MLPy Workshop 3

Alexandru Girban (S2148980), Hariaksh Pandya (S2692608)

January 29, 2025

1 Week 3: Clustering

In this workshop, we will work through a set of problems clustering, another cannonical form of unsupervised learning. Clustering is an important tool that is used to discover homogeneous groups of data points within a heterogeneous population. It can be the main goal in some problems, while in others it may be used in EDA to understand the main types of behavior in the data or in feature engineering.

We will start by generating some artificial data, and then we will utilize clustering algorithms described in lectures and explore the impact of feature engineering on the solution. We will then attempt to find clusters in a gene expression dataset.

As usual, the worksheets will be completed in teams of 2-3, using **pair programming**, and we have provided cues to switch roles between driver and navigator. When completing worksheets:

- You will have tasks tagged by (CORE) and (EXTRA).
- Your primary aim is to complete the (CORE) components during the WS session, afterwards you can try to complete the (EXTRA) tasks for your self-learning process.
- Look for the as cue to switch roles between driver and navigator.

Instructions for submitting your workshops can be found at the end of worksheet. As a reminder, you must submit a pdf of your notebook on Learn by 16:00 PM on the Friday of the week the workshop was given.

As you work through the problems it will help to refer to your lecture notes (navigator). The exercises here are designed to reinforce the topics covered this week. Please discuss with the tutors if you get stuck, even early on!

1.1 Outline

1. Problem Definition and Setup: Simulated Example
2. K-means: Simulated Example
3. Hierarchical Clustering: Simulated Example
4. Gene Expression Data
5. Hierarchical Clustering: Gene Expression Data
6. K-means Clustering: Gene Expression Data

2 Problem Definition and Setup: Simulated Example

2.1 Packages

First, lets load in some packages to get us started.

[3]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from sklearn.cluster import KMeans
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster import hierarchy
```

2.2 Data: Simulated Example

We will begin with a simple simulated example in which there are truly three clusters. We assume that there are $D = 2$ features and within each cluster, the data points are generated from a spherical normal distribution $N(\mathbf{m}_k, \sigma_k^2\mathbf{I})$ for clusters $k = 1, 2, 3$, where both the mean $\mathbf{m}_k$ and variance σ_k^2 are different across clusters. Specifically, we assume that:

- Cluster 1: contains $|C_1| = 500$ points with mean vector $\mathbf{m}_1 = \begin{pmatrix} 0 \\ 4 \end{pmatrix}$ with standard deviation $\sigma_1 = 2$.
- Cluster 2: contains $|C_2| = 250$ points with mean vector $\mathbf{m}_2 = \begin{pmatrix} 0 \\ -4 \end{pmatrix}$ with standard deviation $\sigma_2 = 1$.
- Cluster 3: contains $|C_3| = 100$ points with mean vector $\mathbf{m}_3 = \begin{pmatrix} -4 \\ 0 \end{pmatrix}$ with standard deviation $\sigma_3 = 0.5$.

Run the following code to generate the dataset described above.

[4]:
```
# Number of features
D = 2

# Cluster sizes
N_1 = 500
N_2 = 250
N_3 = 100

# Cluster means
m_1 = np.array([0., 4.])
m_2 = np.array([0., -4.])
m_3 = np.array([-4., 0.])

# Cluster standard deviations
sd_1 = 2.
sd_2 = 1.
```

```
sd_3 = 0.5

# Generate the data
rnd = np.random.RandomState(5)
X_1 = rnd.normal(loc = m_1, scale = sd_1, size = (N_1,D))
X_2 = rnd.normal(loc = m_2, scale = sd_2, size = (N_2,D))
X_3 = rnd.normal(loc = m_3, scale = sd_3, size = (N_3,D))
X = np.vstack((X_1, X_2, X_3))

# Save true cluster labels
cl = np.hstack((np.repeat(1,N_1),np.repeat(2,N_2),np.repeat(3,N_3)))
```

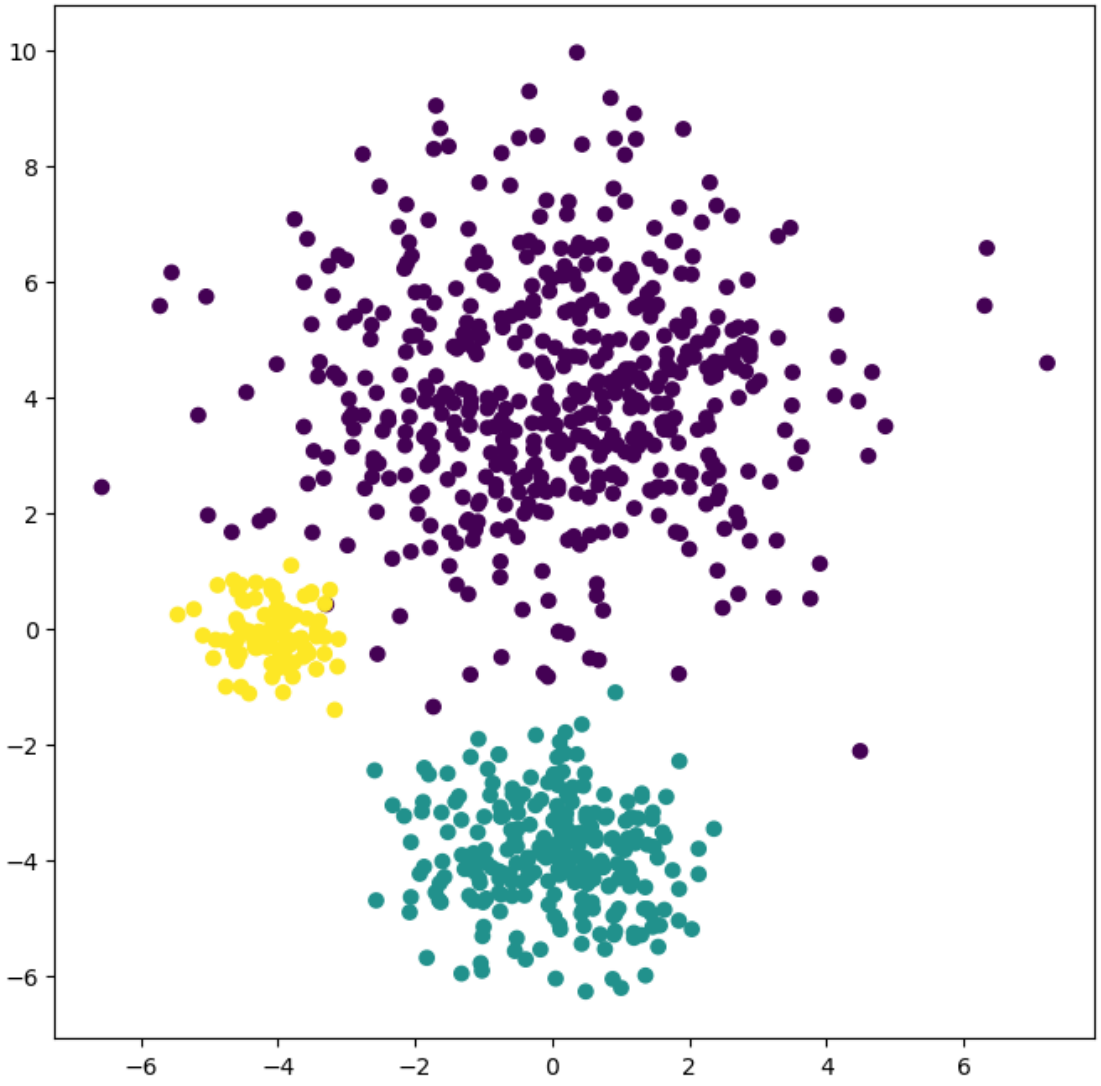
[3]:
```
# Check that the size is correct
print(X.shape)
```

```
(850, 2)
```

2.2.1 Exercise 1 (CORE)

Visualise the data and color by the true cluster labels.

[4]:
```
# Plot the data and color by cluster membership
plt.figure(figsize = (8, 8))
plt.scatter(X[:,0], X[:, 1], c = cl)
plt.show()
```



3 K-means Clustering: Simulated Example

To perform K-means clustering, we will use `KMeans()` in `sklearn.cluster`. Documentation is available here, and for an overview of clustering methods available in `sklearn`, see link. There are different inputs we can specify when calling `KMeans()` such as:

- `n_clusters`: the number of clusters.
- `init`: which specifies the intialization of the centroids, e.g. can be set to `k-means++` for K-means++ initialization or `random` for random initialization.
- `n_init`: which specifies the number of times the algorithm is run with different random initializations
- `random_state`: this can bet set to a fixed number to make results reproducible.

We can then use the `.fit()` method of `KMeans` to run the K-means algorithm on our data.

After fitting, some of the relevant attributes of interest include:

- `labels_`: cluster assignments of the data points.
- `cluster_centers_`: mean corresponding to each cluster, stored in a matrix of size: number of clusters K times number features D.
- `inertia_`: the total within-cluster variation.

3.0.1 Exercise 2 (CORE)

Let's start by exploring how the clustering changes across the K-means iterations. To do, set:

- number of clusters to 3
- initialization to random
- number of times the algorithm is run to 1
- fix the random seed to a number of your choice (e.g. 0)

a) Now, fit the K-means algorithms with different values of the maximum number of iterations fixed to 1,2,3, and the default value of 300.

b) Plot the clustering solution for the four different cases and comment on how it changes.

c) How many iterations are needed for the convergence?

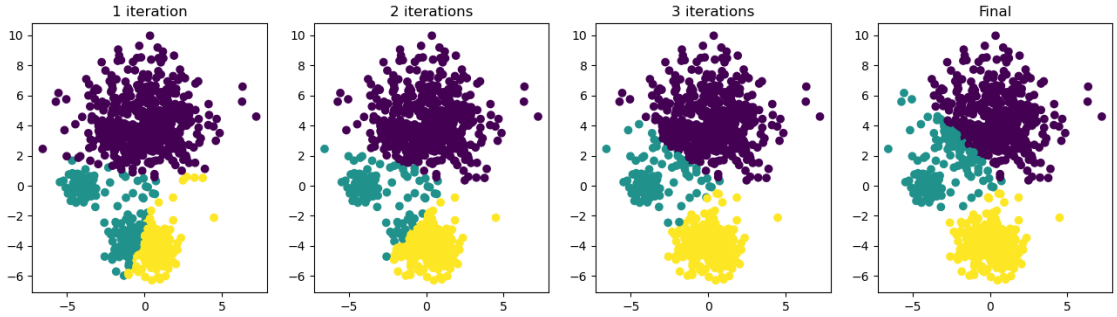
Hint

- To find the number of iterations, check the attributes of `KMeans`

[5]:
```
# Part a

# 1 iteration
km1 = KMeans(n_clusters = 3, init = 'random', n_init = 1, random_state = 0,␣
 ↪max_iter = 1)
km1.fit(X)

# 2 iterations
km2 = KMeans(n_clusters = 3, init = 'random', n_init = 1, random_state = 0,␣
 ↪max_iter = 2)
km2.fit(X)

# 3 iterations
km3 = KMeans(n_clusters = 3, init = 'random', n_init = 1, random_state = 0,␣
 ↪max_iter = 3)
km3.fit(X)

# defalt number of iterations
km = KMeans(n_clusters = 3, init = 'random', n_init = 1, random_state = 0)
km.fit(X)
```

[5]:
```
KMeans(init='random', n_clusters=3, n_init=1, random_state=0)
```

[9]:
```
# Part b
fig, ax = plt.subplots(1, 4, figsize = (16,4))
ax[0].scatter(X[:,0], X[:,1], c = km1.labels_)
ax[0].set_title('1 iteration')
ax[1].scatter(X[:,0], X[:,1], c = km2.labels_)
ax[1].set_title('2 iterations')
ax[2].scatter(X[:,0], X[:,1], c = km3.labels_)
ax[2].set_title('3 iterations')
ax[3].scatter(X[:,0], X[:,1], c = km.labels_)
ax[3].set_title('Final')
plt.show()
```



[6]:
```
# Part c
km.n_iter_
```

[6]:
```
11
```

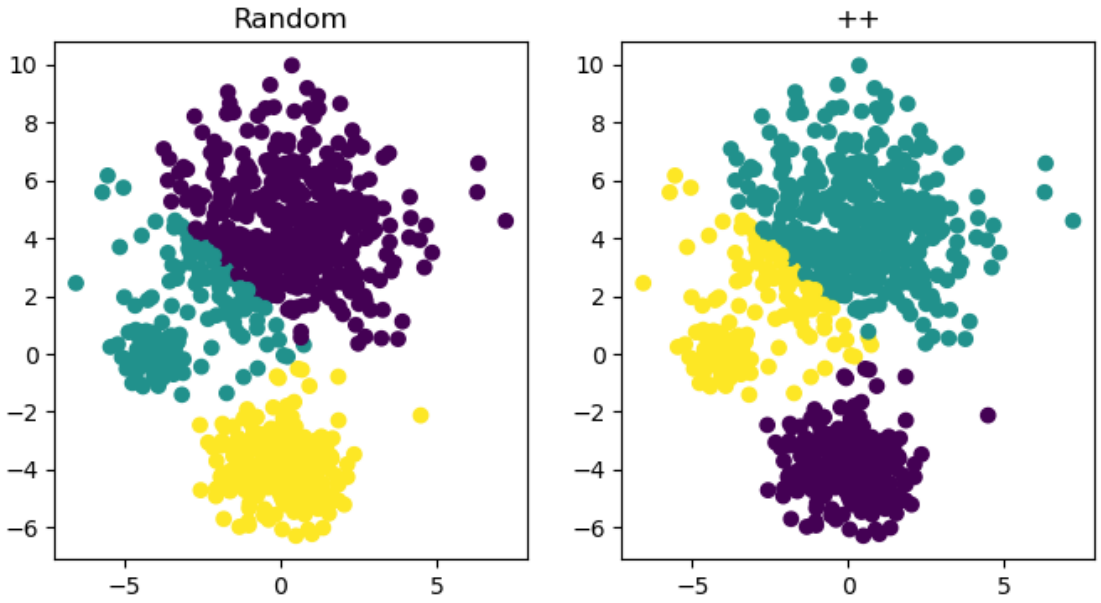
There are 11 iterations required for convergence.

3.0.2 Exercise 3 (CORE)

Next, compare the random intialization with K-means++ (in this case fix the number of different initializations to 10). Plot both clustering solutions. Which requires fewer iterations? and which provides a lower within-cluster variation?

[11]:
```
# random
km = KMeans(n_clusters = 3, init = 'random', n_init = 10, random_state = 0)
km.fit(X)

# ++
kmpp = KMeans(n_clusters = 3, init = 'k-means++', n_init = 10, random_state = 0)
kmpp.fit(X)

# Plot the clustering solutions
fig, ax = plt.subplots(1, 2, figsize = (8,4))
ax[0].scatter(X[:,0], X[:,1], c = km.labels_)
```

```
ax[0].set_title('Random')
ax[1].scatter(X[:,0], X[:,1], c = kmpp.labels_)
ax[1].set_title('++')
plt.show()

# Print the within cluster variation
print('Within cluster variation for Random: ', km.inertia_)
print('Within cluster variation for ++: ', kmpp.inertia_)

# Print the number of iterations required
print('Number of iterations for Random: ', km.n_iter_)
print('Number of iterations for ++: ', kmpp.n_iter_)
```




```
Within cluster variation for Random:  3833.3076162106936
Within cluster variation for ++:  3833.282646884114
Number of iterations for Random:  11
Number of iterations for ++:  4
```

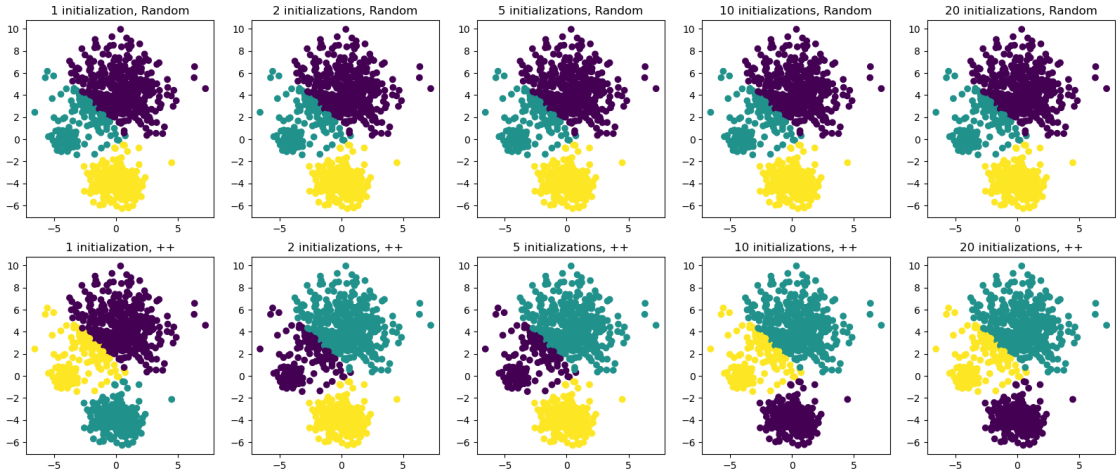
We note that ++ requires a much lower number of iterations than Random (4 vs. 11), and it also produces a configuration with slightly lower within cluster variation.

3.0.3 Exercise 4 (CORE)

Find the clustering solution using a different number of initializations equal to 1, 2, 5, 10, and 20. Visualize the results and print the within-cluster variation. Based on the results, how many intializations are needed? Try changing the random state; how does that change your conclusions?

[14]:
```
# random
km1 = KMeans(n_clusters = 3, init = 'random', n_init = 1, random_state = 0)
km1.fit(X)

km2 = KMeans(n_clusters = 3, init = 'random', n_init = 2, random_state = 0)
km2.fit(X)

km5 = KMeans(n_clusters = 3, init = 'random', n_init = 5, random_state = 0)
km5.fit(X)

km10 = KMeans(n_clusters = 3, init = 'random', n_init = 10, random_state = 0)
km10.fit(X)

km20 = KMeans(n_clusters = 3, init = 'random', n_init = 20, random_state = 0)
km20.fit(X)



# ++
kmpp1 = KMeans(n_clusters = 3, init = 'k-means++', n_init = 1, random_state = 0)
kmpp1.fit(X)

kmpp2 = KMeans(n_clusters = 3, init = 'k-means++', n_init = 2, random_state = 0)
kmpp2.fit(X)

kmpp5 = KMeans(n_clusters = 3, init = 'k-means++', n_init = 5, random_state = 0)
kmpp5.fit(X)

kmpp10 = KMeans(n_clusters = 3, init = 'k-means++', n_init = 10, random_state =␣
↪0)
kmpp10.fit(X)

kmpp20 = KMeans(n_clusters = 3, init = 'k-means++', n_init = 20, random_state =␣
↪0)
kmpp20.fit(X)

# Visualizations
fig, ax = plt.subplots(2, 5, figsize = (20,8))
ax[0][0].scatter(X[:,0], X[:,1], c = km1.labels_)
ax[0][0].set_title('1 initialization, Random')
ax[0][1].scatter(X[:,0], X[:,1], c = km2.labels_)
ax[0][1].set_title('2 initializations, Random')
ax[0][2].scatter(X[:,0], X[:,1], c = km5.labels_)
ax[0][2].set_title('5 initializations, Random')
ax[0][3].scatter(X[:,0], X[:,1], c = km10.labels_)
ax[0][3].set_title('10 initializations, Random')
ax[0][4].scatter(X[:,0], X[:,1], c = km20.labels_)
ax[0][4].set_title('20 initializations, Random')
```

```
ax[1][0].scatter(X[:,0], X[:,1], c = kmpp1.labels_)
ax[1][0].set_title('1 initialization, ++')
ax[1][1].scatter(X[:,0], X[:,1], c = kmpp2.labels_)
ax[1][1].set_title('2 initializations, ++')
ax[1][2].scatter(X[:,0], X[:,1], c = kmpp5.labels_)
ax[1][2].set_title('5 initializations, ++')
ax[1][3].scatter(X[:,0], X[:,1], c = kmpp10.labels_)
ax[1][3].set_title('10 initializations, ++')
ax[1][4].scatter(X[:,0], X[:,1], c = kmpp20.labels_)
ax[1][4].set_title('20 initializations, ++')
plt.show()

# Within-cluster variations
print('Within cluster variation for 1 initialization, Random: ', km1.inertia_)
print('Within cluster variation for 2 initializations, Random: ', km2.inertia_)
print('Within cluster variation for 5 initializations, Random: ', km5.inertia_)
print('Within cluster variation for 10 initializations, Random: ', km10.
 ↪inertia_)
print('Within cluster variation for 20 initializations, Random: ', km20.
 ↪inertia_)

print('Within cluster variation for 1 initialization, ++: ', kmpp1.inertia_)
print('Within cluster variation for 2 initializations, ++: ', kmpp2.inertia_)
print('Within cluster variation for 5 initializations, ++: ', kmpp5.inertia_)
print('Within cluster variation for 10 initializations, ++: ', kmpp10.inertia_)
print('Within cluster variation for 20 initializations, ++: ', kmpp20.inertia_)
```




```
Within cluster variation for 1 initialization, Random:  3833.3076162106936
Within cluster variation for 2 initializations, Random:  3833.3076162106936
Within cluster variation for 5 initializations, Random:  3833.3076162106936
```

```
Within cluster variation for 10 initializations, Random:  3833.3076162106936
Within cluster variation for 20 initializations, Random:  3833.3076162106936
Within cluster variation for 1 initialization, ++:  3833.367971404591
Within cluster variation for 2 initializations, ++:  3833.3076162106936
Within cluster variation for 5 initializations, ++:  3833.3076162106936
Within cluster variation for 10 initializations, ++:  3833.282646884114
Within cluster variation for 20 initializations, ++:  3833.282646884114
```

First of all, we note that for Random initializations, we obtain the same k-means configurations and thus the same within cluster variation regardless of the value of n_init we use. Hence, a value of n_init = 1 will suffice in this situation. However, when we switch from Random to ++, we notice that the value of n_init starts to matter, with higher values yielding slightly better solutions, up to a value of n_init = 10.

Now, is a good point to switch driver and navigator

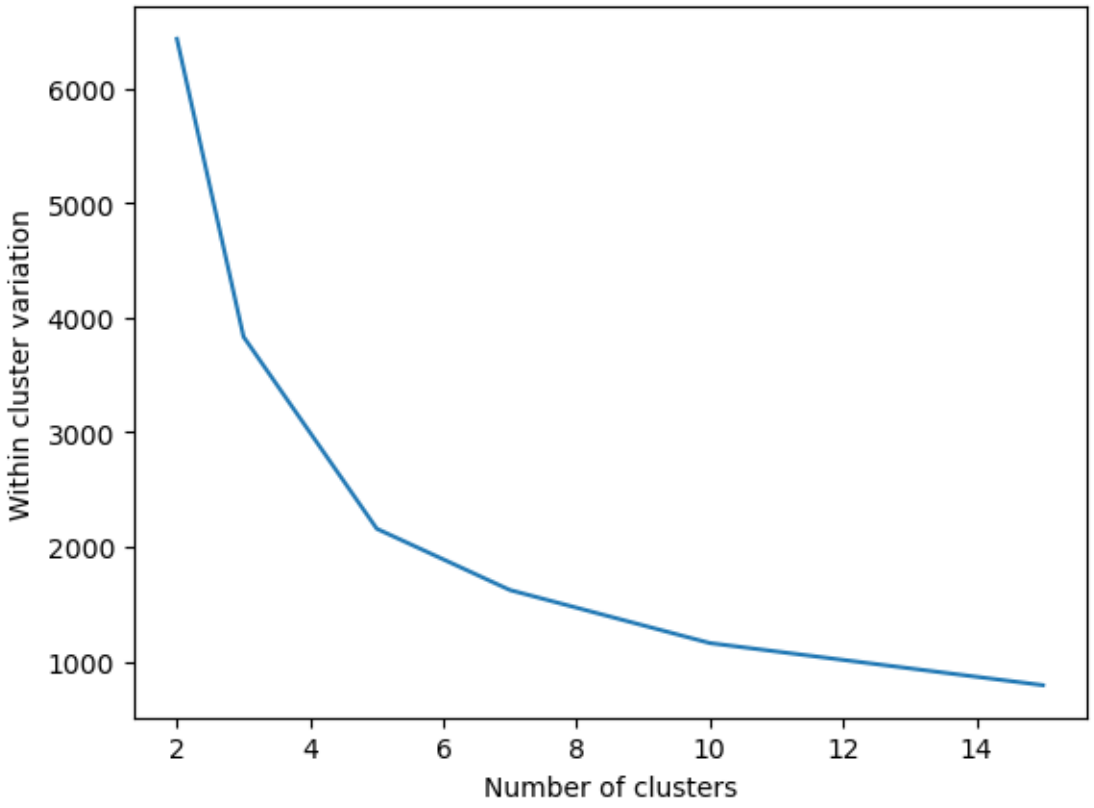
3.0.4 Exercise 5 (CORE)

Since we simulated the data, we know the true number of clusters. However, in practice this number is rarely known. Find the K-means solution with different choices of K and plot the within-cluster variation as a function of K. Comment on the results.

[15]:
```
# Choices for K: 2, 3, 5, 7, 10, 15

km2 = KMeans(n_clusters = 2, init = 'random', n_init = 10, random_state = 0)
km2.fit(X)

km3 = KMeans(n_clusters = 3, init = 'random', n_init = 10, random_state = 0)
km3.fit(X)

km5 = KMeans(n_clusters = 5, init = 'random', n_init = 10, random_state = 0)
km5.fit(X)

km7 = KMeans(n_clusters = 7, init = 'random', n_init = 10, random_state = 0)
km7.fit(X)

km10 = KMeans(n_clusters = 10, init = 'random', n_init = 10, random_state = 0)
km10.fit(X)

km15 = KMeans(n_clusters = 15, init = 'random', n_init = 10, random_state = 0)
km15.fit(X)

# Visualize the solutions
fig, ax = plt.subplots(1, 6, figsize = (24,4))
ax[0].scatter(X[:,0], X[:,1], c = km2.labels_)
ax[0].set_title('2 clusters')
ax[1].scatter(X[:,0], X[:,1], c = km3.labels_)
ax[1].set_title('3 clusters (True)')
ax[2].scatter(X[:,0], X[:,1], c = km5.labels_)
```

```
ax[2].set_title('5 clusters')
ax[3].scatter(X[:,0], X[:,1], c = km7.labels_)
ax[3].set_title('7 clusters')
ax[4].scatter(X[:,0], X[:,1], c = km10.labels_)
ax[4].set_title('10 clusters')
ax[5].scatter(X[:,0], X[:,1], c = km15.labels_)
ax[5].set_title('15 clusters')
plt.show()
```

[16]:
```
# Plotting within cluster variation as a function of K
x = [2, 3, 5, 7, 10, 15]
y = [km2.inertia_, km3.inertia_, km5.inertia_, km7.inertia_, km10.inertia_,␣
 ↪km15.inertia_]

plt.plot(x, y)
plt.xlabel("Number of clusters")
plt.ylabel("Within cluster variation")
plt.show()
```



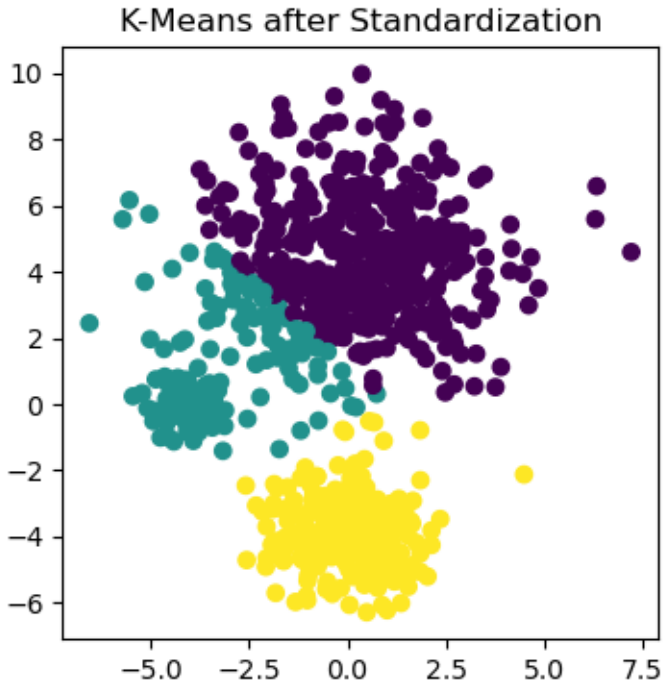
Note that increasing the number of clusters decreases the within cluster variation, with the most significant decreases in within cluster variation occuring when we go from 2 to 3 clusters, and from 3 to 5 clusters. Increasing the number of clusters past 10 has minimal impact on decreasing the within cluster variation.

3.0.5 Exercise 6 (CORE)

Now standardize the data and re-run the K-means algorithm. Qualitatively, how has standardising the data impacted performance? Can you argue why you observe what you see?

[16]:
```
# Standardizing X
X_standard = (X - X.mean())/X.std()

# K-Means
km1 = KMeans(n_clusters = 3, init = 'random', n_init = 1, random_state = 0)
km1.fit(X_standard)

# Visualize
fig, ax = plt.subplots(1, 1, figsize = (4,4))
ax.scatter(X[:,0], X[:,1], c = km1.labels_)
ax.set_title('K-Means after Standardization')
plt.show()
```

```
print('Within cluster variation: ', km1.inertia_)
print('Number of iterations: ', km1.n_iter_)
```




```
Within cluster variation:  362.3745077044642
Number of iterations:  11
```

As it is obviously to be expected, we have significantly reduced within cluster variation after standardizing since we have reduced the absolute values of the numerical data we are clustering. The number of iterations (11) appears to remain unchanged. The clustes are extremely similar geometrically to the original, because the original data was generated from normal distributions.

4 Hierarchical Clustering: Simulated Example

To perform hierarchical clustering, we will use the `linkage()` function from `scipy.cluster.hierarchy`. The inputs to specify include

- the data.
- `metric`: specifies the dissimarlity between data points. Defaults to the Euclidean distance.
- `method`: specifies the type of linkage, e.g. complete, single, or average.

Then, we can use `dendrogram()` from `scipy.cluster.hierarchy` to plot the dendrogram.

Note that you can also use `AgglomerativeClustering` from `sklearn.cluster`, which similarly has options for `metric` to specify the distance and `linkage` to specify the type of linkage. However,

`sklearn` does not have its own functions for plotting the dendogram and use must use the tools from `scipy.cluster.hierarchy`.

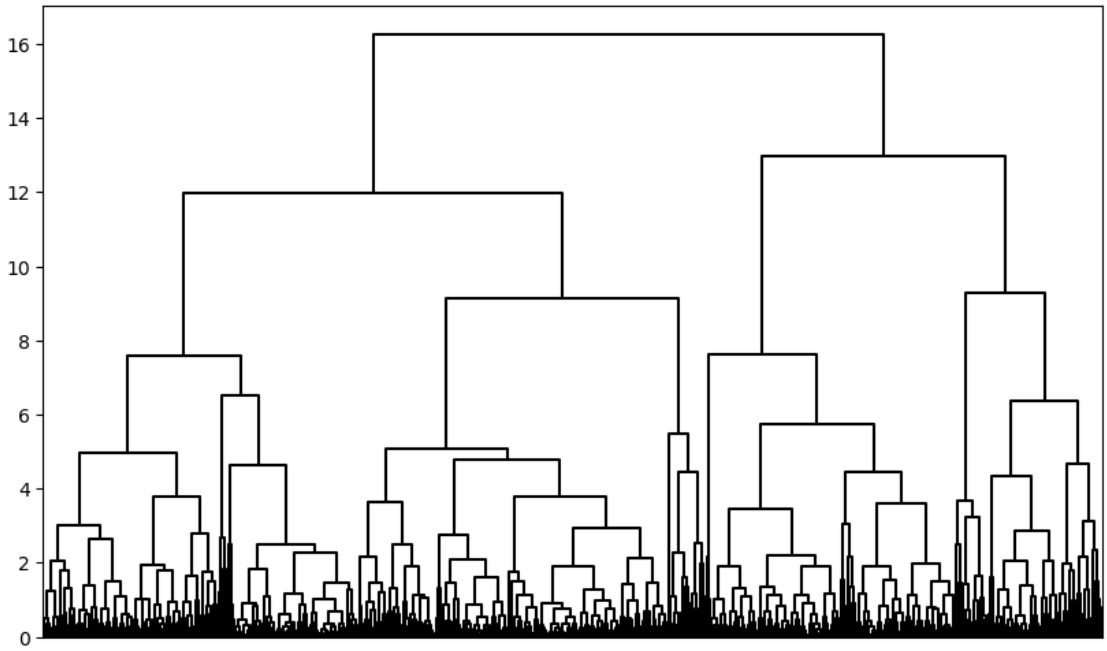
4.0.1 Exercise 7 (CORE)

a) Use hierarchical clustering with complete linkage and the Euclidean distance to cluster the simulated data. Name the object `hc_comp`.

[8]:
```
# The name of the returned object from hierarchy.linkage should be hc_comp for␣
↪part b
hc_comp = hierarchy.linkage(X, method = 'complete')
```

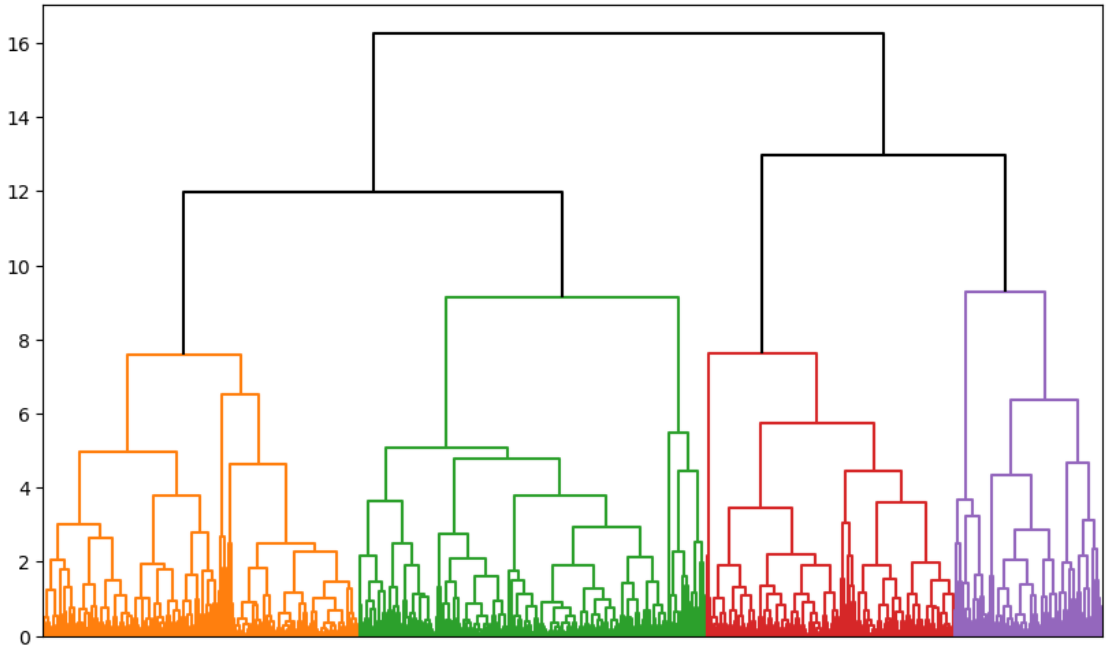
b) Plot the dendogram by running the code below. Try changing the 'color_threshold' to a number (e.g. 11) to color the branches of the tree below the threshold with different colors, thus, identifying the clusters if the tree were to be cut at that threshold.

[9]:
```
# Plot the dendrogram
cargs = {'color_threshold': -np.inf,'above_threshold_color':'black'}
fig, ax = plt.subplots(1, 1, figsize=(10,6))
hierarchy.dendrogram(hc_comp, ax=ax, **cargs, no_labels=True)
plt.show()
```



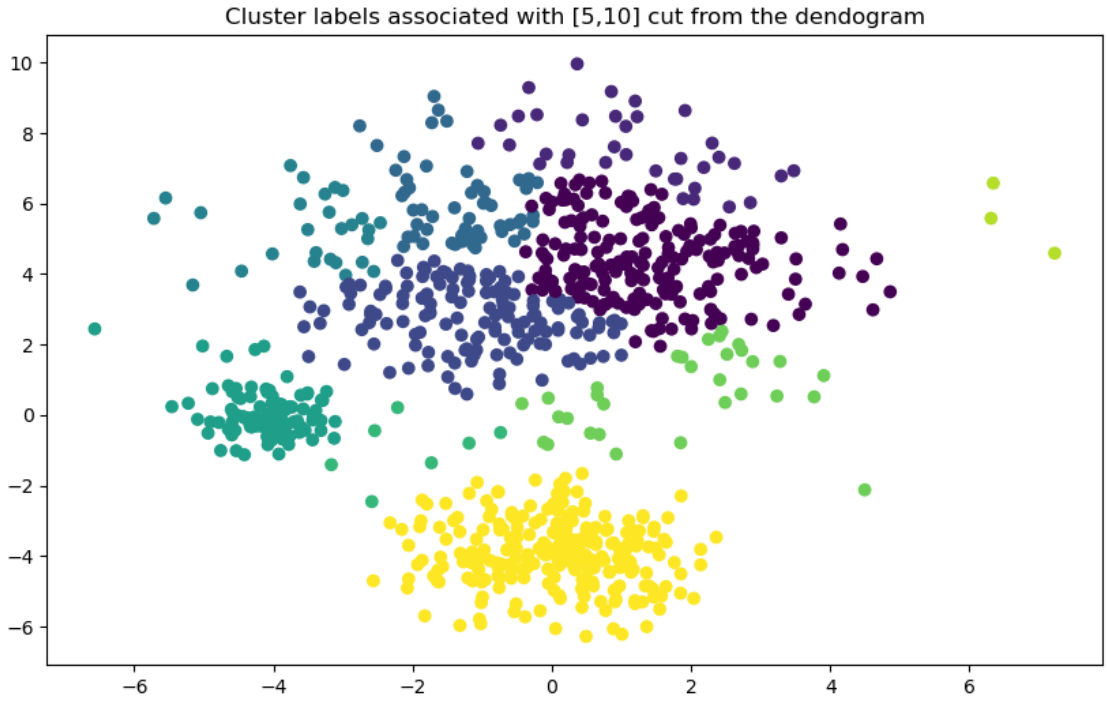
[11]:
```
# Try to change the color threshold to visualize clusters obtained by cutting␣
↪the tree
# Plot the dendrogram
cargs = {'color_threshold': 11,'above_threshold_color':'black'}
```

```
fig, ax = plt.subplots(1, 1, figsize=(10,6))
hierarchy.dendrogram(hc_comp, ax=ax, **cargs, no_labels=True)
plt.show()
```



c) Now, use the function `cut_tree()` from `scipy.cluster.hierarchy` to determine the cluster labels associated with a given cut of the dendrogram. You can either specify the number of clusters via `n_clusters` or the height/threshold at which to cut via `height`. Plot the data colored by cluster membership.

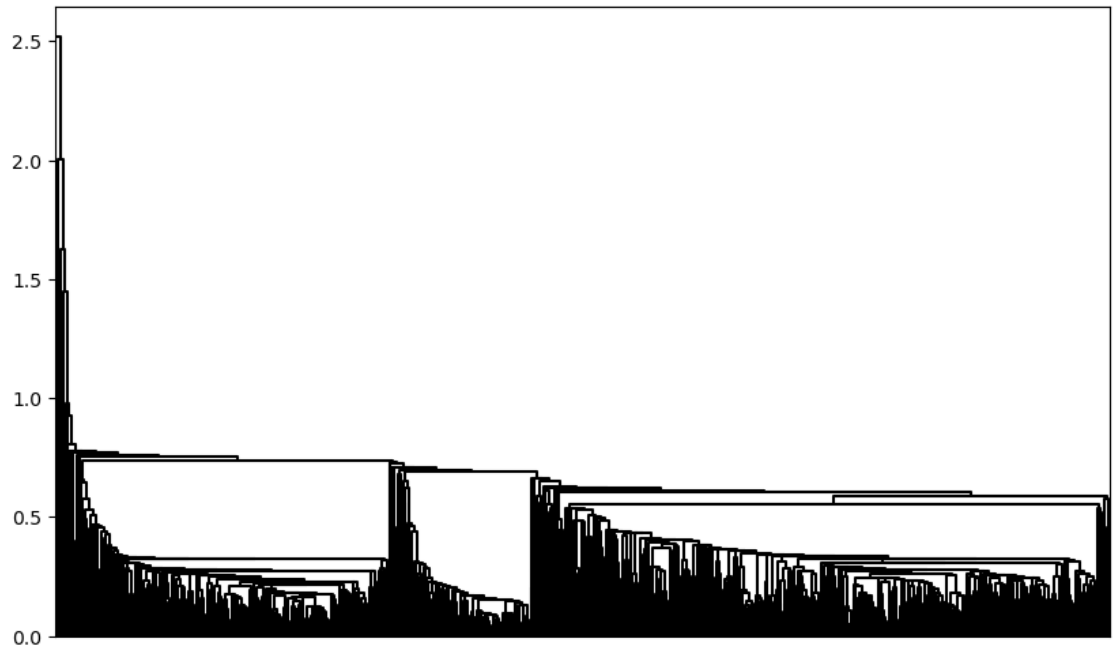
[18]:
```
# Cut the tree at a specified number of clusters
cuttree = hierarchy.cut_tree(hc_comp, n_clusters = [5,10])
fig, ax = plt.subplots(1, 1, figsize=(10,6))
ax.scatter(X[:,0], X[:,1], c = cuttree[:,1])
ax.set_title('Cluster labels associated with [5,10] cut from the dendogram')
plt.show()
```



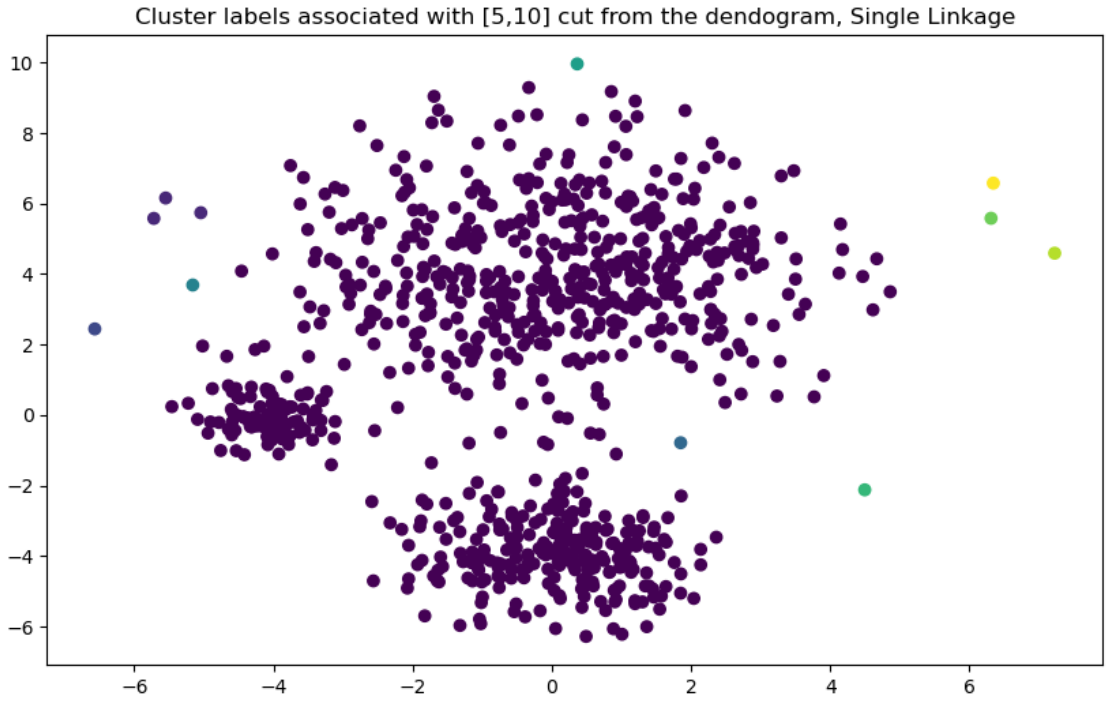
4.0.2 Exercise 8 (CORE)

Now try changing the linkage to single and average. Does this affect on the results?

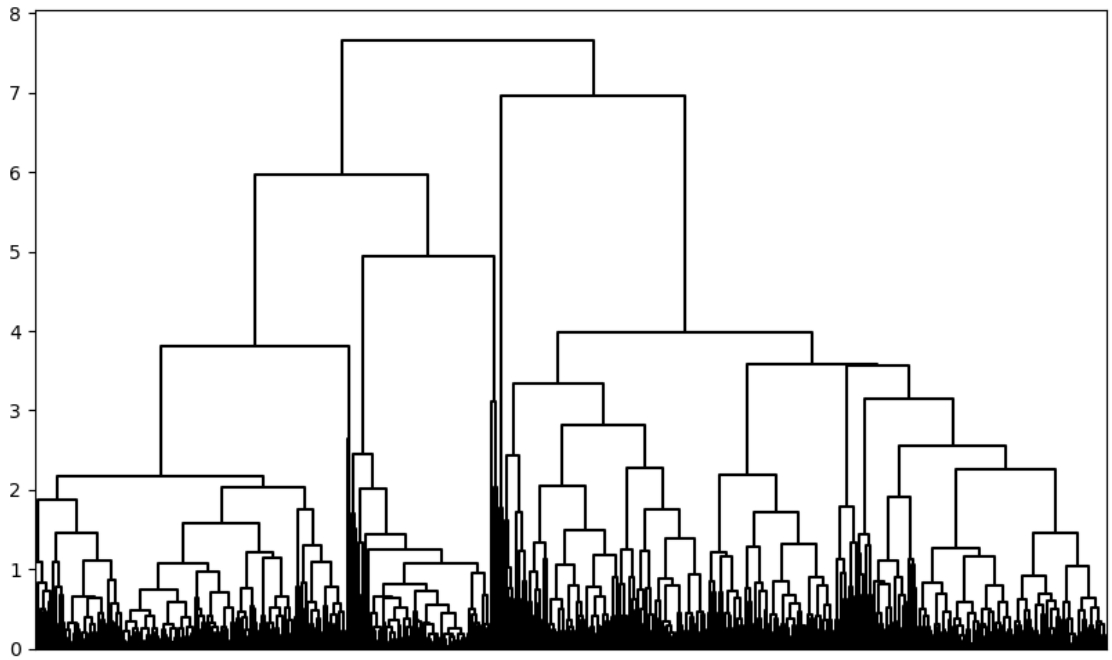
[20]:
```
# Change to single linkage, plot the dendrogram, and visualize the clustering␣
  ↪solution by cutting the tree
hc_single = hierarchy.linkage(X, method = 'single')

# Plot the dendrogram
cargs = {'color_threshold': -np.inf,'above_threshold_color':'black'}
fig, ax = plt.subplots(1, 1, figsize=(10,6))
hierarchy.dendrogram(hc_single, ax=ax, **cargs, no_labels=True)
plt.show()
```



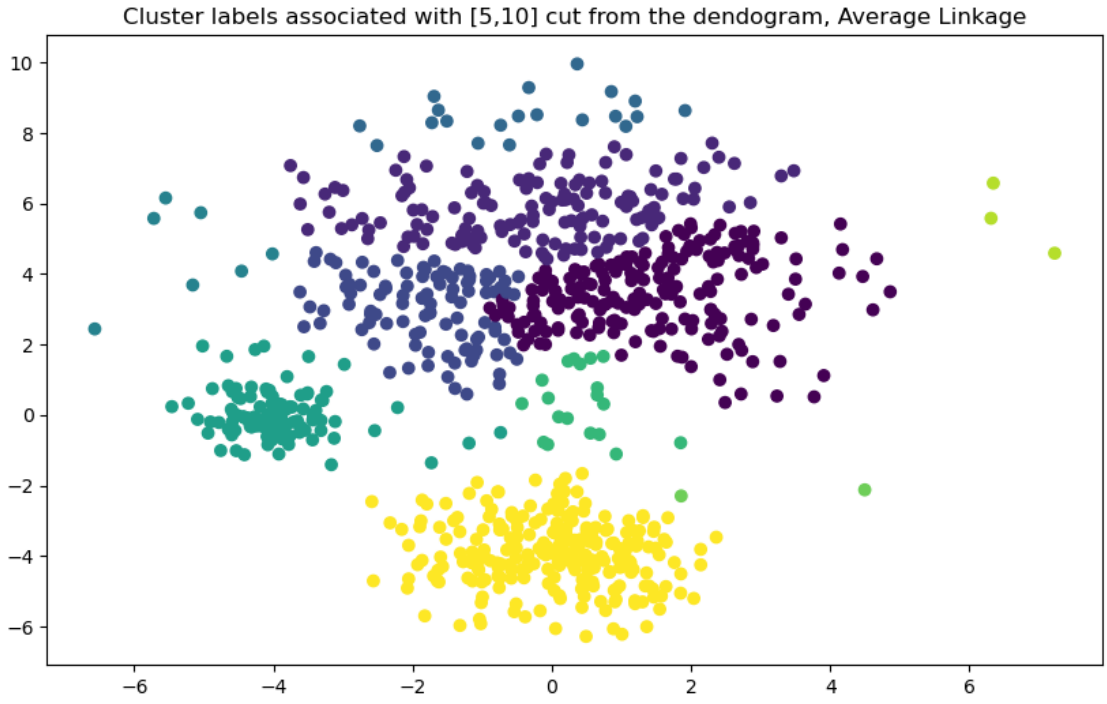
[22]:
```
# Cut the tree at a specified number of clusters
cuttree = hierarchy.cut_tree(hc_single, n_clusters = [5,10])
fig, ax = plt.subplots(1, 1, figsize=(10,6))
ax.scatter(X[:,0], X[:,1], c = cuttree[:,1])
ax.set_title('Cluster labels associated with [5,10] cut from the dendogram,␣
 ↪Single Linkage')
plt.show()
```



[21]:
```
# Change to average linkage, plot the dendrogram, and visualize the clustering␣
 ↪solution by cutting the tree
hc_average = hierarchy.linkage(X, method = 'average')

# Plot the dendrogram
cargs = {'color_threshold': -np.inf,'above_threshold_color':'black'}
fig, ax = plt.subplots(1, 1, figsize=(10,6))
hierarchy.dendrogram(hc_average, ax=ax, **cargs, no_labels=True)
plt.show()
```



[24]:
```
# Cut the tree at a specified number of clusters
cuttree = hierarchy.cut_tree(hc_average, n_clusters = [5,10])
fig, ax = plt.subplots(1, 1, figsize=(10,6))
ax.scatter(X[:,0], X[:,1], c = cuttree[:,1])
ax.set_title('Cluster labels associated with [5,10] cut from the dendogram,␣
 ↪Average Linkage')
plt.show()
```



Note that changing the linkage produces completely different dendograms. The one for single linkage looks completely different. In terms of the generated plots after we cut the trees, for single linkage we obtain one big central cluster, and outliers are differentiated, while the plot for average linkage looks similar to the one obtained for complete linkage.

Now, is a good point to switch driver and navigator

5 Gene Expression Data

Now, we will consider a more complex real dataset with a larger feature space.

The dataset is the **NCI cancer microarray dataset** discussed in both *Introduction to Statistical Learning* and *Elements of Statistical Learning*. The dataset consists of $D = 6830$ gene expression measurements for each of $N = 64$ cancer cell lines. The aim is to determine whether there are groups among the cell lines with similar gene expression patterns. This is an example of a high-dimensional dataset with D much larger than N, which makes visualization difficult. The $N = 64$ cancer cell lines have been obtained from samples of cancerous tisses, corresponding to 14 different types of cancer. However, our focus remains unsupervised learning and we will use the cancer labels only to plot.

We first need to read in the dataset.

[25]:
```
#Fetch the data and cancer labels
url_data = 'https://web.stanford.edu/~hastie/ElemStatLearn/datasets/nci.data.
  ↪csv'
```

```
url_labels = 'https://web.stanford.edu/~hastie/ElemStatLearn/datasets/nci.label.
 ↪txt'

X = pd.read_csv(url_data)
y = pd.read_csv(url_labels, header=None)

# clean data and follow convention in the notes that features are columns:
X = X.drop(labels='Unnamed: 0', axis=1).T
```

[26]:
```
X.shape
```

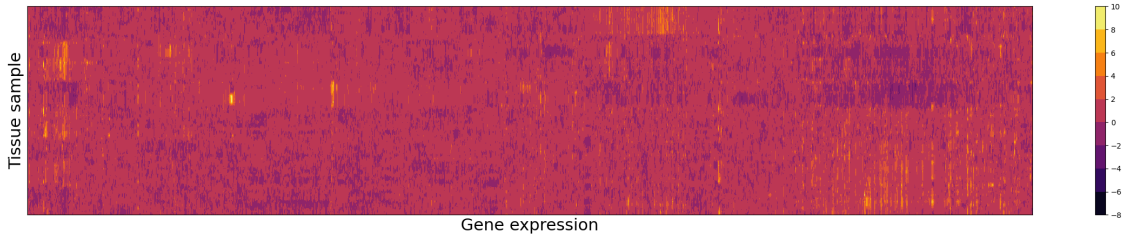
[26]: (64, 6830)

[27]:
```
y.shape
```

[27]: (64, 1)

Let's visualise the data with a contour plot.

[28]:
```
# Contour plot of the gene expression data
fig = plt.figure(figsize=(30,5))
ax = fig.add_subplot(111)
contours = ax.contourf(X, cmap='inferno')#, vmax=4, vmin=-4)
cbar = plt.colorbar(contours)
ax.set_xticks([])
ax.set_yticks([])
ax.set_xlabel("Gene expression", fontsize=22)
ax.set_ylabel("Tissue sample", fontsize=22)
plt.show()
```



We now convert our pandas dataframe into a numpy array and create integer labels for cancer type (for plotting purposes)

If you visualise the labels, you will notice there are lots of inconsistencies with white space etc. Run the following code to clean the labels.

[29]:
```
# Print the unique labels and counts
y.value_counts()
```

[29]:
```
0
OVARIAN                     4
RENAL                       4
MELANOMA                    3
COLON                       3
NSCLC                       3
BREAST                      2
CNS                         2
BREAST                      2
NSCLC                       2
NSCLC                       2
RENAL                       2
PROSTATE                    2
CNS                         2
MELANOMA                    2
LEUKEMIA                    2
COLON                       2
MELANOMA                    2
BREAST                      1
BREAST                      1
BREAST                      1
CNS                         1
LEUKEMIA                    1
COLON                       1
COLON                       1
MCF7D-repro                 1
LEUKEMIA                    1
LEUKEMIA                    1
MCF7A-repro                 1
LEUKEMIA                    1
K562A-repro                 1
K562B-repro                 1
NSCLC                       1
OVARIAN                     1
NSCLC                       1
MELANOMA                    1
OVARIAN                     1
RENAL                       1
RENAL                       1
RENAL                       1
UNKNOWN                     1
Name: count, dtype: int64
```

[30]:
```python
# Clean the labels by stripping the white space
y_clean = np.asarray(y).flatten()
for j in range(y_clean.size):
    y_clean[j] = y_clean[j].strip()
```

```
cancer_types = list(np.unique(y_clean))
cancer_groups = np.array([cancer_types.index(lab) for lab in y_clean])
```

[31]:
```
print(cancer_types)
```

```
['BREAST', 'CNS', 'COLON', 'K562A-repro', 'K562B-repro', 'LEUKEMIA',
'MCF7A-repro', 'MCF7D-repro', 'MELANOMA', 'NSCLC', 'OVARIAN', 'PROSTATE',
'RENAL', 'UNKNOWN']
```

[33]:
```
X_array = np.asarray(X)
```

5.0.1 Exercise 9 (EXTRA)

Perform a PCA of **X** to visualize the data. Plot the first few principal component scores and color by cancer type. Do cell lines within the same cancer types seems to have similar scores? Make a scree plot of the proportion of variance explained. How many components does this suggest?

[167]:
```
from sklearn.decomposition import PCA
```

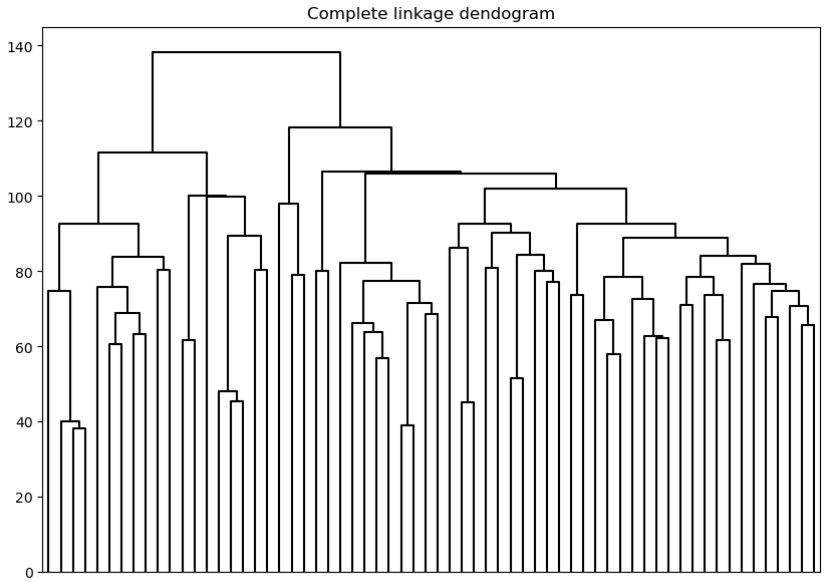
6 Hierarchical Clustering: Gene Expression Data

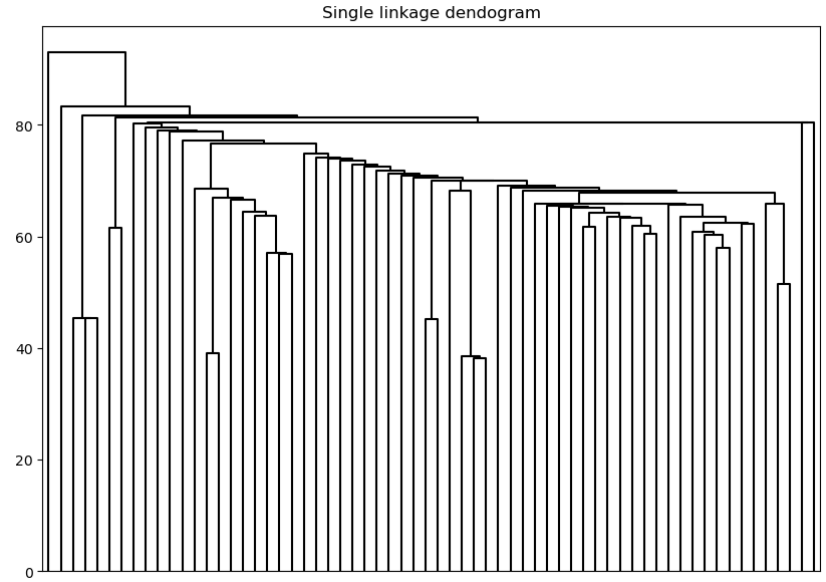
Now, let's perform hierarchical clustering on the gene expression data.

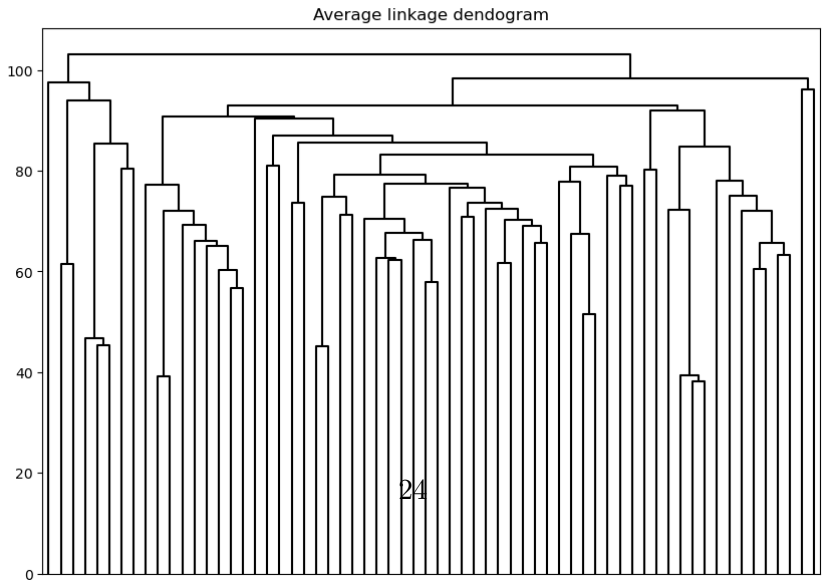
6.0.1 Exercise 10 (CORE)

a) Plot the dendrogram with complete, single, and average linkage. Does the choice of linkage affect the results? Which linkage would you choose?

[41]:
```
# Fit hierarchical clustering with different types of linkage
hc_comp = hierarchy.linkage(X, method = 'complete')
hc_single = hierarchy.linkage(X, method = 'single')
hc_average = hierarchy.linkage(X, method = 'average')

# Plot the dendrograms
cargs = {'color_threshold': -np.inf,'above_threshold_color':'black'}
fig, ax = plt.subplots(3, 1, figsize=(10,24))
hierarchy.dendrogram(hc_comp, ax=ax[0], **cargs, no_labels=True)
ax[0].set_title('Complete linkage dendogram')
hierarchy.dendrogram(hc_single, ax=ax[1], **cargs, no_labels=True)
ax[1].set_title('Single linkage dendogram')
hierarchy.dendrogram(hc_average, ax=ax[2], **cargs, no_labels=True)
ax[2].set_title('Average linkage dendogram')
plt.show()
```


Complete linkage dendogram
140
120
100
80
60
40
20
0

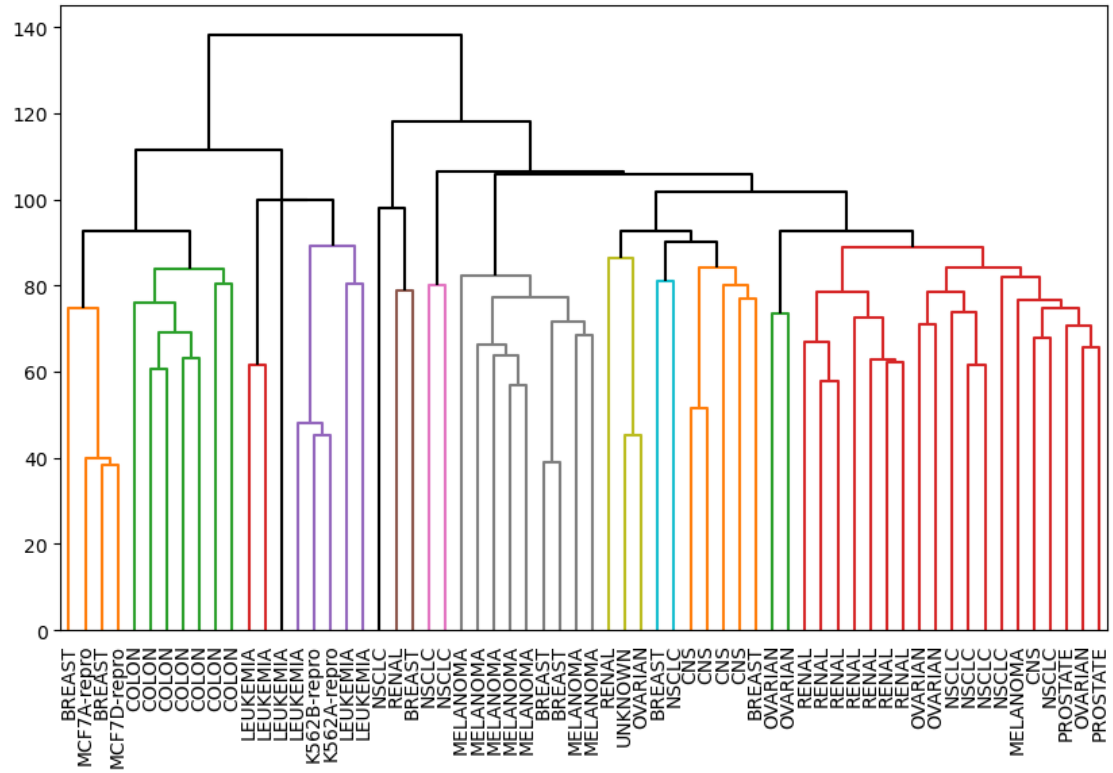

Single linkage dendogram
80
60
40
20
0


Average linkage dendogram
100
80
60
40
20
0

The choice of linkage affects the clusters being formed, as can be seen from the different dendograms above. Complete linkage seems to look like the most balanced.

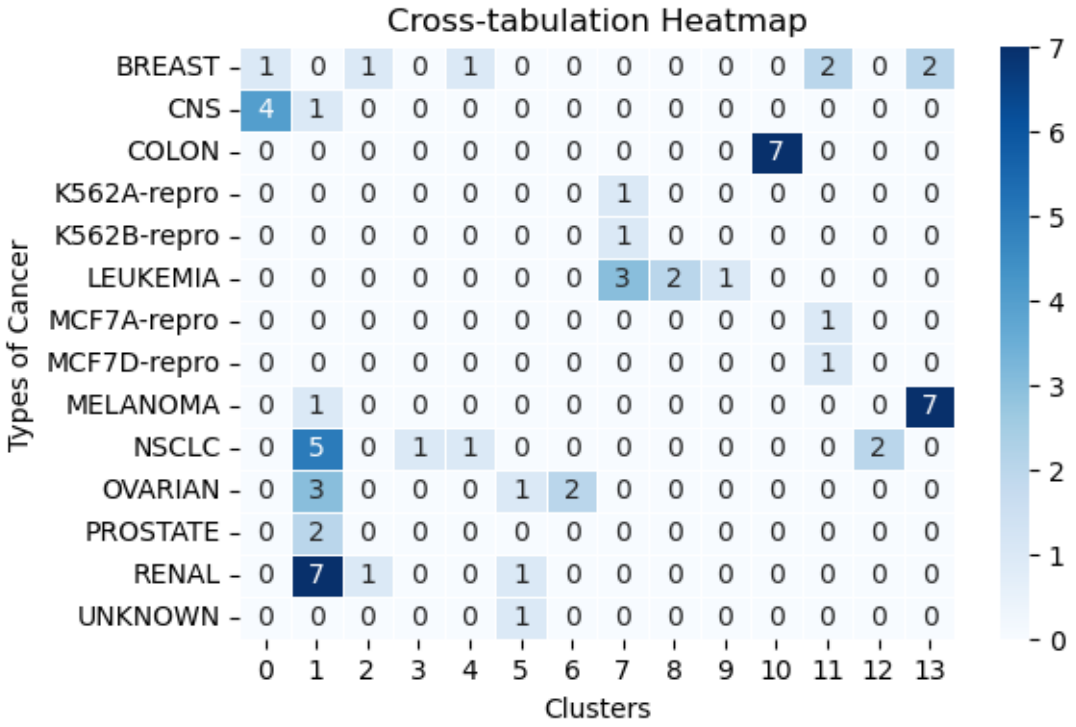
b) Select a linkage and a number of clusters (by examining the dendrogram and jumps in the heights of the clusters merged). Plot the dendogram and color the branches to identify the clusters. Use the option `labels = np.asarray(y_clean), leaf_font_size=10` in `hierarchy.dendrogram` to add the cancer types as labels for each data point. Do you observe any patterns between the clusters and cancer types? You may also want to use `pd.crosstab` to compute a cross-tabulation to compare the clusters and cancer types.

[82]:
```
cargs = {'color_threshold': 90,'above_threshold_color':'black'}
# Choose 14 clusters
fig, ax = plt.subplots(1, 1, figsize=(10,6))
hierarchy.dendrogram(hc_comp, ax=ax, **cargs, labels = np.asarray(y_clean),␣
 ↪leaf_font_size=10)
plt.show()
```



[83]:
```
cuttree = hierarchy.cut_tree(hc_comp, height = 90)

y['clean'] = y_clean
y['cluster'] = cuttree
```

```
cross_tab = pd.crosstab(y['clean'], y['cluster'])

# Plot heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(cross_tab, annot=True, fmt="d", cmap="Blues", linewidths=0.5)

# Labels and title
plt.xlabel('Clusters')
plt.ylabel('Types of Cancer')
plt.title('Cross-tabulation Heatmap')

# Show plot
plt.show()
```



We used complete linkage as it looked the most balanced. We chose 90 as a threshold for height as that will produce exactly 14 clusters, i.e. the number of different types of cancers we have as labels. We do observe a pattern between the clusters and the different types of cancers: indeed, the similarity between Types of cancer and the clusters can also be observed from the Cross-tabulation heatmap above. For instance, the colon cancers line up precisely with the 10th cluster, and for all types of cancer except breast cancer, they are primarily situated within only one of the clusters.

Now, is a good point to switch driver and navigator

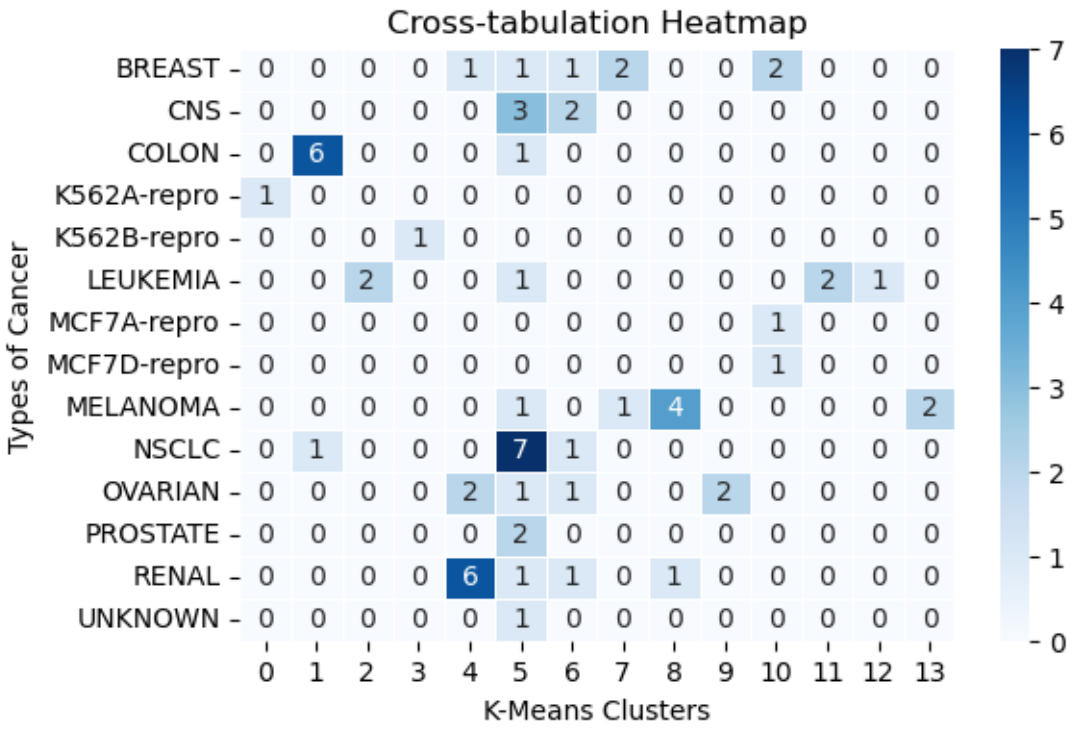
7 K-means Clustering: Gene Expression Data

Now, let's perform k-means clustering on the gene expression data.

7.0.1 Exercise 11 (CORE)

Perform K-means clustering with the same number of clusters that you selected for hierarchical clustering. Are the results similar?

[79]:
```
km = KMeans(n_clusters = 14, init = 'random', n_init = 1, random_state = 0)
km.fit(X)

# Cross-tabulation for KMeans

y['k-means'] = km.labels_

cross_tab = pd.crosstab(y['clean'], y['k-means'])

# Plot heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(cross_tab, annot=True, fmt="d", cmap="Blues", linewidths=0.5)

# Labels and title
plt.xlabel('K-Means Clusters')
plt.ylabel('Types of Cancer')
plt.title('Cross-tabulation Heatmap')

# Show plot
plt.show()
```



Using K-Means produces different clustering (a dissimilar result), although it still performs well. Hierarchical clustering seems to produce a slightly better result in terms of matching with the labels (types of clusters).

7.0.2 Exercise 12 (EXTRA)

Plot the two clustering solutions along with a plot of the data colored by the cancer types in the space spanned by the first two principal components.

[ ]:

8 Competing the Worksheet

At this point you have hopefully been able to complete all the CORE exercises and attempted the EXTRA ones. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Before generating the PDF, please go to Edit -> Edit Notebook Metadata and change 'Student 1' and 'Student 2' in the **name** attribute to include your name.

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF.

[ ]:
```
!jupyter nbconvert --to pdf mlp_week03.ipynb
```

[ ]: