# LAB 7

November 23, 2020

## 1 Lab 7

#### 1.0.1 Xochitl Alvarado, Hideki Nakazawa, Tiffany Ngyuen, Nia Sanchez

#### 1.0.2 BME 455

### 1.1 Supervise Learning - Classification Using Support Vector Machines

**What is a Suppor Vector Machine?** In the previous chapter, you saw how to perform classification using logistics regression. In this chaper, you will learn another supervised machien learning algorithm that is also very popular among data scientists – Support Vector Machines (SVM). Like logistics regression, SVM is also a classfication algorithm. The main idea behind SVM is to draw a line between two or more classes in the best possible manner.

Once the line is drawn to separate the classes, you can then use it to predict future data. For example, given the snout length and ear geometry of a new unknown animal, you can now use the dividing line as a classifier to predict if the animal is a dog or a cat.

In this chapter you will learn how SVM works and the various techniques you can use to adapt SVM for solving nonlinearly-separable datasets.

**Maximum Separability** The goal is to find the largest possible width for the margin that can separate the two groups. Each of the two margins touches the closest poin(s) to each group of points. We use the term "hyperplane" instead of "line" because in SVM we typically deal with more than two dimensions, and using the word "hyperplane" more accurately convest the idea of a plane in multidimensional space.

**Support Vectors** A key term in SVM is support vector. Support vectors are the points that lie on the two margins. Using the example from the previous section, Figure 8.5 show the two support vectors lying on the two margins. In this case we say there are two support vectors–one for each class.

**Formula for the Hyperplane** With the series of points, the next question would be to find the formula for the hyperplane, together with the two margins. The formula for the hyperplane (g) is given as:

$$g(x) = W_0 x_1 + W_1 x_2 + b$$

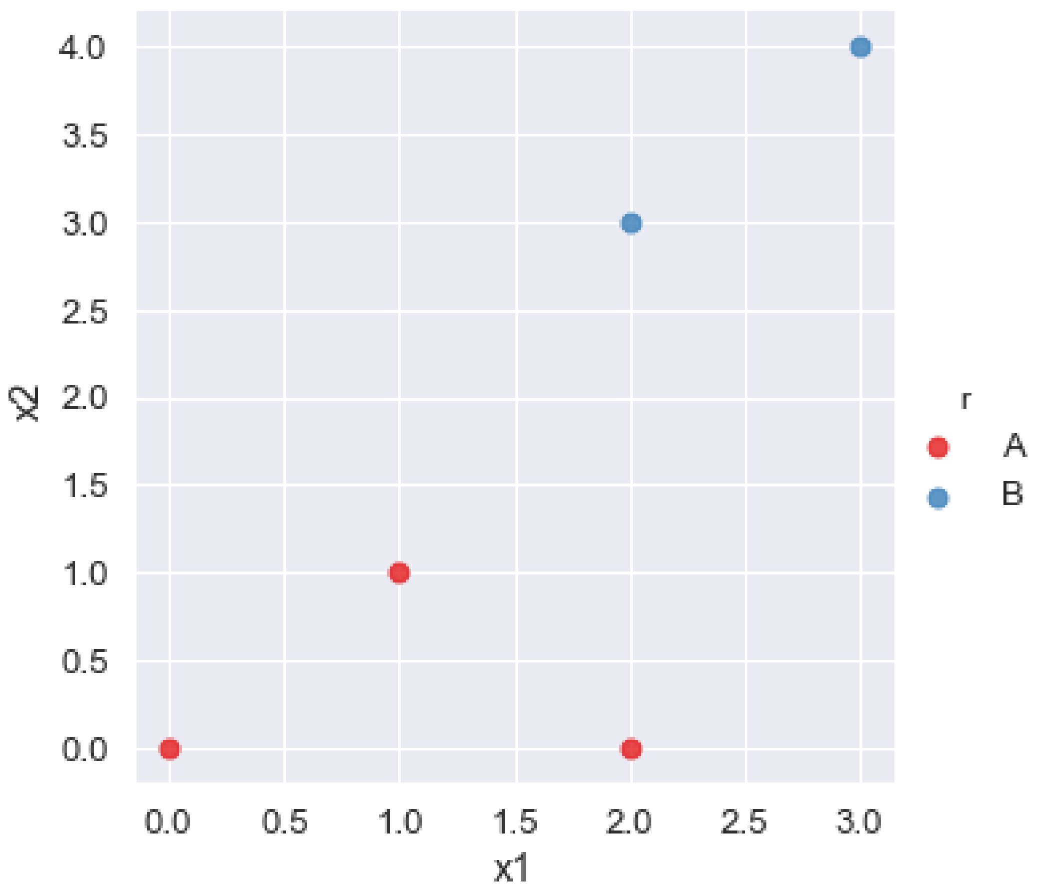
where $x_1$ and $x_2$ are the inputs, $W_0$ and $W_1$ are the weight vectors, and $b$ is the bias. If the value of $g$ is $>= 1$, then the point specified in Class 1, and if the value of $g$ is $<= -1$, then the point specified is in Class 2. As mentioned, the goal of SVM is to find the widest margins that divide the classes, and the total margin (2d) is defined by:

$$2/||w||$$

where $||w||$ is the normalized weight vectors ($W_0 and W_1$. Using the training set, the goal is to minimize the value of $||w||$ so that you can get the maximum separability between the classes. Once this is done, you will be able to get the values of $W_0, W_1$, and $b$. Finding the margin is a Constrained Optimization problem, which can be solved using the Larange Multipliers technique. It is beyond the scope of this book to discuss how to find the margin based on the dataset, but suffice it to say that we will make use of the Scikit-learn library to find them.

**Using Scikit-learn for SVM** Now lets work on an example to see how SVm works and how to implement it using Scikit-learn. FOr this example, we hava file named svm.csv containing the following data:

[54]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import seaborn as sns; sns.set(font_scale=1.2)
import matplotlib.pyplot as plt

data=pd.read_csv('svm.csv')
sns.lmplot('x1','x2',
           data=data,
           hue='r',
           palette='Set1',
           fit_reg=False,
           scatter_kws={"s":50});
```



Using the data points we have previously loaded, now lets use Scikit-learns svm modules SVC class to help us derive the value for the various variables that we need tocompute otherwise. The following code snippet uses linear kernel to solve the problem. The linear kernel assumes that the dataset can be separated linearly.

[55]:
```
from sklearn import svm
#---Converting the Columns as Matrices---#
points=data[['x1','x2']].values
result=data['r']

clf=svm.SVC(kernel='linear')
clf.fit(points,result)

print('Vector of weights (w) = ',clf.coef_[0])
print('b=',clf.intercept_[0])
print('Indices of support vectors = ',clf.support_)
print('Number of support vectors for each class =',clf.n_support_)
print('Coefficients of the support vector in the decision function =',np.abs(clf.
 ↪dual_coef_))
```

```
Vector of weights (w) =  [0.4 0.8]
b= -2.2
Indices of support vectors =  [1 2]
Number of support vectors for each class = [1 1]
Coefficients of the support vector in the decision function = [[0.4 0.4]]
```

svm.LinearSVC : Linear Support Vector Classification svm.LinearSVR: Linear Support Vector Regression svm.NuSVC: Nu-Support Vector Classification svn.NuSVR: Nu-Support Vector Regression svm.OneClassSVM: Unsupervised Outlier Detection svm.SVC: C-Support Vector Classification svm.SVR: Epsilon Support Vector Regression

As you can see the vector of weights has been found to be [0.4 0.8], meaning that $ W_0 $ is now 0.4 and $W_1$ is now 0.8. The value is b is -2.2, and there are two support vectors. The index of the support vectors is 1 and 2, meaning that the points are the ones in bold.

**Plotting the Hyperplane and the Margins** With the values of the variables all obtained, it is now time to plot the hyperplane and its two accompanying margins. Do you remember the formula for the hyperplane? It is as follows:

$$g(x) = W_0 x_1 + W_1 x_2 + b$$

to plot the hyperplane, set $ W_0 x_'1 + W_1 x_2 + b $ to 0 like this:

$$W_0 x1 + W_1 x2 + b = 0$$

In order to plot the hyperplane (which is a straight line in this case), we need two points: one on the x-axis and one on the y-axis. Using the preceding formula, when $x_1 = 0$, we can solve for $x_2$ as follows:

$$W_0(0) + W_1 x_2 + b = 0$$

$$W_1 x_2 = -b$$

$$x_2 = -b/W_1$$

When $x_2 = 0$, we can solve for $x_1$ as follows:

$$W_0 x_1 + W_1(0) + b = 0$$

$$W_0 x_1 = -b$$

$$x_1 = -b/W_0$$

The point $(0, -b/W_1)$ is the y-intercept of the straight line. Once the points on each axis are found, you can now calculate the slope as follows:

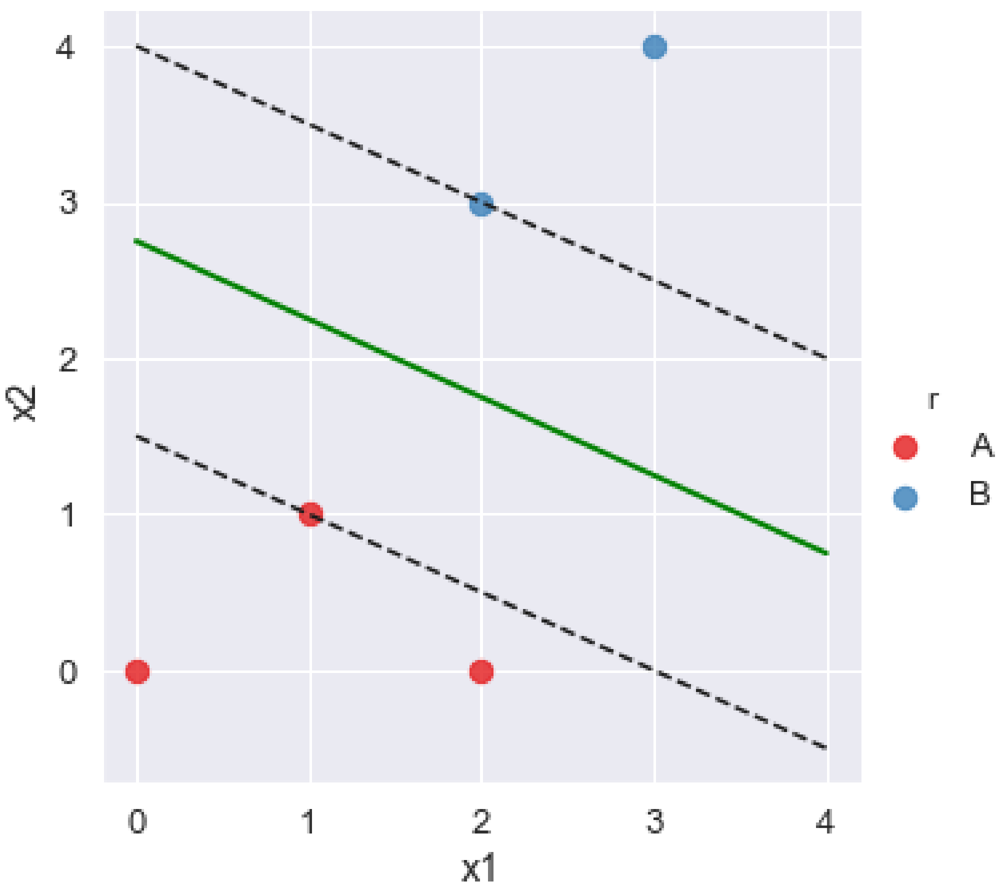
$$Slope = (-b/W_1)/(b/W_0)$$

$$Slope = -(W_0/W_1)$$

With the slope and y-intercept of the line found, you can now go ahead and plot the hyperplane. The following code snippet does just that:

[56]:
```
#---w is the vector of weights---#
w=clf.coef_[0]
#---find the slope of the hyperplane---#
slope=-w[0]/w[1]

b=clf.intercept_[0]
#---Find the coordinates for the hyperplane---#
xx=np.linspace(0,4)
yy=slope*xx - (b/w[1])

#---plot the margins---#
s=clf.support_vectors_[0] #---first support vector---#
yy_down=slope*xx+(s[1]-slope*s[0])

s=clf.support_vectors_[-1] #---last support vector---#
yy_up = slope*xx + (s[1]-slope*s[0])

#---plot the points---#
sns.lmplot('x1','x2',data=data, hue='r',palette='Set1',fit_reg=False,␣
 ↪scatter_kws={"s":70})
#---plot the hyperplane---#
plt.plot(xx,yy,linewidth=2,color='green');

#---plot the 2 margins---#
plt.plot(xx,yy_down,'k--')
plt.plot(xx,yy_up,'k--')
```

[56]:
```
[<matplotlib.lines.Line2D at 0x244885ae4f0>]
```



**Making Predictions** Remember, the goal of SVM is to separate the points into two or more classes, so that you can use it to predict the classes of future points. Having trained your model using SVM, you can now perform some predictions using the model. The following code snippet uses the model that you have trained to perform some predictions:

[57]:
```
print(clf.predict([[3,3]]) [0]) # 'B'
print(clf.predict([[4,0]])[0]) # 'A'
print(clf.predict([[2,2]])[0]) # 'B'
print(clf.predict([[1,2]])[0]) # 'A'
```

```
B
A
B
A
```

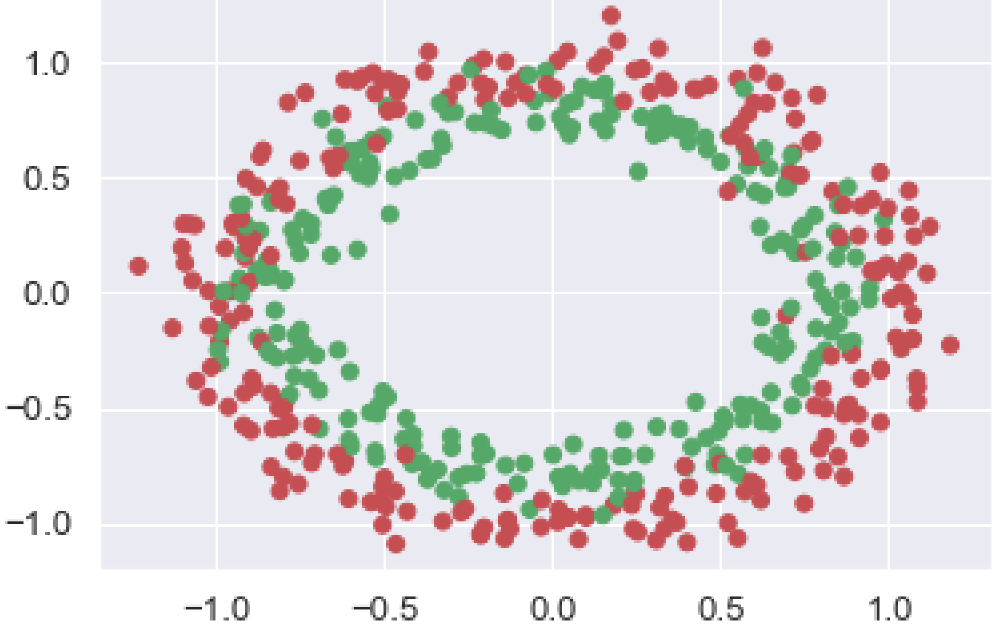
**Kernel Trick** Sometimes, the points in a dataset are not always linearly separable. You can see that it is not possible to draw a straight line to separate the two sets of points. With some manipulation, however, you can make this set of points linearly separable. This technique is known as the kernel trick. The kernel trick is a technique in machine learning that transforms data into a higher dimension space so that, after the transformation, it has a clear dividing margin between classes of data.

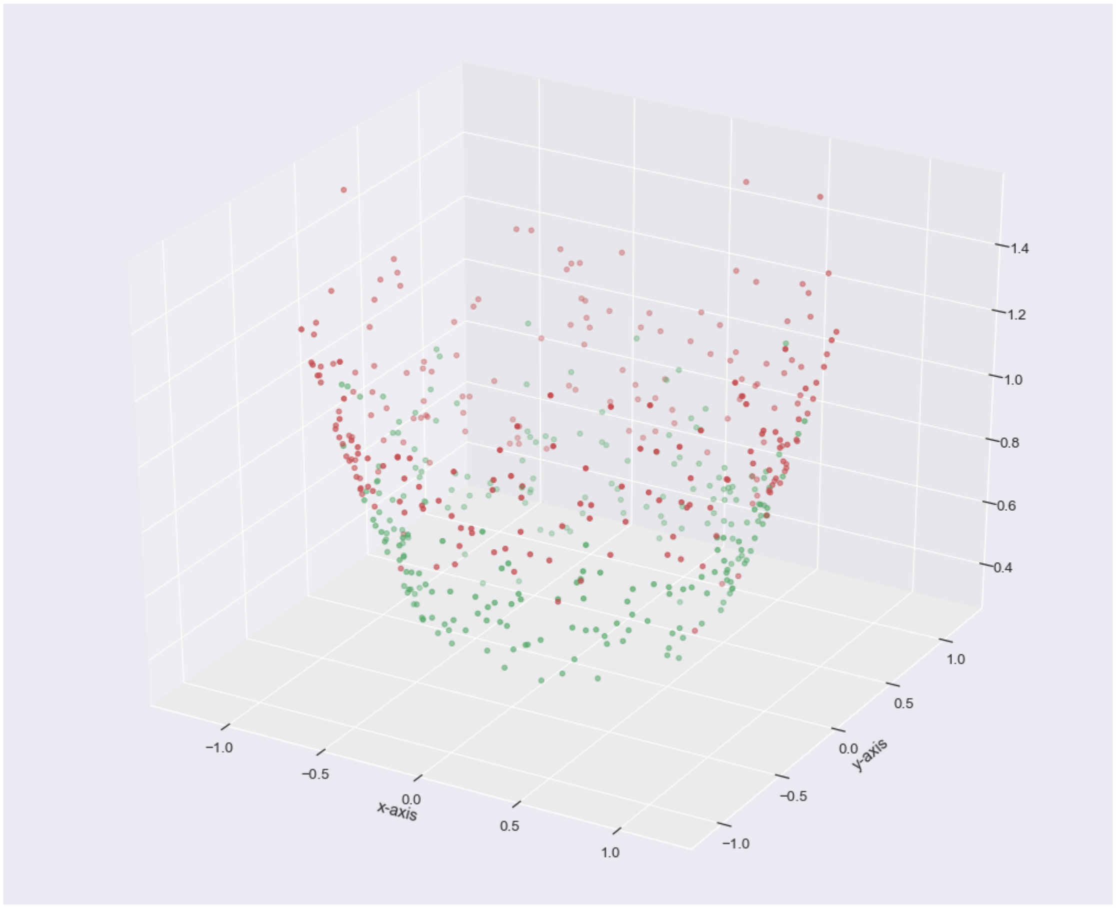
**Adding a Third Dimension** To do so, we can add a third dimension, say the z-axis, and define z to be:

$$z = x^2 + y^2$$

Once we plot the points using a 3D Chart, the points are now linearly separable. It is difficult to visualize this unless you plot the points out. The following code snippet does just that:

[58]:
```
%matplotlib inline

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_circles

#---X is features and c is the class labels---#
X, c= make_circles(n_samples=500,noise=0.09)

rgb=np.array(['r','g'])
plt.scatter(X[:,0],X[:,1],color=rgb[c])
plt.show()

fig=plt.figure(figsize=(18,15))
ax=fig.add_subplot(111,projection='3d')
z=X[:,0]**2 + X[:,1]**2
ax.scatter(X[:,0],X[:,1],z,color=rgb[c])
plt.xlabel("x-axis")
plt.ylabel("y-axis")
plt.show()
```


1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


1.4
1.2
1.0
0.8
0.6
0.4
1.0
0.5
0.0
−0.5
−1.0
y-axis
−1.0
−0.5
0.0
0.5
1.0
x-axis

We first create two sets of random points (a total of 500 points) distributed in circular fashion using the make_circles() function. We then plot them out on a 2D chart (as what was shown in Figure 8.11). We then add the third axis, the z-axis, and plot the chart in 3D.

**Plotting the 3D Hyperplane** With the points plotted in a 3D chart, lets now train the model using the third dimension:

[59]:
```
#---combine X (x-axis, y-axis) and z into single ndarray---#
features=np.concatenate((X,z.reshape(-1,1)), axis=1)

#--use SVM for training---#
from sklearn import svm

clf=svm.SVC(kernel='linear')
clf.fit(features,c)
```

[59]:
```
SVC(kernel='linear')
```

First, we combined the three axes into a single ndarray using the np.concatenate() function. We then trained the model as usual. For a lienarly-separable set of points in two dimensions, the formula the hyperplane is as follows:

$$g(x) = W_0X1 + W_1X2 + W_2X3 + b$$

For the set of points now in three dimensions, the formula now becomes the following:

$$g(x) = W_0X1 + W_1X2 + W_2X3 + b$$

In particular, $W_2$ is now represented by clf.coef_[0][2].

The next step is to draw the hyperplane in 3D. In order to do that, you need to find the value of x3, which can be derived, as shown.

This can be expressed in code as follows:

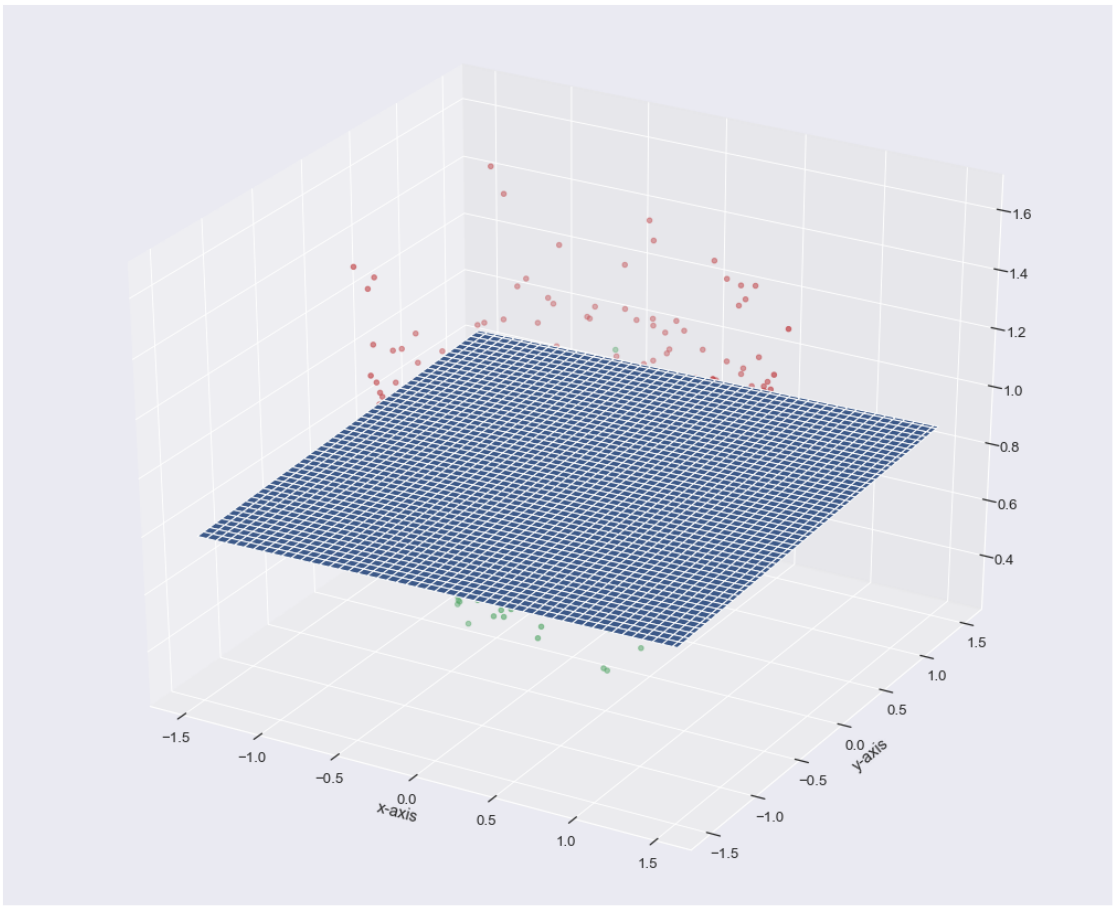
[60]:
```
x3=lambda x,y: (-clf.intercept_[0] - clf.coef_[0] [0] * x-clf.coef_[0][1] *y)/
 ↪clf.coef_[0][2]
```

To plot the hyperplane in 3D, use the plot_surface() function:

[61]:
```
tmp= np.linspace(-1.5,1.5,100)
x,y=np.meshgrid(tmp,tmp)
ax.plot_surface(x,y,x3(x,y))
plt.show()
```

The entire code snippet is as follows:

[62]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import make_circles

#---X is features and c is the class labels---#
X, c = make_circles(n_samples=500, noise=0.09)
z=X[:,0]**2 + X[:,1]**2

rgb=np.array(['r','g'])

fig=plt.figure(figsize=(18,15))
ax=fig.add_subplot(111,projection='3d')
ax.scatter(X[:,0],X[:,1],z,color=rgb[c])
plt.xlabel("x-axis")
plt.ylabel("y-axis")
# plt.show()

#---combine X (x-axis,y-axis) and z into single ndarray---#
features=np.concatenate((X,z.reshape(-1,1)),axis=1)

#---use SVM for training---#
from sklearn import svm

clf= svm.SVC(kernel ='linear')
clf.fit(features, c)
x3 = lambda x,y: (-clf.intercept_[0] - clf.coef_[0][0] * x-clf.coef_[0][1] * y)/
 ↪clf.coef_[0][2]
tmp=np.linspace(-1.5,1.5,100)
x,y=np.meshgrid(tmp,tmp)

ax.plot_surface(x,y,x3(x,y))
plt.show()
```



### 1.1.1 Types of Kernels

Up to this point, we discussed one type of SVM–linear SVM. As the name implies, linear SVM uses a straight line to separate the points. In the previous section, you also learned about the use of kernel tricks to separate two sets of data that are distributed in a circular fashion and then used linear SVM to separate them. Sometimes not all points can be separated linearly, nor can they be separated using the kernel tricks that you observed in the previous section. For this type of data, you need to "bend" the lines to separate them. In machine learning, kernels are functions that transform your data from nonlinear spaces to linear ones. To understand how kernels work, lets use the Iris dataset as an example. The following code snippet loads the Iris dataset and prints out the features, target, and target names:

[63]:
```
%matplotlib inline
import pandas as pd
import numpy as pd

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

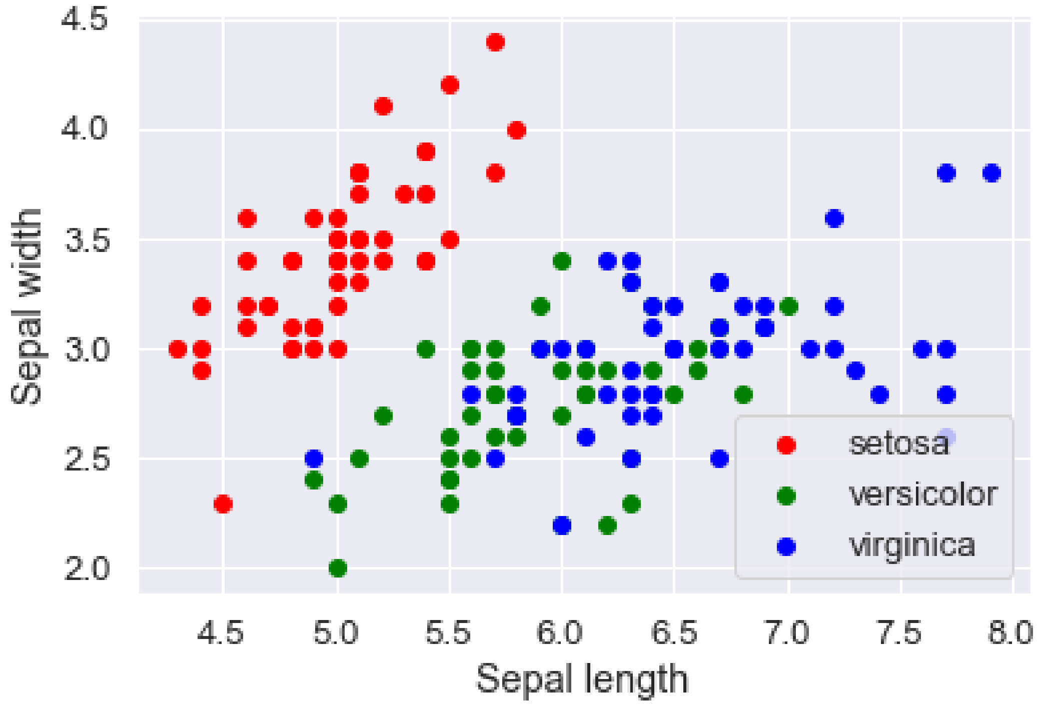
```
iris=datasets.load_iris()
print(iris.data[0:5])    #print first 5 rows
print(iris.feature_names) # ['sepal length (cm)', 'sepal width (cm)',
                           #'petal length (cm)', 'petal width (cm)']

print(iris.target[0:5])    # print first 5 rows
print(iris.target_names)   # ['setosa' 'versicolor' 'virginica']

X=iris.data[:, :2] #take the first 2 features
y=iris.target
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width
(cm)']
[0 0 0 0 0]
['setosa' 'versicolor' 'virginica']
```

We will plot the points using a scatter plot.

[64]:
```
#---plot the points---#
colors = ['red', 'green', 'blue']
for color, i, target in zip(colors, [0,1,2,], iris.target_names):
    plt.scatter(X[y==i,0], X[y==i,1], color=color, label=target)

plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.legend(loc='best',shadow=False, scatterpoints=1)
```

[64]:
```
<matplotlib.legend.Legend at 0x2448866a340>
```



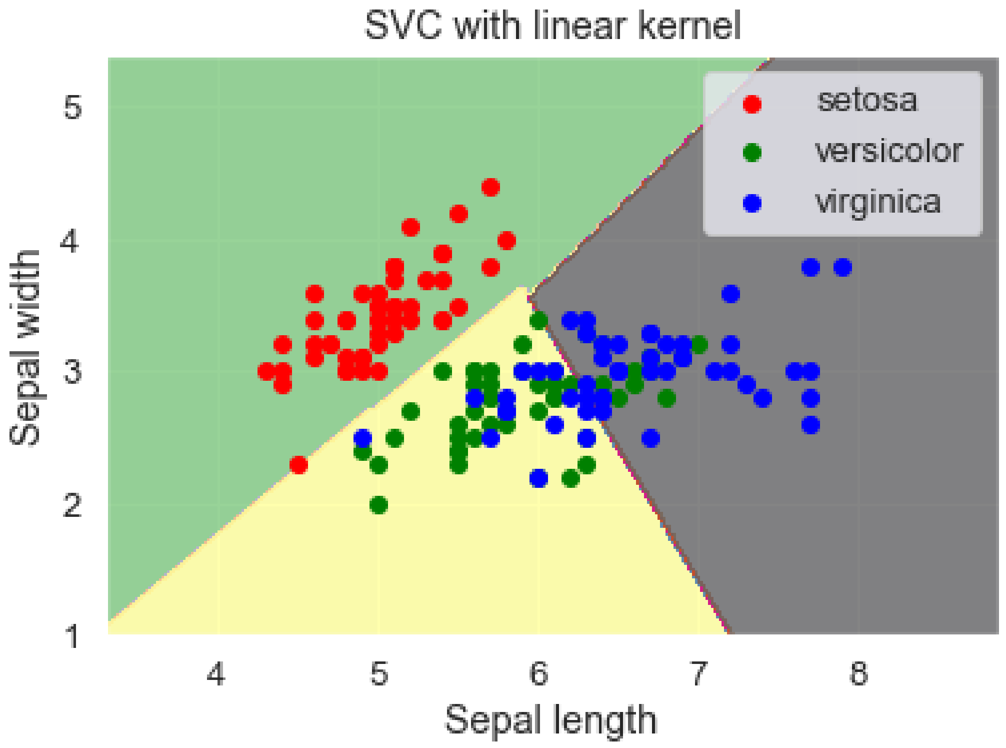
Next, we will use the SVC class with the linear kernel:

[65]:
```
C=1 # SVM regularization parameter
clf= svm.SVC(kernel='linear', C=C).fit(X,y)
title='SVC with linear kernel'
```

Instead of drawing lines to separate the three groups of Iris flowers, this time we will paint the groups in colors using the contourf() function:

[66]:
```
#---min and max for the first feature---#
x_min, x_max= X[:,0].min() - 1, X[:,0].max() +1

#---min and max for the second feature---#
y_min, y_max = X[:,1].min() -1, X[:,1].max()+1

#---step size in the mesh---#
h=(x_max/x_min)/100

#---make predictions for each of the points in xx,yy---#
xx,yy = np.meshgrid(np.arange(x_min,x_max,h),
                    np.arange(y_min,y_max,h))

Z=clf.predict(np.c_[xx.ravel(),yy.ravel()])
```

```
#---draw the result using a color plot---#
Z= Z.reshape(xx.shape)
plt.contourf(xx,yy,Z, cmap=plt.cm.Accent, alpha=0.8)

#---plot the training points---#
colors = ['red','green', 'blue']
for color, i, target in zip(colors, [0,1,2],iris.target_names):
    plt.scatter(X[y==i,0], X[y==i,1], color=color, label=target)

    plt.xlabel('Sepal length')
    plt.ylabel('Sepal width')
    plt.title(title)
    plt.legend(loc='best',shadow =False,scatterpoints=1)
```



Once the training is done, we will perform some predictions:

[67]:
```
predictions = clf.predict(X)
print(np.unique(predictions, return_counts=True))
```

```
(array([0, 1, 2]), array([50, 53, 47], dtype=int64))
```

This means that after feeding the model with the Iris dataset, 50 are classified as "setosa", 53 are classified as "versicolor", and 47 are classified as "virginica"

### 1.1.2 C

In the previous section you saw the use of the C parameter:

```
[68]: C=1
      clf=svm.SVC(kernel='linear',C=C).fit(X,y)
```

C is known as the penalty parameter of the error term. It controls the tradeoff between the smooth decision boundary and classifying the training points correctly. For example, if the value of C is high, then the SVM algorithm will seek to ensure that all points are classified correctly. The downside to this is that it may result in a narrower margin.

In contrast, a lwer C will aim for the widest margin possible, but it will result in some points being classified incorrectly.

Note that when C is 1 or $10^10$, there isn't too much difference among the classification results. However, when C is small ($10^-10$), you can see that a number of points (belonging to "versicolor" and "virginica") are now misclassified as "setosa"

In short, a low C makes the decision surface smooth while trying to classify most points, while a high C tries to classify all of the points correctly.

### 1.1.3 Radia Basis Function (RBF) Kernel

Besides the linear kernel that we have seen so far, there are some commonly used nonlinear kernels:

Radial basis Function (RBF), also known as Gaussian Kernel Polynomial

The first, RBF gies value to each point based on its distance from the origin or a fixed center, commonly on a Euclidean space. Lets use the same example we used in the previous section, but this time modify the kernel to use rbf:

```
[69]: C=1
      clf = svm.SVC(kernel='rbf',gamma='auto',C=C).fit(X,y)
      title='SVC with RBF kernel'
```

### 1.1.4 Gamma

If you look at the code snippet carefully, you will discover a new parameter (gamma). Gamma defines how far the influence of a single training example reaches. A low Gamma value indicates that every point has a far reach, on the other hand a high Gamma means that the points closest to the decision boundary have a close reach. The higher the value of Gamma, the more it will try to fit the training dataset exactly, resulting in overfitting.

Note if Gamma is high (10), overfitting occurs. You can also see from this figure that the value of C controls the smoothness of the curve.

To summarize, C controls the smoothnesso fthe boundary and Gamma determines if the points are overfitted.

### 1.1.5 Polynomial Kernel

Another type of kernel is called the polynomial kernel. a Polynomial kernel of degree 1 is similar to that of the linear kernel. Higher=degree polynomical kernels afford a more flexible decision boundary. The following code snippet shows the Iris dataset trained using the polynomical kernel with degree 4:

[70]:
```
C=1 # SVM regularization parameter
clf= svm.SVC(kernel='poly', degree=4, C=C, gamma='auto').fit(X,y)
title='SVC with polynomical (degree 4) kernel'
```

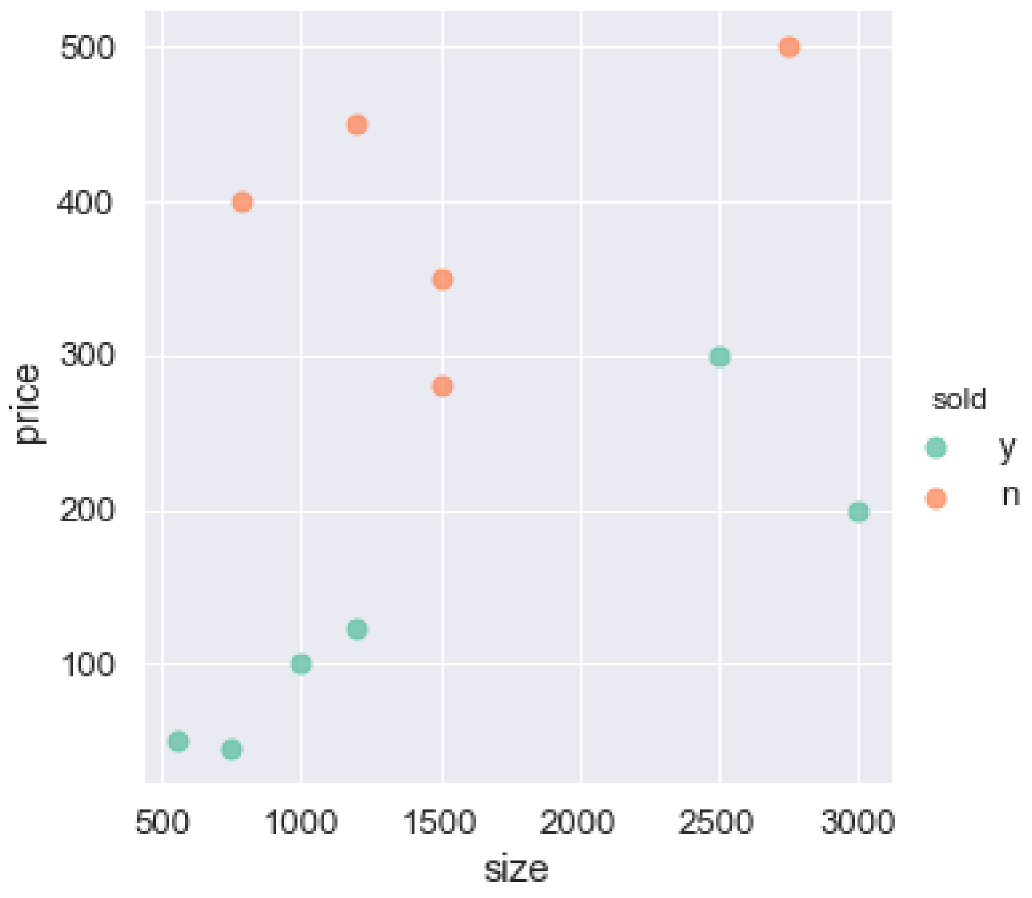
### 1.1.6 Using SVM for Real-Life Problems

We will end this chapter by applying SVM to a common problem in our daily lives. Consider the following dataset (saved in a file named house_sizes_prices_svm.csv) containing the size of houses and their asking prices (in thousands) for a particular area:

The third colum indicates if the house was sold. Using this dataset you want to know if a house with a specific asking price would be albe to sell.

First lets plot out the points:

[71]:
```
%matplotlib inline
import pandas as pd
import numpy as np
from sklearn import svm
import matplotlib.pyplot as plt
import seaborn as sns; sns.set(font_scale=1.2)

data= pd.read_csv('house_sizes_prices_svm.csv')

sns.lmplot('size','price',
           data=data,
           hue='sold',
           palette='Set2',
          fit_reg=False,
          scatter_kws={"s":50});
```



Visually, you can see that is a problem that can be solved with SVM's linear kernel:

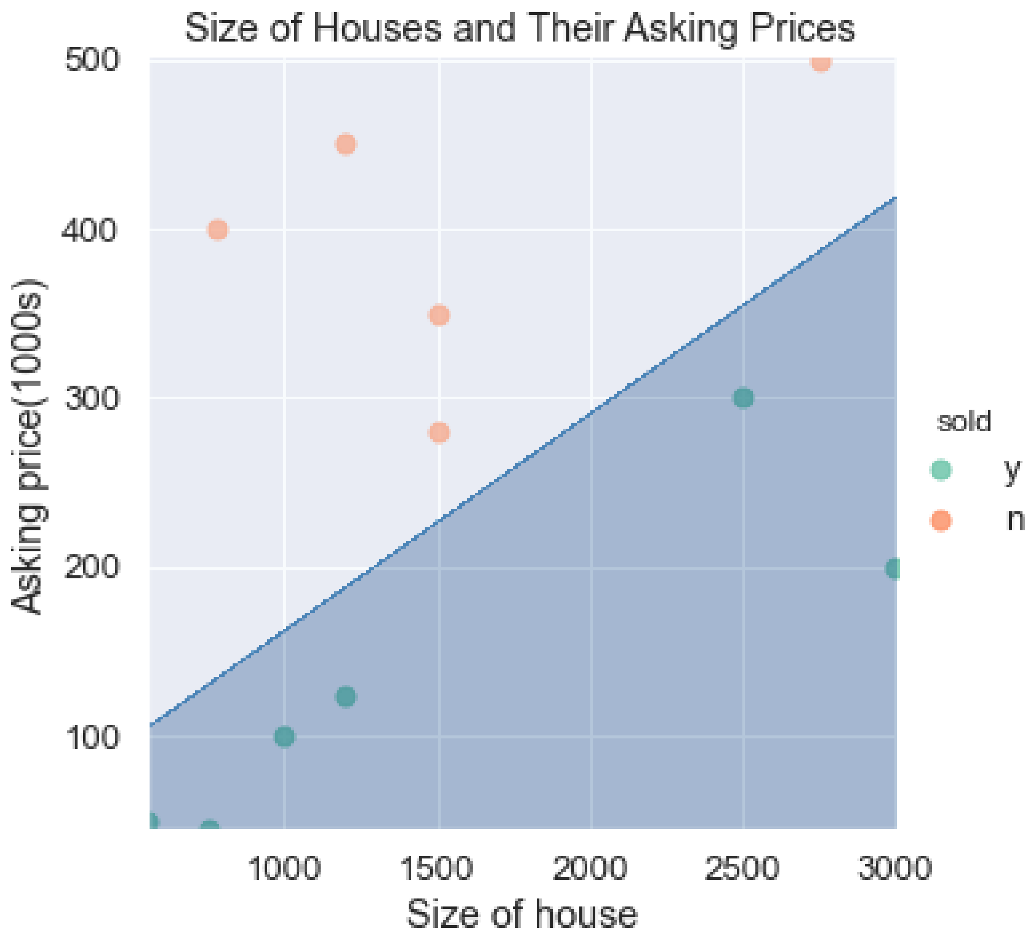
```
[72]: X=data[['size','price']].values
      y=np.where(data['sold']=='y',1,0) #--1 for Y and 0 for N--
      model=svm.SVC(kernel='linear').fit(X,y)
```

With the trained model, you can now perform predictions and paint the two classes:

```
[81]: sns.lmplot('size','price',
                 data=data,
                 hue='sold',
                 palette='Set2',
                fit_reg=False,
                scatter_kws={"s":50});

      #---min and max for the first feature---#
      x_min,x_max=X[:,0].min() - 1, X[:,0].max()+1

      #---min and max for the second feature---#
```

```
y_min,y_max =X[:,1].min()-1,X[:,1].max()+1

#---step size in the mesh---#
h=(x_max/x_min)/20

#---make predictions for each of the points in xx,yy---#
xx, yy=np.meshgrid(np.arange(x_min,x_max,h),
                   np.arange(y_min,y_max,h))

Z=model.predict(np.c_[xx.ravel(),yy.ravel()])

#---draw the result using a color plot---#
Z=Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Blues, alpha=0.3)

plt.xlabel('Size of house')
plt.ylabel('Asking price(1000s)')
plt.title("Size of Houses and Their Asking Prices")
```

```
[81]: Text(0.5, 1.0, 'Size of Houses and Their Asking Prices')
```



[82]:
```
def will_it_sell(size,price):
    if (model.predict([[size,price]]))==0:
        print('Will not sell!')
    else:
        print('Will sell!')

#---do some prediction---#
will_it_sell(2500,400) #Will not sell!
will_it_sell(2500,200) #Will sell!
```

```
Will not sell!
Will sell!
```

### 1.1.7 Summary

In this chapter, you learned about how Support Vector Machines help in classification problems. You learned about the formular for finding the hyperplane, as well as the two accompanying margins. Fortunately, Scikit-learn provides the classes needed for training models using SVm, and with the parameers returned you can plot the hyperplane and margins visually so that you can understand how SVM works. You also learned about the various kernels that you can apply to your SVM algorithms so that the dataset can be separated linearly.