Perspective HW6

Yuxin Wu(yuxinwu@uchicago.edu)

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
import math
from sklearn.linear_model import LogisticRegression
from sklearn import svm
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
```

Non-linear Separation

1.

In [2]:

```
np.random.seed(985)
```

In [3]:

```
x1 = np.random.uniform(-1,1,100)
x2 = np.random.uniform(-1,1,100)
error = np.random.uniform(0,1,100)
```

In [4]:

```
y = x1 + x2 + x1 ** 2 + x2 ** 2 + error
```

In [5]:

```
y = (y - y.min()) / (y.max() - y.min())
```

In [6]:

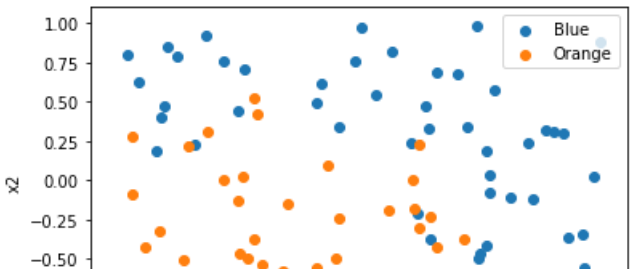
```
Blue = y >= 0.3
Orange = y < 0.3
```

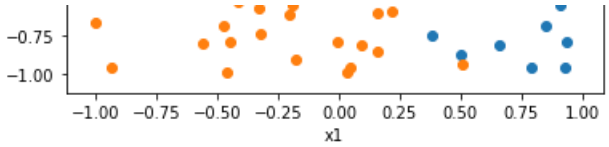
In [7]:

```
plt.scatter(x1[Blue], x2[Blue])
plt.scatter(x1[Orange], x2[Orange])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[7]:

```
<matplotlib.legend.Legend at 0x1ca56ebcc50>
```





In [8]:

```
model_rbf = SVC(kernel='rbf')
model_linear = SVC(kernel='linear')
```

In [9]:

```
x1 = pd.DataFrame(x1)
x2 = pd.DataFrame(x2)
X = pd.concat([x1, x2], axis=1 )
```

In [10]:

```
x_train, x_test, y_train, y_test = train_test_split(X, Blue, test_size=0.2)
```

In [11]:

```
model_rbf.fit(x_train, y_train)
model_linear.fit(x_train, y_train)
```

Out[11]:

```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

In [12]:

```
print('Training Accuracy:',model_rbf.score(x_train, y_train))
print('Testing Accuracy:', model_rbf.score(x_test, y_test))
```

```
Training Accuracy: 0.9
Testing Accuracy: 0.85
```

In [13]:

```
print('Training Accuracy:', model_linear.score(x_train, y_train))
print('Testing Accuracy:', model_linear.score(x_test, y_test))
```

```
Training Accuracy: 0.8125
Testing Accuracy: 0.8
```

As we can see from the result above. A support vector machine with a radial kernel will outperform a support vector classifier (a linear kernel) on the training data. And A support vector machine with a radial kernel will outperform a support vector classifier (a linear kernel) on the testing data.

SVM vs. logistic regression

In [14]:

```
np.random.seed(802)
```

2.

In [15]:

```
x1 = np.random.uniform(-1,1,500)
x2 = np.random.uniform(-1,1,500)
error = np.random.uniform(0,1,500)
```

In [16]:

```
y = x1 + x1 ** 2 + x2 ** 3 + error
y = (y - y.min()) / (y.max() - y.min())
Blue = y >= 0.4
Orange = y < 0.4
```

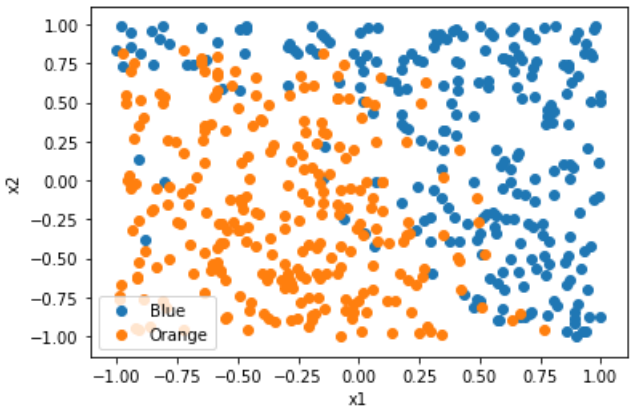
3.

In [17]:

```
plt.scatter(x1[Blue], x2[Blue])
plt.scatter(x1[Orange], x2[Orange])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[17]:

```
<matplotlib.legend.Legend at 0x1ca56fc7da0>
```



In [18]:

```
x1 = pd.DataFrame(x1)
x2 = pd.DataFrame(x2)
X_l = pd.concat([x1, x2], axis=1 )

x3 = pd.DataFrame(x1 * x2)
x1s = pd.DataFrame(x1 ** 2)
x2s = pd.DataFrame(x2 ** 2)
X_n = pd.concat([x1s, x2s, x3], axis=1 )
```

4.

In [19]:

```
model_logis = LogisticRegression()
```

In [20]:

```
model_logis.fit(X_l, Blue)
```

Out[20]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
```

```
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

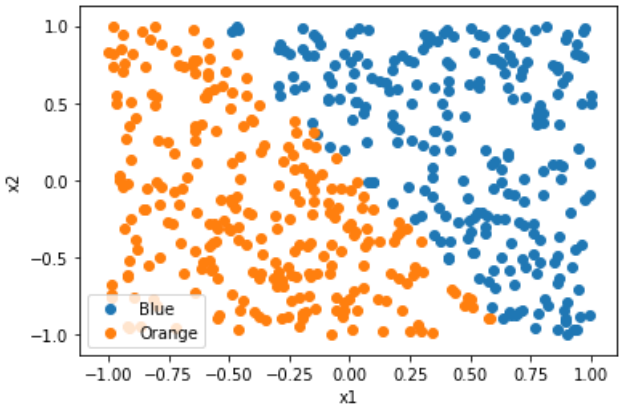
5.

In [21]:

```
pred = model_logis.predict(X_l)
plt.scatter(x1[pred], x2[pred])
plt.scatter(x1[~pred], x2[~pred])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[21]:

```
<matplotlib.legend.Legend at 0x1ca5700bf98>
```



6.

In [22]:

```
model_logism = LogisticRegression()
model_logism.fit(X_n, Blue)
```

Out[22]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

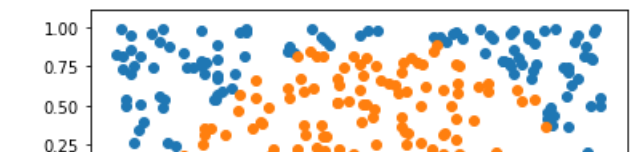
7.

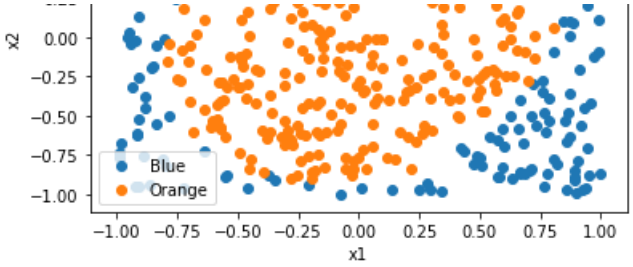
In [23]:

```
pred = model_logism.predict(X_n)
plt.scatter(x1[pred], x2[pred])
plt.scatter(x1[~pred], x2[~pred])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[23]:

```
<matplotlib.legend.Legend at 0x1ca570e1358>
```





8.

In [24]:

```
model_linear = svm.SVC(kernel='linear')
model_linear.fit(X_l, Blue)
```

Out[24]:

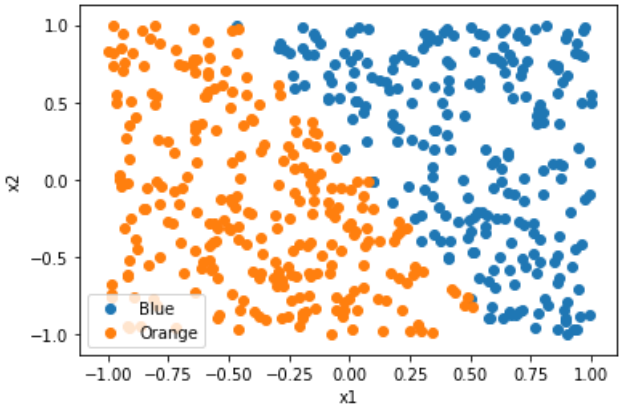
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='linear',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

In [25]:

```
pred_svc = model_linear.predict(X_l)
plt.scatter(x1[pred_svc], x2[pred_svc])
plt.scatter(x1[~pred_svc], x2[~pred_svc])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[25]:

```
<matplotlib.legend.Legend at 0x1ca570c7ef0>
```



9.

In [26]:

```
model_rbf = svm.SVC(kernel='rbf')
model_rbf.fit(X_l, Blue)
```

Out[26]:

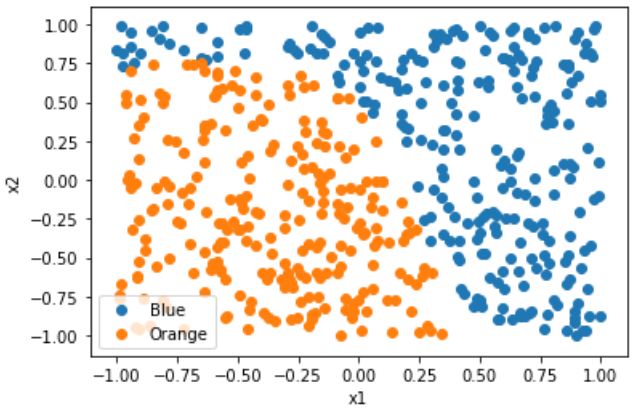
```
SVC(C=1.0, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

In [27]:

```
pred_svc = model_rbf.predict(X_1)
plt.scatter(x1[pred_svc], x2[pred_svc])
plt.scatter(x1[~pred_svc], x2[~pred_svc])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[27]:

```
<matplotlib.legend.Legend at 0x1ca571d4f28>
```



10.

In [28]:

```
print('Accuracy')
print('Logistic regression(linear):', model_logis.score(X_1, Blue))
print('Logistic regression(non-linear function):', model_logism.score(X_n, Blue))
print('SVM linear kernel:', model_linear.score(X_1, Blue))
print('SVM radial kernel:', model_rbf.score(X_1, Blue))
```

```
Accuracy
Logistic regression(linear): 0.858
Logistic regression(non-linear function): 0.666
SVM linear kernel: 0.866
SVM radial kernel: 0.91
```

As we can see from the results presented above, when we use a non-linear function of both X_1 and X_2 as predictors, the result is not so good, the accuracy is aroung 0.65-0.67. With a linear function of X1 and X2, the accuracy can go above 0.85. Among all the models, SVM with radial kernel and non-linear function of X1 and X2 performs the best (the accuracy is 0.91, and the shape of the predicted graph is very similar to our original graph). There is no absolute/big difference between Logistic regression and SVM with linear kernel. In terms of the tradeoff, SVM does not require the user to build a non-linear function previously, but a non-linear logistic requires the user to build a non-linear function previously. But SVM is much complicated than logistic regression (take more time and step to compute). So whether to choose logistic or SVM may vary from situation to situation.

Tuning cost

11.

In [169]:

```
np.random.seed(996)
```

In [170]:

```
x1 = np.random.uniform(-1,1,500)
x2 = np.random.uniform(-1,1,500)
error = np.random.uniform(0,0.4,500)
```

In [171]:

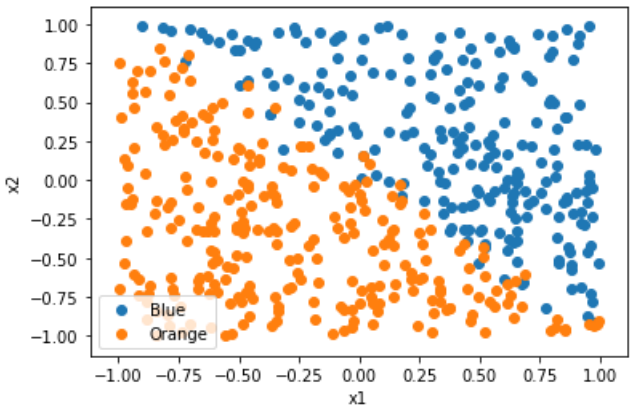
```
y = x1 + x2 + error
y = (y - y.min()) / (y.max() - y.min())
Blue = y >= 0.5
Orange = y < 0.5
```

In [172]:

```
plt.scatter(x1[Blue], x2[Blue])
plt.scatter(x1[Orange], x2[Orange])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Blue', 'Orange'])
```

Out[172]:

```
<matplotlib.legend.Legend at 0x1ca5a5def28>
```



12.

In [173]:

```
x1 = pd.DataFrame(x1)
x2 = pd.DataFrame(x2)
X = pd.concat([x1, x2], axis=1 )
```

In [174]:

```
x_train, x_test, y_train, y_test = train_test_split(X, Blue, test_size=0.2)
```

In [175]:

```
cost_list = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.5, 2, 2.5, 3, 4]
cross_error_list = []
train_error_list = []
for i in cost_list:
    model3 = SVC(C = i, kernel='linear')
    cross_error = 1 - np.mean(cross_val_score(model3, x_train, y_train, cv=10))
    cross_error_list.append(cross_error)
    model3.fit(x_train, y_train)
    train_error = 1 - model3.score(x_train, y_train)
    train_error_list.append(train_error)
```

In [176]:

```
print("Cross Error:")
print(cross_error_list)
print("Train Error:")
print(train_error_list)
```

```
Cross Error:
[0.07749999999999999, 0.04500000000000015, 0.04499999999999993, 0.04250000000000009,
0.04249999999999987, 0.04749999999999999, 0.050000000000000044, 0.04249999999999987,
0.04499999999999993, 0.04499999999999993, 0.04249999999999987, 0.04499999999999993,
0.04749999999999999, 0.04499999999999993, 0.04249999999999987, 0.044999999999999931
```

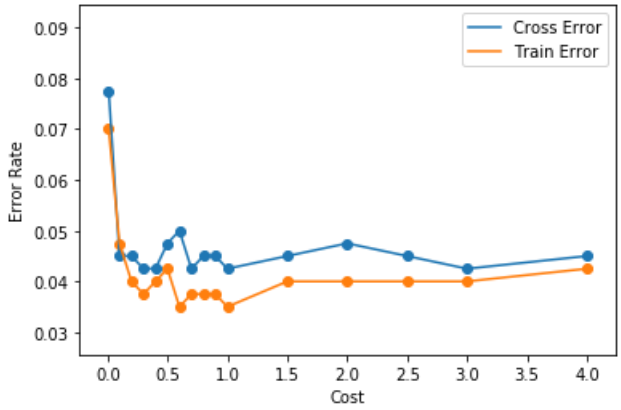
```
Train Error:
[0.06999999999999995, 0.04749999999999999, 0.040000000000000036, 0.03749999999999998,
0.040000000000000036, 0.04249999999999998, 0.03500000000000003, 0.03749999999999998,
0.03749999999999998, 0.03749999999999998, 0.03500000000000003, 0.040000000000000036,
0.040000000000000036, 0.040000000000000036, 0.040000000000000036, 0.04249999999999998]
```

In [177]:

```
plt.scatter(cost_list, cross_error_list)
plt.plot(cost_list, cross_error_list)
plt.scatter(cost_list, train_error_list)
plt.plot(cost_list, train_error_list)
plt.xlabel('Cost')
plt.ylabel('Error Rate')
plt.legend(['Cross Error', 'Train Error'])
```

Out[177]:

```
<matplotlib.legend.Legend at 0x1ca5a751f60>
```



As we can see from the results and graph above. The Cross Error reaches its peak at the cost value of 0.1. While for the train error, for the most cases, it is smaller than the cross error. The Train Error reaches its peak also at the cost value of 0.1. The error rate becomes relatively constant at the cost value of 2. As we can see from the graph the general trend for cross error and train error is very similar.

13.

In [178]:

```
cross_error_test_list = []
for i in cost_list:
    model3 = SVC(C = i, kernel='linear')
    model3.fit(x_train, y_train)
    cross_error_test = 1 - model3.score(x_test, y_test)
    cross_error_test_list.append(cross_error_test)
```

In [179]:

```
print(cross_error_test_list)
```

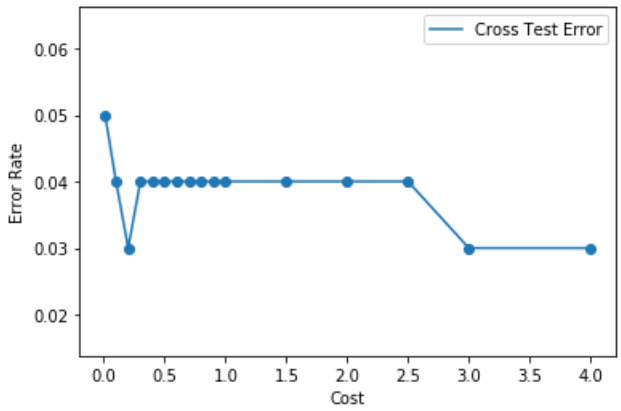
```
[0.050000000000000044, 0.040000000000000036, 0.030000000000000027, 0.040000000000000036,
0.040000000000000036, 0.040000000000000036, 0.040000000000000036, 0.040000000000000036,
0.040000000000000036, 0.040000000000000036, 0.040000000000000036, 0.040000000000000036,
0.040000000000000036, 0.040000000000000036, 0.030000000000000027, 0.030000000000000027]
```

In [180]:

```
plt.scatter(cost_list, cross_error_test_list)
plt.plot(cost_list, cross_error_test_list)
plt.xlabel('Cost')
plt.ylabel('Error Rate')
plt.legend(['Cross Test Error'])
```

Out[180]:

```
<matplotlib.legend.Legend at 0x1ca5a7517f0>
```

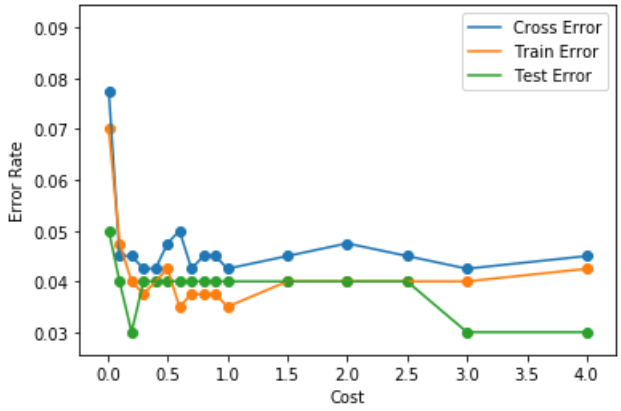


In [181]:

```
plt.scatter(cost_list, cross_error_list)
plt.plot(cost_list, cross_error_list)
plt.scatter(cost_list, train_error_list)
plt.plot(cost_list, train_error_list)
plt.scatter(cost_list, cross_error_test_list)
plt.plot(cost_list, cross_error_test_list)
plt.xlabel('Cost')
plt.ylabel('Error Rate')
plt.legend(['Cross Error', 'Train Error', "Test Error"])
```

Out[181]:

```
<matplotlib.legend.Legend at 0x1ca5a820ba8>
```



When cost value equals to 0.2, it yeilds the fewest error (= 0.03). The fewest test errors is smaller than the fewest train and fewest cross error

14.

As we can see from the results and graphs above, to reach a relatively accurate result, we do not need a very high cost value. A cost vale as small as 0.2 is sufficient enough for SVM to get an accurate result. The result proves that in the case of data that is just barely linearly separable, a support vector classifier with a small value of cost that misclassifies a couple of training observations may perform better on test data than one with a huge value of cost that does not misclassify any training observations.

In [182]:

```
gss_train = pd.read_csv("gss_train.csv")
gss_test = pd.read_csv("gss_test.csv")
y_train = gss_train["colrac"]
x_train = gss_train.drop('colrac', axis = 1)
```

15.

In [186]:

```
cost_list = [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
cross_error_list = []
for i in cost_list:
    model5 = svm.SVC(C = i, kernel='linear')
    cross_error = 1 - np.mean(cross_val_score(model5, x_train, y_train, cv=10))
    cross_error_list.append(cross_error)
```

In [197]:

```
print(cross_error_list)
print("Accuracy:", 1 - cross_error_list[2] )
```

```
[0.20526482858697626, 0.20323780155994908, 0.20188191547251955, 0.20593596952657356,
0.20323326682387077, 0.20593143479049525, 0.20660711046617097, 0.20660711046617097,
0.20660711046617086, 0.20593143479049525, 0.20593143479049525]
Accuracy: 0.7981180845274805
```

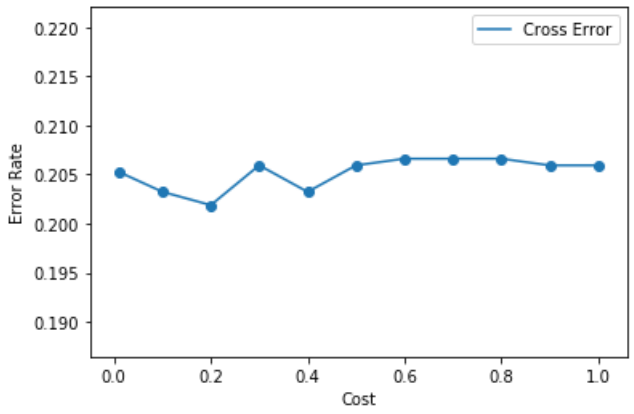
In [187]:

```
plt.scatter(cost_list, cross_error_list)
plt.plot(cost_list, cross_error_list)

plt.xlabel('Cost')
plt.ylabel('Error Rate')
plt.legend(['Cross Error'])
```

Out[187]:

```
<matplotlib.legend.Legend at 0x1ca5b8c17b8>
```



As we can see from the graph above, when cost vlaue equlas to 0.2, we will have the lowest error rate, which suggests the cost value equls to 0.2 is our optimal choice. But as we can see, there is actually no big difference among different cost values, the error rate is within the range of 0.20 and 0.210. The linear SVM works pretty well.

16.

In [190]:

```
#rbf
model = SVC(kernel='rbf')
fea = {'C': [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1],
       'degree':[2,3,4],
       'gamma': ['scale', 'auto']}

cvs = GridSearchCV(model, fea, cv=10)
cvs.fit(x_train, y_train)
print(cvs.best_estimator_)
print(cvs.best_score_)
```

```
SVC(C=1, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=2, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
0.7494830400870669
```

In [191]:

```
#poly
model = SVC(kernel='poly')
fea = {'C': [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1],
       'degree':[2,3,4],
       'gamma': ['scale', 'auto']}

cvs = GridSearchCV(model, fea, cv=10)
cvs.fit(x_train, y_train)
print(cvs.best_estimator_)
print(cvs.best_score_)
```

```
SVC(C=0.01, break_ties=False, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=2, gamma='auto', kernel='poly',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
0.7927036096499183
```

As we can see from the results above. Linear SVM performs the best when the cost value equals to 0.2, the accuracy is 0.7981180845274805. SVM with radial performs the best when the cost value equals to 1, degree = 2, and gamma = 'scale', the accuracy is 0.7494830400870669. SVM with poly performs the best when cost value = 0.01, degree = 2, and gamma = 'auto, the accuracy is 0.7927036096499183. Generally speaking, all three models perform pretty well. The accuracy are pretty similar. But as we can see the linear SVM has the highest accuracy and it is eaiser to train. Thus, I would still choose Linear SVM as the best model among these three models.