Project_3

April 23, 2020

1 Project 3

1.1 A

[77]:
```
import pandas as pd
import numpy as np
import subprocess as sp

sp.call('clear', shell = True)

df = pd.DataFrame(np.array([[0.5,-1],[3.0,-1],[4.5,1],[4.6,1],[4.9,1],[5.
↪2,-1],[5.3,-1],
                  [5.5,1],[7.0,-1],[9.5,-1]]), columns = ['x','y'])

#Standard Manhattan distance
df['Distance'] = np.abs(df.iloc[:,0] - 5.0)
df = df.sort_values(by = ['Distance']).reset_index()
del(df['index'])

neighbors = [1,3,5,9]
labels = ["1-nn","3-nn","5-nn","9-nn"]
answers = []

for neighbor in neighbors:
    sign = 0
    for i in range(neighbor):
        sign += df.loc[i,'y']
    if sign <0:
        answers.append(-1)
    else:
        answers.append(1)

answers = pd.DataFrame(answers, index = labels, columns = ['Classification'])

#Repeated with the Distance Weighting approach
df['Weighted Distance'] = 1/np.square(np.abs(df.iloc[:,0] - 5.0))
weighted_answers = []
for neighbor in neighbors:
```

```
    sign = 0
    for i in range(neighbor):
        sign += df.loc[i,'y'] * df.loc[i,'Weighted Distance']
    if sign < 0:
        weighted_answers.append(-1)
    else:
        weighted_answers.append(1)
answers['Weighted Classification'] = np.asarray(weighted_answers)
display(df)
display(answers)
```

	x	y	Distance	Weighted Distance
0	4.9	1.0	0.1	100.000000
1	5.2	-1.0	0.2	25.000000
2	5.3	-1.0	0.3	11.111111
3	4.6	1.0	0.4	6.250000
4	4.5	1.0	0.5	4.000000
5	5.5	1.0	0.5	4.000000
6	3.0	-1.0	2.0	0.250000
7	7.0	-1.0	2.0	0.250000
8	0.5	-1.0	4.5	0.049383
9	9.5	-1.0	4.5	0.049383

	Classification	Weighted Classification
1-nn	1	1
3-nn	-1	1
5-nn	1	1
9-nn	-1	1

1.2 B

[78]:
```
import pandas as pd
import numpy as np
import subprocess as sp
from pprint import pprint

sp.call('clear', shell = True)
#Functions for calculating variables in the program
#Calculate the updated probabilities each epoch
def probability_next(error,probability,classified):
    if classified == False:
        probability_next = probability/(2 * (error + epsilon))
    else:
        probability_next = probability/(2 * (1 - error + epsilon))
    return probability_next
#Calculate the weight for each individual model
```

```
def calculate_weight(error):
    #When error = 0, alpha is approximately 18, which is very large. And it␣
↪should be larger than the other weights, since that specific
    #model was perfect. However, I wrote this if statement when I thought an␣
↪alpha as large as 18 might be skewing the results. I didn't
    #see any significant change in the final results, so I commented it out.
    #if error == 0:
    #    alpha = 2
    #    return alpha

    alpha = 0.5 * np.log((1 - error + epsilon)/(error + epsilon))
    return alpha
#Calculate the error each epoch
def calculate_error(probabilities):
    error = np.sum(probabilities)
    return error
#Calculate the entropy
def calculate_entropy(df):
    Class = df.keys()[-1]
    entropy = 0
    targets = df[Class].unique()
    for target in targets:
        probability = df[Class].value_counts()[target]/len(df[Class])
        entropy += -1 * probability*np.log2(probability)
    return entropy
```

[79]:
```
################################################################################
#Below are functions I reused from Project 2. Each of them has been updated in
#some way.
#Finding the optimal splitting point for each level of the decision tree
def choose_node(df,information_gain,splits):
    #Check if all instances have the same class. If they do, take the average
    #as the splitting point.
    uniques = df['Class'].nunique()
    if uniques == 1:
        split = df['x'].mean()
    else:
        for index in df.index:
            #Create a list of splits between the instances
            split = df.loc[index,'x'] + 0.01
            splits.append(split)

            #Dataframes to represent the child nodes created by each split
            child_1 = pd.DataFrame(df[df['x'] <= split])
            child_2 = pd.DataFrame(df[df['x'] > split])

            #Calculate entropy and save information gain to a list
```

```
            weighted_average = (len(child_1)/len(df)) *␣
↪calculate_entropy(child_1) + (len(child_2)/len(df)) * calculate_entropy(df)
            information_gain.append(1 - weighted_average)

        #Choose the splitting condition with the lowest entropy.
        split = splits[np.argmax(np.asarray(information_gain))]

    return split
#Create a child node for each level of the decision tree.
def build_subtree(df,split,child):
    if child == 0:
        return df[df['x'] <= split].reset_index(drop = True)
    else:
        return df[df['x'] > split].reset_index(drop = True)
#Main function for building the decision tree
def build_decision_tree(df,tree = None):
    #Calculate the optimal splitting condition
    split = choose_node(df,information_gain = [],splits = [])
    children = [0,1]

    #Create the framework for a new subtree
    if tree is None:
        tree = {}
        tree[split] = {}
    #This checks if the current training set has the same instance for every
    #row. This only happens about 0.1% of the time, but causes problems once
    #the tree gets passed to the predict function.
    if df['x'].nunique() == 1:
        tree[split][children[0]] = df['Class'].max()
        tree[split][children[1]] = df['Class'].max()
        return tree

    #Decide what to construct for each of the node's children, a leaf or a
    #subtree
    for child in children:
        subtree = build_subtree(df,split,child)
        class_values,counts = np.unique(subtree['Class'],return_counts = True)

        if len(counts) == 1:
            tree[split][child] = class_values[0]
        else:
            tree[split][child ] = build_decision_tree(subtree)

    return tree
#Predict and instance's class using the constructed tree
def predict(inst,tree):
    for nodes in tree.keys():
```

```
        #Determine which child node to select
        if inst['x'] <= nodes:
            value = 0
        else:
            value = 1

        #Take the value of whatever is on the current child node, whether it's
        #a leaf or subtree
        tree = tree[nodes][value]
        prediction = 0

        #Check if you've reached a leaf node or another subtree
        if type(tree) is dict:
            prediction = predict(inst, tree)
        else:
            prediction = tree
            break;

    return prediction
```

In the `choose_node` function, my solution for all instances having the same class is different from the books. Rather than choosing the maximum or minimum split, I took the mean. That's because, later on in the `build_decision_tree` function, I split the list into two new dataframes, one containing values smaller than the split and one containing values larger. If I choose an extremum, I'll end up passing an empty dataframe back to the `choose_node` which then crashes at `split = splits[np.argmax(np.asarray(information_gain))]`.

[80]:
```
#Build training dataframe and calculate original probabilities
data_train = pd.DataFrame(np.array([[0.5,-1],[3.0,-1],[4.5,1],[4.6,1],[4.9,1],
                                    [5.2,-1],[5.3,-1],[5.5,1],[7.0,-1],[9.
 ↪5,-1]])
                          , columns = ['x','Class'])
data_train['Probabilities'] = np.zeros(shape = (len(data_train)))
for instance in data_train.index:
    data_train.loc[instance,'Probabilities'] += 1/(len(data_train))
#Global variables
epochs = 10
epoch = 0
model_weights = []
epsilon = np.finfo(float).eps
boosting_ensemble = {}
```

I chose 10 epochs. There's no trend in error as we add more since the datasets are independent of each other. Also, performing 20 and 100 epochs produced the same results.

[81]:
```
#Here starts the main part for the program
while epoch < epochs:
```

```
#Sample for new training data with the current set of probabilities
new_data_train = data_train.iloc[:,0:2].sample(
    n = len(data_train),
    replace = True,
    weights = data_train['Probabilities']).sort_values(
        by = ['x']).reset_index()
del new_data_train['index']

#Construct decision tree and save it to a dictionary
boosting_ensemble[epoch + 1] = {}
tree = build_decision_tree(new_data_train)
boosting_ensemble[epoch + 1] = tree

#Run the decision tree on the original data set and label each instance
#as to whether it was classified or not
data_train['Correctly Classified'] = np.zeros(shape = len(data_train))
#Run each instance through the prediction function
for index in data_train.index:
    predicted_class = predict(data_train.loc[index,:],tree)
    if data_train.loc[index,'Class'] == predicted_class:
        data_train.loc[index,'Correctly Classified'] = 1
    else:
        data_train.loc[index,'Correctly Classified'] = 0

#Calculate the error and break the loop if the model's error satisfies
#the stopping condition
error = calculate_error(
    data_train['Probabilities'][data_train['Correctly Classified'] == 0])
if error >= 0.5:
    print("Boosting Round ", epoch + 1, ". Error, ", error)
    break

#Calculate the weight for this model
model_weights.append(calculate_weight(error))

#Print original probabilities
print("\nBoosting Round ", epoch +1, "\nProbabilities:\n")
display(data_train[['x','Probabilities']])

#Update the probabilities
for index in data_train.index:
    #Case where the instance was classified
    if data_train.loc[index,'Correctly Classified'] == 1:
        data_train.loc[index,'Probabilities'] = probability_next(
            error = error,
            probability = data_train.loc[index,'Probabilities'],
            classified = True)
```

```python
        #Case where the instance was misclassified
        else:
            data_train.loc[index,'Probabilities'] = probability_next(
                error = error,
                probability = data_train.loc[index,'Probabilities'],
                classified = False)

    #Printing the results
    print("Training Data Set:\n")
    display(new_data_train[['x']])
    print("\nDecision Tree: ")
    pprint(boosting_ensemble[epoch + 1], width = 1)
    print("\nError: ", error)
    print("Model Weight: ", model_weights[-1])
    epoch += 1
```

```
Boosting Round  1
Probabilities:


     x  Probabilities
0  0.5            0.1
1  3.0            0.1
2  4.5            0.1
3  4.6            0.1
4  4.9            0.1
5  5.2            0.1
6  5.3            0.1
7  5.5            0.1
8  7.0            0.1
9  9.5            0.1


Training Data Set:


     x
0  0.5
1  0.5
2  3.0
3  4.5
4  4.6
5  4.6
6  5.2
7  5.5
8  5.5
9  9.5
```

```
Decision Tree:
{3.01: {0: -1.0,
        1: {4.609999999999999: {0: 1.0,
                                1: {5.21: {0: -1.0,
                                           1: {5.51: {0: 1.0,
                                                      1: -1.0}}}}}}}}

Error:  0.2
Model Weight:  0.693147180559945

Boosting Round  2
Probabilities:


     x  Probabilities
0  0.5         0.0625
1  3.0         0.0625
2  4.5         0.0625
3  4.6         0.0625
4  4.9         0.2500
5  5.2         0.0625
6  5.3         0.2500
7  5.5         0.0625
8  7.0         0.0625
9  9.5         0.0625

Training Data Set:


     x
0  3.0
1  4.5
2  4.6
3  4.9
4  4.9
5  4.9
6  5.2
7  5.2
8  5.3
9  5.3



Decision Tree:
{4.91: {0: {3.01: {0: -1.0,
                   1: 1.0}},
        1: -1.0}}
```

```
Error:  0.06249999999999986
Model Weight:  1.3540251005511035

Boosting Round  3
Probabilities:


     x  Probabilities
0  0.5       0.033333
1  3.0       0.033333
2  4.5       0.033333
3  4.6       0.033333
4  4.9       0.133333
5  5.2       0.033333
6  5.3       0.133333
7  5.5       0.500000
8  7.0       0.033333
9  9.5       0.033333

Training Data Set:


     x
0  3.0
1  4.5
2  4.9
3  5.3
4  5.3
5  5.3
6  5.5
7  5.5
8  7.0
9  9.5



Decision Tree:
{3.01: {0: -1.0,
        1: {4.91: {0: 1.0,
                   1: {5.31: {0: -1.0,
                              1: {5.51: {0: 1.0,
                                         1: -1.0}}}}}}}}

Error:  0.0
Model Weight:  18.021826694558577

Boosting Round  4
Probabilities:
```

```
     x  Probabilities
0  0.5       0.016667
1  3.0       0.016667
2  4.5       0.016667
3  4.6       0.016667
4  4.9       0.066667
5  5.2       0.016667
6  5.3       0.066667
7  5.5       0.250000
8  7.0       0.016667
9  9.5       0.016667

Training Data Set:


     x
0  0.5
1  3.0
2  4.9
3  4.9
4  5.3
5  5.3
6  5.5
7  5.5
8  5.5
9  9.5



Decision Tree:
{3.01: {0: -1.0,
        1: {4.91: {0: 1.0,
                   1: {5.31: {0: -1.0,
                              1: {5.51: {0: 1.0,
                                         1: -1.0}}}}}}}}

Error:  0.0
Model Weight:  18.021826694558577

Boosting Round  5
Probabilities:


     x  Probabilities
0  0.5       0.008333
1  3.0       0.008333
2  4.5       0.008333
3  4.6       0.008333
4  4.9       0.033333
```

```
5  5.2       0.008333
6  5.3       0.033333
7  5.5       0.125000
8  7.0       0.008333
9  9.5       0.008333
```

Training Data Set:

```
     x
0  3.0
1  4.9
2  4.9
3  4.9
4  4.9
5  5.2
6  5.3
7  5.5
8  5.5
9  7.0
```

Decision Tree:

```
{4.91: {0: {3.01: {0: -1.0,
                   1: 1.0}},
        1: {5.31: {0: -1.0,
                   1: {5.51: {0: 1.0,
                              1: -1.0}}}}}}
```

Error: 0.0
Model Weight: 18.021826694558577

Boosting Round 6
Probabilities:

```
     x  Probabilities
0  0.5       0.004167
1  3.0       0.004167
2  4.5       0.004167
3  4.6       0.004167
4  4.9       0.016667
5  5.2       0.004167
6  5.3       0.016667
7  5.5       0.062500
8  7.0       0.004167
9  9.5       0.004167
```

```
Training Data Set:


     x
0  0.5
1  4.9
2  5.2
3  5.3
4  5.3
5  5.5
6  5.5
7  5.5
8  5.5
9  5.5



Decision Tree:
{5.31: {0: {0.51: {0: -1.0,
                   1: {4.91: {0: 1.0,
                              1: -1.0}}}},
        1: 1.0}}

Error:  0.012499999999999987
Model Weight:  2.1847239262335028

Boosting Round  7
Probabilities:


     x  Probabilities
0  0.5       0.002110
1  3.0       0.166667
2  4.5       0.002110
3  4.6       0.002110
4  4.9       0.008439
5  5.2       0.002110
6  5.3       0.008439
7  5.5       0.031646
8  7.0       0.166667
9  9.5       0.166667

Training Data Set:


     x
0  3.0
1  3.0
2  3.0
```

```
3  3.0
4  4.6
5  5.3
6  7.0
7  7.0
8  9.5
9  9.5


Decision Tree:
{3.01: {0: -1.0,
        1: {4.609999999999999: {0: 1.0,
                                1: -1.0}}}}

Error:  0.040084388185653845
Model Weight:  1.5879292198922543

Boosting Round  8
Probabilities:


     x  Probabilities
0  0.5       0.001099
1  3.0       0.086813
2  4.5       0.001099
3  4.6       0.001099
4  4.9       0.105263
5  5.2       0.001099
6  5.3       0.004396
7  5.5       0.394737
8  7.0       0.086813
9  9.5       0.086813

Training Data Set:


     x
0  3.0
1  3.0
2  3.0
3  3.0
4  5.5
5  5.5
6  5.5
7  5.5
8  7.0
9  9.5
```

```
Decision Tree:
{3.01: {0: -1.0,
        1: {5.51: {0: 1.0,
                   1: -1.0}}}}

Error:  0.005494505494505481
Model Weight:  2.599248515632894

Boosting Round  9
Probabilities:


     x  Probabilities
0  0.5       0.000552
1  3.0       0.043646
2  4.5       0.000552
3  4.6       0.000552
4  4.9       0.052922
5  5.2       0.100000
6  5.3       0.400000
7  5.5       0.198459
8  7.0       0.043646
9  9.5       0.043646

Training Data Set:


     x
0  3.0
1  3.0
2  5.2
3  5.3
4  5.3
5  5.3
6  5.3
7  5.5
8  5.5
9  5.5



Decision Tree:
{5.31: {0: -1.0,
        1: 1.0}}

Error:  0.14132015120674435
Model Weight:  0.9021841289217212
```

```
Boosting Round  10
Probabilities:


     x  Probabilities
0  0.5       0.000322
1  3.0       0.025415
2  4.5       0.001955
3  4.6       0.001955
4  4.9       0.187243
5  5.2       0.058229
6  5.3       0.232916
7  5.5       0.115560
8  7.0       0.154424
9  9.5       0.154424


Training Data Set:


     x
0  3.0
1  4.9
2  4.9
3  4.9
4  5.2
5  5.3
6  5.3
7  5.5
8  7.0
9  9.5



Decision Tree:
{3.01: {0: -1.0,
        1: {4.91: {0: 1.0,
                   1: {5.31: {0: -1.0,
                              1: {5.51: {0: 1.0,
                                         1: -1.0}}}}}}}}

Error:  0.0
Model Weight:  18.021826694558577
```

1.2.1 Running the Final Ensemble Model on the Training Set

[82]:
```
ensemble_results = pd.DataFrame(np.zeros(shape = len(data_train)), columns =␣
 ↪['0'])
for tree in boosting_ensemble.keys():
```

```
    ensemble_results[tree] = np.zeros(shape = len(data_train))
    for index in data_train.index:
        ensemble_results.loc[index,tree] = model_weights[tree - 1] * predict(
        data_train.loc[index,:],
        boosting_ensemble[tree])

del ensemble_results['0']
ensemble_results['Ensemble Results'] = np.zeros(shape = len(ensemble_results))
for index in ensemble_results.index:
    ensemble_results.loc[index,'Ensemble Results'] = ensemble_results.
 ↪loc[index,:].sum()
    if ensemble_results.loc[index,'Ensemble Results'] > 0:
        ensemble_results.loc[index,'Ensemble Results'] = 1
    else:
        ensemble_results.loc[index,'Ensemble Results'] = -1
ensemble_results['True Class'] = data_train['Class']
ensemble_error = np.sum(ensemble_results['Ensemble Results'] -␣
 ↪ensemble_results['True Class'])/(len(ensemble_results))
display(ensemble_results)
print("Ensemble Model Error on Original Training Set:", ensemble_error)
```

```
          1         2          3          4          5         6         7  \
0 -0.693147 -1.354025 -18.021827 -18.021827 -18.021827 -2.184724 -1.587929
1 -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
2  0.693147  1.354025  18.021827  18.021827  18.021827  2.184724  1.587929
3  0.693147  1.354025  18.021827  18.021827  18.021827  2.184724  1.587929
4 -0.693147  1.354025  18.021827  18.021827  18.021827  2.184724 -1.587929
5 -0.693147 -1.354025 -18.021827 -18.021827 -18.021827 -2.184724 -1.587929
6  0.693147 -1.354025 -18.021827 -18.021827 -18.021827 -2.184724 -1.587929
7  0.693147 -1.354025  18.021827  18.021827  18.021827  2.184724 -1.587929
8 -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
9 -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929

          8         9         10  Ensemble Results  True Class
0 -2.599249 -0.902184 -18.021827              -1.0        -1.0
1 -2.599249 -0.902184 -18.021827              -1.0        -1.0
2  2.599249 -0.902184  18.021827               1.0         1.0
3  2.599249 -0.902184  18.021827               1.0         1.0
4  2.599249 -0.902184  18.021827               1.0         1.0
5  2.599249 -0.902184 -18.021827              -1.0        -1.0
6  2.599249 -0.902184 -18.021827              -1.0        -1.0
7  2.599249  0.902184  18.021827               1.0         1.0
8 -2.599249  0.902184 -18.021827              -1.0        -1.0
9 -2.599249  0.902184 -18.021827              -1.0        -1.0


Ensemble Model Error on Original Training Set: 0.0
```

1.2.2 Results on the Test Data Set

[83]:
```
data_val = pd.DataFrame(
    np.arange(1,11,1),
    columns = ['x'])

for tree in boosting_ensemble.keys():
    data_val[tree] = np.zeros(shape = len(data_val))
    for index in data_val.index:
        data_val.loc[index,tree] = model_weights[tree - 1] * predict(
            data_val.loc[index,:],
            boosting_ensemble[tree])

data_val = data_val.set_index('x')
data_val['Class'] = np.zeros(shape = len(data_val))

for index in data_val.index:
    if data_val.loc[index,:].sum() > 0:
        data_val.loc[index,'Class'] = 1
    else:
        data_val.loc[index,'Class'] = -1
display(data_val)
```

```
           1         2          3          4          5         6         7  \
x
1  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
2  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
3  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
4   0.693147  1.354025  18.021827  18.021827  18.021827  2.184724  1.587929
5  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827 -2.184724 -1.587929
6  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
7  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
8  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
9  -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929
10 -0.693147 -1.354025 -18.021827 -18.021827 -18.021827  2.184724 -1.587929

           8         9         10  Class
x
1  -2.599249 -0.902184 -18.021827   -1.0
2  -2.599249 -0.902184 -18.021827   -1.0
3  -2.599249 -0.902184 -18.021827   -1.0
4   2.599249 -0.902184  18.021827    1.0
5   2.599249 -0.902184 -18.021827   -1.0
6  -2.599249  0.902184 -18.021827   -1.0
7  -2.599249  0.902184 -18.021827   -1.0
8  -2.599249  0.902184 -18.021827   -1.0
9  -2.599249  0.902184 -18.021827   -1.0
10 -2.599249  0.902184 -18.021827   -1.0
```

Overall, good results. In the original data set, all the classes of "1" are concentrated around 4. One could argue that 6 was misclassified, since its closest value in the training set, 5.5, had a class of 1. However,...

```
for index in df.index:              #Create a list of splits between the instances
split = df.loc[index,'x'] + 0.01              splits.append(split)
```

In the code above, I create a new split by adding $x = 0.01$ to whatever the current instance is. In order for 6 to be picked up as having a class of "1" I would have to set $x = 0.51$. I tried this at the end of the project and found the change caused my `build_decision_tree` function to enter an infinite loop. I was doing more harm then good at that point so I quit and kept the version of the code that worked. Rather than creating special conditions to capture an odd classification, I think it's better to treat $[5.5, 1]$ as an outlier and claim that 6 should have class "-1."

[ ]:

```

```