Gaurav Sahu

githublink: https://github.com/GauravSahu13

Feature Engineering

1) Data Cleaning:

1.1) Handling Missing Values:

Missing values are common in real-world datasets and can adversely affect model performance. Imputation techniques like mean, median, or mode replacement can be used to fill missing values. More advanced techniques include KNN imputation, which estimates missing values based on the values of the nearest neighbors.

Why are their Missing values?

- Data Entry Errors,Non-Response-hesitate to put down the information (eg: Men--salary / Women---age)

What are the different types of Missing Data?

(i) Missing Completely at Random(MCAR)

- In this scenario, the probability of a value being missing is unrelated to the observed or missing data. It occurs randomly throughout the dataset, and there is no systematic pattern to the missingness.
- Example: A survey where participants fail to answer certain questions due to accidental oversight.

(ii) Missing Data Not At Random(MNAR)

- The missingness is related to the missing values themselves, even after considering observed data. This type of missingness is more challenging to handle because it implies that the missing data is systematically different from the observed data.
- Example: In a survey on income where high-income individuals are less likely to disclose their income, the missingness of income data may be related to the income level itself.

(iii) Missing At Random(MAR)

- The probability of a value being missing depends only on the observed data and not on the missing data itself. In other words, the missingness can be explained by other variables in the dataset
- Example: In a survey where income information is missing for unemployed individuals, the missingness of income may be related to employment status (an observed variable).

Techniques of handling missing values:

- Mean/Median/Mode replacement
- Random Sample Imputation

- Capturing NAN values with a new feature
- End of Distribution Imputation
- Arbitrary Imputation
- Frequent Categories Imputation

1.2) Outlier Detection and Treatment:

Outliers are data points that significantly differ from other observations in the dataset. They can distort statistical analyses and machine learning models. Techniques for handling outliers include truncation (replacing outliers with a specified threshold), winsorization (replacing outliers with the nearest non-outlier value), or removing them altogether if they are deemed erroneous.

Dataset link: https://github.com/GauravSahu13/EDA/tree/Feature_Enginerring

In [133…

```
# Importing Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
# Reading CSV file from local
df = pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv')
df.head()
```

Out[2]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	E
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	

In [3]:

```
# finding null value
df.isnull().sum()
```

Out[3]:
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

- here features 'Age', 'Cabin', 'Embarked' have 177, 687 & 2 missing values respectively

Age and Cabin are related to each other so it is an example of Missing Data Not At Random(MNAR)

Embarked is not related to any other feature so it is an example of Missing Data At Random(MAR)

In [5]:
```
# Nan values replace by '1' & not nan value with '0'

df['cabin_null']=np.where(df['Cabin'].isnull(),1,0)
# find the percentage of null values
df['cabin_null'].mean()
```

Out[5]:
```
0.7710437710437711
```

- feature cabin has 77% of null value

In [6]:
```
df.columns
```

Out[6]:
```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked', 'cabin_null'],
      dtype='object')
```

In [7]:
```
df.groupby(['Survived'])['cabin_null'].mean()
```

Out[7]:
```
Survived
0    0.876138
1    0.602339
Name: cabin_null, dtype: float64
```

- 60% missing value of Cabin ---> Survived
- 87% missing value of Cabin ---> Not Survived

(i) Mean-Median-Mode Replacement

When should we apply?

- We solve this by replacing the NAN value with the most frequent occurance of the variables

Advantages

- Easy to Implement(robust to outliers)
- Faster way to obtain the complete dataset

Disadvantages

- Change or Distortion in the original variance
- Impacts Correlation

In [135…

```
df=pd.read_csv('titanic.csv',usecols=['Age','Fare','Survived'])
## Lets go and see the percentage of missing values
df.isnull().mean()
```

Out[135…

```
Survived    0.000000
Age         0.198653
Fare        0.000000
dtype: float64
```

- Age has 19.8% missing value

filling nan value with median of the data

In [141…

```
median=df.Age.median()
median
# for feature Age median is 28
```

Out[141… 28.0

In [143…

```
def impute_nan(df,feature,median):
    df[feature+"_median"]=df[feature].fillna(median)
impute_nan(df,'Age',median)
df.head()
```

Out[143…

	Survived	Age	Fare	Age_median
0	0	22.0	7.2500	22.0
1	1	38.0	71.2833	38.0
2	1	26.0	7.9250	26.0
3	1	35.0	53.1000	35.0
4	0	35.0	8.0500	35.0

In [145…

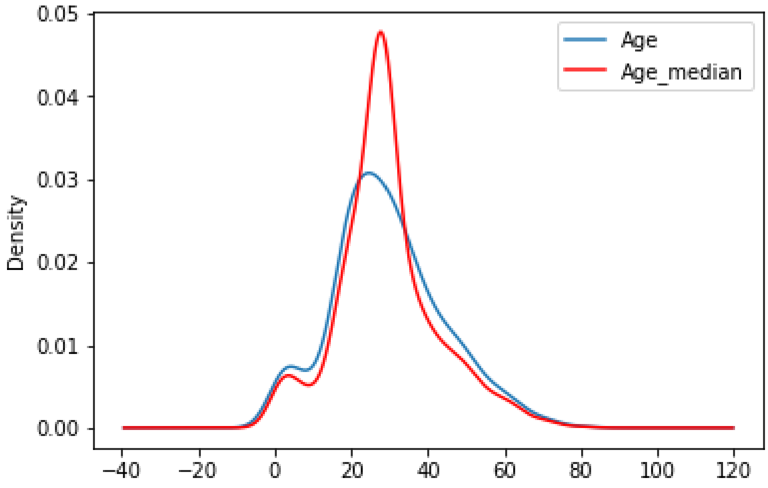
```
print(df['Age'].std())
# after replacement nan value
print(df['Age_median'].std())
```

```
14.526497332334044
13.019696550973194
```

In [14]:

```
fig = plt.figure()
ax = fig.add_subplot(111)
df['Age'].plot(kind='kde', ax=ax)
df.Age_median.plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

Out[14]: <matplotlib.legend.Legend at 0x21358d35820>



- Blue line represents feature 'Age' with nan value
- Red line represents feature 'Age' with not nan value

(ii) Random Sample Imputation

Aim: Random sample imputation consists of taking random observation from the dataset and we use this observation to replace the nan values

When should it be used?

- It assumes that the data are missing completely at random(MCAR)

Advantages

- There is less distortion in variance

Disadvantages

- Every situation randomness wont work

In [146…

```
df.head()
```

Out[146…

	Survived	Age	Fare	Age_median
0	0	22.0	7.2500	22.0
1	1	38.0	71.2833	38.0
2	1	26.0	7.9250	26.0
3	1	35.0	53.1000	35.0
4	0	35.0	8.0500	35.0

In [147…

```
df.isnull().sum()
```

Out[147…

```
Survived        0
Age           177
Fare            0
Age_median      0
dtype: int64
```

In [149…

```
df['Age'].isnull().sum()
```

Out[149…

```
177
```

In [164…

```
# dropping all the nan value
df['Age'].dropna().sample(df['Age'].isnull().sum(),random_state=0)
```

Out[164…

```
423    28.00
177    50.00
305     0.92
292    36.00
889    26.00
       ...
539    22.00
267    25.00
352    15.00
99     34.00
689    15.00
Name: Age, Length: 177, dtype: float64
```

- values randomly replace by another value (eg 423 replace by 28)

In [165…

```
df[df['Age'].isnull()].index
# getting index of nan value
```

Out[165…

```
Int64Index([  5,  17,  19,  26,  28,  29,  31,  32,  36,  42,
            ...
            832, 837, 839, 846, 849, 859, 863, 868, 878, 888],
           dtype='int64', length=177)
```

In [170…

```
median=df.Age.median()
def impute_nan(df,feature,median):
    df[feature+"_median"]=df[feature].fillna(median)
    df[feature+"_random"]=df[feature]
    ##It will have the random sample to fill the na
    random_sample=df[feature].dropna().sample(df[feature].isnull().sum(),random_stat
    ##pandas need to have same index in order to merge the dataset
    random_sample.index=df[df[feature].isnull()].index
    df.loc[df[feature].isnull(),feature+'_random']=random_sample

impute_nan(df,"Age",median)
df.head(20)
```

Out[170…

	Survived	Age	Fare	Age_median	Age_random
0	0	22.0	7.2500	22.0	22.00
1	1	38.0	71.2833	38.0	38.00
2	1	26.0	7.9250	26.0	26.00
3	1	35.0	53.1000	35.0	35.00
4	0	35.0	8.0500	35.0	35.00
5	0	NaN	8.4583	28.0	28.00
6	0	54.0	51.8625	54.0	54.00
7	0	2.0	21.0750	2.0	2.00
8	1	27.0	11.1333	27.0	27.00
9	1	14.0	30.0708	14.0	14.00

	Survived	Age	Fare	Age_median	Age_random
10	1	4.0	16.7000	4.0	4.00
11	1	58.0	26.5500	58.0	58.00
12	0	20.0	8.0500	20.0	20.00
13	0	39.0	31.2750	39.0	39.00
14	0	14.0	7.8542	14.0	14.00
15	1	55.0	16.0000	55.0	55.00
16	0	2.0	29.1250	2.0	2.00
17	1	NaN	13.0000	28.0	50.00
18	0	31.0	18.0000	31.0	31.00
19	1	NaN	7.2250	28.0	0.92

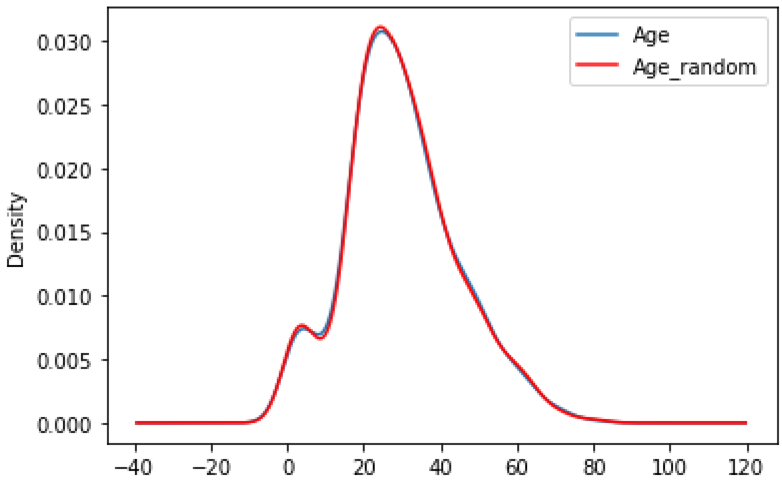
- As we can see Nan value of index 5,17,19 replace by random values (28,50,0.92)

In [176…

```
fig = plt.figure()
ax = fig.add_subplot(111)
df['Age'].plot(kind='kde', ax=ax)
df.Age_random.plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

Out[176…

```
<matplotlib.legend.Legend at 0x2135f318880>
```



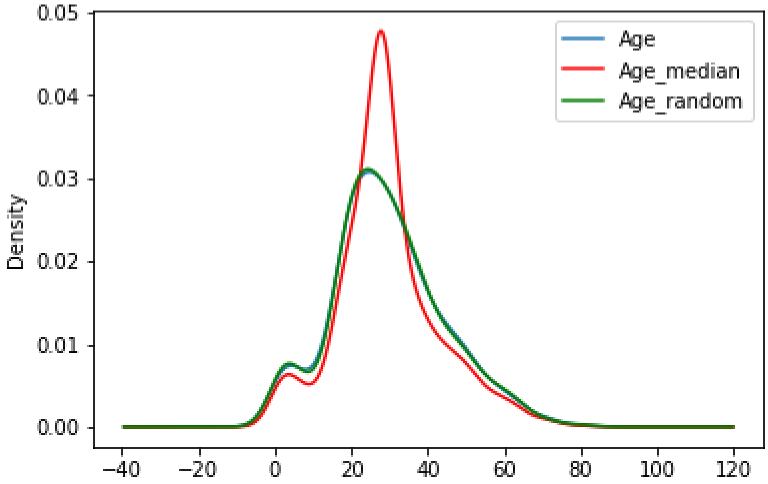
- here distortion is less compare to median graph

In [177…

```
fig = plt.figure()
ax = fig.add_subplot(111)
df['Age'].plot(kind='kde', ax=ax)
df.Age_median.plot(kind='kde', ax=ax, color='red')
df.Age_random.plot(kind='kde', ax=ax, color='green')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

Out[177…

```
<matplotlib.legend.Legend at 0x2135f3ba520>
```



In [ ]:

(iii) Capturing NAN values with a new feature

It works well if the data are not missing completely at random

Advantages

- Easy to implement
- Captures the importance of missing values

Disadvantages

- Creating Additional Features(Curse of Dimensionality)

In [178…

```
df.head()
```

Out[178…

	Survived	Age	Fare	Age_median	Age_random
0	0	22.0	7.2500	22.0	22.0
1	1	38.0	71.2833	38.0	38.0
2	1	26.0	7.9250	26.0	26.0
3	1	35.0	53.1000	35.0	35.0
4	0	35.0	8.0500	35.0	35.0

In [179…

```
df['Age_NAN']=np.where(df['Age'].isnull(),1,0)
df.head()
```

Out[179…

	Survived	Age	Fare	Age_median	Age_random	Age_NAN
0	0	22.0	7.2500	22.0	22.0	0
1	1	38.0	71.2833	38.0	38.0	0
2	1	26.0	7.9250	26.0	26.0	0
3	1	35.0	53.1000	35.0	35.0	0
4	0	35.0	8.0500	35.0	35.0	0

In [180…

```
df.Age.median()
```

Out[180…

```
28.0
```

In [181…

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

In [182…

```
df.head(10)
```

Out[182…

	Survived	Age	Fare	Age_median	Age_random	Age_NAN
0	0	22.0	7.2500	22.0	22.0	0
1	1	38.0	71.2833	38.0	38.0	0
2	1	26.0	7.9250	26.0	26.0	0
3	1	35.0	53.1000	35.0	35.0	0
4	0	35.0	8.0500	35.0	35.0	0
5	0	28.0	8.4583	28.0	28.0	1
6	0	54.0	51.8625	54.0	54.0	0
7	0	2.0	21.0750	2.0	2.0	0
8	1	27.0	11.1333	27.0	27.0	0
9	1	14.0	30.0708	14.0	14.0	0

- NaN value replace by '1' & not NaN by '0'

(iv) End of Distribution imputation

- In this method we replace missing values with far end values or extreme
- Far end value means the values after 3rd standard deviation

Advantages

- Easy to implement & Captures the importance of missing values

Disadvantages

- Distorts the original distribution of the variable.
- If missingness is not important, it may mask the predictive power of the original variable by distorting its distribution.
- If number of NA is big, it will mask true outliers in the distribution

In [190…

```
df1=pd.read_csv('titanic.csv', usecols=['Age','Fare','Survived'])
df1.head()
```

Out[190…

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833

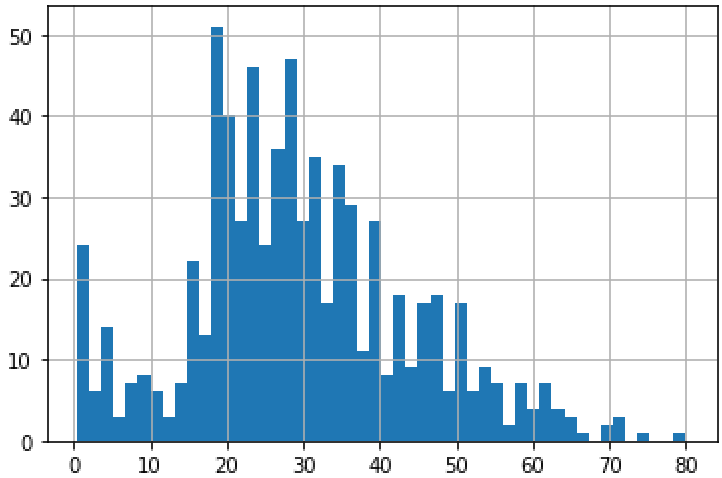
	Survived	Age	Fare
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [191…

```
df1.Age.hist(bins=50)
```

Out[191…

```
<Axes: >
```



In [192…

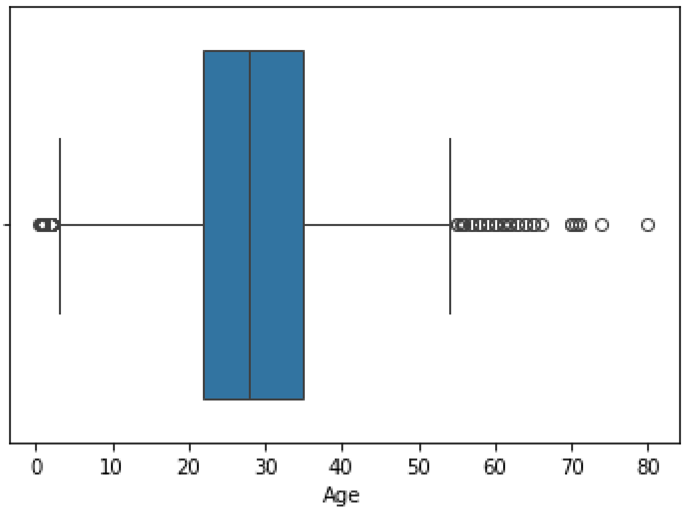
```
extreme=df1.Age.mean()+3*df1.Age.std()
```

In [193…

```
import seaborn as sns
sns.boxplot(x='Age',data=df)
```

Out[193…

```
<Axes: xlabel='Age'>
```



In [194…

```
def impute_nan(df1,variable,median,extreme):
    df1[variable+"_end_distribution"]=df1[variable].fillna(extreme)
    df1[variable].fillna(median,inplace=True)

impute_nan(df1,'Age',df1.Age.median(),extreme)
df1.head(10)
```

Out[194…

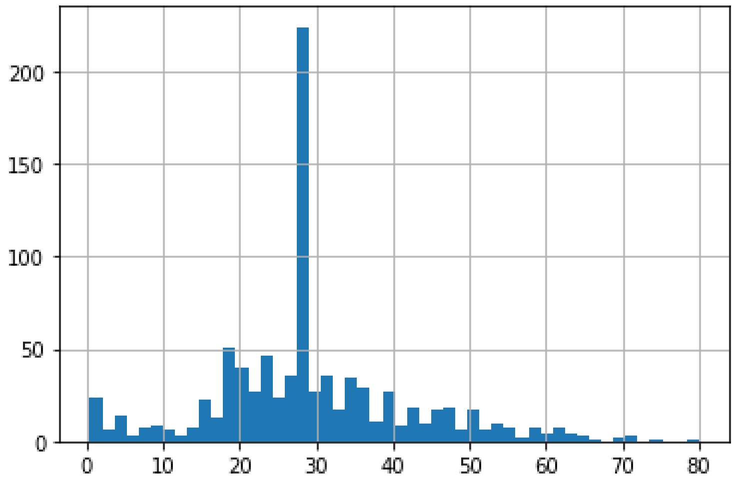
	Survived	Age	Fare	Age_end_distribution
0	0	22.0	7.2500	22.00000
1	1	38.0	71.2833	38.00000
2	1	26.0	7.9250	26.00000
3	1	35.0	53.1000	35.00000
4	0	35.0	8.0500	35.00000
5	0	28.0	8.4583	73.27861
6	0	54.0	51.8625	54.00000
7	0	2.0	21.0750	2.00000
8	1	27.0	11.1333	27.00000
9	1	14.0	30.0708	14.00000

- In Index 5 Nan value with extreme value(73.27)

In [195…

```
df1['Age'].hist(bins=50)
```

Out[195… <Axes: >

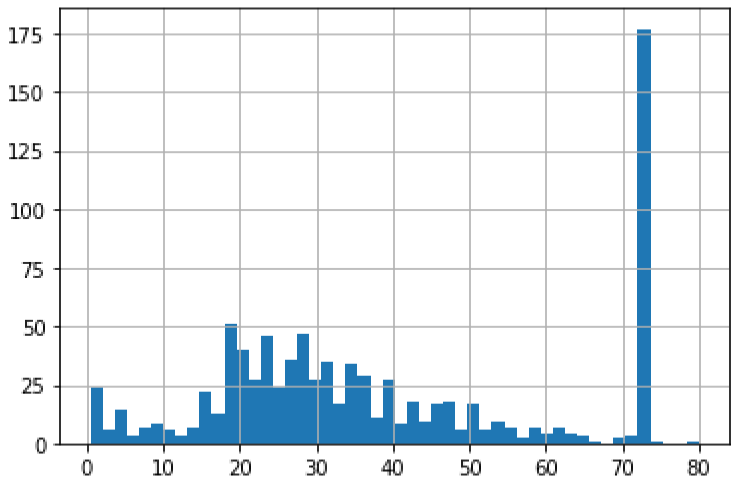


In [196…

```
df1['Age_end_distribution'].hist(bins=50)
```

Out[196…

```
<Axes: >
```

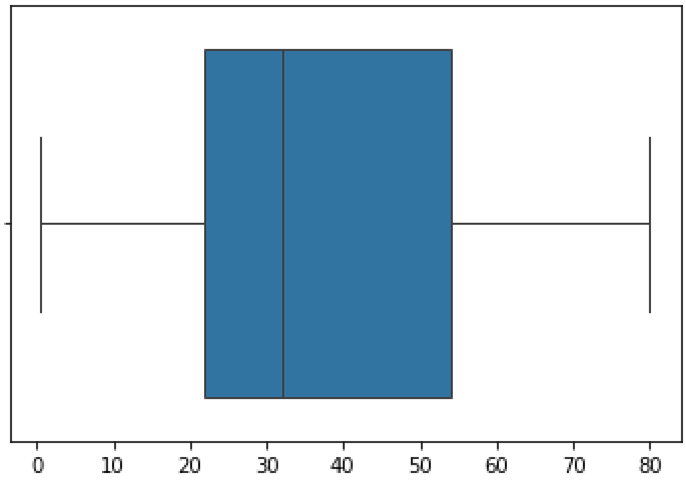


In [197…

```
sns.boxplot(x ='Age_end_distribution',data=df1)
```

Out[197…

```
<Axes: xlabel='Age_end_distribution'>
```



(v) Arbitrary Value Imputation

This technique was derived from kaggle competition It consists of replacing NAN by an arbitrary value

Advantages

- Easy to implement
- Captures the importance of missingess if there is one

Disadvantages

- Distorts the original distribution of the variable
- If missingess is not important, it may mask the predictive power of the original variable by distorting its distribution
- Hard to decide which value to use

In [200…

```
df=pd.read_csv('titanic.csv', usecols=['Age','Fare','Survived'])
df.head()
```

Out[200…

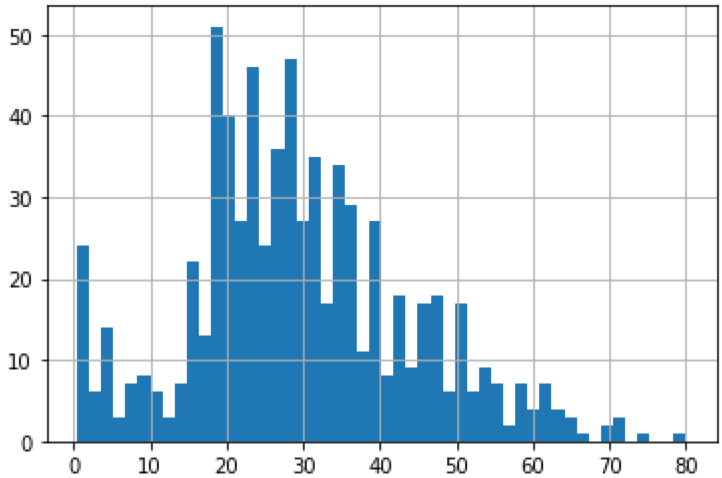
	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [201…

```
### Arbitrary values ---> It should be more frequently present
df['Age'].hist(bins=50)
```

Out[201…

```
<Axes: >
```



In [205…

```
def impute_nan(df,variable):
    df[variable+'_zero']=df[variable].fillna(0)
    df[variable+'_hundred']=df[variable].fillna(100)
impute_nan(df,'Age')
df.head(10)
```

Out[205…

	Survived	Age	Fare	Age_zero	Age_hundred
0	0	22.0	7.2500	22.0	22.0
1	1	38.0	71.2833	38.0	38.0
2	1	26.0	7.9250	26.0	26.0
3	1	35.0	53.1000	35.0	35.0
4	0	35.0	8.0500	35.0	35.0
5	0	NaN	8.4583	0.0	100.0
6	0	54.0	51.8625	54.0	54.0
7	0	2.0	21.0750	2.0	2.0
8	1	27.0	11.1333	27.0	27.0
9	1	14.0	30.0708	14.0	14.0

- Arbitrary value as 100

(vi) Frequent categories imputation

In [206…
```
loan =pd.read_csv(r'C:\Users\Gaurav\Desktop\train.csv', usecols=['BsmtQual','Firepla
loan.columns
```

Out[206…
```
Index(['BsmtQual', 'FireplaceQu', 'GarageType', 'SalePrice'], dtype='object')
```

In [207…
```
loan.shape
```

Out[207…
```
(1460, 4)
```

In [220…
```
# missing values in categorical features
loan.isnull().sum()
```

Out[220…
```
BsmtQual        37
FireplaceQu    690
GarageType      81
SalePrice        0
BsmtQual_Var     0
dtype: int64
```

In [221…
```
# in terms of percentage
loan.isnull().mean().sort_values(ascending=True)
```

Out[221…
```
SalePrice       0.000000
BsmtQual_Var    0.000000
BsmtQual        0.025342
GarageType      0.055479
FireplaceQu     0.472603
dtype: float64
```

1. Compute the frequency with every feature

Advantages

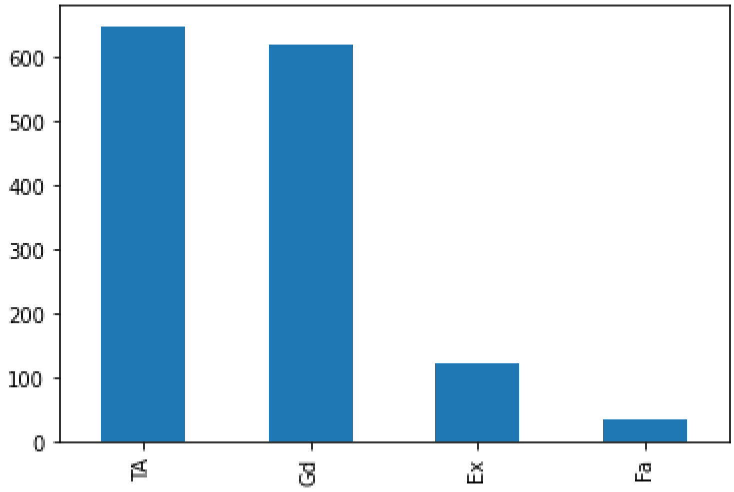
- Easy To implement
- Fater way to implement

Disadvantages

- Since we are using the more frequent labels, it may use them in an over respresented way, if there are many nan's
- It distorts the relation of the most frequent label

In [222…
```
# nan value replace by most freq
loan['BsmtQual'].value_counts().plot.bar()
```

Out[222…
```
<Axes: >
```



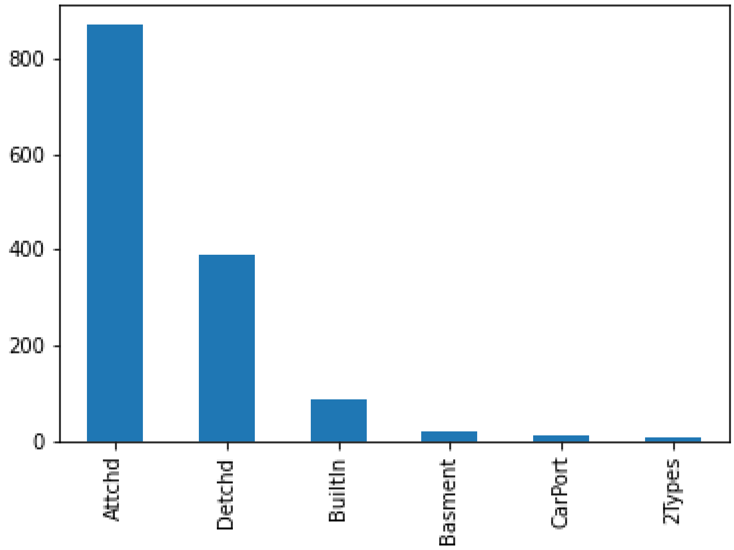
In [ ]:

In [211…

```
loan['GarageType'].value_counts().plot.bar()
```

Out[211…

<Axes: >

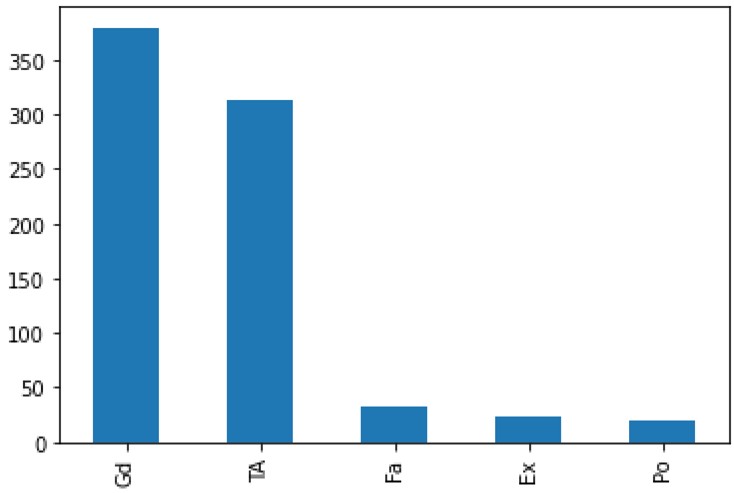


In [212…

```
loan['FireplaceQu'].value_counts().plot.bar()
```

Out[212…

<Axes: >



In [213…

```
# highest category name in specific variable
loan['GarageType'].value_counts().index[0]
#loan['GarageType'].mode()[0]
```

Out[213…

```
'Attchd'
```

- 'Attchd' is more frequently repeated in feature 'GarageType'

In [214…

```
# Replacing Function
def impute_nan(df,variable):
    most_frequent_category=loan[variable].mode()[0]
    loan[variable].fillna(most_frequent_category,inplace=True)
```

In [215…

```
for feature in ['BsmtQual','FireplaceQu','GarageType']:
    impute_nan(loan,feature)
```

In [216…

```
loan.isnull().mean()
```

Out[216…

```
BsmtQual       0.0
FireplaceQu    0.0
GarageType     0.0
SalePrice      0.0
dtype: float64
```

2. Adding a variable to capture NAN

Advantages

- Features have more NaN value

Diadvantages

- Increasing Feature space

In [238…

```
loan =pd.read_csv(r'C:\Users\Gaurav\Desktop\train.csv', usecols=['BsmtQual','Firepla
loan.head()
```

Out[238…

	BsmtQual	FireplaceQu	GarageType	SalePrice
0	Gd	NaN	Attchd	208500
1	Gd	TA	Attchd	181500
2	Gd	TA	Attchd	223500
3	TA	Gd	Detchd	140000
4	Gd	TA	Attchd	250000

In [239…

```
loan['BsmtQual_Var']=np.where(loan['BsmtQual'].isnull(),1,0)
```

In [240…

```
loan['BsmtQual'].mode()[0]
```

Out[240…

```
'TA'
```

In [241…

```
frequent=loan['BsmtQual'].mode()[0]
loan['BsmtQual'].fillna(frequent,inplace=True)
loan.head()
```

Out[241…

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQual_Var
0	Gd	NaN	Attchd	208500	0
1	Gd	TA	Attchd	181500	0
2	Gd	TA	Attchd	223500	0
3	TA	Gd	Detchd	140000	0
4	Gd	TA	Attchd	250000	0

In [242…

```
loan['FireplaceQu_Var']=np.where(loan['FireplaceQu'].isnull(),1,0)
frequent=loan['FireplaceQu'].mode()[0]
loan['FireplaceQu'].fillna(frequent,inplace=True)
```

In [243…

```
loan.head()
```

Out[243…

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQual_Var	FireplaceQu_Var
0	Gd	Gd	Attchd	208500	0	1
1	Gd	TA	Attchd	181500	0	0
2	Gd	TA	Attchd	223500	0	0
3	TA	Gd	Detchd	140000	0	0
4	Gd	TA	Attchd	250000	0	0

- Suppose if you have more frequent categories, we just replace NAN with a new category

In [251…

```
loan =pd.read_csv(r'C:\Users\Gaurav\Desktop\train.csv', usecols=['BsmtQual','Firepla
loan.head()
```

Out[251…

	BsmtQual	FireplaceQu	GarageType	SalePrice
0	Gd	NaN	Attchd	208500
1	Gd	TA	Attchd	181500
2	Gd	TA	Attchd	223500
3	TA	Gd	Detchd	140000
4	Gd	TA	Attchd	250000

In [252…

```
def impute_nan(df,variable):
    loan[variable+"newvar"]=np.where(loan[variable].isnull(),"Missing",loan[variable

for feature in ['BsmtQual','FireplaceQu','GarageType']:
    impute_nan(loan,feature)
```

- Replacing NaN value by Missing

In [253…

```
loan.head()
```

Out[253…

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQualnewvar	FireplaceQunewvar	GarageTypen
0	Gd	NaN	Attchd	208500	Gd	Missing	
1	Gd	TA	Attchd	181500	Gd	TA	
2	Gd	TA	Attchd	223500	Gd	TA	
3	TA	Gd	Detchd	140000	TA	Gd	[
4	Gd	TA	Attchd	250000	Gd	TA	

In [254…

```
loan=loan.drop(['BsmtQual','FireplaceQu','GarageType'],axis=1)
# dropping unnecessary features
```

In [255…

```
loan.head()
```

Out[255…

	SalePrice	BsmtQualnewvar	FireplaceQunewvar	GarageTypenewvar
0	208500	Gd	Missing	Attchd
1	181500	Gd	TA	Attchd
2	223500	Gd	TA	Attchd
3	140000	TA	Gd	Detchd
4	250000	Gd	TA	Attchd

In [ ]:

In [ ]:

Encoding Categorical Variables:

Categorical variables need to be converted into numerical representations for machine learning algorithms to process them. One-hot encoding, label encoding, and target encoding are common techniques for this purpose.

1. One Hot Encoding

Disadvantage : creates more features

In [130…

```
import pandas as pd
import numpy as np
```

In [113…

```
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv',usecols=['Sex'])
df.head()
```

Out[113…

	Sex
0	male
1	female
2	female
3	female
4	male

Deleting one column with help of dummy variable trap

In [114…

```
pd.get_dummies(df).head()
```

Out[114…

	Sex_female	Sex_male
0	0	1
1	1	0
2	1	0
3	1	0
4	0	1

In [115…

```
pd.get_dummies(df,drop_first=True).head()
# male == 1 /  female == 0
```

Out[115…

	Sex_male
0	1
1	0
2	0

	Sex_male
3	0
4	1

In [131…

```
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv',usecols=['Embarked'])
```

In [132…

```
df['Embarked'].unique()
```

Out[132…

```
array(['S', 'C', 'Q', nan], dtype=object)
```

In [133…

```
# droping nan value
df.dropna(inplace=True)
```

In [134…

```
pd.get_dummies(df,drop_first=True).head()
# two features will represent third features
```

Out[134…

	Embarked_Q	Embarked_S
0	0	1
1	0	0
2	0	1
3	0	1
4	0	1

2. One Hot Encoding with many categories in a feature

In [158…

```
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\mercedes.csv',usecols=["X0","X1","X2","X3
```

In [159…

```
df.head()
```

Out[159…

	X0	X1	X2	X3	X4	X5	X6
0	k	v	at	a	d	u	j
1	k	t	av	e	d	y	l
2	az	w	n	c	d	x	j
3	az	t	n	f	d	x	l
4	az	v	n	f	d	h	d

In [160…

```
# let's have a look at how many labels each variable had
for i in df.columns:
    print(i, ':', len(df[i].unique()), 'labels')
```

```
X0 : 47 labels
X1 : 27 labels
X2 : 44 labels
X3 : 7 labels
```

```
X4 : 4 labels
X5 : 29 labels
X6 : 12 labels
```

In [161…

```
df.X1.value_counts()
```

Out[161…

```
aa    833
s     598
b     592
l     590
v     408
r     251
i     203
a     143
c     121
o      82
w      52
z      46
u      37
e      33
m      32
t      31
h      29
y      23
f      23
j      22
n      19
k      17
p       9
g       6
d       3
q       3
ab      3
Name: X1, dtype: int64
```

In [162…

```
# considering only top 10 & dropping rest ---> (KDD cup competition)
df.X1.value_counts().sort_values(ascending=False).head(10)
```

Out[162…

```
aa    833
s     598
b     592
l     590
v     408
r     251
i     203
a     143
c     121
o      82
Name: X1, dtype: int64
```

In [163…

```
# in terms of list
lst_10=df.X1.value_counts().sort_values(ascending=False).head(10).index
lst_10=list(lst_10)
lst_10
```

Out[163…

```
['aa', 's', 'b', 'l', 'v', 'r', 'i', 'a', 'c', 'o']
```

In [164…

```
for categories in lst_10:
    df[categories]=np.where(df['X1']==categories,1,0)
lst_10.append('X1')
df[lst_10]
```

Out[164...

	aa	s	b	l	v	r	i	a	c	o	X1
0	0	0	0	0	1	0	0	0	0	0	v
1	0	0	0	0	0	0	0	0	0	0	t
2	0	0	0	0	0	0	0	0	0	0	w
3	0	0	0	0	0	0	0	0	0	0	t
4	0	0	0	0	1	0	0	0	0	0	v
...	...	...	...	...	...	...	...	...	...	...	...
4204	0	1	0	0	0	0	0	0	0	0	s
4205	0	0	0	0	0	0	0	0	0	1	o
4206	0	0	0	0	1	0	0	0	0	0	v
4207	0	0	0	0	0	1	0	0	0	0	r
4208	0	0	0	0	0	1	0	0	0	0	r

4209 rows × 11 columns

In [165...

```
# get whole set of dummy variables, for all the categorical variables

def one_hot_encoding_top_x(df, variable, top_x_labels):
    # function to create the dummy variables for the most frequent labels
    # we can vary the number of most frequent labels that we encode

    for label in top_x_labels:
        df[variable+'_'+label] = np.where(df[variable]==label, 1, 0)
```

In [166...

```
# read the data again
df = pd.read_csv(r'C:\Users\Gaurav\Downloads\mercedes.csv', usecols=['X1', 'X2'])

# encode X2 into the 10 most frequent categories
one_hot_encoding_top_x(df, 'X2', lst_10)
df.head()
```

Out[166...

	X1	X2	X2_aa	X2_s	X2_b	X2_l	X2_v	X2_r	X2_i	X2_a	X2_c	X2_o	X2_X1
0	v	at	0	0	0	0	0	0	0	0	0	0	0
1	t	av	0	0	0	0	0	0	0	0	0	0	0
2	w	n	0	0	0	0	0	0	0	0	0	0	0
3	t	n	0	0	0	0	0	0	0	0	0	0	0
4	v	n	0	0	0	0	0	0	0	0	0	0	0

3. Ordinal Number Encoding

Ordinal data is a categorical, statistical data type where the variables have natural, ordered categories and the distances between the categories is not known.

For example:

- Student's grade in an exam (A, B, C or Fail).
- Educational level, with the categories: Elementary school, High school, College graduate, PhD ranked from 1 to 4.

When the categorical variables are ordinal, the most straightforward best approach is to replace the labels by some ordinal number based on the ranks.

In [167…

```
import datetime
```

In [168…

```
today_date=datetime.datetime.today()
```

In [169…

```
today_date
```

Out[169…

```
datetime.datetime(2024, 2, 18, 18, 6, 47, 841279)
```

In [170…

```
# differnce btw dates
today_date-datetime.timedelta(1)
```

Out[170…

```
datetime.datetime(2024, 2, 17, 18, 6, 47, 841279)
```

In [171…

```
#### last 15 days dates
days=[today_date-datetime.timedelta(x) for x in range(0,15)]
days
```

Out[171…

```
[datetime.datetime(2024, 2, 18, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 17, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 16, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 15, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 14, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 13, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 12, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 11, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 10, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 9, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 8, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 7, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 6, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 5, 18, 6, 47, 841279),
 datetime.datetime(2024, 2, 4, 18, 6, 47, 841279)]
```

In [172…

```
import pandas as pd
data=pd.DataFrame(days)
data.columns=["Day"]
```

In [173…

```
data.head()
```

Out[173…

	Day
0	2024-02-18 18:06:47.841279
1	2024-02-17 18:06:47.841279
2	2024-02-16 18:06:47.841279
3	2024-02-15 18:06:47.841279

	Day
4	2024-02-14 18:06:47.841279

In [174…

```
data['weekday']=data['Day'].dt.day_name()
data.head()
```

Out[174…

	Day	weekday
0	2024-02-18 18:06:47.841279	Sunday
1	2024-02-17 18:06:47.841279	Saturday
2	2024-02-16 18:06:47.841279	Friday
3	2024-02-15 18:06:47.841279	Thursday
4	2024-02-14 18:06:47.841279	Wednesday

In [175…

```
dictionary={'Monday':1,'Tuesday':2,'Wednesday':3,'Thursday':4,'Friday':5,'Saturday':
```

In [24]:

```
dictionary
```

Out[24]:

```
{'Monday': 1,
 'Tuesday': 2,
 'Wednesday': 3,
 'Thursday': 4,
 'Friday': 5,
 'Saturday': 6,
 'Sunday': 7}
```

In [176…

```
data['weekday_ordinal']=data['weekday'].map(dictionary)
```

In [26]:

```
data
```

Out[26]:

	Day	weekday	weekday_ordinal
0	2024-02-15 20:48:31.566936	Thursday	4
1	2024-02-14 20:48:31.566936	Wednesday	3
2	2024-02-13 20:48:31.566936	Tuesday	2
3	2024-02-12 20:48:31.566936	Monday	1
4	2024-02-11 20:48:31.566936	Sunday	7
5	2024-02-10 20:48:31.566936	Saturday	6
6	2024-02-09 20:48:31.566936	Friday	5
7	2024-02-08 20:48:31.566936	Thursday	4
8	2024-02-07 20:48:31.566936	Wednesday	3
9	2024-02-06 20:48:31.566936	Tuesday	2
10	2024-02-05 20:48:31.566936	Monday	1
11	2024-02-04 20:48:31.566936	Sunday	7

	Day	weekday	weekday_ordinal
12	2024-02-03 20:48:31.566936	Saturday	6
13	2024-02-02 20:48:31.566936	Friday	5
14	2024-02-01 20:48:31.566936	Thursday	4

4. Count Or Frequency Encoding

Another way to refer to variables that have a multitude of categories, is to call them variables with high cardinality.

If we have categorical variables containing many multiple labels or high cardinality,then by using one hot encoding, we will expand the feature space dramatically.

One approach that is heavily used in Kaggle competitions, is to replace each label of the categorical variable by the count, this is the amount of times each label appears in the dataset. Or the frequency, this is the percentage of observations within that category. The 2 are equivalent.

Advantages

1. Easy To Use
2. Not increasing feature space

Disadvantages

1. If some of the labels have the same count, then they will be replaced with the same count and they will loose some valuable information.
2. Adds somewhat arbitrary numbers, and therefore weights to the different labels, that may not be related to their predictive power

In [189…

```
train_set = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/ad
train_set.head()
```

Out[189…

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40

In [190…

```
# cateorgy columns
columns=[1,3,5,6,7,8,9,13]
```

In [191…

```
train_set=train_set[columns]
```

In [192…

```
# assigning column name
train_set.columns=['Employment','Degree','Status','Designation','family_job','Race',
```

In [193…

```
train_set.head()
```

Out[193…

	Employment	Degree	Status	Designation	family_job	Race	Sex	Country
0	State-gov	Bachelors	Never-married	Adm-clerical	Not-in-family	White	Male	United-States
1	Self-emp-not-inc	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	United-States
2	Private	HS-grad	Divorced	Handlers-cleaners	Not-in-family	White	Male	United-States
3	Private	11th	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	United-States
4	Private	Bachelors	Married-civ-spouse	Prof-specialty	Wife	Black	Female	Cuba

In [194…

```
for feature in train_set.columns[:]:
    print(feature,":",len(train_set[feature].unique()),'labels')
```

```
Employment : 9 labels
Degree : 16 labels
Status : 7 labels
Designation : 15 labels
family_job : 6 labels
Race : 5 labels
Sex : 2 labels
Country : 42 labels
```

country-name will replace by Frequency

In [195…

```
train_set['Country'].value_counts().to_dict()
```

Out[195…

```
{' United-States': 29170,
 ' Mexico': 643,
 ' ?': 583,
 ' Philippines': 198,
 ' Germany': 137,
 ' Canada': 121,
 ' Puerto-Rico': 114,
 ' El-Salvador': 106,
 ' India': 100,
 ' Cuba': 95,
 ' England': 90,
 ' Jamaica': 81,
 ' South': 80,
 ' China': 75,
 ' Italy': 73,
 ' Dominican-Republic': 70,
```

```
 ' Vietnam': 67,
 ' Guatemala': 64,
 ' Japan': 62,
 ' Poland': 60,
 ' Columbia': 59,
 ' Taiwan': 51,
 ' Haiti': 44,
 ' Iran': 43,
 ' Portugal': 37,
 ' Nicaragua': 34,
 ' Peru': 31,
 ' France': 29,
 ' Greece': 29,
 ' Ecuador': 28,
 ' Ireland': 24,
 ' Hong': 20,
 ' Cambodia': 19,
 ' Trinadad&Tobago': 19,
 ' Laos': 18,
 ' Thailand': 18,
 ' Yugoslavia': 16,
 ' Outlying-US(Guam-USVI-etc)': 14,
 ' Honduras': 13,
 ' Hungary': 13,
 ' Scotland': 12,
 ' Holand-Netherlands': 1}
```

In [196…

```
country_map=train_set['Country'].value_counts().to_dict()
```

In [197…

```
train_set['Country']=train_set['Country'].map(country_map)
train_set.head(20)
```

Out[197…

	Employment	Degree	Status	Designation	family_job	Race	Sex	Country
0	State-gov	Bachelors	Never-married	Adm-clerical	Not-in-family	White	Male	29170
1	Self-emp-not-inc	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	29170
2	Private	HS-grad	Divorced	Handlers-cleaners	Not-in-family	White	Male	29170
3	Private	11th	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	29170
4	Private	Bachelors	Married-civ-spouse	Prof-specialty	Wife	Black	Female	95
5	Private	Masters	Married-civ-spouse	Exec-managerial	Wife	White	Female	29170
6	Private	9th	Married-spouse-absent	Other-service	Not-in-family	Black	Female	81
7	Self-emp-not-inc	HS-grad	Married-civ-spouse	Exec-managerial	Husband	White	Male	29170
8	Private	Masters	Never-married	Prof-specialty	Not-in-family	White	Female	29170
9	Private	Bachelors	Married-civ-spouse	Exec-managerial	Husband	White	Male	29170
10	Private	Some-	Married-	Exec-	Husband	Black	Male	29170

	Employment	Degree	Status	Designation	family_job	Race	Sex	Country
		college	civ-spouse	managerial				
11	State-gov	Bachelors	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Male	100
12	Private	Bachelors	Never-married	Adm-clerical	Own-child	White	Female	29170
13	Private	Assoc-acdm	Never-married	Sales	Not-in-family	Black	Male	29170
14	Private	Assoc-voc	Married-civ-spouse	Craft-repair	Husband	Asian-Pac-Islander	Male	583
15	Private	7th-8th	Married-civ-spouse	Transport-moving	Husband	Amer-Indian-Eskimo	Male	643
16	Self-emp-not-inc	HS-grad	Never-married	Farming-fishing	Own-child	White	Male	29170
17	Private	HS-grad	Never-married	Machine-op-inspct	Unmarried	White	Male	29170
18	Private	11th	Married-civ-spouse	Sales	Husband	White	Male	29170
19	Self-emp-not-inc	Masters	Divorced	Exec-managerial	Unmarried	White	Female	29170

5. Target Guided Ordinal Encoding

1. Ordering the labels according to the target
2. Replace the labels by the joint probability of being 1 or 0

In [198…

```
import pandas as pd
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv', usecols=['Cabin','Survived'
df.head()
```

Out[198…

	Survived	Cabin
0	0	NaN
1	1	C85
2	1	NaN
3	1	C123
4	0	NaN

In [199…

```
df['Cabin'].fillna('Missing',inplace=True)
```

In [200…

```
df['Cabin']=df['Cabin'].astype(str).str[0]
# considering first letter only
```

In [201…

```
df.head()
```

Out[201…

	Survived	Cabin
0	0	M
1	1	C
2	1	M
3	1	C
4	0	M

In [202…

```
df.Cabin.unique()
```

Out[202…

```
array(['M', 'C', 'E', 'G', 'D', 'A', 'B', 'F', 'T'], dtype=object)
```

In [203…

```
df.groupby(['Cabin'])['Survived'].mean()
# in each column how many people survive
```

Out[203…

```
Cabin
A    0.466667
B    0.744681
C    0.593220
D    0.757576
E    0.750000
F    0.615385
G    0.500000
M    0.299854
T    0.000000
Name: Survived, dtype: float64
```

In [204…

```
df.groupby(['Cabin'])['Survived'].mean().sort_values()
```

Out[204…

```
Cabin
T    0.000000
M    0.299854
A    0.466667
G    0.500000
C    0.593220
F    0.615385
B    0.744681
E    0.750000
D    0.757576
Name: Survived, dtype: float64
```

In [205…

```
ordinal_labels=df.groupby(['Cabin'])['Survived'].mean().sort_values().index
ordinal_labels
```

Out[205…

```
Index(['T', 'M', 'A', 'G', 'C', 'F', 'B', 'E', 'D'], dtype='object', name='Cabin')
```

In [206…

```
enumerate(ordinal_labels,0)
# assign 0,1,2,3 as per rank
```

Out[206…

```
<enumerate at 0x1a19e6b4480>
```

In [207…

```
ordinal_labels2={k:i for i,k in enumerate(ordinal_labels,0)}
ordinal_labels2
```

Out[207…
```
{'T': 0, 'M': 1, 'A': 2, 'G': 3, 'C': 4, 'F': 5, 'B': 6, 'E': 7, 'D': 8}
```

In [208…
```
df['Cabin_ordinal_labels']=df['Cabin'].map(ordinal_labels2)
df.head()
```

Out[208…

	Survived	Cabin	Cabin_ordinal_labels
0	0	M	1
1	1	C	4
2	1	M	1
3	1	C	4
4	0	M	1

5.1. Mean Encoding

replace by mean value

In [83]:
```
mean_ordinal=df.groupby(['Cabin'])['Survived'].mean().to_dict()
```

In [84]:
```
mean_ordinal
```

Out[84]:
```
{'A': 0.4666666666666667,
 'B': 0.7446808510638298,
 'C': 0.5932203389830508,
 'D': 0.7575757575757576,
 'E': 0.75,
 'F': 0.6153846153846154,
 'G': 0.5,
 'M': 0.29985443959243085,
 'T': 0.0}
```

In [86]:
```
df['mean_ordinal_encode']=df['Cabin'].map(mean_ordinal)
df.head()
```

Out[86]:

	Survived	Cabin	Cabin_ordinal_labels	mean_ordinal_encode
0	0	M	1	0.299854
1	1	C	4	0.593220
2	1	M	1	0.299854
3	1	C	4	0.593220
4	0	M	1	0.299854

It leads to overfitting

In [ ]:
```

```

5.2. Probability Ratio Encoding

Steps:

1. Probability of Survived based on Cabin--- Categorical Feature
2. Probability of Not Survived---1-pr(Survived)
3. pr(Survived)/pr(Not Survived)
4. Dictonary to map cabin with probability
5. replace with the categorical feature

In [89]:
```
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv', usecols=['Cabin','Survived'
df.head()
```

Out[89]:

	Survived	Cabin
0	0	NaN
1	1	C85
2	1	NaN
3	1	C123
4	0	NaN

In [90]:
```
### Replacing
df['Cabin'].fillna('Missing',inplace=True)
df.head()
```

Out[90]:

	Survived	Cabin
0	0	Missing
1	1	C85
2	1	Missing
3	1	C123
4	0	Missing

In [91]:
```
df['Cabin'].unique()
```

Out[91]:
```
array(['Missing', 'C85', 'C123', 'E46', 'G6', 'C103', 'D56', 'A6',
       'C23 C25 C27', 'B78', 'D33', 'B30', 'C52', 'B28', 'C83', 'F33',
       'F G73', 'E31', 'A5', 'D10 D12', 'D26', 'C110', 'B58 B60', 'E101',
       'F E69', 'D47', 'B86', 'F2', 'C2', 'E33', 'B19', 'A7', 'C49', 'F4',
       'A32', 'B4', 'B80', 'A31', 'D36', 'D15', 'C93', 'C78', 'D35',
       'C87', 'B77', 'E67', 'B94', 'C125', 'C99', 'C118', 'D7', 'A19',
       'B49', 'D', 'C22 C26', 'C106', 'C65', 'E36', 'C54',
       'B57 B59 B63 B66', 'C7', 'E34', 'C32', 'B18', 'C124', 'C91', 'E40',
       'T', 'C128', 'D37', 'B35', 'E50', 'C82', 'B96 B98', 'E10', 'E44',
       'A34', 'C104', 'C111', 'C92', 'E38', 'D21', 'E12', 'E63', 'A14',
       'B37', 'C30', 'D20', 'B79', 'E25', 'D46', 'B73', 'C95', 'B38',
       'B39', 'B22', 'C86', 'C70', 'A16', 'C101', 'C68', 'A10', 'E68',
       'B41', 'A20', 'D19', 'D50', 'D9', 'A23', 'B50', 'A26', 'D48',
       'E58', 'C126', 'B71', 'B51 B53 B55', 'D49', 'B5', 'B20', 'F G63',
       'C62 C64', 'E24', 'C90', 'C45', 'E8', 'B101', 'D45', 'C46', 'D30',
       'E121', 'D11', 'E77', 'F38', 'B3', 'D6', 'B82 B84', 'D17', 'A36',
       'B102', 'B69', 'E49', 'C47', 'D28', 'E17', 'A24', 'C50', 'B42',
       'C148'], dtype=object)
```

In [92]:
```
# first letter only
df['Cabin']=df['Cabin'].astype(str).str[0]
```

```
df.head()
```

Out[92]:

	Survived	Cabin
0	0	M
1	1	C
2	1	M
3	1	C
4	0	M

In [93]:

```
df.Cabin.unique()
```

Out[93]:

```
array(['M', 'C', 'E', 'G', 'D', 'A', 'B', 'F', 'T'], dtype=object)
```

In [94]:

```
prob_df=df.groupby(['Cabin'])['Survived'].mean()
prob_df=pd.DataFrame(prob_df)
prob_df
```

Out[94]:

Cabin	Survived
A	0.466667
B	0.744681
C	0.593220
D	0.757576
E	0.750000
F	0.615385
G	0.500000
M	0.299854
T	0.000000

In [95]:

```
prob_df['Died']=1-prob_df['Survived']
prob_df.head()
```

Out[95]:

Cabin	Survived	Died
A	0.466667	0.533333
B	0.744681	0.255319
C	0.593220	0.406780
D	0.757576	0.242424
E	0.750000	0.250000

In [96]:
```
prob_df['Probability_ratio']=prob_df['Survived']/prob_df['Died']
prob_df.head()
```

Out[96]:

Cabin	Survived	Died	Probability_ratio
A	0.466667	0.533333	0.875000
B	0.744681	0.255319	2.916667
C	0.593220	0.406780	1.458333
D	0.757576	0.242424	3.125000
E	0.750000	0.250000	3.000000

In [97]:
```
probability_encoded=prob_df['Probability_ratio'].to_dict()
df['Cabin_encoded']=df['Cabin'].map(probability_encoded)
df.head()
```

Out[97]:

	Survived	Cabin	Cabin_encoded
0	0	M	0.428274
1	1	C	1.458333
2	1	M	0.428274
3	1	C	1.458333
4	0	M	0.428274

In [98]:
```
df.head(20)
```

Out[98]:

	Survived	Cabin	Cabin_encoded
0	0	M	0.428274
1	1	C	1.458333
2	1	M	0.428274
3	1	C	1.458333
4	0	M	0.428274
5	0	M	0.428274
6	0	E	3.000000
7	0	M	0.428274
8	1	M	0.428274
9	1	M	0.428274
10	1	G	1.000000
11	1	C	1.458333
12	0	M	0.428274
13	0	M	0.428274

	Survived	Cabin	Cabin_encoded
14	0	M	0.428274
15	1	M	0.428274
16	0	M	0.428274
17	1	M	0.428274
18	0	M	0.428274
19	1	M	0.428274

In [ ]:

Feature Scaling

Normalization:

Normalization scales features to have a magnitude of 1. It is particularly useful for algorithms that rely on distance metrics, such as K-Nearest Neighbors.

Standardization:

Standardization transforms features to have a mean of 0 and a standard deviation of 1. It helps algorithms converge faster, especially gradient-based optimization methods.

1. Standardization

We try to bring all the variables or features to a similar scale.

- It comes into picture when features of input dataset hve large diff. btw their ranges or simply when they are measured in diferent measurement units(eg. pounds, meters, miles)
- Standardization transforms features to have a mean of 0 and a standard deviation of 1. It helps algorithms converge faster, especially gradient-based optimization methods.
- z=(x-x_mean)/std

In [54]:
```
import pandas as pd
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv', usecols=['Pclass','Age','Fa
df.head()
```

Out[54]:

	Survived	Pclass	Age	Fare
0	0	3	22.0	7.2500
1	1	1	38.0	71.2833
2	1	3	26.0	7.9250
3	1	1	35.0	53.1000
4	0	3	35.0	8.0500

In [55]:
```
# removing nan value
```

In [56]:
```
df['Age'].fillna(df.Age.median(),inplace=True)
```

In [57]:
```
df.isnull().sum()
```

Out[57]:
```
Survived    0
Pclass      0
Age         0
```

```
Fare        0
dtype: int64
```

In [58]:
```
#### standarisation: We use the Standardscaler from sklearn library
from sklearn.preprocessing import StandardScaler
```

In [59]:
```
scaler=StandardScaler()
df_scaled=scaler.fit_transform(df)
```

In [60]:
```
pd.DataFrame(df_scaled)
# transforamtion happens through columns wise
```

Out[60]:

	0	1	2	3
0	-0.789272	0.827377	-0.565736	-0.502445
1	1.266990	-1.566107	0.663861	0.786845
2	1.266990	0.827377	-0.258337	-0.488854
3	1.266990	-1.566107	0.433312	0.420730
4	-0.789272	0.827377	0.433312	-0.486337
...	...	...	...	...
886	-0.789272	-0.369365	-0.181487	-0.386671
887	1.266990	-1.566107	-0.796286	-0.044381
888	-0.789272	0.827377	-0.104637	-0.176263
889	1.266990	-1.566107	-0.258337	-0.044381
890	-0.789272	0.827377	0.202762	-0.492378

891 rows × 4 columns

In [61]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

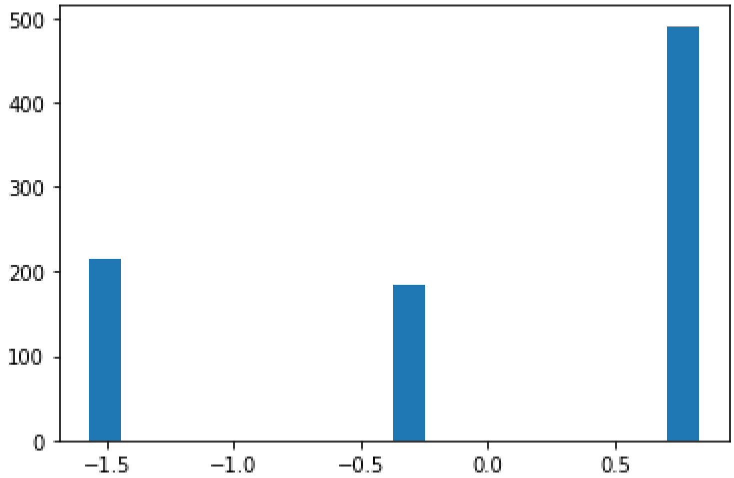
In [62]:
```
df_scaled
```

Out[62]:
```
array([[-0.78927234,  0.82737724, -0.56573646, -0.50244517],
       [ 1.2669898 , -1.56610693,  0.66386103,  0.78684529],
       [ 1.2669898 ,  0.82737724, -0.25833709, -0.48885426],
       ...,
       [-0.78927234,  0.82737724, -0.1046374 , -0.17626324],
       [ 1.2669898 , -1.56610693, -0.25833709, -0.04438104],
       [-0.78927234,  0.82737724,  0.20276197, -0.49237783]])
```

In [63]:
```
#Pclass
plt.hist(df_scaled[:,1],bins=20)
```

Out[63]:
```
(array([216.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 184.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 491.]),
 array([-1.56610693, -1.44643272, -1.32675851, -1.2070843 , -1.08741009,
        -0.96773588, -0.84806167, -0.72838747, -0.60871326, -0.48903905,
        -0.36936484, -0.24969063, -0.13001642, -0.01034222,  0.10933199,
         0.2290062 ,  0.34868041,  0.46835462,  0.58802883,  0.70770304,
```

```
        0.82737724]),
 <BarContainer object of 20 artists>)
```

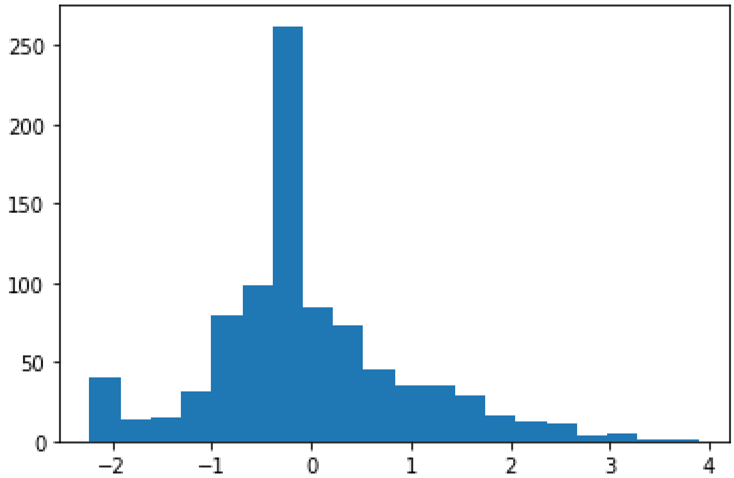


In [64]:

```
# age
plt.hist(df_scaled[:,2],bins=20)
```

Out[64]:

```
(array([ 40.,  14.,  15.,  31.,  79.,  98., 262.,  84.,  73.,  45.,  35.,
         35.,  29.,  16.,  13.,  11.,   4.,   5.,   1.,   1.]),
 array([-2.22415608, -1.91837055, -1.61258503, -1.3067995 , -1.00101397,
       -0.69522845, -0.38944292, -0.08365739,  0.22212813,  0.52791366,
        0.83369919,  1.13948471,  1.44527024,  1.75105577,  2.05684129,
        2.36262682,  2.66841235,  2.97419787,  3.2799834 ,  3.58576892,
        3.89155445]),
 <BarContainer object of 20 artists>)
```

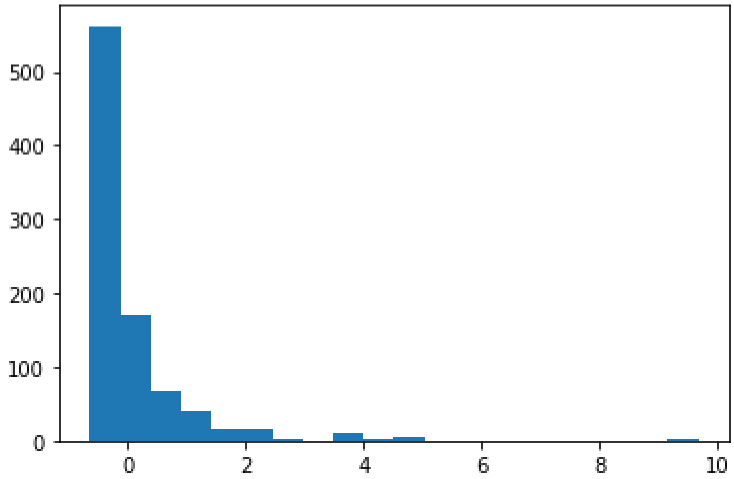


In [65]:

```
# fare
plt.hist(df_scaled[:,3],bins=20)
```

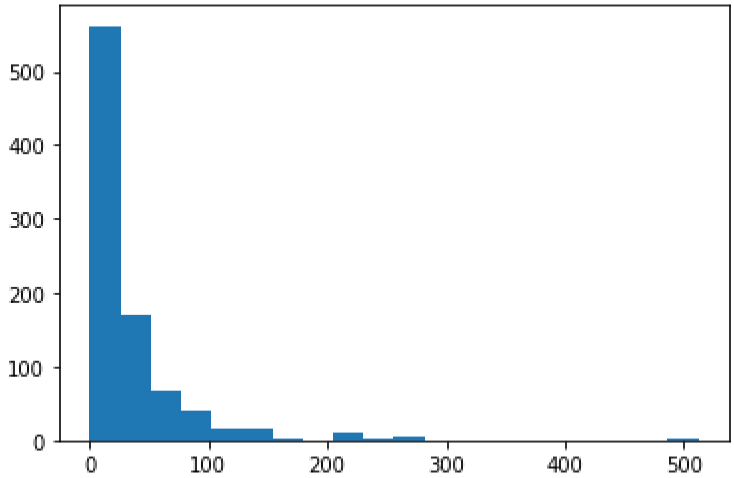
Out[65]:

```
(array([562., 170.,  67.,  39.,  15.,  16.,   2.,   0.,   9.,   2.,   6.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.]),
 array([-0.64842165, -0.13264224,  0.38313716,  0.89891657,  1.41469598,
        1.93047539,  2.4462548 ,  2.96203421,  3.47781362,  3.99359303,
        4.50937244,  5.02515184,  5.54093125,  6.05671066,  6.57249007,
        7.08826948,  7.60404889,  8.1198283 ,  8.63560771,  9.15138712,
        9.66716653]),
 <BarContainer object of 20 artists>)
```



In [66]:
```
plt.hist(df['Fare'],bins=20)
```

Out[66]:
```
(array([562., 170.,  67.,  39.,  15.,  16.,   2.,   0.,   9.,   2.,   6.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.]),
 array([  0.     ,  25.61646,  51.23292,  76.84938, 102.46584, 128.0823 ,
        153.69876, 179.31522, 204.93168, 230.54814, 256.1646 , 281.78106,
        307.39752, 333.01398, 358.63044, 384.2469 , 409.86336, 435.47982,
        461.09628, 486.71274, 512.3292 ]),
 <BarContainer object of 20 artists>)
```



2. Normalization

Normalization scales features to have a magnitude of 1. It is particularly useful for algorithms that rely on distance metrics, such as K-Nearest Neighbors.

Min Max Scaling (### CNN)---Deep Learning Techniques

- Min Max Scaling scales the values between 0 to 1.
- X_scaled = (X - X.min / (X.max - X.min)

In [25]:
```
from sklearn.preprocessing import MinMaxScaler
min_max=MinMaxScaler()
df_minmax=pd.DataFrame(min_max.fit_transform(df),columns=df.columns)
df_minmax.head()
```

Out[25]:

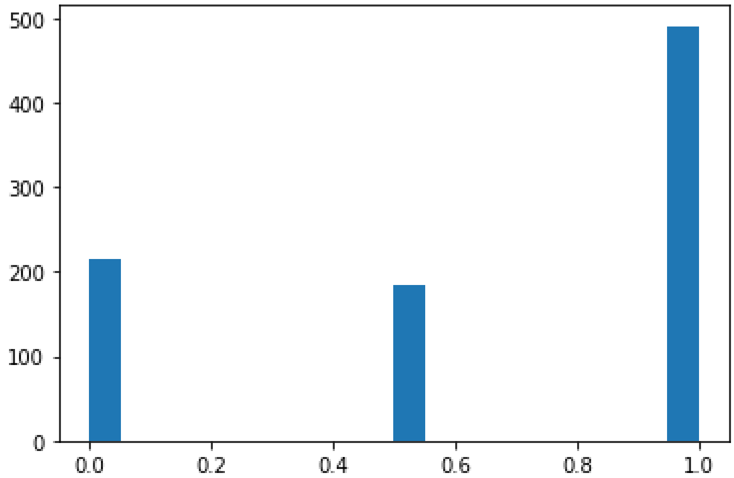
	Survived	Pclass	Age	Fare
0	0.0	1.0	0.271174	0.014151
1	1.0	0.0	0.472229	0.139136
2	1.0	1.0	0.321438	0.015469
3	1.0	0.0	0.434531	0.103644
4	0.0	1.0	0.434531	0.015713

In [26]:

```
plt.hist(df_minmax['Pclass'],bins=20)
```

Out[26]:

```
(array([216.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 184.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 491.]),
 array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 ,
        0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95, 1.  ]),
 <BarContainer object of 20 artists>)
```

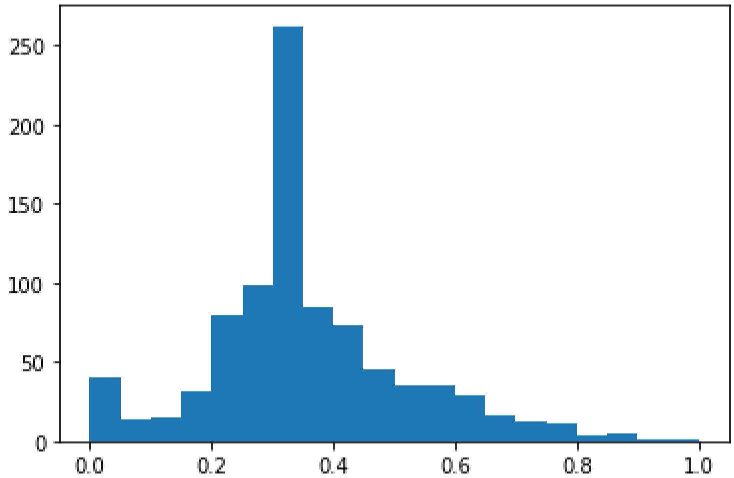


In [27]:

```
plt.hist(df_minmax['Age'],bins=20)
```

Out[27]:

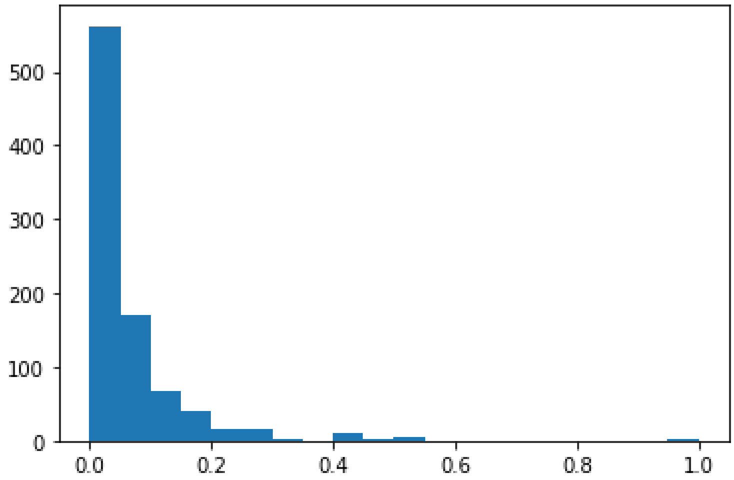
```
(array([ 40.,  14.,  15.,  31.,  79.,  98., 262.,  84.,  73.,  45.,  35.,
         35.,  29.,  16.,  13.,  11.,   4.,   5.,   1.,   1.]),
 array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 ,
        0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95, 1.  ]),
 <BarContainer object of 20 artists>)
```



In [28]:

```
plt.hist(df_minmax['Fare'],bins=20)
```

Out[28]:
```
(array([562., 170.,  67.,  39.,  15.,  16.,   2.,   0.,   9.,   2.,   6.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.]),
 array([0.  , 0.05, 0.1 , 0.15, 0.2 , 0.25, 0.3 , 0.35, 0.4 , 0.45, 0.5 ,
        0.55, 0.6 , 0.65, 0.7 , 0.75, 0.8 , 0.85, 0.9 , 0.95, 1.  ]),
 <BarContainer object of 20 artists>)
```



3. Robust Scaler

1. It is used to scale the feature to median and quantiles

1. Scaling using median and quantiles consists of substracting the median to all the observations, and then dividing by the interquantile difference.

1. The interquantile difference is the difference between the 75th and 25th quantile:

- IQR = 75th quantile - 25th quantile

- X_scaled = (X - X.median) / IQR

Example: 0,1,2,3,4,5,6,7,8,9,10

- 9 means 90 percentile---90% of all values in this group is less than 9

- 1 means 10 precentile---10% of all values in this group is less than 1

In [69]:
```
from sklearn.preprocessing import RobustScaler
```

In [70]:
```
scaler=RobustScaler()
```

In [71]:
```
df_robust_scaler=pd.DataFrame(scaler.fit_transform(df),columns=df.columns)
df_robust_scaler.head()
```

Out[71]:

	Survived	Pclass	Age	Fare
0	0.0	0.0	-0.461538	-0.312011
1	1.0	-2.0	0.769231	2.461242
2	1.0	0.0	-0.153846	-0.282777

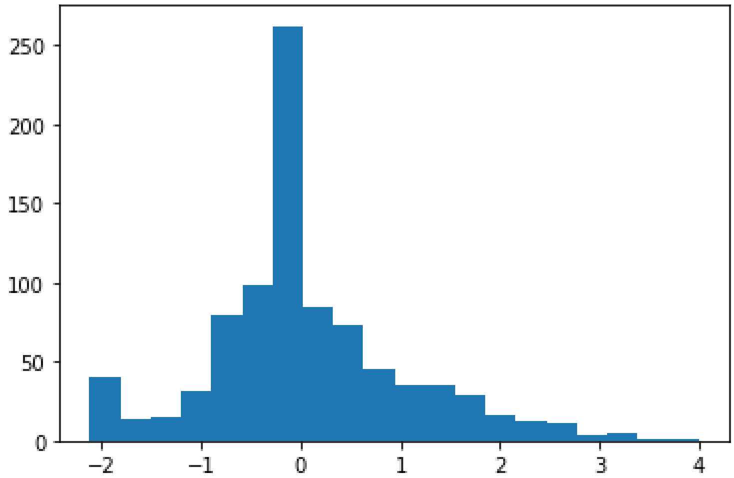
	Survived	Pclass	Age	Fare
3	1.0	-2.0	0.538462	1.673732
4	0.0	0.0	0.538462	-0.277363

In [73]:

```
plt.hist(df_robust_scaler['Age'],bins=20)
```

Out[73]:

```
(array([ 40.,  14.,  15.,  31.,  79.,  98., 262.,  84.,  73.,  45.,  35.,
         35.,  29.,  16.,  13.,  11.,   4.,   5.,   1.,   1.]),
 array([-2.12153846, -1.81546154, -1.50938462, -1.20330769, -0.89723077,
        -0.59115385, -0.28507692,  0.021     ,  0.32707692,  0.63315385,
         0.93923077,  1.24530769,  1.55138462,  1.85746154,  2.16353846,
         2.46961538,  2.77569231,  3.08176923,  3.38784615,  3.69392308,
         4.        ]),
 <BarContainer object of 20 artists>)
```

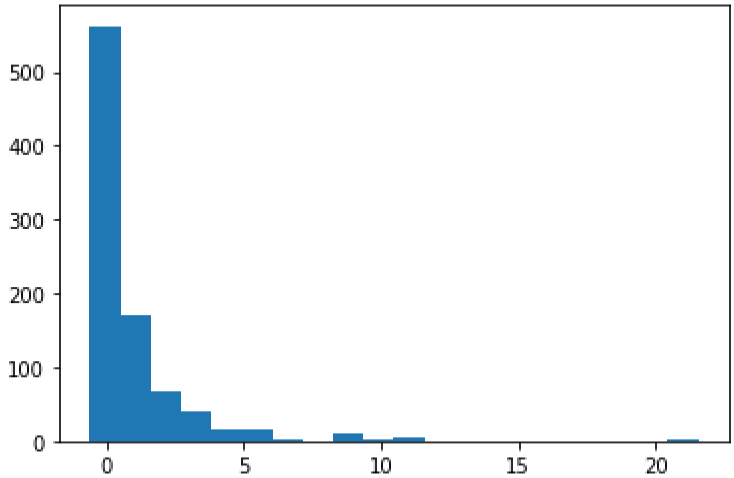


In [74]:

```
plt.hist(df_robust_scaler['Fare'],bins=20)
```

Out[74]:

```
(array([562., 170.,  67.,  39.,  15.,  16.,   2.,   0.,   9.,   2.,   6.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.]),
 array([-0.62600478,  0.48343237,  1.59286952,  2.70230667,  3.81174382,
         4.92118096,  6.03061811,  7.14005526,  8.24949241,  9.35892956,
        10.46836671, 11.57780386, 12.68724101, 13.79667816, 14.90611531,
        16.01555246, 17.12498961, 18.23442675, 19.3438639 , 20.45330105,
        21.5627382 ]),
 <BarContainer object of 20 artists>)
```



Feature Transformation

Types Of Transformation

- Guassian Transformation
- Logarithmic Transformation
- Reciprocal Trnasformation
- Square Root Transformation
- Exponential Trnasformation
- Box Cox Transformation

1. Guassian Transformation

Some machine learning algorithms like linear and logistic assume that the features are normally distributed

- Accuracy
- Performance

In [75]:
```
df=pd.read_csv(r'C:\Users\Gaurav\Downloads\titanic.csv',usecols=['Age','Fare','Survi
df.head()
```

Out[75]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [76]:
```
### fillnan
df['Age']=df['Age'].fillna(df['Age'].median())
```

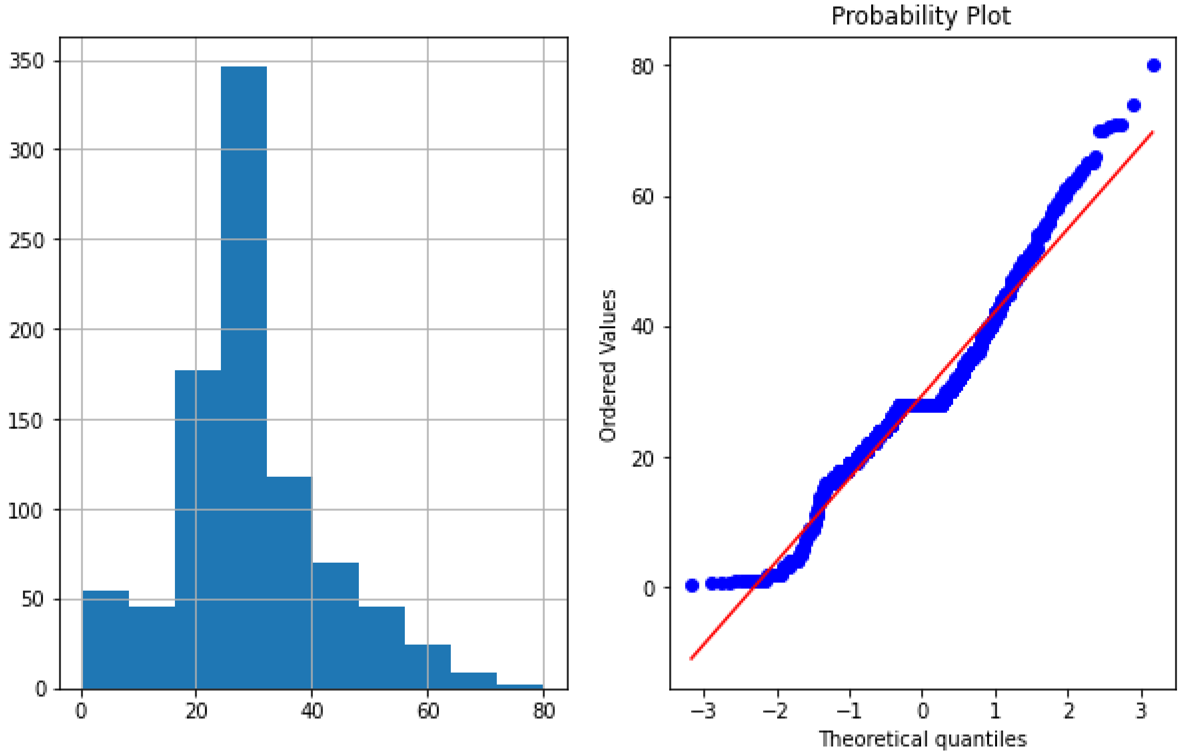
In [77]:
```
df.isnull().sum()
```

Out[77]:
```
Survived    0
Age         0
Fare        0
dtype: int64
```

In [78]:
```
import scipy.stats as stat
import pylab
```

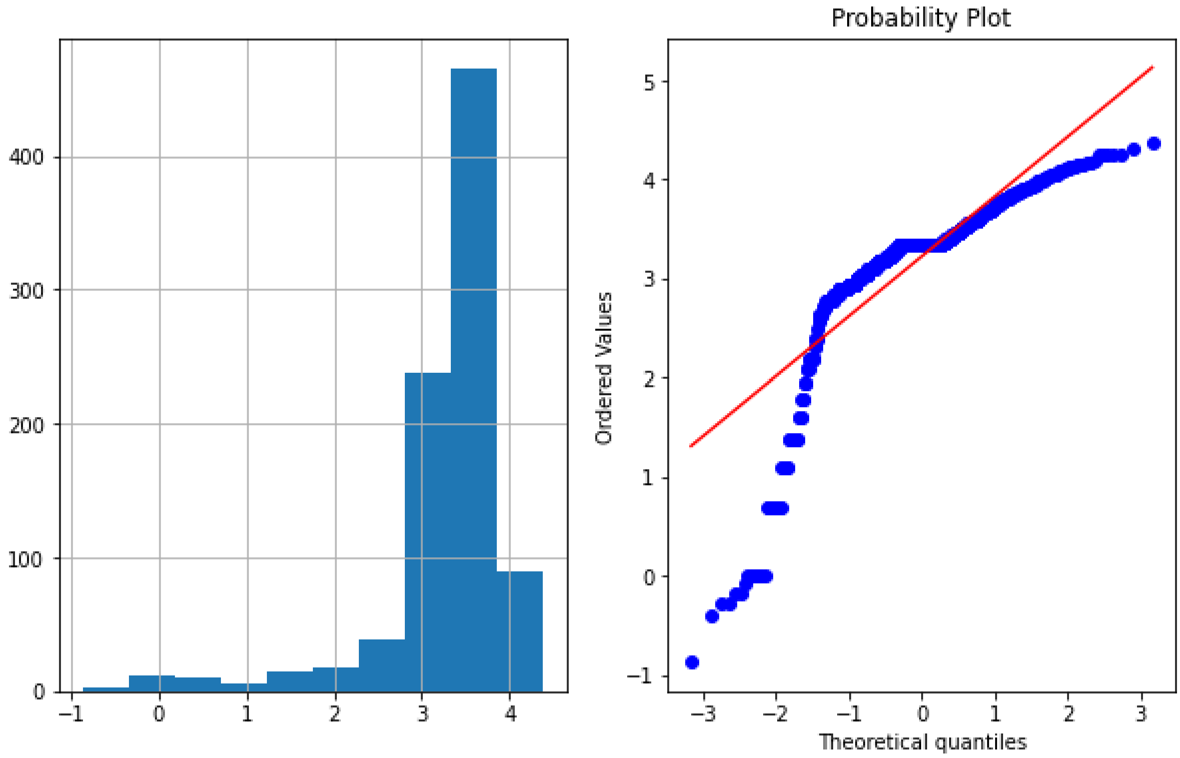
In [79]:
```
#### If you want to check whether feature is guassian or normal distributed
#### Q-Q plot
def plot_data(df,feature):
    plt.figure(figsize=(10,6))
    plt.subplot(1,2,1)
    df[feature].hist()
    plt.subplot(1,2,2)
    stat.probplot(df[feature],dist='norm',plot=pylab)
    plt.show()
```

In [80]:
```
plot_data(df,'Age')
```



2. Logarithmic Transformation

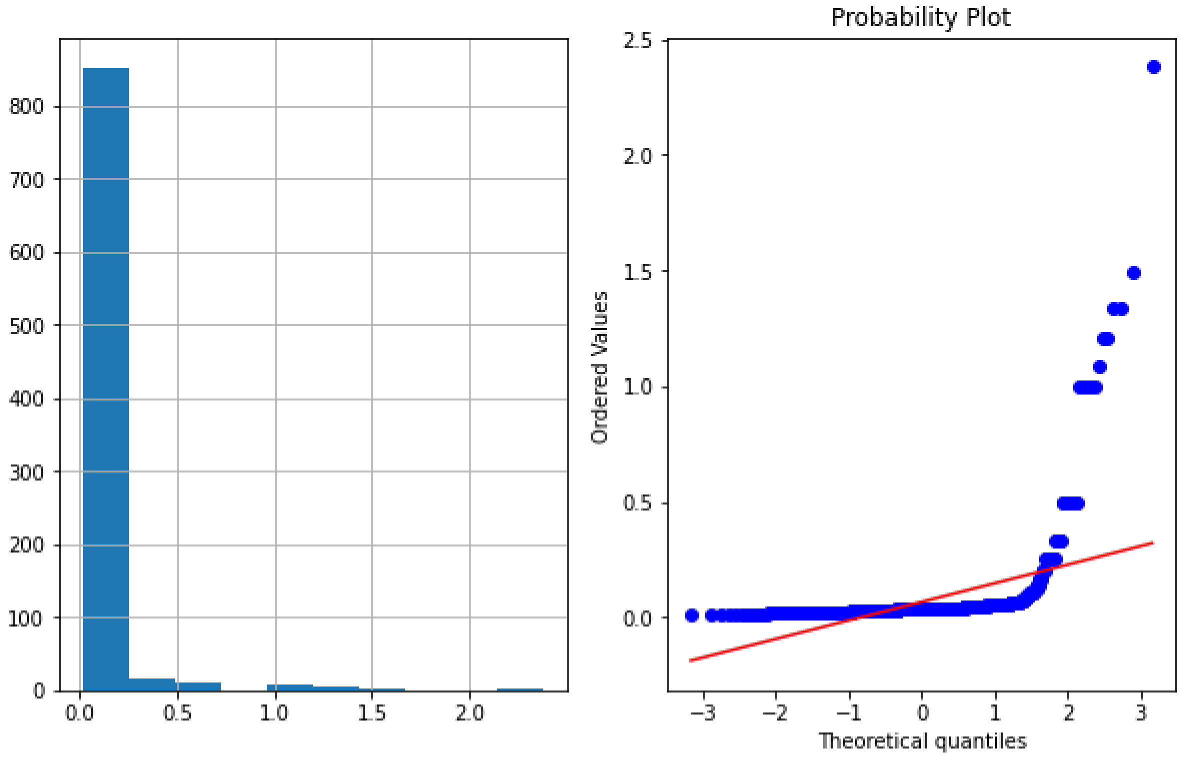
In [81]:
```
import numpy as np
df['Age_log']=np.log(df['Age'])
plot_data(df,'Age_log')
```



3. Reciprocal Trnasformation

In [82]:

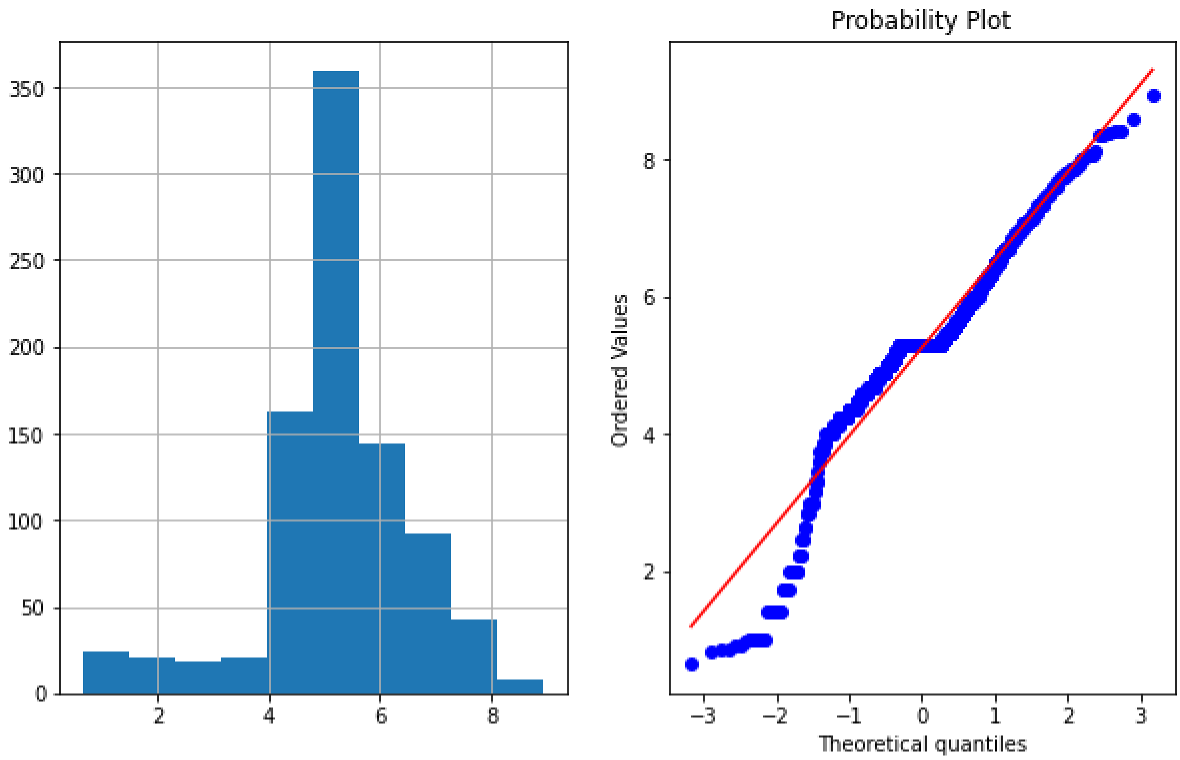
```
df['Age_reciprocal']=1/df.Age
plot_data(df,'Age_reciprocal')
```



4. Square Root Transformation

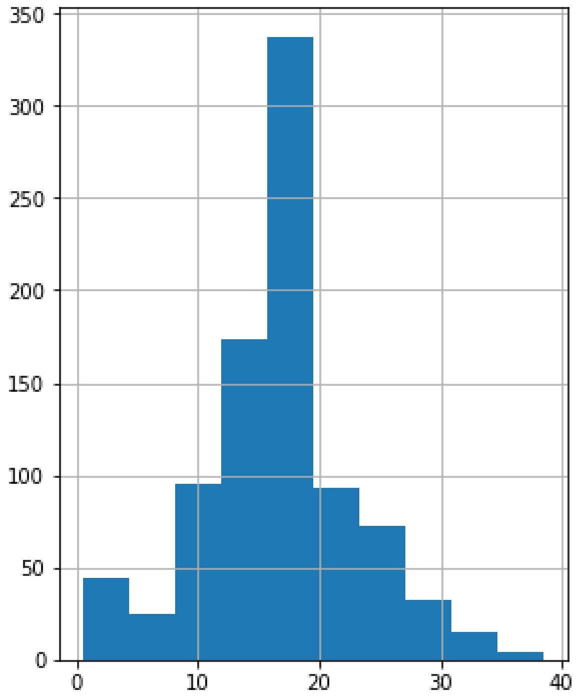
In [86]:

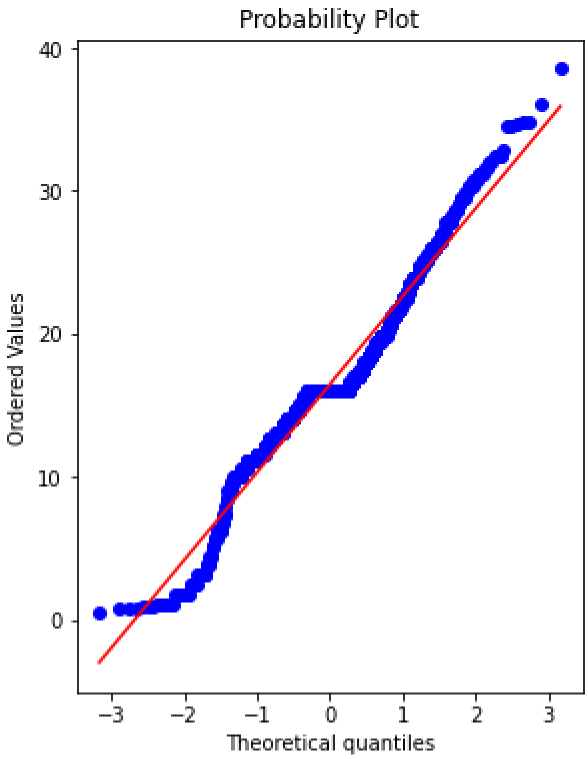
```
df['Age_sqaure']=df.Age**(1/2)
plot_data(df,'Age_sqaure')
```



5. Exponential Transdormation

In [87]:
```
df['Age_exponential']=df.Age**(1/1.2)
plot_data(df,'Age_exponential')
```





6. BoxCOx Transformation

The Box-Cox transformation is defined as:

T(Y)=(Y exp(λ)−1)/λ

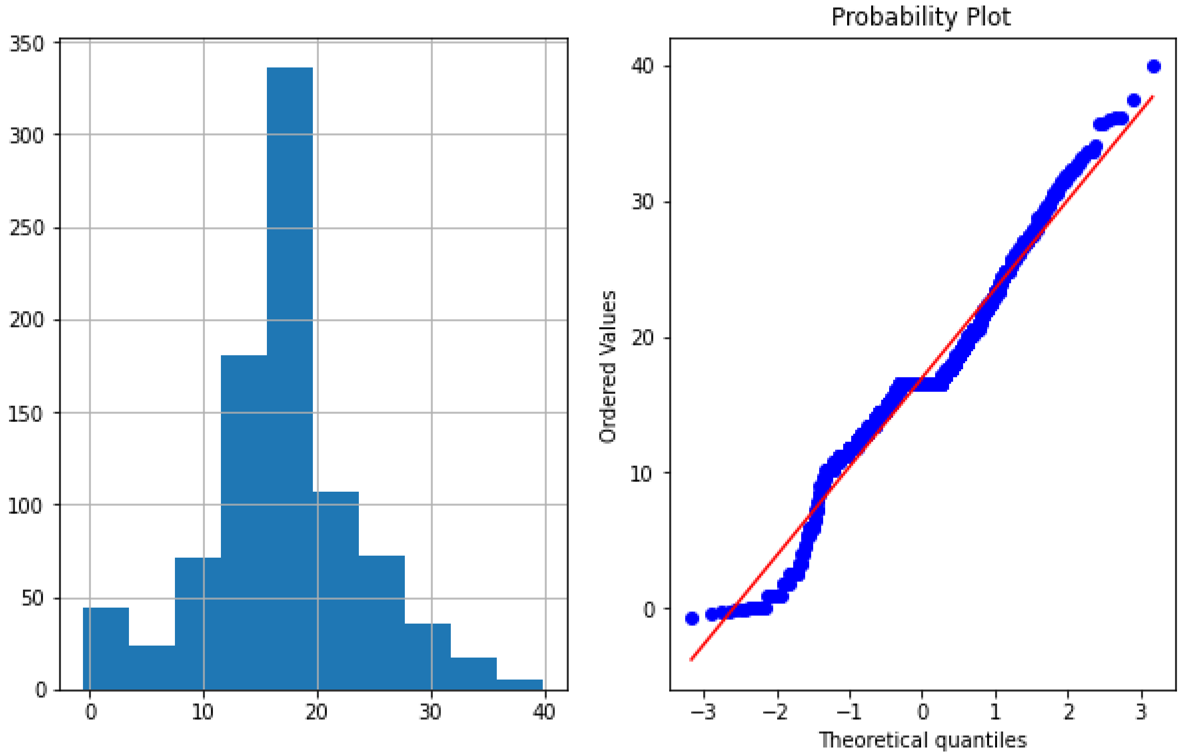
where Y is the response variable and λ is the transformation parameter. λ varies from -5 to 5. In the transformation, all values of λ are considered and the optimal value for a given variable is selected.

In [88]:
```
df['Age_Boxcox'],parameters=stat.boxcox(df['Age'])
```

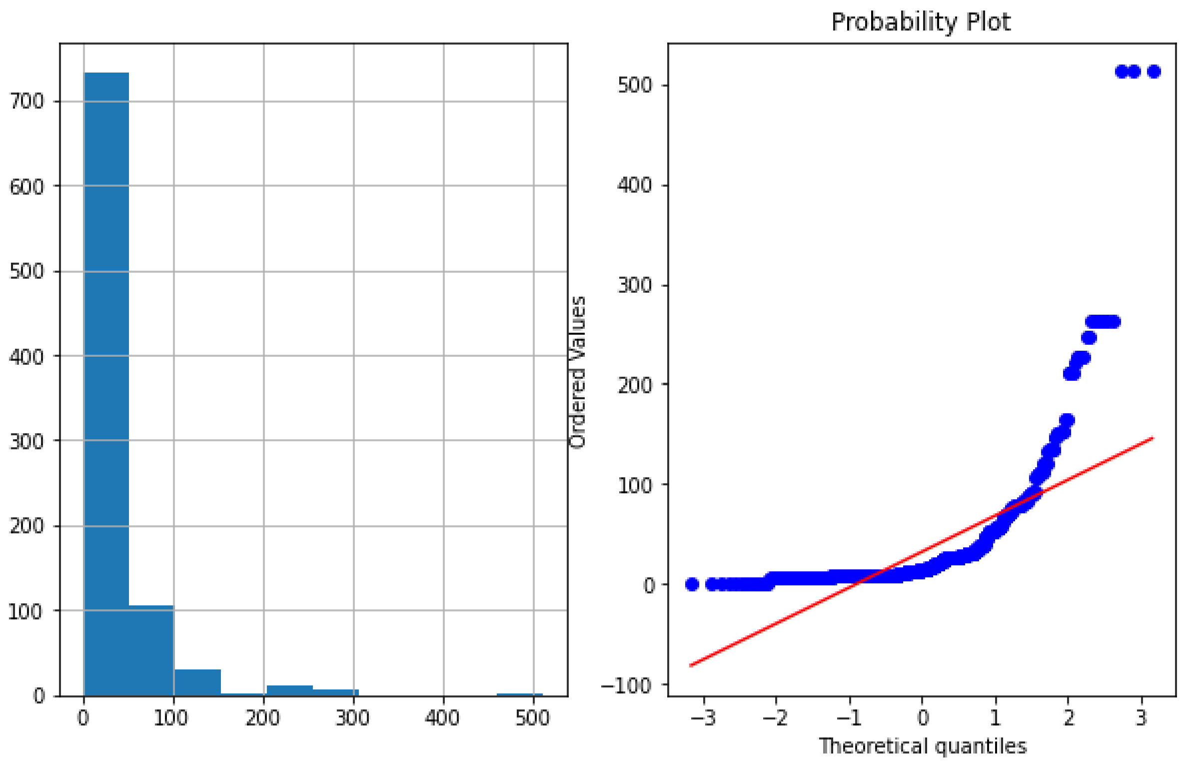
In [89]:
```
print(parameters)
```

```
0.7964531473656952
```

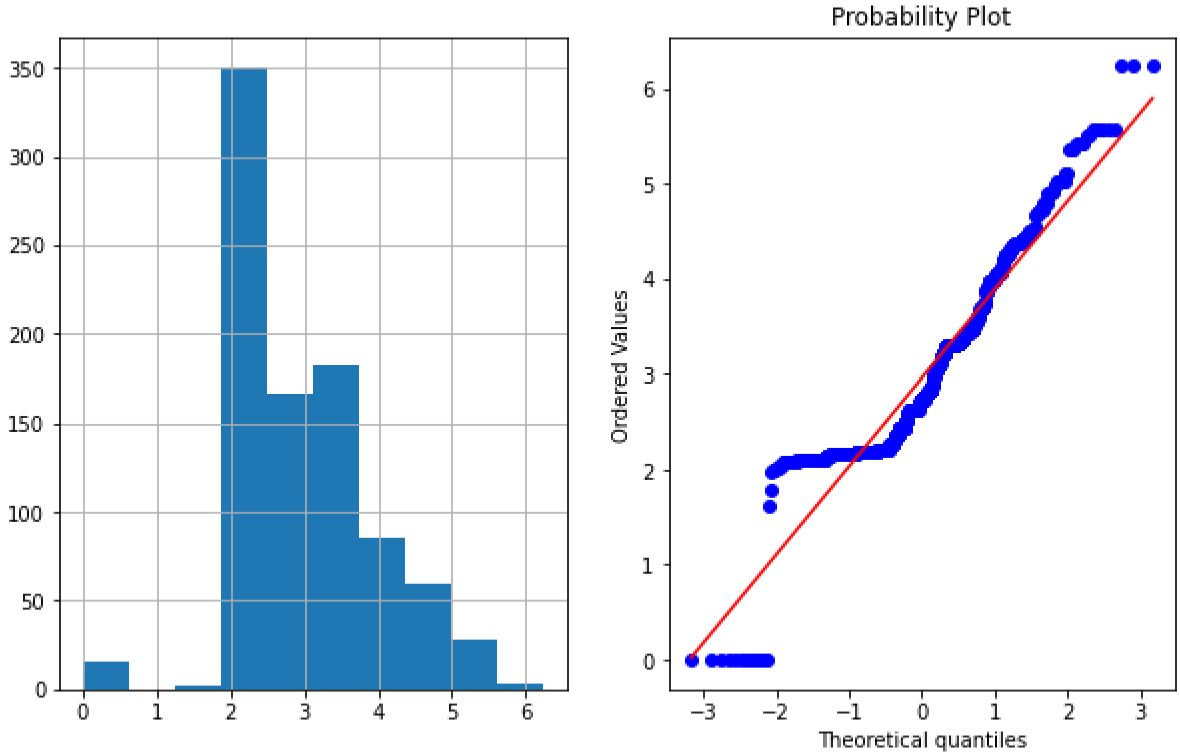
In [90]:
```
plot_data(df,'Age_Boxcox')
```



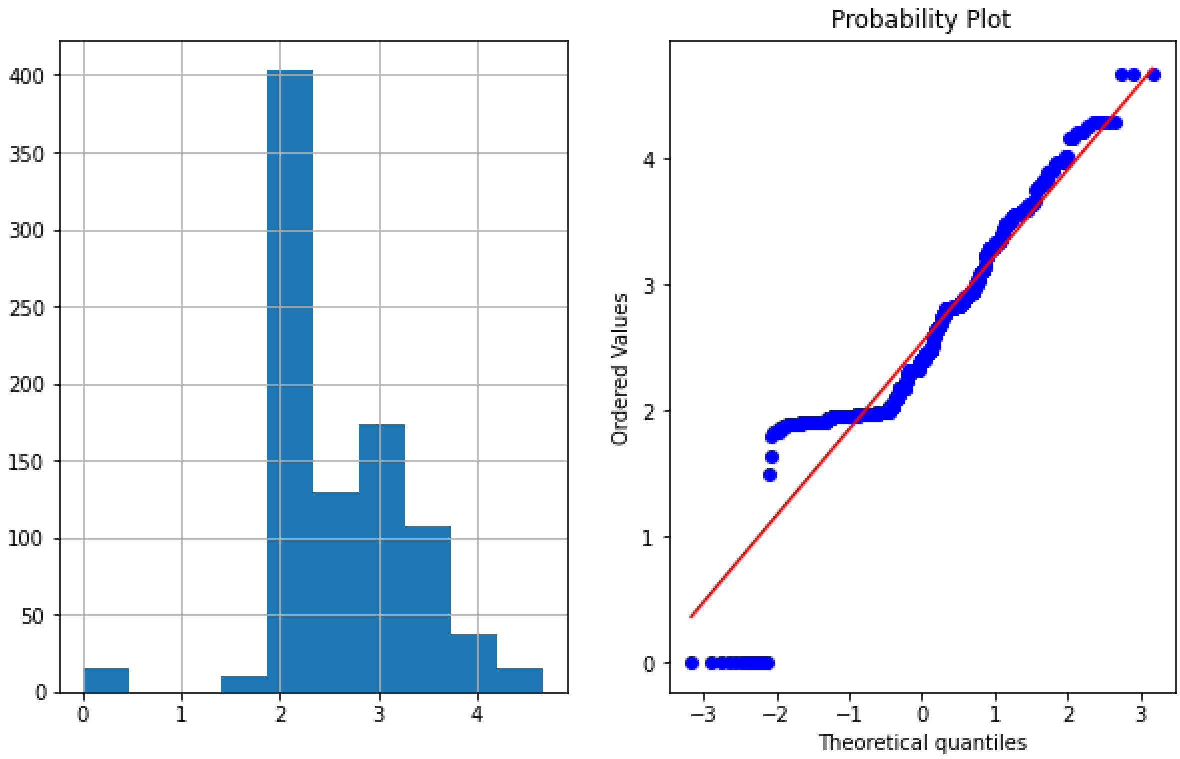
In [91]:
```
plot_data(df,'Fare')
```



In [92]:
```
#### Fare
df['Fare_log']=np.log1p(df['Fare'])
plot_data(df,'Fare_log')
```



In [93]:
```
df['Fare_Boxcox'],parameters=stat.boxcox(df['Fare']+1)
plot_data(df,'Fare_Boxcox')
```



In [ ]:

In [ ]:

In [ ]:

In [ ]: