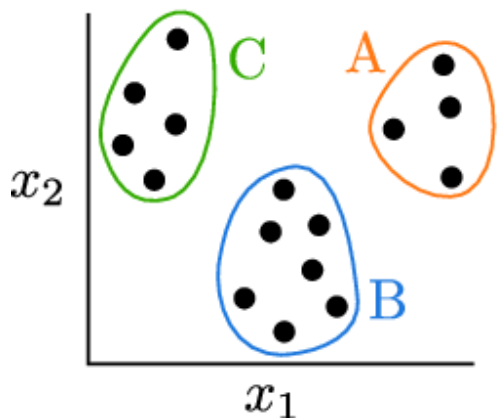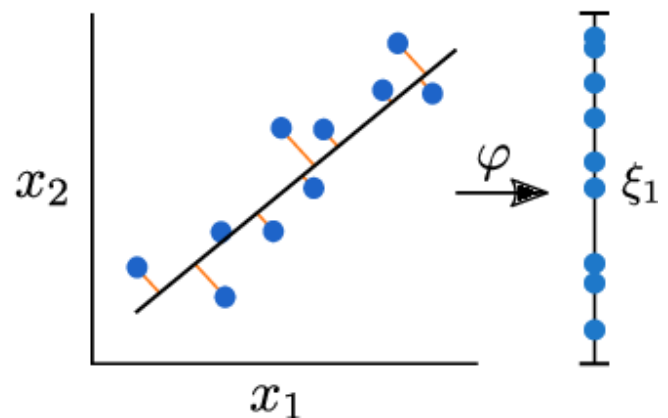# Unsupervised Learning and Recommendations with Scikit Learn

This tutorial is a part of the [Machine Learning with Python: Zero to GBMs](#) and [Zero to Data Science Bootcamp by Jovian](#).

The following topics are covered in this tutorial:

- Overview of unsupervised learning algorithms in Scikit-learn
- Clustering algorithms: K Means, DBScan, Hierarchical clustering etc.
- Dimensionality reduction (PCA) and manifold learning (t-SNE)

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run**

button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Let's install the required libraries.

```
!pip install numpy pandas matplotlib seaborn --quiet
```

```
!pip install jovian opendatasets graphviz scikit-learn plotly --upgrade --quiet
```

## ⌄ Introduction to Unsupervised Learning

Unsupervised machine learning refers to the category of machine learning techniques where models are trained on a dataset without labels. Unsupervised learning is generally use to discover patterns in data and reduce high-dimensional data to fewer dimensions. Here's how unsupervised learning fits into the landscape of machine learning algorithms([source](#)):
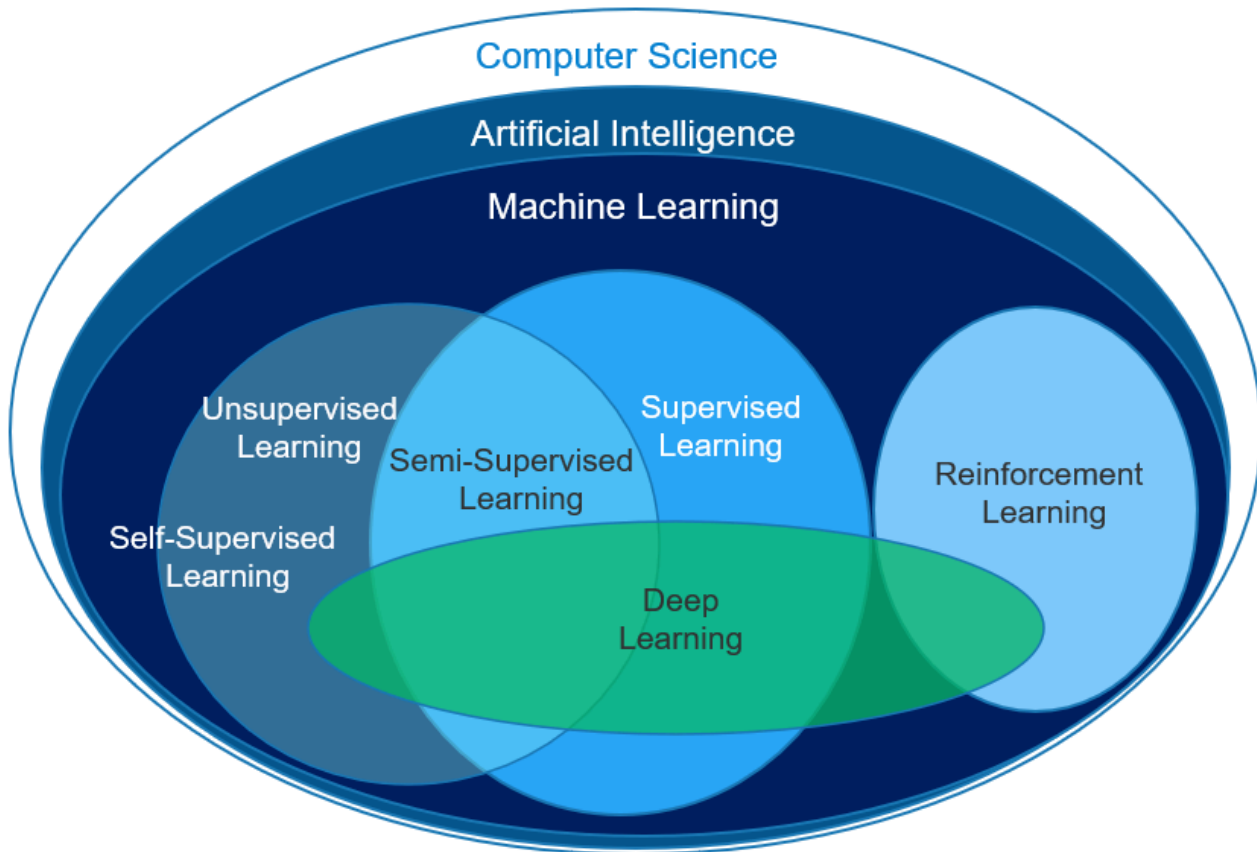
Here are the topics in machine learning that we're studying in this course ([source](#)):
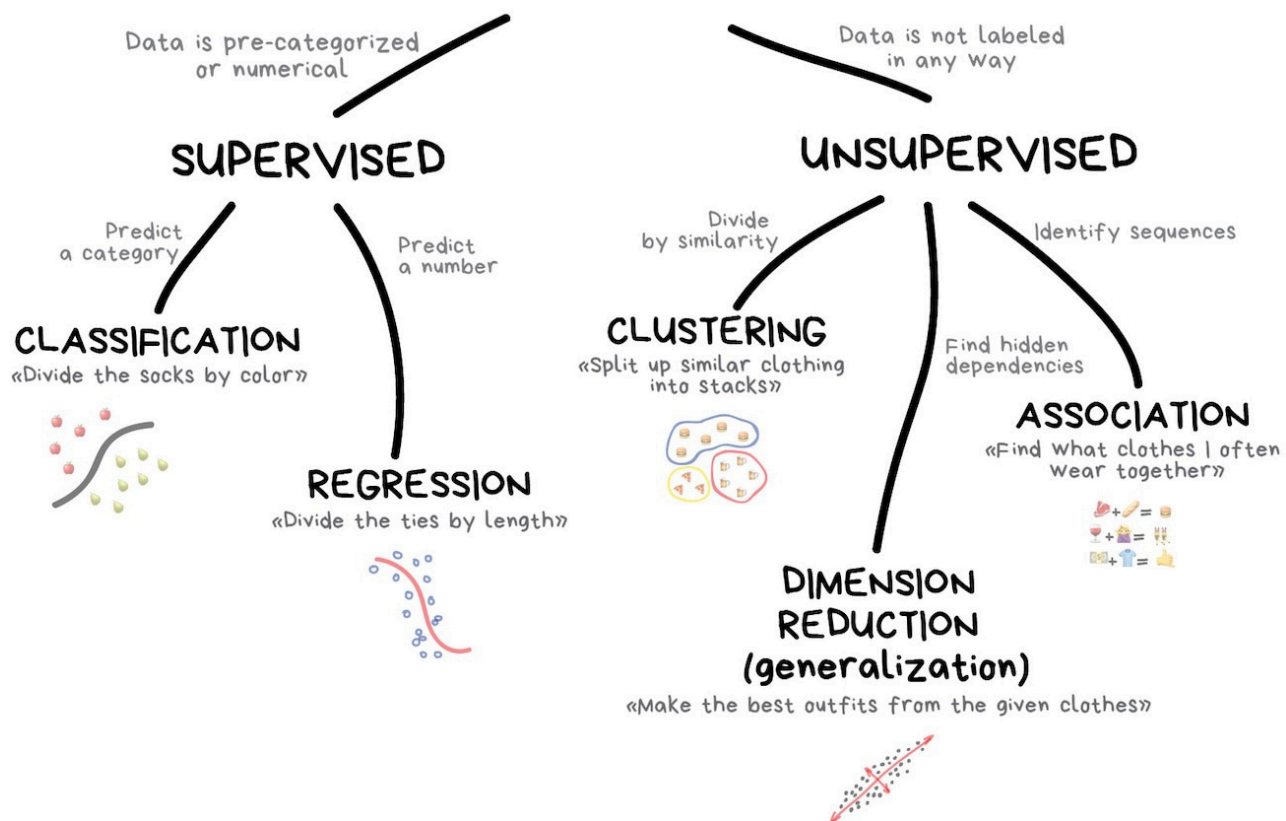
Scikit-learn offers the following cheatsheet to decide which model to pick for a given problem. Can you identify the unsupervised learning algorithms?

Here is a full list of unsupervised learning algorithms available in Scikit-learn: https://scikit-learn.org/stable/unsupervised_learning.html

⌄ Clustering

Clustering is the process of grouping objects from a dataset such that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (Wikipedia). Scikit-learn offers several clustering algorithms. You can learn more about them here: https://scikit-learn.org/stable/modules/clustering.html

Here is a visual representation of clustering:
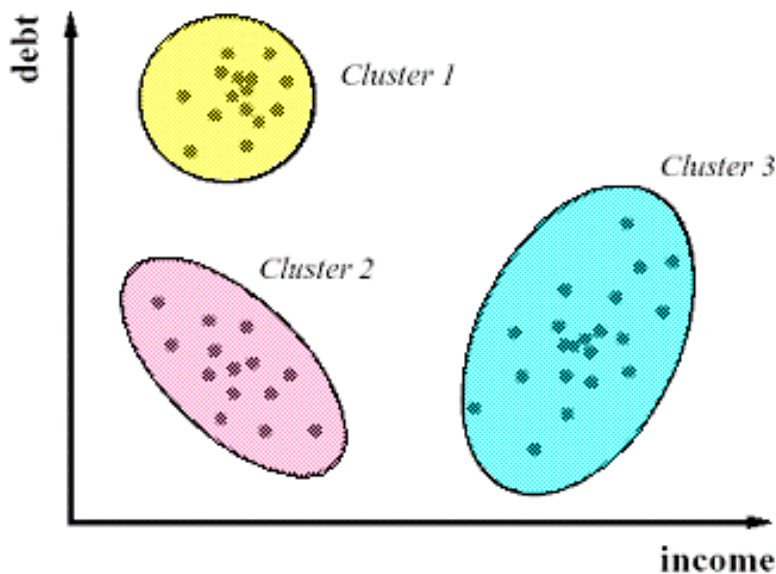
Here are some real-world applications of clustering:

- Customer segmentation
- Product recommendation
- Feature engineering
- Anomaly/fraud detection
- Taxonomy creation

We'll use the Iris flower dataset to study some of the clustering algorithms available in `scikit-learn`. It contains various measurements for 150 flowers belonging to 3 different species.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')
%matplotlib inline
```

```
iris_df = sns.load_dataset('iris')
```

```
iris_df
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

```
sns.scatterplot(data=iris_df, x='sepal_length', y='petal_length', hue='species');
```
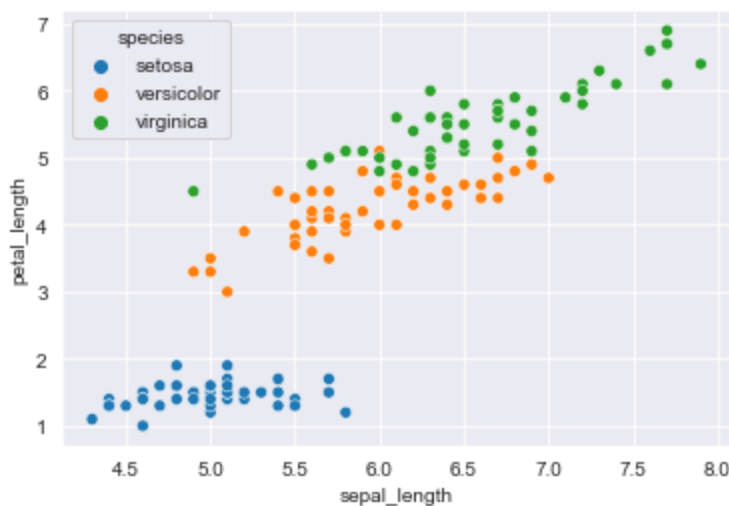
We'll attempt to cluster observations using numeric columns in the data.

```
numeric_cols = ["sepal_length", "sepal_width", "petal_length", "petal_width"]

X = iris_df[numeric_cols]
```

⌄ K Means Clustering

The K-means algorithm attempts to classify objects into a pre-determined number of clusters by finding optimal central points (called centroids) for each cluster. Each object is classifed as belonging the cluster represented by the closest centroid.
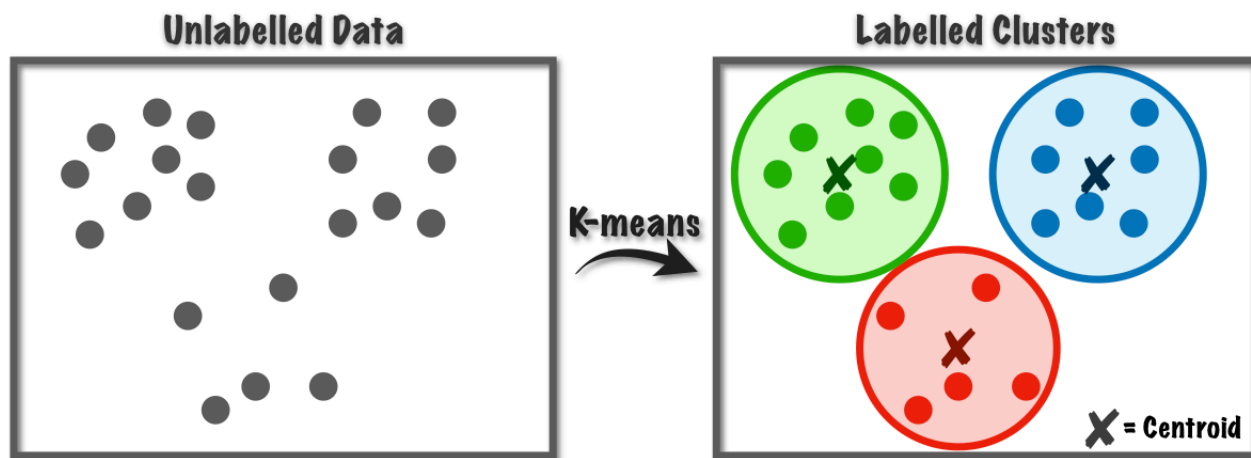
Here's how the K-means algorithm works:

1. Pick K random objects as the initial cluster centers.
2. Classify each object into the cluster whose center is closest to the point.
3. For each cluster of classified objects, compute the centroid (mean).
4. Now reclassify each object using the centroids as cluster centers.
5. Calculate the total variance of the clusters (this is the measure of goodness).
6. Repeat steps 1 to 6 a few more times and pick the cluster centers with the lowest total variance.

Here's a video showing the above steps: https://www.youtube.com/watch?v=4b5d3muPQmA

Let's apply K-means clustering to the Iris dataset.

```
from sklearn.cluster import KMeans
```

```
model = KMeans(n_clusters=3, random_state=42)
```

```
model.fit(X)
```

```
KMeans(n_clusters=3, random_state=42)
```

We can check the cluster centers for each cluster.

```
model.cluster_centers_
```

```
array([[5.9016129 , 2.7483871 , 4.39354839, 1.43387097],
       [5.006     , 3.428     , 1.462     , 0.246     ],
       [6.85      , 3.07368421, 5.74210526, 2.07105263]])
```

We can now classify points using the model.

```
X
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

150 rows × 4 columns

```
preds = model.predict(X)
preds
```

```
array([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, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0,
```
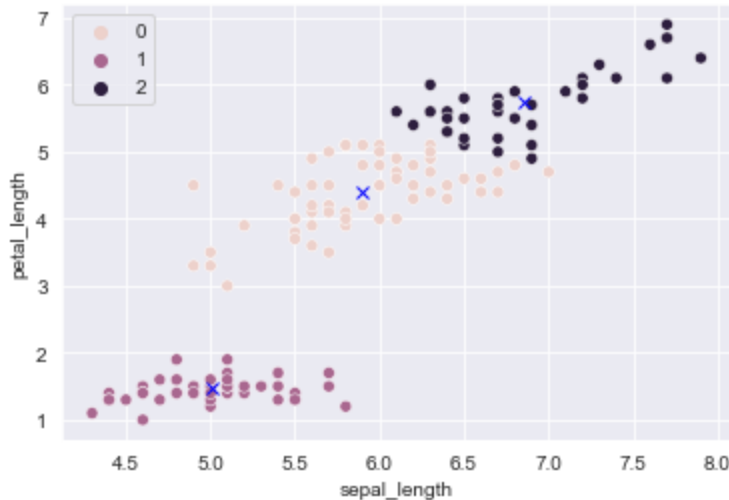
```
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 2, 2, 2, 2, 0, 2, 2, 2,
            2, 2, 2, 0, 0, 2, 2, 2, 2, 0, 2, 0, 2, 0, 2, 2, 0, 0, 2, 2, 2, 2,
            2, 0, 2, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 2, 0, 2, 2, 0], dtype=int32)
```

```python
sns.scatterplot(data=X, x='sepal_length', y='petal_length', hue=preds);
centers_x, centers_y = model.cluster_centers_[:,0], model.cluster_centers_[:,2]
plt.plot(centers_x, centers_y, 'xb')
```

⊋ [<matplotlib.lines.Line2D at 0x7ffc1d12e080>]

As you can see, K-means algorithm was able to classify (for the most part) different specifies of flowers into separate clusters. Note that we did not provide the "species" column as an input to `KMeans`.

We can check the "goodness" of the fit by looking at `model.inertia_`, which contains the sum of squared distances of samples to their closest cluster center. Lower the inertia, better the fit.

```python
model.inertia_
```

⊋ 78.851441426146

Let's try creating 6 clusters.

```python
model = KMeans(n_clusters=6, random_state=42).fit(X)
```
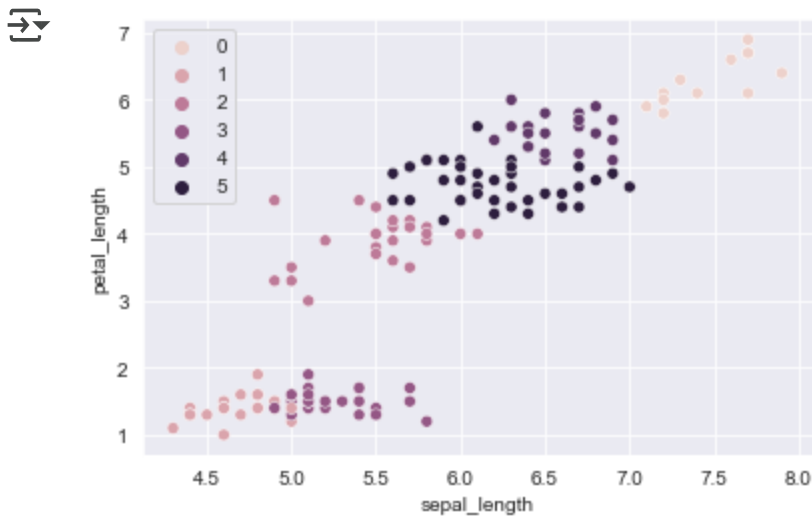
```python
preds = model.predict(X)
preds
```

⊋ array([3, 1, 1, 1, 3, 3, 1, 3, 1, 1, 3, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3,
            1, 3, 1, 1, 3, 3, 3, 1, 1, 3, 3, 3, 1, 1, 3, 3, 1, 3, 3, 1, 1, 3,
            3, 1, 3, 1, 3, 1, 5, 5, 5, 2, 5, 5, 5, 2, 5, 2, 2, 5, 2, 5, 2, 5,
            5, 2, 5, 2, 5, 2, 5, 5, 5, 5, 5, 5, 5, 2, 2, 2, 2, 5, 2, 5, 5, 5,

```
       2, 2, 2, 5, 2, 2, 2, 2, 2, 5, 2, 2, 4, 5, 0, 4, 4, 0, 2, 0, 4, 0,
       4, 4, 4, 5, 4, 4, 4, 0, 0, 5, 4, 5, 0, 5, 4, 0, 5, 5, 4, 0, 0, 0,
       4, 5, 5, 0, 4, 4, 5, 4, 4, 4, 5, 4, 4, 4, 5, 4, 4, 5], dtype=int32)
```

```python
sns.scatterplot(data=X, x='sepal_length', y='petal_length', hue=preds);
```


In most real-world scenarios, there's no predetermined number of clusters. In such a case, you can create a plot of "No. of clusters" vs "Inertia" to pick the right number of clusters.

```python
options = range(2,11)
inertias = []

for n_clusters in options:
    model = KMeans(n_clusters, random_state=42).fit(X)
    inertias.append(model.inertia_)

plt.title("No. of clusters vs. Inertia")
plt.plot(options, inertias, '-o')
plt.xlabel('No. of clusters (K)')
plt.ylabel('Inertia');
```
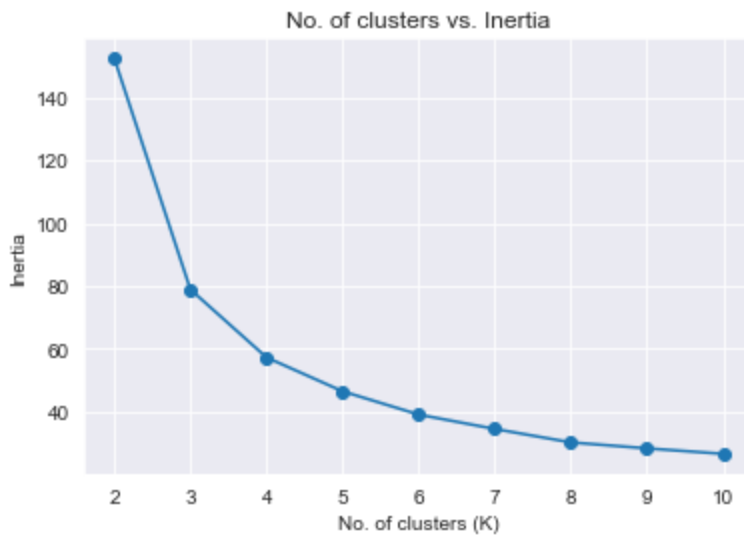
The chart is creates an "elbow" plot, and you can pick the number of clusters beyond which the reduction in inertia decreases sharply.

Mini Batch K Means: The K-means algorithm can be quite slow for really large dataset. Mini-batch K-means is an iterative alternative to K-means that works well for large datasets. Learn more about it here: https://scikit-learn.org/stable/modules/clustering.html#mini-batch-kmeans

> **EXERCISE**: Perform clustering on the Mall customers dataset on Kaggle. Study the segments carefully and report your observations.

Start coding or generate with AI.

Start coding or generate with AI.

⌄ DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) uses the density of points in a region to form clusters. It has two main parameters: "epsilon" and "min samples" using which it classifies each point as a core point, reachable point or noise point (outlier).
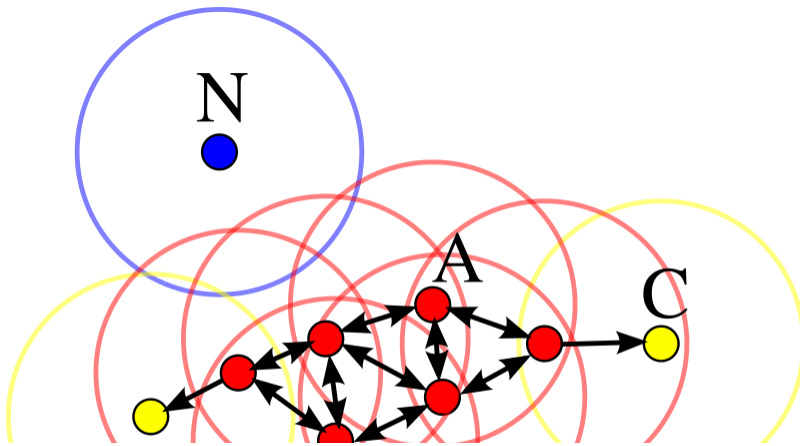
```
from sklearn.cluster import DBSCAN
```

?DBSCAN

Here's a video explaining how the DBSCAN algorithm works: https://www.youtube.com/watch?

```
model = DBSCAN(eps=1.1, min_samples=4)
```

```
model.fit(X)
```

```
DBSCAN(eps=1.1, min_samples=4)
```

In DBSCAN, there's no prediction step. It directly assigns labels to all the inputs.

```
model.labels_
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 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, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
sns.scatterplot(data=X, x='sepal_length', y='petal_length', hue=model.labels_);
```
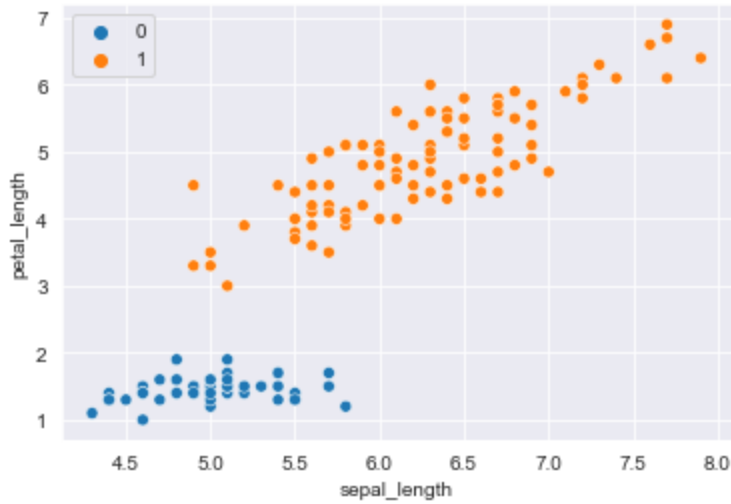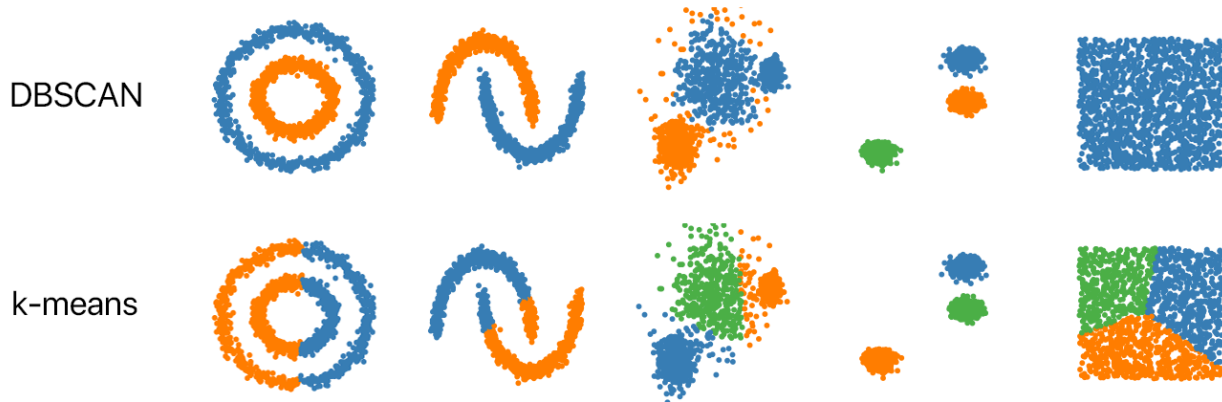
> **EXERCISE**: Try changing the values of `eps` and `min_samples` and observe how the number of clusters the classification changes.

Start coding or generate with AI.

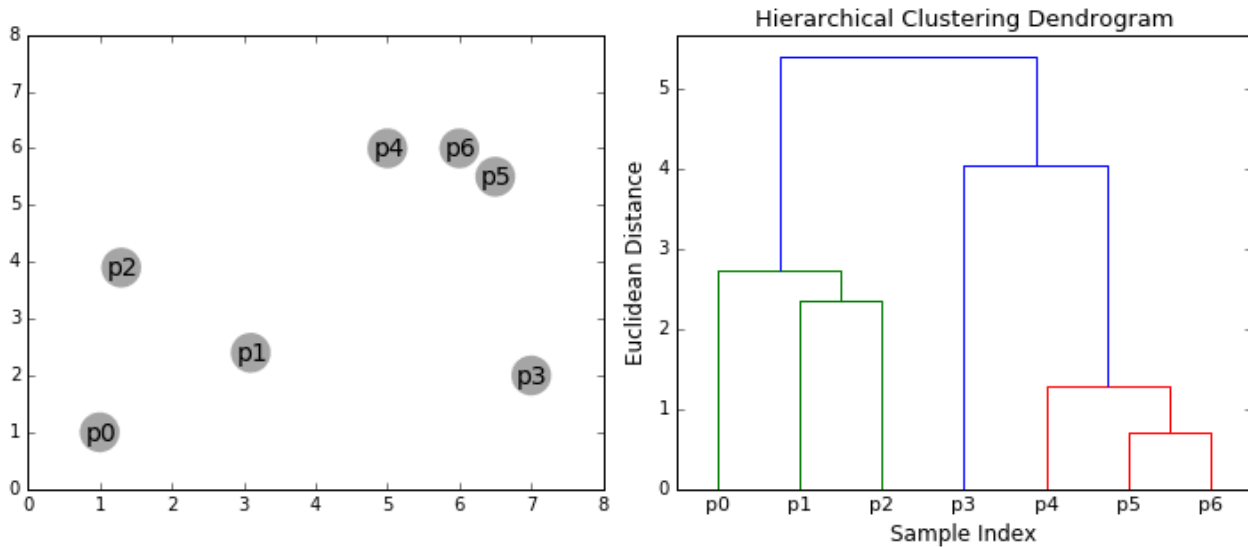Start coding or generate with AI.

Here's how the results of DBSCAN and K Means differ:

Hierarchical Clustering

Hierarchical clustering, as the name suggests, creates a hierarchy or a tree of clusters.

While there are several approaches to hierarchical clustering, the most common approach works as follows:

1. Mark each point in the dataset as a cluster.
2. Pick the two closest cluster centers without a parent and combine them into a new cluster.
3. The new cluster is the parent cluster of the two clusters, and its center is the mean of all the points in the cluster.
4. Repeat steps 2 and 3 till there's just one cluster left.

EXERCISE: Implement hierarchical clustering for the Iris dataset using `scikit-`
Watch this video for a visual explanation of hierarchical clustering:
`learn`.
https://www.youtube.com/watch?v=7xHsRkOdVwo

Start coding or generate with AI.

Start coding or generate with AI.

There are several other clustering algorithms in Scikit-learn. You can learn more about them and when to use them here: https://scikit-learn.org/stable/modules/clustering.html

Let's save our work before continuing.

```
jovian.commit()
```

⇥ [jovian] Committed successfully! https://jovian.ai/aakashns/sklearn-unsupervised
    'https://jovian.ai/aakashns/sklearn-unsupervised-learning'

⌄ Dimensionality Reduction and Manifold Learning

In machine learning problems, we often encounter datasets with a very large number of dimensions (features or columns). Dimensionality reduction techniques are used to reduce the number of dimensions or features within the data to a manageable or convenient number.

Applications of dimensionality reduction:

- Reducing size of data without loss of information
- Training machine learning models efficiently
- Visualizing high-dimensional data in 2/3 dimensions

⌄ Principal Component Analysis (PCA)

Principal component is a dimensionality reduction technique that uses linear projections of data to reduce their dimensions, while attempting to maximize the variance of data in the projection. Watch this video to learn how PCA works: https://www.youtube.com/watch?v=FgakZw6K1QQ

Here's an example of PCA to reduce 2D data to 1D:
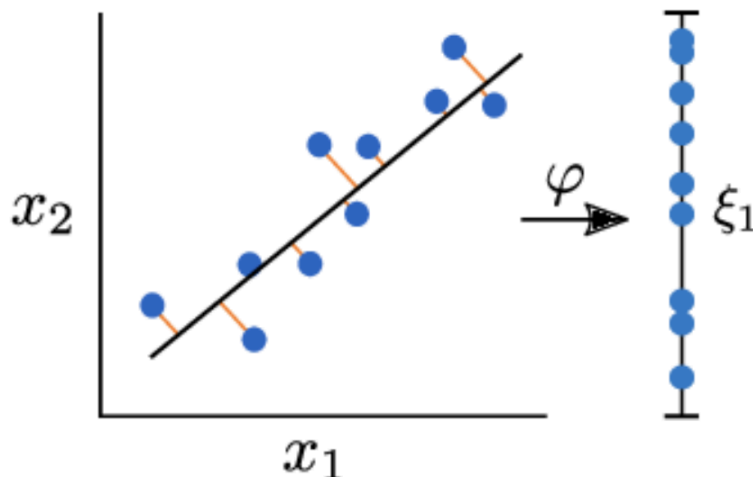
Here's an example of PCA to reduce 3D data to 2D:

Let's apply Principal Component Analysis to the Iris dataset.

```
iris_df = sns.load_dataset('iris')
iris_df
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

```
numeric_cols
```

```
['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
```

```
from sklearn.decomposition import PCA
```

```
?PCA
```

```
pca = PCA(n_components=2)
```

```
pca.fit(iris_df[numeric_cols])
```

```
PCA(n_components=2)
```

```
pca
```

```
PCA(n_components=2)
```

```
transformed = pca.transform(iris_df[numeric_cols])
```

```
sns.scatterplot(x=transformed[:,0], y=transformed[:,1], hue=iris_df['species']);
```

As you can see, the PCA algorithm has done a very good job of separating different species of flowers using just 2 measures.

> **EXERCISE**: Apply Principal Component Analysis to a large high-dimensional dataset and train a machine learning model using the low-dimensional results. Observe the changes in the loss and training time for different numbers of target dimensions.

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Learn more about Principal Component Analysis here:
https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html

⌄ t-Distributed Stochastic Neighbor Embedding (t-SNE)

Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high. Scikit-learn provides many algorithms for manifold learning: https://scikit-learn.org/stable/modules/manifold.html . A commonly-used manifold learning technique is t-Distributed Stochastic Neighbor Embedding or t-SNE, used to visualize high dimensional data in one, two or three dimensions.

Here's a visual representation of t-SNE applied to visualize 2 dimensional data in 1 dimension:
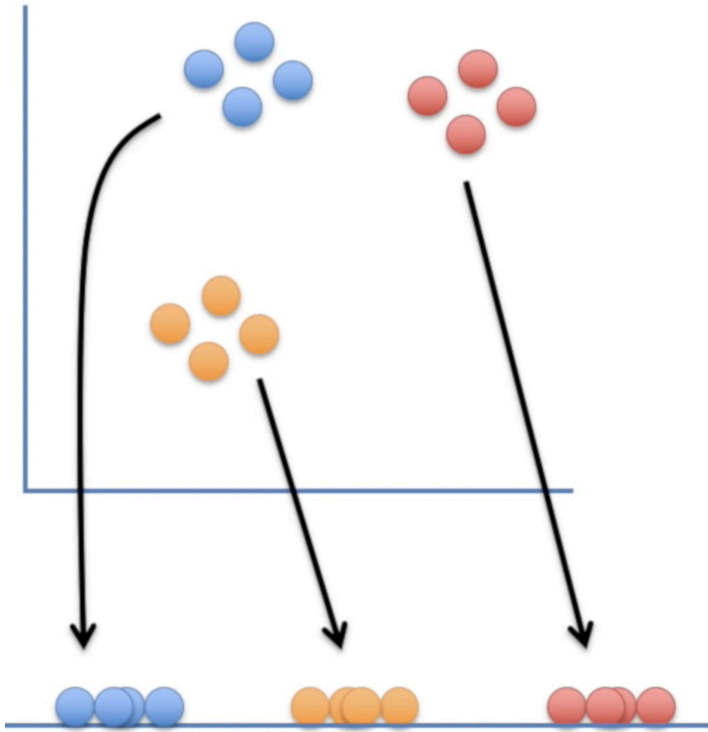
Here's a visual representation of t-SNE applied to the MNIST dataset, which contains 28px x 28px images of handrwritten digits 0 to 9, a reduction from 784 dimensions to 2 dimensions (source):
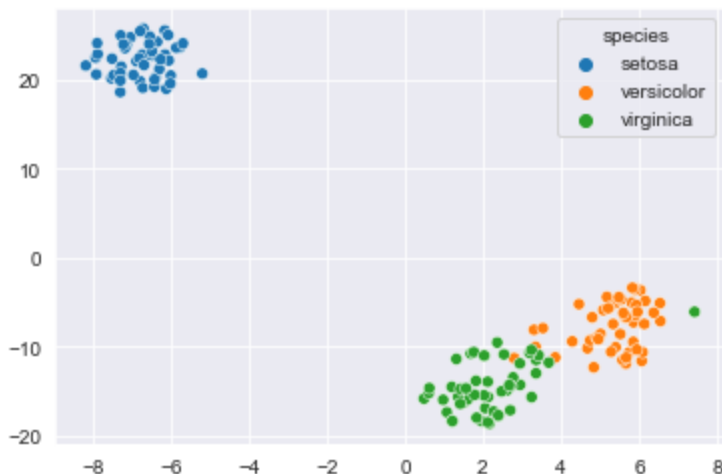
```
from sklearn.manifold import TSNE
```

Here's a video explaining how t-SNE works. https://www.youtube.com/watch?v=NEaUSP4Yerm

```
?TSNE


tsne = TSNE(n_components=2)


transformed = tsne.fit_transform(iris_df[numeric_cols])


sns.scatterplot(x=transformed[:,0], y=transformed[:,1], hue=iris_df['species']);
```

As you can see, the flowers from the same species are clustered very closely together. The relative distance between the species is also conveyed by the gaps between the clusters.

> **EXERCISE**: Use t-SNE to visualize the [MNIST handwritten digits dataset](https://colab.research.google.com/drive/1m9oK65jPhyqEYQOb9GqLDqkwhQqh-TFz#printMode=true).

Start coding or generate with AI.

Start coding or generate with AI.

Let's save our work before continuing.

```
jovian.commit()
```

⇥

⌄ Summary and References

CLASSICAL MACHINE LEARNING