Machine Learning Course Assignment

Nasim Maleki

Dr. Zhang

Department of Computer Science, University of New Brunswick

December 7, 2018

Contents

Chapter 1

ML Algorithm Implementation Report

1.1 Datasets Explanations

There five datasets from UCI data sources we intend to train and test our implmented algorithm with them. However, each dataset has its own characteristics. So based on the type of machine learning algorithm, we need to know these specific feature of each dataset to pre-process them first and them feed our machine learning method. Each dataset is elaborated in the following sections.

1.1.1 Ecoli

In Ecoli Dataset there are 9 columns the first column is the ID of the instance and the last row is the label(class) of instances. The other columns are the features. We need to know that in data pre-processing of this dataset we have to delete the first column because it is not a feature to decide based on it. Without removing this column the final accuracy/results would be wrong. All data are float and real valued in KNN this is okay, but for other algorithms such as ID3 we need to apply a bining method.
labels=('cp', 'im', 'imL', 'imS', 'imU', 'om', 'omL', 'pp')
There are 336 instances.

1.1.2 Breast Cancer

In Breast Cancer Dataset there are 11 columns the first column is the ID of the instance and the last row is the label(class) of instances. The other columns are the features. We need to know that in data preprocessing of this dataset we have to delete the first column becuase it is not a feature to decide based on it. Without removing this column the final accuracy/results would be wrong. In this dataset we have missing value for the 6th column. All data are categorical (numerical).
There are two labels with value of 2 and 4. labels=(2,4)
There are 699 instances.

1.1.3 Car

In Car Dataset there are 7 columns, the last row is the label(class) of instances. The other columns are the features. All data are categorical (numerical). And based on 6 feature we decide the car is acceptable(acc), unacceptable(unacc), etc.
labels=('acc', 'good', 'unacc', 'vgood')
There are 1728 instances.

1.1.4 Mushroom

In Mushroom Dataset there are 23 columns, the first row is the label(class) of instances. The other columns are the features. In this dataset we have missing value for the 11th column All data are categorical.
There are two labels with value of poison and not. labels=('e', 'p)
There are 8124 instances.

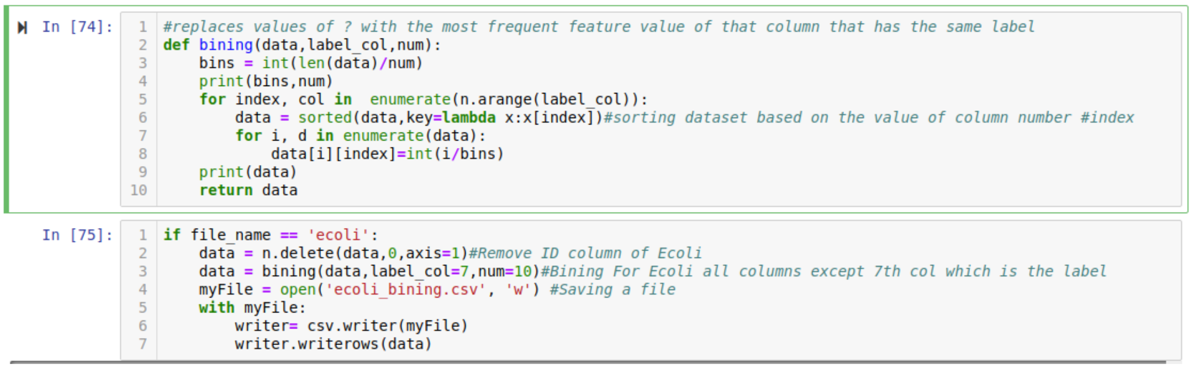
1.1.5 Letter Recognition

In Mushroom Dataset there are 17 columns, the first row is the label(class) of instances which is the alphabet. The other columns are the features. In this dataset we have missing value for the 11th column All data are categorical.
26 labels=('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z')
There are 20000 instances.

1.2 Data Preprocessing and Postprocessing

1.2.1 Bining

When our feature values have continues values we would have problem in classifying them, specifically in ID3, Adaboost,.... We can change the continues values to categorical values by applying bining. Here the method we used for bining continues values is a simple one we just made n bins from minimum value to maximum value(Column is Sorted). All bins have the same number of instances.Then we assign number of each bin to its members. Ecoli has float values and we used this module for this dataset.



In [74]:

```python
#replaces values of ? with the most frequent feature value of that column that has the same label
def bining(data,label_col,num):
    bins = int(len(data)/num)
    print(bins,num)
    for index, col in  enumerate(n.arange(label_col)):
        data = sorted(data,key=lambda x:x[index])#sorting dataset based on the value of column number #index
        for i, d in enumerate(data):
            data[i][index]=int(i/bins)
    print(data)
    return data
```

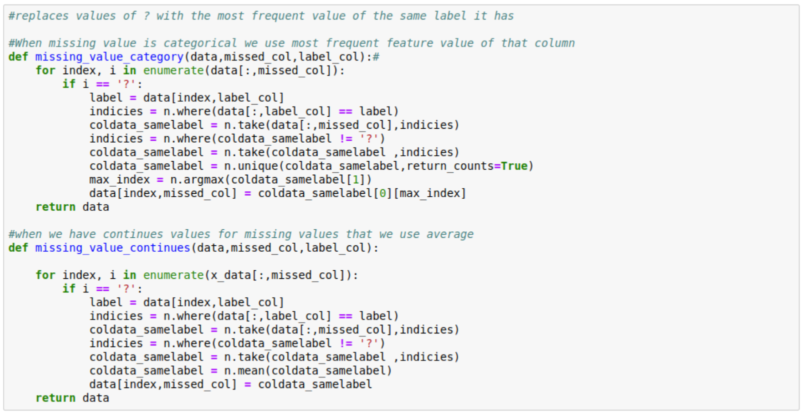
In [75]:

```python
if file_name == 'ecoli':
    data = n.delete(data,0,axis=1)#Remove ID column of Ecoli
    data = bining(data,label_col=7,num=10)#Bining For Ecoli all columns except 7th col which is the label
    myFile = open('ecoli_bining.csv', 'w') #Saving a file
    with myFile:
        writer= csv.writer(myFile)
        writer.writerows(data)
```

Figure 1.1: Bining Function

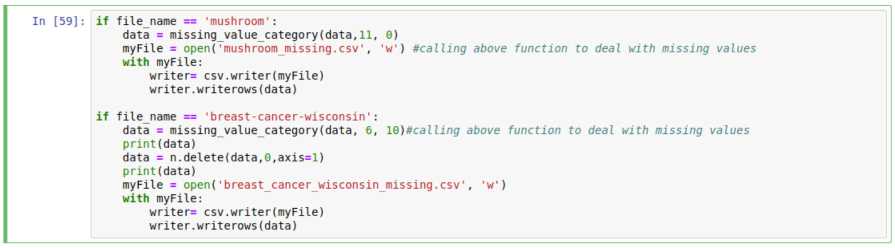
1.2.2 Missing Values

The other important part of preprocessing is to deal with missing values. Here in each column of data that we have missing value we replace it with most frequent feature value of that column that has the same label with the instance that has missing values. Here Mushroom and Breast Cancer datasets have missing values.



```python
#replaces values of ? with the most frequent value of the same label it has

#When missing value is categorical we use most frequent feature value of that column
def missing_value_category(data,missed_col,label_col):#
    for index, i in enumerate(data[:,missed_col]):
        if i == '?':
            label = data[index,label_col]
            indicies = n.where(data[:,label_col] == label)
            coldata_samelabel = n.take(data[:,missed_col],indicies)
            indicies = n.where(coldata_samelabel != '?')
            coldata_samelabel = n.take(coldata_samelabel ,indicies)
            coldata_samelabel = n.unique(coldata_samelabel,return_counts=True)
            max_index = n.argmax(coldata_samelabel[1])
            data[index,missed_col] = coldata_samelabel[0][max_index]
    return data

#when we have continues values for missing values that we use average
def missing_value_continues(data,missed_col,label_col):

    for index, i in enumerate(x_data[:,missed_col]):
        if i == '?':
            label = data[index,label_col]
            indicies = n.where(data[:,label_col] == label)
            coldata_samelabel = n.take(data[:,missed_col],indicies)
            indicies = n.where(coldata_samelabel != '?')
            coldata_samelabel = n.take(coldata_samelabel ,indicies)
            coldata_samelabel = n.mean(coldata_samelabel)
            data[index,missed_col] = coldata_samelabel
    return data
```

Figure 1.2: Missing Value

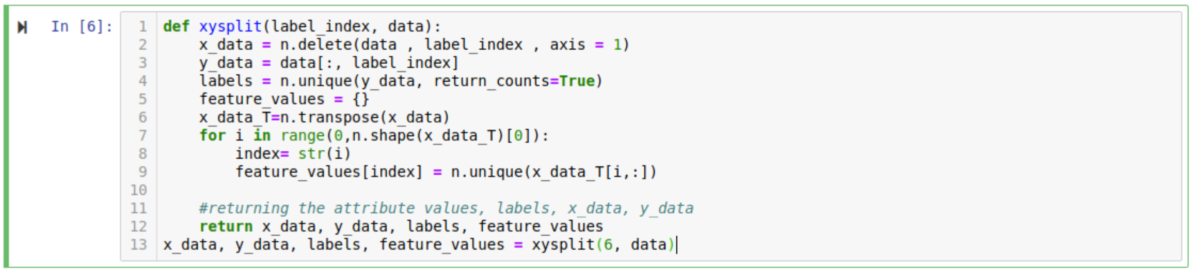


```
In [59]: if file_name == 'mushroom':
             data = missing_value_category(data,11, 0)
             myFile = open('mushroom_missing.csv', 'w') #calling above function to deal with missing values
             with myFile:
                 writer= csv.writer(myFile)
                 writer.writerows(data)

         if file_name == 'breast-cancer-wisconsin':
             data = missing_value_category(data, 6, 10)#calling above function to deal with missing values
             print(data)
             data = n.delete(data,0,axis=1)
             print(data)
             myFile = open('breast_cancer_wisconsin_missing.csv', 'w')
             with myFile:
                 writer= csv.writer(myFile)
                 writer.writerows(data)
```

Figure 1.3: Missing value for Mushroom and Breast Cancer

1.2.3 xysplit Function

When we read our CSV file, we will get a matrix with two axis(0,1). We have to split label column and feature columns. Besides that we get feature values(unique values) of each column and classes.

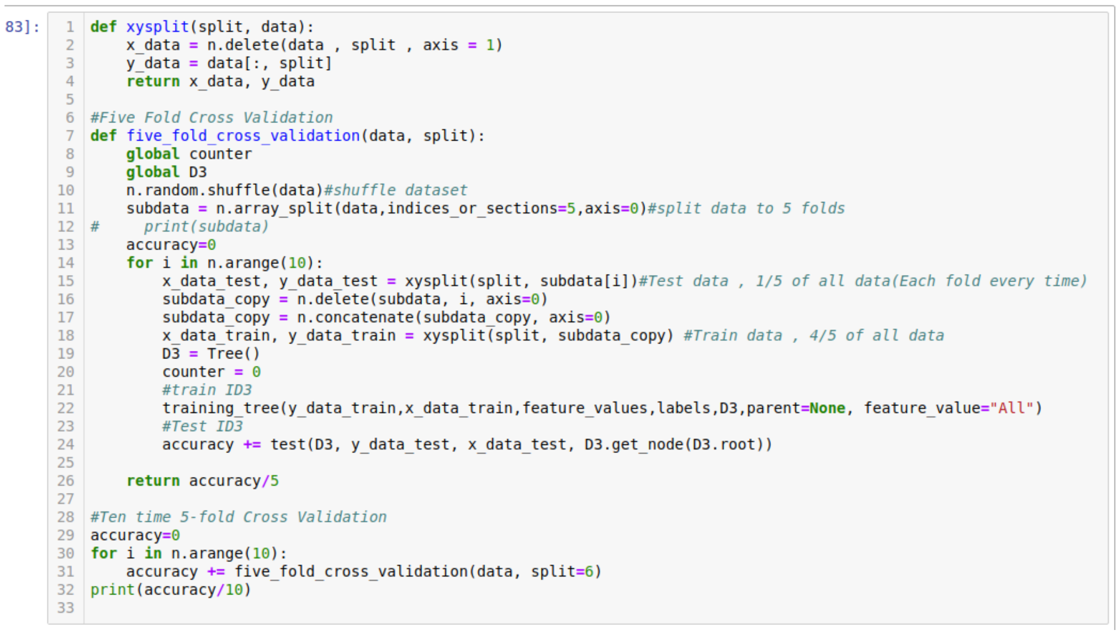


```
In [6]: def xysplit(label_index, data):
            x_data = n.delete(data , label_index , axis = 1)
            y_data = data[:, label_index]
            labels = n.unique(y_data, return_counts=True)
            feature_values = {}
            x_data_T=n.transpose(x_data)
            for i in range(0,n.shape(x_data_T)[0]):
                index= str(i)
                feature_values[index] = n.unique(x_data_T[i,:])

            #returning the attribute values, labels, x_data, y_data
            return x_data, y_data, labels, feature_values
        x_data, y_data, labels, feature_values = xysplit(6, data)
```

Figure 1.4: xysplit function

1.2.4 10 times 5-fold Cross Validation

When we train and then test our classification method. It is better to shuffle our data train it with a part of data and then test it with the rest. If we do this procedure again and again, our result is more fair because all instances will distribute. 5-fold cross validation means our dataset is divided to 5 parts each time we train our dataset with 4/5 of data and test it with 1/5. After 5 rounds again we repeat these training and testing 10 times and get the average of all 10 results.



```
83]:
def xysplit(split, data):
    x_data = n.delete(data , split , axis = 1)
    y_data = data[:, split]
    return x_data, y_data

#Five Fold Cross Validation
def five_fold_cross_validation(data, split):
    global counter
    global D3
    n.random.shuffle(data)#shuffle dataset
    subdata = n.array_split(data,indices_or_sections=5,axis=0)#split data to 5 folds
#     print(subdata)
    accuracy=0
    for i in n.arange(10):
        x_data_test, y_data_test = xysplit(split, subdata[i])#Test data , 1/5 of all data(Each fold every time)
        subdata_copy = n.delete(subdata, i, axis=0)
        subdata_copy = n.concatenate(subdata_copy, axis=0)
        x_data_train, y_data_train = xysplit(split, subdata_copy) #Train data , 4/5 of all data
        D3 = Tree()
        counter = 0
        #train ID3
        training_tree(y_data_train,x_data_train,feature_values,labels,D3,parent=None, feature_value="All")
        #Test ID3
        accuracy += test(D3, y_data_test, x_data_test, D3.get_node(D3.root))

    return accuracy/5

#Ten time 5-fold Cross Validation
accuracy=0
for i in n.arange(10):
    accuracy += five_fold_cross_validation(data, split=6)
print(accuracy/10)
```

Figure 1.5: E.g. 10 times 5-fold Cross Validation for ID3

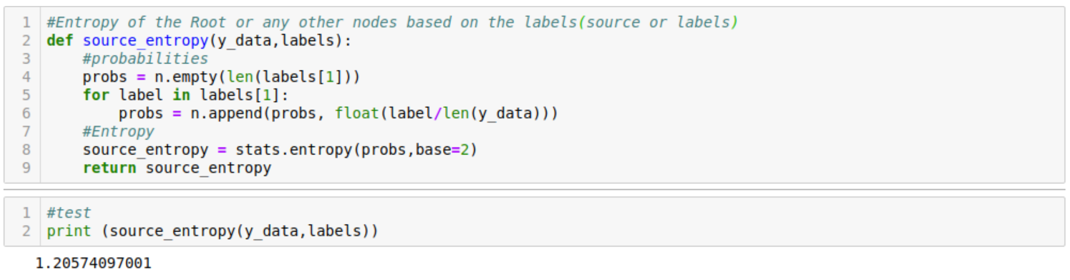
1.3 Implementation

There are 6 algorithms have been implemented. Different functions and modules of each would be elaborated below. 5 mentioned dataset have been fed and then tested in each algorithm with 10 times 5-fold cross validation. The accuracy of each would be shown in the table.

1.3.1 ID3

ID3 algorithm which is a complete decision tree of our dataset has used different functions to learn the data and make the model. The whole idea of decision tree is that first we need to compute entropy of each node (for the part of data it has), then we compute sum of the entropy for all attribute values of a feature. After computing information gain of all features, we will choose a feature with the best information gain. We continue in this way till all the nodes of the tree get built. We stop expanding a node when there is no partition of data in that node or the node is pure (we know the label) or there is no more feature.

- **Source Entropy**
  Here in source entropy function we compute the p*log(p) for all labels in a for loop when input data is the last column of dataset(target value column).



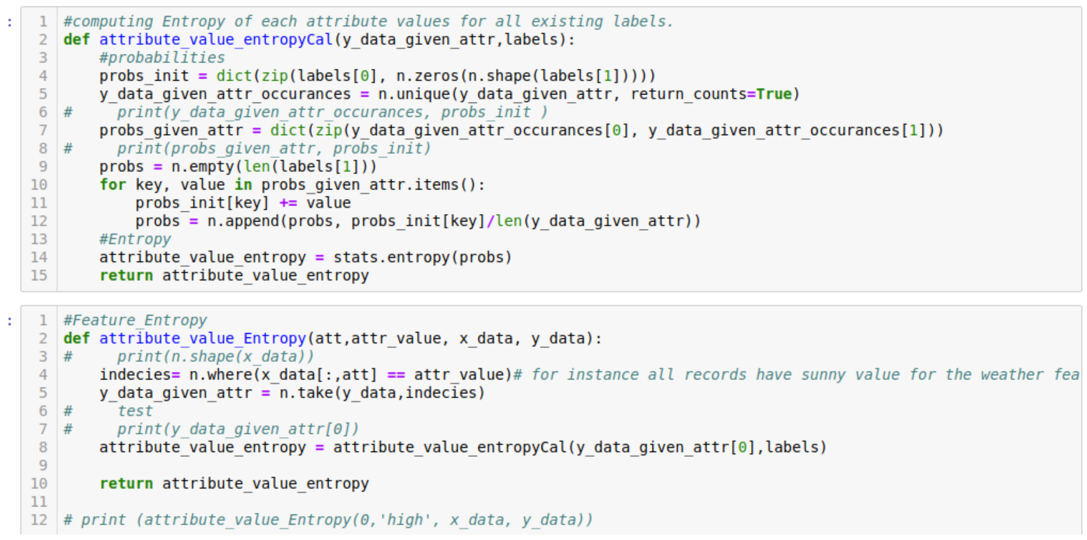
```python
#Entropy of the Root or any other nodes based on the labels(source or labels)
def source_entropy(y_data,labels):
    #probabilities
    probs = n.empty(len(labels[1]))
    for label in labels[1]:
        probs = n.append(probs, float(label/len(y_data)))
    #Entropy
    source_entropy = stats.entropy(probs,base=2)
    return source_entropy
```

```python
#test
print (source_entropy(y_data,labels))
```

```
1.20574097001
```

Figure 1.6: Computing Source Entropy

- **Attribute Value Entropy**
  In "attribute value entropy" function we call the "source entropy" function when input data is the labels of the rows that have a specific attribute value. This function would be called for all attribute values of a feature to compute its information gain.

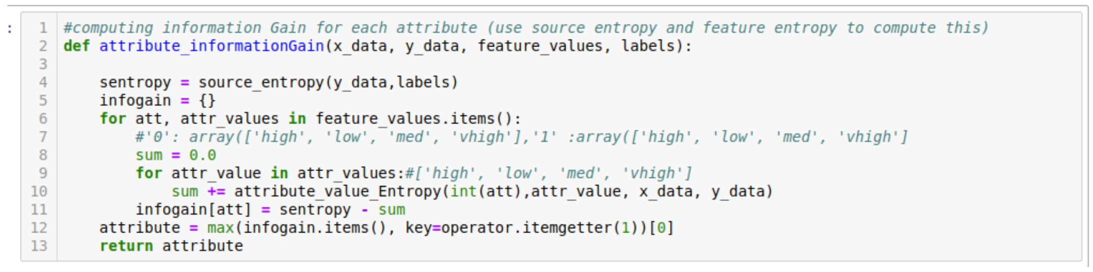


```python
#computing Entropy of each attribute values for all existing labels.
def attribute_value_entropyCal(y_data_given_attr,labels):
    #probabilities
    probs_init = dict(zip(labels[0], n.zeros(n.shape(labels[1]))))
    y_data_given_attr_occurances = n.unique(y_data_given_attr, return_counts=True)
#     print(y_data_given_attr_occurances, probs_init )
    probs_given_attr = dict(zip(y_data_given_attr_occurances[0], y_data_given_attr_occurances[1]))
#     print(probs_given_attr, probs_init)
    probs = n.empty(len(labels[1]))
    for key, value in probs_given_attr.items():
        probs_init[key] += value
        probs = n.append(probs, probs_init[key]/len(y_data_given_attr))
    #Entropy
    attribute_value_entropy = stats.entropy(probs)
    return attribute_value_entropy
```

```python
#Feature Entropy
def attribute_value_Entropy(att,attr_value, x_data, y_data):
#     print(n.shape(x_data))
    indecies= n.where(x_data[:,att] == attr_value)# for instance all records have sunny value for the weather fea
    y_data_given_attr = n.take(y_data,indecies)
#     test
#     print(y_data_given_attr[0])
    attribute_value_entropy = attribute_value_entropyCal(y_data_given_attr[0],labels)

    return attribute_value_entropy

# print (attribute_value_Entropy(0,'high', x_data, y_data))
```

Figure 1.7: Attribute Value Entropy

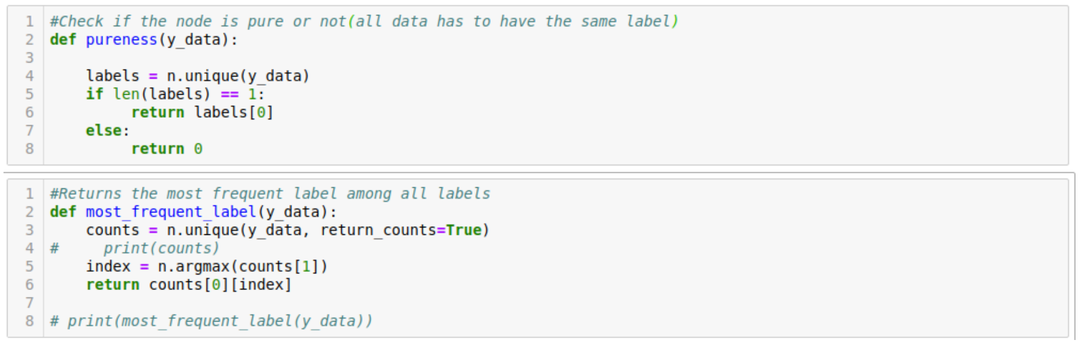
- **Information Gain**
  After Computing source entropy of a node and source entropy for all unselected features. By computing (source_entropy - Sum(Attribute_value_source_entropy)), we will get information gain. The we choose the feature with the most info gain.



```python
#computing information Gain for each attribute (use source entropy and feature entropy to compute this)
def attribute_informationGain(x_data, y_data, feature_values, labels):

    sentropy = source_entropy(y_data,labels)
    infogain = {}
    for att, attr_values in feature_values.items():
        #'0': array(['high', 'low', 'med', 'vhigh'],'1' :array(['high', 'low', 'med', 'vhigh']
        sum = 0.0
        for attr_value in attr_values:#['high', 'low', 'med', 'vhigh']
            sum += attribute_value_Entropy(int(att),attr_value, x_data, y_data)
        infogain[att] = sentropy - sum
    attribute = max(infogain.items(), key=operator.itemgetter(1))[0]
    return attribute
```

Figure 1.8: Computing Information Gain for Each Feature

- **Pureness Test and Most Frequnet Label**
  There are two functions here we implemented. The first one is "pureness. It will check if the partition of the dataset belonging to a node is pure or not. If all the labels are the same it is pure. The second function is implemented to compute the label of data. Argument of this function is last column of data(y__data).

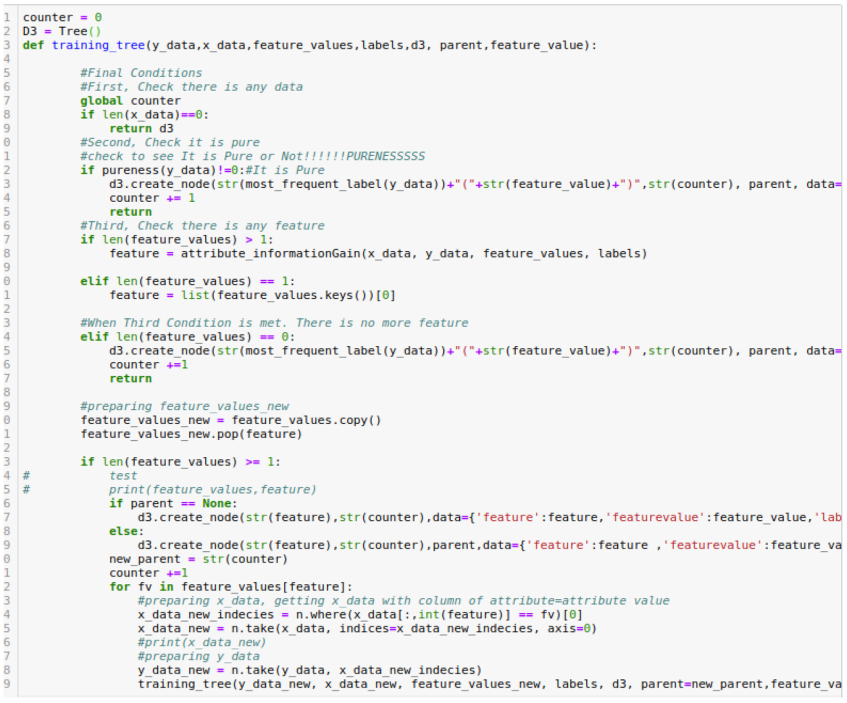


```python
#Check if the node is pure or not(all data has to have the same label)
def pureness(y_data):

    labels = n.unique(y_data)
    if len(labels) == 1:
         return labels[0]
    else:
         return 0
```

```python
#Returns the most frequent label among all labels
def most_frequent_label(y_data):
    counts = n.unique(y_data, return_counts=True)
#     print(counts)
    index = n.argmax(counts[1])
    return counts[0][index]

# print(most_frequent_label(y_data))
```

Figure 1.9: Pureness Test and Most Frequent Label

- **Training Function**
  First it is check three stop condition which are "pure node", "no feature", "no data". If these condition have been met we do not expanding the node and return. Otherwise we expand the node based on its children(each node has edges with attribute values of the parent node). Then we decide for the next feature based on maximum information gain of the rest features in that branch. We used Tree Data structure to make the tree, nodes, and leaves, ... More details have been explained in the source code by blue comments. Test part of ID3 source code has put in "cross validation section" above. For testing we just need to traverse the tree to reach a leaf(label). If the label is similar to the label of test data it means we have detected correctly.



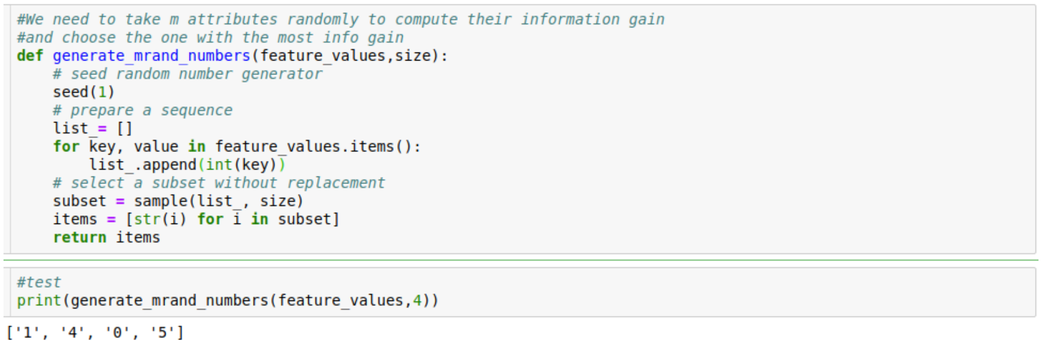
```
counter = 0
D3 = Tree()
def training_tree(y_data,x_data,feature_values,labels,d3, parent,feature_value):

        #Final Conditions
        #First, Check there is any data
        global counter
        if len(x_data)==0:
            return d3
        #Second, Check it is pure
        #check to see It is Pure or Not!!!!!!PURENESSSSS
        if pureness(y_data)!=0:#It is Pure
            d3.create_node(str(most_frequent_label(y_data))+"("+str(feature_value)+")",str(counter), parent, data=
            counter += 1
            return
        #Third, Check there is any feature
        if len(feature_values) > 1:
            feature = attribute_informationGain(x_data, y_data, feature_values, labels)

        elif len(feature_values) == 1:
            feature = list(feature_values.keys())[0]

        #When Third Condition is met. There is no more feature
        elif len(feature_values) == 0:
            d3.create_node(str(most_frequent_label(y_data))+"("+str(feature_value)+")",str(counter), parent, data=
            counter +=1
            return

        #preparing feature_values_new
        feature_values_new = feature_values.copy()
        feature_values_new.pop(feature)

        if len(feature_values) >= 1:
#           test
#           print(feature_values,feature)
            if parent == None:
                d3.create_node(str(feature),str(counter),data={'feature':feature,'featurevalue':feature_value,'lab
            else:
                d3.create_node(str(feature),str(counter),parent,data={'feature':feature ,'featurevalue':feature_va
            new_parent = str(counter)
            counter +=1
            for fv in feature_values[feature]:
                #preparing x_data, getting x_data with column of attribute=attribute value
                x_data_new_indecies = n.where(x_data[:,int(feature)] == fv)[0]
                x_data_new = n.take(x_data, indices=x_data_new_indecies, axis=0)
                #print(x_data_new)
                #preparing y_data
                y_data_new = n.take(y_data, x_data_new_indecies)
                training_tree(y_data_new, x_data_new, feature_values_new, labels, d3, parent=new_parent,feature_va
```

Figure 1.10: Main Module of ID3 for Training

1.3.2 Random Forest

Because ID3 is a greedy algorithm searching hypothesis, we add a randomness feature to improve it. In random forest , there is two main module the first module is Bagging and the second is RF itself which is modification of ID3.

- **Choose M random Features**
  This function is returning m random numbers which can be considered as indices of feature columns.



```
#We need to take m attributes randomly to compute their information gain
#and choose the one with the most info gain
def generate_mrand_numbers(feature_values,size):
    # seed random number generator
    seed(1)
    # prepare a sequence
    list_ = []
    for key, value in feature_values.items():
        list_.append(int(key))
    # select a subset without replacement
    subset = sample(list_, size)
    items = [str(i) for i in subset]
    return items
```

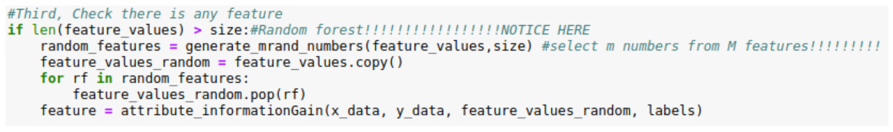
```
#test
print(generate_mrand_numbers(feature_values,4))
```

```
['1', '4', '0', '5']
```

Figure 1.11: Choose M random Features

- **Random Forest Training(Training Modified ID3)**
  The training function for RF is the same as ID3, only we need to change the part that we want to choose best feature based on info gain. Instead

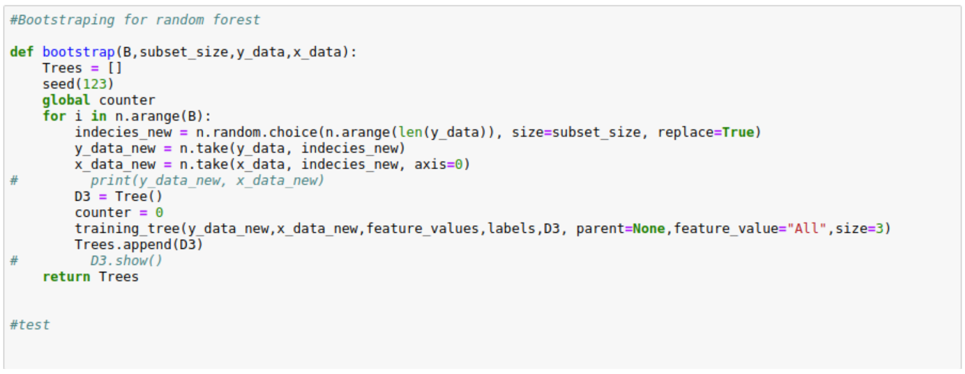
of choosing feature from all remaining features, first we choose m random features from the remaining feature_values list. Then computing information gain just for randomly selected features. Here you see only the part if RF code that has been added to ID3 source code.



```
#Third, Check there is any feature
if len(feature_values) > size:#Random forest!!!!!!!!!!!!!!!!NOTICE HERE
    random_features = generate_mrand_numbers(feature_values,size) #select m numbers from M features!!!!!!!!
    feature_values_random = feature_values.copy()
    for rf in random_features:
        feature_values_random.pop(rf)
    feature = attribute_informationGain(x_data, y_data, feature_values_random, labels)
```

Figure 1.12: Random Forest Training

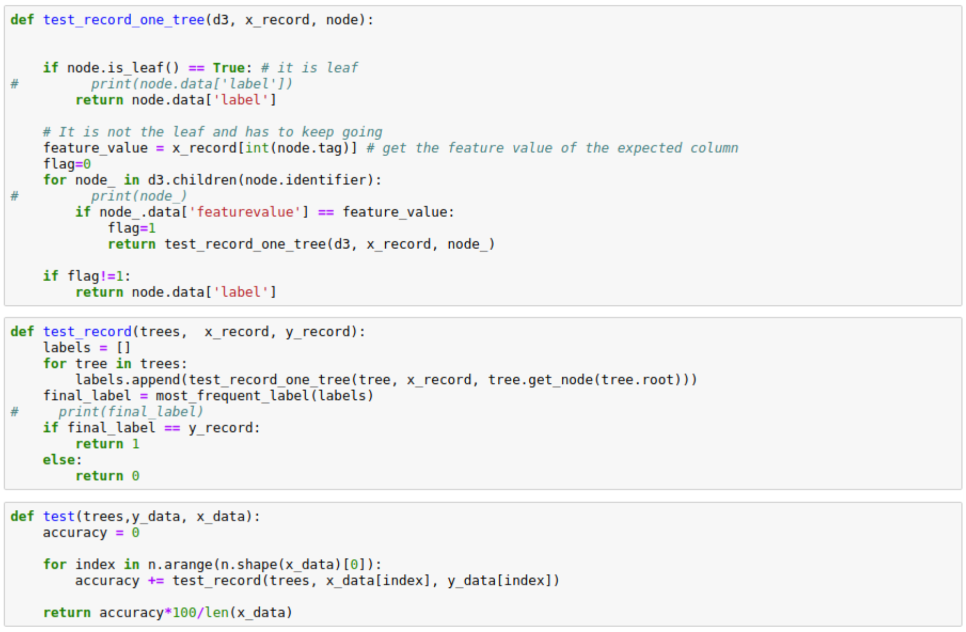
- Make Forest, Training trees(ID3 with Bagging) Another important module of RF is Bagging part. In bagging, we choose n times m random rows of training dataset. Each time we make a classifier(train random forest) with selected rows and make a bag. Final label is the vote among the label of each bag.



```
#Bootstraping for random forest

def bootstrap(B,subset_size,y_data,x_data):
    Trees = []
    seed(123)
    global counter
    for i in n.arange(B):
        indecies_new = n.random.choice(n.arange(len(y_data)), size=subset_size, replace=True)
        y_data_new = n.take(y_data, indecies_new)
        x_data_new = n.take(x_data, indecies_new, axis=0)
#         print(y_data_new, x_data_new)
        D3 = Tree()
        counter = 0
        training_tree(y_data_new,x_data_new,feature_values,labels,D3, parent=None,feature_value="All",size=3)
        Trees.append(D3)
#         D3.show()
    return Trees


#test
```

Figure 1.13: Make Forest

- Test Functions For test, Test_record_one_tree is computing the label of one record of a test set in one bag. Test record would vote among the labels. Test function would test all rows of the test set in the RF Model.


```
def test_record_one_tree(d3, x_record, node):


    if node.is_leaf() == True: # it is leaf
#         print(node.data['label'])
        return node.data['label']

    # It is not the leaf and has to keep going
    feature_value = x_record[int(node.tag)] # get the feature value of the expected column
    flag=0
    for node_ in d3.children(node.identifier):
#         print(node_)
        if node_.data['featurevalue'] == feature_value:
            flag=1
            return test_record_one_tree(d3, x_record, node_)

    if flag!=1:
        return node.data['label']
```

```
def test_record(trees,  x_record, y_record):
    labels = []
    for tree in trees:
        labels.append(test_record_one_tree(tree, x_record, tree.get_node(tree.root)))
    final_label = most_frequent_label(labels)
#     print(final_label)
    if final_label == y_record:
        return 1
    else:
        return 0
```

```
def test(trees,y_data, x_data):
    accuracy = 0

    for index in n.arange(n.shape(x_data)[0]):
        accuracy += test_record(trees, x_data[index], y_data[index])

    return accuracy*100/len(x_data)
```

Figure 1.14: Test Function

1.3.3 Adaboost on Tree Stumps

Tree Stumps is a one level ID3 with a restriction bias of being one level tree. So it can behave not accurately and to boosting this weak classifier we use Adaboost algorithm. For Adaboost algorithm, there are two main modules the first one is implementation of Tree Stumps(simplified ID3). And the second module is Adaboost module.

- **Make subset of main training dataset randomly**
  Following functions are used to initialize the weight for each row of the training data and choose the most probable rows(based on the weights) randomly.

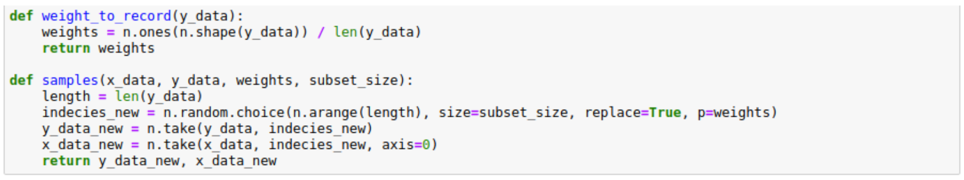
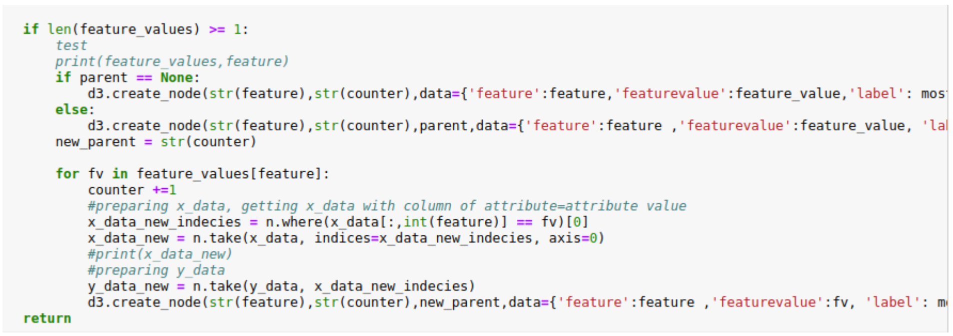

```
def weight_to_record(y_data):
    weights = n.ones(n.shape(y_data)) / len(y_data)
    return weights

def samples(x_data, y_data, weights, subset_size):
    length = len(y_data)
    indecies_new = n.random.choice(n.arange(length), size=subset_size, replace=True, p=weights)
    y_data_new = n.take(y_data, indecies_new)
    x_data_new = n.take(x_data, indecies_new, axis=0)
    return y_data_new, x_data_new
```

Figure 1.15: Make subset of main training dataset randomly

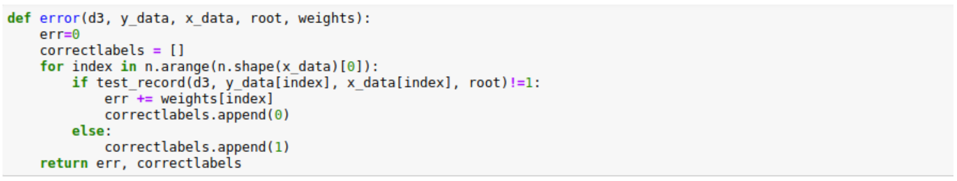
- **Tree Stumps Classifier**
  If you refer to ID3 function above you see we keep going to make the tree level by level , node by node. But here we only have one level tree we do not have a recursive tree. Once the root is made all its children node in the next level would be made. This is one level tree. However in the provided source code, we not only developed adaboost for tree stums, but we used it for ID3 too.



```
if len(feature_values) >= 1:
    test
    print(feature_values,feature)
    if parent == None:
        d3.create_node(str(feature),str(counter),data={'feature':feature,'featurevalue':feature_value,'label': mos
    else:
        d3.create_node(str(feature),str(counter),parent,data={'feature':feature ,'featurevalue':feature_value, 'la
    new_parent = str(counter)

    for fv in feature_values[feature]:
        counter +=1
        #preparing x_data, getting x_data with column of attribute=attribute value
        x_data_new_indecies = n.where(x_data[:,int(feature)] == fv)[0]
        x_data_new = n.take(x_data, indices=x_data_new_indecies, axis=0)
        #print(x_data_new)
        #preparing y_data
        y_data_new = n.take(y_data, x_data_new_indecies)
        d3.create_node(str(feature),str(counter),new_parent,data={'feature':feature ,'featurevalue':fv, 'label': m
return
```

Figure 1.16: Training Adaboost Classifier(One Level ID3)

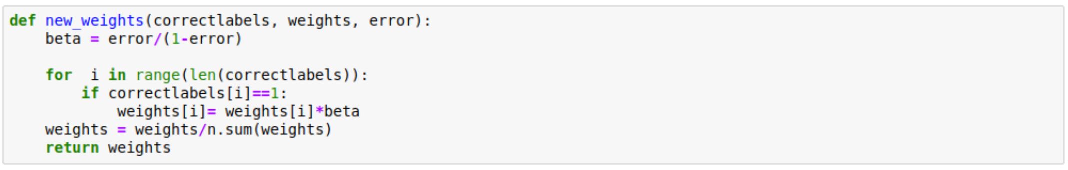
- **Error**
  This is the error function in Adaboost. After making a classifier, it would test all training data in that classifier which is made only by a subset of train dataset. Each instance is detected incorrectly, it increase the error value. Here is the function of error.



```
def error(d3, y_data, x_data, root, weights):
    err=0
    correctlabels = []
    for index in n.arange(n.shape(x_data)[0]):
        if test_record(d3, y_data[index], x_data[index], root)!=1:
            err += weights[index]
            correctlabels.append(0)
        else:
            correctlabels.append(1)
    return err, correctlabels
```

Figure 1.17: Error Function

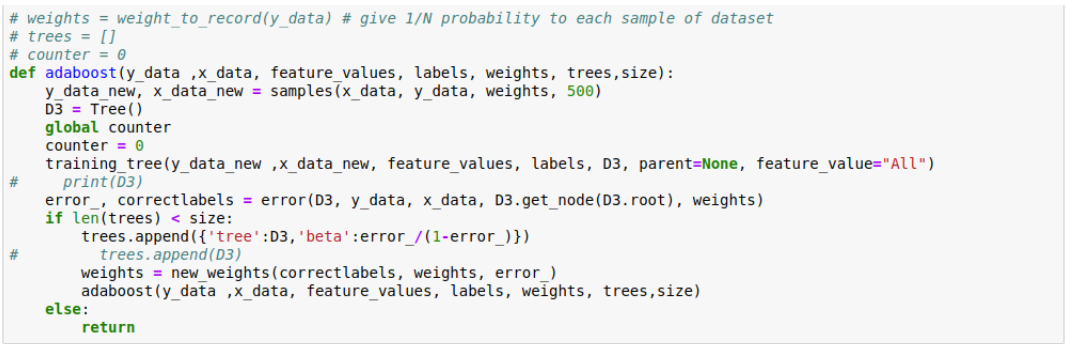
- **Update Weights**
  In adaboost we need to take care of all wrongly detected instances. So we increase the weight of them. After increasing the weights we normalize them.



```
def new_weights(correctlabels, weights, error):
    beta = error/(1-error)

    for  i in range(len(correctlabels)):
        if correctlabels[i]==1:
            weights[i]= weights[i]*beta
    weights = weights/n.sum(weights)
    return weights
```

Figure 1.18: Update Weights

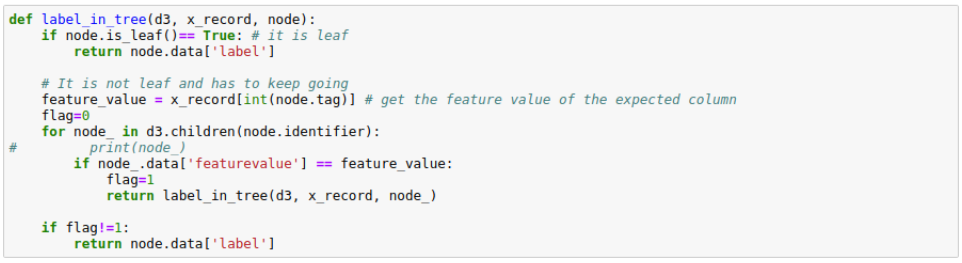
- **Training Adaboost**
  Here we keep going to train tree stumps and continuously compute the errors for each. In each training step we increase the weights of incorrectly classified rows. Then based on new weights, we select subset of training dataset to train another tree stumps. We keep going to do this until get a lower error. After making tree stumps each of which has a score based on their errors.



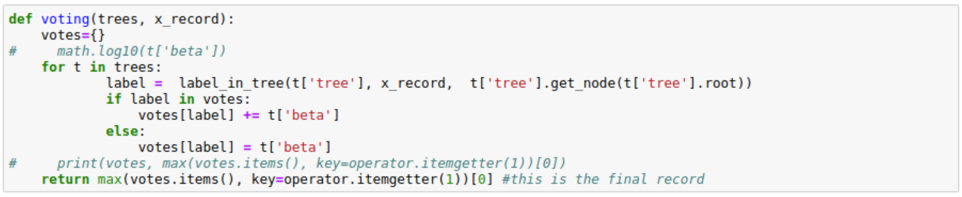
```
# weights = weight_to_record(y_data) # give 1/N probability to each sample of dataset
# trees = []
# counter = 0
def adaboost(y_data ,x_data, feature_values, labels, weights, trees,size):
    y_data_new, x_data_new = samples(x_data, y_data, weights, 500)
    D3 = Tree()
    global counter
    counter = 0
    training_tree(y_data_new ,x_data_new, feature_values, labels, D3, parent=None, feature_value="All")
#     print(D3)
    error_, correctlabels = error(D3, y_data, x_data, D3.get_node(D3.root), weights)
    if len(trees) < size:
        trees.append({'tree':D3,'beta':error_/(1-error_)})
#         trees.append(D3)
        weights = new_weights(correctlabels, weights, error_)
        adaboost(y_data ,x_data, feature_values, labels, weights, trees,size)
    else:
        return
```

Figure 1.19: Training Adaboost

- **Voting Function**
  After training Adaboost, we have to test the data test. The final label is the weighted voting among the label of trained tree stumps.



```
def label_in_tree(d3, x_record, node):
    if node.is_leaf()== True: # it is leaf
        return node.data['label']

    # It is not leaf and has to keep going
    feature_value = x_record[int(node.tag)] # get the feature value of the expected column
    flag=0
    for node_ in d3.children(node.identifier):
#         print(node_)
        if node_.data['featurevalue'] == feature_value:
            flag=1
            return label_in_tree(d3, x_record, node_)

    if flag!=1:
        return node.data['label']
```

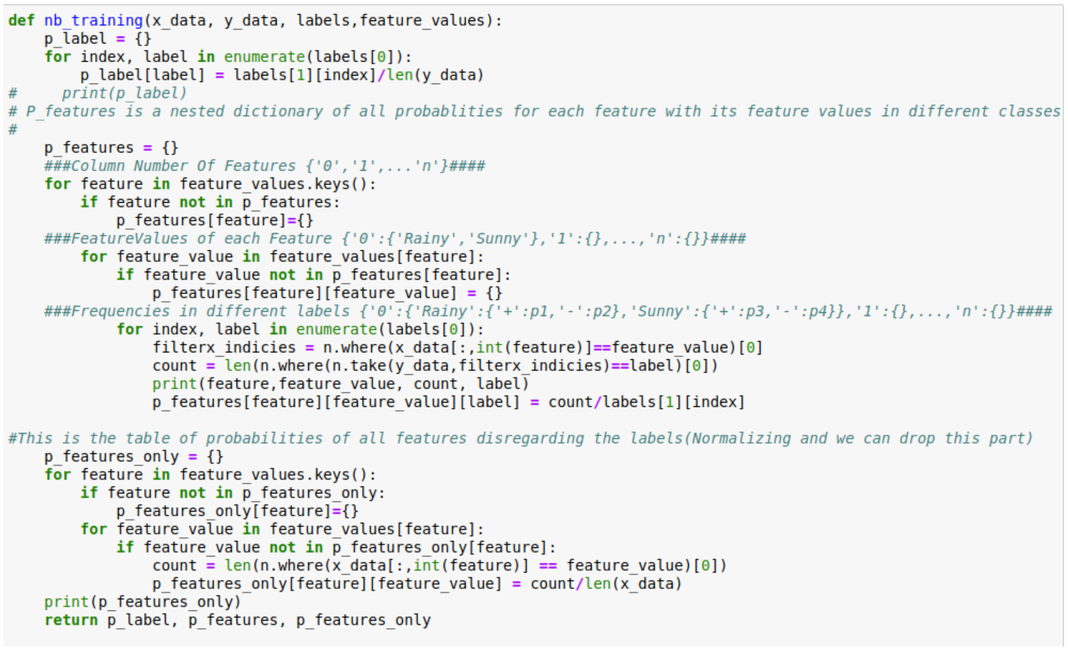


```
def voting(trees, x_record):
    votes={}
#     math.log10(t['beta'])
    for t in trees:
            label =  label_in_tree(t['tree'], x_record,  t['tree'].get_node(t['tree'].root))
            if label in votes:
                votes[label] += t['beta']
            else:
                votes[label] = t['beta']
#     print(votes, max(votes.items(), key=operator.itemgetter(1))[0])
    return max(votes.items(), key=operator.itemgetter(1))[0] #this is the final record
```

Figure 1.20: Voting Function

1.3.4 Naive Bayes

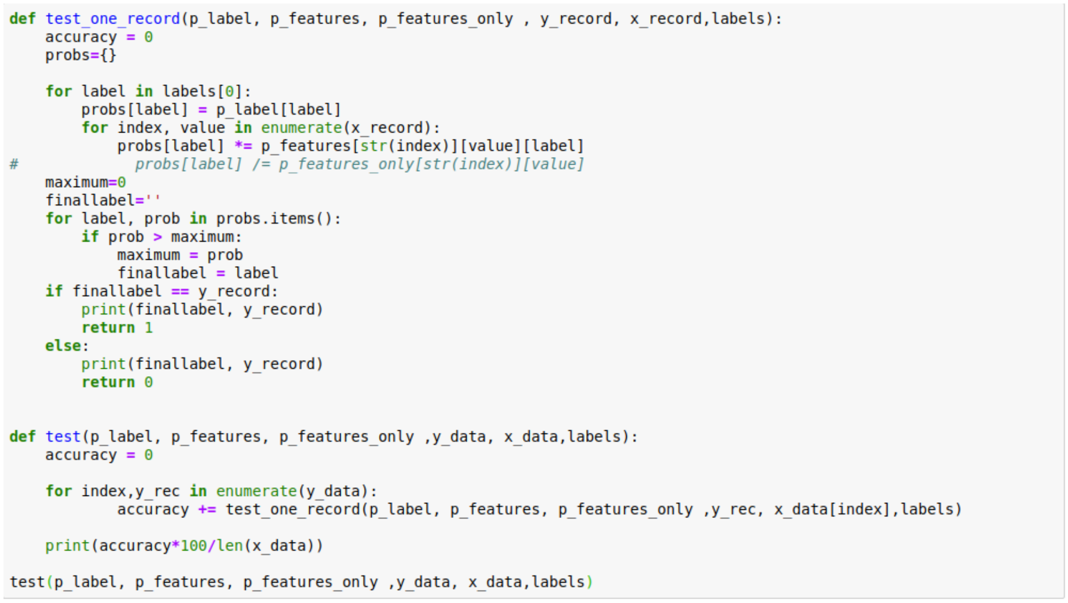
- **Training the Naive Bayes Classifier and Making Probability Tables**
  In training the Naive Bayes Classifier, the only information we need based on our training data is some probability tables. So training phase in Naive Bayes is supposed to make probability tables. The first probability table in the source code is p_table computing the probability of existing labels in the training set. The second table(P_features) is to compute the probability of the labels when a feature has a specific feature value. We also have zero handling to prevent the zero probability and to this end we add one to the count in nominator and add the number of labels to the count of denominator. The last table is to compute the probablity of the feature values in each column in training dataset. However this table is used for normalizing and because it is fixed value can be dropped.



```python
def nb_training(x_data, y_data, labels,feature_values):
    p_label = {}
    for index, label in enumerate(labels[0]):
        p_label[label] = labels[1][index]/len(y_data)
#     print(p_label)
# P_features is a nested dictionary of all probablities for each feature with its feature values in different classes
#
    p_features = {}
    ###Column Number Of Features {'0','1',...'n'}####
    for feature in feature_values.keys():
        if feature not in p_features:
            p_features[feature]={}
    ###FeatureValues of each Feature {'0':{'Rainy','Sunny'},'1':{},...,'n':{}}####
        for feature_value in feature_values[feature]:
            if feature_value not in p_features[feature]:
                p_features[feature][feature_value] = {}
    ###Frequencies in different labels {'0':{'Rainy':{'+':p1,'-':p2},'Sunny':{'+':p3,'-':p4}},'1':{},...,'n':{}}####
            for index, label in enumerate(labels[0]):
                filterx_indicies = n.where(x_data[:,int(feature)]==feature_value)[0]
                count = len(n.where(n.take(y_data,filterx_indicies)==label)[0])
                print(feature,feature_value, count, label)
                p_features[feature][feature_value][label] = count/labels[1][index]

#This is the table of probabilities of all features disregarding the labels(Normalizing and we can drop this part)
    p_features_only = {}
    for feature in feature_values.keys():
        if feature not in p_features_only:
            p_features_only[feature]={}
        for feature_value in feature_values[feature]:
            if feature_value not in p_features_only[feature]:
                count = len(n.where(x_data[:,int(feature)] == feature_value)[0])
                p_features_only[feature][feature_value] = count/len(x_data)
    print(p_features_only)
    return p_label, p_features, p_features_only
```

Figure 1.21: Naive Bayes- Training

- **Testing Naive Bayes**
  To test naive bayes we used naive bayes formula and compute the probability of each label for incoming instance and choose the label for that instance that has the most probability.



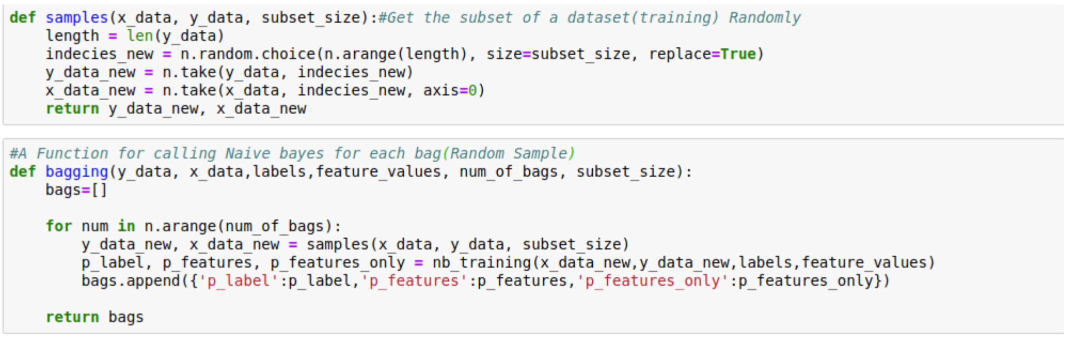
```python
def test_one_record(p_label, p_features, p_features_only , y_record, x_record,labels):
    accuracy = 0
    probs={}

    for label in labels[0]:
        probs[label] = p_label[label]
        for index, value in enumerate(x_record):
            probs[label] *= p_features[str(index)][value][label]
#             probs[label] /= p_features_only[str(index)][value]
    maximum=0
    finallabel=''
    for label, prob in probs.items():
        if prob > maximum:
            maximum = prob
            finallabel = label
    if finallabel == y_record:
        print(finallabel, y_record)
        return 1
    else:
        print(finallabel, y_record)
        return 0


def test(p_label, p_features, p_features_only ,y_data, x_data,labels):
    accuracy = 0

    for index,y_rec in enumerate(y_data):
            accuracy += test_one_record(p_label, p_features, p_features_only ,y_rec, x_data[index],labels)

    print(accuracy*100/len(x_data))

test(p_label, p_features, p_features_only ,y_data, x_data,labels)
```

Figure 1.22: Naive Bayes- Testing

1.3.5 Naive Bayes with Bagging

The only difference between simple NB and NB with Bagging is the bagging module. So we do not explain NB again here. For bagging we need to have few baggs which are the NB classifier. To have this, we randomly get instances from training dataset and feed to NB classification. We do this again in few rounds. Finally we have a number of classifiers. The to decide what is the label, we have voting and we will choose the most frequent label as final label.

- Bagging



```python
def samples(x_data, y_data, subset_size):#Get the subset of a dataset(training) Randomly
    length = len(y_data)
    indecies_new = n.random.choice(n.arange(length), size=subset_size, replace=True)
    y_data_new = n.take(y_data, indecies_new)
    x_data_new = n.take(x_data, indecies_new, axis=0)
    return y_data_new, x_data_new
```

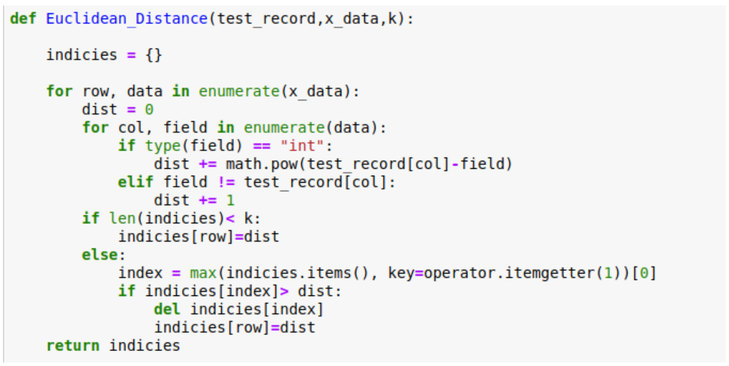
```python
#A Function for calling Naive bayes for each bag(Random Sample)
def bagging(y_data, x_data,labels,feature_values, num_of_bags, subset_size):
    bags=[]

    for num in n.arange(num_of_bags):
        y_data_new, x_data_new = samples(x_data, y_data, subset_size)
        p_label, p_features, p_features_only = nb_training(x_data_new,y_data_new,labels,feature_values)
        bags.append({'p_label':p_label,'p_features':p_features,'p_features_only':p_features_only})

    return bags
```

Figure 1.23: Naive Bayes With Bagging- Training

1.3.6 KNN

KNN is a lazy classification it means we do not have any training phase or a final model as a classifier to test our data. We wait for a test instance and compute the distances of this instance with training data we choose k nearest neighbours and vote among k labels. Here we used two distance function, Euclidean and Manhattan. If our data are categorical for computing their distnaces we used hamming distance it means if the feature value of the instance is the same as feature value of training instance the difference is zero otherwise it is one.

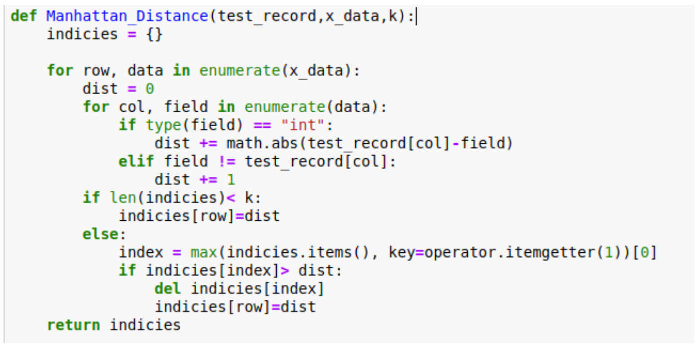
- **Euclidean Distance Function**
  Euclidean Distance $= \sqrt(\sum_{i=1}^{n} (xi - x'i)^2)$



```python
def Euclidean_Distance(test_record,x_data,k):

    indicies = {}

    for row, data in enumerate(x_data):
        dist = 0
        for col, field in enumerate(data):
            if type(field) == "int":
                dist += math.pow(test_record[col]-field)
            elif field != test_record[col]:
                dist += 1
        if len(indicies)< k:
            indicies[row]=dist
        else:
            index = max(indicies.items(), key=operator.itemgetter(1))[0]
            if indicies[index]> dist:
                del indicies[index]
                indicies[row]=dist
    return indicies
```

Figure 1.24: Euclidean Distance Function

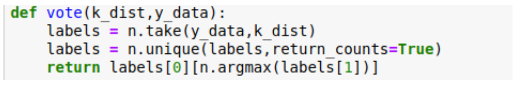
- **Manhattan Distance Function**
  Manhatan Distance $= \sum_{i=1}^{n} |xi - x'i|$



```python
def Manhattan_Distance(test_record,x_data,k):
    indicies = {}

    for row, data in enumerate(x_data):
        dist = 0
        for col, field in enumerate(data):
            if type(field) == "int":
                dist += math.abs(test_record[col]-field)
            elif field != test_record[col]:
                dist += 1
        if len(indicies)< k:
            indicies[row]=dist
        else:
            index = max(indicies.items(), key=operator.itemgetter(1))[0]
            if indicies[index]> dist:
                del indicies[index]
                indicies[row]=dist
    return indicies
```

Figure 1.25: Manhattan Distance Function

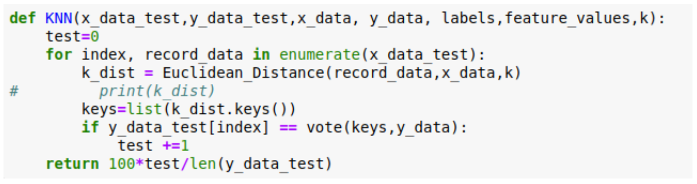
- **Voting Function**
  Voting function get the indices of k nearest neighbours and also get the last column of training (label column), it would take the labels of entered indices and return the most frequent label among them.



```python
def vote(k_dist,y_data):
    labels = n.take(y_data,k_dist)
    labels = n.unique(labels,return_counts=True)
    return labels[0][n.argmax(labels[1])]
```

Figure 1.26: Voting Function

- **KNN Function(Lazy Algorithm)**
  As explained above KNN is lazy method and does not need training. So here the inputs of the KNN functions are test and train data both together. K is the number of the nearest neighbours which is the KNN parameter. labels are the existing classes in dataset. Feature values is dictionary of all features and their possible attribute values.



```python
def KNN(x_data_test,y_data_test,x_data, y_data, labels,feature_values,k):
    test=0
    for index, record_data in enumerate(x_data_test):
        k_dist = Euclidean_Distance(record_data,x_data,k)
#         print(k_dist)
        keys=list(k_dist.keys())
        if y_data_test[index] == vote(keys,y_data):
            test +=1
    return 100*test/len(y_data_test)
```

Figure 1.27: KNN Function

Chapter 2

Experiments and Results

2.1 ID3

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
car	ID3	None	93.0556	0.2597
ecoli	ID3	None	68.9081	0.96221
breast cancer	ID3	None	92.3322	0.3922
letter recognition	ID3	None	39.578	0.1139
mushroom	ID3	None	99.8202	0.0447

2.2 Random Forest

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
car	RF	trees=10 subset size=500 m=3	84.2479	0.9826
car	RF	trees=10 subset size=500 m=2	87.0724	1.093
ecoli	RF	trees=10 subset size=500 m=3	69.7906	1.3387
breast cancer	RF	trees=10 subset size=500 m=3	94.0046	0.6776
letter recognition	RF	trees=10 subset size=500 m=3	12.5895	0.1139
mushroom	RF	trees=10 subset size=500 m=3	96.9731	0.0901

2.3 Adaboost on tree stumps

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
car	Adaboost	trees=10 subset size=500	62.8122	2.1975
car	Adaboost	trees=60 subset size=500	61.4166	1.0988
ecoli	Adaboost	trees=10 subset size=500	59.8735	2.3794
breast cancer	Adaboost	trees=10 subset size=500	72.2614	5.5939
letter recognition	Adaboost	trees=20 subset size=2000	6.2644	0.1516
mushroom	Adaboost	trees=10 subset size=500	71.1722	4.9651

2.4 Naive Bayes

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
car	NB	None	84.0912	0.6560s
ecoli	NB	None	78.1527	0.8347
breast cancer	NB	None	96.1499	0.3523
letter recognition	NB	None	74.5115	0.1427
mushroom	NB	None	44.5679	0.1011

2.5 Naive Bayes with Bagging

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
car	NB bagging	bags=200 subset size=200	82.8786	1.8267
car	NB bagging	bags=20 subset size=200	83.7911	0.5815
ecoli	NB bagging	bags=200 subset size=200	76.4117	2.7014
ecoli	NB bagging	bags=20 subset size=200	75.4705	2.2329
breast cancer	NB bagging	bags=20 size=200	63.072	0.4614
letter recognition	NB bagging	bags=20 subset size=200	63.072	0.4614
mushroom	NB bagging	bags=20 subset size=200	97.0127	0.3421

2.6 KNN with hamming distance

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
car	KNN	k=1	77.0483	0.8387
car	KNN	k=2	77.0483	0.8387
ecoli	KNN	k=3	71.0399	2.2472
breast cancer	KNN	k=3	95.7807	0.4311
letter recognition	KNN	k=10	87.020	0.1519
mushroom	KNN	k=3	100	0
mushroom	KNN	k=3	99.9569	0.0253

2.7 KNN with Euclidean Distance

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
ecoli	KNN	k=1	54.4846	1.6542
ecoli	KNN	k=3	56.3072	1.8308
ecoli	KNN	k=10	54.4899	0.2330
breast cancer	KNN	k=1	95.3369	0.4529
breast cancer	KNN	k=3	95.8794	0.2468
breast cancer	KNN	k=10	94.9787	0.3596

2.8 KNN with Manhattan Distance

Dataset	Algorithm	Setting	Accuracy	Standard Deviation
Letter recognition	KNN	k=3	86.6833	0.0347
ecoli	KNN	k=1	53.8340	2.0723
ecoli	KNN	k=3	55.5043	0.9804
ecoli	KNN	k=10	54.4692	2.0265
breast cancer	KNN	k=1	95.5641	0.4354
breast cancer	KNN	k=3	95.7646	0.3398
breast cancer	KNN	k=10	94.7644	0.2330