upGrad

Summary

Introduction to Principal Component Analysis

In this session, you got an overview of the principal component analysis algorithm and its application. PCA helps in dimensionality reduction by capturing maximum information. It does this by capturing the variance in the data. In order to understand the inner workings of PCA, you need to first learn about its three main building blocks, which are as follows:

1. **Summary statistics:** This includes mean, median, mode, variance, standard deviation and covariance.
2. **Vectors:** This involves vector operations such as addition, scalar multiplication, basis and change of basis.
3. **Matrices:** This involves representation of data in the form of matrices and vectors, matrix operations, etc.

Out of the aforementioned tools, summary statistics are covered in a separate optional module. You are expected to complete the '**Summary Statistics**' module before starting this module. The topics of vectors and matrices are covered in this session.

What Does PCA Do?

Before learning about a new concept, you need to understand why and how that knowledge is useful. So, let's try to understand the problems associated with having many features in the dataset:

- **The predictive model set-up**: When there are many correlated features in a data set, it leads to the problem of multicollinearity. Certain methods can be used to reduce the dimensionality and remove the correlation among the features, but it might lead to some information loss. In contrast, PCA reduces the dimensionality by keeping the maximum information of the data set intact.
- **Data visualisation:** It is not possible to visualise more than two variables simultaneously using a 2-D plot. Therefore, finding relationships between the observations in a data set that has several variables through visualisation is quite difficult.

So, PCA reduces the dimensionality without any information loss. Further, you do not need to eliminate any feature by considering the p-values or VIF values in the case of PCA because it automatically finds the best set of features. This is one of the advantages of PCA.

Introduction to Dimensionality Reduction

In this segment, you learnt about the concept of low rank approximation. The rank of a matrix is the number of linearly independent rows or columns in a matrix, whichever is lower. Let's consider the following matrix.

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Here, **(Third column) = 2 X (Second column) - (First column).** Hence, the rank of the square matrix 'X' is not 3; it is 2 because only two columns are linearly independent. This process, wherein you reduce the rank of a matrix by considering only the independent columns or rows, without taking into account the linearly dependent columns or rows, is called low-rank approximation.

Now, let's slightly change the value '1' to '1.001'.

$$Y = \begin{bmatrix} 1.001 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

As you can see in the 'Y' matrix given above, you cannot write the third column as a linear combination of the first and second columns, or any column as a linear combination of other columns. This matrix is called a **full rank matrix,** and the rank of this matrix is 3.

Now, suppose you want to build an ML model using the three columns of the matrix 'Y' as the features. Although there is no dependency of any column on another column, there is a high correlation (or covariance) among the columns, which means that there is some linear dependency of column 3 on column 1 and column 2. The correlation coefficient (which can be interpreted as covariance) between the columns represents the extent to which they are linearly related to one another.

So, in order to reduce the dimensionality when there is no explicit relationship among the variables, you need to consider the variance rather than the rank. Similar to PCA, there are many algorithms that use variance to extract the low-rank decomposition of the data set based on which the component maximises the variance of that particular data set.

You will notice this phenomenon at many instances in machine learning model building, wherein you cannot express one feature with the exact linear combination of the other features. However, this does not mean that the features are not correlated to one another at all. In such cases, you can use the variance as the measure to decompose the data into a low rank and consider only those components/features that maximise the variance.

Following are other low-rank approximation techniques that do not necessarily consider the assumption of linear combination of columns:

1. Random projections
2. Kernel PCA
3. ISOMAP

4. T-SNE / Manifold learning

In the next segment, you will learn about the PCA algorithm and understand how it works and how it fetches the maximum information from a given data set.

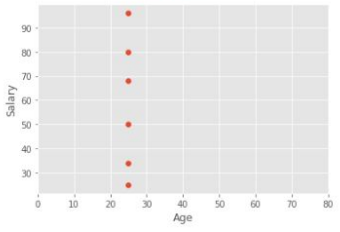
Introduction to PCA - I

You learnt that PCA is a low-rank decomposition algorithm that uses variance as a measure of information. In this segment and the next, you will use a graphical and statistical interpretation of PCA and understand what exactly PCA does.

Let's consider the following three tables.

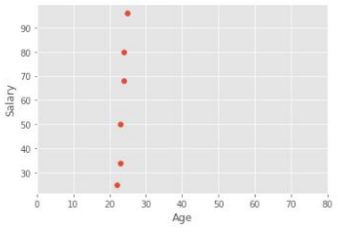
Salary (1000 X 1₹)	Age (Year)
25	25
34	25
50	25
68	25
80	25
96	25

Table A



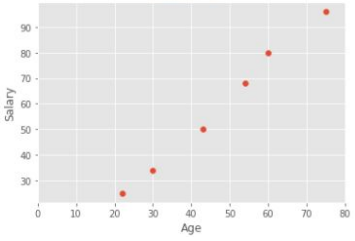
Salary (1000 X 1₹)	Age (Year)
25	22
34	23
50	23
68	24
80	24
96	25

Table B



Salary (1000 X 1₹)	Age (Year)
25	22
34	30
50	43
68	54
80	60
96	75

Table C



In Table A, the age of the employees is not varying with the different data points; hence, it can be concluded that the Age column is not useful for building a machine learning model because it does not contain any information.

On the other hand, the age is slightly varying in Table B and significantly varying in Table C. Now, if you want to build an ML model using the columns of these tables, then only the Age column in Table C would be the most relevant because it has the maximum variance.

Thus, you can conclude that if a particular column has more variance in it, then that column is considered more important than the other columns in the data set. In simple terms, if a column has high variance, then that particular column contains more information.

Now, let's consider Table C and calculate the variance of each column in it.

Variance in age = 386.27
Variance in salary = 750.57
Covariance between age and salary = 447.22

Now, let's perform a short exercise and calculate the percentage of variance in each variable.

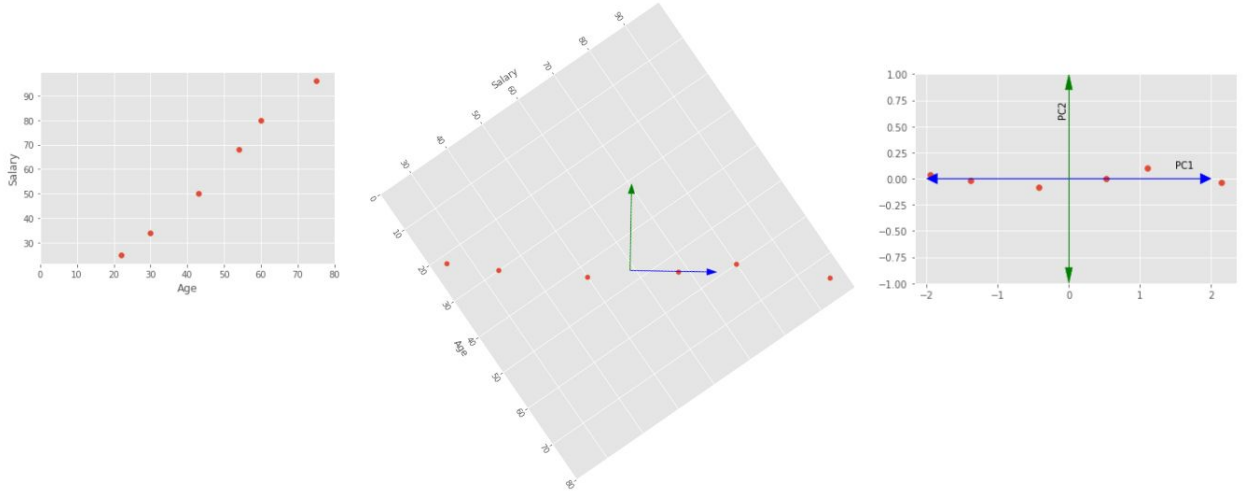
Percentage of variance in age = 34%
Percentage of variance in salary = 66%

As you can see, both columns contain a significant amount of variance; hence, both columns are important for the analysis. Also, the columns are correlated to each other.

Suppose you are able to create two new columns, say, PC1 and PC2, from the original set of columns, i.e., Age and Salary, such that the percentage of variance in PC1 is 99.8% and that of variance in PC2 is 0.2%. Also, the covariance between PC1 and PC2 is almost zero.

Now, you can simply drop PC2, as PC1 and PC2 are uncorrelated to each other (covariance almost equals to zero), and almost 99% of the information, i.e., variance is contained by PC1.

Let's consider Table C and its scatter plot and try to get the feel of PCA by rotating the axis.



In the first graph, there is a significant variance along both axes. Now, let's change this and view the data from the perspective of the blue and green axes.

As you can see in the graphs provided above, the variance along the blue axis, PC1, is maximum and that along the green axis, PC2, is minimum. So, you have finally found the direction along which you can derive the maximum variance, or, in other words, the maximum information.

Now, once you have found the direction of the maximum variance, the next task is to project the original data points on these newly obtained axes. This is what PCA does. In the upcoming segments, you will learn how it manages to do this.

upGrad

Introduction to PCA - II

In this segment, you learnt how to get the projected data points on the newly created axes, PC1 and PC2.

Suppose you have a transformation matrix as shown below. Now, when you multiply this matrix with the data points, you get the projected data points on the principal component axis.

Salary (1000 X ₹)	Age (Year)
25	22
34	30
50	43
68	54
80	60
96	75

$$\begin{bmatrix} Transformation \\ Matrix \end{bmatrix} \begin{bmatrix} 25 \\ 22 \end{bmatrix} = \begin{bmatrix} -1.95503533 \\ 0.04185559 \end{bmatrix}$$

PC1	PC2
-1.95503533	0.04185559
-1.38527443	-0.01898064
-0.42053679	-0.07896334
0.5219234	-0.00357421
1.09768043	0.09923497
2.14124272	-0.03957237

Covariance Matrix

$$\begin{bmatrix} 386.27 & 447.22 \\ 447.22 & 750.57 \end{bmatrix}$$

Covariance Matrix

$$\begin{bmatrix} 2.396 & -1.48367e^{-10} \\ -1.48367e^{-10} & 0.00395 \end{bmatrix}$$

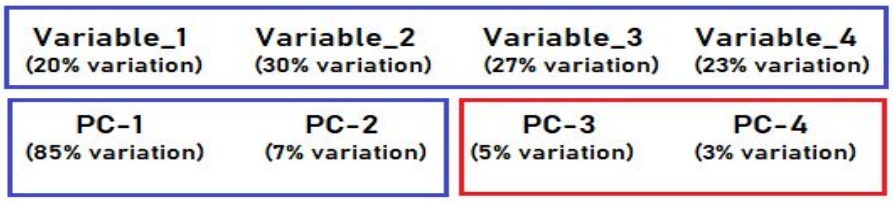
The following points can be observed from the two covariance matrices given above:

- The first covariance matrix corresponds to the original data set. As you can see in this covariance matrix, the values of variance in the Salary and Age columns are 386.27 and 750.57, respectively. Both the variance values are high, which means that both variables are relevant. Also, the values of covariance between the variables are 447.22, which means that the salary and age variables are highly correlated to each other, which will create problems in modelling.
- On the other hand, in the covariance matrix of PC1 and PC2 (which are the projected points of the original data points on the new axes), you will notice that there is a high variance (99.8%) in PC1 and a negligibly small variance (0.8%) in PC2.
- There is almost zero covariance between PC1 and PC2.

So, essentially, PCA converts possibly correlated variables into new features that are known as ‘principal components’ in such a way that:

- They are uncorrelated to each other,
- They are the linear combinations of the original variables, and
- They capture maximum information in the data set.

Note that the number of principal components is the same as that of the columns in the data set. PCs are sorted in descending order of the information content. For example, if you have a data set with four columns, then you will get four PCs, and the first PC will have the maximum variance, the second PC will have the second maximum variance, and so on.



As you can see in the image given above, the original data set contains four variables with significant variance in each variable. Once you find the PCs of this data set, you can get most of the variance (about 92% of the information) in the top two PCs. So, you can discard PC3 and PC4, which would lead to dimensionality reduction in the data set without losing too much of the information. Note that this can change with the use case. In case you need to capture 100% variance, you would use all the four features.

So, PCA helps in the following:

- **Data visualisation and EDA:** It helps in data visualisation and EDA because a few variables in the data set carry maximum information. Plotting the scatter plot of two variables is easier than analysing multiple variables at a time.
- **The predictive model set-up:** When many features in a data set are correlated, it leads to the problem of multicollinearity. Removing features iteratively is time-consuming and also leads to information loss.
- **For creating uncorrelated features** that can be input to a prediction model: With fewer uncorrelated features, the modelling process can be faster and more stable.

Now, the aforementioned definition introduces some new terms and phrases, such as '**linear combinations**' and '**capturing maximum information**'. To understand these terms and phrases, you need to have some knowledge of linear algebra concepts along with **vectors and matrices,** which are the building blocks of PCA.

Vectors

In the last two segments, you understood what PCA does. In order to learn about the PCA algorithm in detail, you need to have an understanding the following three tools:

1. **Summary statistics:** This includes mean, median, mode, variance, standard deviation and covariance.
2. **Vectors:** This involves vector operations such as addition, scalar multiplication and vector representation of the data.
3. **Matrices:** This includes representation of vectors in the form of matrix, matrix operations, basis and change of basis.

You are expected to go through the optional module on summary statistics before proceeding with this module, which serves as the prerequisite module to understand PCA.

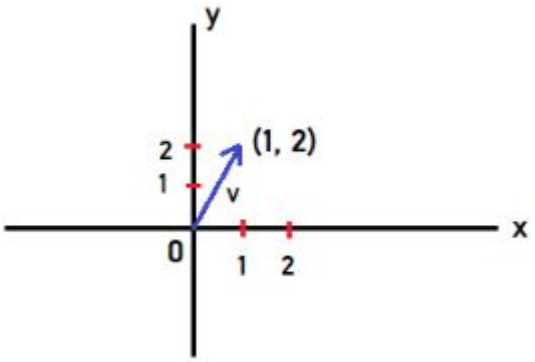
Let's start the discussion with the concept of vectors.

- A vector is an **ordered** collection of numbers. Suppose you have the vector 'x' in the n-dimensional space. Now, the elements in the vector 'x' will be in an order such that x1 is on the first axis, x2 is on the second axis, x3 is on the third axis, and so on, as shown in the image given below.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

- A vector is used to represent a point in a given space. The vector 'v' has two dimensions, as shown below.

$$v = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$



Here, we have taken a vector 'v' to represent a point (1, 2) in the two-dimensional space.

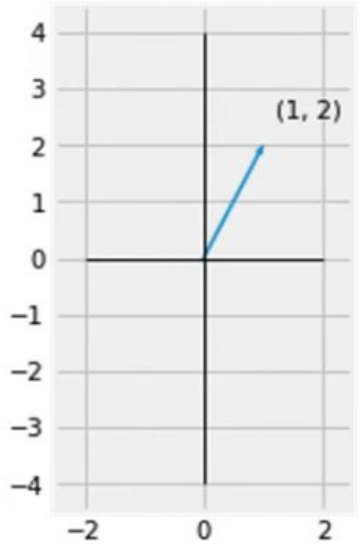
In literature, a vector is represented in the following ways:

- A vector is represented in **lower case and bold** letters. Each element of the vectors is a scalar, denoted by lower case letters.
- A vector is represented as a single-column matrix whose order is **n X 1,** where 'n' is the number of dimensions in the space.

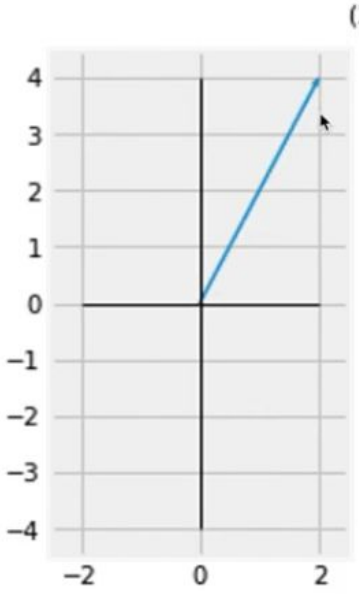
Scalar multiplication of a vector

Suppose you have a vector 'x' denoting a point (1, 2) in a two-dimensional space. If you multiply this vector by a factor of two, it simply scales the vector 'x' by a factor of two, after which the new point that the vector will denote will be (2, 4).

The scalar multiplication of this vector is shown in the image given below.



$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

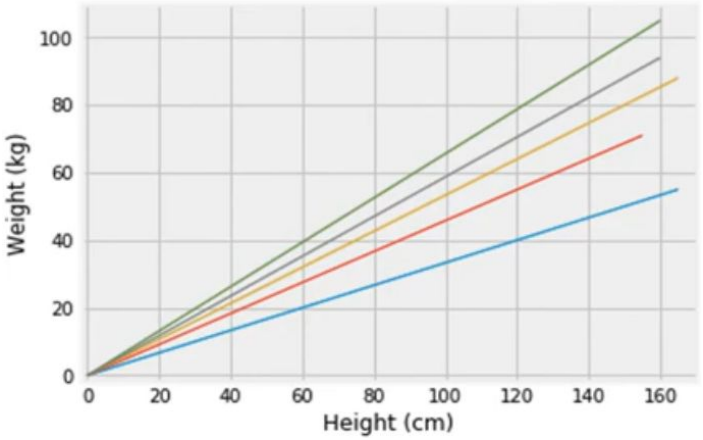


$$2\boldsymbol{x} = 2\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

The basic purpose of understanding vectors is to represent the data points of a data set in the form of vectors. Let's take a look at the data set of different patients containing two columns: Height (cm) and Weight (kg).

Patient ID	Height (cm)	Weight (kg)
P1	165	55
P2	155	71
P3	165	88
P4	160	105
P5	160	94

You can represent this particular data in the form of vectors on a two-dimensional plane, as it contains only two columns.



So, in the graph given above, you can see that there are two axes, Height and Weight. You can represent each data point on this two-dimensional plane.

Vector representation in n-dimensional space

Each vector contains values representing all the dimensions or variables in the data. For example, if there was a variable 'age' included in the data set given above and the first patient was 22 years old, then the vector representing the patient would be written as (165, 55, 22). Similarly, if the data set had 10 variables, the vector representation would have 10 dimensions. Similarly, you can extend it to n dimensions or variables.

Matrices

In this segment, you learnt about matrices. Let's take the example of the patients' data set from the previous segment to understand the concept of a matrix.

Patient ID	Height (cm)	Weight (kg)
P1	165	55
P2	155	71
P3	165	88
P4	160	105
P5	160	94

As you can see in the table given above, each row can be depicted in the form of a vector (a column matrix), i.e., the vector representation of the data points on an n-dimensional space.

What if we could represent the vectors of all the points in a single-grouped manner as shown below?

$$\begin{bmatrix} 165 & 155 & 165 & 160 & 160 \\ 55 & 71 & 88 & 105 & 94 \end{bmatrix}$$

Here, each column represents a vector, and there are five vectors in the form of a matrix.

The features of a matrix can be summarised as follows:

- A matrix is an ordered collection of numbers, just like a vector.
- Unlike vectors, a matrix has more than one index. It has numbers along its rows and columns. The order of a matrix is defined as follows:
  Number of rows X Number of columns

  Suppose you have a matrix 'X' with entries as shown below.

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{n1} \\ x_{12} & x_{22} & \ldots & x_{n2} \\ \vdots & \vdots & \vdots & \vdots \\ x_{1m} & x_{2m} & \cdots & x_{nm} \end{bmatrix}$$

  The order of this matrix is **n X m.**

- A matrix can also be considered a collection of vectors. Let's take the matrix 'X' again. As you can see below, each column of the matrix has a vector such as **x1, x2, x3 and so on**. So, the matrix 'X' is a collection of 'n' vectors, and each vector has 'm' dimensions as shown below.

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{n1} \\ x_{12} & x_{22} & \ldots & x_{n2} \\ \vdots & \vdots & \vdots & \vdots \\ x_{1m} & x_{2m} & \cdots & x_{nm} \end{bmatrix}$$

$$\boldsymbol{X} = [\boldsymbol{x_1} \quad \boldsymbol{x_2} \quad \cdots \quad \boldsymbol{x_n}]$$

Matrix addition operation

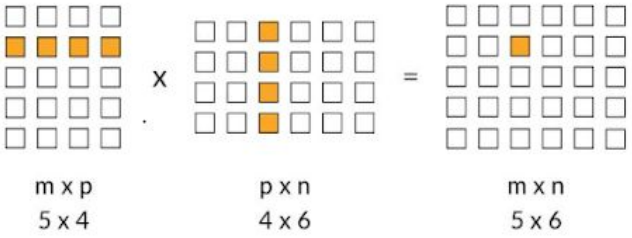
The addition of two matrices (X and Y) can be performed only if the following condition is fulfilled:

Number of col(X) = Number of col(Y) AND Number of rows(X) = Number of rows(Y)

Matrix multiplication operation

The matrix multiplication operation is an ordered operation, which means that AXB is not equal to BXA. The operation AXB is only possible if the following condition is fulfilled:

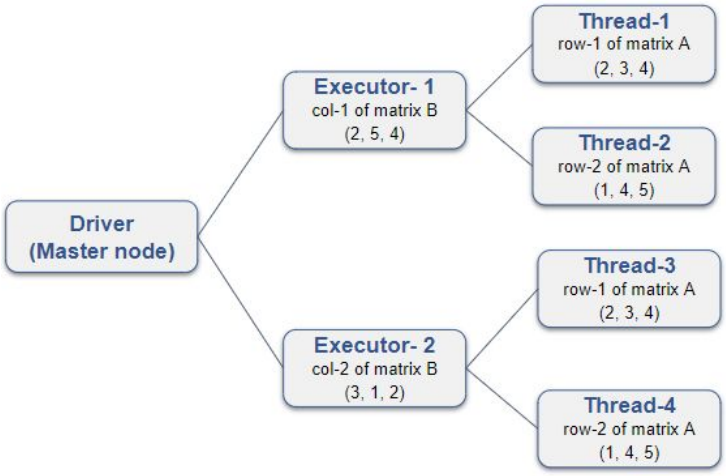
Number of col (A) = Number of rows (B)



Matrix multiplication is a parallel process because all the values in the resultant matrix are independent of one another. Suppose you want to multiply matrix 'A' by matrix 'B'. The process of multiplying a row of matrix 'A' by a column of matrix B is distributed among multiple executors as shown in the diagram given below.

$$A = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 4 & 5 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 3 \\ 5 & 1 \\ 4 & 2 \end{bmatrix}$$



Here, the columns of matrix 'B' are distributed among two different executors so that they can be multiplied by the rows of matrix 'A', and each thread will result in each element of the resultant matrix, matrix 'R'. Hence, matrix 'R' will be represented as shown below.

$$R = \begin{bmatrix} 35 & 17 \\ 42 & 17 \end{bmatrix}$$

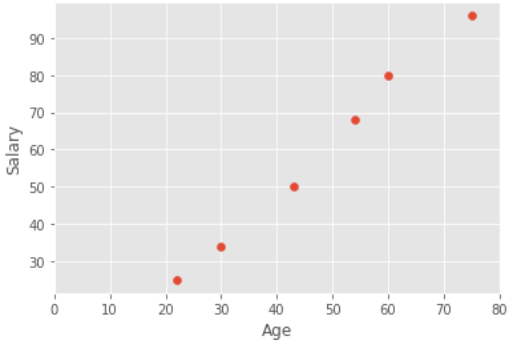
Here, each element in matrix 'R' is obtained parallelly and simultaneously.

Covariance Matrix

By now, you have a basic understanding of a matrix and its operations. Now, let's understand a special type of matrix: the covariance matrix. This matrix is used in the PCA algorithm.

Let's take a look at the dummy data set that we used earlier in which the salary varies with the person's age, as shown in the table given below.

Salary (₹ in thousand)	Age (Years)
25	22
34	30
50	43
68	54
80	60
96	75



As you can see in the graph given above, as the salary of a person increases with their age. The relation between the variables is quantified using a specific measure, 'covariance'.

- You already know that variance refers to the spread of data points around their mean. A high variance means that the points are far from the mean. In other words, variance is the tendency of data points to fluctuate around their own mean.
- Covariance refers to the measure of how two random variables in a data set will change together.
- A **positive covariance** means that the two variables are positively related and they move in the same direction. A **negative covariance** means that the variables are inversely related and they move in the opposite directions.
- The formula to calculate the covariance between two variables 'x' and 'y' is similar to the variance formula. The formula is as follows:

$$\sigma_{xy}^2 = E[(x - \mu_x)(y - \mu_y)]$$ where, E is the mean operator.

Now, let's proceed to the concept of a covariance matrix. Suppose you are given a data set with three columns: 'col-1', 'col-2' and 'col-3'. Can you identify the number of combinations among the three columns of the data set in order to find the covariance between them?

The columns have three combinations, (col-1, col-2), (col-2, col-3) and (col-1, col-3). Also, there are three variance values corresponding to each column in the data set. When you arrange them in the form of a matrix, this matrix is called a covariance matrix.

In a covariance matrix, the diagonal entries are the variances corresponding to each variable, and the non-diagonal entries are the covariances of the variables. Note that the **covariance of 'x' with respect to 'y' is the same as the covariance of 'y' with respect to 'x'.**

So, in the example given above, wherein there are three columns, the size of the covariance matrix will be 3X3. You can depict the covariance matrix of 'n' column data in the format given below.

$$\Sigma_X = \begin{bmatrix} \sigma^2_{x_1x_1} & \sigma^2_{x_2x_1} & \cdots & \sigma^2_{x_nx_1} \\ \sigma^2_{x_1x_2} & \sigma^2_{x_2x_2} & \cdots & \sigma^2_{x_nx_2} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma^2_{x_1x_n} & \sigma^2_{x_2x_n} & \cdots & \sigma^2_{x_nx_n} \end{bmatrix}$$

As you can see, a **covariance matrix is a symmetric matrix** along its diagonal because the sigmasqr(x2, x1) is equal to the sigmasqr(x1, x2), and so on.

Now, let's understand the concept of a correlation matrix.

The formula to calculate the correlation between 'x' and 'y' from the covariance is as follows:

$$\rho_{xy} = \frac{E[(x-\mu_x)(y-\mu_y)]}{\sigma_x\sigma_y} = \frac{\sigma^2_{xy}}{\sigma_x\sigma_y}$$

Where, sigma(x) and sigma(y) are the standard deviations of 'x' and 'y', respectively.

Correlation also depicts the dependency between the two variables. The basic difference between covariance and correlation is as follows:

'Covariance' indicates the direction of the linear relationship between variables. On the other hand, 'correlation' measures both the strength and the direction of the linear relationship between the two variables.

The value of correlation lies between -1 to +1.

- +1 denotes a perfect linear relation between two variables, and if one variable increases, then the other variable also increases.
- -1 denotes a perfect linear relation between two variables in an inverse sense, which means that if one variable increases, then the other variable decreases.
- 0 denotes that there is no linear relationship between the variables.

upGrad

Summary

Basis and Python Demonstration of Matrices

In this session, you learnt how to perform vector and matrix operations using Python. Once the Python demonstration is done, you learned another important concept called **'Basis'**.

The broad flow of this session is as follows:

1. Python demonstration of vectors and matrices
2. Python demonstration of covariance matrices
3. Conceptual understanding of basis
4. Change of basis

Vectors and Matrices Python Demonstration

Let's start the discussion with a function to plot the graph.

1. So, first, you need to import the two important libraries that you are already aware of. They are as follows:
   - *numpy*
   - *matplotlib.pyplot*

2. In the next code, you write a function called **'*draw_vectors*'** to plot vectors on a graph. This function is as follows:

```
COLORS = plt.rcParams['axes.prop_cycle'].by_key()['color'] * 2
def draw_vectors(*vectors, **kwargs):
    X = np.vstack(vectors)
    fig, ax = plt.subplots()
    for i, v in enumerate(X):
        ax.arrow(0, 0, *v, color=COLORS[i], length_includes_head=True,
                 width=0.03, head_width=0.1)
    xmax, ymax = np.abs(X.max(0))
    ax.axis([-xmax - 1, xmax + 1, -ymax -1, ymax + 1])
    ax.set_aspect('equal')
```

   - ***np.vstack(vectors):*** This particular function is used to stack the sequence of input arrays vertically in order to make a single array. So, if you have two arrays such as [1, 2] and [3, 4], then this function will stack them in the following format:
     [[1, 2]
     [3, 4]]

- ***fig, ax = plt.subplots():*** This method provides a way to plot multiple plots in a single figure. Given the number of 'rows' and 'columns', it returns a tuple (fig, ax), giving a single figure 'fig' with an array of axes 'ax'.
  If you want to plot four plots in a single subplot, then write ***fig, ax = plt.subplots(2, 2).*** In this code, you get four plots in a single figure, and the array 'ax' will contain four indexes as ax[0, 0], ax[1, 0], ax[0, 1] and ax[1, 1].

- ***ax.arrow():*** This method is used to draw an arrow. Here, you start a 'for' loop, which receives the vectors as inputs from the stack 'X' one by one.

- ***ax.set_aspect('equal'):*** This function is used in the Matplotlib library to set the aspect of the axis scaling, i.e., the ratio of y-unit to x-unit. As you have used 'equal' here, it will set the scaling ratio of y-unit to x-unit as equal.

Note: In theory, you have understood the vectors in the form of a column matrix. However, in numpy, the vectors are represented in the form of a row matrix.

Covariance Matrix Python Demonstration

In this segment, you learnt to plot a scatter plot and calculate the covariance between two variables. To find the covariance matrix between two variables, you can use the following function:

np.cov(df[var-1], df[var-2])

You also learnt the following points in this segment:

If the value of covariance is high, then there is a high tendency that one variable is linearly associated with the other variable, which means that:

- If there is a high positive covariance, then there is a high linear association between the two variables, and if one variable increases, then the other will also increase.
- Similarly, if there is a high negative covariance, then there is a high linear association between the two variables. But in this case, if one variable increases, then the other will decrease.

In addition to the concept of a covariance matrix, you also learnt how to rotate any vector by an angle of theta using the following transformation matrix.

So, if you want to rotate a particular vector by an angle of θ (in radian) without scaling it up, you can multiply that vector by the following generalised transformation matrix.

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

upGrad

Basis

In this segment, you learnt about another essential building block of the PCA algorithm called **'Basis'**.

Suppose a hospital maintains the health records of patients in terms of temperature, height, weight, heart rate, blood pressure, etc. Let's consider only two parameters, weight and height, for our purpose.

Now, suppose a patient's height is 165 centimetres (cm) and weight is 55 kilograms (kg). In the FPS (foot, pound and second) system of units, you can measure the height as 5.4 feet (ft) and weight as 121.3 pounds (lbs). You would notice here that whether you represent the height in centimetres or feet, the height remains the same. This is what the concept of basis is all about.

Suppose you have the details of the height and weight of 100 patients. You can locate each patient on an x-y plane as a vector, where the x-axis represents weight in 'kg' and the y-axis represents height in 'cm'. So here, the basis will be 'kg-cm'.

Now, suppose you want to locate the height and weight of each patient in the 'ft-lbs' system. For this, you need to have a new axis where 'lbs' is on the horizontal axis' and 'ft' is on the vertical axis. So, here, the basis will be 'lbs-ft'.

Let's consider the patient data set example again, where the height is expressed in 'cm' and the weight, in 'kg'. Suppose a patient's height is 165 cm and weight is 55 kg. You have a vector 'v' that represents this information on an x-y plane where the basis is kg-cm.

$$v = \begin{bmatrix} 165 \\ 55 \end{bmatrix}$$

The vector 'v' can be represented in a linear combination of two basis vectors, b1 and b2:

$$\begin{bmatrix} 165 \\ 55 \end{bmatrix} = 165 \begin{bmatrix} 1\text{ cm} \\ 0\text{ kg} \end{bmatrix} + 55 \begin{bmatrix} 0\text{ cm} \\ 1\text{ kg} \end{bmatrix}$$

Where b1 and b2 are as follows:

$$b_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad b_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

A few basic points regarding basis and basis vectors are given below:

- 'Basis' is a fundamental unit that defines the space in which you express vectors.
- In any dimensional space or matrix, vectors can be represented as a linear combination of basis

vectors.

- The basic definition of basis vectors is that they are a certain set of vectors whose linear combination is able to explain any other vector in that space.
- A space can be defined by infinite combinations of basis vectors.

As you may already know, **1 ft is equal to 30.48 cm** and **1 lbs is equal to 0.45 kg.** So, the height and weight of 165 cm and 55 kg, respectively, can be represented in the following format.

$$\begin{bmatrix} 165 \\ 55 \end{bmatrix} = 165 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 55 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 5.4 \begin{bmatrix} 30.48 \\ 0 \end{bmatrix} + 121.3 \begin{bmatrix} 0 \\ 0.45 \end{bmatrix}$$

Here, 165 cm is equivalent to 5.4 ft, and 55 kg is equivalent to 121.3 lbs. So, you have learnt how to convert the basis from kg-cm to lbs-ft. In the next segment, you will learn how to change the basis.

It is recommended that you do not limit your understanding of the concept basis to a case of unit change. The concept of 'basis' is much broader, and the change of unit is only a part of it. You will get a more detailed understanding of basis in the next segment.

Change of Basis

By now, you have gained a basic understanding of what PCA does. You have also been introduced to the concept of basis. Basis is defined as the fundamental set of vectors, the linear combination of which can represent any data point in that space. In this segment, you learnt how to change the basis from the original to a new one.

Does this seem like something you have already heard of? Well, no. But you saw a similar concept while learning about PCA, where with the rotation of the axis, the covariance values diminished. This change of an axis is known as the change of basis vectors. To understand this in detail, let's consider the same data set of height and weight.

As you saw in the previous example, the height and weight of a patient are 165 cm and 55 kg, respectively, which can be represented as 5.4 ft and 121.3 lbs, respectively, in the ft-lbs basis system. This seems like a change of basis.

As you may already know, **1 ft is equal to 30.48 cm** and **1 lbs is equal to 0.45 kg.** So, the height and weight of 165 cm and 55 kg, respectively, can be represented in the following format.

$$\begin{bmatrix} 165 \\ 55 \end{bmatrix} = 165 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 55 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 5.4 \begin{bmatrix} 30.48 \\ 0 \end{bmatrix} + 121.3 \begin{bmatrix} 0 \\ 0.45 \end{bmatrix}$$

Where 165 cm is equivalent to 5.4 ft, and 55 kg is equivalent to 121.3 lbs.

Let's write the equation given above in a vector format.

You have a vector ‘v’.

$$v = \begin{bmatrix} 165 \\ 55 \end{bmatrix}$$

Where

$$\begin{bmatrix} 165 \\ 55 \end{bmatrix} = a_1 \cdot v_1 + a_2 \cdot v_2$$

$$a_1 = 5.4 \quad a_2 = 121.3$$

And

$$v_1 = \begin{bmatrix} 30.48 \\ 0 \end{bmatrix} \quad v_2 = \begin{bmatrix} 0 \\ 0.45 \end{bmatrix}$$

Let’s represent all of these equations in a **single matrix format** as shown below.

$$\begin{bmatrix} 165 \\ 55 \end{bmatrix} = \begin{bmatrix} 30.48 & 0 \\ 0 & 0.45 \end{bmatrix} \begin{bmatrix} 5.4 \\ 121.3 \end{bmatrix}$$

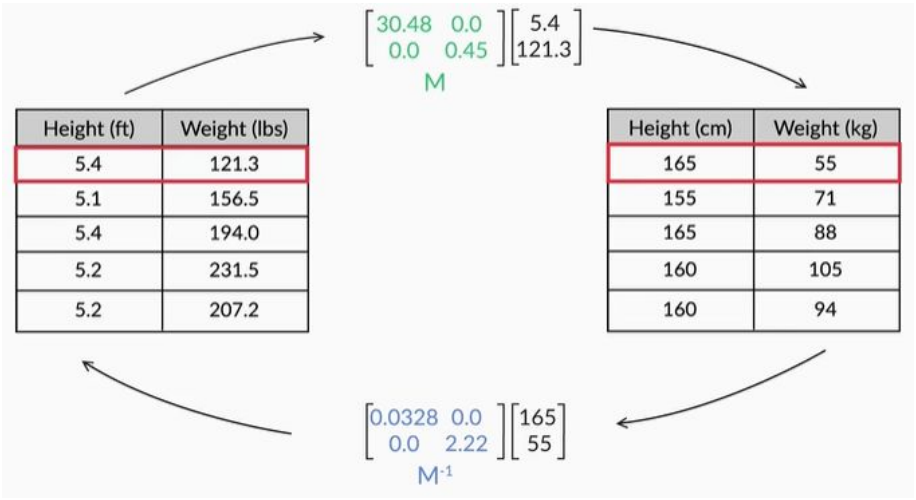
$$\begin{bmatrix} 165 \\ 55 \end{bmatrix} = M \begin{bmatrix} 5.4 \\ 121.3 \end{bmatrix}$$

Here, you can see that **‘M’** is a **change of basis matrix**. Now, if you want to obtain the ft-lbs equivalent of 165 cm and 55 kg, then you need to perform the following operation:

$$\begin{bmatrix} 5.4 \\ 121.3 \end{bmatrix} = \begin{bmatrix} 30.48 & 0 \\ 0 & 0.45 \end{bmatrix}^{-1} \begin{bmatrix} 165 \\ 55 \end{bmatrix}$$

$$\begin{bmatrix} 5.4 \\ 121.3 \end{bmatrix} = M^{-1} \begin{bmatrix} 165 \\ 55 \end{bmatrix}$$

To generalise the result given above, you can refer to the following image.



In this image, both M and inv(M) are **change of basis matrices** according to which basis system you want to convert from which basis system.

Before moving on to the next discussion, let's understand the inverse of a matrix in brief.

Suppose there is a matrix 'A' and its inverse matrix is 'B'. Then AB = I, where 'I' would be an identity matrix: I = [1, 0], [0, 1]

Keep in mind that the inverse is only possible for a square matrix, i.e. a matrix whose number of rows is equal to the number of columns.

Now, let's go through another example of basis change to understand this in detail.

Let's consider the following example where a vector (4, 3) is represented as a linear combination of two other vectors, (-1, 3) and (2, -1).

$$\begin{bmatrix}4\\3\end{bmatrix} = 2\begin{bmatrix}-1\\3\end{bmatrix} + 3\begin{bmatrix}2\\-1\end{bmatrix}$$

Suppose

$$v = \begin{bmatrix}4\\3\end{bmatrix} \quad v_1 = \begin{bmatrix}-1\\3\end{bmatrix} \quad v_2 = \begin{bmatrix}2\\-1\end{bmatrix}$$

Considering the definition of a basis vector, it can be concluded that because you are able to represent the 'v' vector as a linear combination of v1 and v2, the new basis vectors can be v1 and v2. And the corresponding vector in the new basis vectors will be 'b'.

$$b = \begin{bmatrix}2\\3\end{bmatrix}$$

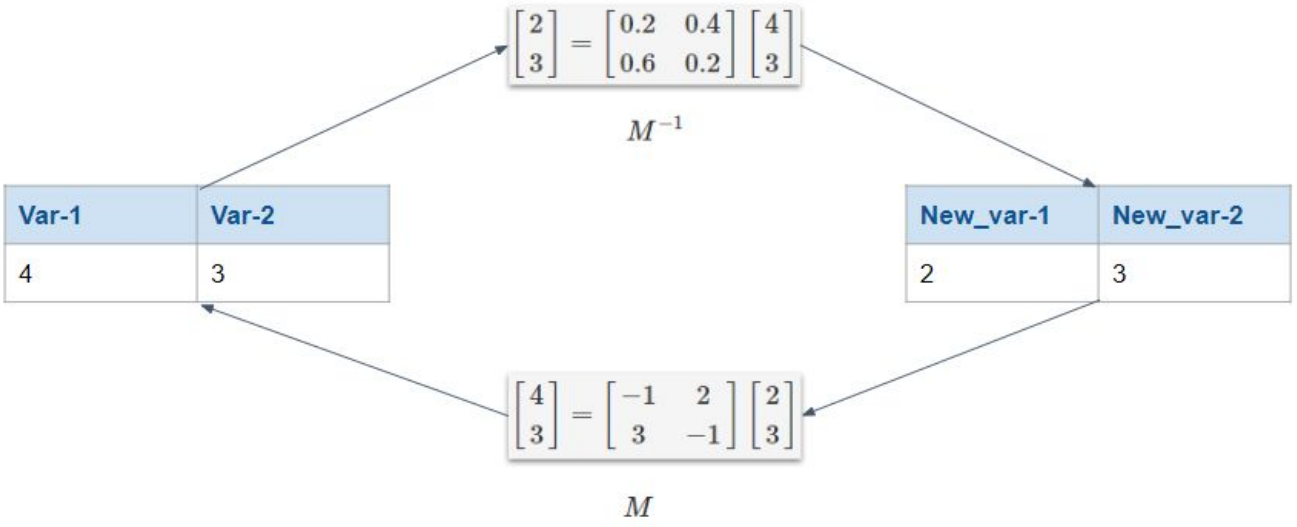
Let's represent the equation given above in a single matrix format.

$$\begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

An important point that you should note is as follows:

Suppose you have a point, say (4, 3), in the x-y / i-j basis and you want to represent the same point in the v1 (-1, 3) and v2 (2, -1) basis. Then you need to represent the (4, 3) point as a linear combination of v1 and v2. So, the coefficient of the linear combination, (2, 3), will be the new representation of the same point in the v1 and v2 basis.

So, you can conclude that the change of basis from one system to another can be done by multiplying the **change of basis matrix** as shown in the image given below.



With this, you have understood the concept of basis change.

upGrad

Summary

PCA Algorithm

This session broadly covered the following topics:

1. Eigenvectors and eigenvalues
2. Diagonalisation of a covariance matrix
3. Eigendecomposition of a covariance matrix
4. PCA algorithm
5. Python demonstration of PCA algorithm
6. Spark MLlib demonstration of PCA

In this session, you learnt about the concepts of eigenvectors and eigenvalues. After getting an understanding of eigenvectors you learnt about the diagonalisation of the covariance matrix.

Eigenvectors and Eigenvalues - I

Until now, you learnt about the basics of vectors and matrices. In this segment, you will learn about certain other concepts such as eigenvectors and the associated eigenvalues, which are required to understand the workings of PCA.

You learnt about the concept of **transformation matrix** in the earlier segments. Let's try to recollect the concept here. When a given matrix is multiplied by a vector, then it simply transforms that particular vector to a new vector.

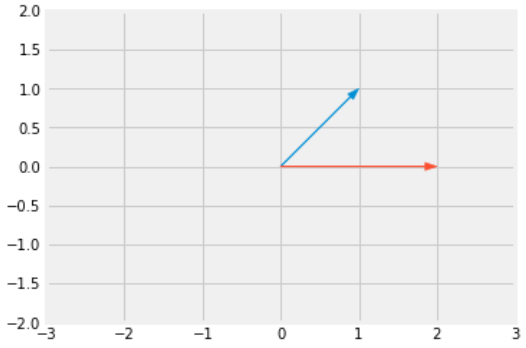
Now, as you saw in the video given above, there is a vector 'x', which is represented as shown below.

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

When you multiply this vector 'x' by matrix 'A', you get the following new, transformed vector 'Ax'.

$$A = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \quad Ax = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

Now, can you see that the transformation matrix 'A' has transformed the given vector 'x' into a new vector 'Ax'? You can visualise these two vectors 'x' and 'Ax' graphically as shown below; the given vector 'x' is shown in blue, and the transformed vector 'Ax' is shown in red.



In the next segment, we will extend the concept of a transformation matrix to learn about eigenvectors.

Eigenvectors and Eigenvalues - II

In this segment, you learnt about eigenvectors and eigenvalues.

Let's assume a transformation matrix 'A', which is shown below.

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

Note: In the x–y plane, the unit magnitude vectors in the direction of the x and y axes are denoted by 'i' and 'j', respectively, which can be represented in the matrix form as shown below.

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \hat{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Now, suppose you have two vectors 'x' and 'i' as shown below.

$$x = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad \hat{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Let's transform these two vectors using the transformation matrix 'A' and examine what the new transformed vectors would be in this case.

$$Ax = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \end{bmatrix} = 2 \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$A\hat{i} = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \end{bmatrix} = 3 \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

An interesting point that you should note here is that when you transform the 'x' and 'i' vectors using the matrix 'A', you get new vectors that are simply the scaled vectors of the original vectors, i.e., the transformed vectors are parallel to the original vectors. At the same time, when you transform the 'x' and 'i' vectors using the transformed matrix 'A', you get the vectors, '2x' and '3i' vectors, respectively.

So, in this case, multiplying the vectors 'x' and 'i' by the matrix 'A' has the same effect as simply multiplying them by the scalars 2 and 3, respectively.

Hence, the vectors 'x' and 'i' are called the **eigenvectors** of matrix 'A', and the scalars '2' and '3', respectively, by which these vectors get scaled are called the **eigenvalues** of matrix 'A'.

In linear algebra, an **eigenvector** of a linear transformation (or a square matrix) is a non-zero vector that changes at most by a scalar factor when that linear transformation is applied to it. The corresponding **eigenvalue** is the factor by which the eigenvector is scaled.

Please note that we are not introducing the mathematical calculation to find out the eigenvectors and eigenvalues of a particular matrix. If you understand the fundamentals of eigenvectors and eigenvalues, then it will be quite easy for you to understand the calculation of the same.

You can refer to the additional reading links given below to know more about the calculation of eigenvectors and eigenvalues.

https://www.youtube.com/watch?v=PFDu9oVAE-g
https://www.mathsisfun.com/algebra/eigenvalue.html

Let's list some interesting properties of eigenvectors:

- Eigenvalues and eigenvectors of a given matrix always occur in pairs.
- Eigenvalues and eigenvectors are defined only for square matrices, and it is not necessary for them to always exist. This means that there could be cases where there are no eigenvectors and eigenvalues for a given matrix, i.e., imaginary eigenvectors and eigenvalues exist.

Diagonalisation of Covariance Matrix

In this segment, you learnt about the application of eigen discussion to diagonalise a covariance matrix using eigenvectors.

But first, you will need to have an understanding of what is actually meant by diagonalisation of a covariance matrix.

In the very first session of this module, you saw the following data set and its covariance matrix.

Salary (1000 X 1₹)	Age (Year)
25	22
34	30
50	43
68	54
80	60
96	75

$$\begin{bmatrix} Change & of \\ Basis & Matrix \end{bmatrix} \begin{bmatrix} 25 \\ 22 \end{bmatrix} = \begin{bmatrix} -1.95503533 \\ 0.04185559 \end{bmatrix}$$

PC1	PC2
-1.95503533	0.04185559
-1.38527443	-0.01898064
-0.42053679	-0.07896334
0.5219234	-0.00357421
1.09768043	0.09923497
2.14124272	-0.03957237

Covariance Matrix

$$\begin{bmatrix} 386.27 & 447.22 \\ 447.22 & 750.57 \end{bmatrix}$$

Covariance Matrix

$$\begin{bmatrix} 2.396 & -1.48367e^{-10} \\ -1.48367e^{-10} & 0.00395 \end{bmatrix}$$

In the first table, the value of covariance is quite high, which means 'Salary' and 'Age' are correlated to each other. But when you represent the same data in a new basis system, you get new, transformed data.

Using these new, transformed data points, the value of the covariance is found to be negligible, that is, almost zero, and the variance is distributed in such a way that PC1 has the highest variance (or information), 2.396, PC2 has the next highest variance, 0.00395.

So, when the diagonal elements of a covariance matrix become 0, it is called a diagonalised covariance matrix. A diagonalised covariance matrix represents non-correlated features, which is the requirement that we need to fulfill using PCA.

So, our task is to transform each original data point to a new basis system such that the covariance matrix of the transformed data set becomes a diagonal matrix in the new basis system.

So, in the process of diagonalisation of the **covariance matrix**, or, in general, **a matrix**, eigenvectors and eigenvalues play a very magical role.

Suppose a matrix 'A' has 'v1' and 'v2' eigenvectors and 'lambda-1' and 'lambda-2' eigenvalues, as shown below.

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

$$v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$\lambda_1 = 3, \lambda_2 = 2$$

Let's define the eigenvector and eigenvalue matrices, 'V' and 'Λ', respectively, as shown below.

$$V = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \Lambda = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$$

When you multiply matrix 'A' by matrix 'V', you get the same results as when you multiply matrix 'V' by matrix 'Λ'.

$$AV = V\Lambda$$

When you **right-multiply** both sides of the equation given above by inv(V), you will get the result as shown below.

$$A = V\Lambda V^{-1}$$

Or when you **left-multiply** both sides of that same equation, you get the result as shown below.

$$V^{-1}AV = \Lambda$$

Now, in this equation, matrix 'Λ' is nothing but a diagonal matrix whose non-diagonal entries are simply 0.

So, based on the analysis above, you can state that:

Both A and Λ represent the same linear transformation but in different basis vectors (i.e., original basis and eigenvector basis, respectively).

Let's try and understand the statement given above.

Suppose you have an original basis, i.e., an x–y plane, and you have a vector 'x' in the same basis. Also, you have a transformation matrix 'A' as shown below.

$$x = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

When you transform the same vector 'x' using the transformation matrix 'A', you get the result as shown below.

$$Ax = \begin{bmatrix} -1 \\ 4 \end{bmatrix}$$

Now, suppose you want to represent vector ‘x’ in a new basis system, which is supposed to be an eigenvector basis this time. This means that you need to represent vector ‘x’ as a linear combination of the eigenvectors of matrix ‘A’. You can represent the linear combination as shown below.

The eigenvectors of ‘A’ are given below.

$$v_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Let’s represent ‘x’ in terms of linear combination of eigenvectors.

$$x = \begin{bmatrix} -1 \\ 2 \end{bmatrix} = 1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 2 \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} = V^{-1}x$$

$$x' = V^{-1}x$$

This means that point x(-1, 2) can be represented as x'(1, 2) in the eigenvector basis, and the change of the basis matrix will be ‘inv(V)’. This is one of the key steps in understanding PCA. Using this, we will proceed to prove that A and capital lambda are transformation matrices but in separate basis system (i.e. eigenvector basis system).

$$V^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}^{-1}$$

Now, think the other way around: Suppose you have vector **x'** in an eigenvector basis and you want to calculate the transformed vector in a new eigenvector basis, A'x', which corresponds to the transformed matrix ‘A’ in the original basis. To do this, you need to first convert x' to x, and then x to Ax, and finally convert Ax to an eigenvector basis system.

So let’s convert x' to x first as shown below.

$$x = Vx'$$

Now, let’s apply the transformation to x, i.e. ‘Ax’.

$$y = Ax = AVx'$$

Now, let’s convert ‘y’, which is ‘Ax’, to an eigenvector basis again, that is, y'.

$$y' = V^{-1}y = V^{-1}AVx'$$

,

Or, in other words:

$$y' = \Lambda x'$$

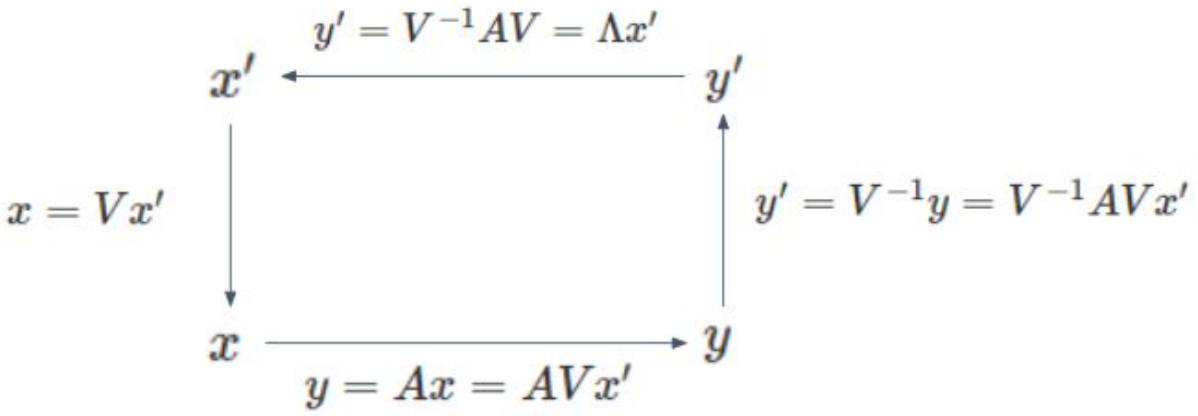
,

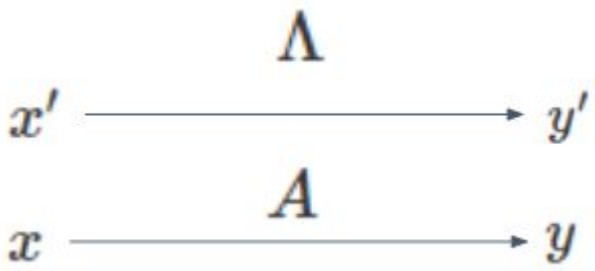
because

$$\Lambda = V^{-1}AV$$

So, as you can see here, A' comes out to be 'Λ', which is the transformation matrix in the new basis system, and 'Λ' is a diagonal matrix.

You can refer to the following summary image to understand the process better.





Therefore, we can say that **both A and Λ represent the same linear transformation but in different basis vectors (i.e., original basis and eigenvector basis, respectively).**

In other words, you can diagonalise matrix 'A' by representing it in a new eigenvector basis system because matrix 'A' will become 'Λ' in the eigenvector basis system.

Now, suppose the 'A' is the **covariance matrix** of the data set in the original basis; for example, let's consider the following data set.

Salary (1000 X 1₹)	Age (Year)
25	**22**
34	**30**
50	**43**
68	**54**
80	**60**
96	**75**

The covariance matrix of this data set would be as shown below.

$$\begin{bmatrix} 386.27 & 447.22 \\ 447.22 & 750.57 \end{bmatrix}$$

Now, when you apply the following operation on the covariance matrix, you will get a diagonalised covariance matrix, as shown below.

$$\text{Diagonalized Covariance Matrix} = V^{-1}AV$$

Here, 'V' is the eigenvector matrix.

Thus, a very important point that you learnt here is that you can diagonalise a covariance matrix only when you represent the data points in an eigenvector basis system.

Eigendecomposition of a covariance matrix: Eigendecomposition of covariance matrix refers to finding the eigenvectors of the covariance matrix so that you can represent all the data points of the original basis system in the eigenvector basis system.

With this, you have a broad understanding of the concept of PCA algorithm. So, the overall algorithm of PCA can be summarised as follows:

- The original data points may not be uncorrelated to each other. Therefore, one of the ways to tackle this problem is to drop the least significant column. The other way is to generate the new features, the PCs that are uncorrelated to each other. Hence, you first calculate the covariance matrix of the original data points and try to diagonalise it, which means that they should have near-to-zero covariance.
- The diagonalisation of the covariance matrix happens when you represent all the original data points in the eigenvector basis.
- The eigenvalues along the diagonal will be of the covariance matrix of the original data points.

upGrad

PCA Algorithm

In this segment, you applied all your learnings so far and understand the PCA algorithm.

But first, let's revise the points given below, which form the basis of the PCA algorithm.

PCA converts possibly correlated variables to 'Principal Components' such that:

- They are uncorrelated to each other,
- They are linear combinations of the original variables, and
- They capture maximum information in the data set.

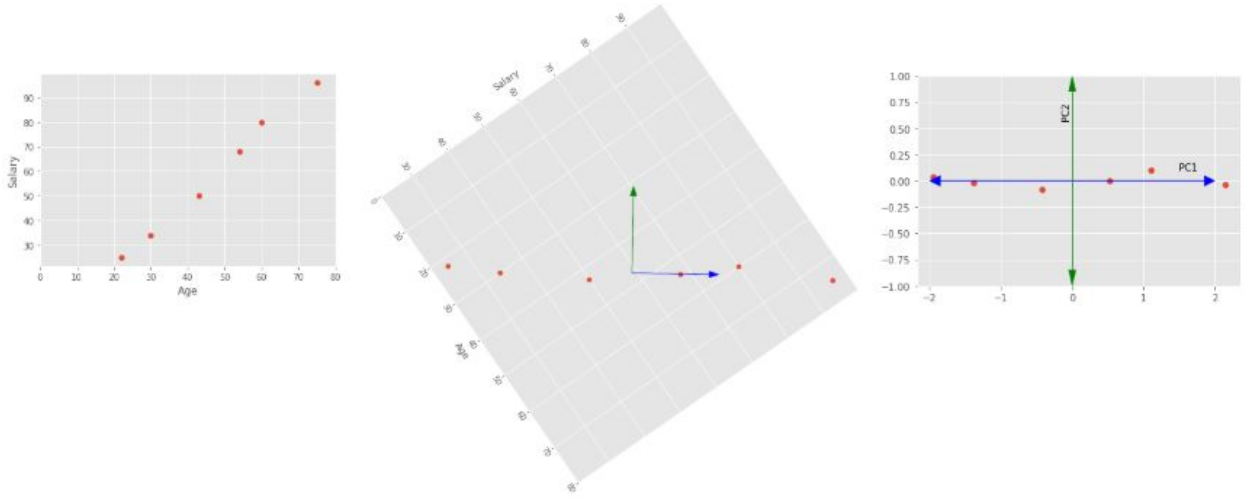
The uncorrelated features can be obtained only when you diagonalise the covariance matrix.

Let's summarise the steps of the PCA algorithm one by one:

- Suppose you have an original data set with **'M' rows and 'N' columns**. Then you will have a **covariance matrix of order 'N X N'** for this data set.
- After getting the covariance matrix for the data set, find its eigenvectors, which will be the new basis vectors.
- Once you have obtained the eigenvectors, simply arrange them in the form of a matrix. This eigenvector matrix will be the '**change of basis matrix'**.
- After getting the **'change of basis matrix'**, multiply each data point in the original basis system by the inverse of the **'change of basis matrix'** to get the new data points in the **eigenvector basis system**.
- In this way, you represent all the data points of the original basis system in the eigenvector basis system so that you can obtain the uncorrelated features, as the covariance matrix has now been diagonalised.

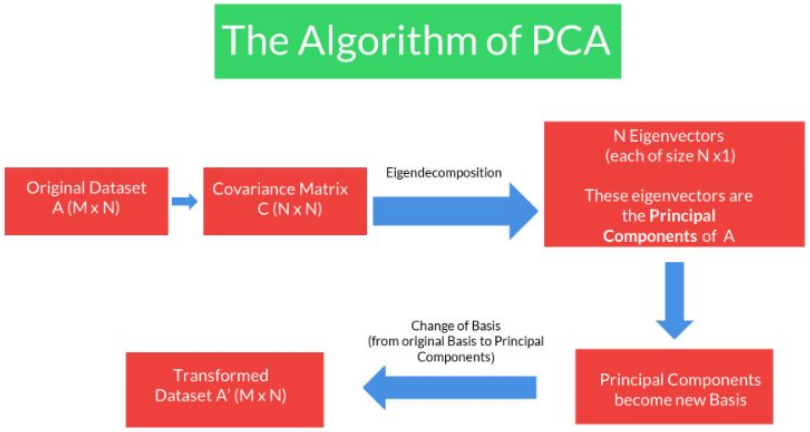
One point that you should remember is that the **'eigenvectors of a covariance matrix are the new directions in which you represent the data in order to get uncorrelated features; or, in other words, the eigenvectors are the new principal components'.**

So, if you look back at where we started our discussion on PCA using the 'Salary' and 'Age' example.



You find the maximum variance along the blue axis. The blue and green lines are the eigenvectors of the covariance matrix.

The image given below summarises the PCA algorithm.



In the next two segments, you saw hands-on coding demonstrations of the PCA algorithm using Python and Spark.

PCA Python Demonstration - I

Welcome to the hands-on demonstration of the PCA algorithm in Python. In the earlier segments, you learnt about the concept of the PCA algorithm. In this segment, you learnt how to implement the PCA algorithm in Python using the **'sklearn'** Python library. You will be implementing PCA on the 'iris' data set in Python.

So, let's get an understanding of the 'iris' data set first.

The 'iris' data set consists of 50 samples from each of the three species of Iris (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Four features, the lengths and the widths of the sepals and petals, were measured for each sample, in centimetres.

So, the 'iris' data set contains the following five columns:

1. Sepal length (cm)
2. Sepal width (cm)
3. Petal length (cm)
4. Petal width (cm)
5. Species type

In this demonstration, we will first build the PCA algorithm, step by step, and after that, you will build the PCA algorithm again using the 'pca' package of the 'sklearn' library. Once you obtain both the results, you will see that both the procedures produce the same results.

So, let's start the step-by-step process of the principal component analysis. The first step in PCA is to **normalise the data set and find the covariance matrix** of the data. Let's start the Python demonstration by loading the data and finding its covariance matrix.

Let's understand the steps implemented, one by one.

1. In the very first part of the code, you need to import certain useful libraries. Some of the important libraries that you need to understand are as follows:

   from sklearn.datasets import load_iris
   from sklearn.decomposition import PCA

   - The 'iris' data is built in the 'sklearn.dataset' package. You just need to import it and load it into a dataframe or in an array.
   - You need to import the 'PCA' package from the 'sklearn.decomposition' library.

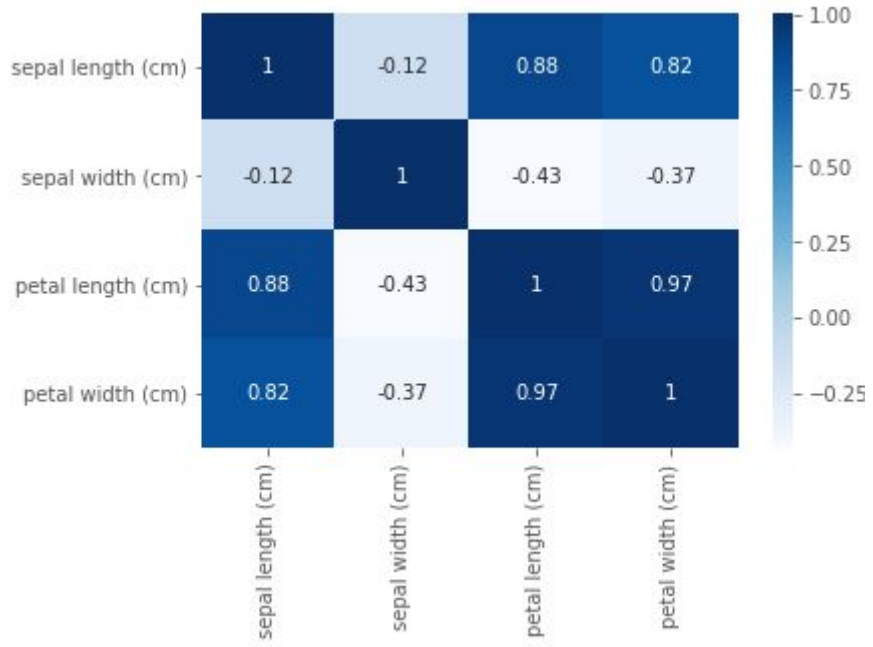
2. Once you have loaded the data into an array X, you need to normalise it. In total, there are four numerical columns corresponding to the sepal and petal lengths and widths. To normalise each numerical column, you need to perform the following steps:

   - First, you need to find the difference between each data point and the mean for each column, and then divide the result by the standard deviation of that particular column:

   - *x_centered = X - X.mean(axis=0)*
     x_scaled = x_centered / x_centered.std(axis=0)

   - In this way, you will get the normalised data into an array, that is, *x_scaled*.
   - You will notice that the mean of this data set is almost equal to 0, and the variance works out to be 1, which is true in the case of a normalised data set.

3. In the next step, you need to find out the covariance matrix of the normalised data set. To find the covariance matrix of the 'iris' data set, you need to first transpose the 'x_scaled' array and then implement the '*np.cov(x_scaled.T)'.*
4. Once you have obtained the covariance matrix of the data set, you need to create its heatmap after converting the covariance matrix array 'C' into a dataframe as shown below.



Here, you can see that there are high covariances of about 0.82, 0.88 or 0.97 between the features.

The next step is to find out the **eigenvectors and eigenvalues of the covariance matrix and represent the original data points in the eigenvector basis system**.

To calculate the eigenvectors and eigenvalues of the covariance matrix, you need to write the following code:

w, v = np.linalg.eig(C)

Here, 'w' will get the eigenvalues, and the array 'v' will get the eigenvectors of the covariance matrix 'C' as shown above.

ix = np.argsort(w)[::-1]
v_sorted = v[:, ix]

The step given above is essential to arrange the eigenvectors in ascending order of their eigenvalues. But why is it an important step? Let's try and understand now.

In the first session of this module, you learnt that PCs are arranged in decreasing order of their variances. The first PC has the maximum variance (information), the next PC has the second highest variance, and so on. Therefore, to arrange the PCs in decreasing order of variance, you need to arrange the eigenvectors in decreasing order of their corresponding eigenvalues.

If you want to dig deeper into the concept of eigenvalues related to PCA, then you need to start by noting an interesting fact. Do you remember that a diagonalize covariance matrix in eigenvectors' basis is nothing but the eigenvalues matrix 'Capital Lambda'?

The diagonalised covariance matrix = Capital Lambda.= $\begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$

So, the variance in each PCs is lambda- 1 and lambda- 2, respectively. You can calculate the percentage of variance in each principal component using the following formula:

The percentage of variance explained by the eigenvector- 1 (PC1), which corresponds to Lambda-1, will be given by the following expression:

Lambda-1 / (Lambda- 1+ Lambda- 2)

The percentage of variance explained by the eigenvector- 2 (PC2), which corresponds to Lambda-2, will be given by the following expression:

Lambda-2 / (Lambda- 1+ Lambda- 2)

And hence, it is necessary to arrange the eigenvectors in decreasing order of their eigenvalues.

Now, the 'iris' data set consists of four variables. Therefore, you will get a total of four PCs in decreasing order of their variances. For now, let's take only two PCs for further analysis.

Next, you need to determine the values of the data points in the newly obtained eigenvector basis system. To do that, you need to multiply the **inverse of the eigenvector matrix** by the **original data points**.

Let's examine the shape of a single row of the scaled 'iris' data set.

	Sepal Length	Sepal Width	Petal Length	Petal Width
Row-1	-9.00681170e-01	1.01900435	-1.34022653	-1.31544430

A single row is a row matrix; but to multiply the data points with the eigenvector matrix, you need to have a column matrix. Therefore, you need to transpose the entire 'x_scaled' array to multiply it by the inverse of the eigenvector matrix.

To calculate the inverse of the 'v_sorted' array, you need to perform the following operation:
v_inv = np.linalg.pinv(v_sorted)

Before moving to the next step, let's understand the concept of the transpose of a matrix.

The transpose of any matrix can be obtained by converting its rows into columns, and vice versa. You will have a better understanding of this using the following example.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Once you get the inverse of the eigenvector matrix, you need to multiply the transpose of each data point in the row format by the inverse of the eigenvector matrix as shown below.

Let's understand the following code.

x_lr = np.dot(v_inv, x_scaled.T).T

In order to understand the transposes we have obtained, let's consider the following two basis vectors, v1 and v2. You want to represent point [a, b] (in an x-y plane) in the new basis system, and then you can use the following formula to get the result.

$$\begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} v_1 & v_2 \end{bmatrix}^{-1} \begin{bmatrix} a \\ b \end{bmatrix}$$

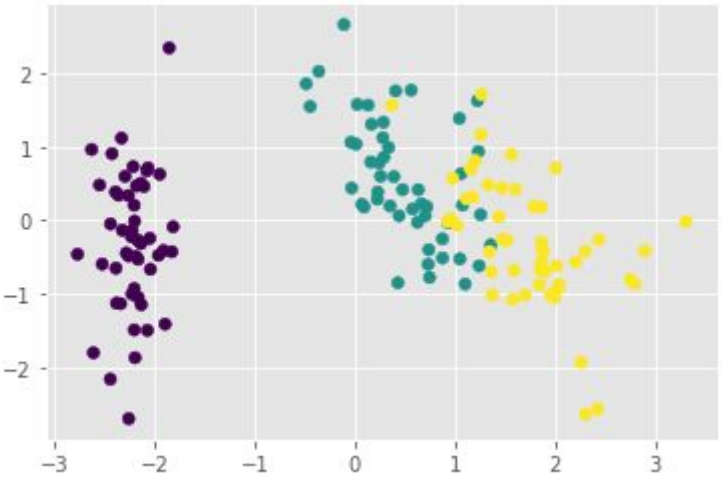
Here, [c, d] is the representation of point [a, b] in the v1 and v2 basis. Note that in the numpy format, [a, b] and [c, d] are represented as row vectors and not column vectors. Hence, we need to take two transposes.

- So, each row of the data set is a vector, which is not in the column vector format. So, to convert it into a column matrix from a row matrix, it is important to transpose it. That is why it is important to perform '*x_scaled.T*'.
- The output of *np.dot(v_inv, x_scaled.T)* will be a column matrix, and so, you need to transpose it again to get the results in the row format.

So, finally, we are focusing on two PCs only. Next, to get the reduced data corresponding to the two PCs only, you need to perform following step:

x_lr_reduced = *x_lr[:, :N]*,
where N= 2

When you plot the two PCs on a scatter plot, you will get the graph as shown below.



In this graph, you can see that the three classes of the *Iris* species are indicated with violet, blue and yellow colours. You can see that with two linear separators, you are able to classify most of the data points correctly.

In the next segment, you will perform PCA analysis using the **pca** package of **'sklearn.decomposition'.**

PCA Python Demonstration - sklearn and Scree Plot

In this segment, you learnt how to perform PCA analysis using the pca package of 'sklearn.decomposition'. Let's watch the upcoming video to learn about it from our expert.

So, it is quite easy to perform PCA using sklearn, and in three steps, you get a scatter plot that is similar to the graph that you saw in the previous segment. Let's consider the same iris data set that is stored in 'X'.

#Step-1: To start the PCA process
pca = PCA()

#Step-2: To transform the data into a scaled form
x_lr = pca.fit_transform(X)

#Step-3: To plot the scatter plot of two PCs
plt.scatter(x_lr[:, 0], x_lr[:, 1], c=y)

Now, let's try and understand the concept of scree plots.

It is quite easy to perform PCA using sklearn, and in three steps, you get a scatter plot that is similar to the graph that you saw in the previous segment.

#Step-1: To start the PCA process
pca = PCA()

#Step-2: To transform the data into a scaled form

x_lr = pca.fit_transform(X)

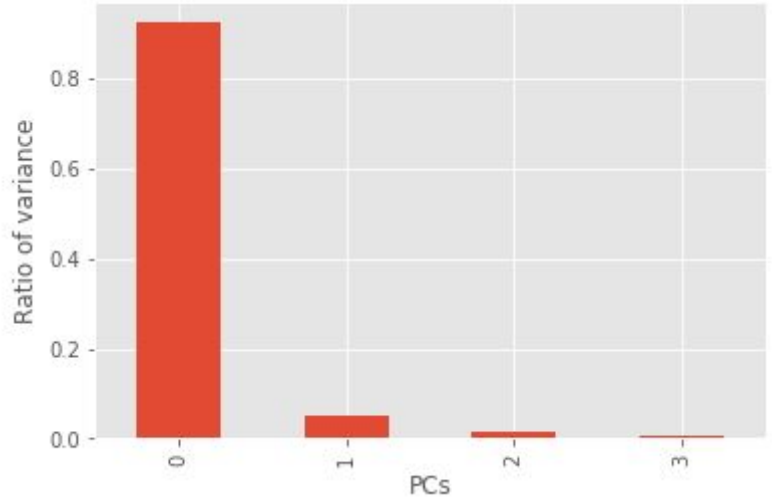
#Step-3: To plot the scatter plot of two PCs
plt.scatter(x_lr[:, 0], x_lr[:, 1], c=y)

Now, let's try and understand the scree plots.

Scree plot: A scree plot visually depicts the percentage of variance for each PC using a histogram.

To make a scree plot of a PCA analysis, you need to simply write the following lines of code. Here, you will get the ratio of the variances of each PC.

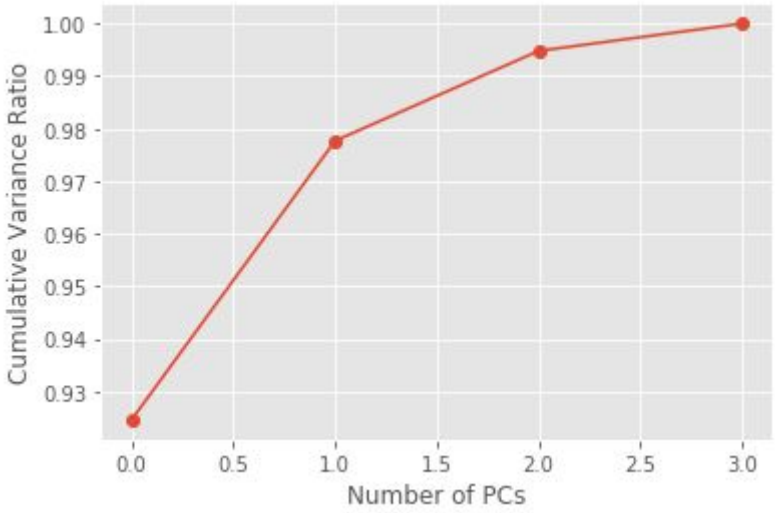
pd.Series(pca.explained_variance_ratio_).plot(kind='bar')
plt.xlabel('PCs')
plt.ylabel('Variance')



In the scree plot given above, you can see that about 90% of the variance is contained in PC0, while the rest is contained in PC1, PC2 and PC3.

Now, suppose you want to plot the cumulative variance ratio. To do this, you can simply write the following code:

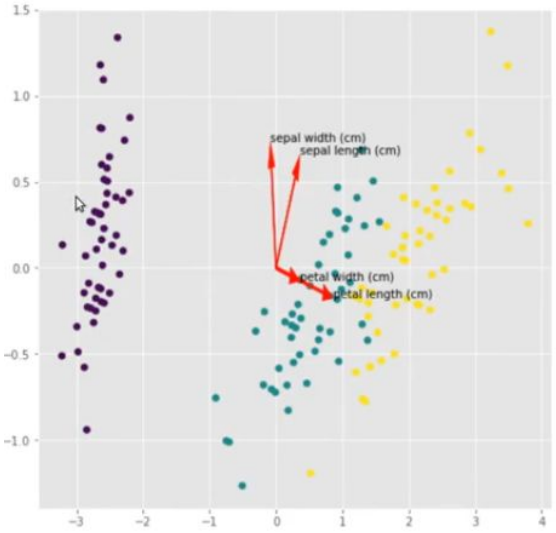
plt.plot(np.cumsum(pca.explained_variance_ratio_), '-o')
plt.xlabel('Number of PCs')
plt.ylabel('Cumulative Variance Ratio')



You can draw quite an interesting insight from the scree plot given above. You can observe that about 98% of the variance can be explained with the help of two PCs only, PC0 and PC1.

Now, let's learn about the concept of biplots.

Biplots are related to loading vectors. Loading vectors show the increasing directions of the original variables on the principal component axes. In this example, the covariance between the petal's length and width is high (about 0.97; please refer to the covariance matrix of the iris data set), and hence, both of them vary in the same direction as depicted in the figure given below.



In this figure, the red vectors are called the loading vectors, and they are the increasing directions of the corresponding features on the PCs.

You will learn about the application of loading vectors in the case study session.

upGrad

PCA Using Spark MLlib

In this segment, you learnt how to perform PCA using Spark MLlib.

For now, let's start with a smaller data set, the 'wine' data set. This data set is the result of a chemical analysis of wines produced in Italy. The analysis determined the quantities of 13 constituents found in each of the three types of wines.

The first step is to create the Spark dataframe from the Pandas dataframe as shown below.

```
df = sqlContext.createDataFrame(df)
```

To get the RDD of the Spark dataframe, you need to write the following code:

```
rdd = df.rdd
```

RDDs are not matrices; so, you need to convert them to row matrices. To do this, you need to import 'RowMatrix'.

```
from pyspark.mllib.linalg.distributed import RowMatrix
```

Row matrices are those matrices whose rows are distributed among multiple clusters of the same machine.

So, to convert the RDDs into a matrix, you need to first convert each row into vectors, and then from the vectors, convert into a matrix whose rows will be distributed on multiple executors of the same machine:

```
from pyspark.mllib.linalg import Vectors
vectors = rdd.map(Vectors.dense)
matrix = RowMatrix(vectors)
```

In the next step, you need to define the number of PCs that you want to get. To do this, you need to first get all the PCs of the 'wine' data set as shown below.

```
pc = matrix.computePrincipalComponents(len(df.columns))
```

Once you have printed 'pc', you will get a 13 X 13 matrix of all the PCs.

Now, to project the original data points on the PCs, you need to multiply the data by the PCs as shown below.

```
matrix_reduced = matrix.multiply(pc)
```

From this step, you will get the 'matrix_reduced' matrix, which is nothing but the projection of all the data points on the PCs. This matrix has a total of 13 columns, and the total number of rows is the same as that in the original data set.

You may be wondering: Why are we doing ***matrix.multiply(pc)*** instead of multiplying the ***inv(pc)*** by the transpose of the ***matrix*** as we did in our earlier discussions?

Let's learn the concept of orthogonality of a matrix to understand this better by solving the questions given below.

The first step is to create the Spark dataframe from the Pandas dataframe as shown below.

df = sqlContext.createDataFrame(df)

Let's normalise the data set first before moving further.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler

assembler = VectorAssembler(inputCols=df.columns, outputCol='features')
scaler = StandardScaler(inputCol='features', outputCol='normFeatures', withMean=True)

df = assembler.transform(df)
scalerModel = scaler.fit(df)
df = scalerModel.transform(df)
```

To get the RDD of the Spark dataframe, you need to write the following code:
rdd = df.rdd

RDDs are not matrices, and so, you need to convert them to row matrices. To do this, you need to import 'RowMatrix'.

from pyspark.mllib.linalg.distributed import *RowMatrix*

Row matrices are those matrices whose rows are distributed among multiple clusters of the same machine.

So, to convert the RDDs into a matrix, you need to first convert each row into vectors, and then from the vectors, convert into a matrix whose rows will be distributed on multiple executors of the same machine:

```
from pyspark.mllib.linalg import Vectors
vectors = rdd.map(Vectors.dense)
matrix = RowMatrix(vectors)
```

Now, in the next step, you need to define the number of PCs that you want to get. To do this, you need to first get all the PCs of the 'wine' data set, as shown below:

pc = matrix.computePrincipalComponents(len(df.columns))

Once you have printed 'pc', you will get a 13 X 13 matrix of all the PCs.

Now, to project the original data points on the PCs, you need to multiply the data with the PCs, as shown below:

matrix_reduced = matrix.multiply(pc)

From this step, you will get the 'matrix_reduced' matrix, which is nothing but the projection of all the data points on the PCs. This matrix has a total of 13 columns, and the total number of rows is the same as that in the original data set.

Don't you have doubt that why are we doing ***matrix.multiply(pc)*** instead of multiplying the ***inv(pc)*** by the transpose of the ***matrix*** that we have done in our entire earlier discussions?

Let's briefly learn the concept of orthogonality of a matrix to understand this better by solving the below questions

1. Suppose you are given a square matrix 'A' and its inverse matrix 'B' as shown below.

$$A = \begin{bmatrix} 3/7 & 2/7 & 6/7 \\ -6/7 & 3/7 & 2/7 \\ 2/7 & 6/7 & -3/7 \end{bmatrix} \quad B = \begin{bmatrix} 3/7 & -6/7 & 2/7 \\ 2/7 & 3/7 & 6/7 \\ 6/7 & ? & -3/7 \end{bmatrix}$$

What is the missing entry in matrix 'B'?

   A. 3/7
   B. 4/7
   C. 2/7
   D. -2/7

Answer: C.
Feedback: Simply multiply matrix 'A' by matrix 'B' and compare the result with the identity matrix 'I' (3/7)(-6/7) + (2/7)(3/7) + (6/7)(?) = 0, and you get the value 2/7.

2. What is the result when you transpose the matrix 'A'?

$$A = \begin{bmatrix} 3/7 & 2/7 & 6/7 \\ -6/7 & 3/7 & 2/7 \\ 2/7 & 6/7 & -3/7 \end{bmatrix}$$

   A. The transpose of the matrix is equal to the matrix 'A'.
   B. The transpose of the matrix is equal to the inverse of the matrix A.
   C. Cannot be determined

Answer: B.

So, in the example given, you can see that the transpose of matrix 'A' is equal to its inverse matrix. This is one of the important properties of orthogonal matrices. So **if you have an orthogonal matrix 'A', then:**

$$A^T = A^{-1}$$

Now, one of the very important facts that you should keep in mind is that **the eigenvector matrix is also an orthogonal matrix. An orthogonal matrix is a matrix whose columns and rows are perpendicular to each other.** In other words, the pairwise dot product of the column vectors is 0; same for the rows. This helps us realise that the vectors in the eigenvector matrix are perpendicular or orthogonal to each other. This is one interesting property that you should remember about eigenvectors. We will not discuss its derivation here.

Suppose you have a data point 'x' in the '**row vector**' format and you want to find its corresponding point in the eigenvector basis and get the result in the '**row matrix**' format. For this, the operation that you will perform will be as follows:

$$x' = [v_1 \quad v_2]^{-1} x^T$$

Here, v1 and v2 are the eigenvectors of the covariance matrix. This will give the x' in the column matrix format. To get the x' in the row matrix format, you need to transpose it again.

$$xfinal = (x')^T$$

The x' is given as follows:

$$x' = [v_1 \quad v_2]^{-1} x^T$$

Let's try to find the transpose of x'.

$$([v_1 \quad v_2]^{-1} x^T)^T = ([v_1 \quad v_2]^T x^T)^T = x[v_1 \quad v_2]$$

Since the eigenvector matrix is an orthogonal matrix, its transpose is equal to its inverse. Please keep in mind that $(AB)^T = B^T A^T$.
So, you can directly perform the following step without finding the inverse or transpose. Pretty interesting, right?

You can refer to the wikipedia page of the orthogonal matrix to understand it better: https://en.wikipedia.org/wiki/Orthogonal_matrix#:~:text=In%20linear%20algebra%2C%20an%20orthogonal,is%20the%20identity%20matrix.

Now, let's convert the 'matrix_reduced' matrix to a numpy array and make the scatter plot of the first two PCs using the following lines of code:

```
import numpy as np
x_red = np.array(matrix_reduced.rows.collect())
import matplotlib.pyplot as plt
plt.scatter(x_red[:, 0], x_red[:, 1], c=wine.target)
```

In this way, you can perform PCA using Spark MLlib.

upGrad

Summary

Movie Recommendation Using PCA

In this session, you are introduced to an end-to-end case study on PCA. This case study is about a movie recommendation system, where the focus will be on reducing the dimensions of the movie tags (or genres). You will have a sufficiently large data set and have hands-on experience of Spark to perform PCA for big data.

The broad flow of this case study is as follows:

1. Describing the movie data set
2. Building a PCA model
3. Selecting the required number of principal components (PCs)
4. Building a movie recommendation system using the nearest neighbour technique

Data Description

Let's begin the case study on recommendation systems by describing the data set that will be used throughout the session to build an end-to-end movie recommendation system. Note that the process described here is one of the ways to provide recommendations. There are many ways to create a recommendation system, one of which you saw earlier, using the ALS algorithm.

The size of the data set is about 250MB. It consists of about 13,000 movies in rows and about 1,000 tags in columns. Let's first learn about the meaning of 'tags' using an example of a movie, which is given below.

Suppose there is a movie named *Conjuring*, which belongs to the horror genre. Let's say you have three tags for this movie: horror, romantic and action. Now, a score out of one corresponds to each tag, say 0.9 for horror, 0.05 for action and 0.05 for romantic, which represents the weightage of each tag for the movie 'Conjuring'.

Similarly, there are 1,000 tags corresponding to the 13,000 movies in the data set, and each tag carries a specific score out of one.

upGrad

Normalisation of Data

Data normalization is the first step before jumping onto the PCA model.

Let's go through the code one by one.

- In the earlier sessions, you learnt about the following lines of code to perform the scaling step in a dataframe.

```
assembler = VectorAssembler(inputCols=[c for c in df.columns if c != 'title'], outputCol='features')
scaler = StandardScaler(inputCol='features', outputCol='normFeats', withMean=True)
df = assembler.transform(df)
scalerModel = scaler.fit(df)
df = scalerModel.transform(df)
```

So, here, the first column, 'title', needs to be removed from the scaling, as this column includes the movie titles.

- You saw that after performing the step given above, you got an error message. This is because some column names contain a dot ('.'), and hence, they have to be rectified. You can apply the following lines of code to replace each dot with an underscore ('_').

```
newCols = []

for c in df.columns:
    if "." in c:
        new_column = c.replace('.', '_')
        df = df.withColumnRenamed(c, new_column)
        newCols.append(new_column)
    else:
        newCols.append(c)
```

Here, you have created a list object named 'newCols' and appended all the column names, one by one, to this list. So, now, if a column name contains a dot, then the dot will be replaced by an underscore, and the 'newCol' list will be appended to the updated column name.

Now, if you perform the normalisation step, there will not be any error, and normalised columns will be stored in *'normFeats'*.

upGrad

PCA Model - I

In the previous segment, you normalised your data set. In this segment, you learnt how to build a PCA model.

Let's discuss the steps one by one:

- After performing the normalisation step, you need to read the data into RDDs.

  rdd = df.select('normFeats').rdd

- Then, you need to import **'RowMatrix'** and **'Vectors'** from the **MLlib** library of PySpark.

  from pyspark.mllib.linalg.distributed import RowMatrix
  from pyspark.mllib.linalg import Vectors

- Next, you need to convert the RDD data into vectors and then convert the vectors into the matrix form. You learnt about this particular step in the previous session, where you performed the PCA analysis using MLlib on a smaller data set.

  vectors = rdd.map(Vectors.dense)
  matrix = RowMatrix(vectors)

  And finally, you need to have a matrix for the entire normalised data set, where each row contains the name of a movie and each column contains a tag.

- In the next step, you will build a PCA model using the following code. Note that you can consider only two principal components in the code given below.

  pc = matrix.computePrincipalComponents(2)

- Once you get the principal components, you need to multiply the 'matrix' with the principal components to get the corresponding points in the principal component basis system.

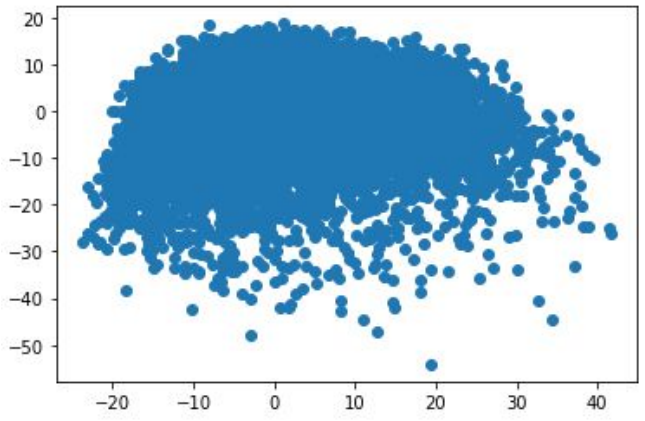
  matrix_reduced = matrix.multiply(pc)

So, the 'matrix_reduces' will be in the order of **Number of movies X 2,** as you can consider only two principal components.

- In the next step, you need to plot a scatter plot for all the data points of these two principal components.

  import numpy as np
  x_red = np.array(matrix_reduced.rows.collect())
  import matplotlib.pyplot as plt

```
%matplotlib inline
plt.scatter(*x_red.T)
```



- The plot given above did not offer much insight into the two PCs.

So, in order to get some useful information from the data set, you need to create a biplot.
As there are more than 1,000 features in this particular data set, you cannot plot each vector corresponding to each tag for the movies. So, let's take the top five tags with the maximum variance to plot their loading vectors on this scatter plot.

To get the top five features (or tags), you need to understand the lines of code given below.

```
from pyspark.ml.stat import Summarizer
summarizer = Summarizer.metrics("variance")
variance = df.select(summarizer.summary(df.features))
variance.show()
```

To get a summary of the columns, you can select a class called 'Summarizer' in ml.stat of PySpark. You can pass any statistical moment such as mean, median, variance and standard deviation to get the values corresponding to each column.

In this case, you have passed 'variance' as a summary parameter. The variances of each column will be stored in 'variance'. Now, to unpack this structure, you need to convert it into an RDD.

```
x = variance.take(1)[0]
variance = x.asDict()['aggregate_metrics(features, 1.0)'].asDict()['variance']
ix = variance.toArray().argsort()
ix = ix[::-1]
```

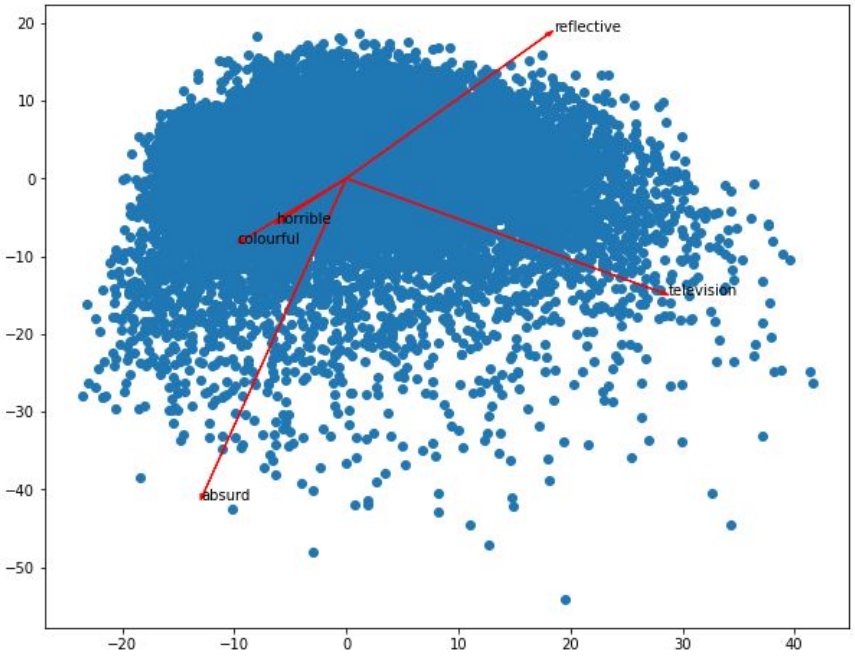
In the first line of code given above, you took the 1st row and 0th element of 'variance' and stored it in the array 'x'. Then, in the second line of code, you converted the variance into the dictionary 'variance'. Next, you sorted the dictionary 'variance' in descending order of the variances corresponding to each column of the movie data set.

Now, to get the first five entries from the sorted dictionary 'variance', let's run the following code.

```
sortedCols = []
for i in ix[:5]:
    sortedCols.append((i, df.columns[i]))
```

Here, in the 'sortedCols' list, you got the variances of the top five columns and their positions in the original data set.

Now, for calculating the variances of these top five columns, you need to plot the loading vectors on the scatter plot of the two principal components. Take a look at the diagram given below.



In the loading vectors diagram given above, which is built only on the scatter plot of the two principal components, you see that the tags 'colourful' and 'horrible' seem to be highly correlated, as the loading vectors of both the tags are in the same direction.

Do you think it is possible for a particular movie to contain both 'horrible' and 'colourful' tags simultaneously?

You will get the answer to this question in the next segment.

PCA Model - II

A model's predictability can be improved upon in a few ways. Recall that in the earlier case study, a lot of feature selection was carried out to improve the model, and this is a good starting point.

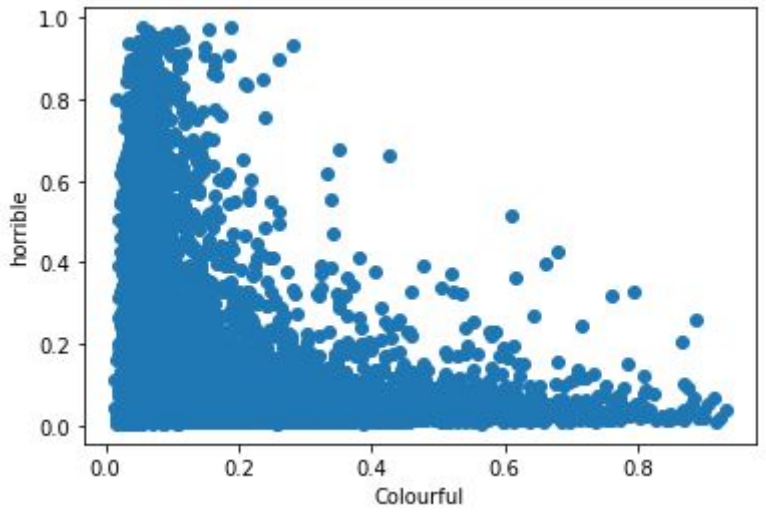
In the previous segment, you saw that after plotting the loading vectors of the top five variance tags, the tags 'colourful' and 'horrible' seemed to be highly correlated.

Now, let's plot the scatter plot of these two tags and understand whether they are actually correlated or not.

Let's extract the two tags 'colourful' and 'horrible' from the original data set.

x = df.select('colourful').rdd
y = df.select('horrible').rdd

Now, let's plot the scatter plot of these two variables and check the correlation between them.



In the plot given above, you can see that as the value of the 'colourful' tag increases, the value of the other 'horrible' tag decreases, which should be true, as a particular movie cannot be colourful as well as horrible simultaneously.

So, when you plotted the loading vectors on the scatter plot of the two PCs, both the loading vectors were in the same direction for both the tags.

Do you think you made a mistake here?
Well, yes. You did make a mistake by plotting in two dimensions, i.e, using only the two principal components. Now, let's take a look at the variance explained by the different principal components.

You already have the projections of all the data points of the two principal components in 'matrix_reduced'. Now, let's convert this matrix into a numpy matrix in the form of an array using the following lines of code.

reduced_rows = matrix_reduced.rows.map(np.array).collect()
reduced_matrix = np.array(reduced_rows)
reduced_matrix

The first line of code creates a numpy array for every row in matrix_reduced using the map function. Hence, reduced_rows is an RDD of numpy arrays, and in order to convert reduced_rows to a numpy array, we execute the second line of code.

So, to calculate the percentage of variance explained by the two principal components, you need to write the following lines of code.

*reduced_matrix.var(axis=0).sum() / (len(df.columns) - 3) * 100*

Here, you first need to calculate the variance of the two PCs individually and then apply sum on it. Once you get the total variance of the two principal components, you need to divide it by the sum of the variance in all the PCs, which will be equal to the total number of tags/ features in the data set because the variance of the normalised tags/features will be 1. However, you need to subtract 3 from the df.columns because you have three extra columns in the data set apart from the numerical tags columns, which are as follows:

- title: Corresponding to the movies titles
- Column corresponding to the assembled features
- Column corresponding to the normalized features

You see that you get only 16% of the variance is described by the two PCs, which is very less for any analysis using principal components.

When you select 100 PCs instead of two PCs, then you get a variance of about 67%. But when you select 500 PCs, then the percentage of variance is approximately 91%.

Recommendation System

In this segment, you learnt how to recommend top movies by finding the nearest neighbour for a particular movie using more number of principal components.

Let's start the discussion by considering the total 500 PCs for the analysis and understand the steps given below:

- First, you need to get the projected data points of 500 principal components.
  You can write the following lines of code to get the projected data points of 500 principal components.

  ```
  # get the matrix of projected data points on 500 PCs.
  pc = matrix.computePrincipalComponents(500)
  matrix_reduced = matrix.multiply(pc)

  # Convert 'matrix_reduced' into a numpy array.
  import numpy as np
  X = matrix_reduced.rows.map(np.array).collect()
  X = np.array(X)

  # get the titles of the movies into 'title'.
  titles = df.select('title').toPandas()

  # prepare a single dataframe with 500 columns of newly projected data points.
  ```

```
import pandas as pd
pdf = pd.DataFrame(X, index=titles['title'])
```

- In the next step, you have to apply the nearest neighbour algorithm. Let's consider the following example so that you understand this better.

  Imagine you are standing in an empty room, which is essentially a three-dimensional space. Suppose there are thousands of tennis balls around you, which are stable, meaning they are not moving. Now, can you identify the top five tennis balls that are closest to your eyes? This is what the nearest neighbour algorithm finds. It calculates the Euclidean distance from the point of interest to the points that lie in the n-dimensional space.

You can understand more about the K-Nearest Neighbour algorithm using the following additional links:
https://www.youtube.com/watch?v=HVXime0nQeI

Now, extend the three-dimensional space to the 500 dimensions, and try to find the top five nearest neighbours for a particular movie.

In order to find the top five movies (or the top five nearest neighbours) for a particular movie using two PCs, you can perform the following lines of code:

```
from sklearn.neighbors import NearestNeighbors
n_pcs = 2
nn = NearestNeighbors()
nn = nn.fit(X[:, :n_pcs])
neighbors = nn.kneighbors(pdf.loc['Toy Story (1995)'].values[:n_pcs].reshape(1, -1), return_distance=False)
pdf.index[neighbors.ravel()].tolist()
```

Now, you need to import the 'NearestNeighbours' class from Sklearn. Here, you will find the top five neighbours corresponding to the movie *Toy Story* (1995). Then, you will get the following five movies:

1. *Toy Story* (1995)
2. *Dirty Laundry*
3. *Empire of Dreams*
4. *The Ten Commandments*
5. *The Blood of Heroes*

You saw that no movie is similar to the movie *Toy Story* (1995) when only two principal components are considered.

Now, let's run the same code with 10, 100 and 500 principal components.

When you find the nearest neighbours corresponding to the movie *Toy Story* (1995) using 10 principal components, you get the following top five movies:

1. *Toy Story* (1995)
2. *Finding Nemo*
3. *Monster*
4. *A Bug's Life*
5. *Ratatouille*

Here, you saw that you are getting only animated movies when you consider 10 PCs.

When you find the nearest neighbours corresponding to the movie *Toy Story* (1995) using 100 principal components, you get the following top five movies:

1. *Toy Story* (1995)
2. *Toy Story 2*
3. *Monsters*
4. *Toy Story 3*
5. *A Bug's life*

Here, you saw that you are getting Toy Story 2 and Toy Story 3 as well.

Hence, as the number of principal components increases, the nearest movie prediction gives better results.

So, instead of using thousands of features in the original data set, you can recommend movies to users using only 100 features, which are essentially the principal components. But there is an interesting point to note here. We get more or less good results with 10 PCs as well. Hence, considering a larger number of PCs is not necessary. Keep in mind the following two key points:

1. The variance captured need not be a metric to decide the number of PCs to be used. However, the final output metric, which is the quality of recommendation, should be considered to decide the number of PCs.
2. It is preferred to have a lower number of PCs because as the number of PCs increases, the computational power required increases, too, and one of the primary aims of PCA is to reduce the number of features to be used for future modelling.

With this, you have completed the PCA module.