Name:Vishal Chauhan
PRN:2020BTECS00090
Ass:6
Sub:AI-ML

Q. For the dataset give below implement Naïve Bayes classifier using python for the classification of records. (perform training and testing)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math


def accuracy_score(y_true, y_pred):
    """ score = (y_true - y_pred) / len(y_true) """

    return round(float(sum(y_pred == y_true))/float(len(y_true)) * 100, 2)


def pre_processing(df):
    """ partioning data into features and target """

    X = df.drop([df.columns[-1]], axis=1)
    y = df[df.columns[-1]]

    return X, y


class NaiveBayes:

    """
            Bayes Theorem:
                                                                                Li
kelihood * Class prior probability
                            Posterior Probability =


      Predictor prior probability


                                                                               P
(x|c) * p(c)
                                                        P(c|x) =
____

        P(x)
    """

    def_init__(self):
        """
                Attributes:
                        likelihoods: Likelihood of each feature per class
                        class_priors: Prior probabilities of classes
```

```
                        pred_priors: Prior probabilities of features
                        features: All features of dataset
        """
        self.features = []
        self.likelihoods = {}
        self.class_priors = {}
        self.pred_priors = {}

        # self.X_train = np.array
        # self.y_train = np.array
        self.train_size = 0
        self.num_feats = 0

    def fit(self, X, y):

        self.features = list(X.columns)
        self.X_train = X
        self.y_train = y
        self.train_size = X.shape[0]
        self.num_feats = X.shape[1]

        for feature in self.features:
            self.likelihoods[feature] = {}
            self.pred_priors[feature] = {}

            for feat_val in np.unique(self.X_train[feature]):
                self.pred_priors[feature].update({feat_val: 0})

                for outcome in np.unique(self.y_train):
                    self.likelihoods[feature].update({feat_val+'_'+outcome:
0})
                    self.class_priors.update({outcome: 0})

        self._calc_class_prior()
        self._calc_likelihoods()
        self._calc_predictor_prior()

        # print(self.likelihoods)
        # print()
        # print(self.pred_priors)
        # print()

    def _calc_class_prior(self):
        """ P(c) - Prior Class Probability """

        for outcome in np.unique(self.y_train):
            outcome_count = sum(self.y_train == outcome)
            self.class_priors[outcome] = outcome_count / self.train_size
```

```
    def _calc_likelihoods(self):
        """ P(x|c) - Likelihood """

        for feature in self.features:

            for outcome in np.unique(self.y_train):
                outcome_count = sum(self.y_train == outcome)
                feat_likelihood =
self.X_train[feature][self.y_train[self.y_train ==
outcome].index.values.tolist(
                )].value_counts().to_dict()

                for feat_val, count in feat_likelihood.items():
                    self.likelihoods[feature][feat_val + '_' + outcome] =
count/outcome_count

    def _calc_predictor_prior(self):
        """ P(x) - Evidence """

        for feature in self.features:
            feat_vals = self.X_train[feature].value_counts().to_dict()

            for feat_val, count in feat_vals.items():
                self.pred_priors[feature][feat_val] = count/self.train_size

    def predict(self, X):
        """ Calculates Posterior probability P(c|x) """

        results = []
        X = np.array(X)

        for query in X:
            probs_outcome = {}
            for outcome in np.unique(self.y_train):
                prior = self.class_priors[outcome]
                likelihood = 1
                evidence = 1

                for feat, feat_val in zip(self.features, query):
                    likelihood *= self.likelihoods[feat][feat_val + '_' +
outcome]
                    evidence *= self.pred_priors[feat][feat_val]

                posterior = (likelihood * prior) / (evidence)

                probs_outcome[outcome] = posterior
```

```
            result = max(probs_outcome, key=lambda x: probs_outcome[x])
            results.append(result)

        return np.array(results)


if __name__ == "__main__":

    df = pd.read_csv("student.csv")
    # shuffle dataset with sample
    df = df.sample(frac=1, random_state=1).reset_index(drop=True)

    # df shape
    print(df.shape)

    # set features and target
    featureDataList, targetDataList = df.iloc[:, 1:-1], df.iloc[:, -1]


    nb = NaiveBayes()
    nb.fit(featureDataList, targetDataList)

    print("Train Accuracy: {}".format(accuracy_score(targetDataList,
nb.predict(featureDataList))))


    # Query 1:
    query = np.array([['>=8', 'no', 'average', 'poor']])
    print("Query 1:- {} ---> {}".format(query, nb.predict(query)))
```

Output:

```
Train Accuracy: 100.0
Query 1:- [['>=8' 'no' 'average' 'poor']] ---> ['no']
```

Q2. For the dataset given below, implement a decision tree classifier using python to classify records. Use entropy and Information gain as an Attribute selection measure (perform training and testing)

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import math
import copy

dataset = pd.read_csv('student.csv')
X = dataset.iloc[:, 1:].values
# print(X)
attribute = ['cgpa','interactive','practical_knowledge','communication_skill']


class Node(object):
    def_init_(self):
        self.value = ""
        self.decision = ""
        self.childs = []


def findEntropy(data, rows):
    yes = 0
    no = 0
    ans = -1
    idx = len(data[0]) - 1
    entropy = 0
    for i in rows:
        if data[i][idx].lower() == 'yes':
            yes += 1
        else:
            no += 1
    x = yes/(yes+no)
    y = no/(yes+no)
    if x != 0 and y != 0:
        entropy = -1 * (x*math.log2(x) + y*math.log2(y))
    if x == 1:
        ans = 1
    if y == 1:
        ans = 0
    return entropy, ans


def findMaxGain(data, rows, columns):
    maxGain = 0
    retidx = -1
    entropy, ans = findEntropy(data, rows)
    if entropy == 0:
        """if ans == 1:
            print("Yes")
        else:
            print("No")"""
        return maxGain, retidx, ans
```

```
    for j in columns:
        mydict = {}
        idx = j
        for i in rows:
            key = data[i][idx]
            if key not in mydict:
                mydict[key] = 1
            else:
                mydict[key] = mydict[key] + 1
        gain = entropy

        # print(mydict)

        for key in mydict:
            yes = 0
            no = 0
            for k in rows:
                if data[k][j] == key:
                    if data[k][-1] == 'Yes':
                        yes = yes + 1
                    else:
                        no = no + 1
            # print(yes, no)
            x = yes/(yes+no)
            y = no/(yes+no)
            # print(x, y)
            if x != 0 and y != 0:
                gain += (mydict[key] * (x*math.log2(x) + y*math.log2(y)))/14
        # print(gain)
        if gain > maxGain:
            # print("hello")
            maxGain = gain
            retidx = j

    return maxGain, retidx, ans


def buildTree(data, rows, columns):

    maxGain, idx, ans = findMaxGain(X, rows, columns)
    root = Node()
    root.childs = []
    # print(maxGain
    #
    # )
    if maxGain == 0:
        if ans == 1:
            root.value = '[YES]'
        else:
            root.value = '[NO]'
        return root

    root.value = "{"+attribute[idx]+"}"
    mydict = {}
    for i in rows:
        key  =  data[i][idx]
        if key not in mydict:
            mydict[key] = 1
        else:
            mydict[key] += 1
```

```
    newcolumns = copy.deepcopy(columns)
    newcolumns.remove(idx)
    for key in mydict:
        newrows = []
        for i in rows:
            if data[i][idx] == key:
                newrows.append(i)
        # print(newrows)
        temp = buildTree(data, newrows, newcolumns)
        temp.decision = key
        root.childs.append(temp)
    return root


def traverse(root):
    print(root.decision)
    print(root.value)

    n = len(root.childs)
    if n > 0:
        for i in range(0, n):
            traverse(root.childs[i])


def calculate():
    rows = [i for i in range(0, len(X))]

    columns = [i for i in range(0, 4)]
    root = buildTree(X, rows, columns)
    root.decision = 'Start'
    traverse(root)


calculate()
```

Output:

Start

{cgpa}

>=9 >=8 <8

{interactive} [YES] [NO]

Yes no

[YES] {practical_knowledge}

Average very_good

[NO] [YES]

3. For the dataset given below, implement a decision tree classifier using python to classify records. Use Gini index as an Attribute selection measure (perform training and testing)

```
import trainDT
import numpy as np


class Node:
    def __init__(self, data_idx, impurity_method, node_level, impurity_value):
        self.data_idx = data_idx
        self.impurity_method = impurity_method
        self.impurity_value = impurity_value
        self.node_level = node_level

        #  defaults
        self.dfeature = -1
        self.nfeatures = []
        self.majority_class = -1
        self.left_child = None
        self.right_child = None

    @staticmethod
    def _init_node(data_idx, impurity_method, level, impurity_value):
        return Node(data_idx, impurity_method, level, impurity_value)

    def get_label_counts(self, label_list):
        labels = [1, 2, 3, 4, 5]
        count_of_labels = []
        for label in labels:
            count_of_labels.append(len(label_list[label_list == label]))
        return count_of_labels

    def build_decision_tree(self, data, indices, impurity_method, nl, p_threshold):
        _impurity_value = -1
        labels = trainDT.train_y
        count_of_labels = self.get_label_counts(labels)
        if self.impurity_method == "gini":
            _impurity_value = self.gini_index(count_of_labels)
        elif self.impurity_method == "entropy":
            _impurity_value = self.entropy(count_of_labels)
        else:
            raise ("Invalid impurity method provided " + str(self.impurity_method))

        print(_impurity_value)

        decision_tree = Node._init_node(indices,
                                        impurity_method=impurity_method,
                                        level=nl,
                                        impurity_value=_impurity_value)
        decision_tree.split_node(max_levels=nl, p_threshold=p_threshold)
        return decision_tree
```

```
    def split_node(self, max_levels, p_threshold):
        if self.node_level > max_levels and self.impurity_value > p_threshold:
            max_gain = -1
            split_feature = -1
            final_left_indices = []
            final_right_indices = []
            final_left_impurity = -1
            final_right_impurity = -1

            for feature in self.nfeatures:
                left_indices = self.get_indices_for_feature(self.data_idx, feature, 0)
                right_indices = self.get_indices_for_feature(self.data_idx, feature, 1)

                p_left = self.calculate_ip(self.data_idx)
                p_right = self.calculate_ip(self.data_idx)

                m = self.get_weighted_impurity(p_left, p_right)
                gain = self.impurity_value - m

                if gain > max_gain:
                    split_feature = feature
                    max_gain = gain
                    final_left_indices = left_indices
                    final_right_indices = right_indices
                    final_left_impurity = p_left
                    final_right_impurity = p_right

            self.dfeature = split_feature
            self.left_child = self._init_node(final_left_indices,
                                              impurity_method=self.impurity_method,
                                              level=self.node_level + 1,
                                              impurity_value=final_left_impurity)

            self.right_child = self._init_node(final_right_indices,
                                               impurity_method=self.impurity_method,
                                               level=self.node_level + 1,
                                               impurity_value=final_right_impurity)

            self.right_child.split_node(max_levels, p_threshold)
            self.left_child.split_node(max_levels, p_threshold)

    def get_weighted_impurity(self, impurity_left, impurity_right, count_left, count_right):
        sum = count_left + count_right
        return impurity_left * (count_left / sum) + impurity_right * (count_right / sum)

    def get_indices_for_feature(self, indices, feature_num, expected_value):
        final_index_set = []
        for index in indices:
            feature_value = trainDT.train_x[index, feature_num]
            if feature_value is expected_value:
                final_index_set.append(index)
        return final_index_set

    def calculate_ip(self, indices):

        final_indexed_labels = trainDT.train_y[indices]
        count_of_each_labels = self.get_label_counts(final_indexed_labels)

        _impurity_value = -1
        if self.impurity_method is "gini":
            _impurity_value = self.gini_index(count_of_each_labels)
        elif self.impurity_method is "entropy":
```

```
            _impurity_value = self.entropy(count_of_each_labels)
        else:
            raise ("Invalid impurity method provided: " + str(self.impurity_method))

        return _impurity_value

    def gini_index(self, counts):
        counts = np.array(counts)
        print(counts)
        print(np.sum(counts))
        probabilities = np.divide(counts, np.sum(counts))
        sum_probabilities = np.sum(np.power(probabilities, 2))
        return 1 - sum_probabilities

    def entropy(self, counts):
        counts = np.array(counts)
        probabilities = np.divide(counts, np.sum(counts))
        entropy = -probabilities * (np.log(probabilities) / np.log(2))
        return entropy
```

trainDT.py

```
import argparse
import numpy as np
import os


class Node:
    """
    Class Node, contains the fields required to build the decision tree
    and also has the functionality and util methods to create a tree
    """

    # Constructor to initialize a node in a tree
    # Takes the following parameters:
    #   data_idx = the data_indices required for splitting the node
    #   impurity_method = To let the tree know what impurity value to be used
    #   node_level = gives the information about the level of the node in the tree
    #   impurity_value = the impurity value at the given node
    #   nfeatrues = number of features to be used for splitting the node
    def __init__(self, data_idx, impurity_method, node_level, impurity_value, nfeatures):
        self.data_idx = data_idx
        self.impurity_method = impurity_method
        self.impurity_value = impurity_value
        self.node_level = node_level
        self.nfeatures = nfeatures

        # default value below
        self.dfeature = -1 # the feature based on which decision will be made
        self.majority_class = -1 # the majority class label at the given node
        self.left_child = None  # reference to the left_child
        self.right_child = None  # reference to the right_child

    # static method to initialize the node, takes the same parameters as required to create a
Node
    @staticmethod
    def _init_node(data_idx, impurity_method, level, impurity_value, nfeatures):
        return Node(data_idx, impurity_method, level, impurity_value, nfeatures)
```

```
    # static method to get the counts of each label in a given labels_list
    @staticmethod
    def get_label_counts(actual_labels_list):
        # getting all the labels from the training set
        labels = set(train_y)

        # list to store the counts of each label
        count_of_labels = []
        for label in labels:
            # getting the count of each label from the list
            count = len(actual_labels_list[actual_labels_list == label])
            if count != 0:
                count_of_labels.append(count)
        return count_of_labels

    # static method to get the weighted impurity
    @staticmethod
    def get_weighted_impurity(impurity_left, impurity_right, count_left, count_right):
        total_sum = count_left + count_right
        return impurity_left * (count_left / total_sum) + impurity_right * (count_right /
total_sum)

    # static method to get the indices based on the feature which equals to expected value
    # We are getting the indices from the training data for the respective index and comparing
with the
    # expected feature value
    @staticmethod
    def get_indices_for_feature(indices, feature_num, expected_value):
        # creating a set to store the results
        final_index_set = []
        for index in indices:
            # getting the feature value from the train data for the particular index and feature
number
            feature_value = train_x[index, feature_num]
            #  if the feature value equals the expected value adding it to the list
            if feature_value == expected_value:
                final_index_set.append(index)
        return final_index_set

    # method to build the decision tree
    # returns the root of the decision tree
    @staticmethod
    def buildDT(_data, _indices, _impurity_method, _nl, _p_threshold):
        _initial_impurity_value = -1

        # calculating the initial gini index at the root
        if _impurity_method == "gini":
            _initial_impurity_value = Node.calculateGINI(_indices)
        elif _impurity_method == "entropy":
            _initial_impurity_value = Node.calculateEntropy(_indices)
        else:
            raise ("Invalid impurity method provided " + str(_impurity_method))

        # creating the root of the decision tree using _init_node method
        _decision_tree = Node._init_node(_indices,
                                         impurity_method=_impurity_method,
                                         level=0,
                                         impurity_value=_initial_impurity_value,
                                         nfeatures=range(_data.shape[1]))

        # splitting the root node until the max level or impurity threshold is reached
```

```
        _decision_tree.split_node(max_levels=_nl, _p_threshold=_p_threshold)
        return _decision_tree

    # calculates the gini index for the given set of indices
    @staticmethod
    def calculateGINI(indices):
        # returning 0 when length of indices is 0, does not matter which value return because
weighted
        # index will make sure that this values is ignored
        if len(indices) == 0:
            return 0

        # getting the labels of the indices provided
        final_indexed_labels = train_y[indices]

        # getting the count of each label for the given indices
        count_of_each_labels = Node.get_label_counts(final_indexed_labels)

        # converting to np array for easier processing
        counts = np.array(count_of_each_labels)

        # getting the probability of each label count
        probabilities = np.divide(counts, np.sum(counts))

        # squaring the probabilities and summing them
        sum_sqaure_probabilities = np.sum(np.power(probabilities, 2))
        return 1 - sum_sqaure_probabilities

    # calculates the entropy for the given set of indices
    @staticmethod
    def calculateEntropy(indices):
        # returning 0 when length of indices is 0, does not matter which value return because
weighted
        # index will make sure that this values is ignored
        if len(indices) == 0:
            return 0
        # getting the labels of the indices provided
        final_indexed_labels = train_y[indices]

        # getting the count of each label for the given indices
        count_of_each_labels = Node.get_label_counts(final_indexed_labels)
        counts = np.array(count_of_each_labels)

        # getting the probability of each label count
        probabilities = np.divide(counts, np.sum(counts))

        # calculating entropy
        entropy = np.sum(-probabilities * (np.log(probabilities) / np.log(2)))
        return entropy

    # utility method to get the majority class label in a given set of indices
    @staticmethod
    def get_maximum_occurring_label(indices):
        labels = set(train_y)
        actual_labels_list = train_y[indices]
        max_occurring_label = -1
        max_count = -1
        for label in labels:
            count = len(actual_labels_list[actual_labels_list == label])
            if count > max_count:
                max_count = count
                max_occurring_label = label
```

```
        return max_occurring_label

    # method to split a given node with maximum depth of tree could be max_levels
    # and with a impurity threshold of _p_threshold
    def split_node(self, max_levels, _p_threshold):
        # assigning the majority class at the given node, helpful in debugging
        # and we will need it for classification
        self.majority_class = Node.get_maximum_occurring_label(self.data_idx)

        # checking if the node is reached the required limits to stop the splitting,
        # it is an early stopping mechanism
        if self.node_level < max_levels and self.impurity_value > _p_threshold:

            # setting the default values which will be calculated in the next steps
            max_gain = -1
            split_feature = -1
            final_left_indices = []
            final_right_indices = []
            final_left_impurity = -1
            final_right_impurity = -1

            # Looping through each feature in the feature set provided
            for feature in self.nfeatures:
                # get the indices for the left child node if the feature value is 0
                left_indices = self.get_indices_for_feature(self.data_idx, feature, 0)
                # get the indices for the right child node if the feature value is 1
                right_indices = self.get_indices_for_feature(self.data_idx, feature, 1)

                # calculate impurity for left child and right child
                p_left = self.calculate_ip(left_indices)
                p_right = self.calculate_ip(right_indices)

                total_sum = len(left_indices) + len(right_indices)
                # calculate the weighted impurity
                m = p_left * (len(left_indices) / total_sum) + p_right * (len(right_indices) /
total_sum)

                # calculate the gain
                gain = self.impurity_value - m

                # if gain is greater than max_gain update the below values
                if gain > max_gain:
                    split_feature = feature
                    max_gain = gain
                    final_left_indices = left_indices
                    final_right_indices = right_indices
                    final_left_impurity = p_left
                    final_right_impurity = p_right

            # once the above is loop is completed assign the feature value
            # through which the node can make a decision during inference.
            # note that the we get this value based on the max_gain
            self.dfeature = split_feature

            # create a node if length of the left child indices is greater than 0, else left
child will be null
            # Also, note that the final_left_indices are calculated based on the
            # feature which provides the max gain and we get this value from the first loop
where
            # we loop through all the features to get the max gain
            if len(final_left_indices) > 0:
```

```
                # initialize the node

                self.left_child = self._init_node(final_left_indices,
                                                  impurity_method=self.impurity_method,
                                                  level=self.node_level + 1,
                                                  impurity_value=final_left_impurity,
                                                  nfeatures=self.nfeatures)
                # split the left child
                self.left_child.split_node(max_levels, p_threshold)

            # create a node if length of the right child indices is greater than 0,
            # else right child will be null
            if len(final_right_indices) > 0:
                self.right_child = self._init_node(final_right_indices,
                                                   impurity_method=self.impurity_method,
                                                   level=self.node_level + 1,
                                                   impurity_value=final_right_impurity,
                                                   nfeatures=self.nfeatures)

                self.right_child.split_node(max_levels, p_threshold)

    # calculating impurities based on gini or entropy, if any other option is provided it will
raise an error
    def calculate_ip(self, indices):
        if self.impurity_method == "gini":
            return self.calculateGINI(indices)
        elif self.impurity_method == "entropy":
            return self.calculateEntropy(indices)
        else:
            raise ("Invalid impurity method provided: " + str(self.impurity_method))

    # classify method to predict the labels of the test data
    # takes two arguments test data and an output file where it appends the output label
    def classify(self, _test_x, _output_file):
        # opens the output file
        _file = open(_output_file, mode="w", encoding="utf-8")

        # looping through all the records in test data
        for record in _test_x:
            # getting the predicted label and writes in the file
            _predicted_label = self.get_predicted_label(record)
            _file.write(str(_predicted_label))
            _file.write("\n")
        _file.close()

    # method to get the prediction of the record, recursively finds out the prediction
    def get_predicted_label(self, _test_x_record):
        # if both the left child and right child are null of the current node
        # then we have reached the leaf node
        # and we will make the decision on the current nodes majority class
        if self.right_child is None and self.left_child is None:
            return self.majority_class
        else:
            # get the feature value based on the feature the node is split
            distinguishing_feature_value = _test_x_record[self.dfeature]
            # go to left child if feature value is 0 else go to right child
            if distinguishing_feature_value == 0:
                return self.left_child.get_predicted_label(_test_x_record)
            else:
                return self.right_child.get_predicted_label(_test_x_record)
```

```
# load the data file and label file using numpy library
def load_data_file_and_label(data_file, label_file):
    data = np.genfromtxt(data_file).astype(int)
    label = np.genfromtxt(label_file).astype(int)
    return data, label


# creating a arg parser for the main method to take the input arguments
def get_parser():
    _parser = argparse.ArgumentParser()

    _parser.add_argument('-train_data')
    _parser.add_argument('-train_label')
    _parser.add_argument('-test_data')
    _parser.add_argument('-test_label')
    _parser.add_argument('-nlevels')
    _parser.add_argument('-pthrd')
    _parser.add_argument('-impurity')
    _parser.add_argument('-pred_file')

    return _parser


# calculates accuracy based on the predicted labels and expected labels
def get_accuracy(expected, predicted):
    count = 0
    for i in range(0, len(expected)):
        if expected[i] == predicted[i]:
            count = count + 1
    return count / len(expected)

# def get_precision_recall(expected, predicted):
#     unique_label_set = np.asarray(list(set(train_y)))
#     precision_recall = []
#     for i in np.arange(0, unique_label_set.size):
#         true_positive = 0
#         false_positive = 0
#         true_negative = 0
#         false_negative = 0
#         for j in np.arange(0, expected.size):
#             # positive true condition
#             if expected[j] == unique_label_set[i]:
#                 # true positive condition
#                 if unique_label_set[i] == predicted[j]:
#                     true_positive += 1
#                 else:
#                     false_negative += 1
#             # negative true condition
#             else:
#                 # false positive condition
#                 if unique_label_set[i] == predicted[j]:
#                     false_positive += 1
#                 else:
#                     true_negative += 1
#
#         precision = true_positive / (true_positive + false_positive)
#         recall = true_positive / (true_positive + false_negative)
#         precision_recall.append([unique_label_set[i], precision, recall])
#
#     return np.asarray(precision_recall)
```

```
def get_precision_recall(expected, predicted):
    true_positive = 0
    false_positive = 0
    true_negative = 0
    false_negative = 0
    for j in np.arange(0, expected.size):
        # positive true condition
        if expected[j] == 1:
            # true positive condition
            if 1 == predicted[j]:
                true_positive += 1
            else:
                false_negative += 1
        # negative true condition
        else:
            # false positive condition
            if 1 == predicted[j]:
                false_positive += 1
            else:
                true_negative += 1

    precision = true_positive / (true_positive + false_positive)
    recall = true_positive / (true_positive + false_negative)

    return precision, recall


# main method where the program starts
if __name__== '__main__':
    parser  =  get_parser()
    args = parser.parse_args()

    # get all the arguments required for the program to run
    train_data_file = os.path.abspath(str(args.train_data))
    train_label_file = os.path.abspath(str(args.train_label))
    test_data_file = os.path.abspath(str(args.test_data))
    test_label_file = os.path.abspath(str(args.test_label))
    max_level_input = int(args.nlevels)
    p_threshold = float(args.pthrd)
    impurity = str(args.impurity)
    pred_output_file = os.path.abspath(str(args.pred_file))

    # get train and test data
    train_x, train_y = load_data_file_and_label(train_data_file, train_label_file)
    test_x, test_y = load_data_file_and_label(test_data_file, test_label_file)

    # build decision tree
    decision_tree = Node.buildDT(train_x,
                                 _indices=list(range(400)),
                                 _impurity_method=impurity,
                                 _nl=max_level_input,
                                 _p_threshold=p_threshold)

    # classify the test data
    decision_tree.classify(test_x, _output_file=pred_output_file)

    # get the predictions from the output file to calculate the accuracy
    predictions = np.genfromtxt(pred_output_file).astype(int)

    _accuracy = get_accuracy(test_y, predictions)
    _precision_recall = get_precision_recall(test_y, predictions)
```

```
print("Accuracy is: " + str(_accuracy))
print("Precision is: " + str(_precision_recall[0]))
print("Recall is: " + str(_precision_recall[1]))
```