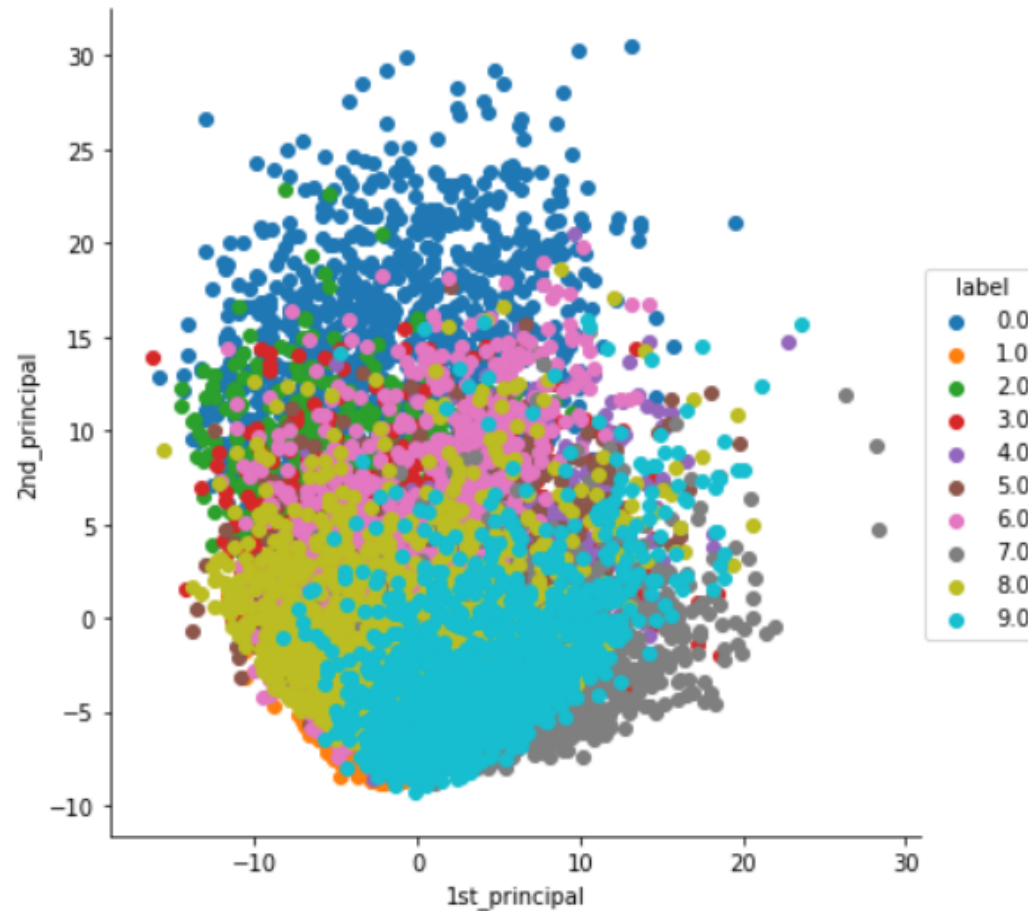


# What is Feature Extraction?

Feature extraction is a process of transforming raw data into a set of features that are more informative and relevant for machine learning tasks. The goal of feature extraction is to reduce the dimensionality of the data while preserving as much information as possible.

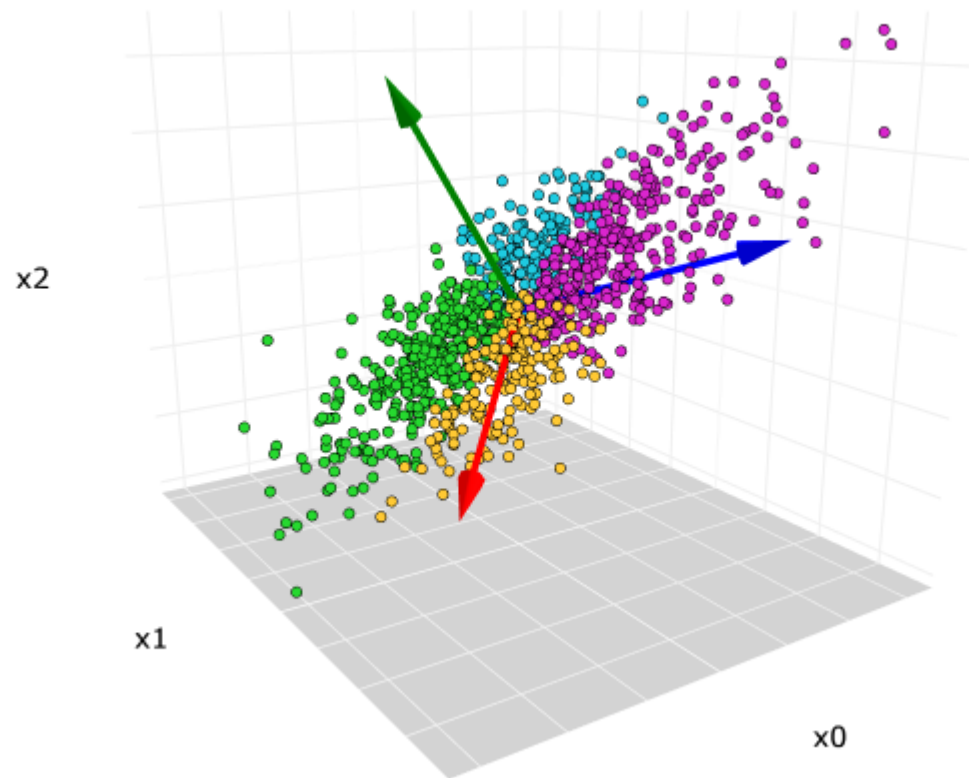


There are two main types of feature extraction: supervised and unsupervised.

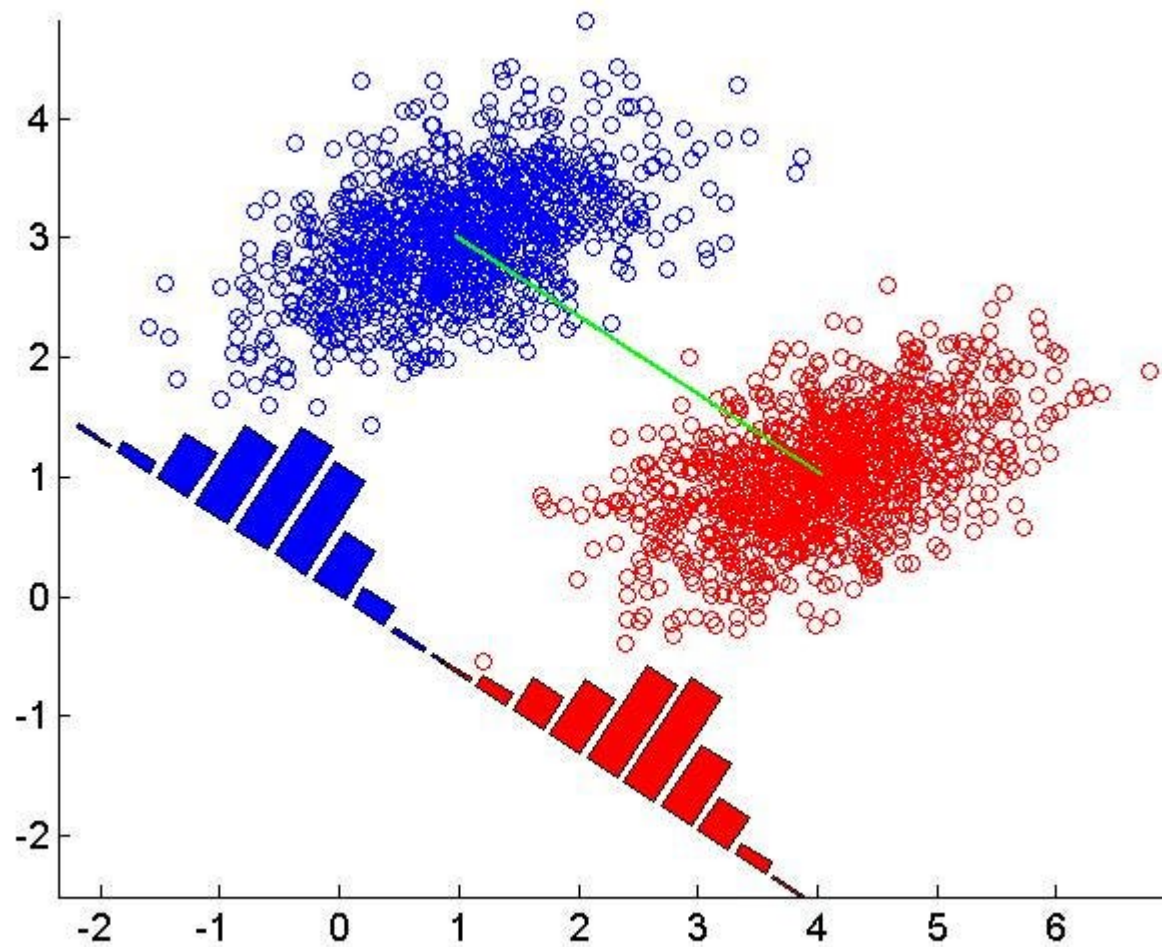
- Supervised feature extraction uses labeled data to learn a mapping from the raw data to a set of features. This type of feature extraction is often used in classification tasks, where the goal is to learn a model that can predict the class of a new data point.
- Unsupervised feature extraction does not use labeled data. Instead, it uses statistical techniques to find patterns in the raw data and to extract features that are likely to be informative. This type of feature extraction is often used in clustering tasks, where the goal is to group similar data points together.

**There are many different feature extraction techniques, some of the most common include:**

- **Principal component analysis (PCA):** PCA is a linear transformation that projects the data onto a lower-dimensional space in such a way that the variance of the projected data is maximized.



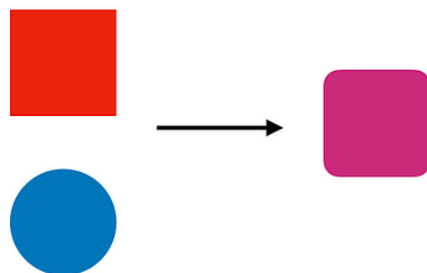
- **Linear discriminant analysis (LDA):** LDA is a supervised learning technique that finds a linear combination of features that maximizes the separation between different classes.



- **Independent component analysis (ICA):** ICA is a non-linear transformation that finds a set of features that are mutually independent.

## PCA

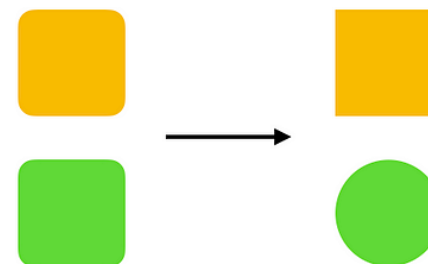
**Compresses information**



Requires preprocessing: autoscaling

## ICA

**Separates information**

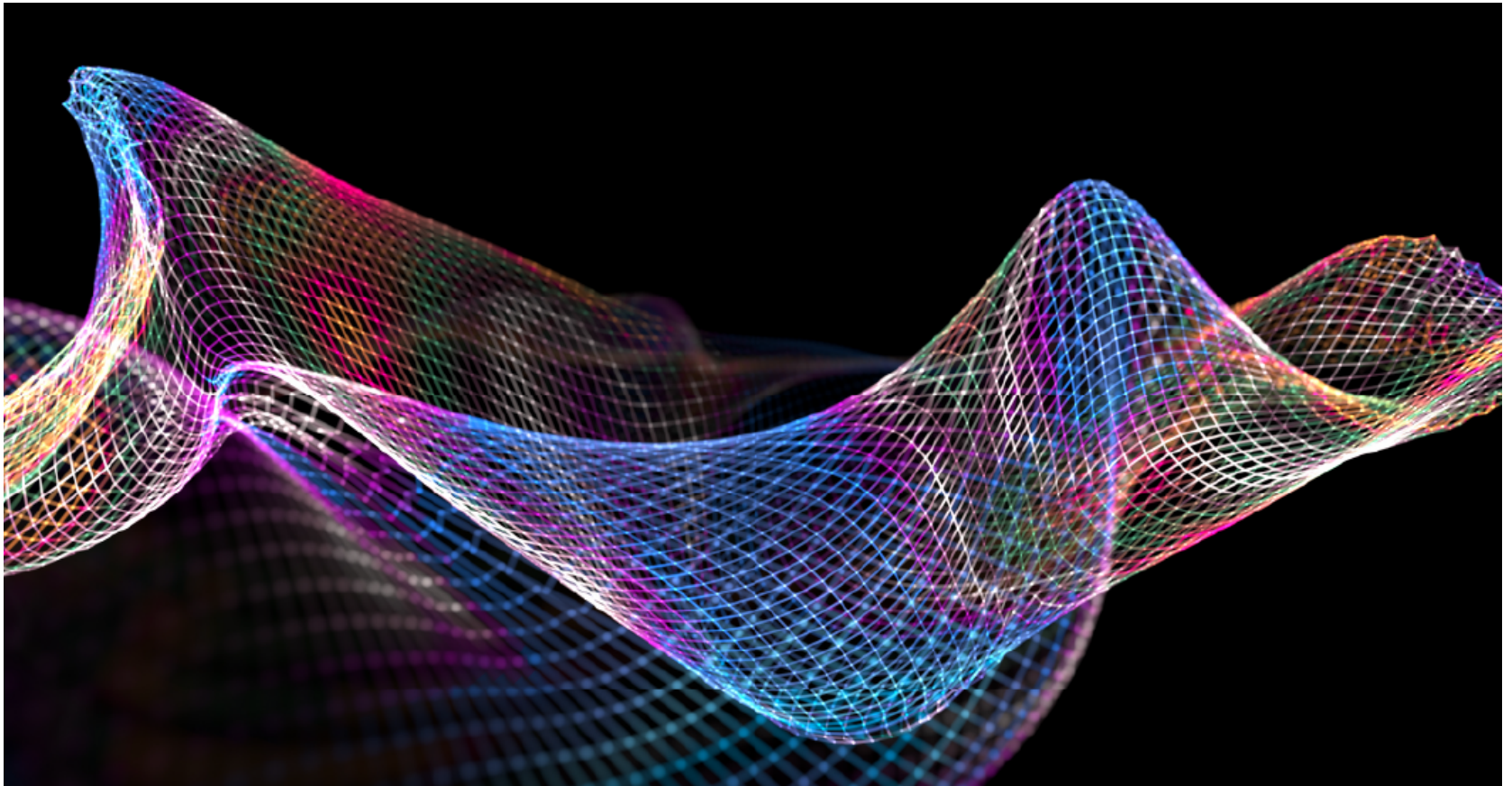


Requires preprocessing: autoscaling

Often benefits from first applying PCA

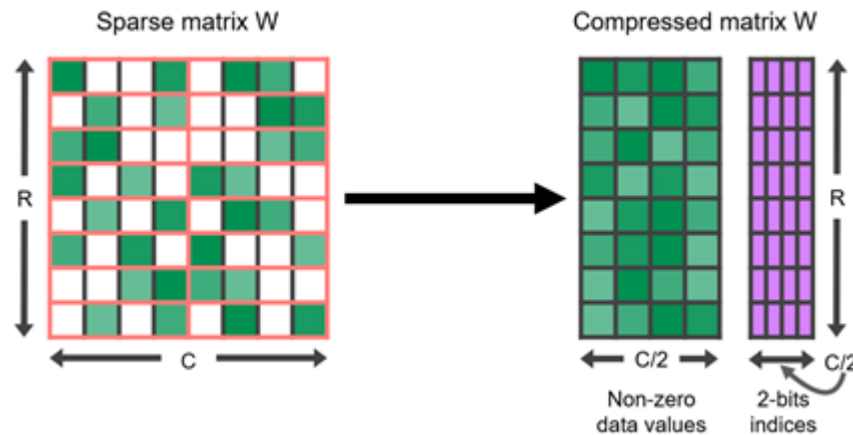
## What is Curse of Dimensionality ?

The "Curse of Dimensionality" refers to a phenomenon that occurs when working with high-dimensional data, where the increase in the number of dimensions leads to various challenges and problems. It is used to describe the challenges that arise when working with data in high-dimensional spaces. These challenges can make it difficult to find patterns in the data, to cluster data points, and to measure distances between points.



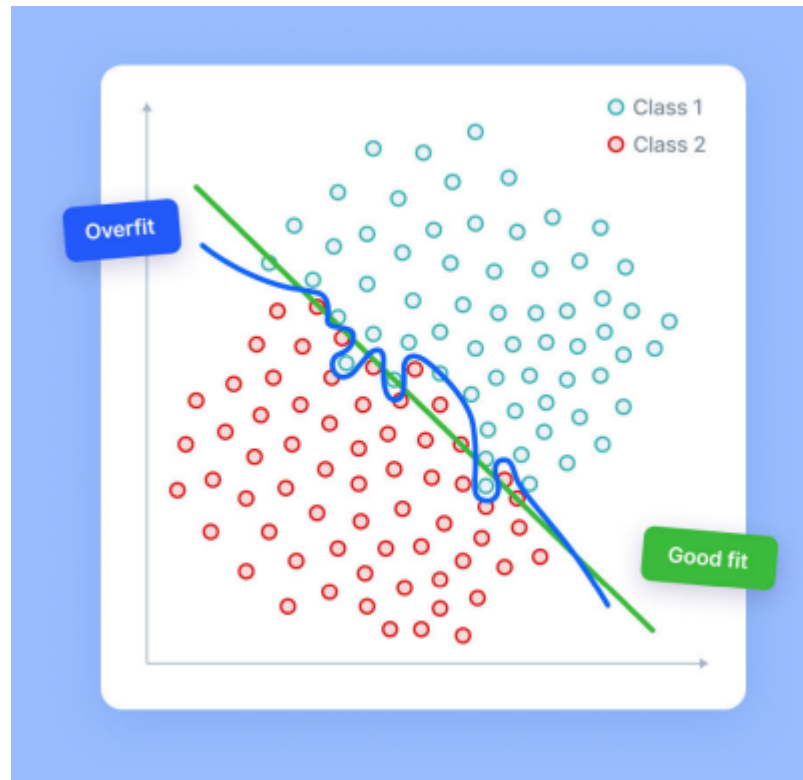
When the dimensionality of a dataset increases, several issues arise that can impact the performance and effectiveness of many algorithms and techniques. Some key challenges associated with the curse of dimensionality include:

1. **Increased sparsity:** As the number of dimensions increases, the available data points become increasingly sparse within the high-dimensional space. This sparsity can make it difficult to find meaningful patterns or relationships between the data points.



2. **Increased computational complexity:** High-dimensional data requires more computational resources to process, analyze, and visualize. Many algorithms have exponential or super-exponential time complexity with respect to the number of dimensions, making them computationally expensive and inefficient.
3. **Increased data requirements:** As the number of dimensions grows, the amount of data needed to achieve reliable statistical significance also increases exponentially. Collecting a sufficient amount of data to represent the high-dimensional space can become impractical or costly.
4. **Degraded performance of algorithms:** Many machine learning and data analysis algorithms rely on distance metrics, such as Euclidean distance or cosine similarity, to measure similarity or dissimilarity between data points. In high-dimensional spaces, these distance metrics become less effective, as the distance between points tends to converge, leading to reduced discriminatory power and degraded algorithm performance.





5. **Overfitting:** High-dimensional spaces provide more freedom for complex models to fit the data perfectly, including noise and spurious correlations. This can result in overfitting, where the model performs well on the training data but fails to generalize to new, unseen data.

To mitigate the curse of dimensionality, several techniques are employed, such as dimensionality reduction (e.g., **Principal Component Analysis**), feature selection, and feature engineering. These methods aim to reduce the number of dimensions while preserving the most relevant information or transforming the data into a lower-dimensional representation where the inherent structure is more apparent.

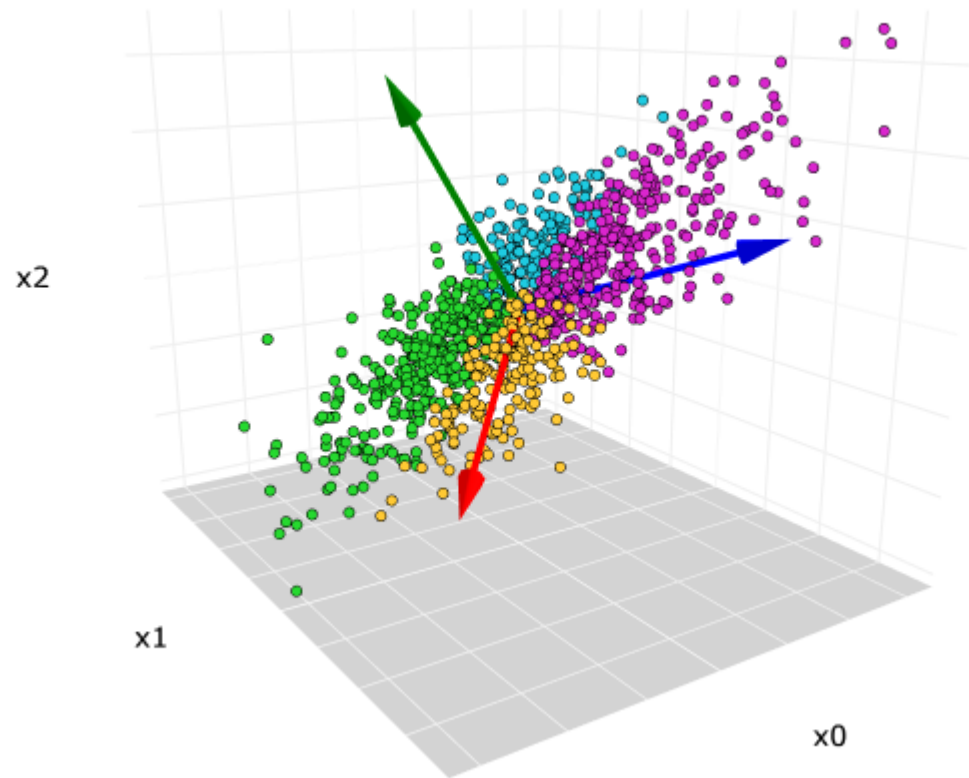
Overall, the curse of dimensionality highlights the challenges that arise when dealing with high-dimensional data, emphasizing the need for careful consideration and appropriate techniques to address these challenges effectively.

## What is PCA (Principal Component Analysis) ?



**Principal Component Analysis**, is a dimensionality reduction technique commonly used in data analysis and machine learning. It helps to identify patterns and structure in high-dimensional data by transforming it into a lower-dimensional representation while retaining the most important information.

- The main goal of PCA is to find a new set of orthogonal axes, called principal components, that capture the maximum variance in the data. The first principal component is the direction along which the data varies the most. The second principal component is orthogonal to the first and captures the next highest variance, and so on. Each principal component is a linear combination of the original features in the dataset.



**The benefits of using PCA:**

1. **Reduces the dimensionality of the data:** This can make it easier for machine learning algorithms to learn and make accurate predictions.
2. **Improves the interpretability of the model:** By extracting meaningful features, we can make it easier to understand how the model works and why it makes the predictions that it does.
3. **Improves the performance of the model:** By extracting relevant features, we can improve the accuracy of the model on both training and test data.
4. **PCA is a relatively simple technique to implement:** There are many software packages that can be used to implement PCA, and the algorithm itself is relatively straightforward.
5. **PCA is available in most statistical software packages:** This makes it easy to use PCA for a variety of tasks, and there is a large body of literature available on the topic.

Overall, PCA is a powerful dimensionality reduction technique that can be used for a variety of purposes. However, it is important to be

**In principal component analysis (PCA), maximum variance refers to the objective of finding the principal components, which are the directions in which the data varies the most. The first principal component is the direction that accounts for the most variance in the data, the second principal component is the direction that accounts for the second most variance in the data, and so on.**

- The reason why PCA maximizes variance is because it is trying to find the directions in which the data is most spread out. This is because the more spread out the data is, the more information there is about the data in that direction.

**For example**, if you have a dataset of points in 2D space, the first principal component will be the direction that the data points spread out the most. This means that the first principal component will be the direction that contains the most information about the data.

- PCA is a powerful dimensionality reduction technique because it can be used to reduce the dimensionality of the data while still preserving most of the information in the data. This is because the principal components contain the most information about the data.
- The maximum variance in PCA is important because it is the goal of the algorithm. By maximizing the variance, PCA is able to find the directions in which the data varies the most, which are the directions that contain the most information about the data.

Here are some of the benefits of maximizing variance in PCA:

1. It can help to improve the accuracy of machine learning models.
2. It can help to reduce the dimensionality of the data without losing too much information.
3. It can help to visualize the data in a lower-dimensional space.

Here are some of the challenges of maximizing variance in PCA:

1. It can be sensitive to the scale of the features.
2. It can be computationally expensive for large datasets.
3. It can be difficult to interpret the results of PCA.

Overall, maximizing variance in PCA is a powerful technique that can be used to improve the performance and interpretability of machine learning models. However, it is important to be aware of the challenges of PCA before using it.

## Step by step PCA implementation

```
In [1]: import numpy as np
import pandas as pd

# Set random seed
np.random.seed(23)

# Generate samples for class 1
mu_vec1 = np.array([0, 0, 0])
cov_mat1 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
class1_sample = np.random.multivariate_normal(mu_vec1, cov_mat1, 20)

# Create DataFrame for class 1
df = pd.DataFrame(class1_sample, columns=['feature1', 'feature2', 'feature3'])
df['target'] = 1

# Generate samples for class 2
mu_vec2 = np.array([1, 1, 1])
cov_mat2 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
class2_sample = np.random.multivariate_normal(mu_vec2, cov_mat2, 20)
```

In [2]:

```
df.head()
```

Out[2]:

	feature1	feature2	feature3	target
0	0.666988	0.025813	-0.777619	1
1	0.948634	0.701672	-1.051082	1
2	-0.367548	-1.137460	-1.322148	1
3	1.772258	-0.347459	0.670140	1
4	0.322272	0.060343	-1.043450	1

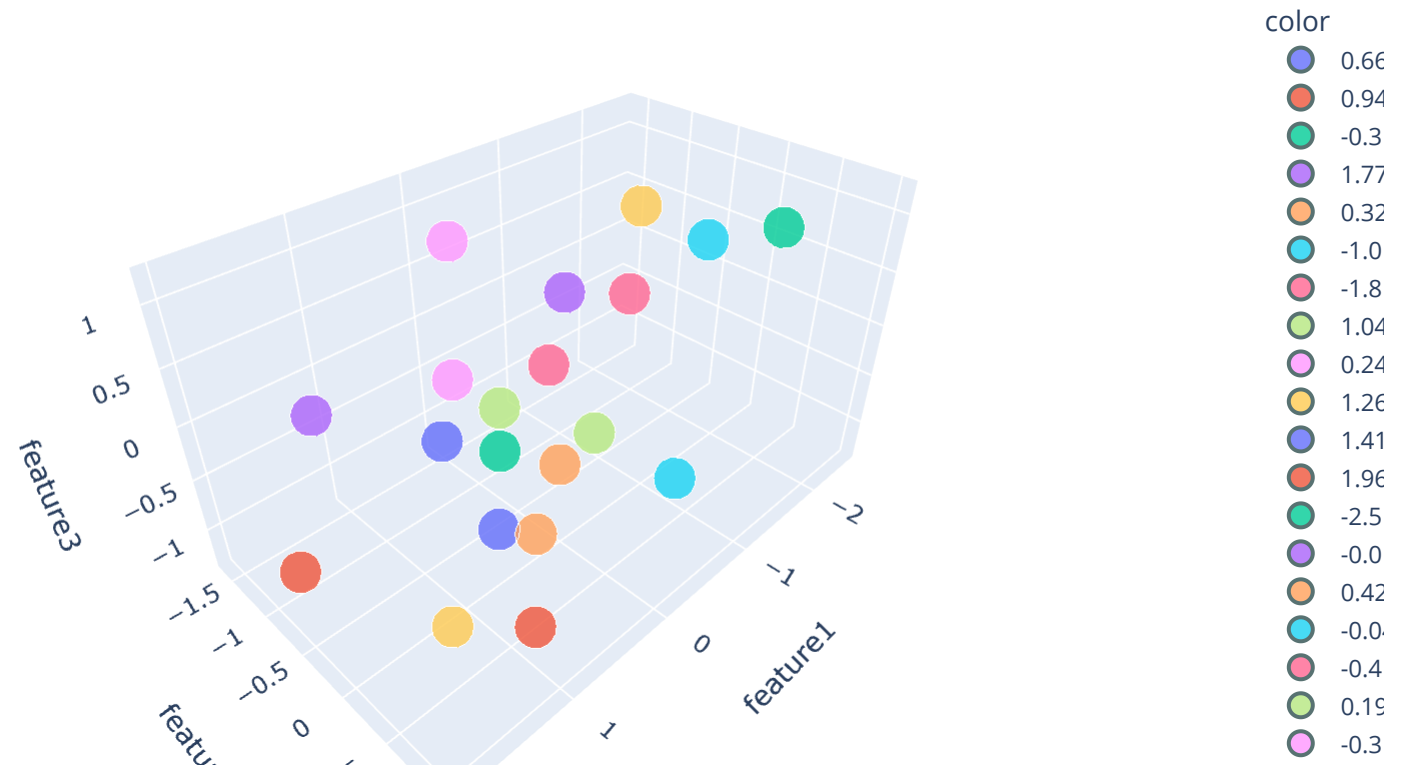
```
In [3]: import plotly.express as px

# Convert y_train to string if needed
# y_train_trf = y_train.astype(str)

# Create a 3D scatter plot using Plotly Express
fig = px.scatter_3d(df,
                    x=df['feature1'],
                    y=df['feature2'],
                    z=df['feature3'],
                    color=df['feature1'].astype(str), # Assign color based on feature1
                    opacity=0.8 # Set opacity of markers
                    )

# Update marker properties
fig.update_traces(marker=dict(size=12,
                              line=dict(width=2,
                                          color='DarkSlateGrey')),
                  selector=dict(mode='markers'))

# Display the plot
fig.show()
```



```
In [4]: # Step 1 - Apply standard scaling
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

df.iloc[:,0:3] = scaler.fit_transform(df.iloc[:,0:3])
```

In [5]:

```
# Step 2 - Find Covariance Matrix
covariance_matrix = np.cov([df.iloc[:,0],df.iloc[:,1],df.iloc[:,2]])
print('Covariance Matrix:\n', covariance_matrix)
```

Covariance Matrix:

```
[[ 1.05263158  0.20397591 -0.28888004]
 [ 0.20397591  1.05263158  0.10956124]
 [-0.28888004  0.10956124  1.05263158]]
```

In [6]:

```
# Step 3 - Finding Eigen Value and Eigen Vectors
eigen_values, eigen_vectors = np.linalg.eig(covariance_matrix)
```

In [7]: eigen\_values

Out[7]: array([0.64212617, 1.36120658, 1.15456198])

In [8]: eigen\_vectors

Out[8]: array([[ -0.65172443, 0.74834128, 0.12345283],
 [ 0.48046517, 0.28140349, 0.8306415 ],
 [-0.58686326, -0.60066414, 0.54294945]])





In [9]: %pylab inline

```

from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
from matplotlib.patches import FancyArrowPatch

# Custom Arrow class for 3D plotting
class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)

# Create a 3D figure
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111, projection='3d')

# Plot the data points
ax.plot(df['feature1'], df['feature2'], df['feature3'], 'o', markersize=8, color='blue', alpha=0.2)

# Plot the mean data point
ax.plot([df['feature1'].mean()], [df['feature2'].mean()], [df['feature3'].mean()], 'o', markersize=10, color='red')

# Plot the eigenvectors
for v in eigen_vectors.T:
    # Create an arrow from mean to eigenvector
    a = Arrow3D([df['feature1'].mean(), v[0]], [df['feature2'].mean(), v[1]], [df['feature3'].mean(), v[2]],
                ax.add_artist(a)

# Set labels for the axes
ax.set_xlabel('x_values')
ax.set_ylabel('y_values')
ax.set_zlabel('z_values')

# Set the plot title
plt.title('Eigenvectors')

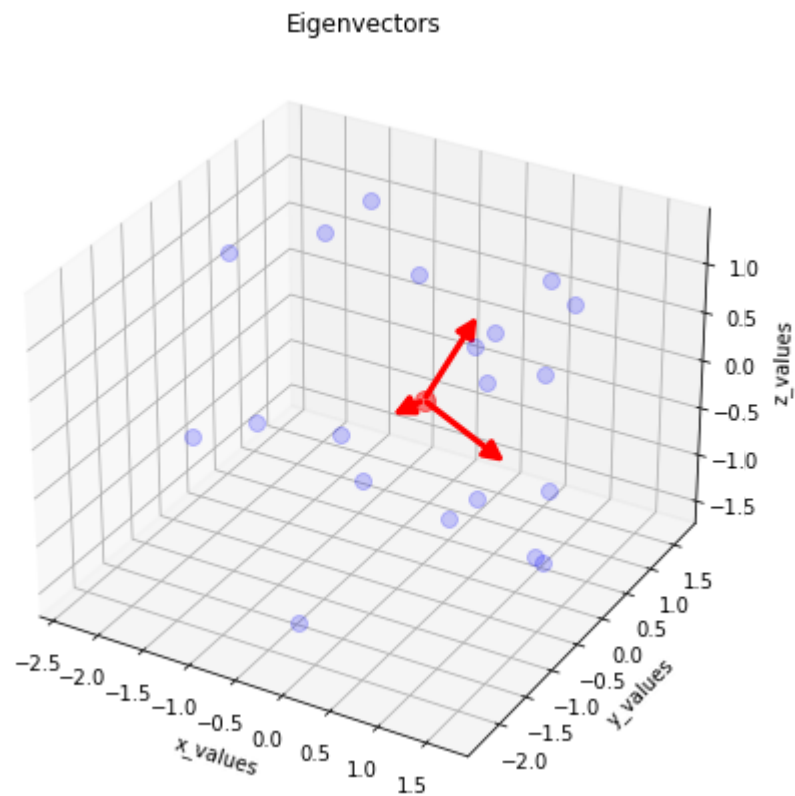
```

```
# Display the plot  
plt.show()
```

Populating the interactive namespace from numpy and matplotlib

C:\Users\user\AppData\Local\Temp\ipykernel\_12340\1008480673.py:16: MatplotlibDeprecationWarning:

The M attribute was deprecated in Matplotlib 3.4 and will be removed two minor releases later. Use self.axes.M instead.



In [10]: *# Returns the first two eigenvectors from the given list of eigenvectors.*

```
pc = eigen_vectors[0:2]
pc
```

Out[10]: array([[ -0.65172443, 0.74834128, 0.12345283],  
[ 0.48046517, 0.28140349, 0.8306415 ]])

In [11]: *# Transform the dataframe using dot product*

```
transformed_df = np.dot(df.iloc[:, 0:3], pc.T)
```

*# 40,3 - 3,2*

*# Create a new dataframe with the transformed values*

```
new_df = pd.DataFrame(transformed_df, columns=['PC1', 'PC2'])
```

```
new_df['target'] = df['target'].values
```

*# Display the first few rows of the new dataframe*

```
new_df.head()
```

Out[11]:

	PC1	PC2	target
0	-0.304909	-0.545559	1
1	0.238477	-0.429512	1
2	-1.078773	-2.044435	1
3	-1.135779	1.289053	1
4	-0.107685	-0.957342	1

```
In [12]: import plotly.express as px

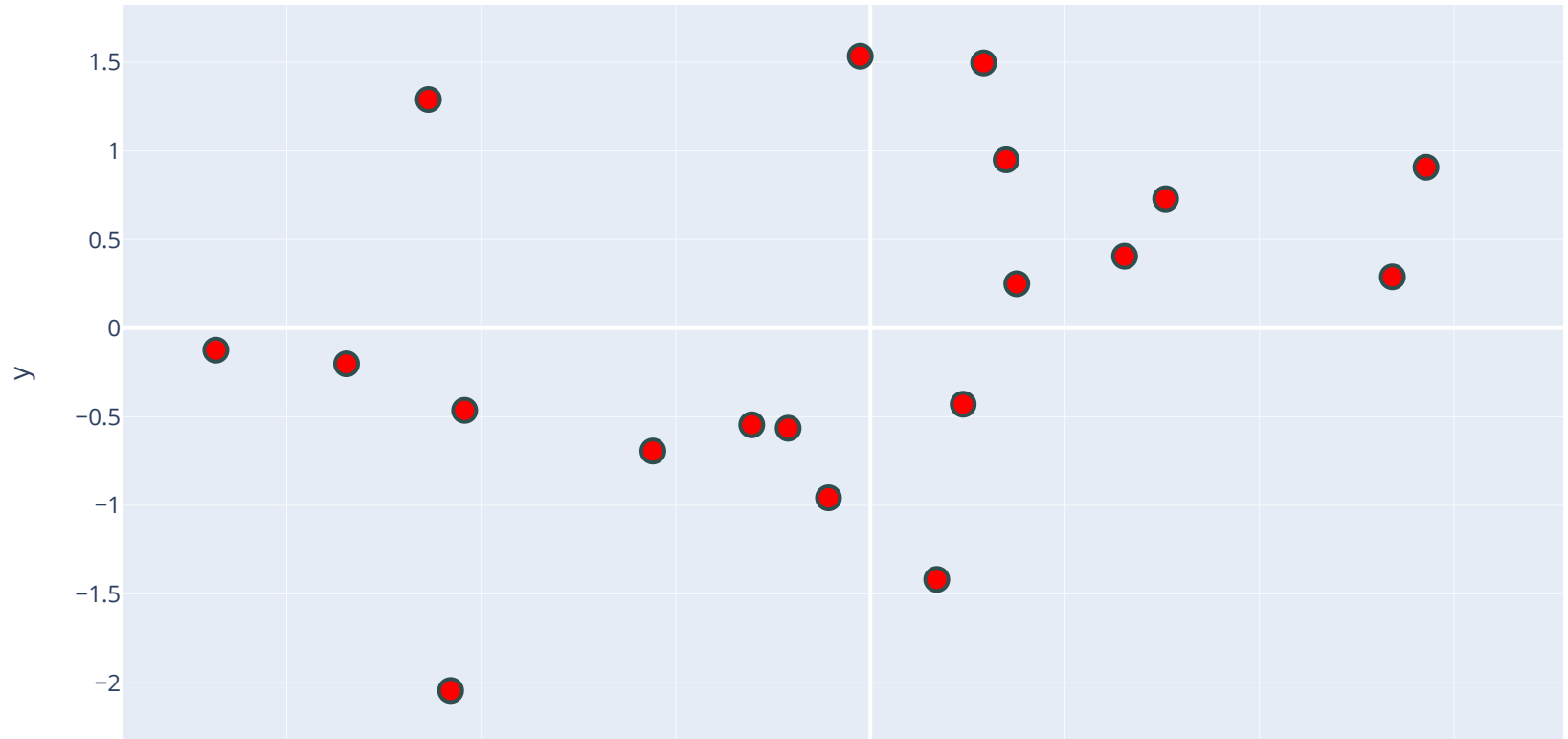
# Convert the 'target' column to string
new_df['target'] = new_df['target'].astype('str')

# Define a custom list of colors
custom_colors = ['#FF0000', '#00FF00', '#0000FF'] # Replace with your desired colors

# Create a scatter plot using Plotly Express
fig = px.scatter(
    x=new_df['PC1'],
    y=new_df['PC2'],
    color=new_df['target'],
    color_discrete_sequence=custom_colors
)

# Update marker properties
fig.update_traces(
    marker=dict(
        size=12,
        line=dict(
            width=2,
            color='DarkSlateGrey'
        )
    ),
    selector=dict(mode='markers')
)

# Display the plot
fig.show()
```



## Practical Example With MNIST Data

```
In [13]: import pandas as pd
df = pd.read_csv("D:\\datascience\\Nitish sir\\Feature Enginnering\\Feature Extraction\\train.csv")
```

```
In [14]: df.head()
```

```
Out[14]:
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0

5 rows × 785 columns



## About DATA

MNIST ("Modified National Institute of Standards and Technology") is the de facto "hello world" dataset of computer vision. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine learning techniques emerge, MNIST remains a reliable resource for researchers and learners alike.

To correctly identify digits from a dataset of tens of thousands of handwritten images. We've curated a set of tutorial-style kernels which cover everything from regression to neural networks. We encourage you to experiment with different algorithms to learn first-hand what works well and how techniques compare.

- In the MNIST dataset, each image consists of **28x28 pixels**. This means that each image is a grayscale image with a resolution of 28 pixels in the horizontal direction (width) and 28 pixels in the vertical direction (height).
- The pixels in the image represent the intensity of the grayscale color, ranging from **0 (black) to 255 (white)**. Each pixel is a single value, indicating the brightness or darkness of that particular pixel.
- Since there are **28x28 = 784 pixels** in each image, the MNIST dataset can be represented as a collection of **784-dimensional feature vectors**, with each element in the vector representing the intensity value of a specific pixel in the corresponding image.

This representation allows the images to be processed by various machine learning algorithms, where each pixel value can be considered as a feature or input for the model.



```
In [15]: # Get the shape of the DataFrame
df.shape
```

Out[15]: (42000, 785)

- The dimensions (42000, 785) represent the shape of the MNIST dataset when it is organized in a tabular format, where each row corresponds to an image and each column represents a feature or attribute.
- In this case, the dataset contains 42,000 images, and each image has 785 elements or features. The additional element is usually a label or target value associated with each image, indicating the digit it represents.
- Out of the 785 elements, the first 784 elements represent the pixel values of the image ( $28 \times 28 = 784$  pixels), and the last element represents the label or digit category for that image.

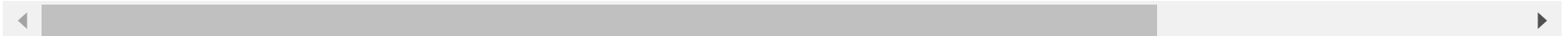
So, the dataset is structured as a 2-dimensional array with 42,000 rows and 785 columns, where each row contains the pixel values and the label for an individual image in the MNIST dataset.

```
In [16]: df.sample()
```

Out[16]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779
36501	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0

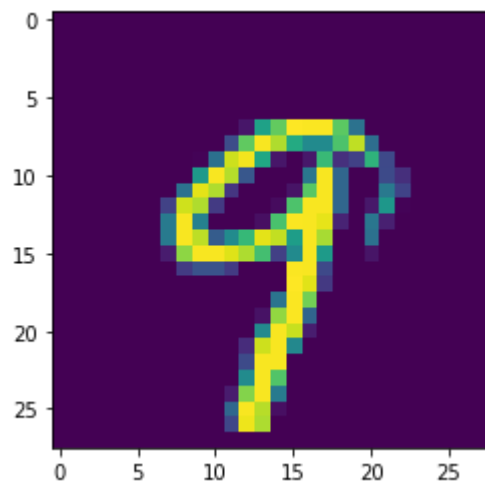
1 rows × 785 columns



```
In [17]: import matplotlib.pyplot as plt

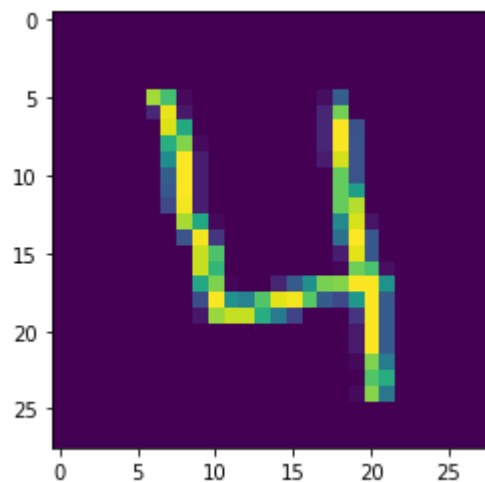
# Displaying the image of the focal cell
# The value 36501 is assigned to 9
plt.imshow(df.iloc[36501,1:].values.reshape(28,28))
```

Out[17]: <matplotlib.image.AxesImage at 0x21fdce515e0>



```
In [18]: # Display an image using matplotlib  
plt.imshow(df.iloc[3,1:].values.reshape(28,28)) # Converted 2D to 1D
```

Out[18]: <matplotlib.image.AxesImage at 0x21fdd2e7130>



```
In [19]: X = df.iloc[:,1:]  
y = df.iloc[:,0]
```

```
In [20]: X
```

Out[20]:

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel777	pixel778	p
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
1	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	
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
41995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
41996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
41997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
41998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	
41999	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	

42000 rows × 784 columns

```
In [21]: y # label
```

Out[21]:

0	1
1	0
2	1
3	4
4	0
	..
41995	0
41996	1
41997	7
41998	6
41999	9

Name: label, Length: 42000, dtype: int64

```
In [22]: X.shape
```

```
Out[22]: (42000, 784)
```

```
In [23]: y.shape
```

```
Out[23]: (42000,)
```

```
In [24]: from sklearn.model_selection import train_test_split  
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2)
```

```
In [25]: X_train.shape
```

```
Out[25]: (33600, 784)
```

```
In [26]: X_test.shape
```

```
Out[26]: (8400, 784)
```

```
In [27]: from sklearn.neighbors import KNeighborsClassifier
```

```
In [28]: knn = KNeighborsClassifier()
```

```
In [29]: knn.fit(X_train,y_train)
```

```
Out[29]: 

▼ KNeighborsClassifier



KNeighborsClassifier()


```

```
In [30]: import time

start = time.time()

y_pred = knn.predict(X_test)

print(time.time() - start)
```

31.31436324119568

```
In [31]: from sklearn.metrics import accuracy_score

accuracy_score(y_test , y_pred)
```

Out[31]: 0.9646428571428571

## The steps involved in performing PCA are as follows:

1. **Standardize the data:** If the features in the dataset have different scales, it is important to standardize them (subtract the mean and divide by the standard deviation) to ensure that each feature contributes equally to the analysis.
2. **Compute the covariance matrix:** The covariance matrix measures the relationships between the different features in the dataset. It provides information about how the features vary together.
3. **Calculate the eigenvectors and eigenvalues:** The eigenvectors and eigenvalues of the covariance matrix represent the principal components. The eigenvectors are the directions of maximum variance, while the corresponding eigenvalues indicate the amount of variance explained by each principal component.
4. **Select the principal components:** To reduce the dimensionality, you can select a subset of the principal components based on their corresponding eigenvalues. Typically, you would choose the top-k components that explain a significant portion of the variance (e.g., 90% or more).
5. **Transform the data:** Multiply the standardized data by the selected principal components to obtain the transformed dataset with reduced dimensionality. Each data point is projected onto the new principal component axes.

```
In [32]: # Standardize the data

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
In [33]: X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
In [34]: # PCA

from sklearn.decomposition import PCA

pca = PCA(n_components=None)
```

```
In [35]: # Perform PCA transformation on X_train
X_train = pca.fit_transform(X_train)

# Perform PCA transformation on X_test
X_test = pca.transform(X_test)
```

```
In [36]: X_train.shape # Same as Old
```

```
Out[36]: (33600, 784)
```



```
In [37]: # We can change the Components to 100

from sklearn.decomposition import PCA

pca = PCA(n_components=100)

# Perform PCA transformation on X_train
X_train_trf = pca.fit_transform(X_train)

# Perform PCA transformation on X_test
X_test_trf = pca.transform(X_test)

X_train_trf.shape # with 100 columns
```

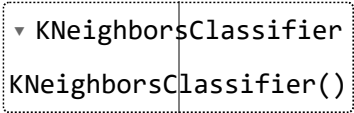
Out[37]: (33600, 100)

```
In [38]: X_train.shape #v Original Columns
```

Out[38]: (33600, 784)

```
In [39]: # With PCA

knn = KNeighborsClassifier()
knn.fit(X_train_trf , y_train)
```

Out[39]: 

```
In [40]: y_pred =knn.predict(X_test_trf)
```

```
In [41]: accuracy_score(y_test,y_pred)
```

Out[41]: 0.9532142857142857

```
In [ ]: # Loop

for i in range(1,785):
    pca = PCA(n_components= i)
    X_train_trf = pca.fit_transform(X_train)
    X_test_trf = pca.transform(X_test)
    knn = KNeighborsClassifier()
    knn.fit(X_train_trf , y_train)
    y_pred =knn.predict(X_test_trf)
    print(accuracy_score(y_test,y_pred))
```

```
In [43]: # transforming to a 2D Co-Ordinate System
```

```
pca = PCA(n_components= 2)

X_train_trf = pca.fit_transform(X_train)

X_test_trf = pca.transform(X_test)
```

```
In [44]: X_train_trf
```

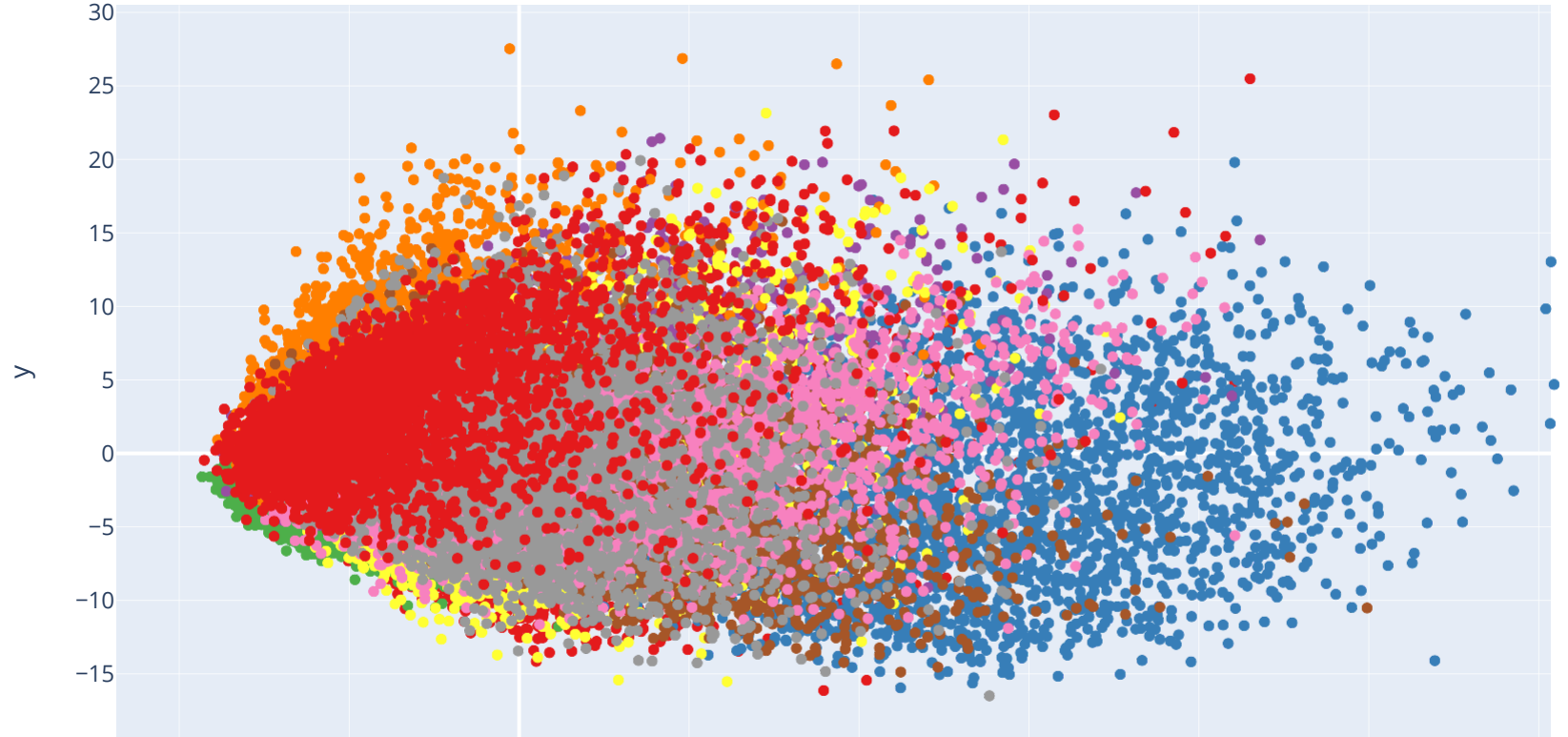
```
Out[44]: array([[ 0.615633 , -2.02992063],
 [18.02942583,  4.53350797],
 [ 5.59729473, -2.78256805],
 ...,
 [ 3.22725707,  1.01716312],
 [ 0.4603045 ,  8.81318573],
 [12.69608333, 10.54865442]])
```

```
In [45]: import plotly.express as px

# Convert y_train to string
y_train_trf = y_train.astype(str)

# Create a scatter plot using Plotly Express
fig = px.scatter(
    x=X_train_trf[:, 0],
    y=X_train_trf[:, 1],
    color=y_train_trf,
    color_discrete_sequence=px.colors.qualitative.Set1
)

# Display the plot
fig.show()
```



In [46]: *# transforming to a 3D Co-Ordinate System*

```
pca = PCA(n_components= 3)

X_train_trf = pca.fit_transform(X_train)

X_test_trf = pca.transform(X_test)
```

In [47]: X\_train\_trf

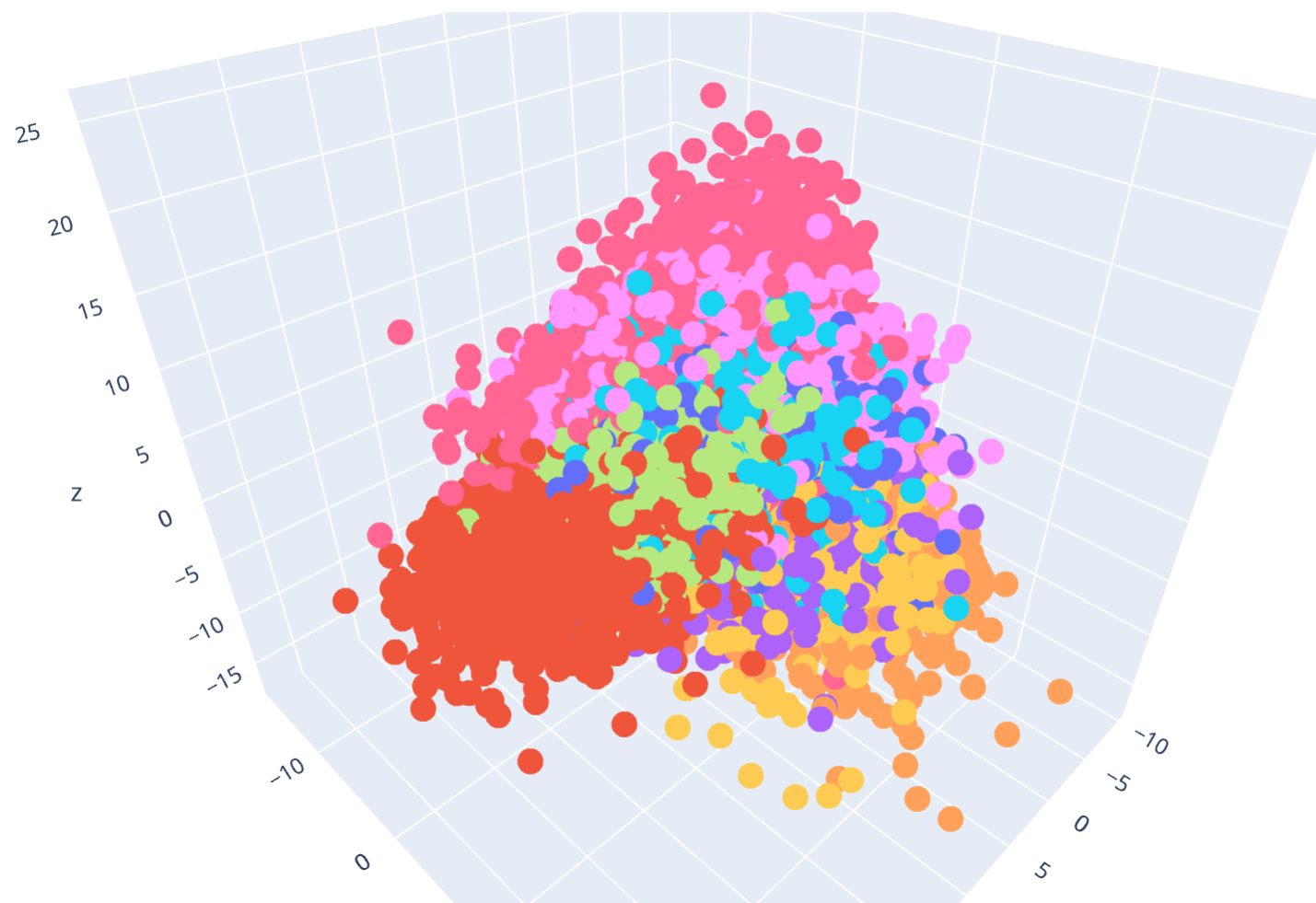
Out[47]: array([[ 0.61563503, -2.03023524, 4.13858804],  
 [18.02937395, 4.53293998, 0.45270315],  
 [ 5.59725383, -2.78307276, -0.53071837],  
 ...,  
 [ 3.22723296, 1.01702138, 5.94237509],  
 [ 0.46031466, 8.81264351, -7.91651999],  
 [12.69609363, 10.54863836, 1.45124835]])

```
In [48]: y_train_trf = y_train.astype(str)

fig = px.scatter_3d(df ,
                    x =X_train_trf[:,0],
                    y = X_train_trf[:,1],
                    z = X_train_trf[:,2],
                    color=y_train_trf,
                    )
fig.update_layout(
    margin =dict(l=20 ,r =20 , t =20 , b =20)

)

fig.show()
```



```
In [50]: pca.explained_variance_
```

```
# Eigen Values
```

```
Out[50]: array([40.73332174, 29.06684409, 26.68759881])
```



```
In [51]: pca.components_
```

```
# Eigen Vectors
```

```
Out[51]: array([[ 1.00000000e+00,  1.91858793e-10, -5.36211357e-10, ...,  
                -1.74842945e-33,  9.43749132e-34,  3.73046951e-33],  
               [-1.71389802e-10,  1.00000000e+00,  4.32922290e-09, ...,  
                -7.28311222e-34, -7.95581012e-34, -4.83815391e-33],  
               [ 4.68368349e-10, -4.10029888e-09,  9.99999999e-01, ...,  
                -6.09016561e-34, -1.33265483e-33,  6.10737704e-34]])
```

```
In [52]: pca.components_.shape
```

```
Out[52]: (3, 784)
```

```
In [53]: pca.explained_variance_ratio_ # Percentages
```

```
Out[53]: array([0.05769421, 0.04116994, 0.03780001])
```

```
In [54]:
```

```
pca = PCA(n_components= None)  
  
X_train_trf = pca.fit_transform(X_train)  
  
X_test_trf = pca.transform(X_test)
```

In [55]: `pca.explained_variance_`

Out[55]: `array([4.07333217e+01, 2.90668441e+01, 2.66875989e+01, 2.08195474e+01, 1.81542661e+01, 1.57978221e+01, 1.38142345e+01, 1.25752024e+01, 1.10264994e+01, 1.00541597e+01, 9.65156705e+00, 8.65657841e+00, 8.01708001e+00, 7.91040721e+00, 7.43572211e+00, 7.10027621e+00, 6.77581867e+00, 6.62870431e+00, 6.40160363e+00, 6.26267068e+00, 5.95303344e+00, 5.77350729e+00, 5.46913990e+00, 5.33683867e+00, 5.15774383e+00, 4.96186657e+00, 4.89282298e+00, 4.71579489e+00, 4.49885931e+00, 4.35432304e+00, 4.33090110e+00, 4.21068613e+00, 4.10779121e+00, 4.04988075e+00, 3.98047882e+00, 3.88681523e+00, 3.84214699e+00, 3.71399789e+00, 3.56404507e+00, 3.50466702e+00, 3.42838556e+00, 3.38051937e+00, 3.31037426e+00, 3.27946762e+00, 3.20393876e+00, 3.17929067e+00, 3.14592715e+00, 3.10684615e+00, 3.08025109e+00, 3.04562488e+00, 2.97404797e+00, 2.93039492e+00, 2.85919587e+00, 2.82445709e+00, 2.81514854e+00, 2.75730138e+00, 2.73091926e+00, 2.64989702e+00, 2.63400660e+00, 2.59598070e+00, 2.51700469e+00, 2.47619303e+00, 2.45280929e+00, 2.41553300e+00, 2.39840596e+00, 2.37028429e+00, 2.36124375e+00, 2.31951290e+00, 2.26846869e+00, 2.25829953e+00, 2.23376422e+00, 2.22121302e+00, 2.16410727e+00, 2.14939140e+00, 2.13181827e+00, 2.10203566e+00, 2.07574345e+00, 2.04644311e+00, 2.02053356e+00, 2.01047443e+00, 1.98574345e+00, 1.96144311e+00, 1.93653356e+00, 1.91147443e+00, 1.88744345e+00, 1.86344311e+00, 1.83953356e+00, 1.81547443e+00, 1.79144345e+00, 1.76744311e+00, 1.74353356e+00, 1.71947443e+00, 1.69544345e+00, 1.67144311e+00, 1.64753356e+00, 1.62347443e+00, 1.60000000e+00, 1.57593356e+00, 1.55187443e+00, 1.52784345e+00, 1.50384311e+00, 1.47984356e+00, 1.45577443e+00, 1.43174345e+00, 1.40774311e+00, 1.38374356e+00, 1.35977443e+00, 1.33574345e+00, 1.31174311e+00, 1.28774356e+00, 1.26377443e+00, 1.23974345e+00, 1.21574311e+00, 1.19174356e+00, 1.16777443e+00, 1.14374345e+00, 1.11974311e+00, 1.09574356e+00, 1.07177443e+00, 1.04774345e+00, 1.02374311e+00, 1.00000000e+00, 9.7593356e-01, 9.5187443e-01, 9.2784345e-01, 9.0384311e-01, 8.7984356e-01, 8.5577443e-01, 8.3174345e-01, 8.0774311e-01, 7.8374356e-01, 7.5977443e-01, 7.3574345e-01, 7.1174311e-01, 6.8774356e-01, 6.6377443e-01, 6.3974345e-01, 6.1574311e-01, 5.9174356e-01, 5.6777443e-01, 5.4374345e-01, 5.1974311e-01, 4.9574356e-01, 4.7177443e-01, 4.4774345e-01, 4.2374311e-01, 4.00000000e+00, 3.7593356e-01, 3.5187443e-01, 3.2784345e-01, 3.0384311e-01, 2.7984356e-01, 2.5577443e-01, 2.3174345e-01, 2.0774311e-01, 1.8374356e-01, 1.5977443e-01, 1.3574345e-01, 1.1174311e-01, 8.774356e-02, 6.377443e-02, 3.974345e-02, 1.574345e-02, 0.00000000e+00, -1.574345e-02, -3.974345e-02, -6.377443e-02, -8.774356e-02, -1.1174311e-01, -1.3574345e-01, -1.5977443e-01, -1.8374356e-01, -2.0774311e-01, -2.3174345e-01, -2.5577443e-01, -2.7984311e-01, -3.0384356e-01, -3.2784345e-01, -3.5187443e-01, -3.7593356e-01, -4.00000000e+00, -4.2374311e-01, -4.4774345e-01, -4.7177443e-01, -4.9574345e-01, -5.1974311e-01, -5.4374356e-01, -5.6777443e-01, -5.9174345e-01, -6.1574311e-01, -6.3974356e-01, -6.6377443e-01, -6.8774345e-01, -7.1174311e-01, -7.3574356e-01, -7.5977443e-01, -7.8374345e-01, -8.0774311e-01, -8.3174356e-01, -8.5577443e-01, -8.7984345e-01, -9.0384311e-01, -9.2784356e-01, -9.5187443e-01, -9.7593356e-01, -1.00000000e+00, -1.02374345e+00, -1.04774311e+00, -1.07177443e+00, -1.09574345e+00, -1.11974311e+00, -1.14374356e+00, -1.16777443e+00, -1.19174345e+00, -1.21574311e+00, -1.23974356e+00, -1.26377443e+00, -1.28774345e+00, -1.31174311e+00, -1.33574356e+00, -1.35977443e+00, -1.38374345e+00, -1.40774311e+00, -1.43174356e+00, -1.45577443e+00, -1.47984345e+00, -1.50384311e+00, -1.52784356e+00, -1.55187443e+00, -1.57593356e+00, -1.60000000e+00, -1.62377443e+00, -1.64774345e+00, -1.67174311e+00, -1.69574356e+00, -1.71977443e+00, -1.74374345e+00, -1.76774311e+00, -1.79174356e+00, -1.81577443e+00, -1.83974345e+00, -1.86374311e+00, -1.88774356e+00, -1.91177443e+00, -1.93674345e+00, -1.96174311e+00, -1.98574356e+00, -2.01077443e+00, -2.03574345e+00, -2.06074311e+00, -2.08574356e+00, -2.11077443e+00, -2.13574345e+00, -2.16074311e+00, -2.18574356e+00, -2.21077443e+00, -2.23574345e+00, -2.26074311e+00, -2.28574356e+00, -2.31077443e+00, -2.33574345e+00, -2.36074311e+00, -2.38574356e+00, -2.41077443e+00, -2.43574345e+00, -2.46074311e+00, -2.48574356e+00, -2.51077443e+00, -2.53574345e+00, -2.56074311e+00, -2.58574356e+00, -2.61077443e+00, -2.63574345e+00, -2.66074311e+00, -2.68574356e+00, -2.71077443e+00, -2.73574345e+00, -2.76074311e+00, -2.78574356e+00, -2.81077443e+00, -2.83574345e+00, -2.86074311e+00, -2.88574356e+00, -2.91077443e+00, -2.93574345e+00, -2.96074311e+00, -2.98574356e+00, -3.01077443e+00, -3.03574345e+00, -3.06074311e+00, -3.08574356e+00, -3.11077443e+00, -3.13574345e+00, -3.16074311e+00, -3.18574356e+00, -3.21077443e+00, -3.23574345e+00, -3.26074311e+00, -3.28574356e+00, -3.31077443e+00, -3.33574345e+00, -3.36074311e+00, -3.38574356e+00, -3.41077443e+00, -3.43574345e+00, -3.46074311e+00, -3.48574356e+00, -3.51077443e+00, -3.53574345e+00, -3.56074311e+00, -3.58574356e+00, -3.61077443e+00, -3.63574345e+00, -3.66074311e+00, -3.68574356e+00, -3.71077443e+00, -3.73574345e+00, -3.76074311e+00, -3.78574356e+00, -3.81077443e+00, -3.83574345e+00, -3.86074311e+00, -3.88574356e+00, -3.91077443e+00, -3.93574345e+00, -3.96074311e+00, -3.98574356e+00, -4.01077443e+00, -4.03574345e+00, -4.06074311e+00, -4.08574356e+00, -4.11077443e+00, -4.13574345e+00, -4.16074311e+00, -4.18574356e+00, -4.21077443e+00, -4.23574345e+00, -4.26074311e+00, -4.28574356e+00, -4.31077443e+00, -4.33574345e+00, -4.36074311e+00, -4.38574356e+00, -4.41077443e+00, -4.43574345e+00, -4.46074311e+00, -4.48574356e+00, -4.51077443e+00, -4.53574345e+00, -4.56074311e+00, -4.58574356e+00, -4.61077443e+00, -4.63574345e+00, -4.66074311e+00, -4.68574356e+00, -4.71077443e+00, -4.73574345e+00, -4.76074311e+00, -4.78574356e+00, -4.81077443e+00, -4.83574345e+00, -4.86074311e+00, -4.88574356e+00, -4.91077443e+00, -4.93574345e+00, -4.96074311e+00, -4.98574356e+00, -5.01077443e+00, -5.03574345e+00, -5.06074311e+00, -5.08574356e+00, -5.11077443e+00, -5.13574345e+00, -5.16074311e+00, -5.18574356e+00, -5.21077443e+00, -5.23574345e+00, -5.26074311e+00, -5.28574356e+00, -5.31077443e+00, -5.33574345e+00, -5.36074311e+00, -5.38574356e+00, -5.41077443e+00, -5.43574345e+00, -5.46074311e+00, -5.48574356e+00, -5.51077443e+00, -5.53574345e+00, -5.56074311e+00, -5.58574356e+00, -5.61077443e+00, -5.63574345e+00, -5.66074311e+00, -5.68574356e+00, -5.71077443e+00, -5.73574345e+00, -5.76074311e+00, -5.78574356e+00, -5.81077443e+00, -5.83574345e+00, -5.86074311e+00, -5.88574356e+00, -5.91077443e+00, -5.93574345e+00, -5.96074311e+00, -5.98574356e+00, -6.01077443e+00, -6.03574345e+00, -6.06074311e+00, -6.08574356e+00, -6.11077443e+00, -6.13574345e+00, -6.16074311e+00, -6.18574356e+00, -6.21077443e+00, -6.23574345e+00, -6.26074311e+00, -6.28574356e+00, -6.31077443e+00, -6.33574345e+00, -6.36074311e+00, -6.38574356e+00, -6.41077443e+00, -6.43574345e+00, -6.46074311e+00, -6.48574356e+00, -6.51077443e+00, -6.53574345e+00, -6.56074311e+00, -6.58574356e+00, -6.61077443e+00, -6.63574345e+00, -6.66074311e+00, -6.68574356e+00, -6.71077443e+00, -6.73574345e+00, -6.76074311e+00, -6.78574356e+00, -6.81077443e+00, -6.83574345e+00, -6.86074311e+00, -6.88574356e+00, -6.91077443e+00, -6.93574345e+00, -6.96074311e+00, -6.98574356e+00, -7.01077443e+00, -7.03574345e+00, -7.06074311e+00, -7.08574356e+00, -7.11077443e+00, -7.13574345e+00, -7.16074311e+00, -7.18574356e+00, -7.21077443e+00, -7.23574345e+00, -7.26074311e+00, -7.28574356e+00, -7.31077443e+00, -7.33574345e+00, -7.36074311e+00, -7.38574356e+00, -7.41077443e+00, -7.43574345e+00, -7.46074311e+00, -7.48574356e+00, -7.51077443e+00, -7.53574345e+00, -7.56074311e+00, -7.58574356e+00, -7.61077443e+00, -7.63574345e+00, -7.66074311e+00, -7.68574356e+00, -7.71077443e+00, -7.73574345e+00, -7.76074311e+00, -7.78574356e+00, -7.81077443e+00, -7.83574345e+00, -7.86074311e+00, -7.88574356e+00, -7.91077443e+00, -7.93574345e+00, -7.96074311e+00, -7.98574356e+00, -8.01077443e+00, -8.03574345e+00, -8.06074311e+00, -8.08574356e+00, -8.11077443e+00, -8.13574345e+00, -8.16074311e+00, -8.18574356e+00, -8.21077443e+00, -8.23574345e+00, -8.26074311e+00, -8.28574356e+00, -8.31077443e+00, -8.33574345e+00, -8.36074311e+00, -8.38574356e+00, -8.41077443e+00, -8.43574345e+00, -8.46074311e+00, -8.48574356e+00, -8.51077443e+00, -8.53574345e+00, -8.56074311e+00, -8.58574356e+00, -8.61077443e+00, -8.63574345e+00, -8.66074311e+00, -8.68574356e+00, -8.71077443e+00, -8.73574345e+00, -8.76074311e+00, -8.78574356e+00, -8.81077443e+00, -8.83574345e+00, -8.86074311e+00, -8.88574356e+00, -8.91077443e+00, -8.93574345e+00, -8.96074311e+00, -8.98574356e+00, -9.01077443e+00, -9.03574345e+00, -9.06074311e+00, -9.08574356e+00, -9.11077443e+00, -9.13574345e+00, -9.16074311e+00, -9.18574356e+00, -9.21077443e+00, -9.23574345e+00, -9.26074311e+00, -9.28574356e+00, -9.31077443e+00, -9.33574345e+00, -9.36074311e+00, -9.38574356e+00, -9.41077443e+00, -9.43574345e+00, -9.46074311e+00, -9.48574356e+00, -9.51077443e+00, -9.53574345e+00, -9.56074311e+00, -9.58574356e+00, -9.61077443e+00, -9.63574345e+00, -9.66074311e+00, -9.68574356e+00, -9.71077443e+00, -9.73574345e+00, -9.76074311e+00, -9.78574356e+00, -9.81077443e+00, -9.83574345e+00, -9.86074311e+00, -9.88574356e+00, -9.91077443e+00, -9.93574345e+00, -9.96074311e+00, -9.98574356e+00, -1.00000000e+00])`

In [56]: `pca.explained_variance_.shape`

Out[56]: `(784,)`

In [57]: `pca.components_`

Out[57]: `array([[ 1.00000000e+00, -7.63006414e-17, -2.03468377e-16, ..., -1.81732303e-33, 9.81673812e-34, 3.96851063e-33], [ 7.63006414e-17, 1.00000000e+00, 2.60902411e-14, ..., -8.16664931e-34, -1.25502863e-33, -5.53621059e-34], [ 2.03468377e-16, -2.59792188e-14, 1.00000000e+00, ..., 2.43949246e-33, -3.61236853e-33, -1.47012382e-32], ..., [ 0.00000000e+00, 9.17467049e-34, 1.94880351e-34, ..., -2.80756515e-04, -3.87673848e-05, -1.91814163e-04], [ 0.00000000e+00, 1.18646442e-33, 6.55402589e-36, ..., -1.20620742e-06, -1.91319546e-05, -5.17030919e-06], [-0.00000000e+00, -1.61858077e-32, 1.51045001e-32, ..., 6.35936057e-02, -3.89132900e-02, -1.67578354e-01]])`

```
In [58]: pca.components_.shape
```

```
Out[58]: (784, 784)
```

```
In [59]: pca.explained_variance_ratio_
```

```
Out[59]: array([5.76942060e-02, 4.11699420e-02, 3.78000066e-02, 2.94885662e-02,
                2.57134926e-02, 2.23758526e-02, 1.95663220e-02, 1.78113713e-02,
                1.56178063e-02, 1.42405956e-02, 1.36703680e-02, 1.22610776e-02,
                1.13552994e-02, 1.12042093e-02, 1.05318708e-02, 1.00567491e-02,
                9.59719123e-03, 9.38882015e-03, 9.06715738e-03, 8.87037435e-03,
                8.43180774e-03, 8.17752898e-03, 7.74642653e-03, 7.55903659e-03,
                7.30536874e-03, 7.02793045e-03, 6.93013790e-03, 6.67939737e-03,
                6.37213232e-03, 6.16741281e-03, 6.13423824e-03, 5.96396716e-03,
                5.81822798e-03, 5.73620427e-03, 5.63790419e-03, 5.50524015e-03,
                5.44197257e-03, 5.26046367e-03, 5.04807223e-03, 4.96396985e-03,
                4.85592567e-03, 4.78812855e-03, 4.68877583e-03, 4.64500003e-03,
                4.53802181e-03, 4.50311055e-03, 4.45585484e-03, 4.40050096e-03,
                4.36283204e-03, 4.31378787e-03, 4.21240717e-03, 4.15057749e-03,
                4.04973197e-03, 4.00052837e-03, 3.98734385e-03, 3.90540980e-03,
                3.86804247e-03, 3.75328350e-03, 3.73077650e-03, 3.67691705e-03,
                3.56505635e-03, 3.50725118e-03, 3.47413072e-03, 3.42133302e-03,
                3.39707447e-03, 3.35724326e-03, 3.34443835e-03, 3.28533126e-03,
                3.21303283e-03, 3.19862935e-03, 3.16387782e-03, 3.14610045e-03,
                3.06521652e-03, 3.04437312e-03, 3.01948276e-03, 2.97729901e-03,
                2.94005002e-03, 2.88855827e-03, 2.87378627e-03, 2.86036022e-03,
```

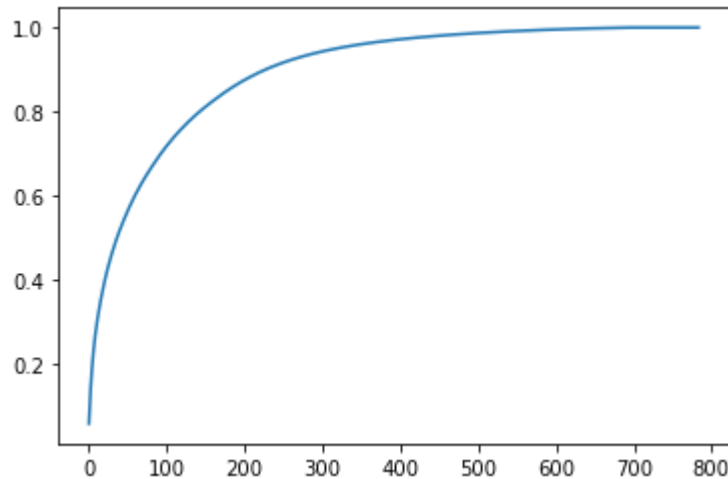
```
In [60]: ## Calculate the cumulative explained variance ratio
```

```
np.cumsum(pca.explained_variance_ratio_)
```

```
Out[60]: array([0.05769421, 0.09886415, 0.13666415, 0.16615272, 0.19186621,
0.21424207, 0.23380839, 0.25161976, 0.26723757, 0.28147816,
0.29514853, 0.30740961, 0.31876491, 0.32996912, 0.34050099,
0.35055774, 0.36015493, 0.36954375, 0.3786109 , 0.38748128,
0.39591309, 0.40409062, 0.41183704, 0.41939608, 0.42670145,
0.43372938, 0.44065952, 0.44733891, 0.45371105, 0.45987846,
0.4660127 , 0.47197666, 0.47779489, 0.4835311 , 0.489169 ,
0.49467424, 0.50011621, 0.50537668, 0.51042475, 0.51538872,
0.52024464, 0.52503277, 0.52972155, 0.53436655, 0.53890457,
0.54340768, 0.54786354, 0.55226404, 0.55662687, 0.56094066,
0.56515306, 0.56930364, 0.57335337, 0.5773539 , 0.58134125,
0.58524666, 0.5891147 , 0.59286798, 0.59659876, 0.60027567,
0.60384073, 0.60734798, 0.61082211, 0.61424345, 0.61764052,
0.62099776, 0.6243422 , 0.62762753, 0.63084057, 0.6340392 ,
0.63720307, 0.64034917, 0.64341439, 0.64645876, 0.64947825,
0.65245555, 0.6553956 , 0.65829416, 0.66116795, 0.66402831,
0.66685254, 0.66967375, 0.67247454, 0.67526914, 0.67804418,
0.68081054, 0.68353216, 0.68623319, 0.68888826, 0.69150838,
0.69412352, 0.69672185, 0.69928988, 0.70184487, 0.70437135,
0.70688134, 0.70932273, 0.71177502, 0.71411816, 0.71655272,
0.71887825, 0.72118516, 0.72347297, 0.72574127, 0.72798957,
0.73021837, 0.73242717, 0.73461557, 0.73678317, 0.73892957,
0.74105437, 0.74315817, 0.74524057, 0.74730117, 0.74934057,
0.75135837, 0.75334517, 0.75530957, 0.75725117, 0.75917057,
0.76106757, 0.76294177, 0.76479277, 0.76661917, 0.76842157,
0.77020057, 0.77195677, 0.77368957, 0.77539957, 0.77708657,
0.77875017, 0.78039017, 0.78200617, 0.78359817, 0.78516577,
0.78670857, 0.78822717, 0.78972117, 0.79119117, 0.79263697,
0.79405817, 0.79545537, 0.79682817, 0.79817717, 0.79950197,
0.80080317, 0.80208037, 0.80333317, 0.80456217, 0.80576717,
0.80694817, 0.80810497, 0.80923717, 0.81034537, 0.81142917,
0.81248817, 0.81352217, 0.81453117, 0.81551497, 0.81647417,
0.81740857, 0.81831817, 0.81920257, 0.82006157, 0.82089577,
0.82170497, 0.82248917, 0.82324817, 0.82398157, 0.82468997,
0.82537317, 0.82603157, 0.82666457, 0.82727257, 0.82785557,
0.82841317, 0.82894517, 0.82945117, 0.82993117, 0.83038517,
0.83081257, 0.83121317, 0.83158757, 0.83193517, 0.83225557,
0.83254857, 0.83281357, 0.83305117, 0.83326197, 0.83344557,
0.83360257, 0.83373157, 0.83383257, 0.83390617, 0.83395257,
0.83397157, 0.83396317, 0.83392757, 0.83386457, 0.83377457,
0.83365817, 0.83351517, 0.83334617, 0.83315117, 0.83292957,
0.83268217, 0.83241857, 0.83213857, 0.83184257, 0.83153057,
0.83119317, 0.83083057, 0.83044317, 0.83003117, 0.82959457,
0.82913317, 0.82864657, 0.82813457, 0.82759757, 0.82703557,
0.82644857, 0.82583657, 0.82519957, 0.82453757, 0.82385057,
0.82313857, 0.82240157, 0.82163957, 0.82085257, 0.82004057,
0.81920357, 0.81834157, 0.81745457, 0.81654257, 0.81560557,
0.81464357, 0.81365657, 0.81264457, 0.81160757, 0.81054557,
0.80945857, 0.80834657, 0.80720957, 0.80604757, 0.80486057,
0.80364857, 0.80241157, 0.80114957, 0.79986257, 0.79855057,
0.79721357, 0.79585157, 0.79446457, 0.79305257, 0.79161557,
0.79015357, 0.78866657, 0.78715457, 0.78561757, 0.78405557,
0.78246857, 0.78085657, 0.77921957, 0.77755757, 0.77587057,
0.77415857, 0.77242157, 0.77065957, 0.76887257, 0.76706057,
0.76522357, 0.76336157, 0.76147457, 0.75956257, 0.75762557,
0.75566357, 0.75367657, 0.75166457, 0.74962757, 0.74756557,
0.74547857, 0.74336657, 0.74122957, 0.73906757, 0.73688057,
0.73466857, 0.73243157, 0.73016957, 0.72789257, 0.72559057,
0.72326357, 0.72091157, 0.71853457, 0.71613257, 0.71370557,
0.71125357, 0.70877657, 0.70627457, 0.70374757, 0.70119557,
0.69861857, 0.69601657, 0.69338957, 0.69073757, 0.68806057,
0.68535857, 0.68263157, 0.67987957, 0.67710257, 0.67429957,
0.67147157, 0.66861857, 0.66574057, 0.66283757, 0.65990057,
0.65693857, 0.65395157, 0.65093957, 0.64790257, 0.64484057,
0.64175357, 0.63864157, 0.63550457, 0.63234257, 0.62915557,
0.62594357, 0.62270657, 0.61944457, 0.61615757, 0.61284557,
0.60950857, 0.60614657, 0.60275957, 0.59934757, 0.59591057,
0.59244857, 0.58896157, 0.58544957, 0.58191257, 0.57835057,
0.57476357, 0.571
```

```
In [61]: plt.plot(np.cumsum(pca.explained_variance_ratio_))
```

```
Out[61]: [<matplotlib.lines.Line2D at 0x21ff2009e80>]
```



```
In [62]: cumulative_variance_ratio = np.cumsum(pca.explained_variance_ratio_)
```

```
In [63]: # Find the number of components that capture a significant amount of variance (e.g., 95%)
```

```
optimal_num_components = np.argmax(cumulative_variance_ratio >= 0.95) + 1
```

```
print(f"Optimal Number of Components: {optimal_num_components}")
```

Optimal Number of Components: 319

## There are a few reasons why PCA might not work.

Here are some of the most common reasons:

- **The data is not normally distributed.** PCA assumes that the data is normally distributed. If the data is not normally distributed, PCA may not be able to find the principal components that explain the most variation in the data.

- **The data is not linearly correlated.** PCA works by finding the principal components that are the linear combinations of the original features that explain the most variation in the data. If the data is not linearly correlated, PCA may not be able to find any principal components that explain a significant amount of variation in the data.
- **The data has too many features.** PCA can only find a limited number of principal components. If the data has too many features, PCA may not be able to find enough principal components to explain a significant amount of variation in the data.
- **The data is not standardized.** PCA assumes that the features have been standardized before it is applied. If the features have not been standardized, PCA may not be able to find the principal components that explain the most variation in the data.

If you are not sure if PCA is working, you can try the following:

- **Check the distribution of the data.** If the data is not normally distributed, you can try transforming the data to make it more normally distributed.
- **Check the correlation between the features.** If the data is not linearly correlated, you can try transforming the data to make it more linearly correlated.
- **Reduce the number of features.** If the data has too many features, you can try reducing the number of features using a technique like feature selection.

If you have tried all of these things and PCA is still not working, it may be that PCA is not the right tool for the job. In this case, you may need to try a different dimensionality reduction technique like LDA or t-SNE.

In [ ]: