Life Cycle of Data Science Projects

1.Data Analysis

2.Feature Engineering

3.Feature Selection

4.Model Building

5.Model Deployment

```
import pandas as pd # python library to
import numpy as np # numerical python .

#python library for data visualisation

import matplotlib.pyplot as plt
import seaborn as sns

#to suppress warnings

import warnings
warnings.filterwarnings("ignore")
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
data=pd.read_excel('/content/drive/MyDrive/HR.xlsx')
```

```
## To display all columns of the Data Frame

pd.pandas.set_option('display.max_columns',None)
```

```
data.head()
```

	Gender	Business	Dependancies	Calls	Type	Billing	Rating	Age	Salar
0	Female	0	No	Yes	Month-to-month	No	Yes	18	5089.0
1	Female	0	No	Yes	Month-to-month	No	Yes	19	5698.1
2	Male	0	No	Yes	Month-to-month	Yes	No	22	5896.6
3	Female	1	No	Yes	Month-to-month	Yes	Yes	21	6125.1
4	Male	0	No	Yes	Month-to-month	Yes	Yes	23	6245.0

```
# To display column names

data.columns
```

```
Index(['Gender', 'Business', 'Dependancies', 'Calls', 'Type', 'Billing',
       'Rating', 'Age', 'Salary', 'Base_pay', 'Bonus', 'Unit_Price',
'Volume',
       'openingbalance', 'closingbalance', 'low', 'Unit_Sales',
'Total_Sales',
       'Months', 'Education'],
      dtype='object')
```

Feature Description

Gender - talks of the gender - Male or female

Business - if the person has another business or no

Dependants - if there are people dependant on the person

Calls - if the person has authority to make calls or not

Type - salary settlement type or contract type

Billing - Subscribed to billing plans or no

Rating - If he has been given a rating by a superior or no

Age - age of the person

Salary - CTC of the employee

Base pay - Base pay of the employee

Bonus - amount received by a person as bonus for sales

Unit price - The Unit price of a sale

Volume - volume allotted to a person

Opening balance - The opening balance of an employee

Closing Balance- The closing balance of an employee

Low - lowest opening balance allotted to a person.

Unit sales - unit sale made by the person

Total sales - total sales made by the person

Months - duration of the person employed with the company

Education- Educational background of an employee

```
# Number of obs#ervations per feature ie 5000 rows per 20 colums

data.shape
```

```
(5000, 20)
```

```
#statistical summary of the data

data.describe()
```

	Business	Age	Salary	Base_pay	Bonus	Unit_Pr:
count	5000.000000	5000.000000	5000.000000	4977.000000	5000.000000	5000.000
mean	0.160000	51.865000	99821.928553	40046.187707	4991.096428	51.258
std	0.366643	8.560691	25376.961744	10135.686075	1268.848087	52.244
min	0.000000	18.000000	5089.000000	2035.600000	254.450000	1.440
25%	0.000000	47.000000	83890.338980	33720.552420	4194.516950	25.727
50%	0.000000	52.000000	100579.378500	40282.016040	5028.968925	39.205
75%	0.000000	57.000000	116912.092475	46792.232410	5845.604624	58.715
max	1.000000	88.000000	199970.740000	79988.296000	9998.537000	629.511

1.Exploratory Data Analysis

In Data Analysis We Try to analyse the following:

1.Missing Values

2.All the Numerical Variables

3.Distribution of Numerical Variables

4.Categorical Variables

5.Cardinality of Categorical Variables

7.Relationship between independent and dependent feature

```python
data.dtypes
```

```
Gender              object
Business             int64
Dependancies        object
Calls               object
Type                object
Billing             object
Rating              object
Age                  int64
Salary             float64
Base_pay           float64
Bonus              float64
Unit_Price         float64
Volume               int64
openingbalance     float64
closingbalance     float64
low                float64
Unit_Sales         float64
Total_Sales         object
Months               int64
Education           object
dtype: object
```

The datatype of the variable 'Total_Sales' is object. It is a numerical variable. It contains space as a value so considered as object. Need to be changed in to numerical type for proper treatment

1.Missing Values

```
# Sum of missing values in each columns



data.isnull().sum()
```

```
Gender               0
Business             0
Dependancies         0
Calls                0
Type                 0
Billing              0
Rating               0
Age                  0
Salary               0
Base_pay            23
Bonus                0
Unit_Price           0
Volume               0
openingbalance    1476
closingbalance       0
low                  0
Unit_Sales           0
Total_Sales          8
Months               0
Education            0
dtype: int64
```

```
#replace the object items in coloumn Total_Sales with "NaN"

data['Total_Sales'] = data['Total_Sales'].replace(' ', pd.NA)
```

```
#change the dtype of "Total_Sales" as float

data['Total_Sales'] = data['Total_Sales'].astype('Float64')
data['Total_Sales'] = data['Total_Sales'].astype('float64')
data['Total_Sales'].dtype
```

```
dtype('float64')
```

```
data.isnull().sum()
```

```
Gender              0
Business            0
Dependancies        0
Calls               0
Type                0
Billing             0
Rating              0
Age                 0
Salary              0
Base_pay           23
Bonus               0
Unit_Price          0
Volume              0
openingbalance   1476
closingbalance      0
low                 0
Unit_Sales          0
Total_Sales        16
Months              0
Education           0
dtype: int64
```

Its evident that Null values increased after changing data type of Total_Sales

```
#Checking the percentage of null values in each feature
#Step 1 - Creating a list of features with null values

features_with_na = [features for features in data.columns if data[features].isnu

#Step2 -Print the feature name and percentage of missing values

for feature in features_with_na:
    print(feature,np.round(data[feature].isnull().mean(),3), ' % missing values'
```

```
Base_pay 0.005  % missing values
openingbalance 0.295  % missing values
Total_Sales 0.003  % missing values
```

There are missing values with 3 columns and only the opening balance shows 30% missing values now lets find the relationship between missing values and Salary(Dependent Variable, target, column

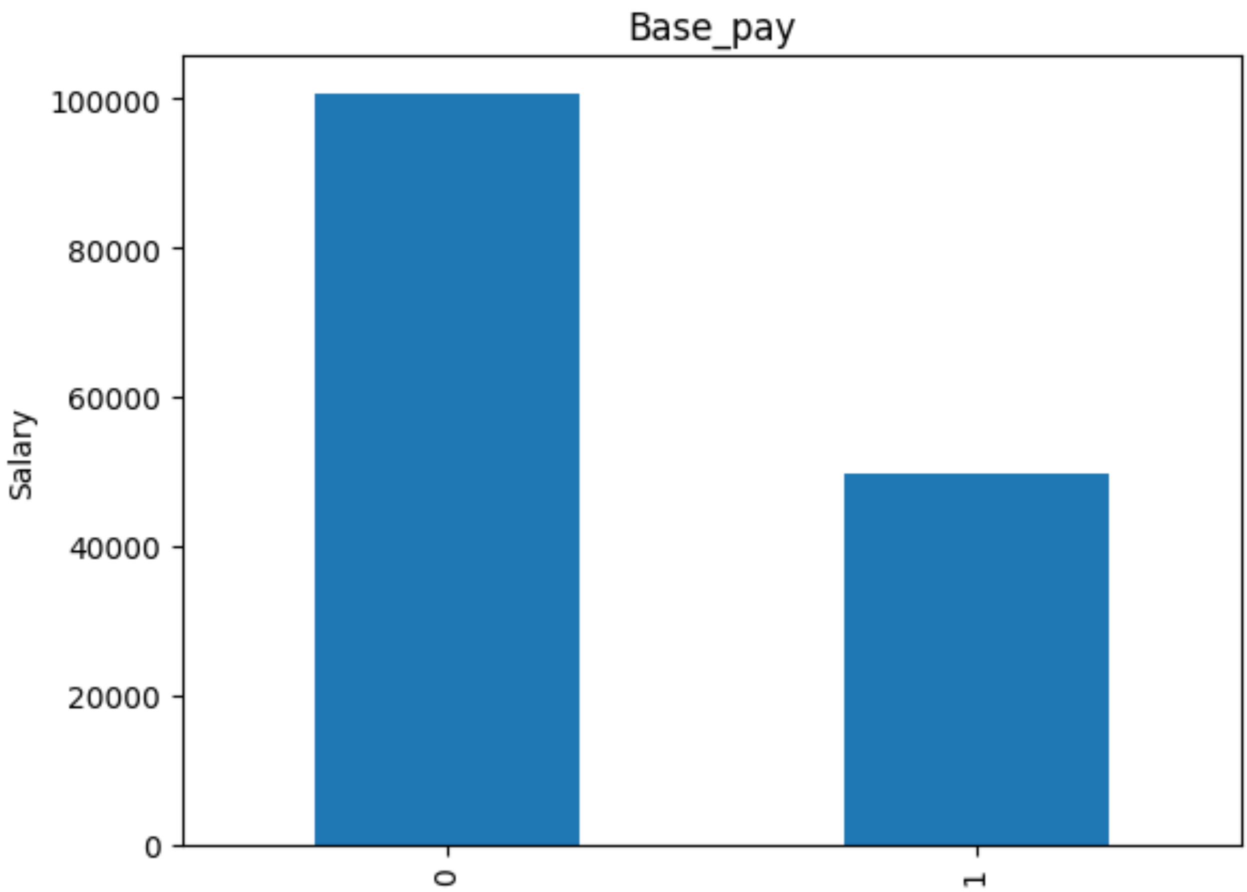
```
# Print features with null values stored to a variable

features_with_na
```

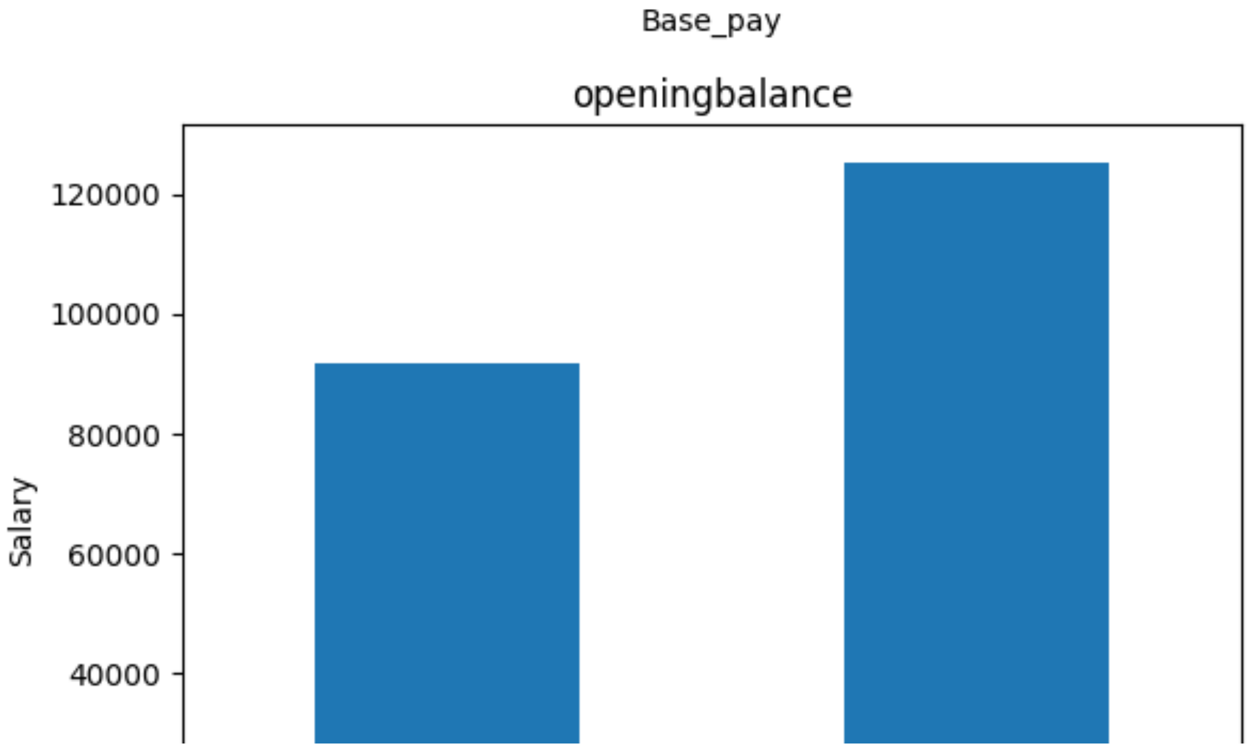
```
['Base_pay', 'openingbalance', 'Total_Sales']
```

BIVARIATE ANALYSIS

Bar Plot

```
# Bivariate analysis to plot relation of null values with target column (salary)

for feature in features_with_na:
    df = data.copy() #copy of data set

    #Creating a variable which indicates 1 for a null value and 0 for all other

    df[feature] = np.where(df[feature].isnull(),1,0)

    #Creating mean Sales Price where information is missing

    df.groupby(feature)['Salary'].median().plot.bar()
    plt.title(feature)
    plt.ylabel('Salary')
    plt.show()
```





Here the relation between missing values and the dependant variable is clearly visible. So we need to replace these missing values with some meaningful values which we will do in the feature engineering time.

2.All the Numerical Variables

```
# Creating a list of numerical values

numerical_features = [feature for feature in data.columns if data[feature].dtype

print("The length of numerical variables: " ,len(numerical_features),'\n')

#display the numerical variables

data[numerical_features].head()
```

```
The length of numerical variables:  13
```

	Business	Age	Salary	Base_pay	Bonus	Unit_Price	Volume	openingbal
0	0	18	5089.00	2035.600	254.4500	3.77	21226600	
1	0	19	5698.12	2279.248	284.9060	3.74	10462800	
2	0	22	5896.65	2358.660	294.8325	3.89	18761000	
3	1	21	6125.12	2450.048	306.2560	4.35	66130600	
4	0	23	6245.00	2498.000	312.2500	4.34	26868200	

3.Distribution of Numerical Variables

3.1. Discrete Variables

```
#Numerical Variables are usually of two types - Continuous and discrete

discrete_feature = [feature for feature in numerical_features if len(data[featur

print("Discrete Variables Count: {}".format(len(discrete_feature)))
```

```
Discrete Variables Count: 1
```

```
discrete_feature
```

```
['Business']
```

```
data['Business'].nunique() # number of unique values in discrete column 'Busness
```

    2

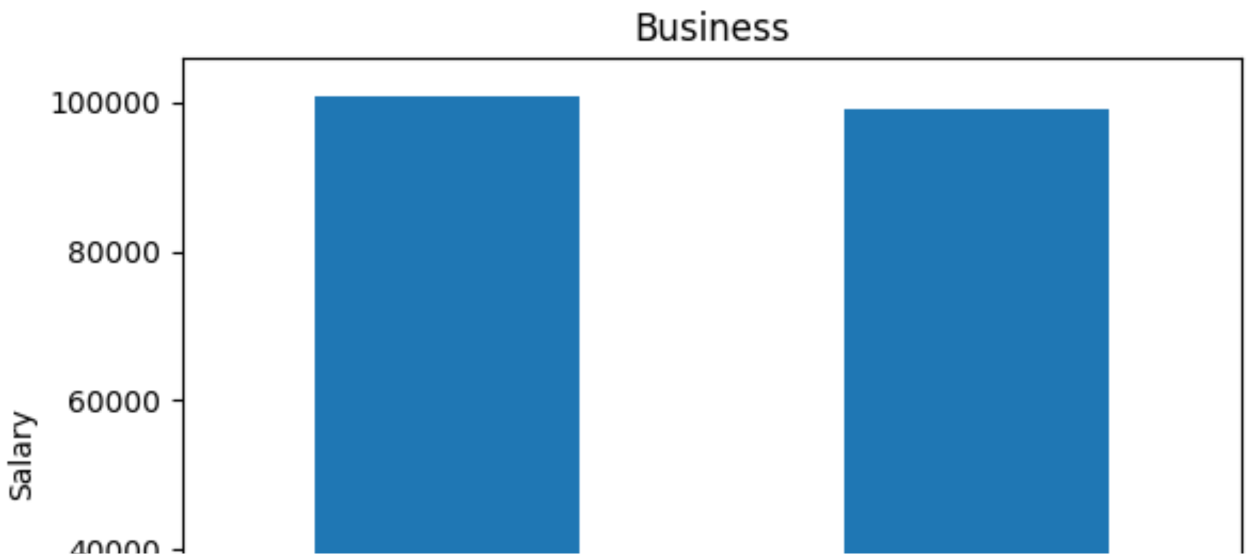
```
data['Business'].unique() # Print unique values
```

    array([0, 1])

BIVARIATE ANALYSIS

Bar Plot

```
#Finding Relationship with Discrete features and Salary

for feature in discrete_feature:
    df = data.copy()
    df.groupby(feature)['Salary'].median().plot.bar()
    plt.xlabel(feature)
    plt.ylabel('Salary')
    plt.title(feature)
    plt.show()
```



Business doesnot make any variation in salary distribution

3.2. Continuous Variables

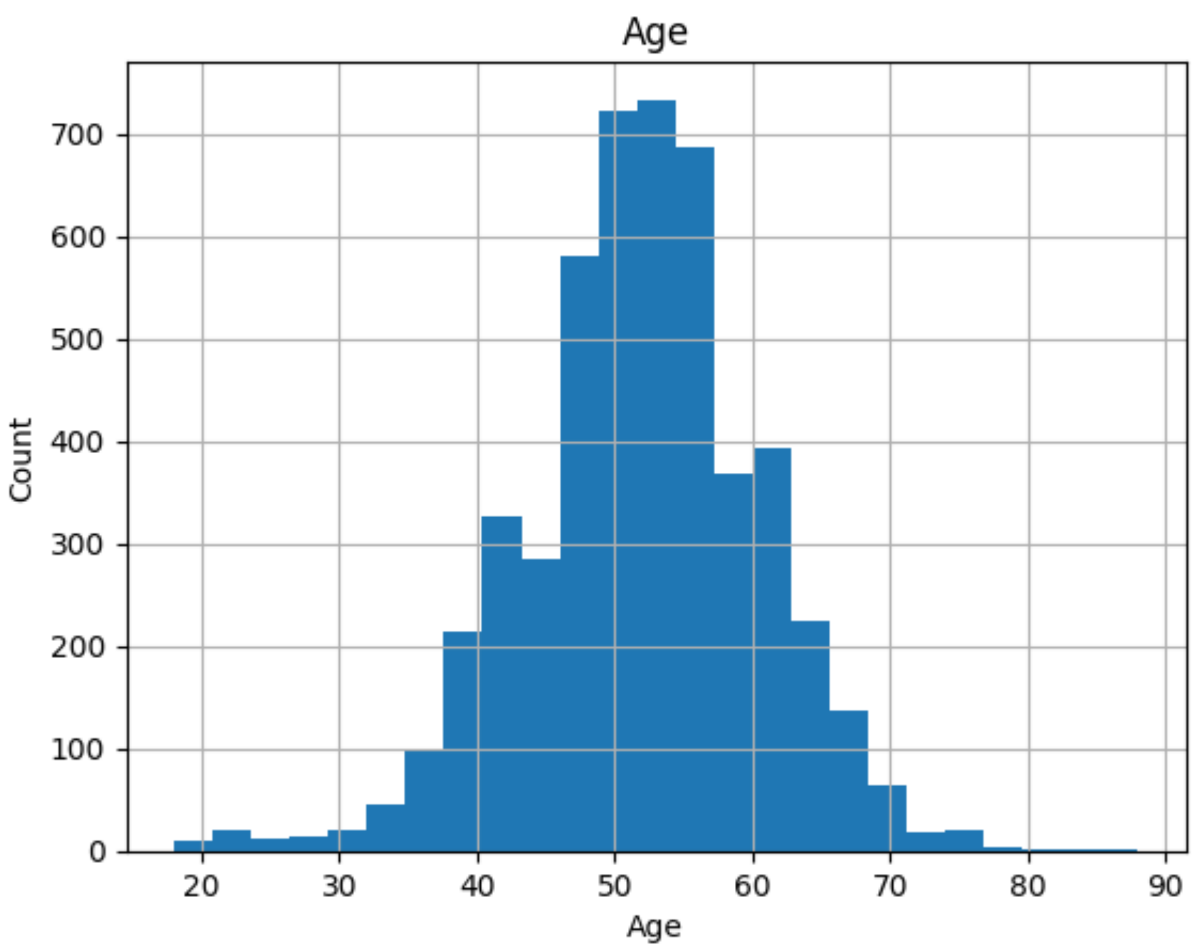
```
continuous_feature = [feature for feature in numerical_features if feature not i
print("Continuous feature Count {}".format(len(continuous_feature)))
```

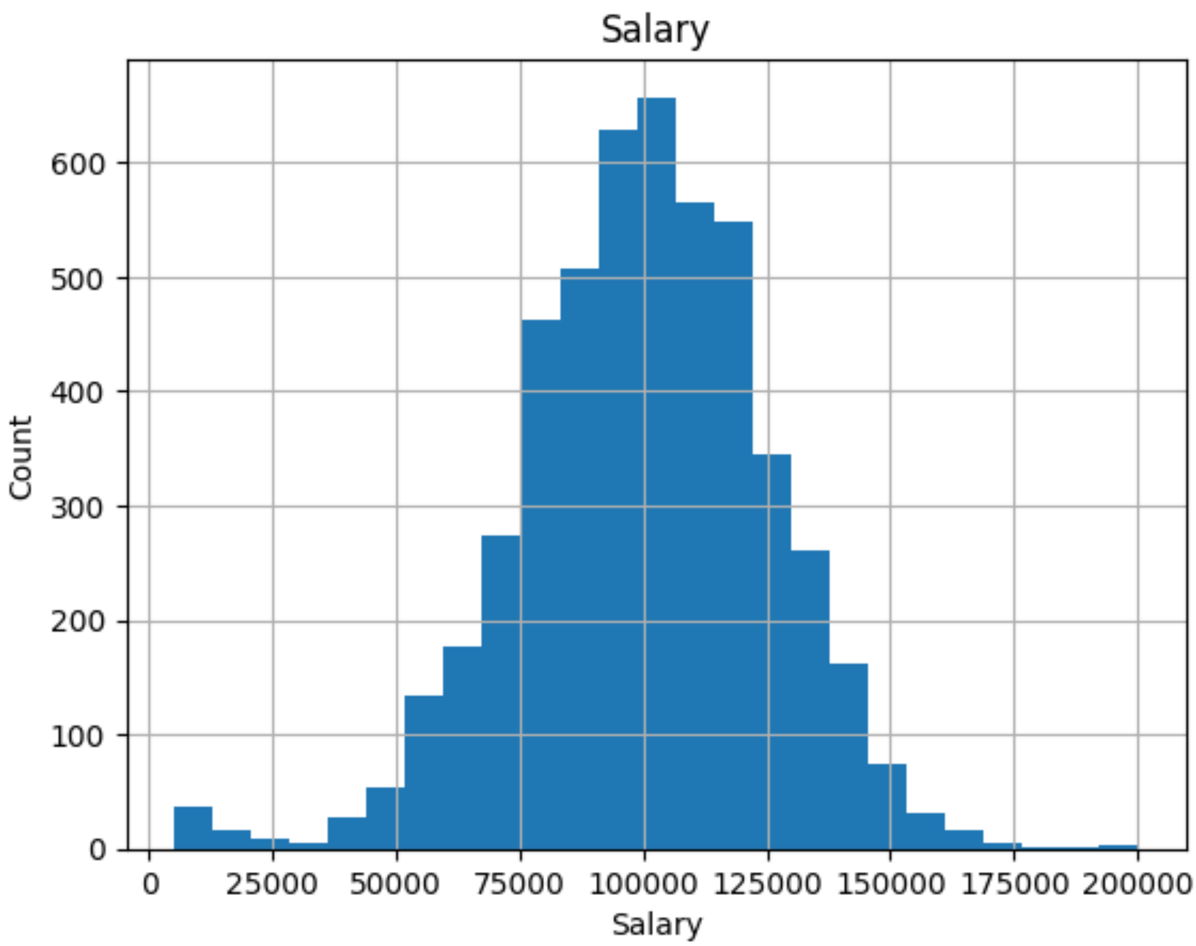
```
Continuous feature Count 12
```

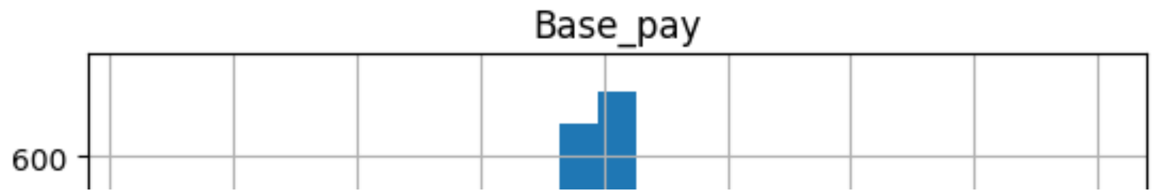
UNIVARIATE ANALYSIS

Histogram

```
#Analyzing the Distribution of Continuous variables

for feature in continuous_feature:
    df = data.copy()
    df[feature].hist(bins=25)
    plt.xlabel(feature)
    plt.ylabel('Count')
    plt.title(feature)
    plt.show()
```







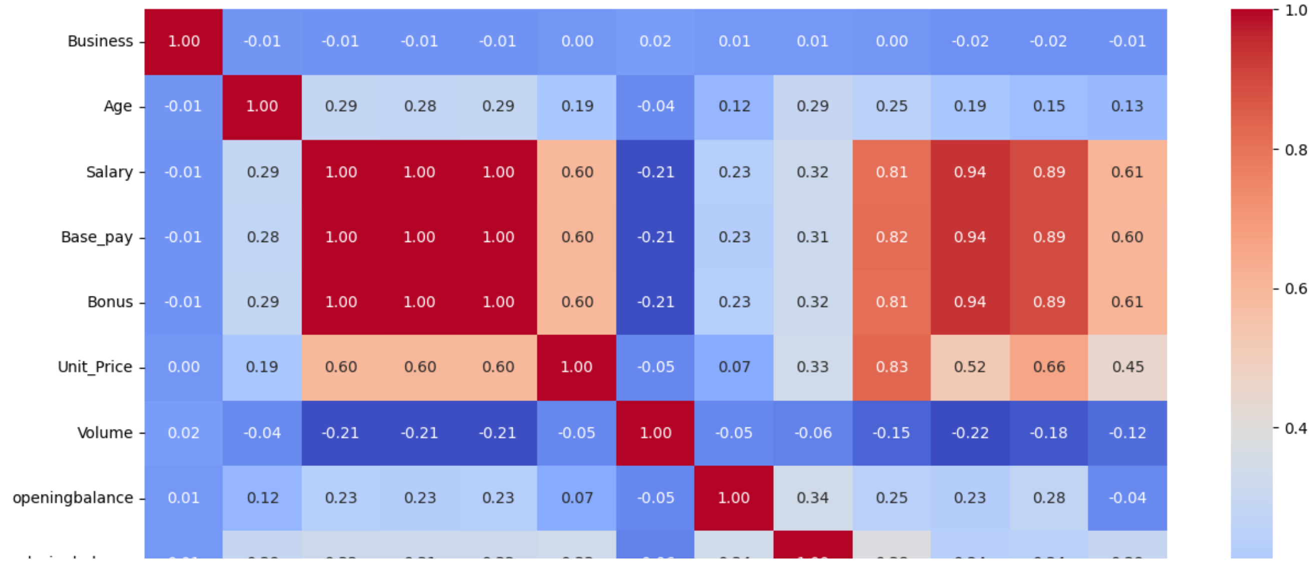
The majority of continuous features resemble a normal distribution, which means the data is symmetrically distributed around the mean, and the majority of the data points cluster near the center of the distribution. The exceptions are the variables "Months" ,"Volume" ,"Total_Sales",'Unit_Sales','low','closingbalance','openingbalance','Unit_Price' which are skewed in nature.Most of the employees work for lower range of unit price earound 150,volume of 2500000,opening and closing balance of 65,unit sales of 20 and then somewhat symmetrically range from 40 to 115,but most people shows very less total sales and number of employees gradually decreases as the sales value increases. work experience of number of employees varies assymetrically. Large number of people leave the job within 5 months and then the next range of people continue 6 years. Work duration of a minimum number distributes between 5 months to 5 years

BIVARIATE ANALYSIS

Heatmap

```
#Heat Map to see the correlation between the numerical features

plt.figure(figsize=(15, 10))
sns.heatmap(data[numerical_features].corr(),cmap='coolwarm',fmt='.2f',annot =Tru
```

```
<Axes: >
```

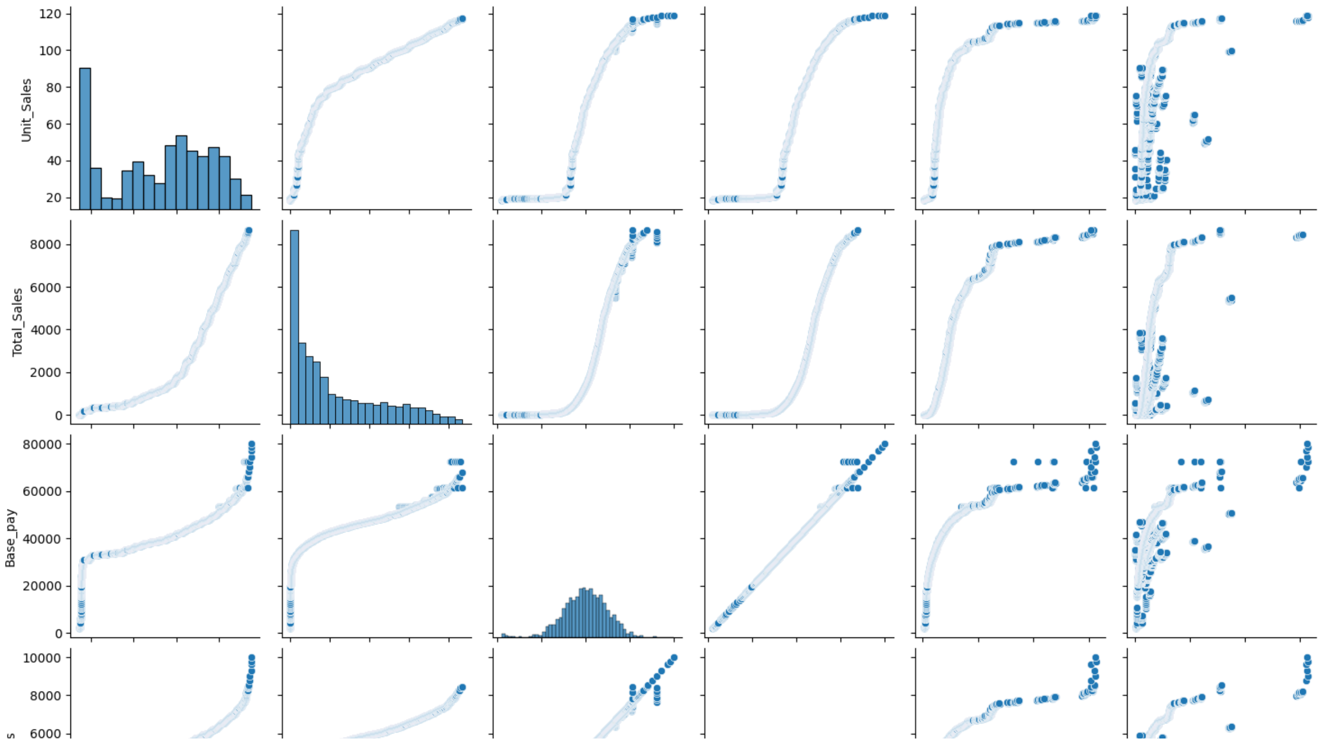


Values close to 1 indicate a strong positive correlation, meaning that when one variable increases, the other tends to increase as well. For example, Salary, Base_pay, and Bonus have high positive correlations with each other (close to 1) since they represent the same aspects of compensation. Values close to -1 indicate a strong negative correlation, meaning that when one variable increases, the other tends to decrease. For example, Unit_Price and Volume have a negative correlation, suggesting that higher unit prices are associated with lower volumes. Values close to 0 indicate a weak or no linear correlation between the variables. For instance, there is a weak correlation between Business and other variables, Age, Months, and low

Pairplot

```
#Pairplot to see the relation among Unit_Sales','Total_Sales','Base_pay','Bonus'

plt.figure(figsize=(15, 100))
sns.pairplot(data[['Unit_Sales','Total_Sales','Base_pay','Bonus','low','Unit_Pri
plt.show()
```

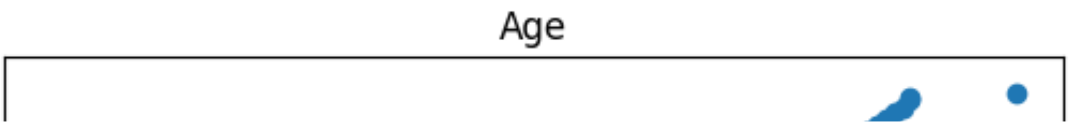
```
<Figure size 1500x10000 with 0 Axes>
```

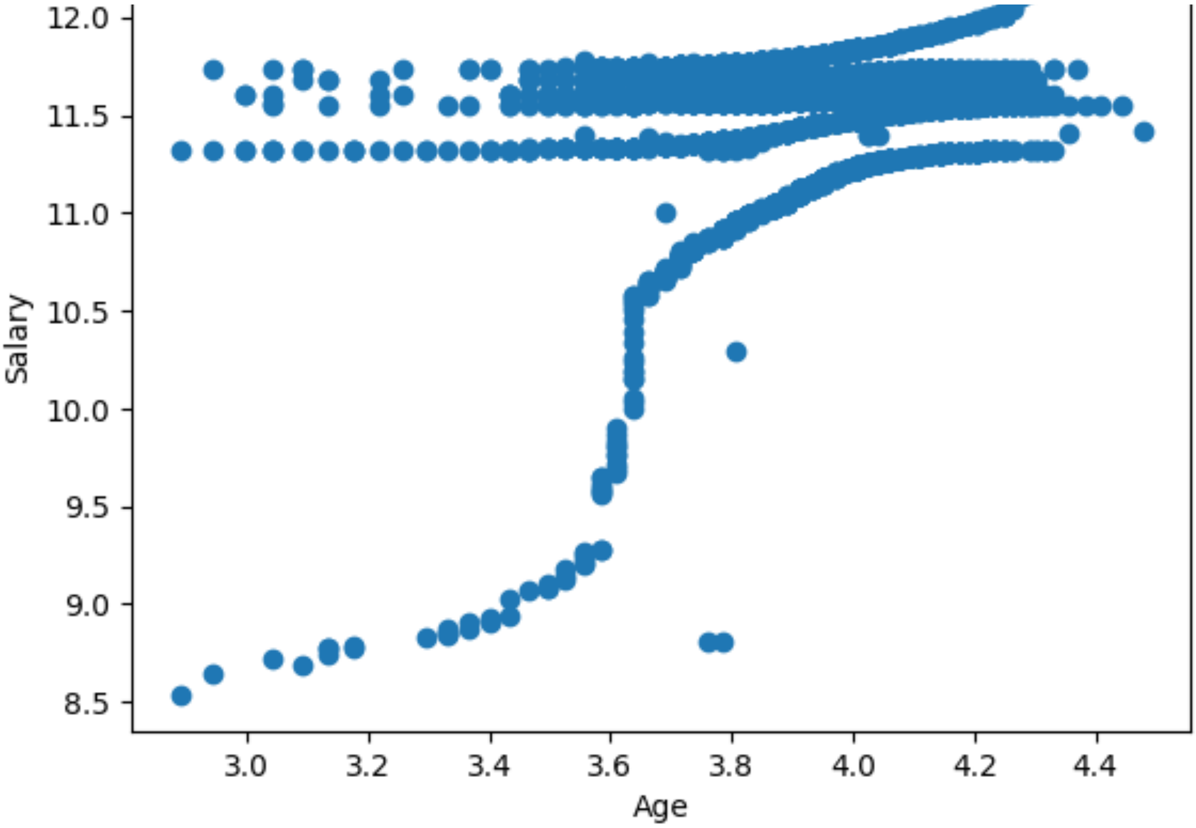


A linear correlation is found between the variables 'Unit_Sales', 'Total_Sales' ,'Base_pay' ,'Bonus' ,'low' and 'Unit_Price'.

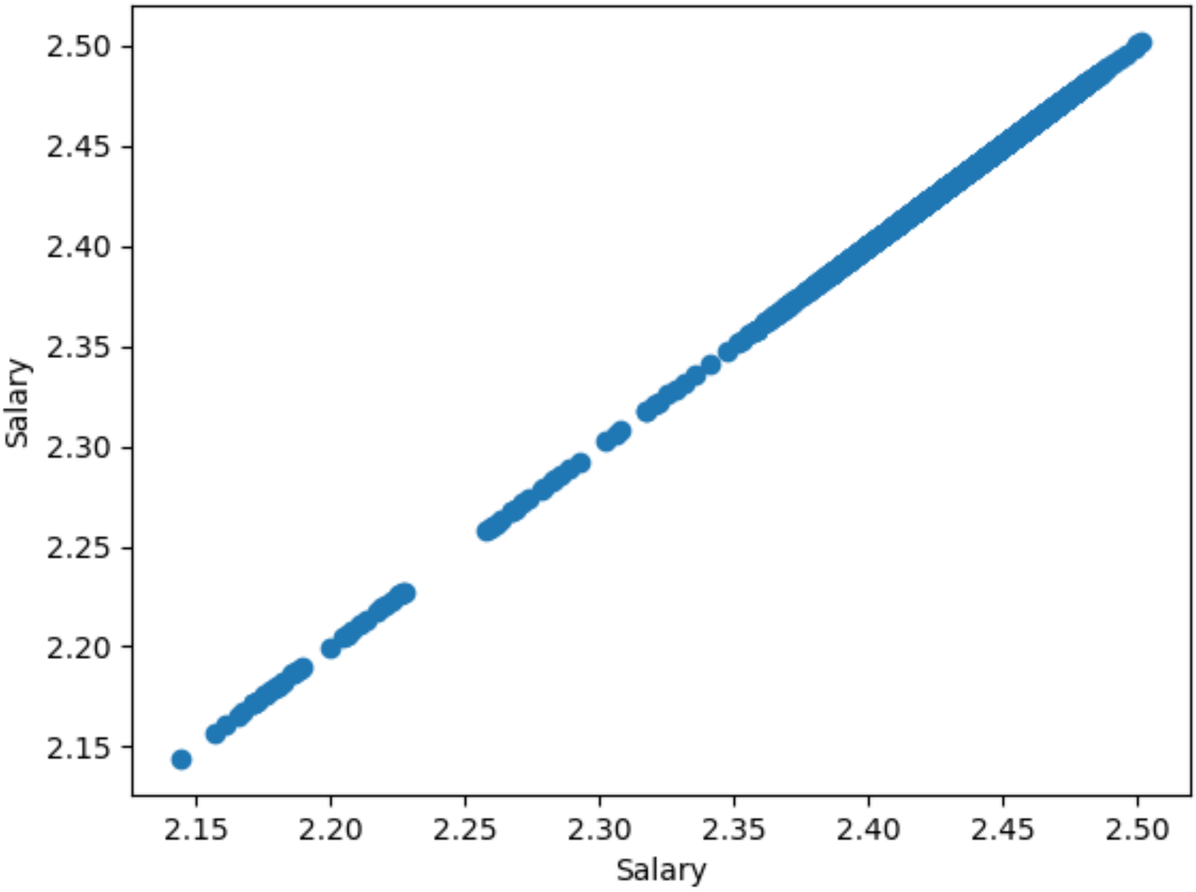
Scatter Plot

```
#To check the correlation between Countinuous features and the response variable

for feature in continuous_feature:
    df = data.copy()
    if 0 in df[feature].unique():
        pass
    else:
        df[feature] = np.log(df[feature])
        df['Salary'] = np.log(df['Salary'])
        plt.scatter(df[feature], df['Salary'])
        plt.xlabel(feature)
        plt.ylabel('Salary')
        plt.title(feature)
        plt.show()
```

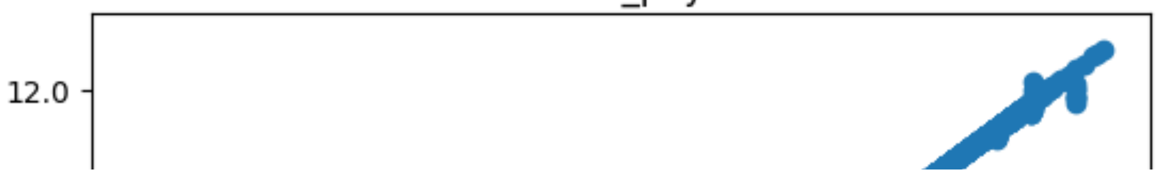




Salary

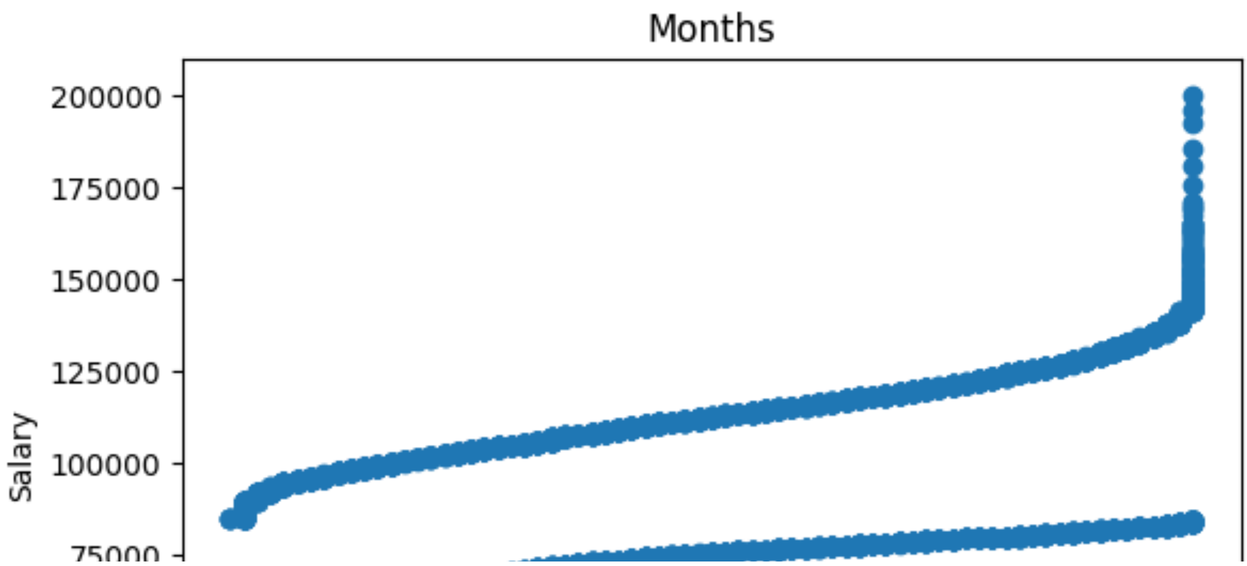


Base_pay



apply a logarithm transformation to the features and the 'Salary' column. Depending on your data and the distribution of values, applying a logarithm transformation can be helpful to visualize relationships more clearly, especially when dealing with data that has a wide range of values or a skewed distribution.["Months" ,"Volume" ,"Total_Sales",'Unit_Sales','low','closingbalance','openingbalance','Unit_Price'], Frm previous histplot its visible that age is normally distributed around 20 to 80 and in same manner salary increasearound 40s and highest salaried at 80s. Unit price is distributed maximum around 150. 'low','Unit_Sales',and 'Total_sales' shows somewhat linear relation. Lowest and highest salaries shows linear relation with closing and opening balances

```
# month contains zeros in values and hence seperately ploted without log transf

plt.scatter(df['Months'], df['Salary'])
plt.xlabel(feature)
plt.ylabel('Salary')
plt.title(feature)
plt.show()
```

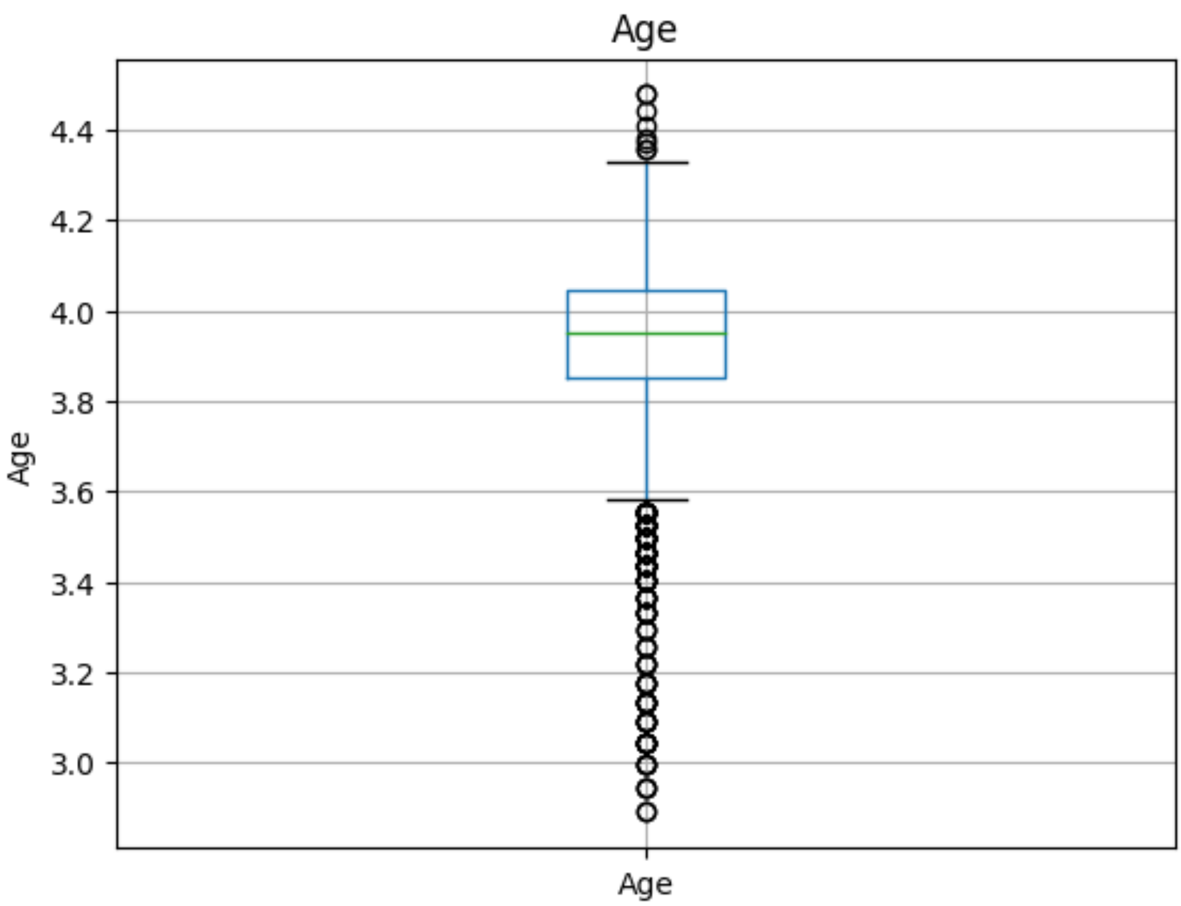


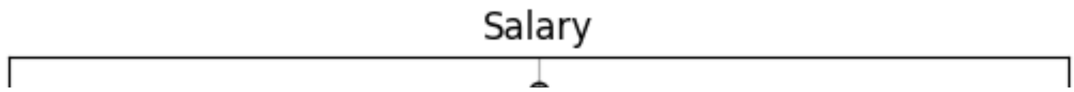
The numerical features "Bonus" , "Base_pay" show a clearly linear relationship with "Salary". There shows steep hike in salary upto 50000 and then some what stable till 1lak up to 5 years,but sudden hike for people 0f 6 years of experience.

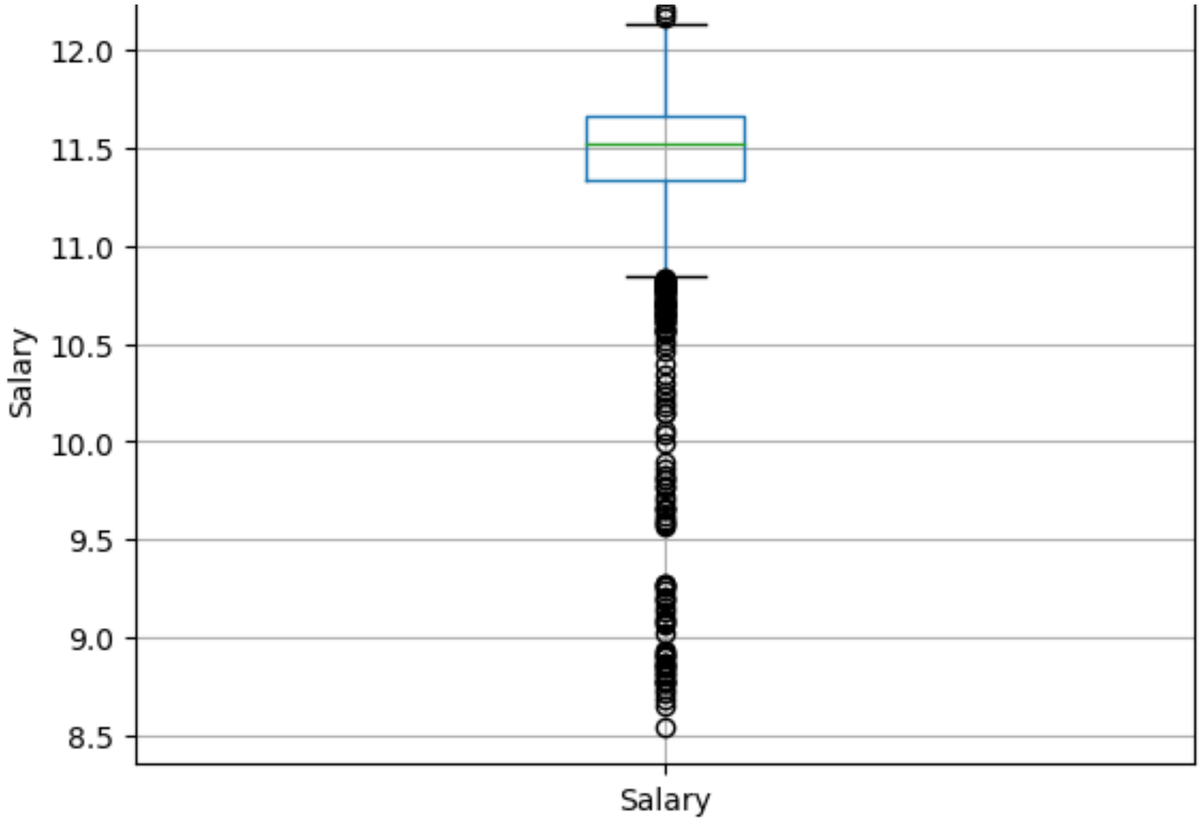
4.Outliers

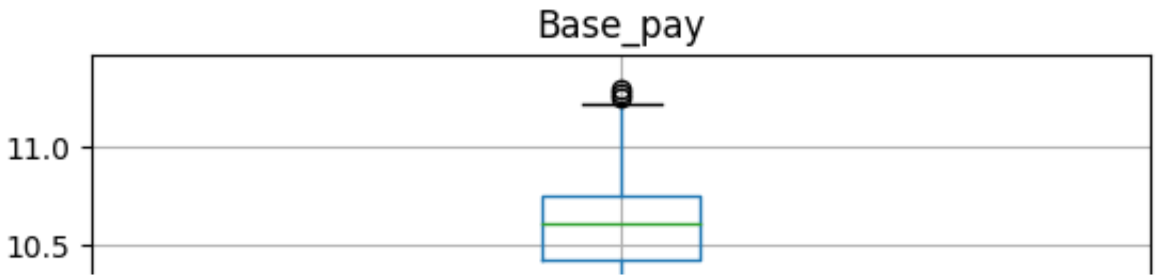
Box Plot

```
#Boxplot of the Continous Features

for feature in continuous_feature:
    df = data.copy()
    if 0 in df[feature].unique():
        pass
    else:
        df[feature] = np.log(df[feature])
        df.boxplot(column = feature)
        plt.ylabel(feature)
        plt.title(feature)
        plt.show()
```

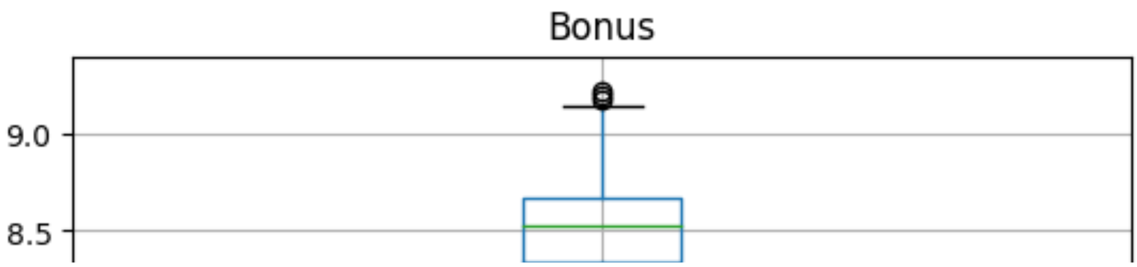




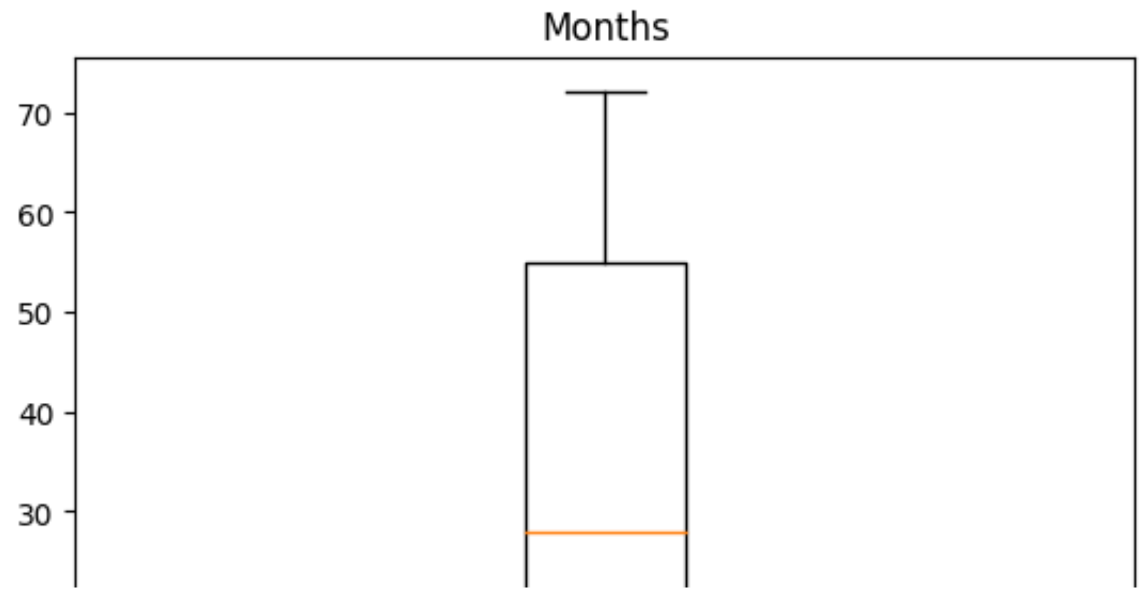




Apply a logarithm transformation to the features and the 'Salary' column. Depending on the data and the distribution of values, A boxplot is a useful way to display the distribution of data through its quartiles and potential outliers. By applying the logarithm transformation, you can better understand the distribution of skewed data and potential patterns or outliers.



```
# month contains zeros in values and hence seperately ploted without log transfo

plt.boxplot(data['Months'])
plt.title('Months')
plt.show()
```



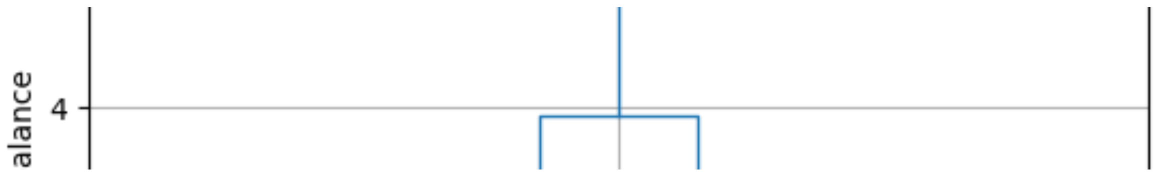
All of the variables in the boxplot clearly exhibit outliers, except for the columns "Total_Sales" ,"Unit_Sales" and 'Months'.

```
outliers=['Age','Salary','Base_pay','Bonus','Unit_Price','openingbalance','closi
```

5.Categorical Variables

```
categorical_features = [feature for feature in data.columns if data[feature].dty
categorical_features
```

```
['Gender', 'Dependancies', 'Calls', 'Type', 'Billing', 'Rating',
'Education']
```



```
print("Categorical feature Count {}".format(len(categorical_features)))
```

```
Categorical feature Count 7
```

```
data[categorical_features].head()
```

	Gender	Dependancies	Calls	Type	Billing	Rating	Education
0	Female	No	Yes	Month-to-month	No	Yes	High School or less
1	Female	No	Yes	Month-to-month	No	Yes	High School or less
2	Male	No	Yes	Month-to-month	Yes	No	High School or less
3	Female	No	Yes	Month-to-month	Yes	Yes	High School or less

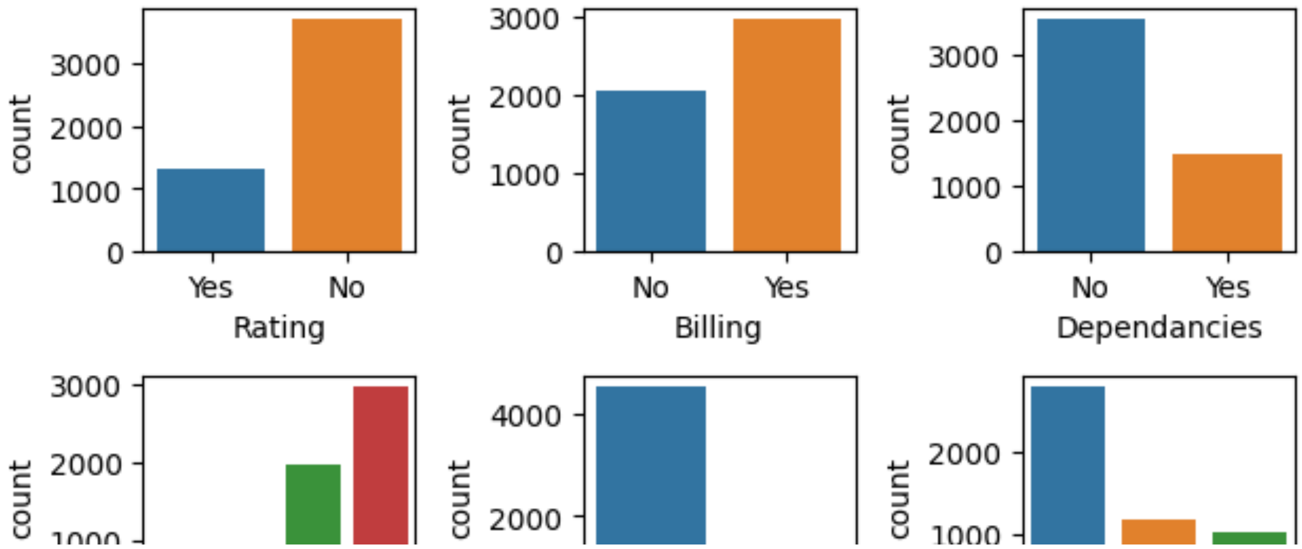
```
#Checking Cardinality

for feature in categorical_features:
    print("The feature is {} and number of labels are {}".format(feature,len(dat
```

```
The feature is Gender and number of labels are 2
The feature is Dependancies and number of labels are 2
The feature is Calls and number of labels are 2
The feature is Type and number of labels are 3
The feature is Billing and number of labels are 2
The feature is Rating and number of labels are 2
The feature is Education and number of labels are 4
```

UNIVARIATE ANALYSIS

Count Plot

```
#to plot the count of each category in the categorical variable

plt.subplot(2, 3, 1)
sns.countplot(x ='Rating', data = data)
plt.subplot(2, 3, 2)
sns.countplot(x ='Billing', data = data)
plt.subplot(2, 3, 3)
sns.countplot(x ='Dependancies', data = data)
plt.subplot(2, 3, 4)
sns.countplot(x ='Education', data = data)
plt.xticks(rotation=90)
plt.subplot(2, 3, 5)
sns.countplot(x ='Calls', data = data)
plt.subplot(2, 3, 6)
sns.countplot(x ='Type', data = data)
plt.xticks(rotation=90)
plt.tight_layout()
plt.show()
```

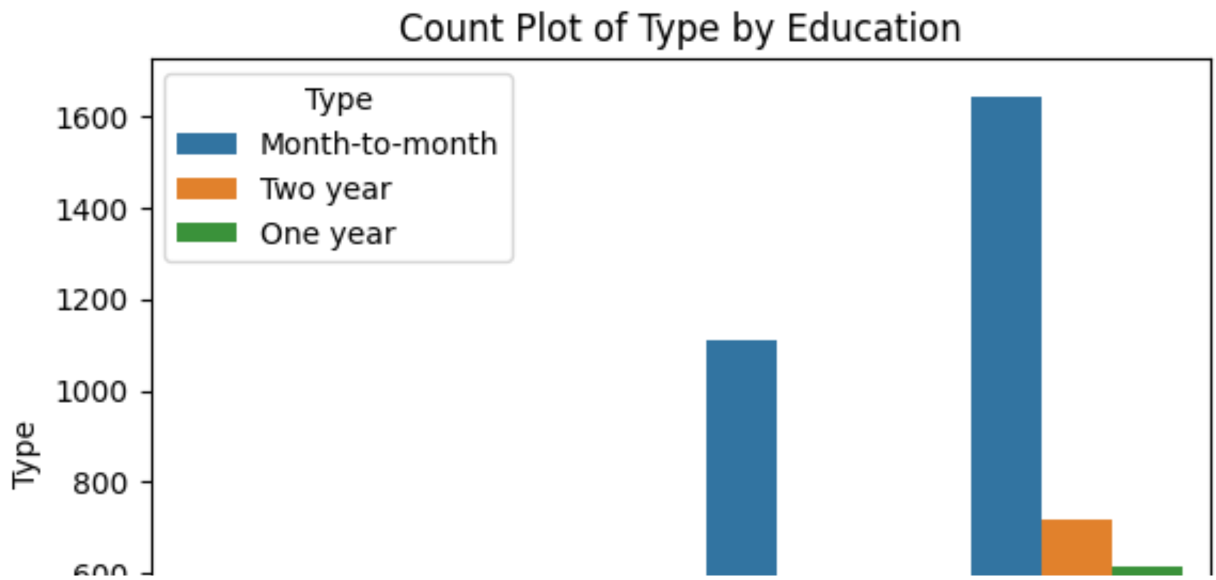


The countplot displays some class imbalance, indicating that the frequency of data points in the various categories is significantly different from one another.

BIVARIATE ANALYSIS

Count Plot

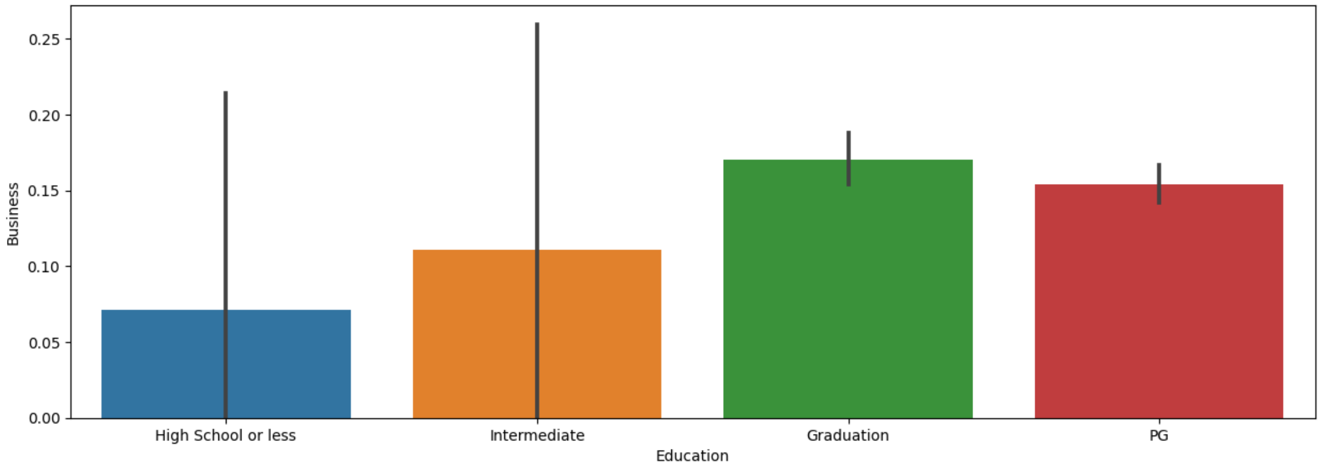
```
#Bivariate analysis of type with Education

sns.countplot(x=data['Education'],hue=data['Type'])
plt.xlabel('Education')  # Label for the x-axis

plt.title('Count Plot of Type by Education')  # Title for the plot
plt.show()
```

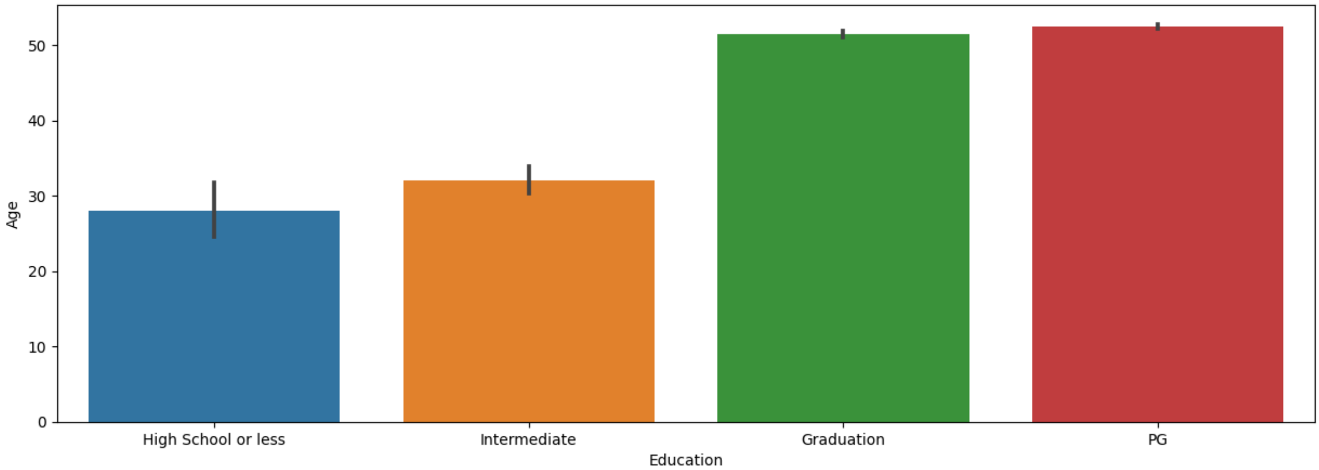


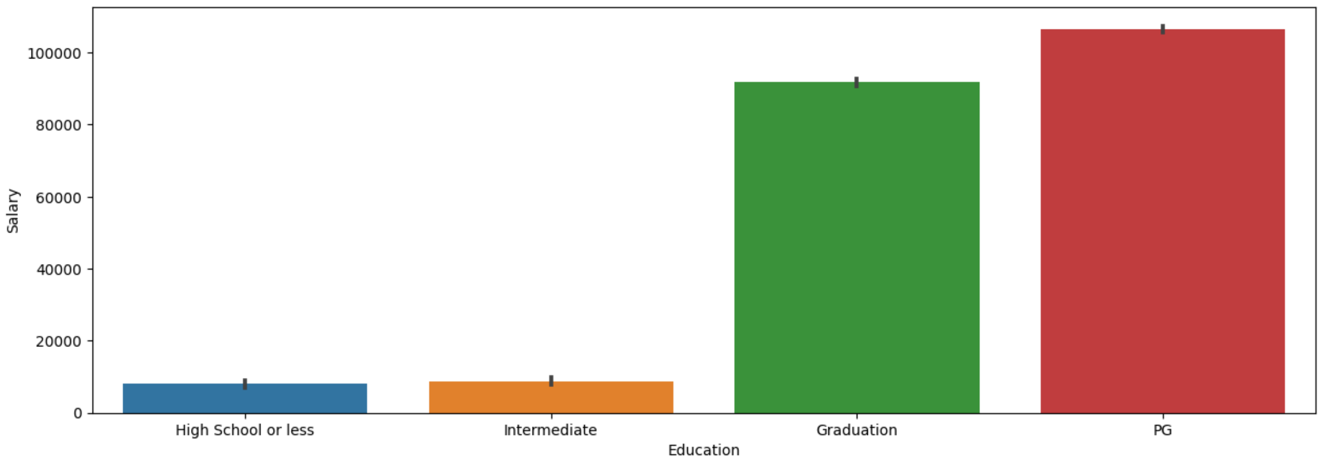
In this graph which plot education by type shows more post gradute employees prefer to work in month to month contract other than yearly basis.month-to-month contracts offer greater flexibility to postgraduate employees. They can easily transition between jobs, take breaks between contracts, or pursue further studies without being tied to long-term commitments.Postgraduate employees might prefer month-to-month contracts to avoid extended trial periods before becoming regular employees.

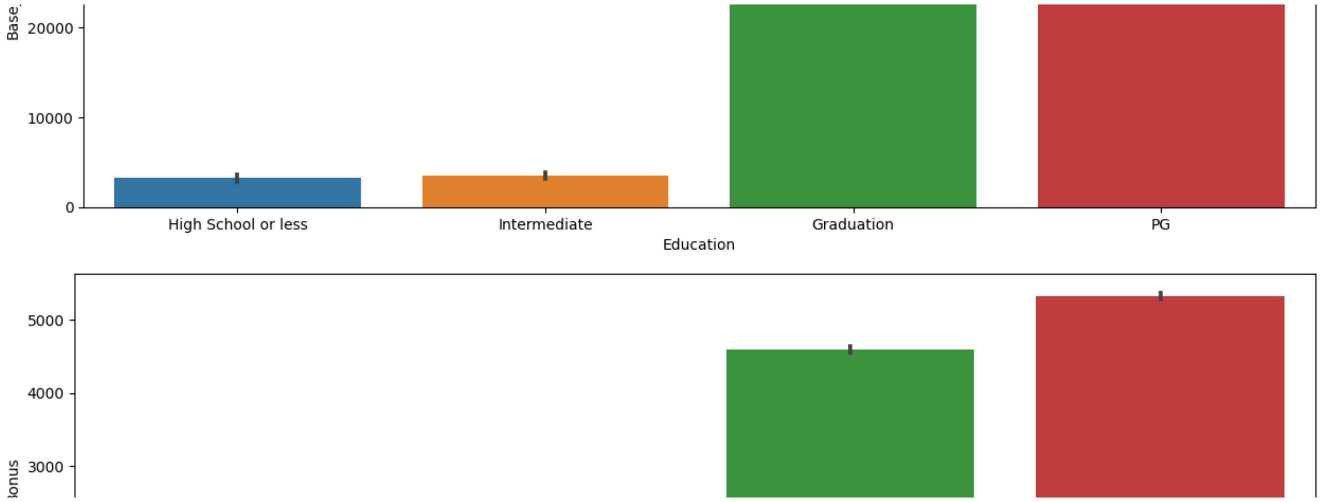
Bar Plot

```
# Analyzing the relationship between numerical varibales and Education

for feature in numerical_features:
  plt.figure(figsize=(15,5))
  sns.barplot(x=data['Education'],y=data[feature])
```

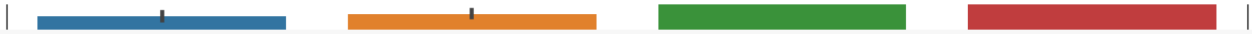




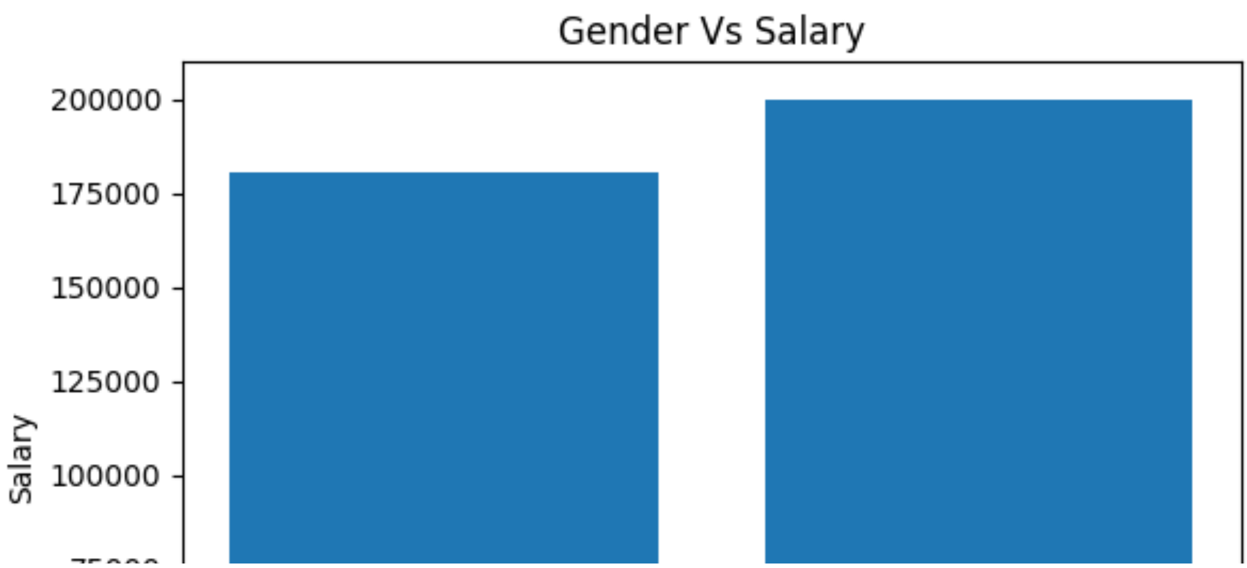




All bivariate analysis show that performance of postgraduate employees have significant importance on most of the numerical features.



```
# Bivariate analysis of Salary and Gender

plt.bar(data['Gender'],data['Salary'])
plt.xlabel('Gender')  # Label for the x-axis
plt.ylabel('Salary')  # Label for the y-axis
plt.title('Gender Vs Salary')  # Title for the plot
plt.show()
```



Compared to female employees, male employees earn higher salaries.Studies have shown that women tend to negotiate salary less frequently than men, which can lead to disparities in initial salary offers and future pay increases.Career interruptions, such as taking time off for caregiving or family responsibilities, can impact the accumulation of work experience and career progression, potentially leading to lower salaries.

2. Data Preprocessing

1. Missing Value Handling
2. Outlier Handling
3. Encoding
4. Feature Scaling
5. Dimensionality reduction
6. Feature Selection

1.Missing Value Handling

```
#features with missing values

features_with_na
```

```
['Base_pay', 'openingbalance', 'Total_Sales']
```

Base pay has normal distribution and hence as the first observation we can use mean to replace the missing values. As we proceed with mean or median filling the values does not fit with the data thrend in the set. So needsome other means of filling

other two shows skewed distribution from the univariate plot. Median can be used to replace missing values.

```
# Index pointsof missing values in 'Base_Pay'

data[data['Base_pay'].isnull()].index.tolist()
```

```
[124,
 125,
 126,
 127,
 128,
 129,
 130,
 131,
 132,
 133,
 134,
 135,
 136,
 137,
 138,
 139,
 140,
 141,
 142,
 143,
 144,
 145,
 146]
```

```
col_select = [8, 9, 10]    #Specify indices of columns to select  ['Salary','Bas
# Get multiple columns into a new data frame
data_new2 = data.iloc[110:155, col_select]   #using slicing to select rows 110 t
print(data_new2)
```

```
          Salary     Base_pay        Bonus
110  46970.19120  18788.07648  2348.509560
111  47243.09710  18897.23884  2362.154855
112  47608.34851  19043.33940  2380.417426
113  47621.52857  19048.61143  2381.076429
114  47647.35545  19058.94218  2382.367773
115  47695.00799  19078.00320  2384.750400
116  48071.75632  19228.70253  2403.587816
117  48081.73044  19232.69218  2404.086522
118  48161.38047  19264.55219  2408.069024
119  48385.07026  19354.02810  2419.253513
120  48390.39438  19356.15775  2419.519719
121  48452.77942  19381.11177  2422.638971
122  48811.05126  19524.42051  2440.552563
123  48892.60518  19557.04207  2444.630259
124  49076.09704          NaN  2453.804852
125  49294.09553          NaN  2464.704777
126  49346.69135          NaN  2467.334568
127  49359.76469          NaN  2467.988235
128  49372.31057          NaN  2468.615529
129  49440.90658          NaN  2472.045329
130  49492.58316          NaN  2474.629158
131  49513.92593          NaN  2475.696297
132  49522.33256          NaN  2476.116628
133  49599.97686          NaN  2479.998843
134  49620.03534          NaN  2481.001767
135  49722.22242          NaN  2486.111121
136  49949.11079          NaN  2497.455540
137  50098.84397          NaN  2504.942199
138  50405.21287          NaN  2520.260644
139  50409.97121          NaN  2520.498561
140  50806.60095          NaN  2540.330048
141  50938.79543          NaN  2546.939772
142  51076.89690          NaN  2553.844845
143  51080.49092          NaN  2554.024546
144  51087.80513          NaN  2554.390257
145  51338.70016          NaN  2566.935008
146  51601.62583          NaN  2580.081292
147  51641.14375  20656.45750  2582.057188
148  51815.12953  20726.05181  2590.756477
149  51976.94617  20790.77847  2598.847309
150  51994.43388  20797.77355  2599.721694
151  52103.81853  20841.52741  2605.190927
152  52130.15956  20852.06383  2606.507978
153  52146.45868  20858.58347  2607.322934
154  52260.97204  20904.38882  2613.048602
```

All continuous 23 values are missing from index 124 to 146:

```
x=data['Base_pay'].mean()
print('Mean of base pay : ',x)
y=data['Base_pay'].median()
print('Median of base pay : ',y)
```

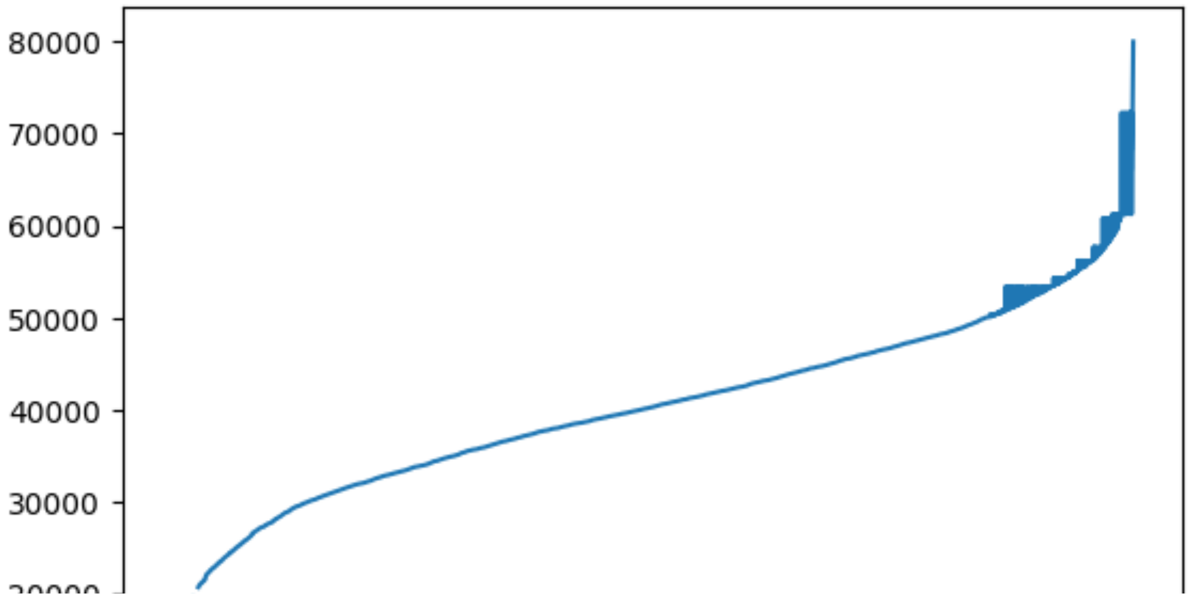
```
    Mean of base pay :  40046.1877067818
    Median of base pay :  40282.01604
```

Both values are above 40000

Line Graph

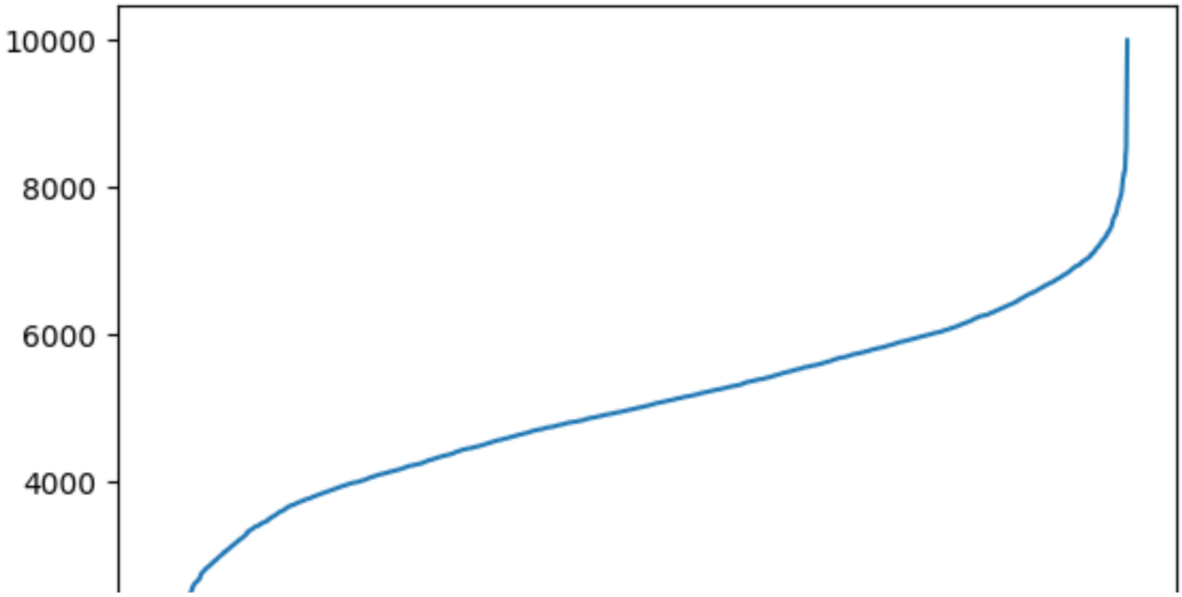
```
# Plot thedistribution of 'Base_Pay'

plotdata=pd.DataFrame(data)

plotdata['Base_pay'].plot(kind='line')
plt.xticks(rotation=90)
```

```
(array([-1000.,     0.,  1000.,  2000.,  3000.,  4000.,  5000.,  6000.]),
 [Text(-1000.0, 0, '-1000'),
  Text(0.0, 0, '0'),
  Text(1000.0, 0, '1000'),
  Text(2000.0, 0, '2000'),
  Text(3000.0, 0, '3000'),
  Text(4000.0, 0, '4000'),
  Text(5000.0, 0, '5000'),
  Text(6000.0, 0, '6000')])
```



```
# Plot thedistribution of 'Bonus'

plotdata=pd.DataFrame(data)

plotdata['Bonus'].plot(kind='line')
plt.xticks(rotation=90)
```

```
(array([-1000.,     0.,  1000.,  2000.,  3000.,  4000.,  5000.,  6000.]),
 [Text(-1000.0, 0, '-1000'),
  Text(0.0, 0, '0'),
  Text(1000.0, 0, '1000'),
  Text(2000.0, 0, '2000'),
  Text(3000.0, 0, '3000'),
  Text(4000.0, 0, '4000'),
  Text(5000.0, 0, '5000'),
  Text(6000.0, 0, '6000')])
```



Both are showinh an increasing trend in its values

1.1. Forward and Backward Fill

```
# Lets explain the reason for not proceeding with mean or median fill

df=data.copy()
df['sum'] = df['Bonus'] + df['Base_pay']  # New column formed with sum of other
```

```
col_select = [8, 9, 10, 20]  # Specify indices of columns to select

data_new2 = df.iloc[110:155, col_select]    # Get multiple columns using index s
print(data_new2)
```

```
          Salary     Base_pay        Bonus           sum
110  46970.19120  18788.07648  2348.509560  21136.586040
111  47243.09710  18897.23884  2362.154855  21259.393695
112  47608.34851  19043.33940  2380.417426  21423.756826
113  47621.52857  19048.61143  2381.076429  21429.687859
114  47647.35545  19058.94218  2382.367773  21441.309953
115  47695.00799  19078.00320  2384.750400  21462.753600
116  48071.75632  19228.70253  2403.587816  21632.290346
117  48081.73044  19232.69218  2404.086522  21636.778702
118  48161.38047  19264.55219  2408.069024  21672.621214
119  48385.07026  19354.02810  2419.253513  21773.281613
120  48390.39438  19356.15775  2419.519719  21775.677469
121  48452.77942  19381.11177  2422.638971  21803.750741
122  48811.05126  19524.42051  2440.552563  21964.973073
123  48892.60518  19557.04207  2444.630259  22001.672329
124  49076.09704          NaN  2453.804852           NaN
125  49294.09553          NaN  2464.704777           NaN
126  49346.69135          NaN  2467.334568           NaN
127  49359.76469          NaN  2467.988235           NaN
128  49372.31057          NaN  2468.615529           NaN
129  49440.90658          NaN  2472.045329           NaN
130  49492.58316          NaN  2474.629158           NaN
131  49513.92593          NaN  2475.696297           NaN
132  49522.33256          NaN  2476.116628           NaN
133  49599.97686          NaN  2479.998843           NaN
134  49620.03534          NaN  2481.001767           NaN
135  49722.22242          NaN  2486.111121           NaN
136  49949.11079          NaN  2497.455540           NaN
137  50098.84397          NaN  2504.942199           NaN
138  50405.21287          NaN  2520.260644           NaN
139  50409.97121          NaN  2520.498561           NaN
140  50806.60095          NaN  2540.330048           NaN
141  50938.79543          NaN  2546.939772           NaN
142  51076.89690          NaN  2553.844845           NaN
143  51080.49092          NaN  2554.024546           NaN
144  51087.80513          NaN  2554.390257           NaN
145  51338.70016          NaN  2566.935008           NaN
146  51601.62583          NaN  2580.081292           NaN
147  51641.14375  20656.45750  2582.057188  23238.514688
148  51815.12953  20726.05181  2590.756477  23316.808287
149  51976.94617  20790.77847  2598.847309  23389.625779
150  51994.43388  20797.77355  2599.721694  23397.495244
151  52103.81853  20841.52741  2605.190927  23446.718337
152  52130.15956  20852.06383  2606.507978  23458.571808
153  52146.45868  20858.58347  2607.322934  23465.906404
154  52260.97204  20904.38882  2613.048602  23517.437422
```

If we replace the missing values with mean or median then the sum will be above 40000 and the final salary will be above the actual. as shown the expected sum is to be somewhere around 22 to 23000. "Due to the continuous increasing trend in the 'Base_pay' and 'Bonus' column and the strong positive correlation with 'Salary' , Forward and Backward Filling is chosen to fill the 23 missing values from row 124 to 146, instead of using the median."

```
# Forward-fill missing values from index 124 to 135
data['Base_pay'] = data['Base_pay'].fillna(method='ffill', limit=12)

# Backward-fill missing values from index 136 to 146
data['Base_pay'] = data['Base_pay'].fillna(method='bfill', limit=11)
```

```
base_pay_subset_new = data.loc[120:150, 'Base_pay']
print(base_pay_subset_new)
```

```
120    19356.15775
121    19381.11177
122    19524.42051
123    19557.04207
124    19557.04207
125    19557.04207
126    19557.04207
127    19557.04207
128    19557.04207
129    19557.04207
130    19557.04207
131    19557.04207
132    19557.04207
133    19557.04207
134    19557.04207
135    19557.04207
136    20656.45750
137    20656.45750
138    20656.45750
139    20656.45750
140    20656.45750
141    20656.45750
142    20656.45750
143    20656.45750
144    20656.45750
145    20656.45750
146    20656.45750
147    20656.45750
148    20726.05181
149    20790.77847
150    20797.77355
Name: Base_pay, dtype: float64
```

Based on exploratory data analysis (EDA), it is evident that the two continuous numerical features 'openingbalance'and 'Total_Sales' contain missing values,and in addition to that, they also have outliers. Therefore, to address these missing values, median is chosen as the appropriate measure for filling in these three columns, considering the presence of outliers in the data.

1.2. Median Imputation

```
# Median filling

for feature  in ['openingbalance','Total_Sales']:
    data[feature ]=data[feature ].fillna(data[feature ].median())
```

```
data.isnull().sum()
```

```
Gender            0
Business          0
Dependancies      0
Calls             0
Type              0
Billing           0
Rating            0
Age               0
Salary            0
Base_pay          0
Bonus             0
Unit_Price        0
Volume            0
openingbalance    0
closingbalance    0
low               0
Unit_Sales        0
Total_Sales       0
Months            0
Education         0
dtype: int64
```

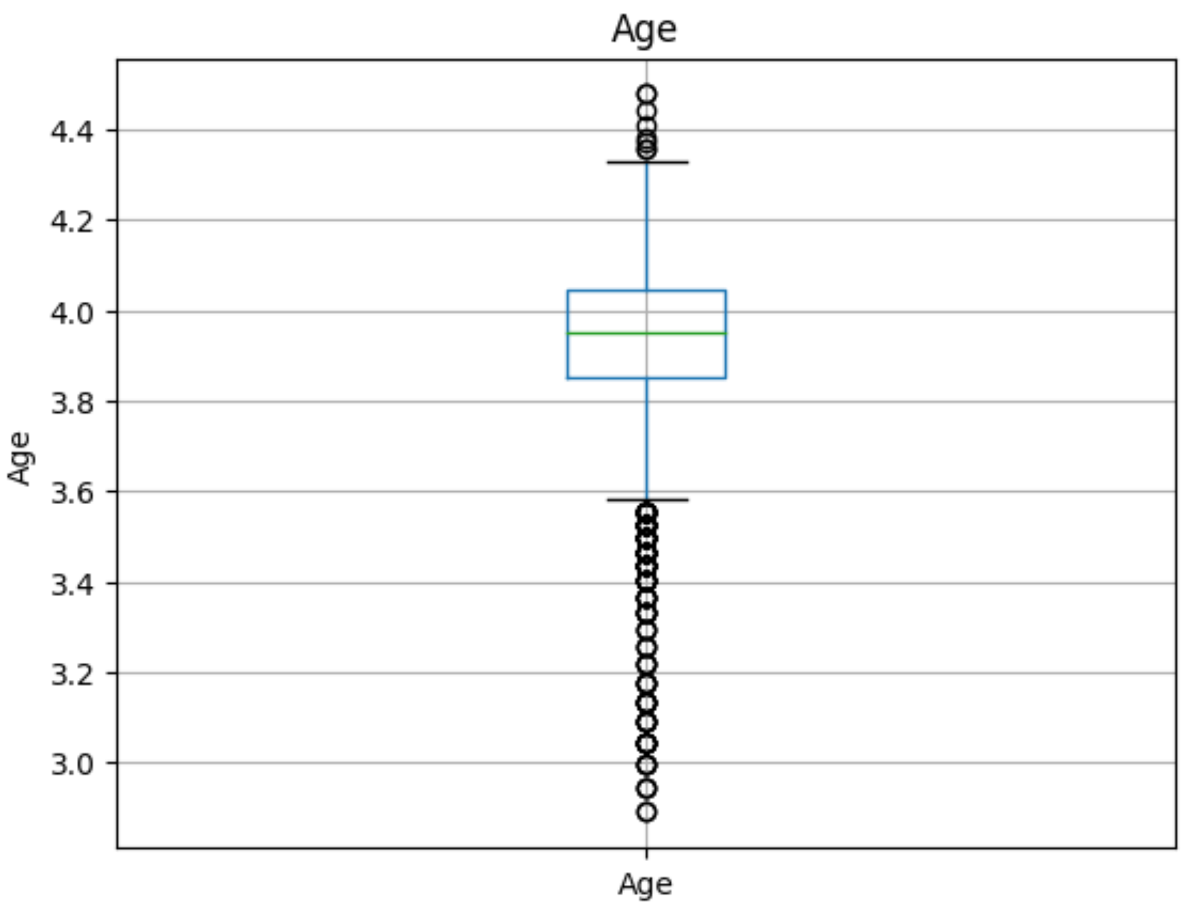
Continuous numerical features with missing values are filled by median and forward filling since the mean filling can't be used as there are outliers or extreme values with these columns.

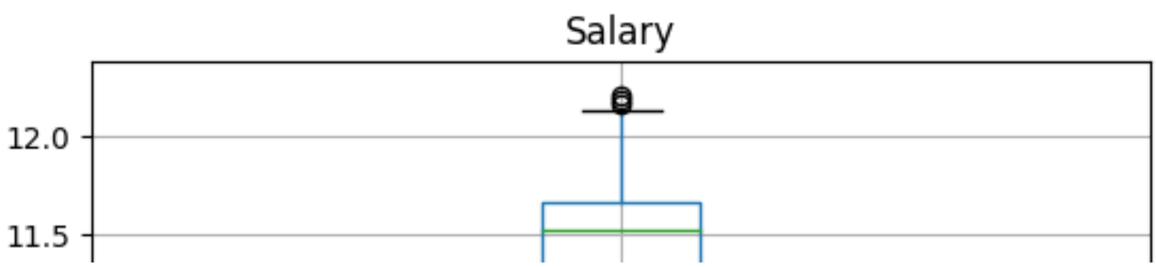
2.Outlier Handling

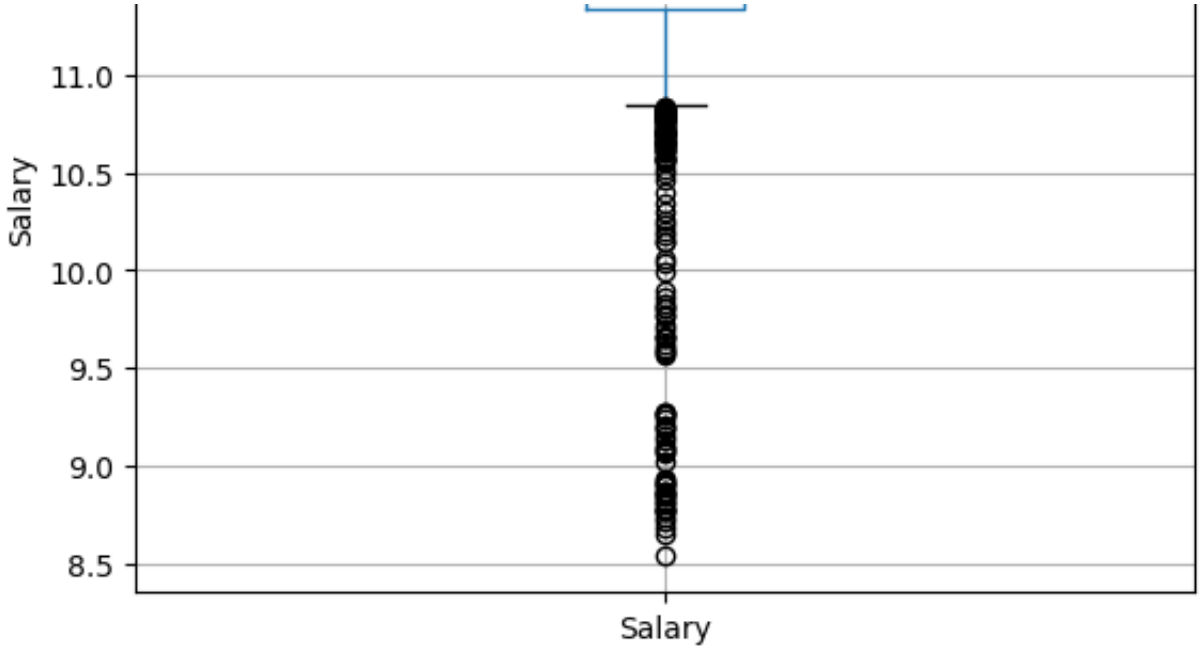
Outliers present in 'Age,' 'Salary,' 'Base_pay,' 'Bonus,' 'Unit_Price,' 'openingbalance,' 'closingbalance,' and 'low' columns

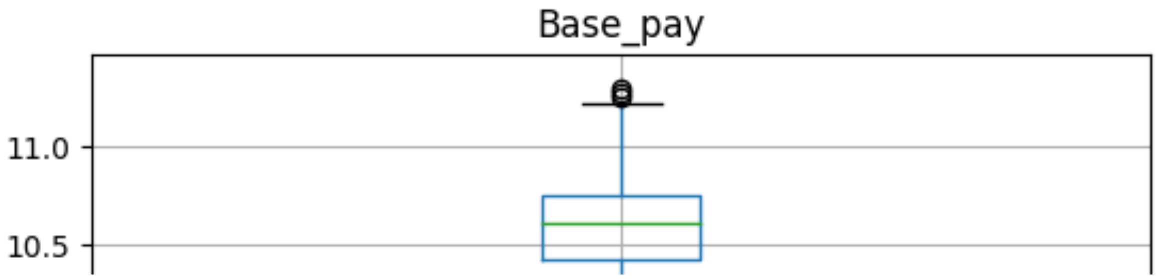
Box Plot

```
# Box plot using log transform for outlier detection after filling null values

for feature in continuous_feature:
    df = data.copy()
    if 0 in df[feature].unique():
        pass
    else:
        df[feature] = np.log(df[feature])
        df.boxplot(column = feature)
        plt.ylabel(feature)
        plt.title(feature)
        plt.show()
```









Majority of outliers in the Age,BasePay, Salary, and Bonus column lies below the lower limit but all are symmetrically distributed. For other columns too shows almost similar pattern of outlier distribution above and below the limits

For Opening balance outliers seems increased both sides after filling null values.

IQR Method

```
outliers=['Age','Salary','Base_pay','Bonus','Unit_Price','openingbalance','closi

 #Outliers are seen above and below the limit

for feature in outliers:
  Q1 = data[feature].quantile(0.25)
  Q2 = data[feature].quantile(0.50)
  Q3 = data[feature].quantile(0.75)
  print('\n',feature,'\n')
  print('Q1  25 % value =',Q1)
  print('Q2  50 % value =',Q2)
  print('Q3  75 % value =',Q3)
  IQR=Q3-Q1
  print('IQR=',IQR)
  up_lim=Q3+1.5*IQR
  low_lim=Q1-1.5*IQR
```

```
print('\nUpper limit=',up_lim)
print('Lower limit=',low_lim)
```

```
 Age

Q1  25 % value = 47.0
Q2  50 % value = 52.0
Q3  75 % value = 57.0
IQR= 10.0

Upper limit= 72.0
Lower limit= 32.0

 Salary

Q1  25 % value = 83890.33898
Q2  50 % value = 100579.37849999999
Q3  75 % value = 116912.092475
IQR= 33021.753495

Upper limit= 166444.7227175
Lower limit= 34357.708737500005

 Base_pay

Q1  25 % value = 33556.1355875
Q2  50 % value = 40231.751415
Q3  75 % value = 46764.836975
IQR= 13208.701387499998

Upper limit= 66577.88905624999
Lower limit= 13743.083506250005

 Bonus

Q1  25 % value = 4194.5169495
Q2  50 % value = 5028.968925
Q3  75 % value = 5845.6046237499995
IQR= 1651.0876742499995

Upper limit= 8322.236135124998
Lower limit= 1717.8854381250007

 Unit_Price

Q1  25 % value = 25.72749975
Q2  50 % value = 39.205
Q3  75 % value = 58.71500025
IQR= 32.9875005

Upper limit= 108.196251
Lower limit= -23.753751

 openingbalance
```

```
openingbalance

Q1  25 % value = 26.397632889999997
Q2  50 % value = 33.119999
Q3  75 % value = 42.525000250000005
IQR= 16.127367360000008

Upper limit= 66.71605129000002
Lower limit= 2.206581849999985
```

Upper and lower limits of outlairs found using IQR method. But handling outliers at this limits may affect our model as we have people with age range 18 to 88, contract period ranging from month to 2 years, educational qualification with PG and even below high school level, work duration from even without completing a month to 6 years of experience in this organisation. So this extreme high and low ranges may lead to create outlier values in bonus, base pay, sales figures and salary too. Since these extreme range of employees are genuine and facts handling outliers at the above mentioned limits is not a better way

```
# To check the number of unique values present in features with outliers

for feature in outliers:
    x=data[feature].nunique()
    print('\n number of unique values in ',feature,'is ',x)
```

```
 number of unique values in  Age is  65

 number of unique values in  Salary is  5000

 number of unique values in  Base_pay is  4883

 number of unique values in  Bonus is  5000

 number of unique values in  Unit_Price is  3836

 number of unique values in  openingbalance is  2986

 number of unique values in  closingbalance is  4011

 number of unique values in  low is  4014
```

```
# Sort and list out the values in the colums showing outliers

list_age = data["Age"].values.tolist()
list_age.sort()
print('age:',list_age)
list_Base_pay=data["Base_pay"].values.tolist()
list_Base_pay.sort()
print('base pay:',list_Base_pay)
list_Bonus=data["Bonus"].values.tolist()
list_Bonus.sort()
print('bonus:',list_Bonus)
list_salary=data["Salary"].values.tolist()
list_salary.sort()
print('salary:',list_salary)
list_Unit_Price=data["Unit_Price"].values.tolist()
list_Unit_Price.sort()
print('unit price:',list_Unit_Price)
list_openingbalance=data["openingbalance"].values.tolist()
list_openingbalance.sort()
print('opening balance:',list_openingbalance)
list_closingbalance=data["closingbalance"].values.tolist()
list_closingbalance.sort()
print('closing balance:',list_closingbalance)
list_low=data["low"].values.tolist()
list_low.sort()
print('low:',list_low)
```

```
age: [18, 18, 19, 19, 19, 20, 20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21,
base pay: [2035.6, 2279.248, 2358.66, 2450.048, 2498.0, 2577.692, 2582.2,
bonus: [254.45, 284.906, 294.8325, 306.256, 312.25, 322.2115, 322.775, 322
salary: [5089.0, 5698.12, 5896.65, 6125.12, 6245.0, 6444.23, 6455.5, 6458.
unit price: [1.44, 1.47, 1.51, 1.52, 1.6, 1.61, 1.62, 1.63, 1.65, 1.66, 1.
opening balance: [3.68, 3.71, 3.75, 3.85, 4.21, 4.23, 4.26, 4.31, 4.4, 4.7
closing balance: [3.68, 3.76, 3.86, 4.21, 4.22, 4.28, 4.29, 4.31, 4.33, 4.
low: [3.65, 3.65, 3.72, 3.83, 4.08, 4.13, 4.15, 4.21, 4.22, 4.27, 4.66, 4.
```

```
data['Age'].unique()
```

```
array([18, 19, 22, 21, 23, 24, 43, 44, 27, 28, 29, 30, 31, 32, 33, 34,
35,
       36, 37, 38, 45, 39, 40, 41, 42, 46, 47, 48, 49, 50, 51, 52, 53,
54,
       55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
71,
       73, 74, 75, 76, 20, 26, 25, 78, 88, 72, 80, 82, 85, 79])
```

All these values are important with the categories of employees so the outliers are seems to be genuine at this point, and we proceed further without handlinh them. It will be better to choose a model less sensitive to outliers

3. Encoding

One Hot Encoding

One Hot Encoding (OHE) is generally preferred over Label Encoding when working with categorical variables in regression models.

```
# As per EDA omitting ['Dependancies','Calls','Billing','Rating']
# Select the columns for encoding
df = data[['Education','Gender','Type']]

from sklearn.preprocessing import OneHotEncoder

# Initialize the OneHotEncoder with dummy variable trap
encoder = OneHotEncoder(drop='first')

# Convert the selected columns to a 2D numpy array and then perform one-hot encc
one_hot_encoded = encoder.fit_transform(df)
```

Only the most relevent categorical columns are selected for encodinh as the other 4 doesnot make any significant relation with the target variable , which we have already seen in the plots represened during EDA

```
#Columns to be encoded

df.head()
```

	Education	Gender	Type
0	High School or less	Female	Month-to-month
1	High School or less	Female	Month-to-month
2	High School or less	Male	Month-to-month
3	High School or less	Female	Month-to-month
4	High School or less	Male	Month-to-month

```
# Convert the sparse matrix to a dense NumPy array
one_hot_encoded_array = one_hot_encoded.toarray()

# Get the column names for the one-hot encoded features
columns = encoder.get_feature_names_out(input_features= ['Education', 'Gender',

# Create a DataFrame with the one-hot encoded array and appropriate column names
df_encoded = pd.DataFrame(one_hot_encoded_array, columns=columns)
```

```
# Display the encoded DataFrame with selected columns of original data frame

df_encoded.head()
```

	Education_High School or less	Education_Intermediate	Education_PG	Gender_Male	Type_ y
0	1.0	0.0	0.0	0.0	
1	1.0	0.0	0.0	0.0	
2	1.0	0.0	0.0	1.0	
3	1.0	0.0	0.0	0.0	
4	1.0	0.0	0.0	1.0	

```
# Number of columns after encoding

df_encoded.shape
```

```
(5000, 6)
```

3 relavent categorical features are one-hot encoded, resulting in 6 one-hot encoded categorical features." With dummy variable trap 3 columns reduced

4. Feature Scaling

```
# Dropping categorical,discrete numerical and Target columns before scaling

x = data.drop(['Gender','Dependancies','Calls','Type','Billing','Rating','Educat
```

Standard Scaling

to ensure that features have similar scales, which can improve the performance of various machine learning algorithms.

```
# x is a DataFrame containing our original data for encoding

from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# Apply the StandardScaler to our data and create a DataFrame with scaled values
scaled = scaler.fit_transform(x)
scaled =pd.DataFrame(scaled,columns=x.columns)

# Display summary statistics of the scaled DataFrame
scaled.describe()
```

	Age	Base_pay	Bonus	Unit_Price	Volume	oper
count	5.000000e+03	5.000000e+03	5.000000e+03	5.000000e+03	5.000000e+03	5
mean	-2.728484e-16	-3.183231e-16	5.456968e-16	1.136868e-16	1.136868e-17	-
std	1.000100e+00	1.000100e+00	1.000100e+00	1.000100e+00	1.000100e+00	1
min	-3.956268e+00	-3.717074e+00	-3.733402e+00	-9.536690e-01	-4.172810e-01	-1
25%	-5.683521e-01	-6.272005e-01	-6.278601e-01	-4.887368e-01	-3.380463e-01	-
50%	1.577133e-02	2.719230e-02	2.985092e-02	-2.307389e-01	-2.401177e-01	-
75%	5.998947e-01	6.676132e-01	6.735193e-01	1.427383e-01	-3.173215e-02	
max	4.221460e+00	3.924420e+00	3.946841e+00	1.106941e+01	1.938558e+01	8

Standardization

-Centers data around the mean and scales to a standard deviation of 1

-Useful when the distribution of the data is Gaussian or unknown

-Less sensitive to outliers

-Changes the shape of the original distribution

-Preserves the relationships between the data points

-Equation: (x – mean)/standard deviation

```
#number of columns in Scaled data set

scaled.shape
```

```
(5000, 11)
```

5. Dimensionality reduction

PCA (Principal Component Analysis)

```
#principal component analysis

from sklearn.decomposition import PCA

# Initialize PCA with the desired explained variance
pca = PCA(0.99)

# 'scaled' contains our scaled data
# Fit and transform PCA to our scaled data
s_pca = pca.fit_transform(scaled)

#shape that reflects the reduced dimensions
s_pca.shape
```

```
(5000, 8)
```

It's reducing the dimensionality of your data while retaining 99% of the variance.The s_pca array should have a shape that reflects the reduced dimensions based on the retained variance.

```
# Get the principal components
selected_columns = pca.components_

#  'scaled' contains our scaled data and 'pca' is our PCA model
# Identify the column names with the highest absolute values in each principal c
selected_columns_names = scaled.columns[np.argmax(np.abs(selected_columns), axis

# Get unique column names
selected_columns_names =np.unique(selected_columns_names)

# Print the selected column names
print(selected_columns_names)
```

```
['Age' 'Base_pay' 'Months' 'Total_Sales' 'Volume' 'closingbalance'
 'openingbalance']
```

Identified the names of the columns that have the highest absolute values in the principal components obtained from the PCA transformation. These columns contribute the most to the variance captured by the principal components.Principal Component Analysis shows that 7 coloumns contribute to the 99% of the variance of the data. The coloumns are 'Age', 'Base_pay', 'Months', 'Total_Sales' ,'Volume', 'closingbalance','openingbalance'.

6. Feature Selection

```
# Display scaled columns after PCA

scaled[selected_columns_names]
```

	Age	Base_pay	Months	Total_Sales	Volume	closingbalance	ope
0	-3.956268	-3.717074	-1.306505	-0.993980	0.892749	-1.071962	
1	-3.839443	-3.693190	-1.306505	-0.993958	0.228446	-1.074116	
2	-3.488969	-3.685406	-1.306505	-0.993936	0.740581	-1.057694	
3	-3.605794	-3.676447	-1.306505	-0.993892	3.664065	-1.057156	
4	-3.372144	-3.671747	-1.265911	-0.993869	1.240929	-1.054463	
...	...	...	...	...	...	...	
4995	2.352265	3.168671	1.616253	-0.385938	-0.174920	6.737445	
4996	2.469090	3.364298	1.616253	-0.385938	-0.045014	6.938616	
4997	2.585914	3.636850	1.616253	-0.385938	0.073328	7.065920	
4998	2.585914	3.767575	1.616253	-0.385938	-0.176068	7.134301	
4999	4.221460	3.924420	1.616253	-0.385938	-0.046779	7.234647	

5000 rows × 7 columns

```
# Concatination

data1  = pd.concat([scaled[selected_columns_names],df_encoded,data['Salary']], a
```

scaled and encoded data combined together with target, the discrete feture Business also is not significant here which is evident from the heat map during EDA

```
# Final Dataset for modeling

data1.head()
```

	Age	Base_pay	Months	Total_Sales	Volume	closingbalance	openin
0	-3.956268	-3.717074	-1.306505	-0.993980	0.892749	-1.071962	
1	-3.839443	-3.693190	-1.306505	-0.993958	0.228446	-1.074116	
2	-3.488969	-3.685406	-1.306505	-0.993936	0.740581	-1.057694	
3	-3.605794	-3.676447	-1.306505	-0.993892	3.664065	-1.057156	
4	-3.372144	-3.671747	-1.265911	-0.993869	1.240929	-1.054463	

```
# Statistical Summary

data1.describe()
```

	Age	Base_pay	Months	Total_Sales	Volume	clo
count	5.000000e+03	5.000000e+03	5.000000e+03	5.000000e+03	5.000000e+03	
mean	-2.728484e-16	-3.183231e-16	-1.818989e-16	4.547474e-17	1.136868e-17	
std	1.000100e+00	1.000100e+00	1.000100e+00	1.000100e+00	1.000100e+00	
min	-3.956268e+00	-3.717074e+00	-1.306505e+00	-9.939799e-01	-4.172810e-01	-
25%	-5.683521e-01	-6.272005e-01	-9.817544e-01	-8.303990e-01	-3.380463e-01	
50%	1.577133e-02	2.719230e-02	-1.698772e-01	-3.859381e-01	-2.401177e-01	
75%	5.998947e-01	6.676132e-01	9.261571e-01	6.415690e-01	-3.173215e-02	
max	4.221460e+00	3.924420e+00	1.616253e+00	2.833082e+00	1.938558e+01	

Modeling

Train Test Split

```
# Target variable
y = data1['Salary']

# Features without target variable
X = data1.drop(['Salary'],axis=1)
```

```
#split the data into training and testing set

from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size =0.25,random_stat
```

Trying different Regression models

Linear Regression

```
# Import the necessary library
from sklearn.linear_model import LinearRegression
```

```
# Create an instance of the LinearRegression class
lr = LinearRegression()

# Fit the linear regression model to the training data
Linear_Model = lr.fit(X_train,y_train)
```

```
# Predict the target variable for the test data
y_pred = Linear_Model.predict(X_test)
```

```
#importing the necessary metrics from scikit-learn to evaluate your linear regre
from sklearn.metrics import mean_squared_error,r2_score
```

The mean_squared_error and r2_score are commonly used metrics to assess how well your model is performing.

```
data1['Salary'].mean()
```

```
99821.9285527176
```

```
print(mean_squared_error(y_test,y_pred))
```

```
1012093.3856638296
```

The mean squared error is a measure of the average squared difference between the predicted values and the actual target value. interpretation of the MSE depends on the scale of our target variable. A larger MSE indicates that the predictions are further away from the actual values, whereas a smaller MSE indicates that the predictions are closer to the actual values.

```
print(r2_score(y_test,y_pred))
```

```
0.9985383861647606
```

our linear regression model is able to explain about 99.85% of the variability in the dependent variable using the independent variables. This is a very high R2 value and suggests that your model is fitting the data very well.

However, it's important to note that a very high R2 value might also indicate overfitting, especially if the model is too complex and is capturing noise in the data. It's always a good practice to evaluate our model's performance on new, unseen data (testing data or cross-validation) to ensure that the high R2 value is not solely a result of fitting noise in the training data.

Lasso Regression

```
# Import the necessary library
from sklearn.linear_model import Lasso
```

```
# Create an instance of the LassoRegression class
lasso = Lasso(alpha = 0.1)#alpha is the regularization Parametre

# Fit the lasso regression model to the training data
Lasso_Model = lasso.fit(X_train,y_train)
```

```
# Predict the target variable for the test data
y_lasso_pred = Lasso_Model.predict(X_test)
```

```
print(mean_squared_error(y_test,y_lasso_pred))
```

```
1011933.2473709969
```

predictions are further away from the actual value

```
print(r2_score(y_test,y_lasso_pred))
```

```
0.9985386174283453
```

99.85% of the variability in the dependent variable

RANSAC Regression

(Random sample consensus (RANSAC) regression)

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import RANSACRegressor

#Create a RANSACRegressor with the base estimator

base_estimator = LinearRegression()

# Create an instance of the RANSACRegressor
regressor = RANSACRegressor(base_estimator=base_estimator)
```

GridSearchCV is used for performing a grid search over specified parameter values for a given estimator, which is helpful for hyperparameter tuning and model selection.

RANSACRegressor is a type of regression model that uses the RANSAC (RANdom SAmple Consensus) algorithm for robustly fitting a regression model to data that may contain outliers.

Hyper Parameter Tuning

```
#Define hyperparameters and their potential values for tuning
param_grid = { 'min_samples': [10, 20, 50], 'residual_threshold': [5, 10, 20], '
```

Perform grid search with cross-validation

```
# Create an instance of GridSearchCV with RANSACRegressor and parameter grid

grid_search = GridSearchCV(regressor, param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

```
▸ GridSearchCV
▸ estimator: RANSACRegressor
▸ base_estimator: LinearRegression
▸ LinearRegression
```

```
#to get the best hyperparameter and the model

best_params = grid_search.best_params_
best_model =grid_search.best_estimator_
```

```
print(best_params)
```

```
{'max_trials': 200, 'min_samples': 10, 'residual_threshold': 10}
```

```
print(best_model)
```

```
RANSACRegressor(base_estimator=LinearRegression(), estimator=LinearRegress
                max_trials=200, min_samples=10, residual_threshold=10)
```

In this example, X_train and y_train represent your training data and target values. The GridSearchCV function performs a grid search over the specified hyperparameters and evaluates the model's performance using cross-validation.

Remember that hyperparameter tuning can be computationally intensive, especially with a large parameter grid or a complex base estimator. It's important to balance the exploration of hyperparameters with your computational resources and time constraints.

```
# Initialize RANSACRegressor with specified parameters

reg = RANSACRegressor(base_estimator=LinearRegression(),
                        min_samples=10, max_trials=100,
                        loss='absolute_error', random_state=42,
                        residual_threshold=5)

 # Fit the RANSACRegressor model to the training data
model_reg= reg.fit(X_train,y_train)

# Predict the target variable for the test data
y_pred_reg = model_reg.predict(X_test)

from sklearn.metrics import mean_squared_error,r2_score

print("\nMean Squared Error in RANSAC Regression is ",mean_squared_error(y_test,
print("R Squared Error in RANSAC Regression is ",r2_score(y_test,y_pred_reg))
```

```
    Mean Squared Error in RANSAC Regression is  1020717.7165551609
    R Squared Error in RANSAC Regression is  0.9985259313443566
```

predictions are further away from the actual value

99.85% of the variability in the dependent variable

hyperparameter tuning, model training, and evaluation are iterative processes, and we may need to adjust our approach based on the results we obtain.

Gradient Boosting

```
from sklearn.ensemble import GradientBoostingRegressor
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

GradientBoostingRegressor model using the X_train and y_train data, and then evaluate the model's performance using the test data

```
from sklearn.model_selection import GridSearchCV

#define the parameter grid for the grid search

param_grid = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
```

```
    'max_depth': [3, 4, 5]
}

# Initialize GridSearchCV

grid_search = GridSearchCV(
    GradientBoostingRegressor(),
    param_grid,# Parameter grid
    cv=5,  # Number of cross-validation folds
    scoring='neg_mean_squared_error',  # Evaluation metric
    n_jobs=-1  # Number of CPU cores to use (-1 for all available cores)
)

# Fit the grid search to the training data
grid_search.fit(X_train, y_train)


# Get the best hyperparameters and best estimator from the grid search

best_params = grid_search.best_params_
best_estimator = grid_search.best_estimator_
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-117-475f08555983> in <cell line: 17>()
     15 )
     16
---> 17 grid_search.fit(X_train, y_train)
     18 best_params = grid_search.best_params_
     19 best_estimator = grid_search.best_estimator_

6 frames
/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in _retrieve(self)
   1705                 (self._jobs[0].get_status(
   1706                     timeout=self.timeout) == TASK_PENDING)):
-> 1707                 time.sleep(0.01)
   1708                 continue
   1709

KeyboardInterrupt:
```

```
best_params
```

```
{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 300}
```

```
best_estimator
```

```
▾ GradientBoostingRegressor
GradientBoostingRegressor(max_depth=5, n_estimators=300)
```

```
# Initialize and train the GradientBoostingRegressor with specified hyperparamet

gb_regressor = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, ma
gb_regressor.fit(X_train, y_train)

# Predict on the test data
y_pred_gb = gb_regressor.predict(X_test)

# Calculate and print the Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred_gb)
print(f"Mean Squared Error: {mse}")

# Calculate and print the R-squared error
print("R Squared Error in GradientBoosting is ",r2_score(y_test,y_pred_gb))
```

```
Mean Squared Error: 87623.2900363851
R Squared Error in GradientBoosting is  0.9998736194172944
```

It looks like our Gradient Boosting Regressor model is performing remarkably well on the test data. An MSE of approximately 87623.29 and an R-squared error of about 0.9999 indicate that our model is fitting the data very closely and explaining a significant amount of the variance in the target variable.

To ensure the model's robustness and generalization ability, need to Perform cross-validation to evaluate the model's performance on multiple folds of the training data. This can provide a more robust estimate of how well our model is likely to perform on new data.

K - Fold Cross_Validation

```
from sklearn.model_selection import KFold

# Create an instance of KFold cross-validator with 100 folds
kfold_validator =KFold(100)

# for train_index,test_index in kfold_validator.split(X,y):
```

```
from sklearn.model_selection import cross_val_score
```

```
# Calculate cross-validated scores using the specified model, feature matrix, ar
cv_score = cross_val_score(model_reg,X,y,cv= kfold_validator)
```

This is a valuable step to assess how well our model generalizes across different folds of the data.

```
cv_score
```

```
array([ 1.00000000e+00,  1.00000000e+00,  9.32185556e-01,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00, -1.34152896e+02,
-5.15220381e+01,
       -8.96553193e+01,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
1.00000000e+00,
        1.00000000e+00,  6.99150317e-01, -1.28576231e+01,
-2.10854306e+01,
       -6.95159603e+00, -2.03226583e+00, -9.65393711e-02,  4.58471109e-
03,
        3.31373314e-01,  7.91656601e-01, -2.74503732e+00, -4.48713507e-
01,
       -1.11079380e+00, -4.30057807e+00, -2.94261506e+00,  4.64237388e-
01])
```

```
np.mean(cv_score)
```

    -2.4567825820202516

However, it's worth noting that this negative value doesn't directly represent an error or loss value.

```
mean_cv_score = -np.mean(cv_score)
print("Mean Cross-Validated Score:", mean_cv_score)
```

This will give you the average performance score (mean squared error, R-squared, or any other scoring metric) of our model across the different folds of cross-validation.

the actual interpretation of this score depends on the specific scoring metric we used when setting up the cross_val_score function.

Random forest Regressor

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [100, 200, 300,500],
}

grid_search = GridSearchCV(
    RandomForestRegressor(),
    param_grid,
    cv=5,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_estimator = grid_search.best_estimator_
```

```
best_params
```

```python
from sklearn.ensemble import RandomForestRegressor
rf_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
rf_regressor.fit(X_train, y_train)
y_pred_rf = rf_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred_rf)
print(f"Mean Squared Error: {mse}")
print("R Squared Error in RandomForestRegressor is ",r2_score(y_test,y_pred_rf))
```

```
Mean Squared Error: 390750.6487156924
R Squared Error in RandomForestRegressor is  0.9994356977702042
```

Decision Tree Regressor

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dist = {
    'max_depth': randint(10, 20),

}

randomized_search = RandomizedSearchCV(
    RandomForestRegressor(),
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings sampled
    cv=5,
    scoring='neg_mean_squared_error',
    n_jobs=-1
)

randomized_search.fit(X_train, y_train)
best_params = randomized_search.best_params_
best_estimator = randomized_search.best_estimator_
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call
last)
<ipython-input-113-7705f68945ed> in <cell line: 18>()
     16 )
     17
---> 18 randomized_search.fit(X_train, y_train)
     19 best_params = randomized_search.best_params_
     20 best_estimator = randomized_search.best_estimator_
```

6 frames

```
/usr/local/lib/python3.10/dist-packages/joblib/parallel.py in
_retrieve(self)
   1705                 (self._jobs[0].get_status(
   1706                     timeout=self.timeout) == TASK_PENDING)):
-> 1707                 time.sleep(0.01)
   1708                 continue
   1709

KeyboardInterrupt:
```

```
best_params
```

```
from sklearn.tree import DecisionTreeRegressor
tree_regressor = DecisionTreeRegressor(max_depth=200)
tree_regressor.fit(X_train, y_train)
y_pred_tree = tree_regressor.predict(X_test)

print("\nMean Squared Error in  Regression is ",mean_squared_error(y_test,y_pred
print("R Squared Error in Regression is ",r2_score(y_test,y_pred_tree))
```

```
Mean Squared Error in RANSAC Regression is  171589.5046780232
R Squared Error in RANSAC Regression is  0.9997521991571412
```

After modeling with Linear Regression, Lasso Regression, Robust Regression, Random Foresrt Regression and Decision Tree Regression models we could find that all these models are performing with higher MSE and almost similar percentage of accuracy or almost similar r2 score.

After fine tuning in Gradient Boosting Regression we get better values and hence lets proceed with this model

Colab paid products - Cancel contracts here