

Machine Learning, Spring 2020

Project Five – KNN + Decision Tree

Python tutorial: <http://learnpython.org/>

TensorFlow tutorial: <https://www.tensorflow.org/tutorials/>

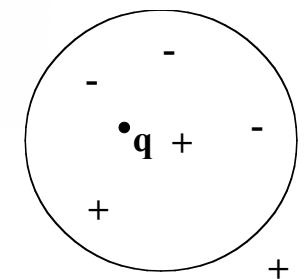
PyTorch tutorial: <https://pytorch.org/tutorials/>

Task1: K-Nearest Neighbor Algorithm

- For each training instance $t=(x, f(x))$
 - Add t to the set Tr_instances
- Given a query instance q to be classified
 - Let x_1, \dots, x_k be the k training instances in Tr_instances nearest to q
 - Return

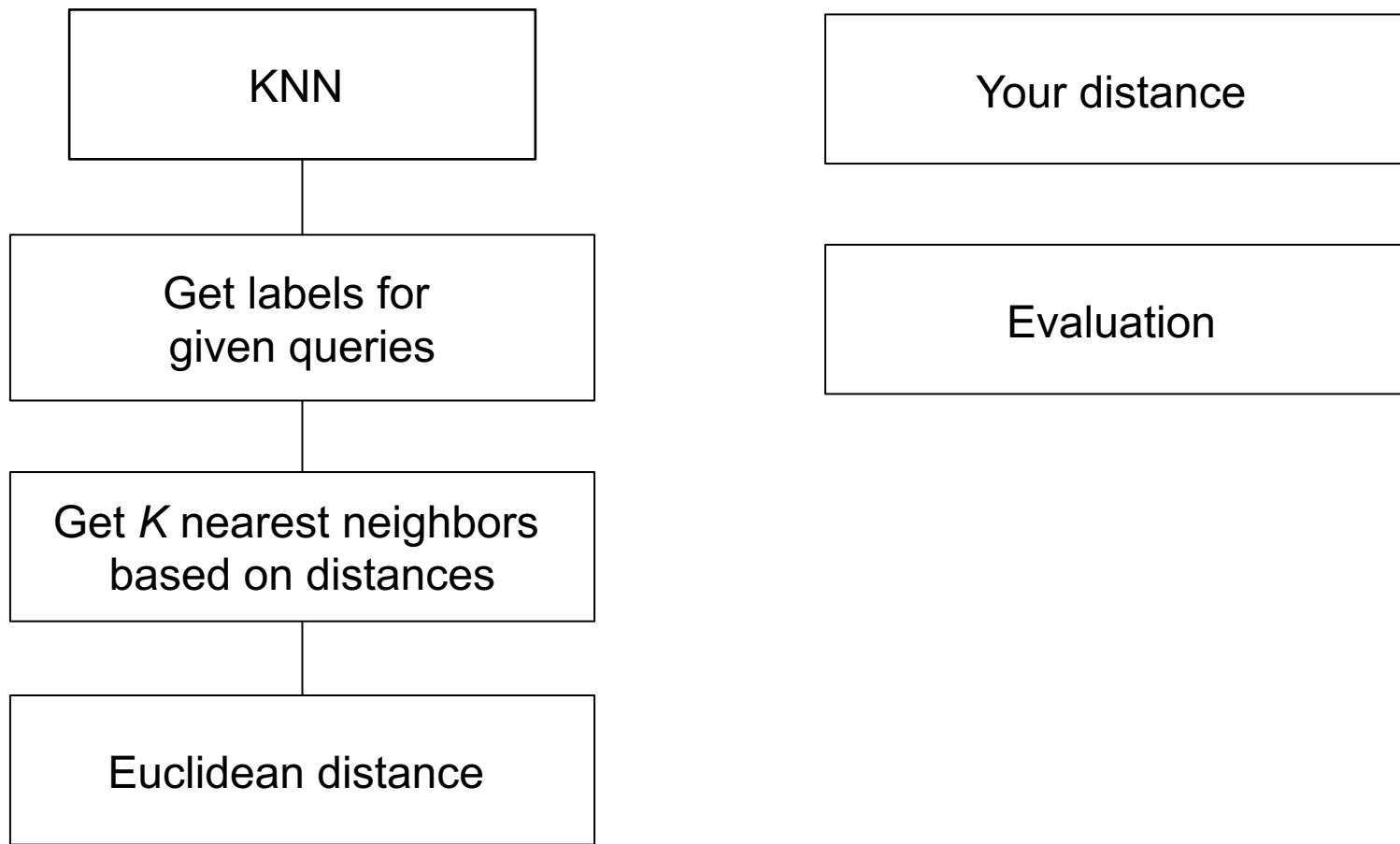
$$\hat{f}(q) = \arg \max_{v \in V} \sum_{i=1}^k \delta(v, f(x_i))$$

- Where V is the finite set of target class values, and $\delta(a,b)=1$ if $a=b$, and 0 otherwise (Kronecker function)
- Intuitively, the k -NN algorithm assigns to each new query instance the majority class among its k nearest neighbors



q is + under 1-NN, but - under 5-NN

Main Modules for KNN



Main Modules for KNN

Euclidean distance

- For each query, calculate the distance between the query and every training sample
- Euclidean distance:

$$D = \sqrt{\sum(x_1 - x_2)^2}$$

```
In [1]: def euclidean_distance(x1, x2):
    #suppose that x1:(1,2) x2:(N*3)

    return
```

Main Modules for KNN

Get K nearest neighbors based on distances

- For each query, calculate the distance between the query and every training sample
- Get the K nearest neighbors based on the calculated distances

```
[2]: def get_neighbors( x, y, x_test, k, distance= euclidean_distance):  
    # X in N*2, y in N*1, x_test in 2,  
    # calculate all the distances  
    # find the nearest K nieghbors' indexes  
    # return k nerest negibors  
    return
```

Main Modules for KNN

Get labels for given queries

- For each query, calculate the distance between the query and every training sample
- Get the K nearest neighbors based on the calculated distances
- Calculate the majority classes from K-NN, and assign the class to the given query

```
In [3]: def get_label(neighbors):  
    # calculate and return label with the majority votes  
    return
```

Main Modules for KNN

KNN

- For each query, calculate the distance between the query and every training sample
- Get the K nearest neighbors based on the calculated distances
- Calculate the majority classes from K-NN, and assign the class to the given query

```
In [4]: def knn(training_data,gt,testing_data,k,distance= euclidean_distance):
    # for each testing data, using functions above to predict the class labes using knn algo

    return
```

Main Modules for KNN

Evaluation

- Given the predicted class labels using KNN and ground truth testing labels:
 - Find out the samples that are not classified correctly
 - Calculate the number of misclassified samples (N_{mc}), then calculate the accuracy :

$$A = \frac{(N - N_{mc})}{N},$$

N : total number of testing samples

```
In [ ]: def eval(y,pred):
    #find out the samples that are not classified correctly
    #calculate the accuracy

    return
```

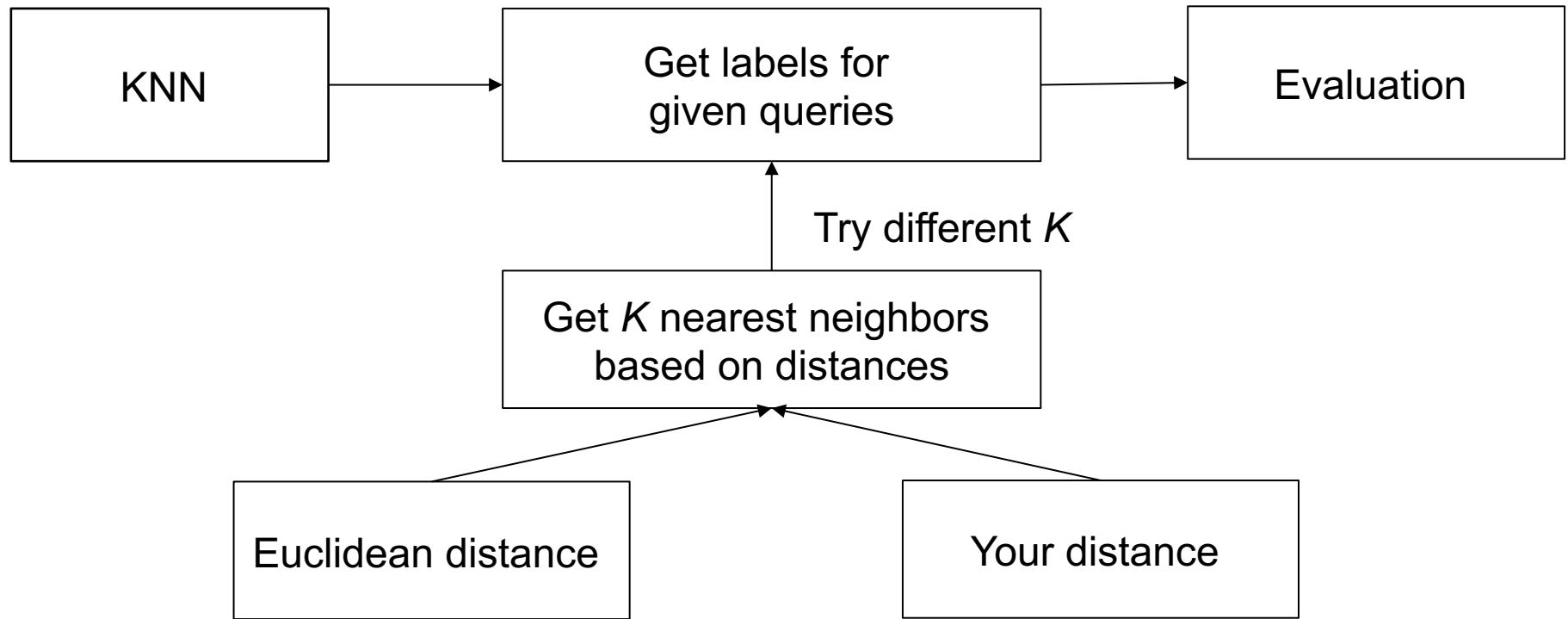
Main Modules for KNN

Your distance

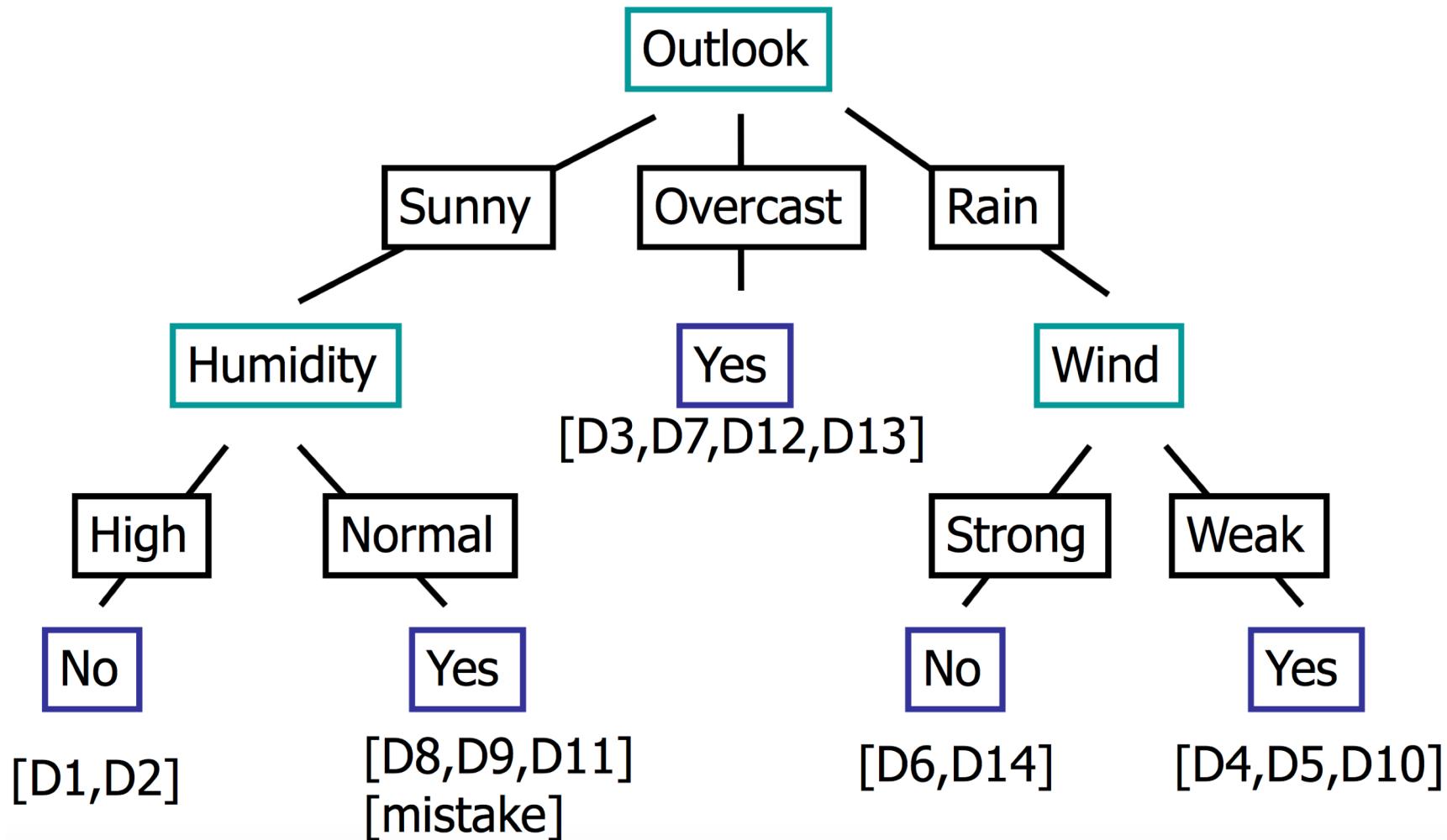
- Choose a new distance function for your k-NN algorithm. You can choose whatever distance function you like but should choose something that you think might yield higher accuracy than the Euclidean distance function.

```
In [ ]: def yourdistance(x1, x2):  
    return
```

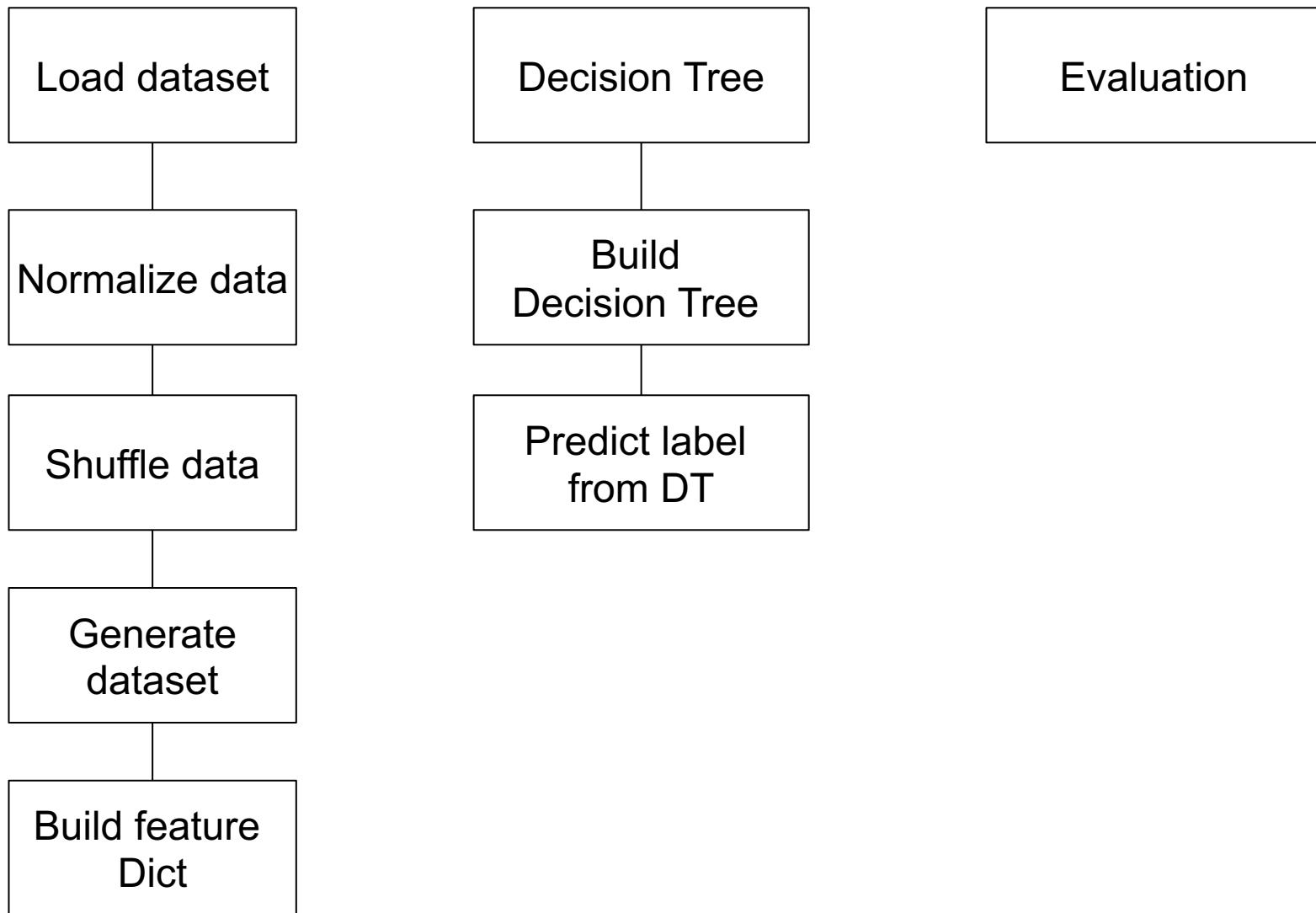
Pipeline for KNN



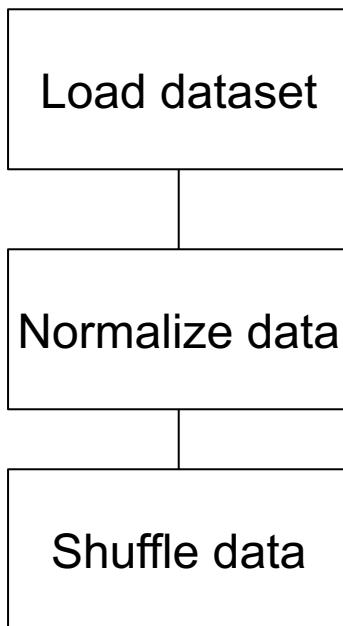
Task 2: Decision Tree



Main Modules Decision Tree



Main Modules Decision Tree



- Load the dataset from a .csv file
- Normalize dataset by $X' = \frac{(X - X_{\min})}{(X_{\max} - X_{\min})}$
- Randomly shuffle the data

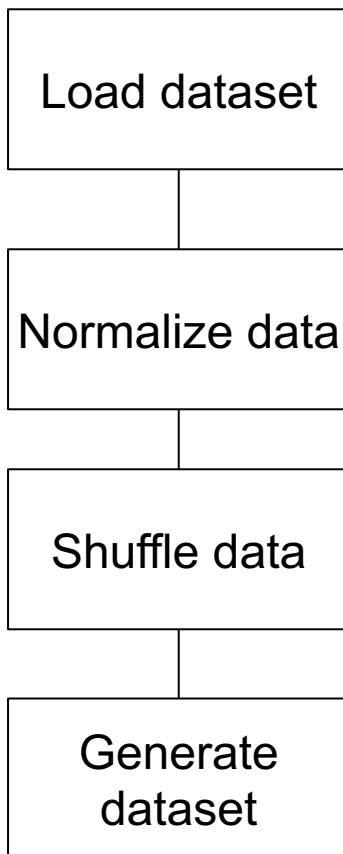
```
In [21]: #normalize the entire dataset prior to learning using min-max normalization
def normalize(matrix):
    # transfer the data matrix to np array in float type.
    print("normalizing the entire dataset:")
    print(a)
    print("Before normalizing")
    # apply the normalization along the 0 axis of a using the formula: (x - x_min)/(x_max - x_min)
    return
```

```
In [22]: # reading from the file using numpy genfromtxt with delimiter ','
def load_csv(file):

    return (X)
```

```
In [23]: #method to randomly shuffle the array using the numpy.random.shuffle()
def random_numpy_array(ar):
    return arr
```

Main Modules Decision Tree



- Load the dataset from a .csv file
- Normalize dataset by $X' = \frac{(X - X_{\min})}{(X_{\max} - X_{\min})}$
- Randomly shuffle the data
- Create training and testing dataset for 10-fold cross validation

```
In [24]: #Normalize the data and generate the training labels,training features, test labels and test training
def generate_set(X):
    print(X.shape[0])
    # store the label X[:, -1] to Y

    # reshape y to (Y's length, 1)
    # store it to j

    print("J is",j)
    # create the new_X which exclude the label X[:, -1]

    # normalize the data step
    # using our implemented function normalize()

    # add the label back to the normalized X
    # using np.concatenate along axis=1

    # store the size of rows of the normalized X with labels

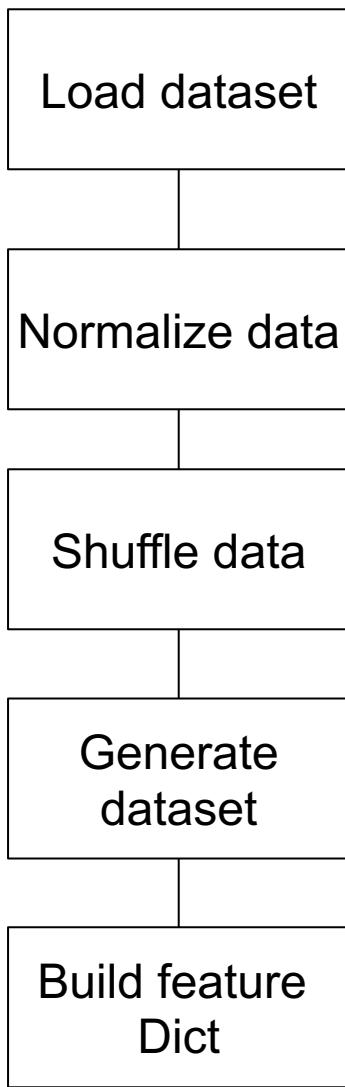
    # use the 10% of the data to be the test set.
    # store the number of testing data

    # set the starting index to be 0

    # set the ending index to be the number of testing data

    # create a list that store all features of the testing data
    # create a list that store all labels of the testing data
    # create a list that store all features of the training data
    # create a list that store all labels of the training data
```

Main Modules Decision Tree



- Load the dataset from a .csv file
- Normalize dataset by $X' = \frac{(X - X_{\min})}{(X_{\max} - X_{\min})}$
- Randomly shuffle the data
- Create training and testing dataset for 10-fold cross validation
- Build a dictionary where the key is the class label and values are the features which belong to that class.

```
In [16]: #build a dictionary where the key is the class label and values are the features which belong to that class.
def build_dict_of_attributes_with_class_values(X,y):
    #   init a dict for attributes

    #   init feature list.

    #   for each feature in the dataset
    for i in range(X.shape[1]):
        #       store the featur index

        #       find all the value correspond to this feature

        #       init an attribute list

        #       init the counter to 0

        #           for each value in the "all the value correspond to this feature"

        #               init a empty list that store the attribute value

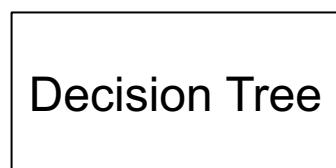
        #               append the this value to the list

        #               append the label of this value to the list

        #               append this list to the attribute list.

        #           increase the counter
```

Main Modules Decision Tree



- Define a node in the tree

```
In [11]: #Class node and explanation is self explanation
class Node(object):
    #      init the node with val,lchild,rchild,theta and leaf.
    def __init__(self, val, lchild, rchild, theta, leaf):
        self.root_value = val
        self.root_left = lchild
        self.root_right = rchild
        self.theta = theta
        self.leaf = leaf

    #      method to identify if the node is leaf
    def is_leaf(self):

        return

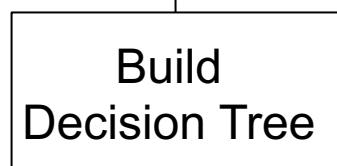
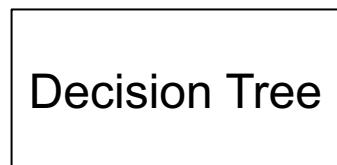
    #      method to return threshold value
    def ret_theta(self):

        return

    #      method return root value
    def ret_root_value(self):

        return
```

Main Modules Decision Tree



In [25]:

```
# Iterative Dichotomiser 3 entropy calculation
def entropy(y):
    #     init a class frequency dict

    #     init the attribute entropy to 0

    #     for each label in y:
    for i in y:
        #             this is label is already in the dict, we increase its freq

        #             else, we set the freq to 1

    #             calculate the cumulative entropy using the formula.

    return
```

Main Modules Decision Tree

Decision Tree

Build
Decision Tree

- Define a node in the tree
- Define the entropy function
- Calculate the information gain to find the best threshold for each feature

```
#method to calculate best threshold value for each feature
def cal_best_theta_value(self,ke,attri_list):
    #      init a list for data

    #      init a list for class labes

    #      for each attribute in the attri_list
    for i in attri_list:
        #          append the data

        #          append the feature value.

    #      calculate the entropy of those feaure values

    #      init the max info gain = 0

    #      init theta=0
```

Main Modules Decision Tree

Decision Tree

Build
Decision Tree

- Define a node in the tree
- Define the entropy function
- Calculate the information gain to find the best threshold for each feature → then select the best feature based on the information gain

```
#method to select the best feature out of all the features.
def best_feature(self,dict_rep):
    #      set key value to none

    #      set best info gain to -1

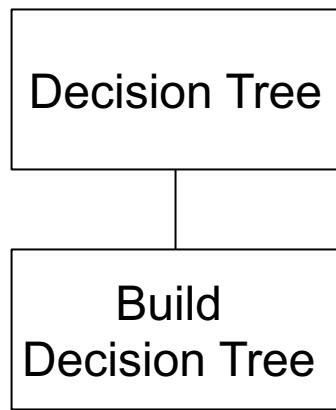
    #      set best theta to 0

    #      set best left list to empty

    #      set best right list to empty

    #      set best class labels after split to empty
```

Main Modules Decision Tree



- Define a node in the tree
- Define the entropy function
- Calculate the information gain to find the best threshold for each feature → then select the best feature based on the information gain → create decision tree (recursively) using the best feature

```

#method to create decision tree
def create_decision_tree(self, dict_of_everything,class_val,eta_min_val):
    global fea_list
    #if all the class labels are same, then we are set
    if len(set(class_val)) ==1:
        #print("Leaf node for set class is",class_val[0],len(class_val))

        return
    #if the no class vales are less than threshold, we assign the class with max values as the class label
    elif len(class_val) < eta_min_val:

        return
    else:
        else:
            #       using the best_feature to get best feature list

            #       store the node name, theta, left split, right split and class labes

            #           call get_remainder_dict to get left tree data

#method to return the major class value using Counter() and .most_common()
def cal_major_class_values(self,class_values):
    return
  
```

For node with instances < n_min, assign class with max instances as the class label



Main Modules Decision Tree

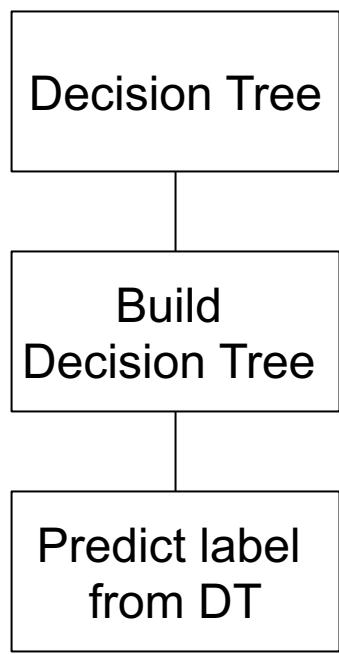
Decision Tree

Build
Decision Tree

- Define a node in the tree
- Define the entropy function
- Calculate the information gain to find the best threshold for each feature → then select the best feature based on the information gain → create decision tree (recursively) using the best feature → get the left subtree feature (left_dict) and right subtree feature

```
def get_remainder_dict(self,dict_of_everything,index_split):  
    global fea_list  
    #     init a split dict  
  
    #     for each key "ke" in dict_of_everything:  
    for ke in dict_of_everything.keys():  
        #         init a value list  
  
        #         init a modified list  
  
        #         get the corresponding values of the key"ke"  
  
        #         for each value and its corresponding index of the key"ke"  
  
        #             if index is not in the index_split:  
  
        #                 append it to the modified list and value list  
  
        #                 add this modified list to the dict  
  
        #             return the splited dict and val list  
    return
```

Main Modules Decision Tree



- Assign a label for each testing sample

```
def classify(self, row, root):  
    #     init the test dict  
  
    #     add row to the dict  
    for k, j in enumerate(row):  
  
        #     set the current node to root  
  
        #     while the current node is not leaf:  
        while not current_node.leaf:  
            #             implement the case whether the current shoud go to the left  
  
            #             implement the case whether the current shoud go to the right  
  
            #             return the calss of the current node  
            return  
  
            #method to the labels for the test data  
def predict(self, X, root):  
    #        predict using the classify()
```

Main Modules Decision Tree

- Calculate accuracy for each fold testing dataset

Evaluation

```
In [26]: #calculating the predicted accuracy
def accuracy_for_predicted_values(test_class_names1,l):
    #      init true and false count to 0

    #      for each prediction,if predict is correct then, true++ else, false++
    for i in range(len(test_class_names1)):

        #      return the acc
    return
```

Pipeline for Decision Tree

