Assignment 3 (DAT340) - AI tools

February 6, 2024

1 Assignment 3 - AI tools

- Student 1 - Luca Modica
- Student 2 - Hugo Manuel Alves Henriques E Silva

1.1 Import libraries

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

sns.set_style('darkgrid')
%matplotlib inline
```

```
/tmp/ipykernel_5908/3608931809.py:2: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of
pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better
interoperability with other libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
```

1.2 Reading data

[2]:
```
import os

def read_csv_files(directory):
    csv_files = [file for file in os.listdir(directory) if file.
 ↪endswith('_labeled.csv')]
    data = {}
```

```python
    for file in csv_files:
        file_path = os.path.join(directory, file)
        df = pd.read_csv(file_path)
        key = file.replace('_labeled.csv', '')
        data[key] = df

    return [file.replace('_labeled.csv', '') for file in csv_files], data
```

[3]:
```python
# with this method we will return an object
# with as property name the city and as
# property value the related dataframe
cities_names, dfs = read_csv_files('./')

print(f'Data from the following cities: {", ".join(cities_names)}.')
```

```
Data from the following cities: Shenyang, Guangzhou, Shanghai, Chengdu, Beijing.
```

[4]:
```python
city = 'Beijing'
print(f'Data from {city}: ')
dfs[city].head()
```

```
Data from Beijing:
```

[4]:
```
   season  DEWP  HUMI    PRES  TEMP     Iws  precipitation  cbwd_NE  cbwd_NW  \
0       4  -8.0  79.0  1026.0  -5.0   23.69            0.0        0        0
1       4 -11.0  85.0  1021.0  -9.0  105.93            1.1        0        0
2       4 -21.0  43.0  1030.0 -11.0  117.55            0.0        0        1
3       4 -25.0  33.0  1034.0 -12.0   39.35            0.0        1        0
4       4 -24.0  30.0  1034.0 -10.0   59.00            0.0        1        0

   cbwd_SE  PM_HIGH
0        1      1.0
1        1      0.0
2        0      0.0
3        0      0.0
4        0      0.0
```

[5]:
```python
city = 'Chengdu'
print(f'Data from {city}: ')
dfs[city].head()
```

```
Data from Chengdu:
```

[5]:
```
   season  DEWP   HUMI    PRES  TEMP  Iws  precipitation  cbwd_NE  cbwd_NW  \
0       2  20.0  88.45  1007.1  22.0  1.0            0.0        0        0
1       2  17.0  54.39  1008.1  27.0  5.0            0.0        0        0
2       2  20.0  78.39  1008.1  24.0  2.0            0.0        0        0
3       2  19.0  65.41  1006.1  26.0  2.0            0.0        0        0
```

```
4        2  20.0  61.90  1003.1  28.0  2.0            0.0         0         0

   cbwd_SE  PM_HIGH
0        0      1.0
1        0      0.0
2        0      0.0
3        0      0.0
4        0      0.0
```

[6]:
```python
city = 'Guangzhou'
print(f'Data from {city}: ')
dfs[city].head()
```

```
Data from Guangzhou:
```

[6]:
```
   season  DEWP  HUMI    PRES  TEMP  Iws  precipitation  cbwd_NE  cbwd_NW  \
0     3.0  15.2  62.0  1013.9  22.9  7.3            0.0        1        0
1     3.0  10.7  43.0  1013.7  24.0  5.2            0.0        1        0
2     3.0   8.8  42.0  1014.4  22.3  9.2            0.0        1        0
3     3.0  12.1  51.0  1013.2  22.7  9.5            0.0        1        0
4     3.0  15.3  76.0  1011.5  19.6  3.0            0.0        0        1

   cbwd_SE  PM_HIGH
0        0      0.0
1        0      0.0
2        0      0.0
3        0      0.0
4        0      0.0
```

[7]:
```python
city = 'Shanghai'
print(f'Data from {city}: ')
dfs[city].head()
```

```
Data from Shanghai:
```

[7]:
```
   season  DEWP   HUMI    PRES  TEMP    Iws  precipitation  cbwd_NE  cbwd_NW  \
0       4   3.0  57.77  1030.1  11.0   66.0            0.0        1        0
1       4  -2.0  49.22  1032.9   8.0  194.0            0.0        1        0
2       4  -1.0  49.51  1029.1   9.0    2.0            0.0        1        0
3       4  -4.0  42.40  1029.1   8.0    7.0            0.0        1        0
4       4  -4.0  45.40  1028.1   7.0    2.0            0.0        0        1

   cbwd_SE  PM_HIGH
0        0      0.0
1        0      0.0
2        0      0.0
3        0      0.0
```

```
4         0      0.0
```

[8]:
```
city = 'Shenyang'
print(f'Data from {city}: ')
dfs[city].head()
```

```
Data from Shenyang:
```

[8]:
```
   season  DEWP   HUMI    PRES  TEMP   Iws  precipitation  cbwd_NE  cbwd_NW  \
0       1  -3.0  26.98  1010.0  16.0  31.0            0.0        0        0
1       1   6.0  58.54  1008.0  14.0  51.0            0.0        0        0
2       1   0.0  43.60  1006.0  12.0   7.0            0.0        0        1
3       1   2.0  41.43  1011.0  15.0  23.0            0.0        0        1
4       1  -5.0  18.06  1013.0  20.0  28.0            0.0        0        0

   cbwd_SE  PM_HIGH
0        0      0.0
1        0      0.0
2        0      0.0
3        0      0.0
4        1      0.0
```

1.3 Model implementation

The Python class below will implement the KMeans algorithm.

[9]:
```
import warnings
from sklearn.metrics import normalized_mutual_info_score, silhouette_score,␣
 ↪adjusted_rand_score
from sklearn.base import BaseEstimator, ClusterMixin, check_array

class KMeans(BaseEstimator, ClusterMixin):

    def __init__(self,
        n_centers=2,
        max_iter=200,
        init_centroids='random',
        random_seed=None,
        distance_metric="euclidean"
        ):
        self.n_centers = n_centers
        self.max_iter = max_iter
        self.init_centroids = init_centroids
        self.random_seed = int(random_seed) if random_seed is not None else None
        self.distance_metric = distance_metric
```

```
    # ***private methods***
    def _set_distance_func(self, distance_metric):
        if distance_metric == "euclidean":
            self.distance_func_ = euclidean_distance
        elif distance_metric == "manhattan":
            self.distance_func_ = manhattan_distance
        else:
            raise ValueError(f"Unknown distance metric: {self.distance_metric}")

    def _initialize_centroids(self, X):
        if self.init_centroids == 'random':
            if X.shape[0] < self.n_centers:
                warnings.warn(
                    "Number of samples in X is less than the number of centers.␣
↪\
                        The number of clusters has changed to number of␣
↪datapoints.", UserWarning)
                self.n_centers = X.shape[0]

            # sample "n_centers" datapoints as first centroids positions
            indices = np.random.choice(range(X.shape[0]), size=self.n_centers)
            self.centroids_ = X[indices]
        elif self.init_centroids == 'kmeans++':
            self.centroids_ = self._kmeans_plus_plus(X)
            # self.centroids_, _ = kmeans_plusplus(X=X, n_clusters=self.
↪n_centers)
        else:
            raise ValueError(f"Unknown centroid initialization method: {self.
↪init_centroids}")

    def _kmeans_plus_plus(self, X):
        centroids = np.empty((self.n_centers, self.n_features_in_))

        # sample a datapoint as first centroid
        index = np.random.choice(range(X.shape[0]), size=1)
        first_centroid = X[index]
        centroids[0] = first_centroid

        for i in range(1, self.n_centers):
            # Calculate distances from each point to its nearest centroid
            distances = np.array(
                [np.min([self.distance_func_(x, c) for c in centroids[:i]]) for␣
↪x in X])

            # Normalize the square of these distances to create a probability␣
↪distribution
```

```
            squared_distances = distances**2
            probabilities = squared_distances / np.sum(squared_distances)

            # Randomly select the next centroid based on the probability␣
↪distribution
            next_centroid_index = np.random.choice(range(X.shape[0]),␣
↪p=probabilities, size=1)
            centroids[i] = X[next_centroid_index]



        return centroids



    # ***public methods***
    def fit(self, X, y=None):
        np.random.seed(self.random_seed)
        X = check_array(X)
        self.n_features_in_ = X.shape[1]

        self._set_distance_func(self.distance_metric)
        self._initialize_centroids(X)

        n_iter = 0
        while n_iter < self.max_iter:
            # calculate distance from each point to each centroid
            distances = np.zeros((X.shape[0], self.n_centers))

            for i in range(self.n_centers):
                for j in range(X.shape[0]):
                    distances[j, i] = self.distance_func_(X[j], self.
↪centroids_[i])

            # assign each point to the closest centroid
            self.labels_ = np.argmin(distances, axis=1)

            # calculate new centroids
            new_centroids = np.zeros((self.n_centers, X.shape[1]))
            for i in range(self.n_centers):
                cluster_points = X[self.labels_ == i]
                if cluster_points.size > 0:
                    new_centroids[i] = np.mean(cluster_points, axis=0)
                else:
                    # Handle the empty cluster case
                    new_centroids[i] = X[np.random.choice(range(X.shape[0]),␣
↪size=1)]
```

```python
            # if centroids have not changed, stop
            if np.allclose(self.centroids_, new_centroids):
                break

            # otherwise, update centroids and continue
            self.centroids_ = new_centroids
            n_iter += 1

        self.n_iter_ = n_iter
        return self

    def transform(self, X):
        """In this custom KMeans implementation, X wont't be changed.
        The implementaion of the method is for consitency puroposes
        with the sklearn APIs and pipeline purposes to extract the
        data tranformed by previous steps (for example)"""
        X = check_array(X)
        if X.shape[1] != self.n_features_in_:
            raise ValueError("The number of features in transform is different␣
↪from the number of features in fit.")

        return check_array(X)

    def fit_transform(self, X, y=None):
        """In this custom KMeans implementation, after .fit() X wont't be␣
↪changed.
        The implementaion of the method is for consitency puroposes
        with the sklearn APIs and pipeline purposes to extract the
        data tranformed by previous steps (for example)"""

        return self.fit(X).transform(X)

    def predict(self, X):
        X = check_array(X, accept_sparse=True)

        pred_labels = np.zeros(X.shape[0], dtype=np.dtype("int64"))
        distances = np.zeros((X.shape[0], self.n_centers))

        for i in range(self.n_centers):
            for j in range(X.shape[0]):
                distances[j, i] = self.distance_func_(X[j], self.centroids_[i])

        pred_labels = np.argmin(distances, axis=1)

        return pred_labels
```

```
    def score(self, X=None, y=None, score_metric='inertia'):
        # raise value error if "y" is none for score
        # metrics that require ground truth  labels
        if y is None and score_metric in ['nmi', 'ari']:
            raise ValueError(
                f"To compute the this score ({score_metric}), 'y' cannot be␣
↪None.")

        # raise value error if "X" is none for score
        # metrics that measures the internal clusters
        # quality
        if X is None and score_metric in ['inertia', 'silhouette']:
            raise ValueError(
                f"To compute the this score ({score_metric}), 'X' cannot be␣
↪None.")


        if score_metric == 'inertia':
            return inertia_score(X, self.centroids_, self.labels_)
        elif score_metric == 'silhouette':
            return effcient_silhouette_score(X, self.labels_, self.
↪distance_metric)
        elif score_metric == 'nmi':
            return nmi_score(y, self.labels_)
        elif score_metric == 'ari':
            return adjusted_rand_score(y, self.labels_)
        else:
            raise ValueError(f"Unknown score metrix: {score_metric}")


# distance functions
def euclidean_distance(x, y):
    return np.sqrt(np.sum((x - y) ** 2))

def manhattan_distance(x, y):
    return np.sum(np.abs(x - y))


# scoring functions
def inertia_score(X, centroids, labels):
    """Inertia measures the sum of squared distances between each sample and␣
↪its
    closest centroid. A lower inertia indicates better clustering."""
    return np.sum((X - centroids[labels]) ** 2)

def custom_silhouette_score(X, labels, n_centers, distance_func):
    """Silhouette score measures how similar an object is to its own cluster
```

```
    (cohesion) compared to other clusters (separation). The silhouette ranges
    from -1 to 1, where a high value indicates that the object is well matched
    to its own cluster and poorly matched to neighboring clusters."""

    silhouette_scores = np.zeros(X.shape[0])

    for i in range(X.shape[0]):
        #get label of the current point
        label = labels[i]
        intra_distances = [distance_func(X[i], X[j]) for j in range(X.shape[0])␣
 ↪if labels[j] == label and i != j]
        average_intra_cluster_distance = np.mean(intra_distances) if␣
 ↪intra_distances else 0

        #distance to points in other clusters
        min_distance_other_clusters = np.min([np.mean([distance_func(X[i],␣
 ↪X[j]) for j in range(X.shape[0]) if labels[j] == k]) for k in␣
 ↪range(n_centers) if k != label])

        if average_intra_cluster_distance or min_distance_other_clusters:
            silhouette_scores[i] = (min_distance_other_clusters -␣
 ↪average_intra_cluster_distance) / max(min_distance_other_clusters,␣
 ↪average_intra_cluster_distance)
        else:
            silhouette_scores[i] = 0

    return np.mean(silhouette_scores)

def effcient_silhouette_score(X, labels, metric):
    return silhouette_score(X=X, labels=labels, metric=metric)

def nmi_score(labels_true, labels_pred):
    """Normalized Mutual Information between the true labels and the predicted␣
 ↪clusters
    divided by the average entropy of the true labels and the predicted␣
 ↪clusters"""
    return normalized_mutual_info_score(labels_true, labels_pred)

def ari_score(labels_true, labels_pred):
    """The Adjusted Rand index is a function that measures the
    similarity of true and predicted assignments, ignoring permutations.
    """
    return adjusted_rand_score(labels_true, labels_pred)
```

1.4 Sanity check

Usage of `check_estimator` to make sure the pattern "fit(), predict(), score()" is followed, alongside the good practises for a new custom cluster algorithm:

```
[10]: from sklearn.utils.estimator_checks import check_estimator

# Check if estimator adheres to scikit-learn conventions.
check_estimator(KMeans(random_seed=0))
```

Down below will define the sklarn pipline that will be used for checking for the usability of our class in there.

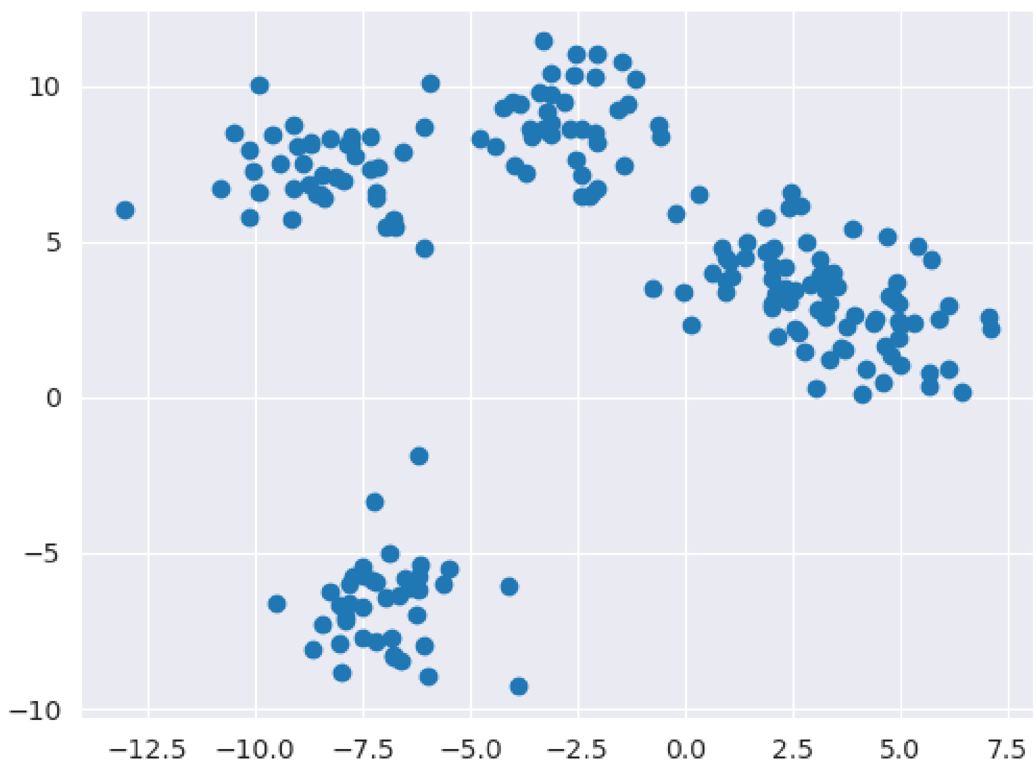
```
[11]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

pipeline_steps = [
    ('scaler', StandardScaler()),
    ('kmeans', KMeans())
]

pipe = Pipeline(pipeline_steps)
```

Function to train and then visualize a 2D dataset. It will return the trained k-means model.

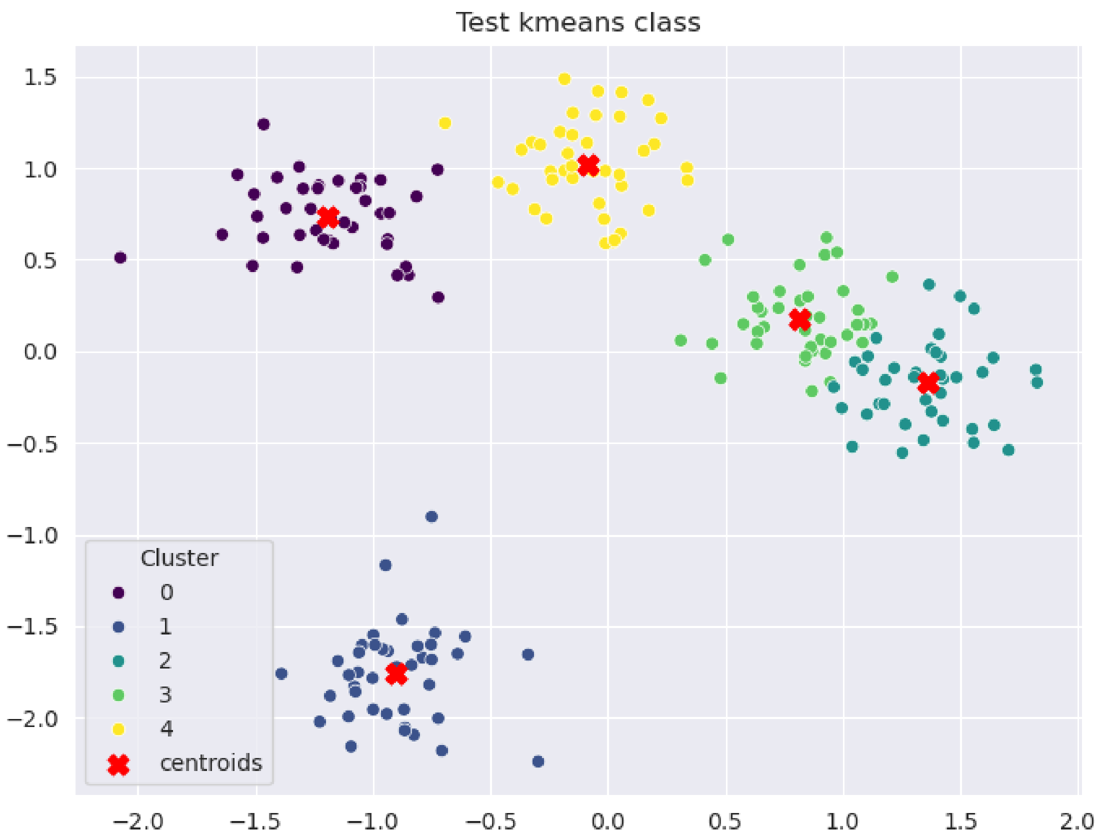
```
[12]: from sklearn.base import clone

def k_means_visualization(k, X, title, init_centroids='random'):
    # k-means training
    pipeline = clone(pipe)
    pipeline.set_params(kmeans__n_centers=k,␣
 ↪kmeans__init_centroids=init_centroids,
        kmeans__random_seed=None)
    X_scaled = pipeline.fit_transform(X)

    # Visualize the K-Means clusters
    kmeans = pipeline.named_steps['kmeans']
    plt.figure(figsize=(8, 6))
    sns.scatterplot(x=X_scaled[:, 0], y=X_scaled[:, 1], hue=kmeans.labels_,␣
 ↪palette='viridis')
    plt.scatter(kmeans.centroids_[:,0], kmeans.centroids_[:,1], marker="X",␣
 ↪c="r", s=80, label="centroids")
    plt.title(title)
    plt.grid(True)
    plt.legend(title='Cluster')

    # highlight clusters' centroids
    plt.scatter(kmeans.centroids_[:,0], kmeans.centroids_[:,1], marker="X",␣
 ↪c="r", s=80, label="centroids")
```

```
    plt.show()
    return kmeans, X_scaled
```

[13]:
```
from sklearn.datasets import make_blobs

# generate 2D data set
k = 5
X, y = make_blobs(
    n_samples=200,
    centers=k,
    cluster_std=1.3,
    random_state=42
)

# plot the data set
plt.scatter(X[:,0], X[:,1])
plt.show()
```



[14]:
```
kmeans, X_scaled = k_means_visualization(k=k, X=X, init_centroids='kmeans++',␣
 ↪title="Test kmeans class")
```



Check for the internal quality of the clusters themselves.

[15]:
```
print(f'Inertia score: {kmeans.score(X=X_scaled)}')
print(f'Silhouette score: {kmeans.score(X=X_scaled,␣
  ↪score_metric="silhouette")}')
```

```
Inertia score: 20.33513776553783
Silhouette score: 0.5859707207927123
```

Check for the for the scoring that cehck similarity with ground truth labels.

[16]:
```
print(f'Normalized Mutual Info score: {kmeans.score(y=y, score_metric="nmi")}')
print(f'Adjusted Rand Index score: {kmeans.score(y=y, score_metric="ari")}')
```

```
Normalized Mutual Info score: 0.9337944107790918
Adjusted Rand Index score: 0.9292019230769231
```

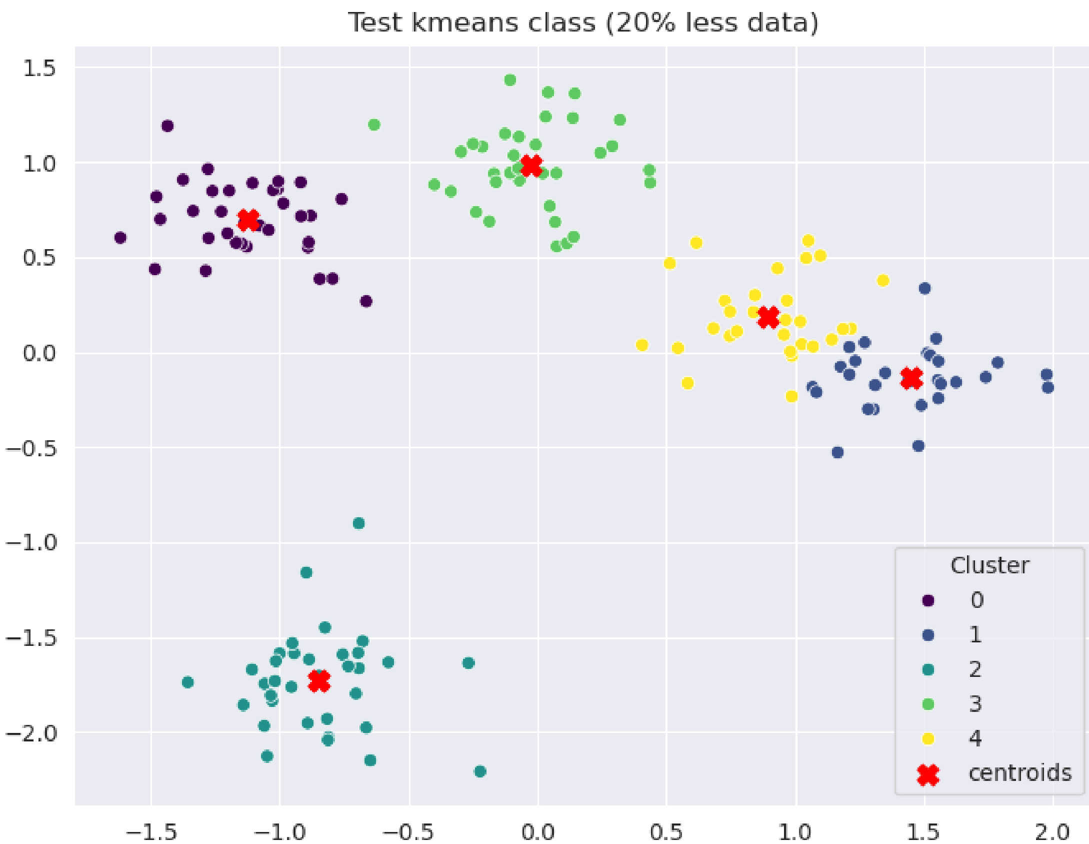
Test printing KMeans main information: the centroids coordinates and the label assignments.

[17]:
```
print(f"kmeans centroids coordinates: \n {kmeans.centroids_}")
print(f"kmeans labels for each datapoint: \n {kmeans.labels_}")
```

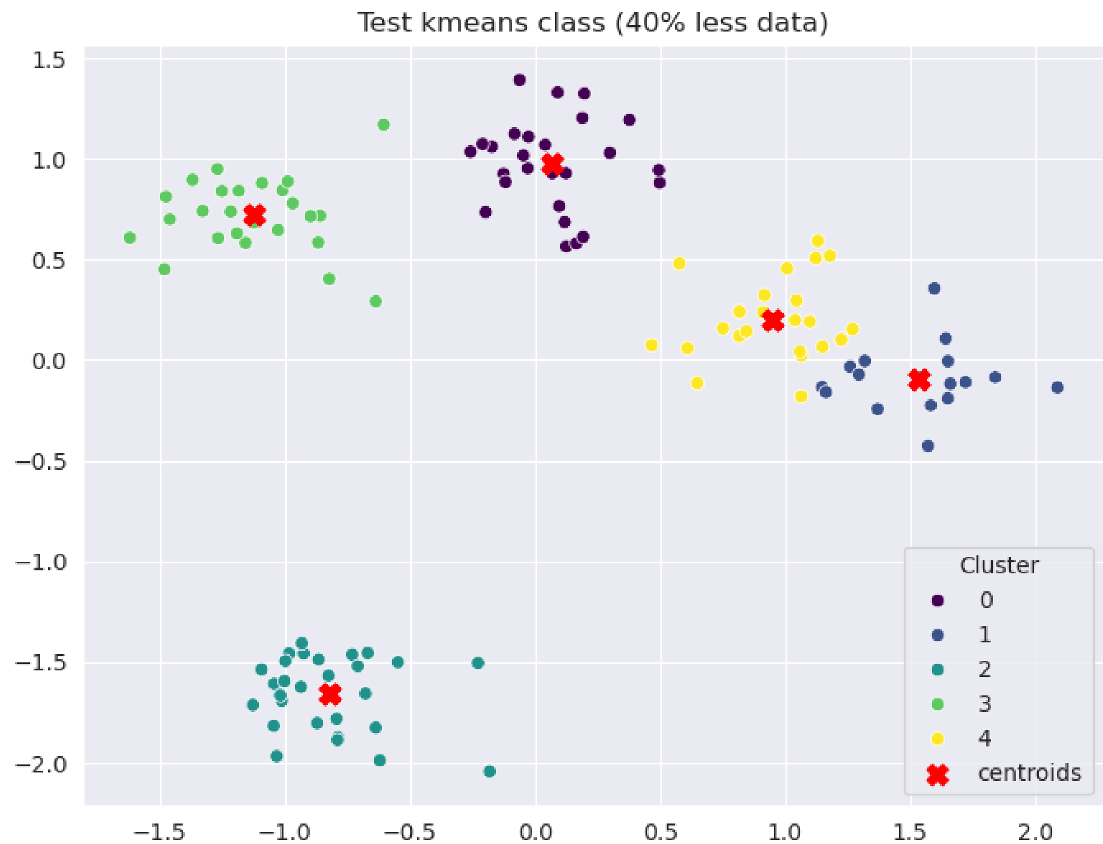
```
kmeans centroids coordinates:
 [[-1.19021388  0.73434046]
 [-0.90010829 -1.76121708]
 [ 1.35835139 -0.17462223]
 [ 0.81516133  0.17952727]
 [-0.08319055  1.02197158]]
kmeans labels for each datapoint:
 [1 4 3 0 2 0 3 3 1 2 2 1 1 1 0 3 4 3 3 0 0 0 2 4 0 4 4 2 4 1 0 0 0 1 4 0 0
 2 0 4 1 3 4 1 1 0 0 3 3 3 3 2 0 4 3 2 4 4 2 0 2 3 1 2 0 3 3 1 3 4 3 4 4 2
 4 0 2 1 3 4 0 4 4 1 1 0 0 0 3 4 1 0 1 2 2 1 2 1 3 4 2 4 2 1 2 2 0 2 1 3 1
 0 4 2 0 4 3 2 3 3 4 4 4 0 0 2 3 2 1 1 4 0 1 0 4 3 3 2 1 1 2 0 2 4 3 2 1 4
 2 3 0 0 4 3 1 3 1 3 3 3 1 2 4 1 1 2 4 3 1 2 2 0 1 0 2 1 0 2 0 2 4 2 0 4 1
 3 3 0 4 2 1 1 4 4 1 3 3 4 3 2]
```

Clustering validation: we will randomly remove 20% and then 40% of the data, to make sure the cluster will approximately maintain the same shapes.

[18]:
```python
from sklearn.model_selection import train_test_split

# removing 20% of the dat points randomly
X_filtered, _, _, _ = train_test_split(X, y, test_size=0.2, random_state=42)


# visualization
_, _ = k_means_visualization(k=k, X=X_filtered, init_centroids='kmeans++',␣
 ↪title="Test kmeans class (20% less data)")
```



[19]:
```
# removing 40% of the dat points randomly
X_filtered, _, _, _ = train_test_split(X, y, test_size=0.4, random_state=42)


# visualization
_, _ = k_means_visualization(k=k, X=X_filtered, init_centroids='kmeans++',␣
 ↪title="Test kmeans class (40% less data)")
```



1.5 Evaluation

Beijing and Shenyang will be the train and validation set: we can do training on a city and then validate it, and viceversa. Then, Guangzhou and Shanghai are 2 test sets.

[20]:
```
train_data = pd.concat([dfs['Beijing'], dfs['Shenyang']])
X = train_data.drop('PM_HIGH', axis=1)
y = train_data['PM_HIGH']

X.head()
```

[20]:

	season	DEWP	HUMI	PRES	TEMP	Iws	precipitation	cbwd_NE	cbwd_NW	\
0	4	-8.0	79.0	1026.0	-5.0	23.69	0.0	0	0	
1	4	-11.0	85.0	1021.0	-9.0	105.93	1.1	0	0	
2	4	-21.0	43.0	1030.0	-11.0	117.55	0.0	0	1	
3	4	-25.0	33.0	1034.0	-12.0	39.35	0.0	1	0	
4	4	-24.0	30.0	1034.0	-10.0	59.00	0.0	1	0	

	cbwd_SE
0	1

```
1          1
2          0
3          0
4          0
```

[21]:
```
from sklearn.metrics import classification_report

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
kmeans = KMeans(n_centers=2, init_centroids='kmeans++', random_seed=42)
kmeans.fit(X_train_scaled)
```

[21]:
```
KMeans(init_centroids='kmeans++', random_seed=42)
```

assessing the quality of the clustering with the internal cluster scoring:

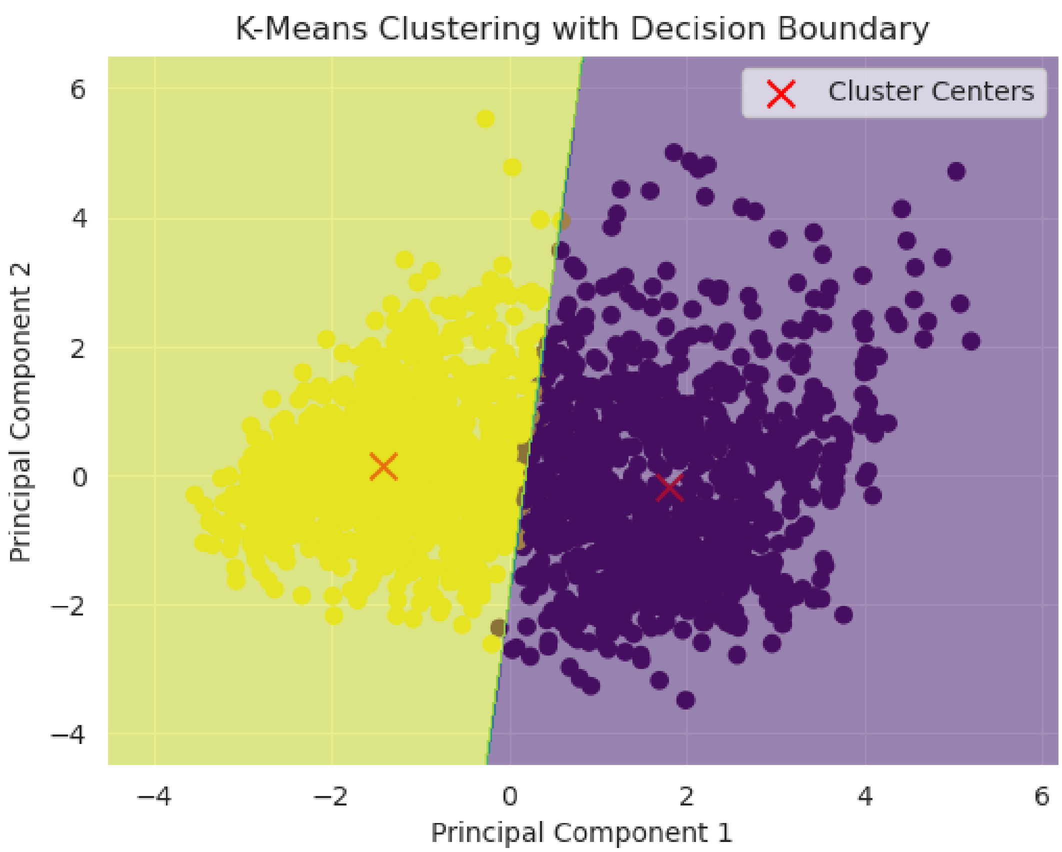
[22]:
```
print(f'Inertia score: {kmeans.score(X_train_scaled)}')
print(f'Silhouette score: {kmeans.score(X_train_scaled,␣
 ↪score_metric="silhouette")}')
```

```
Inertia score: 17196.824183523437
Silhouette score: 0.26843214299414797
```

Plot of the boundaries by applying PCA dimensionality reduction, to justify the score obtained from the Silhouette score.

[23]:
```
from sklearn.decomposition import PCA

kmeans = KMeans(n_centers=2, random_seed=42)
kmeans.fit(X_train_scaled)

# Perform PCA for dimensionality reduction
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_train_scaled)
labels = kmeans.labels_

# Plot the data points with color-coded cluster labels
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels, cmap='viridis')

# Plot the cluster centers
cluster_centers_pca = pca.transform(kmeans.centroids_)
plt.scatter(cluster_centers_pca[:, 0], cluster_centers_pca[:, 1],
            marker='x', color='red', s=100, label='Cluster Centers')
```

```python
# Plot the decision boundary
h = 0.02  # step size of the meshgrid
x_min, x_max = X_pca[:, 0].min() - 1, X_pca[:, 0].max() + 1
y_min, y_max = X_pca[:, 1].min() - 1, X_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = kmeans.predict(pca.inverse_transform(np.c_[xx.ravel(), yy.ravel()]))
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.5, cmap='viridis')

# Add labels and title
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('K-Means Clustering with Decision Boundary')

# Display the plot
plt.legend()
plt.show()
```



Asses the similarity with the ground-truth using external cluster scoring:

[24]:
```
print(f'Normalized Mutual Info score: {kmeans.score(y=y_train,␣
↪score_metric="nmi")}')
print(f'Adjusted Rand Index score: {kmeans.score(y=y_train,␣
↪score_metric="ari")}')
```

```
Normalized Mutual Info score: 0.0028264788660817075
Adjusted Rand Index score: -0.003225155254636967
```

classification scoring metrics on train set:

[25]:
```
y_pred_train = kmeans.predict(X_train_scaled)
print(classification_report(y_train, y_pred_train))
```

```
              precision    recall  f1-score   support

         0.0       0.76      0.46      0.57      1687
         1.0       0.30      0.61      0.40       629

    accuracy                           0.50      2316
   macro avg       0.53      0.53      0.48      2316
weighted avg       0.63      0.50      0.52      2316
```

scoring metrics on validation set:

[26]:
```
y_pred_val = kmeans.predict(X_test_scaled)
print("Classification scores:")
print(classification_report(y_test, y_pred_val))
```

```
Classification scores:
              precision    recall  f1-score   support

         0.0       0.74      0.45      0.56       412
         1.0       0.31      0.61      0.41       167

    accuracy                           0.50       579
   macro avg       0.53      0.53      0.49       579
weighted avg       0.62      0.50      0.52       579
```

Now KMeans will be evaluated on the 2 test set (Guangzhou and Shanghai).

[27]:
```
X_guangzhou = dfs['Guangzhou']
X_guangzhou = X_guangzhou.drop('PM_HIGH', axis=1)
X_guangzhou_scaled = scaler.transform(X_guangzhou)
y_guangzhou = dfs['Guangzhou']['PM_HIGH']
y_pred_guangzhou = kmeans.predict(X_guangzhou_scaled)
print(classification_report(y_guangzhou, y_pred_guangzhou))
```

```
              precision    recall  f1-score   support
```

```
         0.0       0.95      0.10      0.18      1266
         1.0       0.07      0.93      0.12        86

    accuracy                           0.15      1352
   macro avg       0.51      0.52      0.15      1352
weighted avg       0.90      0.15      0.18      1352
```

[28]:

```python
X_shanghai = dfs['Shanghai']
X_shanghai = X_shanghai.drop('PM_HIGH', axis=1)
X_shanghai_scaled = scaler.transform(X_shanghai)
y_shanghai = dfs['Shanghai']['PM_HIGH']
y_pred_shanghai = kmeans.predict(X_shanghai_scaled)
print(classification_report(y_shanghai, y_pred_shanghai))
```

```
              precision    recall  f1-score   support

         0.0       0.80      0.27      0.41      1218
         1.0       0.05      0.36      0.09       133

    accuracy                           0.28      1351
   macro avg       0.42      0.32      0.25      1351
weighted avg       0.72      0.28      0.38      1351
```