

CS57800: Statistical Machine Learning

HOMEWORK 1

Due: Sep 19, 2018 on Wednesday, Using 2 late days

1 Problem Statement

Predict the quality of white wine on scale of 0 to 10 using KNN classifier and Decision Tree Classifier.

2 Shuffle and 4-fold Cross Validation

The first thing which I did was to shuffle the dataset *winequality-white.csv* and stored it into a pickle file named *shuffled_data_wine*. This was done primarily to have a better distribution for both training and testing time. In other words, taking the first n examples from an ordered data set has less probability to generalize on the remaining examples in scenarios where rest of data set might have a different distribution.

4-fold Cross Validation I split the dataset into 4 folds. So, everytime I kept 1 fold aside as the test-fold and combined the remaining 3 folds as the training dataset. Now, out of this training dataset, I used 1/5th data as validation dataset and the remaining 4/5th dataset as training data set. The portion of the code which does this is in the file preprocessing.py and is given below:

```

knn.py  x  distances.py  x  decisiontree.py  x  preprocessing.py
5 def shuffle_data():
6     with open('winequality-white.csv', 'rb') as csvfile:
7         csvreader = csv.reader(csvfile, delimiter = ';')
8         # print type(csvreader)
9         data = []
10        for i, row in enumerate(csvreader):
11            if i != 0:
12                row = [float(i) for i in row]
13                data.append(row)
14        shuffle(data)
15        with open('shuffled_data_wine_knn', 'wb') as fp:
16            pickle.dump(data, fp)
17
18    """
19    The function divide folds is for dividing the data into k folds which is later
20    used for cross validation.
21    """
22    def divide_folds(k, data):
23        if k < 0:
24            return [data]
25
26        fold_size = len(data)/k
27        folds = [data[x:x+fold_size] for x in xrange(0, len(data), fold_size)]
28        if len(folds) == 5:
29            folds[3] += folds[4]
30            del folds[4]
31        return folds
32
33    def split_folds_train_test(test_index, folds):
34        test_fold = folds[test_index]
35        train_fold = []
36        for x in range(len(folds)):
37            if x != test_index:
38                train_fold += folds[x]
39
40        return train_fold, test_fold
41
42    def split_train_validation(train_data):
43        validation_len = len(train_data)/5
44        validation = train_data[:validation_len]
45        train = train_data[validation_len:]
46        # print validation[validation_len-1]
47        # print train[0]
48        return train, validation

```

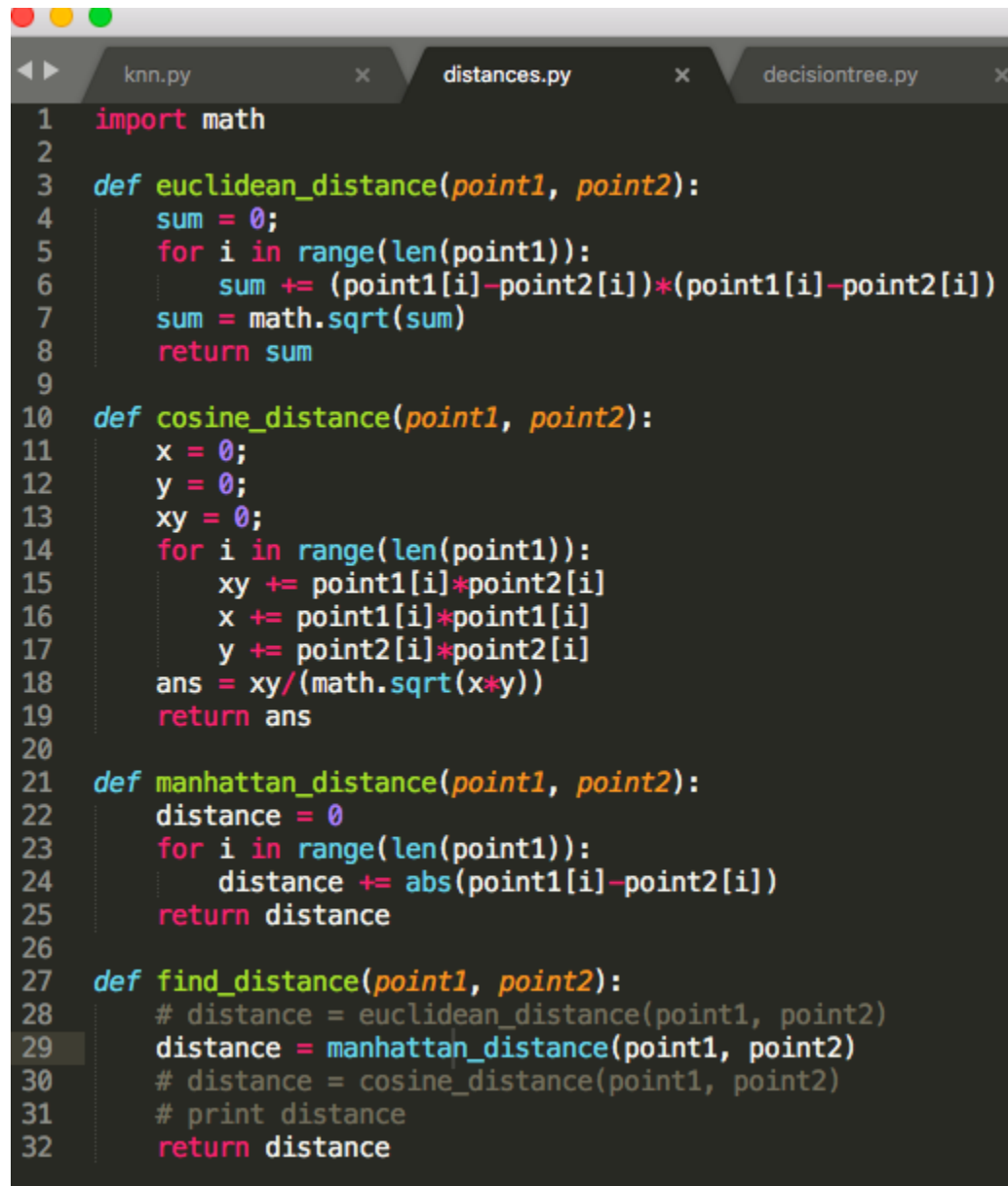
These functions are called in knn.py and decision_tree.py and the code which calls them is given below:

```

knn.py x distances.py x decisiontree.py x preprocessing.py x Hyper-parameters:
79 with open ('shuffled_data_wine_knn', 'rb') as fp:
80     data = pickle.load(fp)
81     # print len(data)
82     no_folds = 4
83
84     """
85     The portion of code which divides into 4 folds for cross validation.
86     The function divide_folds is in preprocessing.py file.
87     """
88     folds = divide_folds(no_folds, data)
89
90     start_k = 2
91     end_k = 14
92     k_axis = [i for i in range(start_k,end_k+1)]
93     folds_axis_f1 = [[-100 for i in range(start_k,end_k+1)] for i in range(1)]
94     folds_axis_accuracy = [[-100 for i in range(start_k,end_k+1)] for i in range(1)]
95
96     """
97     this for loop was used to get scores for a range of k values
98     and then select the best ones on the validation data set.
99     """
100    for k in range(start_k,end_k+1):
101        print "Hyper-parameters:"
102        print "K: ", k
103        print "Distance measure: ", "Euclidean Distance"
104        print ""
105        average_validation_f1 = 0
106        average_validation_accuracy = 0
107        average_test_f1 = 0
108        average_test_accuracy = 0
109        for i in range(no_folds):
110            print "Fold-", i+1, ":"
111            """
112            The function split_folds_train_test takes out the ith fold each time as the test_data which
113            is not touched until after the model is tuned.
114            """
115            train_data, test_data = split_folds_train_test(i, folds)
116
117            """
118            The train_data is split into training and validation using the function spit_train_validati
119            The implementation of this function is in preprocessing.py
120            """
121            train, validation = split_train_validation(train_data)

```

Thus, I use the function `divide_folds` to divide the data into 4 folds. Next, I split the 4 folds 4 times in the for loop using the function `split_folds_train_test` function. Next I split the `train_data` into training and validation datasets. For KNN, the hyperparameters are `k` which is varied as per the variables `start_k` and `end_k` and the distance measure which can be euclidean distance, manhattan distance and cosine distance. This is specified in the `distances.py` file as given below:



```
1 import math
2
3 def euclidean_distance(point1, point2):
4     sum = 0;
5     for i in range(len(point1)):
6         sum += (point1[i]-point2[i])*(point1[i]-point2[i])
7     sum = math.sqrt(sum)
8     return sum
9
10 def cosine_distance(point1, point2):
11     x = 0;
12     y = 0;
13     xy = 0;
14     for i in range(len(point1)):
15         xy += point1[i]*point2[i]
16         x += point1[i]*point1[i]
17         y += point2[i]*point2[i]
18     ans = xy/(math.sqrt(x*y))
19     return ans
20
21 def manhattan_distance(point1, point2):
22     distance = 0
23     for i in range(len(point1)):
24         distance += abs(point1[i]-point2[i])
25     return distance
26
27 def find_distance(point1, point2):
28     # distance = euclidean_distance(point1, point2)
29     distance = manhattan_distance(point1, point2)
30     # distance = cosine_distance(point1, point2)
31     # print distance
32     return distance
```

For decision tree, the hyperparameter is depth of the tree and this I vary from 6 to 23.

3 DecisionTree

I implemented a decision tree using ID3 algorithm. It is a recursive implementation. The main steps are:

- First, I split the data set into 4 folds. I then take out the test fold. The remaining data is split into 4/5th as the training set and 1/5th as the validation set.
- Now, I find the normalization parameters from the training + validation datasets i.e. the minimum value for each feature, the max value for each feature.

- Now, I normalize the dataset using min-max normalization from the normalization parameters calculated in the earlier steps. This gives the data points on scale of 0 to 1.
- For decision tree, I also scale the data points on a scale of 1000. During test point if any of my data point has value < 0 , then the value is marked as 0 and if the value is > 1000 , then its marked as 1000.
- Now, I create the decision tree recursively. In the recursive function which is `create_decision_tree(features, labels, depth)` in my `decisiontree.py` file, I first check what is the predicted label based on a majority vote from the labels present in the current node. Now, if this majority vote comes with a confidence of 100 i.e. all nodes belong to the same class then I mark this as a leaf node in my decision tree with this label as the label of the leaf node.
- If not, I first calculate the entropy for the existing labels in the node.
- Now, I calculate the information gain for a candidate split. For each feature attribute, as mentioned earlier, I have already scaled them on a scale of 0 to 1000. In my experiments, I start from the candidate split of (say) feature 1 being 7 and then increment it by 7 everytime till its value becomes greater than 1000. So, for each feature, I make 142 splits and this is done for 11 features so in total I have 142×11 candidate splits. I calculate the information gain for each of these splits and pick the one which has the max value. The information gain is given by:

$$I = E(S) - (E(l) * EL + E(R) * ER) / (EL + ER)$$

Here, I is Information Gain

$E(S)$ is entropy of labels present on the node

$E(l)$ is entropy of the labels in the left split

EL is total labels in the left split

$E(R)$ is entropy of the labels in the right split

ER is total labels in the right split

- Once, I decide the split, I split the data set and call recursively for the left node and the right node.
- Thus, I recursively build the tree.

After tuning the depth, I found the optimum depth to be 18 as per the f1 score and accuracy on the validation data set. One thing to note is that I only do binary splits at each node. I think that helps avoid overfitting and helps the tree generalize on unknown data.

4 KNN

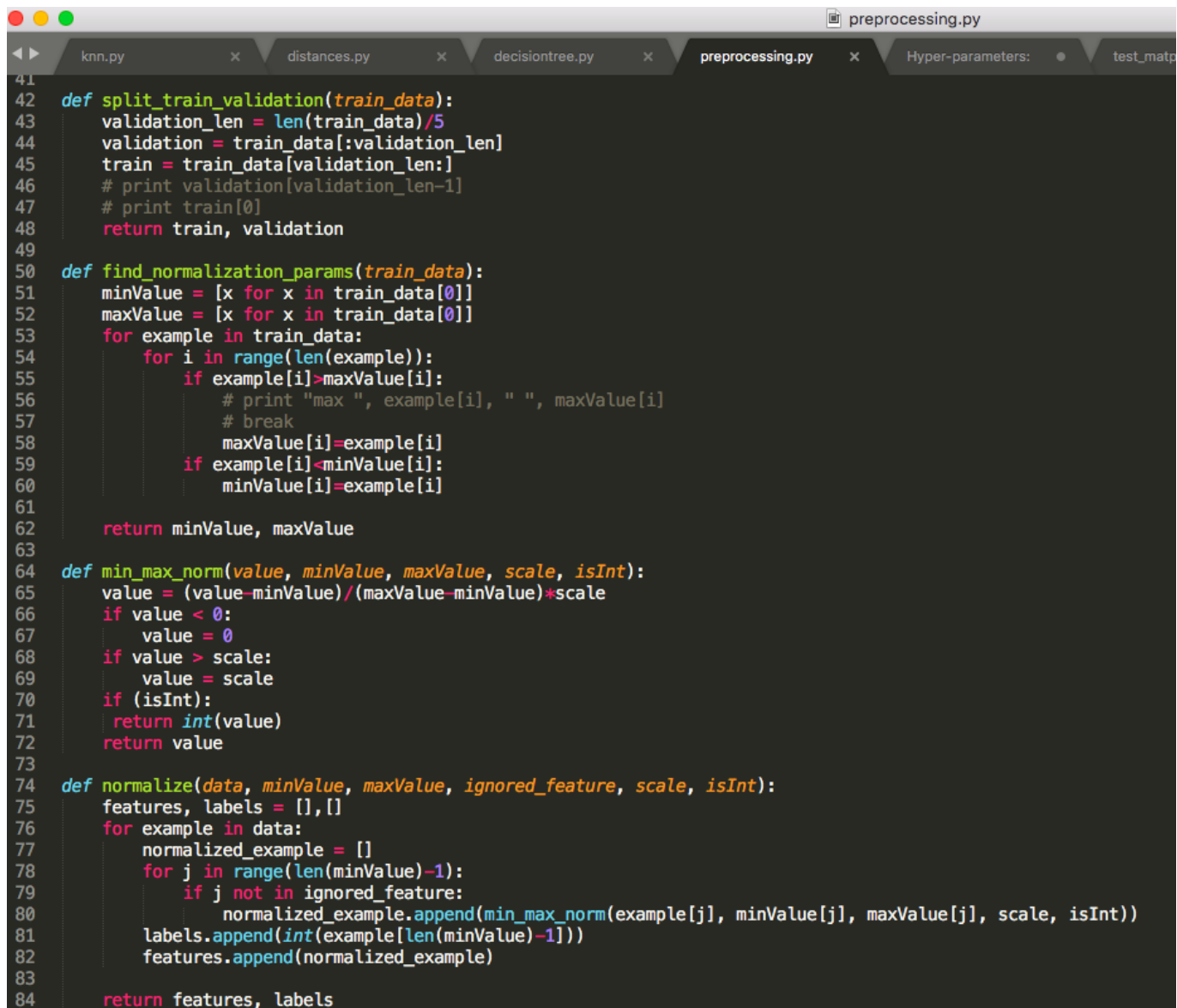
The description of the implementation of KNN is given below:

- First, I split the data set into 4 folds. I then take out the test fold. The remaining data is split into 4/5th as the training set and 1/5th as the validation set.
- Now, I find the normalization parameters from the training + validation datasets i.e. the minimum value for each feature, the max value for each feature.
- Now, I normalize the dataset using min-max normalization from the normalization parameters calculated in the earlier steps. This gives the data points on scale of 0 to 1.
- In knn, the label for a test point on validation data set is given by the weighted majority vote from the k nearest points from training data as per the distance measure defined. I tried both unweighted and weighted majority voting and the performance of weighted majority voting outperformed unweighted majority voting.
- The function which does this in my file knn.py is `knn_output (test_point, train_points, train_labels, k, is_weighting)`

The distance measures that I used were euclidean distance, manhattan distance and cosine distance. After tuning the value of k and distance measures on the validation data set, I found the optimum k to be 12 and the optimum distance measure to be manhattan distance.

5 Scope of Creativity

Now, for decision tree I had the issue of continuous valued features. To overcome this, I used a normalization function which is the min-max normalization function and scaled the values on scale of 0-1000. The portion of code which does this is given below:



```

41
42 def split_train_validation(train_data):
43     validation_len = len(train_data)/5
44     validation = train_data[:validation_len]
45     train = train_data[validation_len:]
46     # print validation[validation_len-1]
47     # print train[0]
48     return train, validation
49
50 def find_normalization_params(train_data):
51     minValue = [x for x in train_data[0]]
52     maxValue = [x for x in train_data[0]]
53     for example in train_data:
54         for i in range(len(example)):
55             if example[i]>maxValue[i]:
56                 # print "max ", example[i], " ", maxValue[i]
57                 # break
58                 maxValue[i]=example[i]
59             if example[i]<minValue[i]:
60                 minValue[i]=example[i]
61
62     return minValue, maxValue
63
64 def min_max_norm(value, minValue, maxValue, scale, isInt):
65     value = (value-minValue)/(maxValue-minValue)*scale
66     if value < 0:
67         value = 0
68     if value > scale:
69         value = scale
70     if (isInt):
71         return int(value)
72     return value
73
74 def normalize(data, minValue, maxValue, ignored_feature, scale, isInt):
75     features, labels = [],[]
76     for example in data:
77         normalized_example = []
78         for j in range(len(minValue)-1):
79             if j not in ignored_feature:
80                 normalized_example.append(min_max_norm(example[j], minValue[j], maxValue[j], scale, isInt))
81             labels.append(int(example[len(minValue)-1]))
82         features.append(normalized_example)
83
84     return features, labels

```

Now, here scale variable was a hyperparameter which I tuned and found the scale of 1000 to be optimum on the validation data set. Now, during testing time if I found the value to be lower than 0, I put this as 0 and if its greater than 1000, then I put it as 1000. One other thing which I tried was using the exact float values or floor int values. I found that floor int values gave similar performance so I used the floor int values i.e. if the feature value is 100.25 then it will become 100.

6 Report

6.1

The results for the KNN model are given below: Please note that these are for $k = 12$ and manhattan distance is used in this model.

k=12, manhattan distance	Validation f1 Score	Test f1 Score	Validation Accuracy Score	Test Accuracy Score
Fold 1	52.93217091	49.9110807	61.98910082	63.4803922
Fold 2	58.12766917	51.3729487	64.85013624	62.0915033
Fold 3	50.59951889	49.8136515	63.6239782	62.6633987
Fold 4	56.52981358	55.948686	64.16893733	65.0897227
Average of folds	54.54729313	51.7615917	63.65803815	63.3312542

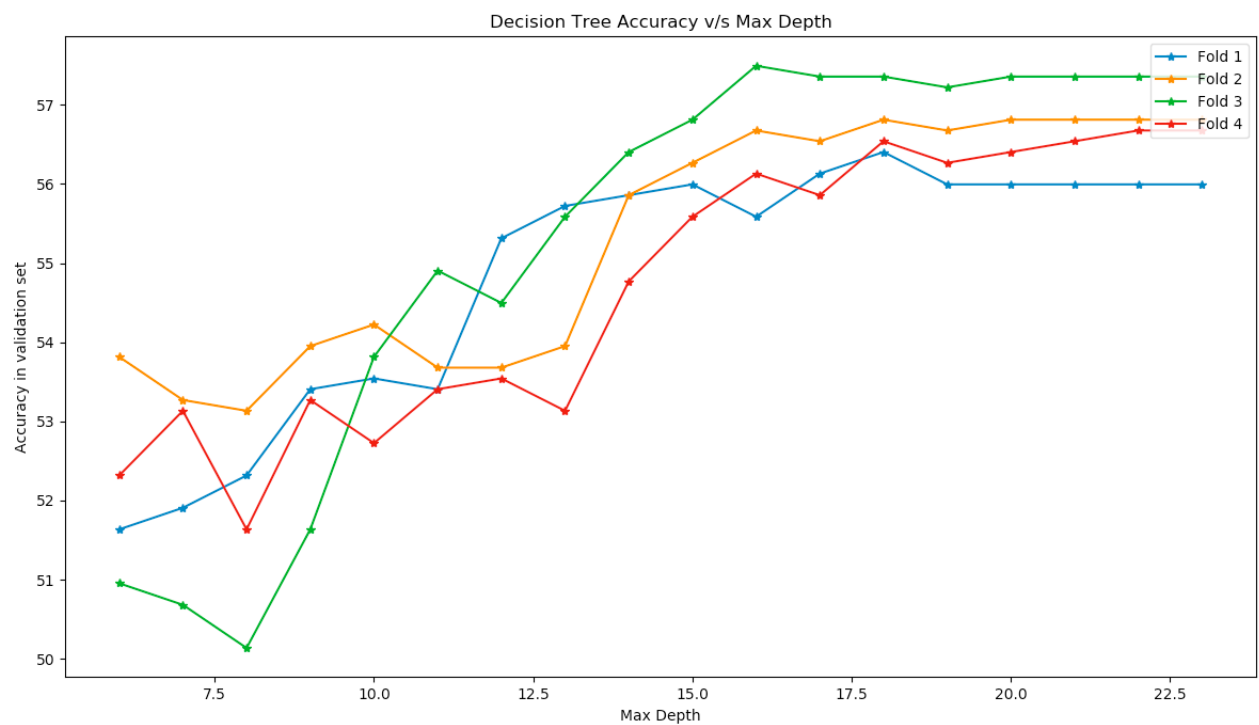
6.2

The results for the decision tree model are given below: Please note that these are for depth = 18.

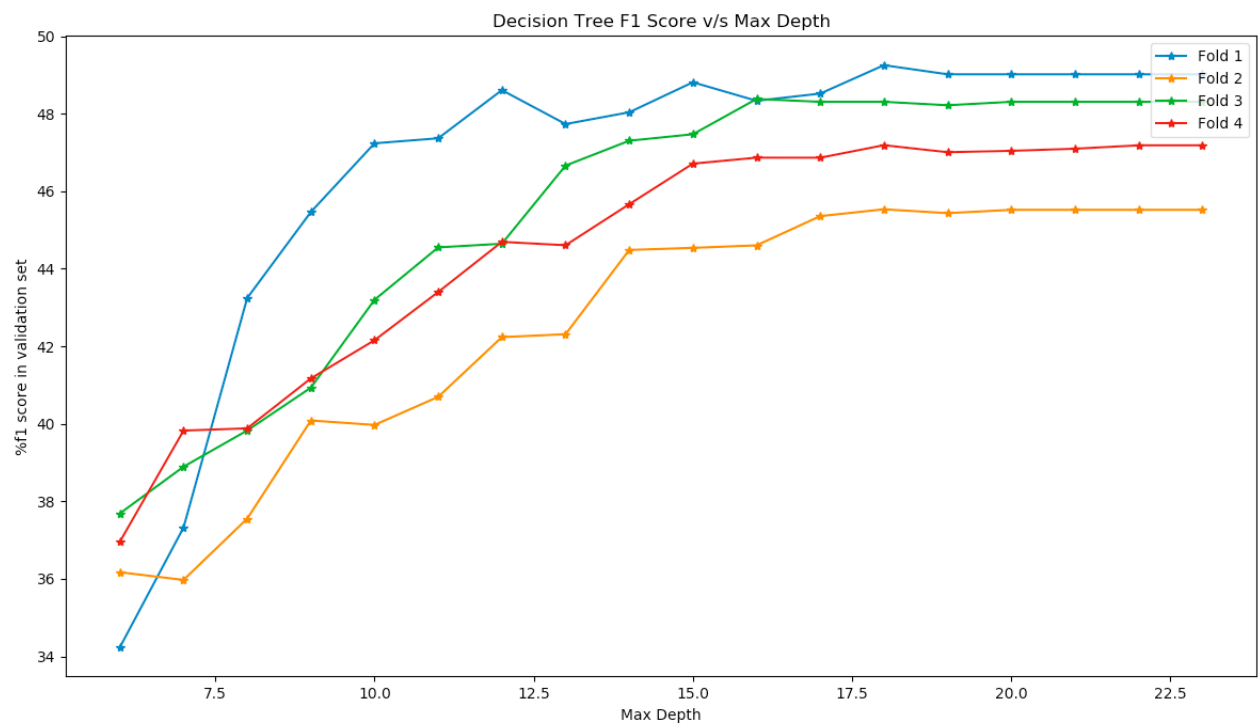
Decision Tree Max Depth 18	Training f1 Score	Validation f1 Score	Test f1 Score	Training Accuracy Score	Validation Accuracy Score	Test Accuracy Score
Fold 1	99.6263626	49.25196855	51.93933642	99.45578231	56.40326975	60.70261438
Fold 2	99.6955827	45.53459776	45.78026439	99.59183673	56.8119891	56.53594771
Fold 3	99.7680217	48.30712871	45.95528587	99.76190476	57.35694823	56.69934641
Fold 4	99.1320651	47.18823984	45.63895878	98.29816202	56.53950954	55.79119086
Average of folds	99.555508	47.57048371	47.32846136	99.27692146	56.77792916	57.43227484

6.3

Graph for Validation accuracy against max-depth for Decision tree is given below:

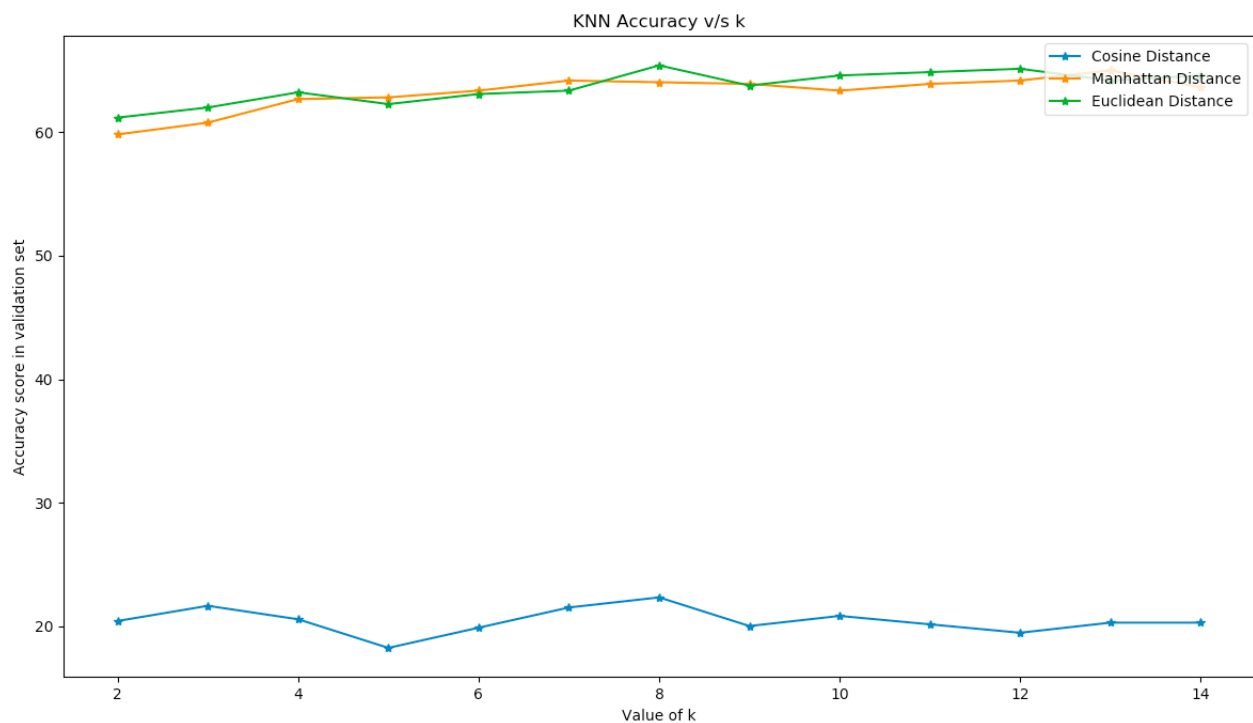


Graph for Validation f1-score against max-depth for Decision tree is given below:

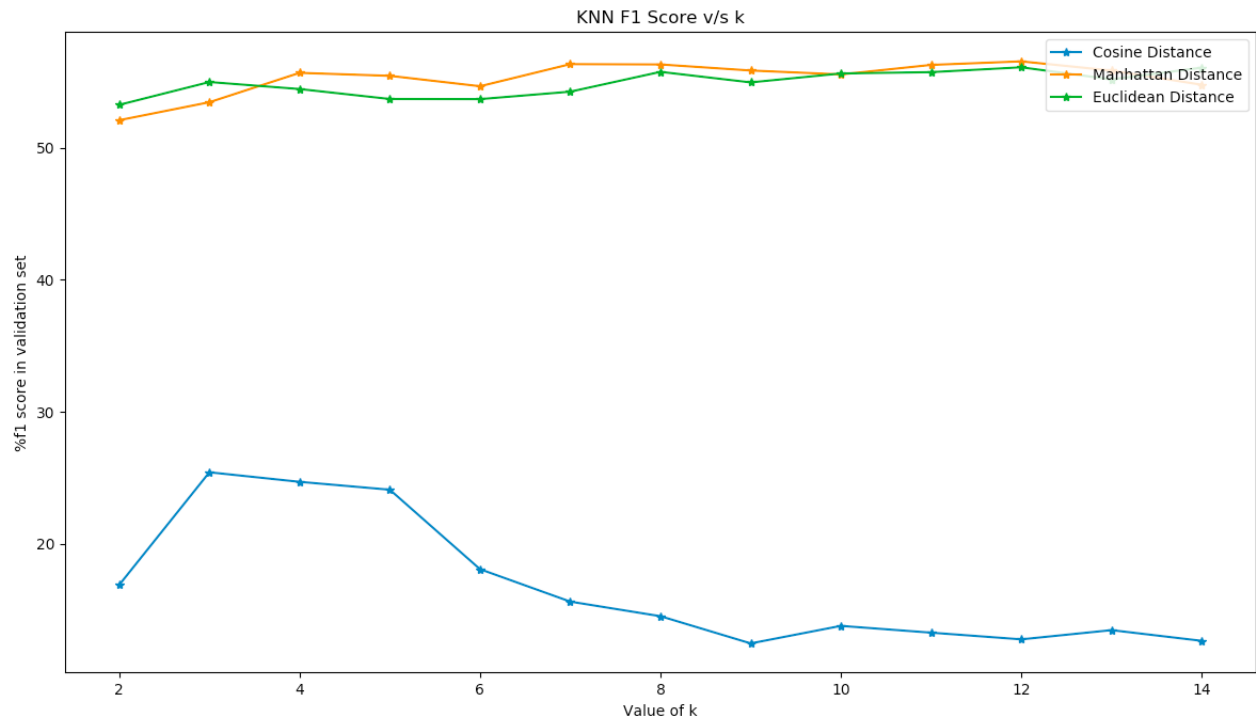


6.4

Graph for Validation accuracy against distance-measures for KNN is given below:



Graph for Validation f1-score against distance-measures for KNN is given below:



6.5

Answer to questions are given below:

- If we allow the max-depth upto the number of features in a decision tree and the splits are not constraint to be binary splits and each feature is split only once, then its likely to lead to overfitting for the decision tree and decision tree may just exactly learn the training data.
- KNN uses training data as it is to guess label for test points. It doesnt compress the training data and as such doesnt have learning step. Decision tree on the other hand compresses the training data using pruning and generalizes it which it then uses during testing time.
- We can convert the decision tree from classification model to ranking model using early pruning and not building a pure tree. Instead of just taking majority vote from labels at leaf nodes in a pure tree, we will have impure leaf nodes, which will have ranks for labels based on percentage of examples found in training step.

7 Additional Works

- The post pruning on decision tree was done by using a confidence score and the number of examples which I currently have for that node. The confidence score is defined as the percentage of labels which gave the majority label. Now, if all labels are the same then the confidence will be 100. This was earlier the break condition for the recursion i.e. this node

was marked as the leaf node.

I experimented with this by using the number of labels which I have for that node and the confidence score was varied from 75-95. Ultimately, the scores which I reported use the check on number of labels to be less than or equal to 10 and if the confidence is ≥ 90 , then I stop and mark this as the leaf node. This I did to avoid overfitting the tree on training data. The code which does this is given below:

```

92
93 ▼ def create_decision_tree(features, labels, depth):
94
95     predicted_label, confidence = predict_label(labels)
96
97 ▼     if depth > MAX_DEPTH or confidence == 100:
98         # print predicted_label, " with confidence ", confidence, " depth is ", depth
99         return Node(None, None, predicted_label)
100
101     #implement logic to prune decision tree if no of samples are less than a threshold
102     if len(labels) <= 10 and confidence >= 90:
103         return Node(None, None, predicted_label)
104
105     entropy_sample = calculate_entropy(labels)
106     max_information_gain = -100
107     max_split_attr = -100
108     max_split_value = -100

```

- The best results for dtree are given below:

Decision Tree Max Depth 18	Validation f1 Score	Test f1 Score	Validation Accuracy Score	Test Accuracy Score
Average of folds	47.570484	47.328461	56.777929	57.432275

The best results for knn are given below:

k=12, manhattan distance	Validation f1 Score	Test f1 Score	Validation Accuracy Score	Test Accuracy Score
Average of folds	54.547293	51.761592	63.658038	63.331254

- The knn model was improved by using the weighted voted distance instead of simple majority irrespective of the distance from the test point. One thing which I did to see if the model can be improved is to correctly predict top 2 labels rather than just the best label. This I did as a way to see if my model can be improved or the data is not spread to form pure clusters. This can be confirmed by not punishing the model if one of the top 2 labels is the actual

label. When I did this practice, I got an accuracy of around 87%. This I think confirmed my intuition that the data forms a lot of overlap clusters between labels like 6 and 7. This I think is a basic limitation of the knn algorithm and thus an accuracy of around 63% seems to be an upper limit to me.

In addition I did feature engineering which helped me get the best results when i ignored the 3rd feature i.e. citric acid content.

...