Homework 6

March 8, 2020

0.1 Homework 6

[1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from pylab import rcParams
import seaborn as sns
import pandas as pd
import sklearn
import warnings
import itertools
import sklearn.metrics
warnings.filterwarnings('ignore')
```

0.1.1 Conceptual Exercises-Non-linear separation

Q1. 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.

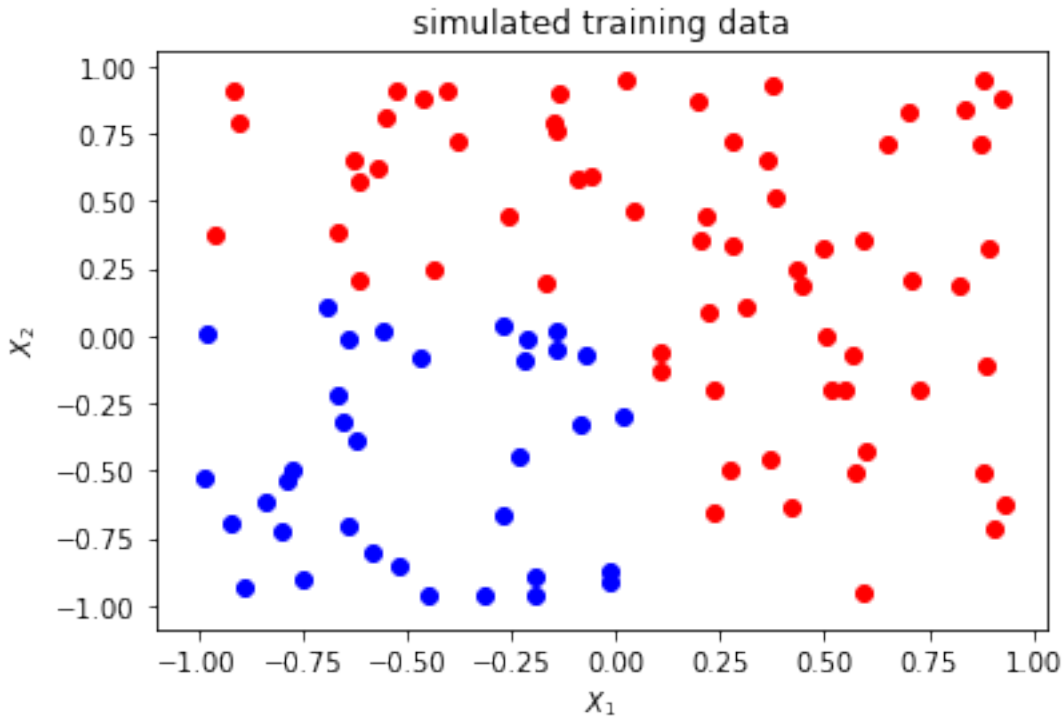
[2]:
```
## training data generation
my_train_df = pd.DataFrame({'X1':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=100),
                            'X2':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=100)})
my_train_df['Y'] = my_train_df['X1'] + my_train_df['X1']**2 + \
                   my_train_df['X2'] + my_train_df['X2']**2
my_train_df['Y'] = my_train_df['Y'].apply(lambda x: 0 if x <= 0 else 1)
my_train_df
```

[2]:

	X1	X2	Y
0	0.361493	0.652931	1
1	0.447484	0.186643	1
2	-0.637110	-0.014590	0
3	-0.913650	0.910230	1

```
4   0.589967 -0.950411  1
..       …       … ..
95 -0.140144 -0.048827  0
96 -0.978061  0.010925  0
97  0.370823 -0.455146  1
98 -0.229718 -0.447561  0
99 -0.668518  0.383990  1

[100 rows x 3 columns]
```

[13]:
```
# visualize the data
plt.scatter(my_train_df.loc[my_train_df['Y']==1]['X1'],
            my_train_df.loc[my_train_df['Y']==1]['X2'], c='r')
plt.scatter(my_train_df.loc[my_train_df['Y']==0]['X1'],
            my_train_df.loc[my_train_df['Y']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('simulated training data')
plt.show()
```



We could see the non-linear separation between the two classes.

[4]:
```
from sklearn.svm import SVC

linearSVM = SVC(kernel='linear')
radialSVM = SVC(kernel='rbf')
linearSVM.fit(my_train_df[['X1','X2']], my_train_df['Y'])
radialSVM.fit(my_train_df[['X1','X2']], my_train_df['Y'])
print("training data error rate (linear): {}".format(\
     1-linearSVM.score(my_train_df[['X1','X2']], my_train_df['Y'])))
print("training data error rate (radial): {}".format(\
     1-radialSVM.score(my_train_df[['X1','X2']], my_train_df['Y'])))
```

```
training data error rate (linear): 0.08999999999999997
training data error rate (radial): 0.010000000000000009
```

[5]:
```
## test data generation, having the same distribution
my_test_df = pd.DataFrame({'X1':sp.stats.uniform.rvs(loc=-1, scale=2, size=100),
                           'X2':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=100)})
my_test_df['Y'] = my_test_df['X1'] + my_test_df['X1']**2 + \
                  my_test_df['X2'] + my_test_df['X2']**2
my_test_df['Y'] = my_test_df['Y'].apply(lambda x: 0 if x <= 0 else 1)
my_test_df
```

[5]:

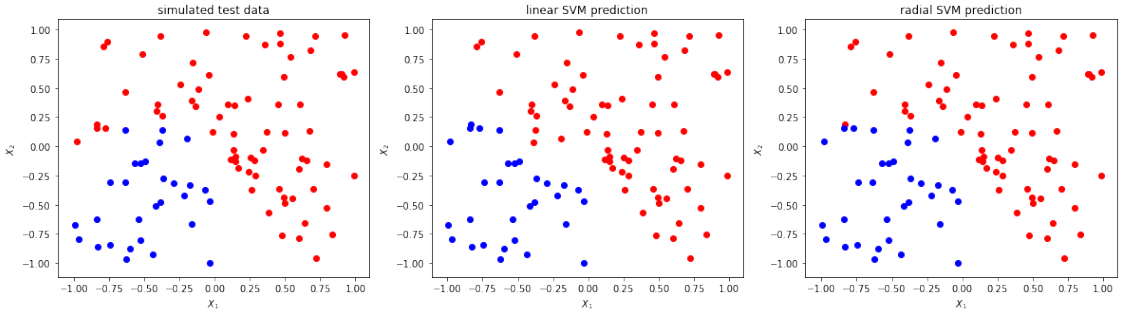
	X1	X2	Y
0	-0.136654	0.347016	1
1	-0.758976	0.896670	1
2	0.495186	0.115983	1
3	0.464462	0.877472	1
4	-0.063417	0.978602	1
..	…	…	..
95	-0.830434	-0.860245	0
96	-0.743988	-0.842591	0
97	0.253029	-0.094075	1
98	-0.159966	-0.669213	0
99	-0.115834	0.487630	1

[100 rows x 3 columns]

[8]:
```
print("test data error rate (linear): {}".format(\
     1-linearSVM.score(my_test_df[['X1','X2']], my_test_df['Y'])))
print("test data error rate (radial): {}".format(\
     1-radialSVM.score(my_test_df[['X1','X2']], my_test_df['Y'])))
```

```
test data error rate (linear): 0.06999999999999995
test data error rate (radial): 0.030000000000000027
```

[16]:
```
# visualize predicted data
fig, axes = plt.subplots(1,3,figsize=[20,5])
# real test data
axes[0].scatter(my_test_df.loc[my_test_df['Y']==1]['X1'],
            my_test_df.loc[my_test_df['Y']==1]['X2'], c='r')
axes[0].scatter(my_test_df.loc[my_test_df['Y']==0]['X1'],
            my_test_df.loc[my_test_df['Y']==0]['X2'], c='b')
axes[0].set_xlabel('$X_{1}$')
axes[0].set_ylabel('$X_{2}$')
axes[0].set_title('simulated test data')

# linear SVM prediction
linear_df = pd.DataFrame({'X1':my_test_df['X1'],
                          'X2':my_test_df['X2'],
'Y':linearSVM.predict(my_test_df[['X1','X2']])})
axes[1].scatter(linear_df.loc[linear_df['Y']==1]['X1'],
            linear_df.loc[linear_df['Y']==1]['X2'], c='r')
axes[1].scatter(linear_df.loc[linear_df['Y']==0]['X1'],
            linear_df.loc[linear_df['Y']==0]['X2'], c='b')
axes[1].set_xlabel('$X_{1}$')
axes[1].set_ylabel('$X_{2}$')
axes[1].set_title('linear SVM prediction')

# radial SVM prediction
radial_df = pd.DataFrame({'X1':my_test_df['X1'],
                          'X2':my_test_df['X2'],
'Y':radialSVM.predict(my_test_df[['X1','X2']])})
axes[2].scatter(radial_df.loc[radial_df['Y']==1]['X1'],
            radial_df.loc[radial_df['Y']==1]['X2'], c='r')
axes[2].scatter(radial_df.loc[radial_df['Y']==0]['X1'],
            radial_df.loc[radial_df['Y']==0]['X2'], c='b')
axes[2].set_xlabel('$X_{1}$')
axes[2].set_ylabel('$X_{2}$')
axes[2].set_title('radial SVM prediction')
plt.show()
```



Based on the plots and the error rates, we can see that within the test set, the radial SVM still outperfoprms the linear SVM.

0.1.2 Conceptual exercises-SVM vs. logistic regression

We have seen that we can fit an SVM with a non-linear kernel in order to perform classification using a non-linear decision boundary. We will now see that we can also obtain a non-linear decision boundary by performing logistic regression using non-linear transformations of the features. Your goal here is to compare different approaches to estimating non-linear decision boundaries, and thus assess the benefits and drawbacks of each.

Q2. 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.

[21]:
```
# data generation
my_second_df = pd.DataFrame({'X1':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=500),
                             'X2':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=500)})
my_second_df['Y'] = my_second_df['X1'] + my_second_df['X1']**2 + \
                    my_second_df['X2'] + my_second_df['X2']**2 + \
                    sp.stats.norm.rvs(loc=0, scale=0.5, size=500)
my_second_df['Y'] = my_second_df['Y'].apply(lambda x: 0 if x <= 0 else 1)
my_second_df
```

[21]:

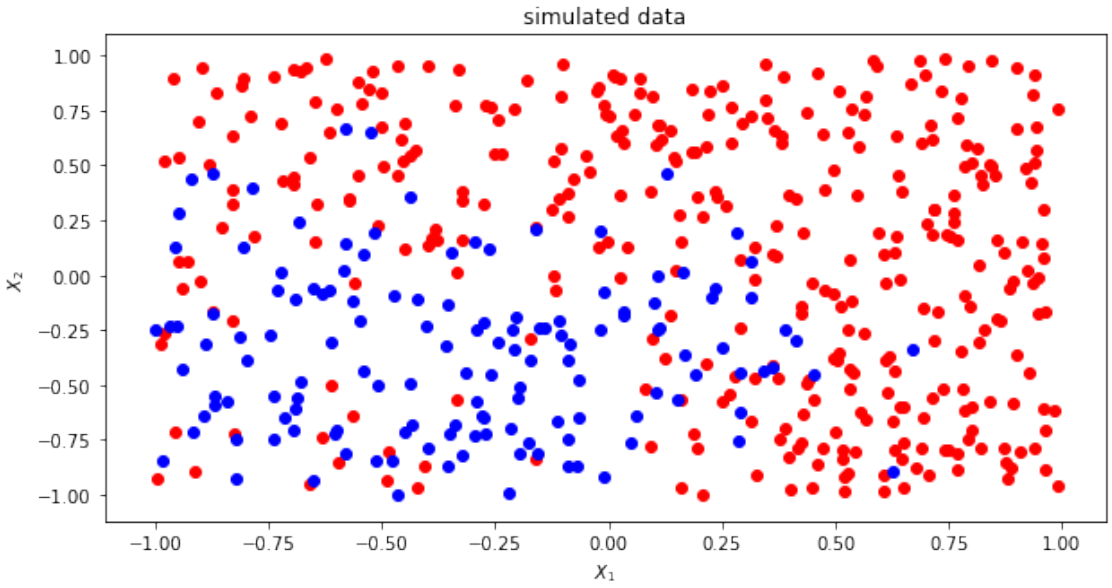
	X1	X2	Y
0	0.441414	-0.479696	1
1	0.541640	-0.806567	1
2	-0.999153	-0.249134	0
3	0.921160	0.490487	1
4	0.386157	0.903619	1
..	…	…	..
495	0.874506	-0.853353	1
496	-0.938042	-0.060140	1
497	-0.109956	0.348007	1
498	-0.320854	0.384140	1
499	0.035108	0.600683	1

[500 rows x 3 columns]

Q3. Plot the observations with colors according to their class labels (y). Your plot should display X_1 on the x-axis and X_2 on the y-axis.

[22]:
```
plt.rcParams['figure.figsize']=[10,5]
plt.scatter(my_second_df.loc[my_second_df['Y']==1]['X1'],
```

```
            my_second_df.loc[my_second_df['Y']==1]['X2'], c='r')
plt.scatter(my_second_df.loc[my_second_df['Y']==0]['X1'],
            my_second_df.loc[my_second_df['Y']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('simulated data')
plt.show()
```



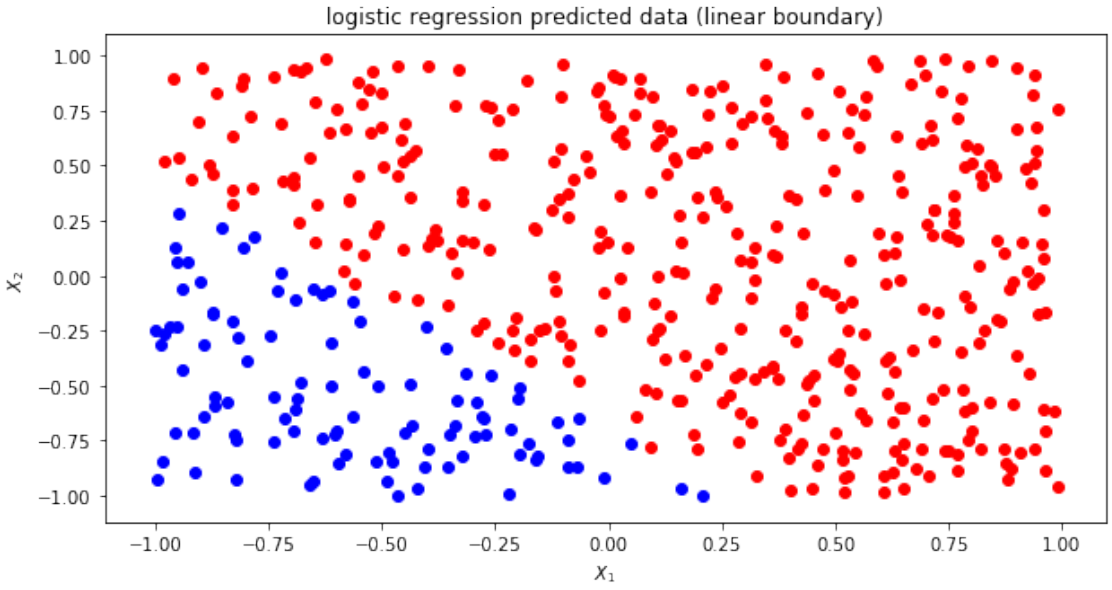
Q4. Fit a logistic regression model to the data, using X_1 and X_2 as predictors

[31]:
```
from sklearn.linear_model import LogisticRegression
LR_clf = LogisticRegression()
LR_clf.fit(my_second_df[['X1','X2']], my_second_df['Y'])
```

[31]:
```
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)
```

Q5. 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).

[32]:
```
LR_df = pd.DataFrame({'X1':my_second_df['X1'],
                      'X2':my_second_df['X2'],
'Y':LR_clf.predict(my_second_df[['X1','X2']])})
plt.rcParams['figure.figsize']=[10,5]
plt.scatter(LR_df.loc[LR_df['Y']==1]['X1'],
            LR_df.loc[LR_df['Y']==1]['X2'], c='r')
plt.scatter(LR_df.loc[LR_df['Y']==0]['X1'],
            LR_df.loc[LR_df['Y']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('logistic regression predicted data (linear boundary)')
plt.show()
```



Q6. Now fit a logistic regression model to the data, but this time using some non-linear function of both X_1 and X_2 as predictors.

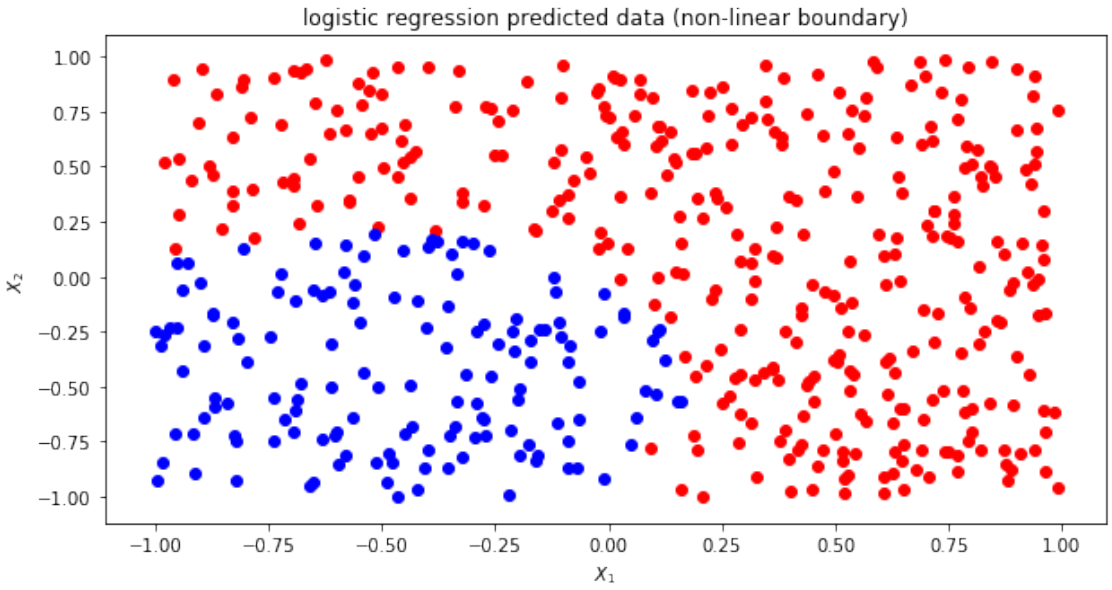
[37]:
```
# producing nonlinear functions
my_second_df['product'] = my_second_df['X1']*my_second_df['X2']
my_second_df['X1_2'] = my_second_df['X1']**2
my_second_df['X2_2'] = my_second_df['X2']**2
LR_clf.fit(my_second_df[['product','X1_2','X2_2','X1','X2']],
           my_second_df['Y'])
```

[37]:
```
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)
```

Q7. 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).

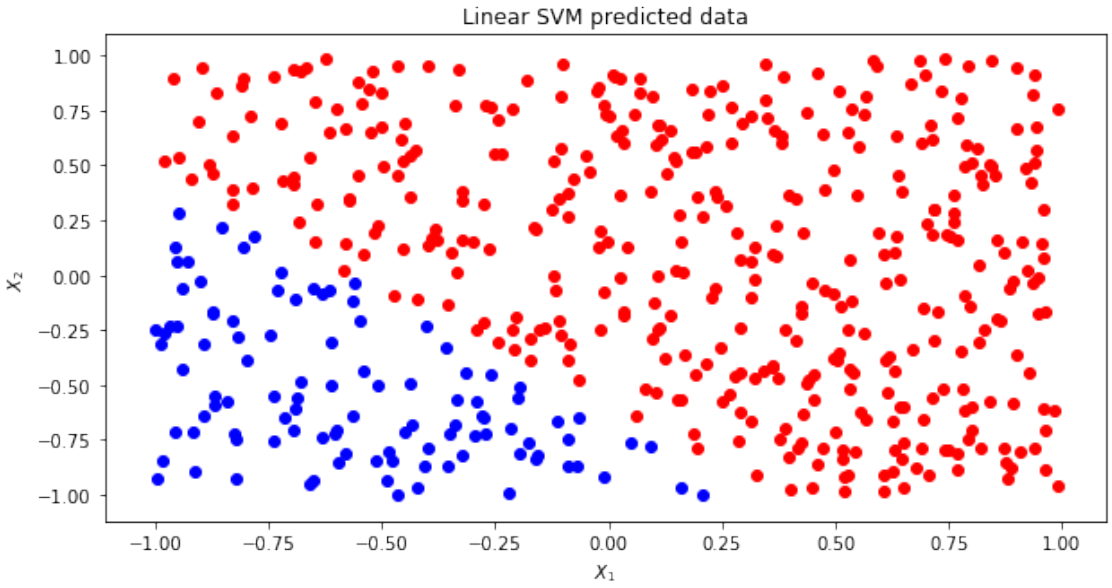
[38]:
```
LR_df['Y*'] =  LR_clf.predict(my_second_df[['product','X1_2','X2_2','X1','X2']])
plt.rcParams['figure.figsize']=[10,5]
plt.scatter(LR_df.loc[LR_df['Y*']==1]['X1'],
            LR_df.loc[LR_df['Y*']==1]['X2'], c='r')
plt.scatter(LR_df.loc[LR_df['Y*']==0]['X1'],
            LR_df.loc[LR_df['Y*']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('logistic regression predicted data (non-linear boundary)')
plt.show()
```



Q8. Now, fit a support vector classifier (linear kernel) to the data with original X_1 and X_2 as predictors. Obtain a class prediction for each observation. Plot the observations, colored according to the predicted class labels.

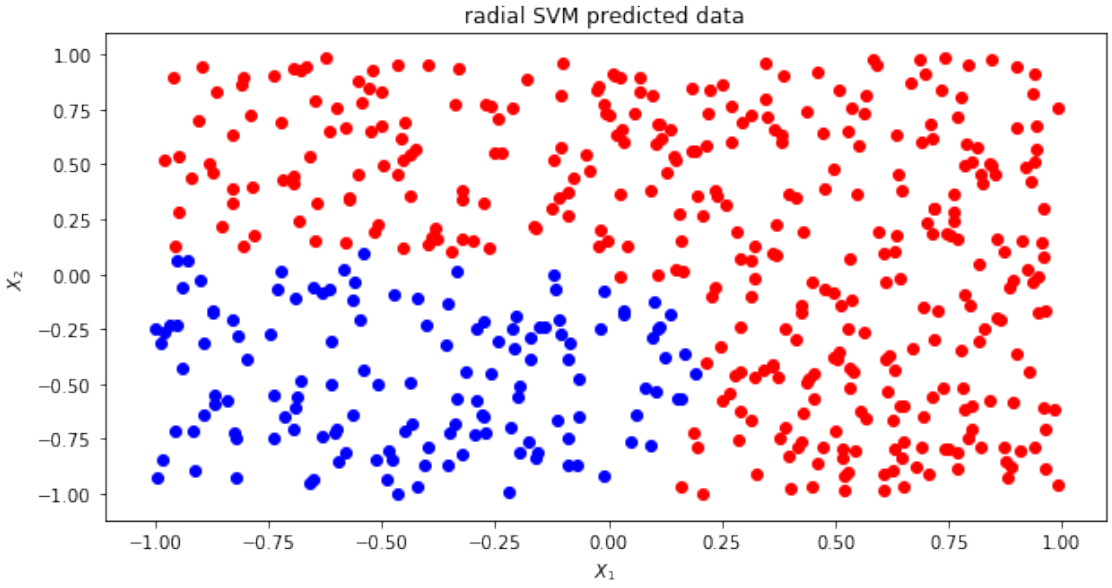
[40]:
```
linearSVM.fit(my_second_df[['X1','X2']], my_second_df['Y'])
linearSVM_df = pd.DataFrame({'X1':my_second_df['X1'],
                             'X2':my_second_df['X2'],
'Y':linearSVM.predict(my_second_df[['X1','X2']])})
```

```
plt.scatter(linearSVM_df.loc[linearSVM_df['Y']==1]['X1'],
            linearSVM_df.loc[linearSVM_df['Y']==1]['X2'], c='r')
plt.scatter(linearSVM_df.loc[linearSVM_df['Y']==0]['X1'],
            linearSVM_df.loc[linearSVM_df['Y']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('Linear SVM predicted data')
plt.show()
```



Q9. 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.

[41]:
```
radialSVM.fit(my_second_df[['X1','X2']], my_second_df['Y'])
radialSVM_df = pd.DataFrame({'X1':my_second_df['X1'],
                             'X2':my_second_df['X2'],
'Y':radialSVM.predict(my_second_df[['X1','X2']])})
plt.scatter(radialSVM_df.loc[radialSVM_df['Y']==1]['X1'],
            radialSVM_df.loc[radialSVM_df['Y']==1]['X2'], c='r')
plt.scatter(radialSVM_df.loc[radialSVM_df['Y']==0]['X1'],
            radialSVM_df.loc[radialSVM_df['Y']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('radial SVM predicted data')
plt.show()
```



Q10. Discuss your results and specifically the tradeoffs between estimating non-linear decision boundaries using these two different approaches.

[42]:
```
# calculating the error rates
print("present the error rate of each classifier:")
print("logistic regression with linear inputs: {}".format(
     sklearn.metrics.accuracy_score(my_second_df['Y'], LR_df['Y'])))
print("logistic regression with non-linear inputs: {}".format(
     sklearn.metrics.accuracy_score(my_second_df['Y'], LR_df['Y*'])))
print("SVM with linear kernel: {}".format(sklearn.metrics.accuracy_score(
     my_second_df['Y'], linearSVM_df['Y'])))
print("SVM with non-linear kernel: {}".format(sklearn.metrics.accuracy_score(
     my_second_df['Y'], radialSVM_df['Y'])))
```

```
present the error rate of each classifier:
logistic regression with linear inputs: 0.818
logistic regression with non-linear inputs: 0.858
SVM with linear kernel: 0.814
SVM with non-linear kernel: 0.86
```

From the results here, I see that generally using non-linear inputs or kernels are more helpful to estimate non-linear decision boundaries for both SVM and logistic regression classifier. But the premise is that you can make sure that the decision boundary is really non-linear. Otherwise, using non-linear models might overfit the data. For my data here, the non-linear SVM has a slight advantage over logistic regression.

However, one thing I think worthy of more consideration is that how much we know about this non-linear boundary. For the situation here, we already know the data generating process. Thus,

we can make more accurate non-linear inputs for the logistic functions. However, in most cases, we do not know so much about the factors and how they combines to affect the decision boundary. I have tested some other less relevant non-linear inputs for the logistic regression classifier, and some yield pretty bad results. So if we know little about the factors that form the decision boundary, then it is better to use SVM with non-linear kernel to detect decision boundary, otherwise you might have difficulty finding suitable non-linear functions/inputs for the logistic regression classifier. On the other hand, if we do know much about the factors related to the decision boundary, then I guess using logistic regression could also become an option. Also, in this case, logistic regression classifier might be able to reduce the possibility of overfitting because it is less sensitive to single obervations, especially those located around the hyperplane, compared to non-linear SVM.

0.1.3 Conceptual exercises-Tuning cost

In class we learned 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. You will now investigate that claim.

Q11. Generate two-class data with $p = 2$ in such a way that the classes are just barely linearly separable.

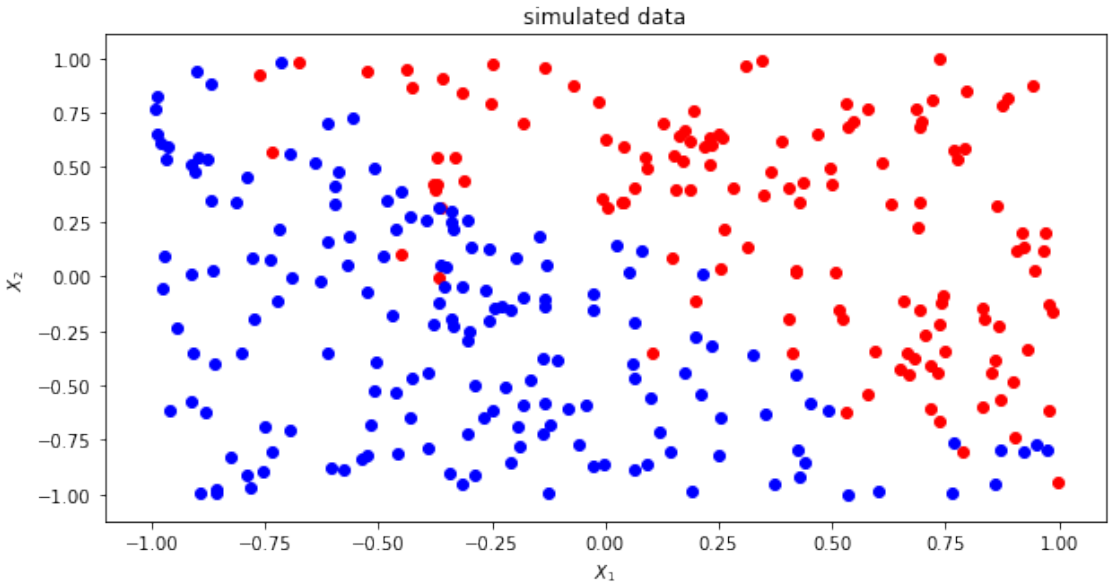
[68]:
```
my_third_train = pd.DataFrame({'X1':sp.stats.uniform.rvs(loc=-1, scale=2,␣
↪size=300),
                               'X2':sp.stats.uniform.rvs(loc=-1, scale=2,␣
↪size=300)})
my_third_train['Y'] = my_third_train['X1'] + my_third_train['X2'] + \
                      sp.stats.norm.rvs(loc=0, scale=0.2, size=300)
my_third_train['Y'] = my_third_train['Y'].apply(lambda x: 0 if x <= 0 else 1)
my_third_train
```

[68]:

	X1	X2	Y
0	-0.869729	0.880231	0
1	0.739699	-0.123542	1
2	-0.133966	-0.138041	0
3	0.694715	0.342149	1
4	-0.587212	0.481670	0
..	…	…	..
295	0.144474	-0.803075	0
296	-0.179225	-0.591955	0
297	0.996953	-0.947397	1
298	-0.304933	0.259912	0
299	-0.460306	0.211887	0

[300 rows x 3 columns]

[69]:
```
plt.rcParams['figure.figsize']=[10,5]
plt.scatter(my_third_train.loc[my_third_train['Y']==1]['X1'],
            my_third_train.loc[my_third_train['Y']==1]['X2'], c='r')
plt.scatter(my_third_train.loc[my_third_train['Y']==0]['X1'],
            my_third_train.loc[my_third_train['Y']==0]['X2'], c='b')
plt.xlabel('$X_{1}$')
plt.ylabel('$X_{2}$')
plt.title('simulated data')
plt.show()
```



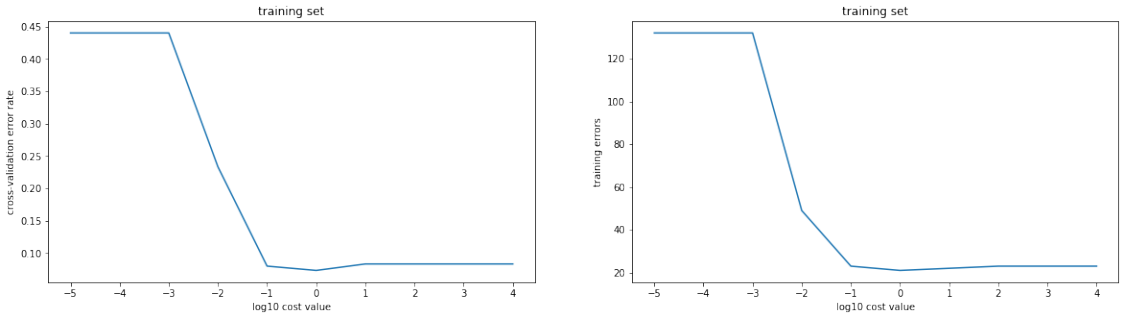
Q12. Compute the cross-validation error rates for support vector classifiers with a range of cost values. How many training errors are made for each value of cost considered, and how does this relate to the cross-validation errors obtained?

[70]:
```
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
training_scores = []
for c in np.power(10.0,range(-5,5)):
    clf = SVC(C=c, kernel='linear')
    clf.fit(my_third_train[['X1','X2']], my_third_train['Y'])
    scores = cross_val_score(clf, my_third_train[['X1','X2']],
                             my_third_train['Y'], cv=10)
    training_errors = len(np.nonzero(my_third_train['Y'].values-\
    clf.predict(my_third_train[['X1','X2']]))[0])
    training_scores.append((c, 1-np.mean(scores), training_errors))
    print("cost value: {}".format(c),
          "average CV error rate: {}".format(1-np.mean(scores)),
```

```
          "training errors: {}".format(training_errors))
```

```
cost value: 1e-05 average CV error rate: 0.44000000000000006 training errors:
132
cost value: 0.0001 average CV error rate: 0.44000000000000006 training errors:
132
cost value: 0.001 average CV error rate: 0.44000000000000006 training errors:
132
cost value: 0.01 average CV error rate: 0.23333333333333328 training errors: 49
cost value: 0.1 average CV error rate: 0.07999999999999985 training errors: 23
cost value: 1.0 average CV error rate: 0.07333333333333325 training errors: 21
cost value: 10.0 average CV error rate: 0.08333333333333326 training errors: 22
cost value: 100.0 average CV error rate: 0.08333333333333326 training errors: 23
cost value: 1000.0 average CV error rate: 0.08333333333333326 training errors:
23
cost value: 10000.0 average CV error rate: 0.08333333333333326 training errors:
23
```

[71]:
```
plt.rcParams['figure.figsize'] = [20,5]
fig, axes = plt.subplots(1,2)
axes[0].plot([np.log10(x[0]) for x in training_scores],
             [x[1] for x in training_scores])
axes[0].set_xticks([np.log10(x[0]) for x in training_scores])
axes[0].set_xlabel('log10 cost value')
axes[0].set_ylabel('cross-validation error rate')
axes[0].set_title('training set')

axes[1].plot([np.log10(x[0]) for x in training_scores],
             [x[2] for x in training_scores])
axes[1].set_xticks([np.log10(x[0]) for x in training_scores])
axes[1].set_xlabel('log10 cost value')
axes[1].set_ylabel('training errors')
axes[1].set_title('training set')
plt.show()
```



Based on this training set, we found that the cross validation error rate and training error do not

always move down forward monotonously as the cost value increases. As the cost value increases, we know that we are including more errors in searching for the optimal hyperplane and also decrease the misclassification in the training process. So we can see that in both graphs, there are periods where the error or error rates drops down quickly. However, when the hyperplane fits too well on the training data (after training errors become stable), that is to say more variances are included, there is likely to be overfitting issues. From the left graph (cross-validation graph) you can see that after 10^(0), the cross validation error rate even increases a bit as the cost increases, which could be viewed as a sign for overfitting.

Q13. 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?

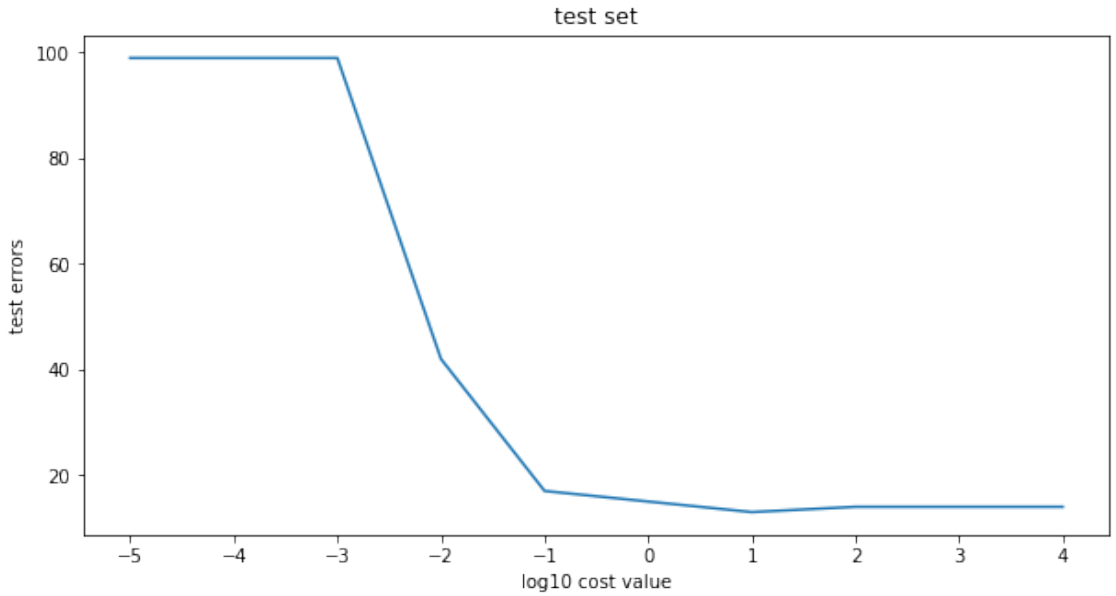
[77]:
```
# generate test data
my_third_test = pd.DataFrame({'X1':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=200),
                              'X2':sp.stats.uniform.rvs(loc=-1, scale=2,␣
 ↪size=200)})
my_third_test['Y'] = my_third_test['X1'] + my_third_test['X2'] + \
                     sp.stats.norm.rvs(loc=0, scale=0.2, size=200)
my_third_test['Y'] = my_third_test['Y'].apply(lambda x: 0 if x <= 0 else 1)
```

[78]:
```
# classify the test data
test_scores = []
for c in np.power(10.0,range(-5,5)):
    clf = SVC(C=c, kernel='linear')
    clf.fit(my_third_train[['X1','X2']], my_third_train['Y'])
    test_errors = len(np.nonzero(my_third_test['Y'].values-\
                  clf.predict(my_third_test[['X1','X2']]))[0])
    test_scores.append((c, test_errors))
    print("cost value: {}".format(c),
          "test errors: {}".format(test_errors))
```

```
cost value: 1e-05 test errors: 99
cost value: 0.0001 test errors: 99
cost value: 0.001 test errors: 99
cost value: 0.01 test errors: 42
cost value: 0.1 test errors: 17
cost value: 1.0 test errors: 15
cost value: 10.0 test errors: 13
cost value: 100.0 test errors: 14
cost value: 1000.0 test errors: 14
cost value: 10000.0 test errors: 14
```

[79]:
```
plt.rcParams['figure.figsize'] = [10,5]
plt.plot([np.log10(x[0]) for x in test_scores],
```

```
         [x[1] for x in test_scores])
plt.xticks([np.log10(x[0]) for x in test_scores])
plt.xlabel('log10 cost value')
plt.ylabel('test errors')
plt.title('test set')
plt.show()
```



From the data and plot above, we can see that the optimal cost value for the test set is around 10^(1), which is quite close to that of the training set cross-validation error rate (10^(0)) or the training errors (10^(0)).

Q14. Discuss your results

From above tuning process of C, we find that a smaller cost value is more likely to obtain lower errors on the test set for SVM because it is less likely to be faced with the issues of overfitting and also captures more information in the data compared to zero cost. As the cost value increases, we know that we are including more errors in searching for the optimal hyperplane that better fits the training data, but at the same time lose some generality of the model (that is overfitting). Also, there is one interesting thing about the training errors I did not mention above. Generally, increasing C will make more accurate classification on the training observations, but I found that the training errors would also slightly increase when C becomes quite large during multiple random simulations. So I suspect that when C is too large, the kind of errors we tolerated might change from more of the type of "on the wrong side of the margin" to more of the type of "on the wrong side of the hyperplane", which might drive the increase in training errors for very large C values.

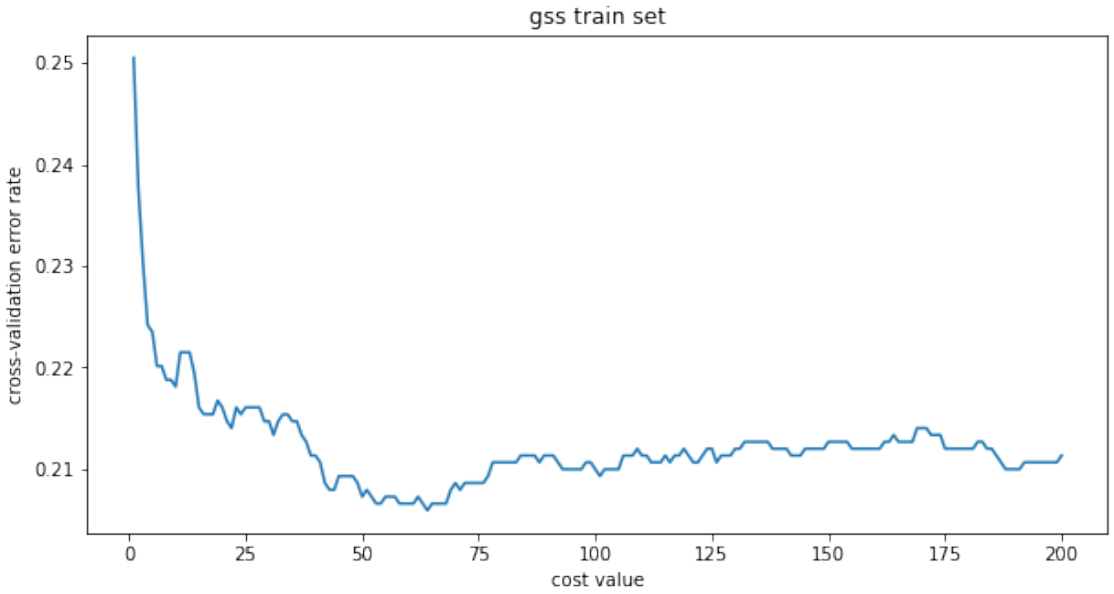
0.1.4 Application: Predicting attitudes towards racist college professors

Q15. 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.

[2]:
```
# load data
gss_train = pd.read_csv('data/gss_train.csv')
gss_test = pd.read_csv('data/gss_test.csv')
gss_train_features = gss_train.drop(columns='colrac')
gss_test_features = gss_test.drop(columns='colrac')
```

[ ]:
```
# For this application, I would try the rbf-SVM
gss_train_scores = []
for c in range(1,201):
    clf = SVC(C=c,kernel='linear')
    #clf.fit(my_third_train[['X1','X2']], my_third_train['Y'])
    scores = cross_val_score(clf, gss_train_features,
    gss_train['colrac'], cv=10, n_jobs=-1)
    gss_train_scores.append((c, 1-np.mean(scores)))
```

[66]:
```
plt.rcParams['figure.figsize'] = [10,5]
plt.plot([x[0] for x in gss_train_scores],[x[1] for x in gss_train_scores])
#plt.xticks([x[0] for x in test_scores])
plt.xlabel('cost value')
plt.ylabel('cross-validation error rate')
plt.title('gss train set')
plt.show()
```



[67]:
```
print("optimal cost value: {}".format(sorted(gss_train_scores,
key=lambda x:x[1])[0][0]))
```

```
optimal cost value: 64
```

The result ia pretty much like what we have observed in the last section (tuning cost). The cross validation score firstly decreases and then increases as the cost value becomes bigger, indicating that neither a too small or too big cost value would be the most effective for the SVM.

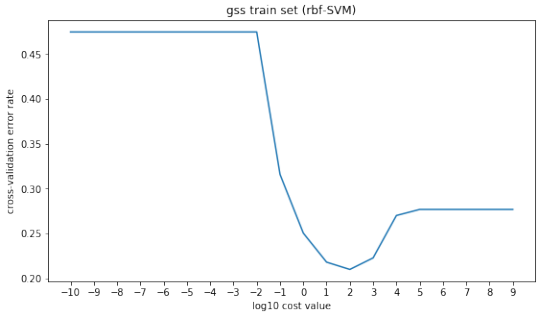
Q16. Repeat the previous question, but this time using SVMs with radial and polynomial basis kernels, with different values for gamma and degree and cost. Present and discuss your results.

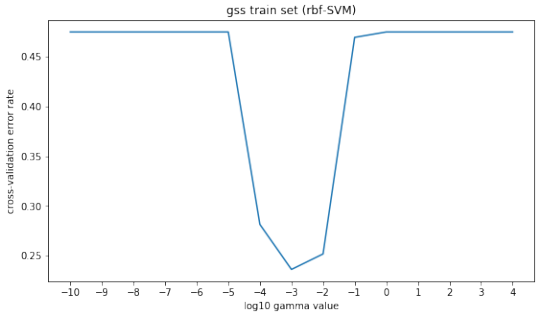
[3]:
```
# rbf-SVM
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
```

[29]:
```
rbf_cost_scores = []
for c in np.power(10.0,range(-10,10)):
    clf = SVC(C=c,kernel='rbf')
    #clf.fit(my_third_train[['X1','X2']], my_third_train['Y'])
    scores = cross_val_score(clf, gss_train_features,
    gss_train['colrac'], cv=10, n_jobs=-1)
    rbf_cost_scores.append((c, 1-np.mean(scores)))

rbf_gamma_scores = []
for g in np.power(10.0,range(-10,5)):
    clf = SVC(gamma=g,kernel='rbf')
    #clf.fit(my_third_train[['X1','X2']], my_third_train['Y'])
    scores = cross_val_score(clf, gss_train_features,
    gss_train['colrac'], cv=10, n_jobs=-1)
    rbf_gamma_scores.append((g, 1-np.mean(scores)))
```

[33]:
```
# plot the result for rbf-SVM
plt.rcParams['figure.figsize'] = [20,5]
fig, rbf_axes = plt.subplots(1,2)
rbf_axes[0].plot([np.log10(x[0]) for x in rbf_cost_scores],
                 [x[1] for x in rbf_cost_scores])
rbf_axes[0].set_xticks([np.log10(x[0]) for x in rbf_cost_scores])
rbf_axes[0].set_xlabel('log10 cost value')
rbf_axes[0].set_ylabel('cross-validation error rate')
rbf_axes[0].set_title('gss train set (rbf-SVM)')

rbf_axes[1].plot([np.log10(x[0]) for x in rbf_gamma_scores],
                 [x[1] for x in rbf_gamma_scores])
rbf_axes[1].set_xticks([np.log10(x[0]) for x in rbf_gamma_scores])
rbf_axes[1].set_xlabel('log10 gamma value')
rbf_axes[1].set_ylabel('cross-validation error rate')
```

```
rbf_axes[1].set_title('gss train set (rbf-SVM)')
plt.show()
```





For the cost value, the RBF kernel SVM shows the same tendency as what we have observed before. The cross validation error rate would first drops and then rises as the cost value increases, because too small cost can not capture the classifiable properties of the data well while too big cost will easily overfit the data. For gamma, the cross-validation error rate has the same tendency as the cost value, and I notice that in scikit-learn, that the default gamma value is "1 / (n_features * X.var())", which is really a small value for our data set. After searching on Internet, we found that the scikit-learn has the following statement:

> "The behavior of the model is very sensitive to the gamma parameter. If gamma is too large, the radius of the area of influence of the support vectors only includes the support vector itself and no amount of regularization with C will be able to prevent overfitting. When gamma is very small, the model is too constrained and cannot capture the complexity or "shape" of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model with a set of hyperplanes that separate the centers of high density of any pair of two classes."

So basically, we need to chose both the cost and gamma of medium values to obtain a better RBF-SVM model.

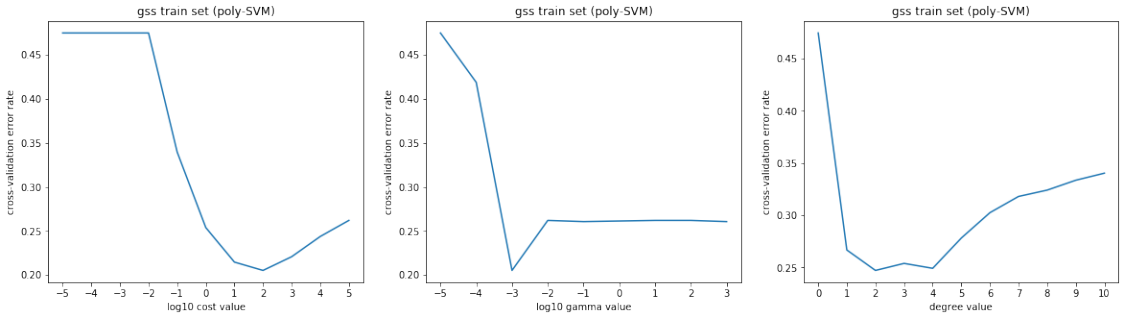
[14]:
```
# poly-SVM
poly_cost_scores = []
for c in np.power(10.0,range(-5,6)):
    clf = SVC(C=c,kernel='poly')
    scores = cross_val_score(clf, gss_train_features,
    gss_train['colrac'], cv=10, n_jobs=-1)
    poly_cost_scores.append((c, 1-np.mean(scores)))
```

[21]:
```
poly_degree_scores = []
for d in range(0,11):
    clf = SVC(degree = d,kernel='poly')
    scores = cross_val_score(clf, gss_train_features,
    gss_train['colrac'], cv=10, n_jobs=-1)
```

```
    poly_degree_scores.append((d, 1-np.mean(scores)))
```

[ ]:
```
poly_gamma_scores = []
for g in np.power(10.0,range(-5,4)):
    clf = SVC(gamma = g ,kernel='poly')
    scores = cross_val_score(clf, gss_train_features,
    gss_train['colrac'], cv=10, n_jobs=-1)
    poly_gamma_scores.append((g, 1-np.mean(scores)))
    print(g)
```

[23]:
```
# plot the result for poly-SVM
plt.rcParams['figure.figsize'] = [20,5]
fig, poly_axes = plt.subplots(1,3)
poly_axes[0].plot([np.log10(x[0]) for x in poly_cost_scores],
                  [x[1] for x in poly_cost_scores])
poly_axes[0].set_xticks([np.log10(x[0]) for x in poly_cost_scores])
poly_axes[0].set_xlabel('log10 cost value')
poly_axes[0].set_ylabel('cross-validation error rate')
poly_axes[0].set_title('gss train set (poly-SVM)')

poly_axes[1].plot([np.log10(x[0]) for x in poly_gamma_scores],
                  [x[1] for x in poly_gamma_scores])
poly_axes[1].set_xticks([np.log10(x[0]) for x in poly_gamma_scores])
poly_axes[1].set_xlabel('log10 gamma value')
poly_axes[1].set_ylabel('cross-validation error rate')
poly_axes[1].set_title('gss train set (poly-SVM)')

poly_axes[2].plot([x[0] for x in poly_degree_scores],
                  [x[1] for x in poly_degree_scores])
poly_axes[2].set_xticks([x[0] for x in poly_degree_scores])
poly_axes[2].set_xlabel('degree value')
poly_axes[2].set_ylabel('cross-validation error rate')
poly_axes[2].set_title('gss train set (poly-SVM)')
plt.show()
```



For the poly-SVM model, the cost value follows the similar tendency as linear or RBF SVM models. For the degree parameter in the polynomial kernels, we found that when the degree value is pretty small, we might be unable to capture the classifiable features from the data. while when we have very higher degrees, the decision boundary we have checked could be overly flexible, which gives rise to the issue of overfitting. As for the gamma in this kernel function, it also affects the scale of the kernal functions. Still, it functions best when its value is neither too big or too small, but compared with the gamma in the rbf-kernel, this gamma is less sensitive after the optimal value (i.e, not make the error rate increase too much).