DATE : / /
PAGE NO. :

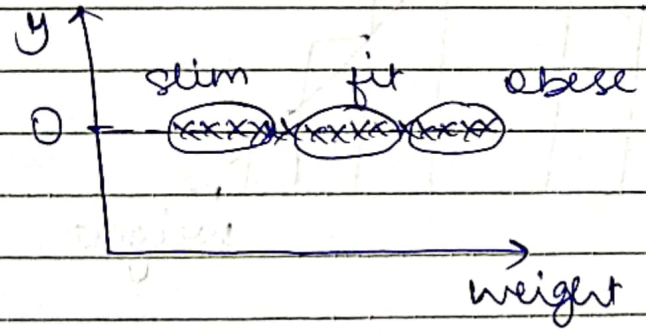
Univariate, Bivariate & Multivariate Analysis

Univariate

- We only take one feature in account

#data

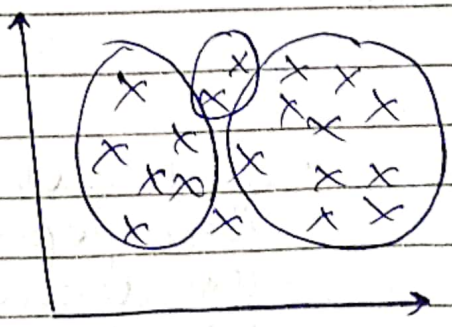
Height	Weight	O/p
180	90	Obese
160	50	slim
170	78	fit
190	90	fit
195	80	slim
⋮	⋮	⋮



- Overlapping of data pts.

Bivariate Analysis

Height & Wt.



DATE : / /
PAGE NO. :

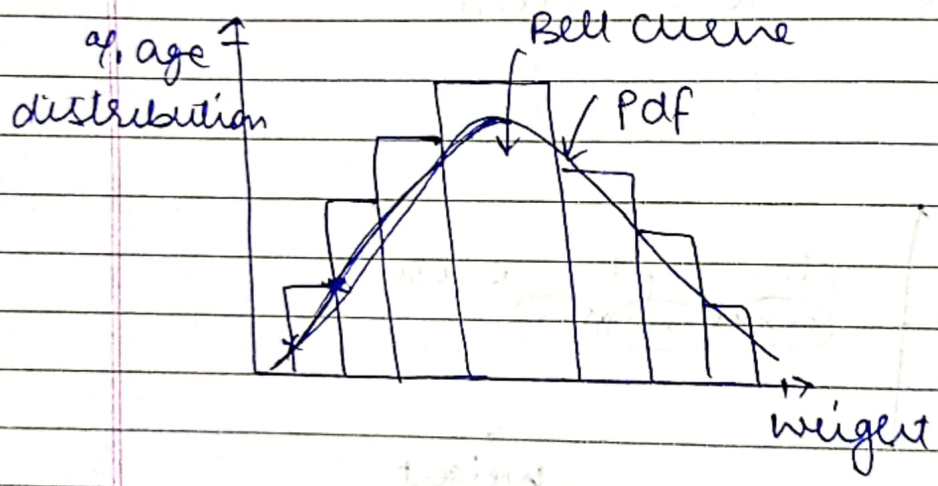
Multivariate Analysis

- when we have multiple features
- we use pairplot of seaborn lib. to plot various graphs

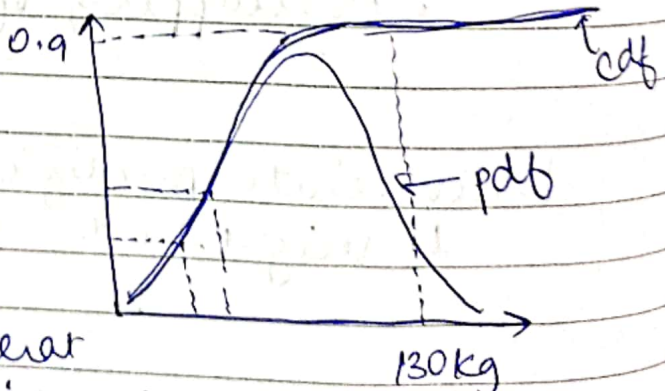
pair plots { Age vs Age, Age vs wt, Age vs hei; Height vs Age, Height vs wt, Height vs He; wt vs Age, wt vs wt, wt vs hei

~~Histogram~~

Probability Density fn.



CDF
Cummilative density fn



- It shows that 90% population is having wt. less than 130kg and 10% has greater wt. than 130kg.

DATE : / /
PAGE NO. :

Types of Encoding

1. Nominal encoding
   - categories do not have any order.
     - One hot encoding
     - One hot encoding with many categories
     - mean encoding

2. Ordinal encoding
   - categories can be rearranged on the basis of ranks.
     - Label encoding
     - Target guided ordinal encoding

value ↑ → label encoding ↑

ex:

BE → 1
MAS → 3
PHD → 4 → Highest imp. high no.
Grad → 2

Handle Missing values in categorical var.

1. Delete Rows
2. Replace with the most frequent values
   - problem: imbalanced dataset within the category.
3. Apply classifier algo to predict.
4. Apply unsupervised ML
   - group them into categories

DATE : / /
PAGE NO. :

Handle categorical features; many categories (count / frequency encoding)

- 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.
- Approach: Each label of the categorical variable by the count, this is the amount of times each label appears in a dataset. Or the frequency, this is the % age of observations within that category.

#

```
mapped = df.X2.value_counts().to_dict()
```

↳ { 'as' : 1659,
'ae' : 496,
'ai' : 415
⋮ } ← { frequency, categories

```
df.X2 = df.X2.map(mapped)
```

↑ maps all the ~~fea~~ categories with their frequency

Advantages:

- Simple to implement
- Does not increase the feature dimensional space.

DATE : / /
PAGE NO. :

1. Disadvantages:
- If some labels have the same count, then they will be replaced with the same count and they will loose some valuable infom.
- Adds somewhat arbitary no, i.e. weights to the different labels, they may not be related to their predictive power.

~~Data to handle Categories~~

Missing values

Different types of missing values :

1. Missing Completely At Random (MCAR)
- A variable is missing completely at random if the probability of missing is the same for all the observations.
- When data is MCAR, there is absolutely no relationship between the missing data.
- In other words, those missing data points are a random subset of the data. There is nothing systematic going on that makes some data more likely to be missing than other.

DATE : / /
PAGE NO. :

2. Missing Data Not at Random (MNAR) (Systematic missing values)
- There is some relationship between the data missing and any other values, observed or missing within the dataset.

3. Missing at Random (MAR)
- It means that the reason for missing values can be explained by variables on which you have complete information as there is some relationship between the missing data and other values/data.
- In this case, the data is not missing for all the observations. It is missing only within sub-samples of the data and there is some pattern in the missing values.

Techniques for handeling missing values
1. Mean / Median / Mode replacement
2. Random sample Imputation
3. Capturing nan values with a new feature
4. End of distribution imputation
5. Arbitary imputation
6. Frequent categories imputation

DATE : / /
PAGE NO. :

1. Mean / Median / Mode

- Mean / Median imputation has the assumption that the data is MCAR.
- We solve this by replacing the nan with the most frequency occurance of the variables.

#

```
df.isnull().mean()
```

↳ mean of null values.

```
def impute_nan(df, variable, median):
    df[variable+"_median"] =
        df[variable].fillna(median)

median = df['Age'].median()

print( df['Age'].std())
print( df['Age_median'].std())
```

↳ 14.5264
13.0196] There is no drastic difference.

#we can also observe the diff. by graph

```
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')
```

DATE : / /
PAGE NO. :

↳ Advantages :
- Easy to implement (robust to outliers)
- Faster way to obtain the complete ds.

↳ Disadvantages :
- Change or distortion in the original variance.
- Impacts correlation

2. Random sample imputation
↳ It consists of taking random observation from the dataset and we use this observation ~~from the data and are use this ds~~ to replace the nan values.
↳ It assumes that the ~~na~~ data that are missing are completely at random (MCAR)

#
```
def impute_nan(df, variable):
    median = df[variable].median()
    df[variable + '_median'] =
        df[variable].fillna(median)
    df[variable + '_random'] = df[variable]
    random_sample =
    df[variable].dropna().sample(
    df[variable].isnull().sum,
        random_state = 0)
    random_sample.index =
     df[df[variable].isnull()].index
```

DATE : / /
PAGE NO. :

```
df.loc[df[variable].isnull(), variable+'_random'] = random_sample

# create plot to see the change in distribution
```

↳ Advantages:
- Easy to implement.
- less distortion in variance.

↳ Disadvantages:
- In Every situation randomness won't work.

3. Capturing NAN values with a new feature
↳ It works well if the data are not missing completely at random (MNAR).

```
#
df['Age_NAN'] = np.where(df['Age'].isnull(),1,0)
# now we can handle missing values
```

↳ Advantages:
- Easy to implement. → with mean, median, random imputation
- Captures the importance of missing values.

↳ Disadvantages:
- Creates additional features.

DATE : / /
PAGE NO. :

4. End of distribution imputation
↳ Here we take the data points from the end of distribution and replace the nan values ↳ tail of distribution.

↳ Here missing values are not at random (MNAR)

```
#
df['Age'].hist(bins=50)
extreme=df['Age'].mean() + 3*df['Age'].std()
```
↳ mean after 3rd S.d.

```
sns.boxplot('Age', data=df)

median = df['Age'].median()

def impute_nan(df, variable, median, extreme)
    df[variable+'_end_dist'] = df[variable].
                                fillna(extreme)
    df[variable].fillna(mean, inplace=True)

impute_nan(df, 'Age', median, extreme)
```

↳ Advantages:
- Easy to implement.
- captures the importance of missingness if there is one.

DATE : / /
PAGE NO. :

4. 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 the no of NA is big, it will mask true outliers in the distribution.
- If the no. of NA is small, the replaced NA may be considered an outlier and pre-processed in a subsequent step of feature engineering.

5. Arbitary value Imputation
- It consists of replacing the nan values by an arbitary value.

```
def impute_nan(df, variable):
    df[variable + "_zero"] = df[variable].
                             fillna(0)
    df[variable + "_hundred"] = df[variable].
                                fillna(100)
```

Advantages:
- Easy to implement.
- Captures the importance of missingness if there is one.

DATE : / /
PAGE NO. :

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.
- Hard to decide which value to use.

Handling categorical missing values

6. Frequent category Imputation

```
def impute_nan(df, variable):
    most_freq_cate = df[variable].value_counts()
                        index[0]
    df[variable].fillna(most_freq_cate,
                        inplace=True)
```

or

```
df[variable].fillna(df[variable].mode()[0],
                    inplace=True)
```

Advantages:
- Easy to implement.
- Faster way to implement

Disadvantages:
- Since we are using the most frequent

DATE : / /
PAGE NO. :

labels, it may use them in order representated way, if there are many nans.
- It differs the relation of the most frequent label.

7. Adding a variable to capture nan

```
df['BsmtQual_var'] = np.where(df['BsmtQual'].
                              isnull(), 1, 0)
#adds new col with val 0
#where there is nan value.
```

Then we can impute the nan with mode().

```
df['BsmtQual'].fillna(df['BsmtQual'].
    mode()[0], inplace=True)
```

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

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

DATE : / /
PAGE NO. :

Handling categorical features

1. One hot encoding

```
pd.get_dummies(df['sex'], drop_first = True)
```

- If many categories are there then we can take top 10 or 15 most frequent categories and one hot encode them.

```
list_10 = [df['x1'].value_counts().
           sort_values(ascending = False).
           index]

for category in list_10:
    df[category] = np.where(df['x1'] =
                   category, 1, 0)
```

2. Ordinal number encoding

```
dictionary = {'monday' : 1,
              'Tuesday': 2,
              'wednesday': 3,
                  :
              'sunday' : 7}

df['week_ordinal'] =
    df['weekday'].map(dictionary)
```

DATE: / /
PAGE NO.:

3. Count of Frequency Encoding
- We map the categories with their frequency.

```
country_map = df['country'].value_counts().to_dict()
df['country'] = df['country'].map(country_map)
```

Advantages:
- Easy to implement.
- Not increasing feature space.

Disadvantages:
- It will provide same weight if the frequencies of the categories are same.

4. Target guided Ordinal Encoding
- Ordering the labels according to the target.
- Replace the labels by the joint probability of being 1 or 0.

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

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

ordinal_dict
{'T': 0, 'M': 1, 'A': 2, ... 'D': 8}
 ↳ max value
```

DATE : / /
PAGE NO. :

```
df['cabin_ordinal'] = df['cabin'].map(
                        ordinal_dict)
```

5. Mean Encoding
- We replace the categories by replacing it with its mean.

```
mean_ordinal = df.groupby(df['cabin'])
    ['Survived'].mean().to_dict()
mean_ordinal
  {'A' : 0.466 ,
   'B' : 0.744 ,
    :
   'M' : 0.299
   'T' : 0.0}

df['mean_encode'] = df['cabin'].map(
                        mean_ordinal)
```

b Advantages:
- Captures the information within the labels. Therefore sendering more predictive features.
- Creates a monotonic relationship between the variable and the target.

b Disadvantages:
- Prones to overfitting.

DATE : / /
PAGE NO. :

6. Probability Ratio Encoding

Probability Ratio = $\frac{P(\text{success})}{P(\text{failure})}$

```
prob_df = df.groupby(['Cabin'])['Survived'].mean()
prob_df = pd.DataFrame(prob_df)
prob_df['Died'] = 1 - prob_df['Survived']
prob_df['Prob_ratio'] = prob_df['Survived'] /
                        prob_df['Died']
prob_map = prob_df['Prob_ratio'].to_dict()
df['Cabin_encoded'] = df['Cabin'].map(
                      prob_map)
```

Types of Transformation

1. Normalization and standardization

a. Standardization:

- We try to bring all the variables and features to a similar scale.
- Standardization means centering the variable at zero

$$Z = \left(\frac{x - \bar{x}}{\sigma_x}\right) \quad \left| \quad \frac{x - x\_mean}{std}\right.$$

DATE : / /
PAGE NO. :

#

```
from sklearn.preprocessing import
                                StandardScaler
scaler= StandardScaler()
df_scaled = scaler.fit_transform(df)
pd.DataFrame(df_scaled)
```

2. Min Max scaling
   - widely used in D.L (CNN)
   - Scales the values bet 0 & 1.

$$X\_scaled = \frac{x - X.min}{X.max - X.min}$$

```
from sklearn.preprocessing import
                                MinMaxScaler
min_max= MinMaxScaler()
df_minmax= pd.DataFrame(min_max.
            fit_transform(df), columns=
                                df.columns)
df_minmax
```

3. RobustScaler
   - It is used to scale the features to median and quantiles
   - Scaling using median and quantiles consists of subtracting the median to all the observations

DATE: / /
PAGE NO.:

and then dividing by the interquartile difference. The interquartile difference is the difference bet. the 75th quantile and 25th quantile.

$$IQR = 75^{th}\ q. - 25^{th}\ q.$$

$$x\_scaled = \frac{(x - x.median)}{IQR}$$

#
```
from sklearn.preprocessing import
                                RobustScaler
scaler = RobustScaler()
df_robust_scaled = pd.DataFrame(
        scaler.fit_transform(df),
        columns = df.columns)
df_robust_scaled.head()
```

4. Guassian Transformation
- some ML algom like linear and logistic assume that the features are normally distributed.

- If we want to check gaussian or normal distributed we use Q-Q plot.

DATE : / /
PAGE NO. :

```
import scipy.stats as stat
import pylab

def plot_data(df, feature):
    plot.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 the plot (1,2,2) if all the pts lie in a straight line then we can say our data is normally distributed.

b Logarithmic transformation

```
df['Age_log'] = np.log(df['Age'])
plot_data(df, 'Age')
```

- If there is error in log fn then use

```
df['Age_log'] = np.log(df['Age'] + 1)
```

b Reciprocal transformation

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

DATE: / /
PAGE NO.:

- Square Root transformation

```
df['Age_sqt'] = df.Age ** (1.2)
plot_data(df, 'Age_sqt')
```

- Exponential Transformation

```
df['Age_exp'] = df.Age ** (1/1.2)
plot_data(df, 'Age_exp')
```

- Box Cox Transformation

~~df['Age_bc'] = stats.boxcox(df['Age'])~~

```
df['Age_boxcox'], params = stat.boxcox(df['Age'])
plot_data(df, 'Age_boxcox')
print(params)
```

Handeling Imbalanced data

```
!pip install imbalanced-learn
```

1. Under Sampling
   - reduce the points of the maximum labels.
   - we should not use undersampling bec. there is loss of data.
   - we should only use it when we have very low data set.

DATE : / /
PAGE NO. :

```
# from collections import Counter
from imblearn.under_sampling import
                                NearMiss
ns = NearMiss(0.8)
X_train_ns, y_train_ns = ns.fit_sample(X_train, y_train)

print ('class before fit {}'.format(
                        Counter(y_train)))
print ("class after fit {}".format(
                        Counter(y_train_ns)))
```

↳ {0: 199017, 1: 347}
{0: 433, 1: 347}

Formula : 0.8 * 433 = 347

2. Over sampling

```
#
from imblearn.over_sampling import
                        RandomOverSampler
os = RandomOverSampler(0.5)
X_train_os, y_train_os = os.fit_sample(
                        X_train, y_train)

print ("No class before fit {}".format(
                        Counter(y_train)))
print ("class after fit {}".format(
                        Counter(y_train_os)))
```

DATE : / /
PAGE NO. :

- {0: 199017, 1: 347}
  {0: 199017, 1: 99508}

3. SMOTE Tomek

```
from imblearn.combine import SMOTETomek
os = SMOTETomek(0.5)
X_train_st, y_train_st = os.fit_sample
                         (X_train, y_train)
print(_____)
print(_____)
```

- {0: 199017, 1: 347}
  {0: 199017, 1: 148640}

4. Ensemble Techniques

```
from imblearn.ensemble import
                    EasyEnsembleClassifier
easy = EasyEnsembleClassifier()
easy.fit(X_train, y_train)
```

Outliers and its Impacts

Q. Which ML algm are sensitive from outliers?

- Naive Bayes classifier – Not Sensitive
- SVM — Not sensitive
- ✓ Linear Regression — Sensitive
- ✓ Logistic Regression — Sensitive
- Decision Trees — Not sensitive

DATE : / /
PAGE NO. :

- Ensemble (RF, XGBoost, GB) — Not Sensitive
- KNN — Not Sensitive
- ✓ KMeans — sensitive
- ✓ Hierarchical — Sensitive
- ✓ PCA — Sensitive
- ✓ Neural Networks — Sensitive

1. Gaussian Distributed Data

upper_boundary = df['Age'].mean() + 3 * df['Age'].std()

lower_boundary = df['Age'].mean() - 3 * df['Age'].std()

IQR = df['Age'].quantile(0.75) - df['Age'].quantile(0.25)

lower_bridge = df['Age'].quantile(0.25) - (1.5 * IQR)

upper_bridge = df['Age'].quantile(0.75) + (1.5 * IQR)

we will consider the values after lower_boundary & upper_boundary as outliers.

DATE : / /
PAGE NO. :

2. Skewed data

IQR = df['Fare'].quantile(0.75) - df['Fare'].quantile(0.25)

lower_bridge : _____

upper_bridge : _____

FEATURE ENGINEERING STEP BY STEP

I. (Exploratory Data Analysis)
- Numerical features, categorical features
- Histogram, pdf, etc.
- Missing values (visualise)
- Outliers (Box plot)
- cleaning or not

II. Handeling the missing values

III. Handeling Imbalanced data

IV. Treating the outliers

V. scaling down the data

VI. Converting categorical features to numerical features

DATE : / /
PAGE NO. :

Feature selection : selecting the best features

1. Correlation
2. K-Neighbours
3. Chisquare
4. Genetic Algm
5. Feature importance (extra tree classifier)