CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep dari pembelajaran mesin kemudian di terapkan pada permasalahan – (C2)
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin untuk mendapatkan pemecahan maslaah dan model yang sesuai – (C3)
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan jika terdapat perkembangan di bidang rekayasa cerdas – (C6)

MINGGU 11

Algoritma Densitas dan Asosiasi

DESKRIPSI TEMA

Mahasiswa mempelajari pembelajaran tidak terbimbing dengan algoritma densitas dan asosiatif yaitu algoritma DBSCAN, algoritma Apriori, algoritma Fp-Growth

CAPAIAN PEMBELAJARAN MINGGUAN (SUB CPMK)

Mahasiswa dapat menerapkan algoritma densitas dan asosiatif sehingga bisa menghasilkan model terbaik

PERALATAN YANG DIGUNAKAN

Anaconda Python 3
Laptop atau Personal Computer

LANGKAH-LANGKAH PRAKTIKUM

In machine learning, the types of learning broadly be classified into 3 types. So this is one of three types. It called Unsupervised Learning. Algorithms belonging to the family of unsupervised learning have no variable to predict tied to the data. Here some of the algorithm.

Density Algorithm : DBSCAN

DBSCAN – Density Based Spatial Clustering of Application with Noise. Find core samples of high density and expands clusters from them. Good for data which contains cluster of similar density. Let's try the data with generate dataset. We using openCv format to convert.

1. Import the Important Library

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from sklearn.cluster import DBSCAN
```

2. Define a function to generate clusters

```python
def cluster_gen(n_clusters, pts_minmax=(10, 100), x_mult=(1, 4), y_mult=(1, 3),
                x_off=(0, 50), y_off=(0, 50)):

    # n_clusters = number of clusters to generate
    # pts_minmax = range of number of points per cluster
    # x_mult = range of multiplier to modify the size of cluster in the x-direction
    # y_mult = range of multiplier to modify the size of cluster in the x-direction
    # x_off = range of cluster position offset in the x-direction
    # y_off = range of cluster position offset in the y-direction

    # Initialize some empty lists to receive cluster member positions
    clusters_x = []
    clusters_y = []
    # Genereate random values given parameter ranges
    n_points = np.random.randint(pts_minmax[0], pts_minmax[1], n_clusters)
    x_multipliers = np.random.randint(x_mult[0], x_mult[1], n_clusters)
    y_multipliers = np.random.randint(y_mult[0], y_mult[1], n_clusters)
    x_offsets = np.random.randint(x_off[0], x_off[1], n_clusters)
    y_offsets = np.random.randint(y_off[0], y_off[1], n_clusters)

    # Generate random clusters given parameter values
    for idx, npts in enumerate(n_points):

        xpts = np.random.randn(npts) * x_multipliers[idx] + x_offsets[idx]
        ypts = np.random.randn(npts) * y_multipliers[idx] + y_offsets[idx]
        clusters_x.append(xpts)
        clusters_y.append(ypts)

    # Return cluster positions
    return clusters_x, clusters_y

```

3. Generate some clusters

```python
n_clusters = 50
clusters_x, clusters_y = cluster_gen(n_clusters)
```

4. Convert to a single dataset in OpenCV format

```python
data = np.float32((np.concatenate(clusters_x),
                   np.concatenate(clusters_y))).transpose()
```

5. Define max_distance (eps parameter in DBSCAN)

```python
max_distance = 1
db = DBSCAN(eps=max_distance, min_samples=10).fit(data)
```

6. Extract a mask of core cluster members

```python
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
```

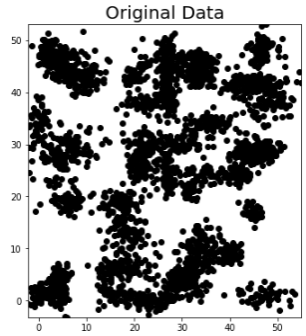
7. Extract labels (-1 is used for outliers)

```python
labels = db.labels_
n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
unique_labels = set(labels)
```

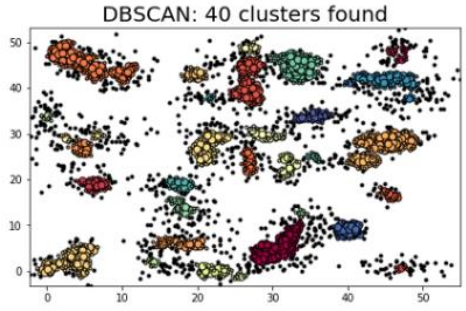
8. Plot the result

```
min_x = np.min(data[:, 0])
max_x = np.max(data[:, 0])
min_y = np.min(data[:, 1])
max_y = np.max(data[:, 1])

fig = plt.figure(figsize=(12,6))
plt.subplot(121)
plt.plot(data[:,0], data[:,1], 'ko')
plt.xlim(min_x, max_x)
plt.ylim(min_y, max_y)
plt.title('Original Data', fontsize = 20)

plt.subplot(122)
```

Original data :




```
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = data[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=7)

    xy = data[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=3)
plt.xlim(min_x, max_x)
plt.ylim(min_y, max_y)
plt.title('DBSCAN: %d clusters found' % n_clusters, fontsize = 20)
fig.tight_layout()
plt.subplots_adjust(left=0.03, right=0.98, top=0.9, bottom=0.05)
```



What if we used DBSCAN algorithm for clustering customer behavior? Lets do with dataset Mall_Customer.csv (clustering customer segmentation).

About dataset content : (describe dataset from kaggle.com)

You are owing a supermarket mall and through membership cards , you have some basic data about your customers like Customer ID, age, gender, annual income and spending score. Spending Score is something you assign to the customer based on your defined parameters like customer behavior and purchasing data.

Problem Statement You own the mall and want to understand the customers like who can be easily converge [Target Customers] so that the sense can be given to marketing team and plan the strategy accordingly.

9. Import the important libraries

```
import numpy as np
import pandas as pd

import os
#print(os.listdir("../input"))

import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

10. Explore the data :

```
1 df = pd.read_csv('Mall_Customers.csv')
2 df.head()
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

```
1 df.isnull().sum()
```

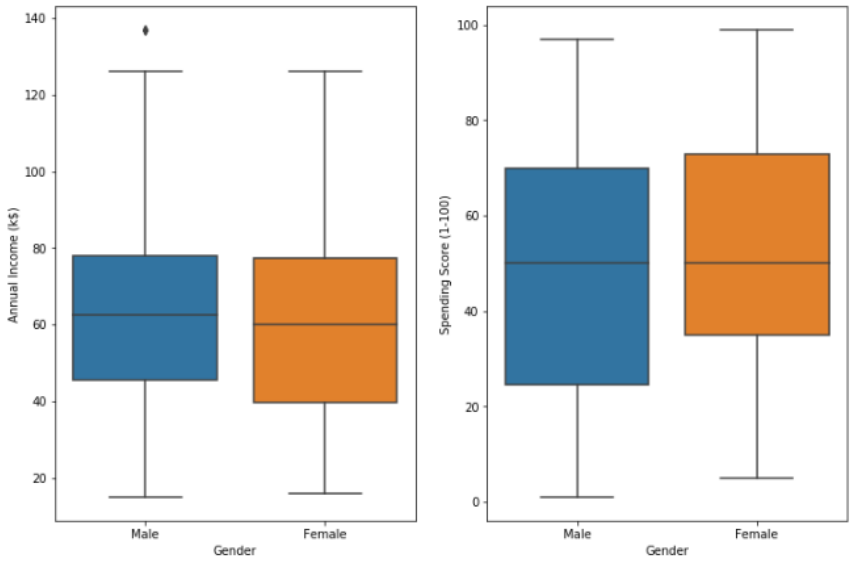
```
CustomerID                0
Gender                    0
Age                       0
Annual Income (k$)        0
Spending Score (1-100)    0
dtype: int64
```

```
1 df.dtypes
```

```
CustomerID                int64
Gender                   object
Age                       int64
Annual Income (k$)        int64
Spending Score (1-100)    int64
dtype: object
```

11. Find insight between male and female

```
1 fig, axes = plt.subplots(1, 2, figsize=(12,8))
2
3 sns.boxplot(x="Gender", y="Annual Income (k$)", data=df, orient='v' , ax=axes[0])
4 sns.boxplot(x="Gender", y="Spending Score (1-100)", data=df, orient='v' , ax=axes[1])
```



12. From graph above, looks like ‘female’ spend more than men. Let’s focus on ‘female’!

Grouping data with ‘female’

```
df_group_one = df[['Gender','Annual Income (k$)','Spending Score (1-100)']]
df_group_one.groupby(['Gender'],as_index=False).mean()
```

	Gender	Annual Income (k$)	Spending Score (1-100)
0	Female	59.250000	51.526786
1	Male	62.227273	48.511364

```
df_female = df[df['Gender'] == "Female"]
print(df_female.shape)
df_female.head()
```

```
(112, 5)
```

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40
5	6	Female	22	17	76
6	7	Female	35	18	6

```
Percentage = (df_female.shape[0]/df.shape[0])*100
print('Female Percentage: ', round(Percentage), '%')
```

```
Female Percentage:  56 %
```

Compute DBSCAN

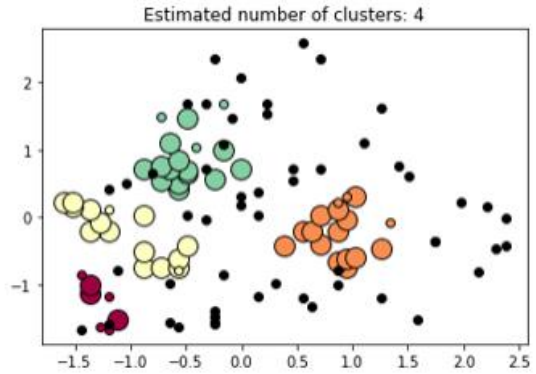
13. We setting with a little so densy. Thus more noises will appear, but we can find a lot more insight with very sensitive result. We set the eps = 0.5 and min_samples = 4.

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler

Clus_dataSet = df_female[['Age','Annual Income (k$)','Spending Score (1-100)']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = np.array(Clus_dataSet, dtype=np.float64)
Clus_dataSet = StandardScaler().fit_transform(Clus_dataSet)

# Compute DBSCAN
db = DBSCAN(eps=0.5, min_samples=4).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
df_female['Clus_Db']=labels

realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)
clusterNum = len(set(labels))

# A sample of clusters
print(df_female[['Age','Annual Income (k$)','Spending Score (1-100)','Clus_Db']].head())

# number of labels
print("number of labels: ", set(labels))
```

```
   Age  Annual Income (k$)  Spending Score (1-100)  Clus_Db
2   20                  16                       6       -1
3   23                  16                      77        0
4   31                  17                      40       -1
5   22                  17                      76        0
6   35                  18                       6       -1
number of labels:  {0, 1, 2, 3, -1}
```

14. Visualize the data :

```
# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = Clus_dataSet[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = Clus_dataSet[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % realClusterNum)
plt.show()

n_noise_ = list(labels).count(-1)
print('number of noise(s): ', n_noise_)
```




```
number of noise(s):  54
```

```
for clust_number in set(labels):
    clust_set = df_female[df_female.Clus_Db == clust_number]
    if clust_number != -1:
        print ("Cluster "+str(clust_number)+', Avg Age: '+ str(round(np.mean(clust_set.Age)))+\
               ', Avg Income: '+ str(round(np.mean(clust_set['Annual Income (k$)'])))+\
               ', Avg Spending: '+ str(round(np.mean(clust_set['Spending Score (1-100)'])))+\
               ', Count: '+ str(np.count_nonzero(clust_set.index)))
```

```
Cluster 0, Avg Age: 22, Avg Income: 26, Avg Spending: 78, Count: 7
Cluster 1, Avg Age: 49, Avg Income: 54, Avg Spending: 50, Count: 17
Cluster 2, Avg Age: 25, Avg Income: 52, Avg Spending: 50, Count: 16
Cluster 3, Avg Age: 32, Avg Income: 82, Avg Spending: 82, Count: 18
```

Conclusion :

With tightly setting, we found that “cluster 3 (average age :32)” which highest average income and spending is the most potential customer group. The second candidate is "Cluster 0 (average age: 22)" which lowest average income but spending a lot. For some countries, "Cluster 3" will represent people who are being settled down (have a family)

which spending most for their children. And "Cluster 0" will represent people who are starting a new work-life after graduated that spending most for their new society life.

Association Learning – Apriori Algorithm

Apriori is a popular algorithm for extracting frequent itemset with applications in association learning. The apriori algorithm has been design to operate databases containing transactions such as purchases by customers of a store. An itemset is considered as "frequent" if it meets a user-specified support threshold. For instance, if the support threshold is set to 0.5 (50%), a frequent itemset is defined as a set of items that occur together in at least 50% of all transactions in the database.

In this section we use libraries that very useful namely **MLxtend** by Sebastian Raschka for extracting frequent item set for further analysis. How to install? Follow the guide in : http://rasbt.github.io/mlxtend/installation/

```
pip install mlxtend
```

15. Lets generating Frequent Itemset :

```
dataset = [['Milk', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
           ['Dill', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
           ['Milk', 'Apple', 'Kidney Beans', 'Eggs'],
           ['Milk', 'Unicorn', 'Corn', 'Kidney Beans', 'Yogurt'],
           ['Corn', 'Onion', 'Onion', 'Kidney Beans', 'Ice cream', 'Eggs']]
```

16. Transform the dataset into the right format via the TransactionEncoder :

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder

te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)
df
```

	Apple	Corn	Dill	Eggs	Ice cream	Kidney Beans	Milk	Nutmeg	Onion	Unicorn	Yogurt
0	False	False	False	True	False	True	True	True	True	False	True
1	False	False	True	True	False	True	False	True	True	False	True
2	True	False	False	True	False	True	True	False	False	False	False
3	False	True	False	False	False	True	True	False	False	True	True
4	False	True	False	True	True	True	False	False	True	False	False

17. Set the items and itemsets with at least 60% support :

```
from mlxtend.frequent_patterns import apriori

apriori(df, min_support=0.6)
```

	support	itemsets
0	0.8	(3)
1	1.0	(5)
2	0.6	(6)

By default, apriori returns the column indices of the items, which may be useful in downstream operations such as association rule mining. For better readability, we can set use_colnames=True to convert these integer values into the respective item names.

```
apriori(df, min_support=0.6, use_colnames=True)
```

	support	itemsets
0	0.8	(Eggs)
1	1.0	(Kidney Beans)
2	0.6	(Milk)
3	0.6	(Onion)
4	0.6	(Yogurt)
5	0.8	(Kidney Beans, Eggs)
6	0.6	(Onion, Eggs)
7	0.6	(Milk, Kidney Beans)
8	0.6	(Onion, Kidney Beans)
9	0.6	(Yogurt, Kidney Beans)
10	0.6	(Onion, Kidney Beans, Eggs)

18. Selecting and Filtering Result for the dataset. For instance, let's assume we are only interested in itemsets of length 2 that have a support at least 80%. First, we create the frequent itemsets via apriori and add new columns that stores the length of each itemset.

```
frequent_itemsets = apriori(df, min_support=0.6, use_colnames=True)
frequent_itemsets['length'] = frequent_itemsets['itemsets'].apply(lambda x: len(x))
frequent_itemsets
```

	support	itemsets	length
0	0.8	(Eggs)	1
1	1.0	(Kidney Beans)	1
2	0.6	(Milk)	1
3	0.6	(Onion)	1
4	0.6	(Yogurt)	1
5	0.8	(Kidney Beans, Eggs)	2
6	0.6	(Onion, Eggs)	2
7	0.6	(Milk, Kidney Beans)	2
8	0.6	(Onion, Kidney Beans)	2
9	0.6	(Yogurt, Kidney Beans)	2
10	0.6	(Onion, Kidney Beans, Eggs)	3

19. Than we can select the results that satisfy our desired criteria as follows :

```
frequent_itemsets[ (frequent_itemsets['length'] == 2) &
                   (frequent_itemsets['support'] >= 0.8) ]
```

	support	itemsets	length
5	0.8	(Kidney Beans, Eggs)	2

20. Similarly, we can select entries based on the itemsets.

```
frequent_itemsets[ frequent_itemsets['itemsets'] == {'Onion', 'Eggs'} ]
```

	support	itemsets	length
6	0.6	(Onion, Eggs)	2

Association Learning – FP Growth Algorithm

FP Growth is an algorithm for extracting frequent itemsets with applications in association rule learning that emerged as a popular alternative to the established Apriori algorighm. In general, the algorithm has been designed to operate on databases containing transactions, such as purchases by customers of a store. An itemset is considered as "frequent" if it meets a user-specified support threshold. For instance, if the support threshold is set to 0.5 (50%), a frequent itemset is defined as a set of items that occur together in at least 50% of all transactions in the database.

In particular, and what makes it different from the Apriori frequent pattern mining algorithm, FP-Growth is an frequent pattern mining algorithm that does not require candidate generation. Internally, it uses a so-called FP-tree (frequent pattern tree) datastrucure without generating the candidate sets explicitely, which makes is particularly attractive for large datasets.

21. Using the same dataset, we import the important libraries with at least 60% support :

```
from mlxtend.frequent_patterns import fpgrowth

fpgrowth(df, min_support=0.6)
```

	support	itemsets
0	1.0	(5)
1	0.8	(3)
2	0.6	(10)
3	0.6	(8)
4	0.6	(6)
5	0.8	(3, 5)
6	0.6	(10, 5)
7	0.6	(8, 3)
8	0.6	(8, 5)
9	0.6	(8, 3, 5)
10	0.6	(5, 6)

22. By default, fpgrowth returns the columns indices of the items which may be useful in downstream operations such as association rule mining. For better readability, we can set use_colnames=True to convert these integer values into the respective item names.

```
fpgrowth(df, min_support=0.6, use_colnames=True)
```

	support	itemsets
0	1.0	(Kidney Beans)
1	0.8	(Eggs)
2	0.6	(Yogurt)
3	0.6	(Onion)
4	0.6	(Milk)
5	0.8	(Kidney Beans, Eggs)
6	0.6	(Yogurt, Kidney Beans)
7	0.6	(Onion, Eggs)
8	0.6	(Onion, Kidney Beans)
9	0.6	(Kidney Beans, Onion, Eggs)
10	0.6	(Milk, Kidney Beans)

23. Apriori Vs FP Growth

```
import pandas as pd
from mlxtend.preprocessing import TransactionEncoder

te = TransactionEncoder()
te_ary = te.fit(dataset).transform(dataset)
df = pd.DataFrame(te_ary, columns=te.columns_)
```

```
from mlxtend.frequent_patterns import apriori

%timeit -n 100 -r 10 apriori(df, min_support=0.6)
```

```
7.14 ms ± 394 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
```

```
from mlxtend.frequent_patterns import fpgrowth

%timeit -n 100 -r 10 fpgrowth(df, min_support=0.6)
```

```
1.65 ms ± 242 µs per loop (mean ± std. dev. of 10 runs, 100 loops each)
```

Since FP Growth doesn't require creating candidate sets explicitly, it can be magnitudes faster than the alternative Apriori algorithm. For instance, the following cells compare the performance of the Apriori algorithm to the performance of FP Growth. From the compare result, FP Growth is about 5 times faseter than Apriori.

Referensi :

Raschka S, Mirjailili. 2017. Phython Machine Leraning 2nd edition. Packt Publishing