-

IBM MACHINE LEARNING

SUPERVISED MACHINE LEARNING: REGRESSION MODELS FOR HOUSE PRICE PREDICTION

Fatima, Sayeda

9/24/2022

Table of Contents

1) Project Overview 2
2) About the Dataset 3
2a) Brief description of the data set you chose: 3
2b) Summary of Data Attributes 3
3) Main Objectives of Analysis 4
4) Data Exploration, Data Cleansing and Features Engineering 5
4a) Data Exploration 5
4b) Data Cleansing & Features Engineering 9
5) Summary of Training Different Regression Models 16
5a) Machine Learning Regression Algorithm Development Approach 16
5b) Summarizing Employed Models 20
1) Ridge Regression (RR) Models 20
2) Lasso Regression (LR) Models 23
3) Elastic-Net (EN) Models 26
4) Extreme Gradient Boosting (XGB) Regression Models 29
6) Key Findings to the Main Objectives of Analysis 32
6a) Result Summary 32
6b) Recommended Model and Justification 32
6c) Summarizing Model Drivers 33
6d) Enlisting Top Contributory Factors 33
6e) Visualizing Top Contributory Factors Driving Sale Value 34
6f) Sale Price Prediction on Test/New Data 35
7) Future Directions 36
7a) Possible Flaws in Chosen Model 36
7b) Recommendations 36
8) Useful Links 37
8a) Link to Other Useful Models 37
8b) Github Link to Assignment Notebook and Other Files 37
References 38

1) Project Overview

A fundamental issue real estate business face is assessing sale price based on house attributes. Grounded on hedonic price modelling theory, not only do neighbourhood-specific characteristics but also unit-specific attributes greatly drive house prices (Herath and Maier, 2010). Hence, identification of key factors driving sale prices of real estate is crucial to facilitate informed purchase decisions. It is here that Regression Machine Learning models can be very useful to gain deeper insight into underlying factors as well as their relationship in driving and estimating fair sale price of houses.

Hence, the main aim of the following regression modelling and analysis approach is to enable the business to:

* Find best regression model for house price prediction

* Identify different factors influencing house prices

* Predict sale price based on contributory factors

2) About the Dataset

2a) Brief description of the data set you chose:

This project uses a hypothetical dataset 'Ames, Iowa Housing Dataset' which was downloaded from the following link:

https://www.kaggle.com/datasets/prevek18/ames-housing-dataset

2b) Summary of Data Attributes

The dataset exhibits 2,930 data points (rows) and 82 features (columns) reflecting on housing characteristics.

The data also comes with 'SalePrice' Column which represents the Class requiring prediction.

```
1 raw_data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2930 entries, 0 to 2929
Data columns (total 82 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Order            2930 non-null   int64
 1   PID              2930 non-null   int64
 2   MS SubClass      2930 non-null   int64
 3   MS Zoning        2930 non-null   object
 4   Lot Frontage     2440 non-null   float64
 5   Lot Area         2930 non-null   int64
 6   Street           2930 non-null   object
 7   Alley            198 non-null    object
 8   Lot Shape        2930 non-null   object
 9   Land Contour     2930 non-null   object
 10  Utilities        2930 non-null   object
 11  Lot Config       2930 non-null   object
 12  Land Slope       2930 non-null   object
 13  Neighborhood     2930 non-null   object
 14  Condition 1      2930 non-null   object
 15  Condition 2      2930 non-null   object
 16  Bldg Type        2930 non-null   object
 17  House Style      2930 non-null   object
 18  Overall Qual     2930 non-null   int64
 19  Overall Cond     2930 non-null   int64
 20  Year Built       2930 non-null   int64
 21  Year Remod/Add   2930 non-null   int64
 22  Roof Style       2930 non-null   object
 23  Roof Matl        2930 non-null   object
 24  Exterior 1st     2930 non-null   object
 25  Exterior 2nd     2930 non-null   object
 26  Mas Vnr Type     2907 non-null   object
 27  Mas Vnr Area     2907 non-null   float64
 28  Exter Qual       2930 non-null   object
 29  Exter Cond       2930 non-null   object
 30  Foundation       2930 non-null   object
 31  Bsmt Qual        2850 non-null   object
 32  Bsmt Cond        2850 non-null   object
 33  Bsmt Exposure    2847 non-null   object
 34  BsmtFin Type 1   2850 non-null   object
 35  BsmtFin SF 1     2929 non-null   float64
 36  BsmtFin Type 2   2849 non-null   object
 37  BsmtFin SF 2     2929 non-null   float64
 38  Bsmt Unf SF      2929 non-null   float64
 39  Total Bsmt SF    2929 non-null   float64
 40  Heating          2930 non-null   object
 41  Heating QC       2930 non-null   object
 42  Central Air      2930 non-null   object
 43  Electrical       2929 non-null   object
```

3) Main Objectives of Analysis

Real estate business performance is largely dependent on paying fair price of assets to prevent overpriced purchases and minimize loss. Hence, these businesses are continuously faced with the challenge to estimate realistic house prices and often rely on manual application of Hedonic Price Method (HPM) or hedonic regression analysis. Consequently, driven by HPM, this analysis is targeted towards answering the following queries

- What are the various contributory factors which drive house prices in a given area?
- Based on important factors, what will be projected price for different which housing units?

As a consequence, implementation of an automated machine learning (ML) HPM regression modelling process will enable the organization to:

- identify key underlying factors which can appreciate or depreciate property values
- save valuable resources and funds in purchasing properties at right values
- effortlessly employ best ML HPM model and generate report with the click of a button

4) Data Exploration, Data Cleansing and Features Engineering

Since the quality of any machine learning model highly depends on quality of data, hence, this stage is not only most important but is also time consuming. Hence, it was conducted in a step-by-step manner.

4a) Data Exploration

- Data was first loaded into pandas dataframe

Section A: About the Dataset

1a) Brief description of the data set you chose:
This project uses dataset for 'Ames Housing' which may also be downloaded from the following link:
https://www.kaggle.com/datasets/prevek18/ames-housing-dataset
This dataset represents housing data for Ames, Iowa region in between 2006 to 2010.

1b) Summary of Data Attributes:
The dataset exhibits 2930 data points (rows) and the following 82 features (columns):

```
raw_data = pd.read_csv('C:/Users/fatima.s/Documents/PythonScripts/DATA SCIENCE/IBM Machine Learning Intermediate/MODULE 2 SU
raw_data
```

	Order	PID	MS SubClass	MS Zoning	Lot Frontage	Lot Area	Street	Alley	Lot Shape	Land Contour	...	Pool Area	Pool QC	Fence	Misc Feature	Misc Val	Mo Sold	Yr Sold	Sale Type	Sa Conditic
0	1	526301100	20	RL	141.0	31770	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	NaN	0	5	2010	WD	Norm
1	2	526350040	20	RH	80.0	11622	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	NaN	0	6	2010	WD	Norm
2	3	526351010	20	RL	81.0	14267	Pave	NaN	IR1	Lvl	...	0	NaN	NaN	Gar2	12500	6	2010	WD	Norm
3	4	526353030	20	RL	93.0	11160	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2010	WD	Norm
4	5	527105010	60	RL	74.0	13830	Pave	NaN	IR1	Lvl	...	0	NaN	MnPrv	NaN	0	3	2010	WD	Norm
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
2925	2926	923275080	80	RL	37.0	7937	Pave	NaN	IR1	Lvl	...	0	NaN	GdPrv	NaN	0	3	2006	WD	Norm
2926	2927	923276100	20	RL	NaN	8885	Pave	NaN	IR1	Low	...	0	NaN	MnPrv	NaN	0	6	2006	WD	Norm
2927	2928	923400125	85	RL	62.0	10441	Pave	NaN	Reg	Lvl	...	0	NaN	MnPrv	Shed	700	7	2006	WD	Norm
2928	2929	924100070	20	RL	77.0	10010	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	4	2006	WD	Norm
2929	2930	924151050	60	RL	74.0	9627	Pave	NaN	Reg	Lvl	...	0	NaN	NaN	NaN	0	11	2006	WD	Norm

2930 rows × 82 columns

- A method was created to conduct preliminary analysis including computation of:
  - ☞ Descriptive statistics to summarize shape of a dataset's distribution, its dispersion and central tendency
  - ☞ Data analysis to depict data types, skewness, kurtosis, etc to facilitate subsequent data cleansing

```
def analysis(*name): # This Method will extract dataframe by name
    n = name # Extract Dataframe by Name...this will create a 3d tuple
    n = (n[0]) # Convert Tuple to To Dataframe
    df_name = [x for x in globals() if globals()[x] is n][0] # Extract Name of Imported Dataframe to print later

    # Perform Statistics
    stats = n.describe(include = 'all').transpose()
    stats = stats.fillna(0) # Replace all Nan Values with Zero

    # Data Analysis
    obs = n.shape[0]
    types = n.dtypes
    counts = n.apply(lambda x: x.count())
    distincts = n.apply(lambda x: x.unique().shape[0])
    nulls = n.apply(lambda x: x.isnull().sum())
    uniques = n.apply(lambda x: [x.unique()])
    per_nulls = (n.isnull().sum()/ obs) * 100
    skewness = n.skew()
    kurtosis = n.kurt()
    corr = n.corrwith(n[target])# "SalePrice"
    #corr = corr.to_string()

    # Transform Data Analysis to Dataframe
    analyze = pd.DataFrame(columns=['Columns','types', 'counts', 'distincts', 'nulls', '% nulls', 'uniques', 'skewness', 'ku
    analyze['types'] = types
    analyze['counts'] = counts
    analyze['distincts'] = distincts
    analyze['nulls'] = nulls
    analyze['% nulls'] = per_nulls
    analyze['uniques'] = uniques
    analyze['skewness'] = skewness
    analyze['kurtosis'] = kurtosis
    analyze['Corr_Sales'] = corr
    analyze['Columns'] = analyze.index
    analyze = analyze.fillna(0).sort_values(by=['Corr_Sales','skewness'], ascending=False) #Fill Remaining Missing Values wi
    analyze = analyze.replace(["NaN"], 0).sort_values(by='Corr_Sales', ascending=False)
    analyze = analyze.reset_index(drop=True)

    print(colored("\nData Analysis for: ", 'green', attrs=['bold'])
          +colored(df_name, 'red', attrs=['bold'])
          + colored("\nData Shape:", 'green', attrs=['bold'])
          +colored(obs, 'magenta', attrs=['bold'])
         )

    return analyze, stats
```

```
Data Analysis for: raw_data
Data Shape:2930

Summary Statistics
```

	count	unique	top	freq	mean	std	min	25%	50%	75%	max
Order	2930.0	0	0	0	1.465500e+03	8.459625e+02	1.0	7.332500e+02	1465.5	2.197750e+03	2.930000e+03
PID	2930.0	0	0	0	7.144645e+08	1.887308e+08	526301100.0	5.284770e+08	535453620.0	9.071811e+08	1.007100e+09
MS SubClass	2930.0	0	0	0	5.738737e+01	4.263802e+01	20.0	2.000000e+01	50.0	7.000000e+01	1.900000e+02
MS Zoning	2930.0	7	RL	2273	0.000000e+00	0.000000e+00	0.0	0.000000e+00	0.0	0.000000e+00	0.000000e+00
Lot Frontage	2440.0	0	0	0	6.922459e+01	2.336533e+01	21.0	5.800000e+01	68.0	8.000000e+01	3.130000e+02
...	...	...	...	...	...	...	...	...	...	...	...
Mo Sold	2930.0	0	0	0	6.216041e+00	2.714492e+00	1.0	4.000000e+00	6.0	8.000000e+00	1.200000e+01
Yr Sold	2930.0	0	0	0	2.007790e+03	1.316613e+00	2006.0	2.007000e+03	2008.0	2.009000e+03	2.010000e+03
Sale Type	2930.0	10	WD	2536	0.000000e+00	0.000000e+00	0.0	0.000000e+00	0.0	0.000000e+00	0.000000e+00
Sale Condition	2930.0	6	Normal	2413	0.000000e+00	0.000000e+00	0.0	0.000000e+00	0.0	0.000000e+00	0.000000e+00
SalePrice	2930.0	0	0	0	1.807961e+05	7.988669e+04	12789.0	1.295000e+05	160000.0	2.135000e+05	7.550000e+05

82 rows × 11 columns

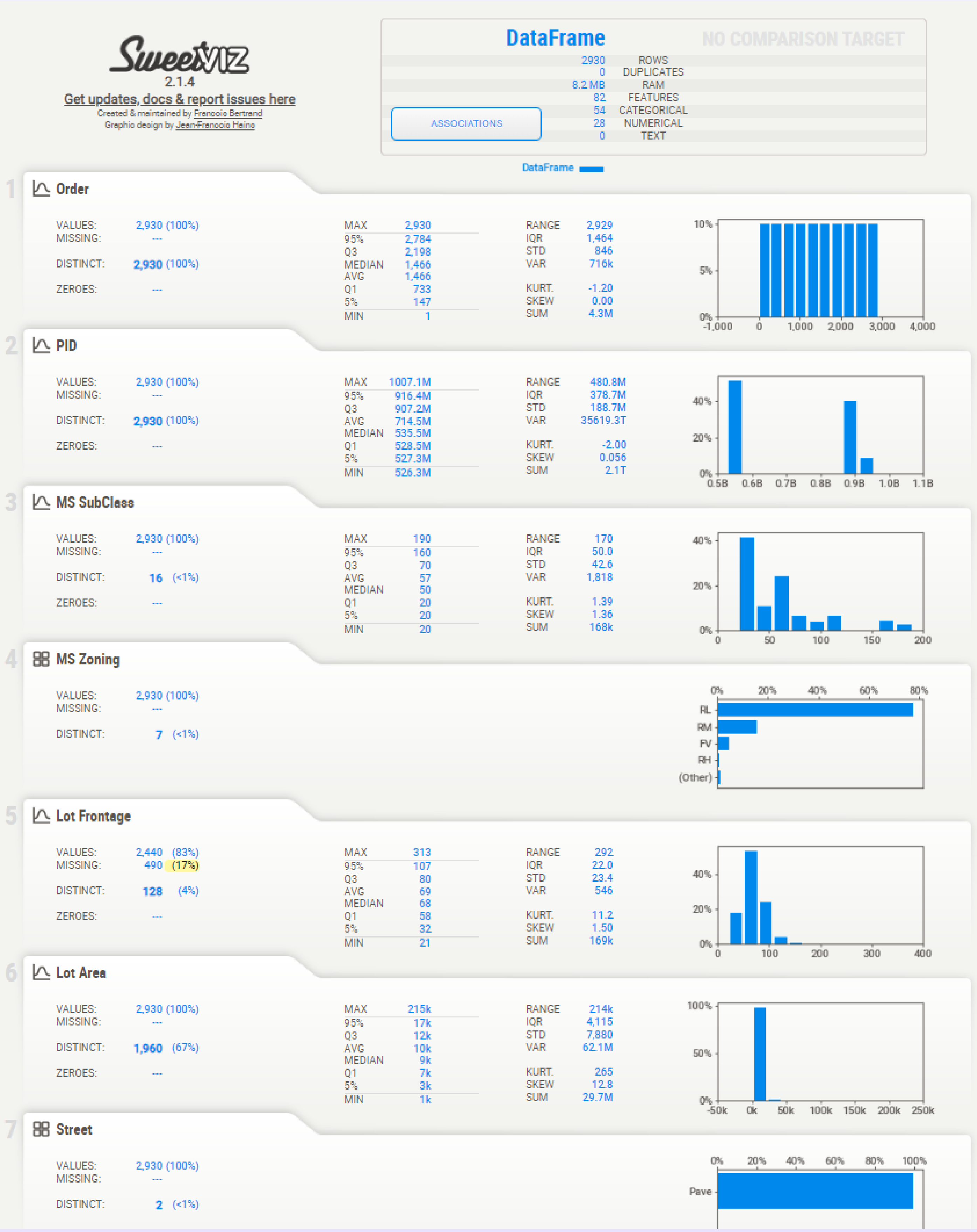
```
print(colored("\nData Analysis Summary:", 'cyan', attrs=['bold']))
analyze
```

```
Data Analysis Summary:
```

	Columns	types	counts	distincts	nulls	% nulls	uniques	skewness	kurtosis	Corr_Sales
0	SalePrice	int64	2930	1032	0	0.00000	0	1.743500	5.118900	1.000000
1	Overall Qual	int64	2930	10	0	0.00000	0	0.190634	0.052412	0.799262
2	Gr Liv Area	int64	2930	1292	0	0.00000	0	1.274110	4.137838	0.706780
3	Garage Cars	float64	2929	7	1	0.03413	0	-0.219836	0.244969	0.647877
4	Garage Area	float64	2929	604	1	0.03413	0	0.241994	0.951023	0.640401
...	...	...	...	...	...	...	...	...	...	...
77	MS SubClass	int64	2930	16	0	0.00000	0	1.357579	1.386775	-0.085092
78	Overall Cond	int64	2930	9	0	0.00000	0	0.574429	1.491450	-0.101697
79	Kitchen AbvGr	int64	2930	4	0	0.00000	0	4.313825	19.869743	-0.119814
80	Enclosed Porch	int64	2930	183	0	0.00000	0	4.014446	28.487205	-0.128787
81	PID	int64	2930	2930	0	0.00000	0	0.055886	-1.995146	-0.246521

82 rows × 10 columns

- Additional Automated Exploratory Data Analysis was performed using Sweetviz to realize visual representation.



4b) Data Cleansing & Features Engineering

In machine learning, feature selection is the method to reduce the number of input variables during developing predictive modelling. This reduction in input variables is necessary not only to minimize computational cost of modelling but also to achieve improved performance of the model itself.

Among widely practices feature selection approaches include statistical-based feature selection methods which use statistical measures to evaluate relationship between each input variable and the target variable and then select those exhibiting strongest relationship with the latter. While these methods can be both speedy and effective, however, the ultimate choice of statistical measure is largely dependent on data types of both of these variables.

Irrespective of the statistical measure being employed, two dominant feature selection techniques, that is supervised and unsupervised, exist where the former can be further categorized into wrapper, filter and intrinsic techniques. Filter-based feature selection methods employs statistical measures to evaluate correlation between input and output variables so that those exhibiting highest correlations are selected. Statistical measures employed in filter-based feature selection are normally univariate in nature since they evaluate relationship of single input variables one by one with target variable, disregarding their interaction with each other.

Consequently, adopting filter-based feature selection methods, the housing price prediction model approached filter engineering in three steps.

1) Data Encoding
2) Managing Multicollinearity
3) Final Data Cleansing
4) Applying Outlier Treatment

1) Prior to final features selection, data encoding of object or string columns was carried out to facilitate any statistical computation during features selection process. Hence, after copying original dataset, an automated method was created and employed to encode object data using Scikit-learn label encoder.

Data Cleansing Actions

Method to Encode Object Type Columns :
1) List Object Type Columns & Encode Data
2) Make Decoder to Decode Encoded Data

```python
# Method to encode object/string columns
def encoder(*name):
    # Accept an argument, return a value.
    n = name # Extract Dataframe by Name...this will create a 3d tuple
    n = (n[0]) # Convert Tuple to To Dataframe
    df_name = [x for x in globals() if globals()[x] is n][0] # Extract Name of Imported Dataframe

    # 1) List all Object/String Columns
    from sklearn import preprocessing
    cat_columns = n.select_dtypes(include=[object]) # Get Object Type Columns to Convert to Encoded Categories

    categorical_column = list(cat_columns.columns)# List of columns to for label encoding

    print(colored("\n\nColumns Requiring Encoding: \n", 'blue', attrs=['bold'])
                 + colored(categorical_column, 'green', attrs=['bold']))

    # Make Empty Dataframe to decode encoded data Later
    decode_features = pd.DataFrame()

    ##### Employ Scikit-Learn label encoding to encode object data #####
    lab_enc = preprocessing.LabelEncoder()
    for col in categorical_column:
            n[col] = lab_enc.fit_transform(n[col])
            name_mapping = dict(zip(lab_enc.classes_, lab_enc.transform(lab_enc.classes_)))

            ##### Decode Encoded Data #####
            feature_df = pd.DataFrame([name_mapping])
            feature_df = feature_df.astype(str)
            feature_df= (col + "_" + feature_df.iloc[0:])
            feature_df["Feature"] = col
            decode_features = decode_features.append(feature_df)# Append Dictionaries to Empty Dataframe for Later Decoding

            ##### Print Encoded Data #####
            print(colored("Feature: \n", 'blue', attrs=['bold'])
                  + colored(col, 'red', attrs=['bold'])
                  + colored("\nMapping: \n", 'blue', attrs=['bold'])
                  + colored(name_mapping, 'green', attrs=['bold'])
                  + colored("\n\nType n: ", 'blue', attrs=['bold'])
                  + colored(type(n), 'magenta', attrs=['bold'])
                 )
    n.head(3)

    ##### 2) Make Decoded Factor Dataframe with Description #####
    factor_list = decode_features.T # Transpose Dataframe and place in new dataframe
    factor_list = factor_list.replace(np.nan, "/") # nan values with forward slash
    factor_list["Factors"] = factor_list.astype(str).agg("".join,axis=1).replace(r'[^\w\s]|/', '', regex=True) # Aggregate A
    factor_list.reset_index() # Reset index before copying/assigning it to a new column
    factor_list['Description'] = factor_list.index  # Assign index to column

    return n, factor_list
```

```python
n, factor_list = encoder(df)
print(colored("\n\nEncoded Dataframe\n", 'blue', attrs=['bold'])
     + colored(type(n), 'blue', attrs=['bold']))

df = n.copy()
```

```
Columns Requiring Encoding:
['MS Zoning', 'Street', 'Alley', 'Lot Shape', 'Land Contour', 'Utilities', 'Lot Config', 'Land Slope', 'Neighborhood', 'Con
dition 1', 'Condition 2', 'Bldg Type', 'House Style', 'Roof Style', 'Roof Matl', 'Exterior 1st', 'Exterior 2nd', 'Mas Vnr T
ype', 'Exter Qual', 'Exter Cond', 'Foundation', 'Bsmt Qual', 'Bsmt Cond', 'Bsmt Exposure', 'BsmtFin Type 1', 'BsmtFin Type
2', 'Heating', 'Heating QC', 'Central Air', 'Electrical', 'Kitchen Qual', 'Functional', 'Fireplace Qu', 'Garage Type', 'Gar
age Finish', 'Garage Qual', 'Garage Cond', 'Paved Drive', 'Pool QC', 'Fence', 'Misc Feature', 'Sale Type', 'Sale Conditio
n']
Feature:
MS Zoning
Mapping:
{'A (agr)': 0, 'C (all)': 1, 'FV': 2, 'I (all)': 3, 'RH': 4, 'RL': 5, 'RM': 6}

Type n: <class 'pandas.core.frame.DataFrame'>
Feature:
Street
Mapping:
{'Grvl': 0, 'Pave': 1}
```

2) Subsequently, multicollinearity was managed by another automated method, as follows:

Method to Eliminate Columns with High Multicollinearity

1) Calculate Variance Inflation Factor
2) Delete features with VIF S above 2.4 but with no significant relationship with target variable
3) Keep features VIF S above 2.4 but with significant relationship with target variable to avoid information loss

Multicollinearity refers to correlation between two or more independent variables which increases standard error (precision of the estimate) of the coefficient. Hence, features exhibiting high multicollinearity can overinflate standard error, thereby, decreasing precision of the estimate. While multicollinearity enlarges model variance, it also expands model dimensions without necessarily enhancing information and so distorts model explainability.

Multicollinearity can easily be computed by the variance inflation factor (VIF) which not only picks out correlation between independent variables but also strength of these correlations. Although, most research papers regard a VIF > 10 as an indicator of strong multicollinearity, nevertheless, there some scholars suggest to select a more cautious threshold of 2.5 which can signal considerable collinearity. Accordingly, Ames House Prediction model implemented a conservative VIF threshold of 2.5 with low correlation to target variable.

```python
def vifs(*name):
    n = name # Extract Dataframe by Name...this will create a 3d tuple
    n = (n[0]) # Convert Tuple to To Dataframe

    analyze, stats = analysis(n) # Call function 'analysis'
    analyze = analyze[(analyze.Columns != target)] # Remove target Column function 'analysis'

    df_name = [x for x in globals() if globals()[x] is n][0] # Extract Name of Imported Dataframe

    vifs = pd.Series(np.linalg.inv(n.corr().to_numpy()).diagonal(),
                     index=n.columns,
                     name='VIF')

    vifs   = vifs.drop([target]); # Remove Traget 'SalePrice' Column
    vifs = vifs.to_frame()
    vifs['Columns'] = vifs.index
    vifs = vifs.sort_values('VIF', ascending=False)

    # Merge with Analysis to get Correlation with Target Variable
    vifs = pd.merge(vifs, analyze, on='Columns', how='left')
    vifs = vifs[(vifs['Corr_Sales'] < 0) & (vifs['VIF']>2.4)]

    vifs = vifs.reset_index()
    vifs = vifs.sort_values('Columns', ascending=True)
    drop1 = vifs.Columns.values.tolist()

    return drop1
```

3) Using an additional automated method, statistical measures were then employed with supervised filter-based feature selection technique. Firstly, aforementioned "vif" method was called to identify columns exhibiting high multicollinearity but low correlation with sales. Then, columns with less unique features were marked. Consequently, highly skewed columns with low correlation to target were also enlisted. Lastly, Columns with extremely high null values were also keyed out. The lists were then combined to filter these columns out of the data-frame.

Method to Drop Columns

1) With High Multicollinearity but Low Correlation to Target
2) With Uniques < 2
3) With High Skewness and Low Correlation to Target
4) Drop Columns With High Nan Values

```
def drop_cols(*name):
    n = name # Extract Dataframe by Name...this will create a 3d tuple
    n = (n[0]) # Convert Tuple to To Dataframe
    df_name = [x for x in globals() if globals()[x] is n][0] # Extract Name of Imported Dataframe

    # 1) Drop Columns with With High Multicollinearity & Low Correlation to Sales
    drop1 = vifs(n) # Call function 'vifs'

    analyze, stats = analysis(n) # Call function 'analysis'
    print(analyze)

    n = n.fillna(0) #Fill Remaining Missing Values with Zero
    # Find Mean of Null, Nan and Zero Values Before Any Drops
    m0 = n.isin([' ','NULL','NaN', 0]).mean().sort_values(axis=0, ascending=False, inplace=False, kind='quicksort', na_posit

    #  2) Drop Columns with Unique Values Less than threshold
    unique_counts = pd.DataFrame.from_records([(col, n[col].nunique()) for col in n.columns], # get unique counts
                          columns=['Column_Name', 'Unique']).sort_values(by=['Unique'])
    unique = unique_counts[(unique_counts['Unique'] < 2)] #If threshold is Lesss than 2 then
    drop2 = (unique['Column_Name'].tolist()) # First List of columns to drop

    # 3) Drop Highly Skewed & Low Sales Correlation Columns OR Low Sales Correlation Columns
    drop3 = analyze[(analyze['Corr_Sales'] != 1) & (analyze['skewness'] > 0) & (analyze['Corr_Sales'] < 0) | (analyze['Corr_

    print(colored("\nDrop 3: \n ", 'blue', attrs=['bold'])
          +colored(drop3, 'magenta', attrs=['bold']))

    drop3 = drop3.sort_values(by='Columns', ascending=True)
    drop3 = drop3['Columns'].tolist() # Second List of columns to drop

    drop = drop1 + drop2 + drop3 + delete_features # Final List of columns to drop
    print(drop)
    n = n.drop(drop,1) # Drop Columns using Final List of columns to drop

    # Find Mean of Null, Nan and Zero Values Before Dropping
    m1 = n.isin([' ','NULL','NaN', 0]).mean().sort_values(axis=0, ascending=False, inplace=False, kind='quicksort', na_posit

    # 4) Drop Columns With High Nan Values
    drop_thresh = .90 # Identify Drop Threshold
    n = n.loc[:, df.isin([' ','NULL', 'NaN',0]).mean() < drop_thresh] # drop columns if Mean is > 0.90

    #df = df.fillna(0) #Fill Remaining Missing Values with Zero
    n = n.replace(["NaN"], 0).sort_values(by=target, ascending=False) # Replace all Nan Values with Zero

    # Find Mean of Null, Nan and Zero Values After Dropping
    m2 = n.isin([' ','NULL','NaN']).mean().sort_values(axis=0, ascending=False, inplace=False, kind='quicksort', na_position

    #Print Results
    print(colored("\nDataframe Average Null Values Before Any Drops\n ", 'blue', attrs=['bold'])
          +colored(m0, 'magenta', attrs=['bold'])
          +colored("\n\n Low Correlation Columns to Drop: ", 'green', attrs=['bold'])
          + colored(drop1, 'red', attrs=['bold'])
          +colored("\n\nDataframe Average Null Values After Low Correlation Columns Drop\n ", 'green', attrs=['bold'])
          +colored(m1, 'red', attrs=['bold'])
          +colored("\n\n Drop Columns if Mean is > 0.90 \n", 'green', attrs=['bold'])
          + colored("\nDataframe Average Null Values After Drop and 'Nan' Replacement\n", 'blue', attrs=['bold'])
          +colored(m2, 'magenta', attrs=['bold'])
          +colored(type(m2), 'magenta', attrs=['bold'])
         )
    return n

# Return Function
n = drop_cols(df)
df = n.copy()
```

4) Lastly, an automated method was employed to replace outliers with mode, that is, most frequent value.

Method to Explore and Adjust Outliers

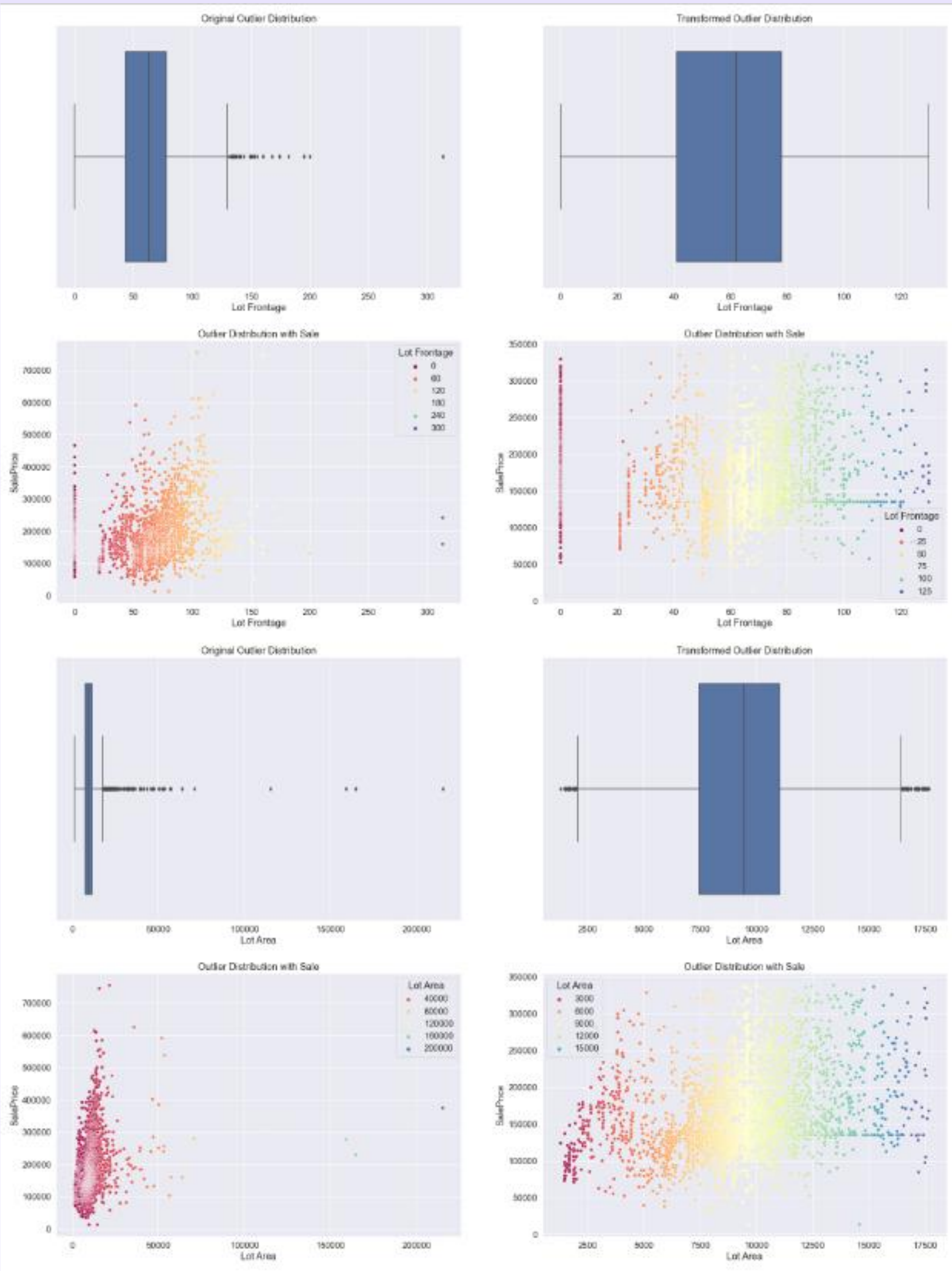
Replace Outlier Values with Mode (Most Frequent Value)

```
def outliers(*name):
    n = name # Extract Dataframe by Name...this will create a 3d tuple
    n = (n[0]) # Convert Tuple to To Dataframe
    df_name = [x for x in globals() if globals()[x] is n][0] # Extract Name of Imported Dataframe

    cols = n.columns # All Columns

    # Numeric Columns
    numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
    numeric_cols = n.select_dtypes(include=numerics)
    numeric_cols = numeric_cols.columns.tolist()

    # Object Columns
    categorical_cols = list(set(cols) - set(numeric_cols))

    # Skewed Columns
    skewed_cols = analyze[(analyze['skewness'] > 0) | (analyze['skewness'] < 0)]
    skewed_cols = skewed_cols['Columns'].tolist()

    # Replace Outliers
    for col in n.columns:
        if col in skewed_cols:
            print(colored(col, 'magenta', attrs=['bold'])
                  + colored(" is Skewed... ", 'blue', attrs=['bold'])
                 )

            mode = n[col].mode()
            mode = mode[0]

            if col in numeric_cols:
                print(colored(col, 'magenta', attrs=['bold'])
                      + colored(" Column Type is: ", 'blue', attrs=['bold'])
                      + colored(n[col].dtypes, 'red', attrs=['bold'])
                     )
                #Calculate quantiles and IQR
                Q1 = n[col].quantile(0.25) # Same as np.percentile but maps (0,1) and not (0,100)
                Q3 = n[col].quantile(0.75)
                IQR = Q3 - Q1
                # Replace with Mode
                n[col] = np.where((n[col] < (Q1 - 1.5 * IQR)) | (n[col] > (Q3 + 1.5 * IQR)), mode, n[col])

                print(colored("\nReplaced ", 'blue', attrs=['bold'])
                      +colored(col, 'magenta', attrs=['bold'])
                      + colored(" Skewed Values by Mode: ", 'blue', attrs=['bold'])
                      + colored(mode, 'red', attrs=['bold'])
                      + colored('\n', 'magenta', attrs=['bold'])
                      + colored((n[col]), 'green', attrs=['bold'])
                     )

            else:
                print("")
    df_transformed = n.copy()
    return df_transformed
```

This was followed by another method to present graphical illustration of original and adjusted data side by side:

Visualize Data Distribution of both Original and New Dataframe

```
df_numeric = df_transformed.select_dtypes(exclude='object')
for col in df_numeric.columns: # Iterate over each Column and Create Visuals
########## New Figure ###########################################################################################
    figa = plt.figure(figsize=(30, 20))
    sns.set(font_scale=1.5)

    fig1 = figa.add_subplot(221); sns.boxplot(df[col])
    fig1 = plt.title('Original Outlier Distribution')

    fig2 = figa.add_subplot(222); sns.boxplot(df_transformed[col])
    fig2 = plt.title('Transformed Outlier Distribution')

    fig3 = figa.add_subplot(223);
    sns.scatterplot(x = df[col], y = df[target], hue=df[col], palette= 'Spectral')
    fig3 = plt.title('Outlier Distribution with Sale')

    fig4 = figa.add_subplot(224);
    sns.scatterplot(x = df_transformed[col], y = df_transformed[target], hue=df_transformed[col], palette= 'Spectral')
    fig4 = plt.title('Outlier Distribution with Sale')

########## New Figure ###########################################################################################
figb = plt.figure(figsize=(20, 10))
sns.set(font_scale=1.5)

fig5 = figb.add_subplot(221);
fig5 = sns.distplot(df[target][~df[target].isnull()], axlabel="Nor. Dist.", fit=st.norm, fit_kws={"color":"red"})
fig5 = plt.title('Distribution of Sales Price')
(mu5, sigma5) = st.norm.fit(df[target])
fig5 = plt.legend(['Normal Distribution \n ($\mu=$ {:.2f} and  $\sigma=$ {:.2f} )'.format(mu5, sigma5)], loc='best', fancybo

fig6 = figb.add_subplot(222);
fig6 = sns.distplot(df_transformed[target][~df_transformed[target].isnull()], axlabel="Nor. Dist.", fit=st.norm, fit_kws={"c
fig6 = plt.title('Distribution of Sales Price')
(mu6, sigma6) = st.norm.fit(df_transformed[target])
fig6 = plt.legend(['Normal Distribution \n ($\mu=$ {:.2f} and  $\sigma=$ {:.2f} )'.format(mu6, sigma6)], loc='best', fancybo

########## New Figure ###########################################################################################
figc = plt.figure(figsize=(20, 10))
sns.set(font_scale=1.5)
fig7 = figc.add_subplot(221);
fig7 = st.probplot(df[target][~df[target].isnull()], plot=plt)
fig7 = plt.title('Probability Plot')

fig8 = figc.add_subplot(222);
fig8 = st.probplot(df_transformed[target][~df_transformed[target].isnull()], plot=plt)
fig8 = plt.title('Probability Plot')
```


Original Outlier Distribution
Lot Frontage
Transformed Outlier Distribution
Lot Frontage
Outlier Distribution with Sale
SalePrice
Lot Frontage
Outlier Distribution with Sale
SalePrice
Lot Frontage
Original Outlier Distribution
Lot Area
Transformed Outlier Distribution
Lot Area
Outlier Distribution with Sale
SalePrice
Lot Area
Outlier Distribution with Sale
SalePrice
Lot Area

5) Summary of Training Different Regression Models

5a) Machine Learning Regression Algorithm Development Approach

Using random search, an automated optimal hyper-parameter search method was created to find optimal model parameters. This approach was employed because best hyperparameters are not automatically learnt within estimators and its manual search not only slows down model development but may also lead to ineffective model construction. Hence, an exhaustive random search approach was used to pass parameter arguments to the constructor in order to find optimal hyperparameters for each model, as shown below:

RIDGE REGRESSION MODELS

Random Search Method to Find 'Best Parameters'to 'Build Ridge Regression Model WITH Optimal Hyperparameters'

```
def random_search_r(X_train, y_train):

    # Define Evaluation
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

    # Define Search Space
    space = dict()
    space['solver'] = ['auto','svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga', 'lbfgs']
    space['alpha'] = loguniform(1e-5, 100)
    space['fit_intercept'] = [True, False]
    space['normalize'] = [True, False]
    space['max_iter'] = [500, 1000, 1500]

    # Define Model
    ridge_model = Ridge(random_state=rs, max_iter=1000)

    # Define Search
    search = RandomizedSearchCV(ridge_model, space, n_iter=500, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv, random_

    # Execute Search
    result_r = search.fit(X_train, y_train)

    # Summarize Result
    best_score_r = result_r.best_score_

    best_params_r = result_r.best_params_
    best_params_r["best_score"] = best_score_r # Add 'best_score' to 'best_params' Dictionary

    best_params_r = pd.DataFrame([best_params_r]) # Dictionary To dataframe

    # Get Optimal Variables
    opt_alpha_r = best_params_r['alpha'].iloc[0]
    opt_alpha_r = '{:.6f}'.format(opt_alpha_r)
    opt_alpha_r = float(opt_alpha_r) # Back to Float

    opt_solver_r = best_params_r['solver'].iloc[0]
    opt_fit_intercept_r = best_params_r['fit_intercept'].iloc[0]
    opt_normalize_r = best_params_r['normalize'].iloc[0]
    opt_max_iter_r = best_params_r['max_iter'].iloc[0]

    # Define Optimal Parameters
    optimal_params_r = {'solver': opt_solver_r,
                                    'alpha': opt_alpha_r,
                                    'fit_intercept': opt_fit_intercept_r,
                                    'normalize': opt_normalize_r,
                                    'max_iter': opt_max_iter_r,
                                    'random_state': rs}

    return best_params_r, optimal_params_r

best_params_r, optimal_params_r = random_search_r(X_train, y_train) # Call Method 'random_search_r' to get optimal hyperpara
best_params_r
```

	alpha	fit_intercept	max_iter	normalize	solver	best_score
0	75.322052	True	500	False	auto	-23514.379169

LASSO REGRESSION MODELS

Random Search Method to Find 'Best Parameters'to 'Build Lasso Regression Model WITH Optimal Hyperparameters'

```
def random_search_l(X_train, y_train):

    # Define Evaluation
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

    # Define Search Space
    space = dict()
    space['alpha'] = loguniform(1e-5, 100)
    space['fit_intercept'] = [True, False]
    space['normalize'] = [True, False]
    space['precompute'] = [True, False]
    space['tol'] = loguniform(1e-4, 100)
    space['selection'] = ['cyclic', 'random']

    # Define Model
    lasso_model = Lasso(random_state=rs, max_iter=1000)

    # Define Search
    search = RandomizedSearchCV(lasso_model, space, n_iter=1000, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv, random

    # Execute Search
    result_l = search.fit(X_train, y_train)

    # Summarize Result
    best_score_l = result_l.best_score_
    best_params_l = result_l.best_params_
    best_params_l["best_score"] = best_params_l # Add 'best_score' to 'best_params' Dictionary

    best_params_l = pd.DataFrame([best_params_l]) # Dictionary To dataframe

    # Get Optimal Variables
    opt_alpha_l = best_params_l['alpha'].iloc[0]
    opt_fit_intercept_l = best_params_l['fit_intercept'].iloc[0]
    opt_normalize_l = best_params_l['normalize'].iloc[0]
    opt_precompute_l = best_params_l['precompute'].iloc[0]
    opt_tol_l = best_params_l['tol'].iloc[0]
    opt_selection_l = best_params_l['selection'].iloc[0]

    # Define Optimal Parameters
    optimal_params_l = {'alpha': opt_alpha_l,
                                    'fit_intercept': opt_fit_intercept_l,
                                    'normalize': opt_normalize_l,
                                    'precompute': opt_precompute_l,
                                    'tol': opt_tol_l,
                                    'selection': opt_selection_l}


    return best_params_l, optimal_params_l

best_params_l, optimal_params_l = random_search_l(X_train, y_train) # Call Method 'random_search_l' to get optimal hyperpara
best_params_l
```

	alpha	fit_intercept	normalize	precompute	selection	tol	best_score
0	9.126019	True	True	False	random	0.031602	{'alpha': 9.126018989876911, 'fit_intercept': ...

ELASTIC-NET REGRESSION MODELS

Randomized Search Method to Find 'Best Parameters'to 'Build Elastic Net Regression Model WITH Optimal Hyperparameters'

```
def random_search_en(X_train, y_train):

    # Define Evaluation
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

    # Define Search Space
    space = dict()
    space['alpha'] = loguniform(1.0, 1.5, 3, 3.5)
    space['l1_ratio'] = loguniform(0.5, 1)
    space['fit_intercept'] = [True, False]
    space['normalize'] = [True, False]
    space['precompute'] = [True, False]
    space['copy_X'] = [True, False]
    space['tol'] = [1e-4, 1e-6, 1e-9]
    space['warm_start'] = [True, False]
    space['positive'] = [True, False]
    space['selection'] = ['cyclic', 'random']

    # Define Model
    en_model = ElasticNet(random_state=rs, max_iter=1000)

    # Define Search
    search_en = RandomizedSearchCV(en_model, space, n_iter=1000, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv, randon

    # Execute Search
    result_en = search_en.fit(X_train, y_train)

    # Summarize esult
    best_score_en= result_en.best_score_

    best_params_en = result_en.best_params_
    best_params_en["best_score"] = best_params_en # Add 'best_score' to 'best_params' Dictionary
    best_params_en = pd.DataFrame([best_params_en]) # Dictionary To dataframe

    # Get Optimal Variables
    opt_alpha_en = best_params_en['alpha'].iloc[0]
    opt_l1_ratio_en = best_params_en['l1_ratio'].iloc[0]
    opt_fit_intercept_en = best_params_en['fit_intercept'].iloc[0]
    opt_normalize_en = best_params_en['normalize'].iloc[0]
    opt_precompute_en = best_params_en['precompute'].iloc[0]
    opt_copy_X_en = best_params_en['copy_X'].iloc[0]
    opt_tol_en = best_params_en['tol'].iloc[0]
    opt_tol_en= best_params_en['tol'].iloc[0]
    opt_warm_start_en = best_params_en['warm_start'].iloc[0]
    opt_positive_en = best_params_en['positive'].iloc[0]
    opt_selection_en = best_params_en['selection'].iloc[0]

    # Define Optimal Parameters
    optimal_params_en = {'alpha': opt_alpha_en,
                                    'l1_ratio': opt_l1_ratio_en,
                                    'fit_intercept': opt_fit_intercept_en,
                                    'normalize': opt_normalize_en,
                                    'precompute': opt_precompute_en,
                                    'copy_X': opt_copy_X_en,
                                    'tol': opt_tol_en,
                                    'warm_start': opt_warm_start_en,
                                    'positive': opt_positive_en,
                                    'selection': opt_selection_en}
    return best_params_en, optimal_params_en

best_params_en, optimal_params_en = random_search_en(X_train, y_train) # Call Function 'random_search_en' to get optimal hyp
best_params_en
```

	alpha	copy_X	fit_intercept	l1_ratio	normalize	positive	precompute	selection	tol	warm_start	best_score
0	6.9506	True	True	0.987434	False	True	False	random	1.000000e-09	True	{'alpha': 6.950599680152801, 'copy_X': True, '...

XGBOOST REGRESSION MODELS ¶

Random Search Method to Find 'Best Parameters'to 'Build XGBoost Regression Model WITH Optimal Hyperparameters'

```
def random_search_xgb(X_train, y_train):

    # Define Evaluation
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

    # Define Search Space
    space = dict()
    space['learning_rate'] = [0.1, 20]
    space['subsample'] = [0.0, 1.0]
    space['criterion'] = ['friedman_mse', 'squared_error', 'mse']
    space['max_features'] = ['auto', 'sqrt', 'log2']

    # Define Model
    xgb_model = GradientBoostingRegressor(random_state=rs)

    # Define Search
    search_xgb = RandomizedSearchCV(xgb_model, space, n_iter=1000, scoring='neg_mean_absolute_error', n_jobs=-1, cv=cv, rand

    # Execute Search
    result_xgb = search_xgb.fit(X_train, y_train)

    # Summarize Result
    best_score_xgb= result_xgb.best_score_
    best_params_xgb = result_xgb.best_params_
    best_params_xgb["best_score"] = best_params_xgb # Add 'best_score' to 'best_params' Dictionary

    best_params_xgb = pd.DataFrame([best_params_xgb]) # Dictionary To dataframe

    # Get Optimal Variables
    opt_learning_rate_xgb = best_params_xgb['learning_rate'].iloc[0]
    opt_subsample_xgb = best_params_xgb['subsample'].iloc[0]
    opt_criterion_xgb = best_params_xgb['criterion'].iloc[0]
    opt_max_features_xgb = best_params_xgb['max_features'].iloc[0]

    # Define Optimal Parameters
    optimal_params_xgb = {'learning_rate': opt_learning_rate_xgb,
                          'subsample': opt_subsample_xgb,
                          'criterion': opt_criterion_xgb,
                          'max_features': opt_max_features_xgb}

    return best_params_xgb, optimal_params_xgb

best_params_xgb, optimal_params_xgb = random_search_xgb(X_train, y_train) # Call Function 'random_search_xgb' to get optimal
best_params_xgb
```

	subsample	max_features	learning_rate	criterion	best_score
0	1.0	auto	0.1	friedman_mse	{'subsample': 1.0, 'max_features': 'auto', 'le...

5b) Summarizing Employed Models

Following four main regression models have been used to predict house prices.

1) Ridge Regression (RR) Models

Due to the dependant nature of multiple variables in predicting variable 'y' where the output is influenced by multiple factors, ridge regression algorithm was employed. A single method was created to measure predictive capability of the following two RR models:

Method to 'Build Ridge Regression WITH & Without Optimal Hyperparameters'

```
# Build a Regression model with Optimal Class Weights
def build_op_r(X_train, y_train, X_test, threshold=0.5, best_params=None):

    model = Ridge(random_state=rs, max_iter = 1000)

    # If best parameters are provided
    if best_params:
        model = make_pipeline(RobustScaler(),
                              Ridge(solver = best_params_r['solver'].iloc[0],
                                    alpha = best_params_r['alpha'].iloc[0],
                                    fit_intercept = best_params_r['fit_intercept'].iloc[0],
                                    normalize = best_params_r['normalize'].iloc[0],
                                    max_iter = best_params_r['max_iter'].iloc[0],
                                    random_state=rs
                                          )
                             )
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model[:-1].get_feature_names_out()
        X_train_preprocessed = pd.DataFrame(model[:-1].transform(X_train), columns=feature_names) # X_train to compute stanc
        features = pd.DataFrame(model[-1].coef_ * X_train_preprocessed.std(axis=0), columns=["Coefficients"], index=feature_


    else:
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model._check_feature_names
        X_train_preprocessed = X_train.copy() #.T # Transpose X_train to compute standard deviation of the related feature

        # Multiply model coefficients by std of features to reduce all coefficients to same unit of measure.
        features  = pd.DataFrame(model.coef_ * X_train_preprocessed.std(axis=0), columns=['Coefficients'], index=X_train.col

    features['Features'] = features.index
    #features = features.loc[features['Coefficients'] > 0] # Uncomment if only features with Coeff > 0 are required
    features  = features.sort_values(by=['Coefficients'], ascending=False).reset_index(drop=True)

    # Plot Features
    data = features.copy()
    data = data[data["Coefficients"] != 0]
    data.set_index('Features', inplace=True)
    data.plot.barh(figsize=(30,10), color='green')

    title = "Feature Importance in Predicting " + target
    plt.title(title)
    plt.axvline(x=0, color="red")
    plt.xlabel("Coefficient Values Corrected by Features' std. dev.")
    plt.subplots_adjust(left=0.3)

    # Get Model Results
    res = result(model)

    return model, res, features, pred
```

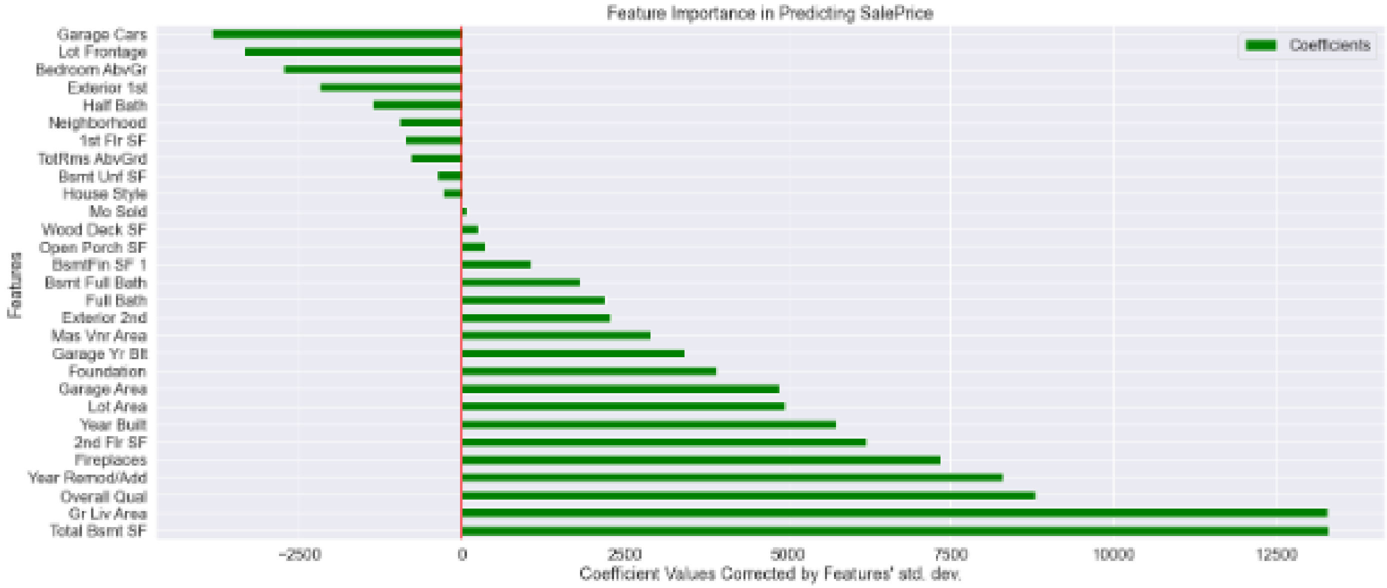
1a) RR Model 1 Without Optimal Hyperparameter Tuning: A simple algorithm was created without optimal hyperparameter tuning.

Model 1: RIDGE Without 'Optimal Hypertuned Parameters'

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model1, res1, features1, pred1 = build_op_r(X_train, y_train, X_test, best_params=None) # Call Method to 'Build Logistic Reg


#Collect & Append Results
r1 = np.array2string(res1)####
r1 = pd.DataFrame([r1.split(';') for x in r1.split('\n')])
r1 = r1.rename(columns={0: 'SCORE'})
r1['MODEL'] = 'Regression'
r1['MODEL#'] = 'Model 1'
r1
```

```
Model Directory:
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_check_feature_names', '_check_n_features', '_decision_function', '_estimator_type', '_get_param_names', '_get_tags', '_more_tags', '_normalize', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_set_intercept', '_validate_data', 'alpha', 'coef_', 'copy_X', 'feature_names_in_', 'fit', 'fit_intercept', 'get_params', 'intercept_', 'max_iter', 'n_features_in_', 'n_iter_', 'normalize', 'positive', 'predict', 'random_state', 'score', 'set_params', 'solver', 'tol']
```

	SCORE	MODEL	MODEL#
0	-1.30784598e+09	Regression	Model 1



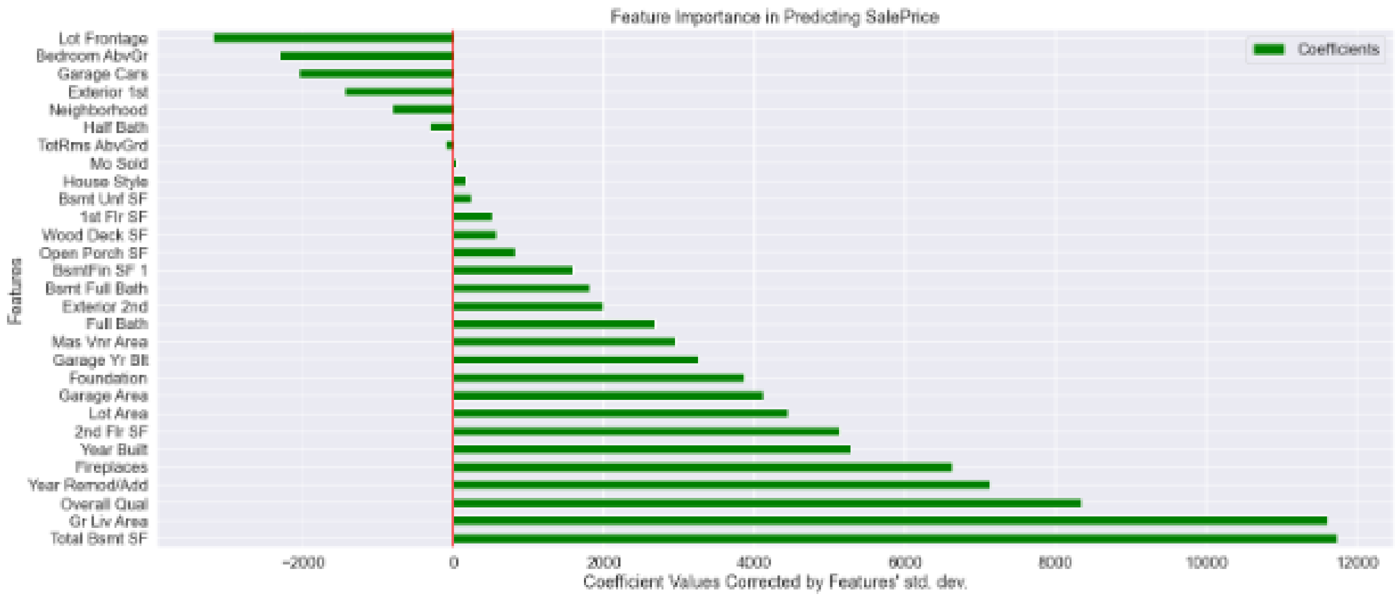
1b) RR Model 2 WITH Optimal Hyperparameter Tuning: A modified algorithm with auto-hyper-tuned parameters was created using random search method described above.

Model 2: RIDGE With 'Optimal Hypertuned Parameters'

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model2, res2, features2, pred2 = build_op_r(X_train, y_train, X_test, best_params=optimal_params_r) # Call Method to 'Build

#Collect & Append Results
r2 = np.array2string(res2)####
r2 = pd.DataFrame([r2.split(';') for x in r2.split('\n')])
r2 = r2.rename(columns={0: 'SCORE'})
r2['MODEL'] = 'Regression Optimal'
r2['MODEL#'] = 'Model 2'
r2
```

```
Model Directory:
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__len__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setsta
te__', '__sizeof__', '__sklearn_is_fitted__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_can_inverse_transfor
m', '_can_transform', '_check_feature_names', '_check_fit_params', '_check_n_features', '_estimator_type', '_final_estimator',
'_fit', '_get_param_names', '_get_params', '_get_tags', '_iter', '_log_message', '_more_tags', '_replace_estimator', '_repr_htm
l_', '_repr_html_inner', '_repr_mimebundle_', '_required_parameters', '_set_params', '_sk_visual_block_', '_validate_data', '_v
alidate_names', '_validate_steps', 'classes_', 'decision_function', 'feature_names_in_', 'fit', 'fit_predict', 'fit_transform',
'get_feature_names_out', 'get_params', 'inverse_transform', 'memory', 'n_features_in_', 'named_steps', 'predict', 'predict_log_
proba', 'predict_proba', 'score', 'score_samples', 'set_params', 'steps', 'transform', 'verbose']
```

	SCORE	MODEL	MODEL#
0	-1.30393778e+09	Regression Optimal	Model 2



2) Lasso Regression (LR) Models

A single method was employed to run lasso regression models with and without optimal hyper-parameter tuning:

Method to 'Build LASSO Regression WITH & Without Optimal Hyperparameters'

```
# Build a LASSO model with Optimal Class Weights
def build_op_l(X_train, y_train, X_test, threshold=0.5, best_params=None):

    model = Ridge(random_state=rs, max_iter = 1000)

    # If best parameters are provided
    if best_params:
        model = make_pipeline(RobustScaler(),
                              Lasso(alpha = best_params_l['alpha'].iloc[0],
                                    fit_intercept = best_params_l['fit_intercept'].iloc[0],
                                    normalize = best_params_l['normalize'].iloc[0],
                                    precompute = best_params_l['precompute'].iloc[0],
                                    tol = best_params_l['tol'].iloc[0],
                                    selection = best_params_l['selection'].iloc[0],
                                    random_state=rs
                                         )
                              )
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model[:-1].get_feature_names_out() #https://scikit-learn.org/stable/auto_examples/inspection/plot_li
        X_train_preprocessed = pd.DataFrame(model[:-1].transform(X_train), columns=feature_names) # X_train to compute stand
        features = pd.DataFrame(model[-1].coef_ * X_train_preprocessed.std(axis=0), columns=["Coefficients"], index=feature_

    else:
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model._check_feature_names
        X_train_preprocessed = X_train.copy() #.T # Transpose X_train to compute standard deviation of the related feature
        # Multiply model coefficients by std of features to reduce all coefficients to same unit of measure.
        features  = pd.DataFrame(model.coef_ * X_train_preprocessed.std(axis=0), columns=['Coefficients'], index=X_train.col

    features['Features'] = features.index
    #features = features.loc[features['Coefficients'] > 0] # Uncomment if only features with Coeff > 0 are required
    features  = features.sort_values(by=['Coefficients'], ascending=False).reset_index(drop=True)

    # Plot Features
    data = features.copy()
    data = data[data["Coefficients"] != 0]
    data.set_index('Features', inplace=True)
    data.plot.barh(figsize=(30,10), color='green')

    title = "Feature Importance in Predicting " + target
    plt.title(title)
    plt.axvline(x=0, color="red")
    plt.xlabel("Coefficient Values Corrected by Features' std. dev.")
    plt.subplots_adjust(left=0.3)

    # Get Model Results
    res = result(model)

    return model, res, features, pred
```

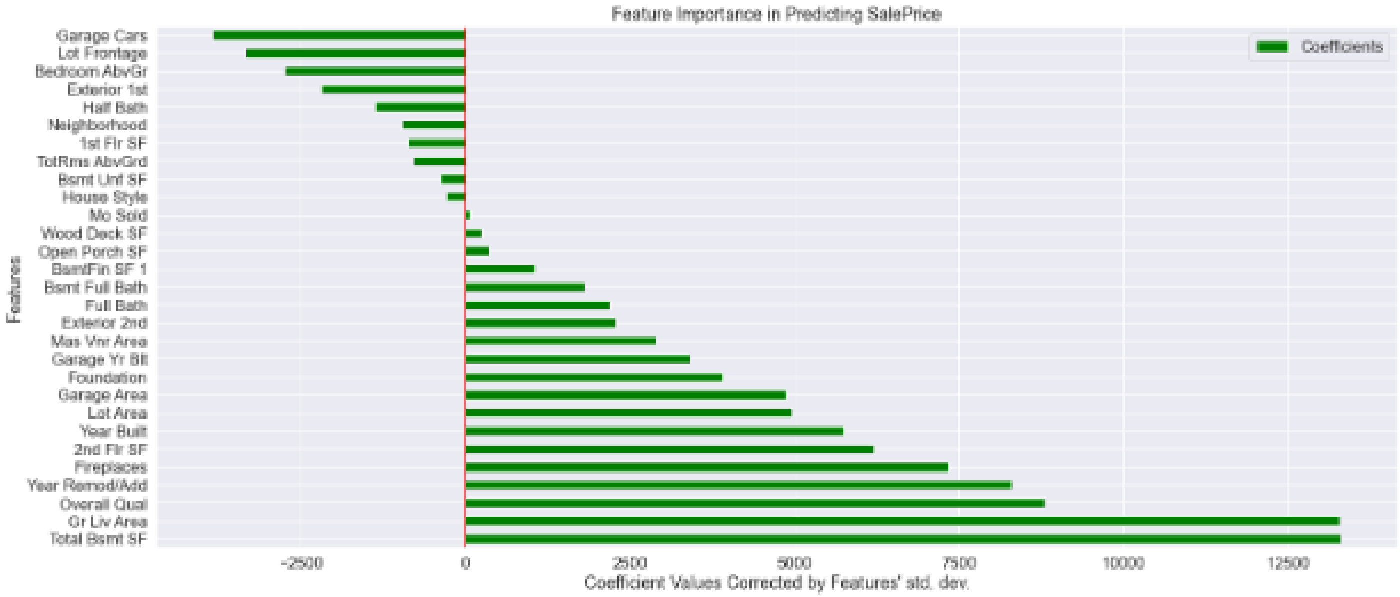
2a) LR Model 1 Without Optimal Hyperparameter Tuning: A simple algorithm was created without any hyperparameter tuning.

Model 3: LASSO Without 'Optimal Hypertuned Parameters'

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model3, res3, features3, pred3 = build_op_l(X_train, y_train, X_test, best_params=None) # Call Method to 'Build Logistic Reg

#Collect & Append Results
r3 = np.array2string(res3)####
r3 = pd.DataFrame([r3.split(';') for x in r3.split('\n')])
r3 = r3.rename(columns={0: 'SCORE'})
r3['MODEL'] = 'Lasso'
r3['MODEL#'] = 'Model 3'
r3
```

```
Model Directory:
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__geta
ttribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook_
_', '__weakref__', '_abc_impl', '_check_feature_names', '_check_n_features', '_decision_function', '_estimator_type', '_get_par
am_names', '_get_tags', '_more_tags', '_normalize', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_set_intercept',
'_validate_data', 'alpha', 'coef_', 'copy_X', 'feature_names_in_', 'fit', 'fit_intercept', 'get_params', 'intercept_', 'max_ite
r', 'n_features_in_', 'n_iter_', 'normalize', 'positive', 'predict', 'random_state', 'score', 'set_params', 'solver', 'tol']
```

	SCORE	MODEL	MODEL#
0	-1.30784598e+09	Lasso	Model 3



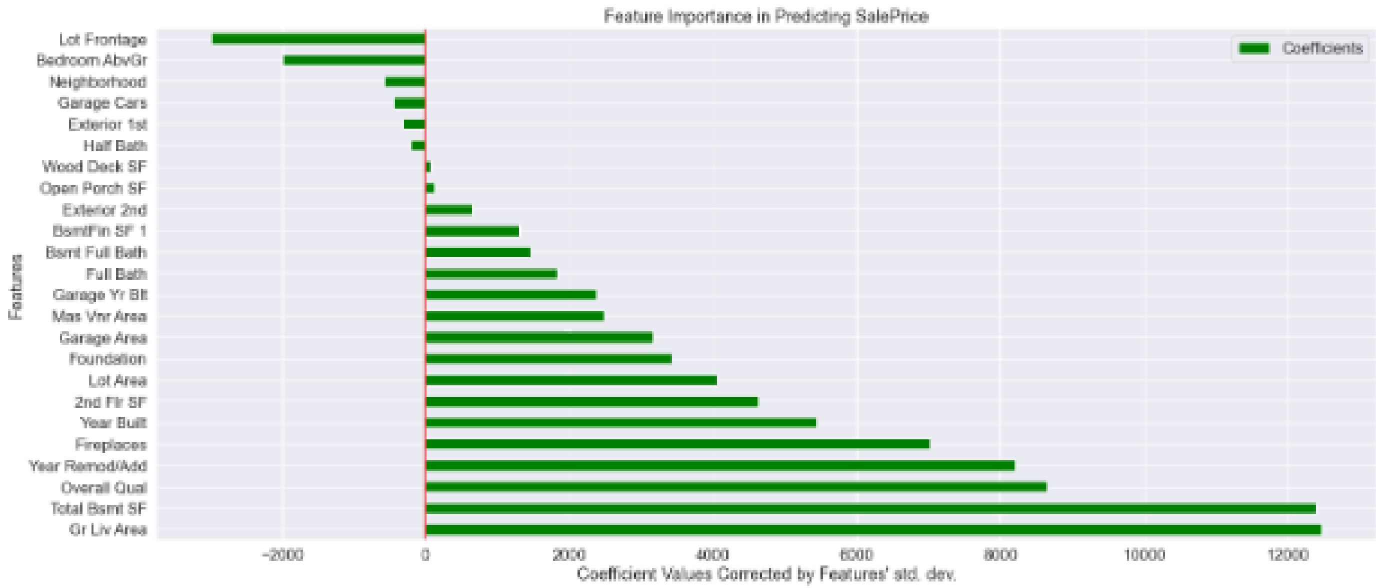
2b) LR Model 2 WITH Optimal Hyperparameter Tuning: A modified algorithm with auto hyper-tuned parameters was created using random search method described above.

Model 4: LASSO With `Optimal Hypertuned Parameters`

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model4, res4, features4, pred4 = build_op_l(X_train, y_train, X_test, best_params=optimal_params_l) # Call Method to 'Build

#Collect & Append Results
r4 = np.array2string(res4)####
r4 = pd.DataFrame([r4.split(';') for x in r4.split('\n')])
r4 = r4.rename(columns={0: 'SCORE'})
r4['MODEL'] = 'Lasso Optimal'
r4['MODEL#'] = 'Model 4'
r4
```

```
Model Directory:
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__len__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__sklearn_is_fitted__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_can_inverse_transform', '_can_transform', '_check_feature_names', '_check_fit_params', '_check_n_features', '_estimator_type', '_final_estimator', '_fit', '_get_param_names', '_get_params', '_get_tags', '_iter', '_log_message', '_more_tags', '_replace_estimator', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_required_parameters', '_set_params', '_sk_visual_block_', '_validate_data', '_validate_names', '_validate_steps', 'classes_', 'decision_function', 'feature_names_in_', 'fit', 'fit_predict', 'fit_transform', 'get_feature_names_out', 'get_params', 'inverse_transform', 'memory', 'n_features_in_', 'named_steps', 'predict', 'predict_log_proba', 'predict_proba', 'score', 'score_samples', 'set_params', 'steps', 'transform', 'verbose']
```

	SCORE	MODEL	MODEL#
0	-1.3033701e+09	Lasso Optimal	Model 4



3) Elastic-Net (EN) Models

A single method was employed to run elastic-net models with and without optimal hyper-parameter tuning:

Method to 'Build ELASTIC-NET Regression WITH & Without Optimal Hyperparameters'

```
# Build a LASSO model with Optimal Class Weights
def build_op_en(X_train, y_train, X_test, threshold=0.5, best_params=None):

    model = ElasticNet(random_state=rs, max_iter = 1000)

    # If best parameters are provided
    if best_params:
        model = make_pipeline(RobustScaler(),
                              ElasticNet(alpha = best_params_en['alpha'].iloc[0],
                                    l1_ratio = best_params_en['l1_ratio'].iloc[0],
                                    fit_intercept = best_params_en['fit_intercept'].iloc[0],
                                    normalize = best_params_en['normalize'].iloc[0],
                                    precompute = best_params_en['precompute'].iloc[0],
                                    copy_X = best_params_en['copy_X'].iloc[0],
                                    tol = best_params_en['tol'].iloc[0],
                                    warm_start = best_params_en['warm_start'].iloc[0],
                                    positive = best_params_en['positive'].iloc[0],
                                    selection = best_params_en['selection'].iloc[0]
                                          )
                              )

        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model[:-1].get_feature_names_out() #https://scikit-learn.org/stable/auto_examples/inspection/plot_li
        X_train_preprocessed = pd.DataFrame(model[:-1].transform(X_train), columns=feature_names) # X_train to compute stanc
        features = pd.DataFrame(model[-1].coef_ * X_train_preprocessed.std(axis=0), columns=["Coefficients"], index=feature_

    else:
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model._check_feature_names
        X_train_preprocessed = X_train.copy() #.T # Transpose X_train to compute standard deviation of the related feature
        # Multiply model coefficients by std of features to reduce all coefficients to same unit of measure.
        features  = pd.DataFrame(model.coef_ * X_train_preprocessed.std(axis=0), columns=['Coefficients'], index=X_train.col

    features['Features'] = features.index
    #features = features.loc[features['Coefficients'] > 0] # Uncomment if only features with Coeff > 0 are required
    features  = features.sort_values(by=['Coefficients'], ascending=False).reset_index(drop=True)

    # Plot Features
    data = features.copy()
    data = data[data["Coefficients"] != 0]
    data.set_index('Features', inplace=True)
    data.plot.barh(figsize=(30,10), color='green')

    title = "Feature Importance in Predicting " + target
    plt.title(title)
    plt.axvline(x=0, color="red")
    plt.xlabel("Coefficient Values Corrected by Features' std. dev.")
    plt.subplots_adjust(left=0.3)

    # Get Model Results
    res = result(model)

    return model, res, features, pred
```

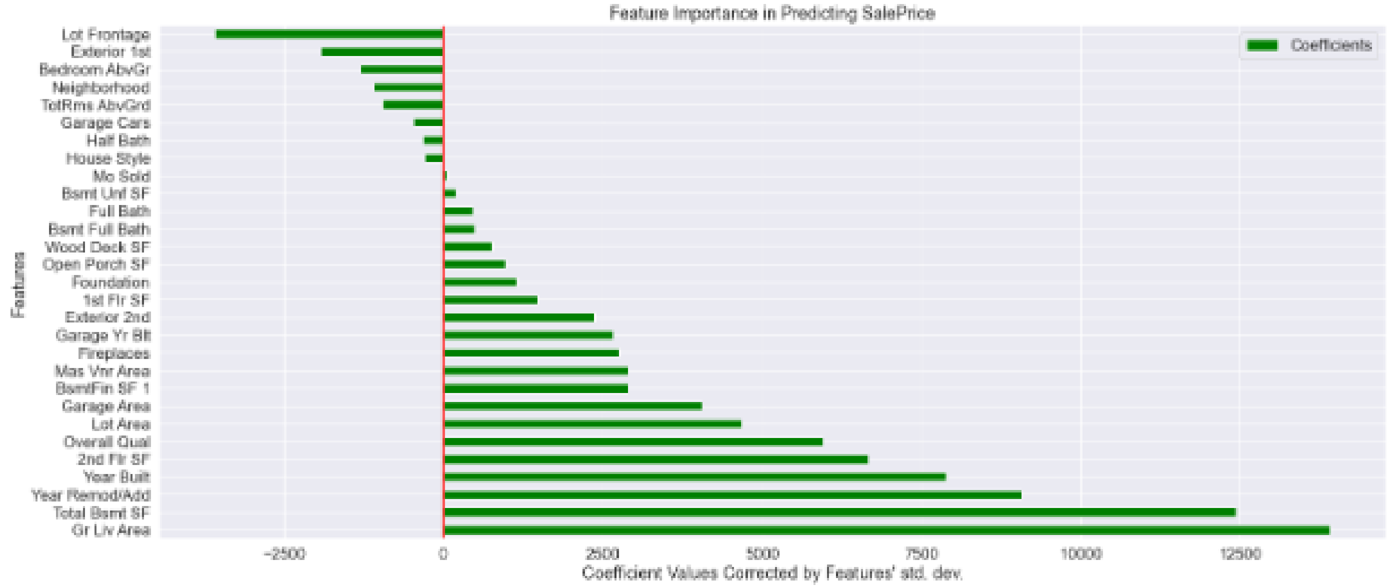
3a) EN Model 1 Without Optimal Hyperparameter Tuning: A simple algorithm was created without any hyperparameter tuning:

Model 5: ELASTIC-NET Without 'Optimal Hypertuned Parameters'

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model5, res5, features5, pred5 = build_op_en(X_train, y_train, X_test, best_params=None) # Call Method to 'Build Logistic Re

#Collect & Append Results
r5 = np.array2string(res5)####
r5 = pd.DataFrame([r5.split(';') for x in r5.split('\n')])
r5 = r5.rename(columns={0: 'SCORE'})
r5['MODEL'] = 'Elastic-Net'
r5['MODEL#'] = 'Model 5'
r5
```

```
Model Directory:
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_check_feature_names', '_check_n_features', '_decision_function', '_estimator_type', '_get_param_names', '_get_tags', '_more_tags', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_set_intercept', '_validate_data', 'alpha', 'coef_', 'copy_X', 'dual_gap_', 'feature_names_in_', 'fit', 'fit_intercept', 'get_params', 'intercept_', 'l1_ratio', 'max_iter', 'n_features_in_', 'n_iter_', 'normalize', 'path', 'positive', 'precompute', 'predict', 'random_state', 'score', 'selection', 'set_params', 'sparse_coef_', 'tol', 'warm_start']
```

	SCORE	MODEL	MODEL#
0	-1.32680508e+09	Elastic-Net	Model 5



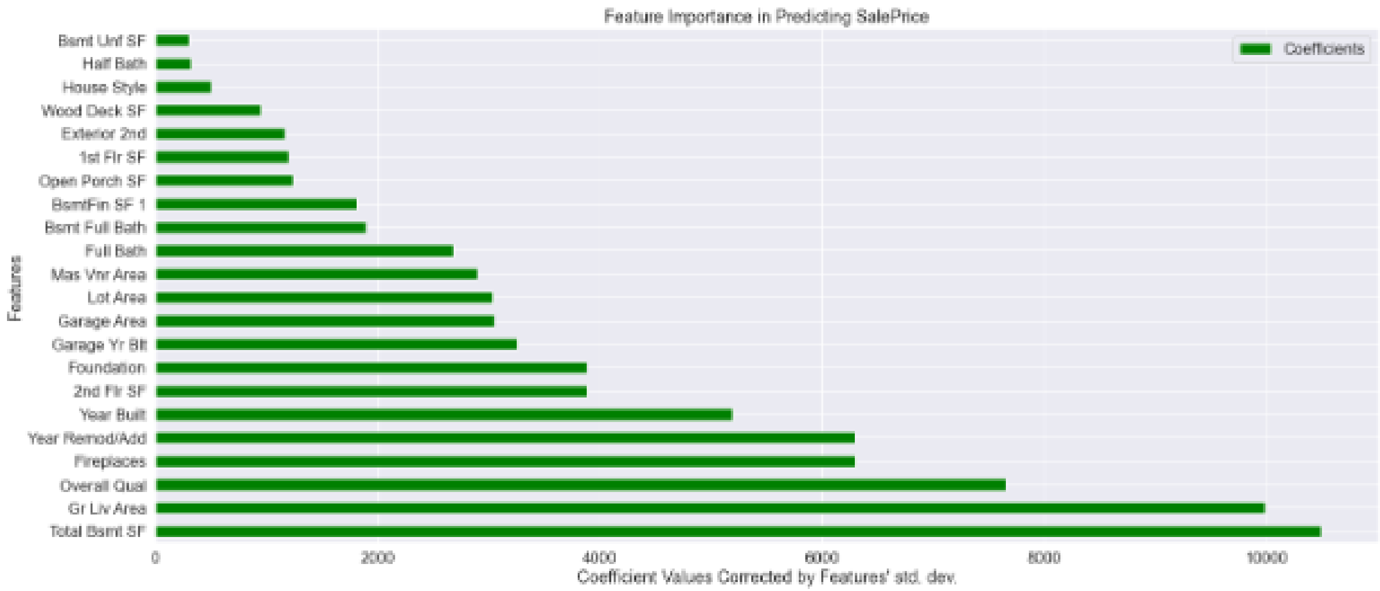
3b) EN Model 2 WITH Optimal Hyperparameter Tuning: A modified algorithm with auto hyper-tuned parameters was created using random search method described above:

Model 6: ELASTIC-NET With 'Optimal Hypertuned Parameters'

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model6, res6, features6, pred6 = build_op_en(X_train, y_train, X_test, best_params=optimal_params_en) # Call Method to 'Buil

#Collect & Append Results
r6 = np.array2string(res6)####
r6 = pd.DataFrame([r6.split(';') for x in r6.split('\n')])
r6 = r6.rename(columns={0: 'SCORE'})
r6['MODEL'] = 'Elastic-Net Optimal'
r6['MODEL#'] = 'Model 6'
r6
```

```
Model Directory:
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__len__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__sklearn_is_fitted__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_can_inverse_transform', '_can_transform', '_check_feature_names', '_check_fit_params', '_check_n_features', '_estimator_type', '_final_estimator', '_fit', '_get_param_names', '_get_params', '_get_tags', '_iter', '_log_message', '_more_tags', '_replace_estimator', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_required_parameters', '_set_params', '_sk_visual_block_', '_validate_data', '_validate_names', '_validate_steps', 'classes_', 'decision_function', 'feature_names_in_', 'fit', 'fit_predict', 'fit_transform', 'get_feature_names_out', 'get_params', 'inverse_transform', 'memory', 'n_features_in_', 'named_steps', 'predict', 'predict_log_proba', 'predict_proba', 'score', 'score_samples', 'set_params', 'steps', 'transform', 'verbose']
```

	SCORE	MODEL	MODEL#
0	-1.32126229e+09	Elastic-Net Optimal	Model 6



4) Extreme Gradient Boosting (XGB) Regression Models

A single method was employed to run XGB regression models with and without optimal hyper-parameter tuning:

Method to 'Build XGB Regression WITH & Without Optimal Hyperparameters'

```
# Build a LASSO model with Optimal Class Weights
def build_op_xgb(X_train, y_train, X_test, threshold=0.5, best_params=None):

    model = GradientBoostingRegressor(random_state=rs)

    # If best parameters are provided
    if best_params:
        model = make_pipeline(RobustScaler(),
                              GradientBoostingRegressor(learning_rate = best_params_xgb['learning_rate'].iloc[0],
                                  subsample = best_params_xgb['subsample'].iloc[0],
                                  criterion = best_params_xgb['criterion'].iloc[0],
                                  max_features = best_params_xgb['max_features'].iloc[0]
                                      )
                             )
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        feature_names = model[:-1].get_feature_names_out() #https://scikit-learn.org/stable/auto_examples/inspection/plot_li
        X_train_preprocessed = pd.DataFrame(model[:-1].transform(X_train), columns=feature_names) # X_train to compute stand
        features = pd.DataFrame(model[-1].feature_importances_ * X_train_preprocessed.std(axis=0), columns=["Coefficients"],

    else:
        model.fit(X_train, y_train)

        # Get Prediction
        pred = model.predict(X_test)
        #pred = pred * (y_train.std() + y_train.mean())
        test_ids = X_test.index
        pred = pd.DataFrame({'Id': test_ids, 'Predicted Price': pred})

        # Get Model Directory
        print(colored("Model Directory: \n", 'cyan', attrs=['bold'])
              + colored(dir(model), 'magenta', attrs=['bold']))

        # Get Model Features
        X_train_preprocessed = X_train.copy()
        features  = pd.DataFrame(model.feature_importances_ * X_train_preprocessed.std(axis=0), columns=['Coefficients'], in

    features['Features'] = features.index
    #features = features.loc[features['Coefficients'] > 0] # Uncomment if only features with Coeff > 0 are required
    features  = features.sort_values(by=['Coefficients'] , ascending=False).reset_index(drop=True)

    # Plot Features
    data = features.copy()
    data = data[data["Coefficients"] != 0]
    data.set_index('Features', inplace=True)
    data.plot.barh(figsize=(30,10), color='green')

    title = "Feature Importance in Predicting " + target
    plt.title(title)
    plt.axvline(x=0, color="red")
    plt.xlabel("Coefficient Values Corrected by Features' std. dev.")
    plt.subplots_adjust(left=0.3)

    # Get Model Results
    res = result(model)

    return model, res, features, pred
```

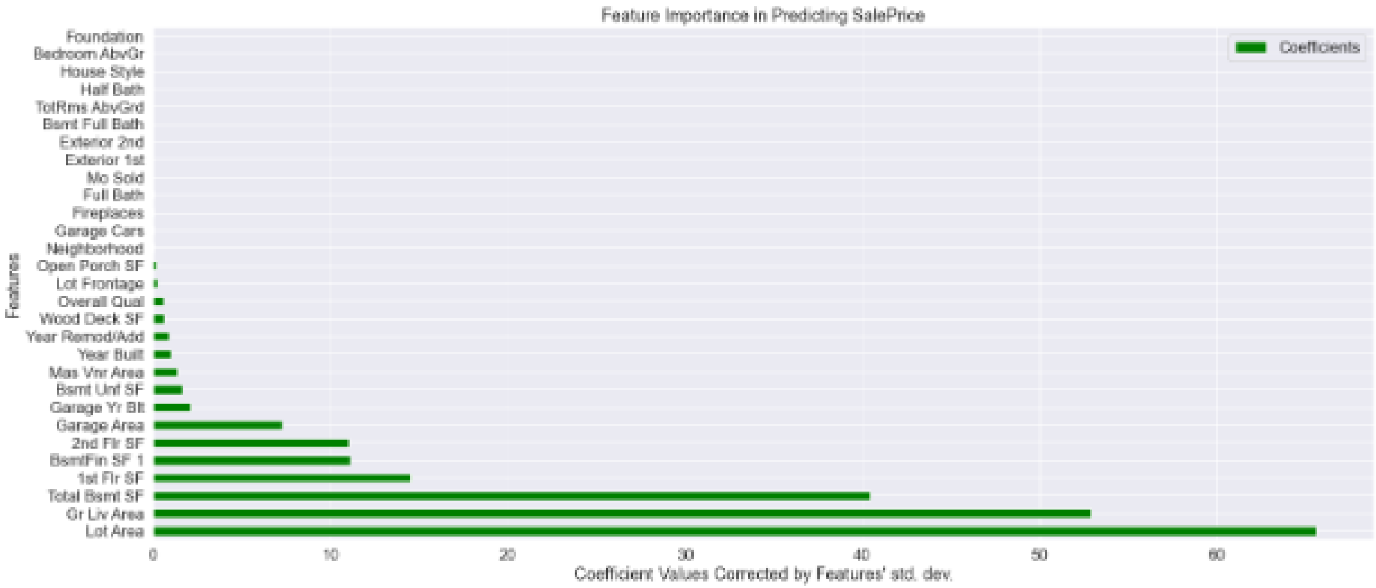
4a) XGB Regression Model 1 Without Optimal Hyperparameter Tuning: A simple algorithm was created without any hyperparameter tuning:

Model 7: XGB Without 'Optimal Hypertuned Parameters'

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model7, res7, features7, pred7 = build_op_xgb(X_train, y_train, X_test, best_params=None) # Call Method to 'Build XGB Withou

#Collect & Append Results
r7 = np.array2string(res7)####
r7 = pd.DataFrame([r7.split(';') for x in r7.split('\n')])
r7 = r7.rename(columns={0: 'SCORE'})
r7['MODEL'] = 'XGB'
r7['MODEL#'] = 'Model 7'
r7
```

```
Model Directory:
['_SUPPORTED_LOSS', '__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_check_feature_names', '_check_initialized', '_check_n_features', '_check_params', '_clear_state', '_compute_partial_dependence_recursion', '_estimator_type', '_fit_stage', '_fit_stages', '_get_param_names', '_get_tags', '_init_state', '_is_initialized', '_loss', '_make_estimator', '_more_tags', '_raw_predict', '_raw_predict_init', '_repr_html_', '_repr_html_inner', '_repr_mimebundle_', '_required_parameters', '_resize_state', '_rng', '_staged_raw_predict', '_validate_data', '_validate_estimator', '_validate_y', 'alpha', 'apply', 'ccp_alpha', 'criterion', 'estimators_', 'feature_importances_', 'feature_names_in_', 'fit', 'get_params', 'init', 'init_', 'learning_rate', 'loss', 'loss_', 'max_depth', 'max_features', 'max_features_', 'max_leaf_nodes', 'min_impurity_decrease', 'min_samples_leaf', 'min_samples_split', 'min_weight_fraction_leaf', 'n_estimators', 'n_estimators_', 'n_features_', 'n_features_in_', 'n_iter_no_change', 'predict', 'random_state', 'score', 'set_params', 'staged_predict', 'subsample', 'tol', 'train_score_', 'validation_fraction', 'verbose', 'warm_start']
```

	SCORE	MODEL	MODEL#
0	-9.9099029e+08	XGB	Model 7



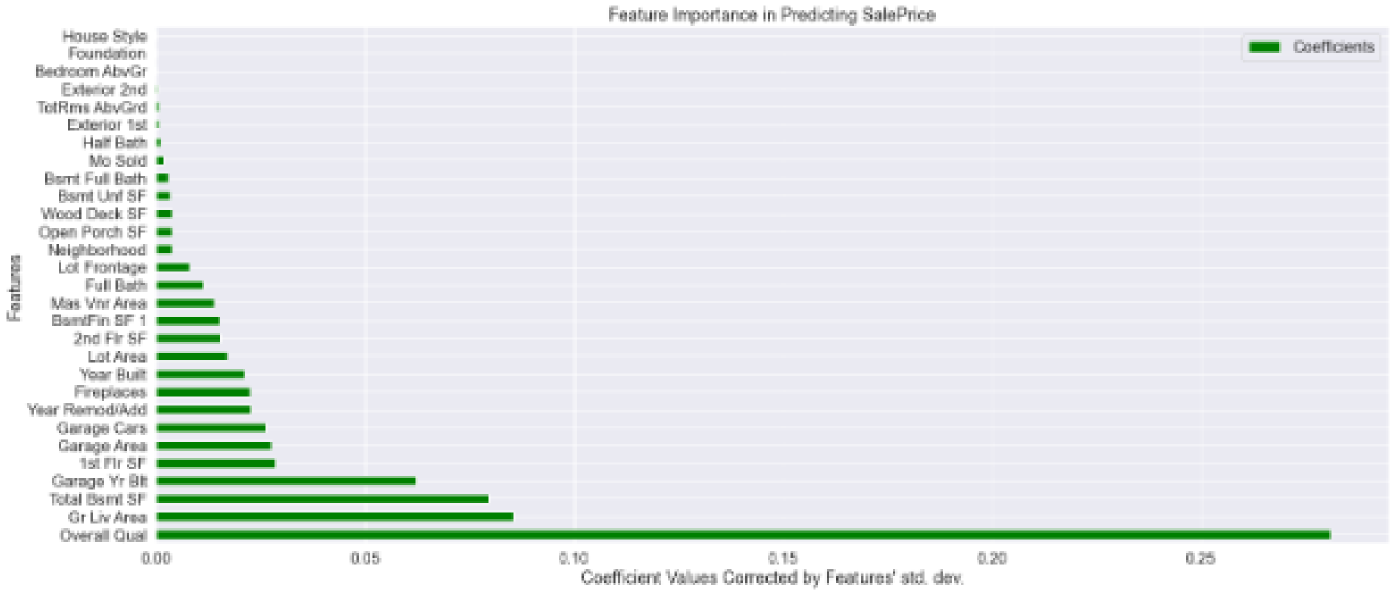
4b) XGB Regression Model 2 WITH Optimal Hyperparameter Tuning: A modified algorithm with auto hyper-tuned parameters was created using random search method described above:

Model 8: XGB With `Optimal Hypertuned Parameters`

```
#Running the Model...Call Method to 'Build LR Without Adjusted Class Weights'
model8, res8, features8, pred8 = build_op_xgb(X_train, y_train, X_test, best_params=optimal_params_xgb) # Call Method to 'Bu

#Collect & Append Results
r8 = np.array2string(res8)####
r8 = pd.DataFrame([r8.split(';') for x in r8.split('\n')])
r8 = r8.rename(columns={0: 'SCORE'})
r8['MODEL'] = 'XGB Optimal'
r8['MODEL#'] = 'Model 8'
r8
```

```
Model Directory:
['__abstractmethods__', '__annotations__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le_
_', '__len__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setsta
te__', '__sizeof__', '__sklearn_is_fitted__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', '_can_inverse_transfor
m', '_can_transform', '_check_feature_names', '_check_fit_params', '_check_n_features', '_estimator_type', '_final_estimator',
'_fit', '_get_param_names', '_get_params', '_get_tags', '_iter', '_log_message', '_more_tags', '_replace_estimator', '_repr_htm
l_', '_repr_html_inner', '_repr_mimebundle_', '_required_parameters', '_set_params', '_sk_visual_block_', '_validate_data', '_v
alidate_names', '_validate_steps', 'classes_', 'decision_function', 'feature_names_in_', 'fit', 'fit_predict', 'fit_transform',
'get_feature_names_out', 'get_params', 'inverse_transform', 'memory', 'n_features_in_', 'named_steps', 'predict', 'predict_log_
proba', 'predict_proba', 'score', 'score_samples', 'set_params', 'steps', 'transform', 'verbose']
```

	SCORE	MODEL	MODEL#
0	-9.92038691e+08	XGB Optimal	Model 8



6) Key Findings to the Main Objectives of Analysis

6a) Result Summary

```
Result Summary is given below:
           SCORE                   MODEL   MODEL#
0    -1.3033701e+09        Lasso Optimal  Model 4
0   -1.30393778e+09   Regression Optimal  Model 2
0   -1.30784598e+09           Regression  Model 1
0   -1.30784598e+09                Lasso  Model 3
0   -1.32126229e+09  Elastic-Net Optimal  Model 6
0   -1.32680508e+09          Elastic-Net  Model 5
0    -9.9099029e+08                  XGB  Model 7
0   -9.92038691e+08          XGB Optimal  Model 8
```

6b) Recommended Model and Justification

```
Recommended Model and Justification:

Model Scoring has been carried out using mean squared error (MSE).
MSE shows closeness of a regression line to a set of points and helps in finding average of a set of errors.
It takes the distances (or errors) from the points to the regression line and squares them to remove any negative signs.
Since it lends more weight to larger differences, hence, the lower the MSE, the better the forecast.
In other words, the smaller the MSE, the closer the model is to the line of best fit.
A Score of Zero would mean the model is perfect.

Therefore, when scoring a regression model with MSE, a minimal score would imply a better prediction
Hence, we will select the model with Minimal Score.

In this case, the Model Lasso Optimal is yielding Minimal Score of -1.3033701e+09.

Hence, we will select this model for Ames, Iowa House Price Prediction.
```

6c) Summarizing Model Drivers

Main Drivers behind top performing Lasso Optimal model are as follows:

- Uses random search method to find optimal parameters to achieve best hyper-tuning
- Runs lasso regression model with above drivers to get top contributory features

6d) Enlisting Top Contributory Factors

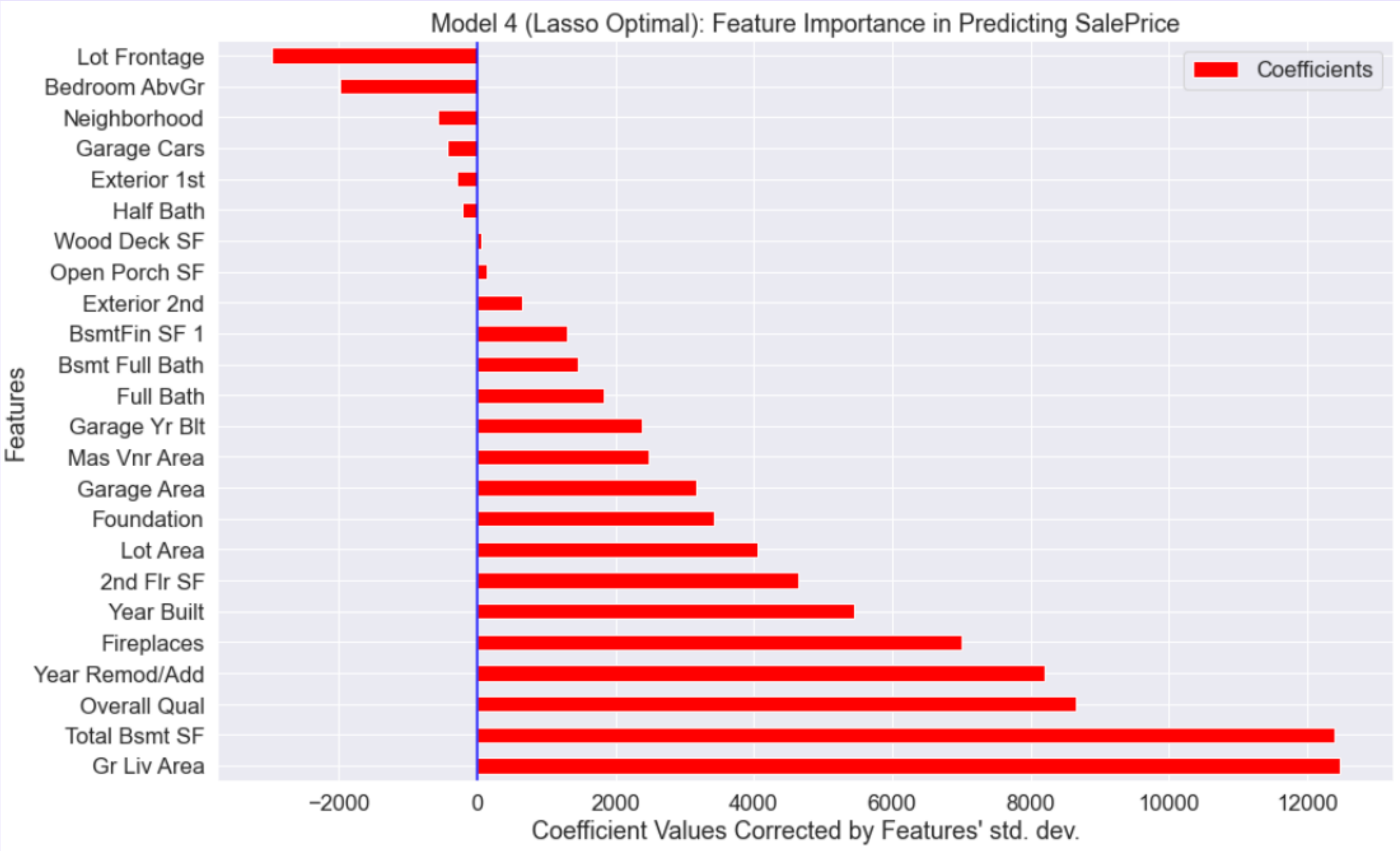
Top Factors Contributing to features driving sale value are cited under in ascending order of importance:

Contributory Features driving Sales Price:

Features Summary with Coefficient Values is given below:

	Features	Coefficients
0	Gr Liv Area	12471
1	Total Bsmt SF	12391
2	Overall Qual	8656
3	Year Remod/Add	8204
4	Fireplaces	7011
5	Year Built	5441
6	2nd Flr SF	4637
7	Lot Area	4052
8	Foundation	3413
9	Garage Area	3170
10	Mas Vnr Area	2474
11	Garage Yr Blt	2374
12	Full Bath	1830
13	Bsmt Full Bath	1448
14	BsmtFin SF 1	1302
15	Exterior 2nd	656
16	Open Porch SF	132
17	Wood Deck SF	67
39	Half Bath	-207
40	Exterior 1st	-301
41	Garage Cars	-435
42	Neighborhood	-574
43	Bedroom AbvGr	-1988
44	Lot Frontage	-2978

6e) Visualizing Top Contributory Factors Driving Sale Value



6f) Sale Price Prediction on Test/New Data

Predicted sale price for test data were extracted from top performing model as follows:

Predict Sale Price on Ames Homes' Original Test Data

```python
# Get Target Variable Predictions
if model == 'Model 1':#
    prediction  = pred1.copy()
elif model == 'Model 2':
    prediction  = pred2.copy()
elif model == 'Model 3':
    prediction  = pred3.copy()
elif model == 'Model 4':
    prediction  = pred4.copy()
elif model == 'Model 5':#
    prediction  = pred5.copy()
elif model == 'Model 6':
    prediction  = pred6.copy()
elif model == 'Model 7':
    prediction  = pred7.copy()
elif model == 'Model 8':
    prediction  = pred8.copy()
else:
    print("")

# Attach Predictions to Test Data
test_data = raw_data.copy()
#cols = ['Id']+test_data.columns.tolist()+['Predicted Price'] # Uncomment if all original columns are desired
cols = ['Id'] + final_list + [target] + [predicted_var] # Comment if previous line has been Uncommented
test_data['Id'] = test_data.index
test_data = test_data.merge(prediction, on='Id', how='left').sort_values(by=['Id'], ascending=True).fillna(0)
test_data = test_data[test_data[predicted_var] != 0]
test_data = test_data[cols]
test_data[predicted_var] = test_data[predicted_var].astype(int)
test_data['Price Difference'] = test_data[predicted_var] - test_data[target]
test_data
```

	Id	Gr Liv Area	Total Bsmt SF	Overall Qual	Year Remod/Add	Fireplaces	Year Built	2nd Flr SF	Lot Area	Foundation	...	Wood Deck SF	Half Bath	Exterior 1st	Garage Cars	Neighborhood	Bedroom AbvGr	Fronta
0	0	1656	1080.0	6	1960	2	1960	0	31770	CBlock	...	210	0	BrkFace	2.0	NAmes	3	14
4	4	1629	928.0	5	1998	1	1997	701	13830	PConc	...	212	1	VinylSd	2.0	Gilbert	3	7
6	6	1338	1338.0	8	2001	0	2001	0	4920	PConc	...	0	0	CemntBd	2.0	StoneBr	2	4
9	9	1804	994.0	7	1999	1	1999	776	7500	PConc	...	140	1	VinylSd	2.0	Gilbert	3	6
15	15	3279	1650.0	8	2003	1	2003	1589	53504	PConc	...	503	1	CemntBd	3.0	StoneBr	4	4

These findings were compiled and subsequently exported in the form of a well formatted coloured excel report.

7) Future Directions

7a) Possible Flaws in Chosen Model

- The model uses Mean Squared Error (MSE) as scoring method which may be highly biased for higher values.
- Just like "delete_features" list, the model needs to incorporate user input list to keep features deemed necessary so that they are not automatically dropped

Outline Editable Variables

```
In [1]:
output_file_path = "C:/Users/fatima.s/Downloads/Ames Housing Price Prediction.xlsx"
#save_plot = "C:/Users/fatima.s/Downloads/Features.png"
save_plot = "Features.png"
delete_features = ['Roof Matl', 'Alley'] # List Specific Features you want to retain for automated drops
```

7b) Recommendations

Following suggestions are likely to improve the model even further:

- Incorporate "keep_features" list and use it to eliminate these from final "drop" list in row 16 of project notebook

```
drop = drop1 + drop2 + drop3 + delete_features # Final List of columns to drop
```

- Introduce more models like Decision Trees and Random Forest
- Implement a method to combine best performing models to ensure enhanced performance and more effective generalization, (see, for example, 8) Useful Links, 8a-a)
- Apply other deep learning models like TensorFlow
- Because of biasness of MSE towards higher values, scoring may be substituted by Root Mean Squared Error (RMSE) which may reflect model performance whilst dealing with increased error values.
- Apply plot to depict both MSE and RMSE (see, Machines, 2022)

8) Useful Links

8a) Link to Other Useful Models

a) https://www.kaggle.com/code/mgmarques/houses-prices-complete-solution
b) https://www.kaggle.com/code/marto24/beginners-prediction-top3
c) https://www.kaggle.com/code/mchatham/ames-housing-regression
d) https://www.kaggle.com/code/mkariithi/real-estate-sales-price-prediction/notebook
e) https://www.kaggle.com/code/bashkeel/eda-to-ensemble-model-lasso-ridge-xgboost
f) https://www.kaggle.com/code/gerlandore/advanced-house-regression-eda-model-comparison
g) https://www.kaggle.com/code/prasadperera/the-boston-housing-dataset/notebook
h) https://www.kaggle.com/search?q=ADVANCED+LINEAR+REGRESSION+BOSTON+HOUSE+PREDICTION
i) https://www.kaggle.com/code/koki25ando/nba-salary-prediction-using-multiple-regression

8b) Github Link to Assignment Notebook and Other Files

https://github.com/FATIMASP/IBM-MACHINE-LEARNING-CERTIFICATION/tree/main/Supervised%20Machine%20Learning:%20Regression

References

Herath, S. & Maier, G., 2010. *The hedonic price method in real estate and housing market research: a review of the literature, (pp. 1-21),* Vienna, Austria: University of Economics and Business: Institute for Regional Development and Environment,

Machines, I. L., 2022. *Mean Squared Error.* [Online]
Available at: https://insidelearningmachines.com/mean_squared_error/
[Accessed 24 09 2022].