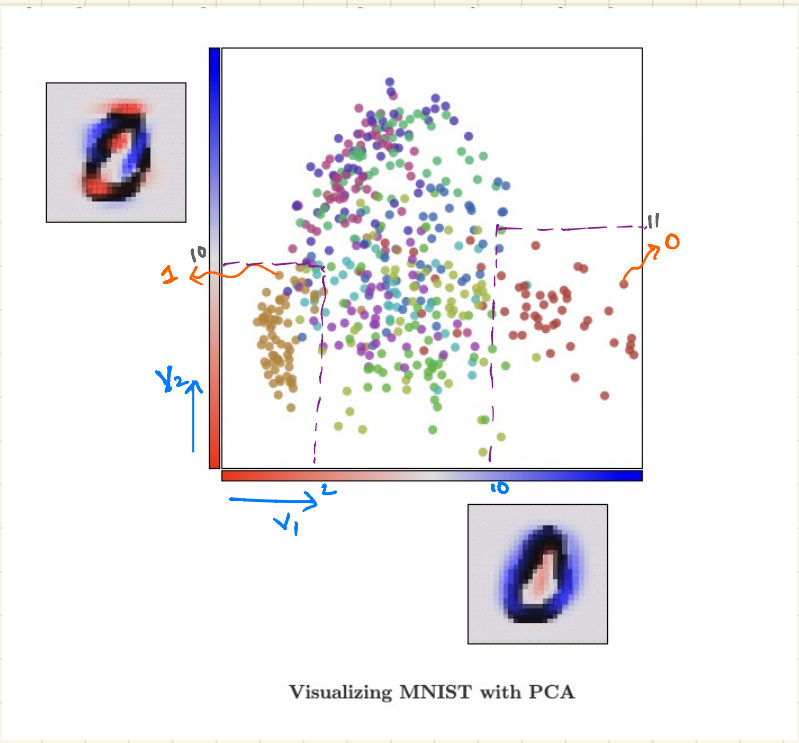
Visualize MNIST dataset



Visualizing MNIST with PCA

red → 0
brown → 1

$y_i \in \{0, 1, 2, 3, \ldots\ldots 9\}$

if $v_1 <= 2$ and $v_2 <= 10$
$y_i = 1$

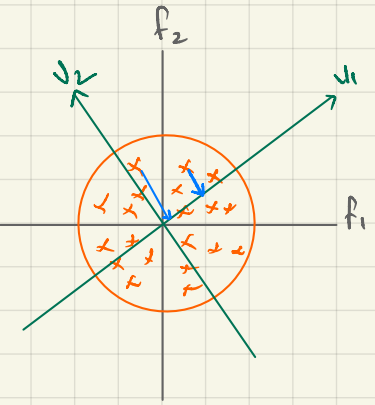
if $v_1 \geq 10$ and $v_2 \leq 11$
$y_i = 0$

→ here, 0 & 1 are only well separated and not other value eg:- 2, 3, 4 --- 9.

→ So, I can only identify '0' and '1' from this (PCA) technique.

Limitations of PCA

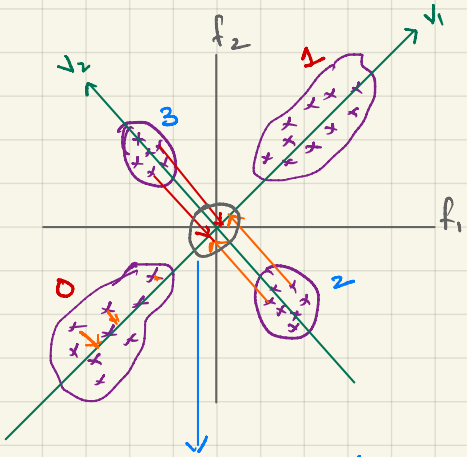
$2D \rightarrow 1D$



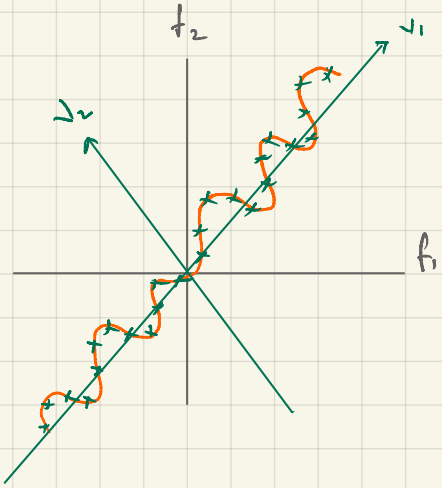
$\lambda_1 \simeq \lambda_2$

(info - lost is very high)

if surface is sphere or hyper-sphere we will lose many data.



It will be hard to distinguish pts are coming from which cluster in 1-D visualization.



If I project all the data on v_1. I will lose the beautiful sinosodal structure. I will lose the shape.

Some fun" & code

fit, transform, fit_transform

EDA → FE → FS → Model creation → Model deployment

FE → Data preprocessing

Model creation → Model training → Models

Eg:- standardscalar, MinMax scales, PCA, Imputer

Eg: Linear Regression, Logistic, DT, RF, AdaBoost

⇒

f	f'
Age	Age
21	0.2
60	0.58
45	0.35
31	0.3

f → f'

standard scalar (sc)
or
PCA
or
⋮

⇒ sc.fit(Age) → it finds out the parameter needed for trasformation.

⇒ sc.transform(Age) → it tranform every data-points in a matrix. But, for that it needs "fit()" parameters.

⇒ sc.fit_transform() → it fits & transform every data points by using a single line fun".

PCA Code example

→ Using eigen value & vectors.
→ Using scikit-Learn.

Pca using eigen value & vector

```
[4] # Pick first 15K data-points to work on for time-effeciency.
    #Excercise: Perform the same analysis on all of 42K data-points.

    labels = l.head(15000)
    data = d.head(15000)

    print("the shape of sample data = ", data.shape)

    the shape of sample data =  (15000, 784)
```

→ 28×28

```
    # Data-preprocessing: Standardizing the data
    from sklearn.preprocessing import StandardScaler
    standardized_data = StandardScaler().fit_transform(data)
    print(standardized_data.shape)

    (15000, 784)
```

pkg → module → function

$\frac{x_i - \mu_i}{\sigma_i}$

sklearn.preprocessing.StandardScaler

class `sklearn.preprocessing.StandardScaler(*, copy=True, with_mean=True, with_std=True)` [source]

Standardize features by removing the mean and scaling to unit variance.

The standard score of a sample `x` is calculated as:

z = (x - u) / s

where `u` is the mean of the training samples or zero if `with_mean=False`, and `s` is the standard deviation of the training samples or one if `with_std=False`.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using `transform`.

→ here, I standardised my data, row-wise.

```
[6] #find the co-variance matrix which is : A^T * A
    sample_data = standardized_data

    # matrix multiplication using numpy
    covar_matrix = np.matmul(sample_data.T , sample_data)

    print ( "The shape of variance matrix = ", covar_matrix.shape)

    The shape of variance matrix =  (784, 784)
```

matmul → matrix multiplication

```
[7] # finding the top two eigen-values and corresponding eigen-vectors
    # for projecting onto a 2-Dim space.

    from scipy.linalg import eigh

    # the parameter 'eigvals' is defined (low value to heigh value)
    # eigh function will return the eigen values in asending order
    # this code generates only the top 2 (782 and 783) eigenvalues.
    values, vectors = eigh(covar_matrix, eigvals=(782,783))

    print("Shape of eigen vectors = ",vectors.shape)
    # converting the eigen vectors into (2,d) shape for easyness of further computations
    vectors = vectors.T

    print("Updated shape of eigen vectors = ",vectors.shape)
```

linear algebra

top two eigen values

```
  # here the vectors[1] represent the eigen vector corresponding 1st principal eigen vector
  # here the vectors[0] represent the eigen vector corresponding 2nd principal eigen vector

  <ipython-input-7-d08a4b58c69a>:9: DeprecationWarning: Keyword argument 'eigvals' is deprecated in favour of 'subset_by_i
    values, vectors = eigh(covar_matrix, eigvals=(782,783))
  Shape of eigen vectors =  (784, 2)
  Updated shape of eigen vectors =  (2, 784)
```

scipy.linalg.eigh

scipy.linalg.eigh(*a, b=None, lower=True, eigvals_only=False, overwrite_a=False, overwrite_b=False, turbo=False, eigvals=None, type=1, check_finite=True, subset_by_index=None, subset_by_value=None, driver=None*) [source]

Solve a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues array `w` and optionally eigenvectors array `v` of array `a`, where `b` is positive definite such that for every eigenvalue λ (i-th entry of w) and its eigenvector `vi` (i-th column of `v`) satisfies:

```
                a @ vi = λ * b @ vi
vi.conj().T @ a @ vi = λ
vi.conj().T @ b @ vi = 1
```

In the standard problem, `b` is assumed to be the identity matrix.

Parameters:

a : *(M, M) array_like*
A complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.

b : *(M, M) array_like, optional*
A complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed.

lower : *bool, optional*
Whether the pertinent array data is taken from the lower or upper triangle of `a` and, if applicable, `b`. (Default: lower)

eigvals_only : *bool, optional*
Whether to calculate only eigenvalues and no eigenvectors. (Default: both are calculated)

eigvals : *tuple (lo, hi), optional, deprecated*

> **Deprecated since version 1.5.0:** `eigh` keyword argument `eigvals` is deprecated in favour of *subset_by_index* keyword instead and will be removed in SciPy 1.12.0.

Returns:

w : *(N,) ndarray*
The N (1<=N<=M) selected eigenvalues, in ascending order, each repeated according to its multiplicity.

v : *(M, N) ndarray*
(if `eigvals_only == False`)

```
[8] # projecting the original data sample on the plane
    #formed by two principal eigen vectors by vector-vector multiplication.

    import matplotlib.pyplot as plt
    new_coordinates = np.matmul(vectors, sample_data.T)

    print (" resultanat new data points' shape ", vectors.shape, "X", sample_data.T.shape," = ", new_coordinates.shape)

     resultanat new data points' shape  (2, 784) X (784, 15000)  =  (2, 15000)
```

784 → 2 dimen.

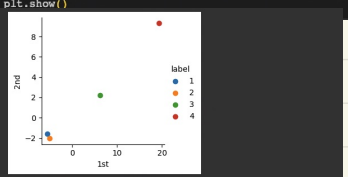
```
[9] import pandas as pd

    # appending label to the 2d projected data
    new_coordinates = np.vstack((new_coordinates, labels)).T

    # creating a new data frame for ploting the labeled points.
    dataframe = pd.DataFrame(data=new_coordinates, columns=("1st_principal", "2nd_principal", "label"))
    print(dataframe.head())

       1st_principal  2nd_principal  label
    0      -5.558661      -5.043558    1.0
    1       6.193635      19.305278    0.0
    2      -1.909878      -7.678775    1.0
    3       5.525748      -0.464845    4.0
    4       6.366527      26.644289    0.0
```

vstack → vertical stacking

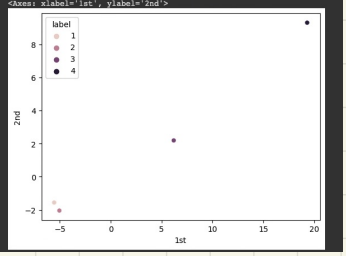
x_i → 784 dim

PCA → x_i → 2 dim

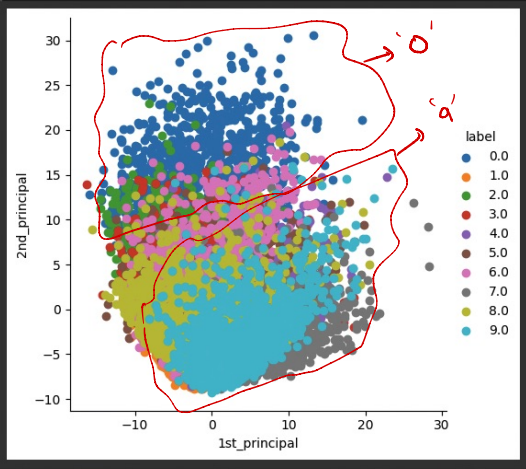
```
[1] import pandas as pd
    df=pd.DataFrame()
    df['1st']=[-5.558661,-5.043558,6.193635 ,19.305278]
    df['2nd']=[-1.558661,-2.043558,2.193635 ,9.305278]
    df['label']=[1,2,3,4]

[3] import seaborn as sn
    import matplotlib.pyplot as plt
    sn.FacetGrid(df, hue="label").map(plt.scatter, '1st', '2nd').add_legend()
    plt.show()
```




```
[14] sn.scatterplot(x='1st',y='2nd',hue='label',data=df)

<Axes: xlabel='1st', ylabel='2nd'>
```



```
import seaborn as sn
sn.FacetGrid(dataframe, hue="label", height = 5, aspect = 1).map(plt.scatter, '1st_principal', '2nd_principal').add_legend()
plt.show()
```



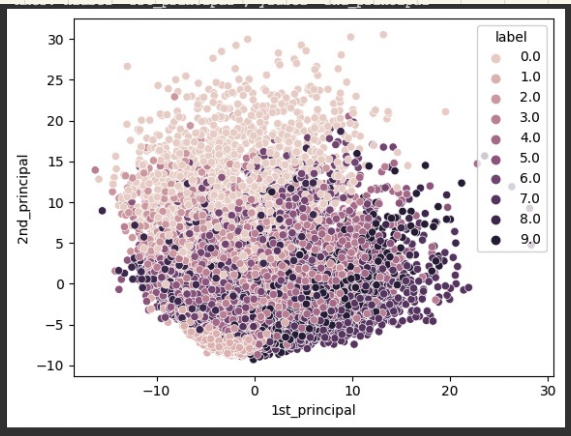
→ too much overlap of data.

→ PCA does not work very very good job in visualization.

→ It does good work in data reduction (dim. reduction)

```
[18] sn.scatterplot(x="1st_principal",y="2nd_principal",legend="full",hue="label",data=dataframe)
```

PCA Using sklearn



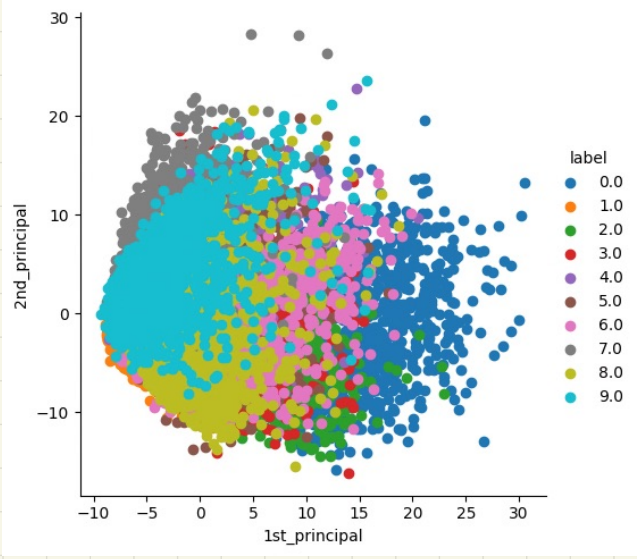

```
[19] # initializing the pca
     from sklearn import decomposition
     pca = decomposition.PCA()

[32] # configuring the parameteres
     # the number of components = 2
     pca.n_components = 2
     pca_data = pca.fit_transform(sample_data)

     # pca_reduced will contain the 2-d projects of simple data
     print("shape of pca_reduced.shape = ", pca_data.shape)

     shape of pca_reduced.shape =  (15000, 2)

[33] # attaching the label for each 2-d data point
     pca_data = np.vstack((pca_data.T, labels)).T

     # creating a new data fram which help us in ploting the result data
     pca_df = pd.DataFrame(data=pca_data, columns=("1st_principal", "2nd_principal", "label"))
     sn.FacetGrid(pca_df, hue="label", height = 5, aspect = 1).map(plt.scatter, '1st_principal', '2nd_principal').add_legend()
     plt.show()
```

→ module

→ function

→ Covariance matrix

→ every thing done with few line of codes.



PCA for dimensionality reduction (not-visualization)

PCA: 784 dim ⟶ 2 dim or 3 dim → for visualization [did that earlier]

784 dim ⟶ (PCA) ⟶ 10 dim. or 100 dim. or 90% of variance covered (how many dim. for that). → For ML model

→ 784 ⟶ 200 dim (how)

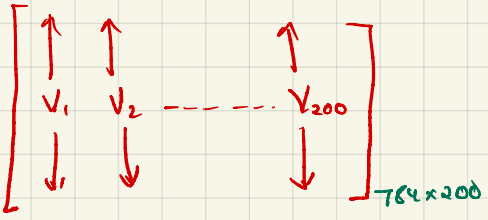
$C = X^T X$

$d_i : i \in 1 \to 784$

$V_i : i \in 1 \to 784$

PCA

$X_{15000 \times 784} \; V_{784 \times 200} = X'_{15K \times 200}$



Q) what is the right no. for dim. reduction? 784 ⟶ 10, 20, ..., 70 → (?)

{PCA: → max-variance of proj.-points}

PCA:-

$d_1 \geq d_2 \geq \dots \geq d_{784}$

$V_1, V_2 \dots V_{784}$

784 ⟶ 10 dim.

% variance explained in 10-dim?

$$= \frac{d_1 + d_2 + \dots + d_{10}}{\sum_{i=1}^{784} d_i} = 0.2$$

20% of the variance of information has been retained when I projected in 10-dim.

Task :- I want to find d', so it retain 90% of info./variance.

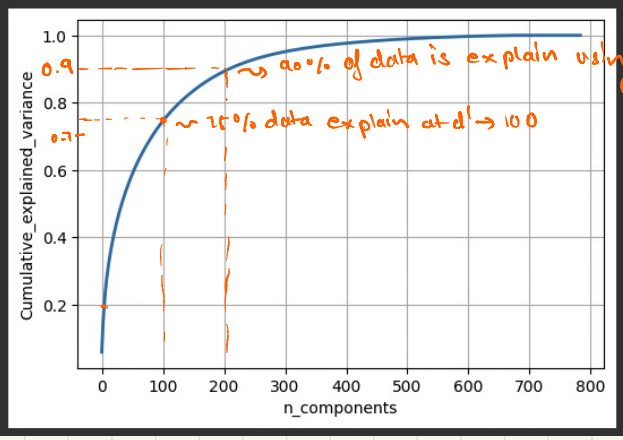
$$\rightarrow \frac{d_1 + d_2 + \cdots d_{d'}}{\sum_{i=1}^{784} d_i} = 0.9$$

```
# PCA for dimensionality redcution (non-visualization)

pca.n_components = 784
pca_data = pca.fit_transform(sample_data)

percentage_var_explained = pca.explained_variance_ / np.sum(pca.explained_variance_);

cum_var_explained = np.cumsum(percentage_var_explained)

# Plot the PCA spectrum
plt.figure(1, figsize=(6, 4))

plt.clf()
plt.plot(cum_var_explained, linewidth=2)
plt.axis('tight')
plt.grid()
plt.xlabel('n_components')
plt.ylabel('Cumulative_explained_variance')
plt.show()


# If we take 200-dimensions, approx. 90% of variance is expalined.
```

→ explained_variance_ → attribute of PCA()

→ cummulative sum

$$\frac{d_i}{\sum d_i} = \frac{d_1}{\sum d_i} + \frac{d_2}{\sum d_i} + \cdots + \frac{d_{784}}{\sum d_{784}}$$



→ It is very imp. for ML model (dim. reduction).