In [13]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
import math
import random
from sklearn import svm
from sklearn.svm import SVC,LinearSVC
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.linear_model import LogisticRegression
import warnings
warnings.filterwarnings('ignore')
```

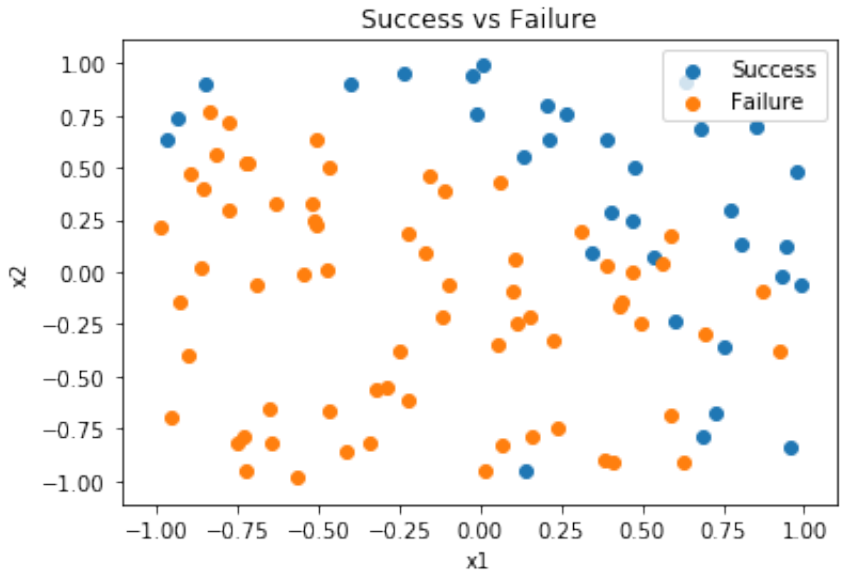
Non-linear separation

1

1.(15 points) Generate a simulated two-class data set with 100 observations and two features in which there is a visible (clear) but still non-linear separation between the two classes. Show that in this setting, a support vector machine with a radial kernel will outperform a support vector classifier (a linear kernel) on the training data. Which technique performs best on the test data? Make plots and report training and test error rates in order to support your conclusions.

In [7]:
```
np.random.seed(8888)
x1 = np.random.uniform(-1,1,100)
x2 = np.random.uniform(-1,1,100)
err = np.random.normal(0,0.5,100)
y = x1 + x1 ** 2 + x2 + x2**2 + err
prob = (y-np.min(y))/(np.max(y)-np.min(y))
success = prob > 0.5
failure = prob <= 0.5
plt.scatter(x1[success], x2[success])
plt.scatter(x1[failure], x2[failure])
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Success vs Failure');
plt.legend(['Success', 'Failure'])
```

Out[7]: <matplotlib.legend.Legend at 0x1a1e589ac8>



In [16]:
```
x = pd.DataFrame({'x1': x1, 'x2': x2})
x_train, x_test, y_train, y_test = train_test_split(x, success, test_s
ize=0.2, random_state=8888)
svm_radial = SVC().fit(x_train, y_train)
svm_linear = SVC(kernel='linear').fit(x_train, y_train)
```

In [17]:
```
print('Training error of radial kernel:', 1-svm_radial.score(x_train,
y_train))
print('Test error of radial kernel:', 1-svm_radial.score(x_test, y_tes
t))
```

```
Training error of radial kernel: 0.125
Test error of radial kernel: 0.09999999999999998
```

In [18]:
```
print('Training error of linear kernel:', 1-svm_linear.score(x_train, y_train))
print('Test error of linear kernel:', 1-svm_linear.score(x_test, y_test))
```

```
Training error of linear kernel: 0.16249999999999998
Test error of linear kernel: 0.19999999999999996
```

From above, we see that in terms of test data, radial kernel performs better than linear kernel;in terms of train data, linear kernel performs better than radial kernel.

SVM vs. logistic regression

2

2.(5 points) Generate a data set with n = 500 and p = 2, such that the observations belong to two classes with some overlapping, non-linear boundary between them.

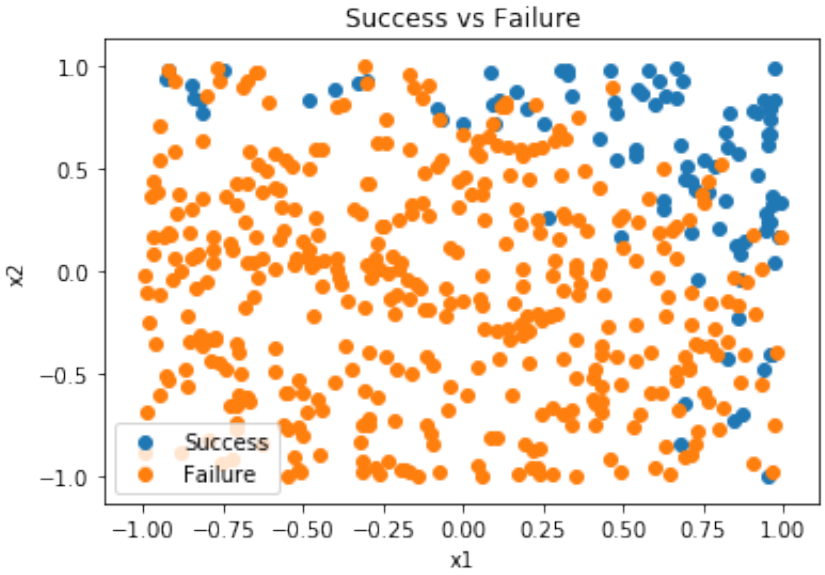
In [21]:
```
x3 = np.random.uniform(-1,1,500)
x4 = np.random.uniform(-1,1,500)
err2 = np.random.normal(0,0.5,500)

y2 = x3 + x3 ** 2 + x4 + x4**2 + err2
prob2 = (y2-np.min(y2))/(np.max(y2)-np.min(y2))
success2 = prob2 > 0.5
failure2 = prob2 <= 0.5
```

3

3.(5 points) Plot the observations with colors according to their class labels (y). Your plot should display X1 on the x-axis and X2 on the y-axis.

In [22]:
```
plt.scatter(x3[success2], x4[success2])
plt.scatter(x3[failure2], x4[failure2])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'])
plt.title('Success vs Failure');
```



4

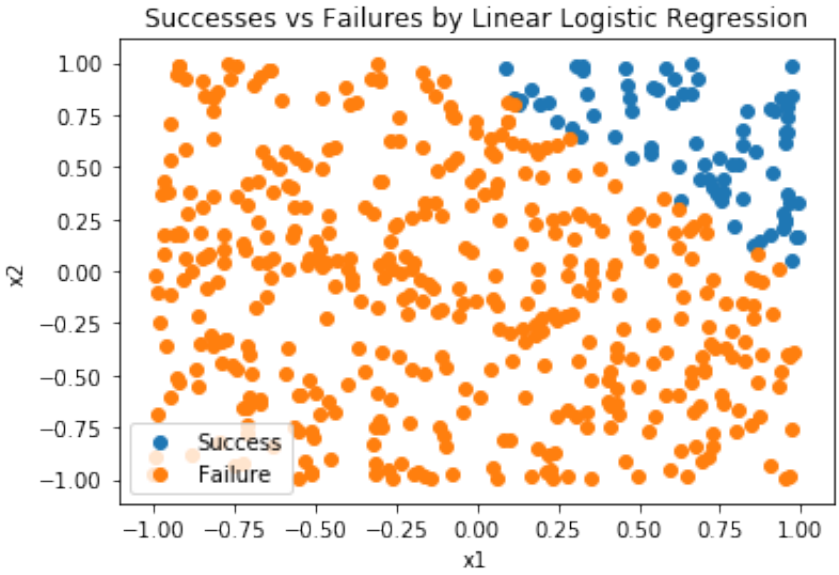
4.(5 points) Fit a logistic regression model to the data, using X1 and X2 as predictors.

In [35]:
```
x = pd.DataFrame({'x1': x3, 'x2': x4})
lr = LogisticRegression().fit(x, success2)
```

5

(5 points) Obtain a predicted class label for each observation based on the logistic model previously fit. Plot the observations, colored according to the predicted class labels (the predicted decision boundary should look linear).

In [36]:
```
lr_pred = lr.predict(x)
plt.scatter(x3[lr_pred], x4[lr_pred])
plt.scatter(x3[~lr_pred], x4[~lr_pred])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'])
plt.title('Successes vs Failures by Linear Logistic Regression');
```



6

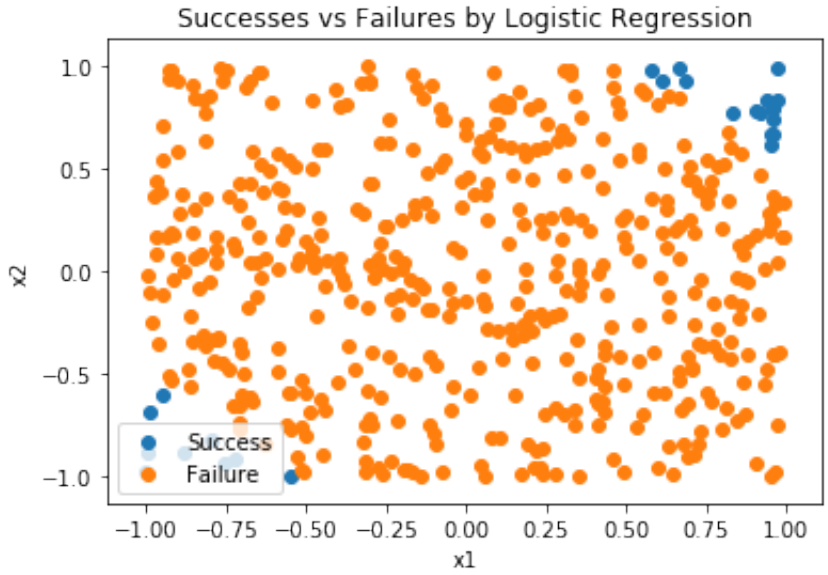
6.(5 points) Now fit a logistic regression model to the data, but this time using some non-linear function of both X1 and X2 as predictors (e.g. X12, X1 × X2, log(X2), and so on).

In [37]:
```
x_square = pd.DataFrame({'x1': x3**2, 'x2': x4**2, 'x1*x2': x3 * x4})
lr_square = LogisticRegression().fit(x_square, success2)
```

7

7.(5 points) Now, obtain a predicted class label for each observation based on the fitted model with non-linear transformations of the X features in the previous question. Plot the observations, colored according to the new predicted class labels from the non-linear model (the decision boundary should now be obviously non-linear). If it is not, then repeat earlier steps until you come up with an example in which the predicted class labels and the resultant decision boundary are clearly non-linear.

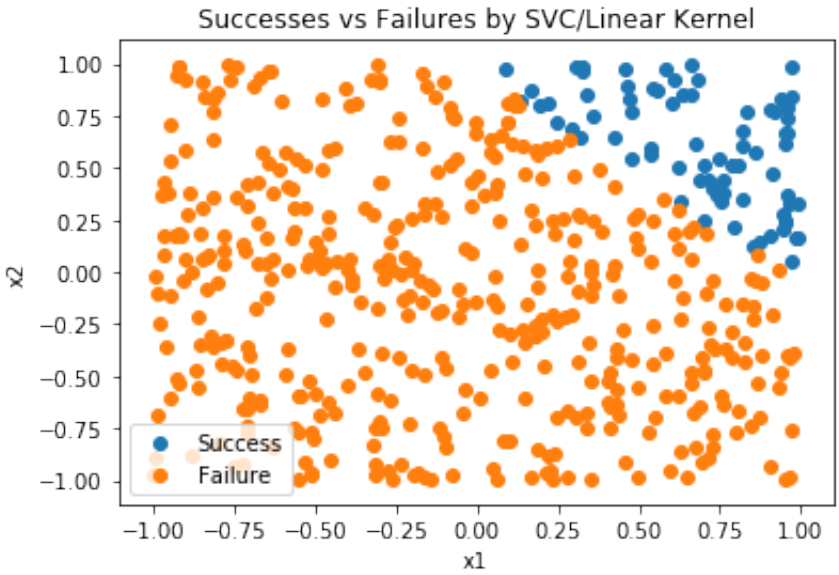
In [38]:
```
lrs_pred = lr_square.predict(x_square)
plt.scatter(x3[lrs_pred], x4[lrs_pred])
plt.scatter(x3[~lrs_pred], x4[~lrs_pred])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'])
plt.title('Successes vs Failures by Logistic Regression');
```



8

8.(5 points) Now, fit a support vector classifier (linear kernel) to the data with original X1 and X2 as predictors. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

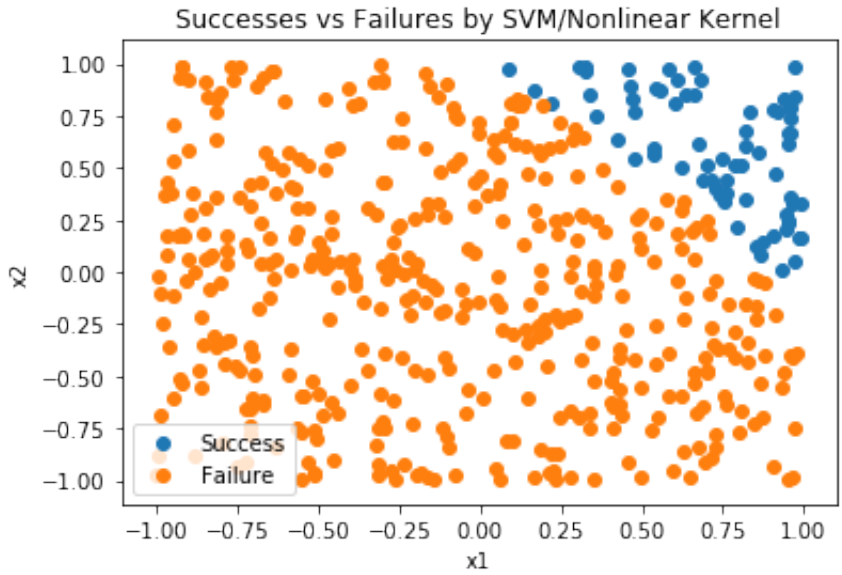
In [43]:
```
svc = SVC(kernel='linear').fit(x, success2)
svc_pred = svc.predict(x)
plt.scatter(x3[svc_pred], x4[svc_pred])
plt.scatter(x3[~svc_pred], x4[~svc_pred])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'])
plt.title('Successes vs Failures by SVC/Linear Kernel');
```



9

9.(5 points) Fit a SVM using a non-linear kernel to the data. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

In [44]:
```
svm = SVC(gamma='scale').fit(x, success2)
svm_pred = svm.predict(x)
plt.scatter(x3[svm_pred], x4[svm_pred])
plt.scatter(x3[~svm_pred], x4[~svm_pred])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'])
plt.title('Successes vs Failures by SVM/Nonlinear Kernel');
```



10

10.(5 points) Discuss your results and specifically the tradeoffs between estimating non-linear decision boundaries using these two different approaches.

In [45]:
```
print('Accuracy for Logistic regression with x1, x2:', lr.score(x, suc
cess2))
print('Accuracy for Logistic regression with non-linear function:', lr
s.score(x_square, success2))
print('Accuracy for SVM linear kernel:', svc.score(x, success2))
print('Accuracy for SVM radial kernel:', svm.score(x, success2))
```

```
Accuracy for Logistic regression with x1, x2: 0.906
Accuracy for Logistic regression with non-linear function: 0.822
Accuracy for SVM linear kernel: 0.904
Accuracy for SVM radial kernel: 0.906
```

From the above measure of accuracy, we see that linear logistic regression and SVM radial kernel perform the best, while non-linear logistic regression performs the worst.As for the tradeoffs when estimating non-linear boundary, the logistic regression are more computationally efficient than SVM, but SVM radial kernel can give a more accurate estimate of the forms of predictors. So there is a tradeoff between computation and accuracy of the model.

Tuning cost

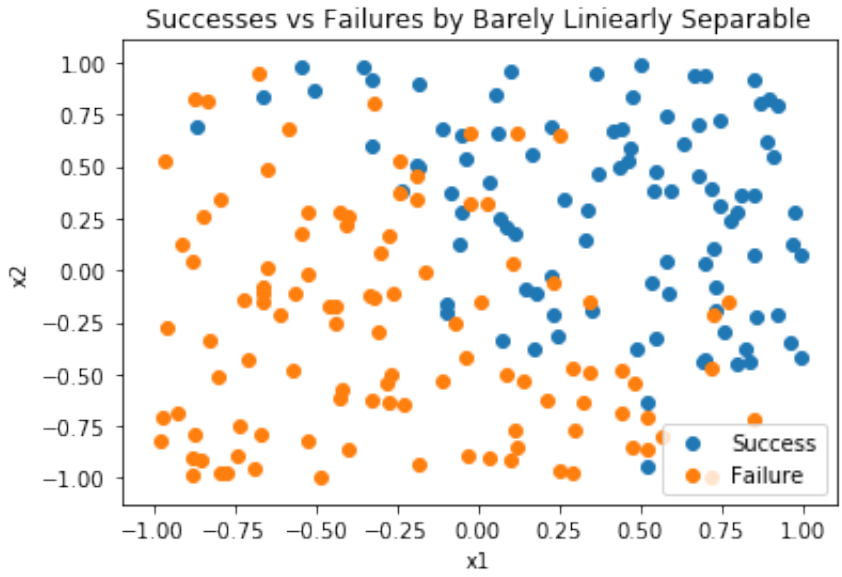
11

11.(5 points) Generate two-class data with p = 2 in such a way that the classes are just barely linearly separable.

In [46]:
```
x1 = np.random.uniform(-1,1,200)
x2 = np.random.uniform(-1,1,200)
err = np.random.normal(0,0.5,200)
y = x1 + x2 + err

prob = (y-np.min(y))/(np.max(y)-np.min(y))
success = prob > 0.5
failure = prob <= 0.5

plt.scatter(x1[success], x2[success])
plt.scatter(x1[failure], x2[failure])
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(['Success', 'Failure'])
plt.title('Successes vs Failures by Barely Liniearly Separable');
```



12

12.(5 points) Compute the cross-validation error rates for support vector classifiers with a range of cost values. How many training errors are made for each value of cost considered, and how does this relate to the cross-validation errors obtained?

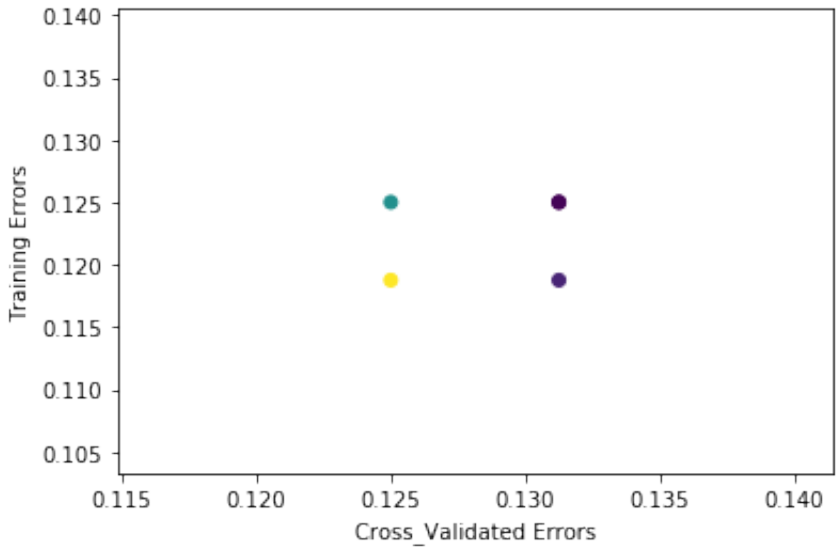
In [60]:
```
x = pd.DataFrame({'x1': x1, 'x2': x2})
x_train, x_test, y_train, y_test = train_test_split(x, success, test_s
ize=0.2, random_state=777)
costs = [0.001, 0.01, 0.1, 0.5, 1]
training_err = []
cv_err = []
for c in costs:
    model = SVC(C=c, kernel='linear')
    cv = 1 - np.mean(cross_val_score(model, x_train, y_train, cv=10, s
coring='accuracy'))
    cv_err.append(cv)
    train = 1 - model.fit(x_train, y_train).score(x_train, y_train)
    training_err.append(train)

pd.DataFrame({'Cost': costs, 'CV Error': cv_err, 'Training Error': tra
ining_err})
```

Out[60]:

	Cost	CV Error	Training Error
0	0.001	0.13125	0.12500
1	0.010	0.13125	0.12500
2	0.100	0.13125	0.11875
3	0.500	0.12500	0.12500
4	1.000	0.12500	0.11875

In [64]:
```
plt.scatter(x = cv_err, y = training_err, c=[0.001, 0.01, 0.1, 0.5, 1]
)
plt.xlabel('Cross_Validated Errors')
plt.ylabel('Training Errors')
plt.show()
```



From the above, we see that when C increases from 0.001 to 0.1, the cross validation error and training error both decrease. When C is larger than 0.5, the cross validation error stays at 0.125, while the training error decreases.

13

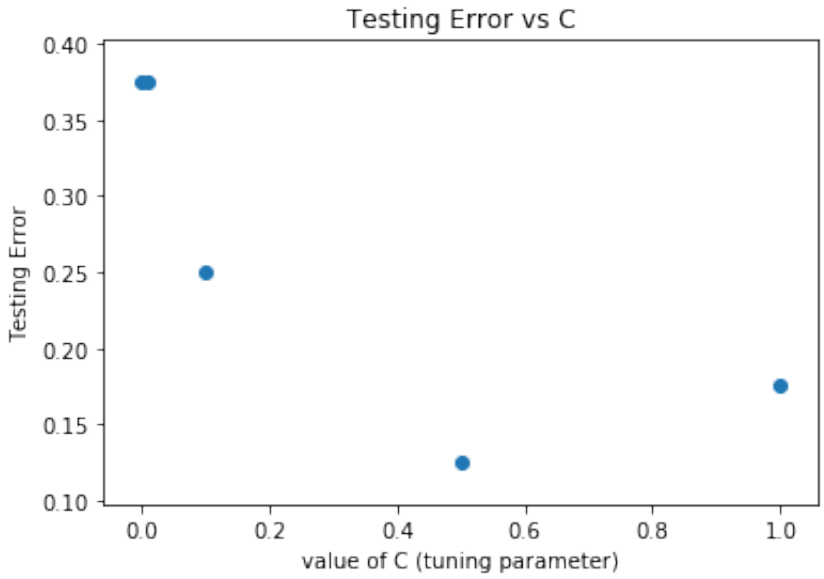
13.(5 points) Generate an appropriate test data set, and compute the test errors corresponding to each of the values of cost considered. Which value of cost leads to the fewest test errors, and how does this compare to the values of cost that yield the fewest training errors and the fewest cross-validation errors?

In [65]:
```
test_err = []
for cost in costs:
    model = SVC(C=cost, kernel='linear')
    test = 1 - model.fit(x_test, y_test).score(x_test, y_test)
    test_err.append(test)
pd.DataFrame({'Cost': costs, 'Test Error': test_err})
```

Out[65]:

	Cost	Test Error
0	0.001	0.375
1	0.010	0.375
2	0.100	0.250
3	0.500	0.125
4	1.000	0.175

In [67]:
```
plt.scatter([0.001, 0.01, 0.1, 0.5, 1], test_err)
plt.ylabel('Testing Error')
plt.xlabel('value of C (tuning parameter)')
plt.title('Testing Error vs C');
```



From the above, we see that the lowest test error occurs when C = 0.5

14

14.(5 points) Discuss your results.

From the above results, we can see that the data set we generate is only barely linear, as a result, when the cost is very low (which is 0.001 in our case), there will be a high error rate. We don't need a high C/cost tuning parameter to severely punish violations to the margin. Instead, a low C would be enough, as suggested by the lowest test error graph.

15

15.(5 points) Fit a support vector classifier to predict colrac as a function of all available predictors, using 10-fold cross-validation to find an optimal value for cost. Report the CV errors associated with different values of cost, and discuss your results.

In [3]:
```
gss_train = pd.read_csv("gss_train.csv")
gss_test = pd.read_csv("gss_test.csv")
x_train = gss_train.drop('colrac',axis=1)
y_train = gss_train['colrac']
```

In [5]:
```
costs = [0.001, 0.01, 0.1, 0.5, 1]
cv_error = []
for c in costs:
    model = SVC(C=c, kernel='linear')
    cv = 1 - np.mean(cross_val_score(model, x_train, y_train, cv=10, s
coring='accuracy'))
    cv_error.append((c, cv))
cv_error
```

Out[5]:
```
[(0.001, 0.2396741283126711),
 (0.01, 0.20594904930954794),
 (0.1, 0.2039447576600022),
 (0.5, 0.20325088134292357),
 (1, 0.20595818047879422)]
```

From the above, we can see that when C = 0.5, we have the lowest cv error.

16

16.(15 points) Repeat the previous question, but this time using SVMs with radial and polynomial basis kernels, with different values for gamma and degree and cost. Present and discuss your results (e.g., fit, compare kernels, cost, substantive conclusions across fits, etc.).

In [14]:
```
parameters = {'kernel':('poly', 'rbf'), 'C':[0.1, 0.5, 1, 5, 10], 'gam
ma':('scale','auto')}
svc = svm.SVC()
model = GridSearchCV(svc, parameters, cv=10)
model.fit(x_train, y_train)
```

Out[14]:
```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None,
coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='auto_deprecated', kernel='rbf', ma
x_iter=-1,
                           probability=False, random_state=None, shr
inking=True,
                           tol=0.001, verbose=False),
             iid='warn', n_jobs=None,
             param_grid={'C': [0.1, 0.5, 1, 5, 10], 'gamma': ('scale
', 'auto'),
                         'kernel': ('poly', 'rbf')},
             pre_dispatch='2*n_jobs', refit=True, return_train_score
=False,
             scoring=None, verbose=0)
```

As the results show, we see that the best model selected has C = 1, gamma = auto (which means gamma = 1 / number of features). Radial kerneling is better than polynomial kerneling (at degree =3).