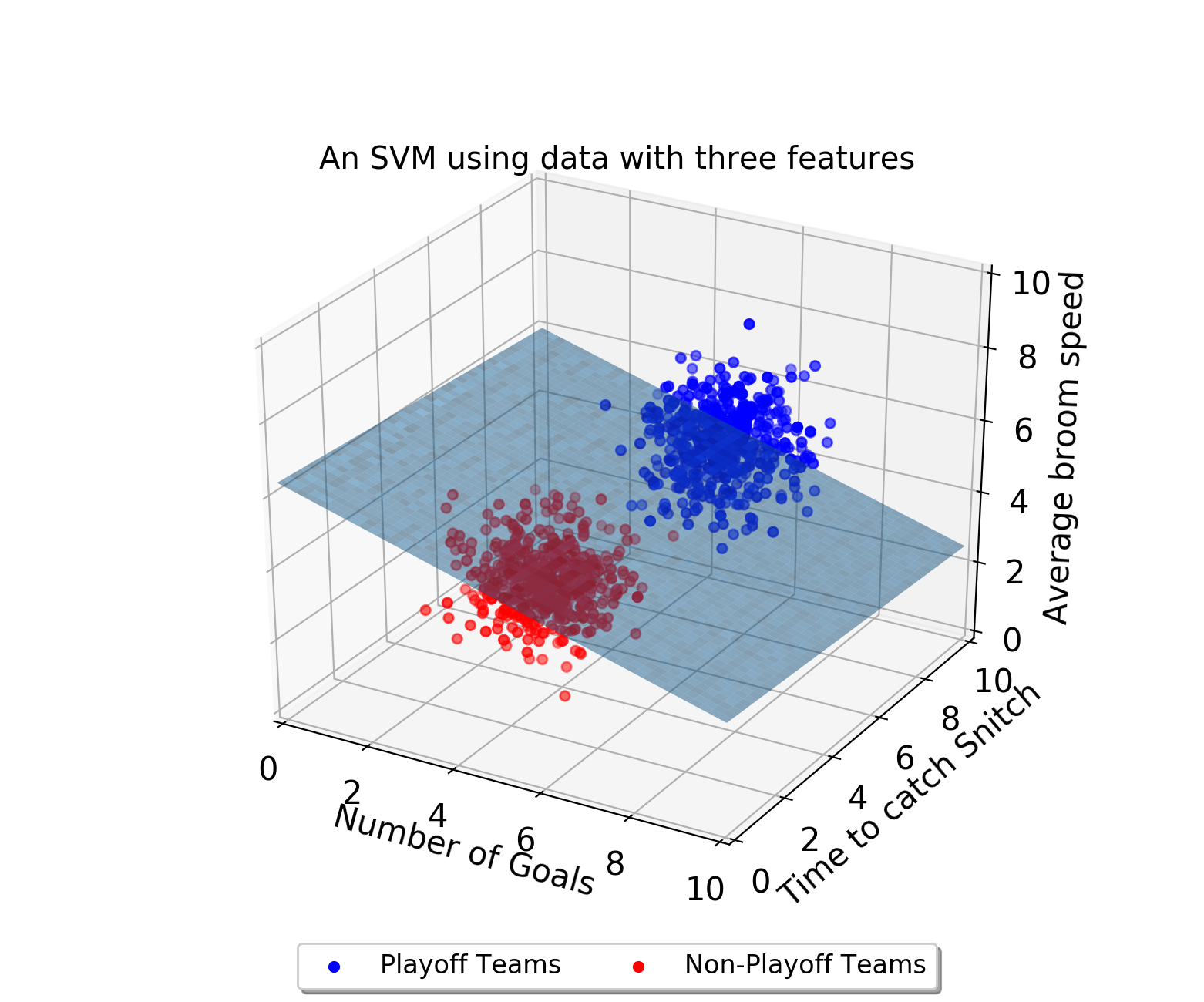


Note that if the labels on the figures in this lesson are too small to read, you can resize this pane to increase the size of the images.

This SVM is using data about fictional games of Quidditch from the Harry Potter universe! The classifier is trying to predict whether a team will make the playoffs or not. Every point in the training set represents a "historical" Quidditch team. Each point has two features — the average number of goals the team scores and the average number of minutes it takes the team to catch the Golden Snitch.

After finding a decision boundary using the training set, you could give the SVM an unlabeled data point, and it will predict whether or not that team will make the playoffs.

Decision boundaries exist even when your data has more than two features. If there are three features, the decision boundary is now a plane rather than a line.



As the number of dimensions grows past 3, it becomes very difficult to visualize these points in space. Nonetheless, SVMs can still find a decision boundary. However, rather than being a separating line, or a separating plane, the decision boundary is called a *separating hyperplane*.

**1.**

Run the code to see two graphs appear. Right now they should be identical. We're going to fix the bottom graph so it has a good decision boundary. Why is this decision boundary bad?

**2.**

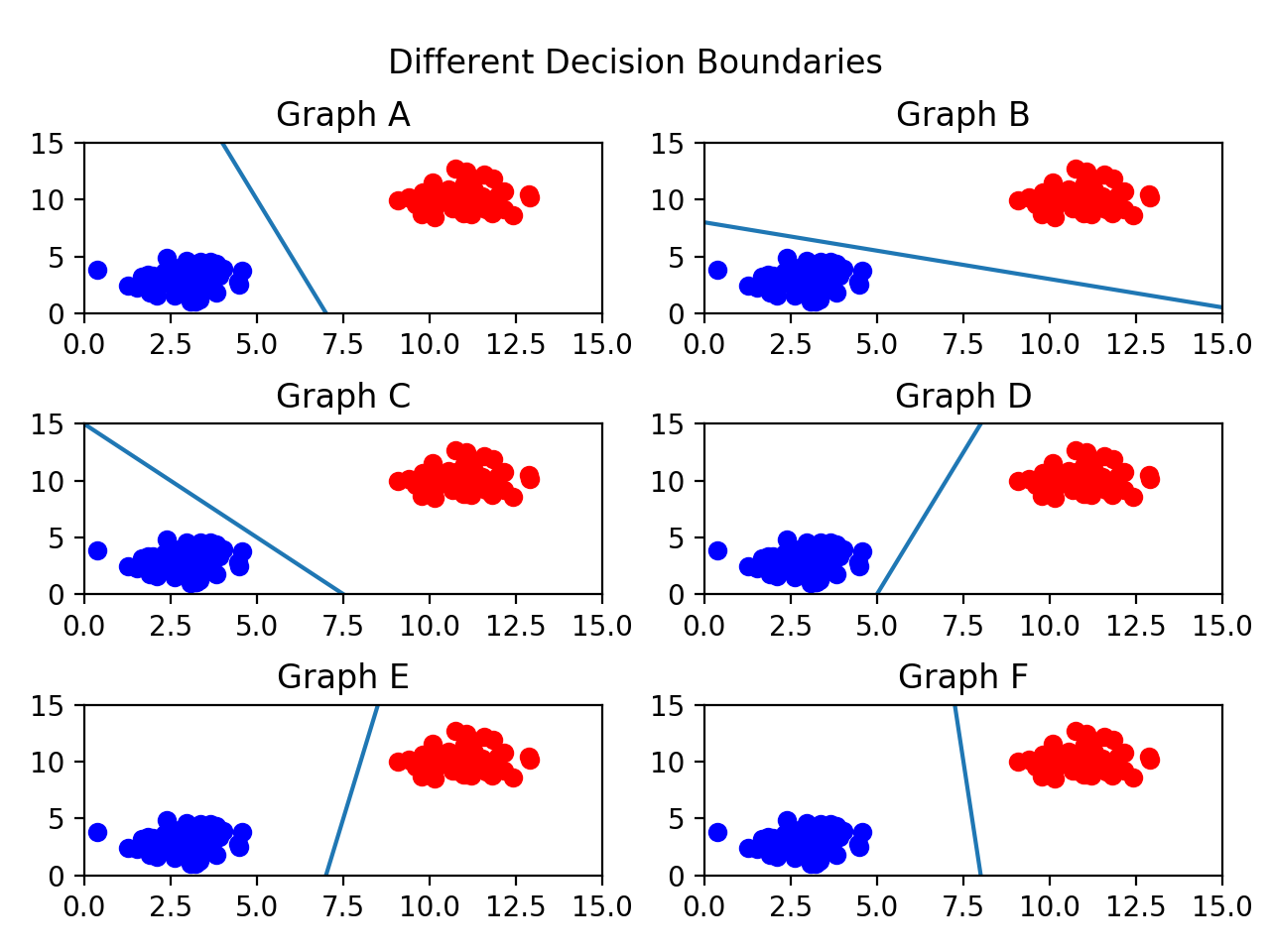
Let's shift the line on the bottom graph to make it separate the two clusters. The slope of the line looks pretty good, so let's keep that at -2.

We want to move the boundary up, so change intercept\_two so the line separates the two clusters.



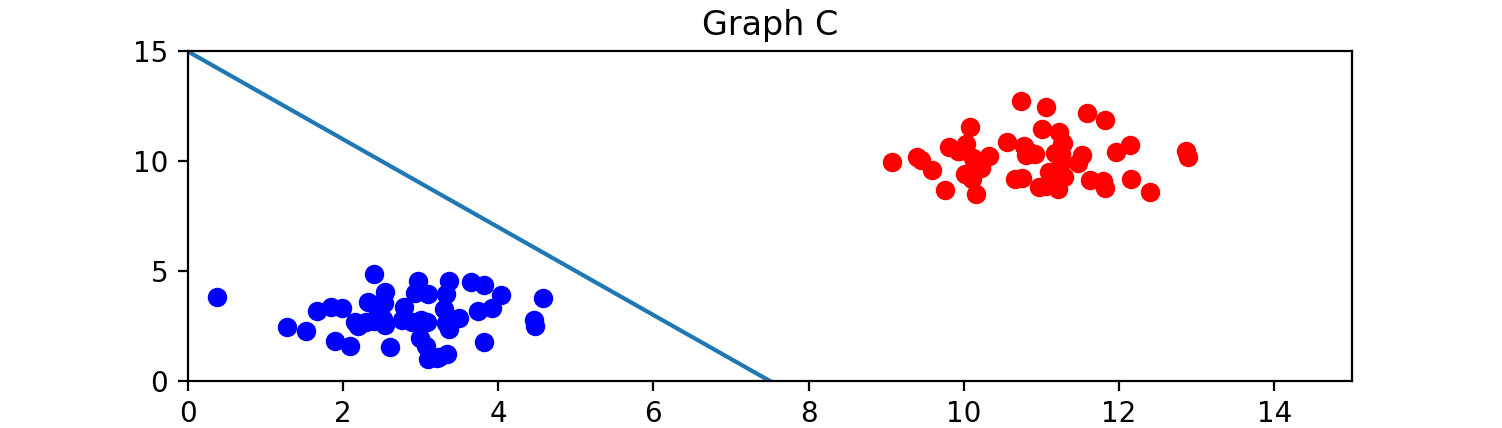
# Optimal Decision Boundaries

One problem that SVMs need to solve is figuring out what decision boundary to use. After all, there could be an infinite number of decision boundaries that correctly separate the two classes. Take a look at the image below:



There are so many valid decision boundaries, but which one is best? In general, we want our decision boundary to be as far away from training points as possible.

Maximizing the distance between the decision boundary and points in each class will decrease the chance of false classification. Take graph C for example.



The decision boundary is close to the blue class, so it is possible that a new point close to the blue cluster would fall on the red side of the line.

Out of all the graphs shown here, graph F has the best decision boundary.

Run the code. Both graphs have suboptimal decision boundaries. Why? We're going to fix the bottom graph.

**2.**

We're going to have to make the decision boundary much flatter, which means we first need to lower its y-intercept. Change intercept\_two to be 8.

**3.**

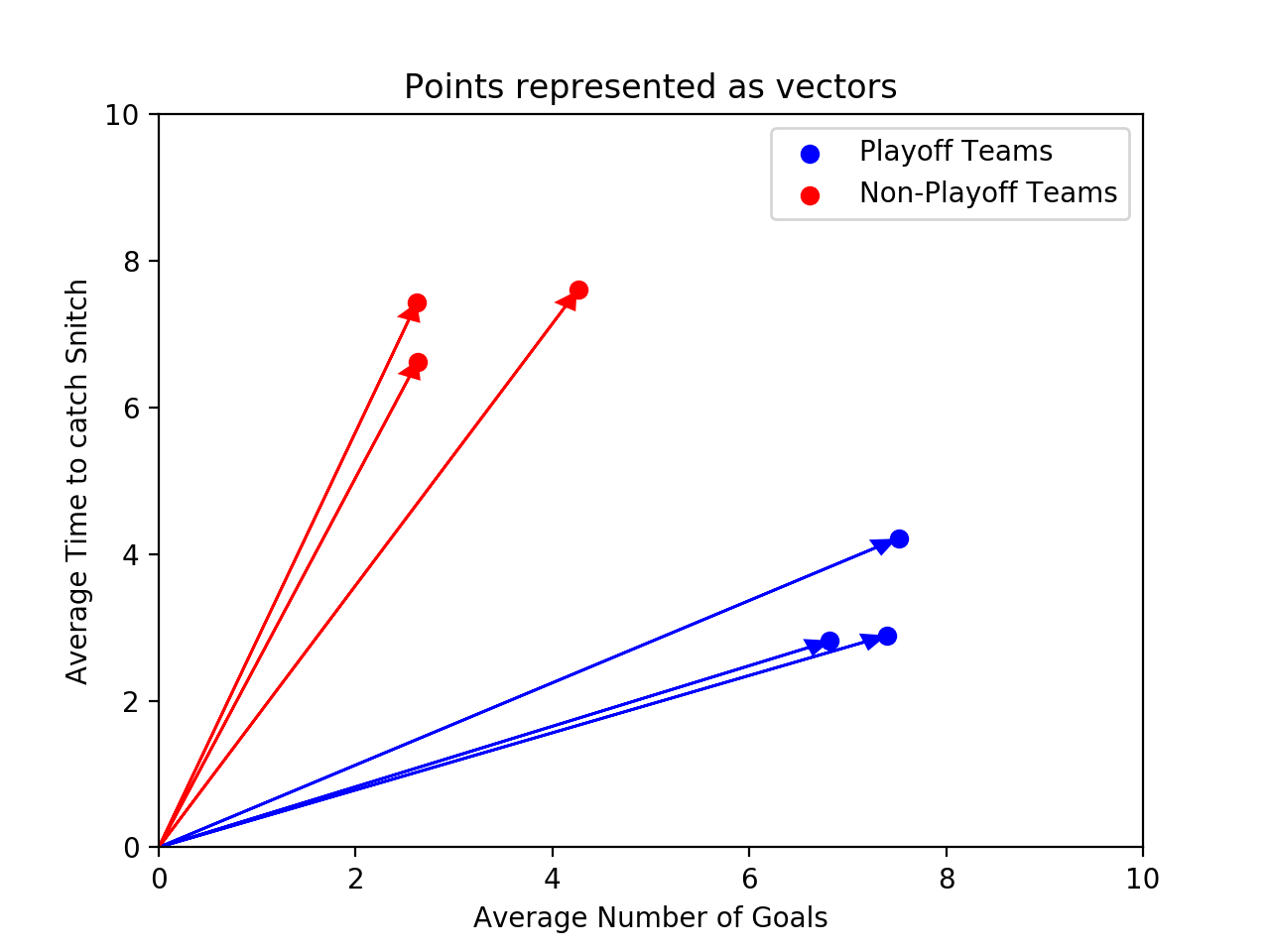
Next, we want the slope to be pretty flat. Change the value of slope\_two. The resulting line should split the two clusters.



# Support Vectors and Margins

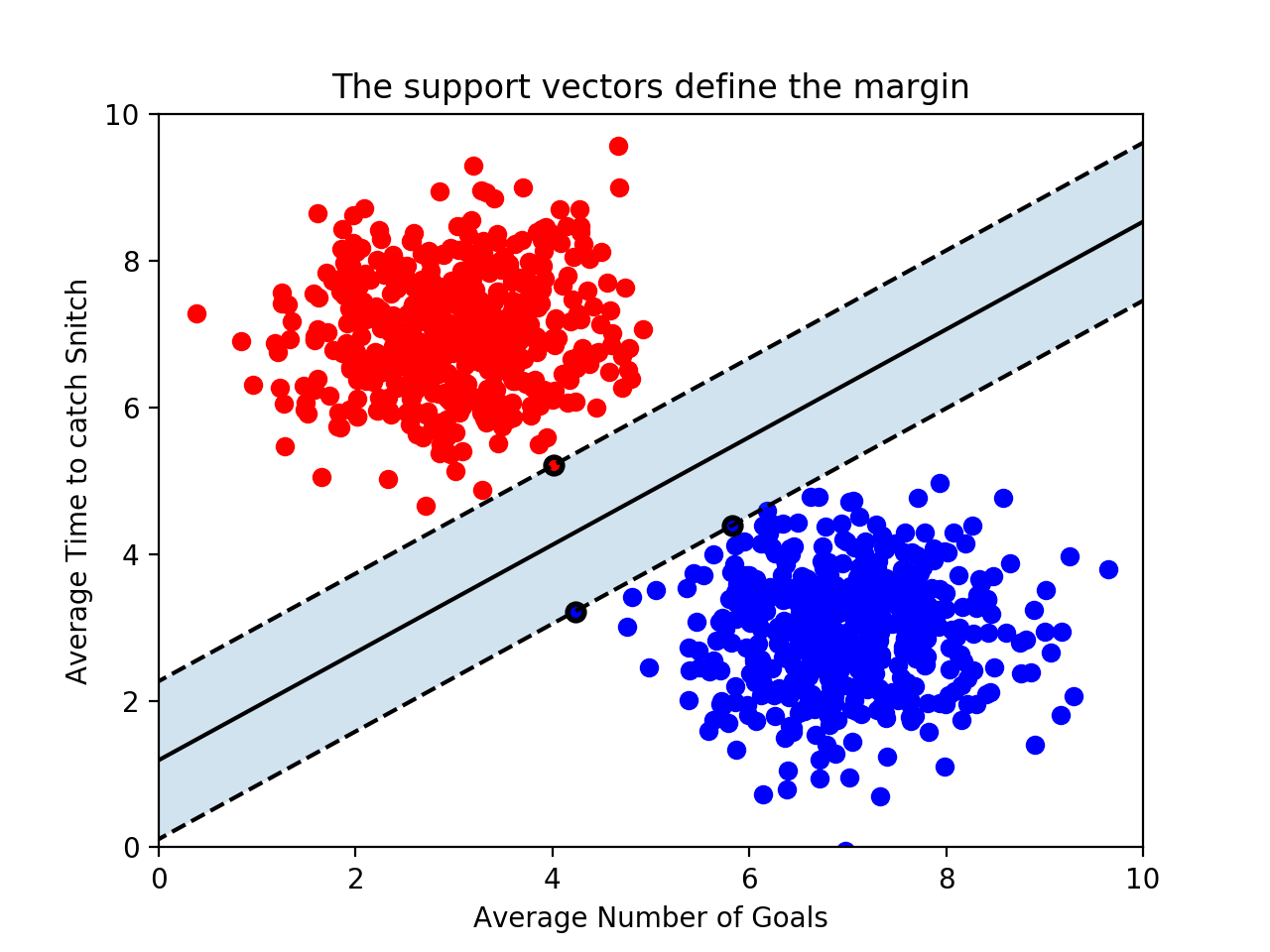
We now know that we want our decision boundary to be as far away from our training points as possible. Let’s introduce some new terms that can help explain this idea.

The support vectors are the points in the training set closest to the decision boundary. In fact, these vectors are what define the decision boundary. But why are they called vectors? Instead of thinking about the training data as points, we can think of them as vectors coming from the origin.



These vectors are crucial in defining the decision boundary — that’s where the "support" comes from. If you are using n features, there are at least n+1 support vectors.

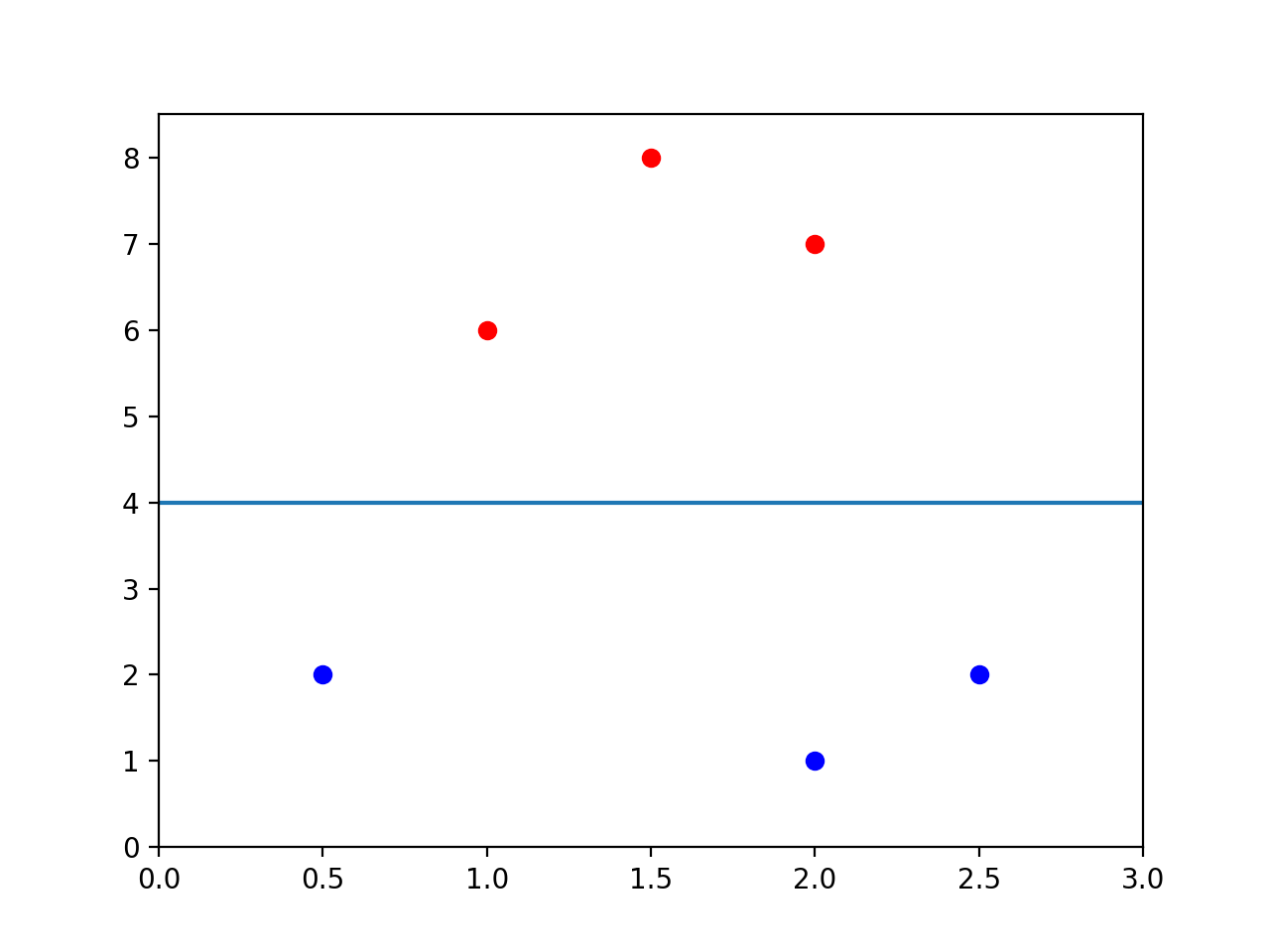
The distance between a support vector and the decision boundary is called the margin. We want to make the margin as large as possible. The support vectors are highlighted in the image below:



Because the support vectors are so critical in defining the decision boundary, many of the other training points can be ignored. This is one of the advantages of SVMs. Many supervised machine learning algorithms use every training point in order to make a prediction, even though many of those training points aren’t relevant. SVMs are fast because they only use the support vectors!

**1.**

What are the support vectors for the SVM pictured below? There should be 2 blue support vectors and 1 red support vector.



Finish defining red\_support\_vector, blue\_support\_vector\_one, and blue\_support\_vector\_two. Set them equal to the correct points. The point should be represented as a list like [1, 0.5].

**2.**

What is the size of the margin? Find the total distance between a support vector and the line by looking at the graph. Create a variable named margin\_size and set it equal to the correct number.

red\_support\_vector = [1, 6]

blue\_support\_vector\_one = [0.5, 2]

blue\_support\_vector\_two = [2.5, 2]

margin\_size = 2

**scikit-learn**

Now that we know the concepts behind SVMs we need to write the code that will find the decision boundary that maximizes the margin. All of the code that we’ve written so far has been guessing and checking — we don’t actually know if we’ve found the best line. Unfortunately, calculating the parameters of the best decision boundary is a fairly complex optimization problem. Luckily, Python’s scikit-learn library has implemented an SVM that will do this for us.

Note that while it is not important to understand how the optimal parameters are found, you should have a strong conceptual understanding of what the model is optimizing.

To use scikit-learn’s SVM we first need to create an SVC object. It is called an SVC because scikit-learn is calling the model a Support Vector Classifier rather than a Support Vector Machine.

classifier = SVC(kernel = 'linear')

We’ll soon go into what the kernel parameter is doing, but for now, let’s use a 'linear' kernel.

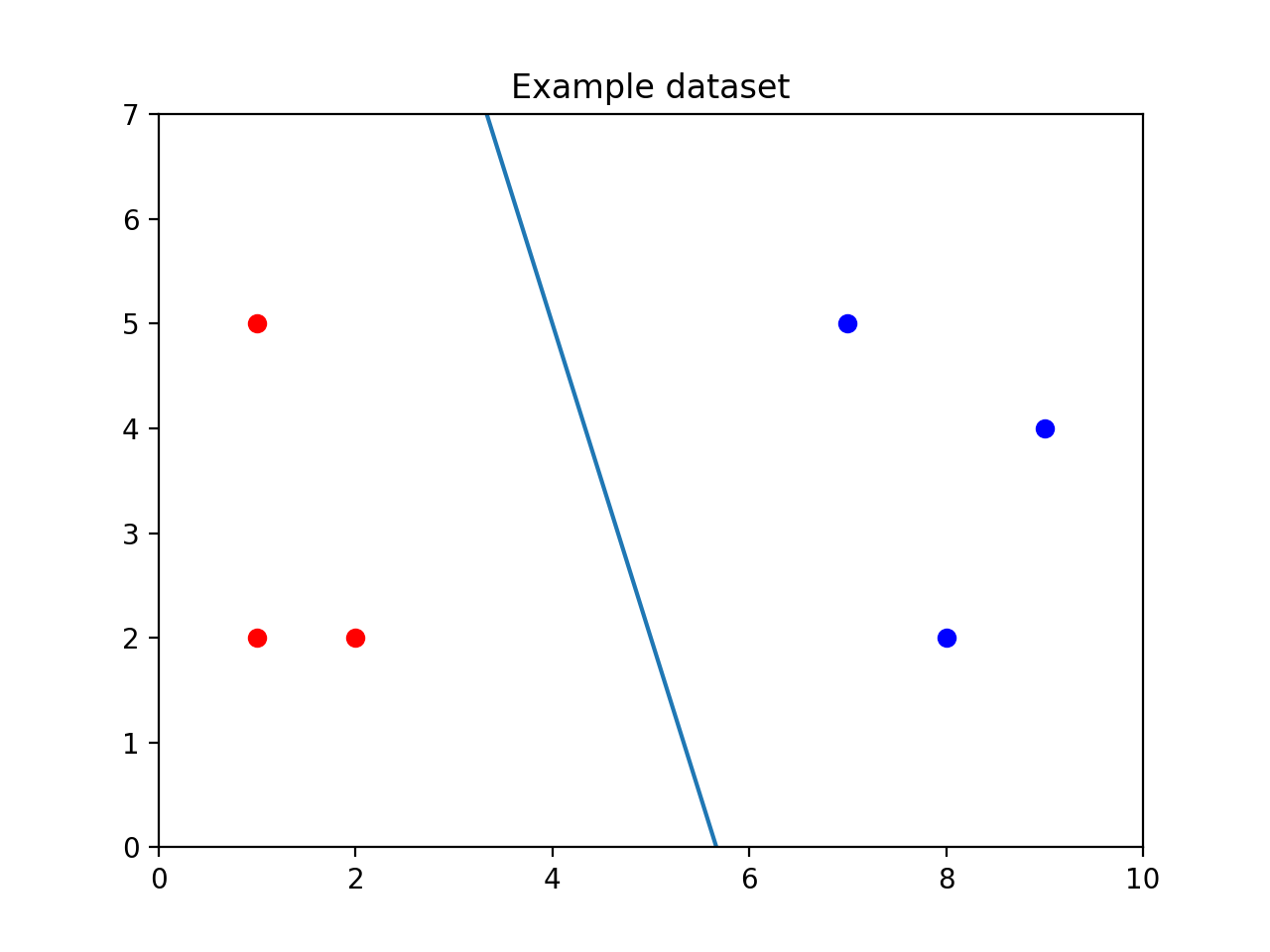
Next, the model needs to be trained on a list of data points and a list of labels associated with those data points. The labels are analogous to the color of the point — you can think of a 1 as a red point and a 0 as a blue point. The training is done using the .fit() method:

training\_points = [[1, 2], [1, 5], [2, 2], [7, 5], [9, 4], [8, 2]]

labels = [1, 1, 1, 0, 0, 0]

classifier.fit(training\_points, labels)

The graph of this dataset would look like this:

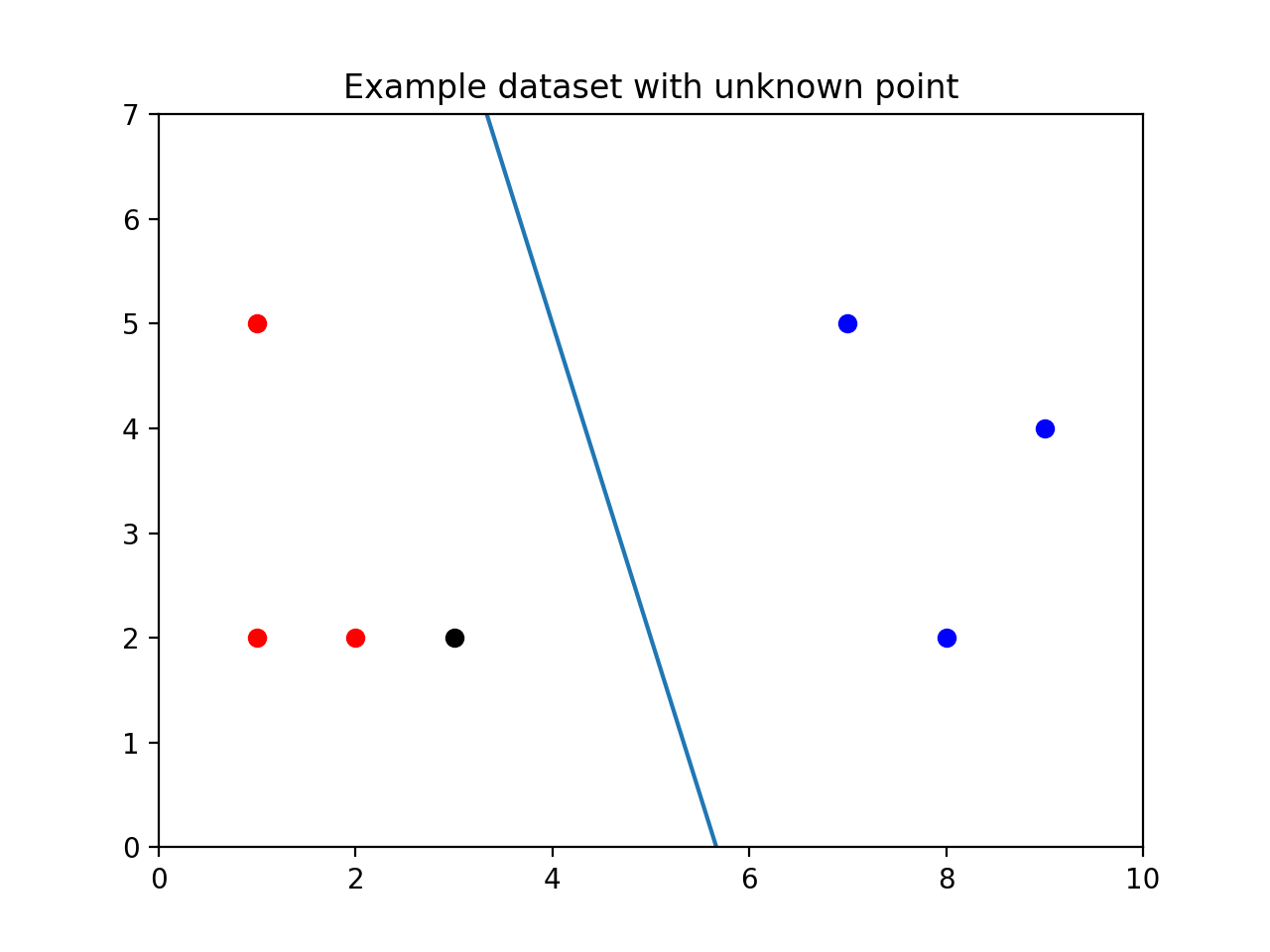


Calling .fit() creates the line between the points.

Finally, the classifier predicts the label of new points using the .predict() method. The .predict() method takes a list of points you want to classify. Even if you only want to classify one point, make sure it is in a list:

print(classifier.predict([[3, 2]]))

In the image below, you can see the unclassified point [3, 2] as a black dot. It falls on the red side of the line, so the SVM would predict it is red.



In addition to using the SVM to make predictions, you can inspect some of its attributes. For example, if you can print classifier.support\_vectors\_ to see which points from the training set are the support vectors.

In this case, the support vectors look like this:

[[7, 5],

[8, 2],

[2, 2]]

**1.**

Let's start by making a SVC object with kernel = 'linear'. Name the object classifier.

**2.**

We've imported the training set and labels for you. Call classifier's .fit() method using points and labels as parameters.

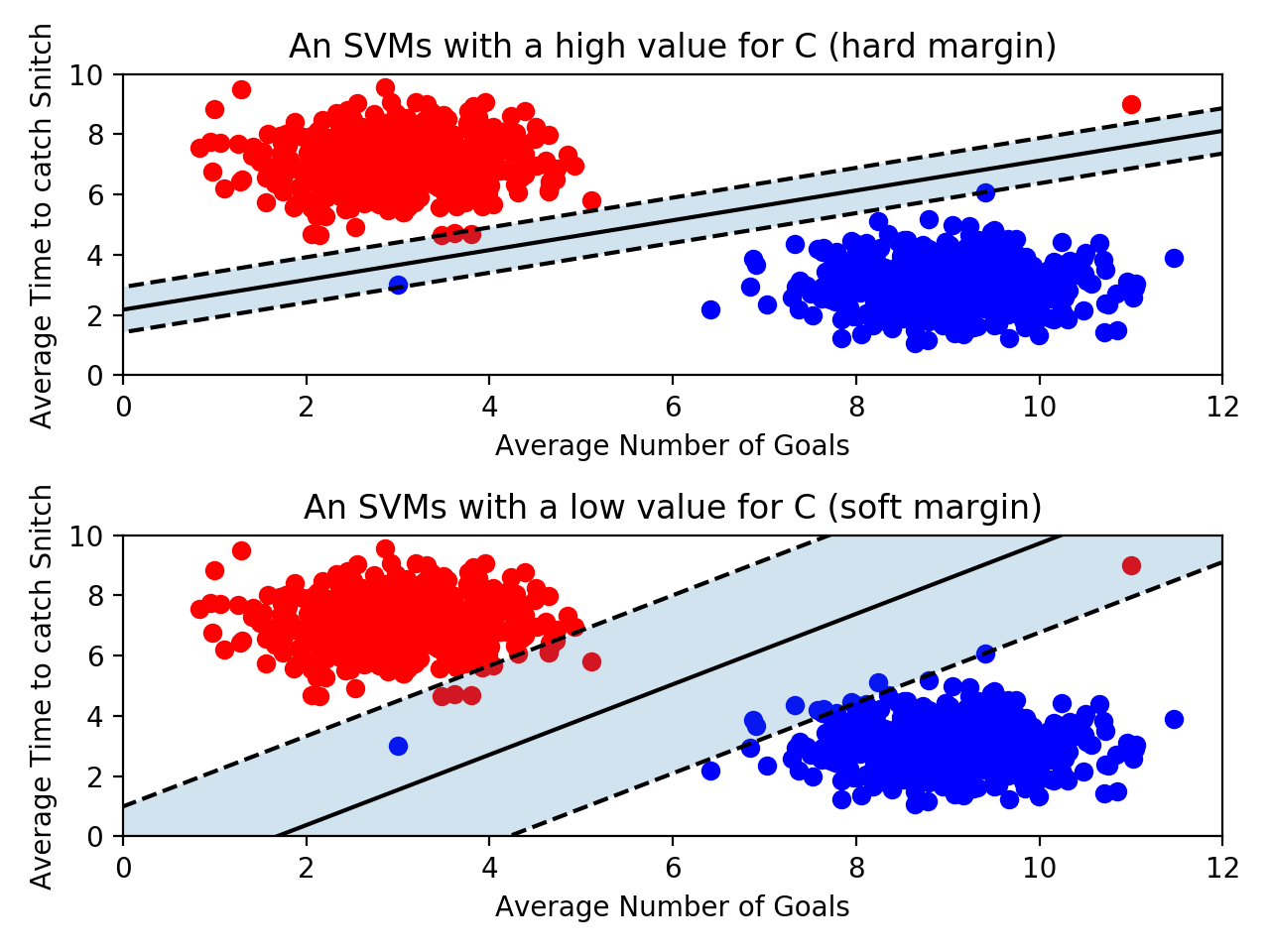
**3.**

We can now classify new points. Try classifying both [3, 4] and [6, 7]. Remember, the .predict() function expects a list of points to predict.

Print the results.

**Outliers**

SVMs try to maximize the size of the margin while still correctly separating the points of each class. As a result, outliers can be a problem. Consider the image below.



The size of the margin decreases when a single outlier is present, and as a result, the decision boundary changes as well. However, if we allowed the decision boundary to have some error, we could still use the original line.

SVMs have a parameter C that determines how much error the SVM will allow for. If C is large, then the SVM has a hard margin — it won’t allow for many misclassifications, and as a result, the margin could be fairly small. If C is too large, the model runs the risk of overfitting. It relies too heavily on the training data, including the outliers.

On the other hand, if C is small, the SVM has a soft margin. Some points might fall on the wrong side of the line, but the margin will be large. This is resistant to outliers, but if C gets too small, you run the risk of underfitting. The SVM will allow for so much error that the training data won’t be represented.

When using scikit-learn’s SVM, you can set the value of C when you create the object:

classifier = SVC(C = 0.01)

The optimal value of C will depend on your data. Don't always maximize margin size at the expense of error. Don't always minimize error at the expense of margin size. The best strategy is to validate your model by testing many different values for C.

**1.**

Run the code to see the SVM's current boundary line. Note that we've imported some helper functions we wrote named draw\_points and draw\_margins to help visualize the SVM.

import codecademylib3\_seaborn

import matplotlib.pyplot as plt

from sklearn.svm import SVC

from graph import points, labels, draw\_points, draw\_margin

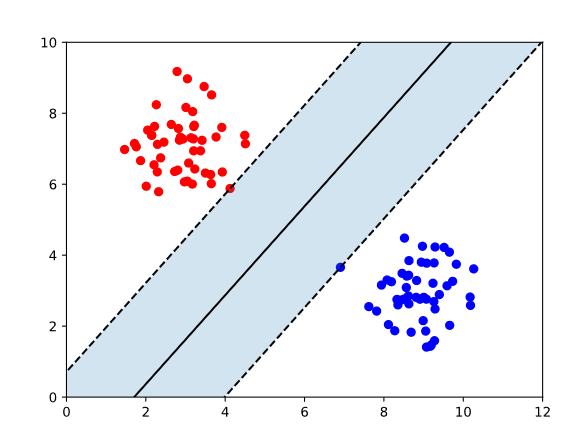
classifier = SVC(kernel='linear', C = 1)

classifier.fit(points, labels)

draw\_points(points, labels)

draw\_margin(classifier)

plt.show()



**2.**

Let's add an outlier! Before calling .fit(), append [3, 3] to points and append 0 to labels. This will add a blue point at [3, 3]

import codecademylib3\_seaborn

import matplotlib.pyplot as plt

from sklearn.svm import SVC

from graph import points, labels, draw\_points, draw\_margin

points.append([3,3])

labels.append(0)

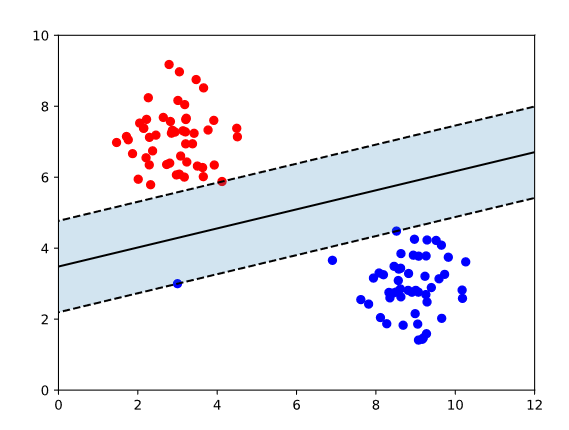
classifier = SVC(kernel='linear', C = 1)

classifier.fit(points, labels)

draw\_points(points, labels)

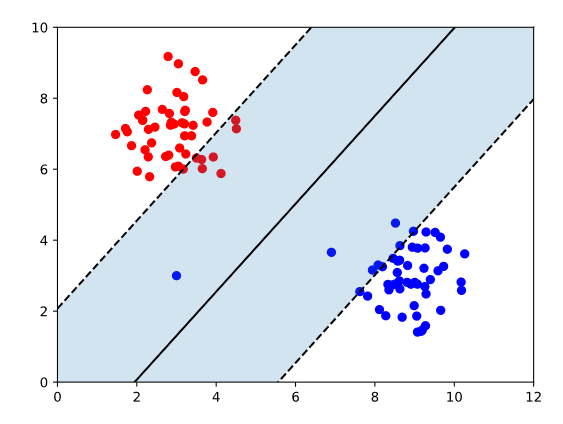
draw\_margin(classifier)

plt.show()



**3.**

Right now, our classifier has hard margins because C = 1. Change the value of C to 0.01 to see what the SVM looks like with soft margins.

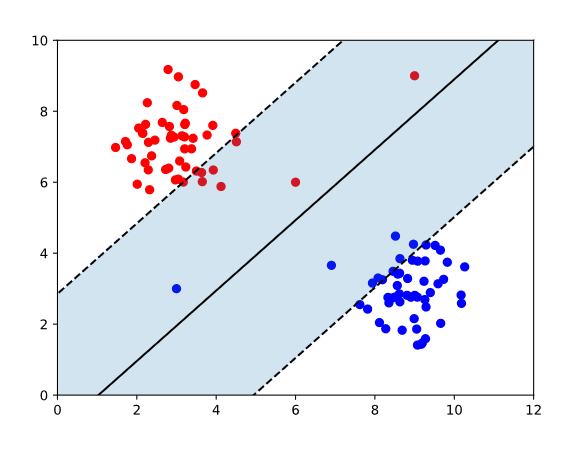


**4.**

append at least two more points to points. If you want the points to appear on the graph, make sure their x and y values are between 0 and 12.

Make sure to also append a label to labels for every point you add. A 0 will make the point blue and a 1 will make the point red.

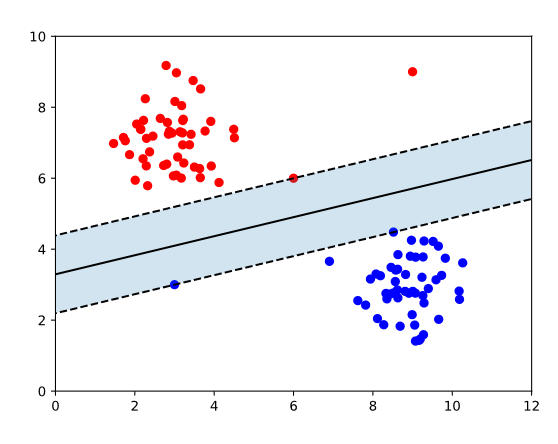
Make sure to add the points before training the SVM.



**5.**

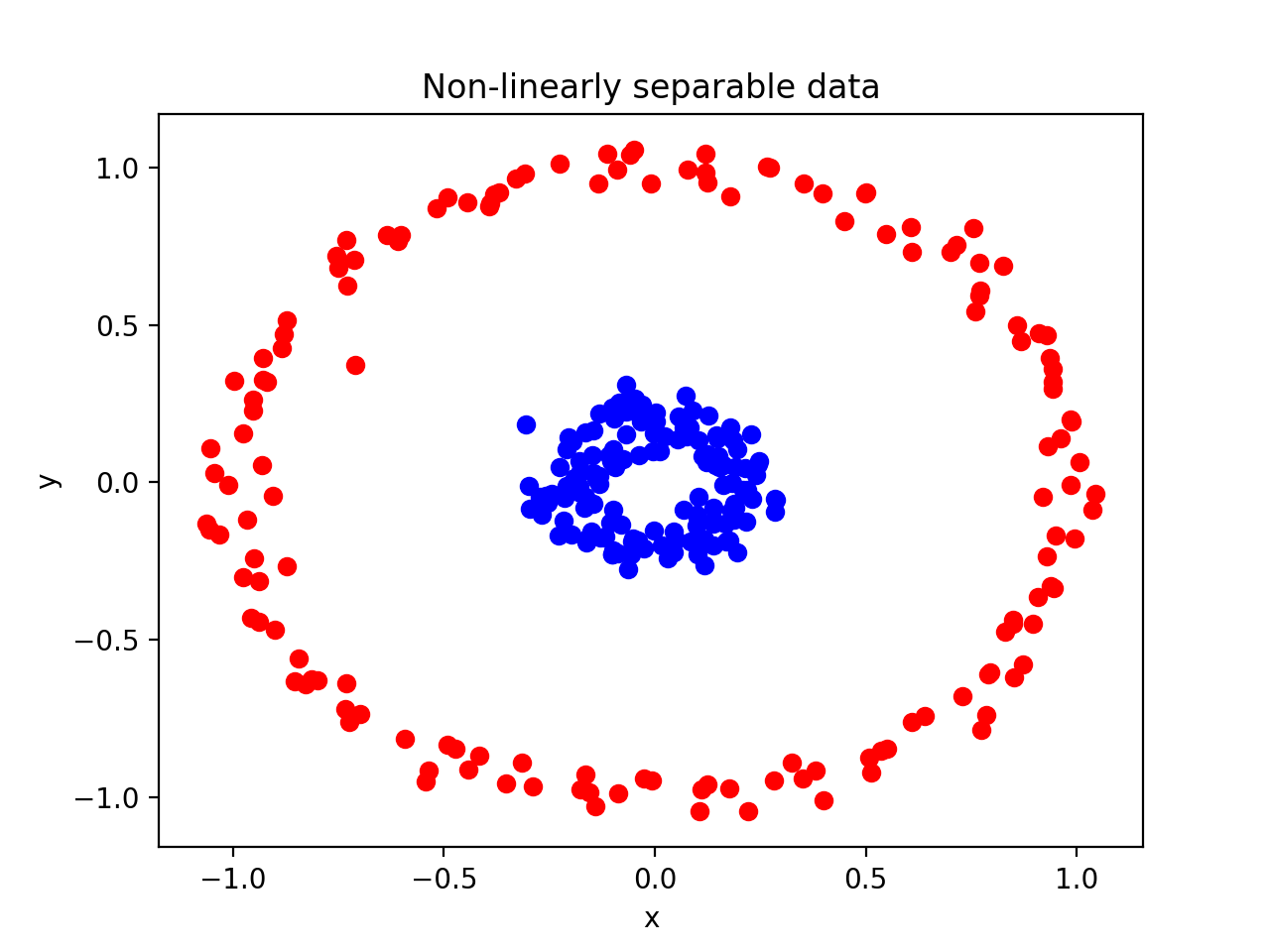
Play around with the C variable to see how the decision boundary changes with your new points added. Change C to be a value between 0.01 and 1.

C= 0.5



**Kernels**

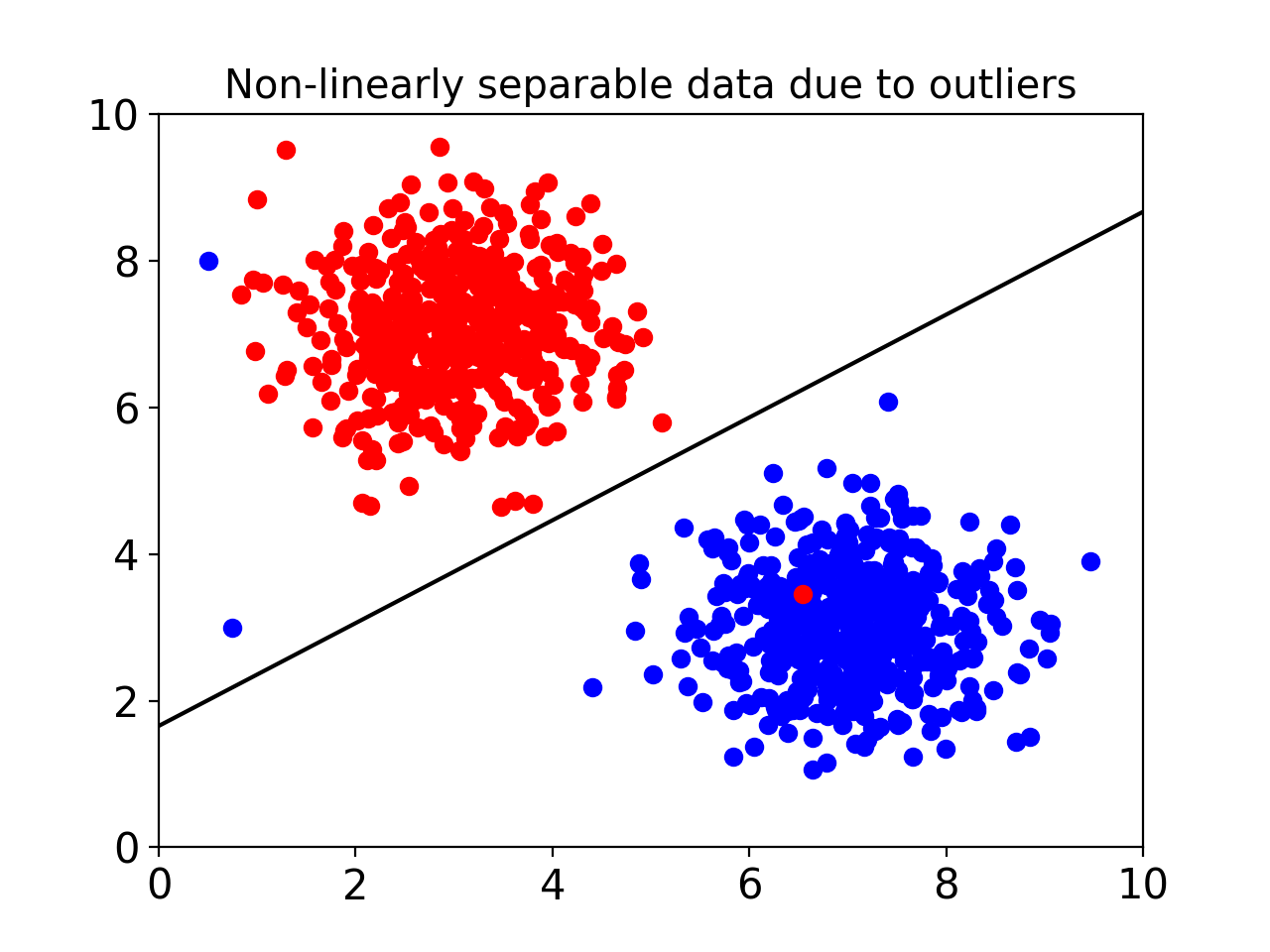
Up to this point, we have been using data sets that are linearly separable. This means that it’s possible to draw a straight decision boundary between the two classes. However, what would happen if an SVM came along a dataset that wasn’t linearly separable?



It's impossible to draw a straight line to separate the red points from the blue points!

Luckily, SVMs have a way of handling these data sets. Remember when we set kernel = 'linear' when creating our SVM? Kernels are the key to creating a decision boundary between data points that are not linearly separable.

Note that most machine learning models should allow for some error. For example, the image below shows data that isn’t linearly separable. However, it is not linearly separable due to a few outliers. We can still draw a straight line that, for the most part, separates the two classes. You shouldn't need to create a non-linear decision boundary just to fit some outliers. Drawing a line that correctly separates every point would be drastically overfitting the model to the data.



**1.**

Let's take a look at the power of kernels. We've created a dataset that isn't linearly separable and split it into a training set and a validation set.

Create an SVC named classifier with a 'linear' kernel.

**2.**

Call the .fit() method using training\_data and training\_labels as parameters.

**3.**

Let's see how accurate our classifier is using a linear kernel.

Call classifier's .score() function using validation\_data and validation\_labels as parameters. Print the results.

This will print the average accuracy of the model.

**4.**

That's pretty bad! The classifier is getting it right less than 50% of the time! Change 'linear' to 'poly' and add the parameter degree = 2. Run the program again and see what happens to the score.

import codecademylib3\_seaborn

from sklearn.svm import SVC

from graph import points, labels

from sklearn.model\_selection import train\_test\_split

training\_data, validation\_data, training\_labels, validation\_labels = train\_test\_split(points, labels, train\_size = 0.8, test\_size = 0.2, random\_state = 100)

classifier = SVC(kernel='poly', degree=2)

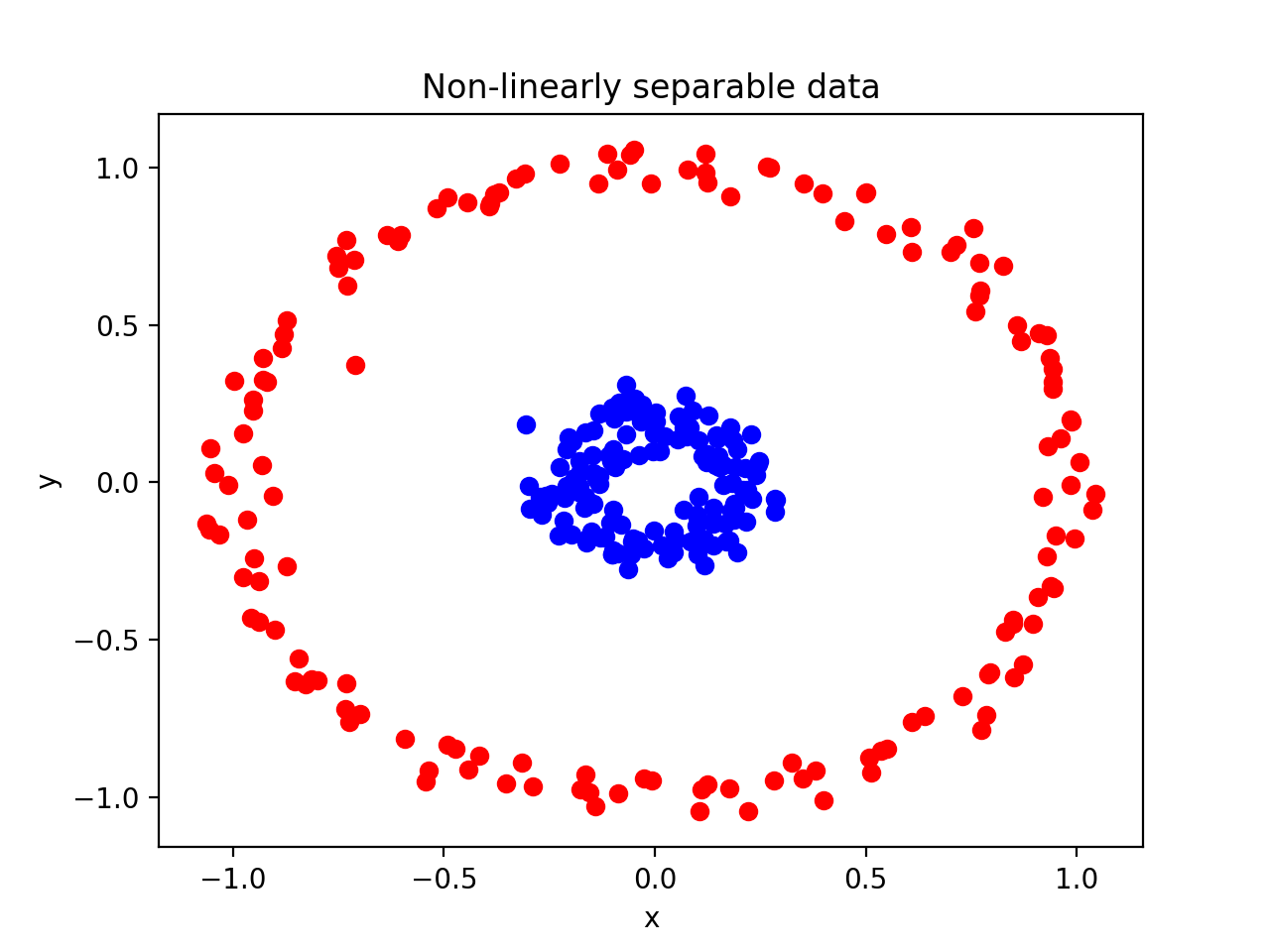
classifier.fit(training\_data, training\_labels)

print(classifier.score(validation\_data, validation\_labels)) #0.4333 linear, 1.0 for poly deg=2

# Polynomial Kernel

That kernel seems pretty magical. It is able to correctly classify every point! Let's take a deeper look at what it was really doing.

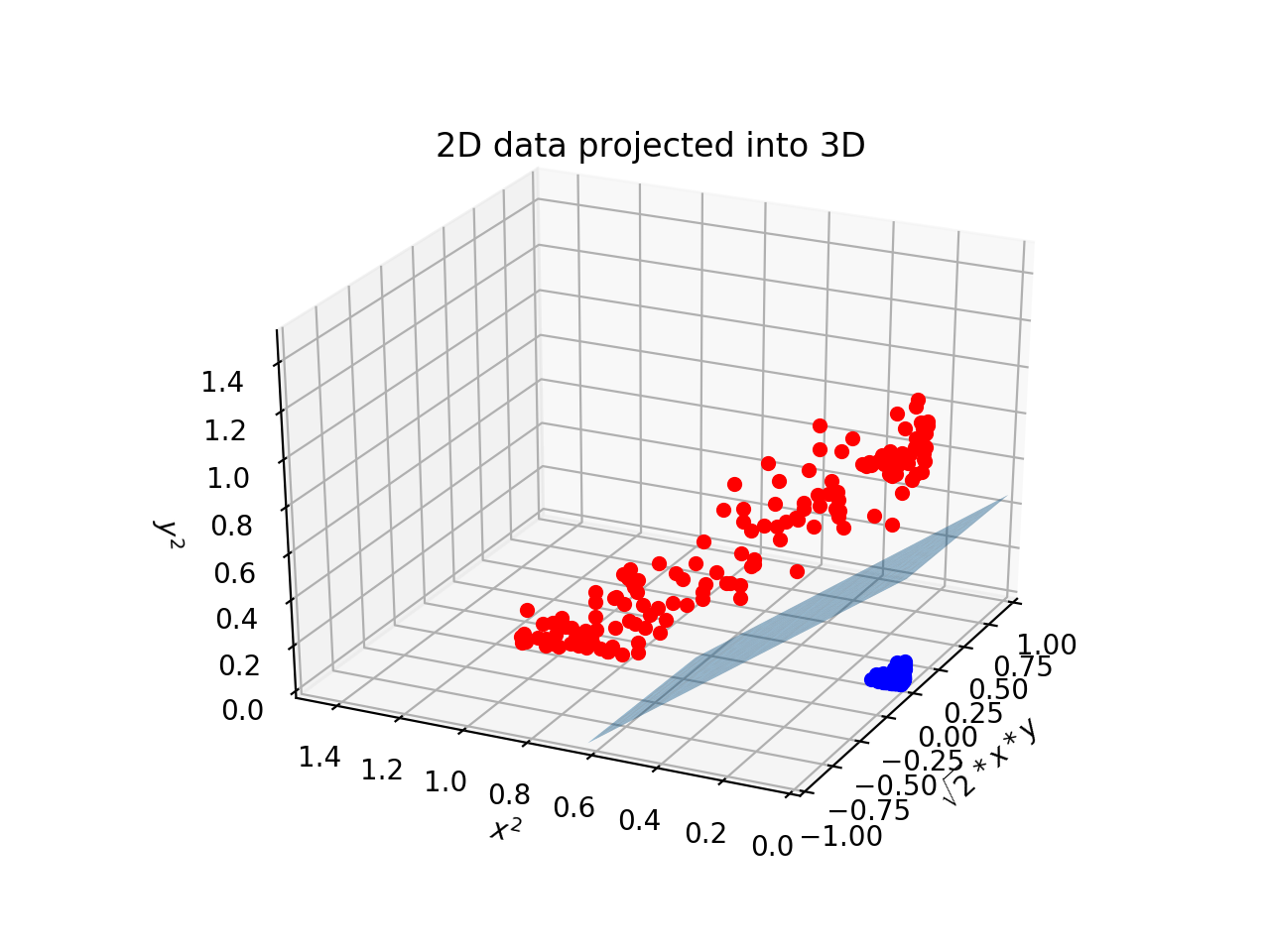
We start with a group of non-linearly separable points that looked like this:



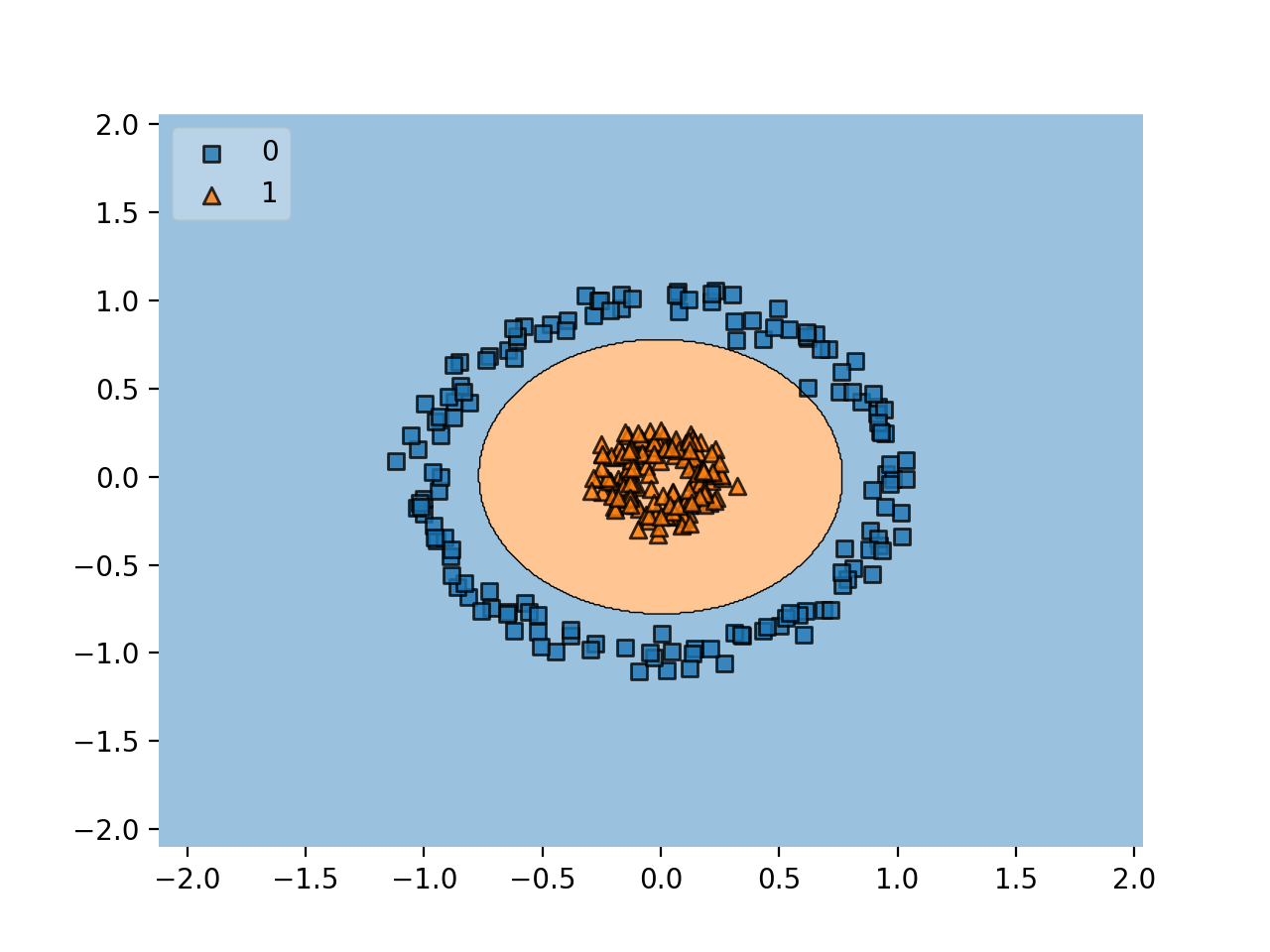
The kernel transforms the data in a clever way to make it linearly separable. We used a polynomial kernel which transforms every point in the following way:

The kernel has added a new dimension to each point! For example, the kernel transforms the point [1, 2] like this:

If we plot these new three dimensional points, we get the following graph:



Look at that! All of the blue points have scooted away from the red ones. By projecting the data into a higher dimension, the two classes are now linearly separable by a plane. We could visualize what this plane would look like in two dimensions to get the following decision boundary.



**1.**

In this exercise, we will be using a non-linearly separable dataset similar to the concentric circles above.

Rather than using a polynomial kernel, we're going to stick with a linear kernel and do the transformation ourselves. The SVM running a linear kernel on the transformed points should perform identically to the SVM running a polynomial kernel on the original points.

To begin, at the bottom of your code, print training\_data[0] to see the first data point. You will also see the accuracy of the SVM when the data is not projected into 3 dimensions.

**2.**

Let's transform the data into three dimensions! Begin by creating two empty lists called new\_training and new\_validation.

**3.**

Loop through every point in training\_data. For every point, append a list to new\_training. The list should contain three numbers:

* The square root of 2 times point[0] times point[1].
* point[0] squared.
* point[1] squared.

Remember, to square a number in Python do number \*\* 2. To take the square root, do number \*\* 0.5.

**4.**

Do the same for every point in validation\_data. For every point in validation\_data, add the new list to new\_validation.

**5.**

Retrain classifier by calling the .fit() method using new\_training and training\_labels as parameters.

**6.**

Finally, run classifier's .score() method using new\_validation and validation\_labels as parameters. Print the results. How did the SVM do when the data was projected to three dimensions?

# Radial Bias Function Kernel

The most commonly used kernel in SVMs is a radial bias function (**rbf**) kernel. This is the default kernel used in scikit-learn's SVC object. If you don't specifically set the kernel to "linear", "poly" the SVC object will use an rbf kernel. If you want to be explicit, you can set kernel = "rbf", although that is redundant.

It is very tricky to visualize how an rbf kernel "transforms" the data. The polynomial kernel we used transformed two-dimensional points into three-dimensional points. An rbf kernel transforms two-dimensional points into points with an infinite number of dimensions!

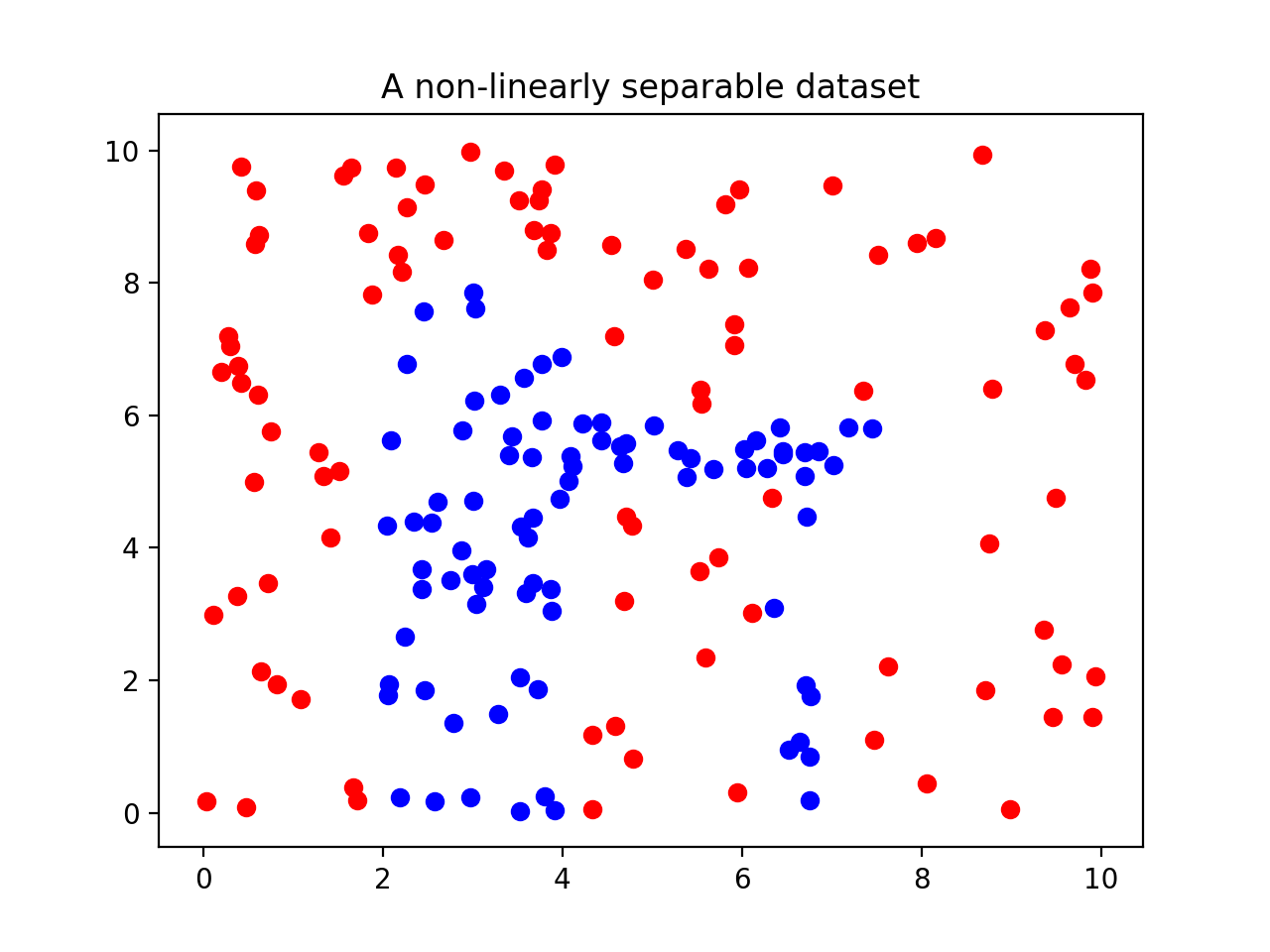
We won't get into how the kernel does this — it involves some fairly complicated linear algebra. However, it is important to know about the rbf kernel's gamma parameter.

classifier = SVC(kernel = "rbf", gamma = 0.5, C = 2)

gamma is similar to the C parameter. You can essentially tune the model to be more or less sensitive to the training data. A higher gamma, say 100, will put more importance on the training data and could result in overfitting. Conversely, A lower gamma like 0.01 makes the points in the training data less relevant and can result in underfitting.

**1.**

We're going to be using a rbf kernel to draw a decision boundary for the following points:



We've imported the data for you and split it into training\_data, validation\_data, training\_labels, and validation\_labels.

Begin by creating an SVC named classifier with an "rbf" kernel. Set the kernel's gamma equal to 1.

**2.**

Next, train the model using the .fit() method using training\_data and training\_labels as parameters.

**3.**

Let's test the classifier's accuracy when its gamma is 1. Print the result of the .score() function using validation\_data and validation\_labels as parameters.

**4.**

Let's see what happens if we increase gamma. Change gamma to 10. What happens to the accuracy of our model?

**5.**

The accuracy went down. We overfit our model. Change gamma to 0.1. What happens to the accuracy of our model this time?

from data import points, labels

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

training\_data, validation\_data, training\_labels, validation\_labels = train\_test\_split(points, labels, train\_size = 0.8, test\_size = 0.2, random\_state = 100)

classifier = SVC(kernel='rbf', gamma=0.1)

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

print(classifier.score(validation\_data, validation\_labels))

# score of 0.8888888888888888 for gamma = 1

# score of 0.8333333333333334 for gamma = 10

# score of 0.7777777777777778 for gamma = 0.1

**Review**

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

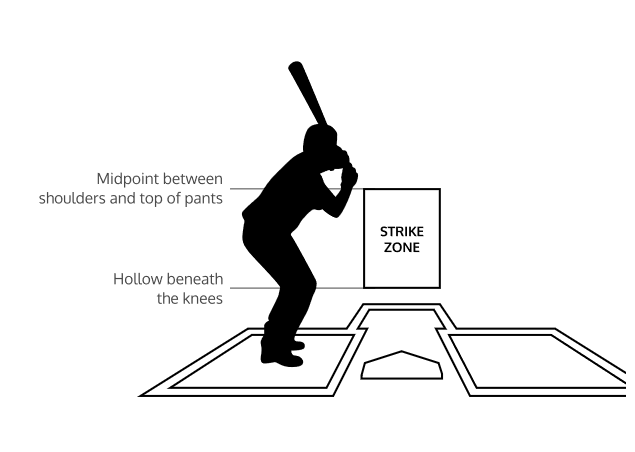
* SVMs are supervised machine learning models used for classification.
* An SVM uses support vectors to define a decision boundary. Classifications are made by comparing unlabeled points to that decision boundary.
* Support vectors are the points of each class closest to the decision boundary. The distance between the support vectors and the decision boundary is called the margin.
* SVMs attempt to create the largest margin possible while staying within an acceptable amount of error.
* The C parameter controls how much error is allowed. A large C allows for little error and creates a hard margin. A small C allows for more error and creates a soft margin.
* SVMs use kernels to classify points that aren't linearly separable.
* Kernels transform points into higher dimensional space. A polynomial kernel transforms points into three dimensions while an rbf kernel transforms points into infinite dimensions.
* An rbf kernel has a gamma parameter. If gamma is large, the training data is more relevant, and as a result overfitting can occur.

# Project

# Sports Vector Machine

Support Vector Machines are powerful machine learning models that can make complex decision boundaries. An SVM’s decision boundary can twist and curve to accommodate the training data.

In this project, we will use an SVM trained using a baseball dataset to find the decision boundary of the strike zone.



The strike zone can be thought of as a decision boundary that determines whether or not a pitch is a strike or a ball. There is a strict definition of the strike zone: however, in practice, it will vary depending on the umpire or the player at bat.

Let's use our knowledge of SVMs to find the real strike zone of several baseball players.

If you get stuck during this project, check out the **project walkthrough video** which can be found at the bottom of the page after the final step of the project.