Analytics Vidhya
Learn everything about analytics

DATA SCIENCE MACHINE LEARNING PYTHON

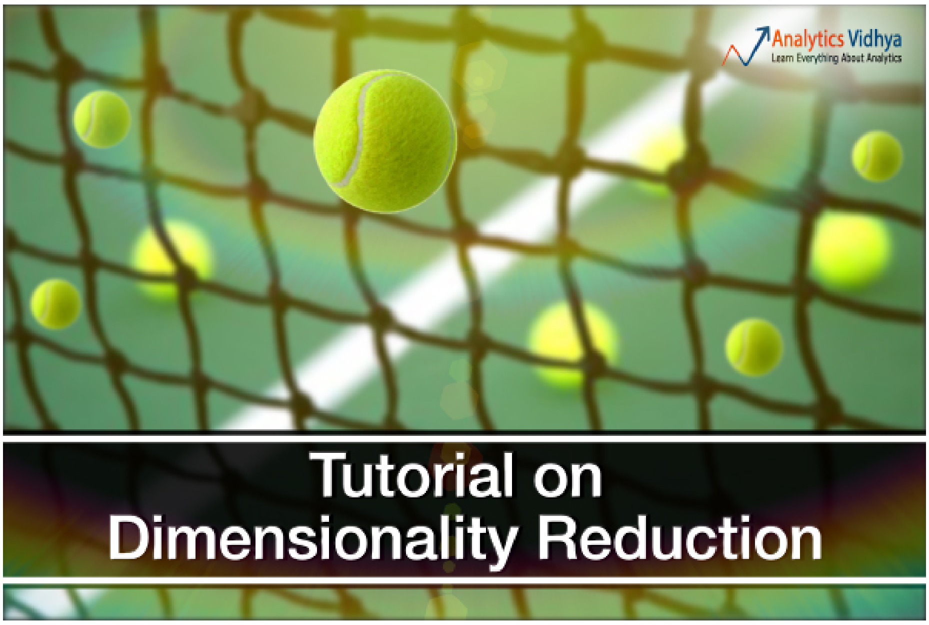
The Ultimate Guide to 12 Dimensionality Reduction Techniques (with Python codes)

PULKIT SHARMA, AUGUST 27, 2018 LOGIN TO BOOKMARK THIS ARTICLE

Introduction

Have you ever worked on a dataset with more than a thousand features? How about over 50,000 features? I have, and let me 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 article.

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 12 different techniques I have covered. Each technique has it's own implementation in Python to get you well acquainted with it.

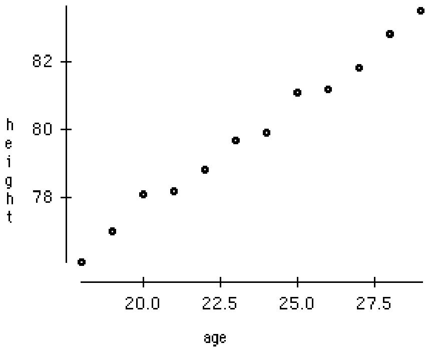
Table of Contents

1. 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:

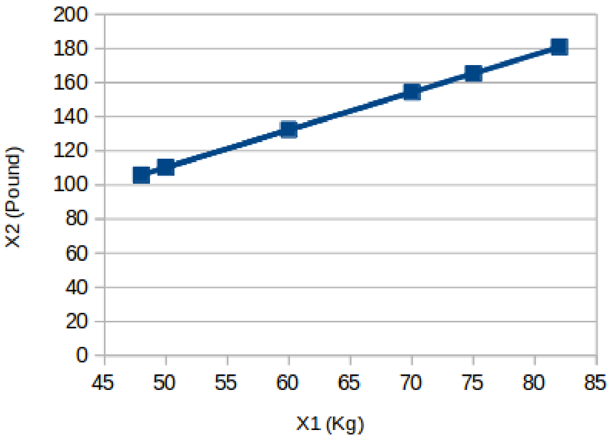
- Facebook collects data of what you like, share, post, places you visit, restaurants you like, etc.
- Your smartphone apps collect a lot of personal information about you
- Amazon collects data of what you buy, view, click, etc. on their site
- Casinos keep a track of every move each customer makes

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:

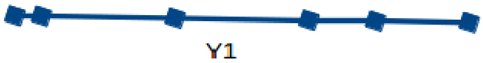


Now consider a case in which we have, say 100 variables (p=100). In this case, we can have 100(100-1)/2 = 5000 different plots. It does not make much sense to visualize each of them separately, right? In such cases where we have a large number of variables, it is better to select a subset of these variables (p<<100) which captures as much information as the original set of variables.

Let us understand this with a simple example. Consider the below image:



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.

2. 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
- Less dimensions lead to less computation/training time
- Some algorithms do not perform well when we have a 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 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

Time to dive into the crux of this article – the various dimensionality reduction techniques!**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).

3. 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 dimensionality reduction)

We will now look at various dimensionality reduction techniques and how to implement each of them in Python.

3.1 Missing Value Ratio

Suppose you’re given a dataset. What would be your first step? You would naturally want to explore the data first before building model. While exploring the data, you find that your dataset has some missing values. Now what? You will try to find out the reason for these missing values and then impute them or drop the variables entirely which have missing values (using appropriate methods).

What if we have too many missing values (say more than 50%)? Should we impute the missing values or drop the variable? I would prefer to drop the variable since it will not have much information. However, this isn’t set in stone. We can set a threshold value and if the percentage of missing values in any variable is more than that threshold, we will drop the variable.

Let’s implement this approach in Python.

```
# import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

First, let’s load the data:

```
# read the data
```

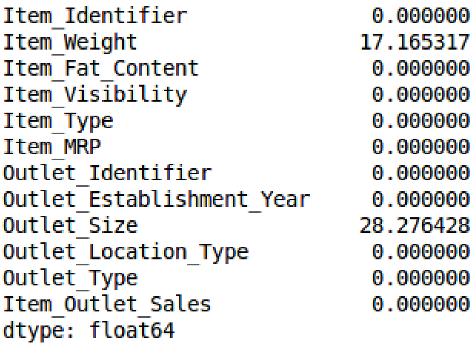
```
train=pd.read_csv("Train_UWu5bXk.csv")
```

Note: The path of the file should be added while reading the data.

Now, we will check the percentage of missing values in each variable. We can use*.isnull().sum()* to calculate this.

```
# checking the percentage of missing values in each variable

train.isnull().sum()/len(train)*100
```



```
Item_Identifier               0.000000
Item_Weight                  17.165317
Item_Fat_Content              0.000000
Item_Visibility               0.000000
Item_Type                     0.000000
Item_MRP                      0.000000
Outlet_Identifier             0.000000
Outlet_Establishment_Year     0.000000
Outlet_Size                  28.276428
Outlet_Location_Type          0.000000
Outlet_Type                   0.000000
Item_Outlet_Sales             0.000000
dtype: float64
```

As you can see in the above table, there aren't too many missing values (just 2 variables have them actually). We can impute the values using appropriate methods, or we can set a threshold of, say 20%, and remove the variable having more than 20% missing values. Let's look at how this can be done in Python:

```
# saving missing values in a variable

a = train.isnull().sum()/len(train)*100

# saving column names in a variable

variables = train.columns

variable = [ ]

for i in range(0,12):

    if a[i]<=20:   #setting the threshold as 20%

        variable.append(variables[i])
```

So the variables to be used are stored in "variable", which contains only those features where the missing values are less than 20%.

3.2 Low Variance Filter

Consider a variable in our dataset where all the observations have the same value, say 1. If we use this variable, do you think it can improve the model we will build? The answer is no, because this variable will have zero variance.

So, we need to calculate the variance of each variable we are given. Then drop the variables having low variance as compared to other variables in our dataset. The reason for doing this, as I mentioned above, is that variables with a low variance will not affect the target variable.

Let's first impute the missing values in the *Item_Weight* column using the median value of the known *Item_Weight* observations. For the *Outlet_Size* column, we will use the mode of the known*Outlet_Size* values to impute the missing values:

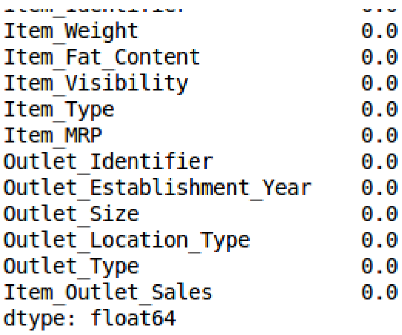
```
train['Item_Weight'].fillna(train['Item_Weight'].median(), inplace=True)

train['Outlet_Size'].fillna(train['Outlet_Size'].mode()[0], inplace=True)
```

Let's check whether all the missing values have been filled:

```
train.isnull().sum()/len(train)*100
```

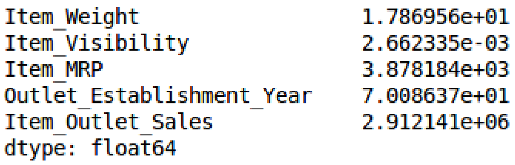
```
Item_Identifier              0.0
```



```
Item_Identifier              0.0
Item_Weight                  0.0
Item_Fat_Content             0.0
Item_Visibility              0.0
Item_Type                    0.0
Item_MRP                     0.0
Outlet_Identifier            0.0
Outlet_Establishment_Year    0.0
Outlet_Size                  0.0
Outlet_Location_Type         0.0
Outlet_Type                  0.0
Item_Outlet_Sales            0.0
dtype: float64
```

Voila! We are all set. Now let's calculate the variance of all the numerical variables.

```
train.var()
```



```
Item_Weight                  1.786956e+01
Item_Visibility              2.662335e-03
Item_MRP                     3.878184e+03
Outlet_Establishment_Year    7.008637e+01
Item_Outlet_Sales            2.912141e+06
dtype: float64
```

As the above output shows, the variance of *Item_Visibility* is very less as compared to the other variables. We can safely drop this column. This is how we apply low variance filter. Let's implement this in Python:

```
numeric = train[['Item_Weight', 'Item_Visibility', 'Item_MRP', 'Outlet_Establishment_Year']]
var = numeric.var()
numeric = numeric.columns
variable = [ ]
for i in range(0,len(var)):
    if var[i]>=10:   #setting the threshold as 10%
       variable.append(numeric[i+1])
```

The above code gives us the list of variables that have a variance greater than 10.

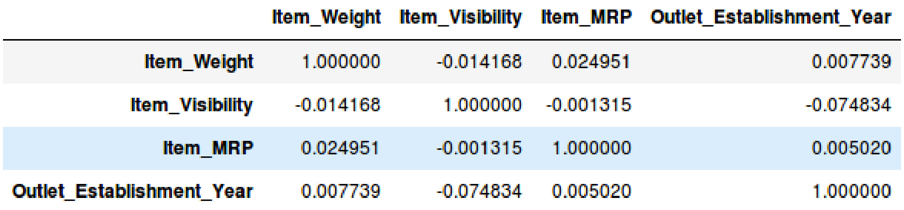
3.3 High Correlation filter

High correlation between two variables means they have similar trends and are likely to carry similar information. This can bring down the performance of some models drastically (linear and logistic regression models, for instance). We can calculate the correlation between independent numerical variables that are numerical in nature. If the correlation coefficient crosses a certain threshold value, we can drop one of the variables (dropping a variable is highly subjective and should always be done keeping the domain in mind).

As a general guideline, we should keep those variables which show a decent or high correlation with the target variable.

Let's perform the correlation calculation in Python. We will drop the dependent variable (*Item_Outlet_Sales*) first and save the remaining variables in a new dataframe (*df*).

```
df=train.drop('Item_Outlet_Sales', 1)
df.corr()
```



	Item_Weight	Item_Visibility	Item_MRP	Outlet_Establishment_Year
Item_Weight	1.000000	-0.014168	0.024951	0.007739
Item_Visibility	-0.014168	1.000000	-0.001315	-0.074834
Item_MRP	0.024951	-0.001315	1.000000	0.005020
Outlet_Establishment_Year	0.007739	-0.074834	0.005020	1.000000

Wonderful, we don’t have any variables with a high correlation in our dataset. Generally, if the correlation between a pair of variables is greater than 0.5-0.6, we should seriously consider dropping one of those variables.

3.4 Random Forest

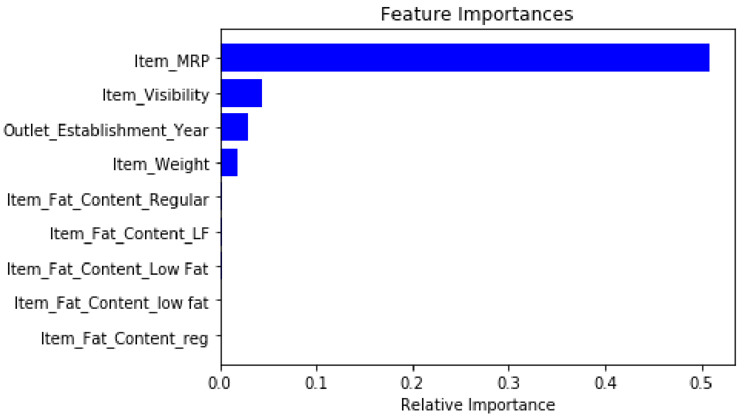
Random Forest is one of the most widely used algorithms for feature selection. It comes packaged with in-built feature importance so you don’t need to program that separately. This helps us select a smaller subset of features.

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 hand pick the top-most features to reduce the dimensionality in our dataset. Alernatively,**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)
```

3.5 Backward Feature Elimination

Follow the below steps to understand and use the ‘Backward Feature Elimination’ technique:

- We first take all the n variables present in our dataset and train the model using them
- We then calculate the performance of the model
- Now, we compute the performance of the model after eliminating each variable (n times), i.e., we drop one variable every time and train the model on the remaining n-1 variables
- We identify the variable whose removal has produced the smallest (or no) change in the performance of the model, and then drop that variable
- Repeat this process until no variable can be dropped

This method can be used when building Linear Regression or Logistic Regression models. Let’s look at it’s Python implementation:

```
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE
from sklearn import datasets
lreg = LinearRegression()
rfe = RFE(lreg, 10)
rfe = rfe.fit_transform(df, train.Item_Outlet_Sales)
```

We need to specify the algorithm and number of features to select, and we get back the list of variables obtained from backward feature elimination. We can also check the ranking of the variables using the “*rfe.ranking_*” command.

3.6 Forward Feature Selection

This is the opposite process of the Backward Feature Elimination we saw above. Instead of eliminating features, we try to find the best features which improve the performance of the model. This technique works as follows:

- We start with a single feature. Essentially, we train the model n number of times using each feature separately
- The variable giving the best performance is selected as the starting variable
- Then we repeat this process and add one variable at a time. The variable that produces the highest increase in performance is retained
- We repeat this process until no significant improvement is seen in the model’s performance

Let’s implement it in Python:

```
from sklearn.feature_selection import f_regression
ffs = f_regression(df,train.Item_Outlet_Sales )
```

This returns an array containing the F-values of the variables and the p-values corresponding to each F value. Refer tothis link to learn more about F-values. For our purpose, we will select the variables having F-value greater than 10:

```
variable = [ ]
for i in range(0,len(df.columns)-1):
    if ffs[0][i] >=10:
       variable.append(df.columns[i])
```

This gives us the top most variables based on the forward feature selection algorithm.

NOTE : **Both Backward Feature Elimination and Forward Feature Selection are time consuming and computationally expensive**.They are practically only used on datasets that have a small number of input variables.

The techniques we have seen so far are generally used when we do not have a very large number of variables in our dataset. These are more or less feature selection techniques. In the upcoming sections, 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.**

The dataset has a total of 70,000 images, out of which 60,000 are in the training set and the remaining 10,000 are test images. For the scope of this article, we will be working only on the training images. The train file is in a zip format. Once you extract the zip file, you will get a .csv file and a train folder which includes these 60,000 images. The corresponding label of each image can be found in the ‘train.csv’ file.

3.7 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 group(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 dataframe 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).

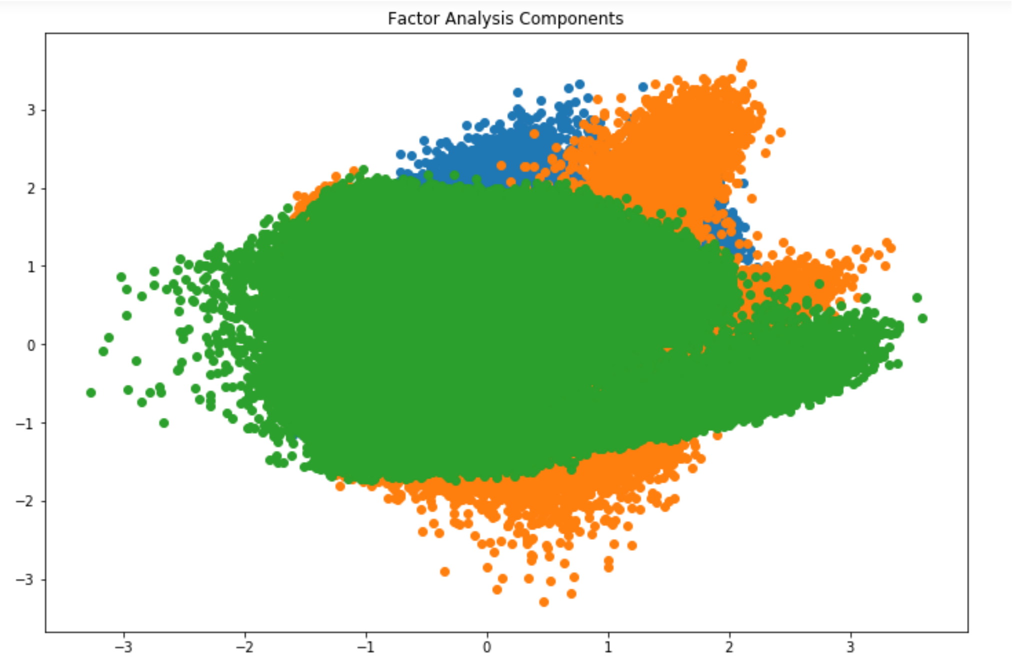
```
train = pd.read_csv("train.csv")     # Give the complete path of your train.csv file
feat_cols = [ 'pixel'+str(i) for i in range(image.shape[1]) ]
df = pd.DataFrame(image,columns=feat_cols)
df['label'] = train['label']
```

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 I mentioned earlier, it is hard to observe these factors individually but we have been able to reduce the dimensions of our data successfully.

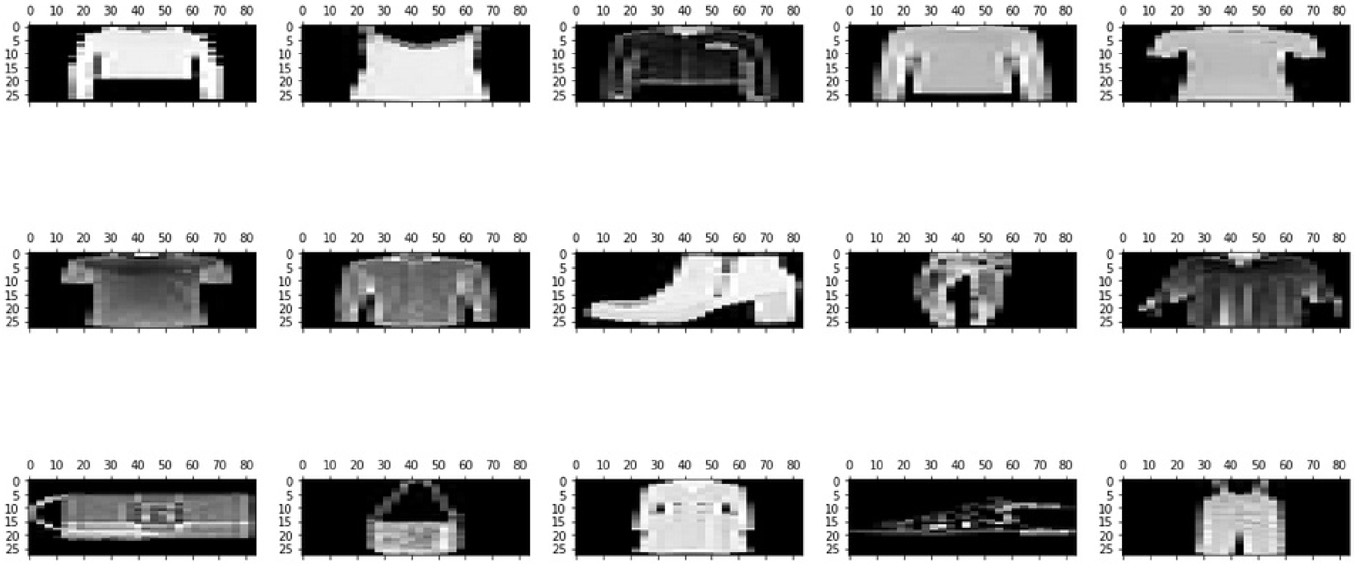
3.8 Principal Component Analysis (PCA)

PCA is a technique which 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))
```

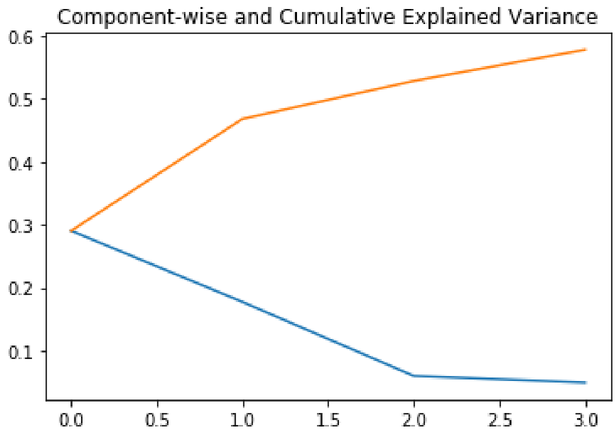


Let’s implement PCA using Python and transform the dataset:

```
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.

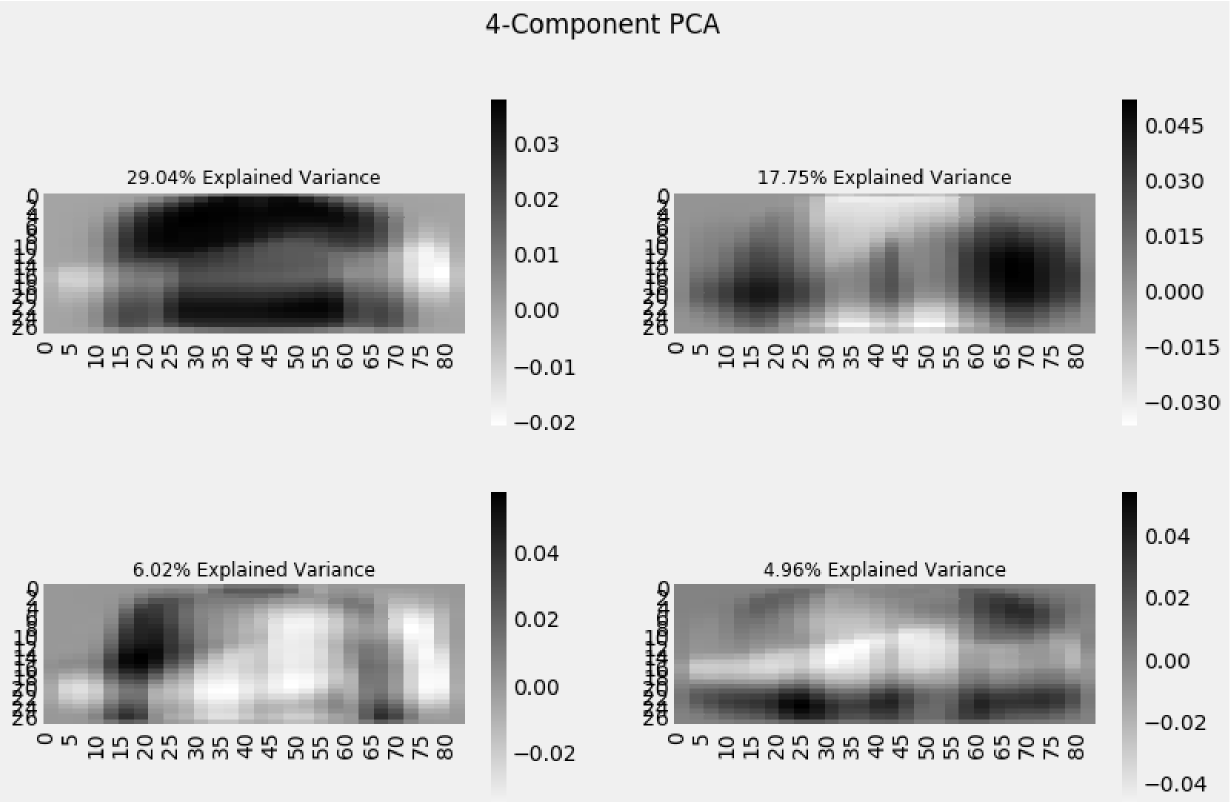
```
plt.plot(range(4), pca.explained_variance_ratio_)
plt.plot(range(4), np.cumsum(pca.explained_variance_ratio_))
plt.title("Component-wise and Cumulative Explained Variance")
```



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:

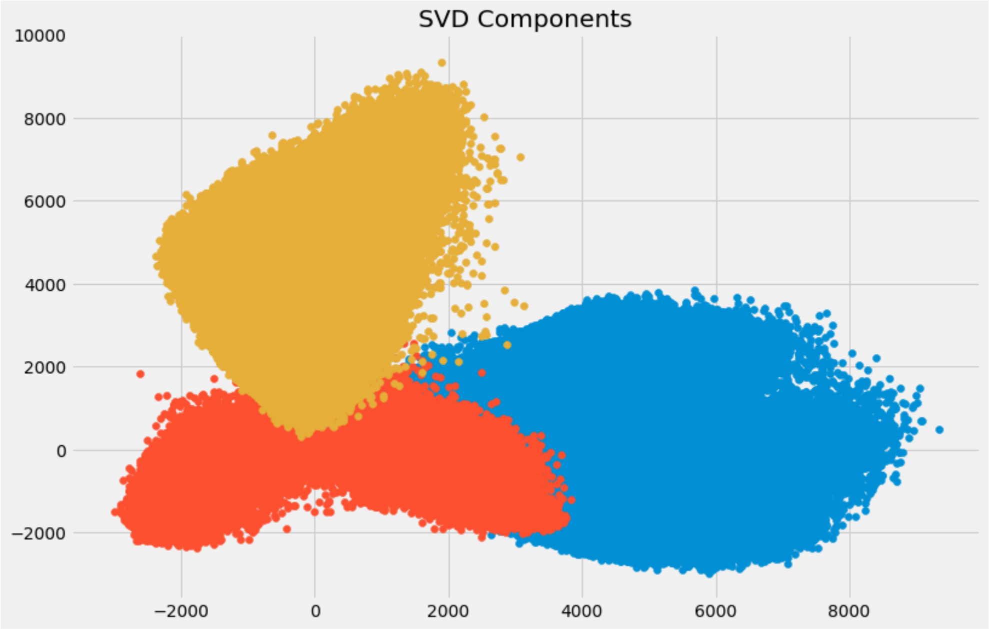
```
plt.figure(figsize=(12,8))

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.

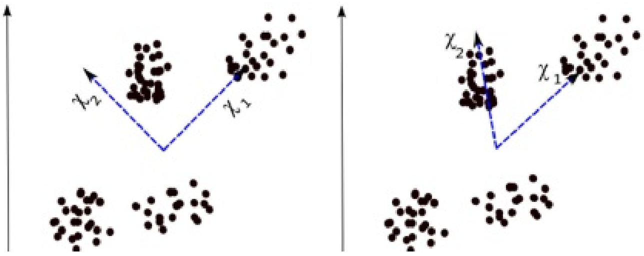
3.9 Independent Component Analysis

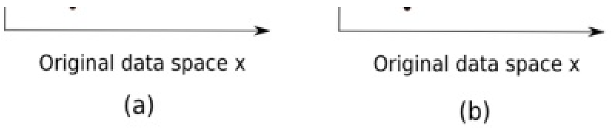
Independent Component Analysis (ICA) is based on information-theory and is also one of the most widely used dimensionality reduction techniques. The major difference between PCA and ICA is that PCA looks for uncorrelated factors while ICA looks for independent factors.

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.

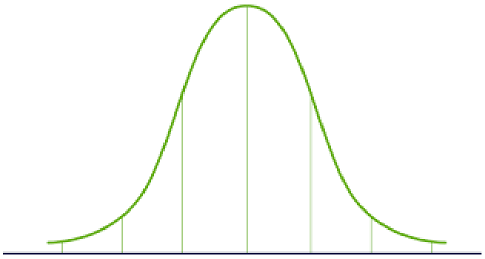
The equation of PCA is x = Wχ.

Here,

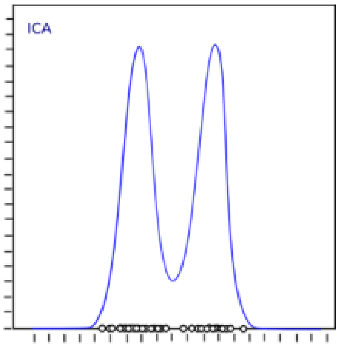
- x is the observations
- W is the mixing matrix
- χ is the source or the independent components

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.

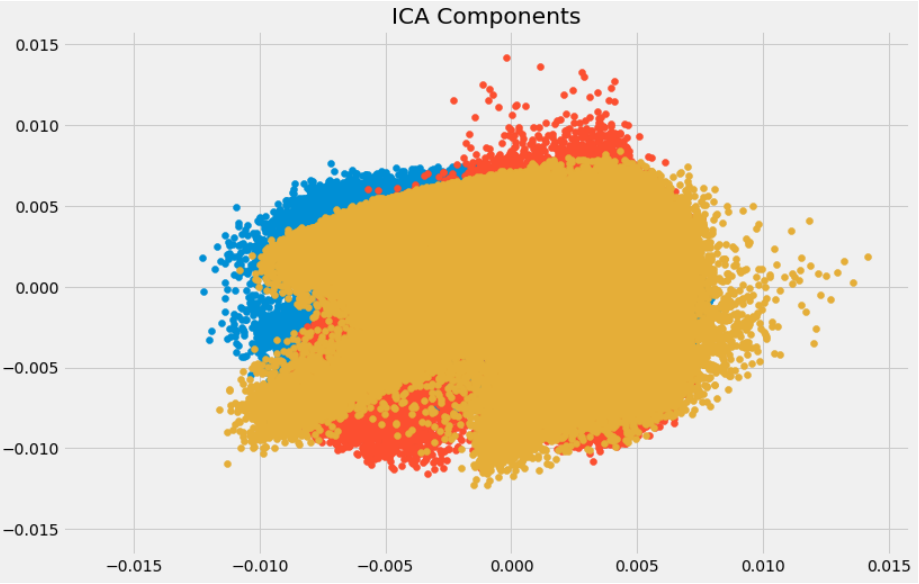


The above distribution is non-gaussian which in turn makes the components independent. Let’s try to implement ICA in Python:

```
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])
```

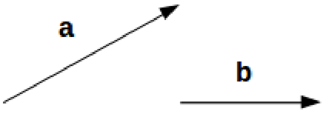


The data has been separated into different independent components which can be seen very clearly in the above image. X-axis and Y-axis represent the value of decomposed independent components.

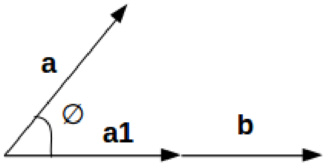
Now we shall look at some of the methods which reduce the dimensions of the data using projection techniques.

3.10 Methods Based on Projections

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
- b̂ = 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.

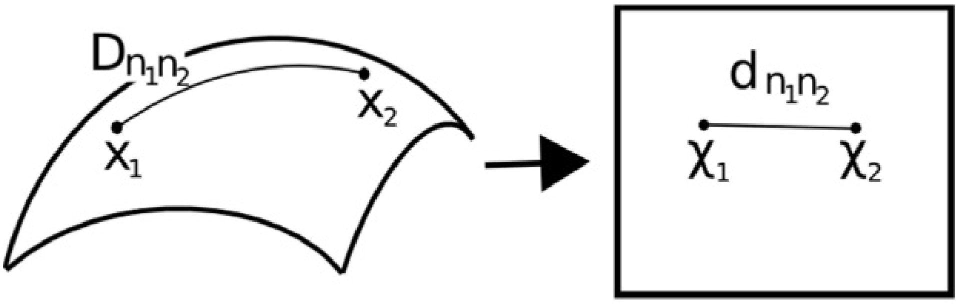
These small portions where the Earth looks flat are manifolds, and if we combine all these manifolds we get a large scale view of the Earth, i.e., original data. Similarly for an n-dimensional curve, small flat pieces are manifolds and a combination of these manifolds will give us the original n-dimensional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

Let us understand manifold projection technique with an example.

If a manifold is continuously differentiable to any order, it is known as smooth or differentiable manifold. ISOMAP is an algorithm which aims to recover 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 manifold, the geodesic distance (shortest distance between two points on a curved surface) between the two points is equal to the Euclidean distance (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:
    - 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 manifold
    - Finally the neighborhood graph is generated: G=G(V,ℰ), where the set of vertices V = {x1, x2,…., xn} are input data points and set of edges 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 a symmetric (nXn) matrix of squared graph distance
    - Now we choose embedding 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:

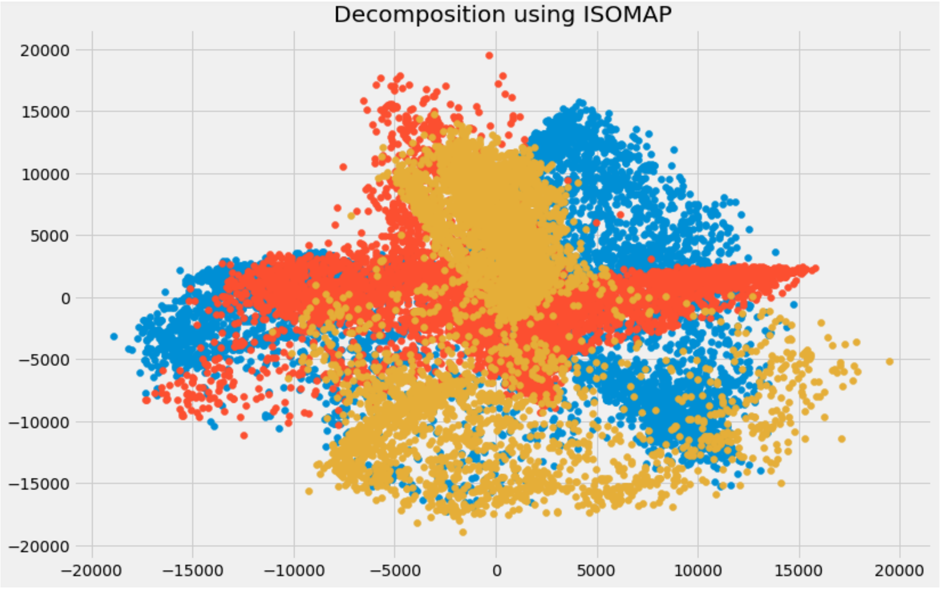
```
plt.figure(figsize=(12,8))

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!

3.11 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 maps 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 manifold to nearby points in low dimensional representation as well as far away points to far away 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:

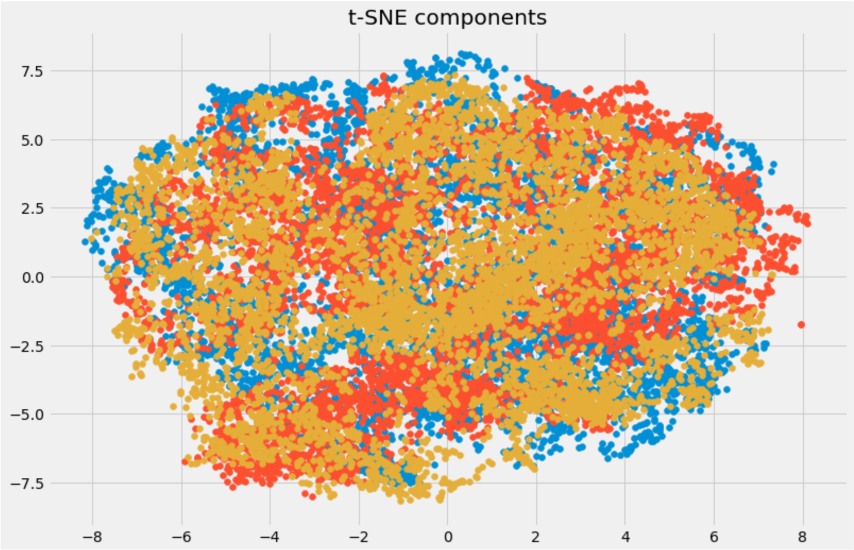
```
plt.figure(figsize=(12,8))

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.

3.12 UMAP

t-SNE works very well on large datasets but it also has it's 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 far away 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:

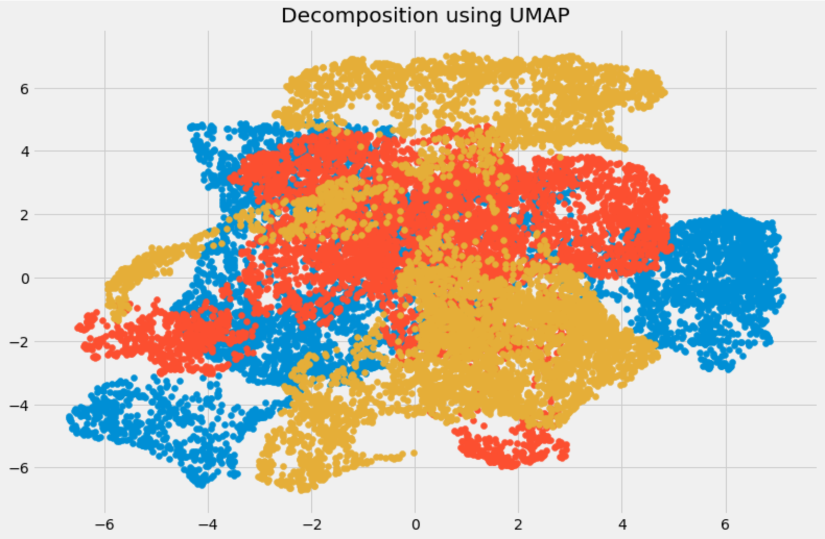
```
import umap
umap_data = umap.UMAP(n_neighbors=5, min_dist=0.3, n_components=3).fit_transform(df[feat_cols][:6000].values)
```

Here,

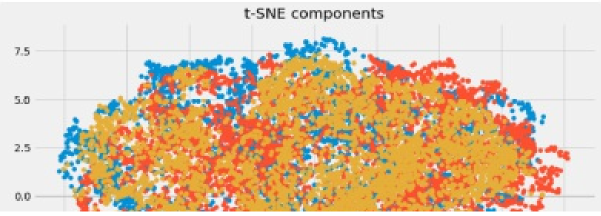
- *n_neighbors* determines the number of neighboring points used
- *min_dist* controls how tightly embedding is allowed. Larger values ensure embedded points are more evenly distributed

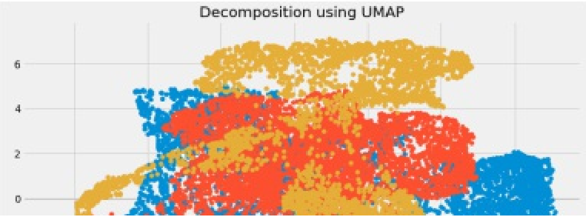
Let us visualize the transformation:

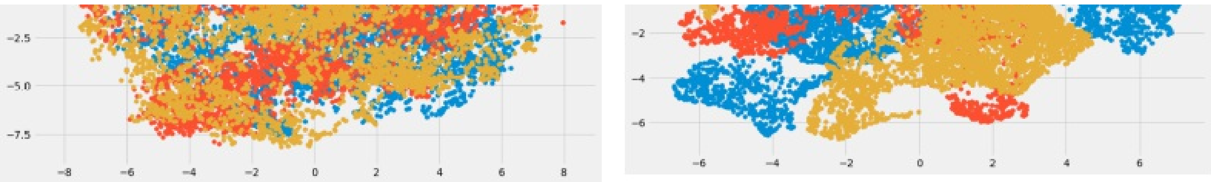
```
plt.figure(figsize=(12,8))
plt.title('Decomposition using UMAP')
plt.scatter(umap_data[:,0], umap_data[:,1])
plt.scatter(umap_data[:,1], umap_data[:,2])
plt.scatter(umap_data[:,2], umap_data[:,0])
```



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







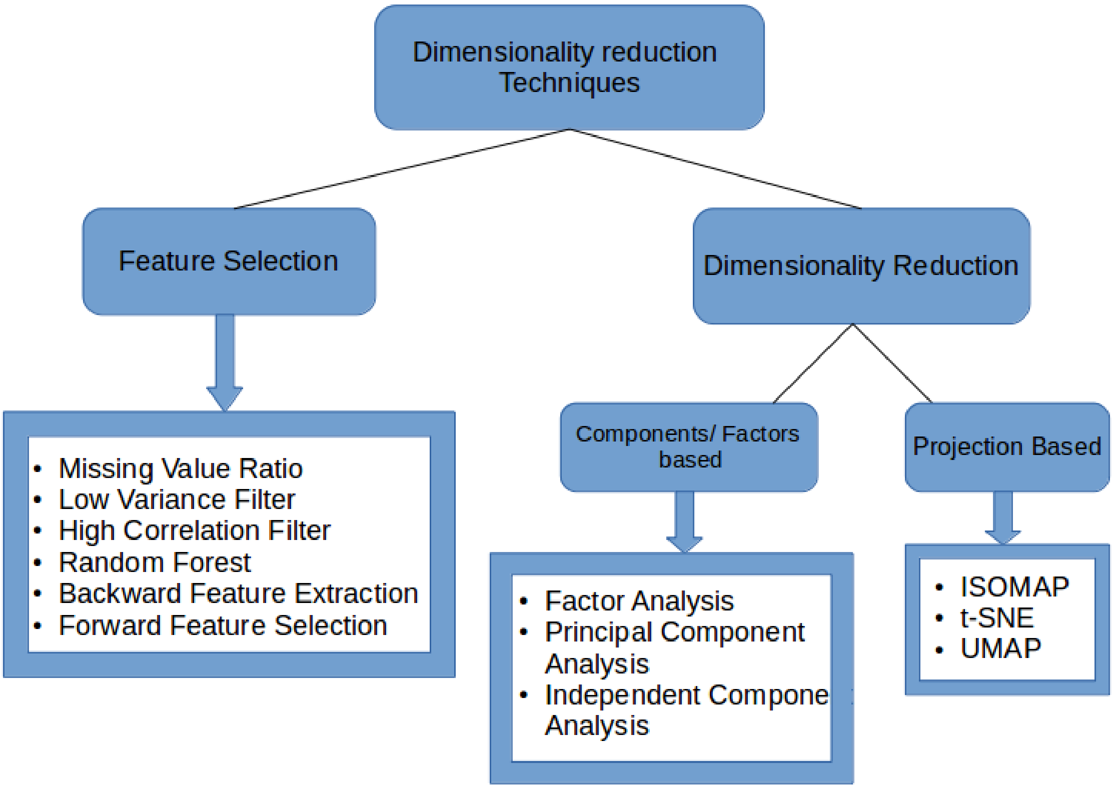
We can see that the correlation between the components obtained from UMAP is quite less as compared to the correlation between the components obtained from t-SNE. Hence, UMAP tends to give better results.

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.

Take a deep breath. We have covered quite a lot of the dimensionality reduction techniques out there. Let's briefly summarize where each of them can be used.

4. Brief Summary of when to use each Dimensionality Reduction Technique

In this section, we will briefly summarize the use cases of each dimensionality reduction technique that we covered. It's important to understand where you can, and should, use a certain technique as it helps save time, effort and computational power.


Dimensionality reduction Techniques
Feature Selection
Dimensionality Reduction
Missing Value Ratio
Low Variance Filter
High Correlation Filter
Random Forest
Backward Feature Extraction
Forward Feature Selection
Components/ Factors based
Projection Based
Factor Analysis
Principal Component Analysis
Independent Compone Analysis
ISOMAP
t-SNE
UMAP

- **Missing Value Ratio**: If the dataset has too many missing values, we use this approach to reduce the number of variables. We can drop the variables having a large number of missing values in them
- **Low Variance filter**: We apply this approach to identify and drop constant variables from the dataset. The target variable is not unduly affected by variables with low variance, and hence these variables can be safely dropped
- **High Correlation filter**: A pair of variables having high correlation increases multicollinearity in the dataset. So, we can use this technique to find highly correlated features and drop them accordingly
- **Random Forest**: This is one of the most commonly used techniques which tells us the importance of each feature present in the dataset. We can find the importance of each feature and keep the top most features, resulting in dimensionality reduction
- Both **Backward Feature Elimination** and **Forward Feature Selection** techniques take a lot of computational time and are thus generally used on smaller datasets
- **Factor Analysis**: This technique is best suited for situations where we have highly correlated set of variables. It divides the variables based on their correlation into different groups, and represents each group with a factor

- **Principal Component Analysis**: This is one of the most widely used techniques for dealing with linear data. It divides the data into a set of components which try to explain as much variance as possible
- **Independent Component Analysis**: We can use ICA to transform the data into independent components which describe the data using less number of components
- **ISOMAP**: We use this technique when the data is strongly non-linear
- **t-SNE**: This technique also works well when the data is strongly non-linear. It works extremely well for visualizations as well
- **UMAP**: This technique works well for high dimensional data. Its run-time is shorter as compared to t-SNE

End Notes

This is as comprehensive an article on dimensionality reduction as you'll find anywhere! I had a lot of fun writing it and found a few new ways of dealing with high number of variables I hadn't used before (like UMAP).

Dealing with thousands and millions of features is a must-have skill for any data scientist. The amount of data we are generating each day is unprecedented and we need to find different ways to figure out how to use it. Dimensionality reduction is a very useful way to do this and has worked wonders for me, both in a professional setting as well as in machine learning hackathons.

I'm looking forward to hearing your feedback and ideas in the comments section below.

Share this:

Like this:

Loading...

TAGS : DIMENSIONALITY REDUCTION, FACTOR ANALYSIS, INDEPENDENT COMPONENT ANALYSIS, ISOMAP, PCA, T-SNE, UMAP

NEXT ARTICLE

AI World Cup is a Fully Simulated Football Tournament Powered by Neural Networks!

PREVIOUS ARTICLE

The Ultimate Data Science and Machine Learning Blogathon – More than $2500 up for grabs!

Pulkit Sharma

My research interests lies in the field of Machine Learning and Deep Learning. Possess an enthusiasm for learning new skills and technologies.