MRIDUL HARISH, CED18I034, PROBLEM SET 6

Q1 - Test Drive ACLOSE algorithm to mine closed frequent patterns on a sample dataset of your choice. Test the same on a FIMI benchmark dataset which you have used for Apriori/FP-growth implementations.

In [ ]:
```
from skmine.datasets.fimi import fetch_mushroom
from skmine.datasets.utils import describe
import pandas as pd
import numpy as np
from sklearn.preprocessing import MultiLabelBinarizer

mushroom = fetch_mushroom()

n_items = describe(mushroom)['n_items']+2
n_transactions = describe(mushroom)['n_transactions']

for i in range(n_transactions):
    temp = []
    for j in mushroom[i]:
        temp.append(j-1)
    mushroom[i] = temp.copy()

describe(mushroom)
```

Out[ ]:
```
{'n_items': 117,
 'avg_transaction_size': 22.0,
 'n_transactions': 8124,
 'density': 0.18803418803418803}
```

In [ ]:

```
def count_itemset(dataset_v, itemset):
    temp_set = set()
    temp_set = dataset_v[itemset[0]]
    for item in itemset:
        temp_set = temp_set.intersection(dataset_v[item])
    return len(temp_set)

def candidate_with_min_support(dataset_v, C, min_support_count):
    L = list()
    for itemset in C:
        if count_itemset(dataset_v, itemset) > min_support_count:
            L.append(itemset)
    return L

def get_vertical_dataset(horizontal_dataset, n_transactions):
    dataset_v = dict()
    for k in range(n_items):
        dataset_v[k] = set()

    for i in range(n_transactions):
        for j in horizontal_dataset[i]:
            dataset_v[j].add(i)

    return dataset_v

def get_c2(L1):
    C2 = list()
    for i in range(len(L1)):
        for j in range(len(L1)):
            if i < j:
                C2.append([L1[i][0], L1[j][0]])
    return C2

def is_compatible(itemset1, itemset2):
    l = len(itemset1)
    flag = True
    for i in range(l-1):
        if itemset1[i] != itemset2[i]:
            flag = False
    return flag

def get_candidate_itemsets_k(L):
    Ck = list()
    for i in range(len(L)):
        for j in range(len(L)):
            if i < j and is_compatible(L[i], L[j]):
                l = len(L[i])
                temp_itemset = list()
                for z in range(l-1):
                    temp_itemset.append(L[i][z])
                temp_itemset.append(L[i][l-1])
                temp_itemset.append(L[j][l-1])
                Ck.append(temp_itemset)
    return Ck

def can_it_be_pruned(itemset, L_prev):
    l = len(itemset)
    flag = False
```

```
    for i in range(l):
        temp_itemset = itemset.copy()
        temp_itemset.remove(itemset[i])
        if temp_itemset not in L_prev:
            flag = True
    return flag

def prune(C_now, L_prev):
    temp_C = list()
    for itemset in C_now:
        if can_it_be_pruned(itemset, L_prev) == False:
            temp_C.append(itemset)
    return temp_C

def t(item_list, dataset_v):
    temp_set = set()
    temp_set = dataset_v[item_list[0]]
    for item in item_list:
        temp_set = temp_set.intersection(dataset_v[item])
    return list(temp_set)

def i(transaction_list, dataset):
    temp_set = set()
    temp_set = set(dataset[transaction_list[0]])
    for transaction in transaction_list:
        temp_set = temp_set.intersection(set(dataset[transaction]))
    return list(temp_set)

def merge_L(L):
    tn = len(L)
    merged_L = list()
    for i in range(1,tn):
        for item_list in L[i]:
            merged_L.append(item_list)
    return merged_L

def is_subset(itemlist_i, itemlist_j):
    return set(itemlist_i).issubset(set(itemlist_j))

def remove_sets_of_same_support(temp_L, L, dataset_v):
    merged_L = merge_L(L)
    index_set = set()
    for i in range(len(merged_L)):
        for j in range(len(temp_L)):
            if (is_subset(merged_L[i], temp_L[j]) and len(t(merged_L[i], dataset_v))
                index_set.add(j)

    final_L = list()
    for i in range(len(temp_L)):
        if i not in index_set:
            final_L.append(temp_L[i])

    return final_L

def aclose(dataset, min_support, n_transactions, n_items):

    dataset_v = get_vertical_dataset(dataset, n_transactions)

    C = list()
```

```
    L = list()
    min_support_count = int(min_support * (n_transactions))
    C.append([False])
    L.append([False])

    temp_C = [[k] for k in range(n_items)]
    C.append(temp_C)

    temp_L = candidate_with_min_support(dataset_v, C[1], min_support_count)
    L.append(temp_L)

    if(len(L[1]) > 0):
        temp_C = get_c2(L[1])
        C.append(temp_C)

        temp_L = candidate_with_min_support(dataset_v, C[2], min_support_count)
        temp_L = remove_sets_of_same_support(temp_L, L, dataset_v)
        L.append(temp_L)

    flag = True
    last_l = len(L)-1;
    k = last_l + 1
    if(len(L[last_l]) == 0):
        flag = False

    while flag:
        #CK
        temp_C = get_candidate_itemsets_k(L[k-1])
        temp_C = prune(temp_C, L[k-1])
        C.append(temp_C)

        temp_L = candidate_with_min_support(dataset_v, C[k], min_support_count)
        temp_L = remove_sets_of_same_support(temp_L, L, dataset_v)
        L.append(temp_L)

        k = k + 1

        flag = True
        last_l = len(L)-1;
        k = last_l + 1
        if(len(L[last_l]) == 0):
            flag = False

    closed_itemsets = set()
    merged_L = merge_L(L)
    for itemlist in merged_L:
        closed_itemsets.add(tuple(i(t(itemlist, dataset_v), dataset)))
    closed_itemlist = list()
    for itemtuple in closed_itemsets:
        closed_itemlist.append(list(itemtuple))

    return closed_itemlist
```

In [ ]:

```
closed_itemsets = aclose(mushroom, 0.4, n_transactions, n_items)
print(closed_itemsets)
```

```
[[33, 38, 84, 85, 58], [33, 66, 84, 85, 89], [33, 35, 84, 85, 58], [33, 84, 85, 89, 9
```

```
2], [58, 84, 38], [89, 84, 38], [84], [33, 84, 23], [33, 75, 84, 85, 58], [33, 35, 3
8, 84, 85], [33, 84, 85, 89, 58], [33, 35, 84, 85, 55, 89], [33, 51, 84, 85], [35, 8
4, 89, 58, 62], [33, 84], [33, 35, 109, 84, 85], [58, 35, 84], [89, 35, 84], [33, 84,
85, 23], [89, 35, 84, 38], [84, 85, 38], [33, 35, 84, 85, 23], [35, 84, 85, 23], [33,
35, 52, 85, 84, 89], [33, 27, 84, 85], [51, 84, 85], [58, 35, 84, 92], [35, 84, 92],
[33, 35, 84, 85, 62], [89, 84, 33], [33, 35, 84, 85], [38, 84, 62], [33, 35, 84, 85,
89], [33, 38, 84, 85, 89], [84, 109], [33, 84, 85, 89, 62], [51, 84], [33, 35, 84, 8
5, 89, 58], [35, 84, 85], [33, 35, 84, 85, 55], [33, 27, 84], [33, 2, 84, 85, 89], [8
4, 38], [35, 84, 23], [89, 35, 84, 92], [2, 84], [33, 84, 85, 38], [58, 84, 92], [89,
84, 92], [58, 35, 84, 62], [35, 84, 62], [33, 84, 85, 55, 89], [33, 84, 85, 92, 62],
[33, 84, 85, 22], [33, 84, 85, 89, 58, 62], [33, 35, 84, 85, 58, 62], [89, 84, 23, 3
3], [33, 75, 84, 85], [33, 75, 84, 85, 89], [89, 84, 23], [33, 84, 85, 23, 89], [84,
62, 92], [84, 38, 92], [89, 35, 84, 62], [33, 66, 84, 85, 58], [89, 58, 84, 62], [58,
84, 62], [89, 84, 62], [89, 2, 84], [89, 58, 84, 92], [33, 66, 75, 84, 85], [33, 66,
84, 85], [84, 92], [89, 84, 85, 33], [33, 35, 38, 84, 85, 55], [33, 109, 84, 85, 89],
[33, 35, 84, 85, 89, 62], [33, 84, 85, 92], [35, 84, 85, 38], [33, 85, 84, 109], [33,
35, 84, 85, 23, 89], [84, 85], [33, 35, 84, 85, 22], [58, 84], [89, 35, 84, 23], [84,
23], [33, 84, 85, 58, 62], [33, 52, 85, 84, 89], [89, 58, 35, 84], [35, 84, 109], [3
3, 51, 84], [33, 35, 109, 84, 85, 89], [33, 84, 85, 58, 92], [84, 62], [35, 84], [33,
35, 38, 84, 85, 89], [84, 85, 23], [35, 84, 38], [33, 84, 85, 62], [33, 35, 38, 84, 8
5, 55, 89], [2, 84, 85], [33, 2, 84, 85], [89, 35, 84, 109], [89, 84], [33, 58, 84, 8
5], [33, 84, 85, 55], [33, 35, 84, 85, 92], [27, 84], [89, 84, 109], [33, 84, 85], [2
```

Q2 - Test Drive Pincer search to mine maximal frequent patterns on a sample dataset of your choice. Test the same on a FIMI benchmark dataset which you have used for Apriori/FP-growth implementations.

In [ ]:

```
from skmine.datasets.fimi import fetch_mushroom
from skmine.datasets.utils import describe
import pandas as pd
import numpy as np
from sklearn.preprocessing import MultiLabelBinarizer

mushroom = fetch_mushroom()

n_items = describe(mushroom)['n_items']+2
n_transactions = describe(mushroom)['n_transactions']

for i in range(n_transactions):
    temp = []
    for j in mushroom[i]:
        temp.append(j-1)
    mushroom[i] = temp.copy()

describe(mushroom)
```

Out[ ]:

```
{'n_items': 117,
 'avg_transaction_size': 22.0,
 'n_transactions': 8124,
 'density': 0.18803418803418803}
```

In [ ]:

```
def count_itemset(dataset_v, itemset):
    temp_set = set()
    temp_set = dataset_v[itemset[0]]
    for item in itemset:
        temp_set = temp_set.intersection(dataset_v[item])
    return len(temp_set)

def get_vertical_dataset(horizontal_dataset):
    dataset_v = dict()
    for k in range(n_items):
        dataset_v[k] = set()

    for i in range(n_transactions):
        for j in horizontal_dataset[i]:
            dataset_v[j].add(i)

    return dataset_v

def is_compatible(itemset1, itemset2):
    l = len(itemset1)
    flag = True
    for i in range(l-1):
        if itemset1[i] != itemset2[i]:
            flag = False
    return flag

def remove_duplicates(new_MFS):
    temp_l = list()
    for itemlist in new_MFS:
        if itemlist not in temp_l:
            temp_l.append(itemlist)
    return temp_l

def update_MFS(MFS, MFCS, min_support_count, dataset_v):
    new_MFS = MFS.copy()
    for itemset in MFCS:
        if count_itemset(dataset_v, itemset) > min_support_count:
            new_MFS.append(itemset)
    new_MFS = remove_duplicates(new_MFS)
    return new_MFS

def is_subset(itemlist_i, itemlist_j):
    return set(itemlist_i).issubset(set(itemlist_j))

def is_subset_any_MFS(itemset, MFS):
    flag = False
    for itemlist in MFS:
        if is_subset(itemset, itemlist):
            flag = True
    return flag

def get_LK_SK(dataset_v, CK, min_support_count, MFS):
    LK = list()
    SK = list()
    for itemset in CK:
        if count_itemset(dataset_v, itemset) > min_support_count:
            LK.append(itemset)
        else:
```

```
            SK.append(itemset)

    LK_new = list()
    for itemset in LK:
        if is_subset_any_MFS(itemset, MFS) == False:
            LK_new.append(itemset)

    return LK_new, SK

def any_itemset_CK_subset_itemset_MFS(CK, MFS):
    flag=False
    for itemset1 in CK:
        for itemset2 in MFS:
            if is_subset(itemset1, itemset2):
                flag=True
    return flag

def recovery(temp_C, LK, MFS, K):
    new_temp_C = temp_C.copy()
    for l in LK:
        for m in MFS:
            if is_compatible(l,m):
                for i in range(K-1, len(m)):
                    temp_l = l.copy()
                    temp_l.append(m[i])
                    new_temp_C.append(temp_l)

    return (new_temp_C)

def get_immediate_subsets(C):
    mega_list = list()
    for c in C:
        C_copy = C.copy()
        C_copy.remove(c)
        temp_list = C_copy.copy()
        mega_list.append(temp_list)
    return mega_list

def new_prune(LK, MFS, temp_C, K):
    new_temp_C = temp_C.copy()
    for c in temp_C:
        if is_subset_any_MFS(c, MFS):
            new_temp_C.remove(c)
        else:
            k_subsets_of_C = get_immediate_subsets(c)
            for s in k_subsets_of_C:
                if (s not in LK) and (c in new_temp_C):
                    new_temp_C.remove(c)
    return new_temp_C

def MFCS_gen(MFCS, SK):
    new_MFCS = MFCS.copy()
    for s in SK:
        for m in new_MFCS:
            temp_MFCS = new_MFCS.copy()
            if (is_subset(s, m)):
                if (m in temp_MFCS):
                    temp_MFCS.remove(m)
                for e in s:
```

```
                    temp_l = m.copy()
                    temp_l.remove(e)

                    if is_subset_any_MFS(temp_l, temp_MFCS) == False:
                        temp_MFCS.append(temp_l)
        new_MFCS = temp_MFCS.copy()
    return new_MFCS

def pincersearch(dataset, min_support, n_transactions, n_items):
    dataset_v = get_vertical_dataset(dataset)

    L = list()
    K = 1
    C = list()
    S = list()
    MFCS = list()
    MFS = list()

    C.append([False])
    L.append([False])
    S.append([False])

    temp_l = list()
    for i in range(n_items):
        temp_l.append(i)
    MFCS.append(temp_l)


    temp_C = [[k] for k in range(n_items)]
    C.append(temp_C)

    min_support_count = int(min_support * n_transactions)

    while len(C[K]):

        MFS = update_MFS(MFS, MFCS, min_support_count, dataset_v)
        LK, SK = get_LK_SK(dataset_v, C[K], min_support_count, MFS)
        L.append(LK)
        S.append(SK)

        temp_C = get_candidate_itemsets_k(L[K])

        if any_itemset_CK_subset_itemset_MFS(C[K], MFS):
            temp_C = recovery(temp_C, L[K], MFS, K)
        temp_C = new_prune(L[K], MFS, temp_C, K)
        C.append(temp_C)

        MFCS = MFCS_gen(MFCS, S[K])
        K = K + 1

    return MFS
```

In [ ]:

```
maximal_freq_itemsets = pincersearch(mushroom, 0.3, n_transactions, n_items)
print(maximal_freq_itemsets)
```

```
[[2, 33, 35, 84, 85, 89]]
```

Q4 - Implement the Decision Tree model on the dataset we have discussed in the class and also

test the same on a benchmark dataset (preferably binary classification dataset). Also explore if there are any packages to display the decision tree.

In [ ]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn import tree
import matplotlib.pyplot as plt

inputs = pd.read_csv("buyscomputer.csv")
```

In [ ]:
```
inputs
```

Out[ ]:

	id	age	income	student	credit_rating	buys_computer
0	1	youth	low	no	fair	yes
1	2	youth	high	yes	excellent	no
2	3	youth	high	no	fair	yes
3	4	middle	medium	yes	fair	yes
4	5	senior	low	yes	fair	yes
5	6	senior	low	yes	excellent	yes
6	7	middle	high	yes	excellent	yes
7	8	youth	medium	no	fair	no
8	9	middle	low	no	excellent	yes
9	10	senior	medium	yes	fair	yes

In [ ]:
```
le_age = LabelEncoder()
le_income = LabelEncoder()
le_student = LabelEncoder()
le_credit_rating = LabelEncoder()
le_buys_computer = LabelEncoder()

inputs["age_n"] = le_age.fit_transform(inputs["age"])
inputs["income_n"] = le_income.fit_transform(inputs["income"])
inputs["student_n"] = le_student.fit_transform(inputs["student"])
inputs["credit_rating_n"] = le_credit_rating.fit_transform(inputs["credit_rating"])
inputs["buys_computer_n"] = le_buys_computer.fit_transform(inputs["buys_computer"])
```

In [ ]:
```
inputs_n = inputs.drop(["id", "age", "income", "student", "credit_rating", "buys_comp
Y = inputs_n["buys_computer_n"]
X = inputs_n.drop(["buys_computer_n"], axis=1)
```

In [ ]:
```
model = tree.DecisionTreeClassifier()
model.fit(X.values,Y.values)
model.score(X.values,Y.values)
```

Out[ ]: 1.0

In [ ]:
```
def buys_computer_predict_api(age, income, student, credit_rating):
    age_en = le_age.transform([age])[0]
    income_en = le_income.transform([income])[0]
    student_en = le_student.transform([student])[0]
    credit_rating_en = le_credit_rating.transform([credit_rating])[0]
    buys_computer_en = model.predict([[age_en, income_en, student_en, credit_rating_e
    return le_buys_computer.inverse_transform([buys_computer_en])[0]
```

In [ ]:
```
#prediction
print(buys_computer_predict_api(age = "senior", income = "low", student = "yes", cre
print(buys_computer_predict_api(age = "youth", income = "low", student = "yes", cred
```

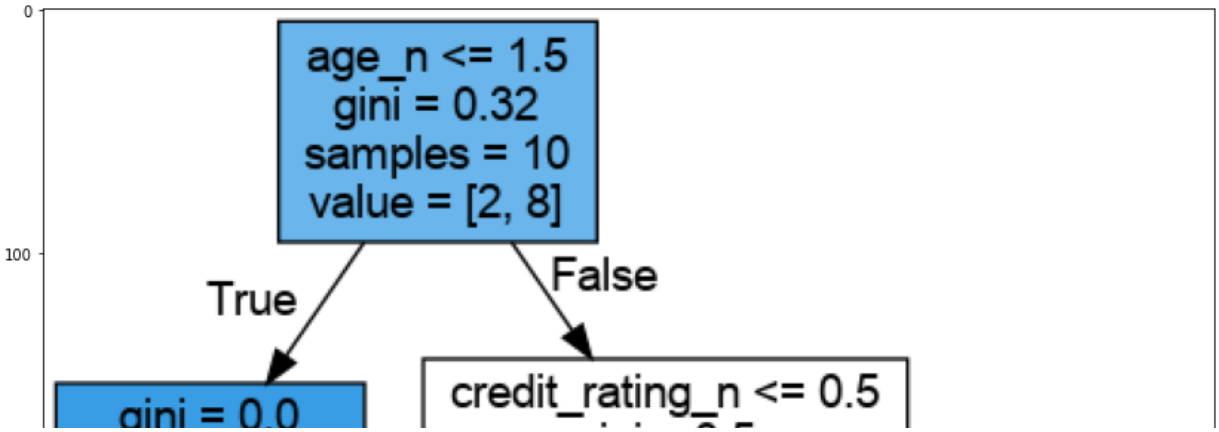
```
yes
no
```

In [ ]:
```
#Display tree
decision_tree = tree.export_graphviz(model, out_file="buys_computer.dot", feature_nar
```

In [ ]:
```
!dot -Tpng buys_computer.dot -o buys_computer.png
```

In [ ]:
```
img = plt.imread("buys_computer.png")
plt.figure(figsize=(15,15))
plt.imshow(img)
```

Out[ ]: <matplotlib.image.AxesImage at 0x7f3af21777c0>



In [ ]:
```
from scipy.io import arff
import pandas as pd

data = arff.loadarff('caesarian.arff')
df = pd.DataFrame(data[0])
df_new = df.copy()

print(df.columns)

df_new['Age'] = df['Age'].str.decode("utf-8")
df_new['Delivery number'] = df['Delivery number'].str.decode("utf-8")
df_new['Delivery time'] = df['Delivery time'].str.decode("utf-8")
df_new['Blood of Pressure'] = df['Blood of Pressure'].str.decode("utf-8")
df_new['Heart Problem'] = df['Heart Problem'].str.decode("utf-8")
df_new['Caesarian'] = df['Caesarian'].str.decode("utf-8")
```

```
Index(['Age', 'Delivery number', 'Delivery time', 'Blood of Pressure',
       'Heart Problem', 'Caesarian'],
      dtype='object')
```

In [ ]:
```
df_new
```

Out[ ]:

	Age	Delivery number	Delivery time	Blood of Pressure	Heart Problem	Caesarian
0	22	1	0	2	0	0
1	26	2	0	1	0	1
2	26	2	1	1	0	0
3	28	1	0	2	0	0
4	22	2	0	1	0	1
...	...	...	...	...	...	...
75	27	2	1	1	0	0
76	33	4	0	1	0	1
77	29	2	1	2	0	1
78	25	1	2	0	0	1
79	24	2	2	1	0	0

80 rows × 6 columns

In [ ]:
```
Y = df_new["Caesarian"]
X = df_new.drop(["Caesarian"], axis=1)
```

In [ ]:
```
model = tree.DecisionTreeClassifier(criterion='entropy')
model.fit(X.values,Y.values)
model.score(X.values,Y.values)
```

Out[ ]: 0.95

In [ ]:
```
print(model.predict([[33,4,0,1,0]])[0])
print(model.predict([[28,2,0,1,1]])[0])
print(model.predict([[25,1,0,0,0]])[0])
```

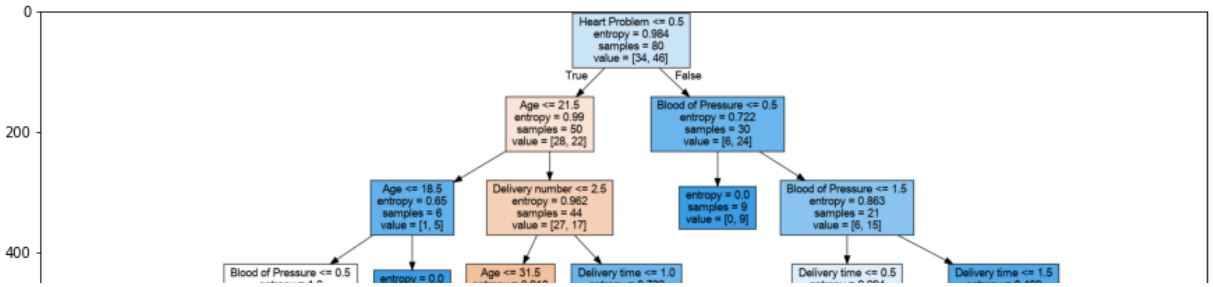
```
1
1
0
```

In [ ]:
```
#Display tree
decision_tree = tree.export_graphviz(model, out_file="caesarian.dot", feature_names=X
```

In [ ]:
```
!dot -Tpng caesarian.dot -o caesarian.png
```

In [ ]:
```
img = plt.imread("caesarian.png")
plt.figure(figsize=(15,15))
plt.imshow(img)
```

Out[ ]: <matplotlib.image.AxesImage at 0x7f3ae067c730>


0
200
400
Heart Problem <= 0.5
entropy = 0.984
samples = 80
value = [34, 46]
True
False
Age <= 21.5
entropy = 0.99
samples = 50
value = [28, 22]
Blood of Pressure <= 0.5
entropy = 0.722
samples = 30
value = [6, 24]
Age <= 18.5
entropy = 0.65
samples = 6
value = [1, 5]
Delivery number <= 2.5
entropy = 0.982
samples = 44
value = [27, 17]
entropy = 0.0
samples = 9
value = [0, 9]
Blood of Pressure <= 1.5
entropy = 0.863
samples = 21
value = [6, 15]
Blood of Pressure <= 0.5
entropy = 0.0
Age <= 31.5
Delivery time <= 1.0
Delivery time <= 0.5
Delivery time <= 1.5