MATH6350_HW4_Part1

November 18, 2019

Statistical Learning and Data Mining
Dr. Azencott
Sable Levy

Part1 : SVM classification for Simulated Data

In [124]:
```
"""
Created on Fri Nov  1 14:43:57 2019

@author: sablelevy
"""

import time
start_time = time.time()
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn import svm
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
from sklearn.model_selection import train_test_split

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

#setting a random seed
seed = 313

#Question 1 : Generate a Data Set by Simulations
#Step 1
np.random.seed(seed)
#dfr = pd.DataFrame(np.random.randn(100, 4), columns=['A', 'B', 'C','D'])
dfr = pd.DataFrame(np.random.uniform(-2, 2, size=(5000, 4)), columns=list('abcd'))

#A = pd.DataFrame(np.random.uniform(-2, 2, size=(4, 4)), columns=list('1234'))
A = np.array(np.random.uniform(-2, 2, size=(4, 4))).round(2)
B = np.array(np.random.uniform(-2, 2, size=(4, 1))).round(2)
c = np.array(np.random.uniform(-2, 2, size=(1, 1)))[0][0].round(2)

print("A:\n",A, "\nB:\n",B,"\nc:\n", c)

x=dfr.iloc[0].values  #.values converts a df (or a df subset) into an arracy
```

```
#x is an array of 4 numbers
#x = arr[row_index]
#r = round = 1 to round
def Pol(x, A, B ,c,r):
    #a list that will store the successive products for every combination of i,j
    ls = []
    for i in range(0,4):
        ls.append(np.multiply(x[i],B[i]))
        for j in range(0,4):
            #np.multiply only accepts 2 arguments, so had to improvise for 3 arguments
            ls.append(np.multiply(np.multiply(x[i],x[j]),A[i][j]))

    result=np.sum(ls)
    #returns an array object without specifying [0]
    if r == 1:
        return (result[0] + c/20).round(2)
    else:
        return (result[0] + c/20)

#Step 2
np.random.seed(312)

dfr2 = pd.DataFrame(np.random.uniform(-2, 2, size=(10000, 4)), columns=list('abcd'))
#arr2 = np.array(np.random.uniform(-2, 2, size=(10000, 4))).round(2)
arr2 = dfr2.values  #same as above

dfr2['class'] = 0
dfr2['pol(x)'] = 0

#creating vector of Pol(x) output
for n in range(len(dfr2)):
    x = arr2[n]
    dfr2.loc[n,'pol(x)'] = Pol(x, A, B ,c,0)

#n is the index of the x
def y_f(n):
    x = arr2[n]
    Un = Pol(x, A, B ,c,0)
    if Un > 0:
        return 1
    elif Un < 0:
        return -1
    else:
        print("On the margin!")

#assigning classes to each case
def sort(df):
    for n in range(len(df)):
        dfr2.loc[n,'class'] = y_f(n)

sort(dfr2)

Cl_plus1 = dfr2[dfr2['class']==1]
```

```
Cl_minus1 = dfr2[dfr2['class']==-1]

#selecting first 2500 cases from each class:
Cl_plus1 = Cl_plus1.iloc[:2500,:]
Cl_minus1 = Cl_minus1.iloc[:2500,:]

Cl_df = pd.concat([Cl_plus1,Cl_minus1]).sort_index()
poly=Cl_df.iloc[:]['pol(x)'].reset_index() #must reset because some have index >5000
poly_to_plot=poly['pol(x)']

#Standardizing columns
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
#sc.fit_transform(Cl_df) makes Cl_df a numpy array, so we can convert it to pd in one line:
Cl_df_sc = pd.DataFrame(sc.fit_transform(Cl_df)) #we lose column names

#Splitting into X data and target y
#dropping target column, and pol(x) column
X = Cl_df_sc.drop(columns = [4,5])
y = Cl_df_sc[4]


from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)

#checking proportion of each class in test set
m1 = 0; m2=0;
for i in range (len(y_test)):
    if y_test.iloc[i] == 1:
        m1+= 1
    elif y_test.iloc[i] == -1:
        m2+=1

#verifying that the class ratios in the test set are roughly equal
print('\nClass ratios in test set:\nCl(+1): {:.2f}, Cl(-1): {:.2f}'.format(m1/len(y_test),
      m2/len(y_test)))

end_time=time.time()
total_time = round(end_time - start_time,3)
print('Time:', total_time, 'seconds.')
```

```
A:
 [[-0.4   0.01  0.82 -1.75]
 [-0.09 -0.59  0.67 -0.03]
 [-1.67 -1.46  0.14  1.06]
 [-1.31  1.71 -1.78 -1.87]]
B:
 [[0.66]
 [1.35]
 [0.36]
 [1.22]]
c:
 -1.37
```

```
Class ratios in test set:
Cl(+1): 0.51, Cl(-1): 0.49
Time: 9.984 seconds.
```

0.0.1 Defining SVM model class, to perform analytics and parameter optimization:

```
In [128]: import time
          start_time = time.time()

          #Defining SVM class
          class SVM:
              def __init__(self, kernel, parameters, X, y, rand_state):
                  self.kernel = kernel
                  self.parameters = parameters
                  self.rand_state=rand_state
                  self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(X,
                              y, test_size=0.2, random_state=self.rand_state)
                  self.best_cost = None

              #To display descriptive analytics
              def analytics(self):
                  #selecting the proper kernel
                  if self.kernel == 'linear':
                      clf = svm.SVC(kernel=self.kernel, C=self.parameters['SVM__C'][0], random_state=973
                  elif self.kernel == 'rbf':
                      clf = svm.SVC(kernel=self.kernel, C=self.parameters['SVM__C'][0],
                                    gamma=self.parameters['SVM__gamma'][0], random_state=973)
                  elif self.kernel == 'poly':
                      clf = svm.SVC(kernel=self.kernel, C=self.parameters['SVM__C'][0],
                                    degree=self.parameters['degree'][0],
                                    coef0=self.parameters['SVM__coef0'][0], gamma='auto',
                                    random_state=973)
                  #fitting pre-scaled data
                  clf.fit(self.X_train, self.y_train)
                  #predictions on test and training set
                  predictions = clf.predict(self.X_test)
                  predictions_train = clf.predict(self.X_train)
                  #finding number of support vectors (sum of number from each class)
                  S = (clf.n_support_[0])+(clf.n_support_[1])
                  print("Number of support vectors in training set:", S)
                  print("Ratio of support vectors in training set = %3.2f" %(S/len(self.X_train)))
                  #percentages of correct predictions on test and training sets
                  print("Test set prediction accuracy = %3.2f" %(clf.score(self.X_test,self.y_test)))
                  print("Training set prediction accuracy = %3.2f" %(clf.score(self.X_train,self.y_trai
                  #printing confusion matrices for both sets
                  cm_test = pd.DataFrame(confusion_matrix(predictions, self.y_test)) #array otherwise
                  cm_train = pd.DataFrame(confusion_matrix(predictions_train, self.y_train))
                  print('Test Set Confusion Matrix: \n{}'.format(cm_test))
                  print('Training Set Confusion Matrix: \n{}'.format(cm_train))
                  #confustion matrices in % form:
                  cm_test_p = cm_test.copy()
                  cm_train_p = cm_train.copy()
                  for i in list(cm_test.index):
```

```
            cm_test_p.iloc[i] =  cm_test.iloc[i].apply(lambda x: x/sum(cm_test.iloc[i]))
            cm_train_p.iloc[i] =  cm_train.iloc[i].apply(lambda x: x/sum(cm_train.iloc[i]))
        print('Test Set Confusion Matrix by %: \n{}'.format(cm_test_p.round(2)))
        print('Training Set Confusion Matrix by %: \n{}'.format(cm_train_p.round(2)))
        #computing performance of testing and training sets, with 95% CI
        p_test = clf.score(X_test,y_test)
        p_train = clf.score(X_train,y_train)
        #standard deviation
        std_ptest=np.sqrt(p_test*(1-p_test)/len(X_test))
        std_ptrain=np.sqrt(p_train*(1-p_train)/len(X_train))

        p_test_CI=p_test-(1.96*std_ptest),p_test+(1.96*std_ptest)
        p_train_CI=p_train-(1.96*std_ptrain),p_train+(1.96*std_ptrain)
        print("95 CI for test set performance: = %3.2f, %3.2f" %(p_test_CI[0],p_test_CI[1]))
        print("95 CI for training set performance: = %3.2f, %3.2f" %(p_train_CI[0],
                                                                    p_train_CI[1]))

    #Parameter optimization and displaying analytical results
    def param_optimize(self):
        if self.kernel == 'linear':
            parameters={k: self.parameters[k] for k in ['SVM__C']}
            clf = svm.SVC(kernel=self.kernel, C=parameters['SVM__C'][0], random_state=seed)
        elif self.kernel == 'rbf':
            parameters={k: self.parameters[k] for k in ['SVM__C','SVM__gamma']}
            clf = svm.SVC(kernel=self.kernel, C=parameters['SVM__C'][0],
                          gamma=parameters['SVM__gamma'][0], random_state=973)
        elif self.kernel == 'poly':
            parameters={k: self.parameters[k] for k in ['SVM__C','SVM__coef0']}
            clf = svm.SVC(kernel=self.kernel, C=parameters['SVM__C'][0],
                          degree=self.parameters['degree'][0],
                          coef0=parameters['SVM__coef0'][0],
                          gamma='auto',
                          random_state=973)
        else:
            return("Error, invalid kernel name.")

        steps = [('scaler', StandardScaler()), ('SVM', clf)]
        pipeline = Pipeline(steps)
        #Iterating through all parameter combinations to find best performance parameters
        grid = GridSearchCV(pipeline, param_grid=parameters, cv=10) #10-fold cross validation
        grid.fit(self.X_train, self.y_train)
        print("\nBest parameters from list of options:", grid.best_params_)
        #best parameters
        best_cost = grid.best_params_.get('SVM__C',None)
        best_gamma = grid.best_params_.get('SVM__gamma',None)
        best_a = grid.best_params_.get('SVM__coef0',None)
        #using best parameters to update svm classifier
        if self.kernel == 'linear':
            clf_best_params = svm.SVC(kernel=self.kernel, C=best_cost, random_state=seed)
        elif self.kernel == 'rbf':
            clf_best_params = svm.SVC(kernel=self.kernel, C=best_cost,
                          gamma=best_gamma, random_state=973)
        elif self.kernel == 'poly':
            clf_best_params = svm.SVC(kernel=self.kernel,
```

```
                C=best_cost, degree=self.parameters['degree'][0], coef0=best_a,
                                        gamma='auto', random_state=973)
        else:
            return("Error, invalid kernel name.")

        clf_best_params.fit(self.X_train, self.y_train)
        predictions = clf_best_params.predict(self.X_test)
        predictions_train = clf_best_params.predict(self.X_train)
        #finding number of support vectors in each set
        S = (clf_best_params.n_support_[0])+(clf_best_params.n_support_[1])
        print("Number of support vectors in training set:", S)
        print("Ratio of support vectors in training set = %3.2f" %(S/4000))
        #percentages of correct predictions
        print("Test set accuracy = %3.2f" %(clf_best_params.score(X_test,y_test)))
        print("Training set accuracy = %3.2f" %(clf_best_params.score(X_train,y_train)))
        #printing confustion matrices for both sets
        cm_test = pd.DataFrame(confusion_matrix(predictions, self.y_test)) #array otherwise
        cm_train = pd.DataFrame(confusion_matrix(predictions_train, self.y_train))
        print('Test Set Confusion Matrix: \n{}'.format(cm_test))
        print('Training Set Confusion Matrix: \n{}'.format(cm_train))
        #confustion matrices in % form:
        cm_test_p = cm_test.copy()
        cm_train_p = cm_train.copy()
        for i in list(cm_test.index):
            cm_test_p.iloc[i] =  cm_test.iloc[i].apply(lambda x: x/sum(cm_test.iloc[i]))
            cm_train_p.iloc[i] =  cm_train.iloc[i].apply(lambda x: x/sum(cm_train.iloc[i]))
        print('Test Set Confusion Matrix by %: \n{}'.format(cm_test_p.round(2)))
        print('Training Set Confusion Matrix by %: \n{}'.format(cm_train_p.round(2)))
        #computing performance of testing and training sets, with 95% CI
        p_test = clf_best_params.score(X_test,y_test)
        p_train = clf_best_params.score(X_train,y_train)
        #standard deviation
        std_ptest=np.sqrt(p_test*(1-p_test)/len(X_test))
        std_ptrain=np.sqrt(p_train*(1-p_train)/len(X_train))
        #95% CI
        p_test_CI=p_test-(1.6*std_ptest),p_test+(1.6*std_ptest)
        p_train_CI=p_train-(1.6*std_ptrain),p_train+(1.6*std_ptrain)

        print("95 CI for test set performance: %3.2f, %3.2f" %(p_test_CI[0],
                                                               p_test_CI[1]))
        print("95 CI for training set performance: %3.2f, %3.2f" %(p_train_CI[0],
                                                                   p_train_CI[1]))

        self.best_cost=best_cost
        self.clf = clf_best_params

    def get_best_cost(self):
        return(self.best_cost)

    def plot(self, poly_to_plot):
        #poly = df.loc[:]['pol(x)']
        #poly =poly_to_plot[poly_to_plot.index.isin(X_train.index)]
        #Z = df.iloc[:5000]['pol(x)']
        #Y = self.clf.decision_function(self.X_train)
```

```
                Y = self.clf.decision_function(X)
                plt.figure(figsize=(12, 8))
                plt.scatter(poly_to_plot, Y, marker='.',color='crimson',alpha=.5)
                plt.grid(True)
                plt.xlabel("Poly(X)")
                plt.ylabel("SVM(X)")
                plt.title('Poly(X) vs. SVM(X)')
                plt.axhline(0,color='black') # x = 0
                plt.axvline(0,color='black') # y = 0
                plt.show()


        end_time=time.time()
        total_time = round(end_time - start_time,3)
        print('Time:', total_time, 'seconds.')
```

```
Time: 0.001 seconds.
```

0.0.2 Question 2: SVM classification by linear kernel

```
In [126]: start_time = time.time()

          #list initial-model parameter value first
          #Setting parameters and performing analytics
          parameters={'SVM__C': [5]}
          SVM_linear = SVM('linear', parameters, X, y, seed)
          SVM_linear.analytics()

          end_time=time.time()
          total_time = round(end_time - start_time,3)
          print('Time:', total_time, 'seconds.')
```

```
Number of support vectors in training set: 2994
Ratio of support vectors in training set = 0.75
Test set prediction accuracy = 0.66
Training set prediction accuracy = 0.67
Test Set Confusion Matrix:
     0    1
0  298  148
1  196  358
Training Set Confusion Matrix:
      0     1
0  1281   578
1   725  1416
Test Set Confusion Matrix by %:
      0     1
0  0.67  0.33
1  0.35  0.65
Training Set Confusion Matrix by %:
      0     1
0  0.69  0.31
1  0.34  0.66
95 CI for test set performance: = 0.63, 0.69
95 CI for training set performance: = 0.66, 0.69
```

```
Time: 0.86 seconds.
```

These results show that the training set accuracy is .67, with a 95% Confidence Interval = (.66,.69), while the test set accuracy is .66, with a 95% Confidence Interval = (.63,.69). This result can also be observed by looking at the mean of the diagonal of the respective confusion matrices given in percent form. Furthermore, there is a large number of support vectors in the training set: 2994, which corresponds to 75% of the cases in the entire training set. This means that the SVM algorithm had difficulty classifying 75% of the training set cases.

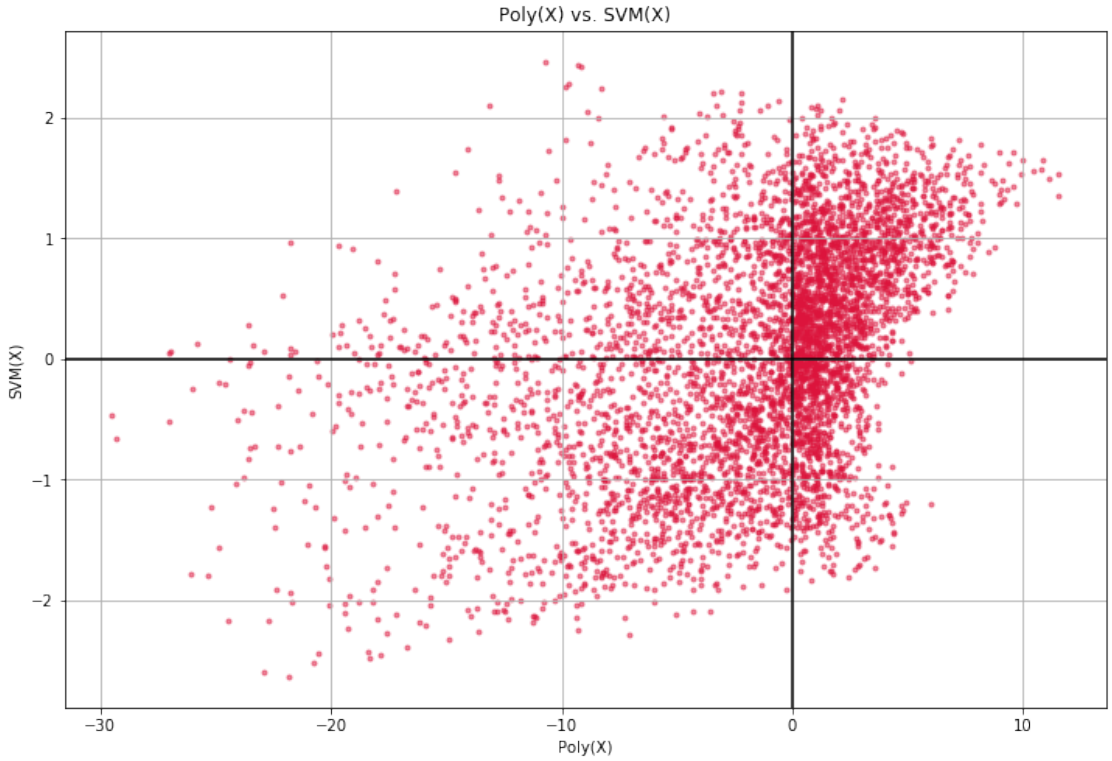
0.0.3 Question 3 : optimize the parameter "cost"

```
In [134]: start_time = time.time()

          #Performing analytics on a list of parameters
          parameters={'SVM__C': [2,5,10,15,20,30]}
          SVM_linear = SVM('linear', parameters, X, y, seed)
          SVM_linear.param_optimize()

          end_time=time.time()
          total_time = round(end_time - start_time,3)
          print('Time:', total_time, 'seconds.')
```

```
Best parameters from list of options: {'SVM__C': 5}
Number of support vectors in training set: 2994
Ratio of support vectors in training set = 0.75
Test set accuracy = 0.66
Training set accuracy = 0.67
Test Set Confusion Matrix:
     0    1
0  298  148
1  196  358
Training Set Confusion Matrix:
      0     1
0  1281   578
1   725  1416
Test Set Confusion Matrix by %:
      0     1
0  0.67  0.33
1  0.35  0.65
Training Set Confusion Matrix by %:
      0     1
0  0.69  0.31
1  0.34  0.66
95 CI for test set performance: 0.63, 0.68
95 CI for training set performance: 0.66, 0.69
Time: 48.829 seconds.
```

These results are same as the those in question 2 because the tuning function found the best cost is C=5, which is what we initially used.

```
In [135]: SVM_linear.plot(poly_to_plot)
```



We have included a plot SVM(x) vs. Poly(x) for all the data in the training set. This plot is very useful; its quadrants correspond to the four cells of the confusion matrix. The first quadrant shows cases where the data were in class CL(+1) and were classified as positive by the SVM classifier, while the third quadrant shows cases where the data were in class CL(-1) and were classified as negative by the SVM classifier. The 2nd and 4th quadrants, conversely, correspond to erroneous classifications. The 2nd quandrant contains cases that are in true class CL(-1) but which were classified as positive by the SVM classifier, while the 4th qudrant contains cases that are in true class CL(+1) but which were classified by the SVM classifier as negative. The points that are in the first and third quadrants correspond to correct classifications, and therefore contain 67% of the data. It's clear that our data are not very well-separated with the linear kernel.

0.0.4 Question 4: SVM classification by radial kernel

In [130]:
```
start_time = time.time()

#getting best cost from linear kernel SVM
best_cost=SVM_linear.get_best_cost()

#Setting parameters and performing analytics
parameters={'SVM__C': [best_cost], 'SVM__gamma': [1]}
SVM_radial = SVM('rbf', parameters, X, y, seed)
SVM_radial.analytics()

end_time=time.time()
total_time = round(end_time - start_time,3)
print('Time:', total_time, 'seconds.')
```

```
Number of support vectors in training set: 453
Ratio of support vectors in training set = 0.11
Test set prediction accuracy = 0.98
Training set prediction accuracy = 0.99
Test Set Confusion Matrix:
     0    1
0  478    8
1   16  498
Training Set Confusion Matrix:
      0     1
0  1987     1
1    19  1993
Test Set Confusion Matrix by %:
      0     1
0  0.98  0.02
1  0.03  0.97
Training Set Confusion Matrix by %:
      0     1
0  1.00  0.00
1  0.01  0.99
95 CI for test set performance: = 0.97, 0.99
95 CI for training set performance: = 0.99, 1.00
Time: 0.159 seconds.
```

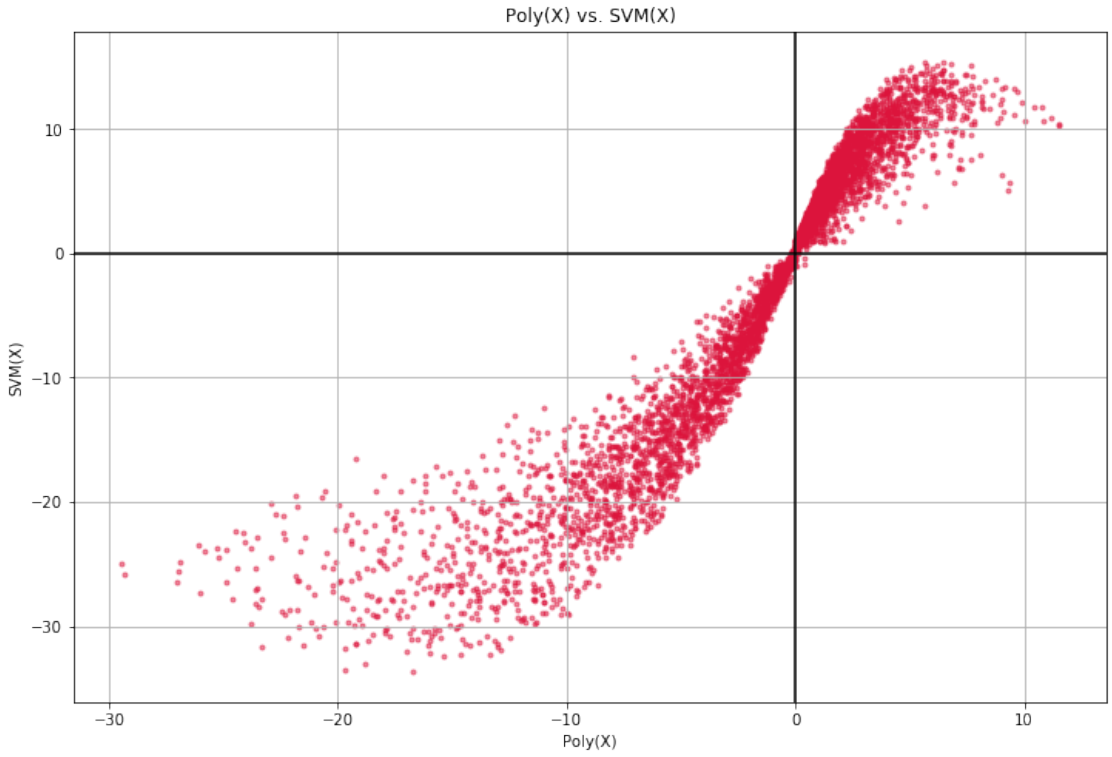
These results show that the training set accuracy is .99, with a 95% Confidence Interval = (.99,1), while the test set accuracy is .98, with a 95% Confidence Interval = (.97,.99). This result can also be observed by looking at the mean of the diagonal of the respective confusion matrices given in percent form. This performance is significantly better than the performance from using a linear kernel. Furthermore, the number of support vectors, 453, has greatly diminished, corresponding to 11% of the cases in the entire training set. This means that the SVM algorithm had difficulty classifying 11% of the training set cases, which is much better than the 75% of support vectors we found using a linear kernel. Naturally, as we can see in these examples, a small % of support vectors corresponds to greater accuracy, while a large % of support vectors corresponds to less accuracy.

0.0.5 Question 5 : optimize the parameter "cost"and "gamma"

In [131]:
```
start_time = time.time()

parameters={'SVM__C':[.1,5,10,20,50,best_cost], 'SVM__gamma':[.01,.1,.25,.5,1]}
SVM_radial = SVM('rbf', parameters, X, y, seed)
SVM_radial.param_optimize()
SVM_radial.plot(poly_to_plot)

end_time=time.time()
total_time = round(end_time - start_time,3)
print('Time:', total_time, 'seconds.')
```

```
Best parameters from list of options: {'SVM__C': 50, 'SVM__gamma': 0.25}
Number of support vectors in training set: 235
Ratio of support vectors in training set = 0.06
Test set accuracy = 0.99
Training set accuracy = 1.00
```

```
Test Set Confusion Matrix:
     0    1
0  485    3
1    9  503
Training Set Confusion Matrix:
      0     1
0  1995     1
1    11  1993
Test Set Confusion Matrix by %:
      0     1
0  0.99  0.01
1  0.02  0.98
Training Set Confusion Matrix by %:
      0     1
0  1.00  0.00
1  0.01  0.99
95 CI for test set performance: 0.98, 0.99
95 CI for training set performance: 1.00, 1.00
```




```
Time: 22.67 seconds.
```

When we optimize on the parameters cost and γ, our program has found that from among our list of possible options, the best cost is C=50 and the best gamma is γ = 0.25. This gives us just slightly better results from as we had in question 4, when we used C=5 and γ = 1. In particular, the number of support vectors in the training set, 235, represents only 6% of the training set, as opposed to the 11% we had

in question 4. Furthermore, the training set accuracy is 100%, with a 95% Confidence Interval = (1,1), while the test set accuracy is .99, with a 95% Confidence Interval = (.98,.99). The performance we get when using these optimal parameters is only marginally better than the performance we had for the non-tuned parameters used in question 4.

We have included a plot SVM(x) vs. Poly(x) for the data. This plot is very useful; its quadrants correspond to the four cells of the confusion matrix. The first quadrant shows cases where the data were in class CL(+1) and were classified as positive by the SVM classifier with a radial kernel, while the third quadrant shows cases where the data were in class CL(-1) and were classified as negative by the SVM classifier. The 2nd and 4th quadrants, conversely, correspond to erroneous classifications. The points that are in the first and third quadrants correspond to correct classifications, and therefore contain almost 100% of the data. We can see by inspection that this is true, which corroborate the performance rate we have calculated, and tells us that our seperator is very good.

0.0.6 Question 6 : SVM classification using a polynomial kernel

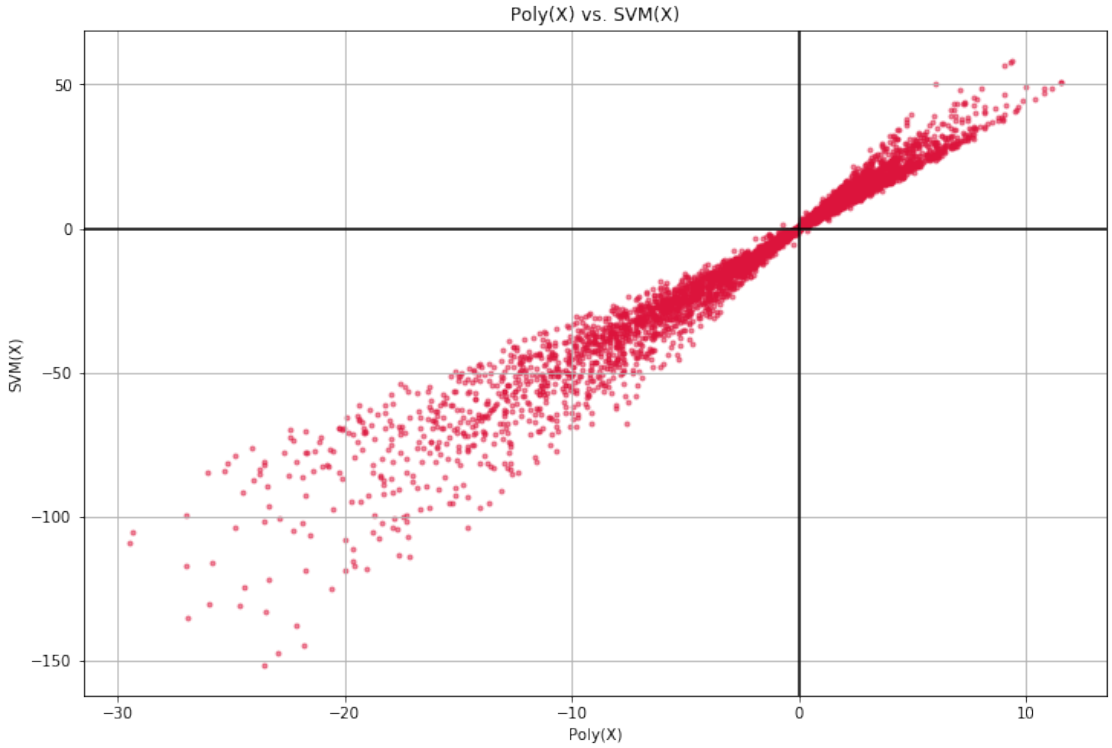
```
In [132]: start_time = time.time()

          parameters={'SVM__C': [best_cost], 'SVM__coef0':[1], 'degree': [4]}
          SVM_polynomial = SVM('poly',parameters,X,y,seed)
          SVM_polynomial.analytics()


          end_time=time.time()
          total_time = round(end_time - start_time,3)
          print('Time:', total_time, 'seconds.')
```

```
Number of support vectors in training set: 217
Ratio of support vectors in training set = 0.05
Test set prediction accuracy = 0.99
Training set prediction accuracy = 0.99
Test Set Confusion Matrix:
     0    1
0  486    2
1    8  504
Training Set Confusion Matrix:
      0     1
0  1988     2
1    18  1992
Test Set Confusion Matrix by %:
      0     1
0  1.00  0.00
1  0.02  0.98
Training Set Confusion Matrix by %:
      0     1
0  1.00  0.00
1  0.01  0.99
95 CI for test set performance: = 0.98, 1.00
95 CI for training set performance: = 0.99, 1.00
Time: 0.096 seconds.
```

```
In [133]: start_time = time.time()
```

```
parameters = {'SVM__C':[10,20,25,30,best_cost], 'SVM__coef0':[.001,.01,.1,.5,1],
              'degree': [4]}
SVM_polynomial = SVM('poly',parameters,X,y,seed)
SVM_polynomial.param_optimize()
SVM_polynomial.plot(poly_to_plot)

end_time=time.time()
total_time = round(end_time - start_time,3)
print('Time:', total_time, 'seconds.')
```

```
Best parameters from list of options: {'SVM__C': 20, 'SVM__coef0': 1}
Number of support vectors in training set: 141
Ratio of support vectors in training set = 0.04
Test set accuracy = 0.99
Training set accuracy = 1.00
Test Set Confusion Matrix:
     0    1
0  487    2
1    7  504
Training Set Confusion Matrix:
      0     1
0  1997     1
1     9  1993
Test Set Confusion Matrix by %:
      0     1
0  1.00  0.00
1  0.01  0.99
Training Set Confusion Matrix by %:
     0    1
0  1.0  0.0
1  0.0  1.0
95 CI for test set performance: 0.99, 1.00
95 CI for training set performance: 1.00, 1.00
```




```
Time: 14.105 seconds.
```

When we optimize on the parameters cost and a, our program has found that from among our list of possible options, the best cost is C=20 and the best a is a = 1. This gives us just slightly better results from as we had initially, when we used C=50 and a = 1. Before tuning the parameters, the number of support vectors in the training set, 141, represented only 4% of the training set, which slightly improves after tuning to 217 support vectors and 5%. Furthermore, the 95% Confidence Interval on training set performance = (.99,1) prior to tuning, and is (1,1) after tuning. The 95% Confidence Interval on test set performance = (.98,1) prior to tuning, and is (.99,1) after tuning. As such, the performance we get when using these optimal parameters is only marginally better than the performance we had for the non-tuned parameters. The similarity of these result should not be too surprising, since the optimal a ended up being a=1, which is what was used prior to tuning. We also note that the overall performance of the SVM function using a polynomial kernel is extremely good, and very close to the performance we found when using the radial basis kernel. Both of these kernels are thus superior than the linear kernel with regards to accuracy of classification.

We have included a plot SVM(x) vs. Poly(x) for the data, with quadrants that correspond to the four cells of the confusion matrix. The first quadrant shows cases where the data were in class CL(+1) and were classified as positive by the SVM classifier with a polynomial kernel, while the third quadrant shows cases where the data were in class CL(-1) and were classified as negative by the SVM classifier. The 2nd and 4th quadrants, conversely, correspond to erroneous classifications. The points that are in the first and third quadrants correspond to correct classifications, and therefore contain almost 100% of the data. We can see by inspection that this is true, which corroborate the performance rate we have calculated, and tells us that our seperator is very good.

It is worth noting that in all of the above SVM models, the performance on the test set is very similar (within a few percentage points) to the performance on the training set. This phenomemon may be explained by the method which these data were generated, namely, a uniform distribution. Because the data are uniformly distributed with the small range [-2,2], the randomly-split test and training sets are likely to be very similar. Furthermore we need not be surprised that the polynomial kernel gave us such a good separator: a polynomial of degree 4 is a good separator of our data because our data were originally generated by a polynomial of degree 4!