Sardar Patel Institute of Technology, Mumbai
Department of Electronics and Telecommunication Engineering
B.E. Sem-VII (2022-2023)
OEIT6 - Data Analytics

Experiment: Support Vector Machine

Name: Pushkar Sutar **Roll No. 2019110060**

Objective : Understanding Support Vector Machine algorithm through building SVM algorithm in Python.

Introduction:

An SVM is a numeric classifier. That means that all of the features of the data must be numeric, not symbolic.

Furthermore, in this class, we'll assume that the SVM is a binary classifier: that is, it classifies points as one of
two classifications. We'll typically call the classifications "+" and " -".

A trained SVM is defined by two values:

- A normal vector w (also called the weight vector), which solely determines the shape and direction of the
  decision boundary.
- A scalar offset b, which solely determines the position of the decision boundary with respect to the origin. A trained SVM can then classify a point x by computing $w \cdot x + b$. If this value is positive, x is classified as +;
  otherwise, x is classified as -.

The decision boundary is coerced by support vectors, so called because these vectors (data points) support the
boundary: if any of these points are moved or eliminated, the decision boundary changes! All support vectors lie
on a gutter, which can be thought of as a line running parallel to the decision boundary. There are two gutters:
one gutter hosts positive support vectors, and the other, negative support vectors.

Note that, though a support vector is always on a gutter, it's not necessarily true that every data point on a gutter
is a support vector.

Code and Output:

We first import all the necessary modules

```
[1]  import pandas as pd
     import numpy as np
     import matplotlib as mpl
     import matplotlib.pyplot as plt
     from sklearn.metrics import confusion_matrix

     %matplotlib inline
```

```
[3]  from sklearn.svm import SVC
```

We'll define a function to draw a nice plot of an SVM.

```
def plot_svc(svc, X, y, h=0.02, pad=0.25):
    x_min, x_max = X[:, 0].min()-pad, X[:, 0].max()+pad
    y_min, y_max = X[:, 1].min()-pad, X[:, 1].max()+pad
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = svc.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.2)

    plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)
    # Support vectors indicated in plot by vertical lines
    sv = svc.support_vectors_
    plt.scatter(sv[:,0], sv[:,1], c='k', marker='x', s=100, linewidths='1')
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()
    print('Number of support vectors: ', svc.support_.size)
```

Building linear SVM-

The SVC() function can be used to fit a support vector classifier when the argument kernel="linear" is used. This function uses a slightly different formulation of the equations we saw in lecture to build the support vector classifier. The c argument allows us to specify the cost of a violation to the margin. When the c argument is small, then the margins will be wide and many support vectors will be on the margin or will violate the margin. When the c argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.

We can use the SVC() function to fit the support vector classifier for a given value of the cost parameter. Here we demonstrate the use of this function on a two-dimensional example so that

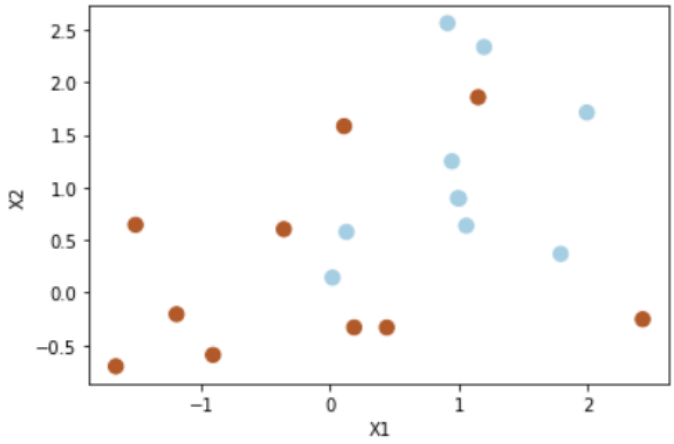
we can plot the resulting decision boundary. Let's start by generating a set of observations, which belong to two classes:

```
[4]  # Generating random data: 20 observations of 2 features and divide into two classes.
     np.random.seed(5)
     X = np.random.randn(20,2)
     y = np.repeat([1,-1], 10)

     X[y == -1] = X[y == -1]+1
```

Let's plot the data to see whether the classes are linearly separable:

```
[5]  plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)
     plt.xlabel('X1')
     plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



Linear or not?
We can clearly see that the classes are not linearly separable.

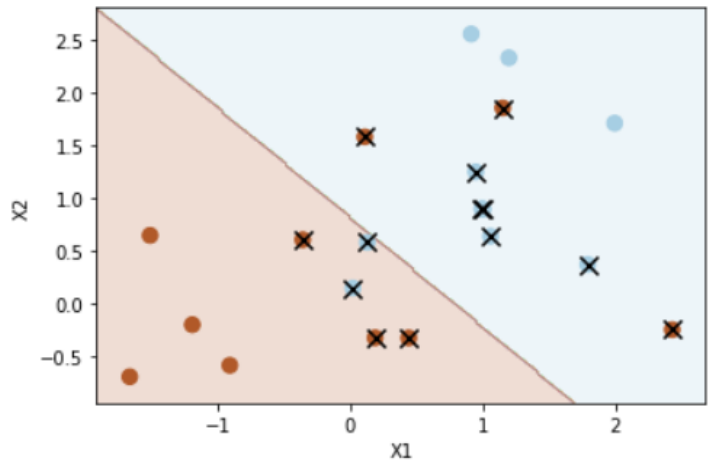
We now fit the support vector classifier.

```
[6]  svc = SVC(C=1, kernel='linear')
     svc.fit(X, y)
```

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

We can now plot the support vector classifier by calling the plot_svc() function on the output of the call to SVC(), as well as the data used in the call to SVC():

```
[7]  plot_svc(svc, X, y)
```




```
Number of support vectors:  13
```

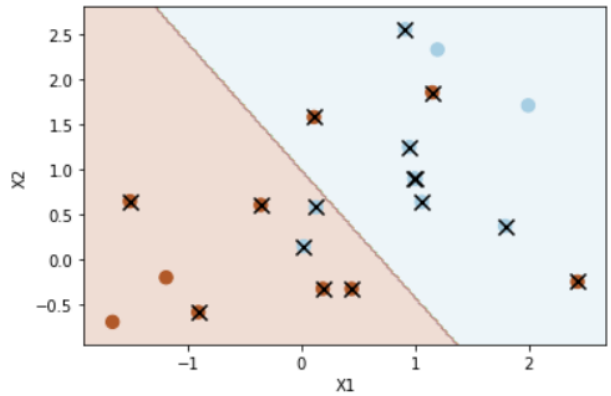
Number of support vectors?
The support vectors are plotted with crosses and the remaining observations are plotted as circles; we see here that there are 13 support vectors.

```
[8]  svc.support_
```

```
array([10, 11, 13, 14, 15, 16, 17,  0,  1,  2,  4,  6,  8], dtype=int32)
```

We now repeat using the smaller value of cost parameter.

```
[9]  svc2 = SVC(C=0.1, kernel='linear')
     svc2.fit(X, y)
     plot_svc(svc2, X, y)
```




```
Number of support vectors:  16
```

Now that a smaller value of the c parameter is being used, we obtain a larger number of support vectors, because the margin is now wider.
Number of support vectors?

There are a total 16 support vectors indicated by cross.

The sklearn.grid_search module includes a a function GridSearchCV() to perform cross-validation. In order to use this function, we pass in relevant information about the set of models that are under consideration. The following command indicates that we want perform 10-fold cross-validation to compare SVMs with a linear kernel, using a range of values of the cost parameter:

```
[10] from sklearn.model_selection import GridSearchCV

     # Select the optimal C parameter by cross-validation
     tuned_parameters = [{'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]}]
     clf = GridSearchCV(SVC(kernel='linear'), tuned_parameters, cv=10, scoring='accuracy')
     clf.fit(X, y)
```

```
GridSearchCV(cv=10, estimator=SVC(kernel='linear'),
             param_grid=[{'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]}],
             scoring='accuracy')
```

We can easily access the cross-validation errors for each of these models:

```
[11] clf.cv_results_
```

```
{'mean_fit_time': array([0.00068347, 0.00054305, 0.00053732, 0.00052991, 0.00056486,
        0.00062144, 0.00070651]),
 'std_fit_time': array([2.16736889e-04, 1.68077658e-05, 3.03715794e-05, 2.45814527e-05,
        4.12574103e-05, 8.26661716e-05, 5.03328726e-05]),
 'mean_score_time': array([0.00038304, 0.00031431, 0.00029902, 0.00030987, 0.00033858,
        0.00032151, 0.00030377]),
 'std_score_time': array([1.45551455e-04, 2.03358749e-05, 1.43387716e-05, 1.52558272e-05,
        9.05304228e-05, 2.89120745e-05, 9.33216125e-06]),
 'param_C': masked_array(data=[0.001, 0.01, 0.1, 1, 5, 10, 100],
              mask=[False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'C': 0.001},
  {'C': 0.01},
  {'C': 0.1},
  {'C': 1},
  {'C': 5},
  {'C': 10},
  {'C': 100}],
 'split0_test_score': array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]),
 'split1_test_score': array([0.5, 0.5, 0.5, 0. , 0. , 0. , 0. ]),
 'split2_test_score': array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]),
 'split3_test_score': array([1., 1., 1., 1., 1., 1., 1.]),
 'split4_test_score': array([1., 1., 1., 1., 1., 1., 1.]),
 'split5_test_score': array([1., 1., 1., 1., 1., 1., 1.]),
 'split6_test_score': array([1., 1., 1., 1., 1., 1., 1.]),
 'split7_test_score': array([1., 1., 1., 1., 1., 1., 1.]),
 'split8_test_score': array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5]),
 'split9_test_score': array([1., 1., 1., 1., 1., 1., 1.]),
 'mean_test_score': array([0.8 , 0.8 , 0.8 , 0.75, 0.75, 0.75, 0.75]),
 'std_test_score': array([0.24494897, 0.24494897, 0.24494897, 0.3354102 , 0.3354102 ,
        0.3354102 , 0.3354102 ]),
 'rank_test_score': array([1, 1, 1, 4, 4, 4, 4], dtype=int32)}
```

The GridSearchCV() function stores the best parameters obtained, which can be accessed as follows:

```
[12] clf.best_params_
```

```
{'C': 0.001}
```

c=0.001 is best according to GridSearchCV .

As usual, the predict() function can be used to predict the class label on a set of test observations, at any given value of the cost parameter. Let's generate a test data set:

```
[13] np.random.seed(1)
     X_test = np.random.randn(20,2)
     y_test = np.random.choice([-1,1], 20)
     X_test[y_test == 1] = X_test[y_test == 1]-1
```

```
[14] svc2 = SVC(C=0.001, kernel='linear')
     svc2.fit(X, y)
     y_pred = svc2.predict(X_test)
     pd.DataFrame(confusion_matrix(y_test, y_pred), index=svc2.classes_, columns=svc2.classes_)
```

	-1	1
-1	2	6
1	0	12

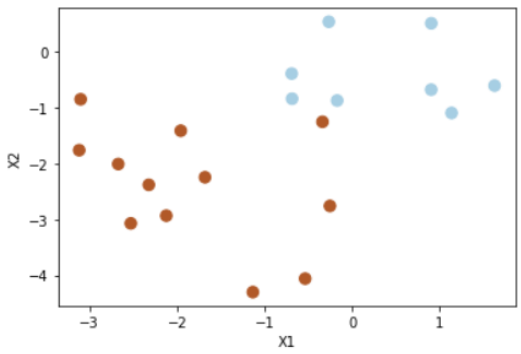
The value of True negative + True positive: 12+2=14.

Thus 14 observations are correctly classified.

With this value of c, 14 of the test observations are correctly classified. Now consider a situation in which the two classes are linearly separable. Then we can find a separating hyperplane using the svm() function. First we'll give our simulated data a little nudge so that they are linearly separable:

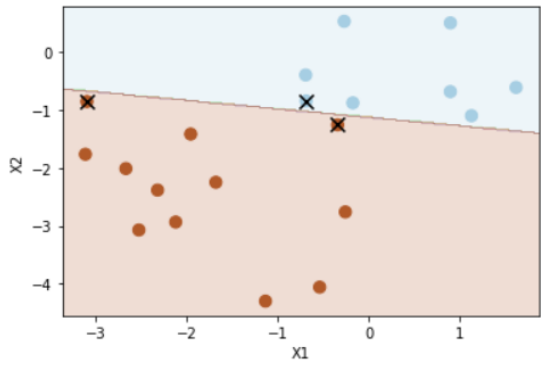
```
[15] X_test[y_test == 1] = X_test[y_test == 1] -1
     plt.scatter(X_test[:,0], X_test[:,1], s=70, c=y_test, cmap=mpl.cm.Paired)
     plt.xlabel('X1')
     plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



Now the observations are just barely linearly separable. We fit the support vector classifier and plot the resulting hyperplane, using a very large value of cost so that no observations are misclassified.

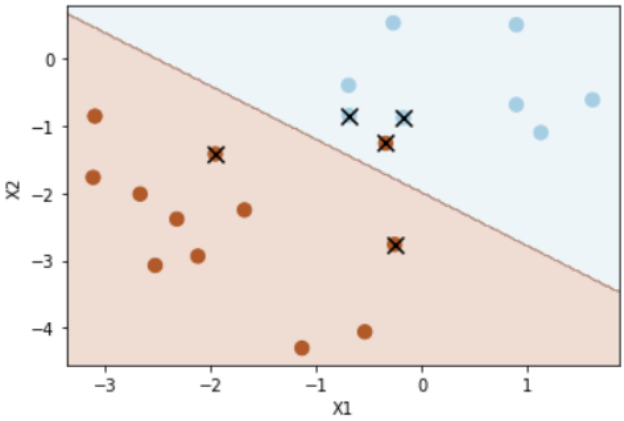
```
[16] svc3 = SVC(C=1e5, kernel='linear')
     svc3.fit(X_test, y_test)
     plot_svc(svc3, X_test, y_test)
```




```
Number of support vectors:  3
```

No training errors were made and only three support vectors were used. However, we can see from the figure that the margin is very narrow (because the observations that are not support vectors, indicated as circles, are very close to the decision boundary). It seems likely that this model will perform poorly on test data. Let's try a smaller value of cost :

```
[17] svc4 = SVC(C=1, kernel='linear')
     svc4.fit(X_test, y_test)
     plot_svc(svc4, X_test, y_test)
```




```
Number of support vectors:  5
```

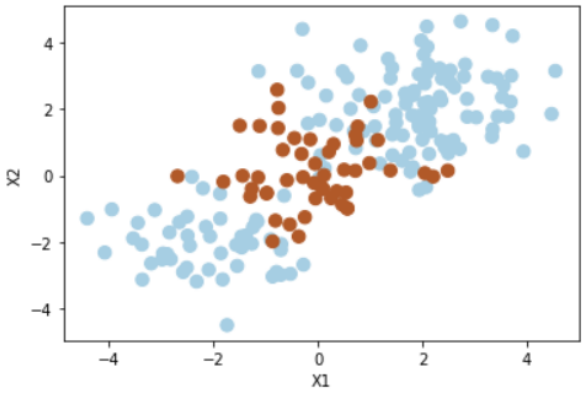
Using cost=1 , we misclassify a training observation, but we also obtain a much wider margin and make use of five support vectors. It seems likely that this model will perform better on test data than the model with cost=1e5 .

SVM using non-linear kernel -

Let's generate some data with a non-linear class boundary:

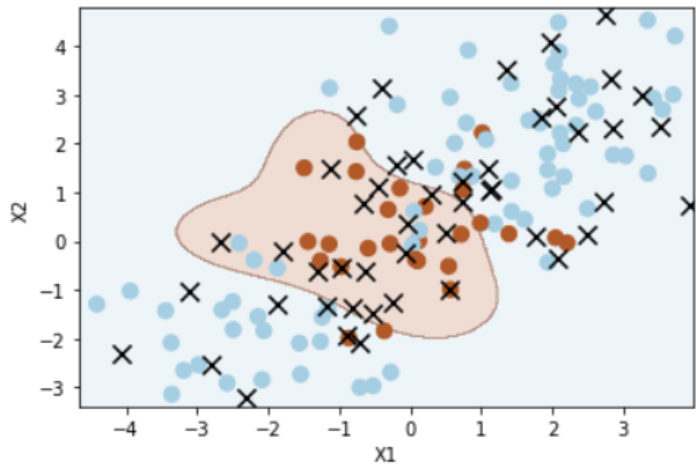
```
[18] from sklearn.model_selection import train_test_split

     np.random.seed(8)
     X = np.random.randn(200,2)
     X[:100] = X[:100] +2
     X[101:150] = X[101:150] -2
     y = np.concatenate([np.repeat(-1, 150), np.repeat(1,50)])

     X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.5, random_state=2)

     plt.scatter(X[:,0], X[:,1], s=70, c=y, cmap=mpl.cm.Paired)
     plt.xlabel('X1')
     plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



See how one class is kind of stuck in the middle of another class? This suggests that we might want to use a radial kernel in our SVM. Now let's fit the training data using the SVC() function with a radial kernel and γ=1 :

```
[19] svm = SVC(C=1.0, kernel='rbf', gamma=1)
     svm.fit(X_train, y_train)
     plot_svc(svm, X_test, y_test)
```

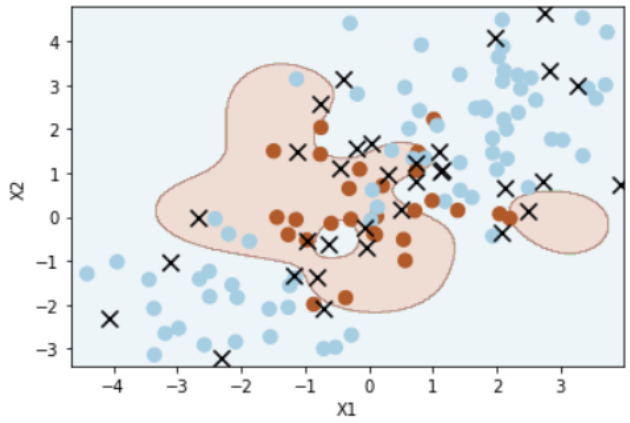



```
Number of support vectors:  51
```

Number of support vectors?
There are a total 51 support vectors.

We can see from the figure that there are a fair number of training errors in this SVM fit. If we increase the value of cost, we can reduce the number of training errors:

```
[20] # Increasing C parameter, allowing more flexibility
     svm2 = SVC(C=100, kernel='rbf', gamma=1.0)
     svm2.fit(X_train, y_train)
     plot_svc(svm2, X_test, y_test)
```




```
Number of support vectors:  36
```

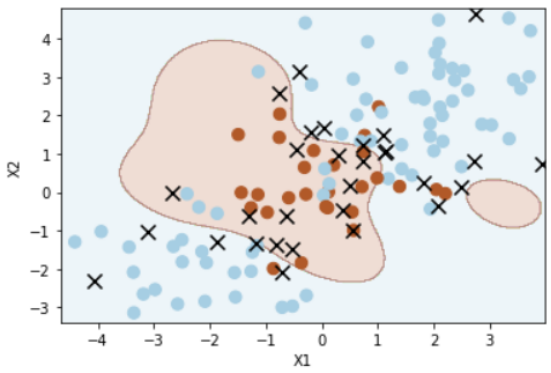
However, this comes at the price of a more irregular decision boundary that seems to be at risk of overfitting the data. We can perform cross-validation using GridSearchCV() to select the best choice of γ and cost for an SVM with a radial kernel:

```
[21] tuned_parameters = [{'C': [0.01, 0.1, 1, 10, 100],
                          'gamma': [0.5, 1,2,3,4]}]
     clf = GridSearchCV(SVC(kernel='rbf'), tuned_parameters, cv=10, scoring='accuracy')
     clf.fit(X_train, y_train)
     clf.best_params_
```

```
{'C': 10, 'gamma': 0.5}
```

Therefore, the best choice of parameters involves cost=1 and gamma=0.5 . We can plot the resulting fit using the plot_svc() function, and view the test set predictions for this model by applying the predict() function to the test data:

```
[22] plot_svc(clf.best_estimator_, X_test, y_test)
     print(confusion_matrix(y_test, clf.best_estimator_.predict(X_test)))
     print(clf.best_estimator_.score(X_test, y_test))
```




```
Number of support vectors:  32
[[66  7]
 [ 6 21]]
0.87
```

Number of support vectors?
There are 32 support vectors.

Test observations correctly classified by this SVM?
87% of the test observations have been correctly classified.

We now draw some ROC curves and analyse the results.
The auc() function from the sklearn.metrics package can be used to produce ROC curves.

```
[23] from sklearn.metrics import auc
     from sklearn.metrics import roc_curve
```

```
[24] # More constrained model
     svm3 = SVC(C=1, kernel='rbf', gamma=1)
     svm3.fit(X_train, y_train)
```

```
SVC(C=1, gamma=1)
```

```
[25] # More flexible model
     svm4 = SVC(C=1, kernel='rbf', gamma=50)
     svm4.fit(X_train, y_train)
```

```
SVC(C=1, gamma=50)
```

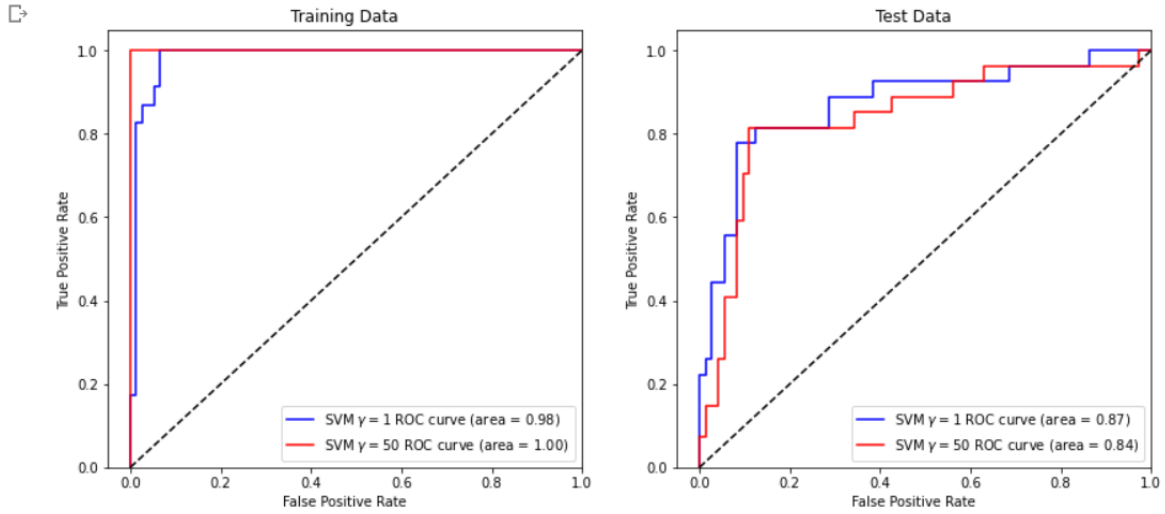
We have made two models one with gamma as 1 and other with gamma as 50.

In order to obtain the fitted values for a given SVM model fit, we use the .decision_function() method of the SVC:

```
[26] y_train_score3 = svm3.decision_function(X_train)
     y_train_score4 = svm4.decision_function(X_train)
```

Now we can produce the ROC plot to see how the models perform on both the training and the test data:

```
[27] y_train_score3 = svm3.decision_function(X_train)
     y_train_score4 = svm4.decision_function(X_train)

     false_pos_rate3, true_pos_rate3, _ = roc_curve(y_train, y_train_score3)
     roc_auc3 = auc(false_pos_rate3, true_pos_rate3)

     false_pos_rate4, true_pos_rate4, _ = roc_curve(y_train, y_train_score4)
     roc_auc4 = auc(false_pos_rate4, true_pos_rate4)

     fig, (ax1,ax2) = plt.subplots(1, 2, figsize=(14,6))
     ax1.plot(false_pos_rate3, true_pos_rate3, label='SVM $\gamma = 1$ ROC curve (area = %0.2f)' % roc_auc3, color='b')
     ax1.plot(false_pos_rate4, true_pos_rate4, label='SVM $\gamma = 50$ ROC curve (area = %0.2f)' % roc_auc4, color='r')
     ax1.set_title('Training Data')

     y_test_score3 = svm3.decision_function(X_test)
     y_test_score4 = svm4.decision_function(X_test)

     false_pos_rate3, true_pos_rate3, _ = roc_curve(y_test, y_test_score3)
     roc_auc3 = auc(false_pos_rate3, true_pos_rate3)

     false_pos_rate4, true_pos_rate4, _ = roc_curve(y_test, y_test_score4)
     roc_auc4 = auc(false_pos_rate4, true_pos_rate4)

     ax2.plot(false_pos_rate3, true_pos_rate3, label='SVM $\gamma = 1$ ROC curve (area = %0.2f)' % roc_auc3, color='b')
     ax2.plot(false_pos_rate4, true_pos_rate4, label='SVM $\gamma = 50$ ROC curve (area = %0.2f)' % roc_auc4, color='r')
     ax2.set_title('Test Data')

     for ax in fig.axes:
         ax.plot([0, 1], [0, 1], 'k--')
         ax.set_xlim([-0.05, 1.0])
         ax.set_ylim([0.0, 1.05])
         ax.set_xlabel('False Positive Rate')
         ax.set_ylabel('True Positive Rate')
```



Conclusions :

- After performing grid search we can conclude that for the generated train and test data cost parameter of 10 and gamma 0.5 gives the best test accuracy of 87%.
- The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve. Second model with gamma equal to 50 is more flexible and offers greater testing accuracy. Hence, the second model is better.
- When the cost argument is small, then the margins will be wide and many support vectors will be on the margin or will violate the margin.
- When the cost argument is large, then the margins will be narrow and there will be few support vectors on the margin or violating the margin.
- We perform various cross validation techniques to determine the best value of cost parameter and gamma.