Principal Component Analysis(PCA)

`from sklearn. decomposition import PCA`

`pca = PCA( n_components=2 )`

`n_components` → Default: No. of columns

> 💡 **It's Unsupervised ML Problem**

- You only have input.
- There's no output.

Overview

- **Purpose**: Dimensionality reduction technique to **transform high-dimensional data into a lower-dimensional space** while retaining most of the variance.
  - 10D → 2D or 3D..so that you can plot its graph
- **Use Case**: Reduces the number of features, removes multicollinearity, and improves model performance.

How Does PCA Work?

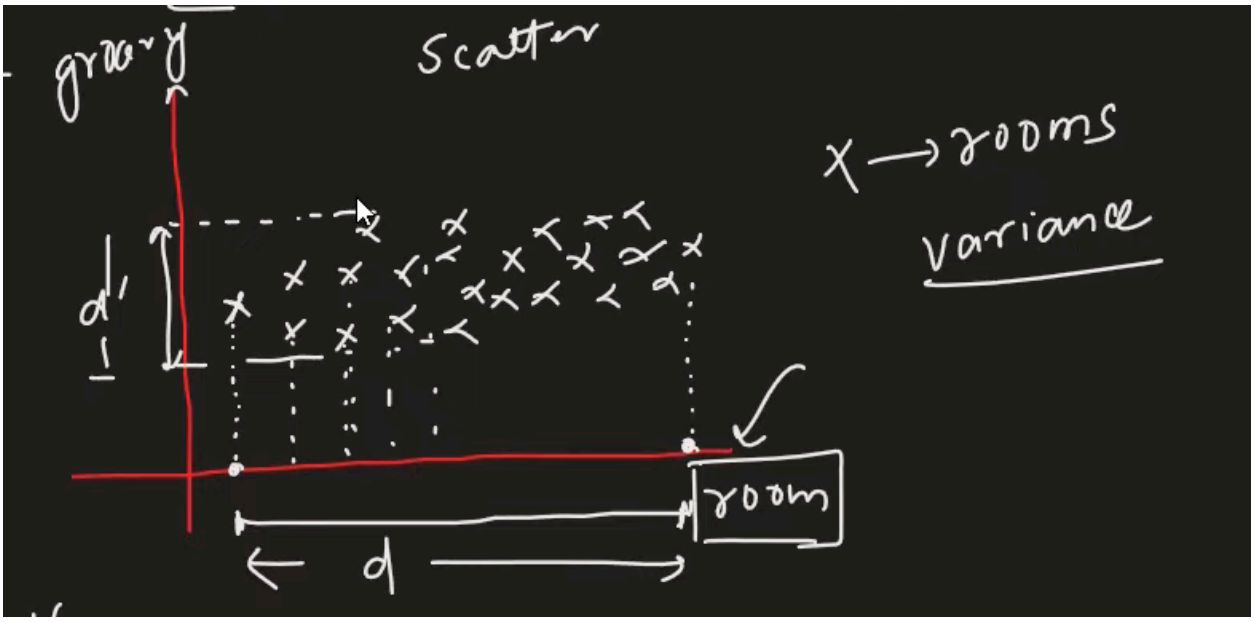
- **Step 1: Standardization**
  - Data is standardized so that each feature has a mean of 0 and a standard deviation of 1. This ensures that all features contribute equally.
- **Step 2: Covariance Matrix Computation**

    - PCA calculates the covariance matrix to understand how variables relate to one another.
- **Step 3: Eigen Decomposition**
    - The eigenvectors (principal components) and eigenvalues of the covariance matrix are computed.
    - **Eigenvectors:** Define the directions of maximum variance.
    - **Eigenvalues:** Indicate the amount of variance captured by each eigenvector.
- **Step 4: Feature Transformation**
    - The original data is projected onto the top k eigenvectors to obtain a lower-dimensional representation.

Why Use PCA?

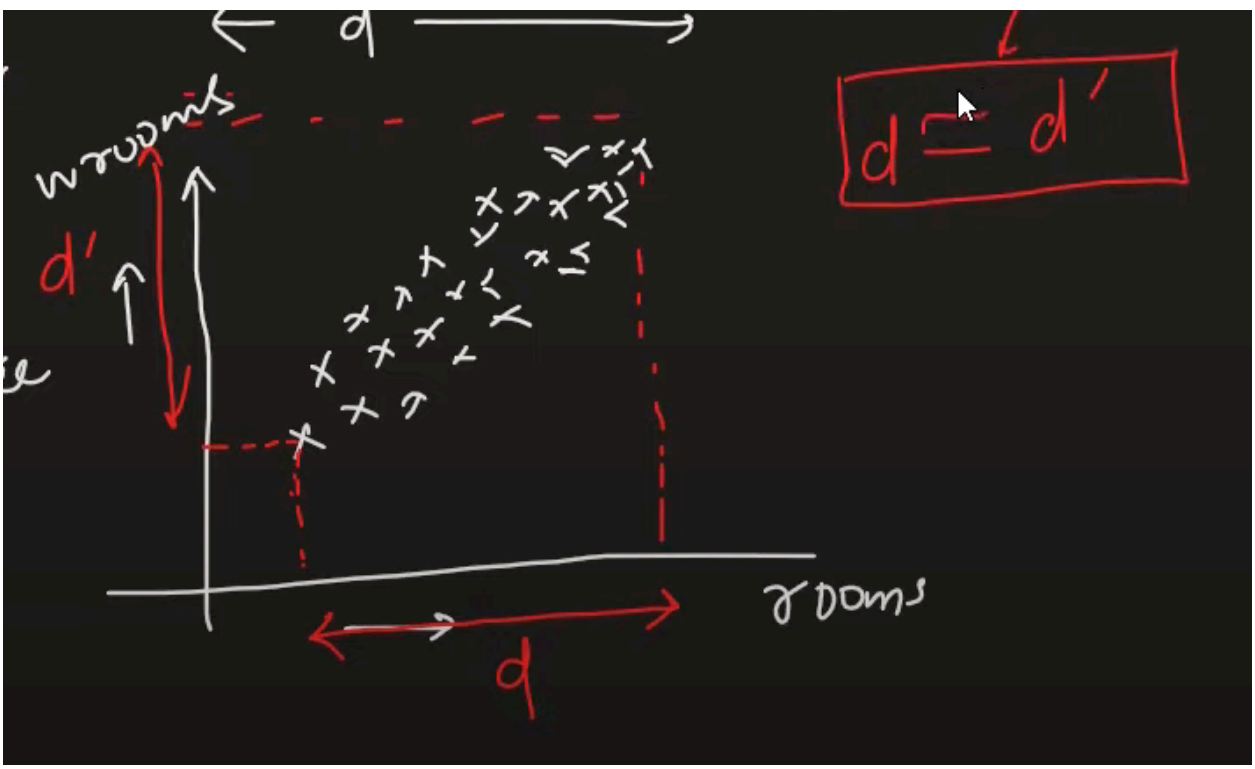
- **Dimensionality Reduction:**Simplifies datasets by reducing the number of features while retaining most of the variance.
- **Noise Reduction:**Filters out noise by keeping only the components with the most significant variance.
- **Visualization:**Facilitates visualization of high-dimensional data in 2D or 3D plots.
- Faster execution of algorithms

Rooms vs Grocery shops 👇



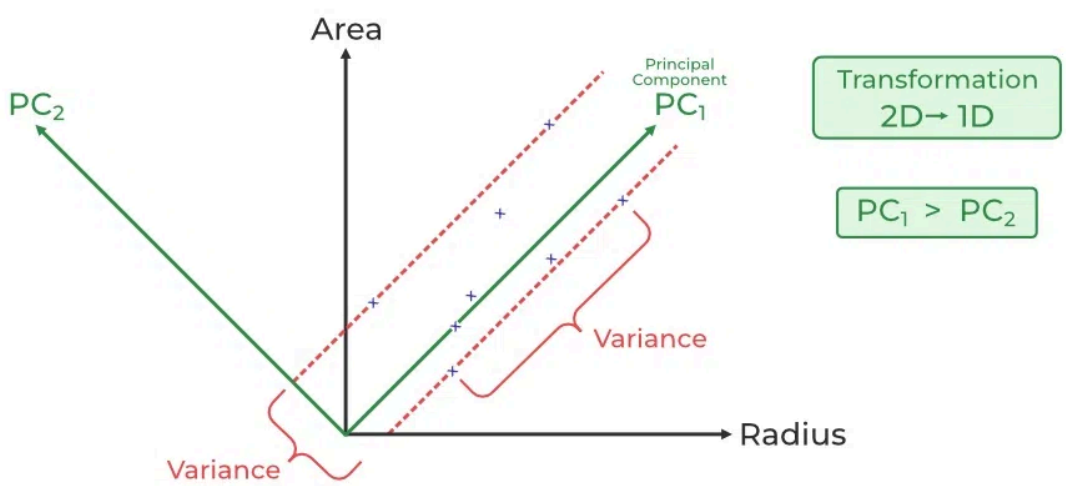
- In above example, we'll select room against grocery shop because it has more variance.

Roomvs vs Washrooms:



- Here, the variance of both are similar.
- So, it's difficult to select one column (feature).
- **Feature extraction** helps in such case.
- Solution for the above👆 problem could be:
  - We can combine **Rooms + Washroom as → Size of Flat**
  - Now it's 1D data
- **PCA forgets about the existing features.**
- It creates a new set of features from the existing features
- & chooses a subset from the new set of features which it thinks is most important.

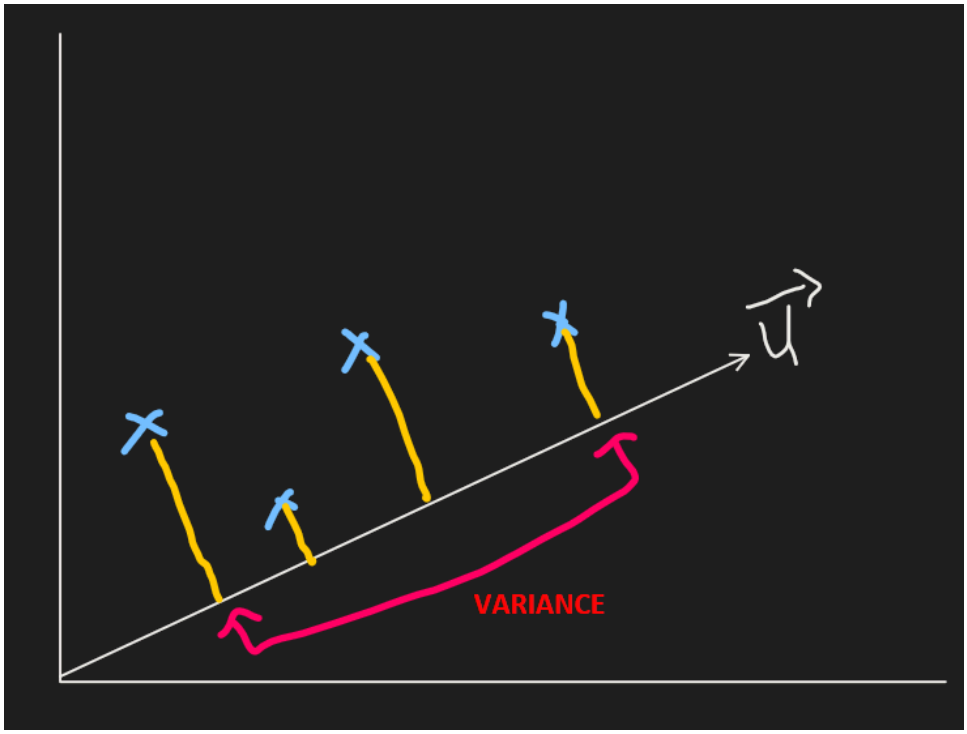
- PCA shifts the axes and gives you new principal components 👇



- As variance of PC1 is greater than Variance of PC2, **PCA will keep PC1**
- We'll transform the data as per PC1

> **No. of Principal Components ≤ No. of Columns (Features)**

💡 **Aim: To maximise the variance when we project the points on a unit vector.**



- Variance tells you the relationship between only 2 variables.
- Covariance & Covariance Matrix solves this problem

Covariance & Covariance Matrix

`.cov()`

- **Covariance** is a measure of the relationship between two random variables.
- It indicates how much two random variables change together.

- If the variables tend to increase or decrease together, the covariance is **positive**.
- If one increases while the other decreases, the covariance is **negative**.
- If the variables are independent, the covariance is **zero**.

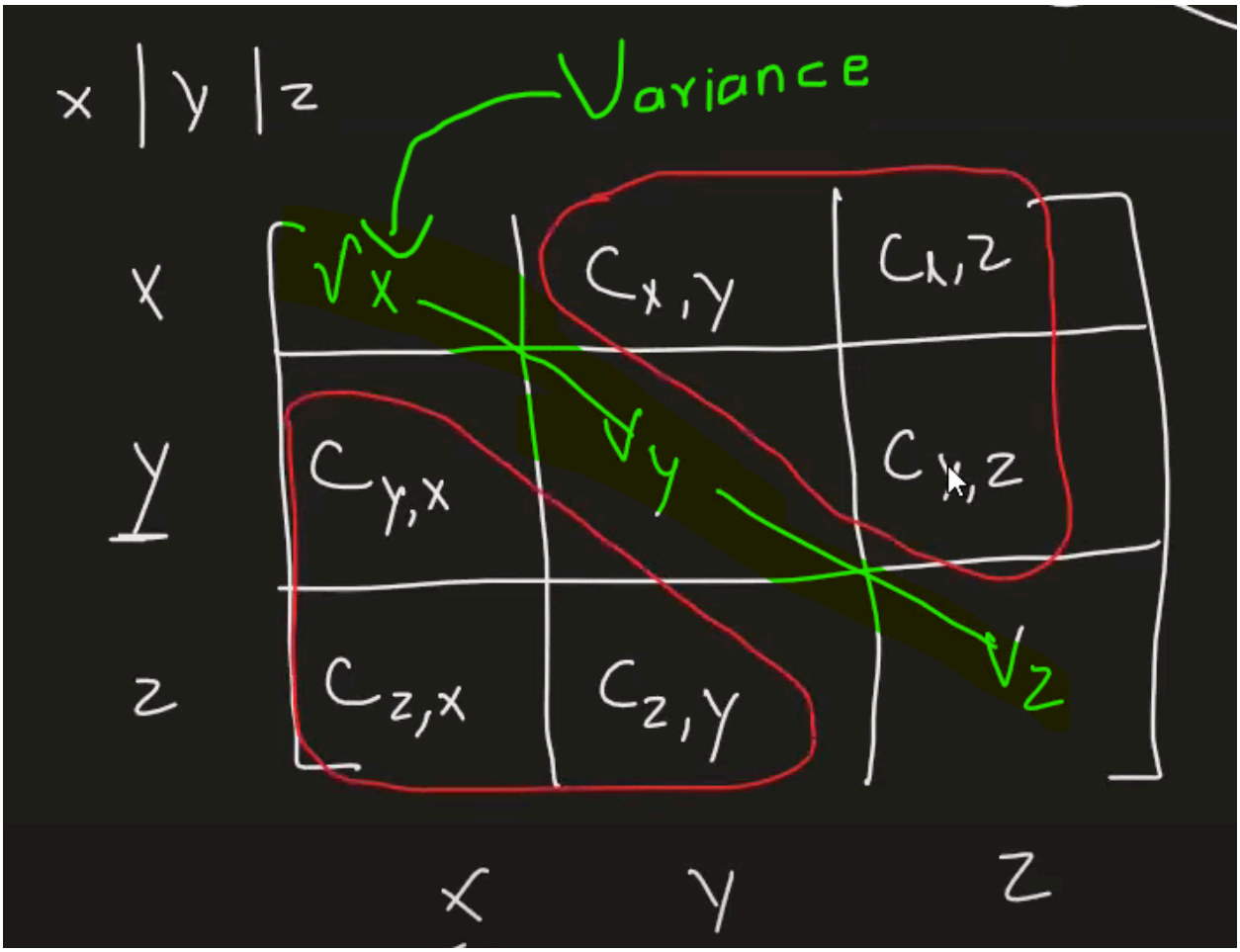
$$\mathrm{Cov}(X, Y) = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})$$

Where:

- X_i and Y_i are the individual data points in each of the variables X and Y,
- $\bar{X}$ and $\bar{Y}$ are the means of X and Y,
- n is the number of data points.

Covariance Matrix

- A **Covariance Matrix** is a square matrix that contains the covariances between pairs of variables in a multivariate dataset.
- If we have multiple variables, the covariance matrix generalizes the covariance concept to multiple dimensions.



💡 **Diagonal columns are variance(spread) of the column.**

Covariance vs Correlation

- Covariance is unnormalized and can take any value between −∞ and +∞.
- Correlation is the normalized version of covariance. It ranges between − 1 and +1.

Aspect	Covariance	Correlation
Scale	Unnormalized	Normalized (between −1 and +1)
Units	Has units of the product of the two variables	Unitless (no dependence on the scale of variables)

Aspect	Covariance	Correlation
Interpretation	Direction of relationship (positive/negative)	Direction and strength of relationship
Range	$-\infty$ to $+\infty$.	-1 to $+1$

Eigen Decomposition of Covariance Matrix

- **Eigen Decomposition**: A process of decomposing a square matrix into its eigenvalues and eigenvectors.

In the Context of PCA:

- You start with a covariance matrix (a square matrix that shows how features vary together).

Eigenvectors:

- Represent the directions (principal components) in the feature space.
- Orthogonal to each other in the case of a symmetric matrix (e.g., covariance matrix).
- **Upon applying transformation, their direction does not change**

Eigenvalues:

- Represent the amount of variance captured by each eigenvector.
- Larger eigenvalues correspond to directions (eigenvectors) with higher variance.

$$A\mathbf{v} = \lambda\mathbf{v}$$

💡 👆 **Meaning: Applying a linear transformation A to a vector v = Multiplying the vector by a scalar.**

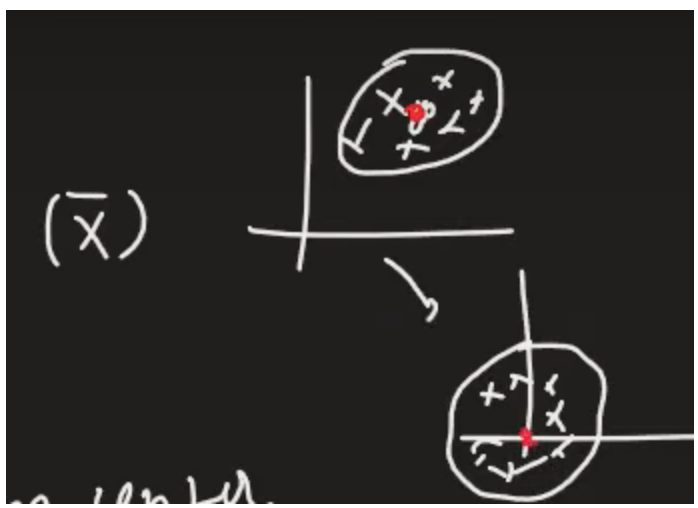
> The Variance is Highest on the Biggest **Eigenvector.**

Visualize metrics transformation → https://www.geogebra.org/m/YCZa8TAH

Steps for Eigen Decomposition

First you do the mean centering of data:

- This is not mandatory step but by doing this, the performance of PCA increases.



1. **Compute the Covariance Matrix**:
    - For a dataset X with *n* samples and *p* features, the covariance matrix

$$\Sigma = \frac{1}{n-1} X^T X$$

2. **Decompose the Covariance Matrix**:
   - Find eigenvalues (λ) and eigenvectors (v) such that:

     $\Sigma v = \lambda v$
3. **Sort Eigenvalues and Eigenvectors**:
   - Sort eigenvalues in descending order and reorder eigenvectors accordingly.
4. **Select Principal Components**:
   - Choose the top k eigenvectors (principal components) corresponding to the largest eigenvalues.
5. Transform the points to lower dimension.

$$u^T . x$$

- You can either go 3D → 2D
- OR 3D → 1D , etc.

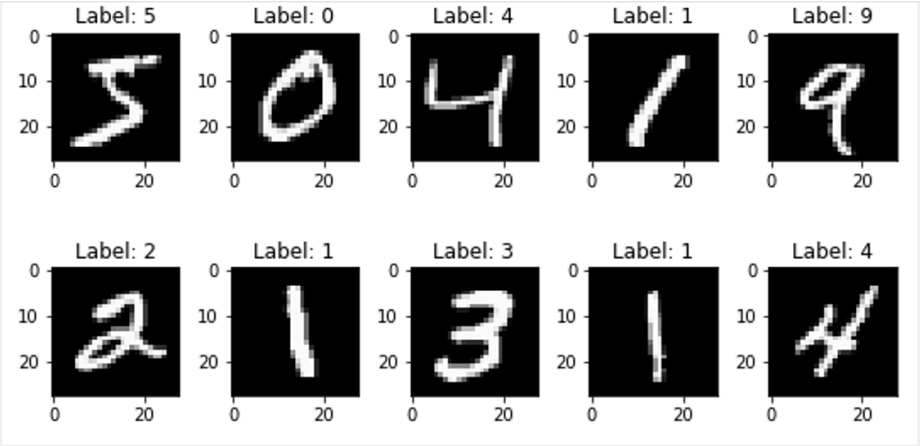
Python Code

- We'll use `digit-recognizer` dataset

```
df = pd.read_csv(r'https://raw.githubusercontent.com/G1Codes/digit-recognizer/refs/heads/main/digit-recognizer_train.csv')

df.shape
```

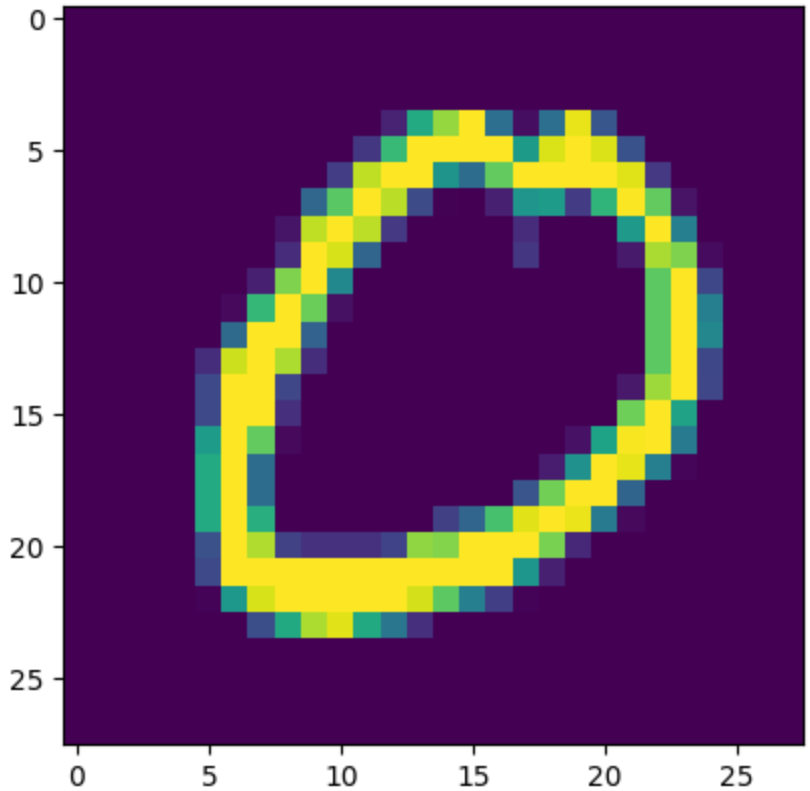
```
Output:
(42000, 785)
```



Samples from the dataset. These are handwritten digits.
Our goal is to train an Artificial Neural Network to be able to receive such an image and decide which digit the image represents.

- Each Image consists of 28×28 pixels
- Total 784 pixels = Columns
- There are 42,000 images
- **Label: Output → Digit**

```
plt.imshow(df.iloc[13051,1:].values.reshape(28,28))
```



- `13051` specifies the row index (this will select the row at index 13051).
- `1:` specifies all columns starting from the second column to the last one. The `1:` s

So,
`df.iloc[13051, 1:]` gets all the pixel values (assuming each column after the first represents a pixel in an image).

`.values` converts the row selected by `iloc` into a → **NumPy array**. This makes it easier to manipulate the data for plotting.

`.reshape(28, 28)` :

- The image data, after being selected from the DataFrame, is likely a flat array (a vector) of 784 values, each representing a pixel.
- The `.reshape(28, 28)` function is used to convert this flat array of 784 values into a 28×28 matrix, which is the shape of the image (for example, in the MNIST dataset, each image is 28 pixels by 28 pixels)

`plt.imshow` is a function from the Matplotlib library that displays an image.

- It assigns a color to a number.

Cool sh!t you can do with `imshow` → https://youtu.be/1eYfoCtMmPY?si=pUGokAkV-JxKvOTe

Run a Machine Learning algo on this data:

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

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,random_state=42)

X_train.shape

Output:
(33600, 784)
```

KNN:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(X_train,y_train)
```

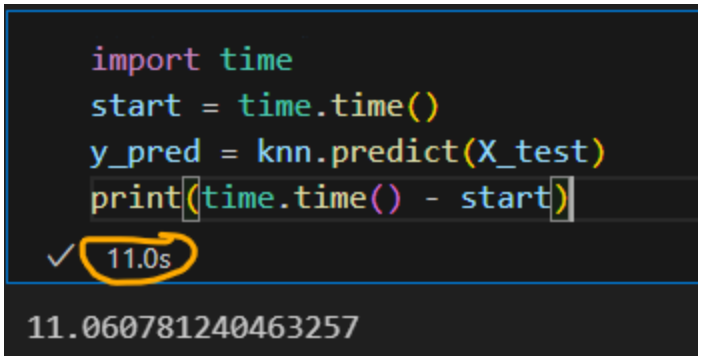
- We used the default settings
- ***By*** default: `n_neighbors=5`

- **It will take a lot of time because there are 784 columns.**

```
import time
start = time.time()
y_pred = knn.predict(X_test)
print(time.time() - start)

Output:
11.060781240463257
```

- It tool 11 sec to run



- VS Code shows time by default.

Calculate the accuracy:

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test,y_pred)

Output:
0.9648809523809524
```

- 96% Accuracy

We'll use use PCA and try to achieve the same accuracy

- First, we have to **standardise** the data

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Apply PCA

```
# PCA
from sklearn.decomposition import PCA

pca = PCA(n_components=200)

X_train_trf = pca.fit_transform(X_train)
X_test_trf = pca.transform(X_test)

X_train_trf.shape

Output:
(33600, 200)
```

- We got 200 columns

Now apply KNN again on this PCA data:

```
knn = KNeighborsClassifier()

knn.fit(X_train_trf,y_train)

y_pred = knn.predict(X_test_trf)
```

```
accuracy_score(y_test,y_pred)

Output:
0.9507142857142857
```

- 👆 **Took 0.95 sec to run against 11 for 784 columns**
- **Accuracy ois 95.07 % compared to 96.48% with all features**

Check the best number of features for KNN:

```
for i in range(1,785):
    pcalist=[]
    pca = PCA(n_components=i)
    X_train_trf = pca.fit_transform(X_train)
    X_test_trf = pca.transform(X_test)

    knn = KNeighborsClassifier(n_jobs=-1)

    knn.fit(X_train_trf,y_train)

    y_pred = knn.predict(X_test_trf)

    pcalist.append(print(i, accuracy_score(y_test,y_pred)))
```

- Result: It went till 95+ on ~ 100 components.
- After that, the accuracy score started to decrease a bit.

NOTE: `n_jobs=-1` **is extremely important as without it, in 13+ min printed ~330 results.**

```
1 0.2580952380952381
2 0.3236904761904762
3 0.5104761904761905
4 0.6663095238095238
5 0.7378571428571429
6 0.8227380952380953
7 0.8436904761904762
8 0.8721428571428571
9 0.886547619047619
10 0.9055952380952381
11 0.9116666666666666
12 0.9184523809523809
13 0.9276190476190476
14 0.9351190476190476
15 0.9384523809523809
16 0.9379761904761905
17 0.9407142857142857
18 0.94
19 0.9425
20 0.9442857142857143
21 0.9438095238095238
22 0.9441666666666667
23 0.9442857142857143
24 0.9457142857142857
25 0.9484523809523809
```

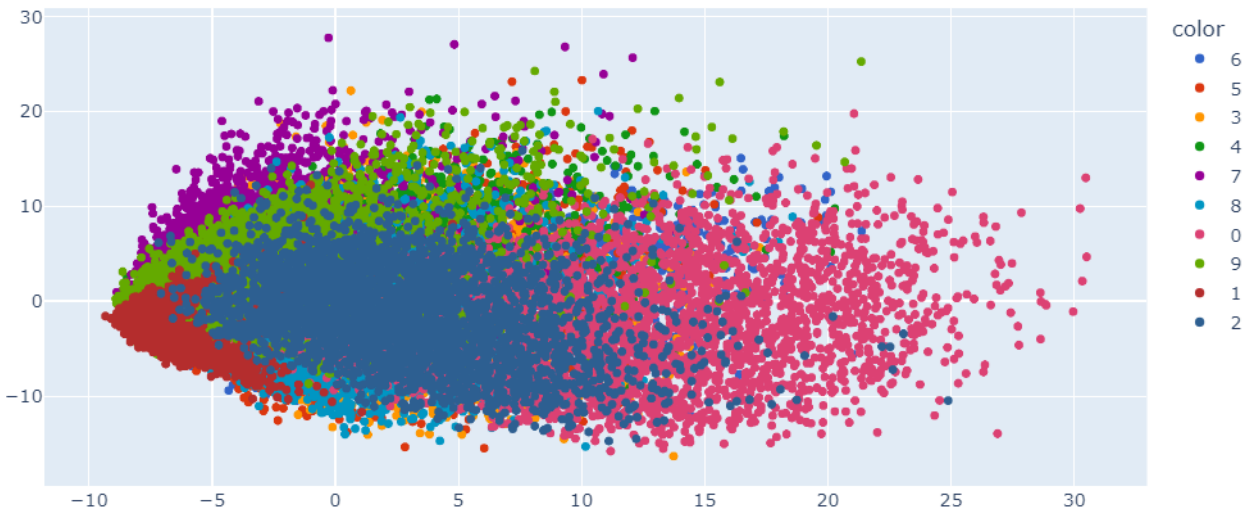
Now use PCA using 2 components

```
# transforming to a 2D coordinate system
pca = PCA(n_components=2)
X_train_trf = pca.fit_transform(X_train)
X_test_trf = pca.transform(X_test)

X_train_trf
```

Plot a scatterplot using plotly:

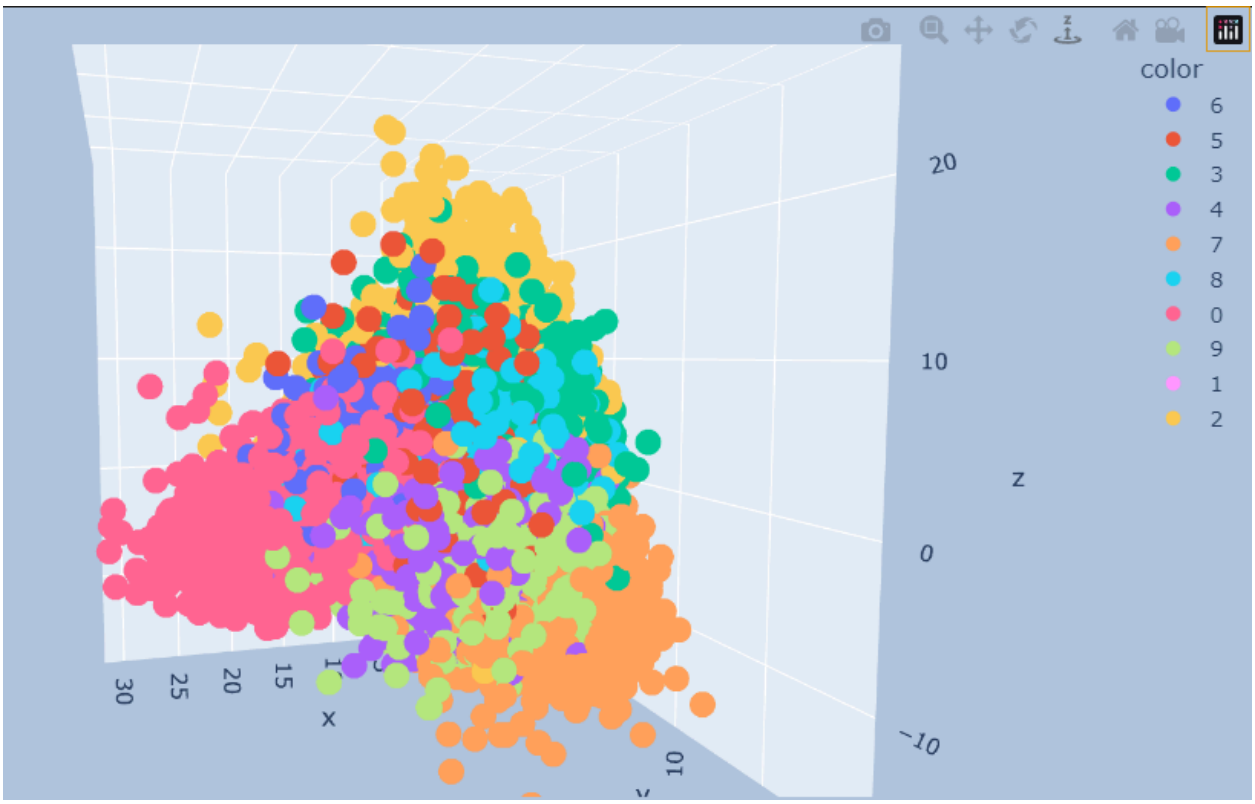
```
import plotly.express as px
y_train_trf = y_train.astype(str)
fig = px.scatter(x=X_train_trf[:,0],
                 y=X_train_trf[:,1],
                 color=y_train_trf,
                 color_discrete_sequence=px.colors.qualitative.G10
                )
fig.show()
```



Do the same in 3D:

```
# transforming in 3D
pca = PCA(n_components=3)
X_train_trf = pca.fit_transform(X_train)
X_test_trf = pca.transform(X_test)
X_train_trf
```

```
import plotly.express as px
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),
    paper_bgcolor="LightSteelBlue",
)
fig.show()
```



Find out Eigen values

```
# Eigen values
pca.explained_variance_
```

```
Output:
array([40.67111198, 29.17023401, 26.74459621])
```

- 👆 This much variance is being explained
- If you average `[40.67111198, 29.17023401, 26.74459621]`, you'll get → **32.195314063333335**
- ***~32% is accuracy score with 3 components***

Eigen vectors

```
# Eigen vectors
pca.components_

Output:
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [-0., -0., -0., ..., -0., -0., -0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

```
pca.components_.shape

Output:
(3, 784)
```

- 👆 There are 3 vectors which has 784 components each.

Find out optimum number of Principal Components

- λ denotes how much variance can be explained
  - Therefore the name `explained_variance_`
  - This is **actual amount of variance (Eigen Values)**

$$\left(\frac{\lambda_1}{\lambda_1+\lambda_2+\lambda_3+\dots\lambda_{784}}\right)\times 100 \rightarrow \text{percentage}$$

- **You want this sum more than 90.**

Find out % of each Eigen vector

```
pca.explained_variance_ratio_*100

Output:
array([5.78519225, 4.14926968, 3.80423901])
```

Find out the optimum number:

- Perform PCA with `n_components=None`

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

pca.explained_variance_.shape

Output:
(784,0)
```

```
pca.components_
```

```
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.]])
```

If `components_ = [[0.7, 0.3], [0.4, -0.6]]`, it means:

- PC1 is a combination of 70% of Feature 1 and 30% of Feature 2.
- PC2 is a combination of 40% of Feature 1 and -60% of Feature 2.

```
pca.components_.shape

Output:
(784, 784)
```

```
pca.explained_variance_ratio_
```

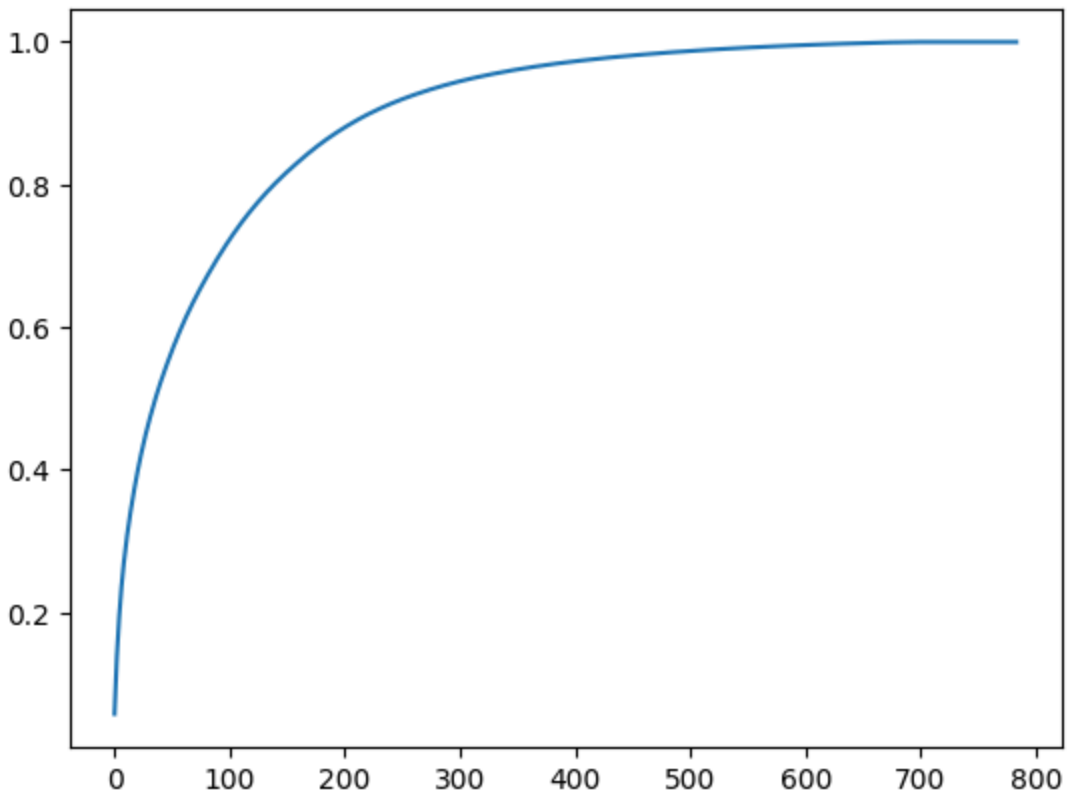
```
array([5.78519225e-02, 4.14926968e-02, 3.80423901e-02, 2.96626277e-02,
       2.58156168e-02, 2.25498018e-02, 1.97306802e-02, 1.77527998e-02,
       1.56865066e-02, 1.43606328e-02, 1.37025484e-02, 1.22725500e-02,
       1.14691200e-02, 1.12302739e-02, 1.05852885e-02, 1.01993106e-02,
       9.57676152e-03, 9.42708817e-03, 9.12489429e-03, 8.90170190e-03,
       8.39940495e-03, 8.20063196e-03, 7.85303229e-03, 7.56739707e-03,
       7.37261022e-03, 7.01884653e-03, 6.97919728e-03, 6.71104483e-03,
       6.39845030e-03, 6.30337291e-03, 6.15350848e-03, 6.02388659e-03,
       5.83673972e-03, 5.78547759e-03, 5.72333148e-03, 5.46399774e-03,
       5.43207369e-03, 5.28172755e-03, 5.13381744e-03, 4.94015474e-03,
       4.87671103e-03, 4.81978768e-03, 4.68204430e-03, 4.59342487e-03,
       4.57660834e-03, 4.49219022e-03, 4.44682221e-03, 4.41540749e-03,
       4.35111936e-03, 4.32032928e-03, 4.21808870e-03, 4.18243127e-03,
       4.07994349e-03, 4.02004913e-03, 3.98335855e-03, 3.94598812e-03,
```

- Apply a `cumsum` on above data

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

```
array([0.05785192, 0.09934462, 0.13738701, 0.16704964, 0.19286525,
       0.21541506, 0.23514574, 0.25289854, 0.26858504, 0.28294568,
       0.29664822, 0.30892077, 0.32038989, 0.33162017, 0.34220546,
       0.35240477, 0.36198153, 0.37140862, 0.38053351, 0.38943521,
       0.39783462, 0.40603525, 0.41388828, 0.42145568, 0.42882829,
       0.43584714, 0.44282633, 0.44953738, 0.45593583, 0.4622392 ,
       0.46839271, 0.4744166 , 0.48025334, 0.48603881, 0.49176214,
       0.49722614, 0.50265822, 0.50793994, 0.51307376, 0.51801392,
       0.52289063, 0.52771041, 0.53239246, 0.53698588, 0.54156249,
       0.54605468, 0.5505015 , 0.55491691, 0.55926803, 0.56358836,
       0.56780645, 0.57198888, 0.57606882, 0.58008887, 0.58407223,
       0.58801822, 0.59188694, 0.59571889, 0.59948441, 0.60322635,
       0.60687487, 0.6104829 , 0.61402176, 0.61750117, 0.62094441,
       0.62432378, 0.62767368, 0.63099377, 0.63426257, 0.63746765,
       0.64065542, 0.64376931, 0.64686564, 0.64992395, 0.65296104,
       0.6559748 , 0.65895992, 0.66192105, 0.66483654, 0.66772536,
       0.67060106, 0.6734651 , 0.67629295, 0.67910917, 0.68191711,
       0.68468646, 0.68743018, 0.69013786, 0.69282727, 0.69548652,
       0.6981275 , 0.70075516, 0.70335672, 0.7059274 , 0.70848658,
       0.71103639, 0.71354984, 0.71605747, 0.71852606, 0.72096777,
       0.72336938, 0.72575703, 0.72811046, 0.73043803, 0.73274819,
       0.73504902, 0.73730548, 0.73955195, 0.74179168, 0.7440142 ,
       0.74620053, 0.7483705 , 0.75051238, 0.7526383 , 0.75471786,
       0.75677534, 0.758826  , 0.76083529, 0.76283446, 0.76482579,
       0.76678392, 0.76873756, 0.77066405, 0.77258   , 0.77447592,
...
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
```

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



When PCA does not work?