Sardar Patel Institute of Technology,Mumbai
Department of Electronics and Telecommunication Engineering
B.E. Sem-VII (2021-2022)
Data Analytics

**Name: Sneha Ghuge** **UID:** 2019110015 **BE ETRX** **DA LAB 8**

**AIM :** To Understand Support Vector Machine algorithm through building SVM algorithm in Python

**PROBLEM STATEMENT :**

1. Support Vector Machines In this lab, we'll use the SVC module from the sklearn.svm package to demonstrate the support vector classifier and the SVM

2. Support vector Classifier 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.

3. The sklearn.grid_search module includes 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 to perform 10-fold cross-validation to compare SVMs with a linear kernel, using a range of values of the cost param-eter.

4. Support Vector Machine: Kernel Method
a. In order to fit an SVM using a non-linear kernel, we once again use the SVC() function. However, now we use a different value of the parameter kernel. To fit an SVM with a polynomial kernel we use kernel ="poly", and to fit an SVM with a radial kernel we use kernel = "rbf". In the former case we also use the degree argument to specify a degree for the polynomial kernel, and in the latter case we use gamma to specify a value of γ for the radial basis kernel. Let's generate some data with a non-linear class boundary

5. Draw ROC curve and Analyze the result

**Code and Output:**

# Importing the required libraries

```
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

# 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)
```

# Support Vector Machines

Here we eill use the **SVC** module from the **sklearn.svm** package to demonstrate the support vector classifier and the SVM:
The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

```
from sklearn.svm import SVC
```

Python

# Support Vector Classifier

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:

```
# 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
```

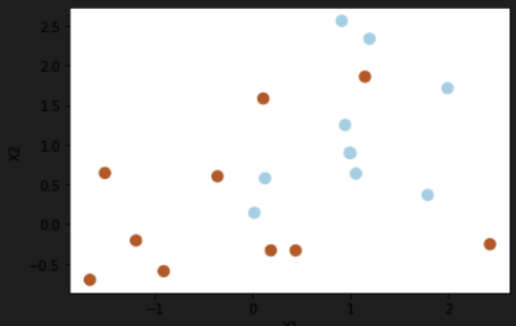
[3] Python

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

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

[4] Pyth

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




**Question : Linear or Non Linear?**
Non linear
The way red dots have an irregular distribution we can see that the classes are not linearly separable

Hence we now fit the support vector classifier using the linear argument for the kernel parameter

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

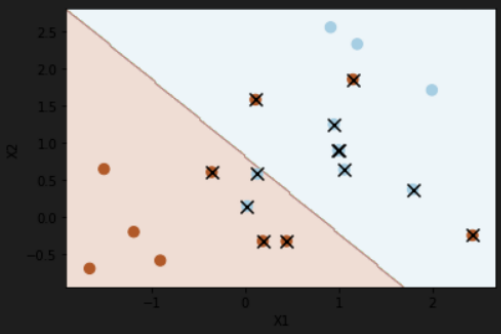
[5] Python

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

After changing the parameter to linear we can now plot the support vector classifier by calling the `plot_svc()` function (defined earlier) on the output of the call to `SVC()`, as well as the data used in the call to `SVC()`:

```
plot_svc(svc, X, y)
```

[6] Python




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

The region of feature space that will be assigned to the −1 class is shown in light blue, and the region that will be assigned to the +1 class is shown in brown. The decision boundary between the two classes is linear (because we used the argument `kernel = "linear"`).

**Question : Number of Support vectors?**
13, they are plotted with crosses where as the observations are plotted as circles.
Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane.

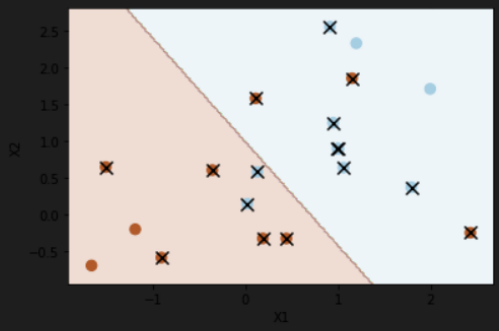
The support_ attribute provides the index of the training data for each of the support vectors in SVC.support_vectors_

```
[7]  svc.support_                                                    Python
```

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

What if we instead used a smaller value of the `cost` parameter?

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




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

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

The **sklearn.grid_search** module includes 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:

```
[9]
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)
```

Python

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

The cross-validation errors for each of these models:

```
[10]
clf.cv_results_
```

Python

```
Output exceeds the size limit. Open the full output data in a text editor
{'mean_fit_time': array([0.0007962 , 0.00038185, 0.00040581, 0.00044677, 0.00039206,
        0.00044398, 0.0005291 ]),
 'std_fit_time': array([6.42355699e-04, 1.60914703e-05, 3.63694941e-05, 1.33526514e-04,
        1.58849051e-05, 8.87448782e-05, 8.33982464e-05]),
 'mean_score_time': array([0.00037389, 0.00021639, 0.0002368 , 0.00021935, 0.00025113,
        0.0002269 , 0.00021975]),
 'std_score_time': array([1.87390384e-04, 9.41991776e-06, 1.61311974e-05, 1.16766701e-05,
        6.72605011e-05, 1.65418686e-05, 1.37301159e-05]),
 '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},
```

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

```
[11]
clf.best_params_
```
Python

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

The cost 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:

```
[12]
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
```
Python

Now we predict the class labels of these test observations. Here we use the best model obtained through cross-validation in order to make predictions which is the second one (svc2):

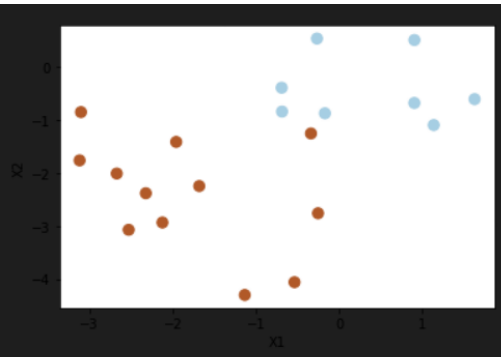
```
[13]
svc2 = SVC(C=0.001, kernel='linear')
svc2.fit(X, y)
y_pred = svc2.predict(X_test)
pd.DataFrame(confusion_matrix(y_test, y_pred), index=svc2.classes_, columns=svc2.classes_)
```
Python

With this value of cost `c`, from the above confusion matrix we can see that 14 of the test observations were 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:

```
[14]
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')
```
Python

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

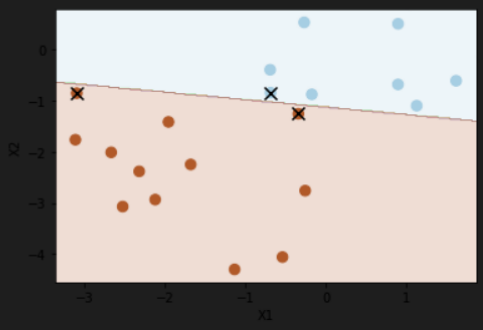



+ Code + Markdown

Now we can see that 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. High value of cost implies no mis classification

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

[15] Python

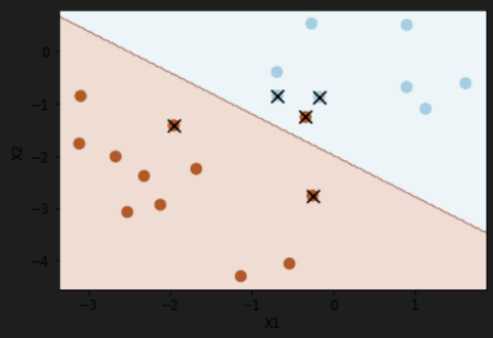



```
Number of support vectors:  3
```

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`:

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

[16] Python




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

Question : 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`**.

# Support Vector Machine

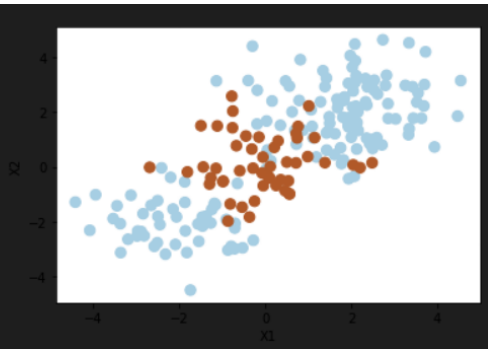
In order to fit an SVM using a **non-linear kernel**, we once again use the **`SVC()`** function. However, now we use a different value of the parameter kernel. To fit an SVM with a polynomial kernel we use **`kernel` = `"poly"`**, and to fit an SVM with a radial kernel we use **`kernel` = `"rbf"`**. In the former case we also use the **`degree`** argument to specify a degree for the polynomial kernel, and in the latter case we use **`gamma`** to specify a value of $\gamma$ for the radial basis kernel.

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

```
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')
```

[17] Python

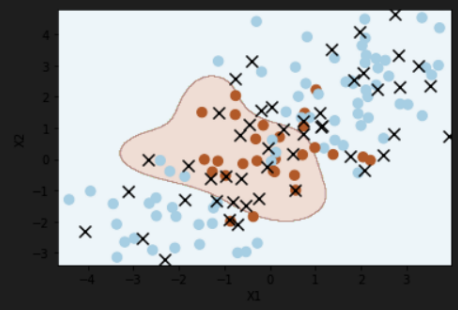
```
Text(0, 0.5, 'X2')
```




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 $\gamma = 1$:

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

[18] Python



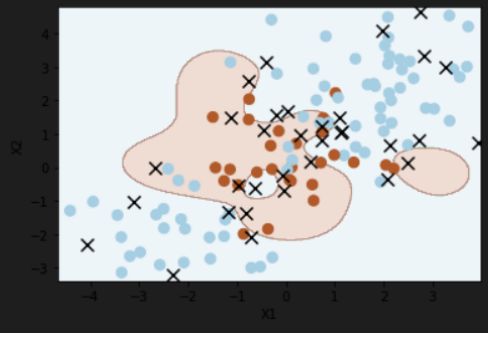

```
Number of support vectors:  51
```

**Question : Number of support vectors?**
51 - denoted with cross

Not too shabby! The plot shows that the resulting SVM has a decidedly non-linear boundary. 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:

```
# 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)
```

[19] Python




**Question : Number of support vectors?**
36 - denoted with cross

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 $\gamma$ and cost for an SVM with a radial kernel:

```
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_
```

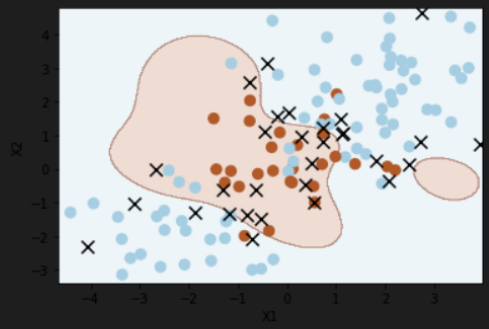
[20] Python

```
{'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:

```
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))
```

[21] Python




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

**Question : Number of support vectors?**
32 - denoted with cross

Hence we have achieved an accuracy of 87%.

# ROC Curves

The **auc**() function from the **sklearn.metrics** package can be used to produce ROC curves.

```
from sklearn.metrics import auc
from sklearn.metrics import roc_curve
```

[22] Python

We create two models here where one of them has a higher gamma showing more flexibility

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

[23] Python

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

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

[24] Python

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

SVMs and support vector classifiers output class labels for each observation. However, it is also possible to obtain fitted values for each observation, which are the numerical scores used to obtain the class labels. For instance, in the case of a support vector classifier, the fitted value for an observation $X = (X_1, X_2, ..., X_p)^T$ takes the form $\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + ... + \hat{\beta}_p X_p$.

In case of an SVM of a non-linear kernel the sign of the fitted value determines on which side of the decision boundary the observation lies. Therefore, the relationship between the fitted value and the class prediction for a given observation is simple: if the fitted value exceeds zero then the observation is assigned to one class, and if it is less than zero than it is assigned to the other.

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

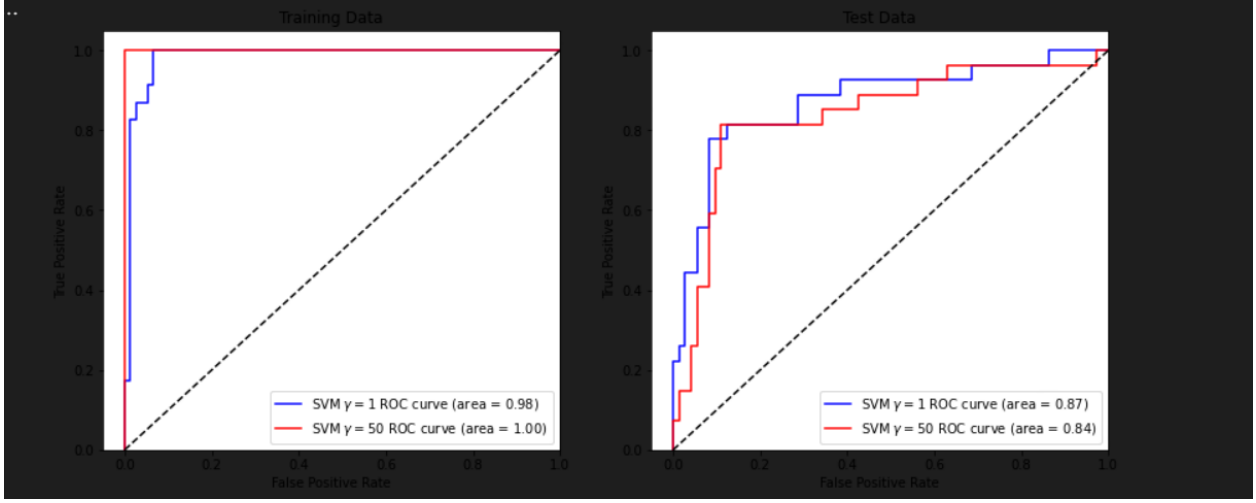
[25]
```python
y_train_score3 = svm3.decision_function(X_train)
y_train_score4 = svm4.decision_function(X_train)
```
Python

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

```python
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')
```

```python
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')
    ax.legend(loc="lower right")
```

26] Python




**Conclusion:**

We know that ROC is a probability curve and or the area under the curve represents the degree or measure of separability. It tells us how much the model is capable of distinguishing between classes. Higher the AUC, the better the model is at predicting 0 classes as 0 and 1.

Hence by seeing at the plots we can see that the training AUC is sufficiently large but the model where the cost function was more higher; which proves the fact that when gamma increases the number of training errors decreases.
But for testing SVM with lesser gamma performed better