Final Report of Term Project

Machine Learning

Thursday class

Team 1

2021.11.19

Table of contents

1. Project Objective 3p

2. Data Curation 3p

3. Data Exploration 3p

4. Model Assessment & Data Analysis 5p

5. AutoML 9p

6. WiseProphet 11p

7. Distribution 13p

8. Appendix 13p

   - code.py 13p
   - Preprocess.py 17p
   - FBClassifier.py 20p
   - FBClustering.py 26p

1. Project Objective

- The goal of our team is to find the most related features with the application rating what is in the Google Playstore.

2. Data Curation

➢ **Reference of the Dataset: Google Play Store Apps**

- https://www.kaggle.com/gauthamp10/google-playstore-apps

➢ **Description**

- Web scraped data from Google PlayStore.
- Android market have over 2-millions of apps.
- Each apps have a lot of information such as app name, most rated, rating score, etc.

➢ **Metadata of dataset**

- Created: April 5, 2019
- Updated: June 17, 2021 (Latest)

3. Data Exploration

➢ **Data Inspection**

There are 24 columns in our dataset. We selected 'Rating' feature for target and selected 4 features to training (Rating Count, Maximum Installs, Ad Supported, In App Purchases). Here is brief description for each column.

0. App Name :	Name of the app
1. App ID :	Package name
2. Category :	App category
3. Rating :	**Average rating**
4. Rating count :	**Number of rating**
5. Installs :	Approximate install count
6. Minimum Installs :	Approximate minimum app install count
7. Maximum Installs :	**Approximate maximum app install count**
8. Free :	Whether app is Free or Paid
9. Price :	App price
10. Currency :	App currency
11. Size :	Size of application package
12. Minimum Android :	Minimum android version supported
13. Developer Id :	Developer Id in Google Playstore
14. Developer Website :	Website of the developer
15. Developer Email :	Email-id of developer
16. Released :	App launch date on Google Playstore
17. Last Updated :	Last app update date
18. Content Rating :	Maturity level of app
19. Privacy Policy :	Privacy policy from developer
20. Ad Support :	**Ad support in app**
21. In App Purchases :	**In-App purchases in app**
22. Editors Choice :	Whether rated as Editor Choice
23. Scraped Time :	Scraped date-time in GMT

Feature Selection

We select Rating feature for target and select 4 other features to training.

```
RangeIndex: 2312944 entries, 0 to 2312943
Data columns (total 24 columns):
 #   Column              Dtype
---  ------              -----
 0   App Name            object
 1   App Id              object
 2   Category            object
 3   Rating              float64
 4   Rating Count        float64
 5   Installs            object
 6   Minimum Installs    float64
 7   Maximum Installs    int64
 8   Free                bool
 9   Price               float64
 10  Currency            object
 11  Size                object
 12  Minimum Android     object
 13  Developer Id        object
 14  Developer Website   object
 15  Developer Email     object
 16  Released            object
 17  Last Updated        object
 18  Content Rating      object
 19  Privacy Policy      object
 20  Ad Supported        bool
 21  In App Purchases    bool
 22  Editors Choice      bool
 23  Scraped Time        object
dtypes: bool(4), float64(4), int64(1), object(15
memory usage: 361.8+ MB
```

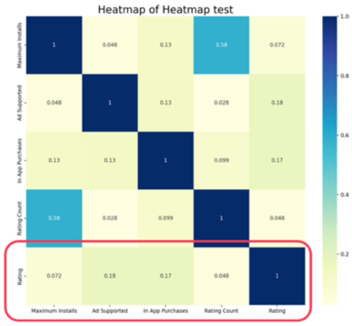
Handle Missing Values

We dropped row what have missing values in `Rating`.

```
App Name                 2
App Id                   0
Category                 0
Rating               22883
Rating Count         22883
Installs               107
Minimum Installs       107
Maximum Installs         0
Free                     0
Price                    0
Currency               135
Size                   196
Minimum Android       6530
Developer Id            33
Developer Website   760835
Developer Email         31
Released             71053
Last Updated             0
Content Rating           0
Privacy Policy      420953
Ad Supported             0
In App Purchases         0
Editors Choice           0
Scraped Time             0
dtype: int64
```

There are lots of missing values. Missing rating value related with no rating because of low reviews. These rows are removed. Other feature's missing values were removed. We check the selected feature's distribution. Several features have biased data. This might be preprocessed by using feature combination and drop rows.

We also checked correlation matrix with selected features and the target feature. Some features (Ad supported, In App Purchases) are related with Rating. But these top features related with `Rating` are not much have a value of correlation. This might be makes more harder to fitting machine learning models that avoid overfitting.



➢ Data Preprocessing

* Feature Selection

In the Feature selection step, we separated columns to 4 groups (Columns to be encoded, scaled, binned, dropped). Details are as follows

1) Columns to be encoded

→ Category, Free, Ad Supported, In App purchases, Editors Choice, Content Rating, Released, Last. Updated

2) Columns to be scaled

→ Rating Count, Installs, Minimum Installs, Maximum Installs

3) Columns to be dropped

→ App Id, Developer Website, Developer Email, Privacy Policy, Currency, Developer Id, Scaped Time, Minimum Android

4) Columns to be binned

→ **Rating** (target feature), Price

* Detailed Range of Binning

1) Rating There are 4 bins (1-4). The binning values are as follows	**2) Price** There are 4 bins (Free, Low, Mid, High). The binning values are as follows
· **1:** -0.1 <= x < 0.1 · **2:** 0.1 <= x < 1.66 · **3:** 1.66 <= x < 3.33 · **4:** 3.33 <= x <5.1	· **Free:** 0 < x < 0.19 · **Low:** 0.19 < x < 9.99 · **Mid:** 9.99 < x < 29.99 · **High:** 29.99 <= x < 410

* Text Preprocessing

1) Installs

→ There are non-numeric values like '+', 'Free'. So, filter these values to numeric value.

2) Last Updated

→ The original format of this column is "Feb 26, 2020" (for example) It is hard to identify. So, translate. this to YYYYMMDD format. e.g) 20200206

3) Category

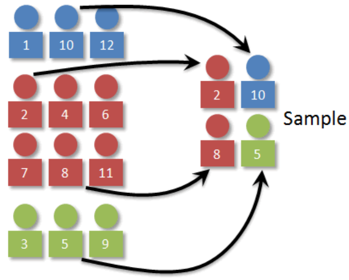
→ There are too many identical categories about 30. So, re-categorize them into 5 categories. (Entertainment, Lifestyle, Social, Education, Health)

<Last Updated>

<Category>

*** Stratified Shuffle Split**

Our team's dataset has **2,000,000 tuples.** It is too large scale to analyze. So, we want 20,000 tuples by 'Rating' bins (1-4) and pick the samples with consistent ratio. Contrast to basic split, it can distribute equally in the sample.



4. Model Assessment & Data Analysis

1) Classification

➢ **Machine learning model what we used (4 models)**

- Decision Tree Classifier, Logistic Classifier, GaussianNB (Naïve Bayes), Gradient Boosting Classifier

➢ **Dataset scaler what we used**

- None (No scale), Standard, Robust, MinMax, MaxAbs scaler

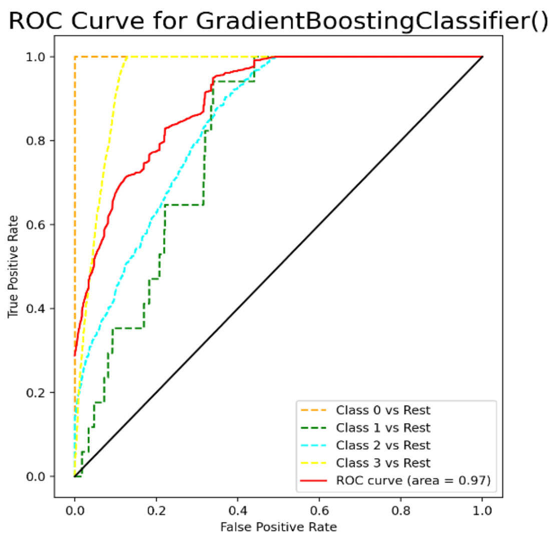
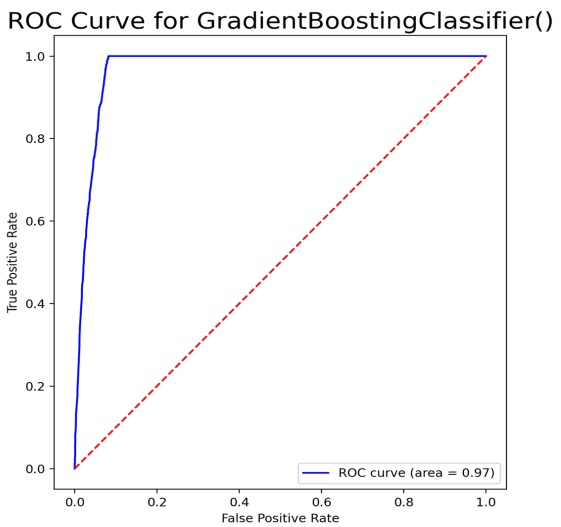
➢ **Model Assessment**

Before classification, the feature affecting the target was checked. The score of each variable was calculated using **SelectKBest()**, and the top 4 features(Maximum Installs, Rating Count, In App Purchases, Ad Supported) were extracted by sorting each feature.

	Col	Score
2	Maximum Installs	6.068258e+09
1	Rating Count	6.203113e+07
7	In App Purchases	5.550902e+02
6	Ad Supported	3.389297e+02
4	Last Updated	2.450677e+02
0	Category	1.782216e+02
8	Editors Choice	1.751640e+01
9	Price	1.234657e+01
5	Content Rating	5.253944e-01
3	Free	2.147794e-01

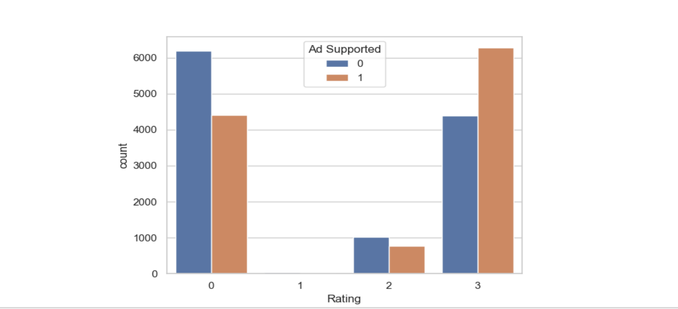
The best score is calculated while running the generated classification function (brute_force or auto_ml), and the scaler, model, and cv_k used at that time are returned.

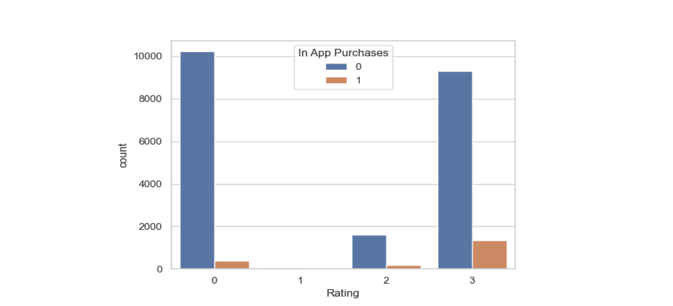
*** The Best Result through the AutoML**

	4 binned target	**2 binned target**
Scores	Best result • Scaler: MinMaxScaler() • Model: GradientBoostingClassifier() • CV: 10 Score • Model score: 92.93% • Weighted precision: 93.63% • Weighted recall: 93.02% • Weighted F1-score: 90.66%	Best result • Scaler: MaxAbsScaler() • Model: GradientBoostingClassifier() • CV: 2 Score • Model score: 95.55% • Weighted precision: 96% • Weighted recall: 96% • Weighted F1-score: 96%
ROC Curve		

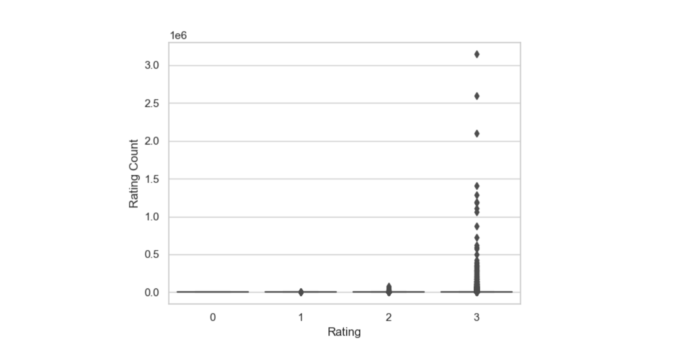
> ➢ **Result of analysis**

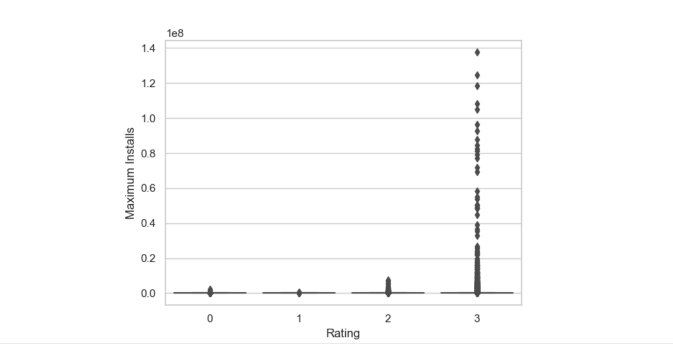
*** Relation with each feature (4-bin)**





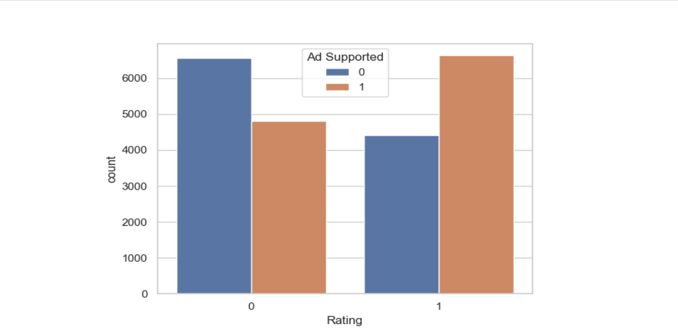
Ad support and In App Purchases are currently labeled as 0 for False and 1 for True. If you look at the graph above, you can see that the higher the rating, the greater the percentage of True.

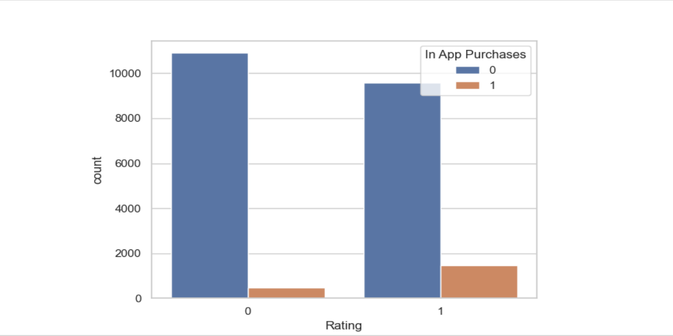




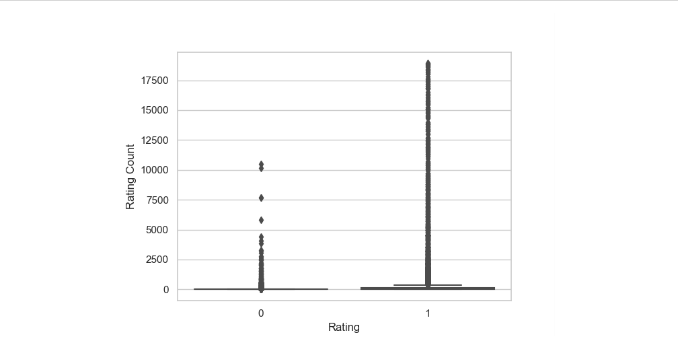
Due to the characteristics of the data, the values are clustered at 0 and there are excessively large values, so the plot is not conspicuous, but the rating count and Maximum Installs seem to increase as the rating increases.

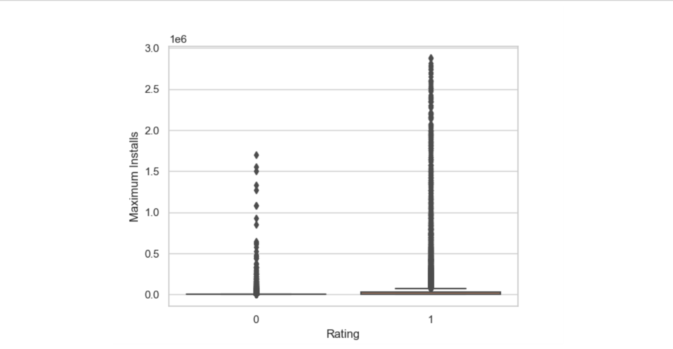
*** Relation with each feature (2-bin)**





Even when two bins were used, it was confirmed that the ratio of ‘Ad Supported’ and ‘In App Purchases’ to True increased more as the Rating increased.





The \`Rating Count\` and \`Maximum Installs\` feature’s result is very similar as 4-bin dataset’s result.

2) Clustering

➢ **Machine learning model what we used (4 models)**

- K-Means, Mean Shift, DBSCAN, and Expectation Maximization

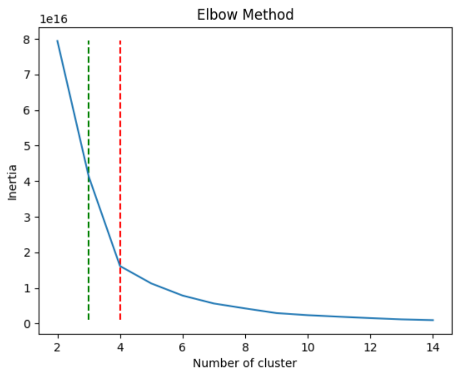
➢ **Dataset scaler what we used**

- None (No scale), Standard, Robust, MinMax, MaxAbs scaler

➢ **Model Assessment**

The best score is calculated while running the generated clustering function (brute_force or auto_ml), and the scaler, model, and cluster_k(the number of cluster) used at that time are returned. In case of the clustering models what is not to need to set clusters number such as DBSCAN, AutoML ignores the specified cluster_k parameter, and returns the cluster numbers what calculated by model.

We calculate and plot the elbow method to find the best cluster's number. From the left graph, we can make decision that 4 cluster is the proper number to analyze.



For calculating purity score, we fix the cluster number to 4 because of the target binning. Before we analyze the dataset, we binned the target features to 4 bins. This means, the answer is formed with 4 groups. This represents the intendancy that clustering algorithm might be make 4 clusters after fitting.

*** The Best Result through the AutoML (4-bin)**

Best result
- Scaler: MaxAbsScaler()
- Model: KMeans()
- Cluster: 4

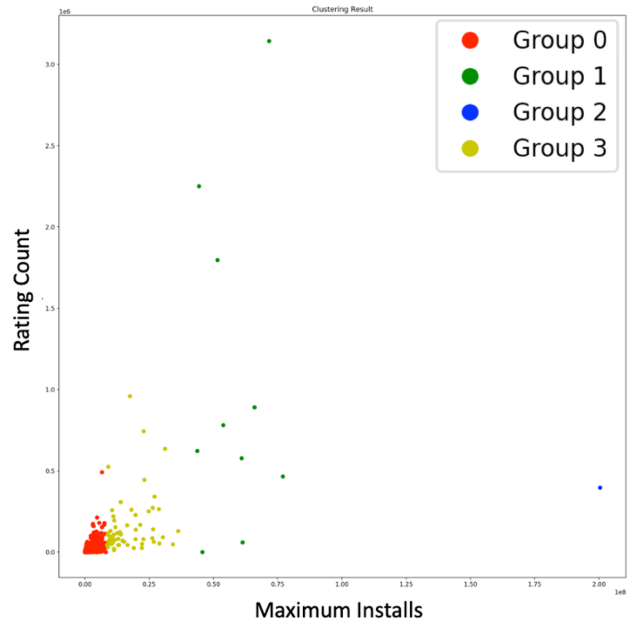
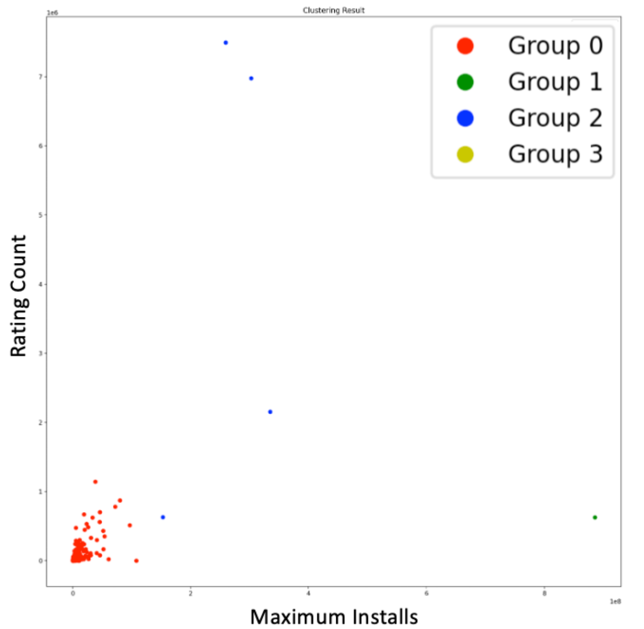
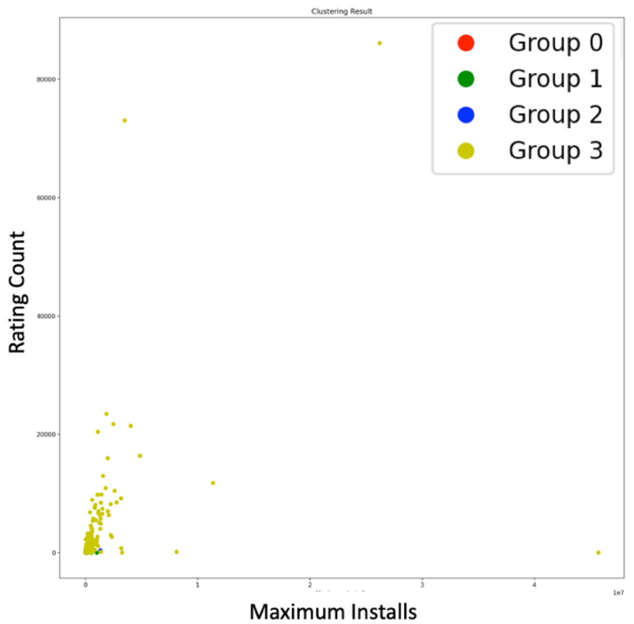
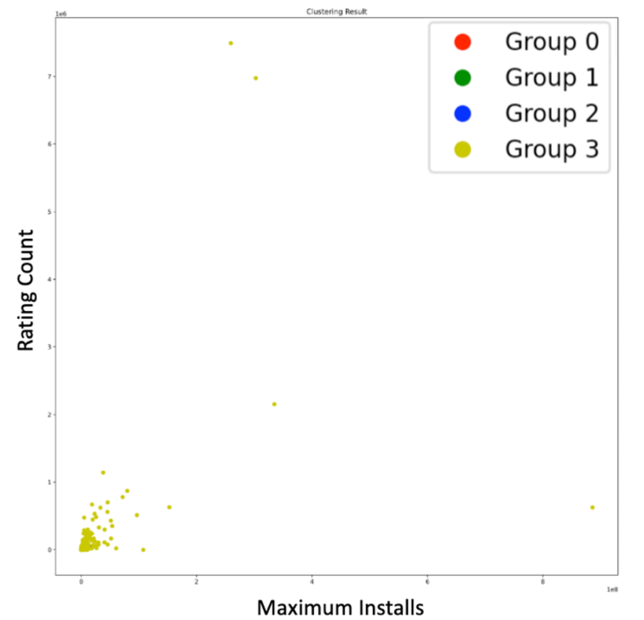
Score
- Silhouette score: 98.60%
- Purity score: 46.73%

*** 3-bin (Removed 0 rating)**

Best result
- Scaler: None
- Model: Gaussian Mixture(n_components=3)
- Cluster: 3

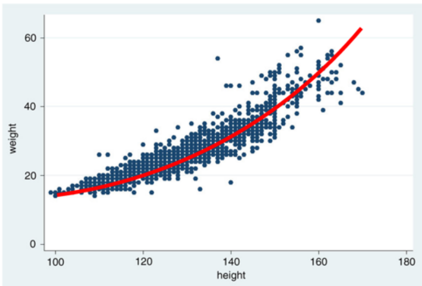
Score
- Silhouette score: 99.76%
- Purity score: 86.98%

*** Analyze of Clustering Model**

<table>
<tr><td>4-bin</td><td>3-bin (Removed 0 rating)</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>Clustering result</td><td>Clustering result</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>Actual target data</td><td>Actual target data</td></tr>
</table>

We find that clustering algorithm performed well through the silhouette score. But we cannot make good score of purity. We can guess that the dataset's actual target value (Rating) is not formed to make a cluster well.

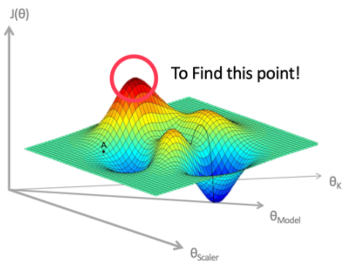
For example, the right graph shows that the relation between height and weight of humans. We can easily inference this data with polynomial regression to represent the data. But it is hard to cluster this data as n-clusters.



Before we use clustering algorithm, we need to analyze dataset's meaning and distributions what is proper to use clustering algorithm.

5. AutoML

- **Model to use**
  - We make AutoML for finding the best options automatically in very short time.
  - Our AutoML is based Stochastic Gradient Descent algorithm.



- **Parameter of AutoML**
  - **scalers, models, k** (the number of cluster or cross validation value): Array type parameter to create vector space.
  - In case of model parameter, we can assign the model what has specific hyperparameters
    - [LogisticRegression(solver="lbfgs", max_iter=100), LogisticRegression(solver="lbfgs", max_iter=200), LogisticRegression(solver="lbfgs", max_iter=300)]
    - If AutoML got the parameter like above, this is work as same as max_iter[100, 200, 300].
  - **thresh_score:** Default is None. If, algorithm find the score what is higher than thresh_score, then stop and terminate searching.
  - **max_iter:** This is meaning that how many iterations in searching loop.

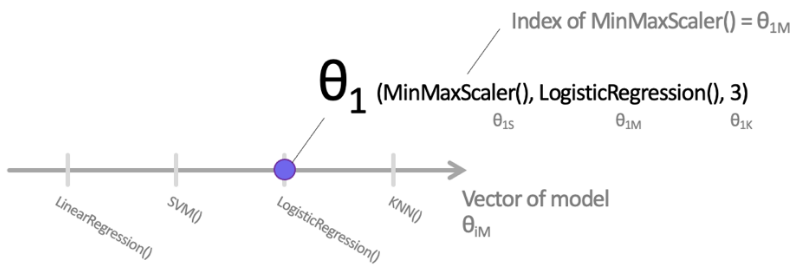
- **AutoML Algorithm's Step**

  0. Set initial θ_1 (point)
  1. Check the previous gradient value
  2. Calculate θ_1, θ_2 score ($J(\theta_1)$, $J(\theta_2)$)
  3. Calculate the gradient of $\Delta J(\theta) / \Delta\theta$
  4. Update the gradient of each vector and update θ_1
  5. Repeat 1~4 steps

- **Detailed Description about AutoML**

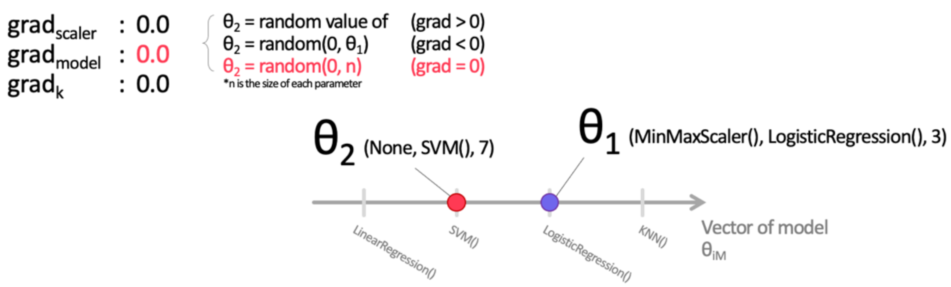
  0. Set initial θ_1 (point)

  Each θ is the point of parameters (scaler, model, k). These parameters have array data to find what is the best array index item of scaler, model, and k. We'll assume that each parameter(array)'s index is size of vector. For setting initial θ1 point, we pick random point of each parameter.



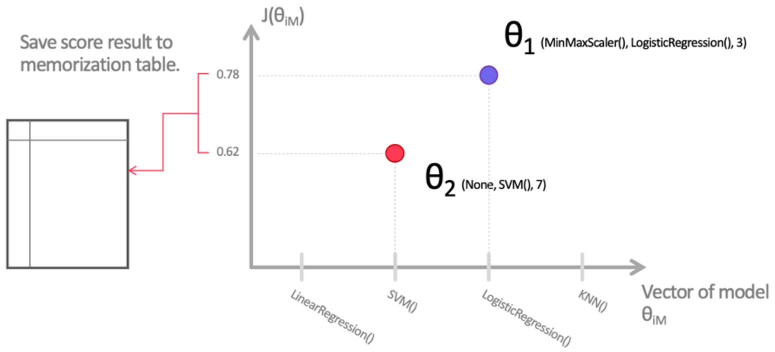
  1. Check the previous gradient value

  Calculate new θ2 coordinate with the formula as below.

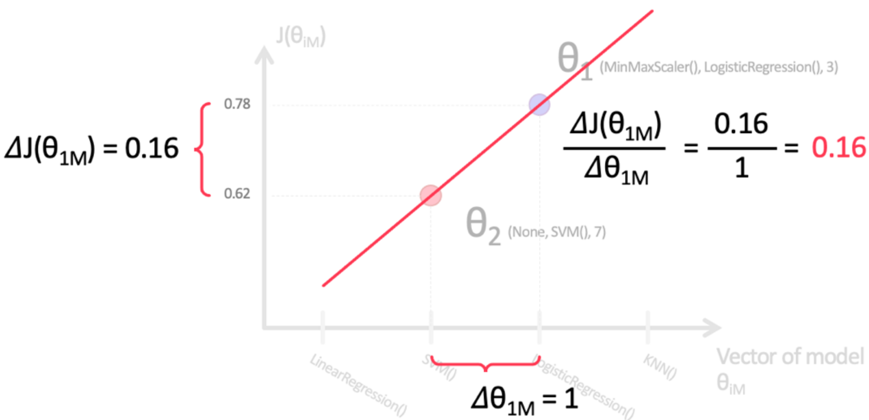


2. Calculate θ_1, θ_2 score ($J(\theta_1)$, $J(\theta_2)$)

Calculate score to make 1-Demention to 2-Demetion.



3. Calculate the gradient of $\Delta J(\theta) / \Delta\theta$



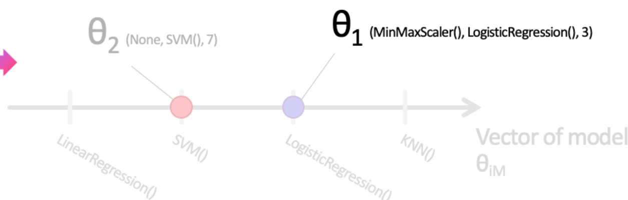
4. Update the gradient of each axis and θ_1

$grad_{scaler}$: 0.32
$grad_{model}$: 0.16
$grad_k$: -0.53

$$\frac{\Delta J(\theta_{1M})}{\Delta\theta_{1M}} = \frac{0.16}{1} = 0.16$$

$$\begin{cases} \theta_1 := \max(\theta_{1j}, \theta_{2j}) & (grad > 0) \\ \theta_1 := \min(\theta_{1j}, \theta_{2j}) & (grad < 0) \\ \theta_1 := \theta_1 & (grad = 0) \end{cases}$$

*J is the index of each parameter

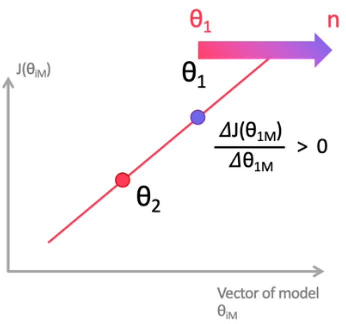


5. Repeat 1~4 steps

$grad_{scaler}$: 0.32
$grad_{model}$: 0.16
$grad_k$: -0.53

$$\begin{cases} \theta_2 = random(\theta_1, n) & (grad > 0) \\ \theta_2 = random(0, \theta_1) & (grad < 0) \\ \theta_2 = random(0, n) & (grad = 0) \end{cases}$$

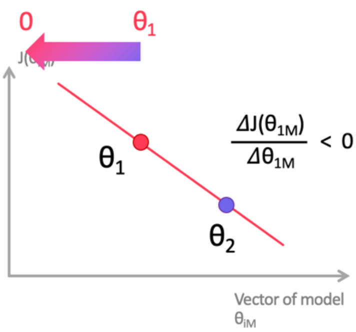
*n is the size of each parameter



$grad_{scaler}$: 0.32
$grad_{model}$: -0.16
$grad_k$: -0.53

$$\begin{cases} \theta_2 = random(\theta_1, n) & (grad > 0) \\ \theta_2 = random(0, \theta_1) & (grad < 0) \\ \theta_2 = random(0, n) & (grad = 0) \end{cases}$$

*n is the size of each parameter



Performance of AutoML

FBClassifier

- **60% Faster** than Brute force searching
- **14% Faster** than Randomize searching

```
# Classification Results

brute force (Search 36 times)
 - Time: 28.775557
 - Score: 0.762974507

Auto ML (10 iter)
 - Time: 7.2952278
 - Score: 0.763623

Auto ML (36 iter)
 - Time: 17.9441926
 - Score: 0.762628589

Auto ML (1000 iter)
 - Time: 25.8670026333
 - Score: 0.7633030802

Simple Random Search (1000 iter)
 - Time: 30.0984897
 - Score: 0.7633333651485362
```

FBClustering

- **246% Faster** than Brute force searching

```
# Clustering Results

brute force (Search 90 times)
 - Time: 352.4571505
 - Score: 0.9994110202825985

Auto ML (50 iter)(defalut)
 - Time: 143.940905
 - Score: 0.9994110202825985
```

Result

Decrease searching through stochastic gradient descent algorithm and **memorization** help to improve searching speed.

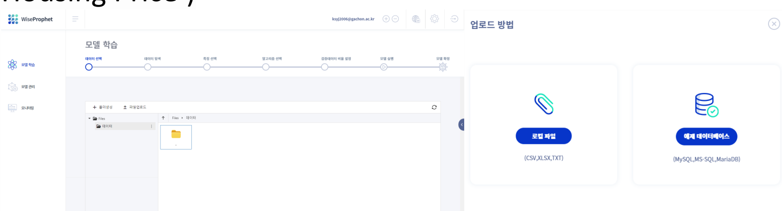
6. WiseProphet

We used WiseProphet. It is a web-based tool that can analyze and visualize data easily and quickly.

Analyzing Steps

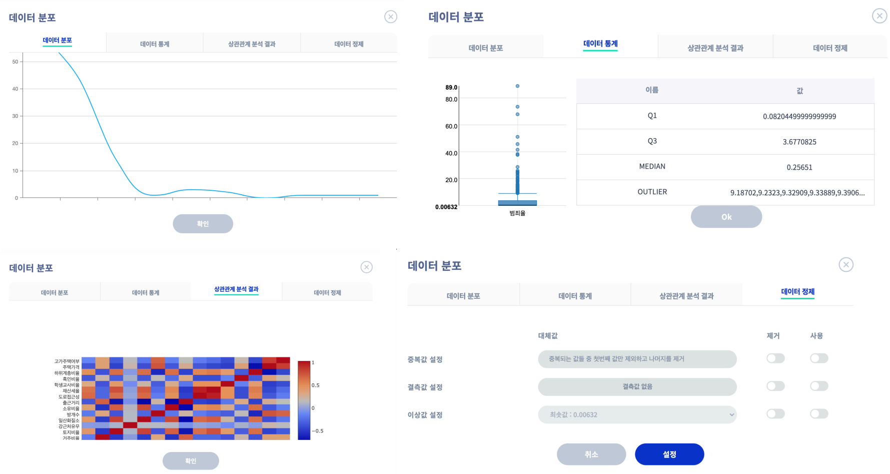
1) Load the Dataset

Left image is the Initial page, and then we can load the dataset. (In this example, we used 'Bosting Housing Price')



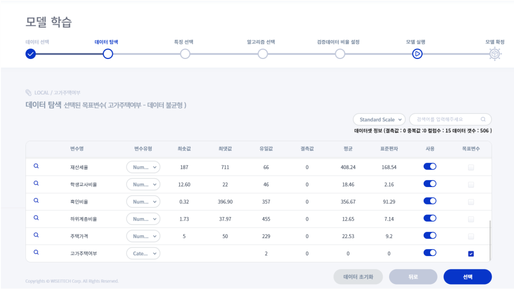
2) Data Exploration

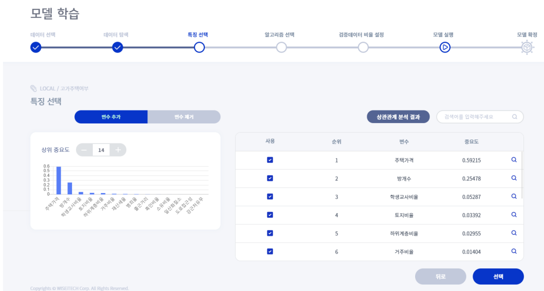
In this page, offers 4 functions to explore data. They are as follows.
(Data distribution, Data statistics, Data correlation, Data cleaning)



3) Feature Selection

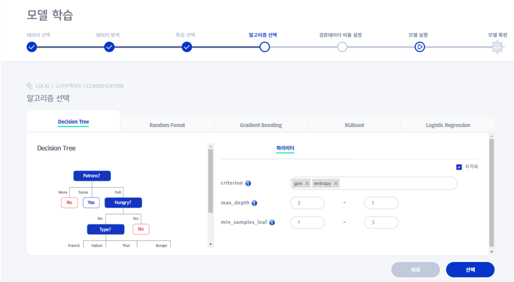
After checking the feature's details, and then select the features to use and the target.

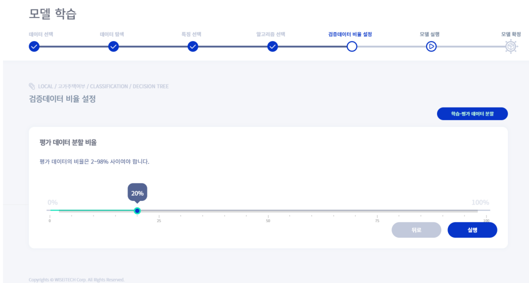




4) Algorithm Selection, Train-Test set split

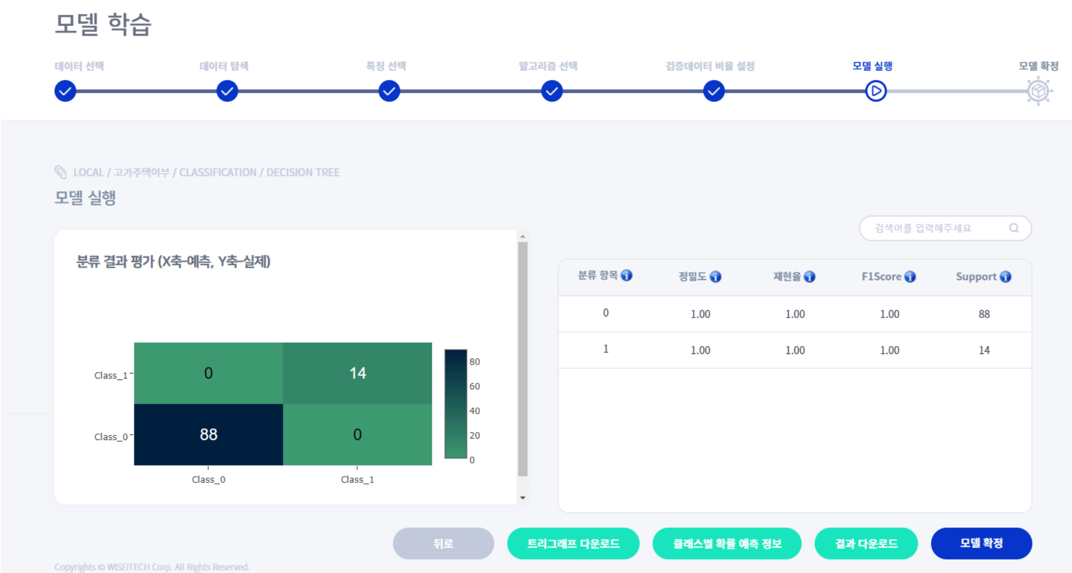
After selecting the features to use, also select the algorithms to use. In this example, the selected classification. And also, there are clustering algorithms and regression algorithms.





5) Execute the model

Every steps are done. Now wee can see the results and visualization. (Can download the result files – Constructed tree's visualization image, Dataset csv file with Actual & predicted values)



➢ Reviews (Strength & Weakness)

Just Clicking the Web-based GUI, the steps are done. After executing the model, the user can manage the created models in the management page.

Using this Tool, our team thought that it has clear pros & cons. First, People who don't know 'coding' can easily analyze data. Also, data visualization can be effectively performed. But it seems that the server that reads the data has not been optimized. That is, it cannot cover every dataset.

7. Distribution

➢ **Team member's Information & Role**

Dokyoon Kim
Implement Auto ML, Program Structure
uhug@gachon.ac.kr

Jaeuk Kim
Preprocessing, Analyze Model Result
ksyj2006@naver.com

Yoonsu La
Implement & Classification & Clustering
lys1@gachon.ac.kr

Soonwan Kwon
Clustering, Documentation
phsons@naver.com

➢ **Distributions (Total: 100%)**

- **김도균: 36 %**
- **라윤수: 32 %**
- **김재욱: 30 %**
- **권순완: 2 %**

8. Appendix

➢ **Github Repository of this project**

- https://github.com/GC212ML2/term-project
- We make documentation (pydoc) about AutoML in the FBClassifer.py, and FBClustering.py.

➢ **Github Organization of our team**

- https://github.com/GC212ML2

➢ **Program code**

· code.py

```
from preprocess import csv_to_dataframe
import FBClassifier
import FBClustering

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import MaxAbsScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import GradientBoostingClassifier

import pandas as pd
from timeit import default_timer as timer
from datetime import timedelta
from matplotlib import pyplot as plt
import seaborn as sns



# Import file and Preprocessing
df, dfs = csv_to_dataframe("./data/Google-Playstore.csv")
# dfs.drop(["index"], axis=1, inplace=True)



# Group again with 5 groups
#  - [Entertainment / Productivity / Lifestyle / Game / Education / Welfare]
category_le = LabelEncoder()
lbl_category = category_le.fit_transform(dfs['Category'])



# Bool
# Free, Ad Supported, In App Purchases, Editors Choice
free_le = LabelEncoder()
```

```
lbl_free = free_le.fit_transform(dfs['Free'])

ad_le = LabelEncoder()
lbl_ad = ad_le.fit_transform(dfs['Ad Supported'])

in_app_purchase_le = LabelEncoder()
lbl_in_app_purchase = in_app_purchase_le.fit_transform(dfs['In App Purchases'])

editors_choice_le = LabelEncoder()
lbl_editors_choice = editors_choice_le.fit_transform(dfs['Editors Choice'])

# Manual encoding for ordering
# ['Everyone', 'Teen', 'Adults']
# print(dfs['Content Rating'].drop_duplicates().tolist())
content_rating_le = LabelEncoder()
lbl_content_rating = content_rating_le.fit_transform(dfs['Content Rating'])
# print(dfs['Content Rating'])
# print(content_rating_le.classes_)



lbl_price = []
# ['Free', 'Low', 'Mid', 'High']
for i in dfs["Price"]:
# print(dfs['Price'].drop_duplicates().tolist())
    if i == "Free": lbl_price.append(0)
    elif i == "Low": lbl_price.append(1)
    elif i == "Mid": lbl_price.append(2)
    elif i == "High": lbl_price.append(3)

price_list_le = ['Free', 'Low', 'Mid', 'High']



dft = pd.DataFrame({
    "Category" : lbl_category,
    "Rating Count" : dfs["Rating Count"],
    "Maximum Installs" : dfs["Maximum Installs"],
    "Free" : lbl_free,
    "Last Updated" : dfs["Last Updated"],
    "Content Rating" : lbl_content_rating,
    "Ad Supported" : lbl_ad,
    "In App Purchases" : lbl_in_app_purchase,
    "Editors Choice" : lbl_editors_choice,
    "Price" : lbl_price,
    "Rating" : dfs["Rating"],
})

# # Print Label
# print(category_le.classes_)
# print(free_le.classes_)
# print(content_rating_le.classes_)
# print(in_app_purchase_le.classes_)
# print(editors_choice_le.classes_)
# print(price_list_le)

# Plot Heatmap
def heatmap(X, title):
    # Calculate correlation matrix and plot them
    plt.figure(figsize=(12,10))
    plt.title('Heatmap of ' + str(title), fontsize=20)
    g=sns.heatmap(X[X.corr().index].corr(), annot=True, cmap="YlGnBu")

    plt.show()

heatmap(dft, "Heatmap test")



# Split to predictor and predicted featrue
X = dft.drop(["Rating"], axis=1)
y = dft["Rating"]
print(X)
print(y)
print(dft.Rating.value_counts())



# Feature selection
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

bestfeatures = SelectKBest(score_func=chi2, k=len(dft.columns)-1)
fit = bestfeatures.fit(X,dft.Rating)
dfcolumns = pd.DataFrame(X.columns)
dfscores = pd.DataFrame(fit.scores_)

featureScores = pd.concat([dfcolumns,dfscores], axis=1)
featureScores.columns = ['Col', 'Score']
print(featureScores.nlargest(len(dft.columns)-1, 'Score'))
```

```
# # select top 4 best features
dft = dft[['Maximum Installs','Ad Supported','In App Purchases','Rating Count','Rating']]



# Training part
## Classification
classifier_result = FBClassifier.auto_ml(
    X,
    dft["Rating"],
    models=[
        DecisionTreeClassifier(criterion="gini"), DecisionTreeClassifier(criterion="entropy"),
        LogisticRegression(solver="lbfgs", max_iter=500, multi_class="ovr", class_weight='balanced'),
        LogisticRegression(solver="lbfgs", max_iter=1000, multi_class="ovr", class_weight='balanced'),
        GaussianNB(),
        GradientBoostingClassifier()
    ],
)

# Print the result
print(classifier_result.best_params)
print('best score :', classifier_result.best_score)
print(FBClassifier.clf_report(X, dft.Rating, classifier_result))
FBClassifier.plot_roc_curve(X, dft.Rating, classifier_result, classifier_result.best_model)



# Plot function for Clustering
def plot_grid(xlist, ylist, colors, title, xlabel, ylabel):
    plt.figure(figsize=(17,17))
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    group1x = []
    group1y = []
    group2x = []
    group2y = []
    group3x = []
    group3y = []
    group4x = []
    group4y = []

    for i in range(0, len(xlist)):
        color = 'ro'
        if colors[i] == 0:
            group1x.append(xlist[i])
            group1y.append(ylist[i])
        if colors[i] == 1:
            group2x.append(xlist[i])
            group2y.append(ylist[i])
        if colors[i] == 2:
            group3x.append(xlist[i])
            group3y.append(ylist[i])
        if colors[i] == 3:
            group4x.append(xlist[i])
            group4y.append(ylist[i])

    plt.plot(group1x, group1y, 'ro', label="Group 0")
    plt.plot(group2x, group2y, 'go', label="Group 1")
    plt.plot(group3x, group3y, 'bo', label="Group 2")
    plt.plot(group4x, group4y, 'yo', label="Group 3")

    plt.legend()
    plt.show()






# Training part
## Classification

# start = timer() # Ticker for performance test
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans
from sklearn.cluster import MeanShift
from sklearn.cluster import DBSCAN

clustering_result = FBClustering.auto_ml(
    X,
    cluster_k=[3],
    models=[
        KMeans(),
        GaussianMixture(),
        MeanShift(),
        DBSCAN()
```

```
    ],
    scalers=[
        None,
        StandardScaler(),
        RobustScaler(),
        MinMaxScaler(),
        MaxAbsScaler()
    ],
)

# end = timer() # Ticker for performance test
# print("Execution time :", timedelta(seconds=end-start)) # Ticker for performance test

print(clustering_result.best_params)
print(clustering_result.best_score)



# Convert for calculate clustering purity score
yt = []
ytt = y.tolist()
for i in range(0, len(ytt)):
    yt.append(ytt[i]-1)
yp = clustering_result.labels.tolist()

purity_result = FBClustering.purity_score(y_true=yt, y_pred=yp)

plot_grid(dft.iloc[:,0], dft.iloc[:,3], yp, "Clustering Result", "Maximum Installs", "Rating Count")
plot_grid(dft.iloc[:,0], dft.iloc[:,3], dft.iloc[:,4], "Clustering Result", "Maximum Installs", "Rating Count")
```

· **preprocess.py**

```
import pandas as pd
from scipy.sparse import data
from sklearn.model_selection import StratifiedShuffleSplit


def csv_to_dataframe(filename, columns=['App Id', 'Developer Website', 'Developer Email', 'Privacy Policy', 'Currency',
                                        'Developer Id', 'Scraped Time', 'Minimum Android']):
    # https://www.kaggle.com/gauthamp10/google-playstore-apps
    df = pd.read_csv(filename)

    if columns.count("Rating") != 0:
        columns.remove("Rating")
        print("Rating is target column, this cannot be deleted.")
        return

    """
    [Preprocessing]
    """
    ### 1. Drop the unnecessary columns
    df = df.drop(columns=columns)


    ### 2. Drop the dirty values
    # Fill NaN of Rating & RatingCount with mean
    df['Rating']  = df['Rating'].astype(float)
    df['Rating'].dropna(inplace=True)

    if df.columns.tolist().count("Rating Count") != 0:
        df['Rating Count']  = df['Rating Count'].astype(float)
        df['Rating Count'].dropna(inplace=True)

    # Replace the values of 'ContentRating'
    if df.columns.tolist().count("Content Rating") != 0:
        df['Content Rating'] = df['Content Rating'].replace('Unrated', "Everyone")
        df['Content Rating'] = df['Content Rating'].replace('Mature 17+', "Adults")
        df['Content Rating'] = df['Content Rating'].replace('Adults only 18+', "Adults")
        df['Content Rating'] = df['Content Rating'].replace('Everyone 10+', "Everyone")

    # Replace 'Installs' to convert numeric value
    if df.columns.tolist().count("Installs") != 0:
        df.Installs = df.Installs.str.replace(',', '')
        df.Installs = df.Installs.str.replace('+', '')
        df['Installs'] = pd.to_numeric(df['Installs'])

    # Replace 'LastUpdated' to YYYYMMDD format
    if df.columns.tolist().count("Last Updated") != 0:
        df = df.rename(columns={'Last Updated': 'LastUpdated'})
        df[['L1', 'L2', 'L3']] = pd.DataFrame(df.LastUpdated.str.split(' ', 3).tolist())

        df['L1'] = df['L1'].replace('Jan', '01')
        df['L1'] = df['L1'].replace('Feb', '02')
        df['L1'] = df['L1'].replace('Mar', '03')
        df['L1'] = df['L1'].replace('Apr', '04')
        df['L1'] = df['L1'].replace('May', '05')
        df['L1'] = df['L1'].replace('Jun', '06')
        df['L1'] = df['L1'].replace('Jul', '07')
        df['L1'] = df['L1'].replace('Aug', '08')
        df['L1'] = df['L1'].replace('Sep', '09')
        df['L1'] = df['L1'].replace('Oct', '10')
        df['L1'] = df['L1'].replace('Nov', '11')
        df['L1'] = df['L1'].replace('Dec', '12')

        df['L2'] = df['L2'].str.slice(start=0, stop=2)

        df['Last Updated'] = df['L3'] + df['L1'] + df['L2']

        df = df.drop(['LastUpdated', 'L1', 'L2', 'L3'], axis=1)

    """"
    Group again with 6 groups
    = Entertainment / Productivity / Lifestyle / Game / Education / Welfare
    """
    if df.columns.tolist().count("Category") != 0:
        # Productivity
        df['Category'] = df['Category'].replace('Productivity', 'Productivity')
        df['Category'] = df['Category'].replace('Tools', 'Productivity')
        df['Category'] = df['Category'].replace('Business', 'Productivity')

        # Lifestyle
        df['Category'] = df['Category'].replace('Social', 'Lifestyle')
        df['Category'] = df['Category'].replace('Food & Drink', 'Lifestyle')
        df['Category'] = df['Category'].replace('Auto & vehicles', 'Lifestyle')
        df['Category'] = df['Category'].replace('House & Home', 'Lifestyle')
        df['Category'] = df['Category'].replace('Shopping', 'Lifestyle')
        df['Category'] = df['Category'].replace('Maps & Navigation', 'Lifestyle')
        df['Category'] = df['Category'].replace('Weather', 'Lifestyle')
        df['Category'] = df['Category'].replace('Events', 'Lifestyle')
        df['Category'] = df['Category'].replace('Auto & Vehicles', 'Lifestyle')
        df['Category'] = df['Category'].replace('Communication', 'Lifestyle')
        df['Category'] = df['Category'].replace('Finance', 'Lifestyle')
```

```
    df['Category'] = df['Category'].replace('Books & Reference', 'Lifestyle')
    df['Category'] = df['Category'].replace('Libraries & Demo', 'Lifestyle')

    # Entertainment
    df['Category'] = df['Category'].replace('Music & Audio', 'Entertainment')
    df['Category'] = df['Category'].replace('Video Players & Editors', 'Entertainment')
    df['Category'] = df['Category'].replace('Personalization', 'Entertainment')
    df['Category'] = df['Category'].replace('News & Magazines', 'Entertainment')
    df['Category'] = df['Category'].replace('Travel & Local', 'Entertainment')
    df['Category'] = df['Category'].replace('Beauty', 'Entertainment')
    df['Category'] = df['Category'].replace('Photography', 'Entertainment')
    df['Category'] = df['Category'].replace('Art & Design', 'Entertainment')
    df['Category'] = df['Category'].replace('Dating', 'Entertainment')
    df['Category'] = df['Category'].replace('Comics', 'Entertainment')
    df['Category'] = df['Category'].replace('Sports', 'Entertainment')

    # Game
    # Based on https://www.appbrain.com/stats/android-market-app-categories
    df['Category'] = df['Category'].replace('Action', 'Game')
    df['Category'] = df['Category'].replace('Adventure', 'Game')
    df['Category'] = df['Category'].replace('Arcade', 'Game')
    df['Category'] = df['Category'].replace('Board', 'Game')
    df['Category'] = df['Category'].replace('Card', 'Game')
    df['Category'] = df['Category'].replace('Casino', 'Game')
    df['Category'] = df['Category'].replace('Casual', 'Game')
    df['Category'] = df['Category'].replace('Music', 'Game')
    df['Category'] = df['Category'].replace('Puzzle', 'Game')
    df['Category'] = df['Category'].replace('Racing', 'Game')
    df['Category'] = df['Category'].replace('Role Playing', 'Game')
    df['Category'] = df['Category'].replace('Simulation', 'Game')
    df['Category'] = df['Category'].replace('Strategy', 'Game')
    df['Category'] = df['Category'].replace('Trivia', 'Game')
    df['Category'] = df['Category'].replace('Word', 'Game')

    # Education
    df['Category'] = df['Category'].replace('Educational', 'Education')

    # Welfare
    df['Category'] = df['Category'].replace('Health & Fitness', 'Welfare')
    df['Category'] = df['Category'].replace('Medical', 'Welfare')
    df['Category'] = df['Category'].replace('Parenting', 'Welfare')

"""
Binning (Price) : There are 4 values (Free, Low, Mid, High)
x = Price
Free : 0 < x < 0.19
Low : 0.19 < x < 9.99
Mid : 9.99 < x < 29.99
High : 29.99 <= x < 410
"""
if df.columns.tolist().count("Price") != 0:
    bins = [-0.1, 0, 9.99, 29.99, 410]
    label = ['Free', 'Low', 'Mid', 'High']
    binning = pd.cut(df['Price'], bins, labels=label)
    df = df.drop('Price', axis=1)
    df['Price'] = binning



## Selection part of binning style of Rating

# """
# Binning (Rating) : There are values of 1-4
# x = Rating
# 1 : -0.1 <= x < 3.0
# 2 : 3.0 <= x < 5.1
# """
# bins = [-0.1, 3.0, 5.1]
# label = ['1', '2']

"""
Binning (Rating) : There are values of 1-4
x = Rating
1 : -0.1 <= x < 0.1
2 : 0.1 <= x < 1.66
3 : 1.66 <= x < 3.33
4 : 3.33 <= x < 5.1
"""
bins = [-0.1, 0.1, 1.66, 3.33, 5.1]
label = ['1', '2', '3', '4']

# """
# # Remove zero value
# # Binning (Rating) : There are values of 1-3
# x = Rating
# 1 : 0.1 <= x < 1.66
# 2 : 1.66 <= x < 3.33
# 3 : 3.33 <= x < 5.1
# """
# bins = [0.1, 1.66, 3.33, 5.1]
```

```
    # label = ['1', '2', '3']


    """
    Binning (Rating) : There are values of 1-4
    x = Rating
    1 : -0.1 <= x < 3.0
    2 : 3.0 <= x < 5.1
    """
    # bins = [-0.1, 3.0, 5.1]
    # label = ['1', '2']



    binning = pd.cut(df['Rating'], bins, labels=label)
    df = df.drop('Rating', axis=1)
    df['Rating'] = binning

    df = df.dropna(axis=0)
    df = df.reset_index()

    df = df.drop('index', axis=1)
    df = df.astype({'Rating': 'int64'})

    # Set the target to Rating
    target = ['Rating']
    X = df.drop(target, axis=1)
    y = df[target]



    # Stratified Shuffle sampling
    split = StratifiedShuffleSplit(n_splits=1, test_size=0.01, random_state=0)

    for train_idx, test_idx in split.split(X, y):
        dataframe = df.loc[train_idx]
        dataframe_sampling = df.loc[test_idx]

    dataframe = dataframe.reset_index()
    dataframe_sampling = dataframe_sampling.reset_index()

    return dataframe, dataframe_sampling
```

· **FBClassifier.py**

```
from math import nan
from matplotlib.pyplot import sca

import numpy as np
import pandas as pd
from pandas.core.frame import DataFrame

from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import MaxAbsScaler

from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

import random
import copy

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, roc_auc_score, auc, classification_report
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import label_binarize



def brute_force(
        X: DataFrame,
        y: DataFrame,
        scalers=[StandardScaler(), RobustScaler(), MinMaxScaler(), MaxAbsScaler()],
        models=[
            DecisionTreeClassifier(criterion="gini"), DecisionTreeClassifier(criterion="entropy"),
        ],
        cv_k=[2, 3, 4, 5, 6, 7, 8, 9, 10],
        is_cv_shuffle=True,
):
    """
    Brute Force Search
    ----------
    - Find the best parameter what has the best score.
    - This function use `Brute Force` method with memoization
    Parameters
    ----------
    - `X`: pandas.DataFrame
      - training dataset.
    - `y`: pandas.DataFrame
      - target value.
    - `scalers`: array
      - Scaler functions to scale data. This can be modified by user.
      - StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler as default.
    - `models`: array
      - Model functions to fitting data and prediction. This can be modified by user.
      - DecisionTreeClassifier(gini, entropy) as default with hyperparameters.
    - `cv_k`: array
      - Cross validation parameter. Default value is [2,3,4,5,6,7,8,9,10].
    - `is_cv_shuffle`
      - To set shuffle or not in cross validation
    Returns
    ----------
    - `best_params`: dictionary type of results.
    - `best_scaler`: Scaler what has best score.
    - `best_model`: Model what has best score.
    - `best_cv_k`: k value in K-fold CV what has best score.
    - `best_score`: double
      - Represent the score of the `best_params`.
    """

    # Initialize variables
    maxScore = -1.0
    best_scaler = None
    best_model = None
    best_cv_k_ = None

    # Find best scaler
    for n in range(0, len(scalers)):
        X = scalers[n].fit_transform(X)

        # Find best model
        for m in range(0, len(models)):
            # Find best k value of CV
            for i in range(0, len(cv_k)):

                kfold = KFold(n_splits=cv_k[i], shuffle=is_cv_shuffle)
                score_result = cross_val_score(models[m], X, y, scoring="accuracy", cv=kfold)

                # if mean value of scores are bigger than max variable,
                # update new options(model, scaler, k) to best options
                if maxScore < score_result.mean():
                    maxScore = score_result.mean()
```

```python
                    best_scaler = copy.deepcopy(scalers[n])
                    best_model = copy.deepcopy(models[m])
                    best_cv_k_ = copy.deepcopy(cv_k[i])

    class res:
        best_params = {}

    res.best_params = {
        'best_scaler': best_scaler,
        'best_model': best_model,
        'best_cv_k': best_cv_k_,
    }
    res.best_scaler = best_scaler
    res.best_model = best_model
    res.best_k = best_cv_k_

    res.best_score = maxScore

    # Return value with dictionary type
    return res


def plot_roc_curve(X, y, model, title):
    # for binary target
    if len(y.unique()) == 2:
        # Calculate False Positive Rate, True Positive Rate
        X = model.best_scaler.fit_transform(X)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
        clf = model.best_model
        y_pred = clf.fit(X_train, y_train).predict_proba(X_test)
        fpr, tpr, _ = roc_curve(y_test, y_pred[:, 1])
        roc_auc = roc_auc_score(y_test, y_pred[:, 1])

        # Plot result
        plt.figure(figsize=(12, 10))
        plt.plot(fpr, tpr, color='b', label='ROC curve (area = %0.2f)' % roc_auc)
        plt.plot([0, 1], [0, 1], color='r', linestyle='--')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve for ' + str(title), fontsize=20)
        plt.legend()
        plt.show()

    # for multiclass target
    else:
        # Calculate False Positive Rate, True Positive Rate
        X = model.best_scaler.fit_transform(X)
        y_unique, counts = np.unique(y, return_counts=True)
        y = label_binarize(y, classes=y_unique)
        n_classes = y.shape[1]
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

        clf = OneVsRestClassifier(model.best_model)
        y_pred = clf.fit(X_train, y_train).predict_proba(X_test)

        # Compute ROC curve and ROC area for each class
        fpr = dict()
        tpr = dict()
        roc_auc = dict()

        for i in range(n_classes):
            fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred[:, i])

        # First aggregate all false positive rates
        all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

        # Then interpolate all ROC curves at this points
        mean_tpr = np.zeros_like(all_fpr)
        for i in range(n_classes):
            mean_tpr += np.interp(all_fpr, fpr[i], tpr[i])

        # Finally average it and compute AUC
        mean_tpr /= n_classes

        fpr["macro"] = all_fpr
        tpr["macro"] = mean_tpr
        weighted_roc_auc = roc_auc_score(y_test, y_pred, multi_class="ovr", average="weighted")
        # Plot result
        plt.figure(figsize=(12, 10))
        plt.plot(fpr[0], tpr[0], linestyle='--', color='orange', label='Class 0 vs Rest')
        plt.plot(fpr[1], tpr[1], linestyle='--', color='green', label='Class 1 vs Rest')
        plt.plot(fpr[2], tpr[2], linestyle='--', color='cyan', label='Class 2 vs Rest')
        plt.plot(fpr[3], tpr[3], linestyle='--', color='yellow', label='Class 3 vs Rest')

        plt.plot(fpr["macro"], tpr["macro"], color='r', label='ROC curve (area = %0.2f)' % weighted_roc_auc)
        plt.plot([0, 1], [0, 1], color='black')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve for ' + str(title), fontsize=20)
```

```python
        plt.legend()
        plt.show()


def clf_report(X, y, model):
    X = model.best_scaler.fit_transform(X)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

    clf = model.best_model
    clf.fit(X_train, y_train)
    pred_test = clf.predict(X_test)
    report = classification_report(y_test, pred_test, zero_division=0)

    return report


def random_search(
        X: DataFrame,
        y: DataFrame,
        scalers=[StandardScaler(), RobustScaler(), MinMaxScaler(), MaxAbsScaler()],
        models=[
            DecisionTreeClassifier(criterion="gini"), DecisionTreeClassifier(criterion="entropy"),
        ],
        cv_k=[2, 3, 4, 5, 6, 7, 8, 9, 10],
        is_cv_shuffle=True,
        thresh_score=None,
        max_iter=50,
):
    """
    Random Search
    ----------
    - Find the best parameter what has the best score.
    - This function use `Random Search` method with memoization
    Parameters
    ----------
    - `X`: pandas.DataFrame
      - training dataset.
    - `y`: pandas.DataFrame
      - target value.
    - `scalers`: array
      - Scaler functions to scale data. This can be modified by user.
      - StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler as default.
    - `models`: array
      - Model functions to fitting data and prediction. This can be modified by user.
      - DecisionTreeClassifier(gini, entropy) as default with hyperparameters.
    - `cv_k`: array
      - Cross validation parameter. Default value is [2,3,4,5,6,7,8,9,10].
    - `is_cv_shuffle`
      - To set shuffle or not in cross validation
    - `thresh_score`
      - Default is None. If, algorithm find the score what is higher than thresh_score, then stop and terminate searching.
    - `max_iter`
      - Default is 100. This is meaning that how many iterations in searching loop.
    Returns
    ----------
    - `best_params_`: dictionary type of results.
    - `best_scaler`: Scaler what has best score.
    - `best_model`: Model what has best score.
    - `best_cv_k`: k value in K-fold CV what has best score.
    - `best_score`: double
      - Represent the score of the `best_params`.
    """

    # 0. Calculate length of each paramenter
    scalers_len = len(scalers)
    models_len = len(models)
    cv_k_len = len(cv_k)

    # 0. Initialize max score point
    max_scalers_idx = 0
    max_models_idx = 0
    max_cv_k_idx = 0
    max_score = 0

    # 0. Create memorize table for memoization
    mem_table = [[[0 for col in range(cv_k_len)] for row in range(models_len)] for col in range(scalers_len)]

    for trial in range(0, max_iter):
        # 0. Pick arbitrary point (theta1 = p1)
        scalers_idx = random.randrange(0, scalers_len)
        models_idx = random.randrange(0, models_len)
        cv_k_idx = random.randrange(0, cv_k_len)

        # 2. Calculate score(J(theta)) of each theta(point)
        score = 0

        # Check mem_table if score already has been calculated
        if mem_table[scalers_idx][models_idx][cv_k_idx] != 0:
            score = mem_table[scalers_idx][models_idx][cv_k_idx]
```

```
        else:
            # if not, calculate score of theta
            if scalers[scalers_idx] != None:
                p1_X = scalers[scalers_idx].fit_transform(X)
            else:
                p1_X = X
            kfold = KFold(n_splits=cv_k[cv_k_idx], shuffle=is_cv_shuffle)
            score = cross_val_score(models[models_idx], p1_X, y, cv=kfold).mean()
            # 2-1. Memoization
            mem_table[scalers_idx][models_idx][cv_k_idx] = score

        # Save point parameter what have best score
        if max_score < score:
            max_scalers_idx = scalers_idx
            max_models_idx = models_idx
            max_cv_k_idx = cv_k_idx
            max_score = score

        # If, score get higher score than thresh, terminate gradient searching
        if thresh_score != None and max_score > thresh_score: break

        # print("Trial: ", end="")
        # print(trial)
        # print(p1.scalers_idx)
        # print(p1.models_idx)
        # print(p1.cv_k_idx)
        # print()

    class res:
        best_params = {}

    res.best_params = {
        'best_scaler': scalers[max_scalers_idx],
        'best_model': models[max_models_idx],
        'best_cv_k': cv_k[max_cv_k_idx],
    }
    res.best_scaler = scalers[max_scalers_idx]
    res.best_model = models[max_models_idx]
    res.best_k = cv_k[max_cv_k_idx]

    res.best_score = max_score

    # Return value with dictionary type
    return res


def auto_ml(
    X:DataFrame,
    y:DataFrame,
    scalers=[StandardScaler(), RobustScaler(), MinMaxScaler(), MaxAbsScaler()],
    models=[
        DecisionTreeClassifier(criterion="gini"), DecisionTreeClassifier(criterion="entropy"),
    ],
    cv_k=[2,3,4,5,6,7,8,9,10],
    is_cv_shuffle = True,
    thresh_score = None,
    max_iter = 50,
):
    """
    Auto ML for Classifier
    ----------
    - Find the best parameter what has the best score.
    - This function use `Auto ML` method. This is similar to the Gradient Descent.
    - This function use memoization technique for faster calculation.
    Parameters
    ----------
    - `X`: pandas.DataFrame
      - training dataset.
    - `y`: pandas.DataFrame
      - target value.
    - `scalers`: array
      - Scaler functions to scale data. This can be modified by user.
      - StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler as default.
    - `models`: array
      - Model functions to fitting data and prediction. This can be modified by user.
      - DecisionTreeClassifier(gini, entropy) as default with hyperparameters.
    - `cv_k`: array
      - Cross validation parameter. Default value is [2,3,4,5,6,7,8,9,10].
    - `is_cv_shuffle`
      - To set shuffle or not in cross validation
    - `thresh_score`
      - Default is None. If, algorithm find the score what is higher than thresh_score, then stop and terminate searching.
    - `max_iter`
      - Default is 50. This is meaning that how many iterations in searching loop.
    Returns
    ----------
    - `best_params`: dictionary type of results.
    - `best_scaler`: Scaler what has best score.
    - `best_model`: Model what has best score.
    - `best_cv_k`: k value in K-fold CV what has best score.
```

```python
    - `best_score`: double
      - Represent the score of the `best_params`.
    """

    logo()

    # 0. Calculate length of each paramenter
    scalers_len = len(scalers)
    models_len = len(models)
    cv_k_len = len(cv_k)

    # 0. Create memorize table for memoization
    mem_table = [[[0 for col in range(cv_k_len)] for row in range(models_len)] for col in range(scalers_len)]

    # 0. Create point(theta) vector class
    class Point():
        scalers_idx = 0
        models_idx = 0
        cv_k_idx = 0

    # 0. Initialize gradient value
    gradient_theta1 = 0
    gradient_theta2 = 0
    gradient_theta3 = 0

    # 0. Pick arbitrary point (theta1 = p1)
    p1 = Point()
    p1.scalers_idx = random.randrange(0, scalers_len)
    p1.models_idx = random.randrange(0, models_len)
    p1.cv_k_idx = random.randrange(0, cv_k_len)

    # 0. Initialize max score point
    max_scalers_idx = 0
    max_models_idx = 0
    max_cv_k_idx = 0
    max_score = 0

    for trial in range(0, max_iter):

        # 1. Check previous gradient value of each theta
        #    and pick another arbitrary point (theta = p2)

        def check_gradient(target_gradient_theta, point_val, max_len):
            result = 0
            if target_gradient_theta > 0 and point_val + 1 != max_len:
                # if point_val+1 == max_len => out of range
                # then, get arbitrary point from 0 to len(target)
                result = random.randrange(point_val + 1, max_len)

            elif target_gradient_theta < 0 and point_val != 0:
                # if point_val == 0 => out of range
                # then, get arbitrary point from 0 to len(target)
                result = random.randrange(0, point_val)

            else:
                result = random.randrange(0, max_len)

            return result

        p2 = Point()
        p2.scalers_idx = check_gradient(gradient_theta1, p1.scalers_idx, scalers_len)
        p2.models_idx = check_gradient(gradient_theta2, p1.models_idx, models_len)
        p2.cv_k_idx = check_gradient(gradient_theta3, p1.cv_k_idx, cv_k_len)

        # 2. Calculate score(J(theta)) of each theta(point)
        p1_score = 0
        p2_score = 0

        # Check mem_table if score already has been calculated
        if mem_table[p1.scalers_idx][p1.models_idx][p1.cv_k_idx] != 0:
            p1_score = mem_table[p1.scalers_idx][p1.models_idx][p1.cv_k_idx]
        else:
            # if not, calculate score of theta
            if scalers[p1.scalers_idx] != None:
                p1_X = scalers[p1.scalers_idx].fit_transform(X)
            else:
                p1_X = X
            kfold = KFold(n_splits=cv_k[p1.cv_k_idx], shuffle=is_cv_shuffle)
            p1_score = cross_val_score(models[p1.models_idx], p1_X, y, cv=kfold).mean()
            # 2-1. Memoization
            mem_table[p1.scalers_idx][p1.models_idx][p1.cv_k_idx] = p1_score

        if mem_table[p2.scalers_idx][p2.models_idx][p2.cv_k_idx] != 0:
            p2_score = mem_table[p2.scalers_idx][p2.models_idx][p2.cv_k_idx]
        else:
            if scalers[p1.scalers_idx] != None:
                p2_X = scalers[p2.scalers_idx].fit_transform(X)
            else:
                p2_X = X
            kfold = KFold(n_splits=cv_k[p2.cv_k_idx], shuffle=is_cv_shuffle)
            p2_score = cross_val_score(models[p2.models_idx], p2_X, y, cv=kfold).mean()
```

```python
            # 2-1. Memoization
            mem_table[p2.scalers_idx][p2.models_idx][p2.cv_k_idx] = p2_score

            # Save point parameter what have best score
        if p1_score > p2_score:
            if max_score < p1_score:
                max_scalers_idx = p1.scalers_idx
                max_models_idx = p1.models_idx
                max_cv_k_idx = p1.cv_k_idx
                max_score = p1_score

        if p1_score < p2_score:
            if max_score < p2_score:
                max_scalers_idx = p2.scalers_idx
                max_models_idx = p2.models_idx
                max_cv_k_idx = p2.cv_k_idx
                max_score = p2_score

        # If, score get higher score than thresh, terminate gradient searching
        if thresh_score != None and max_score > thresh_score: break


        # 3. Calcuate gradient of each theta(point).
        #    with using above theta value, set another theta(point).
        change_of_cost = p2_score - p1_score
        change_of_theta1 = p2.scalers_idx - p1.scalers_idx
        change_of_theta2 = p2.models_idx - p1.models_idx
        change_of_theta3 = p2.cv_k_idx - p1.cv_k_idx

        # If, attribute of theta1 and theta2 are same, set gradient value to 0 (slope = 0)
        def update_gradient_value(change_of_cost, change_of_theta):
            result_gradient = 0
            if change_of_theta != 0:
                result_gradient = change_of_cost / change_of_theta
            return result_gradient

        gradient_theta1 = update_gradient_value(change_of_cost, change_of_theta1)
        gradient_theta2 = update_gradient_value(change_of_cost, change_of_theta2)
        gradient_theta3 = update_gradient_value(change_of_cost, change_of_theta3)


        # 4. Prepare for next gradient (change theta 1 to new position)
        def set_new_point(gradient_theta, compare1, compare2):
            result_idx = 0
            if gradient_theta > 0:
                result_idx = max([compare1, compare2])
            elif gradient_theta < 0:
                result_idx = min([compare1, compare2])
            else:
                result_idx = compare1
            return result_idx

        # Set new theta1 for the next calculation
        p1.scalers_idx = set_new_point(gradient_theta1, p1.scalers_idx, p2.scalers_idx)
        p1.models_idx = set_new_point(gradient_theta2, p1.models_idx, p2.models_idx)
        p1.cv_k_idx = set_new_point(gradient_theta3, p1.cv_k_idx, p2.cv_k_idx)

    class res:
        best_params = {}

    res.best_params = {
        'best_scaler': scalers[max_scalers_idx],
        'best_model': models[max_models_idx],
        'best_cv_k': cv_k[max_cv_k_idx],
    }
    res.best_scaler = scalers[max_scalers_idx]
    res.best_model = models[max_models_idx]
    res.best_k = cv_k[max_cv_k_idx]

    res.best_score = max_score

    # Return value with dictionary type
    return res


def logo():
    print("")
    print("        /‾‾‾‾‾‾\      /‾‾‾\   /‾‾‾/\/‾‾‾‾‾‾‾‾‾‾‾\  /‾‾‾‾‾‾‾‾‾‾‾‾\       /‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾\  /‾‾‾‾\      ")
    print("       /        \    /    /\  /    / /            /\/             /\     /                  /\/    /\     ")
    print("      /    /\    \  /    /  /    / /\‾‾/    /\‾‾\ /    /‾‾‾/    / /    /    /‾/    /‾/    / /    / /     ")
    print("     /    /_/   /\/    /  /    / / ‾ /    / /‾‾‾/    /    /    / /    /    / /    / /    / /    / /      ")
    print("    /    _     / /    /__/    / /   /    / /   /    /___/    / /    /    / /    / /    / /    /_/____    ")
    print("   /    /‾/    / /           / /   /    / /   /             / /    /    / /    / /    / /          /\   ")
    print("  /____/ /____/ /___________/ /   /____/ /   /____________/ /    /____/ /____/ /____/ /___________/ /   ")
    print("  \____\ \____\/\___________\/    \____\/    \____________\/     \____\ \____\ \____\/ \___________\/    ")
    print("                                                                  for Classifier / Version: 2021.11.17 ")
```

· **FBClustering.py**

```
import copy
import numpy as np
import pandas as pd
from pandas.core.frame import DataFrame
from sklearn import metrics

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from pyclustering.cluster.clarans import clarans as CLARANS
from sklearn.cluster import DBSCAN
from sklearn.cluster import MeanShift, estimate_bandwidth

from sklearn.metrics import silhouette_score
import random


# Change CLARANS result to ScikitLearn result
def clarans_label_converter(labels):
    total_len = 0
    for k in range(0, len(labels)):
        total_len += len(labels[k])

    outList = np.empty((total_len), dtype=int)
    cluster_number = 0
    for k in range(0, len(labels)):
        for l in range(0, len(labels[k])):
            outList[labels[k][l]] = cluster_number
        cluster_number += 1
    return outList


# Scoring function through purity check formula
def purity_score(y_true, y_pred):
  # compute contingency matrix (also called confusion matrix)
  contingency_matrix = metrics.cluster.contingency_matrix(y_true, y_pred)
  return np.sum(np.amax(contingency_matrix, axis=0)) / np.sum(contingency_matrix)





def brute_force(
    X:DataFrame,
    scalers=[None, StandardScaler()],
    models=[
        KMeans(n_clusters = 2),
        DBSCAN(eps=0.5, min_samples=5)
    ],
    cluster_k = [2,3,4,5,6,7,8,9,10],
):
    """
    Brute Force Search
    ----------
    - Find the best parameter what has the best score.
    - This function use Silhouette score for scoring cluster models.
    Parameters
    ----------
    - `X`: pandas.DataFrame
      - training dataset.
    - `scalers`: array
      - Scaler functions to scale data. This can be modified by user.
      - `None, StandardScaler()` as default
      - This parameter is compatible with `StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler`.
    - `models`: array
      - Model functions to clustering data. This can be modified by user.
      - KMeans, GaussianMixture, DBSCAN(eps=0.5, min_samples=5) as default with hyperparameters.
      - This parameter is compatible with `KMeans, DBSCAN`.
    - `cluster_k`: array
      - The umber of cluster. Default value is [2,3,4,5,6,7,8,9,10].
      - This can be modified by user.
    Returns
    ----------
    - `best_params`: dictionary
      - Dictionary data what has the information of below.
    - `best_scaler`: Scaler what has best silhouette score.
    - `best_model`: Model what has best silhouette score.
    - `best_k`: Best number of clusters
    - `best_score`: double
      - Represent the silhouette score of the `best_params`.
    - `labels`: List
      - The label information of cluster.
    Examples
    ----------
    result = FBClustering.brute_force(
      df,
```

```
    models=[
      CLARANS(data=df.to_numpy(), number_clusters=1, numlocal=2, maxneighbor=3),
      GaussianMixture(),
      KMeans(),
      DBSCAN(eps=0.5, min_samples=5),
      MeanShift(bandwidth=bandwidth)
    ],
    scalers=[None,], #StandardScaler(), RobustScaler(), MinMaxScaler(), MaxAbsScaler()
    cluster_k = range(2,11)
  )
  # Extract results
  labels = result['labels']
  best_score = result['best_score']
  result = result['best_params']
  best_scaler = result['best_scaler']
  best_model = result['best_model']
  best_k = result['best_k']
  # Print the result of best option
  print("\nBest Scaler: ", end="")
  print(best_scaler)
  print("Best Model: ", end="")
  print(best_model)
  print("Score: ", end="")
  print(best_score)
  print("labels: ", end="")
  print(labels)
  print("k: ", end="")
  print(best_k)
  """

  # Initialize variables
  maxScore = -1.0
  best_scaler = None
  best_model = None
  labels_ = None
  best_k_ = None

  curr_case = 1
  total_case = len(scalers) * len(models) * len(cluster_k)

  # Find best scaler
  for n in range(0, len(scalers)):
      if (scalers[n] != None):
          X = scalers[n].fit_transform(X)

      # Find best model
      for m in range(0, len(models)):

          # Scan once for DBSCAN
          isScaned = False

          # Find best k value of CV
          for i in range(0, len(cluster_k)):
              print("Progressing: (", end="")
              print(curr_case, end="/")
              print(total_case, end=")\n")
              curr_case += 1

              # model fitting
              models[m].n_clusters = cluster_k[i]  # for k-Means
              models[m].n_components = cluster_k[i]  # for Gaussian Mixture

              labels = None
              # calculate silhouette score
              if type(models[m]) == type(CLARANS(X, 1, 0, 0)):
                  models[m] = copy.deepcopy(CLARANS(
                      data=X.to_numpy(),
                      number_clusters=cluster_k[i],  # CLARANS cluster number setting
                      numlocal=models[m].__dict__['_clarans__numlocal'],
                      maxneighbor=models[m].__dict__['_clarans__maxneighbor']
                  ))
                  models[m].process()
                  clarans_label = models[m].get_clusters()
                  labels = clarans_label_converter(labels=clarans_label)

                  score_result = silhouette_score(X, labels)

              elif type(models[m]) == type(DBSCAN()) or type(models[m]) == type(MeanShift()):
                  if isScaned == True:
                      continue

                  isScaned = True
                  labels = models[m].fit_predict(X)

                  # when cluster nuber is just 1, skip scoring
                  gen_cluster_k = len(pd.DataFrame(labels).drop_duplicates().to_numpy().flatten())
                  if gen_cluster_k <= 1:
                      continue
                  score_result = silhouette_score(X, labels)

              else:
```

```
                labels = models[m].fit_predict(X)
                score_result = silhouette_score(X, labels)

            # if mean value of scores are bigger than max variable,
            # update new options(model, scaler, k) to best options
            if maxScore < score_result:
                maxScore = score_result
                best_scaler = copy.deepcopy(scalers[n])
                best_model = copy.deepcopy(models[m])
                best_k_ = cluster_k[i]
                # Calculated by DBSCAN
                if type(best_model) == type(DBSCAN()) or type(best_model) == type(
                    MeanShift()): best_k_ = gen_cluster_k
                labels_ = copy.deepcopy(labels)

    class res:
        best_params = {}

    res.best_params = {
        'best_scaler': best_scaler,
        'best_model': best_model,
        'best_k': best_k_,
    }
    res.best_scaler = best_scaler
    res.best_model = best_model
    res.best_k = best_k_

    res.best_score = maxScore
    res.labels = labels_

    # Return value with dictionary type
    return res




def auto_ml(
    X:DataFrame,
    scalers=[None, StandardScaler()],
    models=[
        KMeans(n_clusters = 2),
        DBSCAN(eps=0.5, min_samples=5),
    ],
    cluster_k = [2,3,4,5,6,7,8,9,10],
    thresh_score = None,
    max_iter = 50,
):
    """
    Auto ML for Clustering
    ----------
    - Find the best parameter what has the best score.
    - This function use `Auto ML` method. This is similar to the Gradient Descent.
    - This function use memoization technique for faster calculation.
    - This function use Silhouette score for scoring cluster models.
    Parameters
    ----------
    - `X`: pandas.DataFrame
      - training dataset.
    - `scalers`: array
      - Scaler functions to scale data. This can be modified by user.
      - `None, StandardScaler()` as default
      - This parameter is compatible with `StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler`.
    - `models`: array
      - Model functions to clustering data. This can be modified by user.
      - KMeans, GaussianMixture, DBSCAN(eps=0.5, min_samples=5) as default with hyperparameters.
      - This parameter is compatible with `KMeans, DBSCAN`.
    - `cluster_k`: array
      - The number of cluster. Default value is [2,3,4,5,6,7,8,9,10].
      - This can be modified by user.
    - `thresh_score`: float
      - Default is None. If, algorithm find the score what is higher than thresh_score, then stop and terminate searching.
    - `max_iter`: integer
      - Default is 50. This is meaning that how many iterations in searching loop.
    Returns
    ----------
    - `best_params`: dictionary
      - Dictionary data what has the information of below.
    - `best_scaler`: Scaler what has best silhouette score.
    - `best_model`: Model what has best silhouette score.
    - `best_k`: Best number of clusters
    - `best_score`: double
      - Represent the silhouette score of the `best_params`.
    - `labels`: List
      - The label information of cluster.
    Examples
    ----------
    result = FBClustering.auto_ml(
      df,
      models=[
```

```
    CLARANS(data=df.to_numpy(), number_clusters=1, numlocal=2, maxneighbor=3),
    GaussianMixture(),
    KMeans(),
    DBSCAN(eps=0.5, min_samples=5),
    MeanShift(bandwidth=bandwidth)
  ],
  scalers=[None,], #StandardScaler(), RobustScaler(), MinMaxScaler(), MaxAbsScaler()
  cluster_k = range(2,11)
)
# Extract results
labels = result['labels']
best_score = result['best_score']
result = result['best_params']
best_scaler = result['best_scaler']
best_model = result['best_model']
best_k = result['best_k']
# Print the result of best option
print("\nBest Scaler: ", end="")
print(best_scaler)
print("Best Model: ", end="")
print(best_model)
print("Score: ", end="")
print(best_score)
print("labels: ", end="")
print(labels)
print("k: ", end="")
print(best_k)
"""

logo()

# 0. Calculate length of each paramenter
scalers_len = len(scalers)
models_len = len(models)
cluster_k_len = len(cluster_k)

# 0. Create memorize table for memoization
mem_table = [[[0 for col in range(cluster_k_len)] for row in range(models_len)] for col in range(scalers_len)]

# 0. Create point(theta) vector class
class Point():
    scalers_idx = 0
    models_idx = 0
    cluster_k_idx = 0

# 0. Initialize gradient value
gradient_theta1 = 0
gradient_theta2 = 0
gradient_theta3 = 0

# 0. Pick arbitrary point (theta1 = p1)
p1 = Point()
p1.scalers_idx = random.randrange(0, scalers_len)
p1.models_idx = random.randrange(0, models_len)
p1.cluster_k_idx = random.randrange(0, cluster_k_len)

# 0. Initialize max score point
max_scalers_idx = 0
max_models_idx = 0
max_cluster_k_idx = 0
max_score = 0
best_labels = None

for trial in range(0, max_iter):

    # 1. Check previous gradient value of each theta
    #    and pick another arbitrary point (theta = p2)

    def check_gradient(target_gradient_theta, point_val, max_len):

        result = 0
        if target_gradient_theta > 0 and point_val + 1 != max_len:
            # if point_val+1 == max_len => out of range
            # then, get arbitrary point from 0 to len(target)
            result = random.randrange(point_val + 1, max_len)

        elif target_gradient_theta < 0 and point_val != 0:
            # if point_val == 0 => out of range
            # then, get arbitrary point from 0 to len(target)
            result = random.randrange(0, point_val)

        else:
            result = random.randrange(0, max_len)

        return result

    p2 = Point()
    p2.scalers_idx = check_gradient(gradient_theta1, p1.scalers_idx, scalers_len)
    p2.models_idx = check_gradient(gradient_theta2, p1.models_idx, models_len)
    p2.cluster_k_idx = check_gradient(gradient_theta3, p1.cluster_k_idx, cluster_k_len)
```

```
        # 2. Calculate score(J(theta)) of each theta(point)
        p1_score = 0
        p2_score = 0
        labels = None

        # Check mem_table if score already has been calculated
        if mem_table[p1.scalers_idx][p1.models_idx][p1.cluster_k_idx] != 0:
            p1_score = mem_table[p1.scalers_idx][p1.models_idx][p1.cluster_k_idx]
        else:
            # model fitting
            models[p1.models_idx].n_clusters = cluster_k[p1.cluster_k_idx]  # for k-Means
            models[p1.models_idx].n_components = cluster_k[p1.cluster_k_idx]  # for Gaussian Mixture

            # calculate silhouette score
            if type(models[p1.models_idx]) == type(CLARANS(X, 1, 0, 0)):
                models[p1.models_idx] = copy.deepcopy(CLARANS(
                    data=X.to_numpy(),
                    number_clusters=cluster_k[p1.cluster_k_idx],  # CLARANS cluster number setting
                    numlocal=models[p1.models_idx].__dict__['_clarans__numlocal'],
                    maxneighbor=models[p1.models_idx].__dict__['_clarans__maxneighbor']
                ))
                models[p1.models_idx].process()
                clarans_label = models[p1.models_idx].get_clusters()
                labels = clarans_label_converter(labels=clarans_label)

                p1_score = silhouette_score(X, labels)

            elif type(models[p1.models_idx]) == type(DBSCAN()) or type(models[p1.models_idx]) == type(MeanShift()):

                labels = models[p1.models_idx].fit_predict(X)

                # when cluster nuber is just 1, skip scoring
                gen_cluster_k = len(pd.DataFrame(labels).drop_duplicates().to_numpy().flatten())
                if gen_cluster_k <= 1:
                    p1_score = -1
                else:
                    p1_score = silhouette_score(X, labels)

            else:
                labels = models[p1.models_idx].fit_predict(X)
                p1_score = silhouette_score(X, labels)

            # 2-1. Memoization
            mem_table[p1.scalers_idx][p1.models_idx][p1.cluster_k_idx] = p1_score

        if mem_table[p2.scalers_idx][p2.models_idx][p2.cluster_k_idx] != 0:
            p2_score = mem_table[p2.scalers_idx][p2.models_idx][p2.cluster_k_idx]
        else:
            # model fitting
            models[p2.models_idx].n_clusters = cluster_k[p2.cluster_k_idx]  # for k-Means
            models[p2.models_idx].n_components = cluster_k[p2.cluster_k_idx]  # for Gaussian Mixture

            # calculate silhouette score
            if type(models[p2.models_idx]) == type(CLARANS(X, 1, 0, 0)):
                models[p2.models_idx] = copy.deepcopy(CLARANS(
                    data=X.to_numpy(),
                    number_clusters=cluster_k[p2.cluster_k_idx],  # CLARANS cluster number setting
                    numlocal=models[p2.models_idx].__dict__['_clarans__numlocal'],
                    maxneighbor=models[p2.models_idx].__dict__['_clarans__maxneighbor']
                ))
                models[p2.models_idx].process()
                clarans_label = models[p2.models_idx].get_clusters()
                labels = clarans_label_converter(labels=clarans_label)

                p2_score = silhouette_score(X, labels)

            elif type(models[p2.models_idx]) == type(DBSCAN()) or type(models[p2.models_idx]) == type(MeanShift()):

                labels = models[p2.models_idx].fit_predict(X)

                # when cluster nuber is just 1, skip scoring
                gen_cluster_k = len(pd.DataFrame(labels).drop_duplicates().to_numpy().flatten())
                if gen_cluster_k <= 1:
                    p1_score = -1
                else:
                    p2_score = silhouette_score(X, labels)

            else:
                labels = models[p2.models_idx].fit_predict(X)
                p2_score = silhouette_score(X, labels)

            # 2-1. Memoization
            mem_table[p2.scalers_idx][p2.models_idx][p2.cluster_k_idx] = p2_score

        # Save point parameter what have best score
        if p1_score > p2_score:
            if max_score < p1_score:
                max_scalers_idx = p1.scalers_idx
                max_models_idx = p1.models_idx
                max_cluster_k_idx = p1.cluster_k_idx
                max_score = p1_score
```

```python
                best_labels = copy.deepcopy(labels)

        if p1_score < p2_score:
            if max_score < p2_score:
                max_scalers_idx = p2.scalers_idx
                max_models_idx = p2.models_idx
                max_cluster_k_idx = p2.cluster_k_idx
                max_score = p2_score
                best_labels = copy.deepcopy(labels)

        # If, score get higher score than thresh, terminate gradient searching
        if thresh_score != None and max_score > thresh_score: break


        # 3. Calcuate gradient of each theta(point).
        #    with using above theta value, set another theta(point).
        change_of_cost = p2_score - p1_score
        change_of_theta1 = p2.scalers_idx - p1.scalers_idx
        change_of_theta2 = p2.models_idx - p1.models_idx
        change_of_theta3 = p2.cluster_k_idx - p1.cluster_k_idx

        # If, attribute of theta1 and theta2 are same, set gradient value to 0 (slope = 0)
        def update_gradient_value(change_of_cost, change_of_theta):
            result_gradient = 0
            if change_of_theta != 0:
                result_gradient = change_of_cost / change_of_theta
            return result_gradient

        gradient_theta1 = update_gradient_value(change_of_cost, change_of_theta1)
        gradient_theta2 = update_gradient_value(change_of_cost, change_of_theta2)
        gradient_theta3 = update_gradient_value(change_of_cost, change_of_theta3)


        # 4. Prepare for next gradient (change theta 1 to new position)
        def set_new_point(gradient_theta, compare1, compare2):
            result_idx = 0
            if gradient_theta > 0:
                result_idx = max([compare1, compare2])
            elif gradient_theta < 0:
                result_idx = min([compare1, compare2])
            else:
                result_idx = compare1
            return result_idx


        # Set new theta1 for the next calculation
        p1.scalers_idx = set_new_point(gradient_theta1, p1.scalers_idx, p2.scalers_idx)
        p1.models_idx = set_new_point(gradient_theta2, p1.models_idx, p2.models_idx)
        p1.cluster_k_idx = set_new_point(gradient_theta3, p1.cluster_k_idx, p2.cluster_k_idx)


    # Return the result
    class res:
        best_params = {}

    res.best_params = {
        'best_scaler': scalers[max_scalers_idx],
        'best_model': models[max_models_idx],
        'best_k': cluster_k[max_cluster_k_idx],
    }
    res.best_scaler = scalers[max_scalers_idx]
    res.best_model = models[max_models_idx]
    res.best_k = cluster_k[max_cluster_k_idx]

    res.best_score = max_score
    res.labels = best_labels

    # Return value with res class
    return res


def logo():
    print("")
    print("        /______\      /___\   /____/\/__________\  /____________\      /_________________\  /____\     ")
    print("       /        \    /    /\ /    / /            /\/            /\    /                   /\/     /\    ")
    print("      /    /\    \  /    / //    / /\__/    /\__\ /    /___/   / /   /    /_/    /_/    / /     / /    ")
    print("     /    /_/    /\/    / //    / / _/    / /__/ /    /   /   / /   /    / /    / /    / /     / /     ")
    print("    /      _    / /    /__/    / /   /    / /   /    /___/   / /   /    / /    / /    / /     /_/      ")
    print("   /    /_/    / /           / /   /    / /   /            / /   /    / /    / /    / /     ______/\  ")
    print("  /____/ /____/ /____________/ /   /____/ /   /____________/ /   /____/ /____/ /____/ /____________/ / ")
    print("  \____\ \____\/\____________\/    \____\/    \____________\/    \____\ \____\ \____\/ \____________\/  ")
    print("                                                                  for Clustering / Version: 2021.11.17 ")
```