MTL 782 Assignment

Submitted by: Mrinal Yadav (2019MT60997) & Rahul Singh (2019MT10285)

Question 1.1

Characterising the DataSet

What the data is about?

The iris dataset contains the following data

50 samples of 3 different species of iris (150 samples total)

Measurements: sepal length, sepal width, petal length, petal width

The format for the data: (sepal length, sepal width, petal length, petal width)

The variables are:

sepal_length: Sepal length, in centimeters, used as input.

sepal_width: Sepal width, in centimeters, used as input.

petal_length: Petal length, in centimeters, used as input.

petal_width: Petal width, in centimeters, used as input.

class: Iris Setosa, Versicolor, or Virginica, used as the target.

What type of benefit you might hope to get from data mining?

1. It is helpful to predict future trends.
2. It signifies customer habits.
3. It helps in decision making.
4. It identifies hidden profitability.
5. It increases company revenue.

Data quality issues:

Unstructured data

Many times, if data has not been entered correctly in the system, or some files may have been corrupted, the remaining data has many missing variables.

Duplicate data

Multiple copies of the same records take a toll on computing and storing, but may also produce skewed or incorrect insights when undetected. One of the critical problems could be human error — someone simply entering data multiple times by accident — or an algorithm that went wrong.

Inaccurate data

Inaccurate data is data in systems filled with human mistakes, like a type or wrong information provided by the customer or inputting details in the wrong field.

For each attribute,

a) Are there problems with the data?

No, there are no problem with the data since there are no missing values or duplicate data present in iris dataset and it is accurate data taken from a reliable source(link given in the assignment).Although there are some outliers present in sepalwidth attribute.

b) What might be an appropriate response to the quality issues.

To overcome data quality issue,we do the following:

For *Unstructured Data*:

Data integration tool help in converting unstructured data to structured data. And also, move data from various formats into one consistent form.

For *Duplicate Data*:

Data deduplication is a combination of human intuition, data analysis, and algorithms to detect possible duplicates based on chance scores and common sense to determine where records look like a near match.

For *Inaccurate Data*:

Automation tools to reduce the amount of manual work when moving data between systems is useful in reducing the risk of mistakes by tired or bored workers.

Importing Libraries

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Importing Iris Data Set

In [2]:
```
url="https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
colnames=['sepallength','sepalwidth','petallength','petalwidth','class']
df = pd.read_csv(url,encoding = 'utf8',names=colnames, header=None)
```

Displaying Data

In [3]:
```
df.head()
```

Out[3]:

	sepallength	sepalwidth	petallength	petalwidth	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [4]:
```
df['class'].unique()
```

Out[4]:
```
array(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica'], dtype=object)
```

In [5]:
```
df.describe(include='all')
```

Out[5]:

	sepallength	sepalwidth	petallength	petalwidth	class
count	150.000000	150.000000	150.000000	150.000000	150
unique	NaN	NaN	NaN	NaN	3
top	NaN	NaN	NaN	NaN	Iris-versicolor
freq	NaN	NaN	NaN	NaN	50
mean	5.843333	3.054000	3.758667	1.198667	NaN
std	0.828066	0.433594	1.764420	0.763161	NaN
min	4.300000	2.000000	1.000000	0.100000	NaN
25%	5.100000	2.800000	1.600000	0.300000	NaN
50%	5.800000	3.000000	4.350000	1.300000	NaN
75%	6.400000	3.300000	5.100000	1.800000	NaN
max	7.900000	4.400000	6.900000	2.500000	NaN

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   sepallength  150 non-null    float64
 1   sepalwidth   150 non-null    float64
 2   petallength  150 non-null    float64
 3   petalwidth   150 non-null    float64
 4   class        150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 5.3+ KB
```

Checking if there are any missing values

In [7]:
```
df.isnull().sum()
```

Out[7]:
```
sepallength    0
sepalwidth     0
petallength    0
petalwidth     0
class          0
dtype: int64
```

There are no missing value present in this dataset

Checking for Outliers

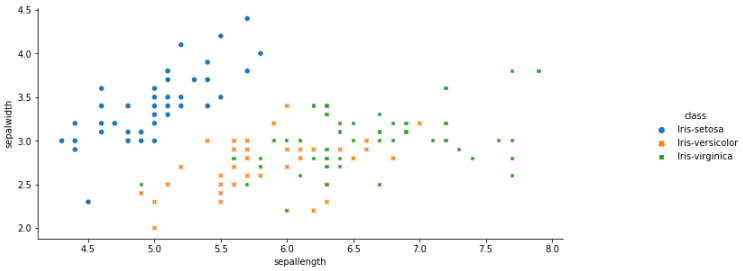
In [8]:
```
def count_outliers(info,column):
        A1 = info[column].quantile(0.25,interpolation='nearest')
        A2 = info[column].quantile(0.5,interpolation='nearest')
        A3 = info[column].quantile(0.75,interpolation='nearest')
        A4 = info[column].quantile(1,interpolation='nearest')
        z = A3 -A1
        global m,n,d
        d=[]
        m = A1-1.5*z
        n = A3+1.5*z
        if info[column].min()>m and info[column].max()<n:
            print("There are no outliers in "+i+" attribute")
        else:
            print("Outliers are present in "+i+" attribute")

            p = info[info[column] < m][column].size

            q = info[info[column] > n][column].size

            d.append(i)

            print("Number of outliers are:",p+q)
for i in df.columns[:-1]:
    count_outliers(df,i)
```

```
There are no outliers in sepallength attribute
Outliers are present in sepalwidth attribute
Number of outliers are: 4
There are no outliers in petallength attribute
There are no outliers in petalwidth attribute
```

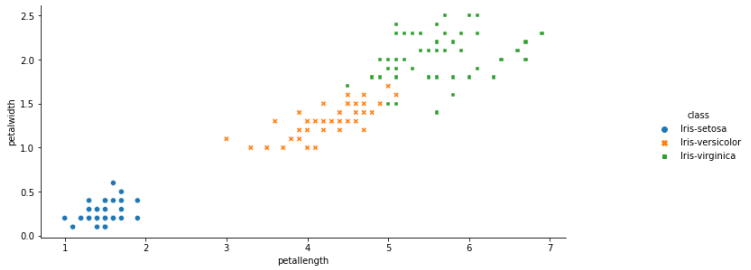
Data Visualisation

ScatterPlot

In [9]:
```
graph=sns.relplot(x='sepallength',y='sepalwidth',data=df,hue='class',style='class')
graph.fig.set_size_inches(12,4)
plt.show()
```



In [10]:
```
graph=sns.relplot(x='petallength',y='petalwidth',data=df,hue='class',style='class')
graph.fig.set_size_inches(12,4)
plt.show()
```



Observation:

We can see that the Petal Features are giving a better cluster division compared to the Sepal features. This is an indication that the Petals can help in better and accurate Predictions over the Sepal.

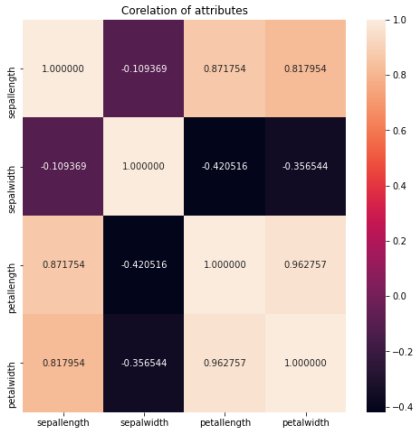
Correlation

In [11]:
```
df.corr()
```

Out[11]:

	sepallength	sepalwidth	petallength	petalwidth
sepallength	1.000000	-0.109369	0.871754	0.817954
sepalwidth	-0.109369	1.000000	-0.420516	-0.356544
petallength	0.871754	-0.420516	1.000000	0.962757
petalwidth	0.817954	-0.356544	0.962757	1.000000

In [12]:
```
plt.subplots(figsize = (8,8))
sns.heatmap(df.corr(),annot=True,fmt="f").set_title("Corelation of attributes")
plt.show()
```



Observation:

The Sepal Width and Length are not correlated. The Petal Width and Length are highly correlated.

In [ ]:

Question 1.2

Implementing Decision Tree, Random Forest, KNN,Naive Bayes Classifier for multi-class Dataset(Iris Dataset)

Dividing data into features and labels

In [13]:
```
y=df.iloc[:,4].values
X=df.iloc[:,0:4].values
```

Label Encoding

Class label attribute is categorical. KNeighborsClassifier does not accept string labels. We need to use LabelEncoder to transform them into numbers.

Iris-setosa correspond to 0, Iris-versicolor correspond to 1 and Iris-virginica correspond to 2.

In [14]:
```
from sklearn.preprocessing import LabelEncoder
LE = LabelEncoder()
y = LE.fit_transform(y)
```

Building Machine Learning Models

In [15]:
```
#Evaluation Metrics
from sklearn.metrics import accuracy_score ,precision_score,recall_score,f1_score,confusion_matrix,classification_report

#Model Select
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.model_selection import KFold,train_test_split,cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn import preprocessing,metrics
from sklearn.pipeline import make_pipeline
```

Splitting The Data into Training And Testing Dataset

In [16]:
```
#Train and Test split
X_train,X_test,y_train,y_test=train_test_split(X,y,random_state=0,test_size=0.2)
```

Implementing Decision Tree

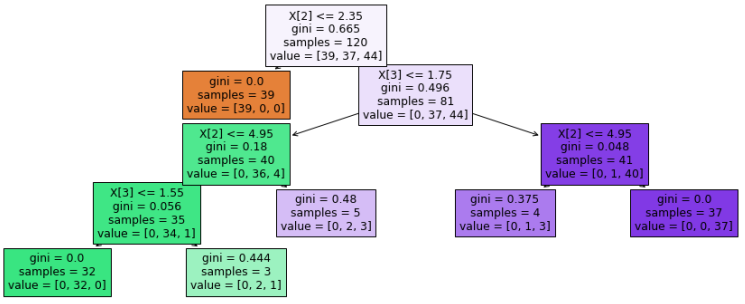
Hyperparameter Tuning(criterion ,min_sample_split_range, max_depth_range ,in_samples_leaf_range,min_leaf_nodes_range) in Decision Tree

In [17]:
```
criterion = ["gini", "entropy"]
min_samples_leaf_range = [1,2,3]
max_leaf_nodes_range = [None, 5, 10, 20]
min_sample_split_range = [2,5,10]
max_depth_range = [None, 2, 5]
par_grid = {"criterion": criterion,
            "min_samples_split": min_sample_split_range,
            "max_depth": max_depth_range,
            "min_samples_leaf": min_samples_leaf_range,
            "max_leaf_nodes": max_leaf_nodes_range
             }
dt=DecisionTreeClassifier()
grid = GridSearchCV(estimator=dt,param_grid=par_grid,cv = 5,scoring='accuracy',refit=True)

t_model = make_pipeline(preprocessing.StandardScaler(), grid)
t_model.fit(X_train, y_train)

print("Tuned model Accuracy: ", grid.best_score_)
print(grid.best_params_)
```

```
Tuned model Accuracy:  0.95
{'criterion': 'gini', 'max_depth': None, 'max_leaf_nodes': None, 'min_samples_leaf': 2, 'min_samples_split': 2}
```

In [18]:
```
model_dt = DecisionTreeClassifier(criterion= 'gini', max_depth=None, max_leaf_nodes= 20, min_samples_leaf= 3, min_samples_split= 5)
model_dt.fit(X_train, y_train)
Y_pred = model_dt.predict(X_test)
dt_acc=round(accuracy_score(y_test,Y_pred)* 100, 2)
dt_acc_score = round(model_dt.score(X_train, y_train) * 100, 2)
```

In [19]:
```
from sklearn.tree import plot_tree
plt.figure(figsize = (16,6))
plot_tree(model_dt.fit(X_train, y_train)  ,filled=True)
plt.show()
```



Comparing the performances of Decision Tree Classifier using k-fold cross validation and other tuning techniques.

In [20]:
```
kfolds = KFold(n_splits=10, random_state=7,shuffle=True)
score = cross_val_score(model_dt, X, y, cv=kfolds)
print('Kfolds_Decision Tree : ',score.mean())
confusion = confusion_matrix(y_test, Y_pred)
sns.heatmap(confusion,annot=True)
print('for Decision Tree, Confusion matrix\n',confusion)
classify=classification_report(y_test, Y_pred)
print('for Decision Tree, Classification Report\n',classify)
print("Heatmap:")
```

```
Kfolds_Decision Tree :  0.9600000000000002
for Decision Tree, Confusion matrix
 [[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
for Decision Tree, Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         6

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Heatmap:
```

Implementing Random Forest

Hyperparameter Tuning(n_estimators,criterion,min_sample_split_range,min_samples_leaf_range,max_leaf_nodes_range) in Random Forest

In [21]:
```
criteria = ["gini", "entropy"]
min_sample_split_range = [2,3]
min_samples_leaf_range = [1,2,3]
n_estimators = [10]
max_leaf_nodes_range = [None, 5,10]
param_grid = {
              "n_estimators":n_estimators,
              "criterion": criteria,
              "min_samples_split": min_sample_split_range,
              "min_samples_leaf": min_samples_leaf_range,
              "max_leaf_nodes": max_leaf_nodes_range
                  }
rf = RandomForestClassifier(n_estimators=n_estimators,criterion= criteria, max_leaf_nodes= max_leaf_nodes_range, min_samples_leaf= min_samples_leaf_range, min_samples_split= min_sample_split_r
grid = GridSearchCV(estimator=rf,
                    param_grid=param_grid,
                    cv = 5,
                    scoring='accuracy',
                    refit=True)
t_model = make_pipeline(preprocessing.StandardScaler(), grid)
t_model.fit(X_train, y_train)

print("Accuracy of the tuned model: ", grid.best_score_)
print(grid.best_params_)
```

```
Accuracy of the tuned model:  0.9583333333333334
{'criterion': 'gini', 'max_leaf_nodes': 5, 'min_samples_leaf': 2, 'min_samples_split': 3, 'n_estimators': 10}
```

In [22]:
```
rf = RandomForestClassifier(n_estimators=10,criterion= 'gini', max_depth=None, max_leaf_nodes= None, min_samples_leaf= 1, min_samples_split= 2)
rf.fit(X_train, y_train)
Y_prediction = rf.predict(X_test)
rf_acc=round(accuracy_score(y_test,Y_prediction)* 100, 2)
random_forest_acc_score = round(rf.score(X_train, y_train) * 100, 2)
```

Comparing the performances of Random Forest Classifier using k-fold cross validation and other tuning techniques.

In [23]:
```
kfolds = KFold(n_splits=10, random_state=7,shuffle=True)
score = cross_val_score(rf, X, y, cv=kfolds)
print('Kfolds_Random Forest : ',score.mean())
confusion = confusion_matrix(y_test, Y_pred)
sns.heatmap(confusion,annot=True)
print('for Random Forest, Confusion matrix\n',confusion)
classify=classification_report(y_test, Y_pred)
print('for Random Forest, Classification Report\n',classify)
print("Heatmap")
```

```
Kfolds_Random Forest :  0.9533333333333334
for Random Forest, Confusion matrix
 [[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
for Random Forest, Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         6

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Heatmap
```

Implementing Gaussian Naive Bayes Classifier

Hyperparameter Tuning (var_smoothing) in Naive Bayes Classifier

In [24]:
```
var_smoothing=[1e-9,1e-8,1e-7]
par_grid = {"var_smoothing": var_smoothing}
nb=GaussianNB()
grid = GridSearchCV(estimator=nb,param_grid=par_grid,cv = 5,scoring='accuracy',refit=True)

t_model = make_pipeline(preprocessing.StandardScaler(), grid)
t_model.fit(X_train, y_train)

print("Tuned model Accuracy: ", grid.best_score_)
print(grid.best_params_)
```

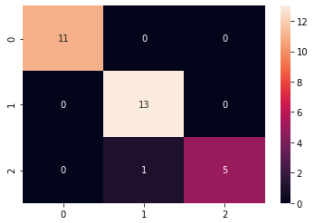
```
Tuned model Accuracy:  0.95
{'var_smoothing': 1e-09}
```

In [25]:
```
nb = GaussianNB(var_smoothing=1e-9)
nb.fit(X_train, y_train)
Y_pred = nb.predict(X_test)
nb_acc=round(accuracy_score(y_test,Y_pred)* 100, 2)
gaussian_acc_score = round(nb.score(X_train, y_train) * 100, 2)
```

Comparing the performances of Gaussian Naive Bayes Classifier using k-fold cross validation and other tuning techniques.

In [26]:
```
kfolds = KFold(n_splits=10, random_state=7,shuffle=True)
score = cross_val_score(nb, X, y, cv=kfolds)
print('Kfolds_Naive Bayes : ',score.mean())
confusion = confusion_matrix(y_test, Y_pred)
print('for Naive Bayes, Confusion matrix\n',confusion)
sns.heatmap(confusion,annot=True)
classify=classification_report(y_test, Y_pred)
print('for Naive Bayes, Classification Report\n',classify)
print("Heatmap:")
```

```
Kfolds_Naive Bayes :  0.9533333333333334
for Naive Bayes, Confusion matrix
 [[11  0  0]
 [ 0 13  0]
 [ 0  1  5]]
for Naive Bayes, Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       0.93      1.00      0.96        13
           2       1.00      0.83      0.91         6

    accuracy                           0.97        30
   macro avg       0.98      0.94      0.96        30
weighted avg       0.97      0.97      0.97        30

Heatmap:
```

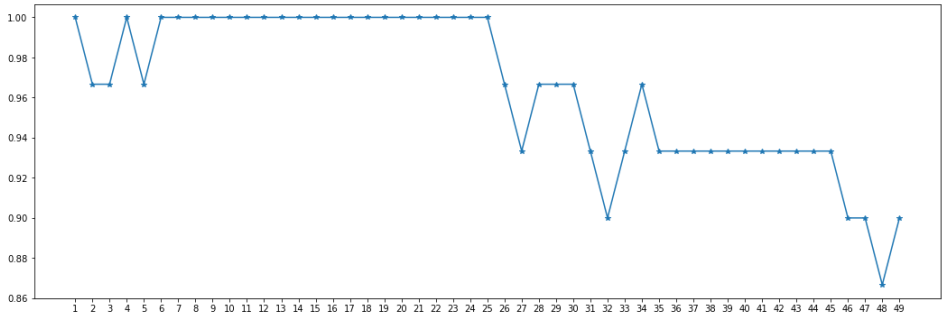


Implementing K-Nearest Neighbor Classifier

Hyperparamter Tunning (value of nearest neighbors in KNN)

Checking the accuracy for various values of n for K-Nearest Neighbors

In [27]:
```
plt.subplots(figsize=(18,6))
n_index=list(range(1,50))
n = pd.Series([],dtype=pd.StringDtype())
x=range(1,50)
for i in list(range(1,50)):
    knn_model=KNeighborsClassifier(n_neighbors=i)
    knn_model.fit(X_train, y_train)
    pred=knn_model.predict(X_test)
    n=n.append(pd.Series(accuracy_score(y_test,pred)))
plt.plot(n_index, n,marker="*")
plt.xticks(x)
plt.show()
```



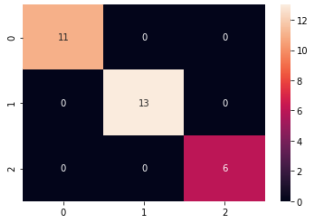
Above is the graph showing the accuracy for the KNN models using different values of n(n-nearest neighbors). We see n=12(not too big and not too small) is good.

In [28]:
```
kNeighbors = KNeighborsClassifier(n_neighbors = 12)
kNeighbors.fit(X_train, y_train)
Y_pred = kNeighbors.predict(X_test)
knn_acc=round(accuracy_score(y_test,Y_pred)* 100, 2)
knn_acc_score = round(kNeighbors.score(X_train, y_train) * 100, 2)
```

Comparing the performances of KNN Classifier using k-fold cross validation and other tuning techniques.

In [29]:
```
kfolds = KFold(n_splits=10, random_state=7,shuffle=True)
score = cross_val_score(kNeighbors, X, y, cv=kfolds)
print('Kfolds_KNN : ',score.mean())
confusion = confusion_matrix(y_test, Y_pred)
sns.heatmap(confusion,annot=True)
print('for KNN, Confusion Matrix:\n',confusion)
classify=classification_report(y_test, Y_pred)
print('for KNN, Classification Report\n',classify)
print("Heatmap:")
```

```
Kfolds_KNN :  0.96
for KNN, Confusion Matrix:
 [[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
for KNN, Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00         6

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Heatmap:
```



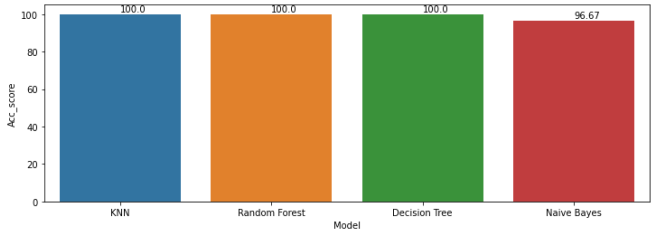
Which is the best Model?

In [30]:
```
best = pd.DataFrame({
    'Model': [ 'KNN','Random Forest','Naive Bayes','Decision Tree'],
    'Score': [ knn_acc_score,random_forest_acc_score,gaussian_acc_score,dt_acc_score],
    "Acc_score":[knn_acc,rf_acc,nb_acc,dt_acc]
                    })

best_df = best.sort_values(by='Acc_score', ascending=False)

best_df = best_df.reset_index(drop=True)

best_df.head(4)
```

Out[30]:

	Model	Score	Acc_score
0	KNN	97.50	100.00
1	Random Forest	100.00	100.00
2	Decision Tree	96.67	100.00
3	Naive Bayes	95.00	96.67

In [31]:
```
plt.subplots(figsize=(12,4))
ax=sns.barplot(x='Model',y="Acc_score",data=best_df)
labels = (best_df["Acc_score"])
for i,j in enumerate(labels):
    ax.text(i,j+1,str(j))
```



In [32]:
```
#Question-2-(a)

#Implementing Apriori Algorithm
from collections import defaultdict
from csv import reader
from itertools import chain,combinations
import pandas as pd
import requests

#Function to generate the candidate set
def generate_candidate(records):
    cand=set()
    for record in records:
        for value in record:
                cand.add(frozenset([value]))
    return cand

#Function to get the frequent set of size (K+1) from the candidate set of size K
def get_frequent(Set,final_set,min_sup,records):
    f_set=set()
    temp=defaultdict(int)
    for value in Set:
        for Set in records:
            if(value.issubset(Set)):
                final_set[value]=final_set[value]+1
                temp[value]=temp[value]+1
    for value, count in temp.items():
        sup=float(count/len(records))
        if(sup>=min_sup):
            f_set.add(value)
    return f_set

#Function to generate the candidate set of size K+1 from the set of size K
def generate_set(Set,k):
    return set([i.union(j) for i in Set for j in Set if len(i.union(j))==k])

#Function implementing the pruning process of a set
def prune(temp,Set,p):
    temp_set=Set.copy()
    for element in Set:
        subclasses=combinations(element,p)
        for subclass in subclasses:
            if(frozenset(subclass) not in temp):
                temp_set.remove(element)
                break
    return temp_set

#Function to derive the association rules based on the input data and the apriori algorithm
def association_rule(final_set,itemset,min_conf):
    rules=[]
    for count, Set in itemset.items():
        for item in Set:
            subclass=chain.from_iterable(combinations(item,x) for x in range(1,len(item)))
            for value in subclass:
                conf=float(final_set[item]/final_set[frozenset(value)])
                if(conf>= min_conf):
                    rules.append([set(value),set(item.difference(value)),conf])
    return rules

#Function defining the working of apriori algorithm
def Apriori(records,min_support,min_confidence):
    itemset=dict()
    final_set=defaultdict(int)
    C1set=generate_candidate(records)
    L1set=get_frequent(C1set,final_set,min_support,records)
    x=2
    temp_set=L1set
    while(temp_set):
        itemset[x-1]=temp_set
        cand_set=generate_set(temp_set,x)
        cand_set=prune(temp_set,cand_set,x-1)
        temp_set=get_frequent(cand_set,final_set,min_support,records)
        x+=1
    rules=association_rule(final_set,itemset,min_confidence)
    rules.sort(key=lambda x:x[2])
#The apriori algorithm retunrns two variables that store the association rules and desired itemset
    return itemset, rules


#ItemSetList to test the working of the implementation of the Apriori Algorithm
itemSetList = [['eggs', 'bacon', 'soup'],
               ['eggs', 'bacon', 'apple'],
               ['soup', 'bacon', 'banana']]
#The min. confidence and min. support are input as values in the function which are 0.5 each
freq_set,rules = Apriori(itemSetList, 0.5, 0.5)
rule=list(rules)
print("The output (desired itemset) of the Apriori algorithm based on the itemset list is",freq_set)
print("The output (association rules) of the Apriori algorithm based on the itemset List is",rule)


#Implementing FP growth algorithm

#Defining the class Node in order to implement the FP tree
class Node:
    def __init__(self,name,frequency,parent):
        self.itemName=name
        self.count=frequency
        self.parent=parent
        self.children={}
        self.next=None

    def display(self, index=1):
        print(' '*index,self.itemName,' ',self.count)
        for child in list(self.children.values()):
            child.display(index+1)

    def increment(self,frequency):
        self.count=self.count+frequency

#Function to generate the header table based on the input provided
def generate_table(records,min_sup,frequency):
    h_table = defaultdict(int)
    for count,Set in enumerate(records):
        for item in Set:
            h_table[item]+=frequency[count]
    h_table=dict((item, support) for item, support in h_table.items() if support>=min_sup)
    if(len(h_table)==0):
        return None
    for item in h_table:
        h_table[item]= [h_table[item],None]
    return h_table

#Function to update the header table in case of new node is inserted in the FP tree
def update_table(element,new_Node,header_table):
    if(header_table[element][1]==None):
        header_table[element][1]=new_Node
    else:
        temp=header_table[element][1]
        while temp.next!=None:
            temp=temp.next
        temp.next=new_Node

#Function to create the FP tree in accordance with the data provided
def Tree(header_table,records,min_sup,frequency):
    fptree=Node("Null",1,None)
```

```
    for count,Set in enumerate(records):
        Set=[element for element in Set if element in header_table]
        Set.sort(key=lambda element: header_table[element][0], reverse=True)
        temp=fptree
        for element in Set:
            sub_tree=temp
            if element in sub_tree.children:
                sub_tree.children[element].increment(frequency[count])
            else:
                new_Node=Node(element,frequency[count],sub_tree)
                sub_tree.children[element]=new_Node
                update_table(element,new_Node,header_table)
            temp=sub_tree.children[element]

    return fptree

#Function used in Prefix Path in order to find the path from the current node to the parent node
def ascend(temp_node,path):
    if temp_node.parent!=None:
        path.append(temp_node.itemName)
        ascend(temp_node.parent, path)

#Prefix Path is used to find the conditional patterns and frequency
def prefix_path(element,header_table):
    temp_node=header_table[element][1]
    freq=[]
    patterns=[]
    while temp_node!=None:
        path=[]
        ascend(temp_node, path)
        if len(path)>1:
            freq.append(temp_node.count)
            patterns.append(path[1:])
        temp_node=temp_node.next
    return patterns, freq

#Function that implements the Mining algorithm as specified in the FP growth algorithm
def mine(headertable,mined_set,min_sup,frequent_set):
    element_list=[item[0] for item in sorted(list(headertable.items()), key=lambda p:p[1][0])]
    for element in element_list:
        temp=mined_set.copy()
        temp.add(element)
        frequent_set.append(temp)
        pattern_base, frequency=prefix_path(element,headertable)
        head_table=generate_table(pattern_base,min_sup,frequency)
        if(head_table!=None):
            tree=Tree(head_table,pattern_base,min_sup,frequency)
            mine(head_table,temp,min_sup,frequent_set)
        else:
            tree=None

#Function to derive the association rules for the input data
def association_rule(frequent_set,records,min_conf):
    rules=[]
    for element in frequent_set:
        Set=chain.from_iterable(combinations(element,x) for x in range(1, len(element)))
        sup=0
        for record in records:
            if(set(element).issubset(record)):
                sup+=1
        for space in Set:
            temp=0
            for record in records:
                if(set(space).issubset(record)):
                    temp+=1
            confidence=float(sup/temp)
            if(confidence>min_conf):
                rules.append([set(space), set(element.difference(space)),confidence])
    return rules

#Function that defines the working of the FP growth algorithm
def fp_growth(records,min_sup_ratio,min_conf):
    frequency=[1 for i in range(len(records))]
    min_sup=len(records)*min_sup_ratio
    header_table=generate_table(records,min_sup,frequency)
    fp_tree=Tree(header_table,records,min_sup,frequency)
    if(fp_tree==None):
        print("There is no frequent set present")
    else:
        frequent_set=[]
        mine(header_table,set(),min_sup,frequent_set)
        rules=association_rule(frequent_set,records,min_conf)
        return frequent_set, rules


#ItemSetList to test the working of the implementation of the FP growth Algorithm
itemSetList = [['eggs', 'bacon', 'soup'],
               ['eggs', 'bacon', 'apple'],
               ['soup', 'bacon', 'banana']]
#The min. confidence and min. support are input as values in the function which are 0.5 each
freqItemSet, fp_rules= fp_growth(itemSetList, 0.5, 0.5)
fp_rule=list(fp_rules)
print("The output (desired itemset) of the FP growth algorithm based on the itemset list is",freqItemSet)
print("The output (association rules) of the FP growth algorithm based on the itemset List is",fp_rule)
```

```
#Question 2-(b)

#Modifications in the Apriori Algorithm

#The complexity while finding the frequent data set and the candidate set for the input data can be reduced futher based on the
#argument that if a certain element of the set used for finding set of (K+1) elements doesn't occur k times in the set then it
#cannot occur in the frequent set and so the entry is removed from the candidate set

#The function get_frequent in the Apriori algorithm is modified by adding the condition as follows
#    for value in Set:
#        for Set in records:
#            if(temp[value]<k):
#                temp.pop(value)

#Implementing Modified Apriori Algorithm

#Function to generate the candidate set
def mod_generate_candidate(records):
    cand=set()
    for record in records:
        for value in record:
                cand.add(frozenset([value]))
    return cand

#Function to get the frequent set of size (K+1) from the candidate set of size K
def mod_get_frequent(Set,final_set,min_sup,records):
    f_set=set()
    temp=defaultdict(int)
    for value in Set:
        for Set in records:
            if(value.issubset(Set)):
                final_set[value]=final_set[value]+1
                temp[value]=temp[value]+1
    for value, count in temp.items():
        sup=float(count/len(records))
        if(sup>=min_sup):
            f_set.add(value)
    return f_set

#Function to generate the candidate set of size K+1 from the set of size K
def mod_generate_set(Set,k):
    return set([i.union(j) for i in Set for j in Set if len(i.union(j))==k])

#Function implementing the pruning process of a set
def mod_prune(temp,Set,p):
    temp_set=Set.copy()
    for element in Set:
        subclasses=combinations(element,p)
        for subclass in subclasses:
            if(frozenset(subclass) not in temp):
                temp_set.remove(element)
                break
    return temp_set

#Function to derive the association rules based on the input data and the apriori algorithm
def mod_association_rule(final_set,itemset,min_conf):
    rules=[]
    for count, Set in itemset.items():
        for item in Set:
            subclass=chain.from_iterable(combinations(item,x) for x in range(1,len(item)))
            for value in subclass:
                conf=float(final_set[item]/final_set[frozenset(value)])
                if(conf>= min_conf):
                    rules.append([set(value),set(item.difference(value)),conf])
    return rules

#Function defining the working of apriori algorithm
def mod_Apriori(records,min_support,min_confidence):
    itemset=dict()
    final_set=defaultdict(int)
    C1set=mod_generate_candidate(records)
    L1set=mod_get_frequent(C1set,final_set,min_support,records)
    x=2
    temp_set=L1set
    while(temp_set):
        itemset[x-1]=temp_set
        cand_set=mod_generate_set(temp_set,x)
        cand_set=mod_prune(temp_set,cand_set,x-1)
        temp_set=mod_get_frequent(cand_set,final_set,min_support,records)
        x+=1
    rules=mod_association_rule(final_set,itemset,min_confidence)
    rules.sort(key=lambda x:x[2])
    return itemset, rules



#Modifications in the FP growth Algorithm

#In the implementation of the FP growth algorithm, instead of creating a tree, we can use the trie data structure which as a
# result gives better performance at mining and also in finding the association rules evidently.

#The trie data structure can be defined as follows as its functions can be called in the implementation


#Implementing Modified FP growth algorithm

#Implementing the Trie Data structure
#Defining TrieNode
class TrieNode:

    def __init__(self,name,frequency,parent):
        self.children = {}
        self.count=frequency
        self.parent=parent
        self.itemName=name
        self.next=None

        # isEndOfWord is True if node represent the end of the word
        self.isEndOfWord = False

    def increment(self,frequency):
        self.count=self.count+frequency

class Trie:
    def __init__(self):
        self.root = self.getNode()

    def getNode(self):
        return TrieNode()

    def _charToIndex(self,ch):
        return ord(ch)-ord('a')

    def insert(self,key):
        temp = self.root
        length = len(key)
        for indices in range(length):
            index = self._charToIndex(key[indices])
            if not temp.children[index]:
                temp.children[index] = self.getNode()
            temp = temp.children[index]
        temp.isEndOfWord = True

    def search(self, key):
        temp = self.root
```

```
        length = len(key)
        for indices in range(length):
            index = self._charToIndex(key[indices])
            if not temp.children[index]:
                return False
            temp = temp.children[index]

        return temp.isEndOfWord

#Function to generate the header table based on the input provided
def mod_generate_table(records,min_sup,frequency):
    h_table = defaultdict(int)
    for count,Set in enumerate(records):
        for item in Set:
            h_table[item]+=frequency[count]
    h_table=dict((item, support) for item, support in h_table.items() if support>=min_sup)
    if(len(h_table)==0):
        return None
    for item in h_table:
        h_table[item]= [h_table[item],None]
    return h_table

#Function to update the header table in case of new node is inserted in the FP tree
def mod_update_table(element,new_Node,header_table):
    if(header_table[element][1]==None):
        header_table[element][1]=new_Node
    else:
        temp=header_table[element][1]
        while temp.next!=None:
            temp=temp.next
        temp.next=new_Node

#Function to create the FP tree in accordance with the data provided
def mod_Tree(header_table,records,min_sup,frequency):
    fptree=TrieNode("Null",1,None)
    for count,Set in enumerate(records):
        Set=[element for element in Set if element in header_table]
        Set.sort(key=lambda element: header_table[element][0], reverse=True)
        temp=fptree
        for element in Set:
            sub_tree=temp
            if element in sub_tree.children:
                sub_tree.children[element].increment(frequency[count])
            else:
                new_Node=TrieNode(element,frequency[count],sub_tree)
                sub_tree.children[element]=new_Node
                mod_update_table(element,new_Node,header_table)
            temp=sub_tree.children[element]

    return fptree

#Function used in Prefix Path in order to find the path from the current node to the parent node
def mod_ascend(temp_node,path):
    if temp_node.parent!=None:
        path.append(temp_node.itemName)
        mod_ascend(temp_node.parent, path)

#Prefix Path is used to find the conditional patterns and frequency
def mod_prefix_path(element,header_table):
    temp_node=header_table[element][1]
    freq=[]
    patterns=[]
    while temp_node!=None:
        path=[]
        mod_ascend(temp_node, path)
        if len(path)>1:
            freq.append(temp_node.count)
            patterns.append(path[1:])
        temp_node=temp_node.next
    return patterns, freq

#Function that implements the Mining algorithm as specified in the FP growth algorithm
def mod_mine(headertable,mined_set,min_sup,frequent_set):
    element_list=[item[0] for item in sorted(list(headertable.items()), key=lambda p:p[1][0])]
    for element in element_list:
        temp=mined_set.copy()
        temp.add(element)
        frequent_set.append(temp)
        pattern_base, frequency=mod_prefix_path(element,headertable)
        head_table=mod_generate_table(pattern_base,min_sup,frequency)
        if(head_table!=None):
            tree=mod_Tree(head_table,pattern_base,min_sup,frequency)
            mod_mine(head_table,temp,min_sup,frequent_set)
        else:
            tree=None

#Function to derive the association rules for the input data
def mod_association_rule(frequent_set,records,min_conf):
    rules=[]
    for element in frequent_set:
        Set=chain.from_iterable(combinations(element,x) for x in range(1, len(element)))
        sup=0
        for record in records:
            if(set(element).issubset(record)):
                sup+=1
        for space in Set:
            temp=0
            for record in records:
                if(set(space).issubset(record)):
                    temp+=1
            confidence=float(sup/temp)
            if(confidence>min_conf):
                rules.append([set(space), set(element.difference(space)),confidence])
    return rules

#Function that defines the working of the FP growth algorithm
def mod_fp_growth(records,min_sup_ratio,min_conf):
    frequency=[1 for i in range(len(records))]
    min_sup=len(records)*min_sup_ratio
    header_table=mod_generate_table(records,min_sup,frequency)
    fp_tree=mod_Tree(header_table,records,min_sup,frequency)
    if(fp_tree==None):
        print("There is no frequent set present")
    else:
        frequent_set=[]
        mod_mine(header_table,set(),min_sup,frequent_set)
        rules=mod_association_rule(frequent_set,records,min_conf)
        return frequent_set, rules
```

```
The output (desired itemset) of the Apriori algorithm based on the itemset list is {1: {frozenset({'bacon'}), frozenset({'soup'}), frozenset({'eggs'})}, 2: {frozenset({'bacon', 'soup'}), fro
zenset({'bacon', 'eggs'})}}
The output (association rules) of the Apriori algorithm based on the itemset List is [[{'bacon'}, {'soup'}, 0.6666666666666666], [{'bacon'}, {'eggs'}, 0.6666666666666666], [{'soup'}, {'baco
n'}, 1.0], [{'eggs'}, {'bacon'}, 1.0]]
The output (desired itemset) of the FP growth algorithm based on the itemset list is [{'eggs'}, {'bacon', 'eggs'}, {'soup'}, {'bacon', 'soup'}, {'bacon'}]
The output (association rules) of the FP growth algorithm based on the itemset List is [[{'bacon'}, {'eggs'}, 0.6666666666666666], [{'eggs'}, {'bacon'}, 1.0], [{'bacon'}, {'soup'}, 0.6666666
666666666], [{'soup'}, {'bacon'}, 1.0]]
```

In [ ]:
```
# Testing of Apriori and FP growth algorithms done on the basis of the data provided in the assignment is as follows
response = requests.get('http://fimi.ua.ac.be/data/retail.dat')
text_data = response.text
store_data = []
for line in text_data.split("\n"):
   word = []
   for number in line.split(" "):
       if(number!=''):
           word.append(int(number))
   store_data.append(word)
num_records = len(store_data)
records = []
for i in range(0,num_records):
   records.append([str(store_data[i][j]) for j in range(0,len(store_data[i]))])
freq_set1,rules1 = Apriori(records, 0.5, 0.5)
freq_set2,rules2 = fp_growth(records, 0.5, 0.5)
rule2=list(rules2)
rule1=list(rules1)
print("The output (desired itemset) of the Apriori algorithm based on the Dataset provided is",freq_set1)
print("The output (association rules) of the Apriori algorithm based on the Dataset provided is",rule1)
print("The output (desired itemset) of the FP growth algorithm based on the Dataset provided is",freq_set2)
print("The output (association rules) of the FP growth algorithm based on the Dataset provided is",rule2)
```