ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

Last lecture reminder

We learned about:

- Hierarchical Clustering - Python Example
- Hierarchical Clustering - Visualization
- DBSCAN - Introduction
- DBSCAN - Key Concepts
- DBSCAN - Theory
- DBSCAN - Visualization

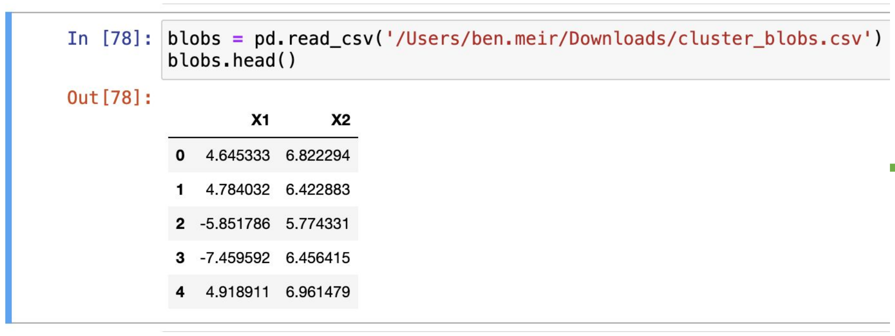
DBSCAN - Python Example

For the example we will use the following csv files:

- **cluster_blobs.csv**
- **cluster_moons.csv**
- **cluster_circles.csv**

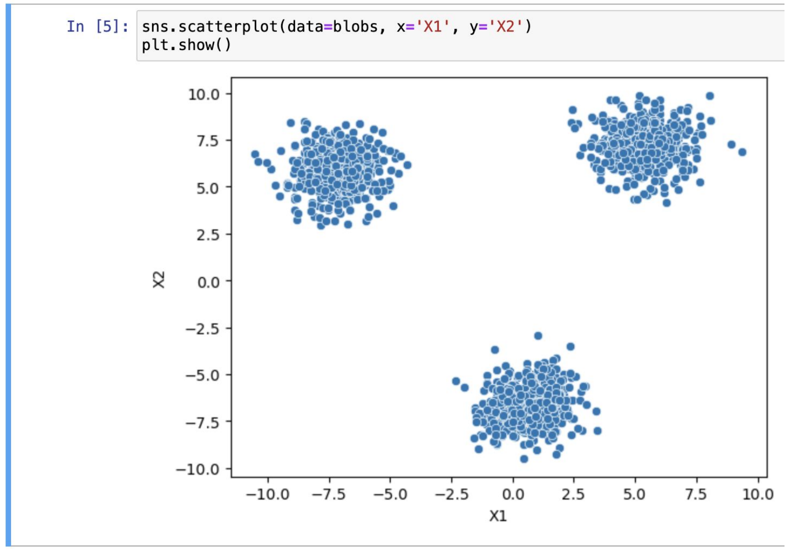
All the following csv files contain data points with 2 features (X1 & X2) that construct specific clusters shapes.

For the **'cluster_blobs.csv'** file:



```
In [78]: blobs = pd.read_csv('/Users/ben.meir/Downloads/cluster_blobs.csv')
         blobs.head()
Out[78]:
```

	X1	X2
0	4.645333	6.822294
1	4.784032	6.422883
2	-5.851786	5.774331
3	-7.459592	6.456415
4	4.918911	6.961479

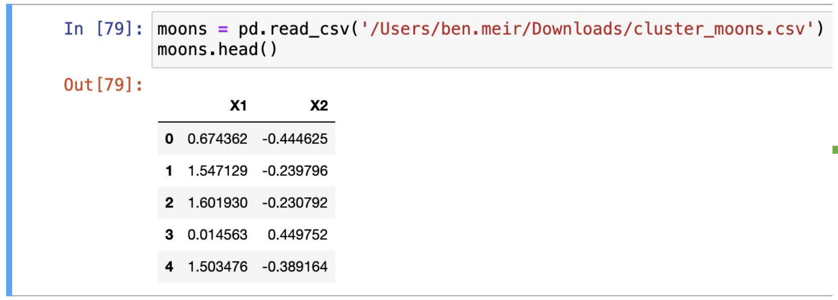


```
In [5]: sns.scatterplot(data=blobs, x='X1', y='X2')
        plt.show()
```

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

DBSCAN - Python Example

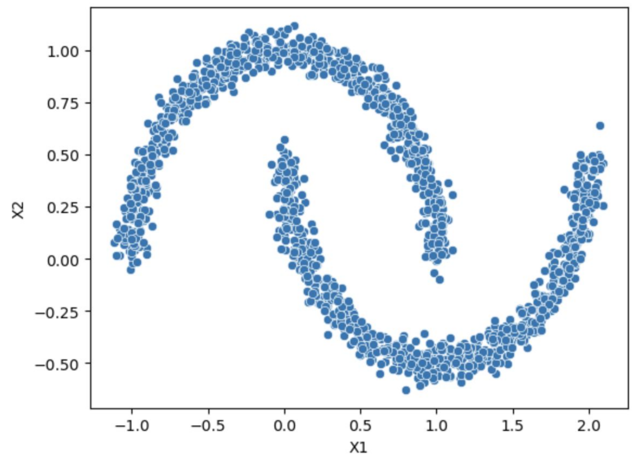
For the ‘**cluster_moons.csv**’ file:



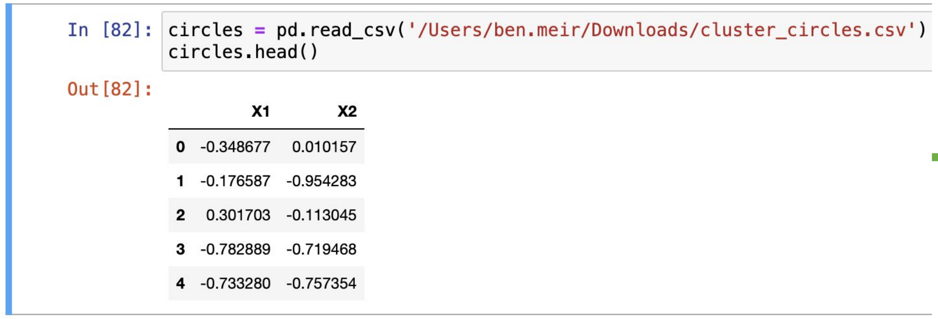
```
In [79]: moons = pd.read_csv('/Users/ben.meir/Downloads/cluster_moons.csv')
         moons.head()
Out[79]:
```

	X1	X2
0	0.674362	-0.444625
1	1.547129	-0.239796
2	1.601930	-0.230792
3	0.014563	0.449752
4	1.503476	-0.389164

```
In [81]: sns.scatterplot(data=moons, x='X1', y='X2')
         plt.show()
```



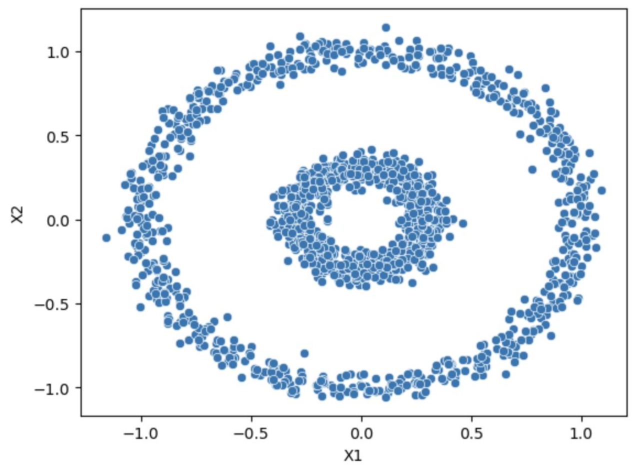
For the ‘**cluster_circles.csv**’ file:



```
In [82]: circles = pd.read_csv('/Users/ben.meir/Downloads/cluster_circles.csv')
         circles.head()
Out[82]:
```

	X1	X2
0	-0.348677	0.010157
1	-0.176587	-0.954283
2	0.301703	-0.113045
3	-0.782889	-0.719468
4	-0.733280	-0.757354

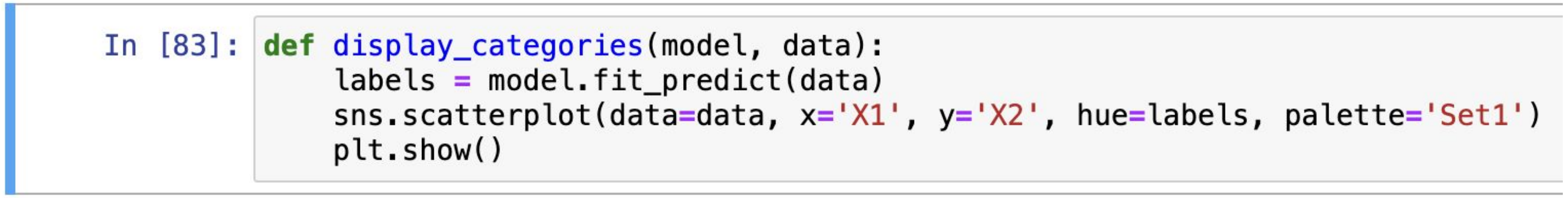
```
In [9]: sns.scatterplot(data=circles, x='X1', y='X2')
        plt.show()
```



ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

DBSCAN - Python Example

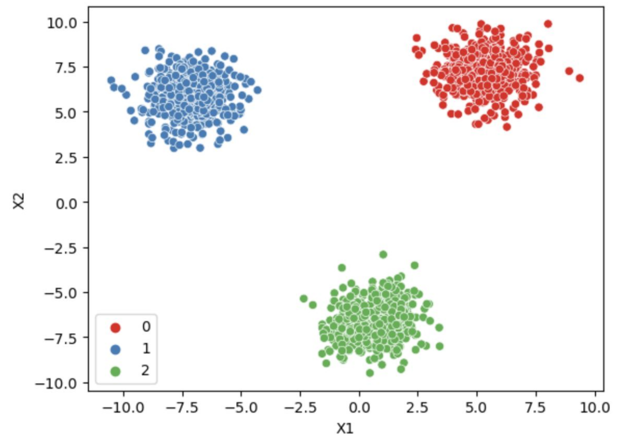
First, we want to write a dedicated function called **‘display_categories’** that will get the dataset and the model we are using and generate the model result clustering prediction as scatterplot.



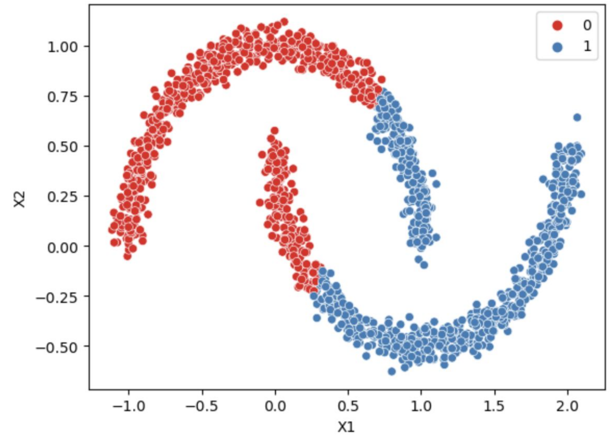
```
In [83]: def display_categories(model, data):
             labels = model.fit_predict(data)
             sns.scatterplot(data=data, x='X1', y='X2', hue=labels, palette='Set1')
             plt.show()
```

Now, let’s examine how the **K-Mean clustering model** (distance based model) perform on each dataset:

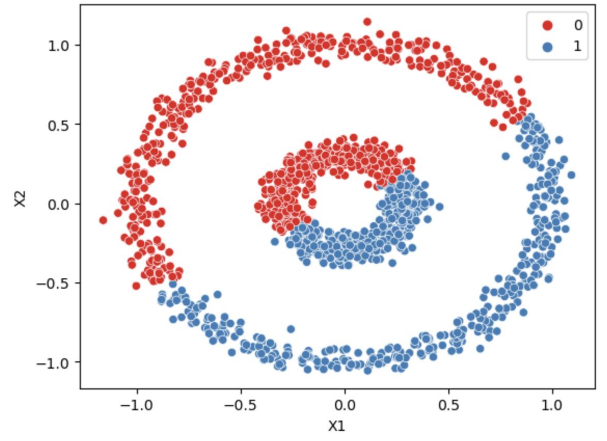
```
In [84]: from sklearn.cluster import KMeans
         model = KMeans(n_clusters=3)
         display_categories(model, blobs)
```



```
In [85]: from sklearn.cluster import KMeans
         model = KMeans(n_clusters=2)
         display_categories(model, moons)
```



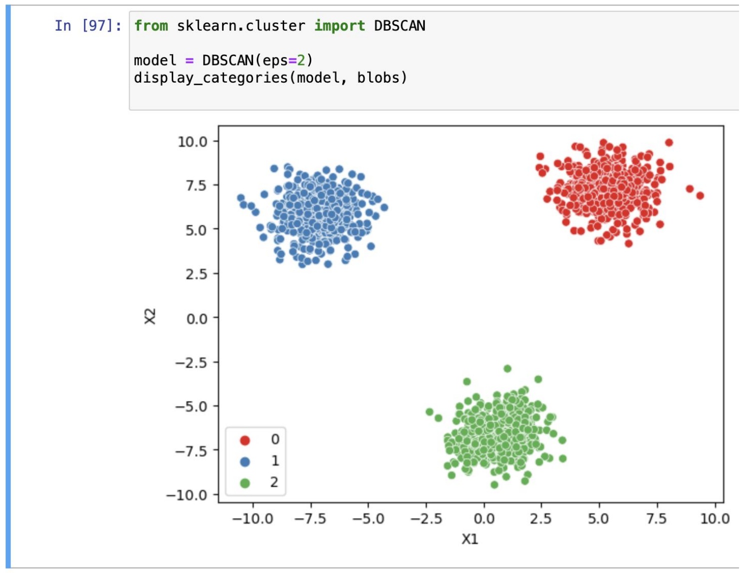
```
In [86]: from sklearn.cluster import KMeans
         model = KMeans(n_clusters=2)
         display_categories(model, circles)
```



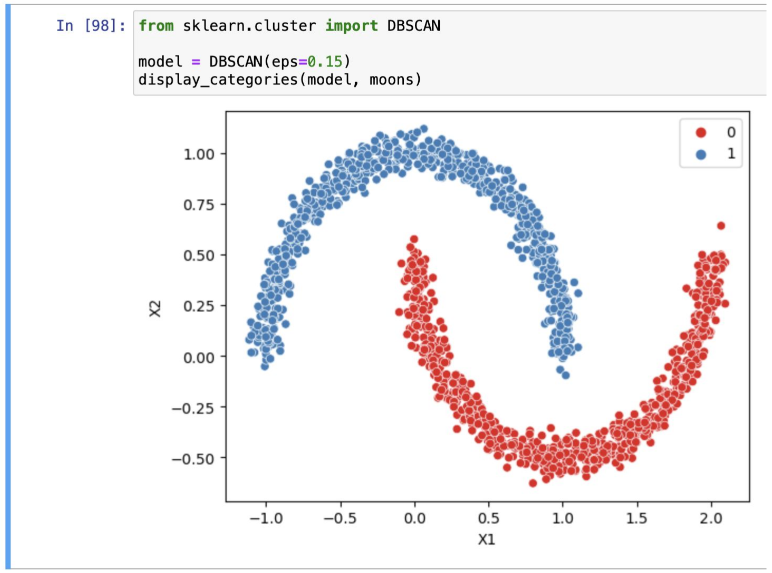
DBSCAN - Python Example

We can clearly see that the distance based algorithm didn’t managed to successfully identify the clusters in the ‘**cluster_moons’** and the **‘cluster_circles’.** This is because relay only on the distance between the data points to the nearest cluster centroid is not enough in this case.

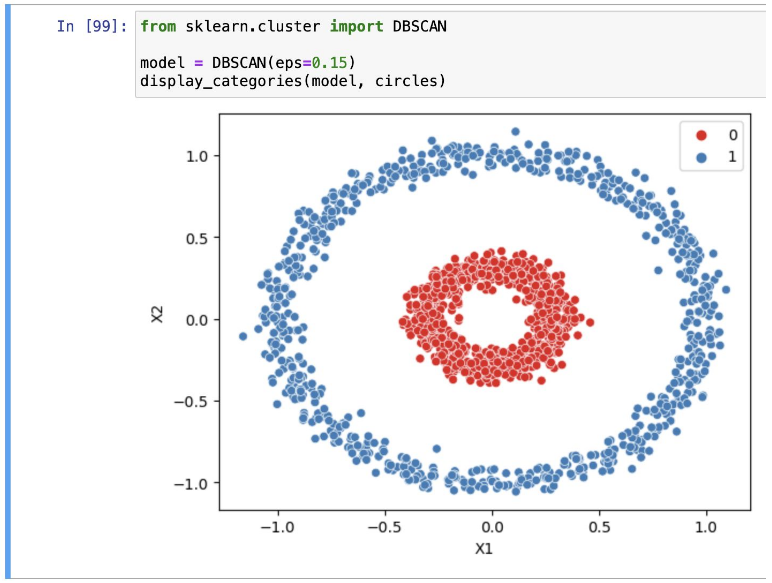
Now, let’s examine how the **DBSCAN model** perform on each of those datasets:



```
In [97]: from sklearn.cluster import DBSCAN

         model = DBSCAN(eps=2)
         display_categories(model, blobs)
```



```
In [98]: from sklearn.cluster import DBSCAN

         model = DBSCAN(eps=0.15)
         display_categories(model, moons)
```



```
In [99]: from sklearn.cluster import DBSCAN

         model = DBSCAN(eps=0.15)
         display_categories(model, circles)
```

DBSCAN - Model Parameters

As we learned in the previous lecture, the way DBSCAN algorithm works is by iterating on each data point and classify it to a specific cluster according to the density-based criteria defined by the parameters **epsilon (ε)** and the **minimum number of points (MinPts)**.

Let's understand how each of those parameters is affecting the clustering result of the model.

Epsilon (ε) → Epsilon defines the radius of the neighborhood around a data point. It directly influences the density considerations of the algorithm:

- **Small epsilon (ε)** → The smallest the epsilon value, the smaller the neighborhoods around each data point. The result will be that Many points may not meet the density requirement (MinPts) to be considered core points and thus may be classified as noise.
- **Large Epsilon (ε)** → The larger the epsilon value, the smaller the neighborhoods around each data point. More points are included within each neighborhood, increasing the likelihood of more points being classified as core points in the cluster.

DBSCAN - Model Parameters

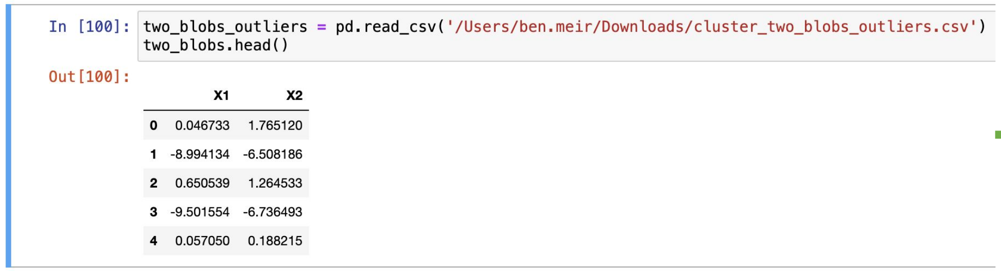
Minimum Number of Points (MinPts) → MinPts determines the minimum number of points required within the ε-radius neighborhood for a point to be classified as a core point.

- **Small MinPts** → The smallest the MinPts value, the lower the threshold for forming dense regions. The result will be that meeting the core point criteria will be easier, resulting in more points being classified as core points. This will reduce the number of outliers and increase the overall number of different clusters.
- **Large MinPts** → The larger the MinPts value, the higher the threshold for forming dense regions. The result will be that meeting the core point criteria will be harder, resulting in less points being classified as core points. This will reduce the number of overall clusters and will create only significant clusters.

Note: The model will consider a data point as an outlier if this points can't be consider as a core point nor a border point. Meaning this point is not a core point and also is not in the ε-radius of any core point.

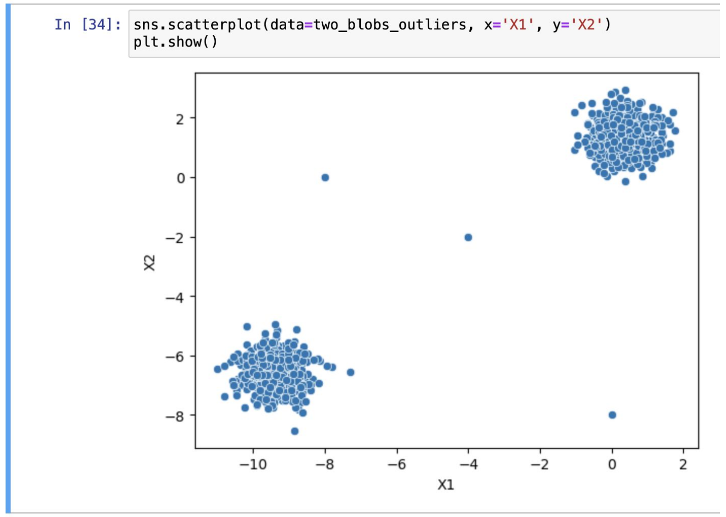
DBSCAN - Model Parameters

In order to show the effect of different Epsilon and MinPts we will use the **'cluster_two_blobs_outliers.csv'** file.

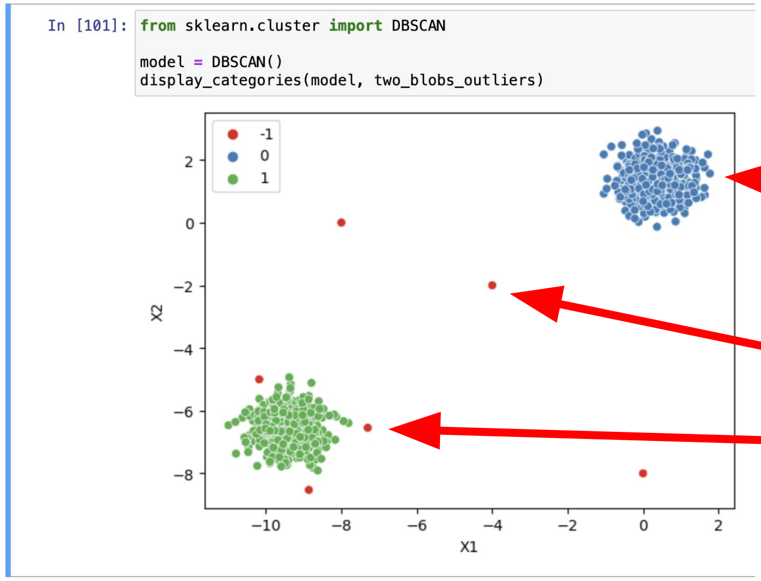


```
In [100]: two_blobs_outliers = pd.read_csv('/Users/ben.meir/Downloads/cluster_two_blobs_outliers.csv')
          two_blobs.head()
Out[100]:
```

	X1	X2
0	0.046733	1.765120
1	-8.994134	-6.508186
2	0.650539	1.264533
3	-9.501554	-6.736493
4	0.057050	0.188215



```
In [34]: sns.scatterplot(data=two_blobs_outliers, x='X1', y='X2')
         plt.show()
```



```
In [101]: from sklearn.cluster import DBSCAN

          model = DBSCAN()
          display_categories(model, two_blobs_outliers)
```

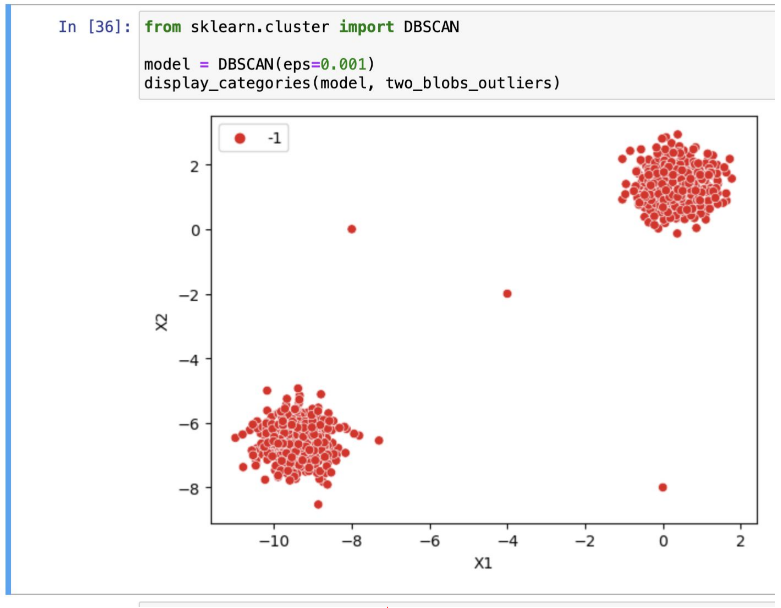
The default DBSCAN parameters (ε = 0.5 & MinPts = 5) managed to recognized the 2 main clusters

The default model also classified some of the points as -1, meaning outliers

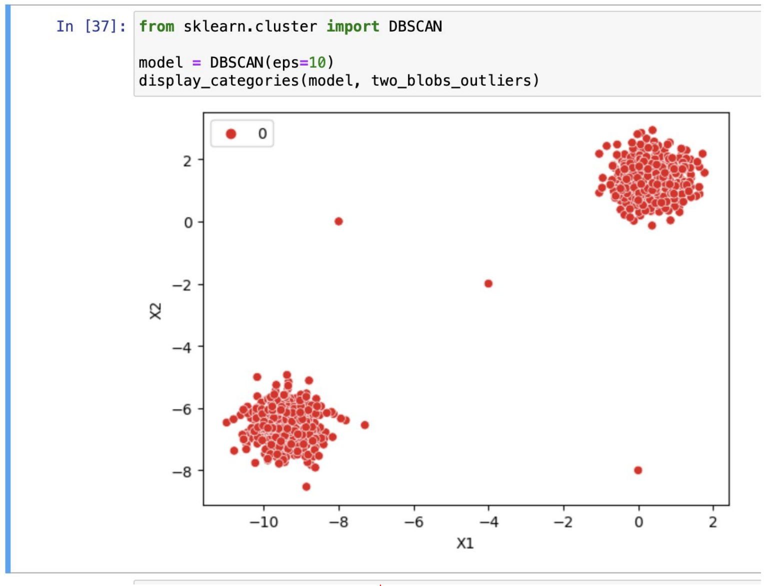
DBSCAN - Model Parameters

As explained, changing the Epsilon and MinPts parameters will change the classification of the data points. It correspondly can result more clusters or more outliers.

For example →



```
In [36]: from sklearn.cluster import DBSCAN

         model = DBSCAN(eps=0.001)
         display_categories(model, two_blobs_outliers)
```



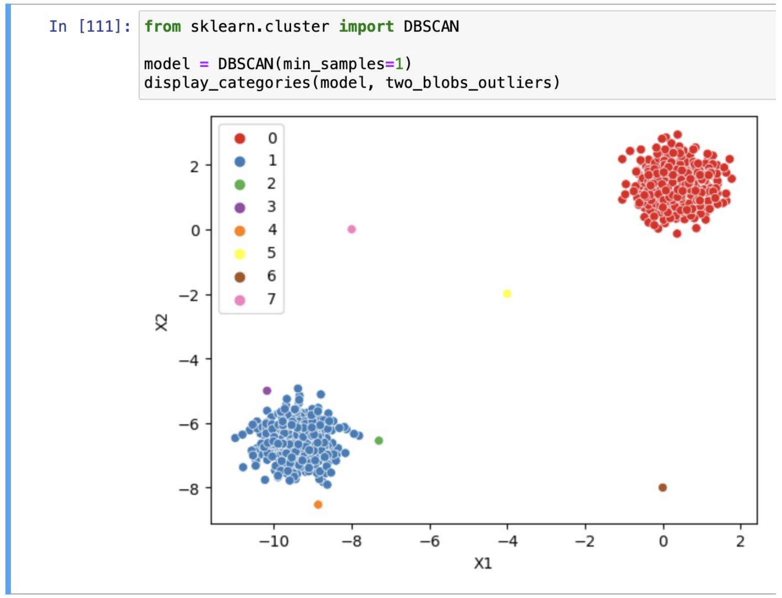
```
In [37]: from sklearn.cluster import DBSCAN

         model = DBSCAN(eps=10)
         display_categories(model, two_blobs_outliers)
```

Epsilon very small will cause the model to classify all data points as outliers

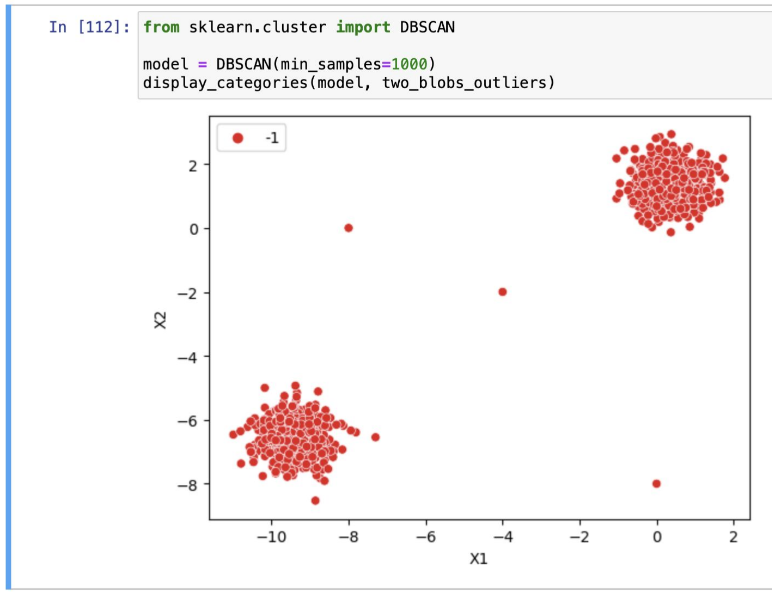
Epsilon very large will cause the model to classify all data points as one cluster

DBSCAN - Model Parameters

For example →



```
In [111]: from sklearn.cluster import DBSCAN

          model = DBSCAN(min_samples=1)
          display_categories(model, two_blobs_outliers)
```



```
In [112]: from sklearn.cluster import DBSCAN

          model = DBSCAN(min_samples=1000)
          display_categories(model, two_blobs_outliers)
```

MinPts very small will create multiple different clusters and reduce the number of outliers

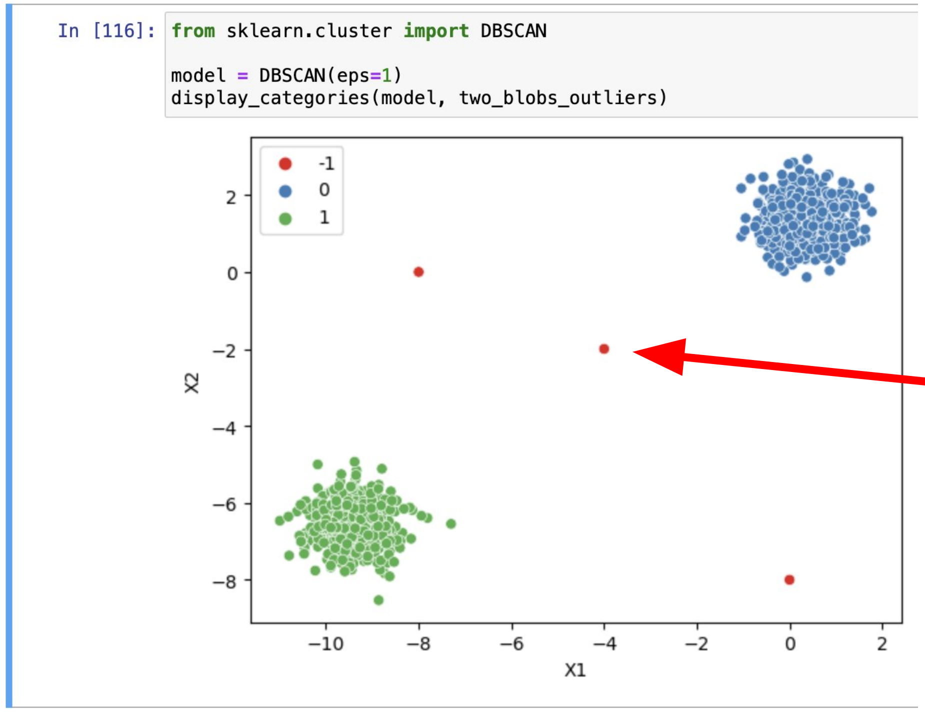
MinPts very large will cause the model to classify all data points as outliers

ECOM SCHOOL
המכללה למקצועות הדיגיטל וההייטק

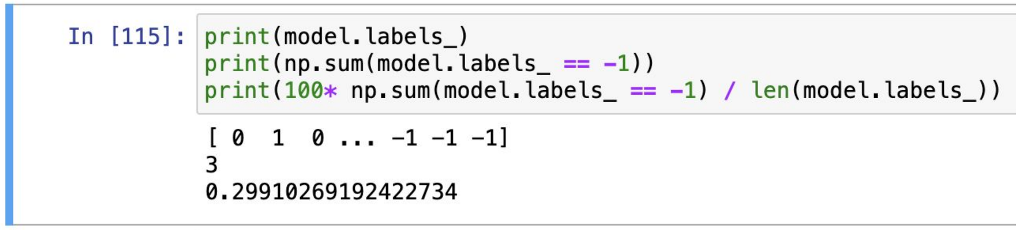
DBSCAN - Optimal Parameters

We can use **elbow method** in order to find the optimal parameters values. This will also required to know in advended the number of clusters that we want to recognize or the number / percent of outliers we allow.

We will use the following calculations to find the number / percentage of outlier data points:



```
In [116]: from sklearn.cluster import DBSCAN

          model = DBSCAN(eps=1)
          display_categories(model, two_blobs_outliers)
```



```
In [115]: print(model.labels_)
          print(np.sum(model.labels_ == -1))
          print(100* np.sum(model.labels_ == -1) / len(model.labels_))

          [ 0  1  0 ... -1 -1 -1]
          3
          0.29910269192422734
```

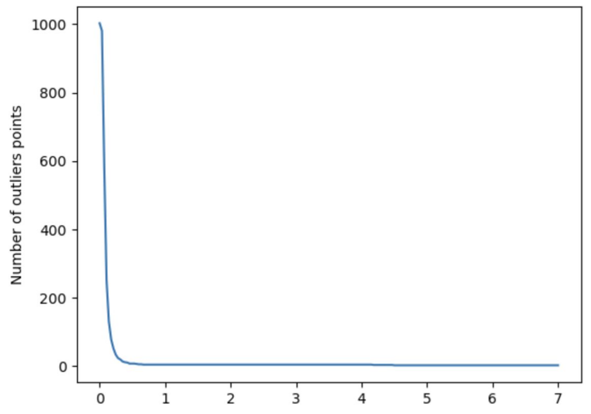
In this model, the number of data points classified as outliers is 3 and the percentage of the total outlier data points is 0.3% from the entire data set

DBSCAN - Optimal Parameters

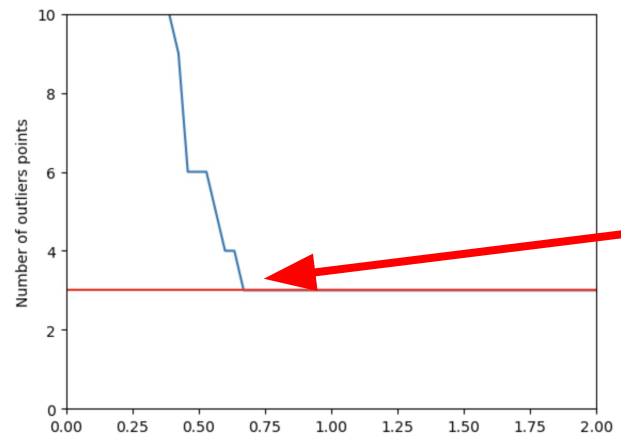
Now, let’s perform the same calculation to find the number / percent of outlier for multiple epsilon values:

```
In [117]: outlier_precent = []
          number_of_outliers = []

          for eps in np.linspace(0.001, 7, 200):
              dbscan = DBSCAN(eps=eps)
              dbscan.fit(two_blobs_outliers)

              ## Total of outliers found:
              number_of_outliers.append(np.sum(dbscan.labels_ == -1))

              ## Precent of points classified as outliers
              outlier_precent.append(100* np.sum(dbscan.labels_ == -1) / len(model.labels_))
```

```
In [118]: sns.lineplot(x=np.linspace(0.001, 7, 200), y=number_of_outliers)
          plt.ylabel('Number of outliers points')
          plt.show()
```



```
In [119]: sns.lineplot(x=np.linspace(0.001, 7, 200), y=number_of_outliers)
          plt.ylabel('Number of outliers points')
          plt.xlim(0,2)
          plt.ylim(0,10)
          plt.hlines(y=3, xmin=0, xmax=2, color='red')
          plt.show()
```

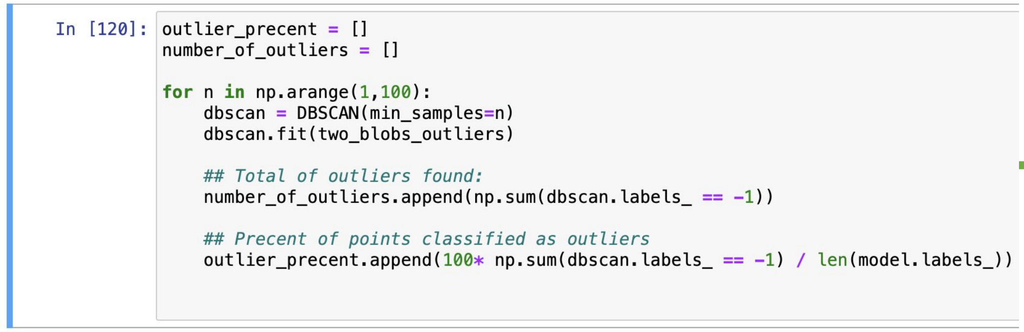


We can also perform some ‘zoom-in’ in order to find the actual epsilon value that we are looking for

In this example, we know we want the algorithm to find only 3 outliers so we can find what is the first epsilon value that find 3 outliers

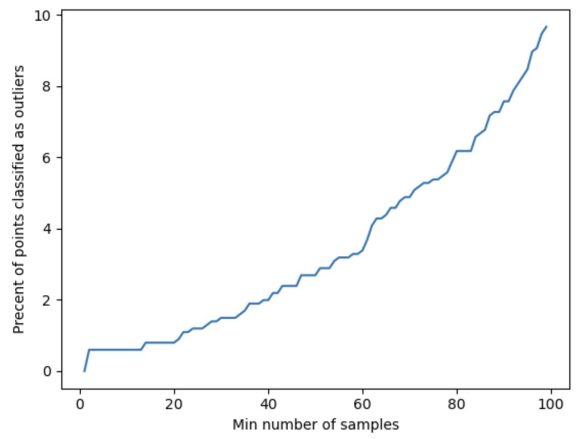
DBSCAN - Optimal Parameters

Same logic can be done for the MinPts parameter:



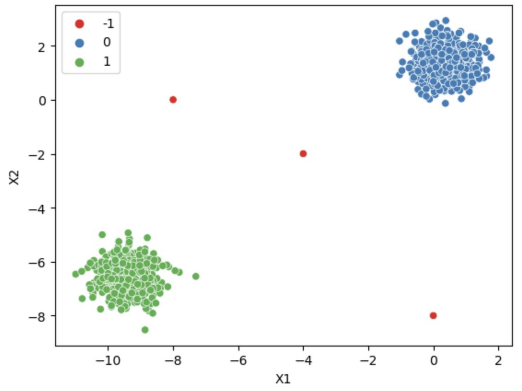
```
In [120]: outlier_precent = []
          number_of_outliers = []

          for n in np.arange(1,100):
              dbscan = DBSCAN(min_samples=n)
              dbscan.fit(two_blobs_outliers)

              ## Total of outliers found:
              number_of_outliers.append(np.sum(dbscan.labels_ == -1))

              ## Precent of points classified as outliers
              outlier_precent.append(100* np.sum(dbscan.labels_ == -1) / len(model.labels_))
```

```
In [122]: sns.lineplot(x=np.arange(1,100), y=outlier_precent)
          plt.xlabel('Min number of samples')
          plt.ylabel('Precent of points classified as outliers')
          plt.show()
```



Regarding the MinPts starting value, it is most likely to select it as → **MinPts = 2 X number of features**

```
In [123]: from sklearn.cluster import DBSCAN

          num_dim = two_blobs_outliers.shape[1]

          model = DBSCAN(eps=0.75, min_samples=2*num_dim)
          display_categories(model, two_blobs_outliers)
```



In our example, selecting the MinPts as 2 X number of features and the Epsilon value as the optimal value we previously found will result the requested outcome (2 clusters and 3 outliers)

Class Exercise - DBSCAN

Instructions:

Use the **'tree_blobs_with_outlier.csv'** file and implement the following:

- Generate a simple scatter plot to see the original dataset pattern.
- Perform default DBSCAN model and plot the classification results.
- Apply elbow method on different Epsilon and MinPts values and find the optimal values for each parameter.
- Perform DBSCAN model with the optimal values you found and plot the classification results.
- Perform DBSCAN model that will generate 0 outlier points and multiple different clusters, plot the model classification results.

Class Exercise Solution - DBSCAN

PCA - Introduction

PCA → PCA stands for Principal Component Analysis. It is a fundamental technique in data analysis, machine learning, and statistics used primarily for dimensionality reduction.
PCA transforms the original features into a set of orthogonal (uncorrelated) components, known as principal components, which represent the data in a way that captures the most variance. This allows the reduction of the number of variables while retaining the most significant information, making PCA especially useful for visualizing high-dimensional data.

Why should we use PCA?
The main challenge with high-dimensional data is that it can lead to overfitting, increased computational complexity, and difficulties in visualization. By reducing the number of dimensions, PCA helps mitigate these issues while maintaining the essence of the data. Additionally, PCA often enhances the performance of various machine learning algorithms by removing noise and reducing redundancy among the features.

PCA - Introduction

How PCA is related to Unsupervised Learning?

In Unsupervised Learning we don’t have label data so we can’t decide which feature is more important than other (unlike in Supervised Learning models which the algorithm itself could reduce the amount of features according to their importance to the label prediction).

PCA help us reduce the amount of features without remove their data completely.

The primary objective of PCA is to project the data into a lower-dimensional space while preserving as much variance (information) as possible.

Note: PCA does not generate a subset of the original features. rather, it creates new values that project the original data onto a new coordinate system defined by the principal components.

PCA - Terminology

Principal Components → Principal components are the new axes or directions that PCA identifies. They are linear combinations of the original variables and are derived in such a way that each succeeding component accounts for the maximum possible variance under the constraint that it is orthogonal to (uncorrelated with) the preceding components.

Explained Variance → Explained Variance is a metric that indicates how much of the total variance (informace) in the dataset is captured by each principal component when performing Principal Component Analysis (PCA). It plays a crucial role in helping to understand the importance of each principal component and in determining how many components to retain for adequate representation of the data.

PCA - Terminology

Eigen values → Eigen values represent the amount of variance captured by each principal component. They help in determining the importance of each component. Larger eigen values indicate components that capture more variance from the data.

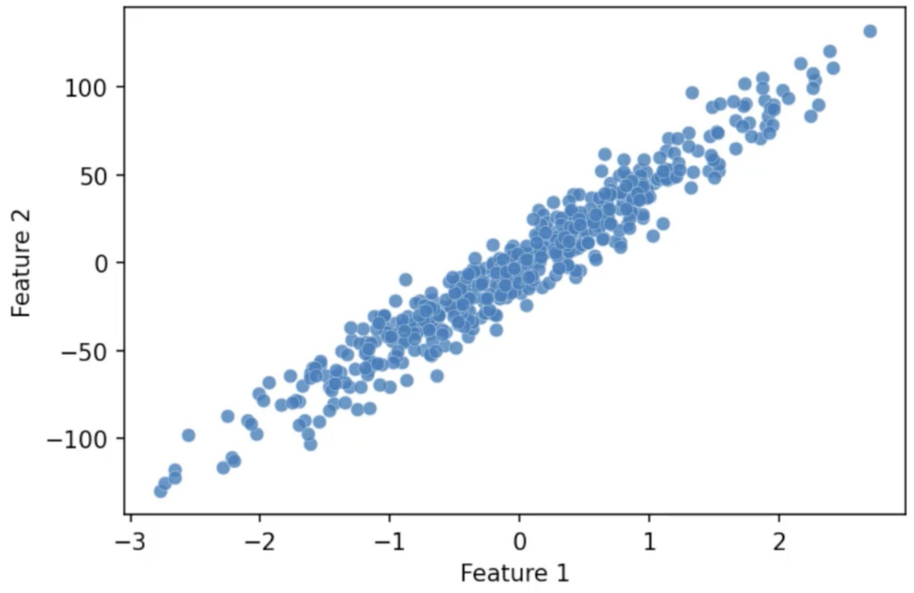
Eigen vectors → Eigen vectors are the directions or weights of the new principal components. They provide the coefficients for combining the original variables to obtain the principal components. These vectors are orthogonal to each other.

Covariance Matrix → The covariance matrix is a square matrix giving the covariance (שונות משותפת) between each pair of variables in the data. PCA uses this matrix to understand how the variables jointly vary and to determine the principal components.

PCA - Theory

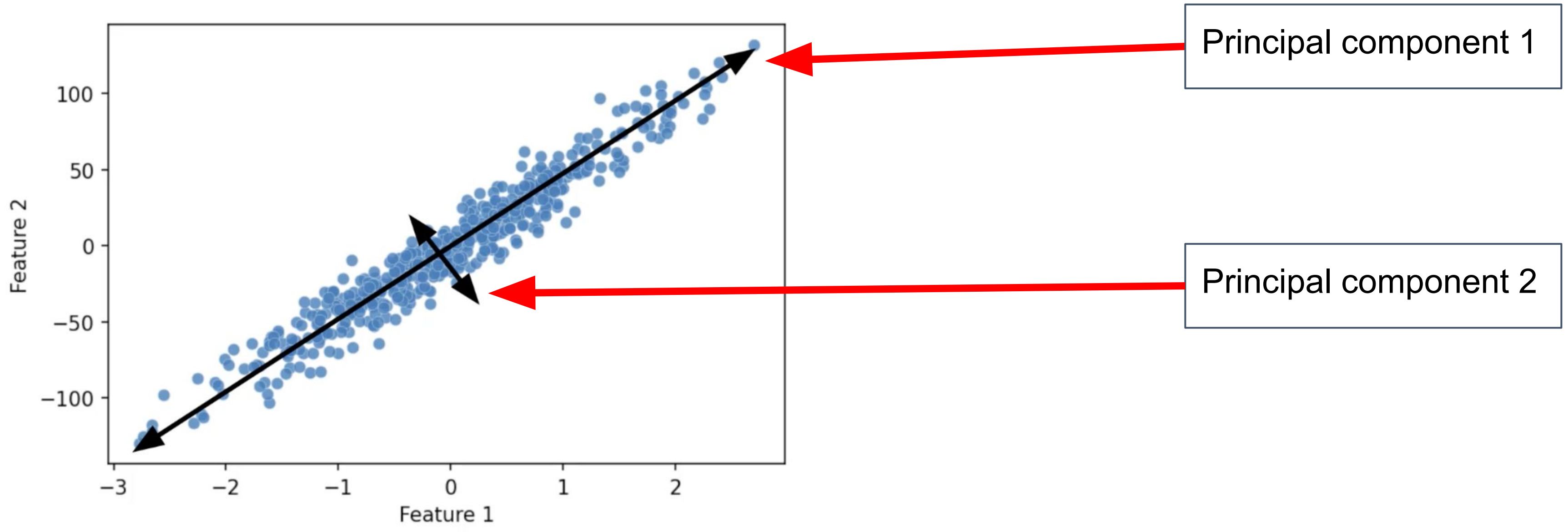
Let’s now explain how exactly PCA works and how it managed to reduce the amount of features without actually dropping entire features data from the data set.

Let’s take a look at the following 2 features data set. We want to reduce the amount of features from 2 to only 1 feature using PCA. We can see that there is a linear dependency between the two features.



PCA - Theory

Because the features has linear dependency, we can describe very similar data by creating a new axis that go across the features data points.



Each optional principal component is built as linear combination of Feature 1 and Feature 2 data.

We now need to decide which principal component we want to use (because we want only 1 feature).

The chosen principal component should be the one that has the higher explained variance.

In our example it will be principal component 1.

PCA - Theory

Now that we chose the optimal principal component, we can plot its data points along a single axis.

Principal Component 1

$$Z_1 = \phi_{11} X_1 + \phi_{21} X_2$$

Principal component 1 is a linear combination of Feature 1 and Feature 2

The final result is a single feature data points that can be plot as a single line.

How can we create the principal components?

We can create the principal components by applying **principal components analysis** which basically create new set of dimensions (principal components) that are normalized linear combinations of the original features.

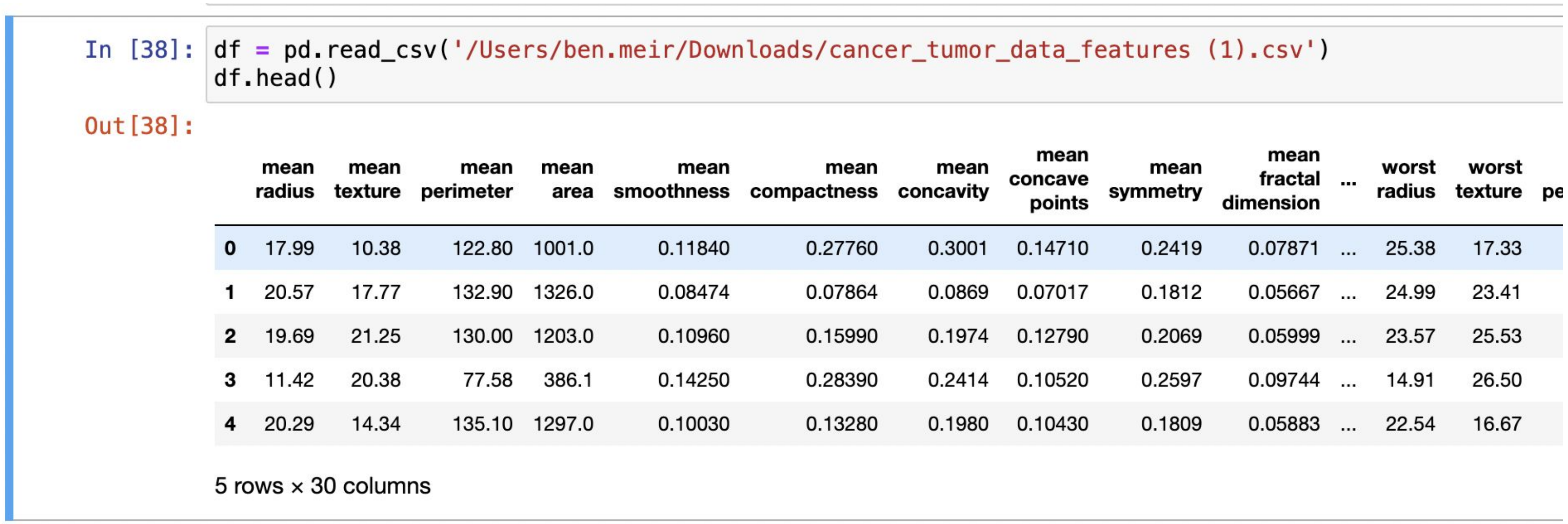
$$Z_1 = \phi_{11} X_1 + \phi_{21} X_2 + \ldots + \phi_{p1} X_p$$

X1 feature coefficient

X2 feature coefficient

PCA - Python Example

For the example we will use the **‘cancer_tumor_data_features.csv’** file. This dataset contains data related to different features of cancer tumors.



```
In [38]: df = pd.read_csv('/Users/ben.meir/Downloads/cancer_tumor_data_features (1).csv')
         df.head()
```

Out[38]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	pe
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	25.38	17.33	
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	24.99	23.41	
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	23.57	25.53	
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	14.91	26.50	
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	22.54	16.67	

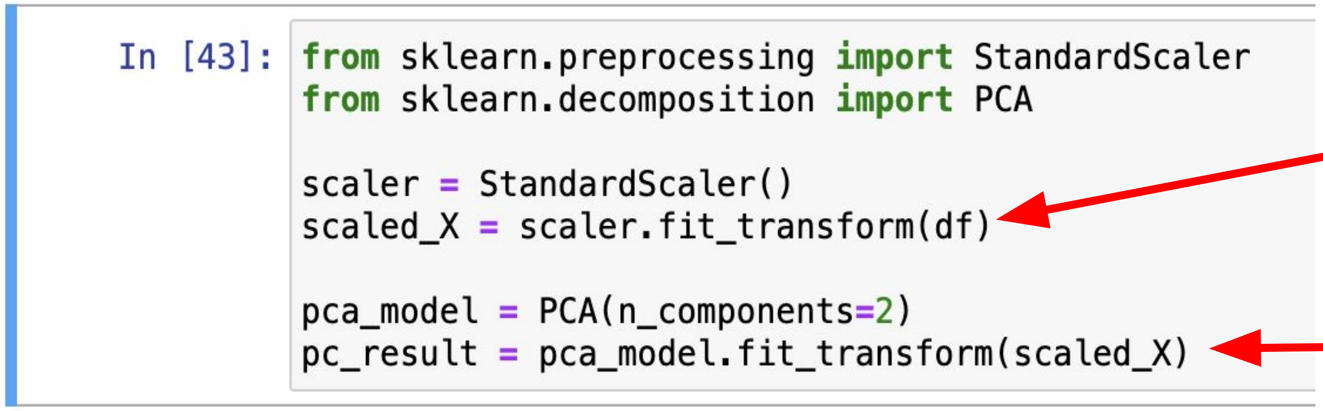
5 rows × 30 columns

As we can see, this dataset contain 30 different columns (features) which can be very complex to our machine learning models. What we want is to use PCA in order to reduce the amount of features from 30 to a lower amount.

PCA - Python Example

First, we will create a PCA instance and provide the amount of principal components we want.

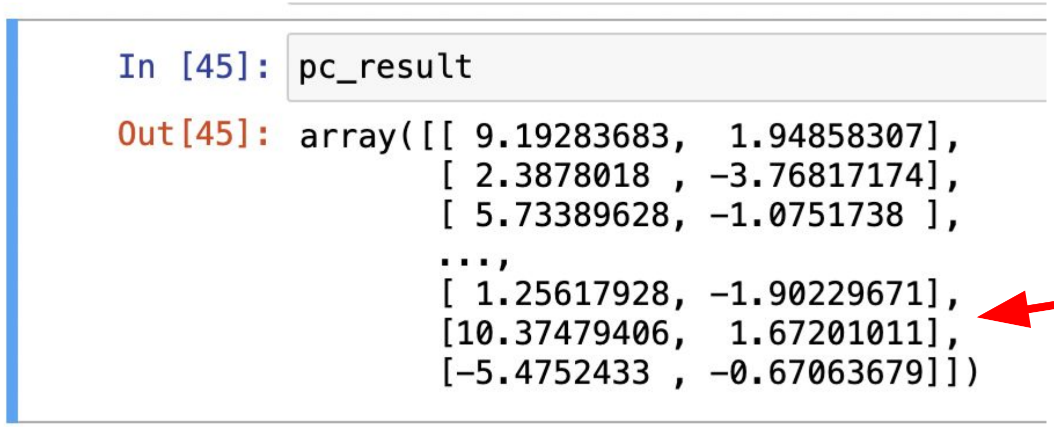
Remember that the amount of principal components determine the amount of features we are reducing our original dataset into. In our case we want to reduce our dataset from 30 features to only 2 features.



```
In [43]: from sklearn.preprocessing import StandardScaler
         from sklearn.decomposition import PCA

         scaler = StandardScaler()
         scaled_X = scaler.fit_transform(df)

         pca_model = PCA(n_components=2)
         pc_result = pca_model.fit_transform(scaled_X)
```

We first want to apply feature scaling on the entire dataset before applying the PCA

The **'fit'** will calculate the eigen vectors and eigen values.
The **'transform'** will project the original data on the chosen eigen vectors.



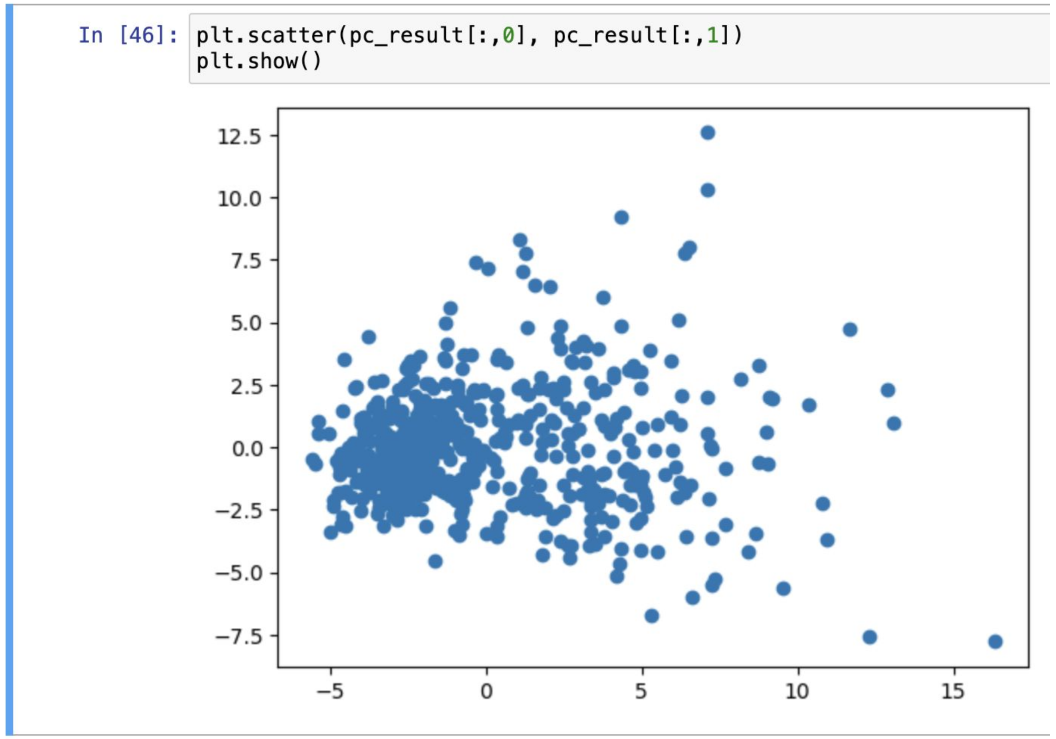
```
In [45]: pc_result

Out[45]: array([[ 9.19283683,  1.94858307],
                [ 2.3878018 , -3.76817174],
                [ 5.73389628, -1.0751738 ],
                ...,
                [ 1.25617928, -1.90229671],
                [10.37479406,  1.67201011],
                [-5.4752433 , -0.67063679]])
```

The PCA transform results are each row with the values of 2 features been calculated

PCA - Python Example

Keep in mind the new 2 features are not a subset from the original 30 features but a linear combination of those features.

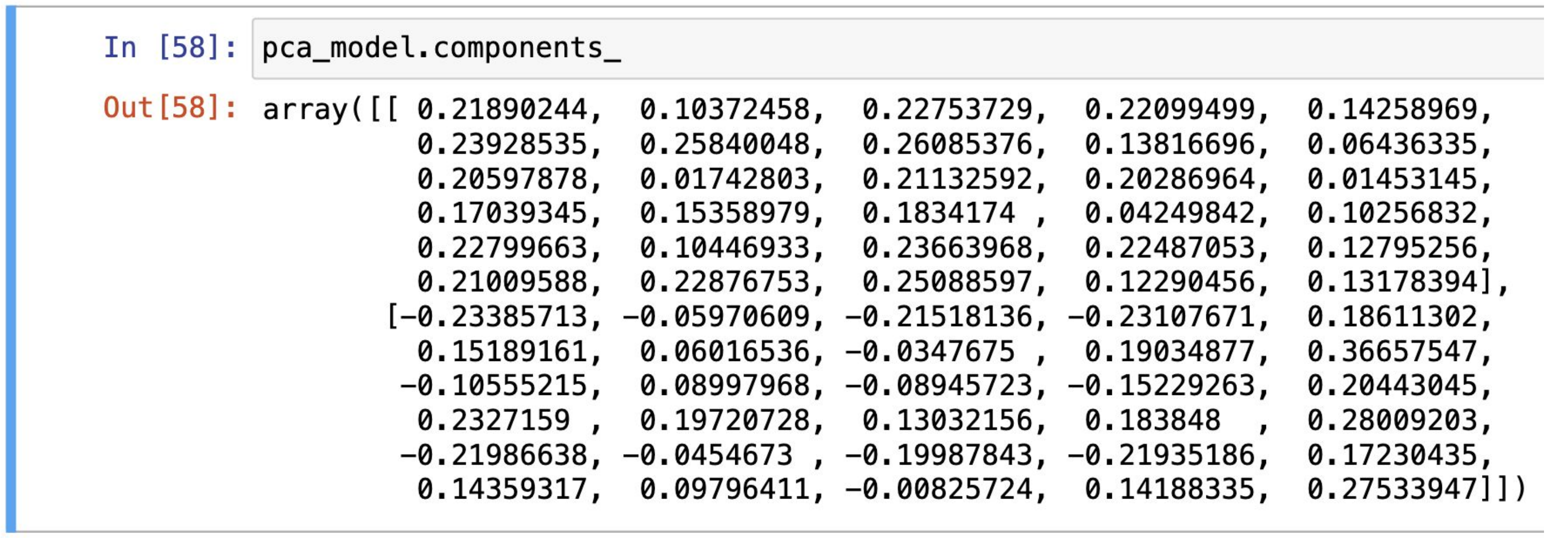
We can also generate a scatterplot to better visualize the feature values accorse the dataset:



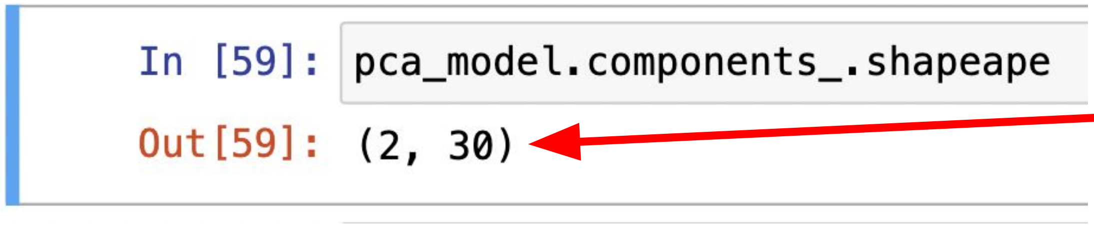
```
In [46]: plt.scatter(pc_result[:,0], pc_result[:,1])
         plt.show()
```

PCA - Python Example

We can also see the components values been generated by the PCA model using the ‘components_’ field. This will return a 2D array that contain for each component (in our case 2 components) the coefficient the model calculated for each original feature.



```
In [58]: pca_model.components_

Out[58]: array([[ 0.21890244,  0.10372458,  0.22753729,  0.22099499,  0.14258969,
                  0.23928535,  0.25840048,  0.26085376,  0.13816696,  0.06436335,
                  0.20597878,  0.01742803,  0.21132592,  0.20286964,  0.01453145,
                  0.17039345,  0.15358979,  0.1834174 ,  0.04249842,  0.10256832,
                  0.22799663,  0.10446933,  0.23663968,  0.22487053,  0.12795256,
                  0.21009588,  0.22876753,  0.25088597,  0.12290456,  0.13178394],
                [-0.23385713, -0.05970609, -0.21518136, -0.23107671,  0.18611302,
                  0.15189161,  0.06016536, -0.0347675 ,  0.19034877,  0.36657547,
                 -0.10555215,  0.08997968, -0.08945723, -0.15229263,  0.20443045,
                  0.2327159 ,  0.19720728,  0.13032156,  0.183848  ,  0.28009203,
                 -0.21986638, -0.0454673 , -0.19987843, -0.21935186,  0.17230435,
                  0.14359317,  0.09796411, -0.00825724,  0.14188335,  0.27533947]])
```



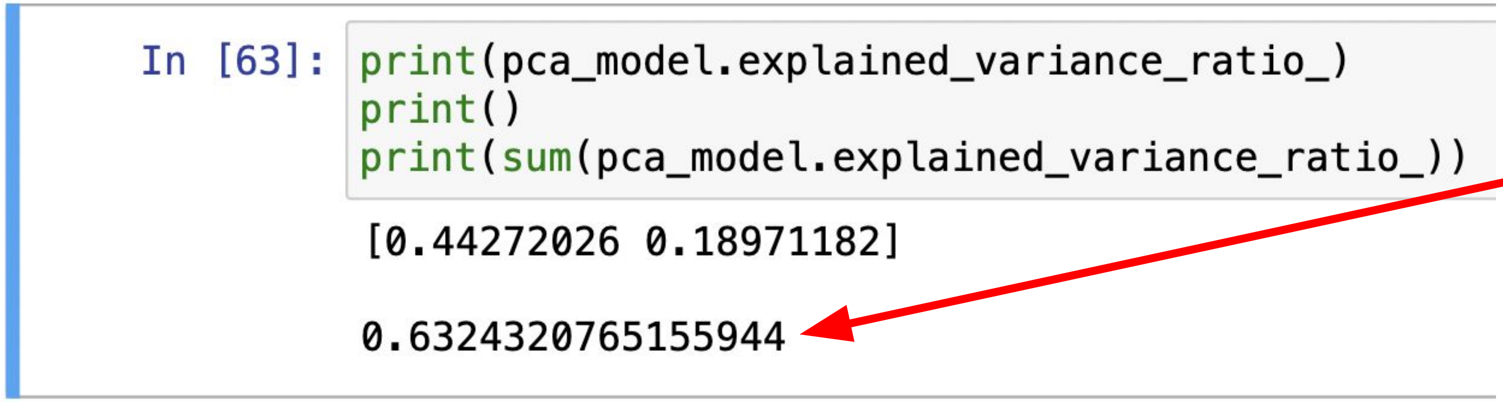
```
In [59]: pca_model.components_.shapeape

Out[59]: (2, 30)
```

The shape of the components array is 2D matrix with 2 rows for the s components and 30 columns for 30 coefficient values to each of the original 30 features.

PCA - Python Example

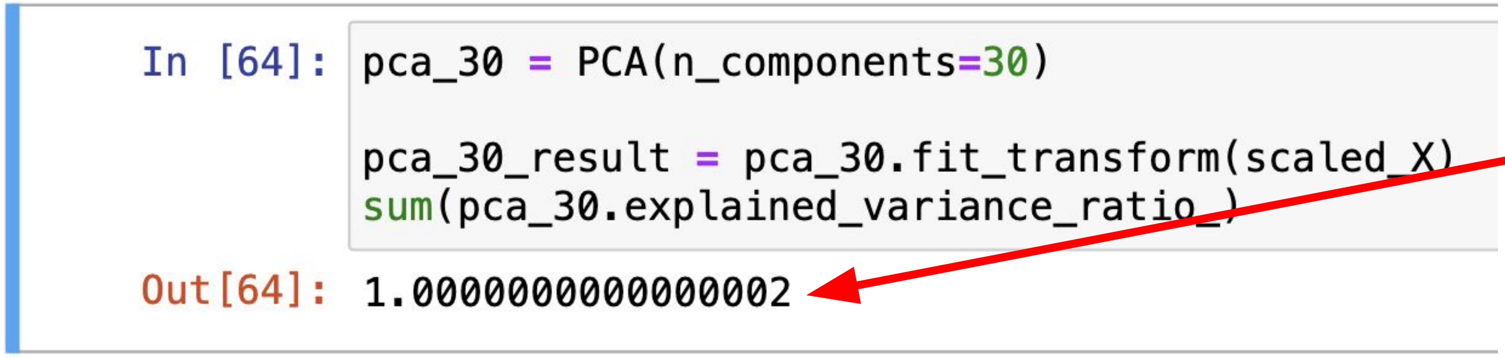
explained_variance_ratio_ → This field will show how much variance each principal component capture. The more variance the component capture the more information it can provide to us.

As we learned, the model will chose the principal components with the highest variance captured.



```
In [63]: print(pca_model.explained_variance_ratio_)
         print()
         print(sum(pca_model.explained_variance_ratio_))

         [0.44272026 0.18971182]

         0.6324320765155944
```

The first 2 principal components capture more than 60% of the variance ratio

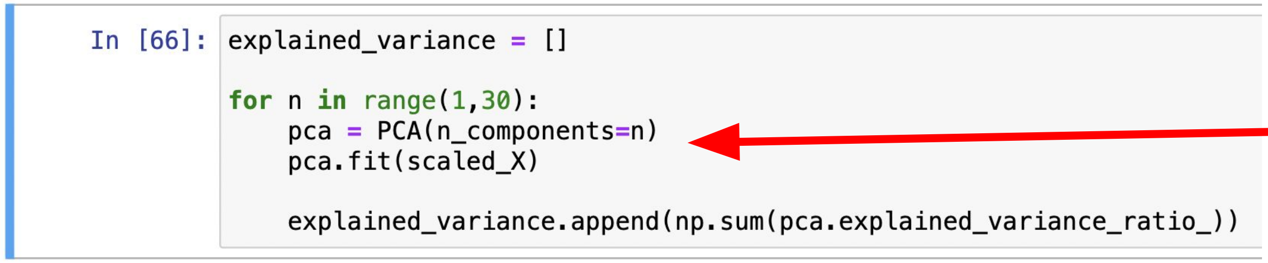


```
In [64]: pca_30 = PCA(n_components=30)

         pca_30_result = pca_30.fit_transform(scaled_X)
         sum(pca_30.explained_variance_ratio_)

Out[64]: 1.0000000000000002
```

If we will generate a PCA with 30 components we will get that the sum explained variance ratio will be 1, this is because we now capture any variance possible using the same amount of principal components as the original features.

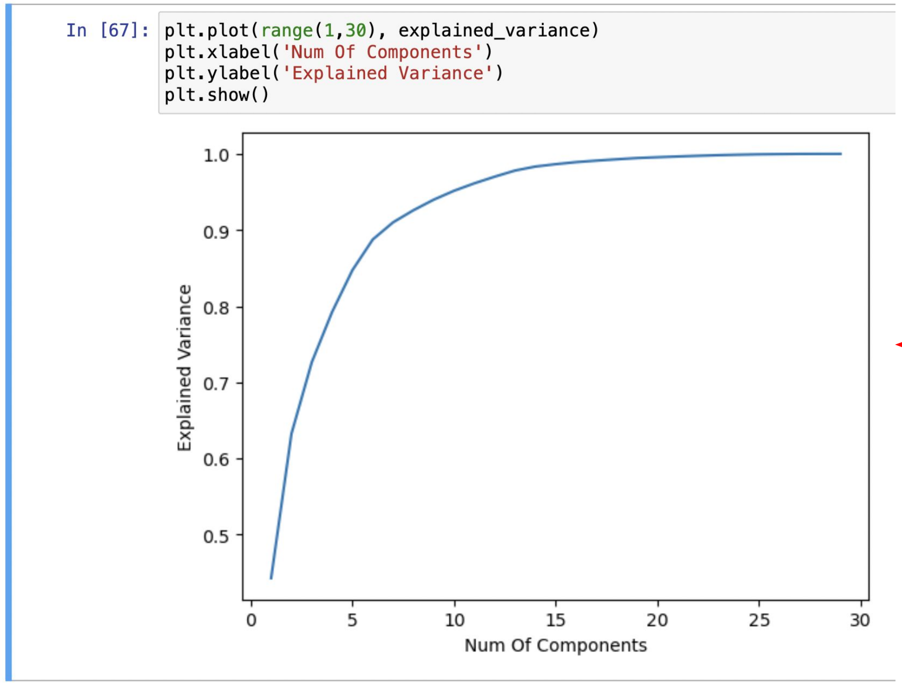
PCA - Python Example

We can also visualized the amount of explained variance we are adding by increasing the amount of principal components. This can help us determine the optimal principal components amount.



```
In [66]: explained_variance = []

         for n in range(1,30):
             pca = PCA(n_components=n)
             pca.fit(scaled_X)

             explained_variance.append(np.sum(pca.explained_variance_ratio_))
```

We generate PCA with different principal components value and save the sum of the explained variance ration of each PCA



```
In [67]: plt.plot(range(1,30), explained_variance)
         plt.xlabel('Num Of Components')
         plt.ylabel('Explained Variance')
         plt.show()
```

We can generate a simple line plot to visualize the increase in explained variance resulting by the increase in the number of principal components