Rodrigo Hernandez
Muzamil Syed
Mayra Gamboa
CS178

Group Project

Introduction to Our Technique

The approach we will be taking will have to do with going into detail with Ensembles. We will each choose our own learning techniques and individually train on the provided data set. We will use multiple instances of each of our learners to store the ensemble??? This will be done using 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.

Questions Are we bagging (bootstrap) where we use part of the data? To prevent overfitting we are using enhancement techniques for each our learners?

In [11]:
```
#Report Format: (Maybe me we could have some kind of title page added
to final pdf with our names)
```

In [5]:
```
##########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 [6]:
```
##########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 [12]:
```
##########Initialization of the Ensemble:##########

'''
We will start with an Ensemble of size 25...
'''

# Ensemble Variables
size = 25  # the amount of learners in the ensemble
features = 55 # the number of features to select from when bagging

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

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

'''
KNN Learner INFO...
-describe problem you use chose and methods to address it
-how did you train the models?
-how you selected any parameters each model/method requires
-how they performed on test data
-consider table of performance of different approaches or plots of perofrmance used to perform model selection
'''

'''
Key Points:
-Explore some aspect of prediction that we have not already done in depth
-Identify a paper that proposes a method you think could be helpful
-Use stacking/information from your leaderboard performance to try and improve your prediction quality
-To explore approach: explore method from class fully enough to understand how changes might affect its performance,
verify your findings make sense, and then use your findings to optimize performance
-In Your Report: describe why you decided to explore this aspect, what you expected to find, and how your findings
matched/ didnt match your expectations
-Beware of the positive/negative aspects of the learners we discussed ie Nearest Neighbor methods can be powerful but can
also be slow for large data sets...perhaps you can reduce the data in some way without sacrificing performance (bootstrap
aggregation)
-Linear methods may be fast but do not provide enough model complexity to provide a good fit so you may been to try and
generate better features
'''
```

In [ ]:
```
##########Code for the Support Vector Machines:##########

'''
Support Vector Machine INFO...
'''
Xi,Yi = ml.utils.bootstrapData(Xtr,Ytr)
clf = svm.SVR(kernel='poly',degree=2,cache_size=7000)
#print(Xi.shape)

A = Xi[0:30,0:55]
B = Yi[0:30]
print(Yi.shape)
# SVMensemble = sklearn.ensemble.BaggingRegressor(clf,7,max_samples=25
,
#  max_features=features)

clf.fit(A,B)
print(clf.predict(Xte[0:2,0:55]))
# SVMensemble.fit(Xtr,Ytr)
# SVMensemble.predict(Xte[0,:])

#Imported Library for Neural Network
```

In [13]:
```
##########Code for the Decision Tree:##########
#Imported Library for Decision Trees
'''
Decision Tree Learner INFO...
'''

'''
Here we will be learning on decision trees. We try to tackle the probl
em 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.
'''

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

'''
We proceed to learning a decision tree regressor on the data, and spec
ifying a maximum depth of
20. We predict on the training data and the testing data and obtain th
```

```
e mean squared averages
for both data sets. The mean squared averages are as follows:
'''
#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))
print('\n')

'''
Now, we will specify a variety of maximum depths for the decision tree
s, ranging anywhere from
1 - 19. We want to find out how adjusting the maximum depths changes t
he complexities of the
trees, and when they begin to overfit. Which maximum depth best handle
s the problem of overfitting?
For each maximum depth, we learn a decision tree regressor and calcula
te the mean squared average
of the training data we predicted on, and also calculate the mean squa
red average of the testing
data we predicted on.
'''
for depth in range(19):
    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))
print('\n')

'''
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 de
cision 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 an
d on the testing data, and we
calculate their mean squared averages. The mean squared averages of th
```

```
e training and testing data
are as follows:
'''
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))

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

```
MSE for training data: 0.0357911227183
MSE for testing data: 0.726991388748


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: 0.43521071237
Depth 08 --> mse train: 0.377611804684, mse validation: 0.440208874745
Depth 09 --> mse train: 0.353281895038, mse validation: 0.446513148684
Depth 10 --> mse train: 0.32512500598, mse validation: 0.464664335836
Depth 11 --> mse train: 0.2920177674, mse validation: 0.484219951949
Depth 12 --> mse train: 0.25566965481, mse validation: 0.514424668305
Depth 13 --> mse train: 0.216916698559, mse validation: 0.55258222697
Depth 14 --> mse train: 0.178733837642, mse validation: 0.580236633767
Depth 15 --> mse train: 0.143084770324, mse validation: 0.621130636606
Depth 16 --> mse train: 0.113559591856, mse validation: 0.645537962467
Depth 17 --> mse train: 0.0871034107474, mse validation: 0.66406547785
8
Depth 18 --> mse train: 0.0657787643252, mse validation: 0.68721879181
1
Depth 19 --> mse train: 0.0492076079306, mse validation: 0.69837227959
7


2^3 data at leaf node --> mse train: 0.159068293718, mse validation: 0
.593354378406
2^4 data at leaf node --> mse train: 0.238927974161, mse validation: 0
.515669493421
2^5 data at leaf node --> mse train: 0.303901057014, mse validation: 0
.457412998062
2^6 data at leaf node --> mse train: 0.348795257147, mse validation: 0
.428665778034
2^7 data at leaf node --> mse train: 0.376680650869, mse validation: 0
.424888196308
2^8 data at leaf node --> mse train: 0.398765237031, mse validation: 0
.42911833535
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
```

Out[13]: '\nWe decide to choose 2^8 minimum number of samples required at each leaf node, based on the results\nof the mean squared averages from the training and testing data. \n'

In [ ]:
```
##########Code to output the predictions and evaluate them on kaggle:#########

# Test correctness of ensemble through MSE
mTest = Xte.shape[0] # Acquire the shape of the test data
Yhat = np.zeros((mTest,num))
MSE = 0

for i in range(size):
    Yhat[:,i] = ensemble[i].predict(Xte).reshape(mTest)

    Yhat = np.mean(Yhat,axis=1)

    MSE = np.mean((Yte - Yhat.reshape(Yte.shape))**2,axis=0)

print(MSE)

'''
Note:
-Should not try to upload every possible model with every possible parameter setting
-Use validation data, or cross-validation to assess which models are worth uploading, and just use the uploads
to verify performance.
'''
#Ye = learner.predict( Xeval ); # make predictions
# Note: be sure Ye is a flat vector, shape (m,)
# otherwise, reshape it using e.g.
# Ye = Ye.ravel()
# or change the indexing in the code below:
fh = open('predictions.csv','w') # open file for upload
fh.write('ID,Prediction\n') # output header line
for i,yi in enumerate(Ye):
fh.write('{},{}\n'.format(i+1,yi)) # output each prediction
fh.close() # close the file
```

In [ ]:
```
##########Code to store each learner in the ensemble:##########
from sklearn.ensemble import BaggingRegressor
'''
How should we store the learners we come up with? Since right now we have an ensemble of size 25 we could maybe
create a list for each of our learners and bring them here then index each list of learners, train it and then store
it into the ensemble before making all the predictions. What do you guys think?
```

```
'''

'''
Now we proceed to storing our individual learners in an ensemble. We s
et the size of the ensemble
to 25 because we will be storing 25 different learners within the ense
mble. 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.
'''

ensemble = [None]*25
# Create learners and add to ensemble
#dtLearners = BaggingRegressor(tree.DecisionTreeRegressor(max_depth =
20, min_samples_leaf = 2**nodes))

'''
We proceed to creating 8 different decision tree learners, we might cr
eate more....
Using a maximum depth of 20 and 8 minimum samples required at each lea
f node as our favorite
parameters, we add each of the learners to the ensemble of different l
earners.
'''
M = Xtr.shape[0]
Me = Xte.shape[0]
YtrHat = np.zeros((M,8))
YteHat = np.zeros((Me,8))
for l in range(8): #add 8 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)

'''
For each learner in the ensemble, we predict on the training data and
the testing data.
'''


#for i in range(size):

    #for learners in each list

        #get KNN learner
        #train KNN learner
        #ensemble[i] = KNN Learner
```

```
        #get NN learner
        #train NN learner
        #ensemble[i] = NN Learner

        #get DT learner
        #train DT learner
        #ensemble[i] = DT Learner

    #dt = dtree.treeRegress()
    #Xi,Yi = ml.utils.bootstrapData(Xtr,Ytr)
    #dt.train(Xi,Yi,maxdepth=depth,nFeatures=features,minParent=nodes)

    #ensemble[i] = dt
```

Conclusion

Here we can probably summarize our results and the learners we were responsible for.

In [ ]: