EDA

August 12, 2021

[1]:
```python
# Data reading and visualization
import os
import numpy as np
import pandas as pd
import seaborn as sns
import plotly.express as px
import matplotlib.pyplot as plt
%matplotlib inline

# Scikit-learn
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# XGBoost & LightGBM
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor

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

[2]:
```python
# CONFIGS
BASE_PATH = "./"

rf_params = {
    'n_estimators': 100,
    'max_depth': 4,
    'min_samples_split': 2,
    'min_samples_leaf': 1
    }

xgb_params = {
    'n_estimators': 1000,
    'max_depth': 4,
    'min_child_weight': 2,
    'learning_rate': 0.01,
```

```
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'objective': 'reg:linear',
    'booster': 'gbtree'
    }

lgb_params = {
    'n_estimators': 1000,
    'max_depth': 4,
    'learning_rate': 0.01,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'objective': 'regression'
    }
```

[3]:
```
# Helper functions

# Print dataset usage stats
def print_info(df):
    print(f"\nDataframe Shape: {df.shape}")
    print(f"\nDataframe Columns: {df.columns}")
    print(f"\nDataframe dtypes: \n{df.dtypes.value_counts()}")
    print(f"\nDataframe memory usage: {round(df.memory_usage().sum() / 1024**2,␣
 ↪2)} MB")

# Fit data to model(s)
def fit_pipeline(scaler, model, X_train, y_train, X_test, y_test):
    pipe = Pipeline([('scaler', scaler), ('model', model)])
    pipe.fit(X_train, y_train)
    score = pipe.score(X_test, y_test)

    return score

# Plot feature importance
def plot_feature_importance(features, title, model):
    fig = px.bar(y=features, x=model.feature_importances_,␣
 ↪template='plotly_dark')
    fig.update_layout(title=f"{title}")
    fig.update_xaxes(title_text="Feature Importance")
    fig.update_yaxes(title_text="Feature")

    fig.show()
```

0.1 # Exploratory Data Analysis (EDA)

[4]:
```python
df = pd.read_csv(os.path.join(BASE_PATH, "dataset/train.csv"))
```

[5]:
```python
# Check shape of dataset
print_info(df)
```

```
Dataframe Shape: (1460, 81)

Dataframe Columns: Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage',
'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')

Dataframe dtypes:
object     43
int64      35
float64     3
dtype: int64

Dataframe memory usage: 0.9 MB
```

[6]:
```python
# Check for null values
df.isnull().sum().value_counts()
```

[6]:
```
0     62
81     5
37     3
8      2
38     2
1      1
```

```
259      1
1179     1
1453     1
690      1
1369     1
1406     1
dtype: int64
```

0.1.1 > Some columns have a lot of missing data, we will either drop them or fill them with the mean value of the column

0.1.2 > To keep things simple, we will only work with numerical data for now

[7]:

```
# Filter and keep only the numerical data
df_num = df.select_dtypes(include=['float64', 'int64'])

# Filter out more catgeorical and time-series data given in `data_description.
 ↪txt`
to_drop = ['Id', 'MSSubClass', 'OverallQual', 'OverallCond', 'YearBuilt',␣
 ↪'YearRemodAdd','MoSold', 'YrSold',  'FullBath', 'HalfBath', 'Fireplaces',␣
 ↪'GarageCars']
df_num.drop(to_drop, axis=1, inplace=True)

# Even more filtering to remove miscellanous features
to_drop = ['MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF',␣
 ↪'BsmtFullBath', 'BsmtHalfBath', 'BedroomAbvGr', 'KitchenAbvGr',␣
 ↪'TotRmsAbvGrd', 'GarageYrBlt', 'MiscVal', 'EnclosedPorch', '3SsnPorch',␣
 ↪'ScreenPorch']
df_num.drop(to_drop, axis=1, inplace=True)

# Check shape of dataset
print_info(df_num)
```

```
Dataframe Shape: (1460, 12)

Dataframe Columns: Index(['LotFrontage', 'LotArea', 'TotalBsmtSF', '1stFlrSF',
'2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF',
       'PoolArea', 'SalePrice'],
      dtype='object')

Dataframe dtypes:
int64      11
float64     1
dtype: int64
```

```
Dataframe memory usage: 0.13 MB
```

[8]:
```python
# Let's check the null values again...
df_num.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   LotFrontage   1201 non-null   float64
 1   LotArea       1460 non-null   int64
 2   TotalBsmtSF   1460 non-null   int64
 3   1stFlrSF      1460 non-null   int64
 4   2ndFlrSF      1460 non-null   int64
 5   LowQualFinSF  1460 non-null   int64
 6   GrLivArea     1460 non-null   int64
 7   GarageArea    1460 non-null   int64
 8   WoodDeckSF    1460 non-null   int64
 9   OpenPorchSF   1460 non-null   int64
 10  PoolArea      1460 non-null   int64
 11  SalePrice     1460 non-null   int64
dtypes: float64(1), int64(11)
memory usage: 137.0 KB
```

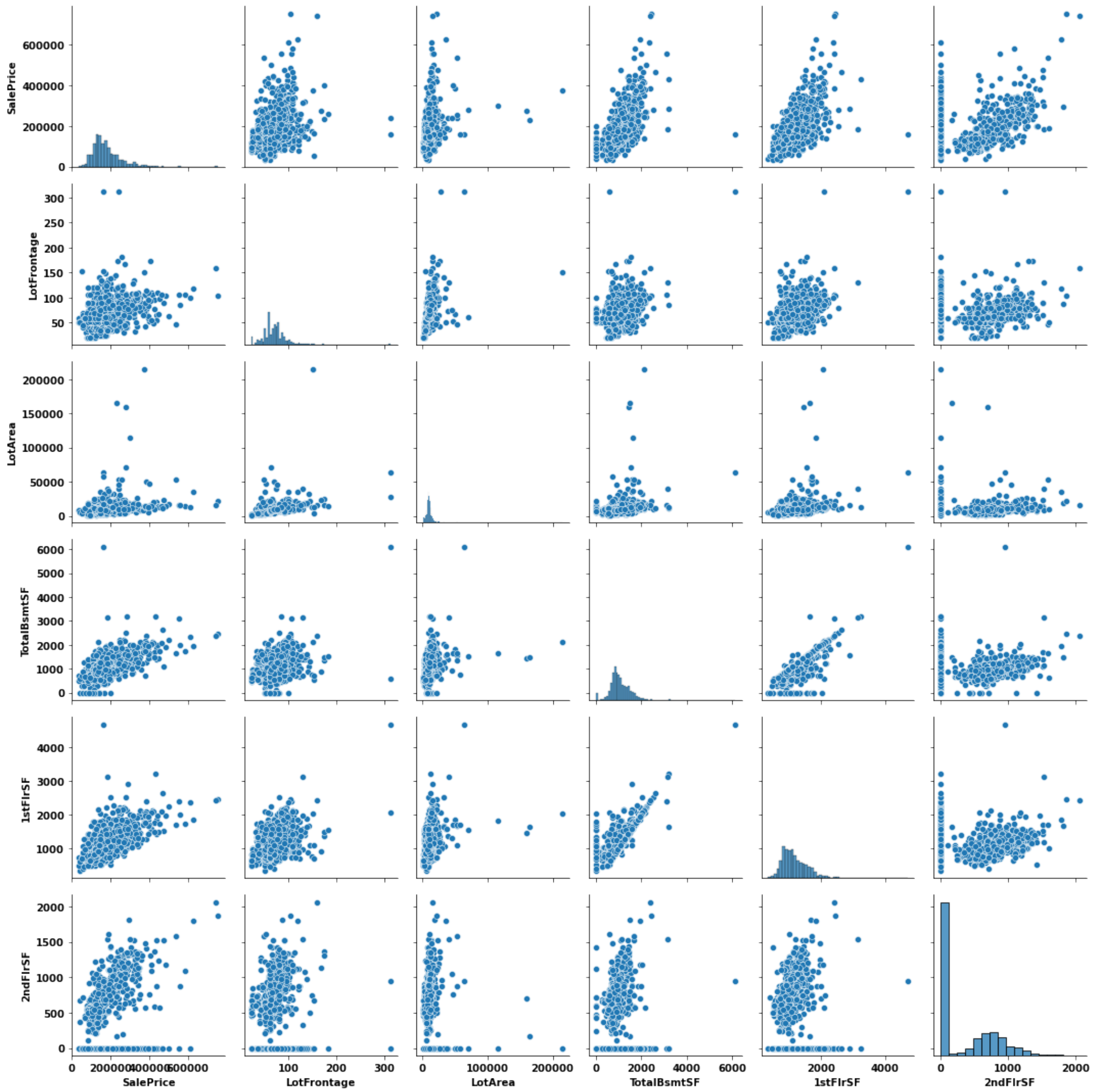
0.2 ## Visualization(s)!

[9]:
```python
# Observe each feature's distribution

# Normal Histogram
hist = df_num.hist(figsize=(15, 15), bins=50)

plt.tight_layout()
plt.show()

# Histogram with distrbution line
# fig = plt.figure(figsize=(15, 15))

# for i, column in enumerate(df_num.columns):
#     plt.subplot(4, 4, i+1)
#     plt.title(column)
#     plt.xlabel(column)
#     sns.distplot(df_num[column])

# plt.tight_layout()
# plt.show()
```

[ ]:
```
# There are a couple of features which remain constant. Let's remove them.
df_num.drop(['LowQualFinSF', 'PoolArea'], axis=1, inplace=True)
```

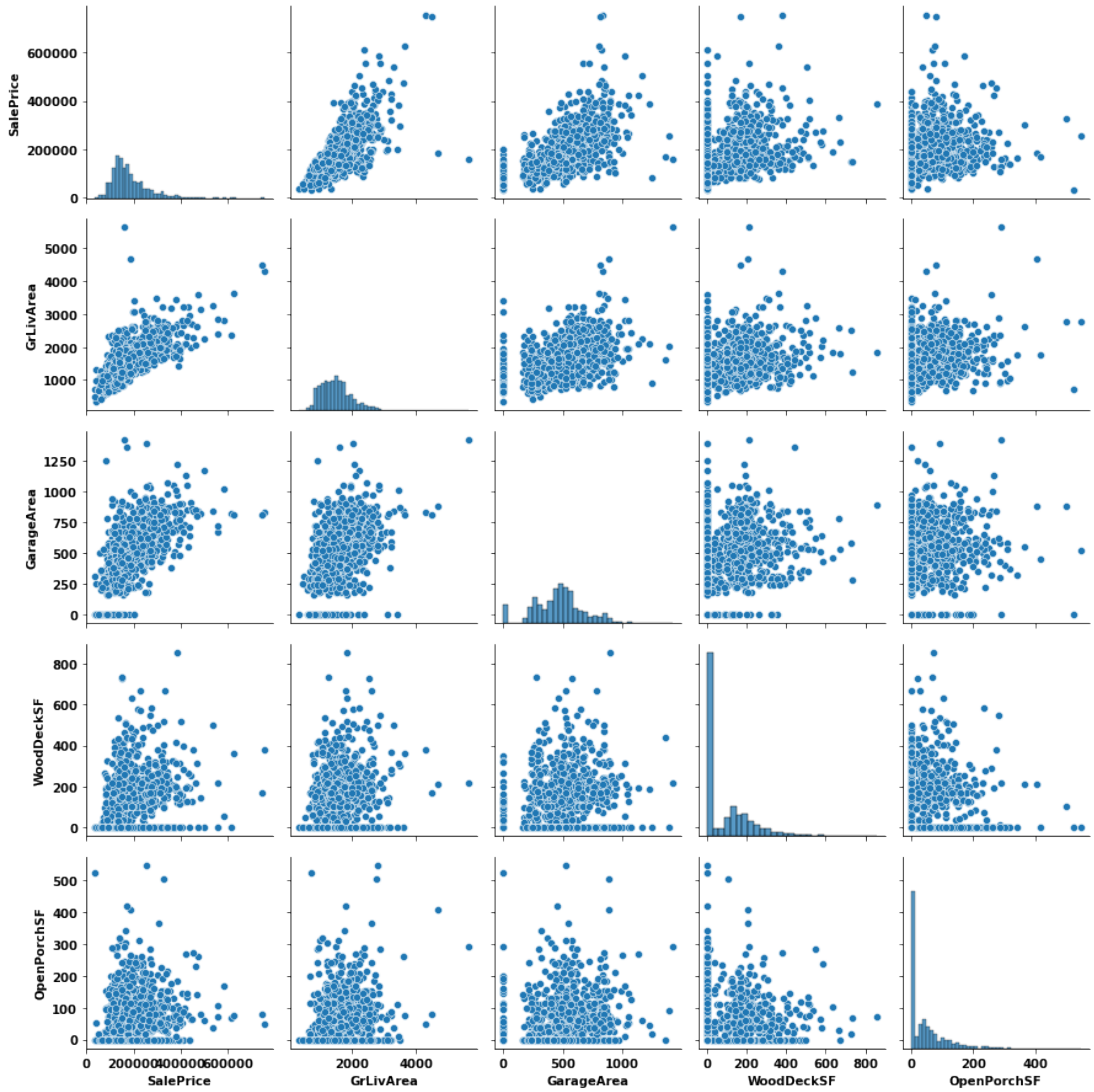
[ ]:
```
# Use pairplots to see how one feature is related to the other

# Features: 'LotFrontage', 'LotArea', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF',␣
↪'GrLivArea', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF'

# Since these can get very large, let's use only the SalePrice and the first 5␣
↪features
fig = plt.figure(figsize=(10, 10))
sns.pairplot(df_num[['SalePrice', 'LotFrontage', 'LotArea', 'TotalBsmtSF',␣
↪'1stFlrSF', '2ndFlrSF']])
plt.tight_layout()
plt.show()
```

```
<Figure size 720x720 with 0 Axes>
```

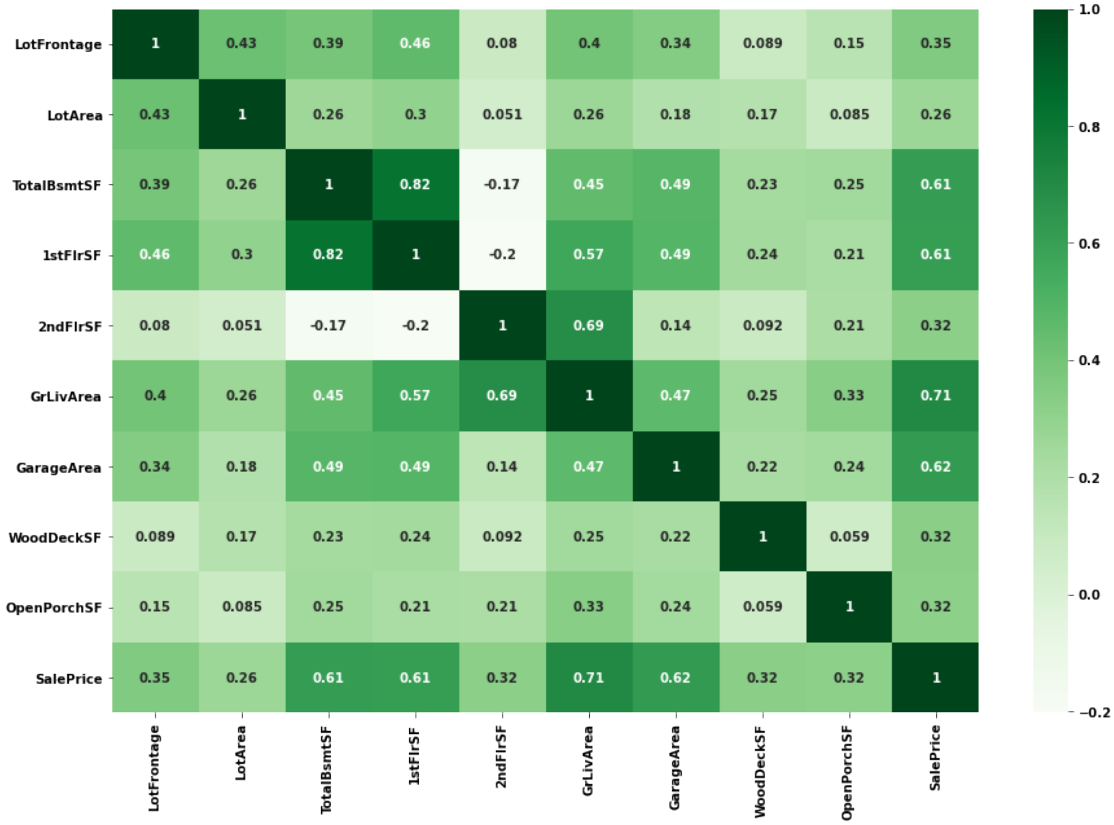


[ ]:
```
# Use pairplots to see how one feature is related to the other

# Features: 'LotFrontage', 'LotArea', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'GrLivArea', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF'

# Since these can get very large, let's use only the SalePrice and the last 4 features
fig = plt.figure(figsize=(10, 10))
sns.pairplot(df_num[['SalePrice', 'GrLivArea', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF']])
plt.tight_layout()
plt.show()
```

<Figure size 720x720 with 0 Axes>




```
[ ]: # Correlation Matrix
fig = plt.figure(figsize=(15, 10))
sns.heatmap(df_num.corr(), cmap='Greens', annot=True)
plt.show()
```



0.3 # Feature Generation

[ ]:
```
# Now that we are done with cleaning the data, let's try to generate more features
# by using the existing ones.

# Current Features: 'LotFrontage', 'LotArea', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'PoolArea'

# Example, we can combine the 'LotArea', 'TotalBsmtSF', 'GarageArea', 'WoodDeckSF', OpenPorchSF' features to generate a new feature: 'ExtArea'

df_num['ExtArea'] = df_num['LotArea'] + df_num['TotalBsmtSF'] + df_num['GarageArea'] + df_num['WoodDeckSF'] + df_num['OpenPorchSF']
```

0.4 # Data Preprocessing

We now need to make sure that the data has a normal distribution.

To achieve this, we can use: - StandardScaler : Normalizes data between 0 and 1 - MinMaxScaler

: Normalizes data betweeen the min and max value in the coolum

[ ]:
```python
# Let's work with the missing values first
df_num.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 11 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   LotFrontage   1201 non-null   float64
 1   LotArea       1460 non-null   int64
 2   TotalBsmtSF   1460 non-null   int64
 3   1stFlrSF      1460 non-null   int64
 4   2ndFlrSF      1460 non-null   int64
 5   GrLivArea     1460 non-null   int64
 6   GarageArea    1460 non-null   int64
 7   WoodDeckSF    1460 non-null   int64
 8   OpenPorchSF   1460 non-null   int64
 9   SalePrice     1460 non-null   int64
 10  ExtArea       1460 non-null   int64
dtypes: float64(1), int64(10)
memory usage: 125.6 KB
```

Since LotFrontage has a lot of missing values, we can: 1. `impute` the mean value from the column 2. drop the rows with the missing values 3. drop the whole column

We will use option 1 as our dataset is relatively small, and losing data will not be beneficial

[ ]:
```python
# Imput mean value in the missing value areas
df_num.fillna(df_num.mean(), inplace=True)
```

[ ]:
```python
# We first need to split the data into train and test sets
X, Y = df_num.drop('SalePrice', axis=1), df_num['SalePrice']

# Split the data into train and test sets: Use 20% of the data for testing
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2,
 ↪random_state=42)
```

[ ]:
```python
# Use baseline models to evaluate the performance before any preprocessing

# Define models
rf = RandomForestRegressor(**rf_params)
xgb = XGBRegressor(**xgb_params)
lgb = LGBMRegressor(**lgb_params)

# Fit models
rf.fit(X_train, y_train)
xgb.fit(X_train, y_train)
```

```
lgb.fit(X_train, y_train)

# Get Predictions
baseline_rf_score = rf.score(X_test, y_test)
baseline_xgb_score = xgb.score(X_test, y_test)
baseline_lgb_score = lgb.score(X_test, y_test)

print(f"\n\nBaseline Scores: \n{'-'*25}\n")
print(f"RandomForestRegressor Score: {baseline_rf_score}")
print(f"XGBRegressor Score: {baseline_xgb_score}")
print(f"LGBMRegressor Score: {baseline_lgb_score}")
```

```
[17:39:17] WARNING: c:\ci\xgboost-
split_1619728435298\work\src\objective\regression_obj.cu:170: reg:linear is now
deprecated in favor of reg:squarederror.


Baseline Scores:
-------------------------

RandomForestRegressor Score: 0.7775756242801215
XGBRegressor Score: 0.8394263530571797
LGBMRegressor Score: 0.8204068943249779
```

[ ]:
```
# Let's apply a preprocessing pipeline to the data using StandardScaler

# Define models
rf  = RandomForestRegressor(**rf_params)
xgb = XGBRegressor(**xgb_params)
lgb = LGBMRegressor(**lgb_params)

# Get scaler
scaler = StandardScaler()

# Pass models to pipeline
rf_score = fit_pipeline(scaler, rf, X_train, y_train, X_test, y_test)
xgb_score = fit_pipeline(scaler, xgb, X_train, y_train, X_test, y_test)
lgb_score = fit_pipeline(scaler, lgb, X_train, y_train, X_test, y_test)

# Print scores
print(f"\n\nStandardScaler Scores: \n{'-'*25}\n")
print(f"RandomForestRegressor Score: {rf_score}")
print(f"XGBRegressor Score: {xgb_score}")
print(f"LGBMRegressor Score: {lgb_score}")
```

```
[17:39:18] WARNING: c:\ci\xgboost-
split_1619728435298\work\src\objective\regression_obj.cu:170: reg:linear is now
deprecated in favor of reg:squarederror.
```

```
StandardScaler Scores:
-------------------------

RandomForestRegressor Score: 0.782707417107054
XGBRegressor Score: 0.8394645654344709
LGBMRegressor Score: 0.8227794311188819
```

[ ]:
```python
# Let's apply a preprocessing pipeline to the data using MinMaxScaler

# Define models
rf = RandomForestRegressor(**rf_params)
xgb = XGBRegressor(**xgb_params)
lgb = LGBMRegressor(**lgb_params)

# Get scaler
scaler = MinMaxScaler()

# Pass models to pipeline
rf_score = fit_pipeline(scaler, rf, X_train, y_train, X_test, y_test)
xgb_score = fit_pipeline(scaler, xgb, X_train, y_train, X_test, y_test)
lgb_score = fit_pipeline(scaler, lgb, X_train, y_train, X_test, y_test)

# Print scores
print(f"\n\nMinMaxScaler Scores: \n{'-'*25}\n")
print(f"RandomForestRegressor Score: {rf_score}")
print(f"XGBRegressor Score: {xgb_score}")
print(f"LGBMRegressor Score: {lgb_score}")
```

```
[17:39:19] WARNING: c:\ci\xgboost-
split_1619728435298\work\src\objective\regression_obj.cu:170: reg:linear is now
deprecated in favor of reg:squarederror.


MinMaxScaler Scores:
-------------------------

RandomForestRegressor Score: 0.7781223199783313
XGBRegressor Score: 0.8401299481241516
LGBMRegressor Score: 0.8204068943249779
```

0.5 # Feature Selection

[ ]:
```python
# Let's see which of the features are the most important by each model

# RandomForestRegressor
```

```
plot_feature_importance(X_train.columns, "RandomForestRegressor Feature␣
 ↪Importance", rf)

# XGBRegressor
plot_feature_importance(X_train.columns, "XGBRegressor Feature Importance", xgb)

# LGBMRegressor
plot_feature_importance(X_train.columns, "LGBMRegressor Feature Importance",␣
 ↪lgb)
```

1 Aaaand.... We're Done! Good Luck Ahead!

[ ]:

```

```