Assignment 4

Teammates: utpatel, naikpa

Random Forest:

- **How we implemented Random Forest Classifier:**

Below are the functions that I used to create Random Forest Classifier. The name of the functions itself will tell what they do.

1. def label_count(rows): This function counts the number of rows that are of certain category and returns dictionary ex: 0 : 3005 90: 4332 180: 4833 270 : 3555

2. def most_common(list): This function counts the most common element from a list and returns that element. This is used to count most common element when we are terminating at leaf node. Example: If at leaf node we want to terminate, we will have to assign one orientation, so we will assign 0 90 180 or 270 and return that as the most common element.

3. def entropy(rows): Given a set of say 4000 rows, it will calculate entropy or disorderness among this 4000 rows. For this, it will have to take into account all label's probability of coming.

4. def info_gain(left, right, current_uncertainty): This function will get left child rows and right child rows (obtained by say : column value <128, put that row in left child set of rows and say if >128 than put that row in right child set of rows) and current uncertainty that will be of parent of this nodes. Function calculates information gain by taking into account parents entropy and left and right childs entropy.

5. def partition(rows,colindex, valueofsplit): This function just does partitions based on the column index given and for that column, the split value given. SO we will compare that whole column and which ever rows are greater than value of split , they will enter in true_rows and rest rows in false_rows. Thus, it returns true_rows and false_rows.

6. def best_split(rows): It counts the best gain possible for set of rows by splitting and column values for which that gain could be achieved and split values for that column which is used.Thus it returns best_gain, best_col, and best_value_of_split

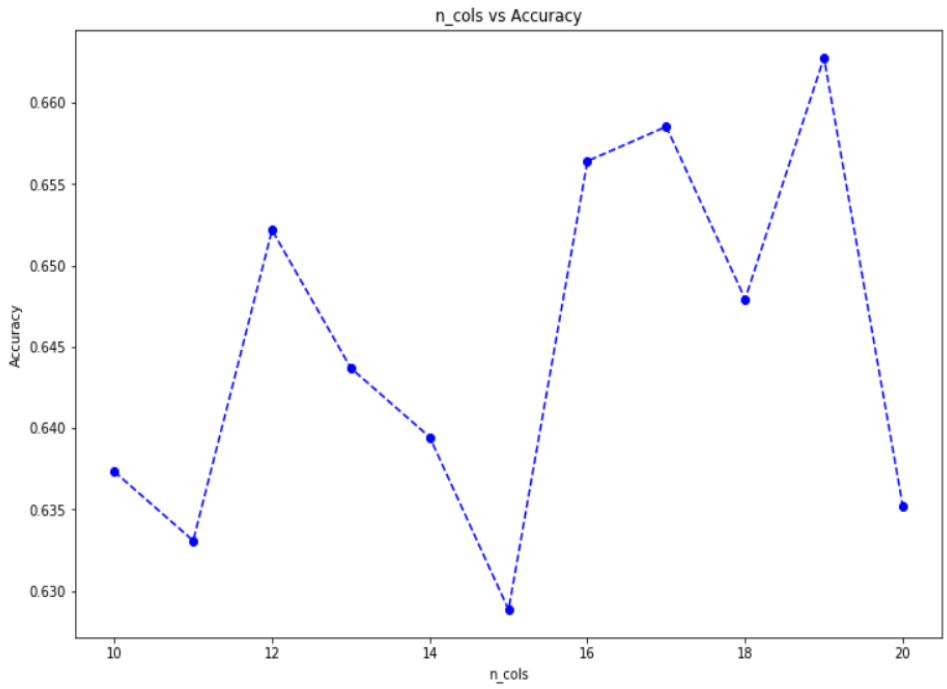
7. def decision_tree(rows,level,randomcols): This function creates decision trees by using all above functions and input given here is dataset that is sliced(3500) and level uptill which that decision tree can grow. This function outputs a dictionary and that dictionary is our single decision tree itself.

8. def random_forest(trainrgbvalues,n_trees,trainorientation,level,n_cols,n_rows): It creates number of decision trees as mentioned in n_trees as argument and uses values passed to it ie. n_rows - no of rows per decision tree , n_cols- no of columns per decision tree, level - level upto which one decision tree can be explored, trainrgbvalues- list that contains all rows which are again in list (all training data is passed here) and trainorientation - the 0 90 180 270 of each row is also passed here.

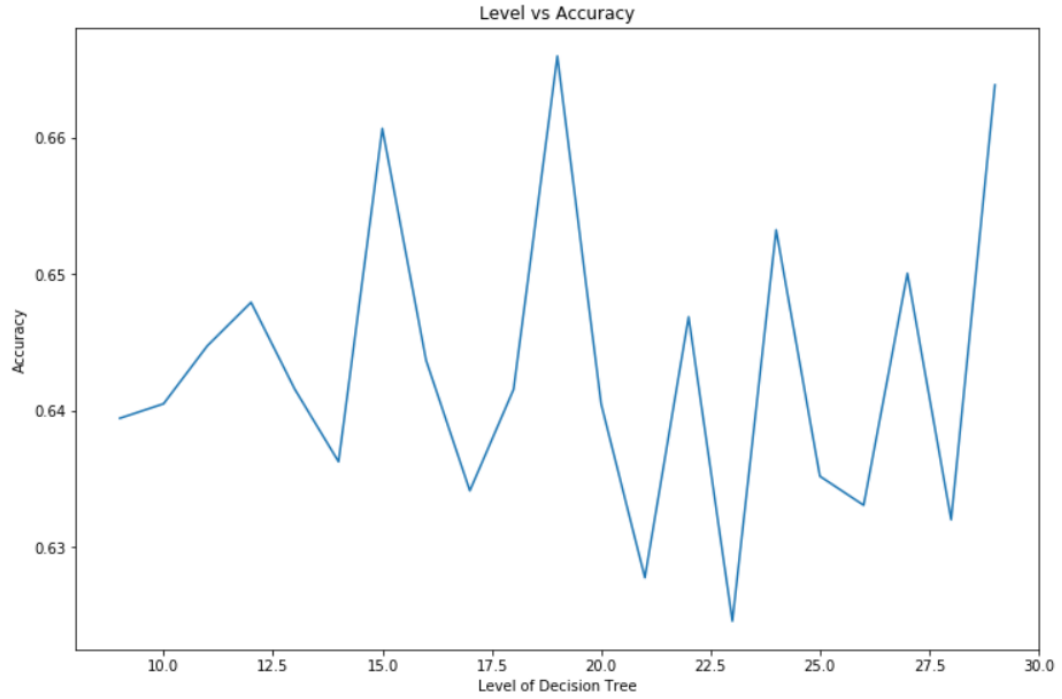
9. def func(tree, testlist): This function takes two arguments: tree- dictionary form of tree and testlist - row for testing. Function will use the tree to explore decision tree and give output orientation. This function will be called for each and every tree that are in a dictionary and this all for each row and at the end if we are having 20 trees in our final model, we will have 20 orientations for each row and that row will be assigned orientation that is repeated the most .

- **Graphs Showing plots of Different Parameters vs Accuracy :**

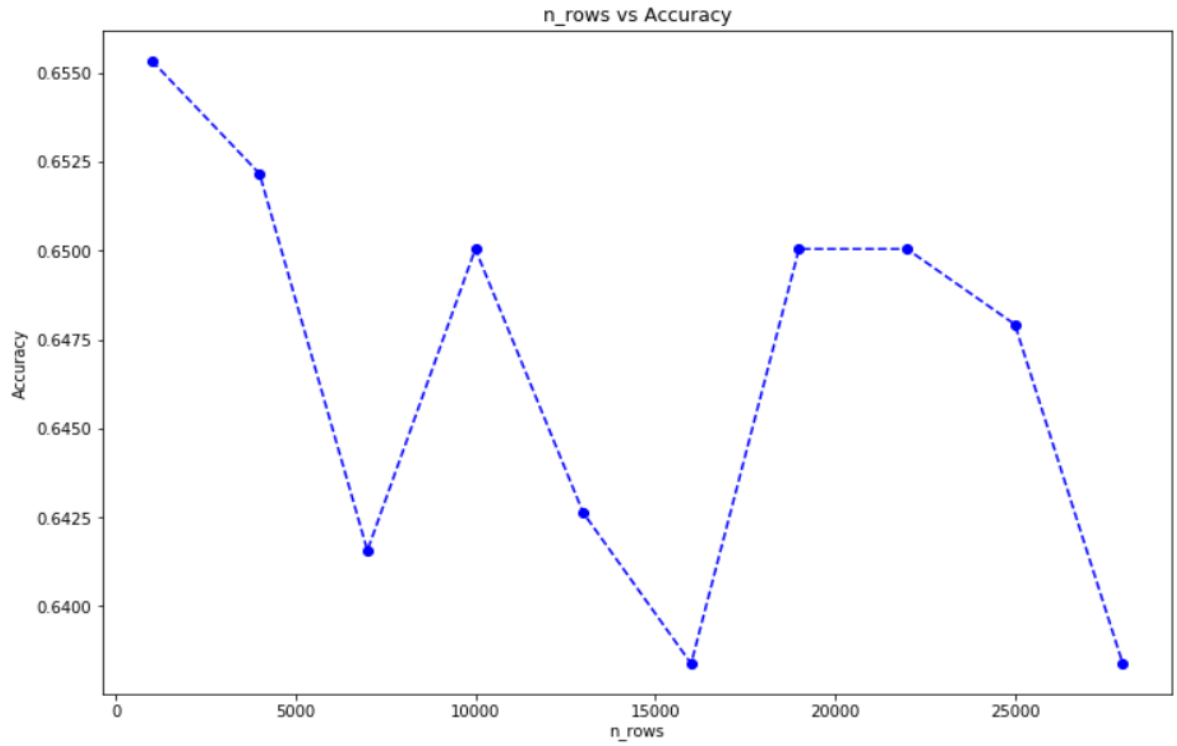
Number of columns vs Accuracy:



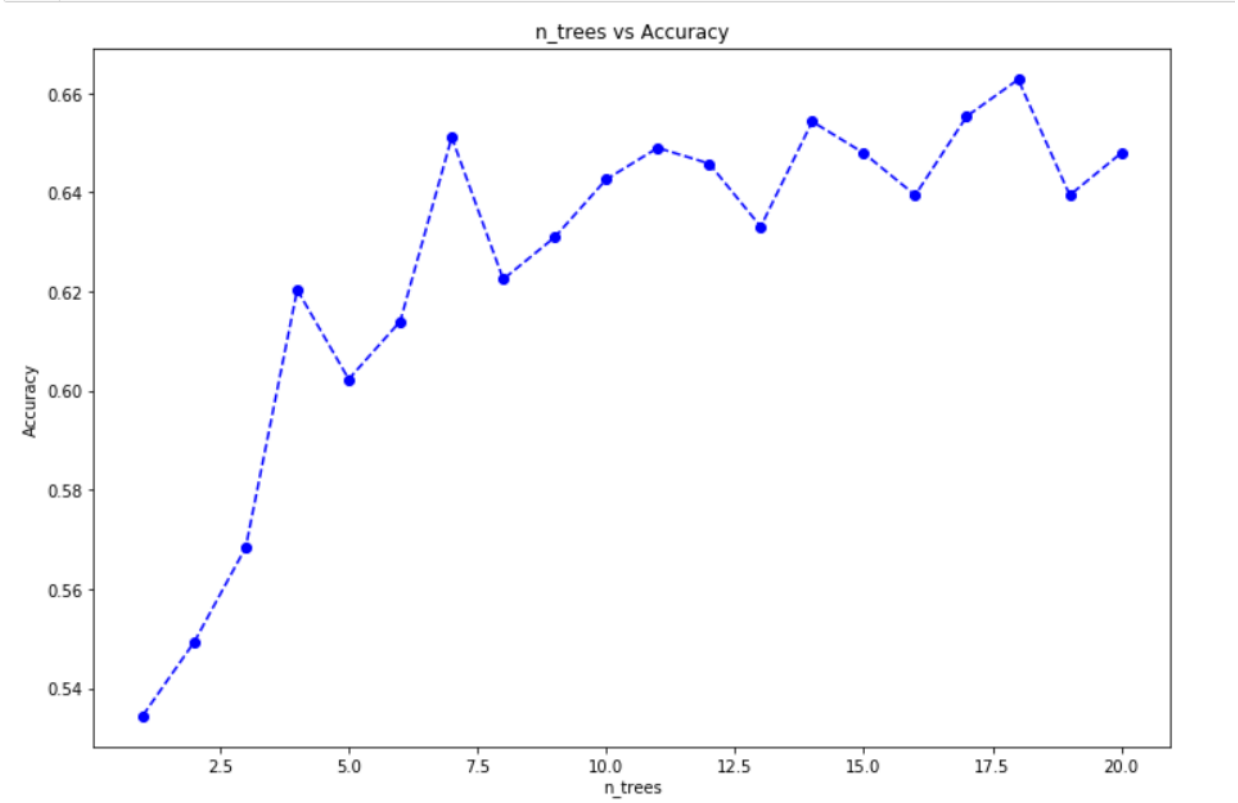
Level or Depth Vs Accuracy



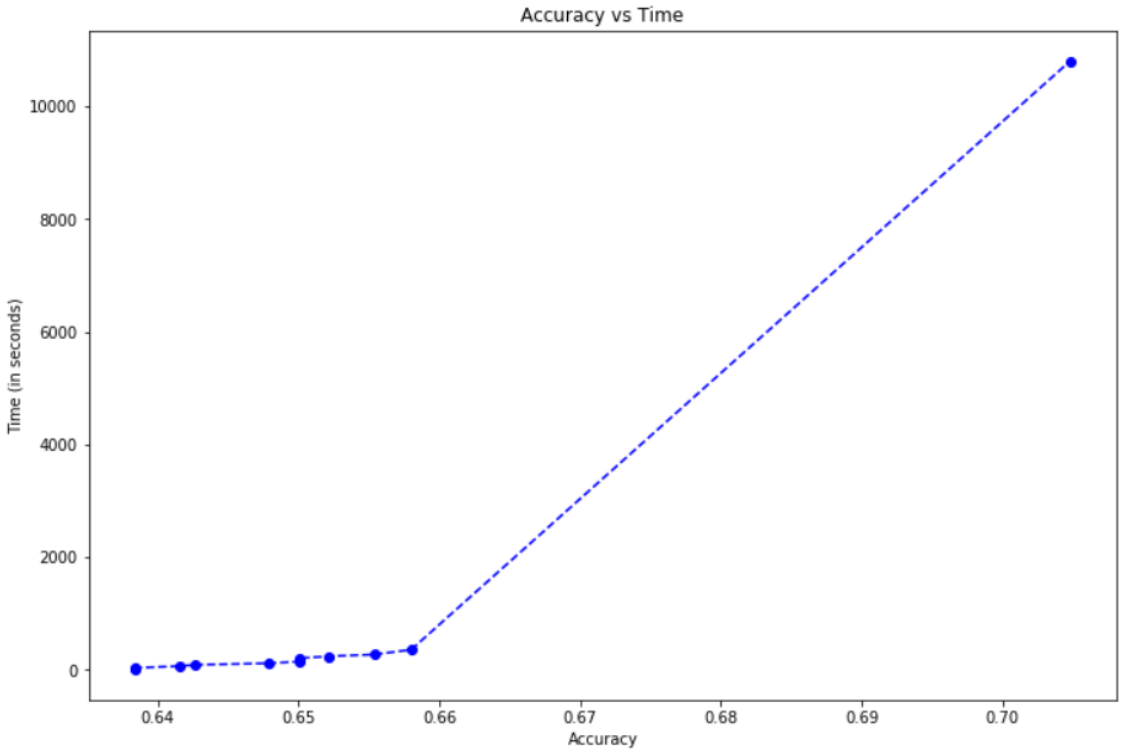
Number of Rows vs Accuracy:



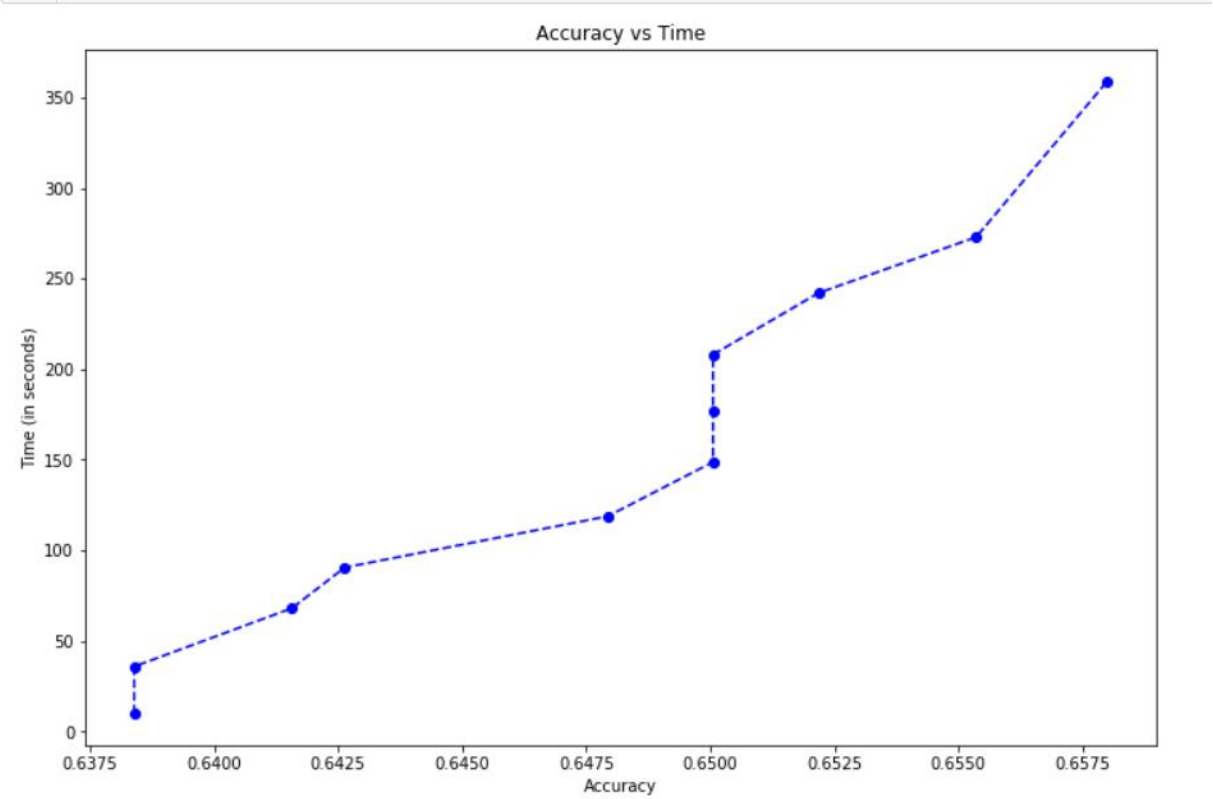
Number of Trees vs Accurcacy



Average Accuracy vs Average Time:



Average Accuracy vs Average Time (Removing last point):



- **Results:**

```
C:\Users\UTSAV\Desktop\AI Assignments\Assignment 4>python temp.py test test-data.txt forest_model.txt forest
Accuracy achieved is :  0.704135737009544
```

Our forest_model.txt file consists of model that consists of accuracy of 70.41% and that model was trained for nearly 3-5 hours. Thus, If you will run test file case against my model made, you would definitely get 70.4% accuracy. This approach was to use value of splits from 1 to 255 for each column and then do this for all columns and thus we would come up with best value of split for best columns and thus I felt, this is the best idea and made model based on this.

Personal Note:

Now, In order for ease of AI's to check my training program works well or not, I created my training program such that it checks only for one value of split for each column and that value was of 128. So, If AI's would train my model, they would achieve accuracy of nearly 63.5 to 67.5%. I asked Question on Piazza and got reply from Prof. Crandall that AI's would check only test model

which was 70.4% accuracy in mine, but they might also train some day. So I created this balance between both. Now If AI's wasn't to train my code for 70.4% accuracy, they will just have to comment one line : -- > for c in [128]:

and uncomment line ---> # for c in range(1,255):
which are on lines 130 and 131.

- **Other Approaches/ Findings :**

Over all in the model We were behind tuning 5 parameters:

1. level or depth of Decision Trees
2. n_trees : number of trees to explore for Random Forest
3. n_cols : number of columns to take in each Decision tree.
4. n_rows: number of rows to take in each Decision Tree.
5. value_of_split: (MOST IMPORTANT) The value at which we need to split for each column for checking.

Approach 1 : We tried to implement by splitting on 128 for each column and after lot of parameter tuning, we achieved accuracy of nearly 68%-70%. I ran 10,000 loops and got 5 parameters that got this much high accuracy, but while checking for same parameters in same program with no parameter tuning loops, we dropped accuracy from 69-71% to directly 65-67%.

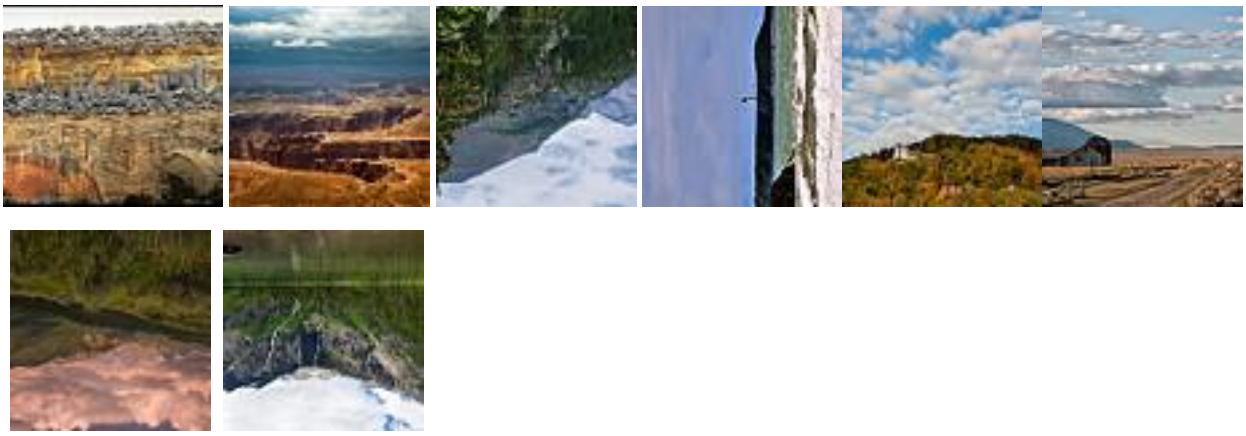
Approach 2: We tried to split values based on Median. We created medians for each column and whenever we need to split on particular column, we would split on that median for that particular column. This approach was also nearly 65% to 68%.

Approach 3: We tried to split on [25,35,45,55,65,75,85,95,105,115, 125,135,145,155,165,175,185,195,205,215,225] values for each column.So whenever we will go for finding best split for a particular set of rows, we will loop through all columns and for all columns, we will split on all above mentioned values, and surprisingly we achieved accuracy of 66%-69%.

Approach 4: Now, we thought that it would be great if we could find best point splitting such that that point is best value of split among all possible splits. So, idea was to use split values of 1 to 255 for each column and to do so for all 192 columns , and then to come up with that golden column and that diamond splitting value for that column. That value was the most accurate split among all possible ways. BUT here I had to do training time of 3-5 hours and then I got my final answer of forest_model.txt.

- **Correctly Classified samples and incorrectly classified samples:**

Sample of Correctly Classified Samples:



Sample of Incorrectly Classified Samples:

Patterns in Error:

```
Missclassified {0: 55, 90: 68, 180: 78, 270: 78}
classified is  {0: 184, 90: 156, 180: 158, 270: 166}
```

One most important pattern that we recognized was that: All images in which we have blue sky, they are almost identified correctly as the blue pixel value at that point would be clearly separating most of the images on their orientation level. Also another pattern we found was that if the sky is somewhat black or other color than blue or Most importantly, the images which are mostly vague or which are distorted from normal order due to the effect of noise(here : light). This was because, if a image is taken in day and it clearly separates the pixels of sky and there are more clear colors in those images, rest images are those which have not clear colors due to less light in sky or are somewhat vague, those images are classified incorrectly.

KNN:

Implementation:

1. Created a class for the KNN model
2. The class is initialized with 2 things a k value and the distance function to be used.
3. We store this k value which is chosen to be 11 as it gives optimal accuracy which can be seen from the graph below of k values vs accuracy and the distance metric used in model.txt file during the training phase.
4. KNN is a lazy algorithm so this is the most we can save as a KNN model.
5. During the testing phase for each testing sample the distances from the testing sample to all other training samples are found out and using those the nearest k neighbors are found out and the mode of the neighbors is given as the prediction for the testing sample.

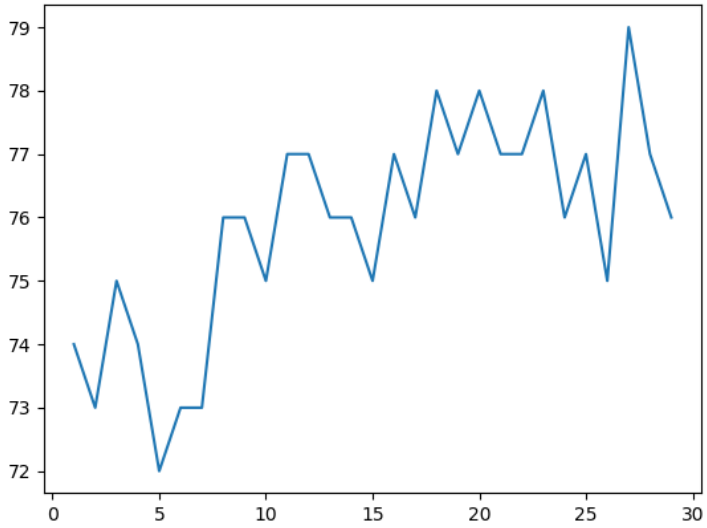
Functions used:

def __init__ - Function to initialize the KNN model.

def distance_l2 - Function calculates the Euclidean distance between given 2 samples using the 192 features.

def predict - Function that generates the predictions for each testing sample by taking the mode of the k closest neighbors.

K vs accuracy:



Time taken on SILO is less than 10 minutes:

```
naikpa@silo:~/B551/Assignment4
[naikpa@silo Assignment4]$ ./ml.py
Train data:
                  label    y    1    2    3 ...   188  189  190  191  192
0  train/10008667845.jpg    0  124   78   50 ...    95   47  111   66   29
1  train/10008667845.jpg   90  119   66    8 ...   134   79  120   73   36
2  train/10008667845.jpg  180  111   66   29 ...    84   53  124   78   50
3  train/10008667845.jpg  270  120   73   36 ...   108   23  119   66    8
4  train/10017728034.jpg    0  206  203  197 ...    23   21   28   27   25

[5 rows x 194 columns]
Test data:
                 label    y    1    2    3 ...   188  189  190  191  192
0  test/10008707066.jpg    0  219  220  221 ...    26   16   47   40   24
1  test/10099910984.jpg    0  135  173  191 ...    86   29  140   78   21
2  test/10107730656.jpg  180   56   72   33 ...   212  227  193  204  224
3  test/10161556064.jpg  270  137  145  183 ...   157  151  224  210  208
4  test/10164298814.jpg    0  158  179  197 ...    89   58  102   74   49

[5 rows x 194 columns]
Score for KNN =  71.04984093319194
Time taken =  564.0246078968048
[naikpa@silo Assignment4]$
```

Correctly classified images :

Incorrectly classified images:

Patterns observed :

- Almost all of the landscape images are classified successfully, this must be because of the clear distinction of light blue pixels for the sky and the darker pixels for the landscape.
- Images where such a contrast is not observed are miss classified by our model.

ADABOOST model :

Implementation:

In the AdaBoost model we use the weak classifier as a decision stump.

Train and test data pre-processing:

- Of the total of 192 features we generate 300 new features.
- We generate the combinations of all features in pairs of two.
- Therefore we have 192x192 number of combinations of pairs.
- Of these we randomly choose 300 pairs
- And we now generate features by subtracting the first feature number from the second.
- We now add a weights column required for boosting whose initial value

Model implementation:

1. As mentioned above we use decision stumps as weak classifier.
2. We have 2 classes:
   a) **DecisionTree**
   b) **AdaBoost**
3. The decision tree class is initialized by giving it the label we are interested in and the feature column to use for the stump.
4. The decision tree then scans for values from -200 to 200 with a step size of 20 as split values, generates temporary splits, calculates the entropies, and the information gain of each split value for the given feature.
5. We use shannon'n formula for entropy calculation.

$$Entropy = \sum -p\log(p)$$

6. We then choose the split value having the max information gain and return this split to the adaboost model.
7. The adaboost model then does the split, calculates the error and then updates the weights of the correctly classified samples using the formula below:

$$weight = weight * error/(1-error)$$

8. We then normalize the weights.

9. The model then calculates the score of the weak classifier as

$$score = \log(1-error/error)$$

10. This is done till there are 100 weak classifiers accumulated.

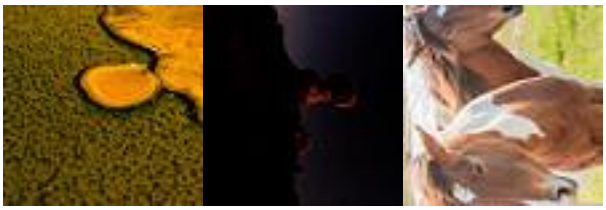
One vs All approach:

We have used a one vs all approach here, so we have 4 models one for 0, one for 90, one for 180 and one for 270. We get the predictions from the 4 models of the positive(0,90,180,270) classes which are in the form of values. We predict a testing sample to be of the class for which the prediction has the highest place.

```
Making predictions
          0         90        180        270
0    2.800751  2.751762  2.333361  1.126182
1    6.245613  2.889086  2.758685  1.855625
2   -1.000000 -1.000000  0.017227 -1.000000
3   -1.000000  0.831025 -1.000000  0.021449
4    2.556211  4.018191  2.005289  1.879926
5    2.427847  1.902117  2.880324  2.224877
6    5.585107  1.863646  3.279206  1.286259
7   -1.000000  3.152681  2.005167  0.291461
8   -1.000000  3.867759  1.461443 -1.000000
9   -1.000000 -1.000000  1.322792 -1.000000
10  -1.000000 -1.000000  0.362806  0.303886
11   2.845394  0.149119  0.831999  1.983430
12   2.108360  0.819607  1.097332  1.102814
13   7.093456  3.240814  3.110034  2.079338
14   0.158022  2.303464  1.436284  2.313393
15   2.926439  0.861068  1.906195  0.270104
16  -1.000000 -1.000000  0.186460 -1.000000
17  -1.000000 -1.000000 -1.000000 -1.000000
18   2.336317  1.503596  2.333361  1.728493
19   0.097727  0.179005  0.175887  0.768324
20   0.093661 -1.000000 -1.000000 -1.000000
21  -1.000000 -1.000000 -1.000000 -1.000000
22  -1.000000  0.387683  1.355862  0.943672
23   1.776837 -1.000000  1.484048 -1.000000
24  -1.000000 -1.000000  0.920781 -1.000000
25   0.930936  3.874707  2.446445  1.271940
26   0.465282  4.404327  1.491923  0.318175
27   1.743197  1.997119  1.922429  1.407744
28  -1.000000  0.411383  0.008347  1.142540
29  -1.000000  0.172762 -1.000000 -1.000000
```

Correctly classified images:

Incorrectly classified images:



Patterns observed :

- Almost all of the landscape images are classified successfully, this must be because of the clear distinction of light blue pixels for the sky and the darker pixels for the landscape.
- Images where such a contrast is not observed are miss classified by our model.