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.

Rodrigo Hernandez
Muzamil Syed
Mayra Gamboa
CS178

Group Project</b>

In [6]:
```
##########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 tree
%matplotlib inline
#Imported Library for Neural Network
#Imported Library for Third Learner
```

In [9]:
```
##########Imported Data:##########
import numpy as np
'''
We will use the provided Class Kaggle Data in our class Kaggle Competi
tion: CS178 Project 2016
https://inclass.kaggle.com/c/cs178-project-2016
'''
# 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)

print(X.shape)
```

```
(60000, 91)
```

In [3]:
```
##########Initialization of the Ensemble:##########
'''
We will start with an Ensemble of size 20...
'''
# Ensemble Variables
size = 20  # the amount of learners in the ensemble
features = 55 # the number of features to select from when bagging

# Create the ensemble
final_ensemble = [None] * size
```

Here we will be learning on decision trees. We try to tackle the problem of overfitting, which occurs with overcomplex trees. These trees can become unstable because small variations in the data might result in a completely different tree being generated. Luckily, we can avoid this problem by either setting the maximum depth of the tree or setting the minimum number of samples required at a leaf node. We proceed to learning a decision tree regressor on the data, and specifying a maximum depth of 20. We predict on the training data and the testing data and obtain the mean squared averages for both data sets. The mean squared averages are displayed as follows. Next, we specify a variety of maximum depths for the decision trees, ranging anywhere from 1 - 19. We want to find out how adjusting the maximum depths changes the complexities of the trees, and when they begin to overfit. Which maximum depth best handles the problem of overfitting? For each maximum depth, we learn a decision tree regressor and calculate the mean squared average of the training data we predicted on, and also calculate the mean squared average of the testing data we predicted on.

In [10]:
```
##########Code for the Decision Tree:##########
#Imported Library for Decision Trees
from sklearn import tree

#Decision Tree Learner INFO...
# Decision Tree Variables
depth = 20 # the maxth depth of the decision tree
nodes = 8 # the minimum number of data to split node

#learn a decision tree regressor on data, specify max depth of 20
learner = tree.DecisionTreeRegressor(max_depth=20)
learner.fit(Xtr, Ytr)
YhatTrain = learner.predict(Xtr)
YhatTest = learner.predict(Xte)
MSETrain = np.mean((Ytr - YhatTrain)**2)
MSETest = np.mean((Yte - YhatTest)**2)
print('{}{}'.format("MSE for training data: ", MSETrain))
print('{}{}'.format("MSE for testing data: ", MSETest))

#try different ranges of maximum depths
for depth in range(20):
    learner = tree.DecisionTreeRegressor(max_depth = depth+1)
    learner.fit(Xtr, Ytr)
    Yhat_train = learner.predict(Xtr)
    Yhat_test = learner.predict(Xte)
    mseTrain = np.mean((Ytr - Yhat_train)**2)
    mseTest = np.mean((Yte - Yhat_test)**2)
    print("Depth {:02d} --> mse train: {}, mse validation: {}".format(
depth+1, mseTrain, mseTest))
```

```
MSE for training data: 0.0357911570162
MSE for testing data: 0.712112171593
Depth 01 --> mse train: 0.557937953351, mse validation: 0.57414551383
Depth 02 --> mse train: 0.505146872922, mse validation: 0.519822936078
Depth 03 --> mse train: 0.472694791396, mse validation: 0.483718589446
Depth 04 --> mse train: 0.451999080531, mse validation: 0.465591656337
Depth 05 --> mse train: 0.434261655119, mse validation: 0.455494004267
Depth 06 --> mse train: 0.418033072227, mse validation: 0.446586295059
Depth 07 --> mse train: 0.397270867629, mse validation: 0.434863566073
Depth 08 --> mse train: 0.377611804684, mse validation: 0.438300262859
Depth 09 --> mse train: 0.353281895038, mse validation: 0.443817028614
Depth 10 --> mse train: 0.32512500598, mse validation: 0.467904685154
Depth 11 --> mse train: 0.2920177674, mse validation: 0.487800548445
Depth 12 --> mse train: 0.25566965481, mse validation: 0.525525238568
Depth 13 --> mse train: 0.216916242863, mse validation: 0.555434048757
Depth 14 --> mse train: 0.178732828402, mse validation: 0.586412749513
Depth 15 --> mse train: 0.143080936773, mse validation: 0.61639887708
Depth 16 --> mse train: 0.11355919281, mse validation: 0.638016511422
Depth 17 --> mse train: 0.0871032692009, mse validation: 0.67195780753
7
Depth 18 --> mse train: 0.0657788229549, mse validation: 0.68580835288
5
Depth 19 --> mse train: 0.0492075353514, mse validation: 0.70996072750
5
Depth 20 --> mse train: 0.0357897739192, mse validation: 0.71208041707
4
```

We stick to specifying a maximum depth of 20, and proceed to learning on decision trees with that fixed depth, and we adjust the next parameter vital to creating the decision trees we want. The next parameter we adjust is the minimum number of samples required at a leaf node, otherwise known as the min_samples_leaf parameter. We learn on a range from 2^3, up to 2^12 minimum number of samples required at a leaf node. We predict on the training data and on the testing data, and we calculate their mean squared averages. The mean squared averages of the training and testing data are displayed as follows.

In [5]:
```
for nodes in range(3, 13):
    learner = tree.DecisionTreeRegressor(max_depth = 20, min_samples_l
eaf = 2**nodes)
    learner.fit(Xtr, Ytr)
    Yhat_train = learner.predict(Xtr)
    Yhat_test = learner.predict(Xte)
    mseTrain = np.mean((Ytr - Yhat_train)**2)
    mseTest = np.mean((Yte - Yhat_test)**2)
    print("2^{} data at leaf node --> mse train: {}, mse validation: {
}".format(nodes, mseTrain, mseTest))
```

```
2^3 data at leaf node --> mse train: 0.159069620689, mse validation: 0
.59296293253
2^4 data at leaf node --> mse train: 0.238927974161, mse validation: 0
.515825198467
2^5 data at leaf node --> mse train: 0.303901057014, mse validation: 0
.457419046287
2^6 data at leaf node --> mse train: 0.348795257147, mse validation: 0
.42866571325
2^7 data at leaf node --> mse train: 0.376680650869, mse validation: 0
.424888119888
2^8 data at leaf node --> mse train: 0.398765237031, mse validation: 0
.429117852099
2^9 data at leaf node --> mse train: 0.414565160586, mse validation: 0
.438279933896
2^10 data at leaf node --> mse train: 0.431358990435, mse validation:
0.450114946475
2^11 data at leaf node --> mse train: 0.452946540527, mse validation:
0.46886867678
2^12 data at leaf node --> mse train: 0.47046962598, mse validation: 0
.483359063336
```

We decide to choose 2^8 minimum number of samples required at each leaf node, based on the results of the mean squared averages from the training and testing data.

Now we proceed to storing our individual learners in an ensemble. We set the size of the ensemble to 20 because we will be storing 20 different learners within the ensemble. We will store the KNN and the decision tree learners in the ensemble. Unfortunately, we will not be able to store the Support Vector learners in this ensemble. We create 10 different decision tree learners. Using a maximum depth of 20 and 8 minimum samples required at each leaf node as our favorite parameters, we add each of the learners to the ensemble of different learners. Finally, we proceed to storing our individual learners in an ensemble. We set the size of the ensemble to 20 because we will be storing 20 different learners within the ensemble. We will store the KNN and the decision tree learners in the ensemble. Unfortunately, we will not be able to store the Support Vector learners in this ensemble.

In [ ]:
```
##########Code to store each learner in the ensemble:##########
from sklearn.ensemble import BaggingRegressor

ensemble = [None]*10 #for now store 10 learners in ensemble before com
bining all learners
# Create learners and add to dtree ensemble
#dtLearners = BaggingRegressor(tree.DecisionTreeRegressor(max_depth =
20, min_samples_leaf = 2**8))
M = Xtr.shape[0]
Me = Xte.shape[0]
YtrHat = np.zeros((M,10))
YteHat = np.zeros((Me,10))
for l in range(10): #add 10 decision tree learners to the ensemble
    Xi, Yi = ml.bootstrapData(Xtr, Ytr, M)
    ensemble[l] = BaggingRegressor(tree.DecisionTreeRegressor(max_dept
h=20,min_samples_leaf = 2**8))
    ensemble[l].fit(Xi, Yi)
    Ythat = ensemble[l].predict(Xtr)
    Yehat = ensemble[l].predict(Xte)
```

In [3]:
```
##########Code for the KNN Learner:##########
from sklearn.neighbors import KNeighborsRegressor  #Imported Library for KNN
Regressor
from sklearn.ensemble import BaggingRegressor

#print Xtr.shape, Ytr.shape

#Create a list to store the top KnnRegressor Performers
topKnnPerformers = [None]*10

#Predict and print the MSE for training and test data on knnRegressors from 1
to 20 Nearest Neighbors
for K in range(1, 21):
    knnRegressor = KNeighborsRegressor(n_neighbors = K)
    yhatTr = knnRegressor.fit(Xtr, Ytr).predict(Xtr)
    yhatVa = knnRegressor.fit(Xte, Yte).predict(Xte)
    print "MSE for K = {:>2} Neighbors     Error(Training) = {:>7.4f}     Err
or(Validation) = {:>7.4f}".format(
            K, np.mean((Ytr - yhatTr)**2), np.mean((Yte - yhatVa)**2))

    #Since the first 10 regressors
    if (K <= 10):
        topKnnPerformers[K-1] = knnRegressor
```

```
MSE for K =  1 Neighbors     Error(Training) =  0.0000     Error(Validation)
=  0.0000
MSE for K =  2 Neighbors     Error(Training) =  0.1863     Error(Validation)
=  0.2122
MSE for K =  3 Neighbors     Error(Training) =  0.2555     Error(Validation)
=  0.2805
MSE for K =  4 Neighbors     Error(Training) =  0.2932     Error(Validation)
=  0.3197
MSE for K =  5 Neighbors     Error(Training) =  0.3175     Error(Validation)
=  0.3458
MSE for K =  6 Neighbors     Error(Training) =  0.3352     Error(Validation)
=  0.3676
MSE for K =  7 Neighbors     Error(Training) =  0.3486     Error(Validation)
=  0.3826
MSE for K =  8 Neighbors     Error(Training) =  0.3578     Error(Validation)
=  0.3965
MSE for K =  9 Neighbors     Error(Training) =  0.3655     Error(Validation)
=  0.4033
MSE for K = 10 Neighbors     Error(Training) =  0.3719     Error(Validation)
=  0.4087
MSE for K = 11 Neighbors     Error(Training) =  0.3771     Error(Validation)
=  0.4145
MSE for K = 12 Neighbors     Error(Training) =  0.3812     Error(Validation)
=  0.4190
MSE for K = 13 Neighbors     Error(Training) =  0.3855     Error(Validation)
=  0.4221
MSE for K = 14 Neighbors     Error(Training) =  0.3895     Error(Validation)
=  0.4259
MSE for K = 15 Neighbors     Error(Training) =  0.3928     Error(Validation)
=  0.4289
MSE for K = 16 Neighbors     Error(Training) =  0.3964     Error(Validation)
=  0.4311
MSE for K = 17 Neighbors     Error(Training) =  0.3993     Error(Validation)
```

```
= 0.4340
MSE for K = 18 Neighbors     Error(Training) =  0.4020     Error(Validation)
= 0.4370
MSE for K = 19 Neighbors     Error(Training) =  0.4041     Error(Validation)
= 0.4393
MSE for K = 20 Neighbors     Error(Training) =  0.4062     Error(Validation)
= 0.4417
```

In [7]:

```
print(topKnnPerformers) #checking to see if the first 10 knn neighbors were s
tored in the list
```

```
[KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=1, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=2, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=3, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=4, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=6, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=7, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=8, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=9, p=2,
          weights='uniform'), KNeighborsRegressor(algorithm='auto', leaf_size
=30, metric='minkowski',
          metric_params=None, n_jobs=1, n_neighbors=10, p=2,
          weights='uniform')]
```

Using the KNeighborsRegressor from the sklearn library a knnregressor learner was constructed with neigbors ranging from 1 to 20. It was expected, as the commentary mentioned on the project writeup that the KNN learners would take some time to train and predict. The process was ended up being fast (30 minutes - 1 hour) and the resulting MSE was very low giving us good results for the FULL data set. As the results are seen above, the error on the training data was < ~.41 and < ~.44 for the test data. Because the model starts underfitting for larger values of K we decided to move forward with neighbors less than K = 10, meaning taking the top 10 performers to store into the ensemble.

```
In [ ]: topEnsemblePerformers = [None]*10

        for i, regressor in enumerate(topKnnPerformers):

            knnEnsemble = BaggingRegressor(base_estimator = topKnnPerformers[i], n_es
        timators = 10)
            yhatTr = knnEnsemble.fit(Xtr, Ytr).predict(Xtr)
            yhatVa = knnEnsemble.fit(Xte, Yte).predict(Xte)

            print "MSE for Ensemble with a K = {:>2} Neighbors Regressor     Error(Tr
        aining) = {:>7.4f}     Error(Validation) = {:>7.4f}".format(
                    regressor.n_neighbors, np.mean((Ytr - yhatTr)**2), np.mean((Yte -
        yhatVa)**2))
```

```
MSE for Ensemble with a K = 1 Neighbors Regressor Error(Training) = 0.0955 Error(Validation) =
0.1055
MSE for Ensemble with a K = 2 Neighbors Regressor Error(Training) = 0.1729 Error(Validation) =
0.1936
MSE for Ensemble with a K = 3 Neighbors Regressor Error(Training) = 0.2359 Error(Validation) =
0.2596
```

Playing around with the BaggingRegressor library, there was no way to store the 10 topKnnPerformers into one Ensemble so we tested storing 10 of each topKnnPerformers and train/predicted to see the results. We found that since it was taking too long to output the MSE and this approach just did not make sense, so we did not move forward with it. However, it is worth noting the drop in MSE for the 3 outputs that did end up printing as shown above. Afterwards, reading through library documentation we moved forward with another approach below.

In [18]:
```
knnEnsemble = BaggingRegressor(n_estimators = len(topKnnPerformers))
knnEnsemble.estimators_ = topKnnPerformers
yhatTr = knnEnsemble.fit(Xtr, Ytr).predict(Xtr)
yhatVa = knnEnsemble.fit(Xte, Yte).predict(Xte)

print "MSE for Ensemble with top 10 KnnRegressors     Error(Training) = {:>7.4f}     Error(Validation) = {:>7.4f}".format(
        np.mean((Ytr - yhatTr)**2), np.mean((Yte - yhatVa)**2))
```

```
MSE for Ensemble with top 10 KnnRegressors     Error(Training) =  0.0719
Error(Validation) =  0.0773
```

MSE for Ensemble with top 10 KnnRegressors Error(Training) = 0.0719 Error(Validation) = 0.0773

Since BaggingRegressors library had an attribute to store subset of estimators, we decided to take our list of topPerformers trained previously and store this list to that attribute. This allowed us to train and predicted on the entire list of learners rather than having to indivudally store them in a BaggingRegressor with 10 copies of each knnRegressor. After training and predicting on this entire list of already well performing learners, the result was an even lower MSE!

In [9]:
```
#Combining the Ensemble

#To output to Kaggle
#Based off of our results above we will use the optimum values given out by t
he knnEnsemble above as well as
#store the optimum Decision Tree leaf value of 2^8 into its own Ensemble repr
edict and average it with knnEnsemble prediction

avgPredTeList = []

optimumKnnEnsemble = BaggingRegressor(n_estimators = len(topKnnPerformers))
optimumDtEnemble = BaggingRegressor(
    base_estimator = tree.DecisionTreeRegressor(max_depth = 20, min_samples_l
eaf = 2**8), n_estimators = 10)

KnnTr = optimumKnnEnsemble.fit(Xtr, Ytr)
DtTr = optimumDtEnemble.fit(Xtr, Ytr)
```

The code above takes the Knn values based on our previous test runs earlier of KNearestNieghbors of up to 10 as well as the optimum Decsion Tree Ensemble based on our previous findings of min_samples_leafs @ 2^8 and retrained aftering storing each to their own Ensemble.

In [15]:
```
yhatKnnTr = KnnTr.predict(Xe1)
yhatDtTr = DtTr.predict(Xe1)

avgPredTeList = (yhatKnnTr + yhatDtTr) / 2
print(len(avgPredTeList))

fh = open('predictions.csv','w') # open file for upload
fh.write('ID,Prediction\n') # output header line
for i,yi in enumerate(avgPredTeList):
    fh.write('{},{}\n'.format(i+1,yi)) # output each prediction

fh.close() # close the file
```

```
40000
```

Because these were our top learners we decided to take the predictions of both the Decision Tree Ensemble and the K nearest Ensemble and average these values to upload to Kaggle. Because of the large amount of time it takes to build these learners and the Ensemble version of these learners, we were only able to upload our predictions once and not reupload with improved prediction quality.

Conclusion

In conclusion, since our document ipython notebook size was exceeding far beyond 6 pages and we were not sure if our writeup was suppose to be in a seperate document, we decided to do our write up as we were writing and testing our code (as shown above). At the beginning we had decided to go more into detail with Ensembles by using learners that we did not go into detail with in class. This enabled us to use 3 different types of learners to equally balance the workload. Rodrigo, started with the intent of using Neural Networks until he realized that they only work with classification models. He decide to switch to Support Vector Machines, also a learner we did not apply in our homeworks. Because the Support Vector Machine took too long to train on the entire dataset, a smaller subset was used. The result of predicting on these smaller subset were very high MSEs for both the training and test data. Because the SVM was constantly underfitting and unable to improve we came to the decision of removing it from the final ensemble. The decision tree, the learner tested by Mayra, had mixed performances. Aftering training and prediction on the decision tree using the library it looked as the depth increased the MSE decreased. Iterating thru different values of the leaf_node parameter, Mayra found that the decision tree at leaf node 2^8 had the optimum learner so we decided to store this learner into the final ensemble. The third learner, explored by Muzamil, was going into detail with K Nearest Nieghbors, but as apposed to using classifiers as in the previous homework, he used the library's knnRegressor model. He decided to train the KNN mode of up to 20 neighbors and the performance, as seen above, turned out to be very good. It was noticed, as expected that the more neighbors you have the more the model was starting to underfit as seen in the increase in the MSEs. He decided to take knnRegressors from 2-10 neighbors to store into the Ensemble. At first an attempt was made to try to use these 10 different learners in the baggingRegressor library but the only way to do this was store multiple of the same learners into baggingRegressor meaning he would have had to create 10 different ensembles to train on. After looking more into the library an attribute in the baggingRegressor was discovered where a list of sublearners can be set. He decided to take this approach and train the ensemble on the list of 10 KnnRegressors of up to 10 neighbors. The training and predicted process ended up being quick and the MSE was even less! We decided to take the predictions along with the optimum Decision Tree learner to upload to Kaggle. Overall, our final goal of creating an emsemble containing 7 or so of each our learners did not go as planned so we decided to choose the 2 learners and use 10 of each of those. There were many times that we could not see the results of our trainers due to very long run times. If something were to be changed in order to get the results we intended, we could probably use partitions of the dataset whenever possible.