**Dimension Reduction Algorithms**

**Common Techniques:**

1. **Missing Values Ratio**
2. **Low variance filter**
3. **High correlation filter**
4. **Random Forest**
5. **Backward feature selection**
6. **Forward feature selection**
7. **Factor analysis**
8. **PCA**
9. **Independent component analysis**
10. **Projection methods**
11. **t-SNE (t- Distributed Stochastic Neighbour Embedding**
12. **UMP**

**Not covered in this Covered in this**

**Missing value Dataset:**

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.

**Python Code:**

# import required libraries

import pandas as pd

import numpy as np

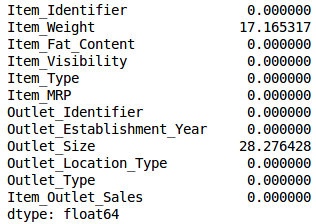
import matplotlib.pyplot as plt

# read the data

train=pd.read\_csv("Train\_UWu5bXk.csv")

# checking the percentage of missing values in each variable

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



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.

# 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%.

**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.

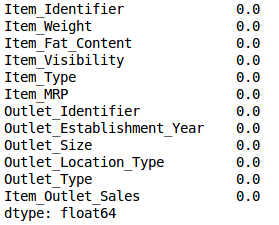
Let’s first impute the missing values in the *Item\_Weight* column using the median value of the known *Item\_Weight* observations. For the *Outlet\_Size* column, we will use the mode of the known *Outlet\_Size* values to impute the missing values:

**Python Code:**

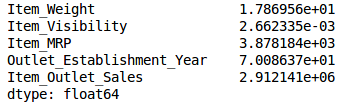
train['Item\_Weight'].fillna(train['Item\_Weight'].median(), inplace=True)

train['Outlet\_Size'].fillna(train['Outlet\_Size'].mode()[0], inplace=True)

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



train.var()



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])

**High Correlation Filter:**

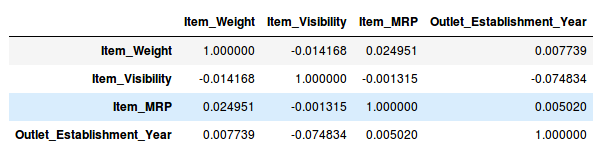
High correlation between two variables means they have similar trends and are likely to carry similar information. This can bring down the performance of some models drastically (linear and logistic regression models, for instance). We can calculate the correlation between independent numerical variables 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.**

**Python Code:**

df=train.drop('Item\_Outlet\_Sales', 1)

df.corr()



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.

**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.

**Python Code:**

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

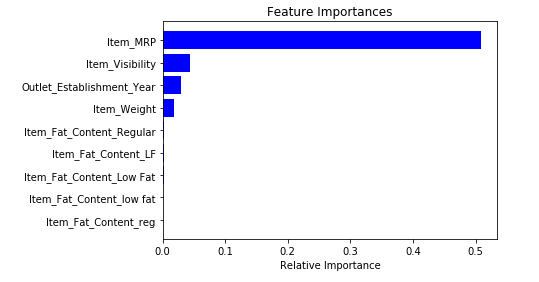
plt.title('Feature Importances')

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)

**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 its Python implementation:

**Python Code**:

from sklearn.linear\_model import LinearRegression

from sklearn.feature\_selection import RFE

from sklearn import datasets

lreg = LinearRegression()

rfe = RFE(lreg, 10) # RFE is recursive feature elimination

rfe = rfe.fit\_transform(df, train.Item\_Outlet\_Sales)

# We need to specify the algo and the number of features we want.

**Principal Component Analysis:**

PCA is used for two purposes:

1. Data visualization.
2. Speeding up the ML algorithm, if there are a lot of features.

Say we are working on the Iris dataset which has got 4 features, sepal length, sepal width, petal length and petal width. And a target that is the output variable.

**Standardizing the data using StandardScalar:**

**scalar = StandardScalar()**

**scalar.fit(training\_data)**

Standardizing the data can sometimes prove to be very crucial for PCA, as PCA’s can get effected by scaling. Standardizing the data converts the dataset features onto a **UNIT SCALE** (mean=0, variance=1) which is the requirement for optimal performance of many ML algorithms.

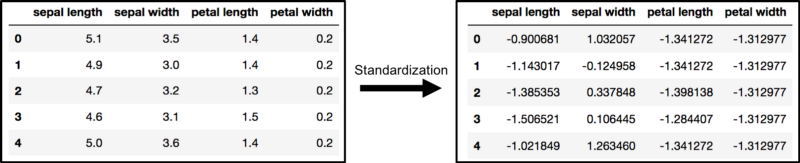
**from sklearn.preprocessing import StandardScaler**

**features = ['sepal length', 'sepal width', 'petal length', 'petal width']**

**# Separating out the features  
x = df.loc[:, features].values**

**# Separating out the target  
y = df.loc[:,['target']].values**

**# Standardizing the features  
x = StandardScaler().fit\_transform(x)**



**PCA Projection from 4D to 2D:**

The original dataset had 4 columns or features. Hence a 4D plot was reuired to visualize the data which is practically not possible. Hence using PCA we project this data onto a 2D plot, where each axis represents a Principle component, and the values in each PC are the projected distance of the sample points along each of these Principal components. PC usually don’t have any meaning associated with it. These new components or axises are the two main dimensions of variation.

**from sklearn.decomposition import PCA**

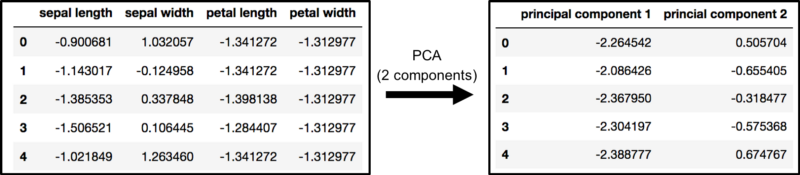
**pca = PCA(n\_components=2) # The top 2 components only**

**#We can also use this: pca=PCA(0.95)**

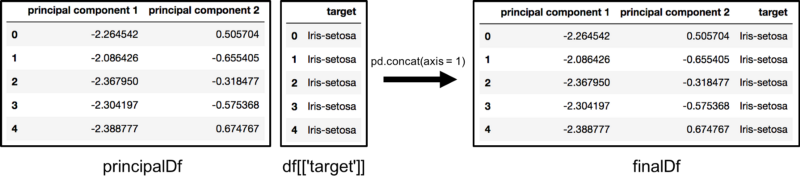
**# This tells scikitlearn choose the minimum number of parameters such that 95% of variance is retained.**

**principalComponents = pca.fit\_transform(x)**

**principalDf = pd.DataFrame(data = principalComponents  
 , columns = ['principal component 1', 'principal component 2'])**



**finalDf = pd.concat([principalDf, df[['target']]], axis = 1)**



**Explained Variance**:

The variance of a PC tells us how much variance or information is being contributed by a particular PC. Say we have 4 features x1, x2, x3, x4 and there are 4 PC as well each associated to a feature. By using the attribute **explained\_variance\_ratio**\_, we can see which out of the three features contribute most to the variance.

**pca.explained\_variance\_ratio\_**

**Sometimes a scree plot is also used to visualize the variance in form of a plot.**