Team 4: Housing Price Prediction

Git Link: AA Case Study

Problem Statement:

It is your job to predict the sales price for each house. For each Id in the test set, you must predict the value of the SalePrice variable.

Goal:

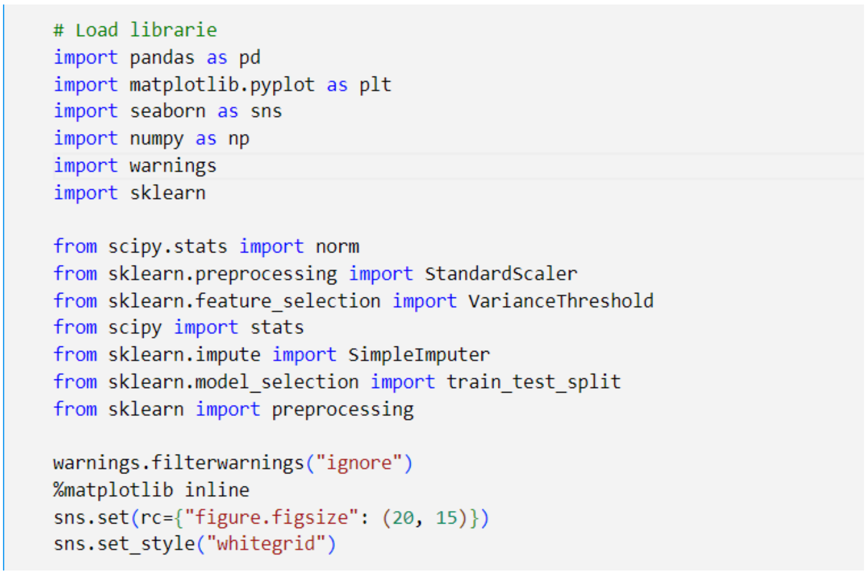
It is your job to predict the sales price for each house. For each Id in the test set, you must predict the value of the SalePrice variable.
Submissions are evaluated on Root-Mean-Squared-Error (RMSE) between the logarithm of the predicted value and the logarithm of the observed sales price. (Taking logs means that errors in predicting expensive houses and cheap houses will affect the result equally.)

In this case study we did EDA, feature selection and modelling to predict housing prices using Advanced Regression Analysis. We have listed out the steps below:

Sr. No.	Title	
1	**Exploratory Data Analysis**	
	1.1	General Exploration
	1.2	Numerical Features 1.2.1 Explore and clean Numerical features 1.2.2. Missing data of Numerical features
	1.3	Categorical features 1.3.1. Explore and clean Categorical features 1.3.2. Missing data of Categorical features 1.3.3. Transform Categorical features into Binary features
	1.4	Merge numerical and binary features into one data set
	1.5	Drop outliers from the train set
2	**Feature engineering**	
3	**Preparing data for modelling**	
	3.1	Target variable 'SalePrice'
	3.2	Split data into train and test and Standardization
	3.3	Backward Stepwise Regression
4	**Modelling**	
	4.1	Models and metrics selection
	4.2	Hyperparameters tuning and model optimization 4.2.1. Ridge regression 4.2.2. Lasso regression 4.2.3. XGBoost regression
	4.3	Choosing the best model
	4.4	Prediction on 'House Prices-Advanced Regression Techniques' test data set
5	**Conclusion**	

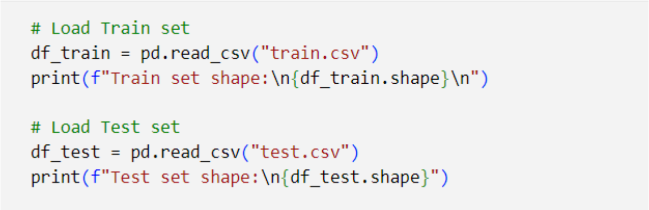
1. Exploratory Data Analysis

1.1 General Exploration



```
# Load librarie
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import warnings
import sklearn

from scipy.stats import norm
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import VarianceThreshold
from scipy import stats
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

warnings.filterwarnings("ignore")
%matplotlib inline
sns.set(rc={"figure.figsize": (20, 15)})
sns.set_style("whitegrid")
```

Loading data and reading:

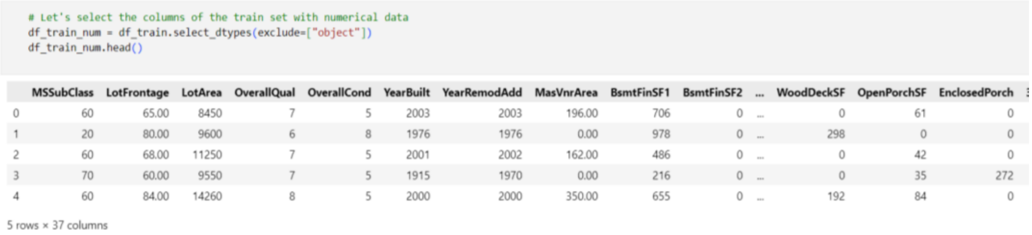


```
# Load Train set
df_train = pd.read_csv("train.csv")
print(f"Train set shape:\n{df_train.shape}\n")

# Load Test set
df_test = pd.read_csv("test.csv")
print(f"Test set shape:\n{df_test.shape}")
```

The only column present in the train set and absent in the test set is 'SalePrice' which is the target variable to be predicted.

1.2 Numerical Features:

1.2.1 Explore and clean numerical features

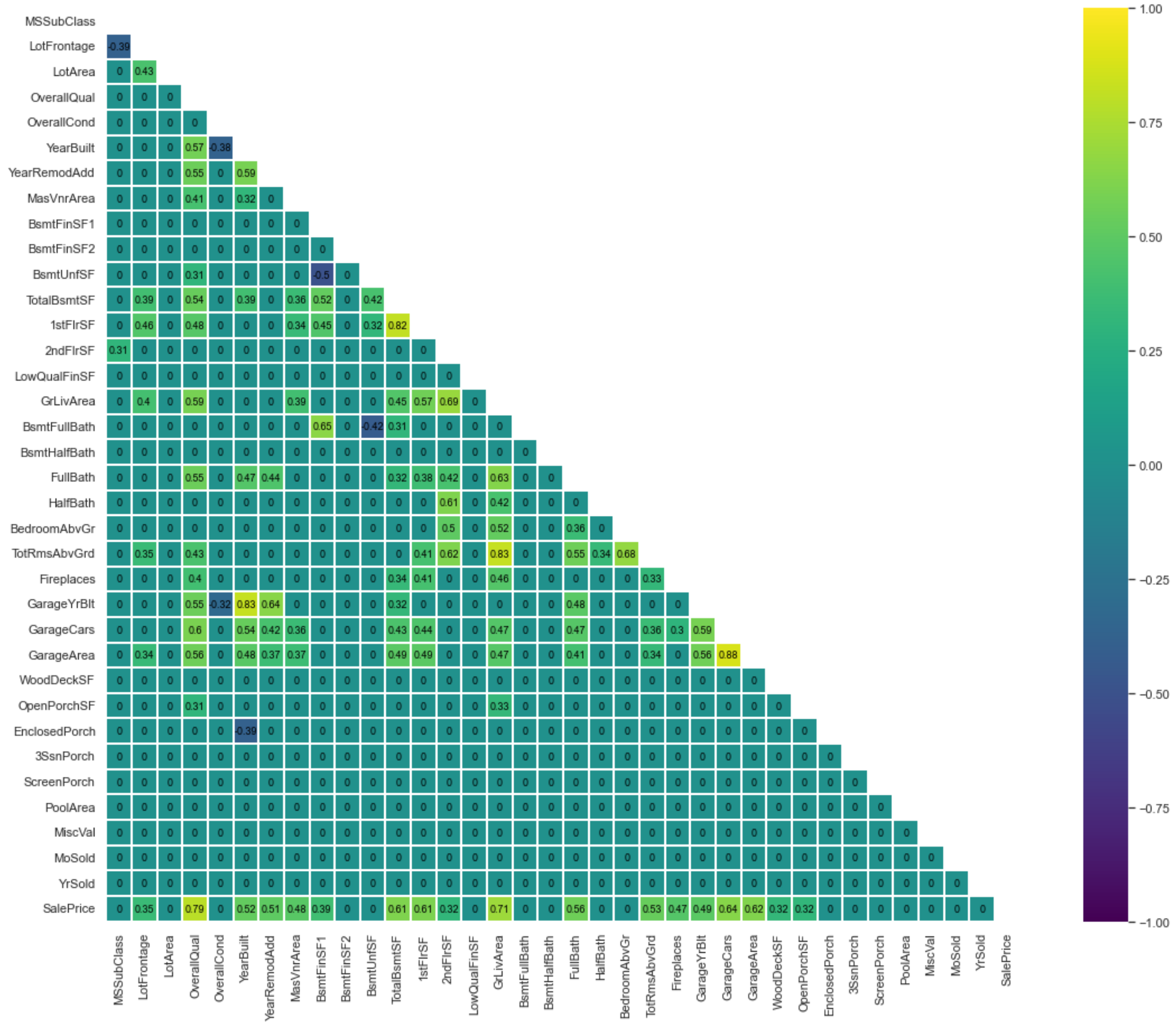


```
# Let's select the columns of the train set with numerical data
df_train_num = df_train.select_dtypes(exclude=["object"])
df_train_num.head()
```

	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	...	WoodDeckSF	OpenPorchSF	EnclosedPorch
0	60	65.00	8450	7	5	2003	2003	196.00	706	0	...	0	61	0
1	20	80.00	9600	6	8	1976	1976	0.00	978	0	...	298	0	0
2	60	68.00	11250	7	5	2001	2002	162.00	486	0	...	0	42	0
3	70	60.00	9550	7	5	1915	1970	0.00	216	0	...	0	35	272
4	60	84.00	14260	8	5	2000	2000	350.00	655	0	...	192	84	0

5 rows × 37 columns

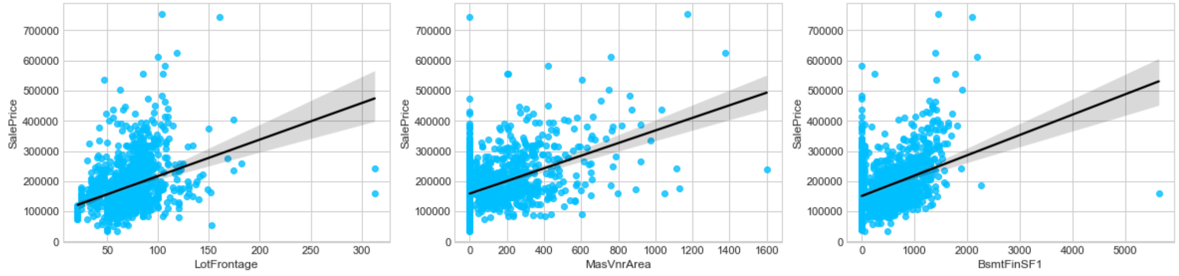
Heatmap for all the remaining numerical data including the taget 'SalePrice'



From the distribution of each numerical variables as well as the heatmap we can notice 18 features that are important and correlated (correlation higher than absolute 0.3) with our target variable 'SalePrice'.

We can also notice that a lot of features are correlated with each other. We will handle these correlations while selecting the features for our models.

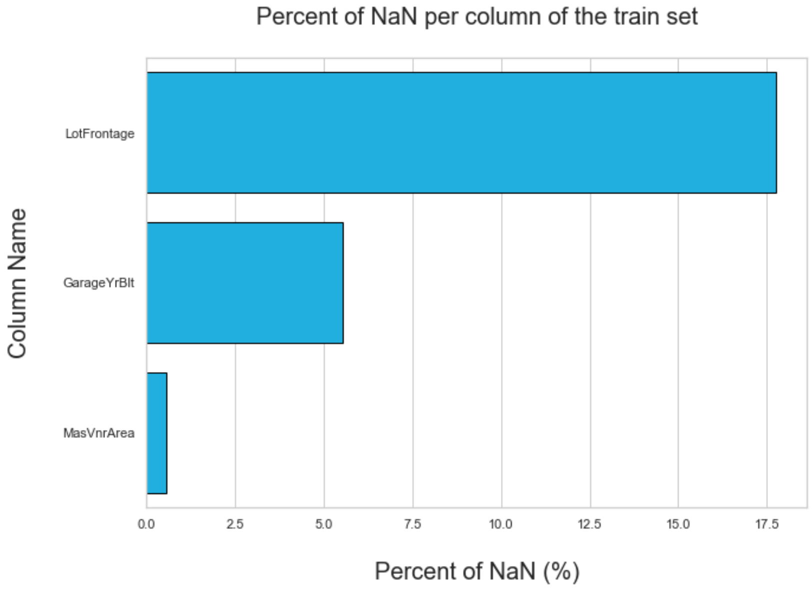
We separated features with high co-relation and low co-relation with 'SalePrice'.



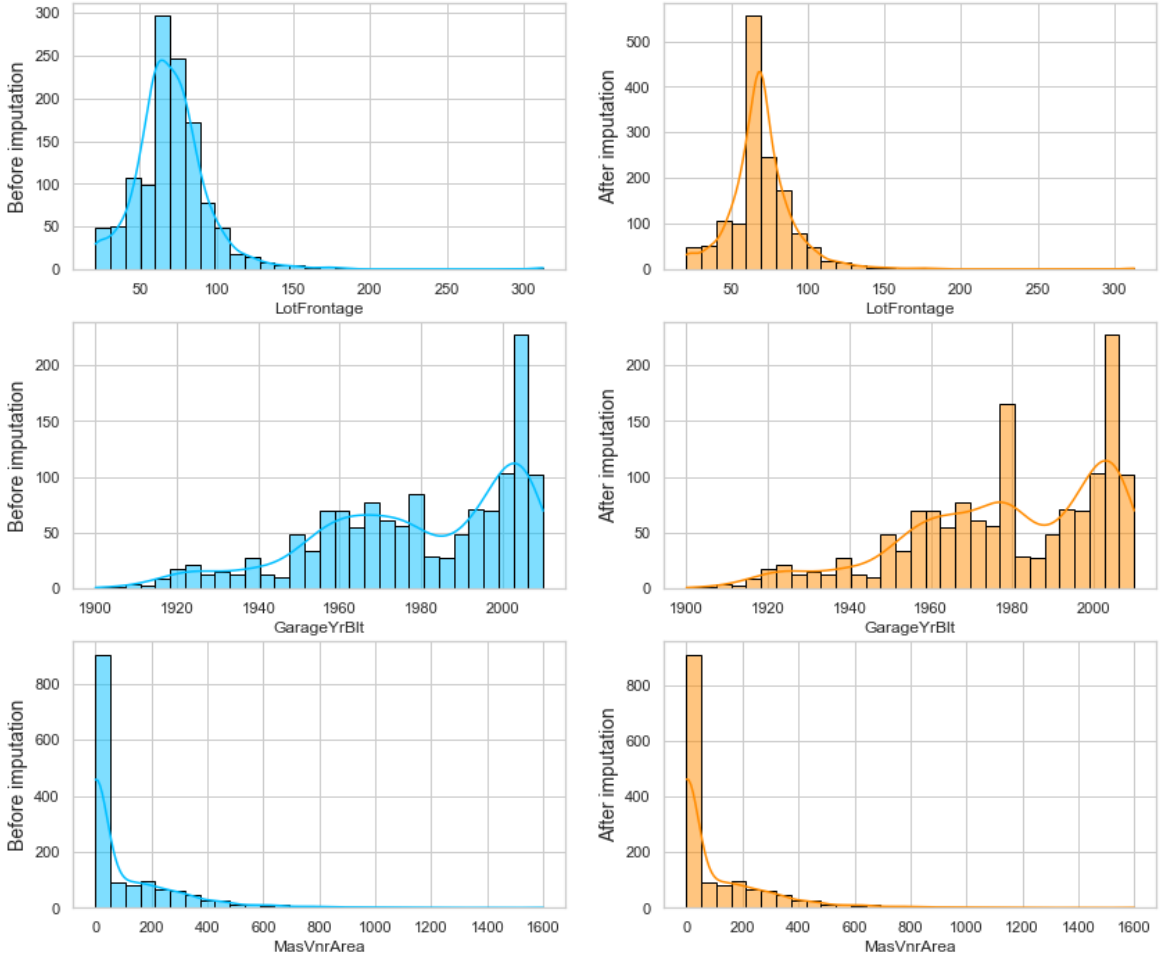
Here we kept the 18 most correlated numerical features with 'SalePrice'

1.2.2 Missing data of Numerical Features

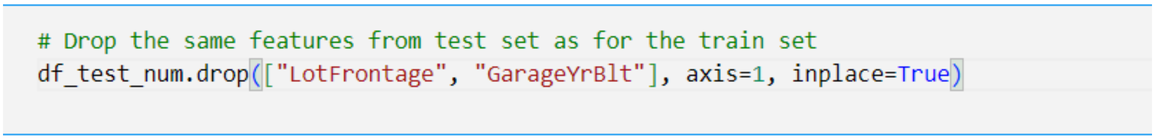
```
Check the NaN of the train set by ploting percent of missing values per column
```



Imputation of missing values (NaNs) with SimpleImputer
Checking the distribution of each imputed feature before and after imputation

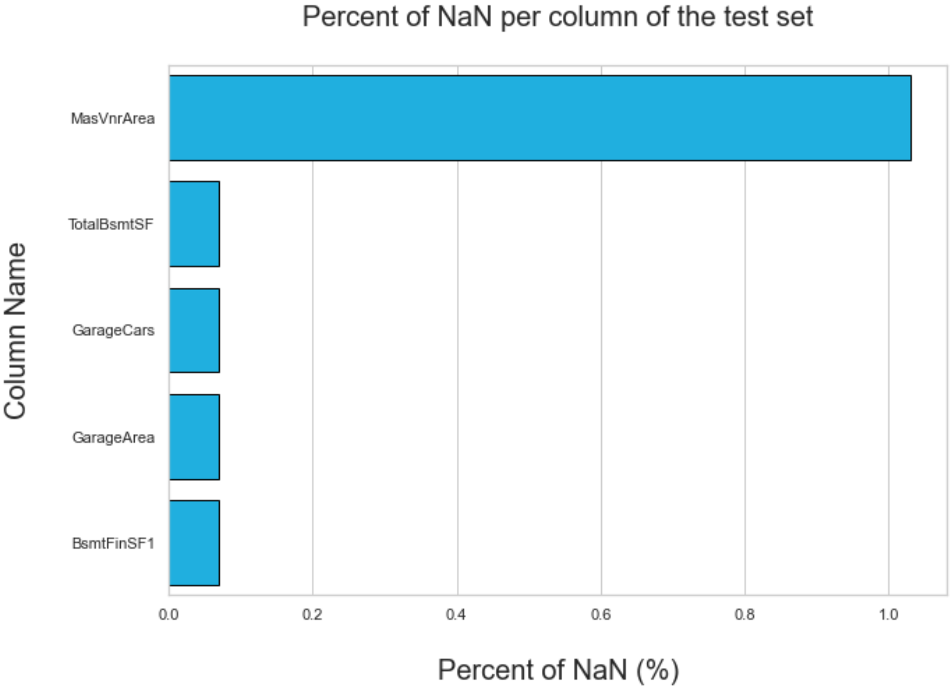


Drop the same features from test set as for the train set



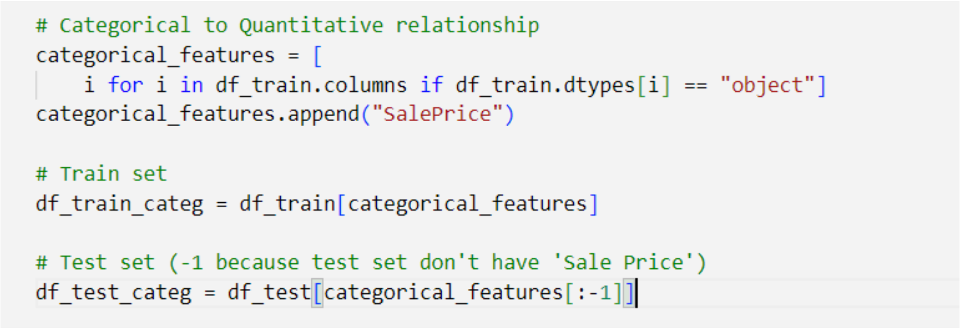
```
# Drop the same features from test set as for the train set
df_test_num.drop(["LotFrontage", "GarageYrBlt"], axis=1, inplace=True)
```

Check the NaN of the test set by ploting percent of missing values per column



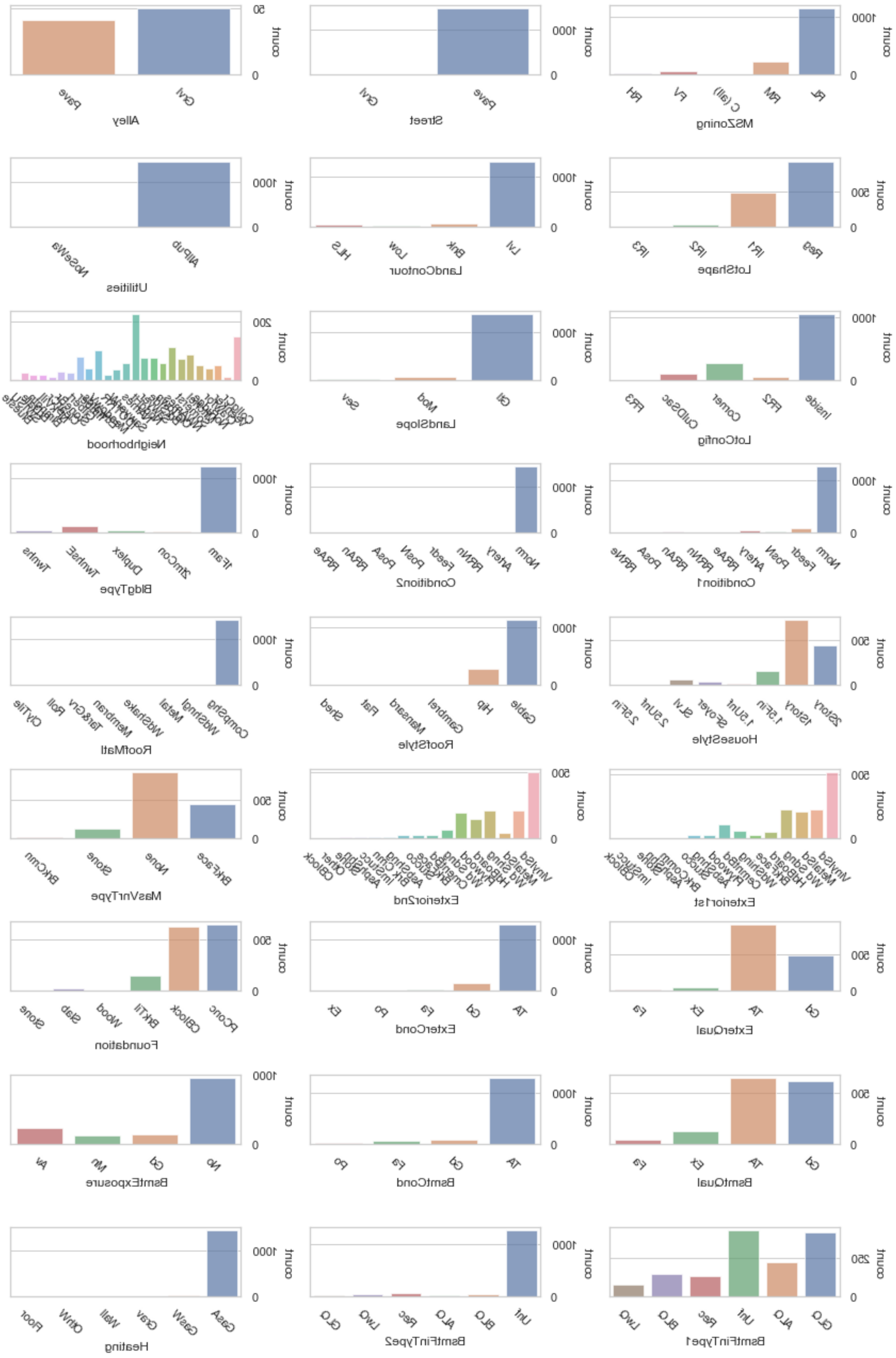
1.3 Categorical Features

1.3.1 Explore and clean Categorical Features



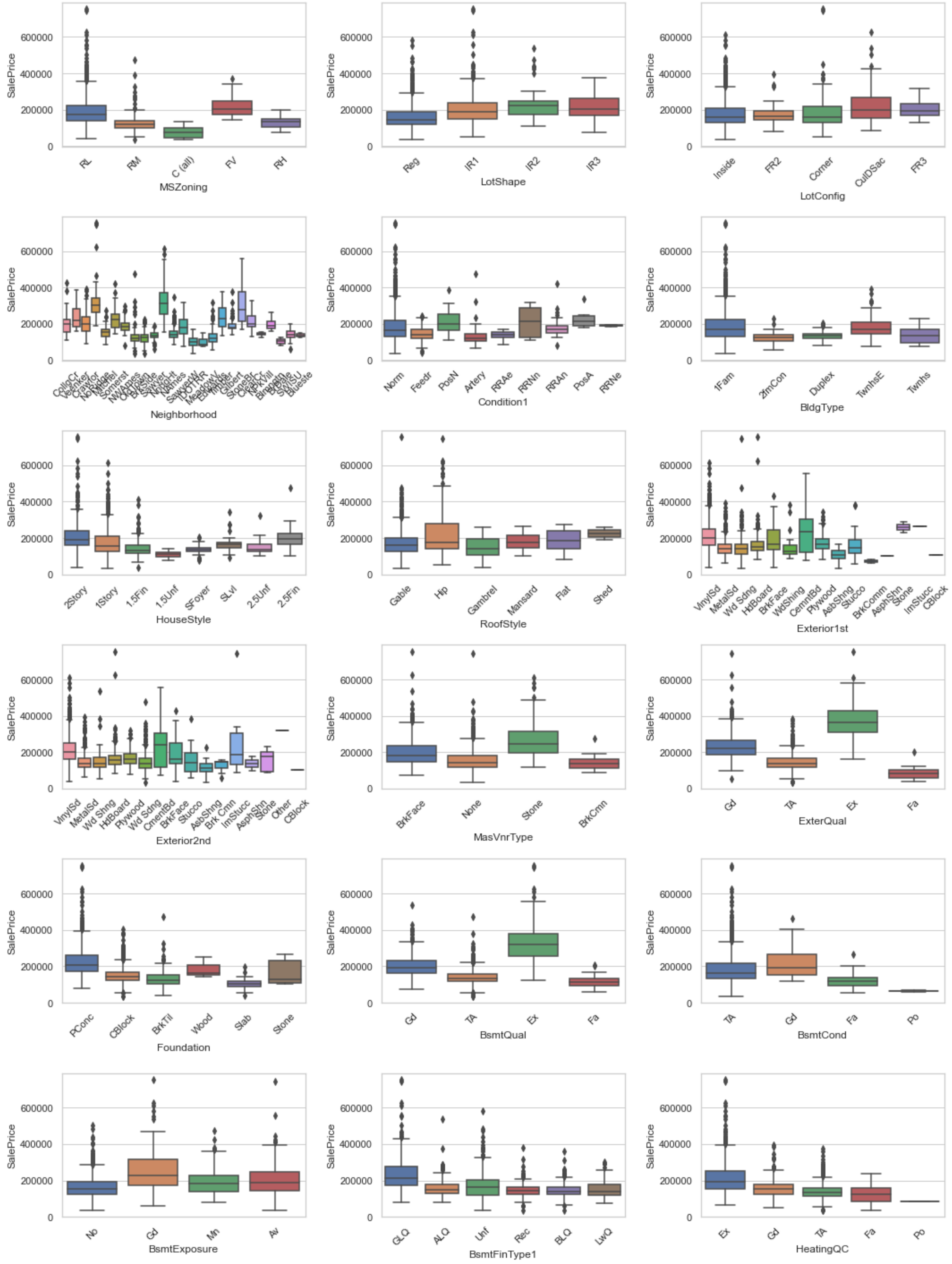
```
# Categorical to Quantitative relationship
categorical_features = [
    i for i in df_train.columns if df_train.dtypes[i] == "object"]
categorical_features.append("SalePrice")

# Train set
df_train_categ = df_train[categorical_features]

# Test set (-1 because test set don't have 'Sale Price')
df_test_categ = df_test[categorical_features[:-1]]
```

Count plot for each of the categorical features in the train set –

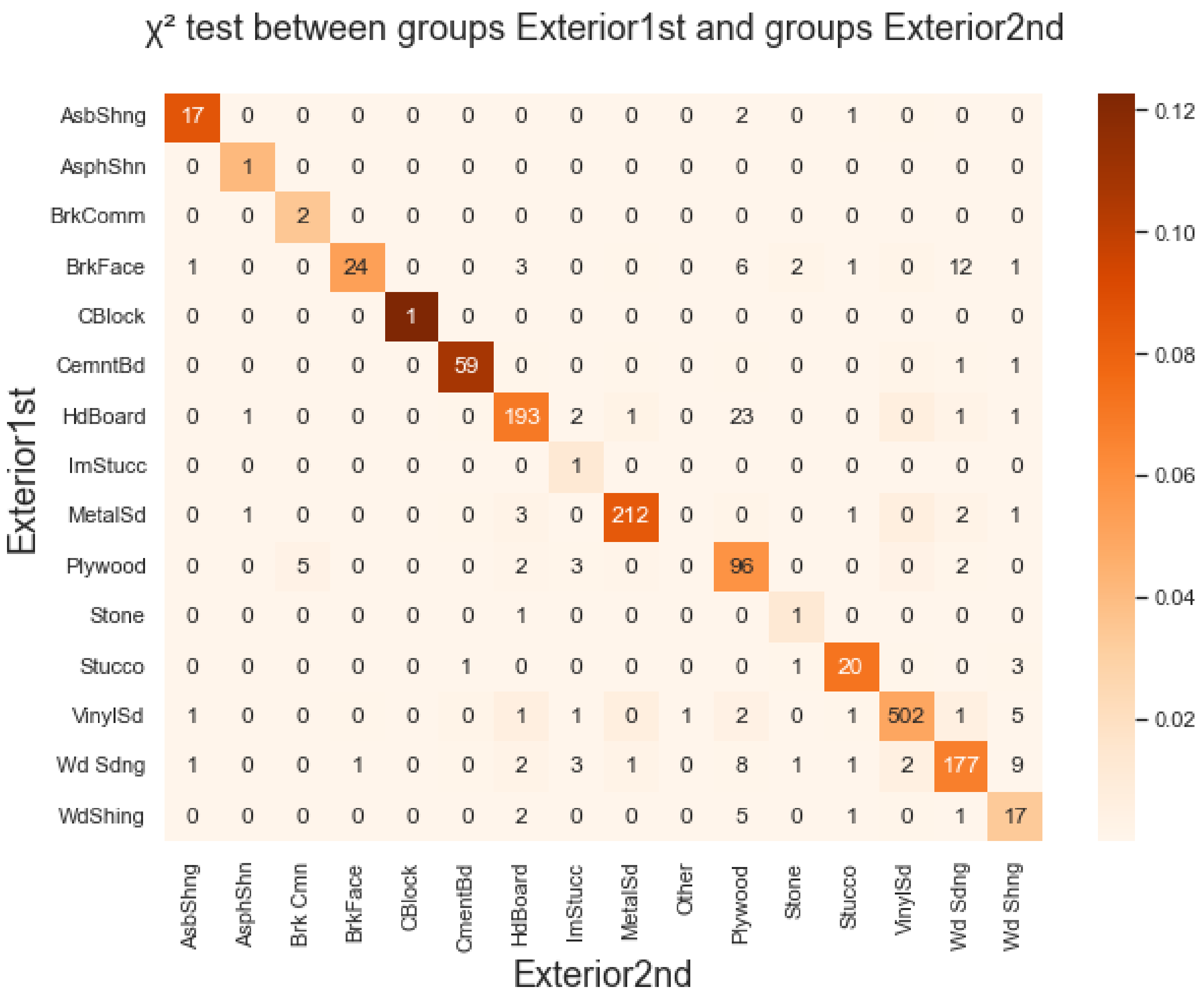


By looking closely at each of the count plots we can notice that for some categorical features, the observations are concentrated in a single level of the category. These features are less informative for our model, so it would be better to remove them.

With the boxplot we can see the variation of the target 'SalePrice' in each of the categorical features.



Some of these features seem to be codependent such as 'Exterior1st' & ‘Exterior2nd’, 'BsmtQual' & 'BsmtCond', 'MasVnrType' & 'ExterQual' etc.
So let's plot the contingency table and perform the Chi square test in order to identify these co-dependency.

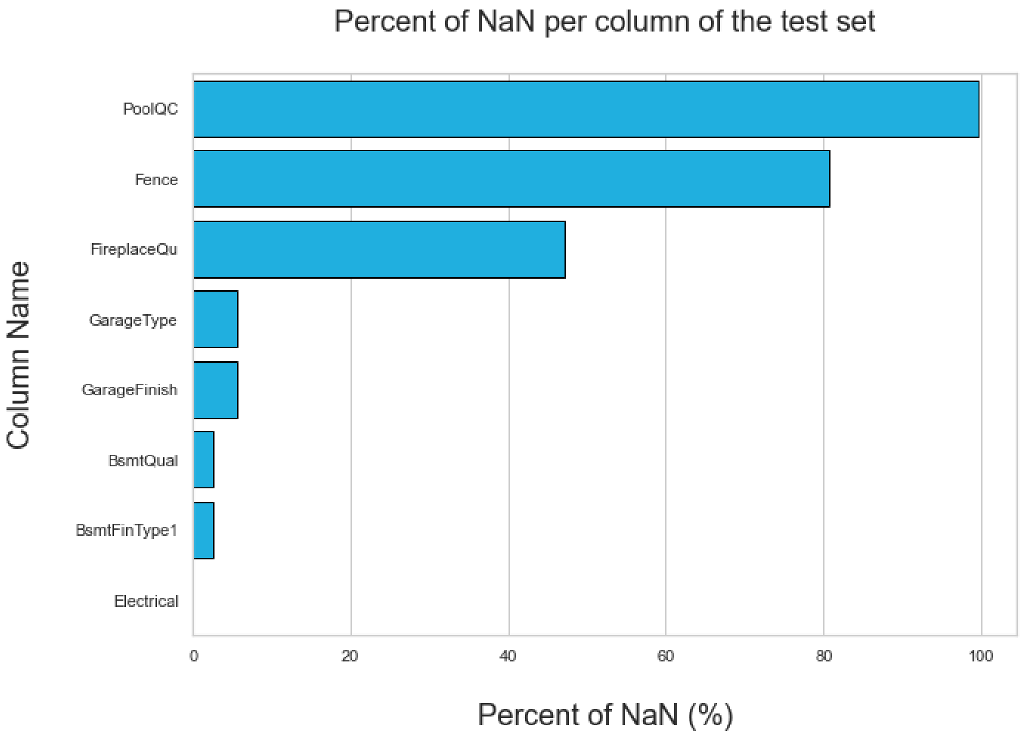


χ² test between groups Exterior1st and groups Exterior2nd

	AsbShng	AsphShn	Brk Cmn	BrkFace	CBlock	CmentBd	HdBoard	ImStucc	MetalSd	Other	Plywood	Stone	Stucco	VinylSd	Wd Sdng	Wd Shng
AsbShng	17	0	0	0	0	0	0	0	0	0	2	0	1	0	0	0
AsphShn	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BrkComm	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
BrkFace	1	0	0	24	0	0	3	0	0	0	6	2	1	0	12	1
CBlock	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
CemntBd	0	0	0	0	0	59	0	0	0	0	0	0	0	0	1	1
HdBoard	0	1	0	0	0	0	193	2	1	0	23	0	0	0	1	1
ImStucc	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
MetalSd	0	1	0	0	0	0	3	0	212	0	0	0	1	0	2	1
Plywood	0	0	5	0	0	0	2	3	0	0	96	0	0	0	2	0
Stone	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
Stucco	0	0	0	0	0	1	0	0	0	0	0	1	20	0	0	3
VinylSd	1	0	0	0	0	0	1	1	0	1	2	0	1	502	1	5
Wd Sdng	1	0	0	1	0	0	2	3	1	0	8	1	1	2	177	9
WdShing	0	0	0	0	0	0	2	0	0	0	5	0	1	0	1	17

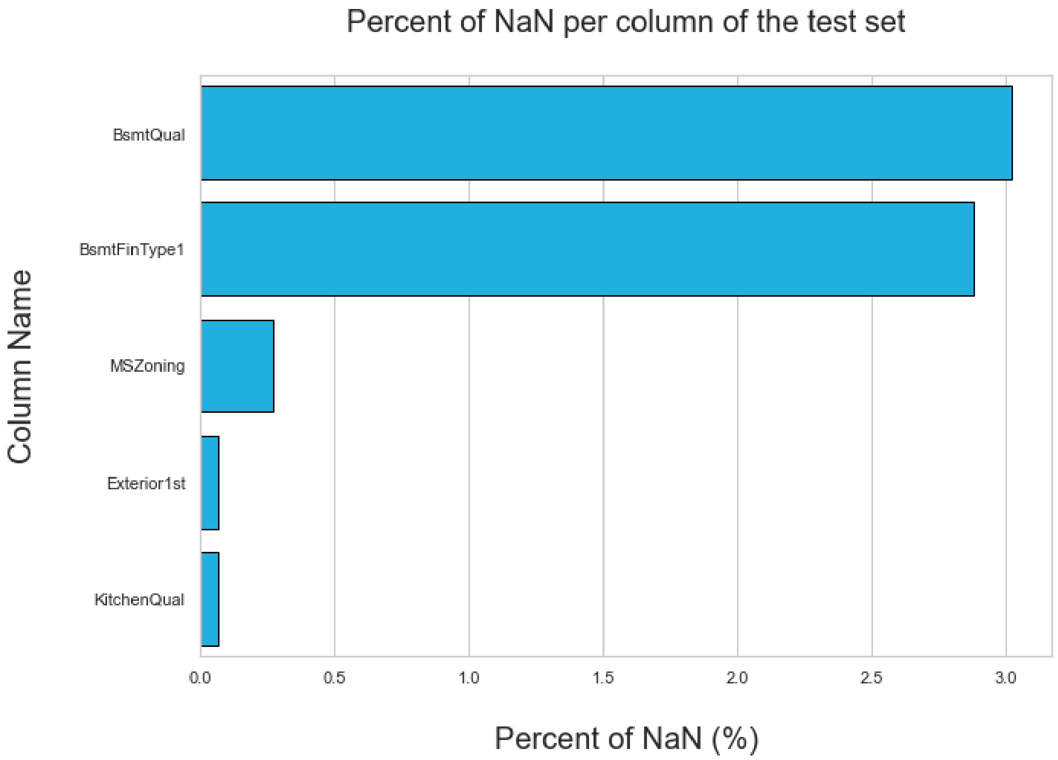
1.3.2 Missing data of Categorical Features

Train Set:

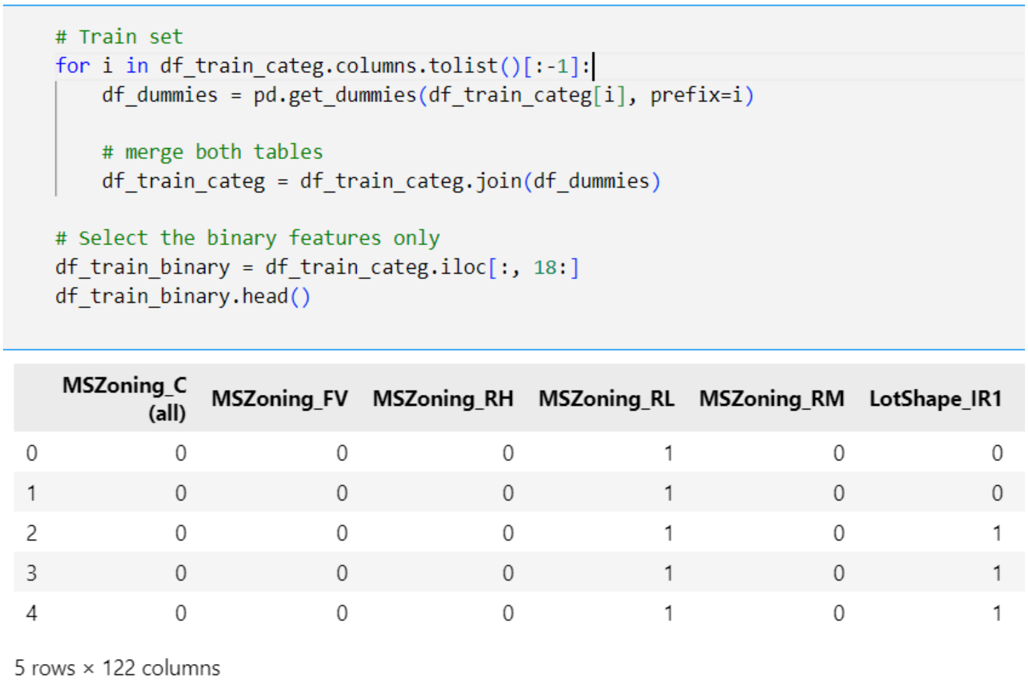


Drop the features where the percentage of NaN is higher than 5% to avoid introducing any error. Then impute the NaN of 'BsmtQual' and 'Electrical' by the corresponding modal.

Test Set:



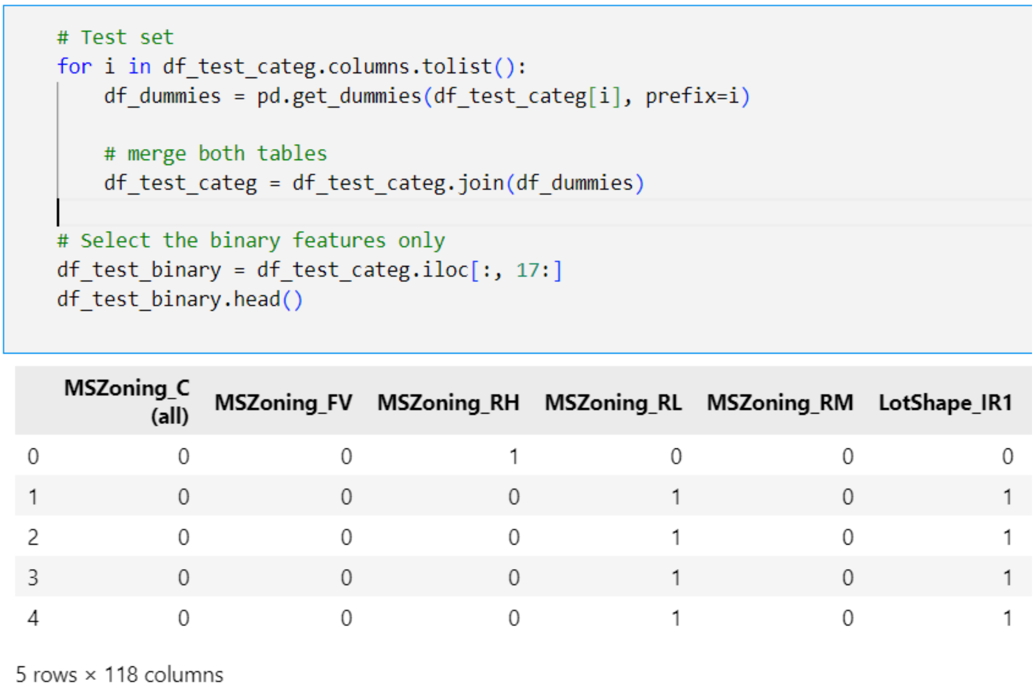
1.3.3 Transform Categorical features into Binary features using get_dummies in both training and testing set.



```
# Train set
for i in df_train_categ.columns.tolist()[:-1]:
    df_dummies = pd.get_dummies(df_train_categ[i], prefix=i)

    # merge both tables
    df_train_categ = df_train_categ.join(df_dummies)

# Select the binary features only
df_train_binary = df_train_categ.iloc[:, 18:]
df_train_binary.head()
```

	MSZoning_C (all)	MSZoning_FV	MSZoning_RH	MSZoning_RL	MSZoning_RM	LotShape_IR1
0	0	0	0	1	0	0
1	0	0	0	1	0	0
2	0	0	0	1	0	1
3	0	0	0	1	0	1
4	0	0	0	1	0	1

5 rows × 122 columns



```
# Test set
for i in df_test_categ.columns.tolist():
    df_dummies = pd.get_dummies(df_test_categ[i], prefix=i)

    # merge both tables
    df_test_categ = df_test_categ.join(df_dummies)

# Select the binary features only
df_test_binary = df_test_categ.iloc[:, 17:]
df_test_binary.head()
```

	MSZoning_C (all)	MSZoning_FV	MSZoning_RH	MSZoning_RL	MSZoning_RM	LotShape_IR1
0	0	0	1	0	0	0
1	0	0	0	1	0	1
2	0	0	0	1	0	1
3	0	0	0	1	0	1
4	0	0	0	1	0	1

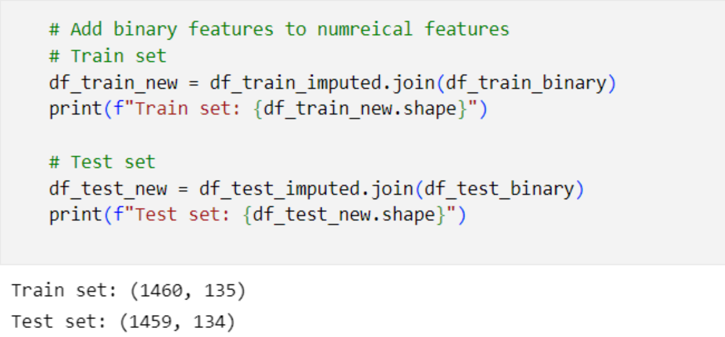
5 rows × 118 columns

We can notice that in df_test_binary there is 118 columns while in df_train_binary there is 122. Let's see which columns are missing from df_test_binary

Features present in df_train_categ and absent in df_test_categ: ['HouseStyle_2.5Fin', 'Exterior1st_ImStucc', 'Exterior1st_Stone', 'Electrical_Mix'] Features present in df_test_categ set and absent in df_train_categ: []

Four of the binary features are absent from the test set. Thus, these features will be dropped from the train in order to have the same columns in both data set.

1.4 Merge Numerical and Binary Features into one dataset.

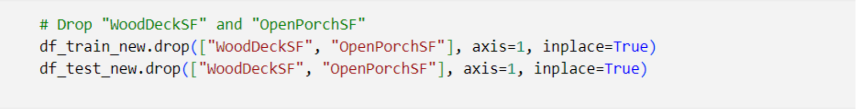


```
# Add binary features to numreical features
# Train set
df_train_new = df_train_imputed.join(df_train_binary)
print(f"Train set: {df_train_new.shape}")

# Test set
df_test_new = df_test_imputed.join(df_test_binary)
print(f"Test set: {df_test_new.shape}")
```

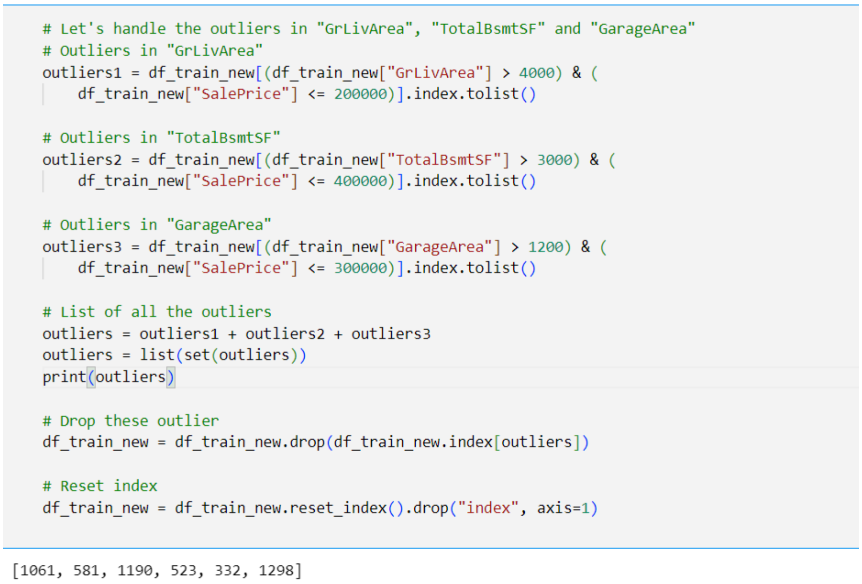
```
Train set: (1460, 135)
Test set: (1459, 134)
```

1.5 Drop the outliers from train set.

Previoulsy in the part 'I.2. Numerical Features' of this notebook I noticed some houses with large surface ("GrLivArea", "TotalBsmtSF" and "GarageArea") and with a very low Price. It is better for our models to drop theese outliers.
We also noticed for both features "WoodDeckSF" and "OpenPorchSF" a high number of 0 values with a correspondingly high price variation. These outliers should be deleted. However, since the number of these outliers is very important, the best thing to do is to drop these columns.



```
# Drop "WoodDeckSF" and "OpenPorchSF"
df_train_new.drop(["WoodDeckSF", "OpenPorchSF"], axis=1, inplace=True)
df_test_new.drop(["WoodDeckSF", "OpenPorchSF"], axis=1, inplace=True)
```

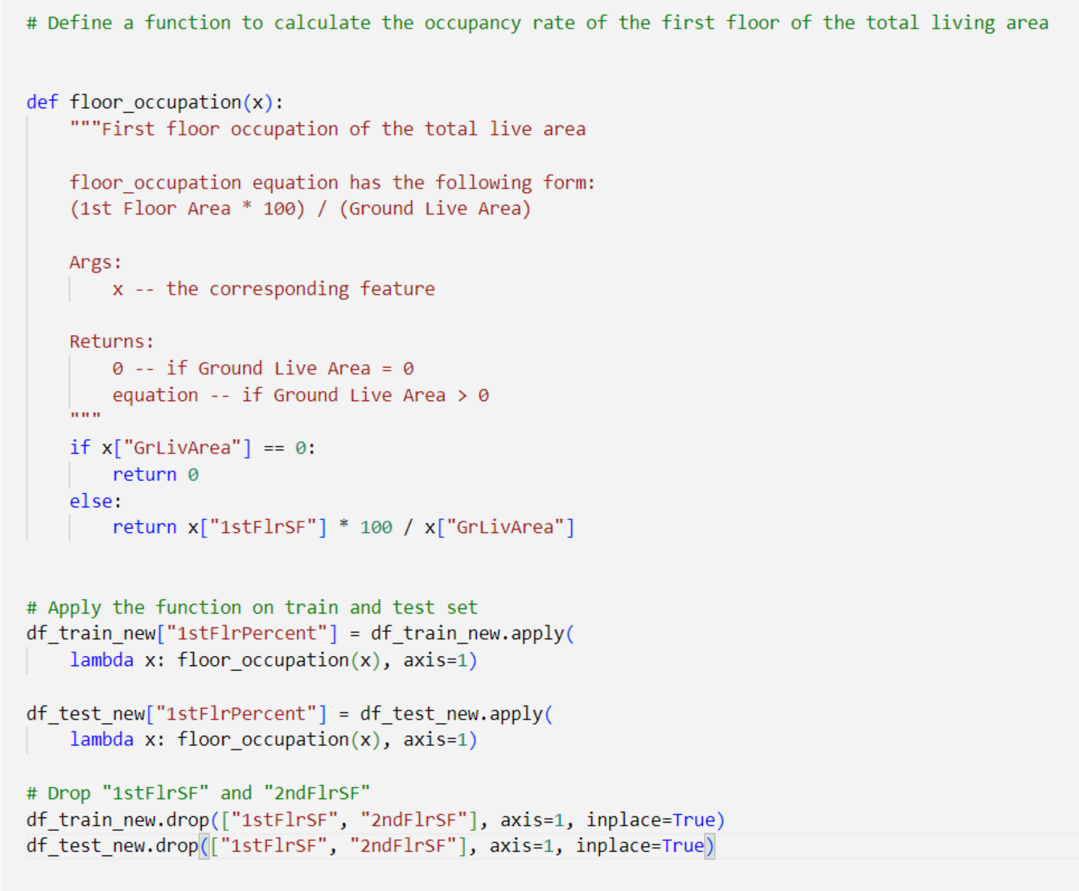


```
# Let's handle the outliers in "GrLivArea", "TotalBsmtSF" and "GarageArea"
# Outliers in "GrLivArea"
outliers1 = df_train_new[(df_train_new["GrLivArea"] > 4000) & (
    df_train_new["SalePrice"] <= 200000)].index.tolist()

# Outliers in "TotalBsmtSF"
outliers2 = df_train_new[(df_train_new["TotalBsmtSF"] > 3000) & (
    df_train_new["SalePrice"] <= 400000)].index.tolist()

# Outliers in "GarageArea"
outliers3 = df_train_new[(df_train_new["GarageArea"] > 1200) & (
    df_train_new["SalePrice"] <= 300000)].index.tolist()

# List of all the outliers
outliers = outliers1 + outliers2 + outliers3
outliers = list(set(outliers))
print(outliers)

# Drop these outlier
df_train_new = df_train_new.drop(df_train_new.index[outliers])

# Reset index
df_train_new = df_train_new.reset_index().drop("index", axis=1)
```

```
[1061, 581, 1190, 523, 332, 1298]
```

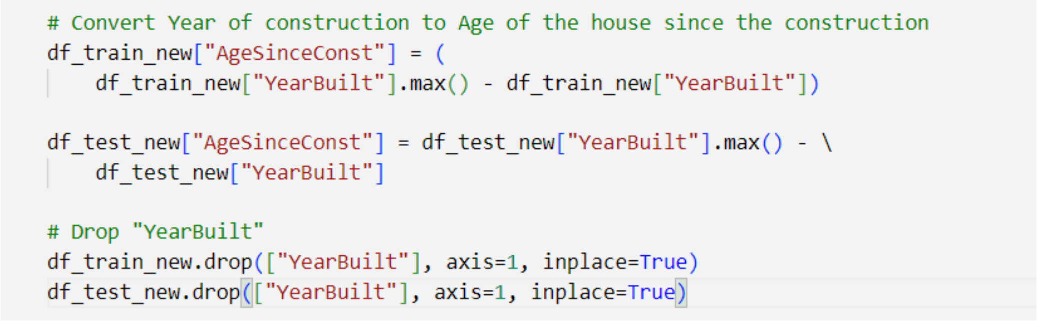
2. Feature Engineering

While looking closely at the remaining features, we can notice that several of them designate a given surface of the property. Thus, we will try to combine some of these surfaces into indicators without losing the information they provide.



```python
# Define a function to calculate the occupancy rate of the first floor of the total living area


def floor_occupation(x):
    """First floor occupation of the total live area

    floor_occupation equation has the following form:
    (1st Floor Area * 100) / (Ground Live Area)

    Args:
        x -- the corresponding feature

    Returns:
        0 -- if Ground Live Area = 0
        equation -- if Ground Live Area > 0
    """
    if x["GrLivArea"] == 0:
        return 0
    else:
        return x["1stFlrSF"] * 100 / x["GrLivArea"]


# Apply the function on train and test set
df_train_new["1stFlrPercent"] = df_train_new.apply(
    lambda x: floor_occupation(x), axis=1)

df_test_new["1stFlrPercent"] = df_test_new.apply(
    lambda x: floor_occupation(x), axis=1)

# Drop "1stFlrSF" and "2ndFlrSF"
df_train_new.drop(["1stFlrSF", "2ndFlrSF"], axis=1, inplace=True)
df_test_new.drop(["1stFlrSF", "2ndFlrSF"], axis=1, inplace=True)
```

In addition, we will turn years into age, e.g., year of construction will be transformed into age of the house since the construction.



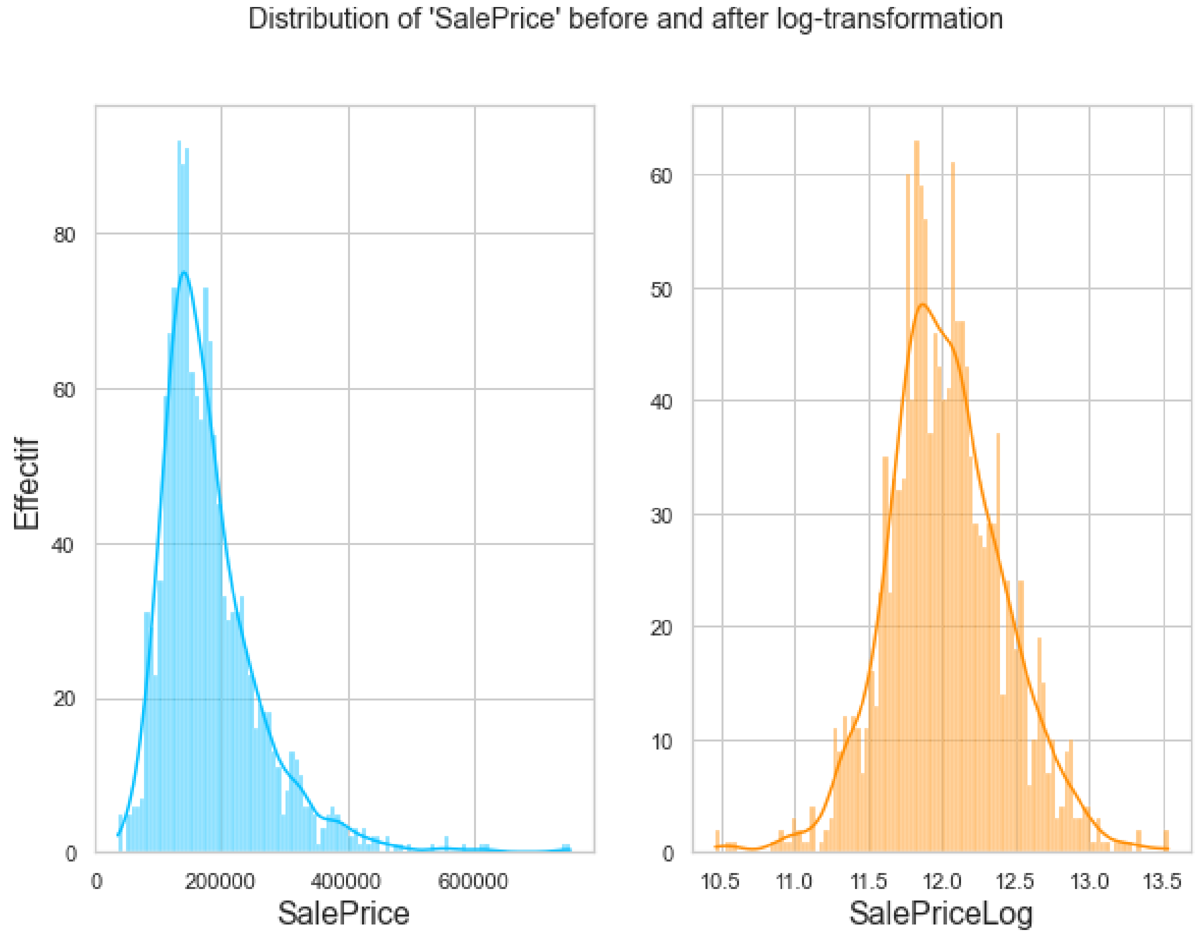
```python
# Convert Year of construction to Age of the house since the construction
df_train_new["AgeSinceConst"] = (
    df_train_new["YearBuilt"].max() - df_train_new["YearBuilt"])

df_test_new["AgeSinceConst"] = df_test_new["YearBuilt"].max() - \
    df_test_new["YearBuilt"]

# Drop "YearBuilt"
df_train_new.drop(["YearBuilt"], axis=1, inplace=True)
df_test_new.drop(["YearBuilt"], axis=1, inplace=True)
```

To avoid redundancy and to mitigate the strong variations of some features according to the SalePrice. We will use a log transformation for skewed features. Generally, in real estate, as the area of the property increases, the price per square feet decreases, hence the use of log.

3. Preparing Data for Modelling.

3.1 Target Variable ‘SalePrice’

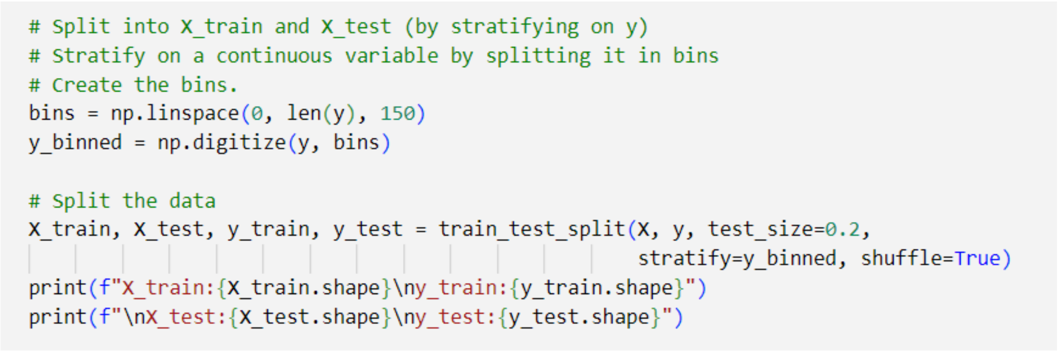
Log transformation of the independent variable ('SalePrice') to have a distribution that approaches the normal distribution.


Distribution of 'SalePrice' before and after log-transformation

```
# Drop the original SalePrice
df_train_new.drop(["SalePrice"], axis=1, inplace=True)
```

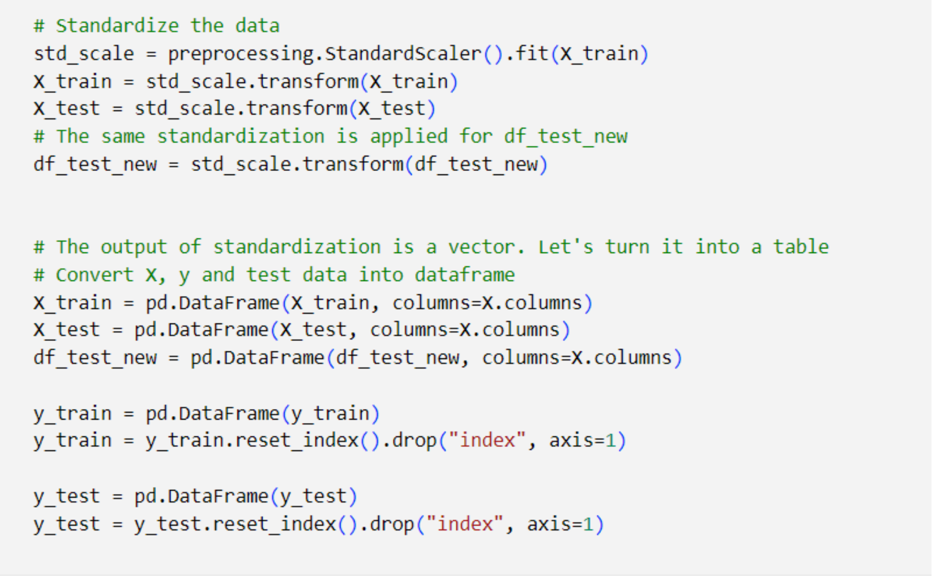
3.2 Split data into train and test set and Standardisation.

Splitting the Data into train and Test:



```
# Split into X_train and X_test (by stratifying on y)
# Stratify on a continuous variable by splitting it in bins
# Create the bins.
bins = np.linspace(0, len(y), 150)
y_binned = np.digitize(y, bins)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    stratify=y_binned, shuffle=True)
print(f"X_train:{X_train.shape}\ny_train:{y_train.shape}")
print(f"\nX_test:{X_test.shape}\ny_test:{y_test.shape}")
```

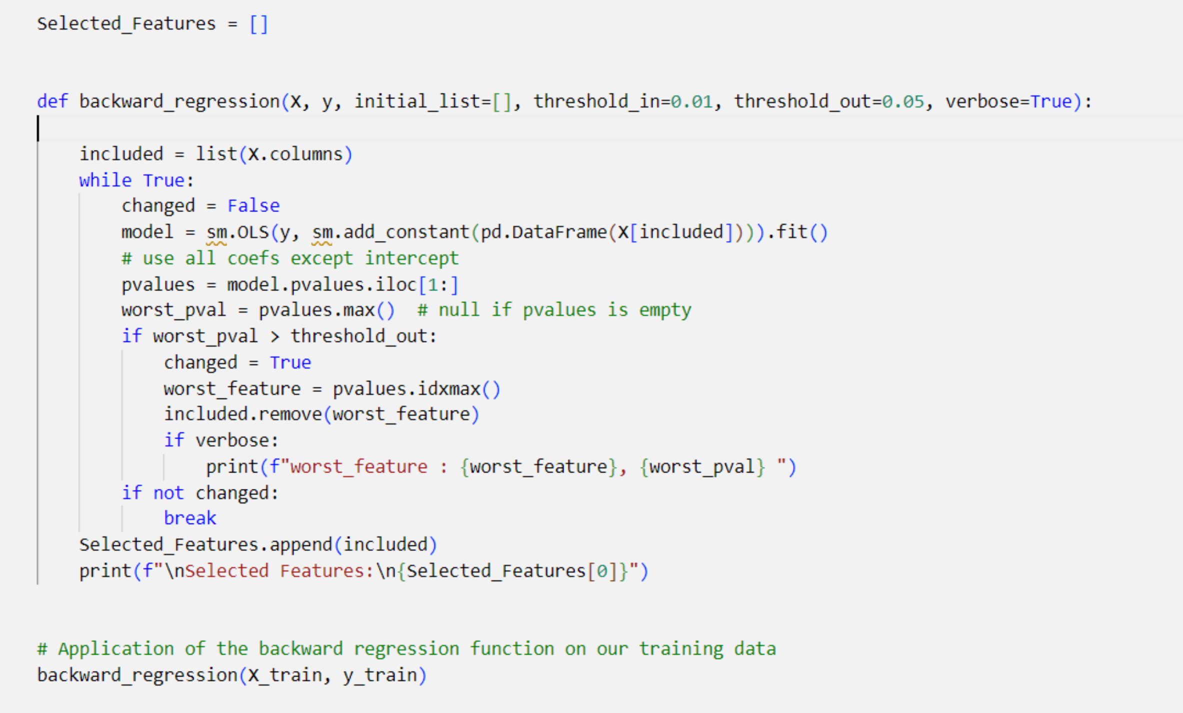
Standardisation:



```
# Standardize the data
std_scale = preprocessing.StandardScaler().fit(X_train)
X_train = std_scale.transform(X_train)
X_test = std_scale.transform(X_test)
# The same standardization is applied for df_test_new
df_test_new = std_scale.transform(df_test_new)


# The output of standardization is a vector. Let's turn it into a table
# Convert X, y and test data into dataframe
X_train = pd.DataFrame(X_train, columns=X.columns)
X_test = pd.DataFrame(X_test, columns=X.columns)
df_test_new = pd.DataFrame(df_test_new, columns=X.columns)

y_train = pd.DataFrame(y_train)
y_train = y_train.reset_index().drop("index", axis=1)

y_test = pd.DataFrame(y_test)
y_test = y_test.reset_index().drop("index", axis=1)
```

3.3 Backward Stepwise Regression.

Let's lighten the model with a Backward Stepwise Regression.
Backward Stepwise Regression is a stepwise regression approach that begins with a full (saturated) model and at each step gradually eliminates variables from the regression model to find a reduced model that best explains the data. Also known as Backward Elimination regression.



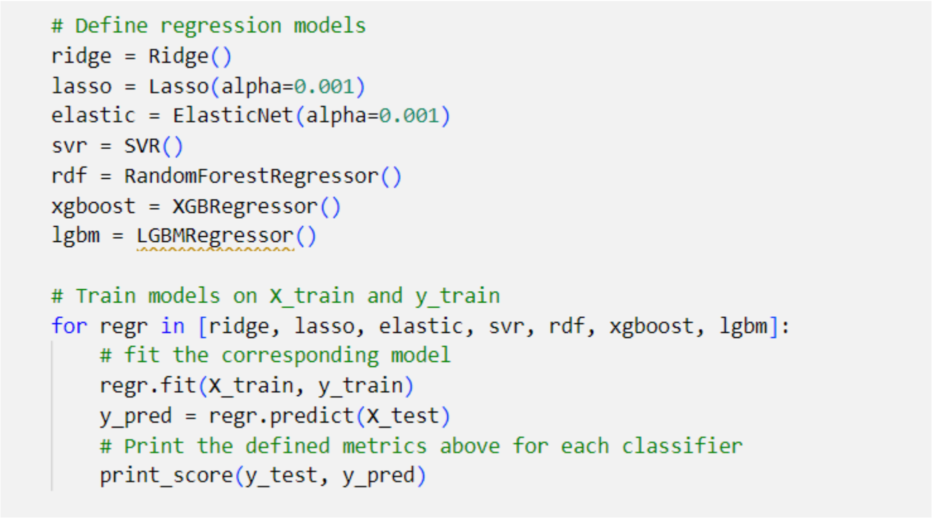
```
Selected_Features = []


def backward_regression(X, y, initial_list=[], threshold_in=0.01, threshold_out=0.05, verbose=True):

    included = list(X.columns)
    while True:
        changed = False
        model = sm.OLS(y, sm.add_constant(pd.DataFrame(X[included]))).fit()
        # use all coefs except intercept
        pvalues = model.pvalues.iloc[1:]
        worst_pval = pvalues.max()  # null if pvalues is empty
        if worst_pval > threshold_out:
            changed = True
            worst_feature = pvalues.idxmax()
            included.remove(worst_feature)
            if verbose:
                print(f"worst_feature : {worst_feature}, {worst_pval} ")
        if not changed:
            break
    Selected_Features.append(included)
    print(f"\nSelected Features:\n{Selected_Features[0]}")


# Application of the backward regression function on our training data
backward_regression(X_train, y_train)
```

4. Modelling

4.1 Models and metrics selection

Here we are going to use RMSE and R² metrics in order to measure the performance of the selected models and their predictions.
Then we will test the models that best meet the estimation of house prices. Here are the following models we will use for regression:

- Ridge regression
- Lasso regression
- Elastic Net regression
- Support Vector regression (SVR)
- Random Forest regression
- XGBoost



```
# Define regression models
ridge = Ridge()
lasso = Lasso(alpha=0.001)
elastic = ElasticNet(alpha=0.001)
svr = SVR()
rdf = RandomForestRegressor()
xgboost = XGBRegressor()
lgbm = LGBMRegressor()

# Train models on X_train and y_train
for regr in [ridge, lasso, elastic, svr, rdf, xgboost, lgbm]:
    # fit the corresponding model
    regr.fit(X_train, y_train)
    y_pred = regr.predict(X_test)
    # Print the defined metrics above for each classifier
    print_score(y_test, y_pred)
```

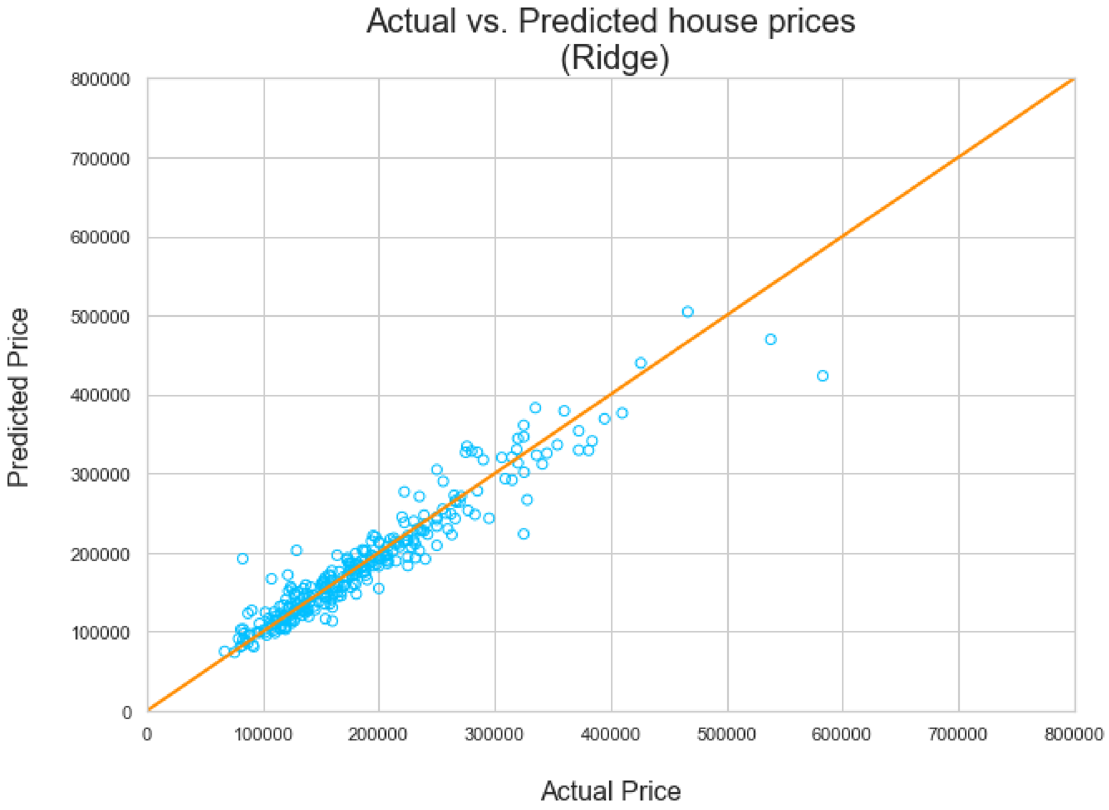
According to the result of R² and Root Mean Squared of these 7 models, we can conclude that the relationship between the features and the target variable is clearly linear. Thus, in the next part we will optimize the hyperparameters of all the models except SVR.

Indeed, SVR acknowledges the presence of non-linearity in the data and provides a proficient prediction model. The basic idea behind SVR is to find the best fit line which is a hyperplane that has the maximum number of points. It is clear that our model is linear that's why we will not optimize the SVR model

4.2 Hyperparameters tuning and model Optimisation

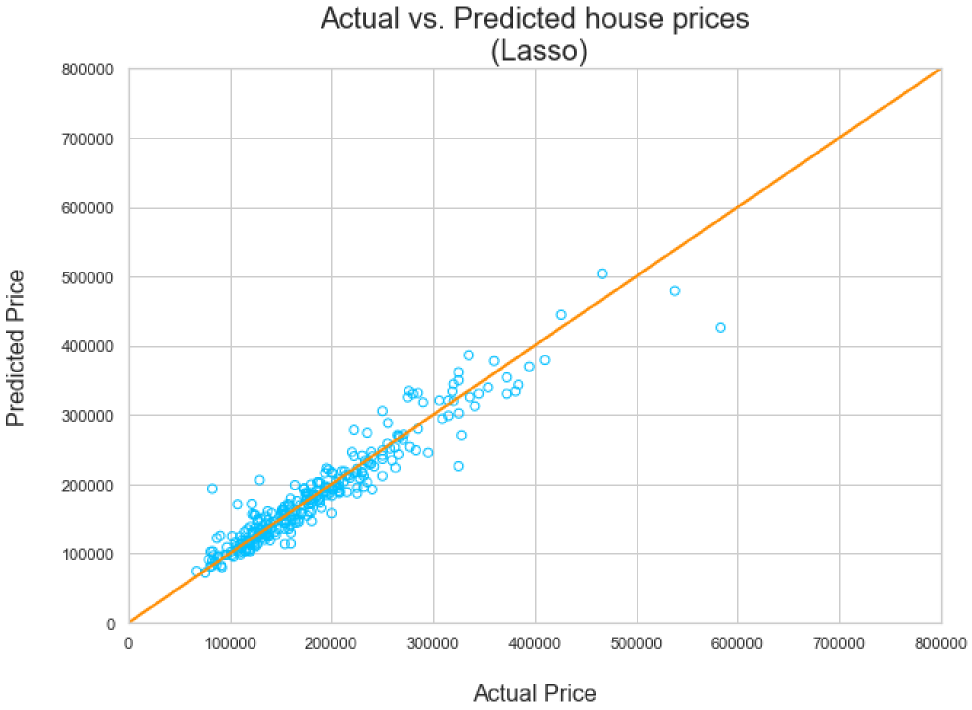
4.2.1 Ridge Regression -

Ridge will reduce the impact of features that are not important in predicting the target values.



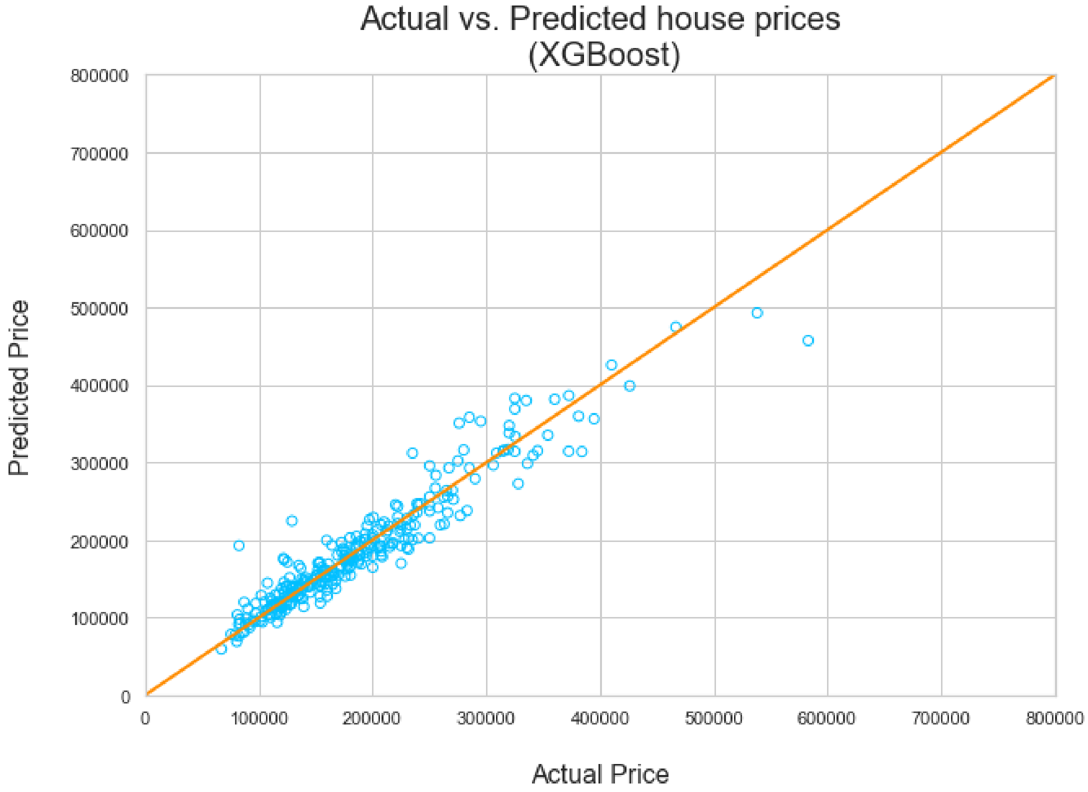
4.2.2 Lasso Regression -

Lasso will eliminate many features and reduce overfitting in the linear model.

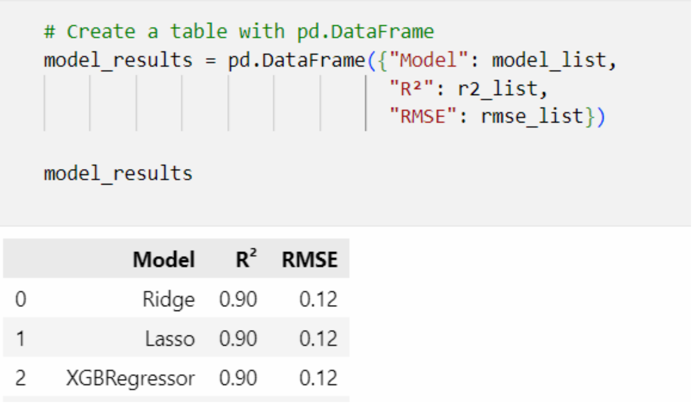


4.2.3 XGBoost Regression -

XGBoost is one of the most popular algorithms that are based on Gradient Boosted Machines. Gradient Boosting refers to a methodology where an ensemble of weak learners is used to improve the model performance in terms of efficiency, accuracy, and interpretability. Gradient Boosting can be applied to a regression by taking the average of the outputs by the weak learners.



4.3 Choosing the best Model

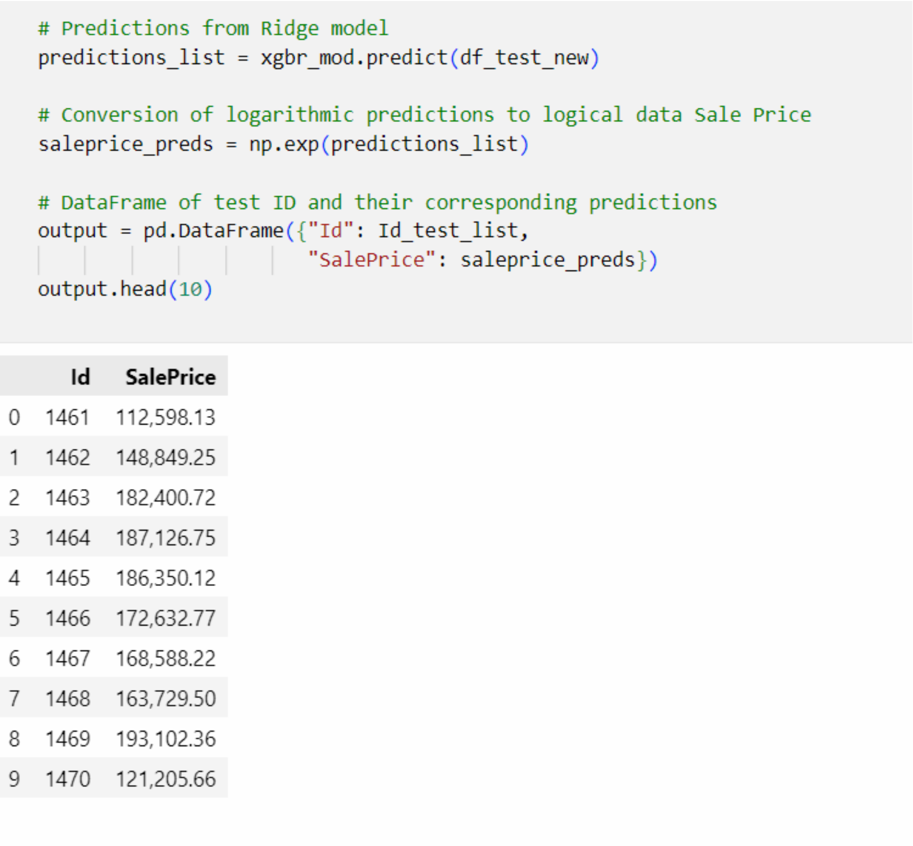


```
# Create a table with pd.DataFrame
model_results = pd.DataFrame({"Model": model_list,
                              "R²": r2_list,
                              "RMSE": rmse_list})

model_results
```

	Model	R²	RMSE
0	Ridge	0.90	0.12
1	Lasso	0.90	0.12
2	XGBRegressor	0.90	0.12

XGB Regressor model will be chosen to predict house prices of the Test set.

4.4 Prediction on ‘Housing Prices’ using XGBoost on test dataset



```
# Predictions from Ridge model
predictions_list = xgbr_mod.predict(df_test_new)

# Conversion of logarithmic predictions to logical data Sale Price
saleprice_preds = np.exp(predictions_list)

# DataFrame of test ID and their corresponding predictions
output = pd.DataFrame({"Id": Id_test_list,
                       "SalePrice": saleprice_preds})
output.head(10)
```

	Id	SalePrice
0	1461	112,598.13
1	1462	148,849.25
2	1463	182,400.72
3	1464	187,126.75
4	1465	186,350.12
5	1466	172,632.77
6	1467	168,588.22
7	1468	163,729.50
8	1469	193,102.36
9	1470	121,205.66

5. Conclusion:

In this model we trained model using Ridge Regression, Lasso Regression and XGBoost Regression. We found that all the regressions showed almost similar predictions.

For improving the accuracy, we did feature selection and feature engineering

For prediction on test dataset, we used XGBoost Regression and from the graphs we can conclude that the predicted prices are near to the prices in training dataset.