feature transformation is a common technique used to improve the accuracy of models. One of the reasons for transformation is to handle skewed data, which can negatively affect the performance of many machine learning algorithms.

**What is Feature Transformation?**

**1.**It is a technique by which we can boost our model performance. Feature transformation is a mathematical transformation in which we apply a mathematical formula to a particular column(feature) and transform the values which are useful for our further analysis.

**2.** It is also known as **Feature Engineering,** which is creating new features from existing features that may help in improving the model performance.

**3.** It refers to the family of algorithms that create new features using the existing features. These new features may not have the same interpretation as the original features, but they may have more explanatory power in a different space rather than in the original space.

**4.**This can also be used for **Feature Reduction**. It can be done in many ways, by linear combinations of original features or by using non-linear functions.

**5.** It helps machine learning algorithms to converge faster.

**Why These Transformations?**

**1.** Some Machine Learning models, like **Linear and Logistic regression**, assume that the variables follow a normal distribution. More likely, variables in real datasets will follow a skewed distribution.

**2.** By applying some transformations to these skewed variables, we can map this skewed distribution to a normal distribution so, this can increase the performance of our models.

**Goal of Feature Transformations**

As we know that **Normal Distribution** is a very important distribution in **Statistics**, which is key to many statisticians for solving problems in statistics. Usually, the data distribution in Nature follows a Normal distribution (**examples like – age, income, height, weight, etc.,**). But the features in the real-life data are not normally distributed, however it is the best approximation when we are not aware of the underlying distribution pattern.

**Transformations present in scikit-learn**

Sklearn has three Transformations-

**1.**  Function Transformation

**2.**  Power Transformation

**3.**  Quantile transformation

**Function Transformations**

**LOG TRANSFORMATION:**

– Generally, these transformations make our data close to a normal distribution but are not able to exactly abide by a normal distribution.

–  This transformation is not applied to those features which have negative values.

– This transformation is mostly applied to**right-skewed data.**

– Convert data from addictive Scale to multiplicative scale i,e, **linearly distributed data**.

**RECIPROCAL TRANSFORMATION**

– This transformation is not defined for zero.

– It is a powerful transformation with a **radical effect**.

– This transformation reverses the order among values of the same sign, so large values become smaller and vice-versa.

**SQUARE TRANSFORMATION**

– This transformation mostly applies to **left-skewed data**.

**SQUARE ROOT TRANSFORMATION:**

– This transformation is defined only for **positive numbers**.

– This transformation is weaker than Log Transformation.

– This can be used for reducing the **skewness of right-skewed data**.

**CUSTOM TRANSFORMATION**

You can refer to the [Link](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html#:~:text=A%20FunctionTransformer%20forwards%20its%20X,%2C%20doing%20custom%20scaling%2C%20etc.) to read more about Function Transformations.

**Power Transformations**

– Used when the desired output is more **“Gaussian”** like.

– Currently has**‘Box-Cox’** and **‘Yeo-Johnson’** transforms.

– **Box-cox** requires the input data to be strictly positive(not even zero is acceptable).

– for features that have zeroes or negative values, **Yeo-Johnson** comes to the rescue.

**BOX-COX TRANSFORMATION:**  Sqrt/sqr/log are the special cases of this transformation.

**YEO-JOHNSON TRANSFORMATION:**  It is a variation of the **Box-Cox** transform.

What is the Skewness Threshold to Apply a Transformation?

Skewness is a measure of the asymmetry of a probability distribution, and a threshold value for skewness to determine when to apply a transformation is not fixed.

A common approach to determining whether to transform a feature with skewness is to use a threshold value of 0.5 or higher. This is based on the observation that most statistical distributions have a skewness between -0.5 and 0.5. Therefore, if the skewness of a feature is above this threshold, a transformation is applied to make the data less skewed and more normally distributed.

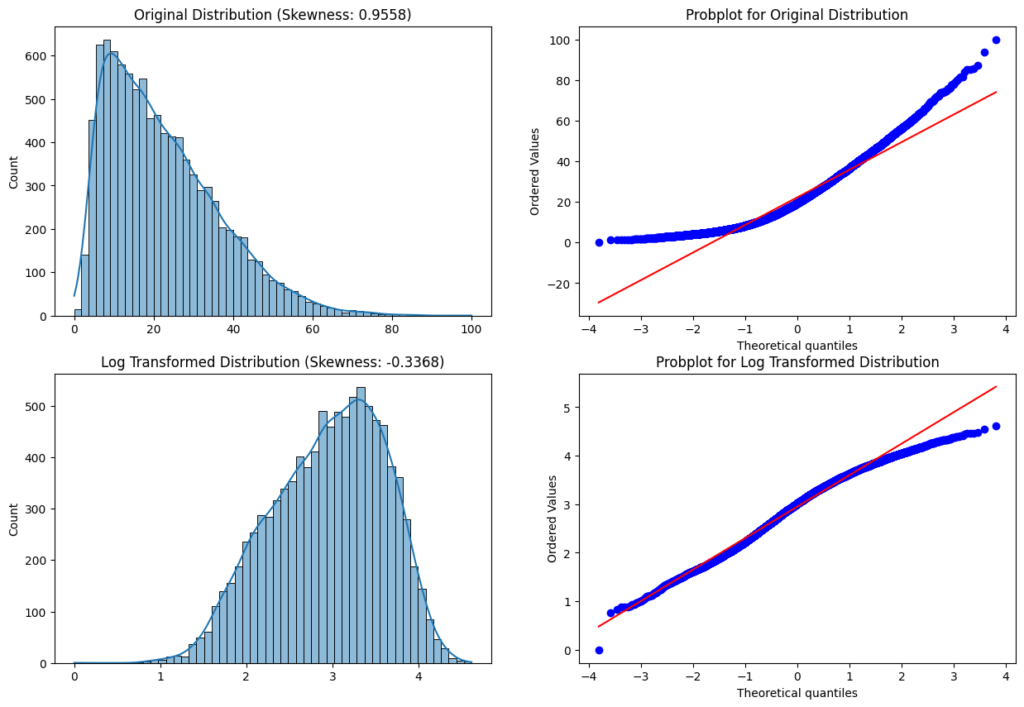
However, this threshold value is not fixed and can depend on the specific problem, the dataset, and the algorithm being used.

If you want to know when and how to transform the target variable in a machine-learning project, take a look at the article [Transform Target Variable](https://datasciencewithchris.com/transform-the-target-variable/).

Log Transformation

The log transformation is a commonly used technique in machine learning to transform skewed or non-normal data into a more normal distribution by computing the natural logarithm of a variable. The log transformation works by compressing large values and expanding small values. When applied to skewed data, it can reduce the influence of extreme values and make the distribution more symmetrical.

df['feature'] = np.log(df['feature'])



Log Transform Features that contain the value 0

The general problem with log transforming features that contain the value 0 is that the log of 0 is not defined (-inf).  
Therefore you can use the little trick to add 1 to the value: log(x+1). Numpy has therefore a build-in function: np.log1p(x). In this case, the log1p transformation of the value 0 is still 0.

df['feature'] = np.log1p(df['feature'])

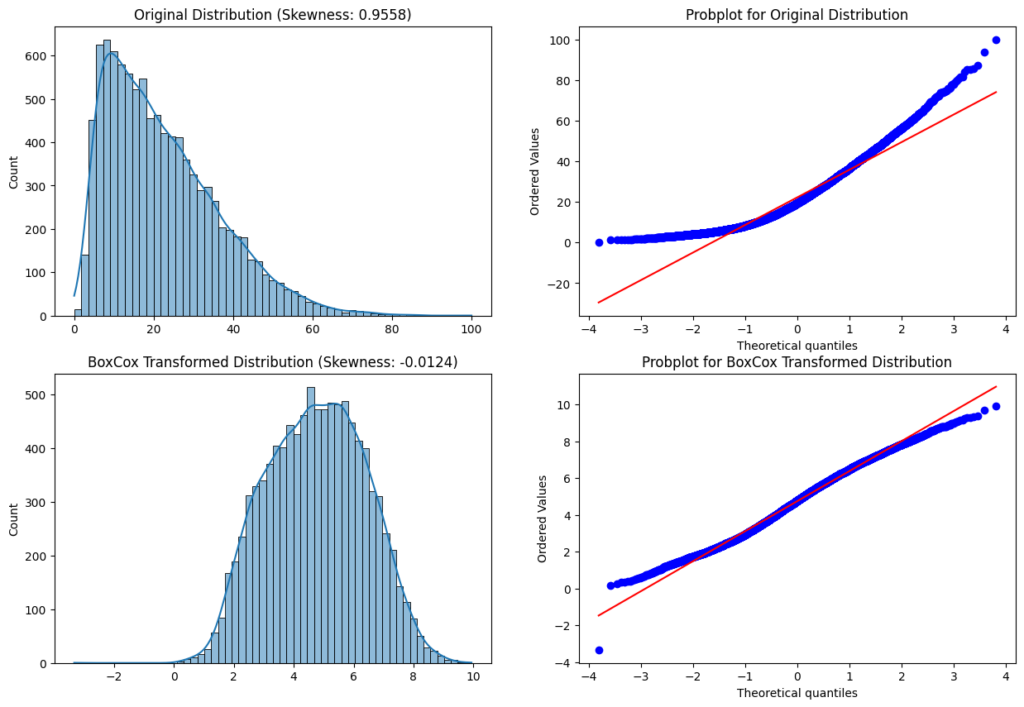
Disadvantages of Log Transformation

* The main disadvantage of the log transformation is that it requires the input data to be positive.

Box-Cox Transformation

The Box-Cox transformation is a mathematical transformation that can be used to transform non-normal data into a more normal distribution. It was introduced by statisticians George Box and David Cox in 1964.

df['feature'] = scipy.stats.boxcox(df['feature'], lambda)



The value lambda is the power of the transformation. If you use the [scipy library](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.boxcox.html" \t "_blank) for the transformation you don’t have to set a value for lambda. If lambda is None a range of possible values is tested for maximizing the log-likelihood function.

w={log(x)if λ=0,(x−1)λotherwise*w*={*log*(*x*)*λ*(*x*−1)​​if *λ*=0,otherwise​

Notice that when lambda =1 the transformed data shifts down by 1 but the distribution does not change. In this case, the data is already normally distributed.

Advantages of Box-Cox Transformation

* The Box-Cox transformation has the advantage over other power transformations in that it can automatically determine the appropriate power to use based on the data.

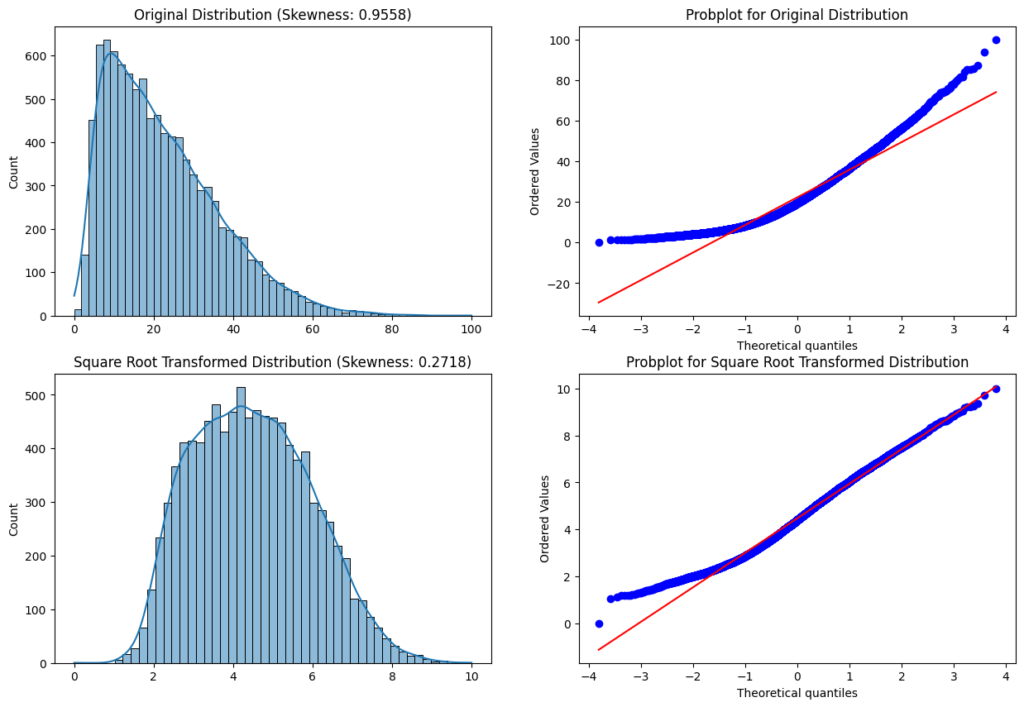
Disadvantages of Box-Cox Transformation

* The Box-Cox transformation requires the input data to be positive, but the Yeo-Johnson transformation is quite similar and does not have the restriction that all values have to be positive.

Square Root Transformation

The square root transformation works analog to the log transformation by applying the square root to a variable or feature. When the square root transformation is applied to a positively skewed dataset, the transformed dataset will have a more normal distribution.

df['feature'] = np.sqrt(df['feature'])



Advantages of Square Root Transformation

* The main advantage of square root transformation is, it can be applied to zero values.

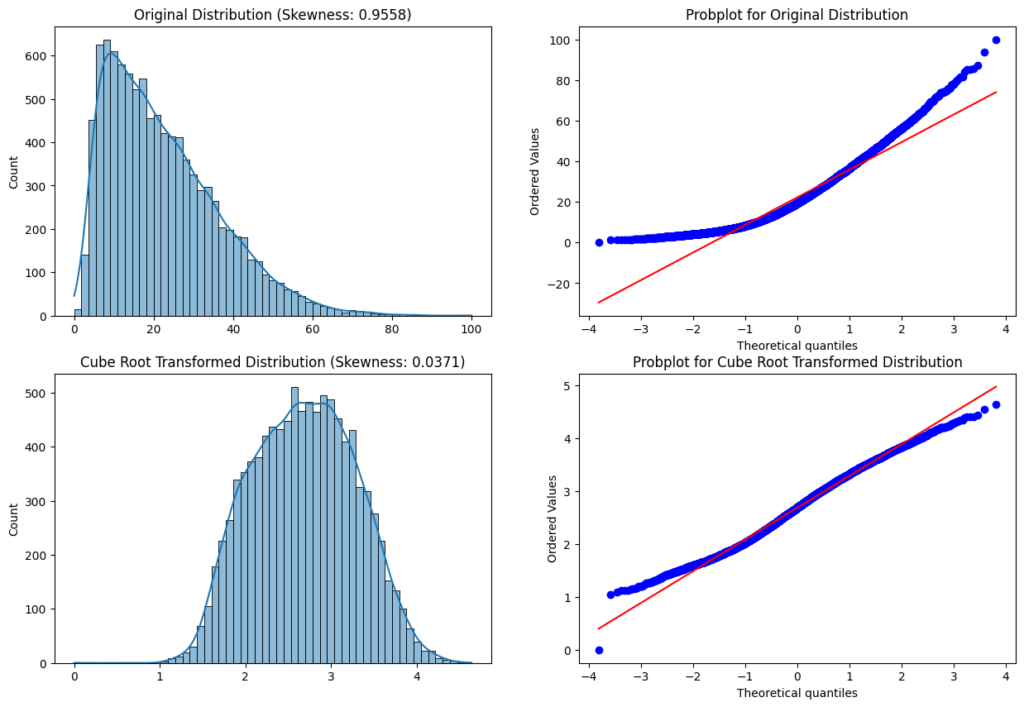
Disadvantages of Square Root Transformation

* The transformation is weaker than the log transformation.
* The square root transformation can not be applied to negative values

Cube Root Transformation

The cube root transformation is a mathematical transformation that takes the cube root of a variable to make a distribution more normal. It is similar to other power transformations such as the square root and the logarithmic transformation but has different properties. The cube root transformation works by compressing large values and expanding small values.

df['feature'] = np.cbrt(df['feature'])



Advantages of Cube Root Transformation

* The main advantage of the cube root transformation is, it can be applied to zero and negative values.

Yeo-Johnson Transformation

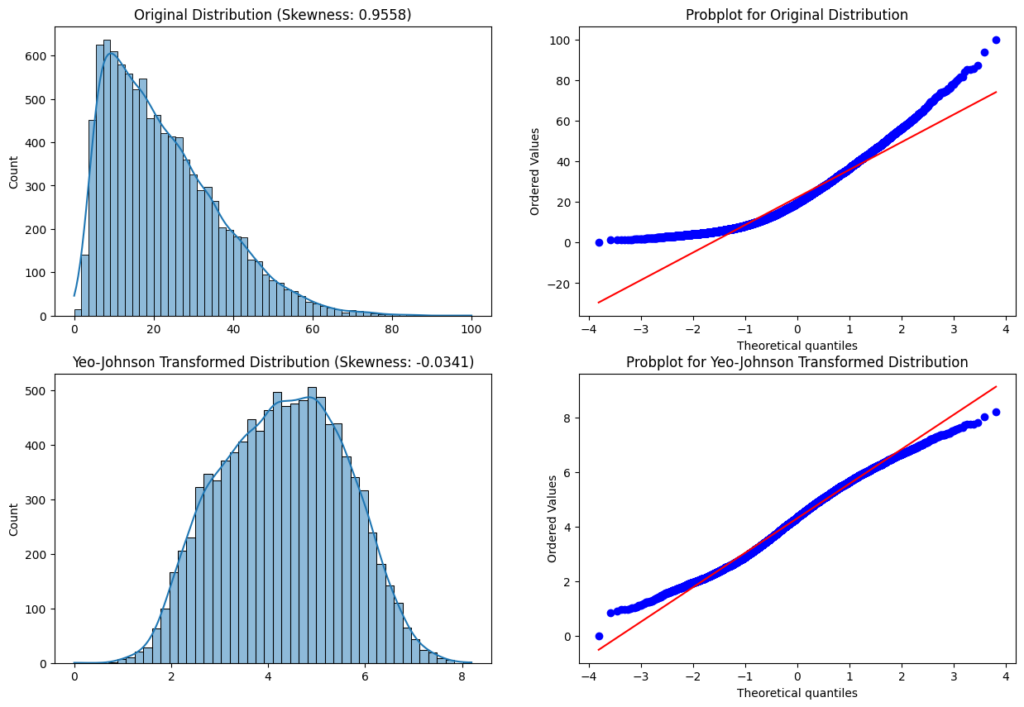
The Yeo-Johnson transformation is a widely used data transformation technique that can be used to transform non-normal data into a more normal distribution. It was introduced by Robert Yeo and Robert Johnson in 2000 as an improvement over the Box-Cox transformation, which has limitations when dealing with data that contain negative values.

df['feature'] = scipy.stats.yeojohnson(df['feature'], lambda)

The value lambda is the power of the transformation. If you use the [scipy library](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.yeojohnson.html" \t "_blank) for the transformation you don’t have to set a value for lambda. If lambda is None a range of possible values is tested for maximizing the log-likelihood function.

w={((x+1)λ−1λif λ≠0,x>=0ln(x+1)if λ=0,y>=0−((x+1)(2−λ)−1)2−λif λ≠2,y<0−ln(−x+1)if λ=2,y<0*w*=⎩⎨⎧​*λ*((*x*+1)*λ*−1​*ln*(*x*+1)2−*λ*−((*x*+1)(2−*λ*)−1)​−*ln*(−*x*+1)​if *λ*=0,*x*>=0if *λ*=0,*y*>=0if *λ*=2,*y*<0if *λ*=2,*y*<0​

A value of lambda=1 produces the identity transformation.



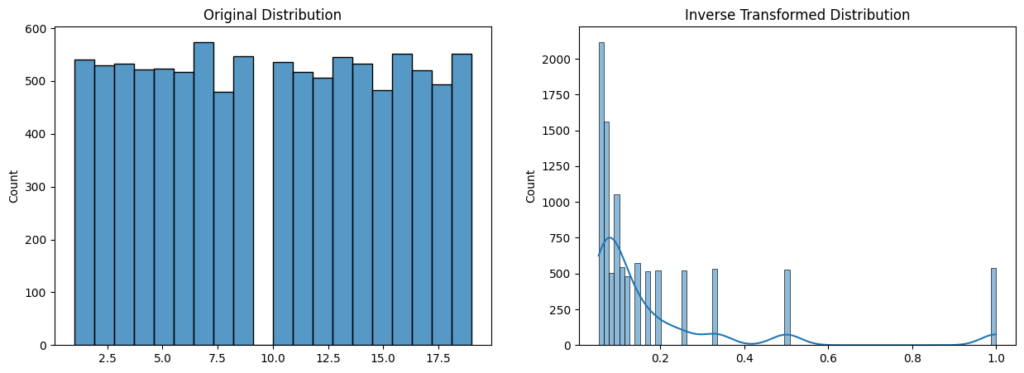
Advantages of Yeo-Johnson Transformation

* The Yeo-Johnson transformation transforms zero, positive as well as negative values.

Inverse Transformation

The inverse transformation is very useful when the feature is a count or a rate, and we want to transform it into a continuous variable.

df['feature'] = 1/(df['feature'])



Advantages of Inverse Transformation

* The transformation can be used on a discrete variable or series.

Disadvantages of Inverse Transformation

The inverse transformation can only be applied to non-0 values.

**Function Transformations**

**Step-1: Import necessary Dependencies**

import pandas as pd

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

import seaborn as sns

**Step-2: Import useful packages**

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.model\_selection import cross\_val\_score

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.preprocessing import FunctionTransformer

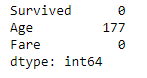
import pandas as pd

df = pd.read\_csv('titanic.csv',usecols=['Age','Fare','Survived'])

print(df.head())

**Step-4: Find the number of missing values per column**

print(df.isnull().sum())

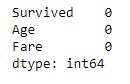


**Step-5: Fill the missing values of the “Age” column with the median of the non-missing values**

df['Age'].fillna(df['Age'].median(),inplace=True)

**Step-6: Now, again check there is any missing value or not**

print(df.isnull().sum())



**Step-7: Separate independent and dependent variables**

X = df.iloc[:,1:3]

y = df.iloc[:,0]

**Step-8: Split our dataset into train and test subsets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.33,random\_state=105)

**Step-9: Plot the probability density function(pdf) and the Q-Q plot for the “Age” column**

import warnings

warnings.filterwarnings('ignore')

plt.figure(figsize=(10,2))

plt.subplot(121)

sns.distplot(X\_train['Age'])

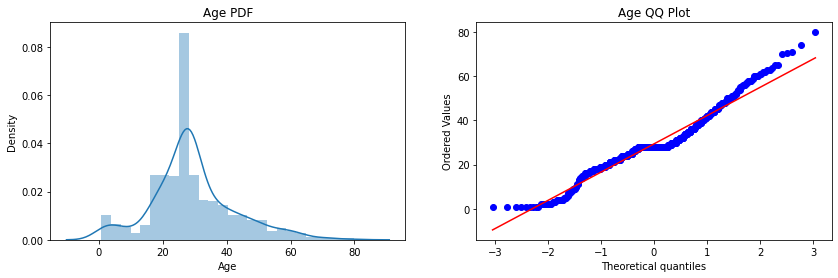
plt.title('Age PDF')

plt.subplot(122)

stats.probplot(X\_train['Age'], dist="norm", plot=plt)

plt.title('Age QQ Plot')

plt.show()



**Step-10: Plot the probability density function(pdf) and the Q-Q plot for the “Fare” column**

plt.figure(figsize=(10,2))

plt.subplot(121)

sns.distplot(X\_train['Fare'])

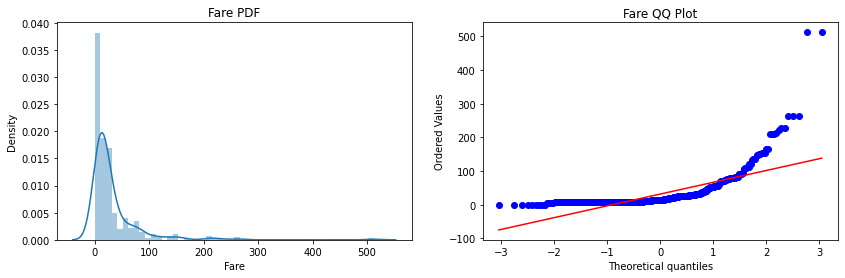
plt.title('Age PDF')

plt.subplot(122)

stats.probplot(X\_train['Fare'], dist="norm", plot=plt)

plt.title('Age QQ Plot')

plt.show()



**Step-11: Form our Logistic Regression and Decision Tree Classifier**

lr = LogisticRegression()

dt = DecisionTreeClassifier()

**Step-12: Check the performance of both the classifiers on the test dataset(before transformation)**

lr.fit(X\_train,y\_train)

dt.fit(X\_train,y\_train)

y\_pred\_lr = lr.predict(X\_test)

y\_pred\_dt = dt.predict(X\_test)

print("Accuracy LR",accuracy\_score(y\_test,y\_pred\_lr))

print("Accuracy DT",accuracy\_score(y\_test,y\_pred\_dt))

accuracy

**Step-13: Transform our dataset using log transformation and then repeat step-11 and step-12.**

trf = FunctionTransformer(func=np.log1p)

X\_train\_transformed = trf.fit\_transform(X\_train)

X\_test\_transformed = trf.transform(X\_test)

lr = LogisticRegression()

dt= DecisionTreeClassifier()

lr.fit(X\_train\_transformed,y\_train)

dt.fit(X\_train\_transformed,y\_train)

y\_pred\_lr = lr.predict(X\_test\_transformed)

y\_pred\_dt = dt.predict(X\_test\_transformed)

print("Accuracy LR",accuracy\_score(y\_test,y\_pred\_lr))

print("Accuracy DT",accuracy\_score(y\_test,y\_pred\_dt))

Transform our dataset

**Step-14: Plot the probability density function(pdf) and the Q-Q plot for the “Age” column**

plt.figure(figsize=(10,2))

plt.subplot(121)

stats.probplot(X\_train['Age'], dist="norm", plot=plt)

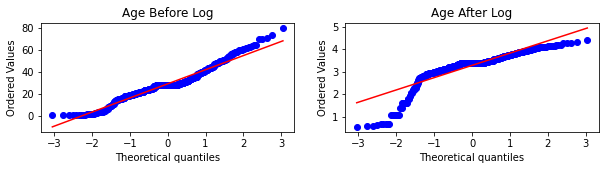
plt.title('Age Before Log')

plt.subplot(122)

stats.probplot(X\_train\_transformed['Age'], dist="norm", plot=plt)

plt.title('Age After Log')

plt.show()



**Step-15: Plot the probability density function(pdf) and the Q-Q plot for the “Fare” column**

plt.figure(figsize=(10,2))

plt.subplot(121)

stats.probplot(X\_train['Fare'], dist="norm", plot=plt)

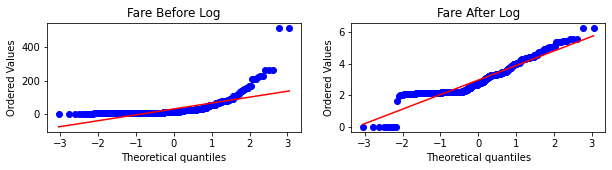
plt.title('Fare Before Log')

plt.subplot(122)

stats.probplot(X\_train\_transformed['Fare'], dist="norm", plot=plt)

plt.title('Fare After Log')

plt.show()



**Conclusion:**Here, we implement the **“Log-Transformation”** but by changing the parameters inside the functions you can easily implement other transformations as well.

– The idea behind training two models is to verify that **Tree-based model** accuracy is not much affected by doing feature transformations, but models like **Linear Regression, Logistic Regression** performance increase up to some extent.

**– How to Decide whether the given Q-Q plot corresponds to normal distribution or not?**

In the Q-Q plots, if the variable follows a Normal distribution, then the variable’s values should fall in a line of slope **45-degree(y=x)** when plotted against the theoretical quantiles.

– The main observation is that here the**“Fare”** column before transformation is **right-skewed** but after transformation comes closer to a normal distribution, but the **“Age”**column is not more affected since it is approx. normally distributed before the transformation.

– The accuracy of the model after doing transformation is increased for Logistic regression, but for verification of how much accuracy changes by transformation, you have to **cross-validate** over the data and find the average accuracy to gain better insights.

**Power Transformations**

**Step-1: Import necessary Dependencies**

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import scipy.stats as stats

**Step-2: Import useful packages**

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import cross\_val\_score

from sklearn.linear\_model import LinearRegression

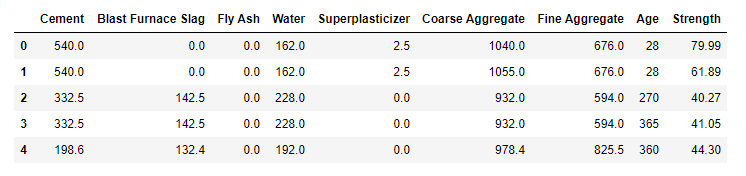
from sklearn.metrics import r2\_score

from sklearn.preprocessing import PowerTransformer

**Step-3: Read and Load the dataset**

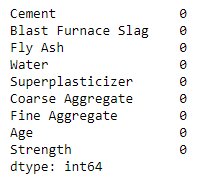
df = pd.read\_csv('concrete\_data.csv')

df.head()



**Step-4: Find the number of missing values per column**

print(df.isnull().sum())



**Step-5: Finding Statistical measures for columns**

df.describe()

**Step-6: Separate independent and dependent variables**

X = df.iloc[:,:8]

y = df.iloc[:,-1]

**Step-7: Split our dataset into train and test subsets**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.33,random\_state=105)

**Step-8: Train our Linear Regression model and check the metric**

lr = LinearRegression()

lr.fit(X\_train,y\_train)

y\_pred = lr.predict(X\_test)

print(r2\_score(y\_test,y\_pred))

feature transfoemation step 8 score

**Step-9: Plotting the distplots without any transformation**

import warnings

warnings.filterwarnings('ignore')

for col in X\_train.columns:

plt.figure(figsize=(14,4))

plt.subplot(121)

sns.distplot(X\_train[col])

plt.title(col)

plt.subplot(122)

stats.probplot(X\_train[col], dist="norm", plot=plt)

plt.title(col)

plt.show()

**Step-10: Apply the Box-Cox transformation**

pt = PowerTransformer(method='box-cox')

X\_train\_transformed = pt.fit\_transform(X\_train+0.0000001)

X\_test\_transformed = pt.transform(X\_test+0.0000001)

pd.DataFrame({'cols':X\_train.columns,'box\_cox\_lambdas':pt.lambdas\_})

**Step-11: Train our model on transformed data and check the metric**

lr = LinearRegression()

lr.fit(X\_train\_transformed,y\_train)

y\_pred2 = lr.predict(X\_test\_transformed)

print(r2\_score(y\_test,y\_pred2))

**Step-12: Plotting the distplots after transformation**

X\_train\_transformed = pd.DataFrame(X\_train\_transformed,columns=X\_train.columns)

for col in X\_train\_transformed.columns:

plt.figure(figsize=(14,4))

plt.subplot(121)

sns.distplot(X\_train[col])

plt.title(col)

plt.subplot(122)

sns.distplot(X\_train\_transformed[col])

plt.title(col)

plt.show()

**Conclusion:**Here we implement the **Box-Cox** transformation but by changing the parameters inside the function you can implement **Yeo-Johnson** Transformation also.

– The idea behind running the **describe()** function is to check the values present in the columns and verify the assumptions of Power Transformation i.e, Box-Cox transformation only accepts strictly positive numbers.

– We also observe that there is an increment in the accuracy of the model, since our problem statement is a **“Regression”** Problem statement and we apply the linear regression, and by transformations, we make the columns closer to a normal distribution, which satisfies the assumptions of the linear regression algorithm.

– We add a very small value to all the points of the dataset so that no point value remains exactly zero and our assumption still holds for Box-Cox transformation.