StackAbuse

Home > Articles

Definitive Guide to Hierarchical Clustering with Python and Scikit-Learn

Cássia Sampaio

Introduction

> *In this guide, we will focus on implementing the **Hierarchical Clustering Algorithm with Scikit-Learn** to solve a marketing problem.*

After reading the guide, you will understand:

- When to apply Hierarchical Clustering
- How to visualize the dataset to understand if it is fit for clustering
- How to pre-process features and engineer new features based on the dataset
- How to reduce the dimensionality of the dataset using PCA
- How to use and read a dendrogram to separate groups
- What are the different linking methods and distance metrics applied to dendrograms and clustering algorithms
- What are the agglomerative and divisive clustering strategies and how they work
- How to implement the Agglomerative Hierarchical Clustering with Scikit-Learn
- What are the most frequent problems when dealing with clustering algorithms and how to solve them

> **Note:** You can download the notebook containing all of the code in this guide here.

Motivation

Imagine a scenario in which you are part of a data science team that interfaces with the marketing department. Marketing has been gathering customer shopping data for a while, and they want to understand, based on the collected data, if there are *similarities between customers*. Those similarities divide customers into groups and having customer groups helps in the targeting of campaigns, promotions, conversions, and building better customer relationships.

> *Is there a way you could help determine which customers are similar? How many of them belong to the same group? And how many different groups there are?*

One way of answering those questions is by using a **clustering** algorithm, such as K-Means, DBSCAN, Hierarchical Clustering, etc. In general terms, clustering algorithms find similarities between data points and group them.

In this case, our marketing data is fairly small. We have information on only 200 customers. Considering the marketing team, it is important that we can clearly explain to them how the decisions were made based on the number of clusters, therefore explaining to them how the algorithm actually works.

Since our data is small and explicability is a major factor, we can leverage **Hierarchical Clustering** to solve this problem. This process is also known as **Hierarchical Clustering Analysis (HCA)**.

> *One of the advantages of HCA is that it is interpretable and works well on small datasets.*

Another thing to take into consideration in this scenario is that HCA is an **unsupervised** algorithm. When grouping data, we won't have a way to verify that we are correctly identifying that a user belongs to a specific group (we don't know the groups). There are no labels for us to compare our results to. If we identify the groups correctly, it will be later confirmed by the marketing department on a day-to-day basis (as measured by metrics such as ROI, conversion rates, etc.).

Now that we have understood the problem we are trying to solve and how to solve it, we can start to take a look at our data!

Brief Exploratory Data Analysis

> **Note:** You can download the dataset used in this guide here.

After downloading the dataset, notice that it is a *CSV (comma-separated values)* file called `shopping-data.csv`. To make it easier to explore and manipulate the data, we'll load it into a `DataFrame` using Pandas:

```
import pandas as pd

# Substitute the path_to_file content by the path to your shopping-data.csv file
path_to_file = 'home/projects/datasets/shopping-data.csv'
customer_data = pd.read_csv(path_to_file)
```

> **Advice:** If you're new to Pandas and DataFrames, you should read our *"Guide to Python with Pandas: DataFrame Tutorial with Examples"*!

Marketing said it had collected 200 customer records. We can check if the downloaded data is complete with 200 rows using the `shape` attribute. It will tell us how many rows and columns we have, respectively:

```
customer_data.shape
```

This results in:

```
(200, 5)
```

Great! Our data is complete with 200 rows *(client records)* and we have also 5 columns *(features)*. To see what characteristics the marketing department has collected from customers, we can see column names with the `columns` attribute. To do that, execute:

```
customer_data.columns
```

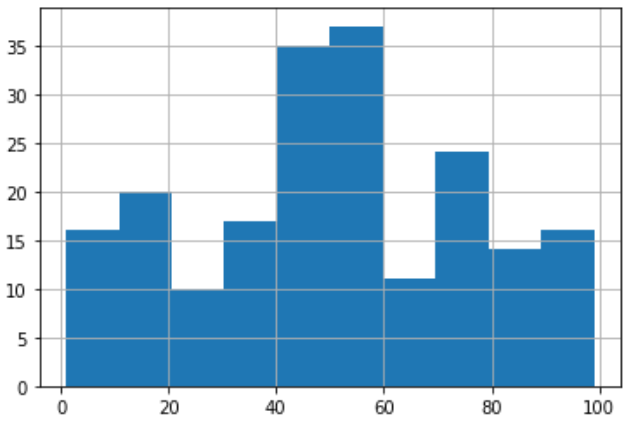
The script above returns:


```
Index(['CustomerID', 'Genre', 'Age', 'Annual Income (k$)',
       'Spending Score (1-100)'],
      dtype='object')
```

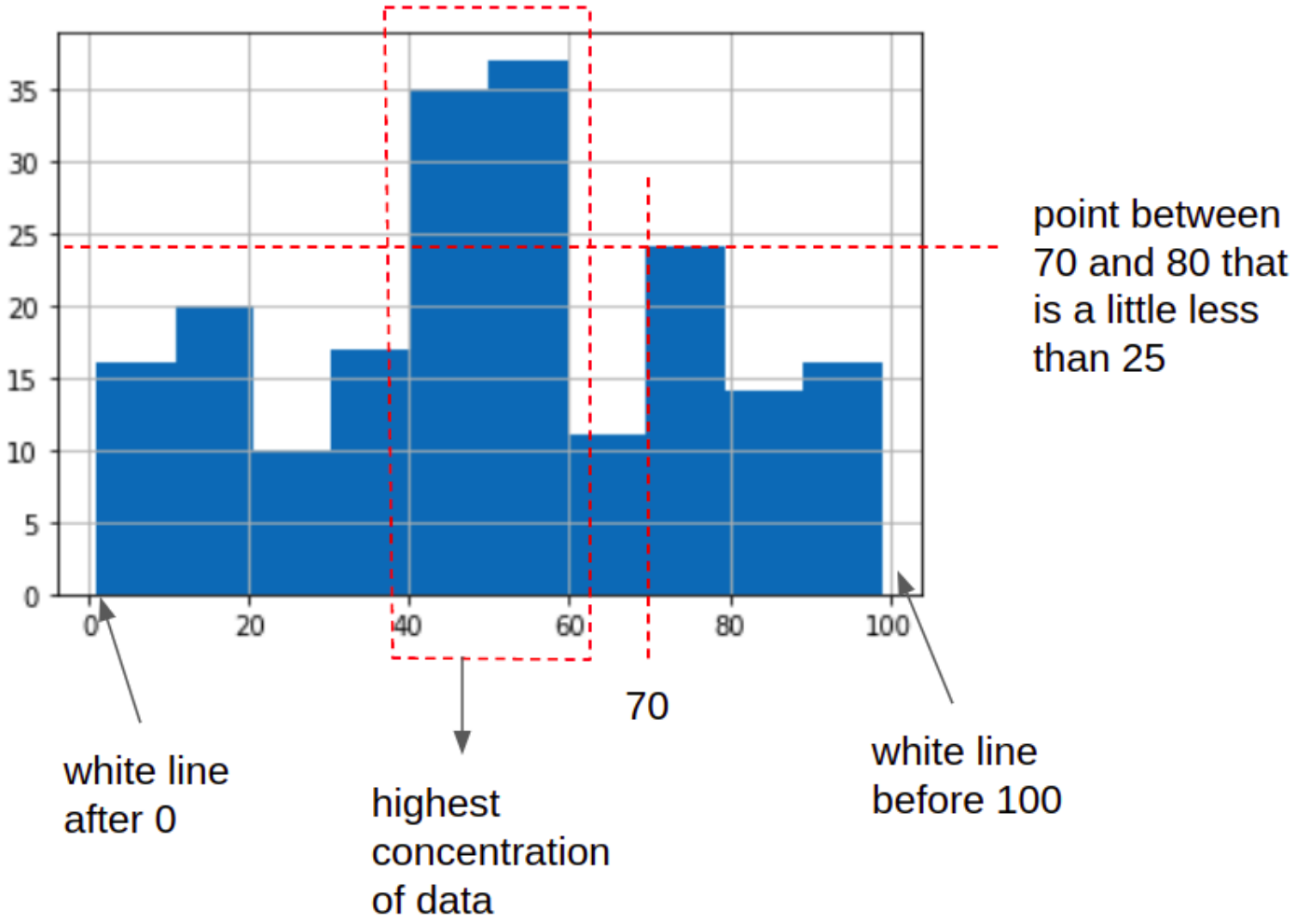
Here, we see that marketing has generated a `CustomerID`, gathered the `Genre`, `Age`, `Annual Income` (in thousands of dollars), and a `Spending Score` going from 1 to 100 for each of the 200 customers. When asked for clarification, they said that the values in the `Spending Score` column signify how often a person spends money in a mall on a scale of 1 to 100. In other words, if a customer has a score of 0, this person never spends money, and if the score is 100, we have just spotted the highest spender.

Let's take a quick look at the distribution of this score to inspect the spending habits of users in our dataset. That's where the Pandas `hist()` method comes in to help:

```
customer_data['Spending Score (1-100)'].hist()
```



By looking at the histogram we see that more than 35 customers have scores between `40` and `60`, then less than 25 have scores between `70` and `80`. So most of our customers are *balanced spenders*, followed by moderate to high spenders. We can also see that there is a line after `0`, to the left of the distribution, and another line before 100, to the right of the distribution. These blank spaces probably mean that the distribution doesn't contain non-spenders, which would have a score of `0`, and that there are also no high spenders with a score of `100`.



To verify if that is true, we can look at the minimum and maximum values of the distribution. Those values can be easily found as part of the descriptive statistics, so we can use the `describe()` method to get an understanding of other numeric values distributions:

```
# transpose() transposes the table, making it easier for us to compare values
customer_data.describe().transpose()
```

This will give us a table from where we can read distributions of other values of our dataset:

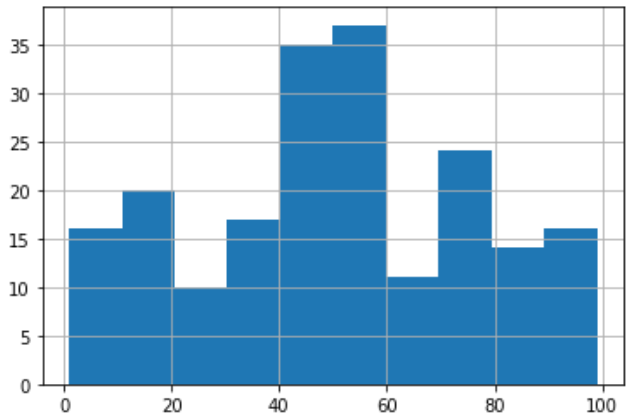
	count	mean	std	min	25%	50%	75%	max
CustomerID	200.0	100.50	57.879185	1.0	50.75	100.5	150.25	200.
Age	200.0	38.85	13.969007	18.0	28.75	36.0	49.00	70.0
Annual Income (k$)	200.0	60.56	26.264721	15.0	41.50	61.5	78.00	137.
Spending Score (1-100)	200.0	50.20	25.823522	1.0	34.75	50.0	73.00	99.0

Our hypothesis is confirmed. The `min` value of the `Spending Score` is `1` and the max is `99`. So we don't have `0` or `100` score spenders. Let's then take a look at the other columns of the transposed `describe` table. When looking at the `mean` and `std` columns, we can see that for `Age` the `mean` is `38.85` and the `std` is approximately `13.97`. The same happens for `Annual Income`, with a `mean` of `60.56` and `std` `26.26`, and for `Spending Score` with a `mean` of `50` and `std` of `25.82`. For all features, the `mean` is far from the standard deviation, which indicates *our data has high variability*.

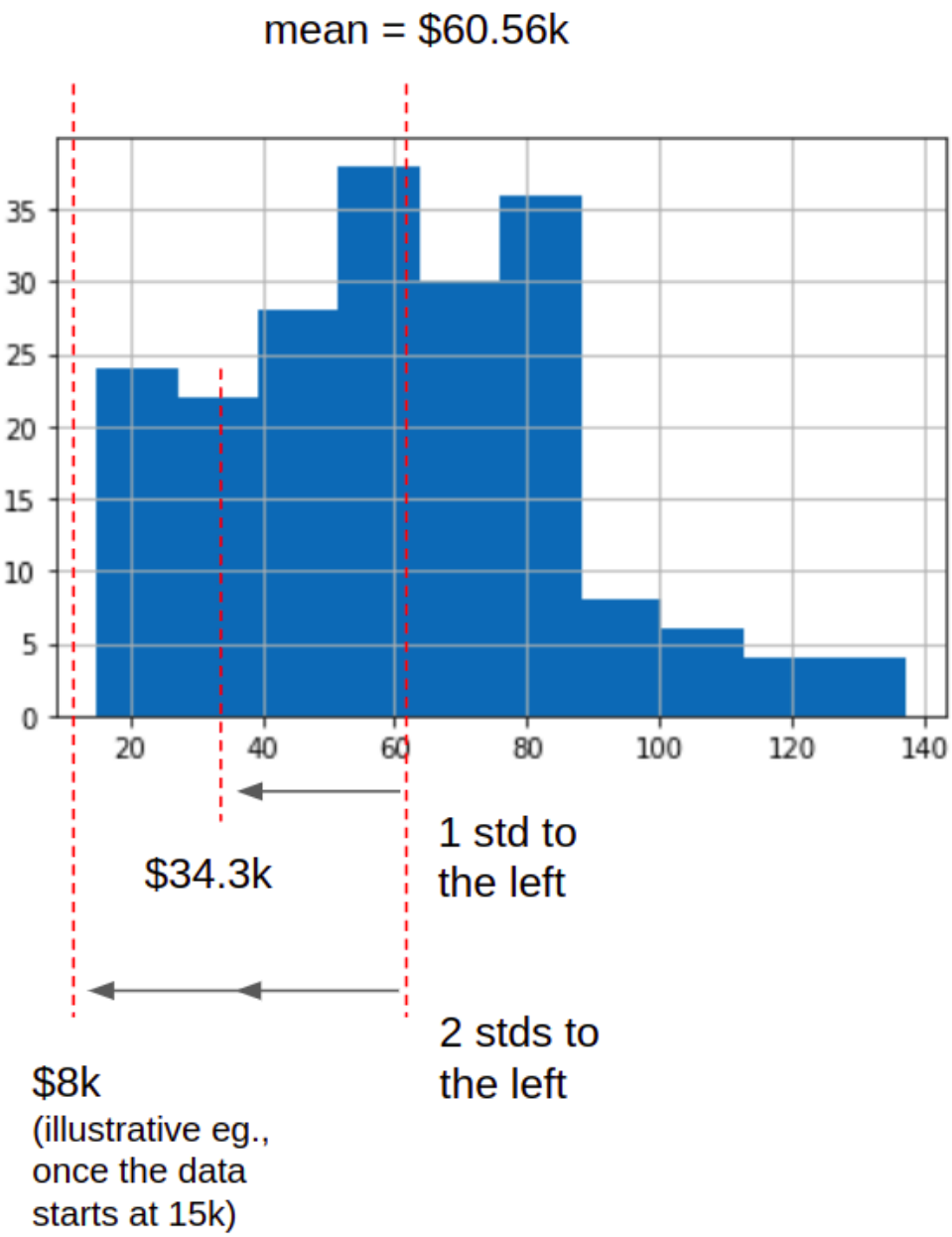
To understand better how our data varies, let's plot the `Annual Income` distribution:

```
customer_data['Annual Income (k$)'].hist()
```

Which will give us:



Notice in the histogram that most of our data, more than 35 customers, is concentrated near the number `60`, on our `mean`, in the horizontal axis. But what happens as we move towards the ends of the distribution? When going towards the left, from the $60.560 mean, the next value we will encounter is $34.300 - the mean ($60.560) minus the standard variation ($26.260). If we go further away to the left of our data distribution a similar rule applies, we subtract the standard variation ($26.260) from the current value ($34.300). Therefore, we'll encounter a value of $8.040. Notice how our data went from $60k to $8k quickly. It is "jumping" $26.260 each time - varying a lot, and that is why we have such high variability.



> *The variability and the size of the data are important in clustering analysis because distance measurements of most clustering algorithms are sensitive to data magnitudes. The difference in size can change the clustering results by making one point seem closer or more distant to another than it actually is, distorting the actual grouping of data.*

So far, we have seen the shape of our data, some of its distributions, and descriptive statistics. With Pandas, we can also list our data types and see if all of our 200 rows are filled or have some `null` values:

```
customer_data.info()
```

This results in:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 5 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   CustomerID              200 non-null    int64
 1   Genre                   200 non-null    object
 2   Age                     200 non-null    int64
 3   Annual Income (k$)      200 non-null    int64
 4   Spending Score (1-100)  200 non-null    int64
dtypes: int64(4), object(1)
memory usage: 7.9+ KB
```

Here, we can see that there are no `null` values in the data and that we have only one categorical column - `Genre`. At this stage, it is important that we have in mind what features seem interesting to be added to the clustering model. If we want to add the Genre column to our model, we will need to transform its values from *categorical* to *numerical*.

Let's see how `Genre` is filled by taking a quick peek at the first 5 values of our data:

```
customer_data.head()
```

This results in:

```
    CustomerID  Genre   Age     Annual Income (k$)  Spending Score (1-100)
0   1           Male    19      15                  39
1   2           Male    21      15                  81
2   3           Female  20      16                  6
3   4           Female  23      16                  77
4   5           Female  31      17                  40
```

It seems that it has only `Female` and `Male` categories. We can be sure of that by taking a look at its unique values with `unique`:

```
customer_data['Genre'].unique()
```

This confirms our assumption:

```
array(['Male', 'Female'], dtype=object)
```

So far, we know that we have only two genres, if we plan to use this feature on our model, `Male` could be transformed to `0` and `Female` to `1`. It is also important to check the proportion between genres, to see if they are balanced. We can do that with the `value_counts()` method and its argument `normalize=True` to show the percentage between `Male` and `Female`:

```
customer_data['Genre'].value_counts(normalize=True)
```

This outputs:

```
Female    0.56
Male      0.44
Name: Genre, dtype: float64
```


We have 56% of women in the dataset and 44% of men. The difference between them is only 16%, and our data is not 50/50 but is *balanced enough* not to cause any trouble. If the results were 70/30, 60/40, then it might have been needed either to collect more data or to employ some kind of data augmentation technique to make that ratio more balanced.

Until now, all features but `Age`, have been briefly explored. In what concerns `Age`, it is usually interesting to divide it into bins to be able to segment customers based on their age groups. If we do that, we would need to transform the age categories into one number before adding them to our model. That way, instead of using the category 15-20 years, we would count how many customers there are in the `15-20` category, and that would be a number in a new column called `15-20`.

> **Advice:** In this guide, we present only brief exploratory data analysis. But you can go further and you should go further. You can see if there are income differences and scoring differences based on genre and age. This not only enriches the analysis but leads to better model results. To go deeper into Exploratory

> Data Analysis, check out the EDA chapter in the "*Hands-On House Price Prediction - Machine Learning in Python*" Guided Project.

After conjecturing on what could be done with both categorical - or categorical to be - `Genre` and `Age` columns, let's apply what has been discussed.

Encoding Variables and Feature Engineering

Let's start by dividing the `Age` into groups that vary in 10, so that we have 20-30, 30-40, 40-50, and so on. Since our youngest customer is 15, we can start at 15 and end at 70, which is the age of the oldest customer in the data. Starting at 15, and ending at 70, we would have 15-20, 20-30, 30-40, 40-50, 50-60, and 60-70 intervals.

To group or *bin* `Age` values into these intervals, we can use the Pandas `cut()` method to cut them into bins and then assign the bins to a new `Age Groups` column:

```
intervals = [15, 20, 30, 40, 50, 60, 70]
col = customer_data['Age']
customer_data['Age Groups'] = pd.cut(x=col, bins=intervals)

# To be able to look at the result stored in the variable
customer_data['Age Groups']
```

This results in:

```
0      (15, 20]
1      (20, 30]
2      (15, 20]
3      (20, 30]
4      (30, 40]
         ...
195    (30, 40]
196    (40, 50]
197    (30, 40]
198    (30, 40]
199    (20, 30]
Name: Age Groups, Length: 200, dtype: category
Categories (6, interval[int64, right]): [(15, 20] < (20, 30] < (30, 40] < (40, 50] < (50
```

Notice that when looking at the column values, there is also a line that specifies we have 6 categories and displays all the binned data intervals. This way, we have categorized our previously numerical data and created a new `Age Groups` feature.

And how many customers do we have in each category? We can quickly know that by grouping the column and counting the values with `groupby()` and `count()`:

```
customer_data.groupby('Age Groups')['Age Groups'].count()
```

This results in:

```
Age Groups
(15, 20]    17
(20, 30]    45
(30, 40]    60
(40, 50]    38
(50, 60]    23
(60, 70]    17
Name: Age Groups, dtype: int64
```

It is easy to spot that most customers are between 30 and 40 years of age, followed by customers between 20 and 30 and then customers between 40 and 50. This is also good information for the Marketing department.

At the moment, we have two categorical variables, `Age` and `Genre`, which we need to transform into numbers to be able to use in our model. There are many different ways of making that transformation - we will use the Pandas `get_dummies()` method that creates a new column for each interval and genre and then fill its values with 0s and 1s- this kind of operation is called *one-hot encoding*. Let's see how it looks:

```
# The _oh means one-hot
customer_data_oh = pd.get_dummies(customer_data)
# Display the one-hot encoded dataframe
customer_data_oh
```

This will give us a preview of the resulting table:

	CustomerID	Age	Annual Income (k$)	Spending Score (1-100)	Genre_Female	Genre_Male	Age Groups_(15, 20]	Age Groups_(20, 30]	Age Groups_(30, 40]	Age Groups_(40, 50]	Age Groups_(50, 60]	Age Groups_(60, 70]
0	1	19	15	39	0	1	1	0	0	0	0	0
1	2	21	15	81	0	1	0	1	0	0	0	0
2	3	20	16	6	1	0	1	0	0	0	0	0
3	4	23	16	77	1	0	0	1	0	0	0	0
4	5	31	17	40	1	0	0	0	1	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...
195	196	35	120	79	1	0	0	0	1	0	0	0
196	197	45	126	28	1	0	0	0	0	1	0	0
197	198	32	126	74	0	1	0	0	1	0	0	0
198	199	32	137	18	0	1	0	0	1	0	0	0
199	200	30	137	83	0	1	0	1	0	0	0	0

200 rows × 12 columns

With the output, it is easy to see that the column `Genre` was split into columns - `Genre_Female` and `Genre_Male`. When the customer is female, `Genre_Female` is equal to `1`, and when the customer is male, it equals `0`.

> **Advice**: If you'd like to read more about One-Hot encoding (also known as categorical encoding sometimes) - read our *"One-Hot Encoding in Python with Pandas and Scikit-Learn"*!

Also, the `Age Groups` column was split into 6 columns, one for each interval, such as `Age Groups_(15, 20]`, `Age Groups_(20, 30]`, and so on. In the same way as `Genre`, when the customer is 18 years old, the `Age Groups_(15, 20]` value is `1` and the value of all other columns is `0`.

The *advantage* of one-hot encoding is the simplicity in representing the column values, it is straightforward to understand what is happening - while the *disadvantage* is that we have now created 8 additional columns, to sum up with the columns we already had.

> **Warning**: If you have a dataset in which the number of one-hot encoded columns exceeds the number of rows, it is best to employ another encoding method to avoid data dimensionality issues.

One-hot encoding also adds 0s to our data, making it more sparse, which can be a problem for some algorithms that are sensitive to data sparsity.

For our clustering needs, one-hot encoding seems to work. But we can plot the data to see if there really are distinct groups for us to cluster.

Basic Plotting and Dimensionality Reduction

Our dataset has 11 columns, and there are some ways in which we can visualize that data. The first one is by plotting it in 10-dimensions (good luck with that). Ten because the `Customer_ID` column is not being considered. The second one is by plotting our initial numerical features, and the third is by transforming our 10 features into 2 - therefore, performing a dimensionality reduction.

Plotting Each Pair of Data

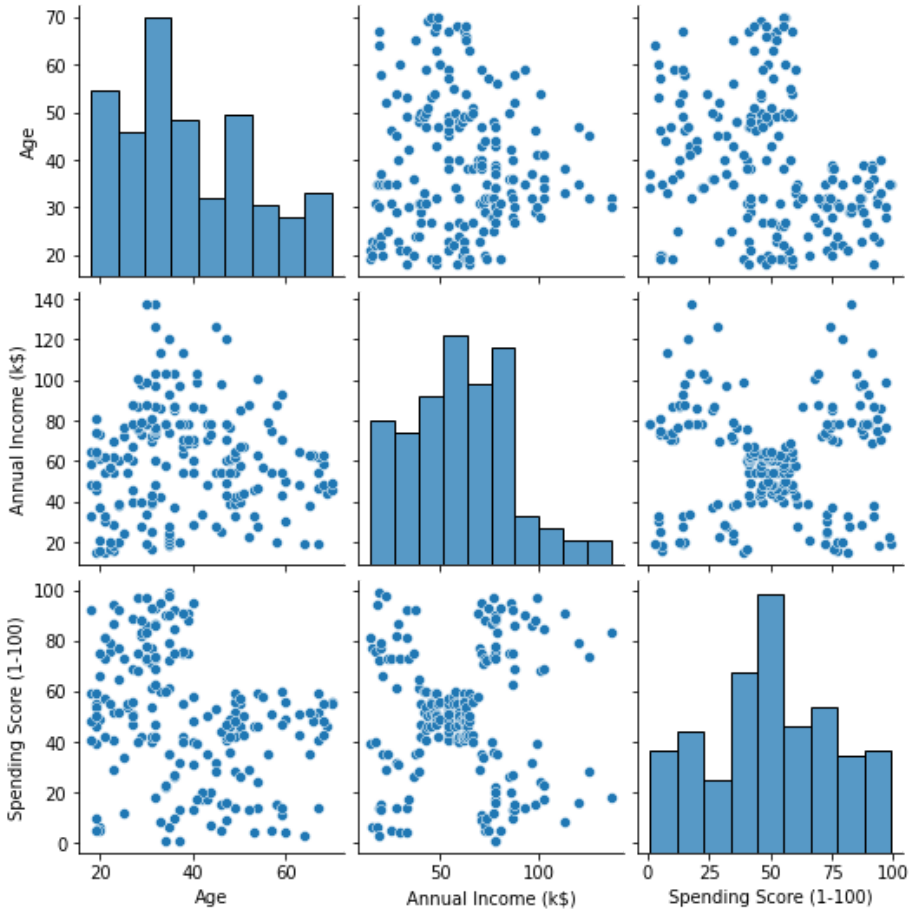
Since plotting 10 dimensions is a bit impossible, we'll opt to go with the second approach - we'll plot our initial features. We can choose two of them for our clustering analysis. One way we can see all of our data pairs combined is with a Seaborn `pairplot()`:

```
import seaborn as sns

# Dropping CustomerID column from data
customer_data = customer_data.drop('CustomerID', axis=1)

sns.pairplot(customer_data)
```

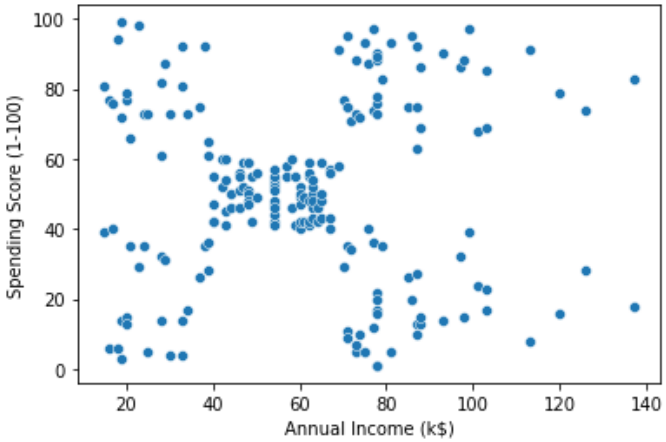
Which displays:



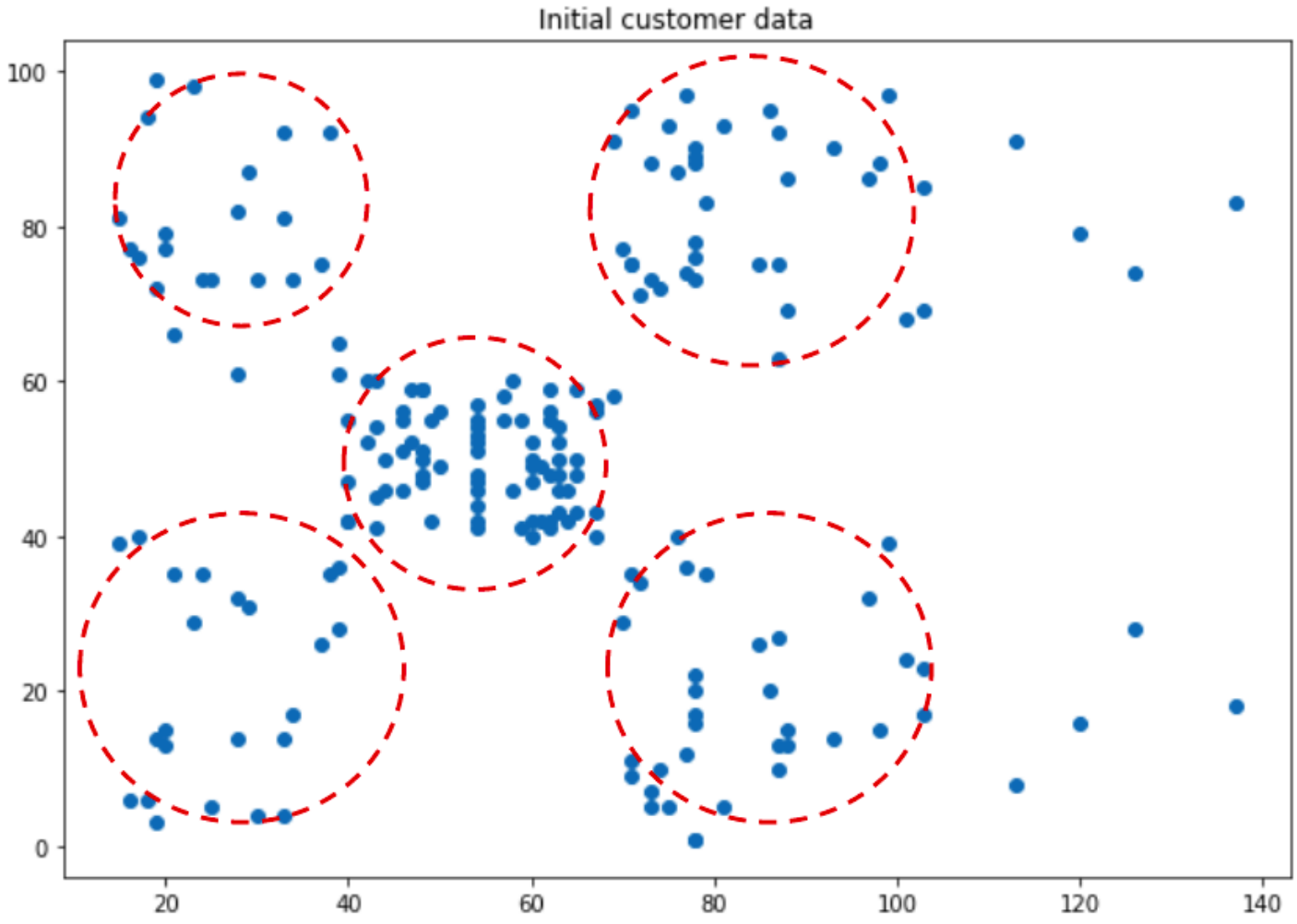
At a glance, we can spot the scatter plots that seem to have groups of data. One that seems interesting is the scatter plot that combines `Annual Income` and `Spending Score`. Notice that there is no clear separation between other variable scatter plots. At the most, we can maybe tell that there are two distinct concentrations of points in the `Spending Score` vs `Age` scatter plot.

Both scatter plots consisting of `Annual Income` and `Spending Score` are essentially the same. We can see it twice because the x and y-axis were exchanged. By taking a look at any of them, we can see what appears to be five different groups. Let's plot just those two features with a Seaborn `scatterplot()` to take a closer look:

```
sns.scatterplot(x=customer_data['Annual Income (k$)'],
                y=customer_data['Spending Score (1-100)'])
```



By looking closer, we can definitely distinguish 5 different groups of data. It seems our customers can be clustered based on how much they make in a year and how much they spend. This is another relevant point in our analysis. It is important that we are only taking two features into consideration to group our clients. Any other information we have about them is not entering the equation. This gives the analysis meaning - if we know how much a client earns and spends, we can easily find the similarities we need.



That's great! So far, we already have two variables to build our model. Besides what this represents, it also makes the model simpler, parsimonious, and more explainable.

> **Note:** Data Science usually favors as simple approaches as possible. Not only because it is easier to explain for the business, but also because it is more direct - with 2 features and an explainable model, it is clear what the model is doing and how it is working.

Plotting Data After Using PCA

It seems our second approach is probably the best, but let's also take a look at our third approach. It can be useful when we can't plot the data because it has too many dimensions, or when there are no data concentrations or clear separation in groups. When those situations occur, it's recommended to try reducing data dimensions with a method called *Principal Component Analysis (PCA)*.

> **Note:** Most people use PCA for dimensionality reduction before visualization. There are other methods that help in data visualization prior to clustering, such as *Density-Based Spatial Clustering of Applications with Noise (DBSCAN)* and *Self-Organizing Maps (SOM)* clustering. Both are clustering algorithms, but can also be used for data visualization. Since clustering analysis has no golden standard, it is important to compare different visualizations and different algorithms.

PCA will reduce the dimensions of our data while trying to preserve as much of its information as possible. Let's first get an idea about how PCA works, and then we can choose how many data dimensions we will reduce our data to.

For each pair of features, PCA sees if the greater values of one variable correspond with the greater values of the other variable, and it does the same for the lesser values. So, it essentially computes how much the feature values vary towards one another - we call that their *covariance*. Those results are then organized into a matrix, obtaining a *covariance matrix*.

After getting the covariance matrix, PCA tries to find a linear combination of features that best explains it - it fits linear models until it identifies the one that explains the ***maximum*** *amount of variance*.

> **Note**: PCA is a linear transformation, and linearity is sensitive to the scale of data. Therefore, PCA works best when all data values are on the same scale. This can be done by subtracting the column *mean* from its values and dividing the result by its standard deviation. That is called *data standardization*. Prior to using PCA, make sure the data is scaled! If you're not sure how, read our *"Feature Scaling Data with Scikit-Learn for Machine Learning in Python"*!

With the best line (linear combination) found, PCA gets the directions of its axes, called *eigenvectors*, and its linear coefficients, the *eigenvalues*. The combination of the eigenvectors and eigenvalues - or axes directions and coefficients - are the *Principal Components* of PCA. And that is when we can choose our number of dimensions based on the explained variance of each feature, by understanding which principal components we want to keep or discard based on how much variance they explain.

After obtaining the principal components, PCA uses the eigenvectors to form a vector of features that reorient the data from the original axes to the ones represented by the principal components - that's how the data dimensions are reduced.

> **Note:** One important detail to take into consideration here is that, due to its linear nature, PCA will concentrate most of the explained variance in the first principal components. So, when looking at the explained variance, usually our first two components will suffice. But that might be misleading in some cases - so try to keep comparing different plots and algorithms when clustering to see if they hold similar results.

Before applying PCA, we need to choose between the `Age` column or the `Age Groups` columns in our previously one-hot encoded data. Since both columns represent the same information, introducing it twice affects our data variance. If the `Age Groups` column is chosen, simply remove the `Age` column using the Pandas `drop()` method and reassign it to the `customer_data_oh` variable:

```
customer_data_oh = customer_data_oh.drop(['Age'], axis=1)
customer_data_oh.shape # (200, 10)
```

Now our data has 10 columns, which means we can obtain one principal component by column and choose how many of them we will use by measuring how much introducing one new dimension explains more of our data variance.

Let's do that with Scikit-Learn `PCA`. We will calculate the explained variance of each dimension, given by `explained_variance_ratio_`, and then look at their cumulative sum with `cumsum()`:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
pca.fit_transform(customer_data_oh)
pca.explained_variance_ratio_.cumsum()
```

Our cumulative explained variances are:

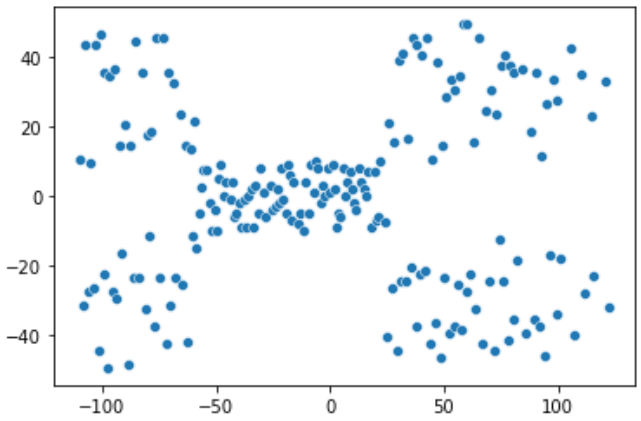
```
array([0.509337  , 0.99909504, 0.99946364, 0.99965506, 0.99977937,
       0.99986848, 0.99993716, 1.        , 1.        , 1.        ])
```

We can see that the first dimension explains 50% of the data, and when combined to the second dimension, they explain 99% percent. This means that the first 2 dimensions already explain 99% of our data. So we can apply a PCA with 2 components, obtain our principal components and plot them:

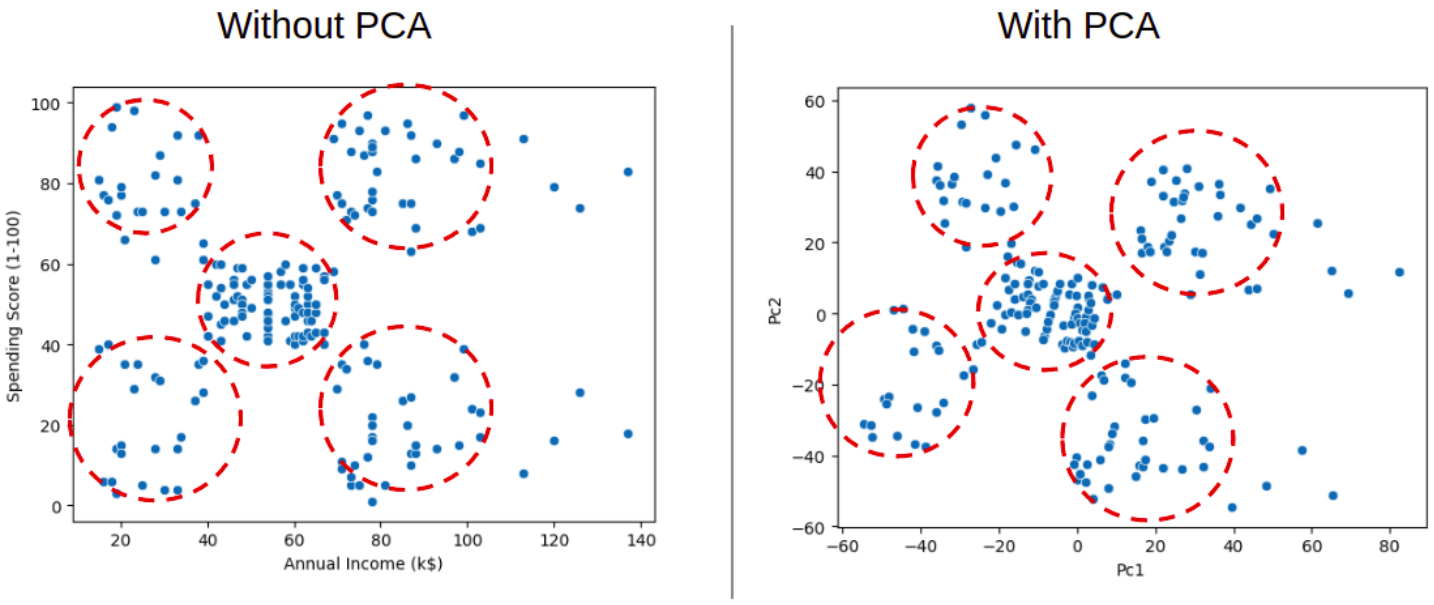
```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
pcs = pca.fit_transform(customer_data_oh)

pc1_values = pcs[:,0]
pc2_values = pcs[:,1]
sns.scatterplot(x=pc1_values, y=pc2_values)
```



The data plot after PCA is very similar to the plot that is using only two columns of the data without PCA. Notice that the points that are forming groups are closer, and a little more concentrated after the PCA than before.



> **Advice**: To see other applications of PCA, take a look at the *"Implementing PCA in Python with Scikit-Learn"* guide.

Visualizing Hierarchical Structure with Dendrograms

So far, we have explored the data, one-hot encoded categorical columns, decided which columns were fit for clustering, and reduced data dimensionality. The plots indicate we have 5 clusters in our data, but there's also another way to visualize the relationships between our points and help determine the number of clusters - by creating a **dendrogram** (commonly misspelled as "dendogram"). *Dendro* means *tree* in Latin.

The *dendrogram* is a result of the linking of points in a dataset. It is a visual representation of the hierarchical clustering process. And how does the hierarchical clustering process work? Well... it depends - probably an answer you've already heard a lot in Data Science.

Understanding Hierarchical Clustering

When the ***Hierarchical Clustering Algorithm (HCA)*** starts to link the points and find clusters, it can first split points into 2 large groups, and then split each of those two groups into smaller 2 groups, having 4 groups in total, which is the **divisive** and **top-down** approach.

Alternatively, it can do the opposite - it can look at all the data points, find 2 points that are closer to each other, link them, and then find other points that are the closest ones to those linked points and keep building the 2 groups from the **bottom-up**. Which is the **agglomerative** approach we will develop.

Steps to Perform Agglomerative Hierarchical Clustering

To make the agglomerative approach even clear, there are steps of the *Agglomerative Hierarchical Clustering (AHC)* algorithm:

1. At the start, treat each data point as one cluster. Therefore, the number of clusters at the start will be K - while K is an integer representing the number of data points.
2. Form a cluster by joining the two closest data points resulting in K-1 clusters.
3. Form more clusters by joining the two closest clusters resulting in K-2 clusters.
4. Repeat the above three steps until one big cluster is formed.

> **Note**: For simplification, we are saying "two closest" data points in steps 2 and 3. But there are more ways of linking points as we will see in a bit.

> *If you invert the steps of the ACH algorithm, going from 4 to 1 - those would be the steps to **Divisive Hierarchical Clustering (DHC)**.*

Notice that HCAs can be either divisive and top-down, or agglomerative and bottom-up. The top-down DHC approach works best when you have fewer, but larger clusters, hence it's more computationally expensive. On the other hand, the bottom-up AHC approach is fitted for when you have many smaller clusters. It is computationally simpler, more used, and more available.

> **Note:** Either top-down or bottom-up, the dendrogram representation of the clustering process will always start with a division in two and end up with each individual point discriminated, once its underlying structure is of a binary tree.

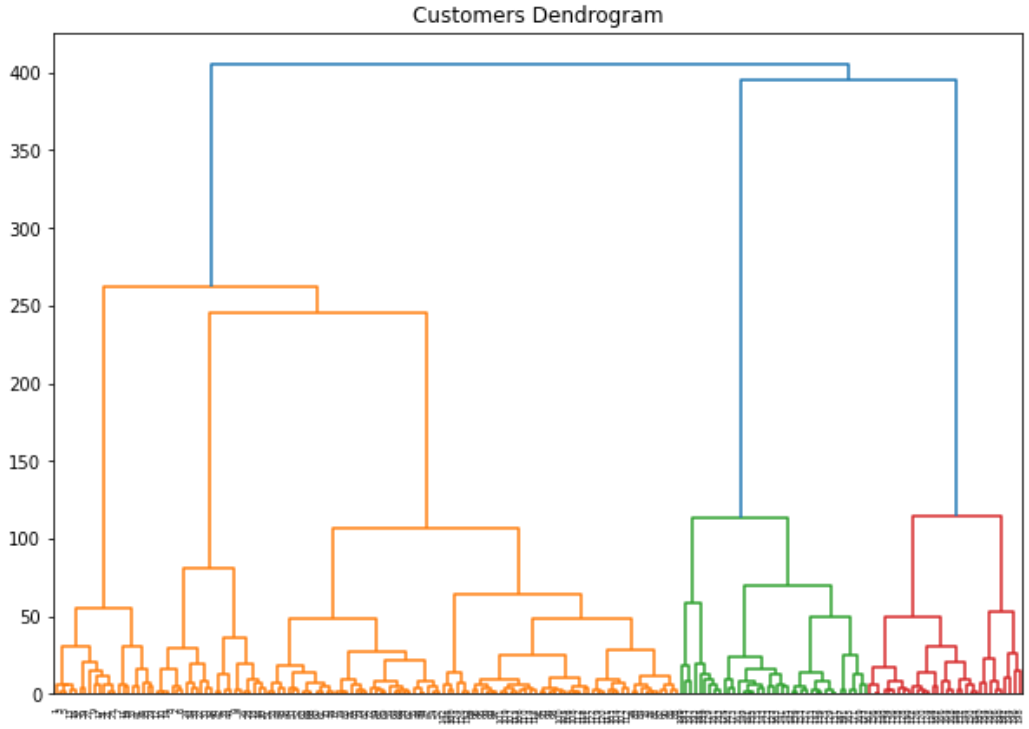
Let's plot our customer data dendrogram to visualize the hierarchical relationships of the data. This time, we will use the `scipy` library to create the dendrogram for our dataset:

```
import scipy.cluster.hierarchy as shc
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 7))
plt.title("Customers Dendrogram")

# Selecting Annual Income and Spending Scores by index
selected_data = customer_data_oh.iloc[:, 1:3]
clusters = shc.linkage(selected_data,
            method='ward',
            metric="euclidean")
shc.dendrogram(Z=clusters)
plt.show()
```

The output of the script looks like this:



In the script above, we've generated the clusters and sub-clusters with our points, defined how our points would link (by applying the `ward` method), and how to measure the distance between points (by using the `euclidean` metric).

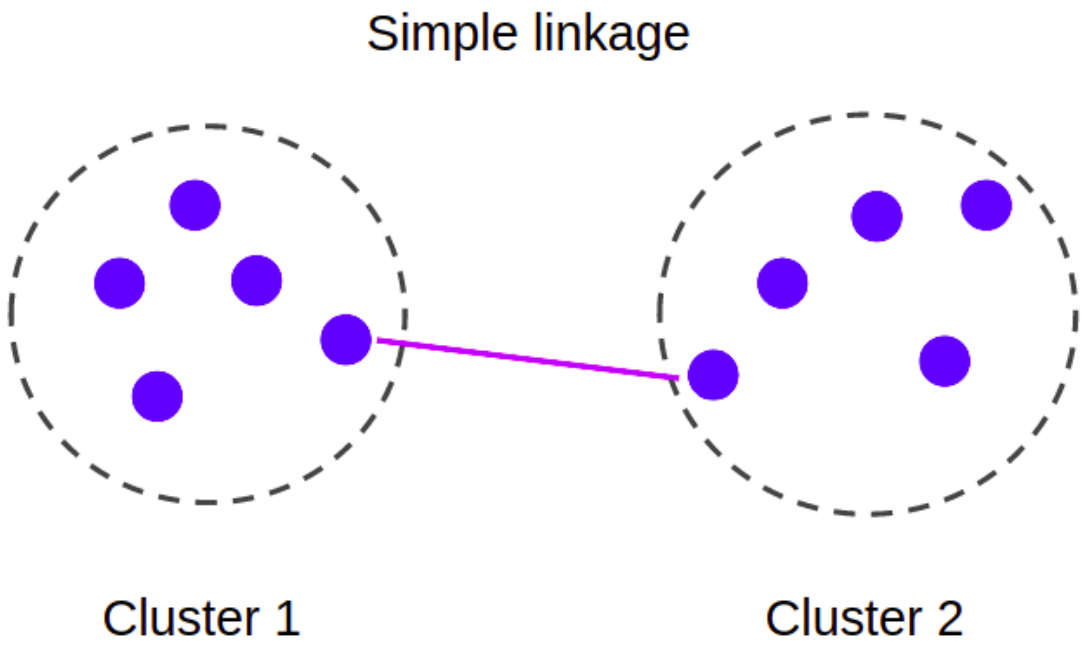
With the plot of the dendrogram, the described processes of DHC and AHC can be visualized. To visualize the top-down approach, start from the top of the dendrogram and go down, and do the opposite, starting down and moving upwards to visualize the bottom-up approach.

Linkage Methods

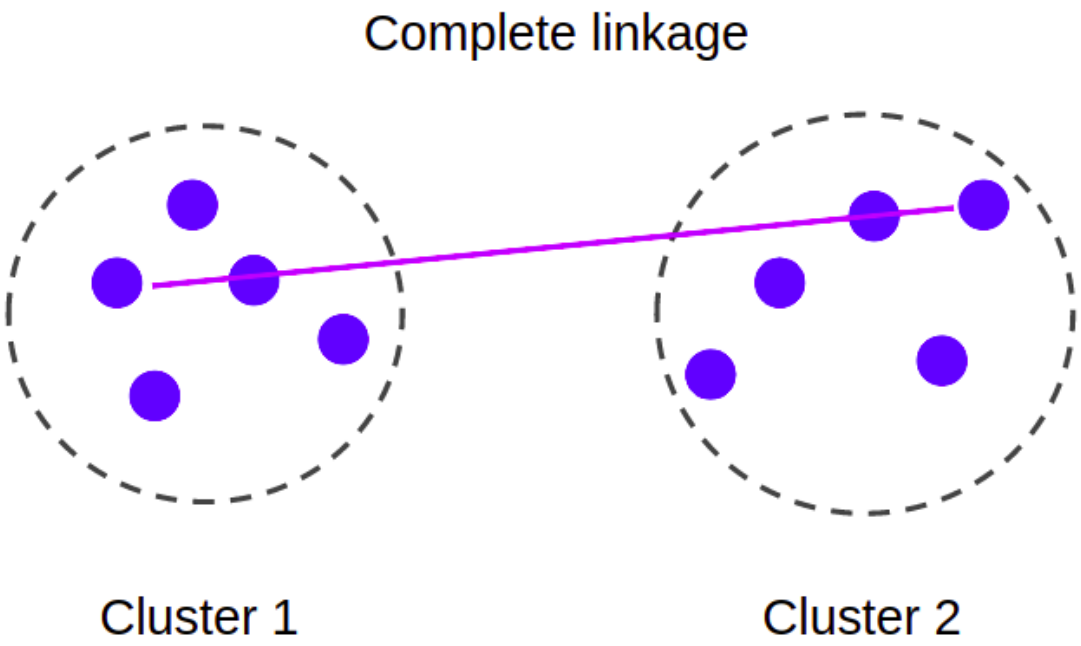
There are many other linkage methods, by understanding more about how they work, you will be able to choose the appropriate one for your needs. Besides that, each of them will yield different results when applied. There is not a fixed rule in clustering analysis, if possible, study the nature of the problem to see which fits its best, test different methods, and inspect the results.

Some of the linkage methods are:

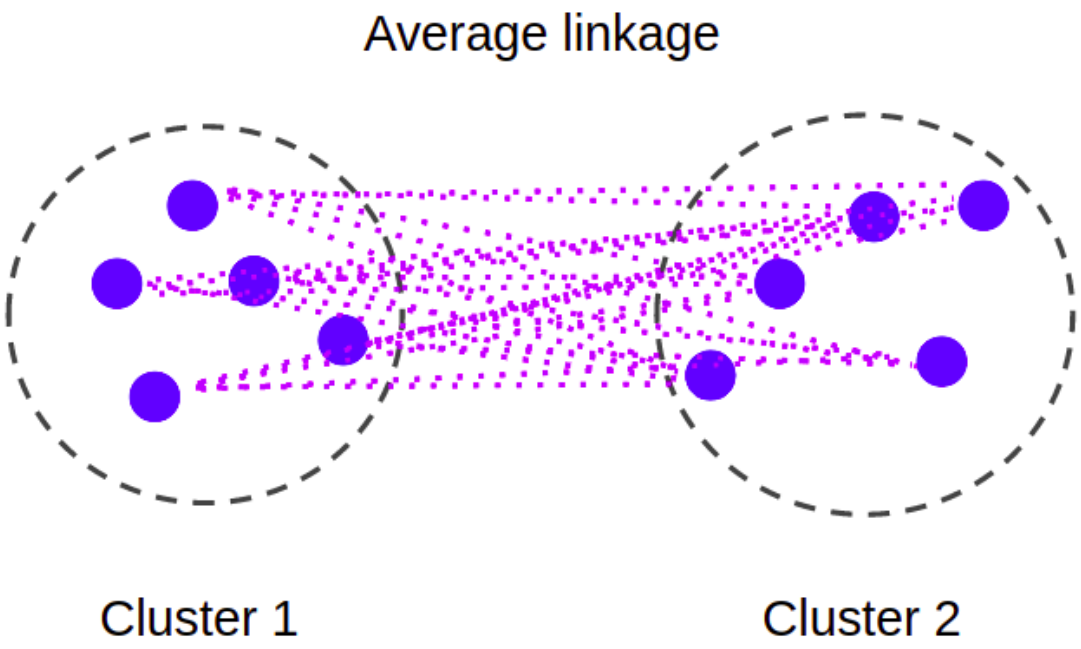
- **Single linkage**: also referred to as ***Nearest Neighbor (NN)***. The distance between clusters is defined by the distance between their closest members.



- **Complete linkage**: also referred to as ***Furthest Neighbor (FN)***, ***Farthest Point Algorithm***, or ***VoorHees Algorithm***. The distance between clusters is defined by the distance between their furthest members. This method is computationally expensive.

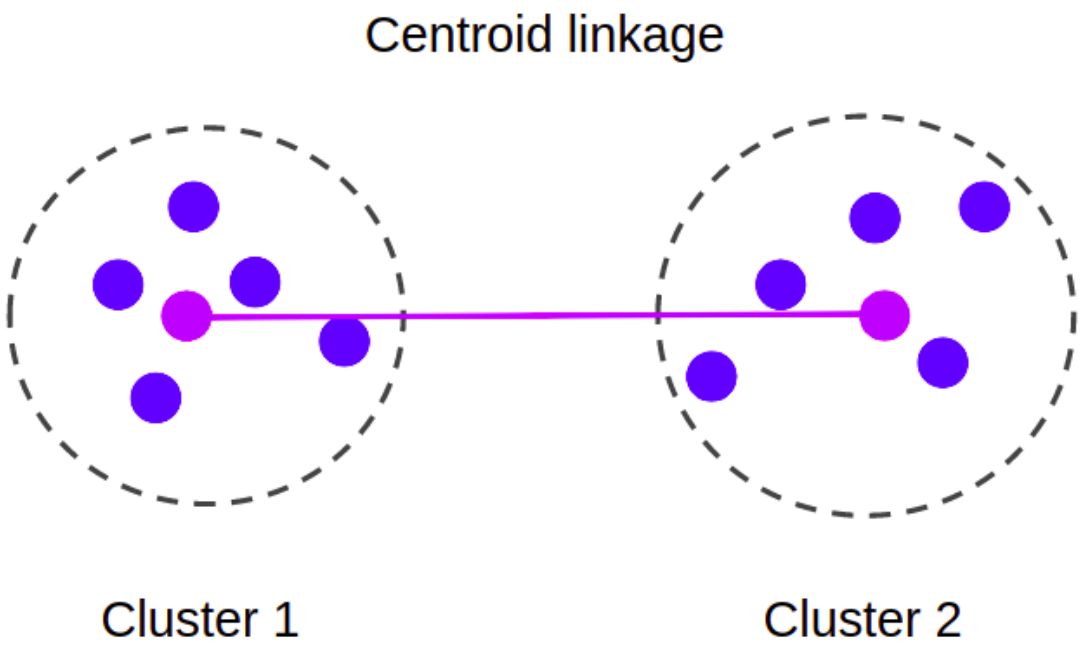


- **Average linkage**: also known as **UPGMA** (Unweighted Pair Group Method with Arithmetic mean)*. The percentage of the number of points of each cluster is calculated with respect to the number of points of the two clusters if they were merged.

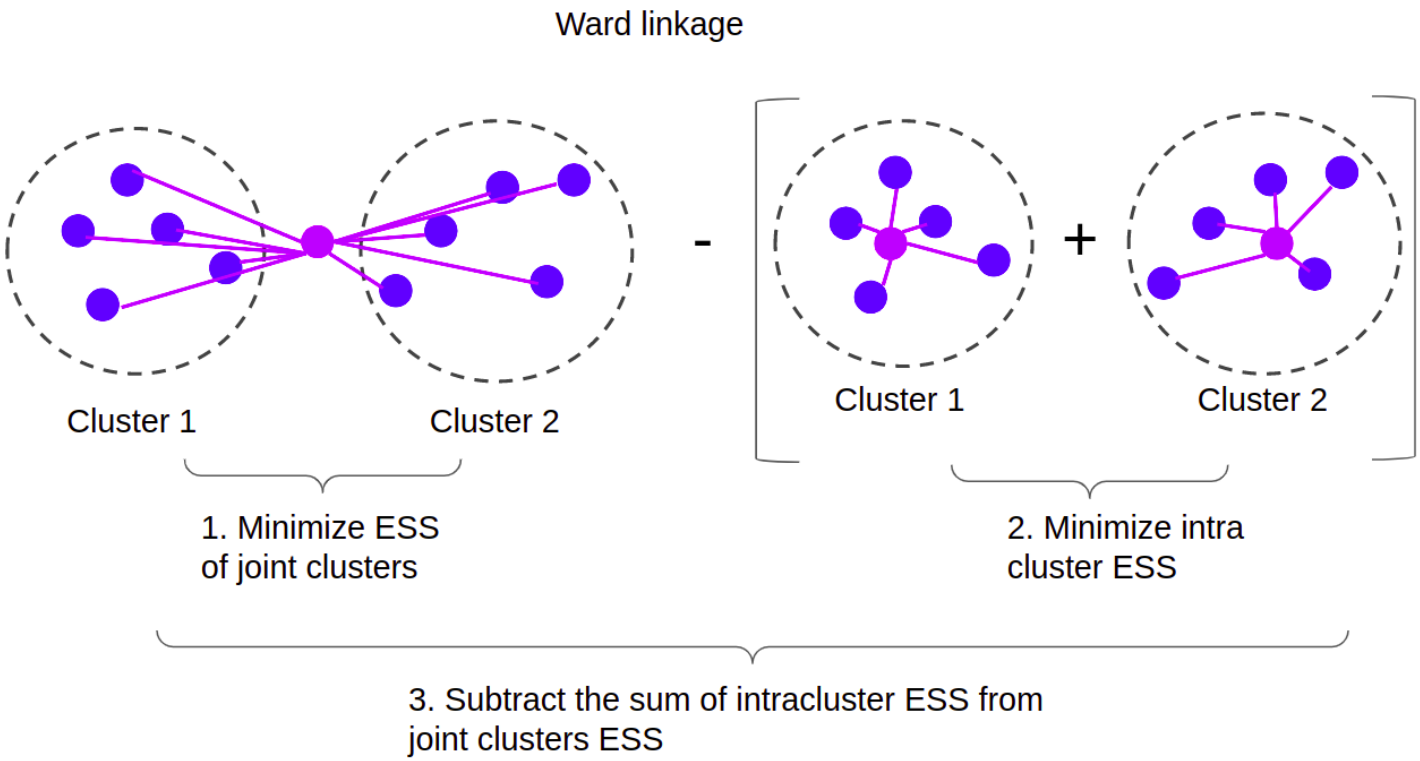


- **Weighted linkage**: also known as **WPGMA** (Weighted Pair Group Method with Arithmetic mean)*. The individual points of the two clusters contribute to the aggregated distance between a smaller and a bigger cluster.

- **Centroid linkage**: also referred to as **UPGMC** (Unweighted Pair Group Method using Centroids)*. A point defined by the mean of all points (centroid) is calculated for each cluster and the distance between clusters is the distance between their respective centroids.



- **Ward linkage**: Also known as **MISSQ** (Minimal Increase of Sum-of-Squares)*. It specifies the distance between two clusters, computes the sum of squares error (ESS), and successively chooses the next clusters based on the smaller ESS. Ward's Method seeks to minimize the increase of ESS at each step. Therefore, minimizing error.



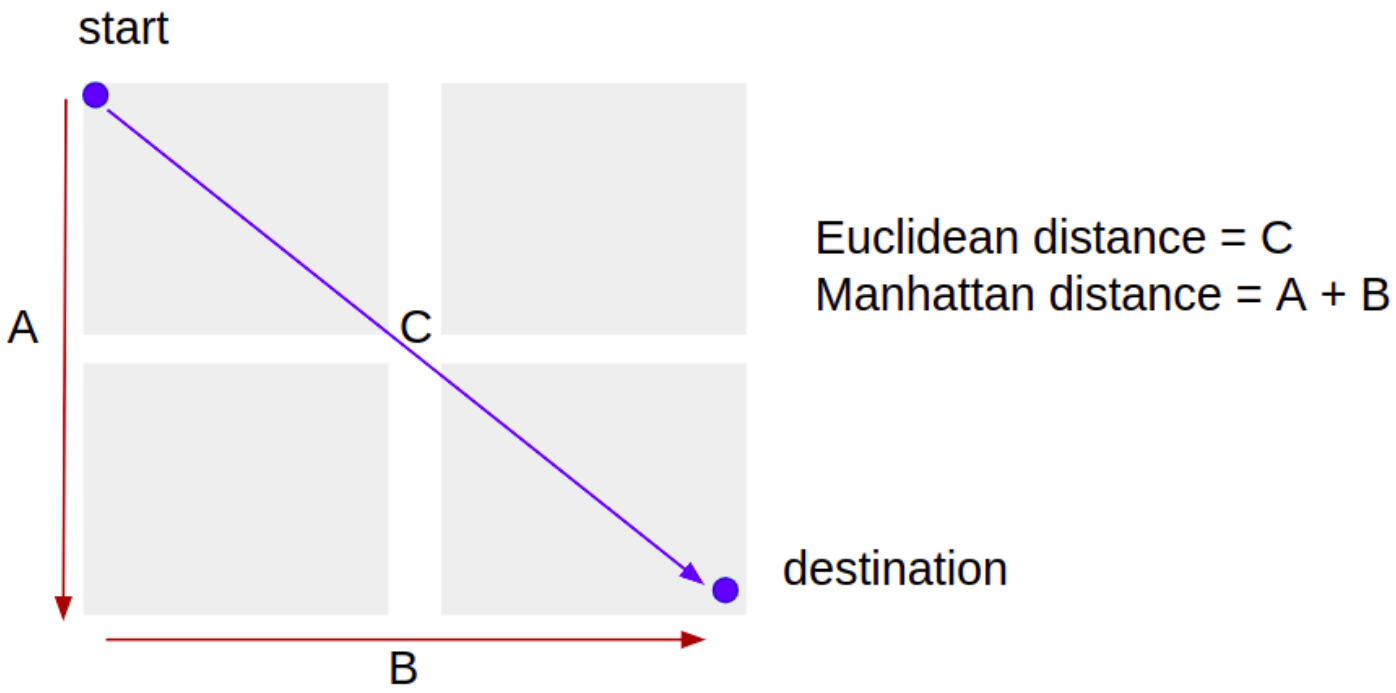
Distance Metrics

Besides the linkage, we can also specify some of the most used distance metrics:

- **Euclidean**: also referred to as **Pythagorean or straight-line** distance. It computes the distance between two points in space, by measuring the length of a line segment that passes between them. It uses the Pythagorean theorem and the distance value is the result `(c)` of the equation:

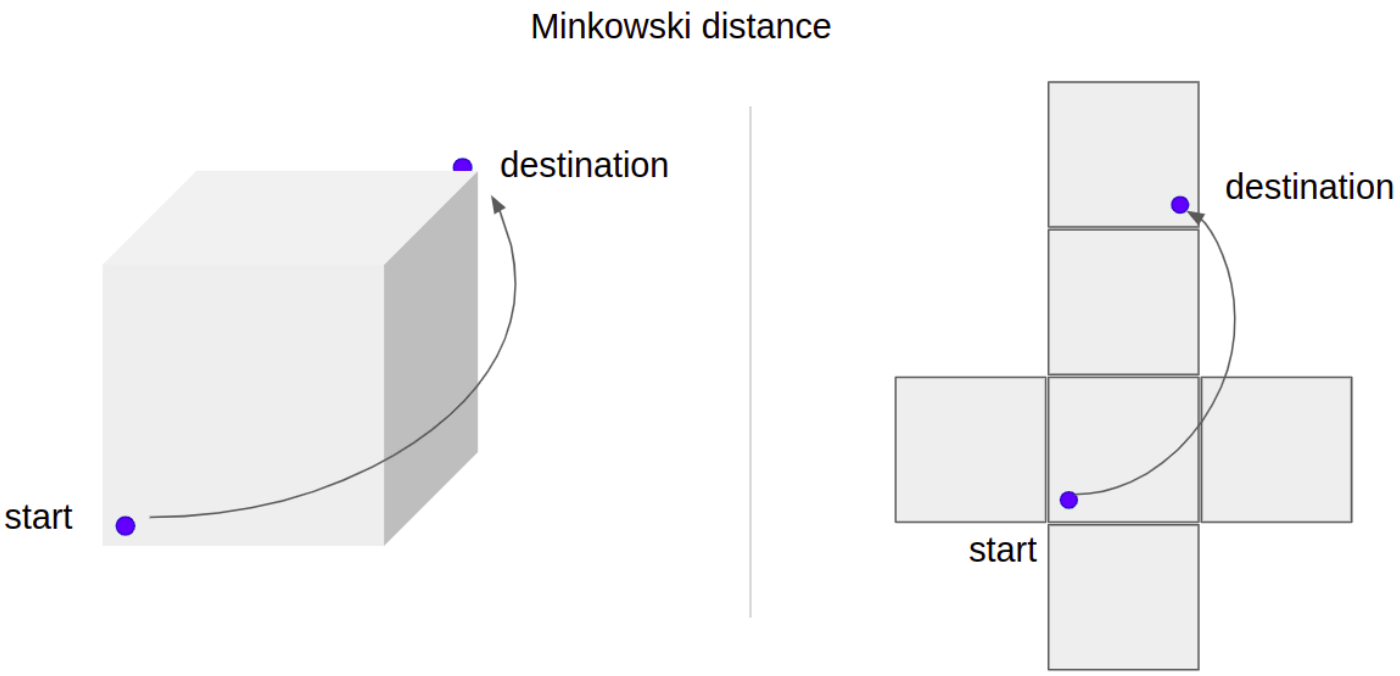
$$c^2 = a^2 + b^2$$

- **Manhattan**: also called **City-block, Taxicab** distance. It is the sum of absolute differences between the measures in all dimensions of two points. If those dimensions are two, it is analogous to making a right and then left when walking one block.



- **Minkowski**: it is a generalization of both Euclidean and Manhattan distances. It is a way to calculate distances based on the absolute differences to the order of the Minkowski metric **p**. Although it is defined for any `p > 0`, it is rarely used for values other than 1, 2, and ∞ (infinite). Minkowski distance is the same as Manhattan distance when `p=1`, and the same as Euclidean distance when `p=2`.

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$



- **Chebyshev**: also known as **Chessboard** distance. It is the extreme case of Minkowski distance. When we use infinity as the value of the parameter `p (p = ∞)`, we end up with a metric that defines distance as the maximal absolute difference between coordinates.
- **Cosine**: it is the angular cosine distance between two sequences of points, or vectors. The cosine similarity is the dot product of the vectors divided by the product of their lengths.
- **Jaccard**: measures the similarity between finite sets of points. It is defined as the total number of points (cardinality) in the common points in each set (intersection), divided by the total number of points (cardinality) of the total points of both sets (union).
- **Jensen-Shannon**: based on the Kullback-Leibler divergence. It considers the points' probability distributions and measures the similarity between those distributions. It is a popular method of probability theory and statistics.

> **Note:** For a complete list of available linkages, visit the Scipy documentation on linkages.
> Also, for a complete list of available metrics, and what they're used for, visit the SciPy point distance documentation.

We have chosen *Ward* and *Euclidean* for the dendrogram because they are the most commonly used method and metric. They usually give good results since Ward links points based on minimizing the errors, and Euclidean works well in lower dimensions.

In this example, we are working with two features (columns) of the marketing data, and 200 observations or rows. Since the number of observations is larger than the number of features (200 > 2), we are working in a low-dimensional space.

> *When the number of features `(f)` is larger than the number of observations `(N)` - mostly written as `f >> N`, it means that we have a high dimensional space.*

If we were to include more attributes, so we have more than 200 features, the Euclidean distance might not work very well, since it would have difficulty in measuring all the small distances in a very large space that only gets larger. In other words, the Euclidean distance approach has difficulties working with the data **sparsity**. This is an issue that is called **the curse of dimensionality**. The distance values would get so small, as if they became "diluted" in the larger space, distorted until they became 0.

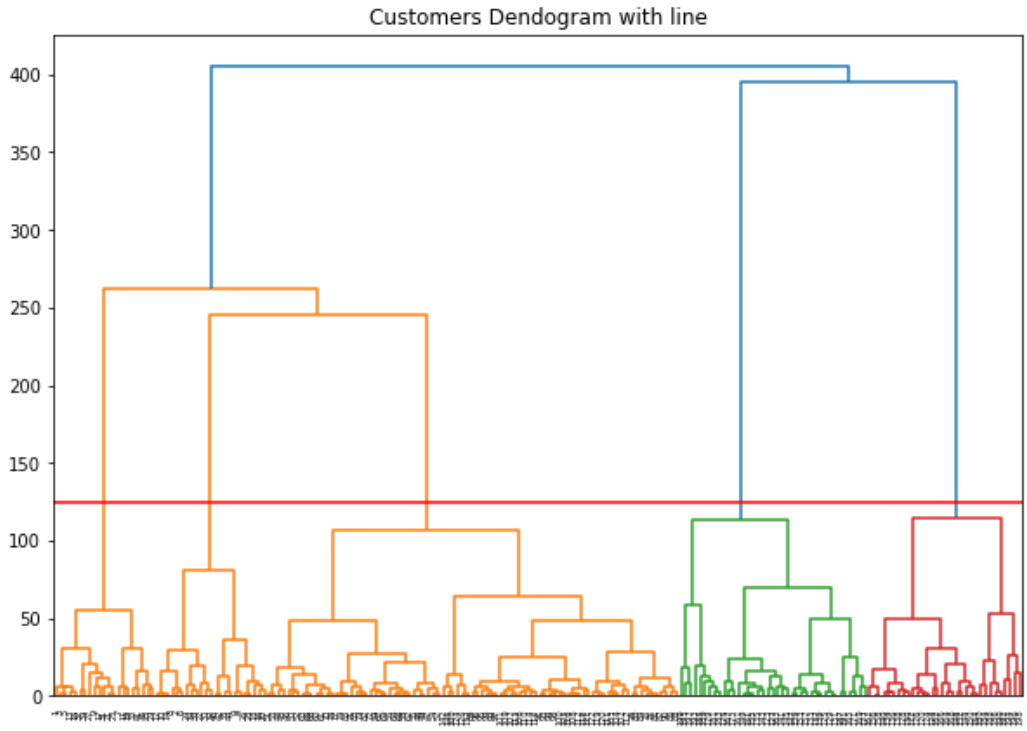
> **Note:** If you ever encounter a dataset with `f >> p`, you will probably use other distance metrics, such as the **Mahalanobis** distance. Alternatively, you can also reduce the dataset dimensions, by using **Principal Component Analysis (PCA)**. This problem is frequent especially when clustering biological sequencing data.

We've already discussed metrics, linkages, and how each one of them can impact our results. Let's now continue the dendrogram analysis and see how it can give us an indication of the number of clusters in our dataset.

Finding an interesting number of clusters in a dendrogram is the same as finding the largest horizontal space that doesn't have any vertical lines (the space with the longest vertical lines). This means that there's more separation between the clusters.

We can draw a horizontal line that passes through that longest distance:

```
plt.figure(figsize=(10, 7))
plt.title("Customers Dendrogram with line")
clusters = shc.linkage(selected_data,
            method='ward',
            metric="euclidean")
shc.dendrogram(clusters)
plt.axhline(y = 125, color = 'r', linestyle = '-')
```



After locating the horizontal line, we count how many times our vertical lines were crossed by it - in this example, 5 times. So 5 seems a good indication of the number of clusters that have the most distance between them.

> **Note**: The dendrogram should be considered only as a reference when used to choose the number of clusters. It can easily get that number way off and is completely influenced by the type of linkage and distance metrics. When conducting an in-depth cluster analysis, it is advised to look at dendrograms with different linkages and metrics and to look at the results generated with the first three lines in which the clusters have the most distance between them.

Implementing an Agglomerative Hierarchical Clustering

Using Original Data

So far we've calculated the suggested number of clusters for our dataset that corroborate with our initial analysis and our PCA analysis. Now we can create our agglomerative hierarchical clustering model using Scikit-Learn `AgglomerativeClustering` and find out the labels of marketing points with `labels_` :

```
from sklearn.cluster import AgglomerativeClustering

clustering_model = AgglomerativeClustering(n_clusters=5, affinity='euclidean', linkage='
clustering_model.fit(selected_data)
clustering_model.labels_
```

This results in:


```
array([4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3,
       4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 1,
       4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 2, 0, 2, 0, 2,
       1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 1, 2, 0, 2, 1, 2, 0, 2, 0, 2, 0, 2,
       0, 2, 0, 2, 0, 2, 1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
       0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
       0, 2])
```

We have investigated a lot to get to this point. And what does these labels mean? Here, we have each point of our data labeled as a group from 0 to 4:

```
data_labels = clustering_model.labels_
sns.scatterplot(x='Annual Income (k$)',
                y='Spending Score (1-100)',
                data=selected_data,
                hue=data_labels,
                palette="rainbow").set_title('Labeled Customer Data')
```

This is our final "cluster-ized" data. You can see the color-coded data points in the form of five clusters.

The data points in the bottom right (label: `0`, purple data points) belong to the customers with high salaries but low spending. These are the customers that spend their money carefully.

Similarly, the customers at the top right (label: `2`, green data points), are the customers with high salaries and high spending. These are the type of customers that companies target.

The customers in the middle (label: `1`, blue data points) are the ones with average income and average spending. The highest numbers of customers belong to this category. Companies can also target these customers given the fact that they are in huge numbers.

The customers in the bottom left (label: `4`, red) are the customers that have low salaries and low spending, they might be attracted by offering promotions.

And finally, the customers in the upper left (label: `3`, orange data points) are the ones with high income and low spending, which are ideally targeted by marketing.

Using the Result from PCA

If we were in a different scenario, in which we had to reduce the dimensionality of data. We could also easily plot the "cluster-ized" PCA results. That can be done by creating another agglomerative clustering model and obtaining a data label for each principal component:

```
clustering_model_pca = AgglomerativeClustering(n_clusters=5, affinity='euclidean', linka
clustering_model_pca.fit(pcs)
```

```
data_labels_pca = clustering_model_pca.labels_

sns.scatterplot(x=pc1_values,
                y=pc2_values,
                hue=data_labels_pca,
                palette="rainbow").set_title('Labeled Customer Data Reduced with PCA')
```

Observe that both results are very similar. The main difference is that the first result with the original data is much easier to explain. It is clear to see that customers can be divided into five groups by their annual income and spending score. While, in the PCA approach, we are taking all of our features into consideration, as much as we can look at the variance explained by each of them, this is a harder concept to grasp, especially when reporting to a Marketing department.

> *The least we have to transform our data, the better.*

If you have a very large and complex dataset in which you must perform a dimensionality reduction prior to clustering - try to analyze the linear relationships between each of the features and their residuals to back up the use of PCA and enhance the explicability of the process. By making a linear model per pair of features, you will be able to understand how the features interact.

If the data volume is so large, it becomes impossible to plot the pairs of features, select a sample of your data, as balanced and close to the normal distribution as possible and perform the analysis on the sample first, understand it, fine-tune it - and apply it later to the whole dataset.

You can always choose different clustering visualization techniques according to the nature of your data (linear, non-linear) and combine or test all of them if necessary.

Conclusion

The clustering technique can be very handy when it comes to unlabeled data. Since most of the data in the real world are unlabeled and annotating the data has higher costs, clustering techniques can be used to label unlabeled data.

In this guide, we have brought up a real data science problem, since clustering techniques are largely used in marketing analysis (and also in biological analysis). We have also explained many of the investigation steps to get to a good hierarchical clustering model and how to read dendrograms and questioned if PCA is a necessary step. Our main objective is that some of the pitfalls and different scenarios in which we can find hierarchical clustering are covered.

Happy clustering!

#python #scikit-learn #data science #pandas #matplotlib #seaborn #data visualization #clustering

Last Updated: November 16th, 2023

You might also like...

- Definitive Guide to Logistic Regression in Python
- Definitive Guide to K-Means Clustering with Scikit-Learn
- Seaborn Boxplot - Tutorial and Examples
- Seaborn Scatter Plot - Tutorial and Examples

Cássia Sampaio *Author*

Data Scientist, Research Software Engineer, and teacher. Cassia is passionate about transformative processes in data, technology and life. She is graduated in Philosophy and Information Systems, with a Strictu Sensu Master's Degree in the field of Foundations Of Mathematics.

Dimitrije Stamenic
Editor

Project

Data Visualization in Python: Visualizing EEG Brainwave Data

#python #matplotlib #seaborn #data visualization

Electroencephalography (EEG) is the process of recording an individual's brain activity - from a macroscopic scale. It's a non-invasive (external) procedure and collects aggregate, not...

Details →