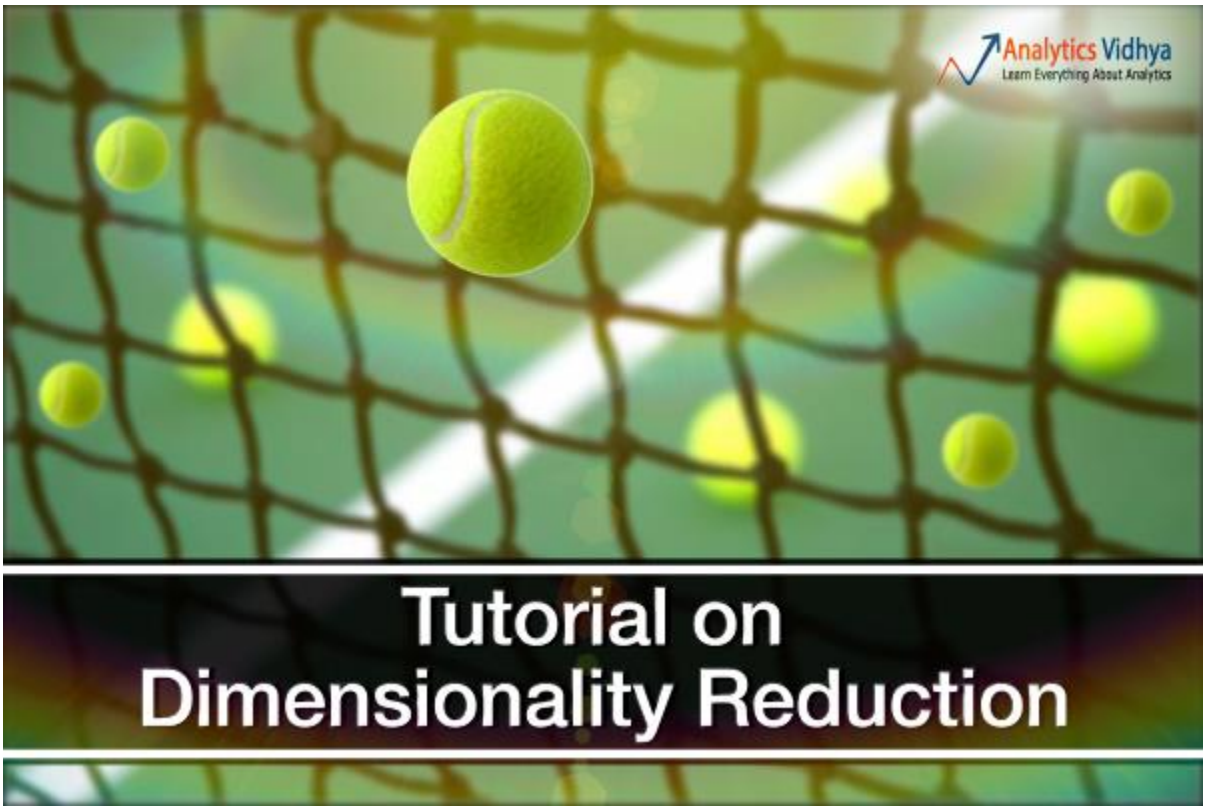
Dimensionality Reduction for Machine Learning

Introduction

Have you ever worked on a dataset with more than a thousand features? How about over 50,000 features? We have, and let us tell you it's a very challenging task, especially if you don't know where to start! Having a high number of variables is both a boon and a curse. It's great that we have loads of data for analysis, but it is challenging due to size.

It's not feasible to analyze each and every variable at a microscopic level. It might take us days or months to perform any meaningful analysis and we'll lose a ton of time and money for our business! Not to mention the amount of computational power this will take. We need a better way to deal with high dimensional data so that we can quickly extract patterns and insights from it. So how do we approach such a dataset?



Using dimensionality reduction techniques, of course. You can use this concept to reduce the number of features in your dataset without having to lose much information

and keep (or improve) the model's performance. It's a really powerful way to deal with huge datasets, as you'll see in this course.

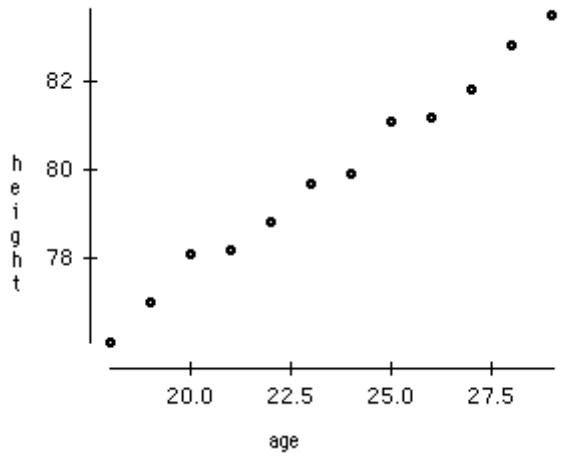
This is a comprehensive guide to various dimensionality reduction techniques that can be used in practical scenarios. We will first understand what this concept is and why we should use it, before diving into the different techniques we have covered. Each technique has it's own implementation in Python to get you well acquainted with it.

What is Dimensionality Reduction?

We are generating a tremendous amount of data daily. In fact, 90% of the data in the world has been generated in the last 3-4 years! The numbers are truly mind-boggling. Below are just some of the examples of the kind of data being collected:

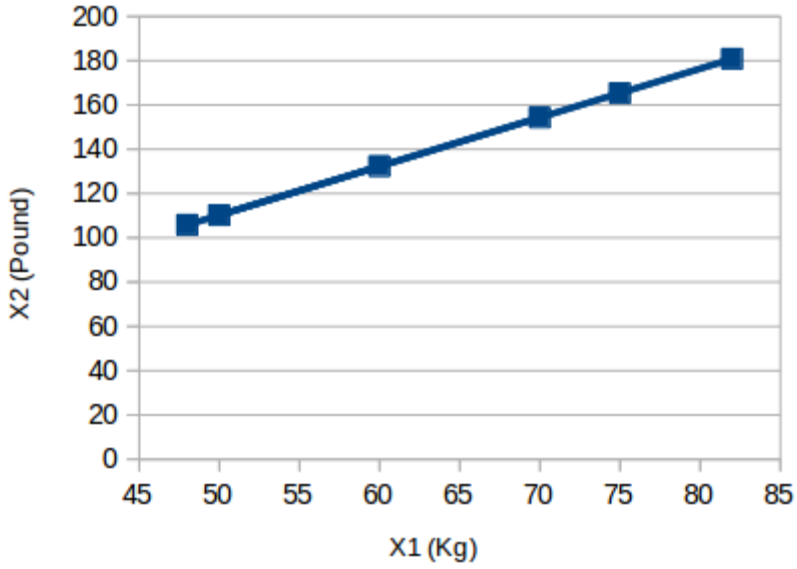
- Facebook collects data about what you like, share, post, places you visit, restaurants you like, etc.
- Your smartphone apps collect a lot of personal information about you
- Amazon collects data of what you buy, view, click, etc. on their site
- Casinos keep a track of every move each customer makes

As data generation and collection keeps increasing, visualizing it and drawing inferences becomes more and more challenging. One of the most common ways of doing visualization is through charts. Suppose we have 2 variables, Age and Height. We can use a scatter or line plot between Age and Height and visualize their relationship easily:

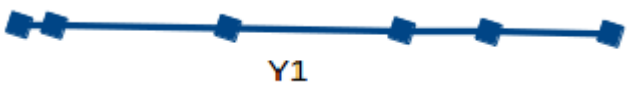


Now consider a case in which we have, say 100 variables (p=100). In this case, we can have 100(100-1)/2 = 5000 different plots. It does not make much sense to visualize each

of them separately, right? In such cases where we have a large number of variables, it is better to select a subset of these variables (p<<100) which captures as much information as the original set of variables.



Here we have weights of similar objects in Kg (X1) and Pound (X2). If we use both of these variables, they will convey similar information. So, it would make sense to use only one variable. We can convert the data from 2D (X1 and X2) to 1D (Y1) as shown below:



Similarly, we can reduce p dimensions of the data into a subset of k dimensions (k<<n). This is called dimensionality reduction.

Why is Dimensionality Reduction required?

Here are some of the benefits of applying dimensionality reduction to a dataset:

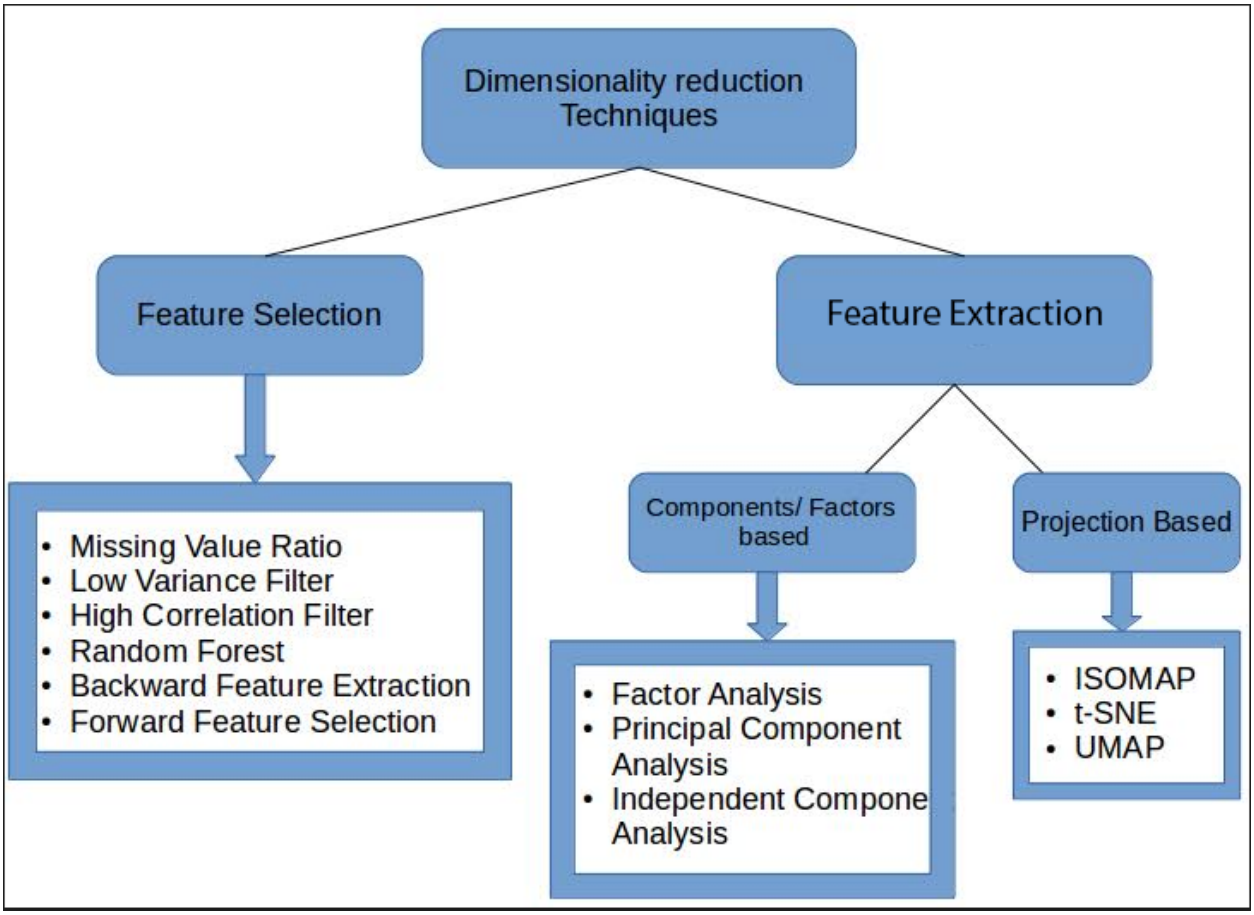
- Space required to store the data is reduced as the number of dimensions comes down
- Fewer dimensions lead to less computation/training time
- Some algorithms do not perform well when we have large dimensions. So reducing these dimensions needs to happen for the algorithm to be useful

- It takes care of multicollinearity by removing redundant features. For example, you have two variables – 'time spent on a treadmill in minutes' and 'calories burnt'. These variables are highly correlated as the more time you spend running on a treadmill, the more calories you will burn. Hence, there is no point in storing both as just one of them does what you require
- It helps in visualizing data. As discussed earlier, it is very difficult to visualize data in higher dimensions so reducing our space to 2D or 3D may allow us to plot and observe patterns more clearly

Common Dimensionality Reduction Techniques

Dimensionality reduction can be done in two different ways:

- By only keeping the most relevant variables from the original dataset (this technique is called **feature selection**)
- By finding a smaller set of new variables, each being a combination of the input variables, containing basically the same information as the input variables (this technique is called **feature extraction**). Feature extraction will be divided into 2 parts:
    - Factor/Component-based method
    - Projection Based Methods



In the next module, we will look at various feature selection techniques and how to implement each of them in Python.

Missing Value Ratio

Feature Selection : Missing Value Ratio

$$\text{Ratio of missing values} = \frac{\text{Number of missing values}}{\text{Total number of observations}} * 100$$

Feature Selection : Missing Value Ratio

Variable	Missing value ratio
ID	0%
season	20%
holiday	30%
workingday	30%
weather	30%
temp	0%
atemp	0%
humidity	20%
windspeed	90%
count	0%

Feature Selection : Missing Value Ratio

Decide a threshold 70%

Feature Selection : Missing Value Ratio

Guideline : Can drop a variable having missing value ratio more than 60 - 70%

Feature Selection : Missing Value Ratio

- Find out the reason for these missing values
  - Non-response
  - Error in data collection
  - Error in Reading Data

Feature Selection : Missing Value Ratio

- Find out the reason for these missing values
- Impute missing values
  - Mean
  - Median
  - Mode
  - Model

Low Variance Filter

Feature Selection : Low Variance

ID	season	holiday	workingday	weather	f5	temp	atemp	humidity	windspeed	count
AB101	1	0	0	1	7	9.84	14.395	81	0.0000	16
AB102	1	0	0	1	7	9.02	13.635	80	0.0000	40
AB103	1	0	0	1	7	9.02	13.635	80	0.0000	32
AB104	1	0	0	1	7	9.84	14.395	75	0.0000	13
AB105	1	0	0	1	7	9.84	14.395	75	0.0000	1
AB106	1	0	0	2	7	9.84	12.880	75	6.0032	1
AB107	1	0	0	1	7	9.02	13.635	80	0.0000	2
AB108	1	0	0	1	7	8.20	12.880	86	0.0000	3
AB109	1	0	0	1	7	9.84	14.395	75	0.0000	8
AB110	1	0	0	1	7	13.12	17.425	76	0.0000	14

Feature Selection : Low Variance

$$\sigma^2 = \frac{\Sigma(x-\bar{x})^2}{n}$$

Feature Selection : Low Variance

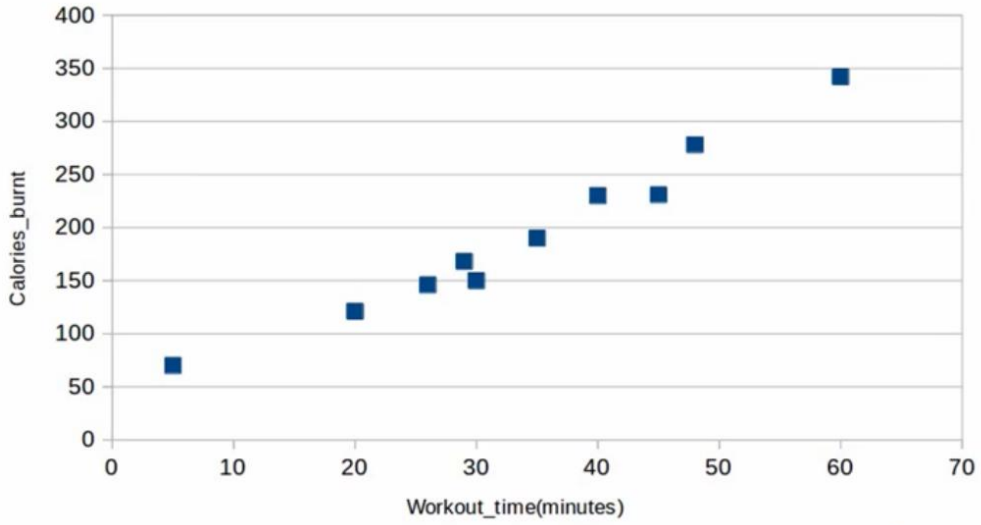
- Decide a threshold variance value
- Drop the variables having less variance than the threshold
  - Normalize the variables before calculating variance

High Correlation Filter

Feature Selection : High Correlation

ID	Workout_time(minutes)	Calories_burnt	Gender	Plays_Sport?	Fitness_Level
1	20	121	M	Yes	Fit
2	40	230	M	No	Fit
3	60	342	F	No	Unfit
4	5	70	M	Yes	Fit
5	48	278	F	Yes	Unfit
6	26	146	M	Yes	Fit
7	29	168	F	No	Unfit
8	45	231	F	Yes	Fit
9	30	150	M	No	Fit
10	35	190	F	No	Fit

Feature Selection : High Correlation



Pearson Correlation = 0.9819

Analytics Vidl

Feature Selection : High Correlation

Correlation

- Determines the relationship between two variables
- Ranges between -1 to 1
- Higher magnitude represents stronger relationship

Feature Selection : High Correlation

- Calculate the correlation between all the independent variables
- Drop the variable, if the correlation crosses a certain threshold (generally 0.5-0.6)
- **Guideline** : Drop the variables which have less correlation with the target variable

Backward Feature Elimination

Feature Selection : Backward Feature Elimination

- No missing values in the dataset
- Variance of the variables is high
- Low correlation between the independent variables

Steps to perform Backward Feature Elimination

1. Train the model using all the variables (n)

ID	Calories_burnt	Gender	Plays_Sport?	Fitness_Level
1	121	M	Yes	Fit
2	230	M	No	Fit
3	342	F	No	Unfit
4	70	M	Yes	Fit
5	278	F	Yes	Unfit
6	146	M	Yes	Fit
7	168	F	No	Unfit
8	231	F	Yes	Fit
9	150	M	No	Fit
10	190	F	No	Fit

Steps to perform Backward Feature Elimination

1. Train the model using all the variables (n)
2. Calculate the performance of the model
3. Eliminate a variable, train the model on remaining variables (n-1)

Steps to perform Backward Feature Elimination

ID	Calories_burnt	Gender	Plays_Sport?	Fitness_Level
1	121	M	Yes	Fit
2	230	M	No	Fit
3	342	F	No	Unfit
4	70	M	Yes	Fit
5	278	F	Yes	Unfit
6	146	M	Yes	Fit
7	168	F	No	Unfit
8	231	F	Yes	Fit
9	150	M	No	Fit
10	190	F	No	Fit

Steps to perform Backward Feature Elimination

ID	Calories_burnt	Gender	Plays_Sport?	Fitness_Level
1	121	M	Yes	Fit
2	230	M	No	Fit
3	342	F	No	Unfit
4	70	M	Yes	Fit
5	278	F	Yes	Unfit
6	146	M	Yes	Fit
7	168	F	No	Unfit
8	231	F	Yes	Fit
9	150	M	No	Fit
10	190	F	No	Fit

Accuracy = 90%

Steps to perform Backward Feature Elimination

1. Train the model using all the variables (n)
2. Calculate the performance of the model
3. Eliminate a variable, train the model on remaining variables (n-1)
4. Calculate the performance of the model on new data
5. Identify the eliminated variable which does not impact the performance much

Steps to perform Backward Feature Elimination

Accuracy using all the variables = 92%

Variable_dropped	Accuracy
Calories_burnt	90%
Gender	91.60%
Plays_Sport?	88%

Steps to perform Backward Feature Elimination

1. Train the model using all the variables (n)
2. Calculate the performance of the model
3. Eliminate a variable, train the model on remaining variables (n-1)
4. Calculate the performance of the model on new data
5. Identify the eliminated variable which does not impact the performance much
6. Repeat until no more variables can be dropped

Forward Feature Selection

Feature Selection: Forward Feature Selection

ID	Calories_burnt	Gender	Plays_Sport?	Fitness_Level
1	121	M	Yes	Fit
2	230	M	No	Fit
3	342	F	No	Unfit
4	70	M	Yes	Fit
5	278	F	Yes	Unfit
6	146	M	Yes	Fit
7	168	F	No	Unfit
8	231	F	Yes	Fit
9	150	M	No	Fit
10	190	F	No	Fit

Steps to perform Forward Feature Selection

ID	Calories_bumt	Gender	Plays_Sport?	Fitness_Level
1	121	M	Yes	Fit
2	230	M	No	Fit
3	342	F	No	Unfit
4	70	M	Yes	Fit
5	278	F	Yes	Unfit
6	146	M	Yes	Fit
7	168	F	No	Unfit
8	231	F	Yes	Fit
9	150	M	No	Fit
10	190	F	No	Fit

Accuracy = 87%

Steps to perform Forward Feature Selection

1. Train n model using each feature (n) individually and check the performance
2. Choose the variable which gives the best performance

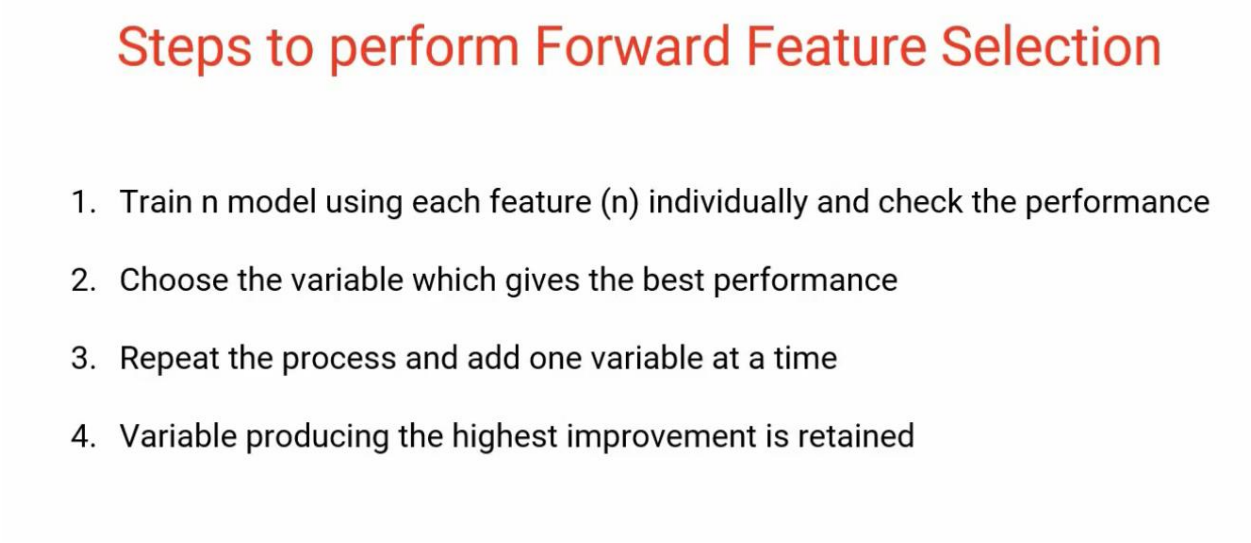
Variable used	Accuracy
Calories_bumt	87.00%
Gender	80.00%
Plays_Sport?	85.00%

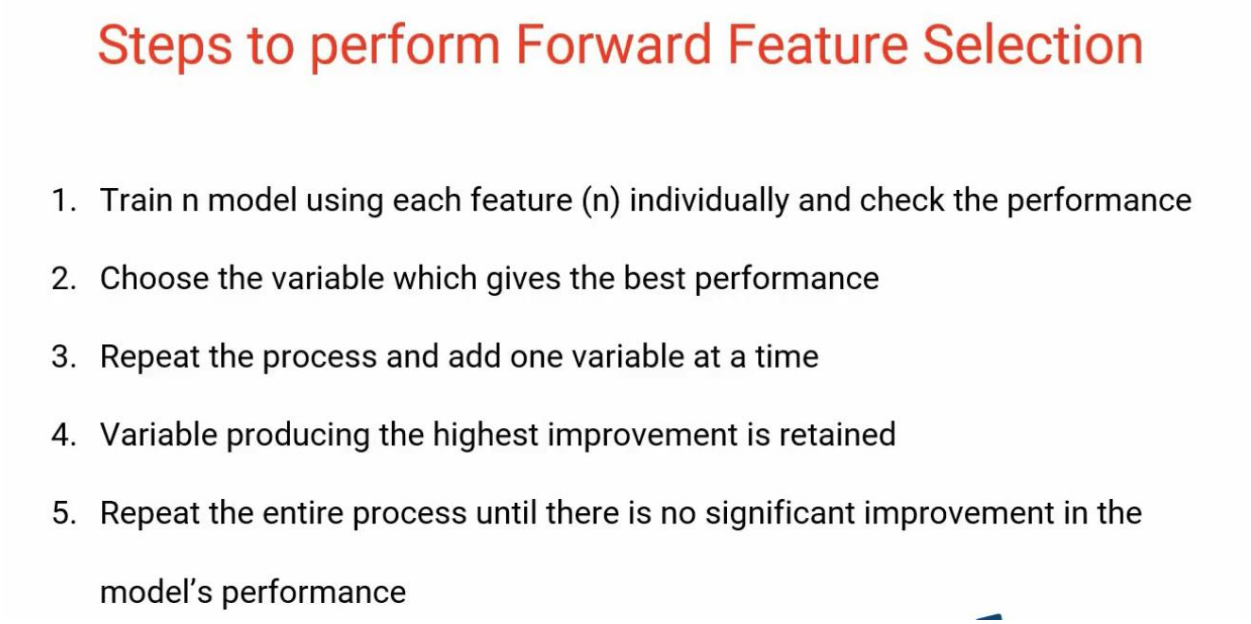
Steps to perform Forward Feature Selection

1. Train n model using each feature (n) individually and check the performance
2. Choose the variable which gives the best performance
3. Repeat the process and add one variable at a time

Steps to perform Forward Feature Selection

ID	Calories_burnt	Gender	Plays_Sport?	Fitness_Level
1	121	M	Yes	Fit
2	230	M	No	Fit
3	342	F	No	Unfit
4	70	M	Yes	Fit
5	278	F	Yes	Unfit
6	146	M	Yes	Fit
7	168	F	No	Unfit
8	231	F	Yes	Fit
9	150	M	No	Fit
10	190	F	No	Fit





Random Forest

Note: We will be using the dataset from AV's [Practice Problem: Big Mart Sales III] (register on this link and download the dataset from the data section)

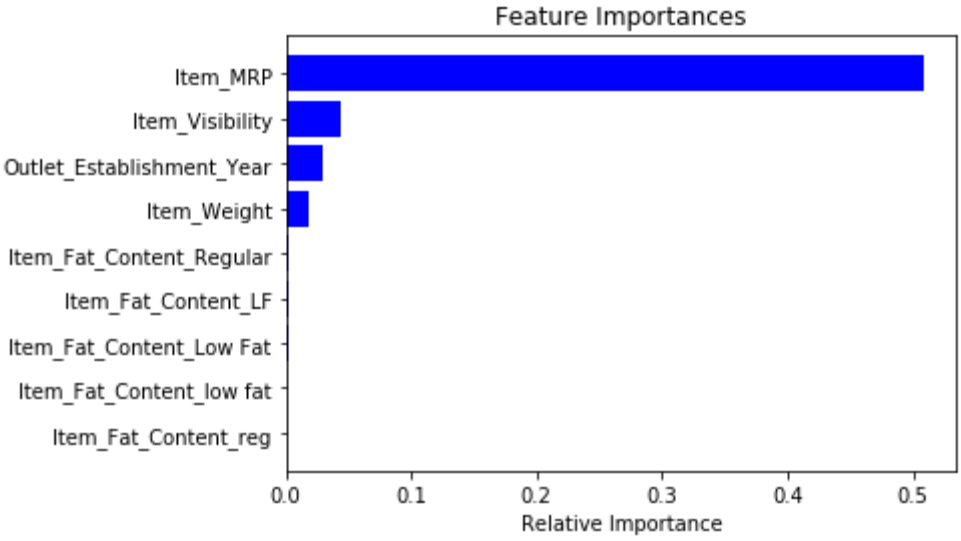
Random Forest is one of the most widely used algorithms for feature selection. It comes packaged with in-built feature importance so you don't need to program that separately. This helps us select a smaller subset of features.

We need to convert the data into numeric form by applying one-hot encoding, as Random Forest (Scikit-Learn Implementation) takes only numeric inputs. Let's also drop the ID variables (*Item_Identifier* and *Outlet_Identifier*) as these are just unique numbers and hold no significant importance for us currently.

```
from sklearn.ensemble import RandomForestRegressor
df=df.drop(['Item_Identifier', 'Outlet_Identifier'], axis=1)
model = RandomForestRegressor(random_state=1, max_depth=10)
df=pd.get_dummies(df)
model.fit(df,train.Item_Outlet_Sales)
```

After fitting the model, plot the feature importance graph:

```
features = df.columns
importances = model.feature_importances_
indices = np.argsort(importances)[-9:]  # top 10 features
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='b',
align='center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Based on the above graph, we can handpick the top-most features to reduce the dimensionality in our dataset. Alternatively, **we can use the *SelectFromModel* of *sklearn* to do so**. It selects the features based on the importance of their weights.

```
from sklearn.feature_selection import SelectFromModel
feature = SelectFromModel(model)
Fit = feature.fit_transform(df, train.Item_Outlet_Sales)
```

Factor Based Feature Extraction Techniques

Introduction to the Module

Feature Extraction is the technique where you extract new features using the existing one.

In this module, we will look at some prominent **Factor Based** feature extraction techniques based on along with its implementation in python.

Please note- For this module, we will be working with the Fashion MNIST dataset, which consists of images belonging to different types of apparel, e.g. T-shirt, trousers, bag, etc. **The dataset can be downloaded from the "IDENTIFY THE APPAREL" practice problem.**

Factor Analysis

Suppose we have two variables: Income and Education. These variables will potentially have a high correlation as people with a higher education level tend to have significantly higher income, and vice versa.

In the Factor Analysis technique, variables are grouped by their correlations, i.e., all variables in a particular group will have a high correlation among themselves, but a low correlation with variables of other groups (s). Here, each group is known as a factor. These factors are small in number as compared to the original dimensions of the data. However, these factors are difficult to observe.

Let's first read in all the images contained in the train folder:

```
import pandas as pd
import numpy as np
from glob import glob
import cv2
images = [cv2.imread(file) for file in glob('train/*.png')]
```

NOTE: You must replace the path inside the glob function with the path of your train folder.

Now we will convert these images into a *NumPy* array format so that we can perform mathematical operations and also plot the images.

```
images = np.array(images)
images.shape
```

(60000, 28, 28, 3)

As you can see above, it's a 3-dimensional array. We must convert it to 1-dimension as all the upcoming techniques only take 1-dimensional input. To do this, we need to flatten the images:

```
image = []
for i in range(0,60000):
    img = images[i].flatten()
    image.append(img)
image = np.array(image)
```

Let us now create a data frame containing the pixel values of every individual pixel present in each image, and also their corresponding labels (for labels, we will make use of the *train.csv* file).

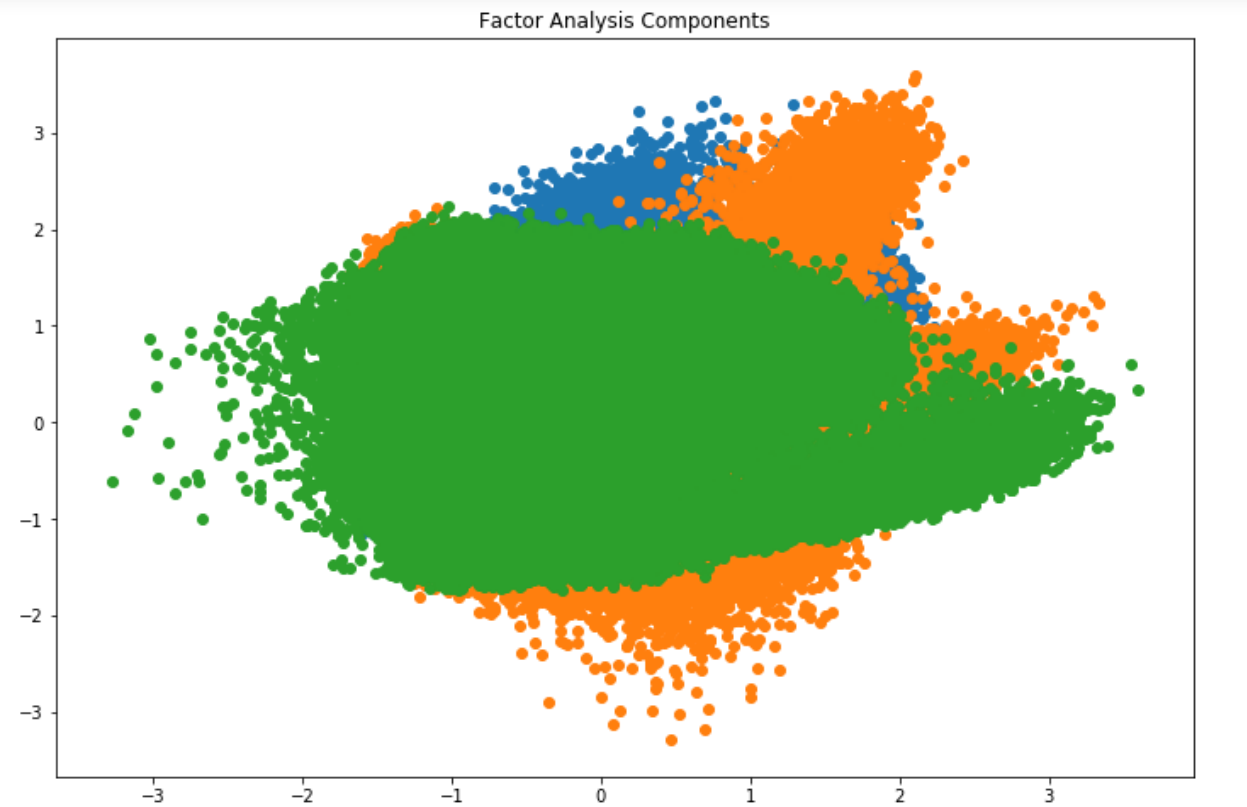
```
train = pd.read_csv("train.csv")      # Give the complete path of your
train.csv file
feat_cols = [ 'pixel'+str(i) for i in range(image.shape[1]) ]
df = pd.DataFrame(image,columns=feat_cols)
df['label'] = train['label']
```

Now we will decompose the dataset using Factor Analysis:

```
from sklearn.decomposition import FactorAnalysis
FA = FactorAnalysis(n_components = 3).fit_transform(df[feat_cols].values)
```

Here, *n_components* will decide the number of factors in the transformed data. After transforming the data, it's time to visualize the results:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.figure(figsize=(12,8))
plt.title('Factor Analysis Components')
plt.scatter(FA[:,0], FA[:,1])
plt.scatter(FA[:,1], FA[:,2])
plt.scatter(FA[:,2],FA[:,0])
```



Looks amazing, doesn't it? We can see all the different factors in the above graph. Here, the x-axis and y-axis represent the values of decomposed factors. As we mentioned earlier, it is hard to observe these factors individually but we have been able to reduce the dimensions of our data successfully.

Principal Component Analysis

Enable fullscreen

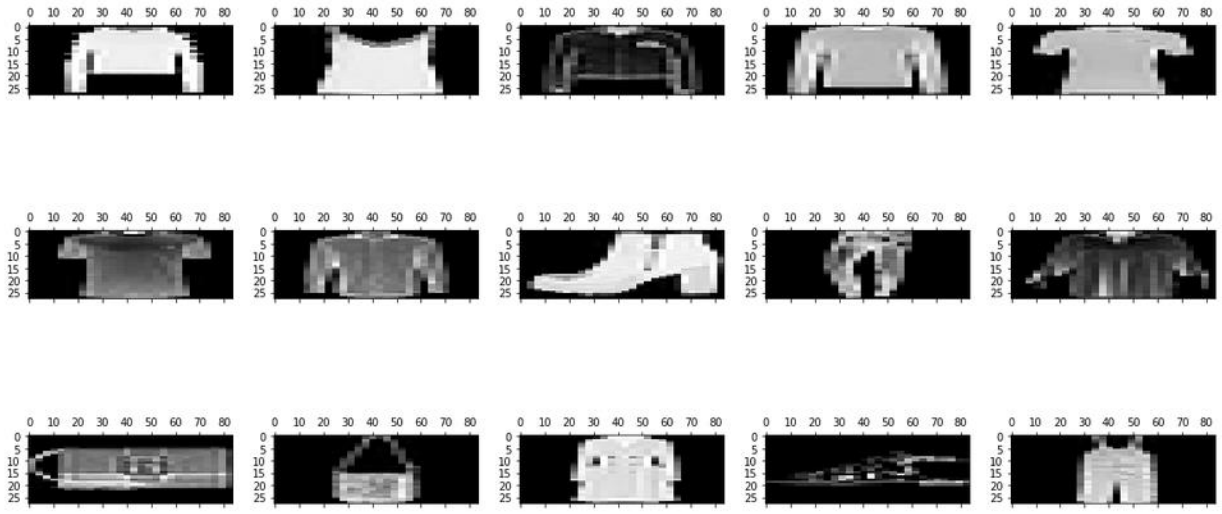
PCA is a technique that helps us in extracting a new set of variables from an existing large set of variables. These newly extracted variables are called Principal Components. You can refer to this article to learn more about PCA. For your quick reference, below are some of the key points you should know about PCA before proceeding further:

- A principal component is a linear combination of the original variables
- Principal components are extracted in such a way that the first principal component explains maximum variance in the dataset

- Second principal component tries to explain the remaining variance in the dataset and is uncorrelated to the first principal component
- Third principal component tries to explain the variance which is not explained by the first two principal components and so on

Before moving further, we'll randomly plot some of the images from our dataset:

```
rndperm = np.random.permutation(df.shape[0])
plt.gray()
fig = plt.figure(figsize=(20,10))
for i in range(0,15):
    ax = fig.add_subplot(3,5,i+1)
    ax.matshow(df.loc[rndperm[i],feat_cols].values.reshape((28,28*3)).astype(
float))
```

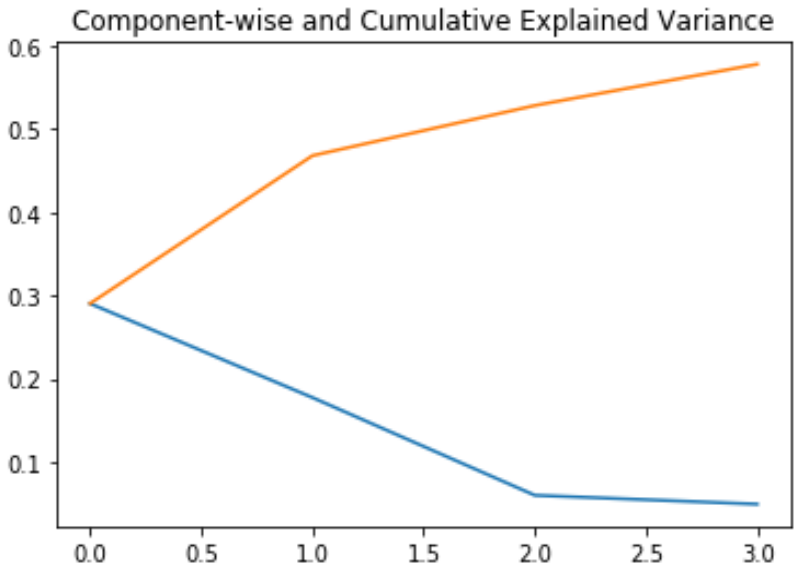


Let's implement PCA using Python and transform the dataset:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=4)
pca_result = pca.fit_transform(df[feat_cols].values)
```

In this case, *n_components* will decide the number of principal components in the transformed data. Let's visualize how much variance has been explained using these 4 components. We will use *explained_variance_ratio_* to calculate the same.

```
plt.plot(range(4), pca.explained_variance_ratio_)
plt.plot(range(4), np.cumsum(pca.explained_variance_ratio_))
plt.title("Component-wise and Cumulative Explained Variance")
```

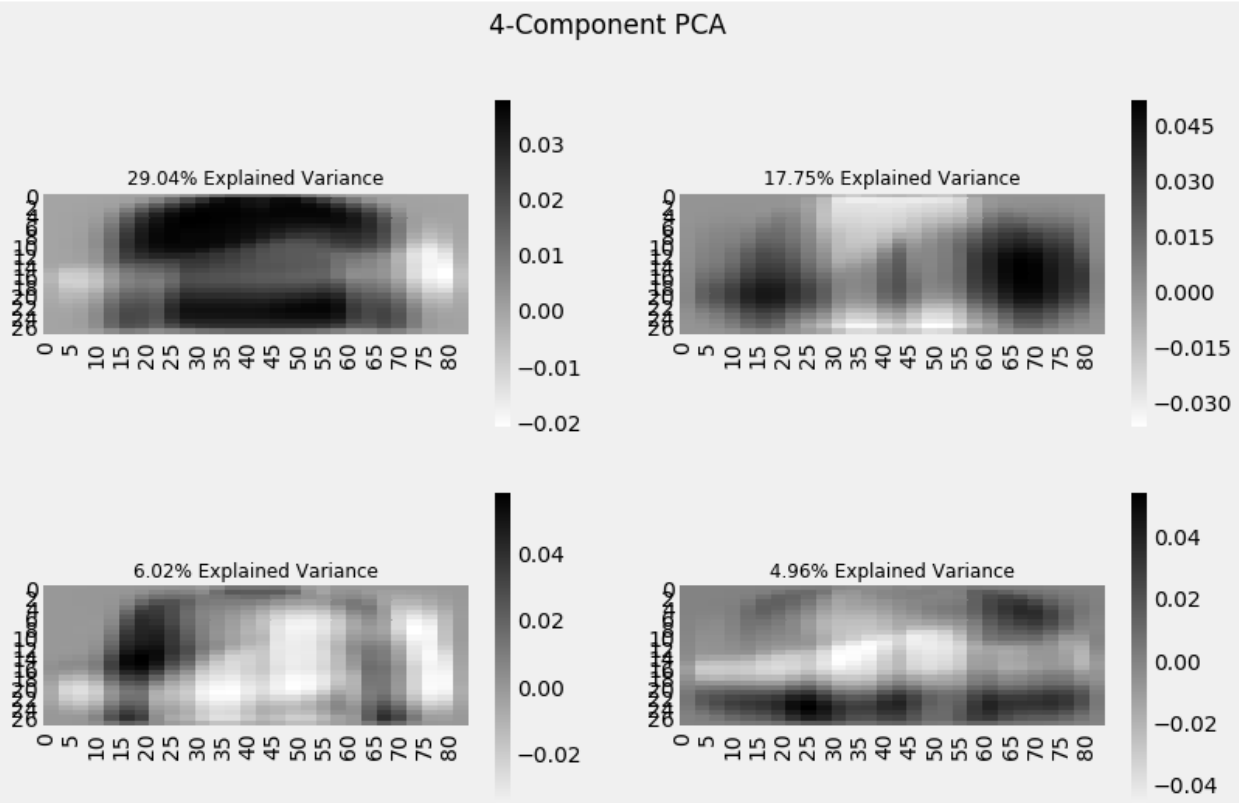


In the above graph, the blue line represents component-wise explained variance while the orange line represents the cumulative explained variance. **We are able to explain around 60% variance in the dataset using just four components.** Let us now try to visualize each of these decomposed components:

```
import seaborn as sns
plt.style.use('fivethirtyeight')
fig, axarr = plt.subplots(2, 2, figsize=(12, 8))
sns.heatmap(pca.components_[0, :].reshape(28, 84), ax=axarr[0][0],
cmap='gray_r')
sns.heatmap(pca.components_[1, :].reshape(28, 84), ax=axarr[0][1],
cmap='gray_r')
sns.heatmap(pca.components_[2, :].reshape(28, 84), ax=axarr[1][0],
cmap='gray_r')
sns.heatmap(pca.components_[3, :].reshape(28, 84), ax=axarr[1][1],
cmap='gray_r')
axarr[0][0].set_title(
"{0:.2f}% Explained Variance".format(pca.explained_variance_ratio_[0]*100),
fontsize=12
)
axarr[0][1].set_title(
"{0:.2f}% Explained Variance".format(pca.explained_variance_ratio_[1]*100),
fontsize=12
)
axarr[1][0].set_title(
"{0:.2f}% Explained Variance".format(pca.explained_variance_ratio_[2]*100),
fontsize=12
)
axarr[1][1].set_title(
"{0:.2f}% Explained Variance".format(pca.explained_variance_ratio_[3]*100),
fontsize=12
)
axarr[0][0].set_aspect('equal')
axarr[0][1].set_aspect('equal')
```

```
axarr[1][0].set_aspect('equal')
axarr[1][1].set_aspect('equal')

plt.suptitle('4-Component PCA')
```



Each additional dimension we add to the PCA technique captures less and less of the variance in the model. The first component is the most important one, followed by the second, then the third, and so on.

We can also use **Singular Value Decomposition** (SVD) to decompose our original dataset into its constituents, resulting in dimensionality reduction. To learn the mathematics behind SVD, refer to this article.

SVD decomposes the original variables into three constituent matrices. It is essentially used to remove redundant features from the dataset. It uses the concept of Eigenvalues and Eigenvectors to determine those three matrices. We will not go into the mathematics of it due to the scope of this article, but let's stick to our plan, i.e. reducing the dimensions in our dataset.

Let's implement SVD and decompose our original variables:

```
from sklearn.decomposition import TruncatedSVD
```

```
svd = TruncatedSVD(n_components=3,
random_state=42).fit_transform(df[feat_cols].values)
```

Let us visualize the transformed variables by plotting the first two principal components:

```
plt.figure(figsize=(12,8))
plt.title('SVD Components')
plt.scatter(svd[:,0], svd[:,1])
plt.scatter(svd[:,1], svd[:,2])
plt.scatter(svd[:,2],svd[:,0])
```

The above scatter plot shows us the decomposed components very neatly. As described earlier, there is not much correlation between these components.

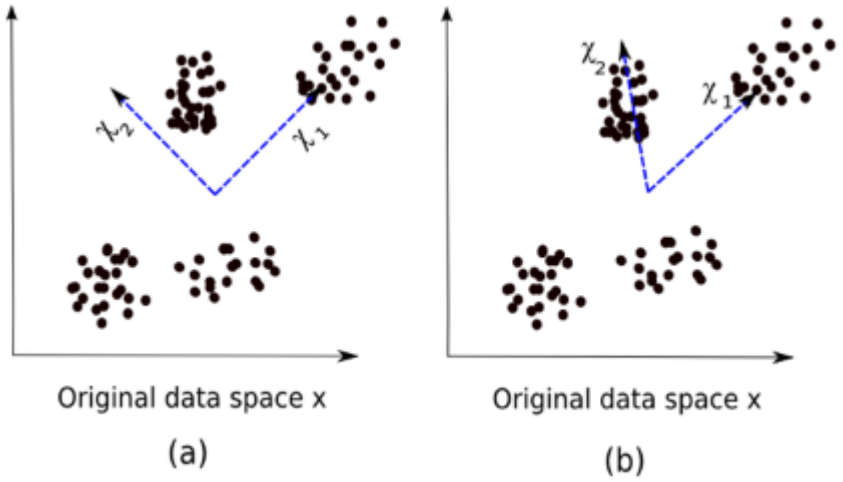
Independent Component Analysis

Independent Component Analysis (ICA) is based on information-theory and is also one of the most widely used dimensionality reduction techniques. The major difference between PCA and ICA is that PCA looks for uncorrelated factors while ICA looks for independent factors.

If two variables are uncorrelated, it means there is no linear relation between them. If they are independent, it means they are not dependent on other variables. For example, the age of a person is independent of what that person eats, or how much television he/she watches.

This algorithm assumes that the given variables are linear mixtures of some unknown latent variables. It also assumes that these latent variables are mutually independent, i.e., they are not dependent on other variables and hence they are called the independent components of the observed data.

Let’s compare PCA and ICA visually to get a better understanding of how they are different:



Here, image (a) represents the PCA results while image (b) represents the ICA results on the same dataset.

The equation of PCA is $x = W\chi$.

Here,

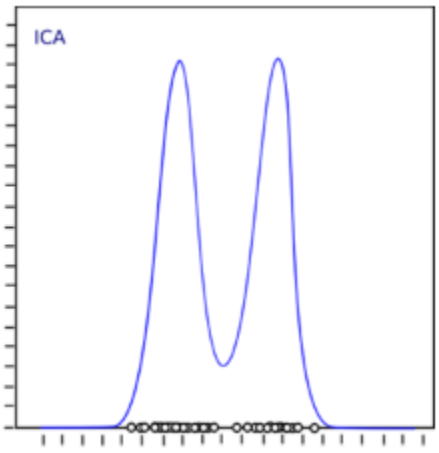
- x is the observations
- W is the mixing matrix
- χ is the source or the independent components

Now we have to find an un-mixing matrix such that the components become as independent as possible. Most common method to measure independence of components is Non-Gaussianity:

- As per the central limit theorem, distribution of the sum of independent components tends to be normally distributed (Gaussian).

- So we can look for the transformations that maximize the kurtosis of each component of the independent components. Kurtosis is the third order moment of the distribution. To learn more about kurtosis, head over here.

- Maximizing the kurtosis will make the distribution non-gaussian and hence we will get independent components.

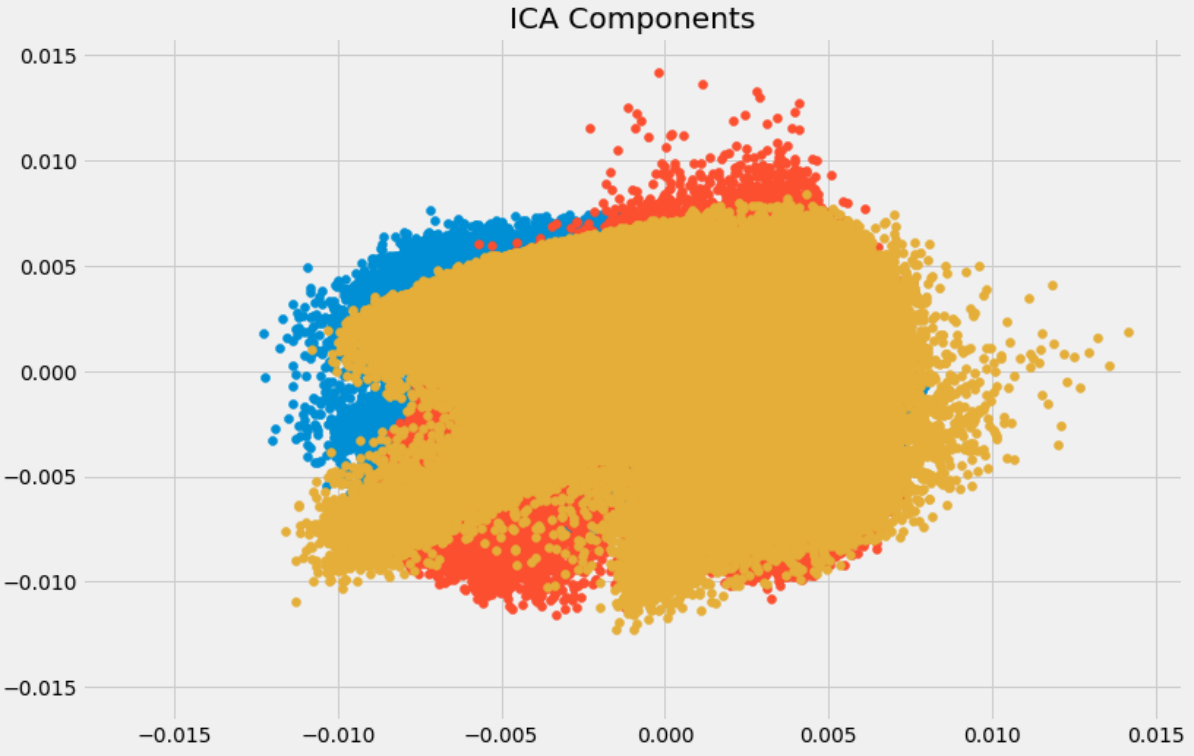


The above distribution is non-gaussian which in turn makes the components independent. Let's try to implement ICA in Python:

```
from sklearn.decomposition import FastICA
ICA = FastICA(n_components=3, random_state=12)
X=ICA.fit_transform(df[feat_cols].values)
```

Here, *n_components* will decide the number of components in the transformed data. We have transformed the data into 3 components using ICA. Let's visualize how well it has transformed the data:

```
plt.figure(figsize=(12,8))
plt.title('ICA Components')
plt.scatter(X[:,0], X[:,1])
plt.scatter(X[:,1], X[:,2])
plt.scatter(X[:,2], X[:,0])
```

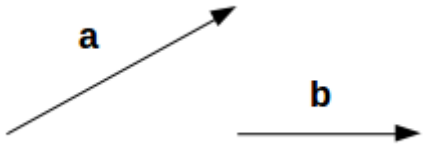


The data has been separated into different independent components which can be seen very clearly in the above image. X-axis and Y-axis represent the value of decomposed independent components.

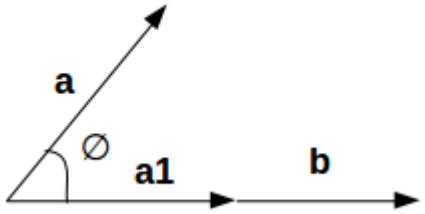
Projection Based Feature Extraction Techniques

Understanding Projection

To start off, we need to understand what projection is. Suppose we have two vectors, vector **a** and vector **b**, as shown below:



We want to find the projection of **a** on **b**. Let the angle between a and b be ∅. The projection (**a1**) will look like:



a1 is the vector parallel to **b.** So, we can get the projection of vector a on vector b using the below equation:

$$a_1 = |\mathbf{a}| \cos\theta = \mathbf{a} \cdot \hat{\mathbf{b}} = \mathbf{a} \cdot \frac{\mathbf{b}}{|\mathbf{b}|}$$

Here,

- a1 = projection of a onto b
- $\hat{b}$ = unit vector in the direction of b

By projecting one vector onto the other, dimensionality can be reduced.

In projection techniques, multi-dimensional data is represented by projecting its points onto a lower-dimensional space. Now we will discuss different methods of projections:

- Projection onto interesting directions:
    - Interesting directions depend on specific problems but generally, directions in which the projected values are non-gaussian are considered to be interesting
    - Similar to ICA (Independent Component Analysis), projection looks for directions maximizing the kurtosis of the projected values as a measure of non-gaussianity

- Projection onto Manifolds:

Once upon a time, it was assumed that the Earth was flat. No matter where you go on Earth, it keeps looking flat (let's ignore the mountains for a while). But if you keep walking in one direction, you will end up where you started. That wouldn't happen if the Earth was flat. The Earth only looks flat because we are minuscule as compared to the size of the Earth.

These small portions where the Earth looks flat are manifolds, and if we combine all these manifolds we get a large scale view of the Earth, i.e., original data. Similarly, for an n-dimensional curve, small flat pieces are manifolds and a combination of these manifolds will give us the original n-dimensional curve. Let us look at the steps for projection onto manifolds:

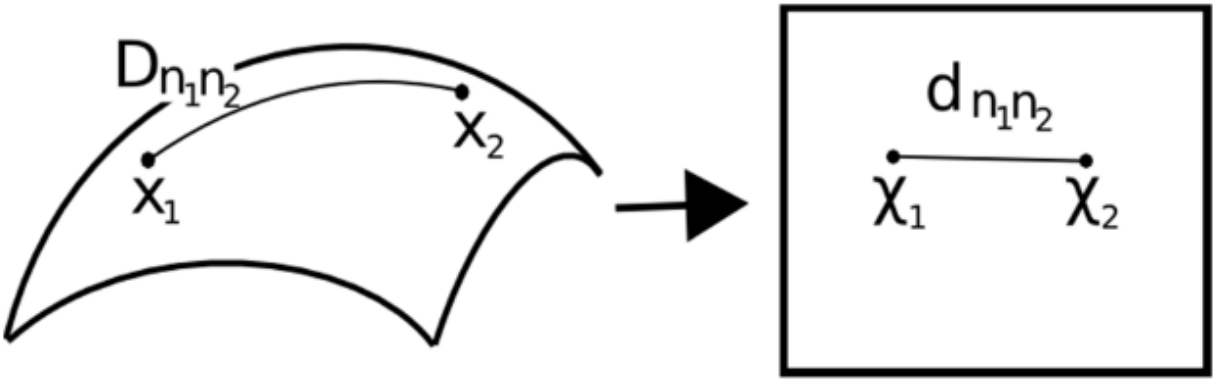
- We first look for a manifold that is close to the data
- Then project the data onto that manifold
- Finally, for representation, we unfold the manifold
- There are various techniques to get the manifold, and all of these techniques consist of a three-step approach:
    - Collecting information from each data point to construct a graph having data points as vertices
    - Transforming the above-generated graph into suitable input for embedding steps
    - Computing an (nXn) eigen equation

ISOMAP

Let us understand the manifold projection technique with an example before moving to ISOMAP.

If a manifold is continuously differentiable to any order, it is known as smooth or differentiable manifold. ISOMAP is an algorithm that aims to recover the full low-dimensional representation of a non-linear manifold. It assumes that the manifold is smooth.

It also assumes that for any pair of points on the manifold, the geodesic distance (the shortest distance between two points on a curved surface) between the two points is equal to the Euclidean distance (the shortest distance between two points on a straight line). Let's first visualize the geodesic and Euclidean distance between a pair of points:



Here,

- Dn1n2 = geodesic distance between X1 and X2
- dn1n2 = Euclidean distance between X1 and X2

ISOMAP assumes both of these distances to be equal. Let's now look at a more detailed explanation of this technique. As mentioned earlier, all these techniques work on a three-step approach. We will look at each of these steps in detail:

- Neighborhood Graph:
  - The first step is to calculate the distance between all pairs of data points:
    dij = dχ(xi,xj) = || xi-xj || χ
    Here,
    dχ(xi,xj) = geodesic distance between xi and xj
    || xi-xj || = Euclidean distance between xi and xj
  - After calculating the distance, we determine which data points are neighbors of the manifold
  - Finally, the neighborhood graph is generated: G=G(V,$\mathcal{E}$), where the set of vertices V = {x1, x2,...., xn} are input data points and set of edges $\mathcal{E}$ = {eij} indicate neighborhood relationship between the points
- Compute Graph Distances:
  - Now we calculate the geodesic distance between pairs of points in manifold by graph distances
  - Graph distance is the shortest path distance between all pairs of points in graph G
- Embedding:
  - Once we have the distances, we form the asymmetric (nXn) matrix of squared graph distance

- Now we choose to embed vectors to minimize the difference between geodesic distance and graph distance
- Finally, the graph G is embedded into Y by the (t Xn) matrix

Let's implement it in Python and get a clearer picture of what I'm talking about. We will perform non-linear dimensionality reduction through Isometric Mapping. For visualization, we will only take a subset of our dataset as running it on the entire dataset will require a lot of time.

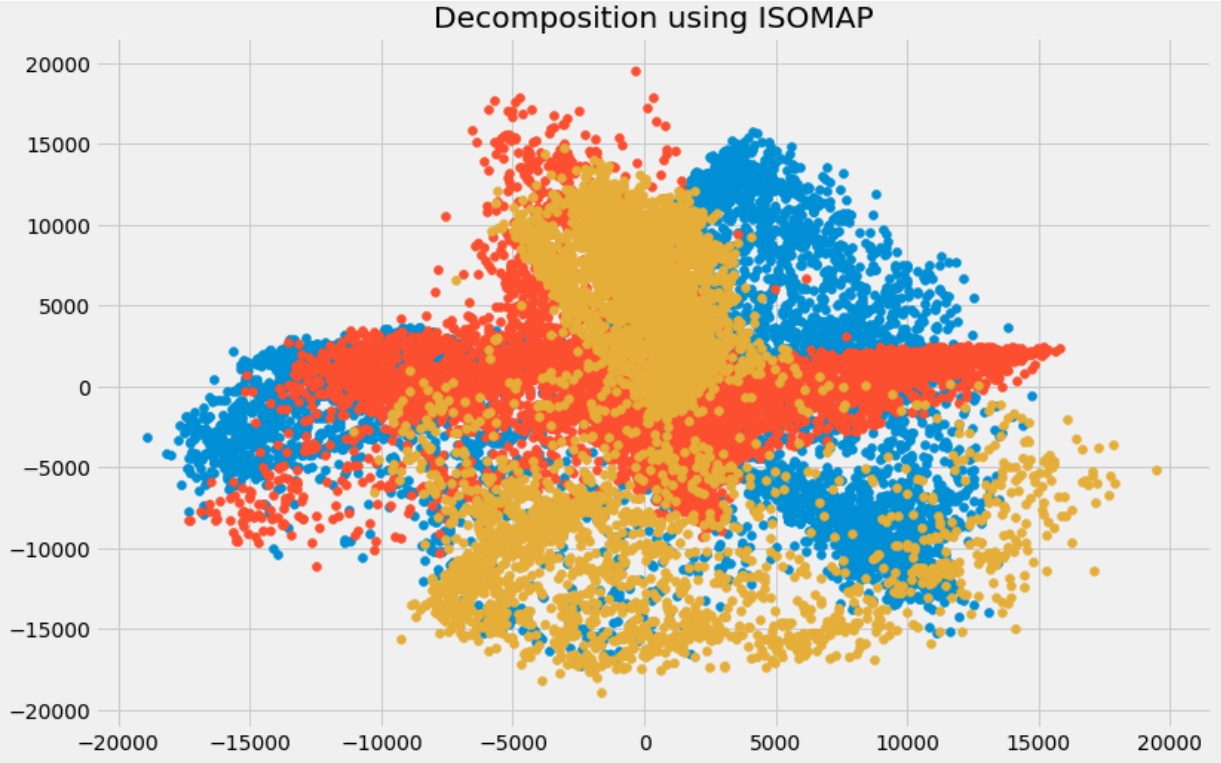
```
from sklearn import manifold
trans_data = manifold.Isomap(n_neighbors=5, n_components=3, n_jobs=-
1).fit_transform(df[feat_cols][:6000].values)
```

Parameters used:

- *n_neighbors* decides the number of neighbors for each point
- *n_components* decides the number of coordinates for manifold
- *n_jobs* = -1 will use all the CPU cores available

Visualizing the transformed data:

```
plt.figure(figsize=(12,8))
plt.title('Decomposition using ISOMAP')
plt.scatter(trans_data[:,0], trans_data[:,1])
plt.scatter(trans_data[:,1], trans_data[:,2])
plt.scatter(trans_data[:,2], trans_data[:,0])
```



You can see above that the correlation between these components is very low. In fact, they are even less correlated as compared to the components we obtained using SVD earlier!

t- Distributed Stochastic Neighbor Embedding (t-SNE)

So far we have learned that PCA is a good choice for dimensionality reduction and visualization for datasets with a large number of variables. But what if we could use something more advanced? What if we can easily search for patterns in a non-linear way? t-SNE is one such technique. There are mainly two types of approaches we can use to map the data points:

- Local approaches: They map nearby points on the manifold to nearby points in the low dimensional representation.

- Global approaches: They attempt to preserve geometry at all scales, i.e. mapping nearby points on the manifold to nearby points in low dimensional representation as well as far away points to faraway points.

    - t-SNE is one of the few algorithms which is capable of retaining both local and global structure of the data at the same time
    - It calculates the probability similarity of points in high dimensional space as well as in low dimensional space
    - High-dimensional Euclidean distances between data points are converted into conditional probabilities that represent similarities:

$$p_{j|i} = \frac{\exp\left(-\|x_i - x_j\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq i} \exp\left(-\|x_i - x_k\|^2 / 2\sigma_i^2\right)}$$

      xi and xj are data points, ||xi-xj|| represents the Euclidean distance between these data points, and **σ**i is the variance of data points in high dimensional space
    - For the low-dimensional data points yi and yj corresponding to the high-dimensional data points xi and xj, it is possible to compute a similar conditional probability using:

$$q_{j|i} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq i} \exp\left(-\|y_i - y_k\|^2\right)}$$

      where ||yi-yj|| represents the Euclidean distance between yi and yj
    - After calculating both the probabilities, it minimizes the difference between both the probabilities

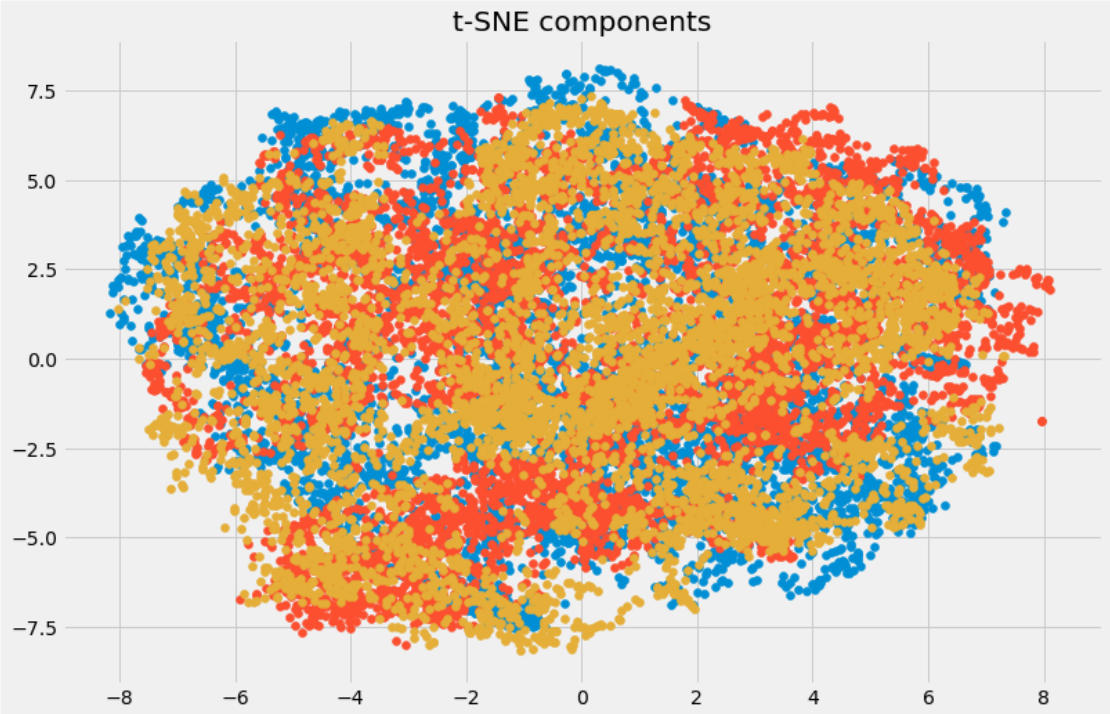
You can refer to this article to learn about t-SNE in more detail.

We will now implement it in Python and visualize the outcomes:

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=3,
n_iter=300).fit_transform(df[feat_cols][:6000].values)
```

n_components will decide the number of components in the transformed data. Time to visualize the transformed data:

```
plt.figure(figsize=(12,8))
plt.title('t-SNE components')
plt.scatter(tsne[:,0], tsne[:,1])
plt.scatter(tsne[:,1], tsne[:,2])
plt.scatter(tsne[:,2], tsne[:,0])
```



Here you can clearly see the different components that have been transformed using the powerful t-SNE technique.

UMAP

t-SNE works very well on large datasets but it also has its limitations, such as loss of large-scale information, slow computation time, and inability to meaningfully represent very large datasets. Uniform Manifold Approximation and Projection (UMAP) is a dimension reduction technique that can preserve as much of the local and more of the global data structure as compared to t-SNE, with a shorter runtime. Sounds intriguing, right?

Some of the key advantages of UMAP are:

- It can handle large datasets and high dimensional data without too much difficulty

- It combines the power of visualization with the ability to reduce the dimensions of the data
- Along with preserving the local structure, it also preserves the global structure of the data. UMAP maps nearby points on the manifold to nearby points in the low dimensional representation and does the same for faraway points

This method uses the concept of k-nearest neighbor and optimizes the results using stochastic gradient descent. It first calculates the distance between the points in high dimensional space, projects them onto the low dimensional space, and calculates the distance between points in this low dimensional space. It then uses Stochastic Gradient Descent to minimize the difference between these distances. To get a more in-depth understanding of how UMAP works, check out this paper.

Refer here to see the documentation and installation guide of UMAP. We will now implement it in Python:

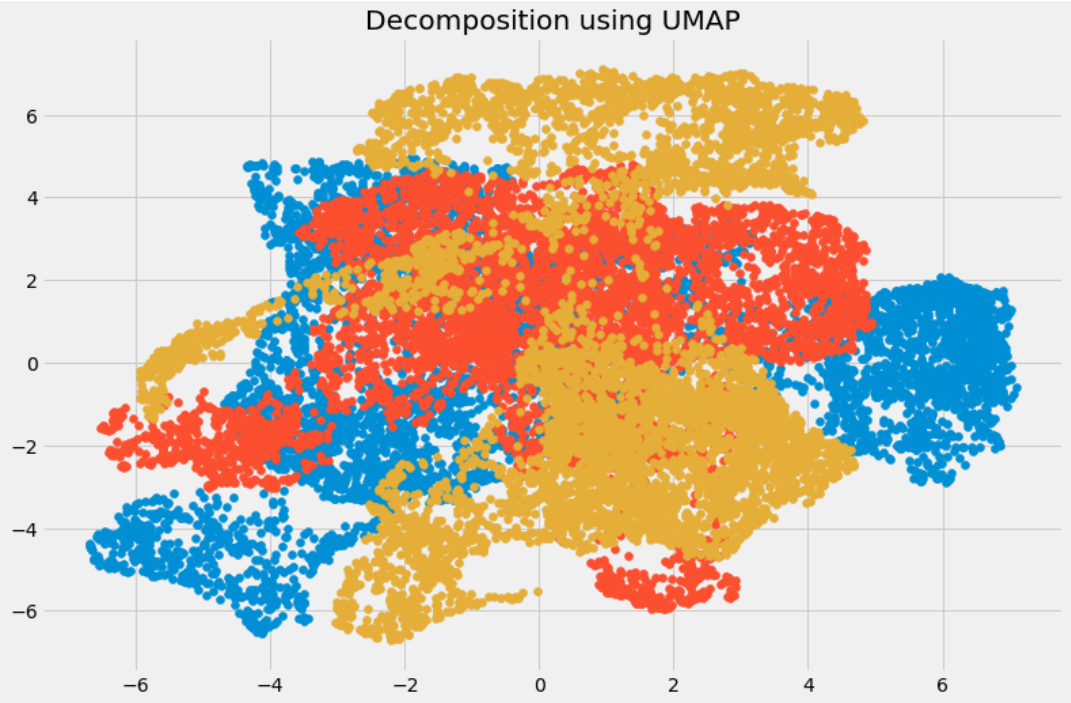
```
import umap
umap_data = umap.UMAP(n_neighbors=5, min_dist=0.3,
n_components=3).fit_transform(df[feat_cols][:6000].values)
```

Here,

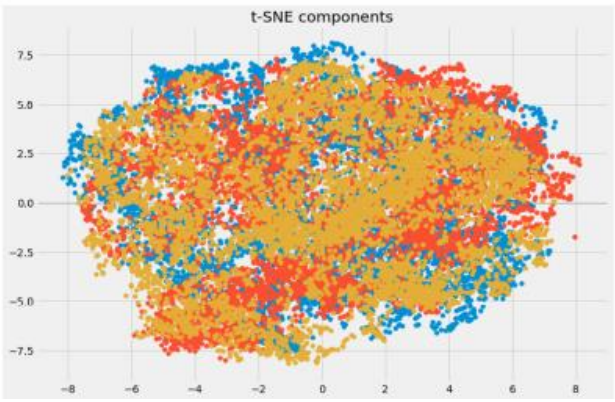
- *n_neighbors* determines the number of neighboring points used
- *min_dist* controls how tightly embedding is allowed. Larger values ensure embedded points are more evenly distributed

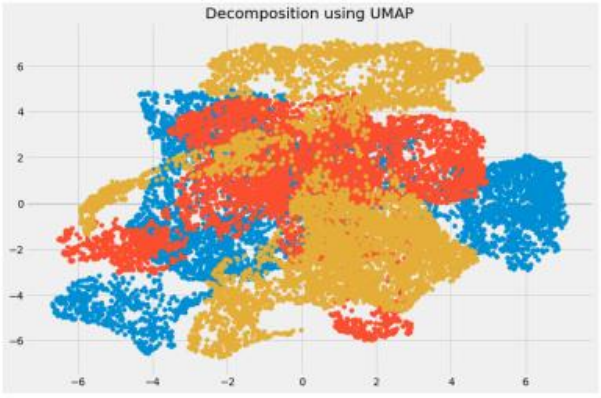
Let us visualize the transformation:

```
plt.figure(figsize=(12,8))
plt.title('Decomposition using UMAP')
plt.scatter(umap_data[:,0], umap_data[:,1])
plt.scatter(umap_data[:,1], umap_data[:,2])
plt.scatter(umap_data[:,2], umap_data[:,0])
```



The dimensions have been reduced and we can visualize the different transformed components. There is very little correlation between the transformed variables. Let us compare the results from UMAP and t-SNE:





We can see that the correlation between the components obtained from UMAP is quite less as compared to the correlation between the components obtained from t-SNE. Hence, UMAP tends to give better results.

As mentioned in UMAP's GitHub repository, it often performs better at preserving aspects of the global structure of the data than t-SNE. This means that it can often provide a better "big picture" view of the data as well as preserving local neighbor relations.