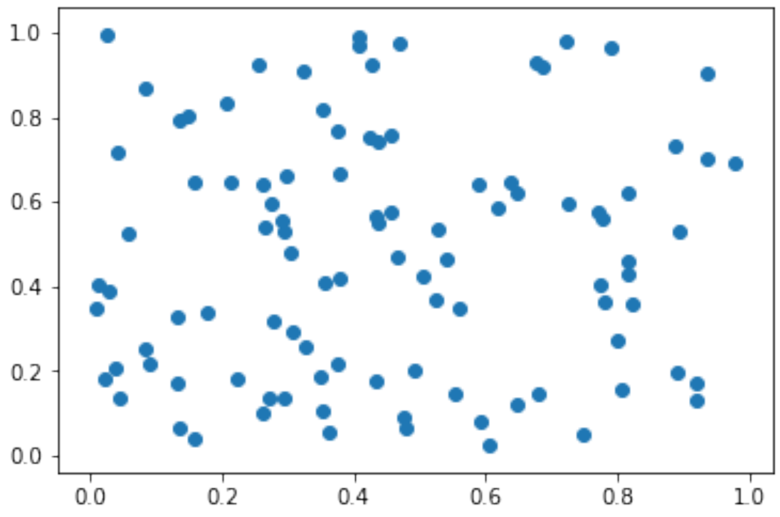
```
import numpy as np
import random
import matplotlib.pyplot as plt
```

## Q1: svm with radial kernel vs linear kernel

```
datapoint = np.random.rand(100,2)
```

```
plt.scatter(datapoint[:,0],datapoint[:,1])
plt.show()
```



```
np.sum(Y)
```

```
16
```

```
Y = []
for i in range(100):
    if (datapoint[i,0]-0.5)**2 + (datapoint[i,1]-0.5)**2 > 0.25:
        Y.append(1)
    else:
        Y.append(0)
```

```
Y = np.array(Y)
indice = np.random.permutation(list(range(100)))
train_indice = indice[:int(0.5*len(indice))]
test_indice = indice[int(0.5*len(indice)):]
X_train = datapoint[train_indice]
X_test = datapoint[test_indice]
Y_train = Y[train_indice]
Y_test = Y[test_indice]
```

```
## model
from sklearn import svm
linear_clf = svm.SVC(kernel = 'linear').fit(X_train,Y_train)
radial_clf = svm.SVC(kernel = 'rbf').fit(X_train,Y_train)
error_rate_linear = 1-np.sum(np.ones(len(X_train))[linear_clf.predict(X_train)
== Y_train])/len(X_train)
error_rate_radial = 1-np.sum(np.ones(len(X_train))[radial_clf.predict(X_train)
== Y_train])/len(X_train)
```

```
error_rate_linear,error_rate_radial
```

```
(0.33999999999999997, 0.07999999999999996)
```

```
error_rate_linear_test = 1-np.sum(np.ones(len(X_test))
[linear_clf.predict(X_test) == Y_test])/len(X_test)
error_rate_radial_test = 1-np.sum(np.ones(len(X_test))
[radial_clf.predict(X_test) == Y_test])/len(X_test)
```

```
error_rate_linear_test,error_rate_radial_test
```

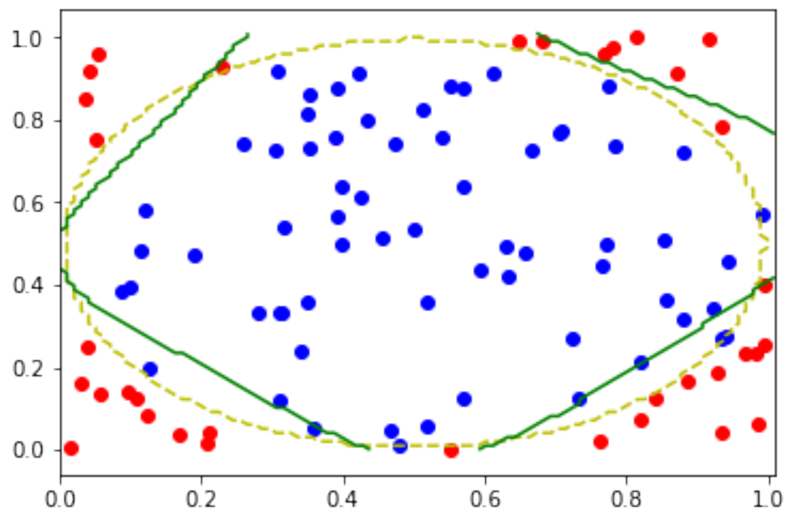
```
(0.35999999999999999, 0.12)
```

```
#plot
xx = np.linspace(0,1.01,100)
yy = np.linspace(0,1.01,100)
X1, X2 = np.meshgrid(xx,yy)
```

```
Z_linear = np.empty(X1.shape)
Z_radial = np.empty(X1.shape)
Z_true = np.empty(X1.shape)
for (i,j), val in np.ndenumerate(X1):
    x1 = val
    x2 = X2[i,j]
    if (x1 - 0.5)**2 + (x2 - 0.5)**2 > 0.25:
        Z_true[i,j] = 1
    else:
        Z_true[i,j] = 0
    Z_linear[i,j] = linear_clf.predict(np.array([x1,x2]).reshape(1,-1))
    Z_radial[i,j] = radial_clf.predict(np.array([x1,x2]).reshape(1,-1))
```

```
for i in range(len(datapoint)):
    if Y[i] == 1:
        plt.scatter(datapoint[i,0],datapoint[i,1],c='r')
    else:
        plt.scatter(datapoint[i,0],datapoint[i,1],c='b')
plt.contour(X1,X2,Z_true, linestyles = 'dashed', colors='y',levels = 1)
#plt.contour(X1,X2,Z_linear)
plt.contour(X1,X2,Z_radial, linestyles = 'solid',colors='g',levels = 1)
plt.legend()
plt.show()
```

```
/Users/apple/anaconda3/lib/python3.6/site-packages/matplotlib/contour.py:1180:
UserWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```



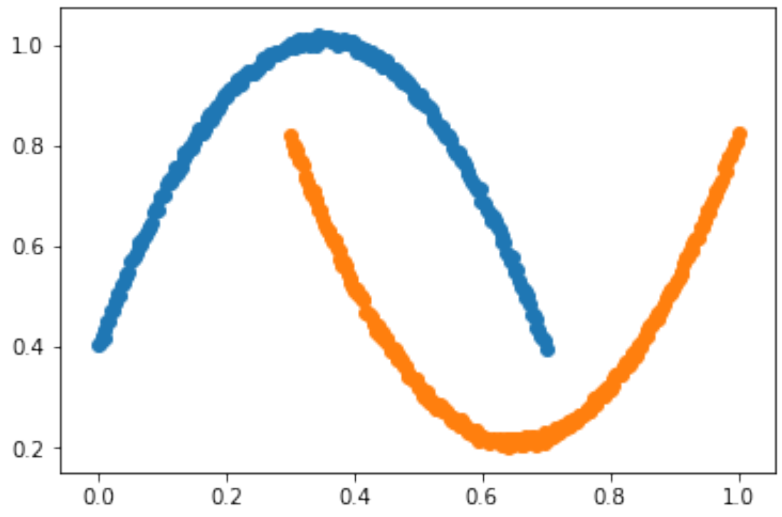
**comments:**

we can see that svm with radial kernel outperforms svm with linear kernel both in training dataset and testing dataset. And we can see that svm with radial kernel( lines in green) depict the original decision boundary(line in yellow), while svm with linear kernel can not make a decision in our dataset (there is no decision boundary formed by svm with linear kernel).

## Q 2 & 3: nonlinear and overlapping boundary

```
x1 = np.linspace(0,0.7,250)
x2 = np.linspace(0.3,1,250)
y1 = -5*(x1-0.35)**2 + 1 + np.random.rand(1,250)*0.02
y2 = 5*(x2-0.65)**2 + 0.2 + np.random.rand(1,250)*0.02
```

```
plt.scatter(x1, y1)
plt.scatter(x2, y2)
plt.show()
```



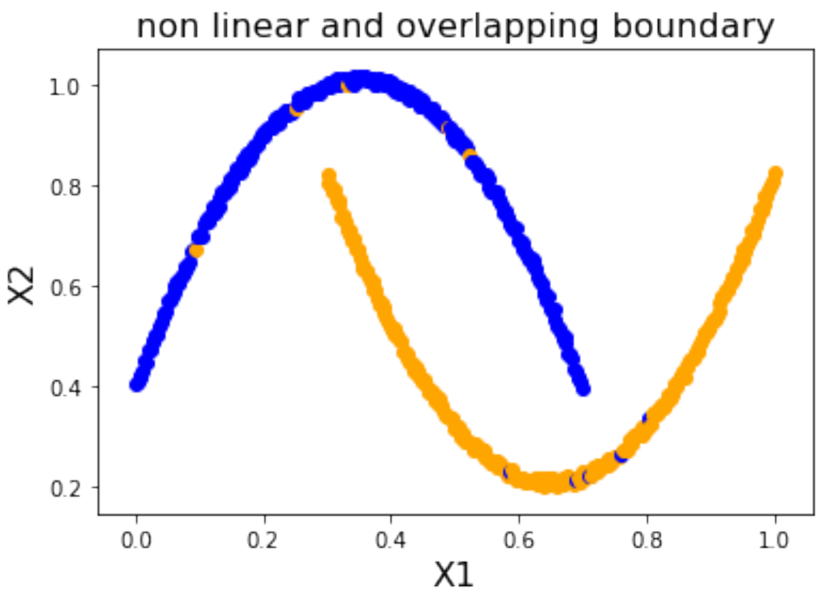
```
import copy
```

```
X1 = list(x1)
X1.extend(list(x2))
X2 = list(y1[0])
X2.extend(list(y2[0]))

X_feature =
np.concatenate((np.array(X1).reshape(-1,1),np.array(X2).reshape(-1,1)),axis =
1)
Y = [0]*250
Y.extend([1]*250)
Y = np.array(Y)

overlapping_point = random.sample(list(range(500)),10)
for i in overlapping_point:
    Y[i] = abs(Y[i] - 1)
```

```
color = np.array(['b','orange'])
for i in range(len(X_feature)):
    plt.scatter(X_feature[i,0],X_feature[i,1],c = color[Y[i]])
plt.xlabel('X1', size = 16)
plt.ylabel('X2', size = 16)
plt.title('non linear and overlapping boundary', size =16)
plt.show()
```



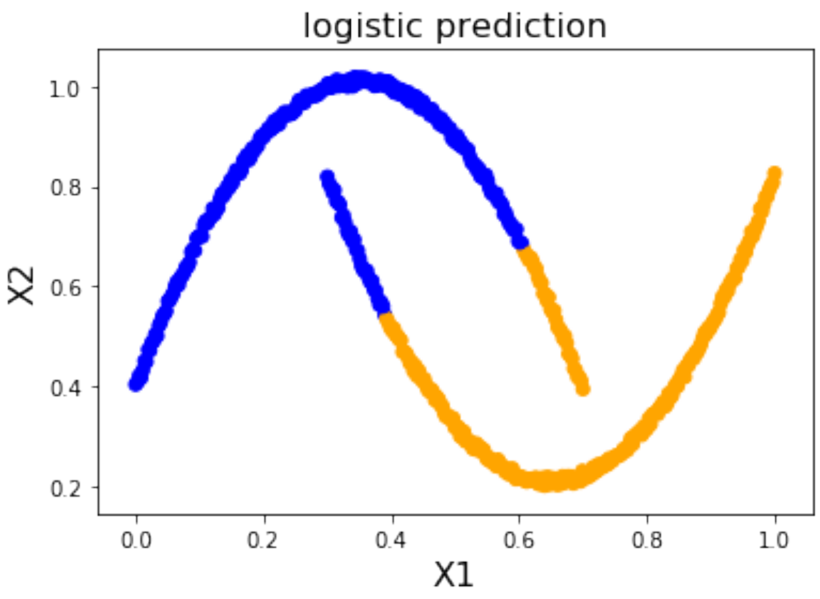
## Q4 & 5

```
from sklearn.linear_model import LogisticRegression
```

```
indice = np.random.permutation(range(500))
X_train = X_feature[indice]
Y_train = Y[indice]
```

```python
clf = LogisticRegression(random_state=0).fit(X_train, Y_train)
```

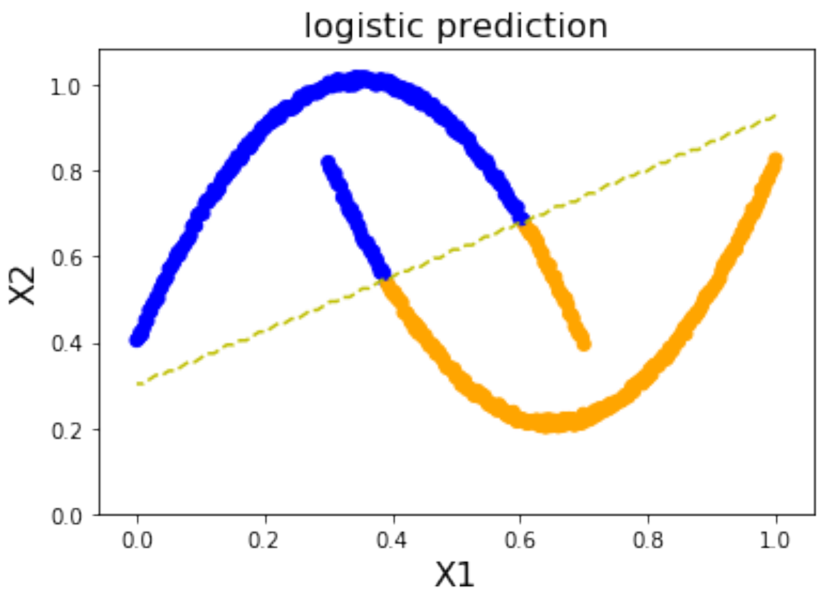
```python
Y_predict = clf.predict(X_train)
plt.scatter(X_train[:,0],X_train[:,1],c = color[Y_predict])
plt.xlabel('X1', size = 16)
plt.ylabel('X2', size = 16)
plt.title('logistic prediction', size =16)
plt.show()
```



```python
def calc_contour_2d(x_min,x_max, clf, func):
    xx = np.linspace(x_min,x_max,100)
    yy = np.linspace(x_min,x_max,100)
    X1, X2 = np.meshgrid(xx,yy)
    Z = np.empty(X1.shape)
    for (i,j), val in np.ndenumerate(X1):
        x1 = val
        x2 = X2[i,j]
        Z[i,j] = clf.predict(np.array(func(x1,x2)).reshape(1,-1))
    return Z, X1, X2
```

```python
def func_linear(x1,x2):
    return([x1,x2])
Z_logistic, X1, X2 = calc_contour_2d(0,1,clf, func_linear)
plt.contour(X1,X2,Z_logistic, linestyles = 'dashed', colors='y',levels = 1)
plt.scatter(X_train[:,0],X_train[:,1],c = color[Y_predict])
plt.xlabel('X1', size = 16)
plt.ylabel('X2', size = 16)
plt.title('logistic prediction', size =16)
plt.show()
```

```
/Users/apple/anaconda3/lib/python3.6/site-packages/matplotlib/contour.py:1180:
UserWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```



## Q6 & 7

```
X1 = X_train[:,0].reshape(-1,1)
X1_square = (X_train[:,0]**2).reshape(-1,1)
X2_square = (X_train[:,1]**2).reshape(-1,1)
X1X2 = (X_train[:,0]*X_train[:,1]).reshape(-1,1)
X1_log = np.log((X_train[:,0]).reshape(-1,1)+1)
X2_log = np.log((X_train[:,1]).reshape(-1,1)+1)
X_train_nonlinear = np.concatenate((X1X2, X1_log, X1_square, X1),axis =1)
```

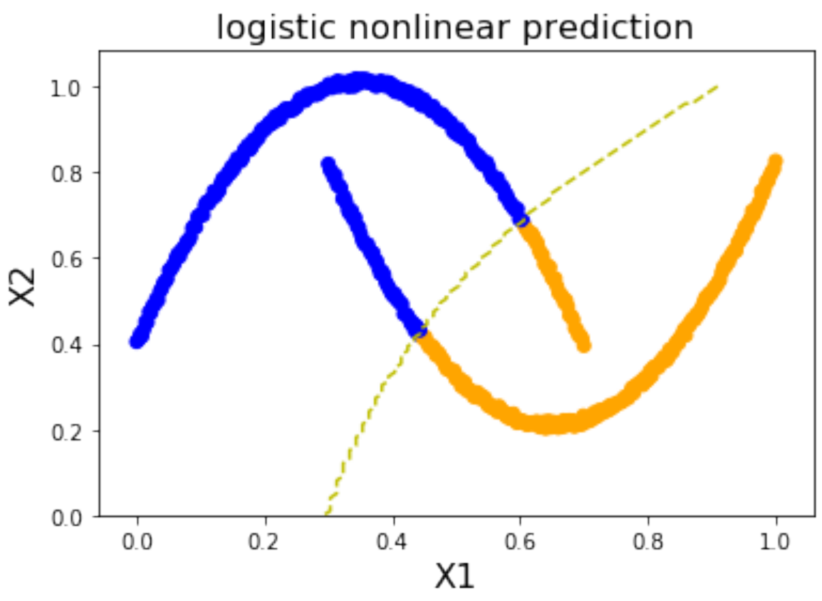
```
X_train_nonlinear.shape
```

```
(500, 3)
```

```
clf_logistic_nonlinear =
LogisticRegression(random_state=0).fit(X_train_nonlinear, Y_train)
```

```
def func_lg_non_linear(x1,x2):
    return ([x1*x2, np.log(x1+1), x1**2, x1])
Z_logistic, X1, X2 =
calc_contour_2d(0,1,clf_logistic_nonlinear,func_lg_non_linear)
plt.contour(X1,X2,Z_logistic, linestyles = 'dashed', colors='y',levels = 1)
Y_predict = clf_logistic_nonlinear.predict(X_train_nonlinear)
plt.scatter(X_train[:,0],X_train[:,1],c = color[Y_predict])
plt.xlabel('X1', size = 16)
plt.ylabel('X2', size = 16)
plt.title('logistic nonlinear prediction', size =16)
plt.show()
```

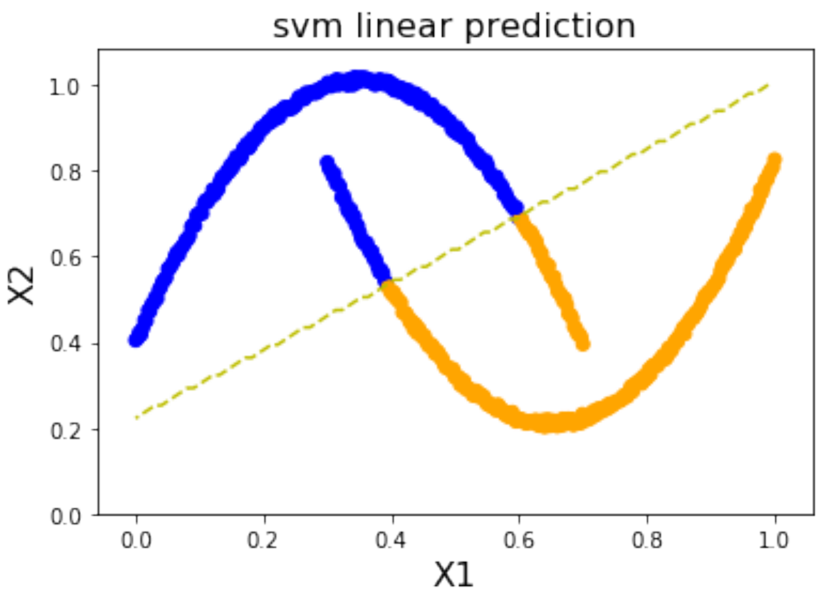
```
/Users/apple/anaconda3/lib/python3.6/site-packages/matplotlib/contour.py:1180:
UserWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```



## Q8:

```
svm_linear_clf = svm.SVC(kernel = 'linear').fit(X_train,Y_train)
Z_svm, X1, X2 = calc_contour_2d(0,1,svm_linear_clf,func_linear)
plt.contour(X1, X2, Z_svm, linestyles = 'dashed', colors='y',levels = 1)
Y_predict = svm_linear_clf.predict(X_train)
plt.scatter(X_train[:,0],X_train[:,1],c = color[Y_predict])
plt.xlabel('X1', size = 16)
plt.ylabel('X2', size = 16)
plt.title('svm linear prediction', size =16)
plt.show()
```

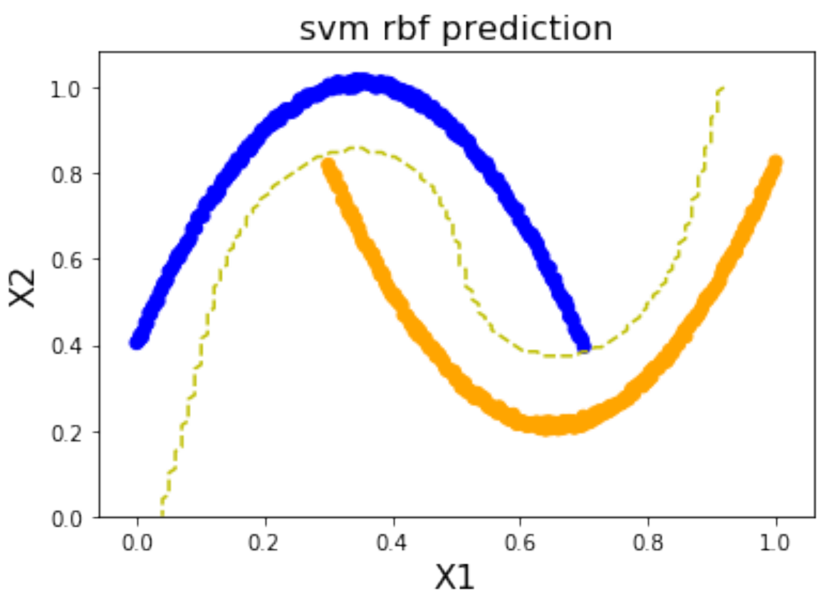
```
/Users/apple/anaconda3/lib/python3.6/site-packages/matplotlib/contour.py:1180:
UserWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```



## Q9:

```
svm_rbf_clf = svm.SVC(kernel = 'rbf').fit(X_train,Y_train)
Z_svm, X1, X2 = calc_contour_2d(0,1,svm_rbf_clf,func_linear)
plt.contour(X1, X2, Z_svm, linestyles = 'dashed', colors='y',levels = 1)
Y_predict = svm_rbf_clf.predict(X_train)
plt.scatter(X_train[:,0],X_train[:,1],c = color[Y_predict])
plt.xlabel('X1', size = 16)
plt.ylabel('X2', size = 16)
plt.title('svm rbf prediction', size =16)
plt.show()
```

```
/Users/apple/anaconda3/lib/python3.6/site-packages/matplotlib/contour.py:1180:
UserWarning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```



## Q10: results comments:

When the decision boundary of data is nonlinear, we can see that svm with radial kernel outperforms logistics model and svm linear model, and the svm with radial kernel can depict the boundary clearly for the training data. However, svm with radial kernel will introduce high variance to the classification, and the model is hard to interpret, and may even result in poor performance of classificaiton on test dataset. (Although this is not seen in our datasetx)

To balance the trade-off between the bias and variance when classifying data has nonlinear boundaries, we should not just believe in every results the SVM throws to us, since the model has a high probability of overfitting(the radial kernel is more complex in nature), but still try to use more interpretable model like the logistics regression or svm with linear kernel, and feed the model with nonlinear features that are constructed through understanding of the data, reducing the probability of overfitting.

## Q11 & 12:

```
x1 = np.linspace(0,0.7,250)
x2 = np.linspace(0.3,1,250)
y1 = -5*(x1-0.35)**2 + 1 + np.random.rand(1,250)*0.3
y2 = 5*(x2-0.65)**2 + 0.2 + np.random.rand(1,250)*0.3
X1 = list(x1)
X1.extend(list(x2))
X2 = list(y1[0])
X2.extend(list(y2[0]))

X_feature =
np.concatenate((np.array(X1).reshape(-1,1),np.array(X2).reshape(-1,1)),axis =
1)
Y = [0]*250
Y.extend([1]*250)
Y = np.array(Y)
```

```
indice = np.random.permutation(range(500))
```

```
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
kf = KFold(n_splits=10)
cost_train_errot_rate = {}
cost_test_errot_rate = {}
for cost in list(np.linspace(0.1,1,11))+[5,10,100,1000]:
    train_error_rate = []
    test_error_rate = []
    for train_indice, test_indice in kf.split(indice):
        x_train = X_feature[train_indice]
        y_train = Y[train_indice]
        x_test = X_feature[test_indice]
        y_test = Y[test_indice]
        clf = svm.SVC(C = cost, kernel='rbf').fit(x_train, y_train)
        train_error_rate.append(1-np.sum(np.ones(len(x_train))
[clf.predict(x_train) == y_train])/len(x_train))
        test_error_rate.append(1-np.sum(np.ones(len(x_test))
[clf.predict(x_test) == y_test])/len(x_test))
    cost_train_errot_rate[cost] = np.mean(train_error_rate)
    cost_test_errot_rate[cost] = np.mean(test_error_rate)
```

```
sorted(cost_train_errot_rate.items(), key = lambda x:x[1])
```

```
[(1000, 0.012222222222222212),
 (100, 0.014222222222222225),
 (10, 0.025777777777777778),
 (5, 0.030444444444444441),
 (1.0, 0.051333333333333356),
 (0.90999999999999992, 0.052222222222222246),
 (0.81999999999999995, 0.054666666666666683),
 (0.72999999999999998, 0.056888888888888892),
 (0.64000000000000001, 0.060444444444444446),
 (0.54999999999999993, 0.064222222222222208),
 (0.45999999999999996, 0.067333333333333314),
 (0.37, 0.071333333333333318),
 (0.28000000000000003, 0.077111111111111103),
 (0.19, 0.081555555555555534),
 (0.10000000000000001, 0.088888888888888878)]
```

```
sorted(cost_test_errot_rate.items(), key = lambda x:x[1])
```

```
[(5, 0.091999999999999998),
 (10, 0.10400000000000001),
 (0.90999999999999992, 0.184),
 (1.0, 0.184),
 (0.81999999999999995, 0.192),
 (0.72999999999999998, 0.20999999999999996),
 (0.54999999999999993, 0.21600000000000003),
 (0.64000000000000001, 0.21799999999999997),
 (100, 0.21799999999999997),
 (0.45999999999999996, 0.22800000000000004),
 (0.37, 0.24399999999999999),
 (1000, 0.24399999999999999),
 (0.28000000000000003, 0.27400000000000002),
 (0.10000000000000001, 0.28800000000000003),
 (0.19, 0.28800000000000003)]
```

### comments:

we can see that, as the cost value increases, the training error rate will decrease and finally is decreased to 0, while the cross validation error rate will decrease at first, and then it will slightly increase as the cost value becomes larger.

## Q13:

```
x1 = np.linspace(0,0.7,50)
x2 = np.linspace(0.3,1,50)
y1 = -5*(x1-0.35)**2 + 1 + np.random.rand(1,50)*0.3
y2 = 5*(x2-0.65)**2 + 0.2 + np.random.rand(1,50)*0.3
X1 = list(x1)
X1.extend(list(x2))
X2 = list(y1[0])
X2.extend(list(y2[0]))

X_feature_test =
np.concatenate((np.array(X1).reshape(-1,1),np.array(X2).reshape(-1,1)),axis =
1)
Y_test = [0]*50
Y_test.extend([1]*50)
Y_test = np.array(Y_test)
cost_Test_error_rate = {}
for cost in list(np.linspace(0.1,1,11))+[5,10,100,1000]:
    clf = svm.SVC(C = cost, kernel='rbf').fit(X_feature, Y)
    cost_Test_error_rate[cost] = 1-np.sum(np.ones(len(X_feature_test))\
                                          [clf.predict(X_feature_test) ==
Y_test])/len(Y_test)
```

```
sorted(cost_Test_error_rate.items(), key = lambda x:x[1])
```

```
[(100, 0.0),
 (1000, 0.010000000000000009),
 (5, 0.030000000000000027),
 (10, 0.030000000000000027),
 (0.72999999999999998, 0.060000000000000053),
 (0.81999999999999995, 0.060000000000000053),
 (0.90999999999999992, 0.060000000000000053),
 (1.0, 0.060000000000000053),
 (0.19, 0.069999999999999951),
 (0.28000000000000003, 0.069999999999999951),
 (0.37, 0.069999999999999951),
 (0.45999999999999996, 0.069999999999999951),
 (0.54999999999999993, 0.069999999999999951),
 (0.64000000000000001, 0.069999999999999951),
 (0.10000000000000001, 0.10999999999999999)]
```

### comments:

As we can see, the cost value that best predicts the test data is 100, which is similiar to the cost value which gives the best performance on training data, while different to the cost value that best predicts the classification of cross validation data: 5. This may due the fact that when we have comparatively fewer data for training( in the case of conducting the cross validation), and a model with less penalty on the misclassifying value can perform better on unseen dataset; and when we have more data for training(in the case of conducting the testing ), we can have a higher penalty on the misclassifying value since the possibility of overfitting is reduced by the increasing training dataset.

## Q15:

```
import pandas as pd
from sklearn import preprocessing
```

```
df_train = pd.read_csv('gss_train.csv')

data_train = copy.copy(df_train)

Y_train = df_train['colrac']

data_train.drop(['colrac'],axis = 1, inplace=True)

X_train = np.array(data_train)
X_scaled_train = preprocessing.scale(X_train)

indice = np.random.permutation(list(range(len(X_scaled_train))))
```

```
Y_train.describe()
```

```
count    1476.000000
mean        0.525068
std         0.499540
min         0.000000
25%         0.000000
50%         1.000000
75%         1.000000
max         1.000000
Name: colrac, dtype: float64
```

```
kf = KFold(n_splits=10)
cost_train_errot_rate = {}
cost_test_errot_rate = {}
for cost in list(np.linspace(0.1,1,11))+[5,10,100,1000]:
    train_error_rate = []
    test_error_rate = []
    for train_indice, test_indice in kf.split(indice):
        x_train = X_scaled_train[train_indice]
        y_train = Y_train[train_indice]
        x_test = X_scaled_train[test_indice]
        y_test = Y_train[test_indice]
        clf = svm.SVC(C = cost, kernel='rbf').fit(x_train, y_train)
        train_error_rate.append(1-np.sum(np.ones(len(x_train))
[clf.predict(x_train) == y_train])/len(x_train))
        test_error_rate.append(1-np.sum(np.ones(len(x_test))
[clf.predict(x_test) == y_test])/len(x_test))
    cost_train_errot_rate[cost] = np.mean(train_error_rate)
    cost_test_errot_rate[cost] = np.mean(test_error_rate)
```

## training error rate with differenct cost value:

```
sorted(cost_train_errot_rate.items(), key = lambda x:x[1])
```

```
[(100, 0.0),
 (1000, 0.0),
 (10, 0.0012044793168157985),
 (5, 0.0037640403600859517),
 (1.0, 0.079870724432719573),
 (0.90999999999999992, 0.087775084536792766),
 (0.81999999999999995, 0.094775603542839537),
 (0.72999999999999998, 0.10207789396865112),
 (0.64000000000000001, 0.11156357937392913),
 (0.54999999999999993, 0.12210229178565277),
 (0.45999999999999996, 0.13158746725049181),
 (0.37, 0.14295443625517873),
 (0.28000000000000003, 0.15974150552548796),
 (0.19, 0.17961490431250962),
 (0.10000000000000001, 0.2019720529975432)]
```

## cross validation error rate with different cost value

```
sorted(cost_test_errot_rate.items(), key = lambda x:x[1])
```

```
[(5, 0.20596157381871666),
 (0.90999999999999992, 0.20868725868725865),
 (1.0, 0.20868725868725865),
 (0.81999999999999995, 0.21207023349880494),
 (10, 0.21273671630814489),
 (100, 0.21341698841698845),
 (1000, 0.21341698841698845),
 (0.72999999999999998, 0.2141018569589998),
 (0.37, 0.21412024269167124),
 (0.45999999999999996, 0.21480051480051482),
 (0.64000000000000001, 0.2154715940430226),
 (0.28000000000000003, 0.21614267328553044),
 (0.54999999999999993, 0.2161472697186983),
 (0.19, 0.21818808604522894),
 (0.10000000000000001, 0.23107648464791325)]
```

### comments:

Model with cost value of 5 give the best performance on the cross validation dataset with the error rate of 0.205. When the features are complex and the relation between the features and the target is non-linear, we would better  use a svm with a cost value that is not so large and to avoid the problem of overfitting.

## Q16:

```
kf = KFold(n_splits=10)
Cost = [0.5,1,5,10]
Kernel = ['rbf','poly']
Degree = [1,3,5,10]
Gamma = ['scale','auto']
Parameter_train_errot_rate = {}
Parameter_test_errot_rate = {}
for c in Cost:
    for k in Kernel:
        for d in Degree:
            for g in Gamma:
                train_error_rate = []
                test_error_rate = []
                for train_indice, test_indice in kf.split(indice):
                    x_train = X_scaled_train[train_indice]
                    y_train = Y_train[train_indice]
                    x_test = X_scaled_train[test_indice]
                    y_test = Y_train[test_indice]
                    clf = svm.SVC(C = c, kernel = k, degree = d, gamma =
g).fit(x_train, y_train)
                    train_error_rate.append(1-np.sum(np.ones(len(x_train))
[clf.predict(x_train) == y_train])/len(x_train))
                    test_error_rate.append(1-np.sum(np.ones(len(x_test))
[clf.predict(x_test) == y_test])/len(x_test))
                Parameter_train_errot_rate[(c,k,d,g)] =
np.mean(train_error_rate)
                Parameter_test_errot_rate[(c,k,d,g)] = np.mean(test_error_rate)
```

```
sorted(Parameter_train_errot_rate.items(),key = lambda x:x[1])
```

```
[((10, 'poly', 5, 'scale'), 0.0),
 ((10, 'poly', 5, 'auto'), 0.0),
 ((10, 'poly', 3, 'scale'), 0.00015054574958978106),
 ((10, 'poly', 3, 'auto'), 0.00015054574958978106),
 ((5, 'poly', 5, 'scale'), 0.0012044226567670235),
 ((5, 'poly', 5, 'auto'), 0.0012044226567670235),
 ((10, 'rbf', 1, 'scale'), 0.0012044793168157985),
 ((10, 'rbf', 1, 'auto'), 0.0012044793168157985),
 ((10, 'rbf', 3, 'scale'), 0.0012044793168157985),
 ((10, 'rbf', 3, 'auto'), 0.0012044793168157985),
 ((10, 'rbf', 5, 'scale'), 0.0012044793168157985),
 ((10, 'rbf', 5, 'auto'), 0.0012044793168157985),
 ((10, 'rbf', 10, 'scale'), 0.0012044793168157985),
 ((10, 'rbf', 10, 'auto'), 0.0012044793168157985),
 ((5, 'poly', 3, 'scale'), 0.0020325092695839819),
 ((5, 'poly', 3, 'auto'), 0.0020325092695839819),
 ((5, 'rbf', 1, 'auto'), 0.0036887391552666736),
 ((5, 'rbf', 3, 'auto'), 0.0036887391552666736),
 ((5, 'rbf', 5, 'auto'), 0.0036887391552666736),
 ((5, 'rbf', 10, 'auto'), 0.0036887391552666736),
 ((5, 'rbf', 1, 'scale'), 0.0037640403600859517),
 ((5, 'rbf', 3, 'scale'), 0.0037640403600859517),
 ((5, 'rbf', 5, 'scale'), 0.0037640403600859517),
 ((5, 'rbf', 10, 'scale'), 0.0037640403600859517),
 ((10, 'poly', 10, 'auto'), 0.009636571115160409),
 ((10, 'poly', 10, 'scale'), 0.009711985640077237l),
 ((1, 'poly', 5, 'auto'), 0.010086791862710432),
 ((1, 'poly', 5, 'scale'), 0.010086848522759196),
 ((5, 'poly', 10, 'auto'), 0.024240925326588537),
 ((5, 'poly', 10, 'scale'), 0.024542300126011961),
 ((0.5, 'poly', 5, 'scale'), 0.0283773355272104),
 ((0.5, 'poly', 5, 'auto'), 0.028452353431785815),
 ((1, 'poly', 3, 'auto'), 0.029358347611665635),
 ((1, 'poly', 3, 'scale'), 0.029433648816484914),
 ((1, 'poly', 10, 'scale'), 0.059473276854596725),
 ((1, 'poly', 10, 'auto'), 0.060225892282448082),
 ((0.5, 'poly', 3, 'auto'), 0.067524273164894344),
 ((0.5, 'poly', 3, 'scale'), 0.067674932234581667),
 ((1, 'rbf', 1, 'scale'), 0.079870724432719573),
 ((1, 'rbf', 3, 'scale'), 0.079870724432719573),
 ((1, 'rbf', 5, 'scale'), 0.079870724432719573),
 ((1, 'rbf', 10, 'scale'), 0.079870724432719573),
 ((1, 'rbf', 1, 'auto'), 0.08002121352226059),
 ((1, 'rbf', 3, 'auto'), 0.08002121352226059),
 ((1, 'rbf', 5, 'auto'), 0.08002121352226059),
 ((1, 'rbf', 10, 'auto'), 0.08002121352226059),
 ((0.5, 'poly', 10, 'scale'), 0.095973170333705018),
 ((0.5, 'poly', 10, 'auto'), 0.098682540545930911),
 ((0.5, 'rbf', 1, 'scale'), 0.12699517029744262),
 ((0.5, 'rbf', 3, 'scale'), 0.12699517029744262),
 ((0.5, 'rbf', 5, 'scale'), 0.12699517029744262),
 ((0.5, 'rbf', 10, 'scale'), 0.12699517029744262),
 ((0.5, 'rbf', 1, 'auto'), 0.12714560272693481),
 ((0.5, 'rbf', 3, 'auto'), 0.12714560272693481),
 ((0.5, 'rbf', 5, 'auto'), 0.12714560272693481),
 ((0.5, 'rbf', 10, 'auto'), 0.12714560272693481),
 ((5, 'poly', 1, 'scale'), 0.17946350866218827),
 ((5, 'poly', 1, 'auto'), 0.17961416773187561),
 ((1, 'poly', 1, 'auto'), 0.17991491927076253),
 ((10, 'poly', 1, 'scale'), 0.17999095705621587),
 ((10, 'poly', 1, 'auto'), 0.18006614494093756),
 ((1, 'poly', 1, 'scale'), 0.18014070956512279),
 ((0.5, 'poly', 1, 'auto'), 0.18413071019971533),
 ((0.5, 'poly', 1, 'scale'), 0.18428131260935388)]
```

```
sorted(Parameter_test_errot_rate.items(),key = lambda x:x[1])
```

```
[((1, 'poly', 3, 'scale'), 0.19715480786909359),
 ((1, 'poly', 3, 'auto'), 0.19715480786909359),
 ((0.5, 'poly', 3, 'auto'), 0.20192130906416622),
 ((0.5, 'poly', 3, 'scale'), 0.20259698473984189),
 ((5, 'rbf', 1, 'scale'), 0.20596157381871666),
 ((5, 'rbf', 3, 'scale'), 0.20596157381871666),
 ((5, 'rbf', 5, 'scale'), 0.20596157381871666),
 ((5, 'rbf', 10, 'scale'), 0.20596157381871666),
 ((5, 'rbf', 1, 'auto'), 0.20664184592756021),
 ((5, 'rbf', 3, 'auto'), 0.20664184592756021),
 ((5, 'rbf', 5, 'auto'), 0.20664184592756021),
 ((5, 'rbf', 10, 'auto'), 0.20664184592756021),
 ((0.5, 'poly', 1, 'scale'), 0.20731292517006805),
 ((0.5, 'poly', 1, 'auto'), 0.20731292517006805),
 ((5, 'poly', 1, 'scale'), 0.20734510020224306),
 ((5, 'poly', 1, 'auto'), 0.20734510020224306),
 ((10, 'poly', 1, 'scale'), 0.20802077587791876),
 ((10, 'poly', 1, 'auto'), 0.20802077587791876),
 ((1, 'rbf', 1, 'scale'), 0.20868725868725865),
 ((1, 'rbf', 1, 'auto'), 0.20868725868725865),
 ((1, 'rbf', 3, 'scale'), 0.20868725868725865),
 ((1, 'rbf', 3, 'auto'), 0.20868725868725865),
 ((1, 'rbf', 5, 'scale'), 0.20868725868725865),
 ((1, 'rbf', 5, 'auto'), 0.20868725868725865),
 ((1, 'rbf', 10, 'scale'), 0.20868725868725865),
 ((1, 'rbf', 10, 'auto'), 0.20868725868725865),
 ((5, 'poly', 5, 'auto'), 0.20869645155359445),
 ((5, 'poly', 5, 'scale'), 0.20870104798676231),
 ((5, 'poly', 3, 'scale'), 0.21070509284794997),
 ((5, 'poly', 3, 'auto'), 0.21070509284794997),
 ((1, 'poly', 1, 'scale'), 0.21072347858062149),
 ((1, 'poly', 1, 'auto'), 0.21072347858062149),
 ((10, 'rbf', 1, 'scale'), 0.21273671630814489),
 ((10, 'rbf', 1, 'auto'), 0.21273671630814489),
 ((10, 'rbf', 3, 'scale'), 0.21273671630814489),
 ((10, 'rbf', 3, 'auto'), 0.21273671630814489),
 ((10, 'rbf', 5, 'scale'), 0.21273671630814489),
 ((10, 'rbf', 5, 'auto'), 0.21273671630814489),
 ((10, 'rbf', 10, 'scale'), 0.21273671630814489),
 ((10, 'rbf', 10, 'auto'), 0.21273671630814489),
 ((10, 'poly', 5, 'scale'), 0.21343537414965988),
 ((10, 'poly', 5, 'auto'), 0.21343537414965988),
 ((10, 'poly', 3, 'scale'), 0.215457804743519),
 ((10, 'poly', 3, 'auto'), 0.2161334804191947),
 ((0.5, 'rbf', 1, 'scale'), 0.21615186615186613),
 ((0.5, 'rbf', 1, 'auto'), 0.21615186615186613),
 ((0.5, 'rbf', 3, 'scale'), 0.21615186615186613),
 ((0.5, 'rbf', 3, 'auto'), 0.21615186615186613),
 ((0.5, 'rbf', 5, 'scale'), 0.21615186615186613),
 ((0.5, 'rbf', 5, 'auto'), 0.21615186615186613),
 ((0.5, 'rbf', 10, 'scale'), 0.21615186615186613),
 ((0.5, 'rbf', 10, 'auto'), 0.21615186615186613),
 ((1, 'poly', 5, 'scale'), 0.22498621070049643),
 ((1, 'poly', 5, 'auto'), 0.22498621070049643),
 ((0.5, 'poly', 5, 'scale'), 0.25483544769259059),
 ((0.5, 'poly', 5, 'auto'), 0.25552031623460197),
 ((10, 'poly', 10, 'scale'), 0.45650395293252427),
 ((10, 'poly', 10, 'auto'), 0.45717962860819999),
 ((5, 'poly', 10, 'scale'), 0.47891156462585027),
 ((5, 'poly', 10, 'auto'), 0.47891156462585027),
 ((0.5, 'poly', 10, 'scale'), 0.48718054789483362),
 ((1, 'poly', 10, 'scale'), 0.48907887479316053),
 ((0.5, 'poly', 10, 'auto'), 0.49259974259974254),
 ((1, 'poly', 10, 'auto'), 0.49384077955506533)]
```

## comments:

We can see that svm with polynomial kernel outperform the best model with radial kernel in question 15 on cross validation dataset. And for polynomial kernel svm, a smaller cost value will give a better performance, and a degree of 3 is suitable for our dataset than other degree(1,5,10), so the complexity of our model is not high.
In terms of gamma in radial kernel, 'scale' gammas may perform better than 'auto' ones in general, but the difference is not so distinctive.