Solutions 4

Jumping Rivers

For our clustering example we will explore some data on credit card usage behaviour. This sort of data might be used to develop a customer segmentation in order to assist with a marketing strategy.

The data can be described with the following variables:

- CUST_ID : Identification of Credit Card holder (Categorical)
- BALANCE : Balance amount left in their account to make purchases
- BALANCE_FREQUENCY : How frequently the Balance is updated, score between 0 and 1 (1 = frequently updated, 0 = not frequently updated)
- PURCHASES : Amount of purchases made from account
- ONEOFF_PURCHASES : Maximum purchase amount done in one-go
- INSTALLMENTS_PURCHASES : Amount of purchase done in installment
- CASH_ADVANCE : Cash in advance given by the user
- PURCHASES_FREQUENCY : How frequently the Purchases are being made, score between 0 and 1 (1 = frequently purchased, 0 = not frequently purchased)
- ONEOFF_PURCHASES_FREQUENCY : How frequently Purchases are happening in one-go (1 = frequently purchased, 0 = not frequently purchased)
- PURCHASES_INSTALLMENTS_FREQUENCY : How frequently purchases in installments are being done (1 = frequently done, 0 = not frequently done)
- CASH_ADVANCE_FREQUENCY : How frequently the cash in advance being paid
- CASH_ADVANCE_TRX : Number of Transactions made with "Cash in Advanced"
- PURCHASES_TRX : Numbe of purchase transactions made CREDIT_LIMIT : Limit of Credit Card for user
- PAYMENTS : Amount of Payment done by user
- MINIMUM_PAYMENTS : Minimum amount of payments made by user
- PRC_FULL_PAYMENT : Percent of full payment paid by user
- TENURE : Tenure of credit card service for user

It can be loaded from the package as follows

```
import jrpyml
ccdata = jrpyml.datasets.ccdata.load_data()
```

This data is a real sample so is a little messy, for example we have a number of missing values. The easiest way to deal with this for now is to remove them, however we could in theory deal with this with some form of imputation.

- Remove the missing values from the data

```
ccdata = ccdata.dropna()
```

- Fit KMeans clustering to the data across a range of values to produce an elbow plot. How many cluster do you think are apparent here? Don't forget to scale your data appropriately first.

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
ccdata_scaled = scaler.fit_transform(ccdata.drop('CUST_ID', axis=1))

max_clusters = 25
sse = []
for i in range(1, max_clusters):
  print(i)
  cluster = KMeans(i)
  cluster.fit(ccdata_scaled)
  sse.append(cluster.inertia_)
```

```
## 1
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=1, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 2
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 3
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 4
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=4, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 5
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=5, n_init=10, n_jobs=None, precompute_distances='auto',
```

```
##        random_state=None, tol=0.0001, verbose=0)
## 6
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=6, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 7
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=7, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 8
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=8, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 9
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 10
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=10, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 11
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=11, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 12
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=12, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 13
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=13, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 14
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=14, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 15
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=15, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 16
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=16, n_init=10, n_jobs=None, precompute_distances='auto',
```

```
##        random_state=None, tol=0.0001, verbose=0)
## 17
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=17, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 18
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=18, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 19
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=19, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 20
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=20, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 21
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=21, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 22
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=22, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 23
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=23, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
## 24
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=24, n_init=10, n_jobs=None, precompute_distances='auto',
##        random_state=None, tol=0.0001, verbose=0)
```

```
import matplotlib.pyplot as plt
plt.plot(sse, 'bx-')
plt.show()
```

- Add the new cluster labels into the data set as a variable named cluster.

```
kmean = KMeans(9)
kmean.fit(ccdata_scaled)
```

```
## KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
##        n_clusters=9, n_init=10, n_jobs=None, precompute_distances='auto',
```

```
##        random_state=None, tol=0.0001, verbose=0)
```

```
labels = kmean.labels_
```

```
ccdata['cluster'] = labels
```

- Now that we have different clusters we might look to try to understand the customer segments, below is a definition of a function which will let you explore individual variables across the clusters.

```
import seaborn as sns
```

```
plot_data = ccdata.drop('CUST_ID', axis=1)
```

```
def examine(plot_data, column):
  grid = sns.FacetGrid(plot_data, col='cluster')
  grid.map(sns.distplot, column)
  plt.show()
```

```
examine(plot_data, 'CREDIT_LIMIT')
```

- It is possible that some outliers make understanding the clusters more difficult. We haven't really discussed outlier detection during the material but a `LocalOutlierFactor` measures deviation of a given sample with respect to its surrounding neighbourhood. We might trim outliers measured this way with the following code.

```
from sklearn.neighbors import LocalOutlierFactor
outlier = LocalOutlierFactor()
```

```
labels = outlier.fit_predict(ccdata_scaled)
```

```
## /filestore/ubuntu/anaconda3/lib/python3.7/site-packages/sklearn/neighbors/lof.py:236: FutureWarning:
##   FutureWarning)
```

```
ccdata_trimmed = ccdata_scaled[labels == 1]
```

- Does this changes anything about your cluster analysis?