How to Implement Hierarchical Clustering for Direct Marketing Campaigns— with Python Code

Understand the ins and outs of hierarchical clustering, and how it applies to marketing campaign analysis in the banking industry.

Zoumana Keita

.

Published in

Towards Data Science

.

11 min read

.

Aug 28

Frederick Warren on Unsplash

Motivation

Imagine being a Data Scientist at a leading financial institution, and your task is to assist your team in categorizing existing clients into distinct profiles: `low` , `average` , `medium` and `platinum` for loan approval.

But, here is the catch:

> There is no such historical label attached to these customers, so how do you proceed with the creation of these categories?

This is where clustering can help, an unsupervised machine-learning technique to group unlabeled data into similar categories.

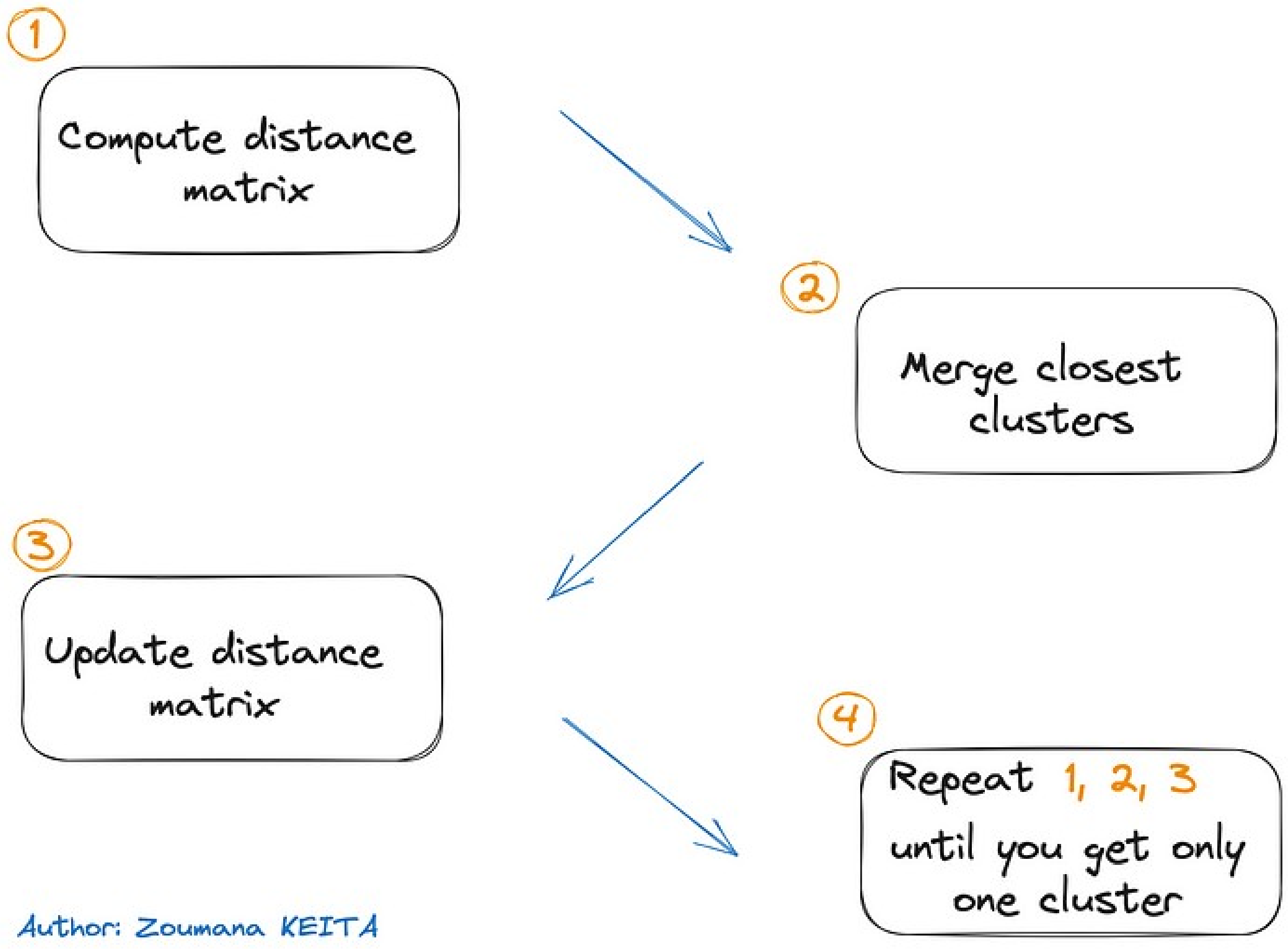
Multiple clustering techniques exist, but this tutorial will focus more on the `hierarchical clustering` approach.

It starts by providing an overview of what `hierarchical clustering` is, before walking you through a step-by-step implementation in `Python` using the popular `Scipy` library.

What is hierarchical clustering?

`Hierarchical clustering` is a technique for grouping data into a tree of clusters called dendrograms, representing the hierarchical relationship between the underlying clusters.

The hierarchical clustering algorithm relies on distance measures to form clusters, and it typically involves the following main steps:



- Computation of the distance matrix containing the distance between each pair of data points using a particular distance metric such as Euclidean distance, Manhattan distance, or cosine similarity
- Merge the two clusters that are the closest in distance
- Update the distance matrix with regard to the new clusters
- Repeat steps 1, 2, and 3 until all the clusters are merged together to create a single cluster

Some graphical illustrations of the hierarchical clustering

Before diving into the technical implementation, let's have an understanding of two main hierarchical clustering approaches: `agglomerative` and `divisive` clustering.

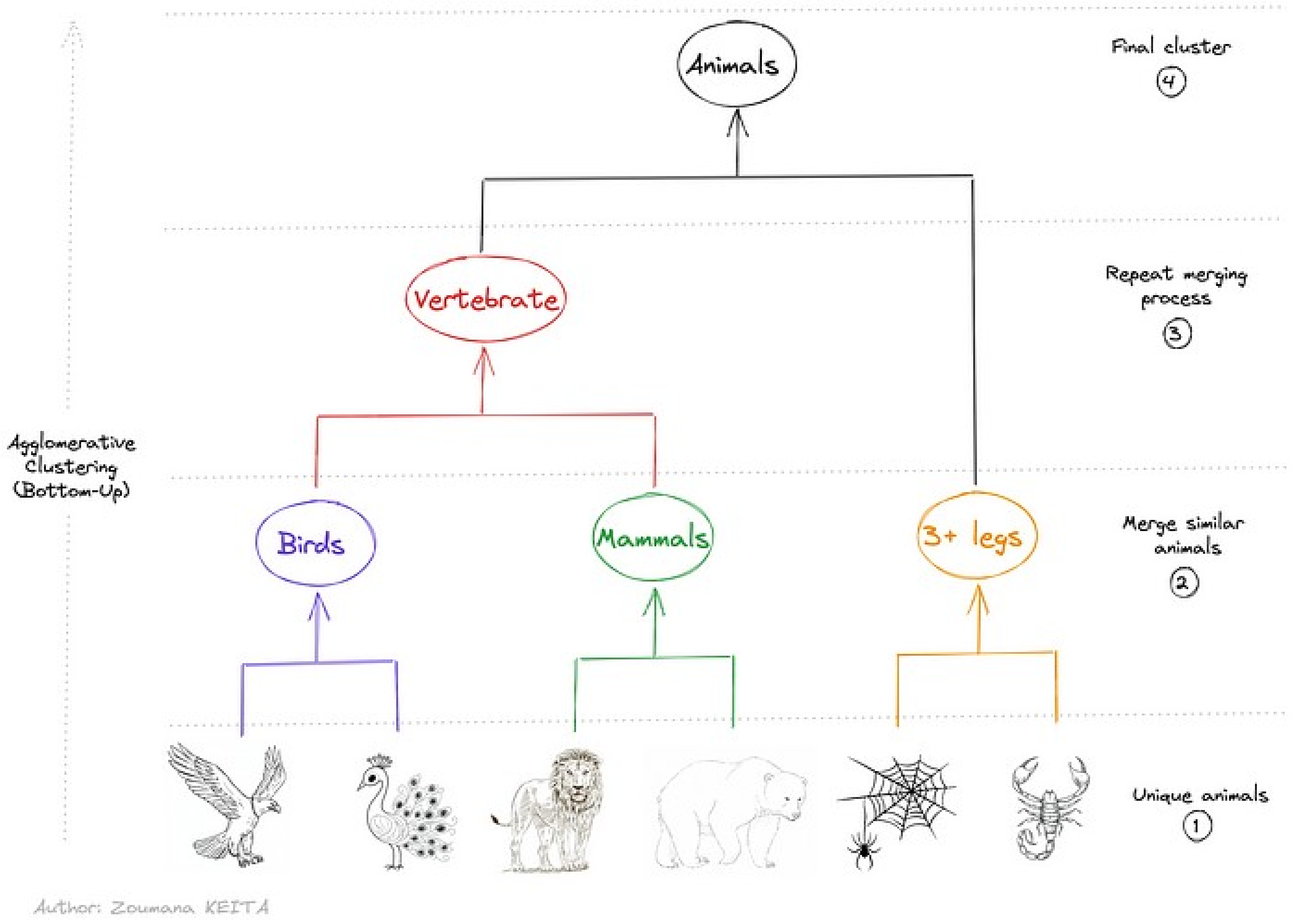
#1. Agglomerative clustering

Also known as a bottom-up approach, agglomerative clustering starts by considering each data point as an individual cluster. It then iteratively merges these clusters until only one remains.

Let's consider the illustration below where:

- We begin by treating each animal as a unique cluster
- Then based on the list of animals, three different clusters are formed according to their similarities: Eagles and Peacock categorized as `Birds` , Lions and bears as `Mammals` , Scorpion and Spiders as `3+ legs`

- We continue the merging process to create the `Vertebrate` cluster by combining the two most similar clusters: `Birds` and `Mammals`
- Lastly, the remaining two clusters, `Vertebrate` and `3+ legs`are merged to create a single `Animals` cluster.

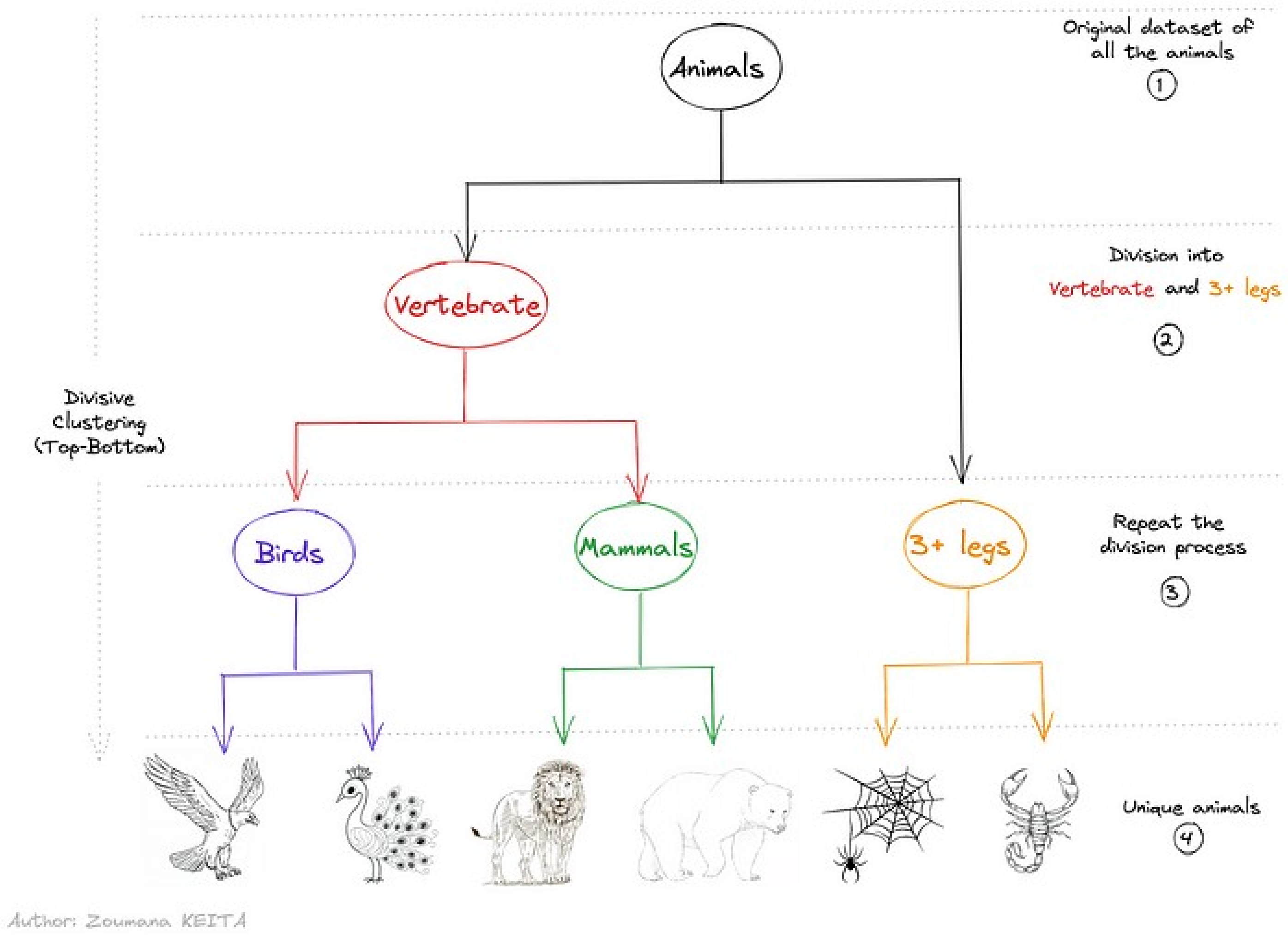


#2. Divisive clustering

Divisive clustering on the other hand is top-down. It begins by considering all the data points as a unified cluster and then progressively splits them until each data point stands as a unique cluster.

By observing the graphic of the divisive approach:

- We notice that the entire `Animal` dataset is considered a unified bloc
- Then, this block is split into two different clusters: `Vertebrate` and `3+ legs`
- The division process is iteratively applied to the previously created clusters until each animal is distinguished as its own unique cluster


Animals
Original dataset of all the animals
1
Vertebrate
Division into Vertebrate and 3+ legs
2
Divisive Clustering (Top-Bottom)
Birds
Mammals
3+ legs
Repeat the division process
3
Unique animals
4
Author: Zoumana KEITA

Choosing the right distance measure

The choice of an appropriate distance measure is a critical step in clustering, and it depends on the specific problem at hand.

For instance, a group of students can be clustered according to their country of origin, gender, or previous academic background. While each of these criteria is valid for clustering, they convey a unique significance.

The Euclidean distance is the most frequently used measure in many clustering software. However other distance measures like Manhattan, Canberra, Pearson correlation, and Minkowski distances also exist.

How to measure clusters before merging them

Clustering might be considered a straightforward process of grouping data. But, it is more than that.

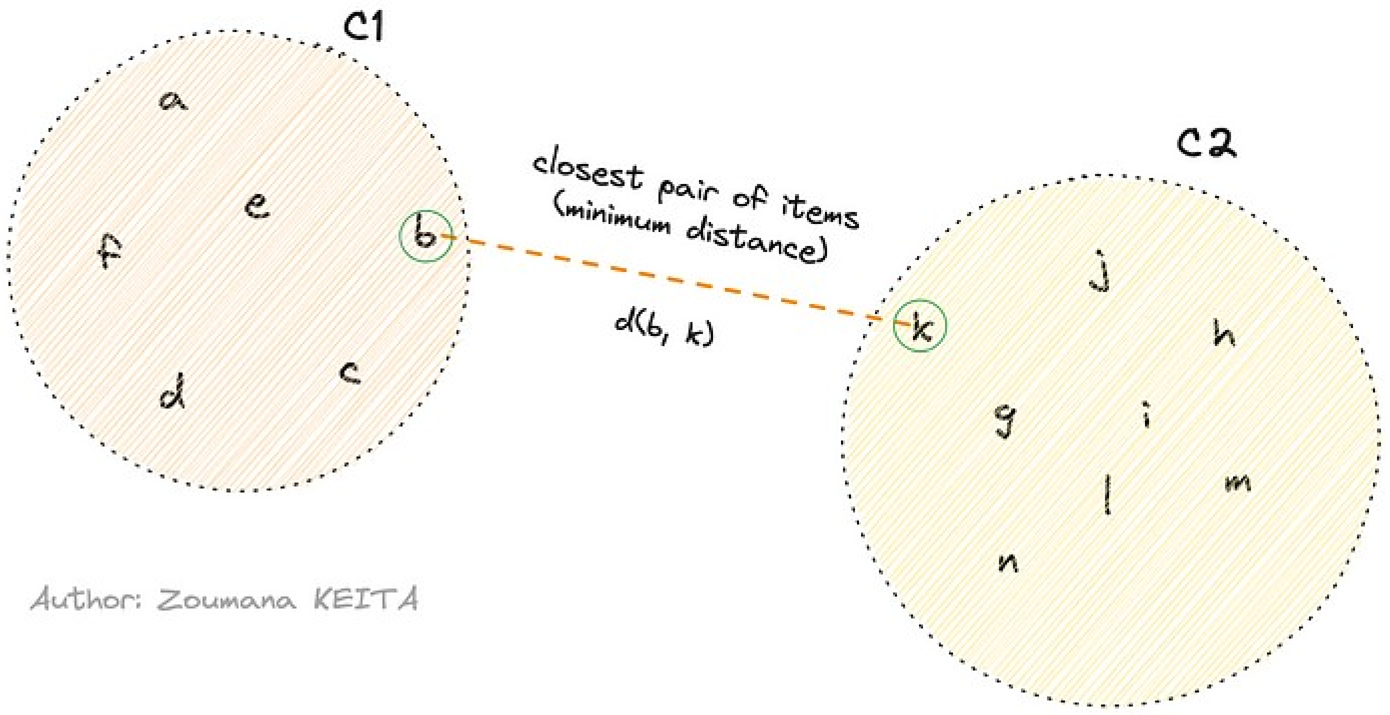
There are three main standard ways to measure the nearest pair of clusters before merging them: `(1) single linkage`, `(2) complete linkage`, and `(3) average linkage`. Let’s explore each one in more detail.

#1. Single linkage

In the single linkage clustering, the distance between two given clusters **C1** and **C2** corresponds to the minimum distances between all pairs of items in the two clusters.

$$\text{Distance} = \text{Min}\,\{d(i, j), \text{ where item } i \text{ is within C1, and item } j \text{ is within C2}\}$$

Out of all the pairs of items from the two clusters, **b** and **k** have the minimum distance.



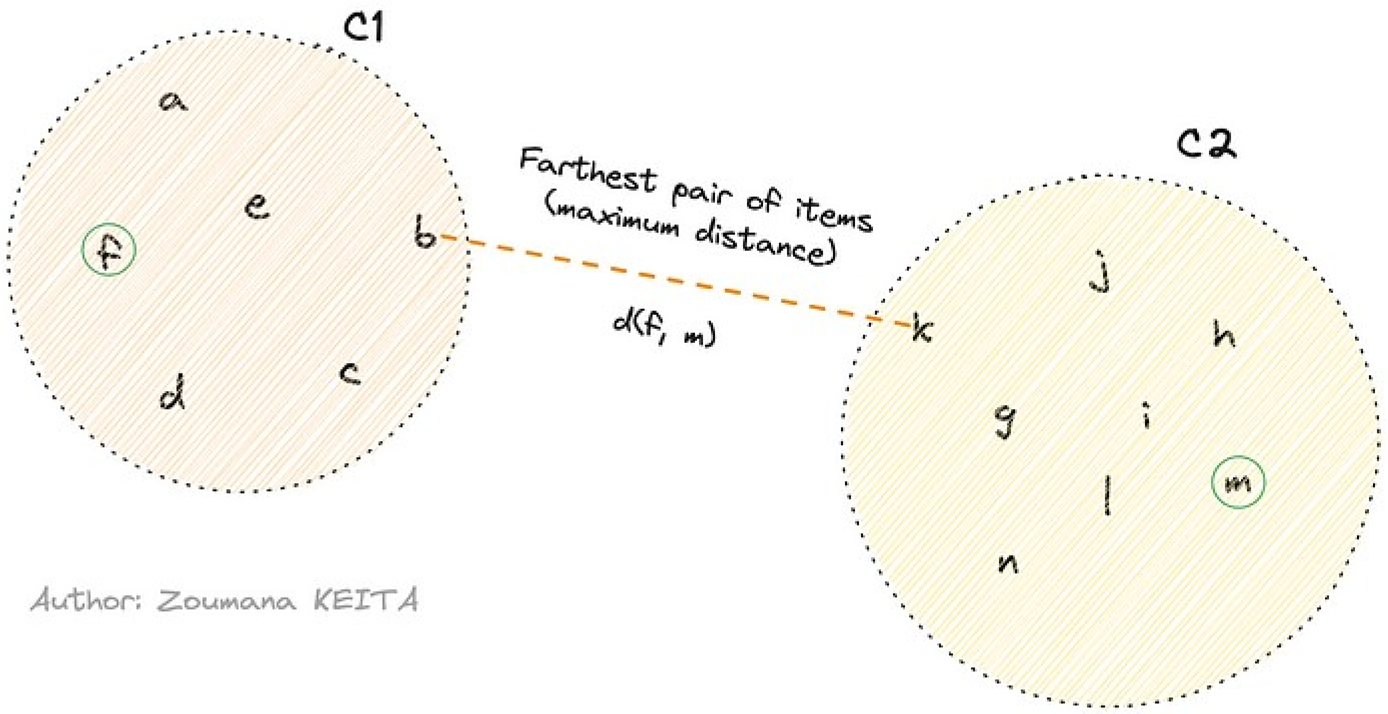
Single linkage illustration (Image by Author)

#2. Complete linkage

For the complete linkage clustering, the distance between two given clusters **C1** and **C2** is the maximum distance between all pairs of items in the two clusters.

$$\text{Distance} = \text{Max}\,\{d(i, j), \text{ where item } i \text{ is within C1, and item } j \text{ is within C2}\}$$

Out of all the pairs of items from the two clusters, the ones highlighted in green (**f** and **m**) have the maximum distance.

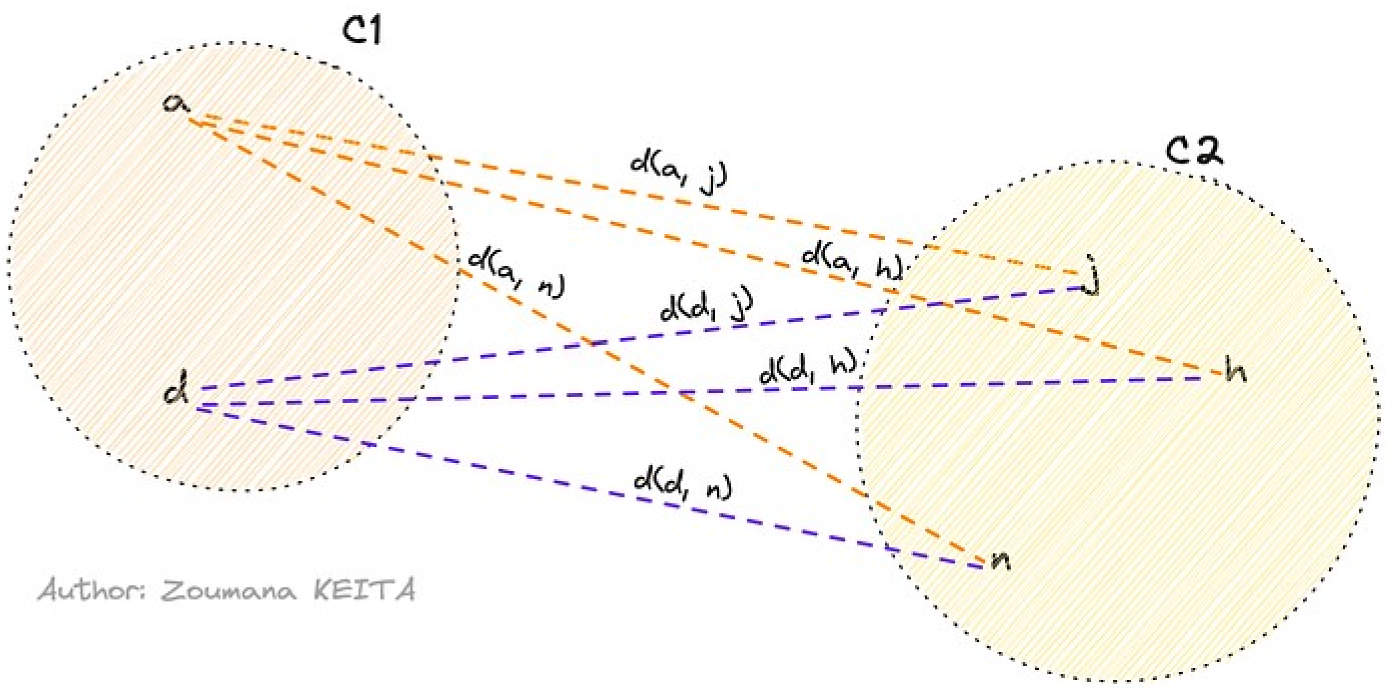


Complete linkage illustration (Image by Author)

#3. Average linkage

In the average linkage clustering, the distance between two given clusters **`C1`** and **`C2`** is computed using the average of all the distances between each pair of items in the two clusters.

$$\text{Distance}(C1, C2) = \frac{\Sigma(d(i, j))}{\text{Total number of distances}}$$



Average linkage illustration (Image by Author)

From the above formula, the average distance can be computed as follows:

$$\text{Distance } (C1, C2) = \frac{d(a, j) + d(d, j) + d(a, h) + d(a, n) + d(d, n) + d(d, h)}{6}$$

Implementing Hierarchical Clustering in Python

Now you have an understanding of how hierarchical clustering works, let's dive deep into the technical implementation using `Python`.

We start by configuring the environment, understanding the data along with the relevant preprocessing tasks, and lastly applying the clustering.

Configure the environment

`Python` is required and needs to be installed along with the following libraries:

- `Pandas` for loading the data frame
- `Scikit-learn` for data normalization
- `Seaborn and Matplotlib` for data visualization
- `Scipy` to apply the clustering

All these libraries are installed using the `pip` command as follows from your notebook:

```
%%bash
pip install scikit-learn
pip install pandas
pip install matplotlib seaborn
pip install scipy
```

Instead of individually installing each library using the `!pip [library]` we use the `%%bash` statement instead so that the notebook cell is considered a shell command, which ignores the `!` hence facilitates the installation.

Understanding the data

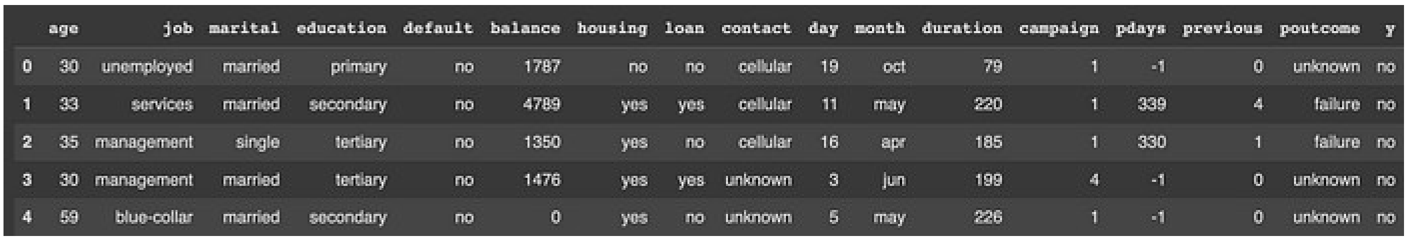
We use a subset of the bank marketing campaigns (phone calls) data of a Portuguese banking institution.

This dataset is from UCI and is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

Due to the unsupervised nature of this tutorial, we get rid of the target column `y` column specifying if the client subscribed to a deposit or not.

Using the `head` function only returns the first five entries, which does not provide enough information about the structure of the data.

```
import pandas as pd
URL = "https://raw.githubusercontent.com/keitazoumana/Medium-Articles-Notebooks/main/data/bank.csv"
bank_data = pd.read_csv(URL, sep=";")
bank_data.head()
```



	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	30	unemployed	married	primary	no	1787	no	no	cellular	19	oct	79	1	-1	0	unknown	no
1	33	services	married	secondary	no	4789	yes	yes	cellular	11	may	220	1	339	4	failure	no
2	35	management	single	tertiary	no	1350	yes	no	cellular	16	apr	185	1	330	1	failure	no
3	30	management	married	tertiary	no	1476	yes	yes	unknown	3	jun	199	4	-1	0	unknown	no
4	59	blue-collar	married	secondary	no	0	yes	no	unknown	5	may	226	1	-1	0	unknown	no

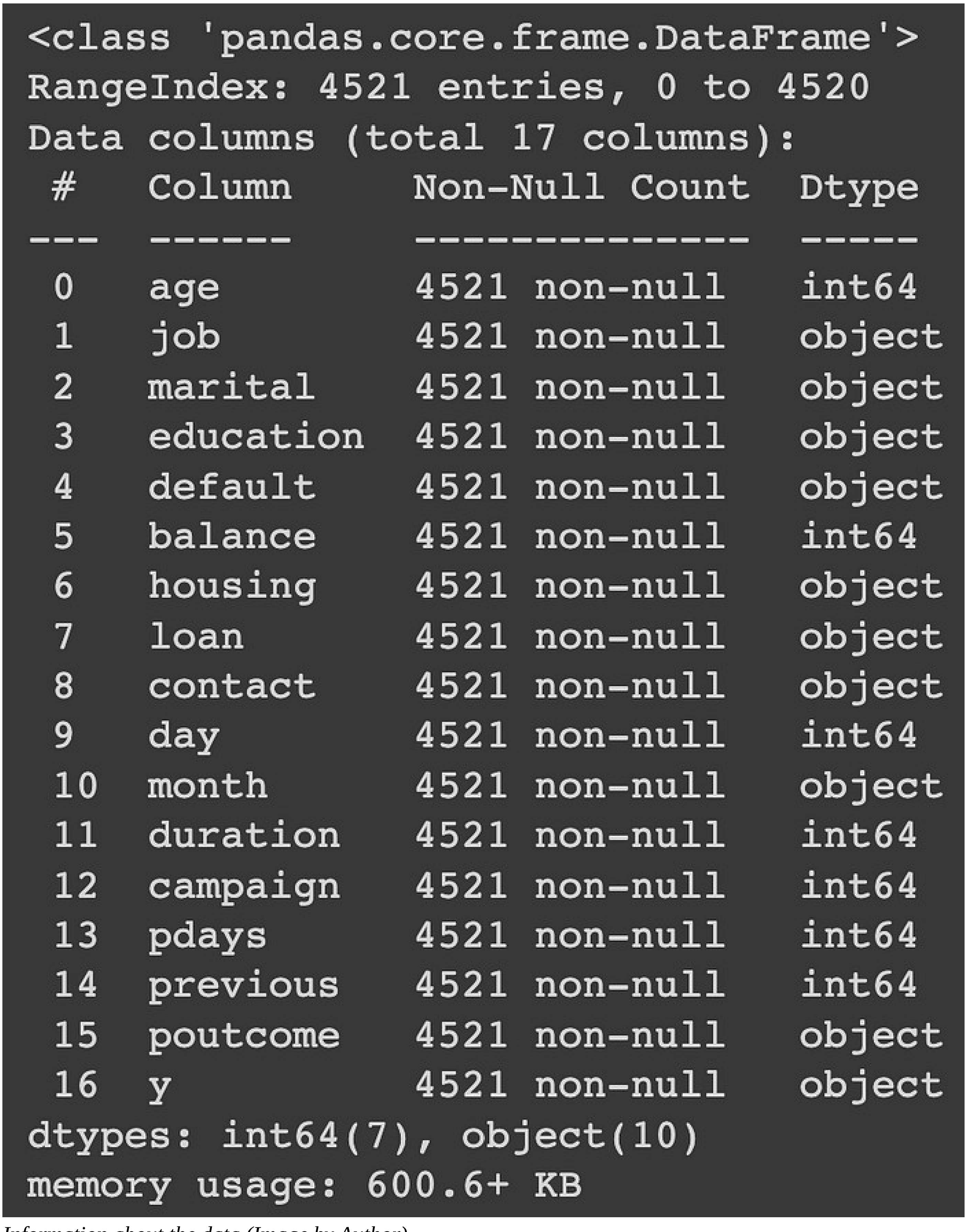
The first five rows of the load data (Image by Author)

However, if we use the `info` function, we can have more granular information about the dataset such as:

- The total number of entries (4,521) and columns (17)
- The name of each column and its type. We can observe that there are two main types of columns: `int64` and `object`
- The total number of missing values in each column

```
bank_data.info()
```

Output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 17 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   age        4521 non-null   int64
 1   job        4521 non-null   object
 2   marital    4521 non-null   object
 3   education  4521 non-null   object
 4   default    4521 non-null   object
 5   balance    4521 non-null   int64
 6   housing    4521 non-null   object
 7   loan       4521 non-null   object
 8   contact    4521 non-null   object
 9   day        4521 non-null   int64
 10  month      4521 non-null   object
 11  duration   4521 non-null   int64
 12  campaign   4521 non-null   int64
 13  pdays      4521 non-null   int64
 14  previous   4521 non-null   int64
 15  poutcome   4521 non-null   object
 16  y          4521 non-null   object
dtypes: int64(7), object(10)
memory usage: 600.6+ KB
```

Information about the data (Image by Author)

Preprocessing the data

Data preprocessing is a major step in every data science task, and clustering is not an exception. The main tasks applied to this data include:

- Filling missing values with appropriate information
- Normalizing the column values
- Finally, dropping irrelevant columns

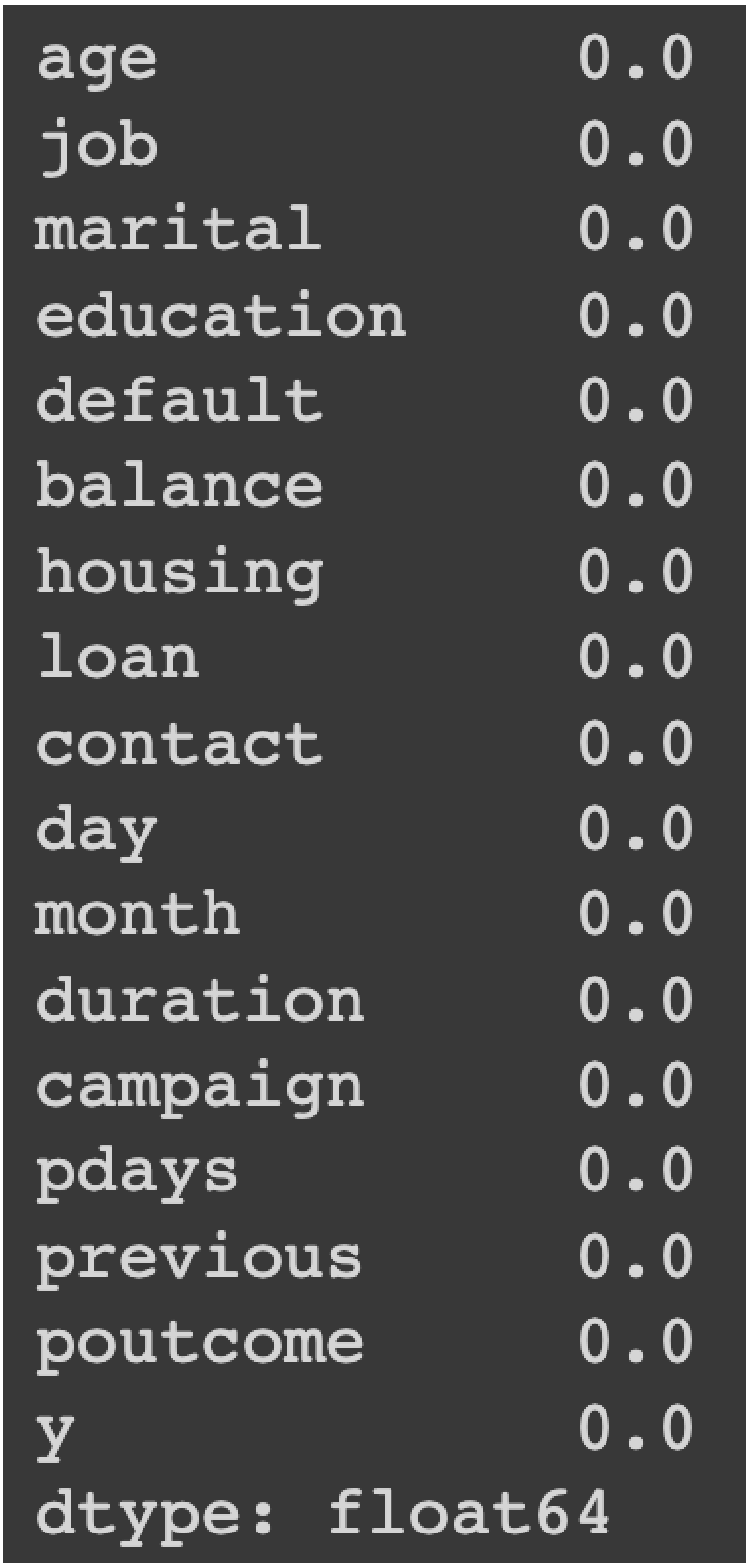
#1. Dealing with missing values

Missing values can significantly damage the overall quality of the analysis and multiple imputation techniques can be applied to efficiently tackle them.

The `percent_missing` reports the percentage of missing value in each column, and luckily, there is no missing value in the data.

```
percent_missing =round(100*(loan_data.isnull().sum())/len(loan_data),2)
percent_missing
```

Output:



```
age          0.0
job          0.0
marital      0.0
education    0.0
default      0.0
balance      0.0
housing      0.0
loan         0.0
contact      0.0
day          0.0
month        0.0
duration     0.0
campaign     0.0
pdays        0.0
previous     0.0
poutcome     0.0
y            0.0
dtype: float64
```

Percentage of missing values in the data (Image by Author)

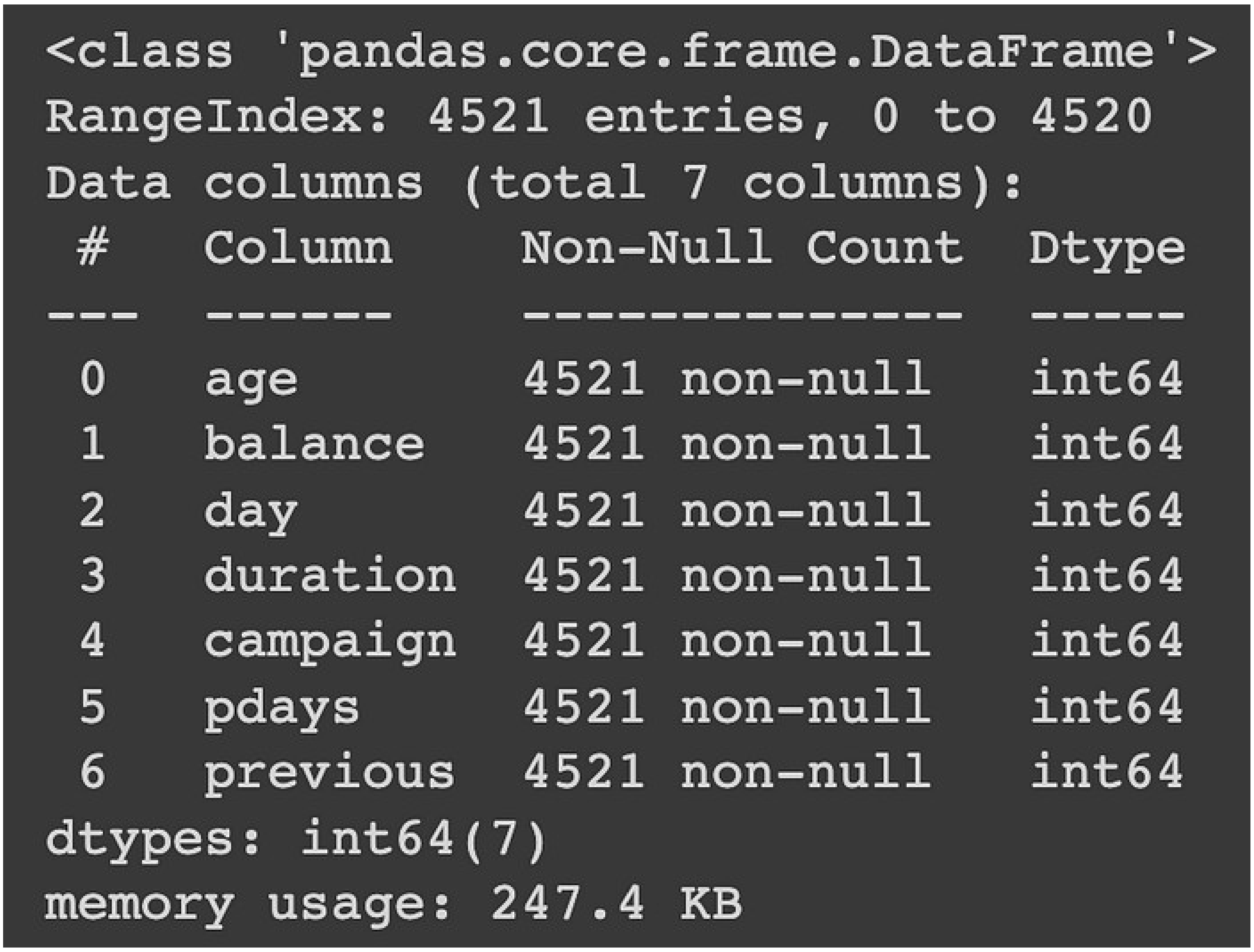
#2. Drop irrelevant columns

Keeping the `object` columns in the dataset would require more processing tasks such as using the relevant encoding technics to encode categorical data into their numerical representation.

Only `int64` (numerical) columns are used in the analysis for simplicity's sake. With the `select_dtypes` function, we select the desired column type to preserve.

```
import numpy as np
cleaned_data = bank_data.select_dtypes(include=[np.int64])
cleaned_data.info()
```

Output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4521 entries, 0 to 4520
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       4521 non-null   int64
 1   balance   4521 non-null   int64
 2   day       4521 non-null   int64
 3   duration  4521 non-null   int64
 4   campaign  4521 non-null   int64
 5   pdays     4521 non-null   int64
 6   previous  4521 non-null   int64
dtypes: int64(7)
memory usage: 247.4 KB
```

New data without the unwanted columns (Image by Author)

#3. Analyze outliers

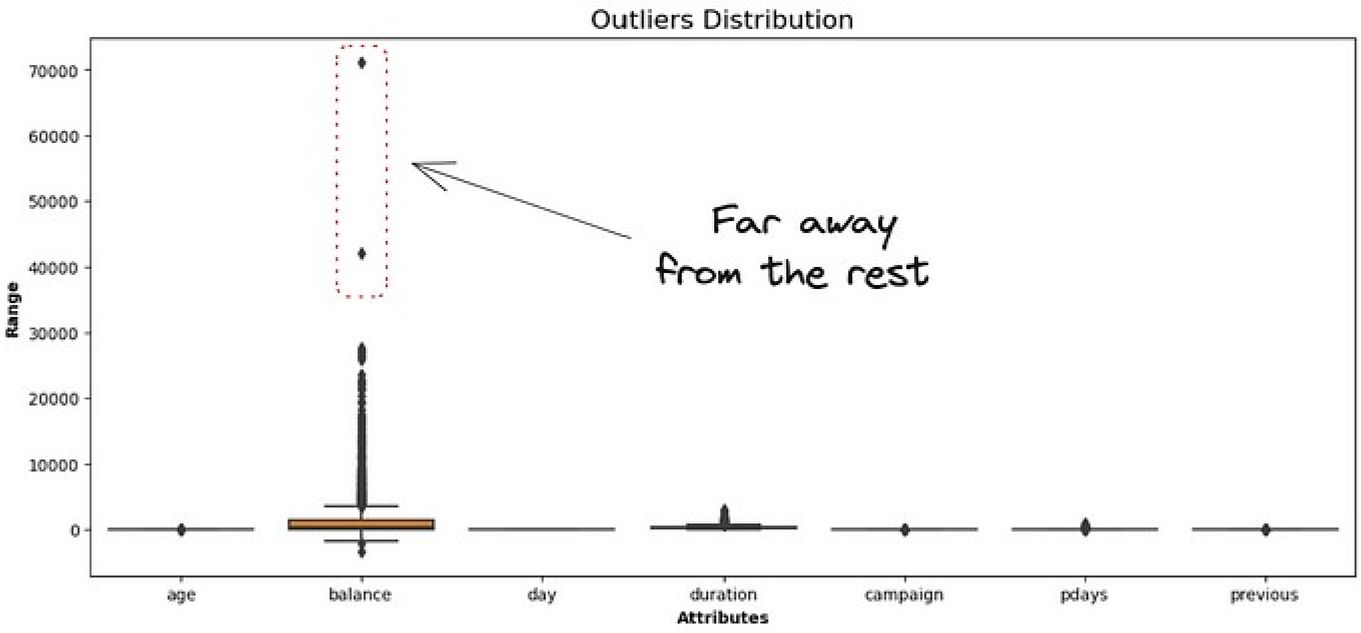
A notable drawback of hierarchical clustering is its sensitivity to outliers, which can skew the distance calculations between data points or clusters.

A simple way to determine those outliers is to analyze the distribution of the data using a `boxplot` as illustrated below in the `show_boxplot` helper function which leverages the `Seaborn` built-in `boxplot` function.

```
import matplotlib.pyplot as plt
import seaborn as sns

def show_boxplot(df):
  plt.rcParams['figure.figsize'] = [14,6]
  sns.boxplot(data = df, orient="v")
  plt.title("Outliers Distribution", fontsize = 16)
  plt.ylabel("Range", fontweight = 'bold')
  plt.xlabel("Attributes", fontweight = 'bold')
```

```
show_boxplot(cleaned_data)
```

Output:



Boxplot of all the variables in the data (Image by Author)

The `balance` attribute representing the clients' average yearly balance is the only one having data points far away from the rest.

By using the interquartile range approach, we can remove all such points that lie outside the range defined by the quartiles `+/-1.5*IQR`, where `IQR` is the `InterQuartile Range`.

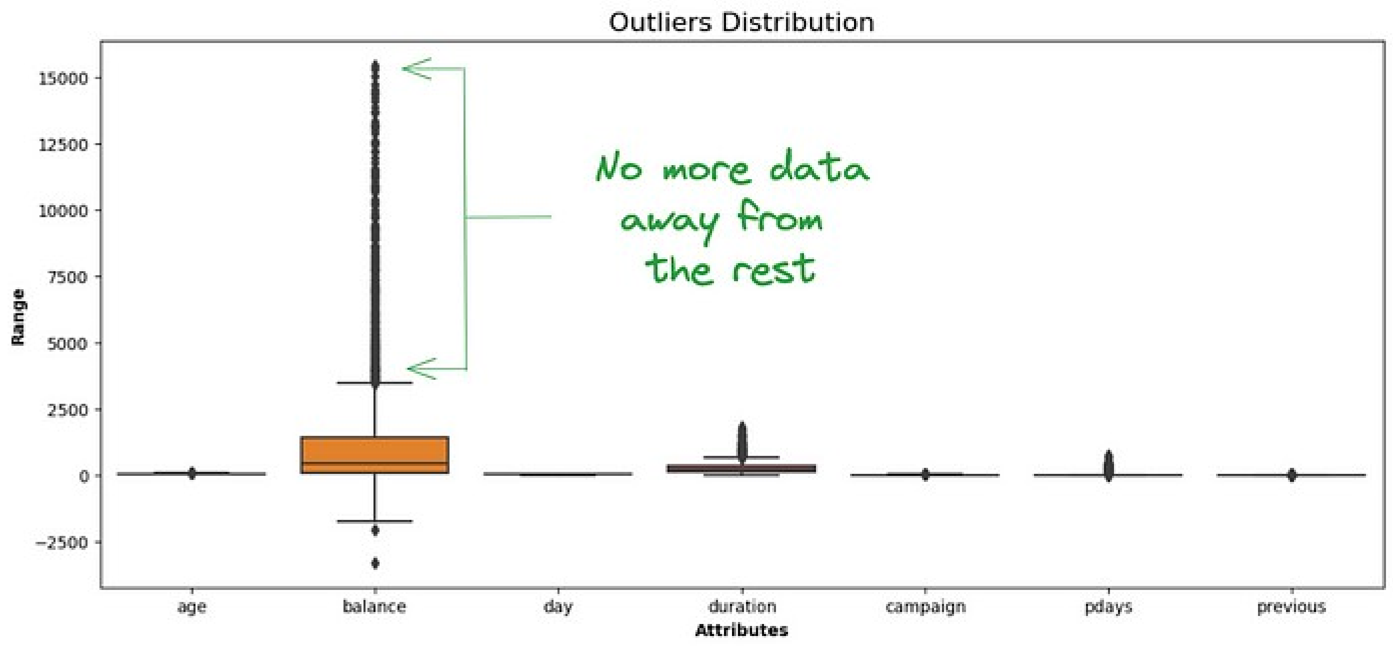
The overall logic is implemented in the `remove_outliers` helper function.

```
def remove_outliers(data):

  df = data.copy()

  for col in list(df.columns):

        Q1 = df[str(col)].quantile(0.05)
        Q3 = df[str(col)].quantile(0.95)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5*IQR
        upper_bound = Q3 + 1.5*IQR

        df = df[(df[str(col)] >= lower_bound) & (df[str(col)] <= upper_bound)]

  return df
```

Then, we can apply the function to the data set, and compare the new boxplot to the one before removing the outliers.

```
without_outliers = remove_outliers(cleaned_data)
# Display the new boxplot
show_boxplot(without_outliers)
```

Output:



```
without_outliers.shape
# (4393, 7)
```

We ended up having a dataset of 4,393 rows and 7 columns, which means that the remaining 127 observations dropped from the data were outliers.

#4. Rescale the data

Given that hierarchical clustering uses Euclidean distance, which is sensitive to variables on different scales, it's better to rescale all the variables prior to distance computing.

The `fit_transform` function from the `StandardScaler` class transforms the original data so that each column has a mean of zero and a standard deviation of one.

```
from sklearn.preprocessing import StandardScaler
data_scaler = StandardScaler()
scaled_data = data_scaler.fit_transform(without_outliers)
scaled_data.shape
# (4393, 7)
```

The shape of the data remains unchanged (4,393 rows and 7 columns) since the normalization does not affect the shape of the data.

Apply the hierarchical clustering algorithm

We are all set to dive deep into the implementation of the clustering algorithm!

At this stage, we can decide which linkage approach to adopt for the clustering of the `method` attribute of `linkage()` function.

Instead of focusing on only one method, let's cover all three linkage techniques using the Euclidean distance.

```
from scipy.cluster.hierarchy import linkage, dendrogram
complete_clustering = linkage(scaled_data, method="complete",
metric="euclidean")
average_clustering = linkage(scaled_data, method="average", metric="euclidean")
```

```
single_clustering = linkage(scaled_data, method="single", metric="euclidean")
```

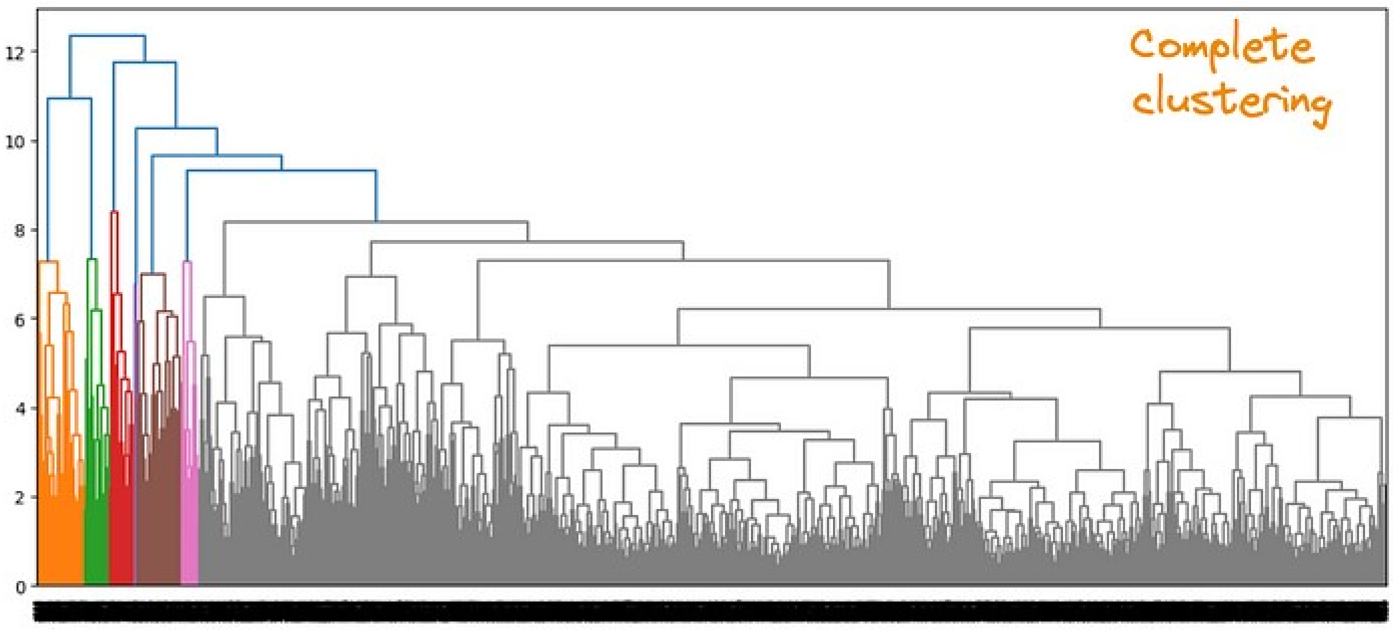
After computing all three clusterings, the respective dendrograms are visualized using the `dendogram` function from `scipy.cluster` module and the `pyplot` function from `matplotlib`.

Each dendrogram is organized as follows:

- The `x-axis` represents the clusters in the data
- The `y-axis` corresponds to the distance between those samples. The higher the line, the more dissimilar are those clusters
- The appropriate number of clusters is obtained by drawing a horizontal line through that highest vertical line
- The number of intersections with the horizontal line corresponds to the number of clusters

```
dendrogram(complete_clustering)
plt.show()
```

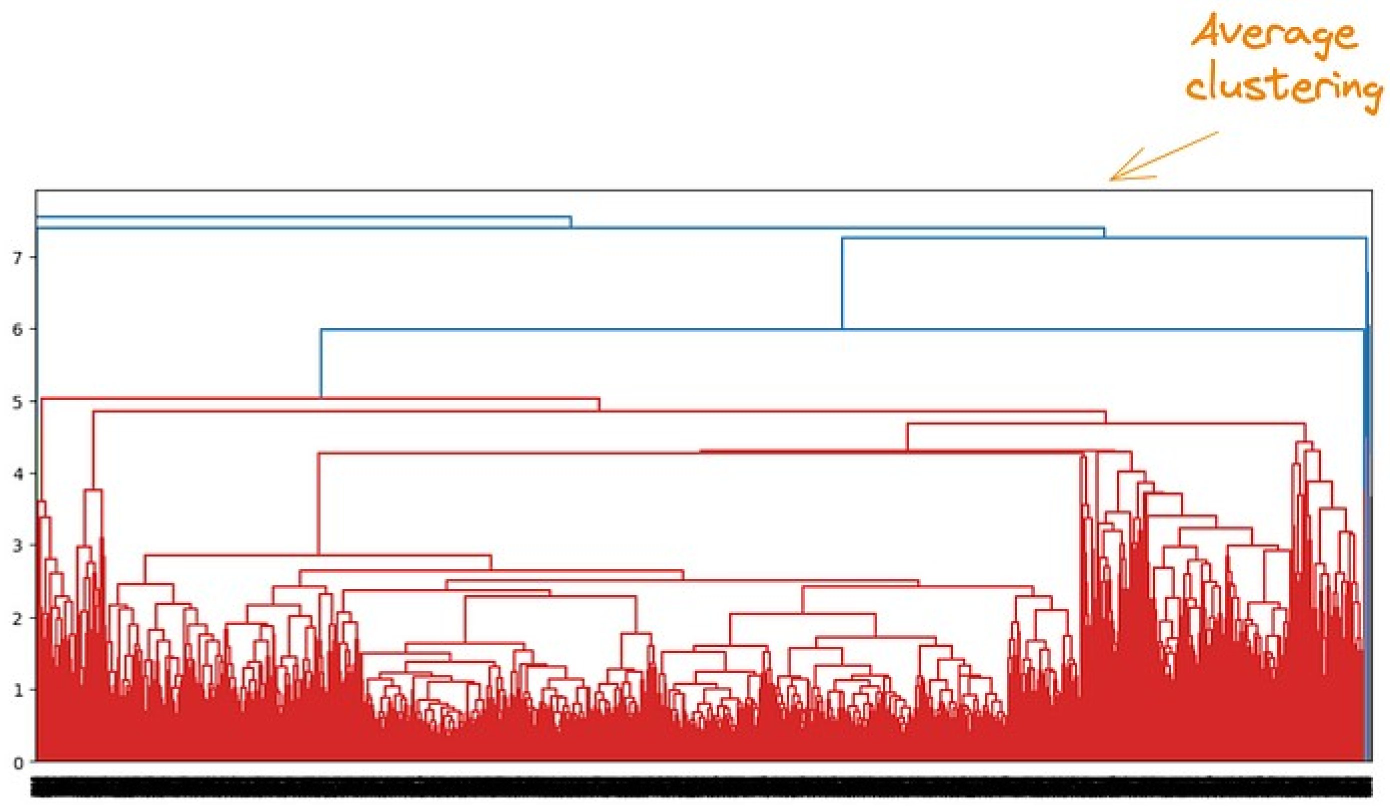
Output:



Dendrogram of the complete clustering approach (Image by Author)

```
dendrogram(average_clustering)
plt.show()
```

Output:



Dendrogram of the average clustering approach (Image by Author)

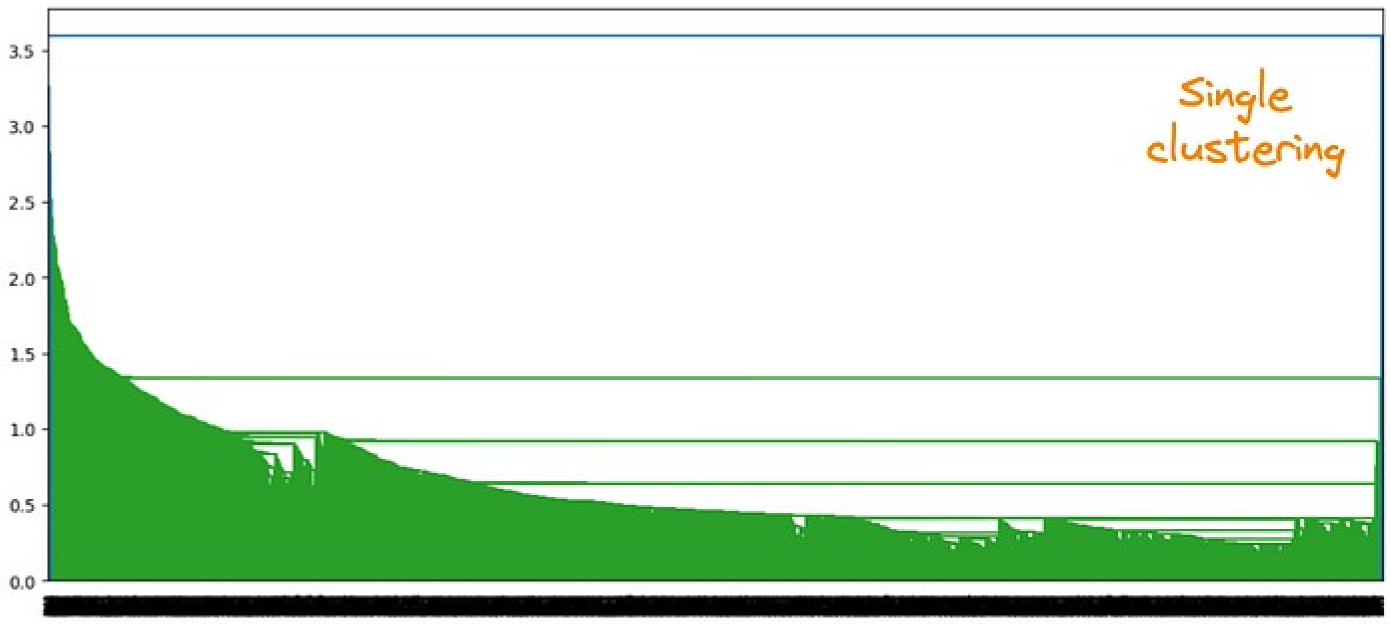
When running the `single clustering` we might face the `recursion limit` issue. This is tackled by using the `setrecursionlimit` function with a large enough value:

```
import sys
sys.setrecursionlimit(1000000)
```

Now we display the dendrogram:

```
dendrogram(single_clustering)
plt.show()
```

Output:

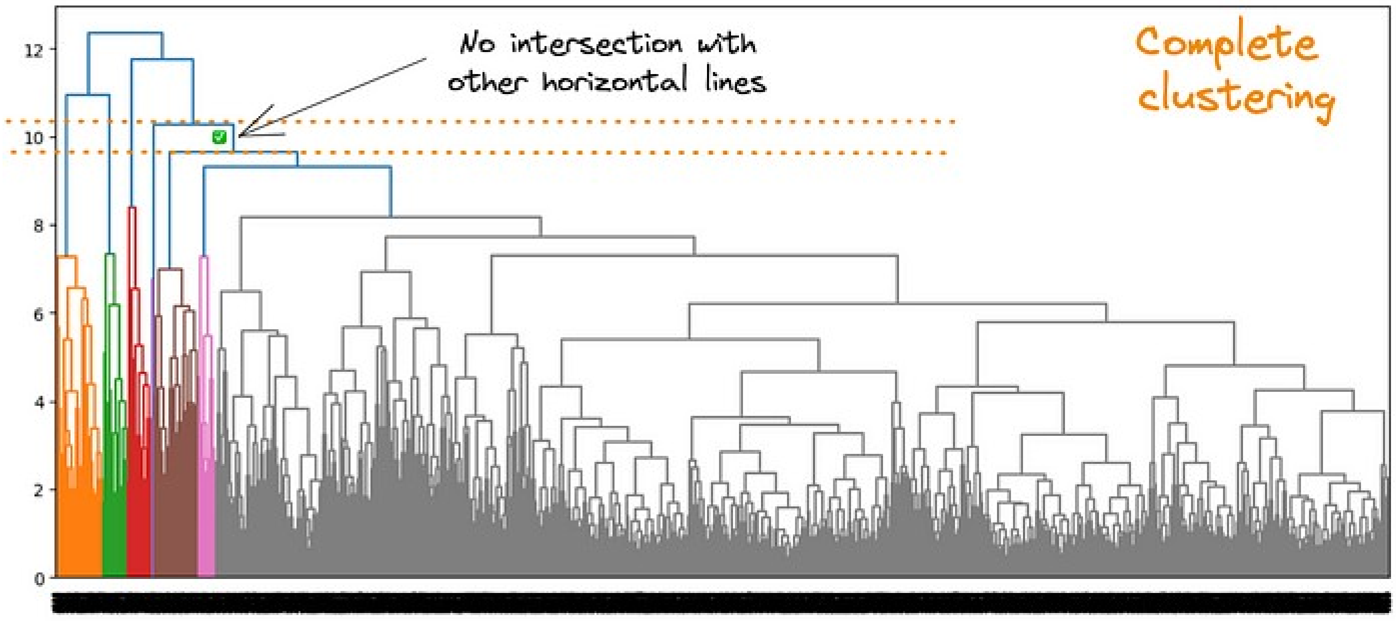


Dendrogram of the single clustering approach (Image by Author)

Determine the number of optimal clusters in the dendrograms

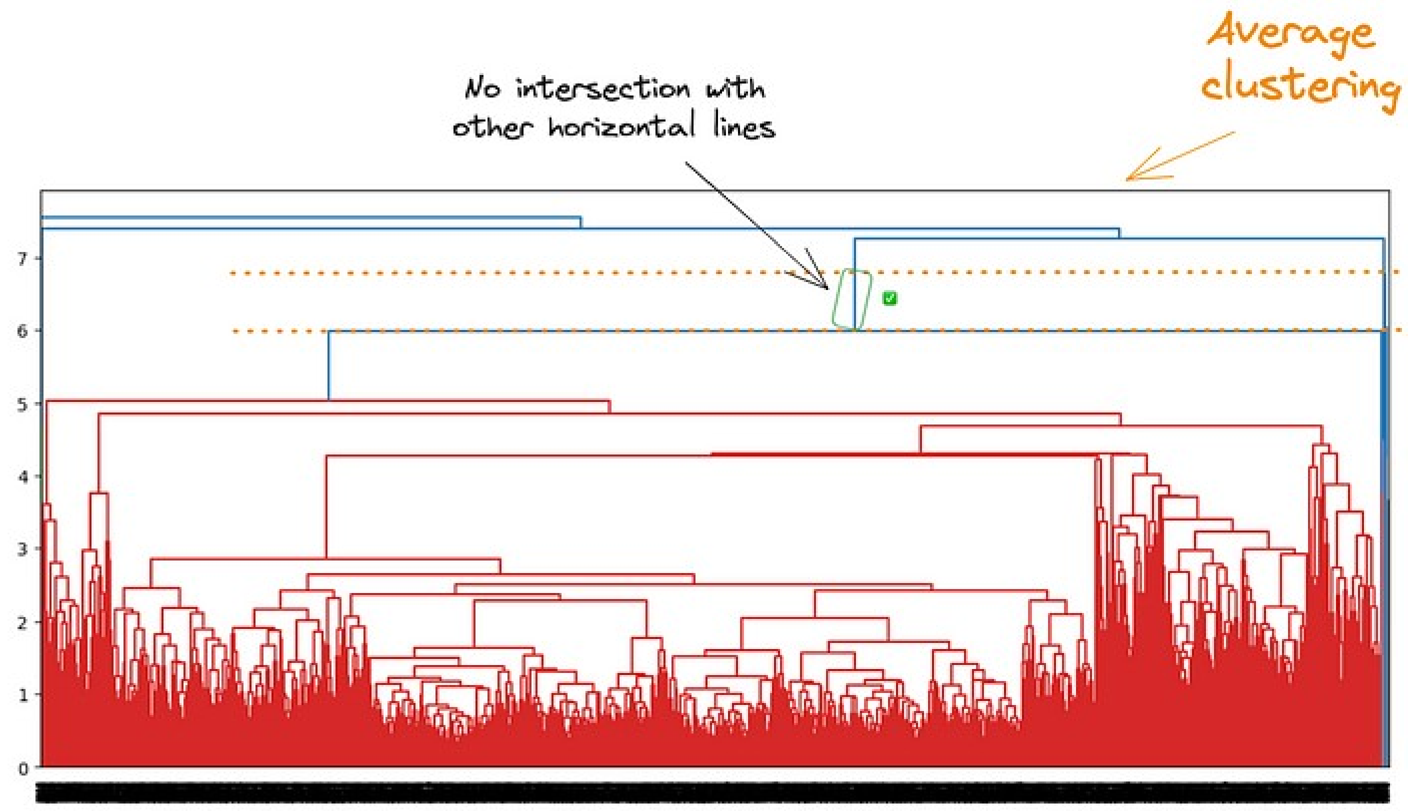
The optimal number of clusters can be obtained by identifying the highest vertical line that does not intersect with any other clusters (horizontal line). Such a line is found below with a red circle and green check mark.

- For complete linkage: there is no significant number of clusters generated



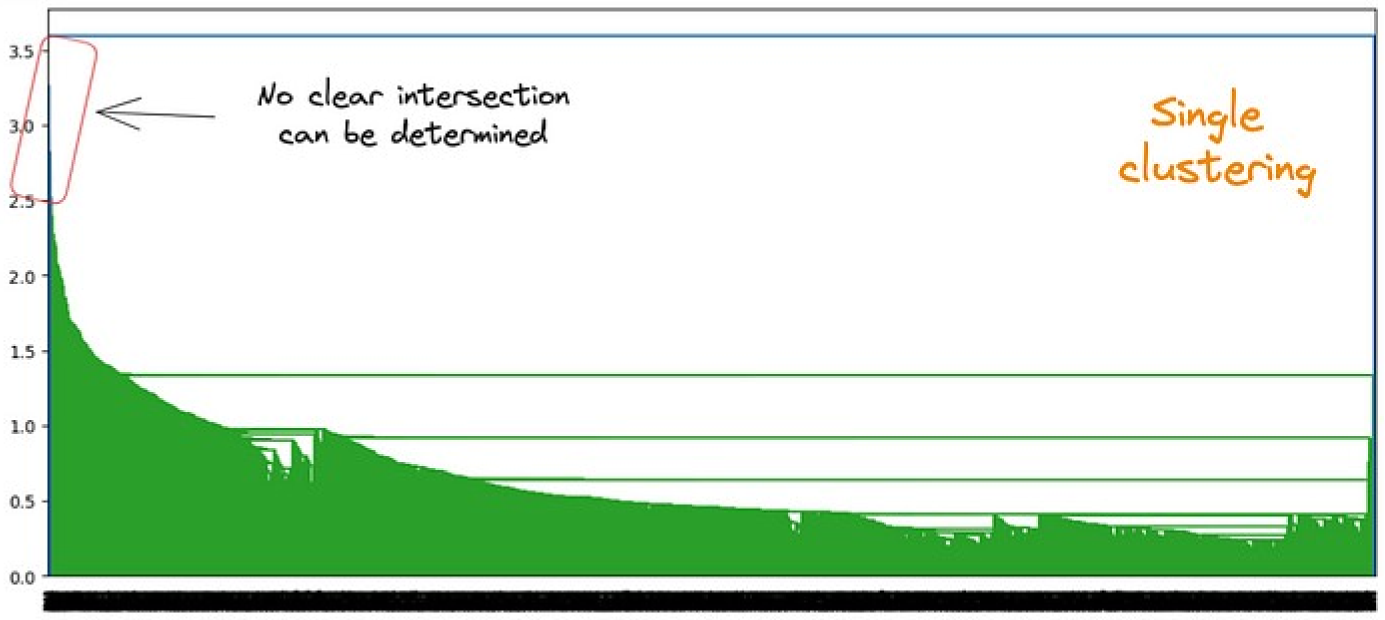
Complete linkage: The optimal number of clusters from the highest distance without intersection (Image by Author)

- For the average linkage: the difference between the two horizontal orange lines is slightly more than one. We can consider two clusters instead.



Average linkage: The optimal number of clusters from the highest distance without intersection (Image by Author)

- For the single linkage: no clear number of cluster can be determined

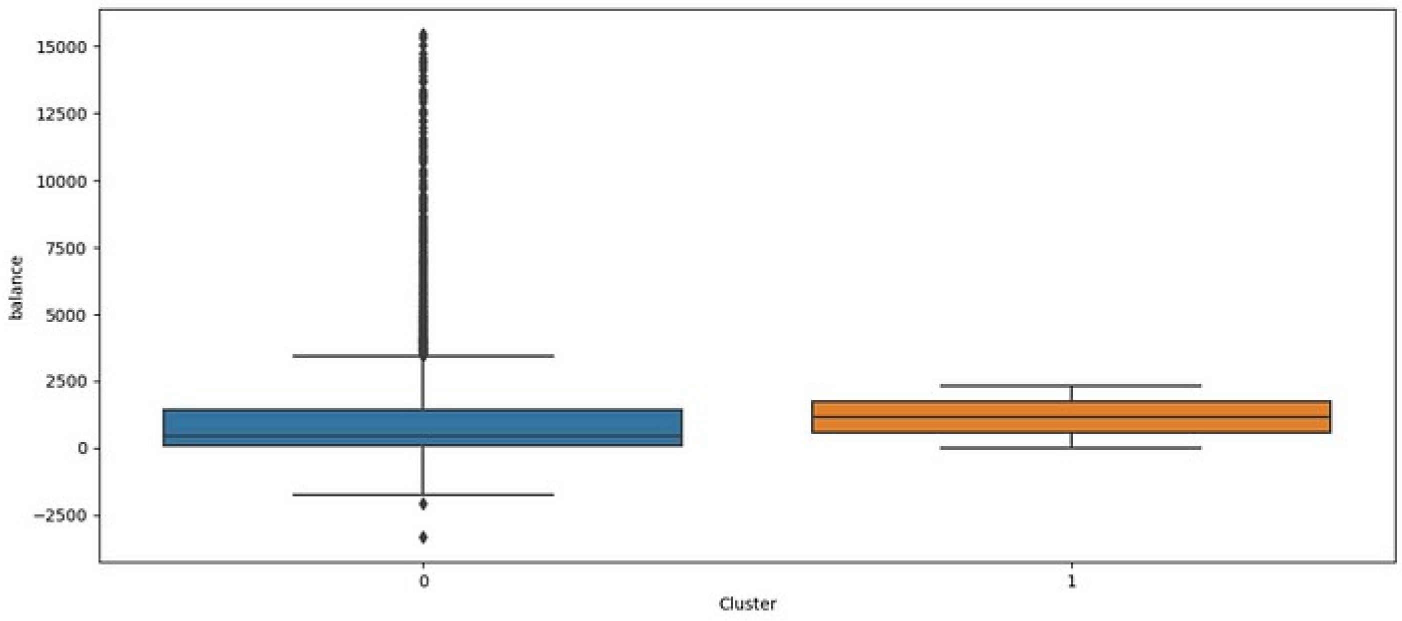


Single linkage: The optimal number of clusters from the highest distance without intersection (Image by Author)

Based on the analysis above, the average linkage seems to provide the optimal number of clusters compared to the single and complete linkages which do not provide a clear understanding of the number of clusters.

Now that we have found the optimal number of clusters let’s interpret these clusters in the context of the clients’ average yearly balance using the `cut_tree` function.

```
cluster_labels = cut_tree(average_clustering, n_clusters=2).reshape(-1, )
without_outliers["Cluster"] = cluster_labels
sns.boxplot(x='Cluster', y='balance', data=without_outliers)
```



(Image by Author)

From the above `boxplot`, we can observe that:

- Clients from cluster 0 possess the highest average annual balance
- Borrowers from cluster 1 have a comparatively lower average annual balance

Conclusion

Congratulations!!!

I hope this article provided enough tools to help you take your knowledge to the next level. The code is available on my GitHub.

Also, If you enjoy reading my stories and wish to support my writing, consider becoming a Medium member. It’s $5 a month, giving you unlimited access to thousands of Python guides and Data science articles.

By signing up using my link, I will earn a small commission at no extra cost to you.

Join Medium with my referral link - Zoumana Keita

As a Medium member, a portion of your membership fee goes to writers you read, and you get full access to every story…

zoumanakeita.medium.com

Feel free to follow me on YouTube, or say Hi on LinkedIn. I am also open to a 1–1 discussion if you need further information.