중앙대학교
CHUNG ANG UNIVERSITY

정보와 지식: 탐구 과제 1 보고서

(2023-1)

강의 / 분반:	정보와 지식 02분반
교수:	박재화
제출일자:	2023. 05. 14
학번:	20184757
이름:	주영석

목차

1. 서론 3

2. 본론 및 구현 3

2-1. Data Preprocessing 3
2-2. Distance Metric 5
2-3. Evaluation 5
2-4. K-means 5
2-5. K-medoids 6

3. 결과 및 분석 8

3-1. K-means 8
3-2. K-medoids 11

4. 프로그램 실행 방법 12

1. 서론

군집화란 (Clustering), 동일한 군집 (Cluster) 내의 개체들이(Object) 서로 다른 군집의 개체들보다 더 유사하도록 개체 집합의 그룹을 나누는 작업이다. 군집화를 데이터의 모음인 데이터셋에 적용할 경우, 데이터셋에서 어떠한 패턴이나 구조를 알아낼 수 있고, 데이터셋을 사용하려는 목적에 부합하는 가치 있는 정보를 얻을 수 있다. 인간의 경우 시각적으로 3차원 이하의 그래프에 표현된 데이터에 대해 미세한 밀도 차이를 직관적으로 구분할 수 있다. 이 과정은 빠르지만, 우리가 지각할 수 있는 차원 이상의 데이터에 대해 이 과정을 수행하기란 쉽지 않은 일이 된다. 데이터를 직관적으로 우리의 감각 기관에 표현할 방법이 존재하지 않으므로, 우리는 직관적으로 표현되지 않은 데이터를 많은 시간을 들여 분석을 해야 할 것이다. 반면, 이 과정을 컴퓨터를 이용해 빠르게 연산 가능하도록 많은 방법이 만들어졌다. 대표적으로 k-means, k-medoids와 같은 중심 기반의 군집화 방법과 계층적 군집화 방법, 밀도 기반, 배포 기반 군집화 방법 등이 개발되었다. 컴퓨터를 이용해 어떤 데이터셋을 군집화할 경우, 데이터를 어떤 초공간에서의 (Hyperspace) 하나의 점 및 벡터로 간주하고, 데이터 간의 정량적인 유사도를 기반으로 계산이 이루어지기 때문에, 데이터 간의 유사도 또는 거리를 측정하는 방식을 반드시 설정해야 한다. 수치 데이터의 경우 대표적으로 유클리드 거리 (Euclidean distance), 맨해튼 거리 (Manhattan distance), 코사인 유사도 (Cosine similarity) 등을 사용할 수 있다.

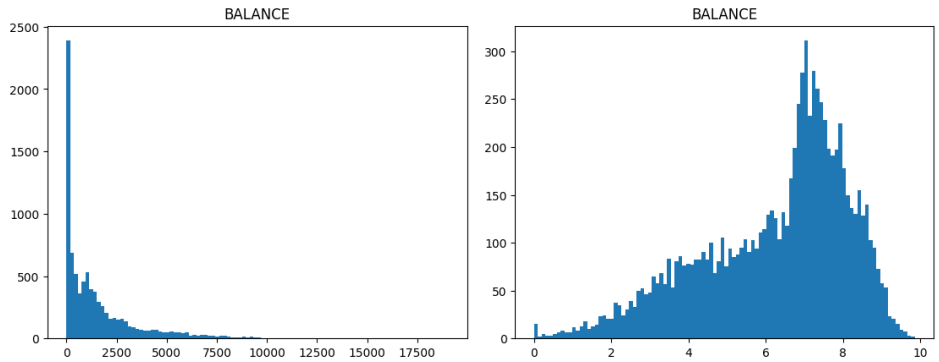
이 글에서는 Kaggle의 Credit Card Dataset을 (https://www.kaggle.com/datasets/arjunbhasin2013/ccdata) 사용한다. 해당 데이터는 총 8950명의 18가지 신용카드 정보를 포함하고 있다. 포함된 데이터의 예시로는 계좌 잔액, 결제 횟수, 최대 결제 금액, 한도, 이용 년 수 등 다양한 데이터가 존재한다. 이를 분석하여 카드 사용자의 집단을 6개의 그룹으로 나누고자 한다. 나눈 그룹을 토대로, 카드사의 경우에는 각 그룹에 알맞은 카드 상품을 출시하여 수익을 극대화할 수 있을 것이다. 군집화는 k-means와 k-medoid 두 가지 방법의 알고리즘을 Python3 언어를 이용해 구현하고, 성능을 비교하는 것을 목표로 한다.

2. 본론 및 구현

2-1. Data Preprocessing

선정한 Credit Card Dataset은 총 8950명의 18가지 신용카드 정보를 포함하고 있다. 이 중 사용자 번호는 유용하지 않은 정보이므로 제거했다. 또한, 값이 존재하지 않는 데이터 314개를 제거하여 8636명의 데이터만을 사용했다. 따라서 데이터의 모양은 8636*17의 수치 데이터만을 가진 matrix이다.

각 column의 값의 분포를 알아보기 위해, column 별로 히스토그램을 그려 분포를 확인해 본 결과, 일부 column의 경우 아래의 왼쪽 그림처럼 극단적으로 편향된 분포를 가지고 있다는 것을 알게 되었다. 따라서, 해당 column 들에 로그를 취하여 분포를 아래의 오른쪽 그림처럼 고르게 만들어 주는 작업을 했다.



마지막으로, 모든 column에 표준화를 적용했다. x가 데이터, μ가 column의 평균, σ가 column의 표준편차일 때, 아래의 수식을 통해 표준화가 적용되었다.

$$\frac{x-\mu}{\sigma}$$

해당 코드는 아래와 같다. csv 파일을 읽기 위해 pandas 패키지와, 배열 연산을 위해 numpy 패키지를 사용했다.

```
# Import library
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity

# Read csv data file
pd_csv = pd.read_csv('./CC GENERAL.csv')

# Remove unnecessary column and blank rows
data = pd_csv.drop(columns=['CUST_ID']).dropna().to_numpy()

# Get column names
column_names = pd_csv.columns[1:]

# Apply log to skewed columns
data_log = data
data_log[:,0] = np.log1p(data_log[:,0])
data_log[:,2] = np.log1p(data_log[:,2])
data_log[:,3] = np.log1p(data_log[:,3])
data_log[:,4] = np.log1p(data_log[:,4])
data_log[:,5] = np.log1p(data_log[:,5])
data_log[:,10] = np.log1p(data_log[:,10])
data_log[:,11] = np.log1p(data_log[:,11])
data_log[:,12] = np.log1p(data_log[:,12])
data_log[:,13] = np.log1p(data_log[:,13])
data_log[:,14] = np.log1p(data_log[:,14])

# Apply standardization
data_log_standardized = (data_log - np.mean(data_log, axis=0)) / np.std(data_log, axis=0)
```

2-2. Distance Metric

사용한 거리는 가장 기본적인 유클리드 거리를 사용했다. 데이터가 n 차원일 때, 두 데이터 p, q 간의 거리는 아래의 수식으로 계산되었다.

$$d(p,q) = \sqrt{\sum_{i=0}^{n-1}(p_i - q_i)^2}$$

해당 코드는 아래와 같다.

```
def euclidean_distance(point1, point2):
    return np.linalg.norm(point1 - point2)
```

2-3. Evaluation

각 군집화 알고리즘을 실행하기 이전에, 평가 방식을 미리 정의한다. 우선, 각 데이터가 속한 군집의 중심점과의 거리의 평균을 구한다. K-means의 경우 centroid, k-medoids의 경우 medoid와의 거리를 구한 다음 평균을 구한다. 거리는 앞서 정의한 유클리드 거리를 계산한다. 해당 코드는 아래와 같다.

```
def intra_cluster_distance(data_scaled, center, data_label):
    return np.mean([euclidean_distance(data_scaled[i], center[data_label[i]]) for i in range(data_scaled.shape[0])])
```

다음으로, 군집화 알고리즘의 결과를 평가하기 위해, 각 데이터가 속한 군집의 중심점과의 거리의 평균을, 각 군집의 중심점 간의 거리의 평균으로 나누어 준다. 해당 코드는 아래와 같다.

```
def intra_inter_variation(data_scaled, center, data_label):
    intra_variation = intra_cluster_distance(data_scaled, center, data_label)
    inter_variation = np.mean([
        euclidean_distance(centroid[i], centroid[j])
        for i in range(0, center.shape[0]-1)
        for j in range(i, center.shape[0])
    ])
    return intra_variation/inter_variation
```

2-4. K-means

먼저, k-means 알고리즘을 구현한다. 먼저, k 개의 centroid의 시작점을 초기화한다. 데이터에서 k 개를 임의로 추출한 뒤, 이를 centroid의 시작점으로 삼았다. 그다음, 최대 50회의 반복 횟수 동안, 각 데이터가 속한 군집의 번호를 centroid와의 거리가 최소인 centroid의 번호로 계산하고, 이를 토대로 다시 centroid를 계산한다. 만약 이전의 centroid와 새로 계산된 centroid가 동일하다면, 수렴한 것으로 간주하고 반복을 멈춘다. 수렴한 상태에서의 평가치를 계산하고, 각각의 상태를 저장한다. 이 과정을 10번 반복하며 실행 시간을 측정한다.

코드는 아래와 같다.

```
# Parameters
k=6
num_epoch=50
num_trial=10
data_num = data_log_standardized.shape[0]
data_dim = data_log_standardized.shape[1]

# Array for saving each trial results
centroid_trial = np.zeros((num_trial, k, data_dim), dtype=np.float64)
data_label_trial = np.zeros((num_trial, data_num), dtype=np.int32)
distance_trial = []
intra_inter_variation_trial = np.zeros(num_trial, dtype=np.float64)

# 10 trials
for trial in range(num_trial):

    # Time check
    start_time = time.time()

    # Initialize centroids with k random data points
    centroid = data_log_standardized[np.random.choice(np.arange(data_num), k)]
    centroid_history = np.zeros((num_epoch, k, data_dim), dtype=np.float64)

    # Initialize label of each data
    data_label = np.zeros((data_num), dtype=np.int32)

    # Array for saving sum of distances
    distance_temp = []
```

```
    # Repeat for 50 epochs
    for epoch in range(num_epoch):

        # Calculate each data label as the minimum distance with centroids
        for data_index, data_point in enumerate(data_log_standardized):
            data_label[data_index] = np.argmin(
                np.array([euclidean_distance(data_point, centroid[i]) for i in range(k)])
            )

        # Calculate new centroid with new labels
        for i in range(k):
            if len(np.where(data_label==i)[0])>0:
                centroid[i,:] = np.mean(data_log_standardized[np.where(data_label==i)[0],:], axis=0)

        # Save centroid
        centroid_history[epoch,:,:] = centroid

        # Calculate distance and save
        distance_temp.append(intra_cluster_distance(data_log_standardized, centroid, data_label))

        # If centroids didn't change, terminate
        if np.array_equal(centroid_history[epoch], centroid_history[epoch-1]):
            break

    # Save results
    centroid_trial[trial] = centroid
    data_label_trial[trial] = data_label
    distance_trial.append(distance_temp)
    intra_inter_variation_trial[trial] = intra_inter_variation(data_log_standardized, centroid, data_label)

    print(f'Trial #{trial} execution time: {time.time()-start_time}s')
```

2-5. K-medoids

K-medoids는 PAM 알고리즘에 따라 개발되었다.

```
# Parameters
k=6
num_epoch=50
num_trial=10
data_num = data_log_standardized.shape[0]
data_dim = data_log_standardized.shape[1]

# Array for saving each trial results
medoid_trial = np.zeros((num_trial, k), dtype=np.float64)
data_label_trial = np.zeros((num_trial, data_num), dtype=np.int32)
distance_trial = []
intra_inter_variation_trial = np.zeros(num_trial, dtype=np.float64)

# Calculate distance matrix
start_time = time.time()
distance_matrix = np.zeros((data_num, data_num), dtype=np.float64)
for i in range(0, data_num-1):
    for j in range(i+1, data_num):
        distance_matrix[i, j] = euclidean_distance(data_log_standardized[i], data_log_standardized[j])
distance_matrix = distance_matrix + distance_matrix.T
print(f'Distance matrix calculation time: {time.time()-start_time}s')
```

```
# 10 trials
for trial in range(num_trial):

    # Time check
    start_time = time.time()

    # Initialize medoids
    medoid = np.full(k, fill_value=-1, dtype=np.int32)
    medoid_history = np.zeros((num_epoch, k), dtype=np.int32)

    # Initialize label of each data
    data_label = np.zeros((data_log_standardized.shape[0]), dtype=np.int32)

    # Select first medoid as random
    medoid[0] = np.random.randint(data_num)

    # Initialize distance from data to each medoid
    distance_medoid = distance_matrix[medoid[0],:]
    distance_sum = np.sum(distance_medoid)
    distance_record = []

    # Greedily select other medoids
    for m in range(k):
        distance_medoid_temp = distance_medoid
        distance_sum_temp = distance_sum
        data_label_temp = data_label
        medoid_temp = 0
        for i in range(data_num):
            if i in medoid:
                continue
            distance_new = np.min(np.array([distance_medoid, distance_matrix[i,:]]).T, axis=1)
            if np.sum(distance_new)<distance_sum_temp:
                data_label_temp = np.argmin(np.array([distance_medoid, distance_matrix[i,:]]).T, axis=1)
                distance_medoid_temp = distance_new
                distance_sum_temp = np.sum(distance_new)
                medoid_temp = i
        medoid[m] = medoid_temp
        distance_medoid = distance_medoid_temp
        distance_sum = distance_sum_temp
        data_label[np.where(data_label_temp == 1)[0]] = m
```

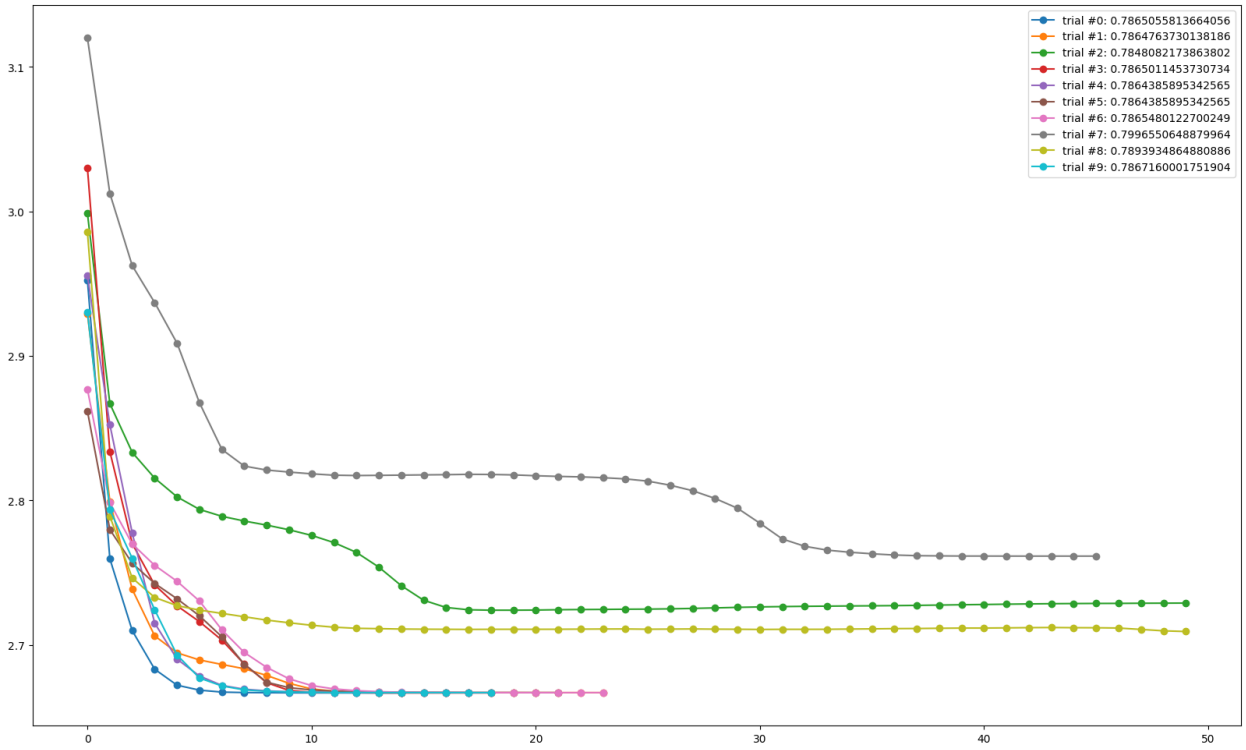
```
    distance_record.append(distance_sum)

    medoid_new = medoid
    for epoch in range(num_epoch):
        isbreak = False
        for m in range(k):
            distance_medoid_temp = distance_medoid
            distance_sum_temp = distance_sum
            data_label_temp = data_label
            medoid_temp = 0
            for i in range(data_num):
                if data_label[i]!=m:
                    continue
                if i==medoid[m]:
                    continue
                distance_new = np.min(np.array([distance_medoid, distance_matrix[i,:]]).T, axis=1)
                if np.sum(distance_new) < distance_sum_temp:
                    data_label_temp = np.argmin(np.array([distance_medoid, distance_matrix[i,:]]).T)
                    distance_medoid_temp = distance_new
                    distance_sum_temp = np.sum(distance_new)
                    medoid_temp = i
            if distance_sum_temp<distance_sum:
                medoid[m] = medoid_temp
                distance_medoid = distance_medoid_temp
                distance_sum = distance_sum_temp
                data_label[np.where(data_label_temp == 1)[0]] = m
                distance_record.append(distance_sum)
            else:
                isbreak = True
                break
        if isbreak:
            break

    medoid_trial[trial] = medoid
    data_label_trial[trial] = data_label
    distance_trial.append(distance_record)
    intra_inter_variation_trial[trial] = \
    intra_inter_variation(data_log_standardized, data_log_standardized[medoid], data_label)
    print(f'Trial #{trial} execution time: {time.time()-start_time}s')
```

3. 결과 및 분석

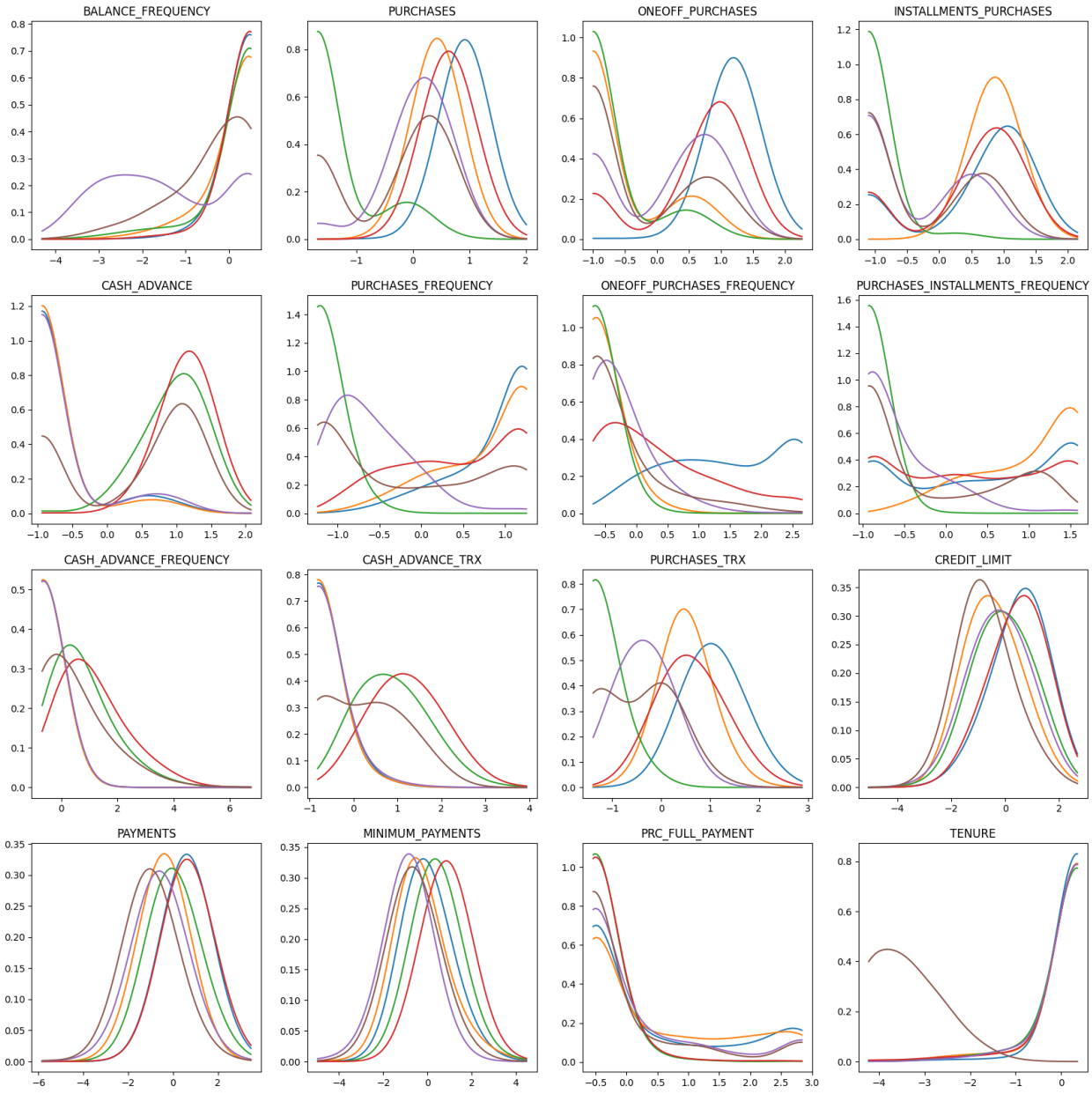
결과 그래프를 그리기 위해 matplotlib 패키지를 이용했다.

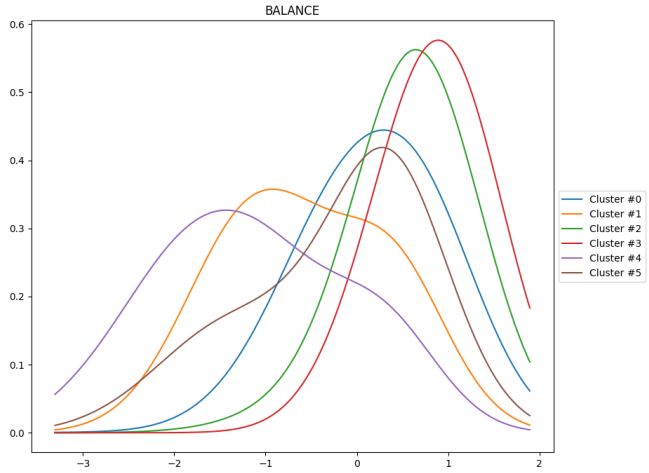
3-1. K-means

먼저, 10번의 시도에 대해 각 데이터와 해당하는 centroid와의 거리의 평균과, 해당하는 평가치는 아래와 같은 결과를 보였다.



이들 중, 평가치가 가장 낮은 결과를 사용해 다음의 결과를 분석하였다. 가장 낮은 결과를 사용한 이유는, 클러스터 내의 거리는 작을수록 좋고, 클러스터 간의 거리는 클수록 좋기 때문에, 두 값을 나눈 값은 결과적으로 낮을수록 클러스터링이 잘 되었다고 판단할 수 있기 때문이다. 아래의 사진은 각 column에서 각 클러스터의 데이터 분포를 Gaussian kernel을 이용한 kernel density estimation을 통해 확률밀도함수의 형태로 나타낸 결과이다. Kernel density estimation은 scikit-learn 패키지를 이용해 구현하였다.


BALANCE_FREQUENCY
PURCHASES
ONEOFF_PURCHASES
INSTALLMENTS_PURCHASES
CASH_ADVANCE
PURCHASES_FREQUENCY
ONEOFF_PURCHASES_FREQUENCY
PURCHASES_INSTALLMENTS_FREQUENCY
CASH_ADVANCE_FREQUENCY
CASH_ADVANCE_TRX
PURCHASES_TRX
CREDIT_LIMIT
PAYMENTS
MINIMUM_PAYMENTS
PRC_FULL_PAYMENT
TENURE



위의 그래프로 각 클러스터를 분석했을 때, 각 군집에 속하는 사용자의 특성을 다음과 같이 적을 수 있다.

- 클러스터 0: 계좌에 돈이 중간 정도로 있고, 카드 결제 횟수가 많고 잦으며, 최대 일시불 결제 금액이 가장 높다. 할부 결제 및 일시불 결제 빈도, 신용 한도가 가장 높은 그룹이다. 반면 선불 결제는 거의 이용하지 않는다.
- 클러스터 1: 계좌에 상대적으로 돈은 적고, 카드 결제 횟수, 최대 결제 금액도 적은 편이다. 할부 결제 횟수, 빈도가 높으며, 신용 한도는 작은 편이다. 마찬가지로 선불 결제를 거의 이용하지 않는다.
- 클러스터 2: 계좌에 돈이 많은 편이나, 결제 횟수 및 빈도, 최대 결제 금액은 작다. 할부 결제 비율도 가장 낮으며, 신용 한도는 중간이다. 선불 결제는 상대적으로 자주 한다.
- 클러스터 3: 클러스터 0과 거의 동일하나, 계좌에 돈이 가장 많다. 일시불 결제 빈도는 높지 않으나, 현금 결제 비율이 높다.
- 클러스터 4: 계좌에 돈이 가장 적으며, 잔액이 업데이트되는 빈도 또한 낮다. 모든 종류의 결제 양과 빈도가 가장 낮다.
- 클러스터 5: 계좌에 돈은 중간 정도로 있으나, 결제 횟수는 적은 편이다. 현금 결제 비율은 상대적으로 높다. 신용 한도는 가장 낮으며, 다른 그룹과는 다르게 카드 이용 기간이 짧다.

3-2. K-medoids

K-means의 10회 실행시간은 평균 5.5초의 실행시간을 보였지만, k-medoid는 10초정도의 시간을 보였다. 또한, 평가치도 k-means의 경우 0.7정도의 값을 보였지만, k-medoids의 경우 0.9정도의 값을 보였다.

4. 프로그램 실행 방법

각 프로그램 k_means.py, k_medoids.py는 Python3으로 작성되어있다. 필요한 라이브러리는 pandas, numpy, matplotlib, scikit-learn이다. 해당 라이브러리를 설치하기 위해서는 pip이 설치되어 있어야 한다. 터미널에서 다음의 명령어를 실행하면 모든 라이브러리를 설치할 수 있다.

```
$ pip3 install pandas numpy matplotlib scikit-learn
```

다음으로, k-means.py를 실행하기 위해서는, 다음의 명령어를 실행하면 된다. 프로그램 실행이 완료되면, 처음으로 나오는 그림은 10번의 시도에 대한 distance의 변화 그래프이고, 두번째와 세번째는 10번의 시도 중 가장 클러스터링 결과가 좋은 결과에 대해 데이터의 각 column에서 각 클러스터의 데이터 분포를 나타내는 probability density function이다.

```
$ python3 k_means.py
```

또한, k-medoids.py를 실행하기 위해서는, 다음의 명령어를 실행하면 된다.

```
$ python3 k_medoids.py
```