Group Project

March 13, 2016

Rodrigo Hernandez
Muzamil Syed
Mayra Gamboa
CS178

Introduction to Our Technique

The approach we will be taking will have to do with going into detail with Ensembles. Instead of having an ensemble of only one type of learner, we decided to have an ensemble with three different types of learners. These learners will be decision trees, K-Nearest-Neighbours, and support vector machines.

We will each choose our own type of learner and perform our own experiments to try and maximize the accuracy of our learner's predictions. After we have chosen our optimal learners, we will combine them into an ensemble of size 24 (8 learners of each type) and predict the test data using this new ensemble.

We will use existing packages provided by sklearn library. We will use the methods provided by the library and explore additional techniques to supplement these methods to vary the complexities of the models that result.

In [1]:
```
##########Libraries Used Throughout The Code:##########

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import mltools as ml
import mltools.dtree as dtree
import mltools.logistic2 as lcs2
import sklearn
from sklearn import svm
from sklearn import preprocessing
%matplotlib inline
```

We will used the provided Kaggle data in our class Kaggle Competition.

In [2]:
```
##########Imported Data:##########
# Get Kaggle training data
X = np.genfromtxt("data/kaggle.X1.train.txt",delimiter=",")
Y = np.genfromtxt("data/kaggle.Y.train.txt",delimiter=",")

# also load features of the test data (to be predicted)
Xe1 = np.genfromtxt("data/kaggle.X1.test.txt",delimiter=",")

perSplit = 0.8 # Percent at which to split the training data
               # (e.g 0.8 = 80/20 split)

Xtr,Xte,Ytr,Yte = ml.splitData(X,Y,0.8)
```

Since the Kaggle data has 91 dimensions, one of the type of learners we chose was a support vector machine, as they perform well in high dimensionality. First, we tested different types of kernels to find which one performed the best. It is obvious that linear kernel was not going to estimate the data correctly, so a linear kernal was not tested.

```
In [3]: # First we scale the data for the SVM
        XiTe = preprocessing.scale(Xte)
        XiTr = preprocessing.scale(Xtr)
```

```
In [4]: # Test accuracy of each type of SVM
        kernels = ['rbf','sigmoid','poly']

        for k in kernels:
            clf = svm.SVR(kernel=k)
            clf.fit(XiTr[:10000,],Ytr[:10000])

            YhatTrain = clf.predict(Xtr)
            YhatTest = clf.predict(Xte)

            MSEtrain = np.mean((Ytr - YhatTrain)**2)
            MSEtest = np.mean((Yte - YhatTest)**2)

            print("SVM with {} as kernal".format(k))
            print("\tMSE of training data: " + str(MSEtrain))
            print("\tMSE of test data: " + str(MSEtest))
```

```
SVM with rbf as kernal
        MSE of training data: 0.694313336835
        MSE of test data: 0.720472853771
SVM with sigmoid as kernal
        MSE of training data: 0.786630556699
        MSE of test data: 0.821199078992
SVM with poly as kernal
        MSE of training data: 7.17265890208e+27
        MSE of test data: 7.2573234728e+27
```

Based on the data, the RBF kernel had the best performance. Now, since SVM can take a long time with large amounts of data, we will see how long an SVM with a RBF kernel takes with different subsets of data and it's performance on said data.

```
In [28]: import time
         clf = svm.SVR(kernel='rbf')
         Xi = preprocessing.scale(X)
         for i in [5000,10000,20000,40000,60000]:
             print("Data for {} data points".format(i))
             t0 = time.time()
             clf.fit(Xi[:i,],Y[:i])
             print("\tTraining: {:.2f} seconds".format(time.time()-t0))
             t0 = time.time()
             Yhat = clf.predict(X[:i,])
             print("\tPredicting: {:.2f} seconds".format(time.time()-t0))
             MSE = np.mean((Y[:i] - Yhat)**2
             print("\tMSE of data: {:.2f}".format(MSE))
```

```
Data for 5000 data points
        Training: 3.24 seconds
        Predicting: 1.79 seconds
```

```
        MSE of data: 0.71
Data for 10000 data points
        Training: 12.60 seconds
        Predicting: 6.96 seconds
        MSE of data: 0.70
Data for 20000 data points
        Training: 180.33 seconds
        Predicting: 28.04 seconds
        MSE of data: 0.70
Data for 40000 data points
        Training: 890.83 seconds
        Predicting: 112.21 seconds
        MSE of data: 0.69
Data for 60000 data points
        Training: 3303.78 seconds
        Predicting: 245.58 seconds
        MSE of data: 0.70
```

Based on these results, it is best to train on a subset of the data of size 20000 since the accuracy is not improved much after 20000 data points but is taking much longer. This number will be used when training the final data in the ensemble.

Since the SVM learner seems to be underfitting, I have decided to try and increase the value of the penalty (parameter C) and increasing the value of gamma, the kernel coefficient, to try and increase the complexity to see if it would do better. The following are the result of the test:

Data for C=0.1
MSE for training data: 0.61558137485
MSE for test data: 1.33784400503
Data for C=0.5
MSE for training data: 0.537344845127
MSE for test data: 1.19044021805
Data for C=1
MSE for training data: 0.497589581634
MSE for test data: 1.06304257652
Data for C=20
MSE for training data: 0.393138627212
MSE for test data: 0.822942582304
Data for C=50
MSE for training data: 0.403543750388
MSE for test data: 0.829147117563
Data for gamma=0.1
MSE for training data: 0.665729419872
MSE for test data: 1.23645277441
Data for gamma=0.5
MSE for training data: 0.728801622922
MSE for test data: 1.2831635786
Data for gamma=1
MSE for training data: 0.729558885724
MSE for test data: 1.28298434064
Data for gamma=10
MSE for training data: 0.729791044544
MSE for test data: 1.28297833932
Data for gamma=20
MSE for training data: 0.729791076776
MSE for test data: 1.28297833925

Changing these parameters did not do much to make the learner better. Since it seems to be underfitting, we tried to make an AdaBoost ensemble of SVMs to try and remedy the fact that it is underfitting.

```
In [ ]: # Scale data for learner
        group = ensemble.AdaBoostRegressor(svm.SVR(kernel='rbf',C=20),n_estimators=8)

        group.fit(Xi[:20000,],Yi[:20000])

        YhatTe = group.predict(Xte)

        print("MSE for test data: {}".format(np.mean((Yte-YhatTe)**2)))
```

This resulted in an MSE of 0.807 for the test data. Since we were not able to improve on the SVM model, we decided to drop it entirely on our final ensemble.