PREDICTING HOUSE PRICE USING MACHINE LEARNING

712521104021______M.MERWIN SAMSON

PPG INSTITUTE OF TEHNOLOGY

BE-CSE 3RD YEAR

Phase 3 submission document

Project Title: House Price Prediction

Phase 3: Development Part 1

Topic: ***Start building the house price prediction model by loading and pre-processing the dataset.***

\

House Price Prediction

Introduction:

★ Whether you're a homeowner looking to estimate the value of your property, a real estate investor seeking profitable opportunities, or a data scientist aiming to build a predictive model, the foundation of this endeavor lies in loading and preprocessing the dataset.

★ Building a house price prediction model is a data-driven process that involves harnessing the power of machine learning to analyze historical housing data and make informed price predictions. This journey begins with the fundamental steps of data loading and preprocessing.

★ This introduction will guide you through the initial steps of the process. We'll explore how to import essential libraries, load the housing dataset, and perfor critical preprocessing steps. Data preprocessing is crucial as it helps clean, format, and prepare the data for further analysis. This includes handling missing values, encoding categorical variables, and ensuring that the data is appropriately scaled.

GIVEN DATASET

	Avg.Area income	Avg.Area house age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Address
1.	79545.45857431678	5.682861321615587	7.009188142792237	4.09	23086.800502686456	1059033.5578701235208	Michael Ferry Apt. 674 Laurabury, NE 37010-5101
2.	79248.64245482568	6.0028998082752425	6.730821019094919	3.09	40173.07217364482	1505890.91484695	188,Johnson Views Suite 079 Lake Kathleen, CA 48958
3.	61287.067178656784	5.86588984031001	8.512727430375099	5.13	36882.15939970458	1058987.9878760849	9127, Elizabeth Stravenue Danieltown, WI 06482-3489

4.	63345.24004622798	7.1882360945186425	5.586728664827653	3.26	34310.24283090706	1260616.8066294468	USS Barnett FPO AP 44820
5.	59982.197225708034	5.040554523106283	7.839387785120487	4.23	26354.109472103148	630943.4893385402	USNS Raymond FPO AE 09386
...	.....	.....	...	...	...	...	...
4996	73060.85	5.29	6.31	4.16	22695.7	905354.91	5224 Lamb Passage Nancystad, GA 16579
4997	60567.94	7.83	6.14	3.46	22837.36	1060193.79	USNS Williams FPO AP 30153-7653
4998	63390.69	7.25	4.81	2.13	33266.15	1030729.58	4215 Tracy Garden Suite 076 Joshualand, VA 01707-9165
4999	68001.33	5.53	7.13	5.44	42625.62	1198656.87	USS Wallace FPO AE 73316
5000	65510.58	5.99	6.79	4.07	46501.28	1298950.48	37778 George Ridges Apt. 509 East Holly, NV 29290-3595

Necessary step to follow:

1.Import Libraries:

Start by importing the necessary libraries:

Program:

```
import pandas as pd

import numpy as np

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler
```

2.Load the Dataset:

Load your dataset into a Pandas DataFrame. You can typically find house price datasets in CSV format, but you can adapt this code to other formats as needed.

Program:

```
 df = pd.read_csv(' E:\USA_Housing.csv ')
 Pd.read()
```

3. Exploratory Data Analysis (EDA):

Perform EDA to understand your data better. This includes checking for missing values, exploring the data's statistics, and visualizing it to identify patterns.

Program:

```
# Check for missing values
print(df.isnull().sum())
# Explore statistics
print(df.describe())
# Visualize the data (e.g., histograms, scatter plots, etc.)
```

4. Feature Engineering:

Depending on your dataset, you may need to create new features or transform existing ones. This can involve one-hot encoding categorical variables, handling date/time data, or scaling numerical features.

Program:

```
# Example: One-hot encoding for categorical variables
 df = pd.get_dummies(df, columns=[' Avg. Area Income ', ' Avg. Area
 House Age '])
```

5. Split the Data:

Split your dataset into training and testing sets. This helps you evaluate your model's performance later.

```
X = df.drop('price', axis=1) # Features
y = df['price'] # Target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

6. Feature Scaling:

Apply feature scaling to normalize your data, ensuring that all features have similar scales. Standardization (scaling to mean=0 and std=1) is a common choice.

Program:

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Data Preprocessing:

Depending on your dataset, you might need to perform various preprocessing tasks:

a. **Handling Missing Values**: Use Pandas methods like `fillna()` or `dropna()` to handle missing data.

b. **Encoding Categorical Variables**: Convert categorical variables into numerical format using techniques like one-hot encoding or label encoding.

c. **Feature Scaling**: Normalize or standardize numerical features, if necessary, using methods like Min-Max scaling or Z-score normalization.

d. **Feature Selection**: Select the most relevant features for your machine learning task.

e. **Data Splitting**: Split your data into training and testing sets to evaluate the model's performance.

python

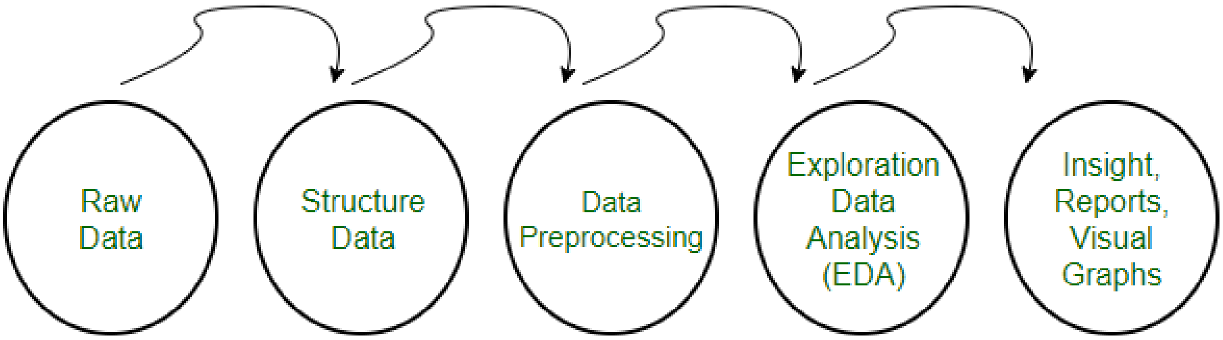
```
    from sklearn.model_selection import train_test_split
    X = data.drop('target_column', axis=1)  # Features (remove target column)
    y = data['target_column']  # Target variable

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

5. **Machine Learning**: If you're working on a machine learning project, you can now use `X_train` and `y_train` to train your model and `X_test` to evaluate its performance.
6. **Save Preprocessed Data (Optional)**: If needed, you can save the preprocessed data to a new CSV file for later use.

python

```
preprocessed_data.to_csv('preprocessed_data.csv', index=False)
```

Remember that the preprocessing steps will vary depending on your specific dataset and project goals. Be sure to tailor the steps above to your requirements and explore additional preprocessing techniques when necessary.


Raw Data
Structure Data
Data Preprocessing
Exploration Data Analysis (EDA)
Insight, Reports, Visual Graphs

Need of Data Preprocessing

- For achieving better results from the applied model in Machine Learning projects the format of the data has to be in a proper manner. Some specified Machine Learning model needs information in a specified format, for example, Random Forest algorithm does not support null values, therefore to execute random forest algorithm null values have to be managed from the original raw data set.
- Another aspect is that the data set should be formatted in such a way that more than one Machine Learning and Deep Learning algorithm are executed in one data set, and best out of them is chosen.

Steps in Data Preprocessing

Step 1: Import the necessary libraries

```
# importing libraries
import pandas as pd
import scipy
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import matplotlib.pyplot as plt
```

Step 2: Load the dataset

Dataset link: [https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database]

```
# Load the dataset
df =pd.read_csv('Geeksforgeeks/Data/diabetes.csv')
print(df.head())
```

Output:

```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6   
1            1       85             66             29        0  26.6   
2            8      183             64              0        0  23.3   
3            1       89             66             23       94  28.1   
4            0      137             40             35      168  43.1   

   DiabetesPedigreeFunction  Age  Outcome  
0                     0.627   50        1  
1                     0.351   31        0  
2                     0.672   32        1  
3                     0.167   21        0  
4                     2.288   33        1  
```

Check the data info

```
df.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

As we can see from the above info that the our dataset has 9 columns and each columns has 768 values. There is no Null values in the dataset.

We can also check the null values using df.isnull()

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

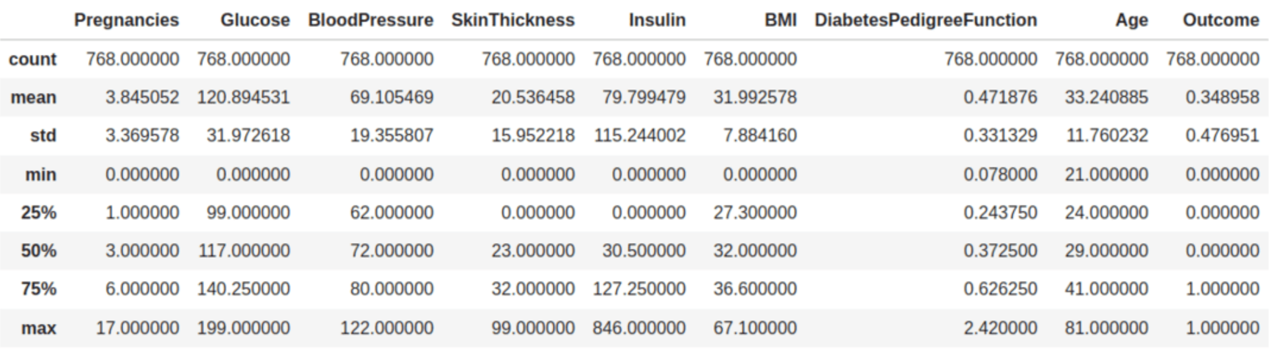
Output:

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Step 3: Statistical Analysis

In statistical analysis, first, we use the df.describe() which will give a descriptive overview of the dataset.

```
df.describe()
```



	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

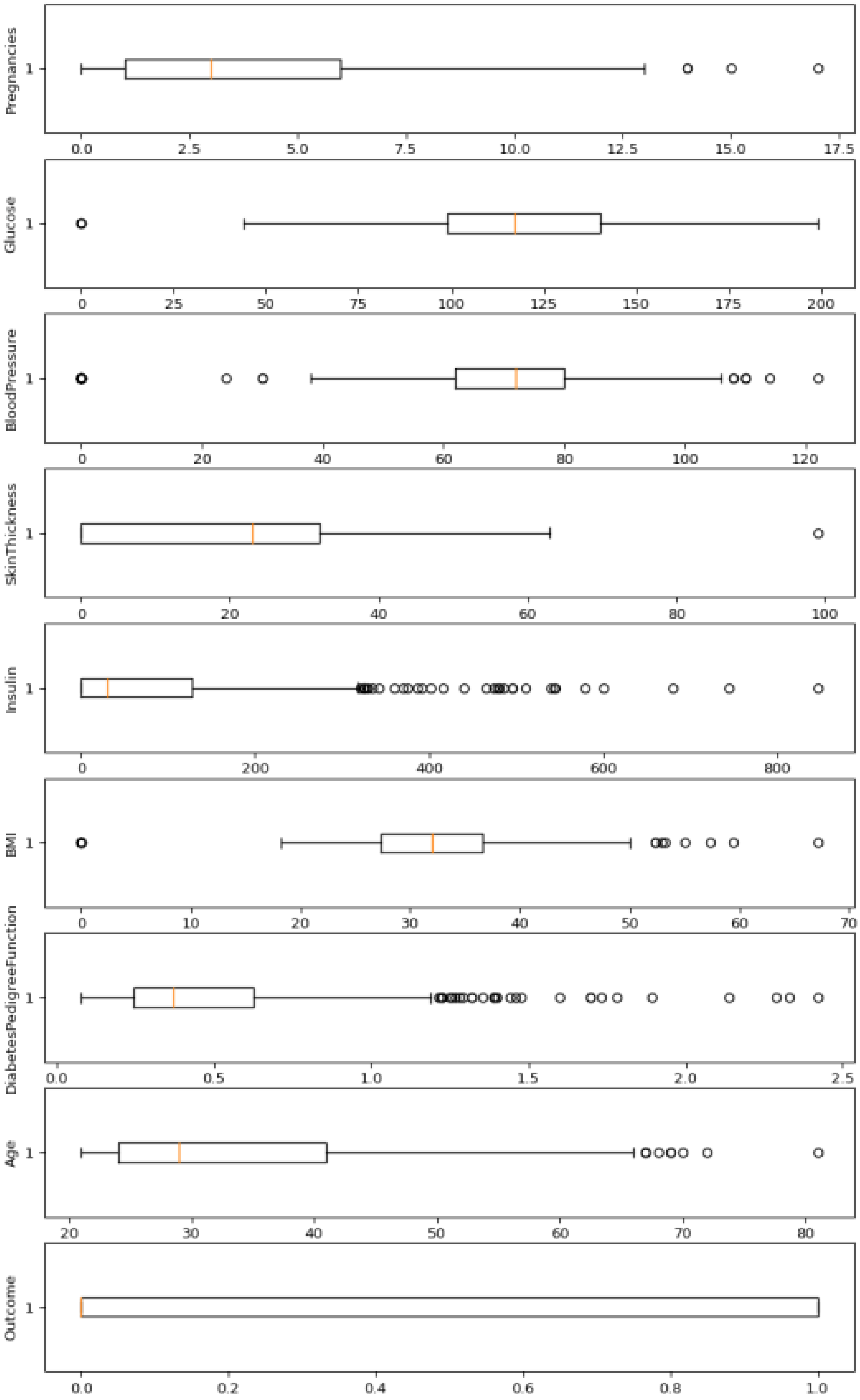
The above table shows the count, mean, standard deviation, min, 25%, 50%, 75%, and max values for each column. When we carefully observe the table we will find that. Insulin, Pregnancies, BMI, BloodPressure columns has outliers.

Let's plot the boxplot for each column for easy understanding.

Step 4: Check the outliers:

```
# Box Plots
fig, axs =plt.subplots(9,1,dpi=95, figsize=(7,17))
i =0
for col in df.columns:
    axs[i].boxplot(df[col], vert=False)
    axs[i].set_ylabel(col)
    i+=1
plt.show()
```

Output:



Pregnancies
1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Glucose
1
0
25
50
75
100
125
150
175
200
BloodPressure
1
0
20
40
60
80
100
120
SkinThickness
1
0
20
40
60
80
100
Insulin
1
0
200
400
600
800
BMI
1
0
10
20
30
40
50
60
70
DiabetesPedigreeFunction
1
0.0
0.5
1.0
1.5
2.0
2.5
Age
1
20
30
40
50
60
70
80
Outcome
1
0.0
0.2
0.4
0.6
0.8
1.0

from the above boxplot, we can clearly see that all most every column has some amounts of outliers.

Drop the outliers

```
# Identify the quartiles
q1, q3 = np.percentile(df['Insulin'], [25, 75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (1.5 * iqr)
upper_bound = q3 + (1.5 * iqr)
# Drop the outliers
clean_data = df[(df['Insulin'] >= lower_bound)
                & (df['Insulin'] <= upper_bound)]


# Identify the quartiles
q1, q3 = np.percentile(clean_data['Pregnancies'], [25, 75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (1.5 * iqr)
upper_bound = q3 + (1.5 * iqr)
# Drop the outliers
clean_data = clean_data[(clean_data['Pregnancies'] >= lower_bound)
                        & (clean_data['Pregnancies'] <=
upper_bound)]


# Identify the quartiles
q1, q3 = np.percentile(clean_data['Age'], [25, 75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (1.5 * iqr)
upper_bound = q3 + (1.5 * iqr)
# Drop the outliers
clean_data = clean_data[(clean_data['Age'] >= lower_bound)
                        & (clean_data['Age'] <= upper_bound)]


# Identify the quartiles
q1, q3 = np.percentile(clean_data['Glucose'], [25, 75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (1.5 * iqr)
upper_bound = q3 + (1.5 * iqr)
```

```
# Drop the outliers
clean_data = clean_data[(clean_data['Glucose'] >= lower_bound)
                        & (clean_data['Glucose'] <= upper_bound)]


# Identify the quartiles
q1, q3 = np.percentile(clean_data['BloodPressure'], [25, 75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (0.75 * iqr)
upper_bound = q3 + (0.75 * iqr)
# Drop the outliers
clean_data = clean_data[(clean_data['BloodPressure'] >= lower_bound)
                        & (clean_data['BloodPressure'] <=
upper_bound)]


# Identify the quartiles
q1, q3 = np.percentile(clean_data['BMI'], [25, 75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (1.5 * iqr)
upper_bound = q3 + (1.5 * iqr)
# Drop the outliers
clean_data = clean_data[(clean_data['BMI'] >= lower_bound)
                        & (clean_data['BMI'] <= upper_bound)]


# Identify the quartiles
q1, q3 = np.percentile(clean_data['DiabetesPedigreeFunction'], [25,
75])
# Calculate the interquartile range
iqr = q3 - q1
# Calculate the lower and upper bounds
lower_bound = q1 - (1.5 * iqr)
upper_bound = q3 + (1.5 * iqr)

# Drop the outliers
clean_data = clean_data[(clean_data['DiabetesPedigreeFunction'] >=
lower_bound)
                        & (clean_data['DiabetesPedigreeFunction']
<= upper_bound)]
```

Step 5: Correlation

```
#correlation
corr = df.corr()
```

```
plt.figure(dpi=130)
sns.heatmap(df.corr(), annot=True, fmt= '.2f')
plt.show()
```

Output:

Correlation

We can also camapare by single columns in descending order

```
corr['Outcome'].sort_values(ascending = False)
```

Output:

```
Outcome                     1.000000
Glucose                     0.466581
BMI                         0.292695
Age                         0.238356
Pregnancies                 0.221898
DiabetesPedigreeFunction    0.173844
Insulin                     0.130548
SkinThickness               0.074752
BloodPressure               0.0
```

Check Outcomes Proportionality

```
plt.pie(df.Outcome.value_counts(),
          labels= ['Diabetes', 'Not Diabetes'],
          autopct='%.f', shadow=True)
plt.title('Outcome Proportionality')
plt.show()
```

Output:

Correlation

We can also camapare by single columns in descending order

```
corr['Outcome'].sort_values(ascending = False)
```

Output:

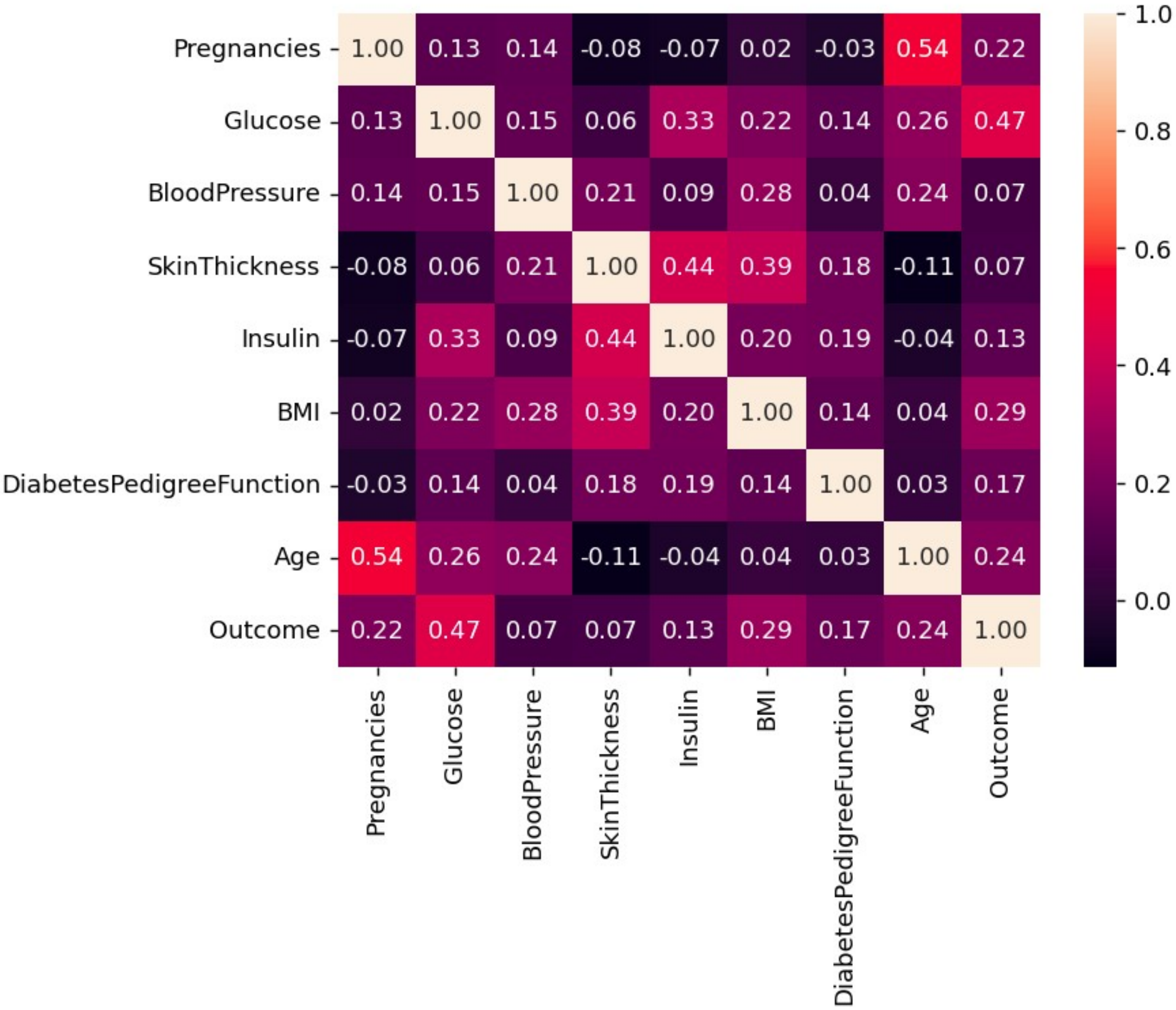
```
Outcome                     1.000000
Glucose                     0.466581
BMI                         0.292695
Age                         0.238356
Pregnancies                 0.221898
DiabetesPedigreeFunction    0.173844
Insulin                     0.130548
```

```
SkinThickness               0.074752
BloodPressure               0.0
```

Check Outcomes Proportionality

```
plt.pie(df.Outcome.value_counts(),
        labels= ['Diabetes', 'Not Diabetes'],
        autopct='%.f', shadow=True)
plt.title('Outcome Proportionality')
plt.show()
```

Output:



Correlation

We can also camapare by single columns in descending order

```
corr['Outcome'].sort_values(ascending = False)
```

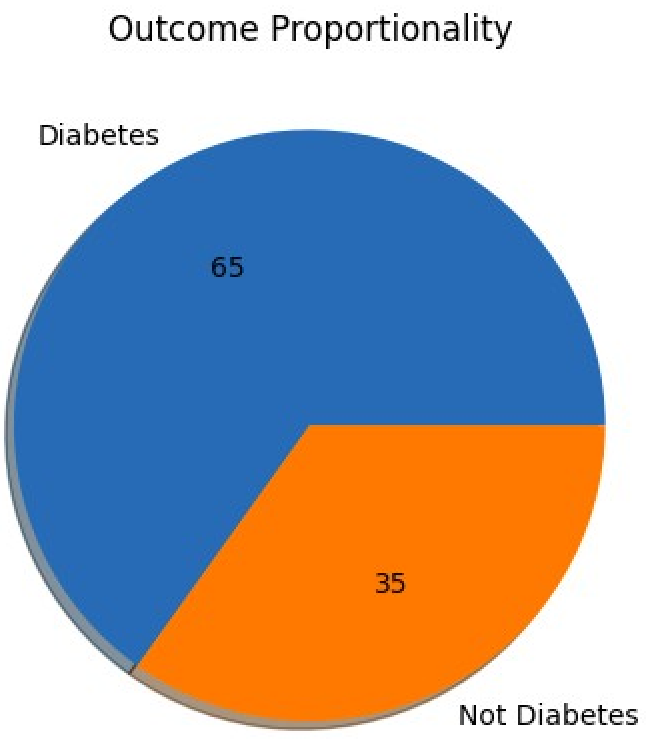
Output:

```
Outcome                     1.000000
Glucose                     0.466581
BMI                         0.292695
Age                         0.238356
Pregnancies                 0.221898
DiabetesPedigreeFunction    0.173844
Insulin                     0.130548
SkinThickness               0.074752
BloodPressure               0.0
```

Check Outcomes Proportionality

```
plt.pie(df.Outcome.value_counts(),
         labels= ['Diabetes', 'Not Diabetes'],
         autopct='%.f', shadow=True)
plt.title('Outcome Proportionality')
plt.show()
```

output



Step 6: Separate independent features and Target Variables

```
# separate array into input and output components
X = df.drop(columns =['Outcome'])
Y = df.Outcome
```

Step 7: Normalization or Standardization

Normalization

- MinMaxScaler scales the data so that each feature is in the range [0, 1].
- It works well when the features have different scales and the algorithm being used is sensitive to the scale of the features, such as k-nearest neighbors or neural networks.
- Rescale your data using scikit-learn using the MinMaxScaler.

```
# initialising the MinMaxScaler
scaler = MinMaxScaler(feature_range=(0, 1))

# learning the statistical parameters for each of the data and
transforming
rescaledX = scaler.fit_transform(X)
rescaledX[:5]
```

Output:

```
array([[0.353, 0.744, 0.59 , 0.354, 0.   , 0.501, 0.234, 0.483],
       [0.059, 0.427, 0.541, 0.293, 0.   , 0.396, 0.117, 0.167],
       [0.471, 0.92 , 0.525, 0.   , 0.   , 0.347, 0.254, 0.183],
       [0.059, 0.447, 0.541, 0.232, 0.111, 0.419, 0.038, 0.   ],
       [0.   , 0.688, 0.328, 0.354, 0.199, 0.642, 0.944, 0.2  ]])
```

Standardization

- Standardization is a useful technique to transform attributes with a Gaussian distribution and differing means and standard deviations to a standard Gaussian distribution with a mean of 0 and a standard deviation of 1.
- We can standardize data using scikit-learn with the StandardScaler class.
- It works well when the features have a normal distribution or when the algorithm being used is not sensitive to the scale of the features

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(X)
rescaledX = scaler.transform(X)
rescaledX[:5]
```

Output:

```
array([[ 0.64 ,  0.848,  0.15 ,  0.907, -0.693,  0.204,  0.468,  1.426],
       [-0.845, -1.123, -0.161,  0.531, -0.693, -0.684, -0.365, -0.191],
       [ 1.234,  1.944, -0.264, -1.288, -0.693, -1.103,  0.604, -0.106],
       [-0.845, -0.998, -0.161,  0.155,  0.123, -0.494, -0.921, -1.042],
       [-1.142,  0.504, -1.505,  0.907,  0.766,  1.41 ,  5.485,
```

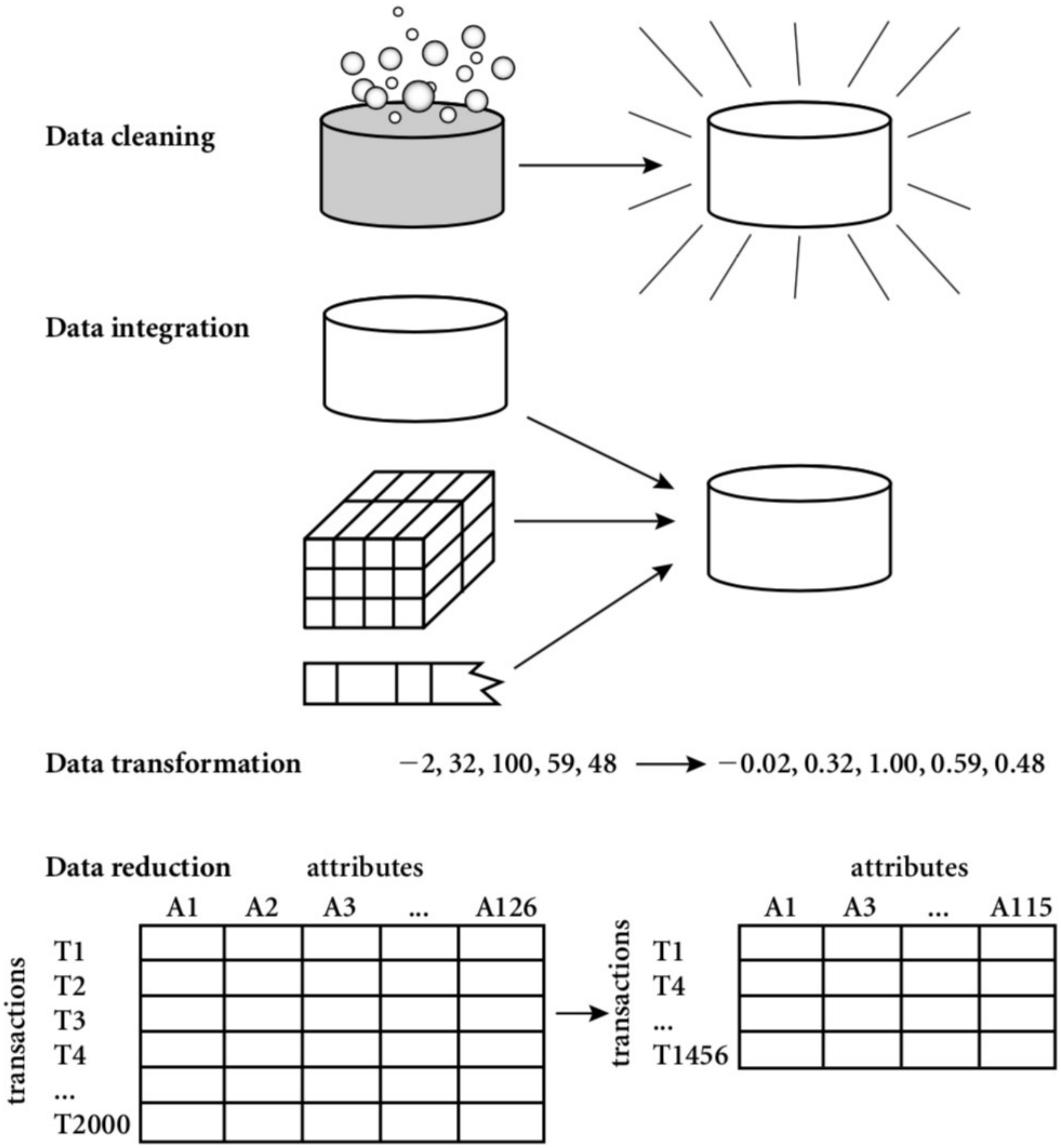
Program:

```
    # Importing necessary libraries
    import pandas as pd
    import numpy as np
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler,
OneHotEncoder
    from sklearn.compose import ColumnTransformer
    from sklearn.pipeline import Pipeline

 # Step 1: Load the dataset

    data = pd.read_csv('E:\USA_Housing.csv')

 # Step 2: Exploratory Data Analysis (EDA)

    print("--- Exploratory Data Analysis ---")
    print("1. Checking for Missing Values:")

    missing_values = data.isnull().sum()
    print(missing_values)
    print("\n2. Descriptive Statistics:")
    description = data.describe()Page | 22
    print(description)

 # Step 3: Feature Engineering

   print("\n--- Feature Engineering ---")
    # Separate features and target variable
   X = data.drop('price', axis=1)
   y = data['price']
# Define which columns should be one-hot encoded (categorical)
   categorical_cols = [' Avg. Area House Age ']
# Define preprocessing steps using ColumnTransformer and Pipeline
   preprocessor = ColumnTransformer(
   transformers=[
   ('num', StandardScaler(), [' Avg. Area Number of Rooms ', '
Avg.
    Area Number of Bedrooms ', ' Area Population ', ' Avg. Area
Income ']),
    ('cat', OneHotEncoder(), categorical_cols)])Page | 23

# Step 4: Data Splitting

   print("\n--- Data Splitting ---")
```

```
    X_train, X_test, y_train, y_test = train_test_split(X,
y,test_size=0.2,random_state=42)

    print(f"X_train shape: {X_train.shape}")
    print(f"X_test shape: {X_test.shape}")
    print(f"y_train shape: {y_train.shape}")
    print(f"y_test shape: {y_test.shape}")
# Step 5:
Preprocessing and Feature Scaling using Pipeline
    print("\n--- Feature Scaling ---")
    model = Pipeline([
    ('preprocessor', preprocessor),])

# Fit the preprocessing pipeline on the training data

    X_train = model.fit_transform(X_train)

# Transform the testing data using the fitted pipeline

    X_test = model.transform(X_test)
    print("--- Preprocessing Complete! ---")
```

There are a number of data preprocessing techniques available such as,

1. **Data Cleaning**
2. **Data Integration**
3. **Data Transformation**
4. **Data Reduction**



Data cleaning can be applied to filling in missing values, remove noise, resolving inconsistencies, identifying and removing outliers in the data.

- **Data integration** merges data from multiple sources into a coherent data store, such as a data warehouse.
- **Data transformations**, such as normalization, may be applied. For example, normalization may improve the accuracy and efficiency of mining algorithms involving distance measurements.
- **Data reduction** can reduce the data size by eliminating redundant features, or clustering, for instance.

Reference: Data Mining:Concepts and Techniques Second Edition, Jiawei Han, Micheline Kamber.

PS: This is my first kaggle notebook contribution. Hope you like it!!

Import the required libraries

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from operator import itemgetter
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.preprocessing import OrdinalEncoder
from category_encoders.target_encoder import TargetEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import (GradientBoostingRegressor,
GradientBoostingClassifier)
import xgboost
```

Load the dataset for training and testing

```
train = pd.read_csv('../input/house-prices-advanced-regression-techniques/
train.csv')
test = pd.read_csv('../input/house-prices-advanced-regression-techniques/
test.csv')
```

Data Cleaning

Find the missing percentage of each columns in training set.

```
def find_missing_percent(data):
    """
    Returns dataframe containing the total missing values and percentage
of total
    missing values of a column.
    """
    miss_df = pd.DataFrame({'ColumnName':[],'TotalMissingVals':
[],'PercentMissing':[]})
    for col in data.columns:
        sum_miss_val = data[col].isnull().sum()
        percent_miss_val = round((sum_miss_val/data.shape[0])*100,2)
        miss_df = miss_df.append(dict(zip(miss_df.columns,
[col,sum_miss_val,percent_miss_val])),ignore_index=True)
    return miss_df
```

```
miss_df = find_missing_percent(train)
'''Displays columns with missing values'''
display(miss_df[miss_df['PercentMissing']>0.0])
print("\n")
print(f"Number of columns with missing values:
{str(miss_df[miss_df['PercentMissing']>0.0].shape[0])}")
```

	ColumnName	TotalMissingVals	PercentMissing
3	LotFrontage	259.0	17.74
6	Alley	1369.0	93.77
25	MasVnrType	8.0	0.55
26	MasVnrArea	8.0	0.55
30	BsmtQual	37.0	2.53
31	BsmtCond	37.0	2.53
32	BsmtExposure	38.0	2.60
33	BsmtFinType1	37.0	2.53
35	BsmtFinType2	38.0	2.60
42	Electrical	1.0	0.07
57	FireplaceQu	690.0	47.26
58	GarageType	81.0	5.55
59	GarageYrBlt	81.0	5.55
60	GarageFinish	81.0	5.55
63	GarageQual	81.0	5.55
64	GarageCond	81.0	5.55
72	PoolQC	1453.0	99.52
73	Fence	1179.0	80.75
74	MiscFeature	1406.0	96.30

```
Number of columns with missing values:19
```

1.2 Drop the columns which have more than 70% of missing values

```
drop_cols = miss_df[miss_df['PercentMissing'] >70.0].ColumnName.tolist()
print(f"Number of columns with more than 70%: {len(drop_cols)}")
train = train.drop(drop_cols,axis=1)
test = test.drop(drop_cols,axis =1)

miss_df = miss_df[miss_df['ColumnName'].isin(train.columns)]
'''Columns to Impute'''
impute_cols = miss_df[miss_df['TotalMissingVals']>0.0].ColumnName.tolist()
miss_df[miss_df['TotalMissingVals']>0.0]
```

```
Number of columns with more than 70%: 4
```

	ColumnName	TotalMissingVals	PercentMissing
3	LotFrontage	259.0	17.74
25	MasVnrType	8.0	0.55
26	MasVnrArea	8.0	0.55
30	BsmtQual	37.0	2.53
31	BsmtCond	37.0	2.53
32	BsmtExposure	38.0	2.60
33	BsmtFinType1	37.0	2.53
35	BsmtFinType2	38.0	2.60
42	Electrical	1.0	0.07
57	FireplaceQu	690.0	47.26
58	GarageType	81.0	5.55
59	GarageYrBlt	81.0	5.55
60	GarageFinish	81.0	5.55
63	GarageQual	81.0	5.55
64	GarageCond	81.0	5.55

```
'''Segregate the numeric and categoric data'''
numeric_cols = train.select_dtypes(['float','int']).columns
categoric_cols = train.select_dtypes('object').columns

train_numeric = train[numeric_cols[:-1]]
train_categoric = train[categoric_cols]

test_numeric = test[numeric_cols[:-1]]
test_categoric = test[categoric_cols]

nominal_cols = ['MSZoning',
'Street','LandContour','Neighborhood','Condition1','Condition2',

'RoofStyle','RoofMatl','Exterior1st','Exterior2nd','MasVnrType','Foundati
on',
                'Heating','GarageType','SaleType','SaleCondition']
ordinal_cols =
['ExterQual','ExterCond','BsmtQual','BsmtCond','BsmtExposure','BsmtFinTyp
e1',

'BsmtFinType2','HeatingQC','CentralAir','Electrical','KitchenQual','Funct
ional',

'FireplaceQu','GarageFinish','GarageQual','GarageCond','PavedDrive','LotS
hape',

'Utilities','LandSlope','BldgType','HouseStyle','LotConfig']
```

MICE (Multiple Imputation by Chained Equation)

Imputation of missing values can be done using two techniques,

- **Single Imputation**
  - Single imputation denotes that the missing value is replaced by a value only once.
- **Multiple Imputation**
  - In multiple imputation, the imputation process is repeated multiple times resulting in multiple imputed datasets.

MICE Algorithm:

The chained equation process can be broken down into four general steps:

- **Step 1:** A simple imputation, such as imputing the mean, is performed for every missing value in the dataset. These mean imputations can be thought of as “place holders.”
- **Step 2:** The “place holder” mean imputations for one variable (“var”) are set back to missing.
- **Step 3:** The observed values from the variable “var” in Step 2 are regressed(can use any other regressors like Gradient Boosting Regressor or XGBoost Regressor for numeric data) on the other variables in the imputation model, which may or may not consist of all of the variables in the dataset. In other words, “var” is the dependent variable in a regression model and all the other variables are independent variables in the regression model. These regression models operate under the same assumptions that one would make when performing linear, logistic, or Poison regression models outside of the context of imputing missing data.
- **Step 4:** The missing values for “var” are then replaced with predictions (imputations) from the regression model. When “var” is subsequently used as an independent variable in the regression models for other variables, both the observed and these imputed values will be used.
- **Step 5:** Steps 2–4 are then repeated for each variable that has missing data. The cycling through each of the variables constitutes one iteration or “cycle.” At the end of one cycle all of the missing values have been replaced with predictions from regressions that reflect the relationships observed in the data.
- **Step 6:** Steps 2–4 are repeated for a number of cycles, with the imputations being updated at each cycle.

Reference: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3074241/

MICE Algorithm for Categorical data:

Before going through the steps 1 to 6 in MICE algorithm the following steps must be done in order to impute categorical data.

- **Step 1:** Ordinal Encode the non-null values
- **Step 2:** Use MICE imputation with Gradient Boosting Classifier to impute the ordinal encoded data
- **Step 3:** Convert back from ordinal values to categorical values.
- **Step 4:** Follow steps 1 to 6 in MICE Algorithm. Instead of using Mean imputation for initial strategy use **Mode imputation**.

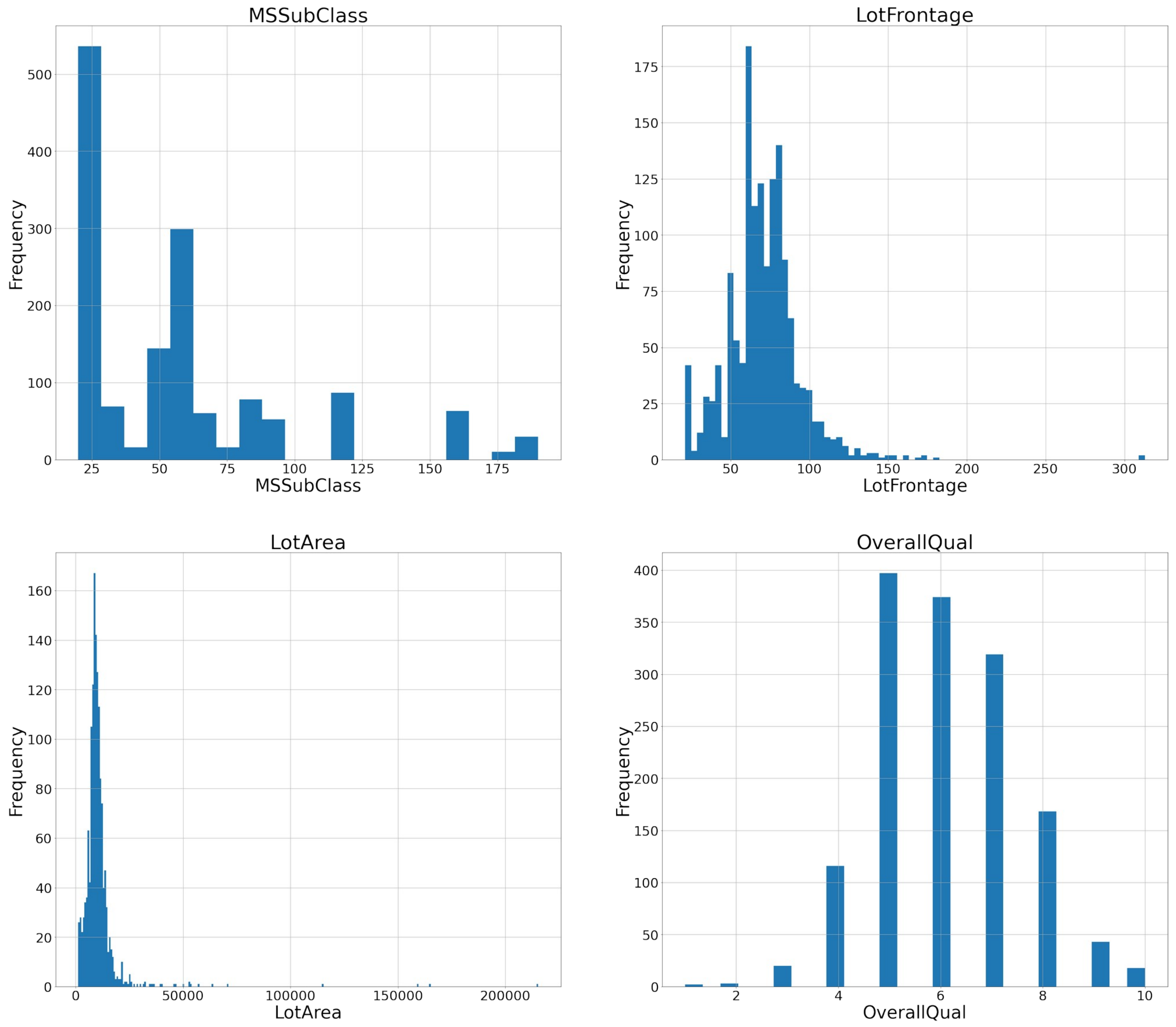
Reference: https://projector-video-pdf-converter.datacamp.com/17404/chapter4.pdf

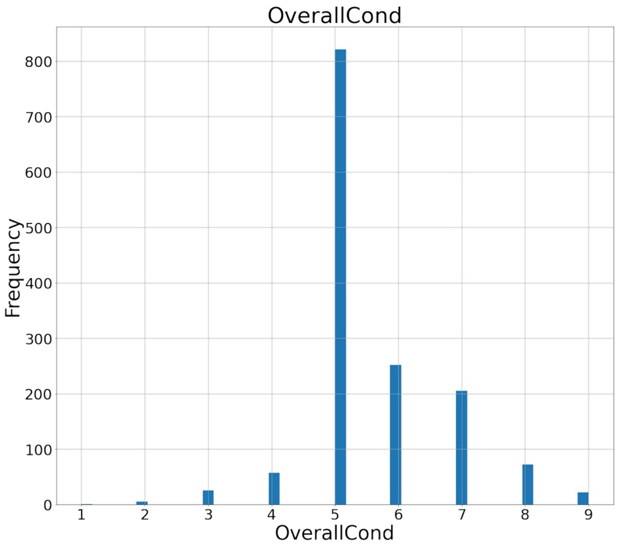
```
def mice_imputation_numeric(train_numeric, test_numeric):
    """
    Impute numeric data using MICE imputation with Gradient Boosting
Regressor.
    """
    iter_imp_numeric = IterativeImputer(GradientBoostingRegressor())
    imputed_train = iter_imp_numeric.fit_transform(train_numeric)
    imputed_test = iter_imp_numeric.transform(test_numeric)
    train_numeric_imp = pd.DataFrame(imputed_train, columns =
train_numeric.columns, index= train_numeric.index)
    test_numeric_imp = pd.DataFrame(imputed_test, columns =
test_numeric.columns, index = test_numeric.index)
    return train_numeric_imp, test_numeric_imp

def mice_imputation_categoric(train_categoric, test_categoric):
    """
    Impute categoric data using MICE imputation with Gradient Boosting
Classifier.
    Steps:
    1. Ordinal Encode the non-null values
    2. Use MICE imputation with Gradient Boosting Classifier to impute
the ordinal encoded data
    3. Inverse transform the ordinal encoded data.
    """
    ordinal_dict={}
    for col in train_categoric:
        '''Ordinal encode train data'''
        ordinal_dict[col] = OrdinalEncoder()
        nn_vals = np.array(train_categoric[col]
[train_categoric[col].notnull()]).reshape(-1,1)
        nn_vals_arr =
np.array(ordinal_dict[col].fit_transform(nn_vals)).reshape(-1,)
        train_categoric[col].loc[train_categoric[col].notnull()] =
nn_vals_arr

    for col in test_categoric:
        '''Ordinal encode test data'''
        nn_vals = np.array(test_categoric[col]
[test_categoric[col].notnull()]).reshape(-1,1)
        nn_vals_arr =
np.array(ordinal_dict[col].transform(nn_vals)).reshape(-1,)
        test_categoric[col].loc[test_categoric[col].notnull()] =
nn_vals_arr

    '''Impute the data using MICE with Gradient Boosting Classifier'''
    iter_imp_categoric = IterativeImputer(GradientBoostingClassifier(),
max_iter =5, initial_strategy='most_frequent')
    imputed_train = iter_imp_categoric.fit_transform(train_categoric)
    imputed_test = iter_imp_categoric.transform(test_categoric)
    train_categoric_imp = pd.DataFrame(imputed_train, columns
=train_categoric.columns,index = train_categoric.index).astype(int)
    test_categoric_imp = pd.DataFrame(imputed_test,
columns=test_categoric.columns,index =test_categoric.index).astype(int)

    '''Inverse Transform'''
    for col in train_categoric_imp.columns:
        oe = ordinal_dict[col]
        train_arr= np.array(train_categoric_imp[col]).reshape(-1,1)
        test_arr = np.array(test_categoric_imp[col]).reshape(-1,1)
        train_categoric_imp[col] = oe.inverse_transform(train_arr)
        test_categoric_imp[col] = oe.inverse_transform(test_arr)

    return train_categoric_imp, test_categoric_imp




train_numeric_imp, test_numeric_imp =
mice_imputation_numeric(train_numeric,test_numeric)
train_categoric_imp, test_categoric_imp =
mice_imputation_categoric(train_categoric, test_categoric)

'''Concatenate Numeric and Categoric Training and Test set data '''
train = pd.concat([train_numeric_imp, train_categoric_imp,
train['SalePrice']], axis = 1)
test = pd.concat([test_numeric_imp, test_categoric_imp], axis =1)
```

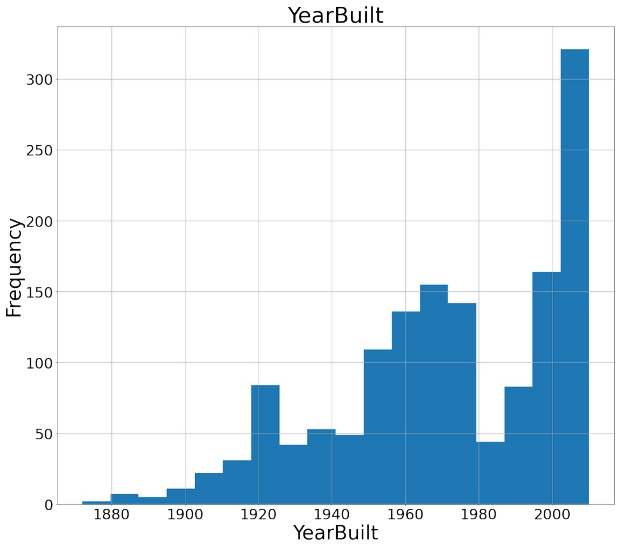
Data Visualization

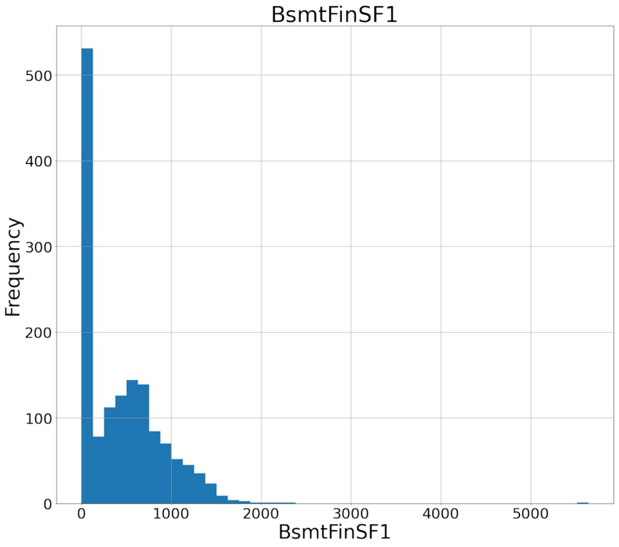
```
def plot_histogram(train, col1, col2, cols_list, last_one =False):
    """
    Plot the histogram for the numerical columns. The bin width
    is calculated by Freedman Diaconis Rule and Sturges rule.

    Freedman-Diaconis Rule:
    Freedman-Diaconis Rule is a rule to find the optimal number of bins.
    Bin width: (2 * IQR)/(N^1/3)
    N - Size of the data
    Number of bins : (Range/ bin-width)
    Disadvantage: The IQR might be zero for certain columns. In
    that case the bin width might be equal to infinity. In that case
    the actual range of the data is returned as bin width.

    Sturges Rule:
    Sturges Rule is a rule to find the optimal number of bins.
    Bin width: (Range/ bin-width)
    N - Size of the data
    Number of bins : ceil(log2(N))+1

    """
    if(col1 in cols_list):
        freq1, bin_edges1 = np.histogram(train[col1],bins='sturges')
    else:
        freq1, bin_edges1 = np.histogram(train[col1],bins='fd')
    if(col2 in cols_list):
        freq2, bin_edges2 = np.histogram(train[col2],bins='sturges')
    else:
        freq2, bin_edges2 = np.histogram(train[col2],bins='fd')

    if(last_one!=True):
        plt.figure(figsize=(45,18))
        ax1 = plt.subplot(1,2,1)
        ax1.set_title(col1,fontsize=45)
        ax1.set_xlabel(col1,fontsize=40)
        ax1.set_ylabel('Frequency',fontsize=40)
        train[col1].hist(bins=bin_edges1,ax = ax1, xlabelsize=30,
ylabelsize=30)

    else:
        plt.figure(figsize=(20,10))
        ax1 = plt.subplot(1,2,1)
        ax1.set_title(col1,fontsize=25)
        ax1.set_xlabel(col1,fontsize=20)
        ax1.set_ylabel('Frequency',fontsize=20)
        train[col1].hist(bins=bin_edges1,ax = ax1, xlabelsize=15,
ylabelsize=15)

    if(last_one != True):
        ax2 = plt.subplot(1,2,2)
        ax2.set_title(col2,fontsize=45)
        ax2.set_xlabel(col2,fontsize=40)
        ax2.set_ylabel('Frequency',fontsize=40)
        train[col2].hist(bins=bin_edges2, ax = ax2, xlabelsize=30,
ylabelsize=30)




'''
These columns have IQR equal to zero. Freedman Diaconis Rule doesn't work
significantly well for these columns.
Use sturges rule to find the optimal number of bins for the columns.
'''
cols_list = ['LowQualFinSF','BsmtFinSF2','BsmtHalfBath','KitchenAbvGr',

'EnclosedPorch','3SsnPorch','ScreenPorch','PoolArea','MiscVal']
```

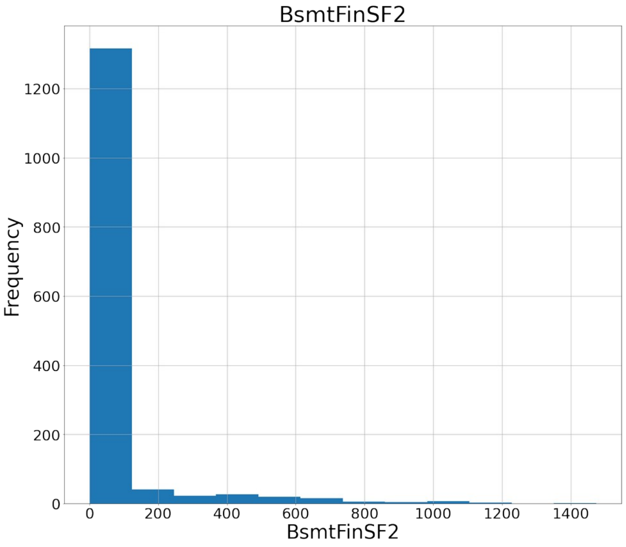
```
# Except ID
hist_cols = numeric_cols[1:]
for i in range(0,len(hist_cols),2):
    if(i == len(hist_cols)-1):
        plot_histogram(train,hist_cols[i],hist_cols[i],cols_list,True)
    else:
        plot_histogram(train,hist_cols[i],hist_cols[i+1],cols_list)
```

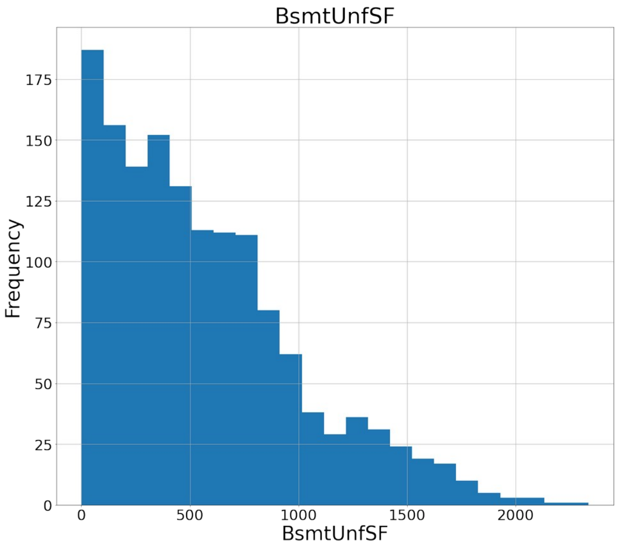


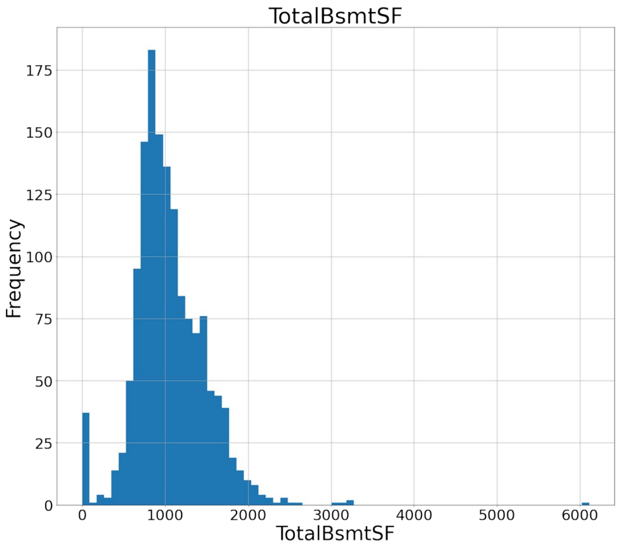

OverallCond
Frequency
0
100
200
300
400
500
600
700
800
1
2
3
4
5
6
7
8
9
OverallCond

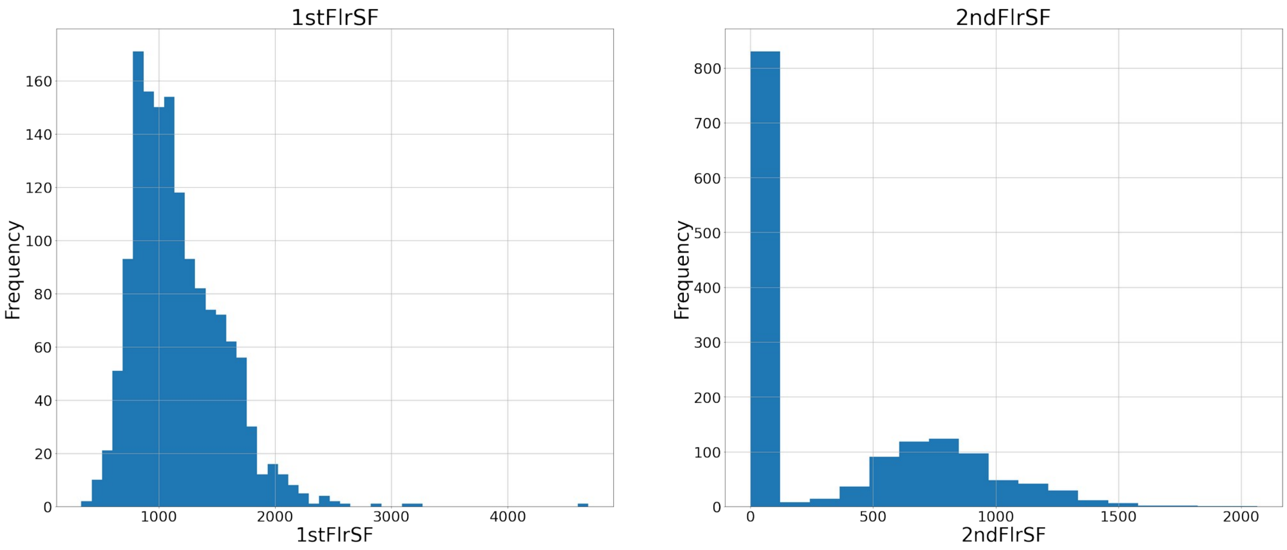

YearBuilt
Frequency
0
50
100
150
200
250
300
1880
1900
1920
1940
1960
1980
2000
YearBuilt

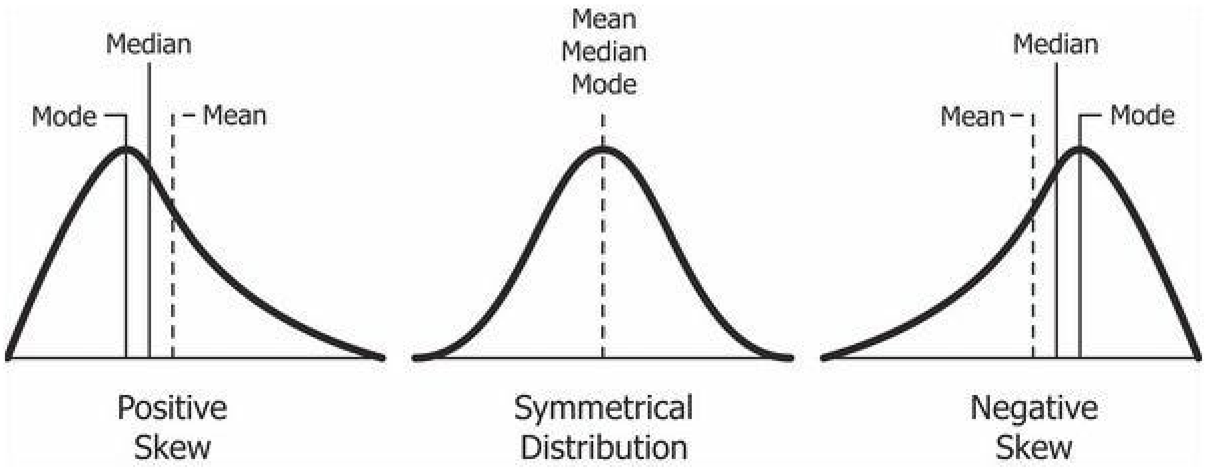

BsmtFinSF1
Frequency
0
100
200
300
400
500
0
1000
2000
3000
4000
5000
BsmtFinSF1


BsmtFinSF2
Frequency
0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200
1400
BsmtFinSF2


BsmtUnfSF
Frequency
0
25
50
75
100
125
150
175
0
500
1000
1500
2000
BsmtUnfSF


TotalBsmtSF
Frequency
0
25
50
75
100
125
150
175
0
1000
2000
3000
4000
5000
6000
TotalBsmtSF





```
    and negative skewed data.Data Preprocessing:
    """
    skew_dict = {}
    for col in numeric_cols:
        skew_dict[col] = train[col].skew()

    skew_dict = dict(sorted(skew_dict.items(),key=itemgetter(1)))
    positive_skew_dict = {k:v for (k,v) in skew_dict.items() if v>0}
    negative_skew_dict = {k:v for (k,v) in skew_dict.items() if v<0}
    return skew_dict, positive_skew_dict, negative_skew_dict

def add_constant(data, highly_pos_skewed):
    """
    Look for zeros in the columns. If zeros are present then the log(0) would
result in -infinity.
    So before transforming it we need to add it with some constant.
    """
    C = 1
    for col in highly_pos_skewed.keys():
        if(col != 'SalePrice'):
            if(len(data[data[col] == 0]) > 0):
                data[col] = data[col] + C
    return data
```

```
def log_transform(data, highly_pos_skewed):
    """
    Log transformation of highly positively skewed columns.
    """
    for col in highly_pos_skewed.keys():
        if(col != 'SalePrice'):
            data[col] = np.log10(data[col])
    return data

def sqrt_transform(data, moderately_pos_skewed):
    """
    Square root transformation of moderately skewed columns.
    """
    for col in moderately_pos_skewed.keys():
        if(col != 'SalePrice'):
            data[col] = np.sqrt(data[col])
    return data

def reflect_sqrt_transform(data, moderately_neg_skewed):
    """
    Reflection and log transformation of highly negatively skewed
    columns.
    """
    for col in moderately_neg_skewed.keys():
        if(col != 'SalePrice'):
            K = max(data[col]) + 1
            data[col] = np.sqrt(K - data[col])
    return data




"""
If skewness is less than -1 or greater than 1, the distribution is highly
skewed.
If skewness is between -1 and -0.5 or between 0.5 and 1, the distribution is
moderately skewed.
If skewness is between -0.5 and 0.5, the distribution is approximately
symmetric.
"""
skew_dict, positive_skew_dict, negative_skew_dict = find_skewness(train,
numeric_cols)
moderately_pos_skewed = {k:v for (k,v) in positive_skew_dict.items() if v>0.5
and v<=1}
highly_pos_skewed = {k:v for (k,v) in positive_skew_dict.items() if v>1}
moderately_neg_skewed = {k:v for (k,v) in negative_skew_dict.items() if v>-1 and
v<=0.5}
highly_neg_skewed = {k:v for (k,v) in negative_skew_dict.items() if v<-1}

'''Transform train data.'''
train = add_constant(train, highly_pos_skewed)
train = log_transform(train, highly_pos_skewed)
train = sqrt_transform(train, moderately_pos_skewed)
train = reflect_sqrt_transform(train, moderately_neg_skewed )
'''Transform test data.'''
test = add_constant(test, highly_pos_skewed)
test = log_transform(test, highly_pos_skewed)
test = sqrt_transform(test, moderately_pos_skewed)
test = reflect_sqrt_transform(test, moderately_neg_skewed )
```

Categorical Encoding

Ordinal Encoding:

Ordinal columns are the ones which have ordinality or inherent order in themselves. Example ratings and feedback like excellent, good, fair, poor.

- Various **Ordinal Encoding techniques** are,
    - Label Encoding
    - Binary Encoding

```
ordinal_col_dicts = {
    'ExterQual':{'TA': 3, 'Gd': 2, 'Ex': 1, 'Fa': 4,'Po':5},
    'ExterCond': {'TA': 3, 'Gd': 2, 'Fa': 4, 'Ex': 1, 'Po': 5},
    'BsmtQual': {'TA': 3, 'Gd': 2, 'Ex': 1, 'Fa': 4,'Po':5},
    'BsmtCond': {'Fa': 4, 'Gd': 2, 'Po': 5, 'TA': 3,'Ex':1, 'NA':6},
    'BsmtExposure': {'No': 4, 'Av': 2, 'Gd': 1, 'Mn': 3,'NA':5},
    'BsmtFinType1': {'Unf': 6, 'GLQ': 1, 'ALQ': 2, 'BLQ': 3, 'Rec': 4,
'LwQ': 5, 'NA':7},
    'BsmtFinType2': {'Unf': 6, 'Rec': 4, 'LwQ': 5, 'BLQ': 3, 'ALQ': 2,
'GLQ': 1, 'NA':7},
    'HeatingQC': {'Ex': 1, 'TA': 3, 'Gd': 2, 'Fa': 4, 'Po': 5},
    'CentralAir': {'Y': 1, 'N': 2},
    'Electrical': {'SBrkr': 1, 'FuseA': 2, 'FuseF': 3, 'FuseP': 4, 'Mix':
5},
    'KitchenQual': {'TA': 3, 'Gd': 2, 'Ex': 1, 'Fa': 4,'Po':5},
    'Functional': {'Typ': 1, 'Min2': 3, 'Min1': 2, 'Mod': 4, 'Maj1': 5,
'Maj2': 6, 'Sev': 7, 'Sal':8},
    'FireplaceQu': {'Gd': 2, 'TA': 3, 'Fa': 4, 'Ex': 1, 'Po': 5},
    'GarageFinish': {'Unf': 3, 'RFn': 2, 'Fin': 1, 'NA':4},
    'GarageQual': {'TA': 3, 'Fa': 4, 'Gd': 2, 'Ex': 1, 'Po': 5},
    'GarageCond': {'TA': 3, 'Fa': 4, 'Gd': 2, 'Po': 5, 'Ex': 1},
    'PavedDrive': {'Y': 1, 'N': 3, 'P': 2},
    'LotShape': {'Reg': 1, 'IR1': 2, 'IR2': 3, 'IR3': 4},
    'Utilities': {'AllPub': 1, 'NoSeWa': 3, 'NoSewr':2, 'ELO':4},
    'LandSlope': {'Gtl': 1, 'Mod': 2, 'Sev': 3},
    'BldgType': {'1Fam': 1, 'TwnhsE': 4, 'Duplex': 3, 'Twnhs': 5,
'2fmCon': 2},
    'HouseStyle': {'1Story': 1,  '2Story': 4, '1.5Fin': 2, 'SLvl': 8,
'SFoyer': 7, '1.5Unf': 3, '2.5Unf': 6, '2.5Fin': 5},
    'LotConfig': {'Inside': 1, 'Corner': 2, 'CulDSac': 3, 'FR2': 4,
'FR3': 5}
}

def ordinal_encode(data, ordinal_col_dicts):
    """
    Ordinal encode the ordinal columns according to the values in
    ordinal_col_dicts.
    """
    for ord_col in ordinal_col_dicts:
        ord_dict = ordinal_col_dicts[ord_col]
        data[ord_col] = data[ord_col].map(ord_dict)
    return data

train = ordinal_encode(train, ordinal_col_dicts)
test = ordinal_encode(test, ordinal_col_dicts)
```

Nominal Encoding:

Nominal columns are the ones which does not have any ordinality or inherent order. Example country names, gender (male, female).

- Various **Nominal Encoding techniques** available are,
    - Frequency Encoding
    - Target Encoding
    - MEstimate Encoding
    - Leave One Out Encoding
    - One-Hot Encoding

Target Encoding:

Target encoding is the process of replacing a categorical value with the mean of the target variable.

```
def target_encode(train, test):
    """
    Target encoding uses the mean of the target to encode
    categorical data.
    """
    target_enc = TargetEncoder()
    x_train, y_train = train[train.columns[:-1]], train[train.columns[-1]]
    x_train = target_enc.fit_transform(x_train,y_train)
    test = target_enc.transform(test)
    train = pd.concat([x_train, y_train], axis = 1)
    return train, test

train, test = target_encode(train, test)
```

Normalization:

- Normalization is also called as **Feature Scaling**. Normalization scales the values of features between a certain interval. Eg: [0,1]

```
def standard_scale(train, test):
    """
    Built - in function to normalize data.
    """
    ss = StandardScaler()
    x_train, y_train = train[train.columns[:-1]], train[train.columns[-
1]]
    x_train =
pd.DataFrame(ss.fit_transform(x_train),columns=x_train.columns,index=x_tr
ain.index)
    test =
pd.DataFrame(ss.transform(test),columns=test.columns,index=test.index)
    return x_train, y_train, test

x_train, y_train, test =standard_scale(train, test)
```

Data Modeling

Fit XGBoost Regressor model to the preprocessed data.

```
def fit_model(x_train,y_train, model):
    """
    Fits x_train to y_train for the given
    model.
    """
    model.fit(x_train,y_train)
    return model

'''Xtreme Gradient Boosting Regressor'''
model = xgboost.XGBRegressor(objective="reg:squarederror",
random_state=42)
model = fit_model(x_train,y_train, model)
'''Predict the outcomes'''
predictions = model.predict(test)
```

```
submission = pd.read_csv('../input/house-prices-advanced-regression-
techniques/sample_submission.csv')
submission['SalePrice'] = predictions
submission.to_csv('submission.csv',index=False)
```

Conclusion:

In the quest to build a house price prediction model, we have embarked on a critical journey that begins with loading and preprocessing the dataset.We have traversed through essentialsteps, starting with importing the necessary libraries to facilitate data manipulation and analysis.

Understanding the data's structure, characteristics, and any potential issues through exploratory data analysis (EDA) is essential for informed decision-making.

Data preprocessing emerged as a pivotal aspect of this process. It involves cleaning, transforming, and refining the dataset to ensurethat it aligns with the requirements of machine learning algorithms.

With these foundational steps completed, our dataset is now primed for the subsequent stages of building and training a house price prediction model.