Capstone 2 Project Report

Predicting Sale Price for Real Estate Property in Maricopa County, Arizona

06/03/2024 By Sebastian Ivan

Problem Statement

1. Context: Maricopa County in Arizona is one of the hottest real estate markets in the country. The sales volume for real estate assets is substantial. Net migration in Arizona for 2023 was 77,759 more people coming in than going. Housing inventory was not able to keep up with increasing housing demand and house pricing for first time buyers, and resellers alike, became a daunting task. House price evaluation and actual selling price have seen large volatility over short period of times. Traditionally, an appraiser is allowed to consider sales up to 6 months old to valuate a house. If nearby properties are not available, due to low inventory, appraiser is allowed to extend the area to look for comps. The valuation or appraised price differs from the actual sold price more often than not, causing subsequent buyers and sellers to have distorted and divergent views of fair market value. This model aids consumers to gauge their expectations, predicting a sale price with a margin error of up to 1% of actual sold price.

2. Criteria for success: To predict a sale price that within 1% of the actual sold price. The statistical model uses data from sold MLS listings and should predict a future sale price when a house valuation (based on past sales, hence, fair market value) and zip code is provided.

3. Scope of Solution Space: The model can be used by any stake holder in real estate market, as a tool to reduce risk of real estate investment.

4. Constraints: The data used in this analysis was obtained from MLS (Multiple Listings Service) only. Properties listed 'For Sale by Owners', or private sales outside MLS system were not accessible. The model is built on limited data spanning from February 01 – March 01, 2024. Results may vary when new data is fed into the model. Market distortions may affect prediction accuracy, also. Fire/distress sales may contribute to outliers causing predictions above or below the 1% prediction accuracy.

5. Stakeholders: Real estate agents, appraisers, lenders, housing authorities, institutional and private buyers and sellers.

6. Method: The model is using MLS data readily available to real estate agents. The Maricopa Housing Project intends to aid consumers to estimate the sale price for a house valued at a specific value and sold for same value, in this case $600,000. The model can be used to obtain an estimated sale price for different valuation values for different zip codes. In this example I used 85202 zip code and a house valued and sold for $600,000. Three models were used to predict the sold price: Linear Regression, Extreme Gradient Booster and Random Forest classification. The last model, Random Forest, predicted a sale price of $594,151 for a house valued and actually sold for $600,000 in 85202 zip code. This translates to a sold price less by 0.97% from actual sold value, which meets the criteria for success of within the 1 % margin of error.

Data Collection

Data set was obtained from Arizona MLS data base. Data includes all real estate properties sold between February 01 and March 01, 2024 in Maricopa county, Arizona. There were 3117 properties sold over the time period considered. There were 29 features included in the analysis, such as number of bedrooms, bathrooms and so on.

	0	1	2	
Zip_Code	85003	85003	85003	85
Dwelling_Type	AF	LS	SF	
Nr_Bedrooms	1	1	4	
Nr_Bathrooms	1.0	1.0	3.0	
Approx_SQFT	720	755	3,415	
Price_per_SqFt	340.28	430.46	439.23	43
Property_Type	Residential	Residential	Residential	Reside
Dwelling_Styles	Stacked	Stacked	Detached	Sta
Year_Built	1964	2005	1940	2
Approx_Lot_SqFt	652	751	11,696	
Pool	Community	Private	NaN	Commu
HOA_Fee	567.0	631.0	555.63	67
Land_Lease_Fee	N	N	N	
Clubhouse_Rec_Room	Yes	Yes	Yes	
Basement	BASEMENT Y/N: N	BASEMENT Y/N: N	BASEMENT Y/N: Y	BASEM Y/
RV_Gate	Yes	Yes	Yes	
List_Price	247,000	335,000	1,595,000	315
Sold_Price	245,000	325,000	1,500,000	310
Building_Style	High Rise	High Rise	High Rise	High
Gated_Community	Yes	Yes	Yes	
Workout_Facility	Yes	Yes	Yes	
Garage_Spaces	0.0	0.0	2.0	
Carport_Spaces	1.0	0.0	0.0	
Loan_Type	Conventional	VA	Conventional	Conventi
Payment_Type	Fixed	Fixed	Fixed	Adjust
Buyer_Concession_to_Seller	0.0	0.0	0.0	
Buyer_Concession_type	$	%	$	
Seller_Concession_to_Buyer	3.0	0.0	0.0	60
Seller_Concession_type	%	$	$	

29 rows × 3117 columns

Data Cleaning

- The ‘Approx_SqFt’, ‘Price_per_SqFt’, ‘Approx_Lot_SqFt’, ‘Land_Lease_Fee’, ‘Sold_Price’ and ‘List_Price were changed from object to numerical type (float or integer). Commas had to be deleted from the imported format before object type was changed.

- Property_Type feature was deleted, the only value contained was ‘Residential’.

- The ‘nan’ value in Pool feature was changed to ‘no_pool’.

- There were 917 properties out of 3117 with no HOA fees, storing NaN values in place. I created a new binary (0’s and 1’s) feature in place HOA_Missing with zero values when value in HOA_Fee was greater than zero, and zero when HOA_Fee value was NaN.

- The Basement feature contained two values with unusable characters: 'BASEMENT Y/N:\xa0N', 'BASEMENT Y/N:\xa0Y' accounting for a binary Yes or No. I changed the values in place, to show only Y or N values.

- Values in ‘Workout_Facility’, ‘Gated_Community’, ‘RV_Gate’, Clubhouse_Rec_Room’ and ‘Building_Style’ features were changed from ‘nan’ to “No”.

- There were 950 missing data in ‘Payment_Type’ feature. I replaced them with in place ‘Missing.

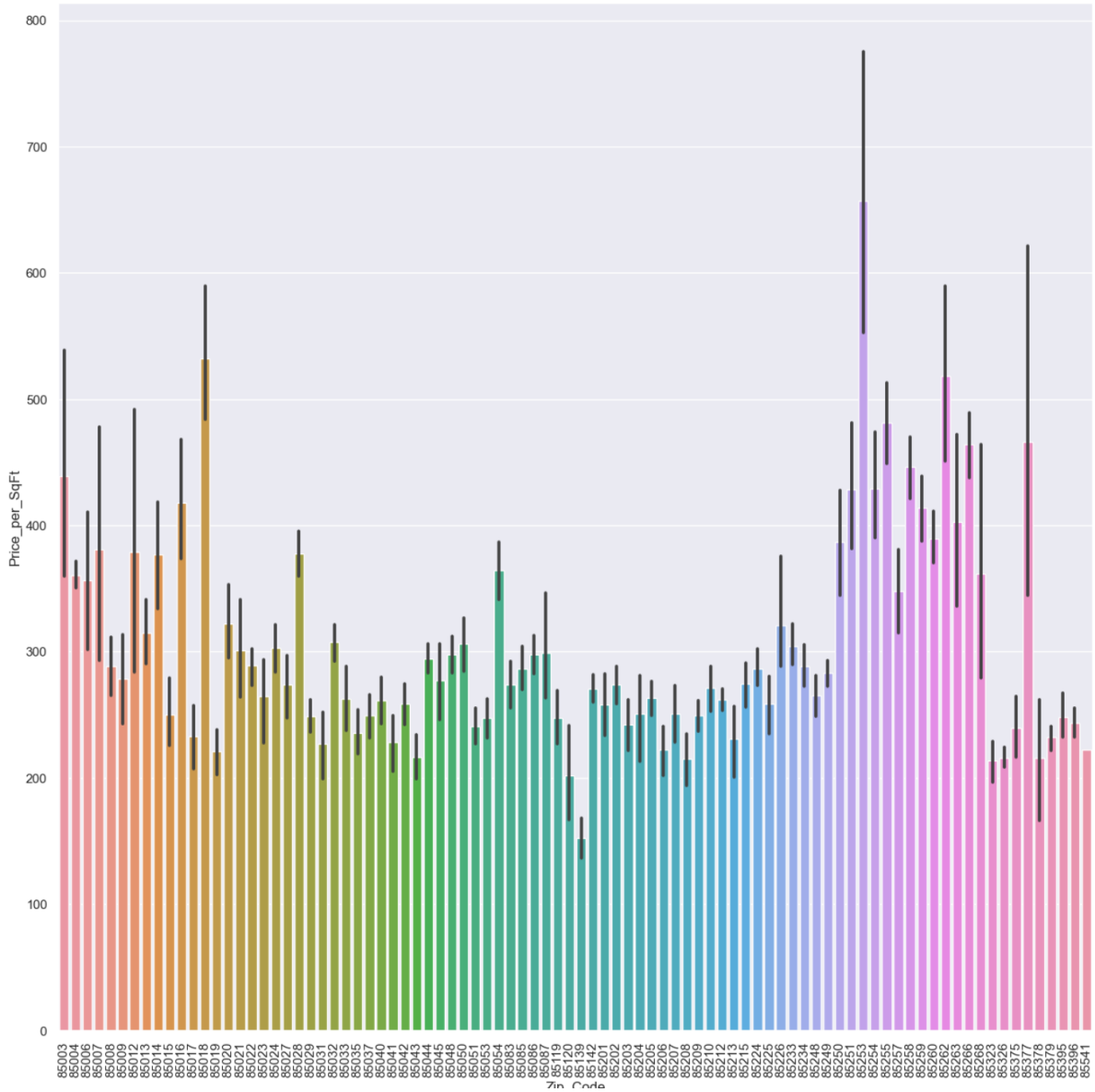
- ‘Buyer and Seller concession type and amount was changed to reflect only a dollar amount. Before that I applied a filter to find errors where an agent would type $ instead of %, or viceversa. I found 42 entries where agents wrongly typed % and $, which would have drastically distort the values, if they weren’t found. For example a $3000 concession is very different compared to a 3,000% concession.

- There another two similar errors, for example one of the entry showed a Price_per_SqFt of $175,000, Turns out the agents for that specific listing typed value of 1 in Approx_Sqft column, thus resulting in the Price_per_Sqft anomaly. This value would have skewed and distorted the data if not corrected.

Exploratory Data Analysis

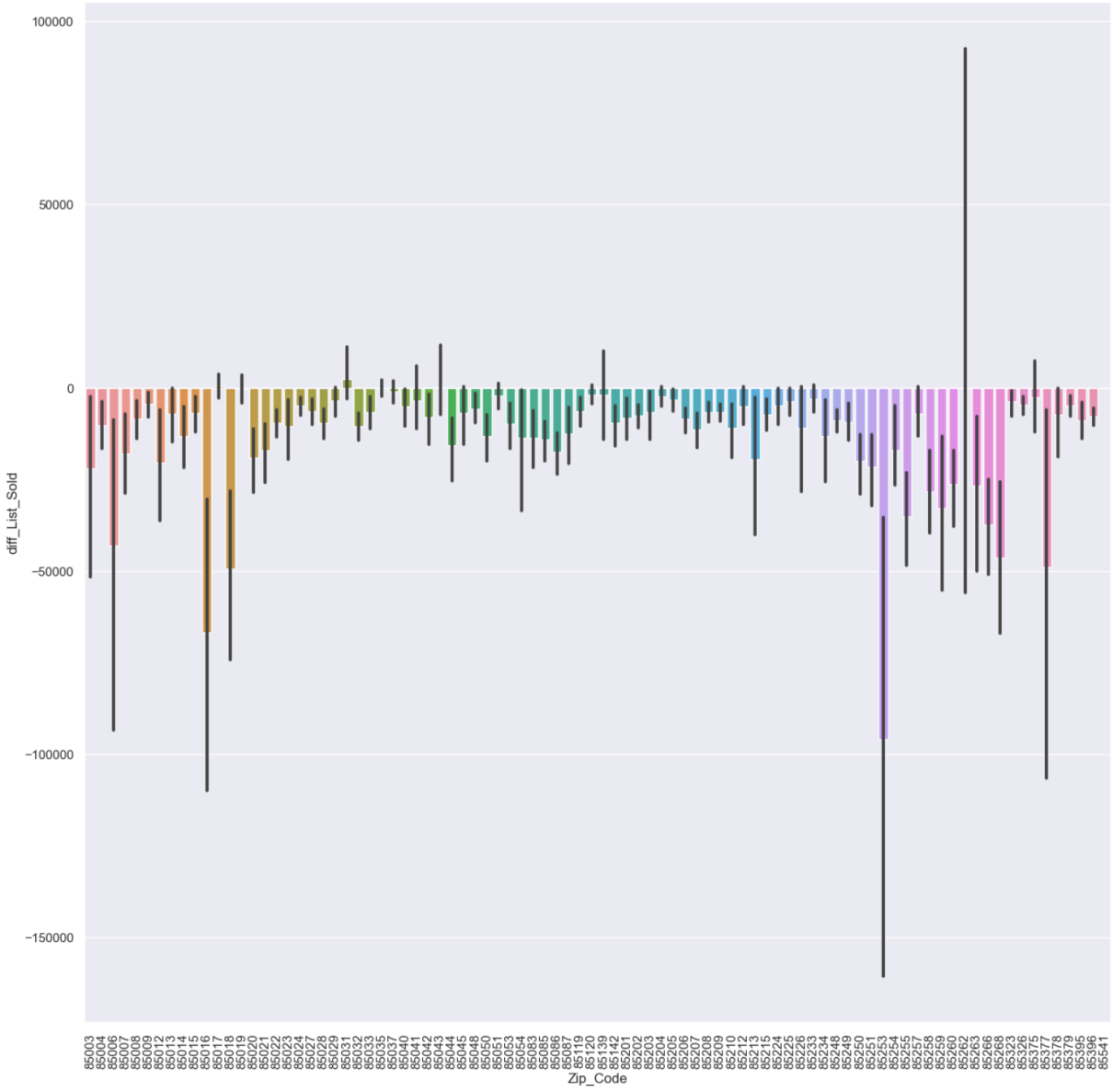
Most 5 expensive houses were sold in Zip codes 85018, 85253, 85016 and 85377, corresponding to Phoenix, Paradise Valley, and Carefree cities.

	Zip_Code	Sold_Price
242	85018	10364000
2332	85253	9391112
148	85016	7626000
2946	85377	6295000
219	85018	5800000



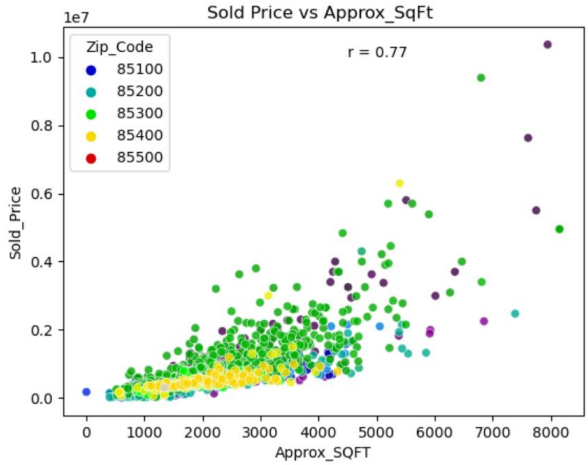
Majority of houses sold for less than the listing price, as reflected by diff_List_Sold feature which shows the difference between listing and sold price.

Note: The best model in this project also predicted a sold value of $594,151 vs $600,000 (less than Listing price) which is consistent with data visualized below.

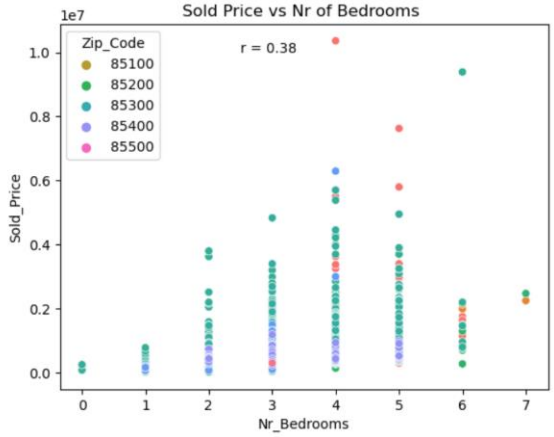


The Sold Price is positively correlated with price per Square Foot, with a **0.76** correlation coefficient

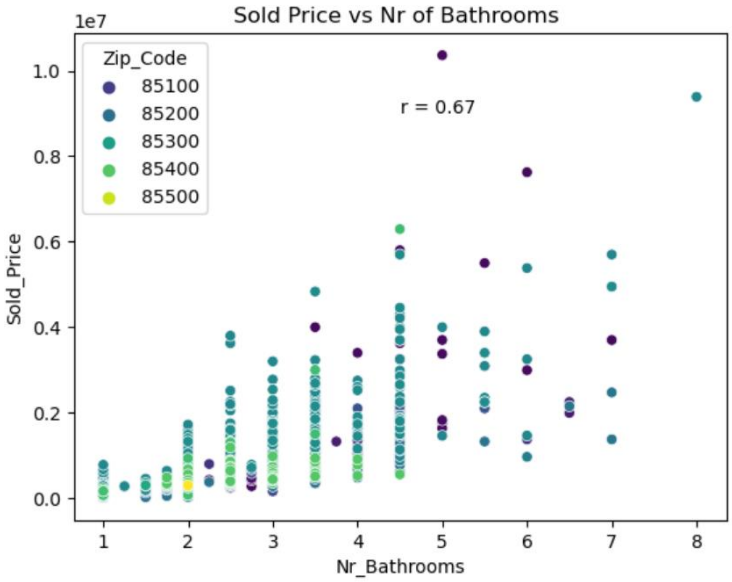
Similar correlation 0.77 is seen between Sold Price and Approx SqFt



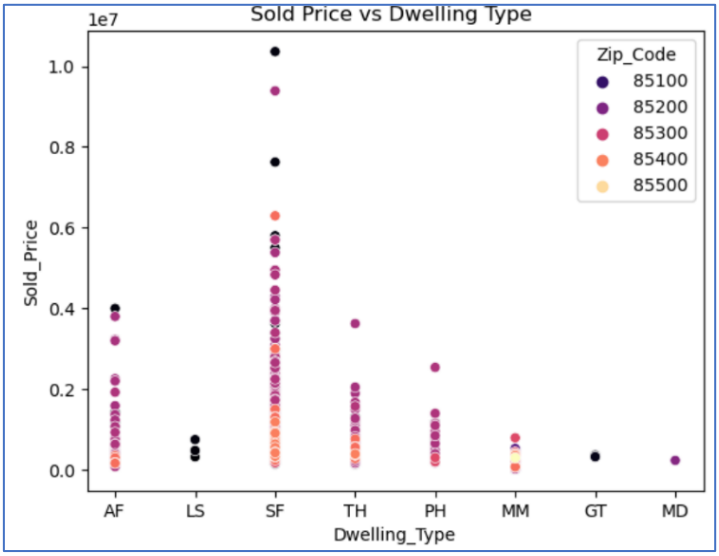
Correlation between Sold price and Nr of Bedrooms is 0.38, not strong but trend underlines the intuitive positive correlation.



Correlation between Sold price and Nr of Bathrooms is surprisingly better at 0.67 compared to Nr od Bedrooms. .

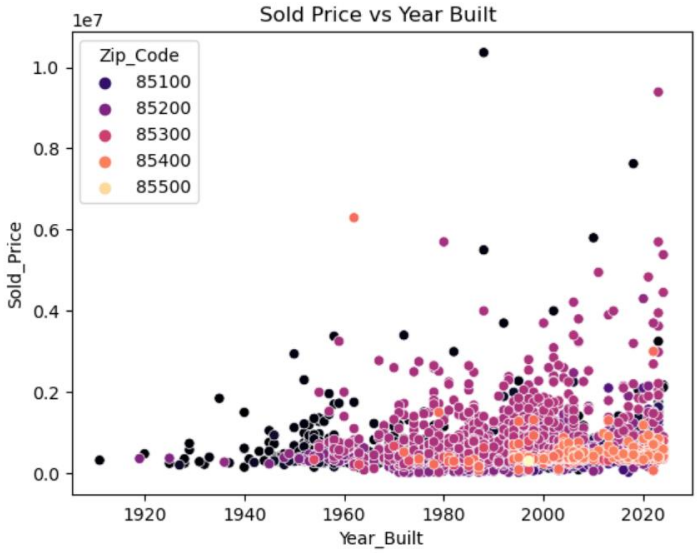


The Sold Price vs Dwelling type indicates that single family residence are by far the most valuable properties compared to Manufactured homes, town homes, Apartments style, patio homes, loft style

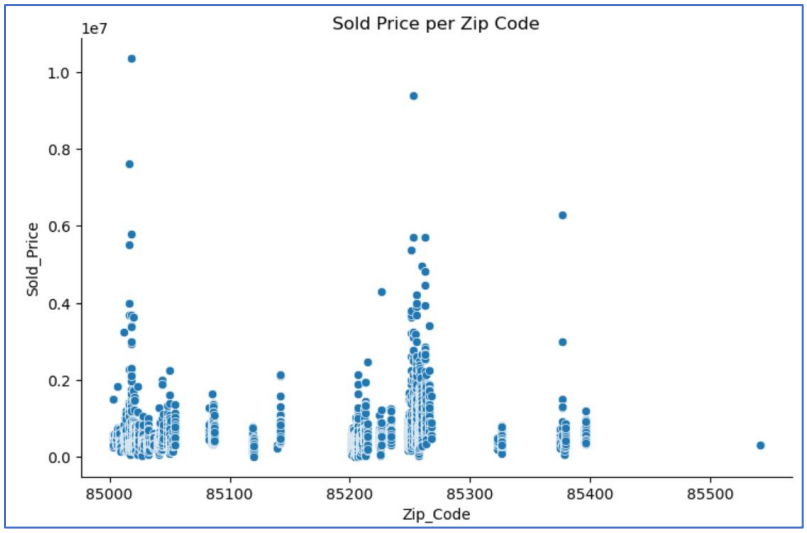


SF	single family
PH	patio home
TH	Townhouse
AF	ApartmentStyle/Flat
GT	Gemini/Twin Home
MM	Mfg/Mobile Housing
MD	Modular/Pre-Fab
LS	Loft style

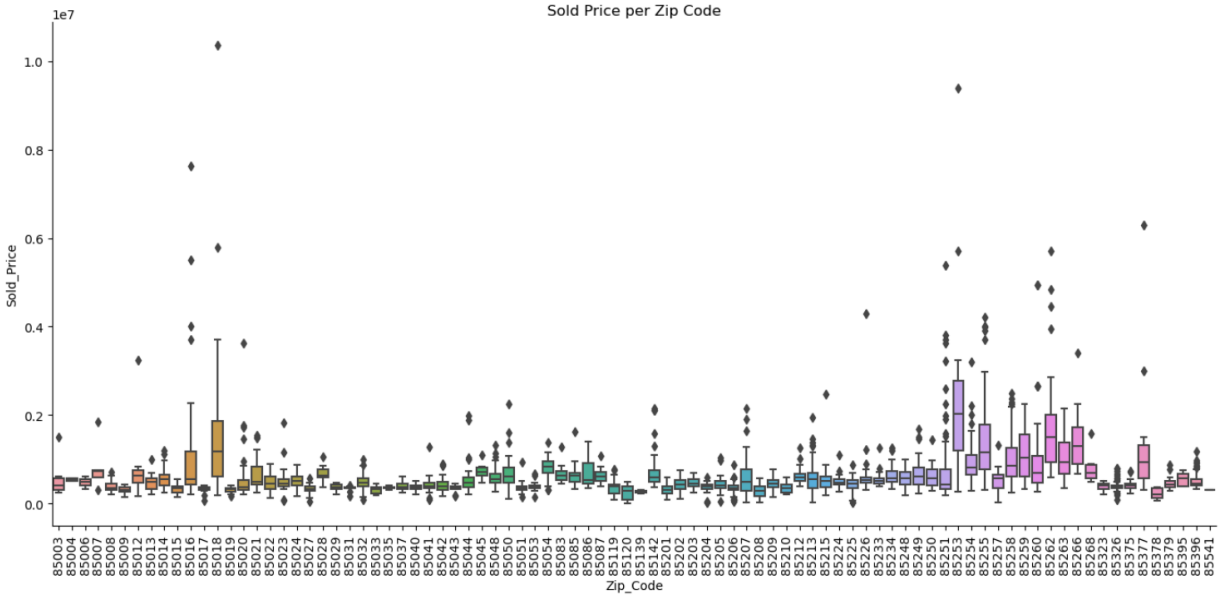
Looking at Year Built, there is a weak correlation between Sold Price and Year Built, though a positive correlation trend is visible



Sold price distribution per zip code shows a wide spread of price for zipcodes ranging from 85000 to 85050 and 85200 to 80275



Same information but using boxplots for every individual zip code to show volatility and spread



A quick look at Sold Prices between $400,000 and $600,000 indicates:

```
Sold_Price          485144.191829
Approx_SQFT           1762.596822
Price_per_SqFt         288.174879
dtype: float64
```

- an average Sold Price of $485,144
- average Approx_SQFT of 1,762 sqft
- an average Price per Sqft of $288/sqft

A quick look at Sold Prices greater than $2,000,000 indicates:

```
Sold_Price          3.111262e+06
Approx_SQFT         4.924365e+03
Price_per_SqFt      6.667067e+02
dtype: float64
```

- an average Sold Price of $3,100,000
- average Approx_SQFT of 4,924 sqft
- an average Price per Sqft of $667/sqft

These lower and higher priced houses show two distinct groups where Approx sqft and Price per Sqft is evidently distinct.

To address these data differences and vastly different value scales, , a scaling and standardization is applied in preprocessing phase.

Preprocessing and Training

Due to data leakage, 3 columns needed to be dropped:

'List_Price',

'diff_List_Sold',

‘Price_per_SqFt',

An unnecessary indexing column 'Unnamed: 0’ was also dropped.

The cleaned data set contains numerical and categorical type values. To employ statistical models the value type must be numerical. To change categorical values to numerical the get_dummies method is applied for to 12 features containing object type values .

After get_dummies application, data set have 63 columns vs 25 initial columns.

The HOA_Fee feature contained ‘NaN’ value types were replaced with zero int type values.

An info() method revealed the data set value types as follows:

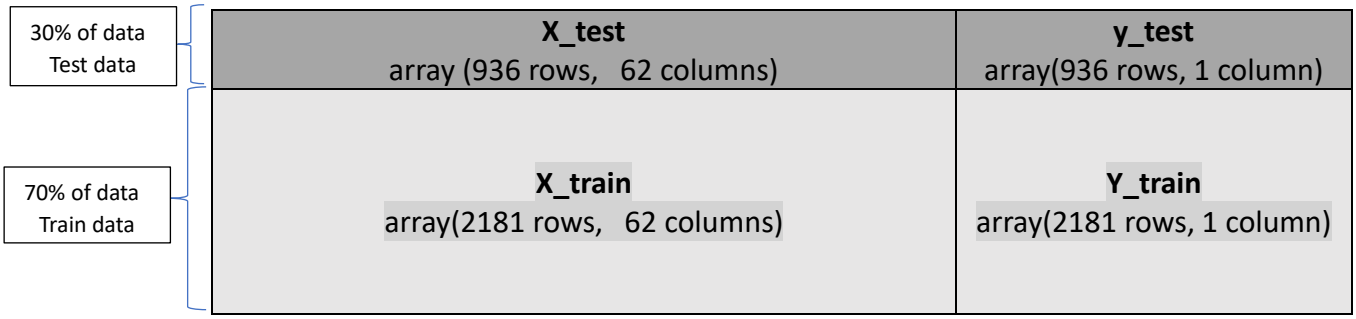
```
dtypes: float64(6), int32(50), int64(7)
memory usage: 925.5 KB
```

- 6 features: float64 type
- 50 features: int32 type
- 7 features: int64 type

Data set is ready for training.

Train/Test Split

To build a model I need to randomly divide data in two sets: a training and a test set, in a 70/30 split. To do this, I apply a function 'train_test_split' which takes all features, except the target feature, in our case Sold_Price, as arguments for **X** and target feature 'Sold_Price' as **y**.



Machine Learning

The model should predict the Sold_Price of a house based on the 62 features. The easiest prediction would be the mean of all target values. As a starting point, I calculate the mean as a threshold for model's prediction power.

This can be done in many ways. I have chosen the DummyRegressor model provided by the scikit_learn machine learning library, which is the baseline model for regression tasks.

```
dumb_reg = DummyRegressor(strategy='mean')
dumb_reg.fit(X_train, y_train)
dumb_reg.constant_
```

```
▾ DummyRegressor
DummyRegressor()
```

```
array([[649888.36359468]])
```

The predicted Sold_Price is $649,888, which, of course, is the mean value of all Sold_Price in training data set.

Metrics

To evaluate the predicted Sold_Price values we must compare it to the actual Sold_Price. I have used three metrics for this purpose:

1. R-squared (coefficient of determination)
2. MAE (mean absolute error)
3. MSE (mean squared error)

As expected, the predicted values **y_train_pred** was an array of values equal to 649,888, obtained by calling predict method on the **dumb_reg** variable.

```
y_tr_pred = dumb_reg.predict(X_train)
y_tr_pred[:5]
```

```
array([649888.36359468, 649888.36359468, 649888.36359468, 649888.36359468,
       649888.36359468])
```

1. R-squared

Train

```
r2_score(y_train, y_tr_pred)
```

```
0.0
```

Test

```
r2_score(y_test, y_te_pred)
```

```
-0.0078084421886748245
```

2. MAE

Train

```
mean_absolute_error(y_train, y_tr_pred)
```

```
347574.7837341418
```

Test

```
mean_absolute_error(y_test, y_te_pred)
```

```
313711.9215035054
```

3. MSE

Train

```
mean_squared_error(y_train, y_tr_pred)
```

```
407213445225.1448
```

Test

```
mean_squared_error(y_test, y_te_pred)
```

```
306122830945.3079
```

Metric results show the MAE performed slightly better on unseen data which is not surprising considering the underfitting of data. The R-squared showed a negative value

With these baseline metric results we can now look at more complex statistical models

Model 1: LM - Linear Regression model

Step 1: Scaling

Features have different scales, different units. We need them to be transformed in feature populations with zero mean and unit variance. To do this, we call **StandardScaler**'s fit method on both train and the split data set. Two new X variables will be obtained: **X_tr_scaled** and **X_te_scaled**, for train and test data set , respectively.

```
scaler = StandardScaler()
scaler.fit(X_train)
X_tr_scaled = scaler.transform(X_train)
X_te_scaled = scaler.transform(X_test)
```

```
▾ StandardScaler
StandardScaler()
```

Step 2: Train the model on the train split

First, we create an instance **lm** of the linear regression then we call **fit** method.

```
lm = LinearRegression().fit(X_tr_scaled, y_train)
```

To predict Sold_Price values we call **predict** method on both train and test data set:

```
y_tr_pred = lm.predict(X_tr_scaled)
y_te_pred = lm.predict(X_te_scaled)
```

To assess model performance, we use same 3 metrics from before. Results are shown below:

1. R-squared

   Train

```
r2_score(y_train, y_tr_pred)
```

```
0.6780740252759134
```

   Test

```
r2_score(y_test, y_te_pred)
```

```
0.6240249288467528
```

2. MAE

   Train

```
mean_absolute_error(y_train, y_tr_pred)
```

```
213687.1086254819
```

   Test

```
mean_absolute_error(y_test, y_te_pred)
```

```
191696.22263150994
```

3. MSE

Train

```
mean_squared_error(y_train, y_tr_pred)
```

```
131092585274.85815
```

Test

```
mean_squared_error(y_test, y_te_pred)
```

```
114202807129.04407
```

The MAE and MSE metrics show substantial improvements from the mean baseline model. The closer MAE is to 0, the more accurate the model's predictions are.

	MAE		MSE	
	Train	**Test**	**Train**	**Test**
Mean – base model	347,574	313,711	407213445225	306122830945
LM model	213,687	191,696	131092585274	114202807219

The R-squared value of 0.678 for **LM** model explains 67.8% of the target (dependent) value variation calculated from data set features (independent variables). R-squared quantifies the fit of the model, not the prediction accuracy.

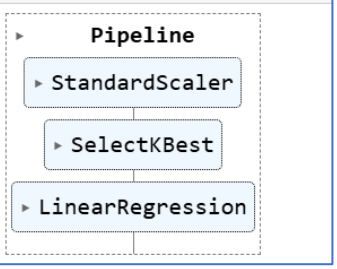
To optimize the model, I use a pipeline **pipe5** allowing to apply a list of transformers (intermediate steps) sequentially to preprocess the data. Pipelines avoid data leakage during cross-validation and hyperparameter tuning.

Third step in the pipeline is when I use **SelectKBest** featured selection method to chose the 5 most relevant features from the data set. Selection is evaluated with the **f_regression** scoring function.

```
pipe5 = make_pipeline(
    StandardScaler(),
    SelectKBest(f_regression, k=5),
    LinearRegression()
)
```

Next, I fit the pipeline and call predict method on X_train and X_test to obtain predicted values y_tr_pred and y_te_pred.

```
pipe5.fit(X_train,y_train)
```




```
y_tr_pred = pipe5.predict(X_train)
y_te_pred = pipe5.predict(X_test)
```

Assess performance of the **LM** model using the 5 most relevant features as resulted from the use of **pipe5**

The R-squared is 64.9 for train and 0.596 for test data set, which is worse than initial LM

	MAE		MSE	
	Train	**Test**	**Train**	**Test**
Mean – base model	347,574	313,711	407213445225	306122830945
LM model	213,687	191,696	131092585274	114202807219
LM – using pipe5	218,657	199,089	143050188547	122793575291

The MAE and MSE performed poorer, as well.

Cross Validation

For finetuning I apply cross validation technique by splitting the dataset into multiple subsets (folds) and training the model on different combinations of these subsets. The goal is to evaluate the model's performance on unseen data by simulating how it would generalize to new samples.

I choose the a 5 folds validation on pipe5 estimator using X_train and y_train data.

```
cv_results = cross_validate(pipe5, X_train, y_train, cv=5)
```

The cross validation test scores (f_regression) are extracted from **cv_results** dictionary as seen below:

```
cv_scores = cv_results['test_score']
```

```
array([0.6891245 , 0.6951945 , 0.66614678, 0.58246098, 0.586157  ])
```

A maximum of **0.69 R-squared** can be achieved from a LM - 5 most relevant features model.

Grid Search and hyperparameters tuning

Further tunning is done by using GridSearchCV on **pipe5** to find the optimal hyperparameters for LM model. For this I select the **selectkbest__k** parameter to tune from GridSearchCV. Loop over all features columns.

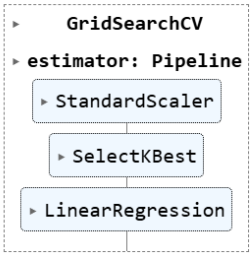
```
k = [k+1 for k in range(len(X_train.columns))]
grid_params = {'selectkbest__k': k}
```

Instantiate the hyperparameter tuning GridSearchCV, stored and defined as **lr_grid_cv**

```
lr_grid_cv = GridSearchCV(pipe5, param_grid=grid_params, cv=5, n_jobs=-1)
```

Fit the model:

```
lr_grid_cv.fit(X_train, y_train)
```



Using the **cv_results_** attribute for **lr_grid_cv** and selecting '**mean_test_score'**, '**std_test_score**' and '**param_selectkbest__k**' from the **cv_results_** dictionary, I found there are 8 most relevant features, see below:

```
score_mean = lr_grid_cv.cv_results_['mean_test_score']
score_std = lr_grid_cv.cv_results_['std_test_score']
cv_k = [k for k in lr_grid_cv.cv_results_['param_selectkbest__k']]
```

```
lr_grid_cv.best_params_
```

```
{'selectkbest__k': 8}
```

The grid search object **lr_grid_cv** is built with hyperparameters, which are parameters that control the behavior of the model during training. To access the coefficients of the best 8 hyperparameters I use this code:

```
coefs = lr_grid_cv.best_estimator_.named_steps.linearregression.coef_
features = X_train.columns[selected]
pd.Series(coefs, index=features).sort_values(ascending=False)
```

```
Approx_SQFT       563235.043130
Nr_Bathrooms       84083.553228
Pool_Private       74571.110439
Pool_no_pool       32677.590669
Type_SF           -22289.515730
Carport_Spaces    -22609.821765
Garage_Spaces     -51154.683709
Nr_Bedrooms      -198498.692108
dtype: float64
```

Predict target values using the grid search object with best 8 hyperparameters

```
y_tr_grid = lr_grid_cv.best_estimator_.predict(X_train)
y_te_grid = lr_grid_cv.best_estimator_.predict(X_test)
```

Evaluate performance using same 3 metrics:

The R-squared is 65.1 for train and 0.597 for test data set, which is not a much of an improvement compared to other models

	MAE		MSE	
	Train	**Test**	**Train**	**Test**
Mean – base model	347,574	313,711	407213445225	306122830945
LM model	213,687	191,696	131092585274	114202807219
LM – using pipe5	218,657	199,089	143050188547	122793575291
Grid Search	218120	198864	142056506910	122299181468

All attempts to optimize the linear regression models were unsuccessful, the best results are obtained for the Linear Model where all 62 features are included in the model.

Model 2: RF - Random Forest Model

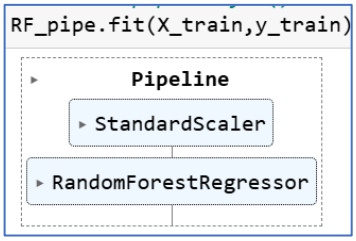
Following similar steps, I build a pipeline where the estimator is a Random Forest instead of linear regression

- Instantiate the RF_pipe

```
RF_pipe = make_pipeline(
    StandardScaler(),
    RandomForestRegressor(random_state=47)
)
```

- Fit the RF_pipe

```
RF_pipe.fit(X_train,y_train)
```



- Predict target values:

```
y_rf_tr_pred = RF_pipe.predict(X_train)
y_rf_te_pred = RF_pipe.predict(X_test)
```

- Assess RF prediction using the 3 metrics

The R-squared is **0.96** for training data and **0.73** for test data. This is a markedly improved performance compared to linear regression model.

	MAE		MSE	
	Train	**Test**	**Train**	**Test**
RF model	56671	130173	15231692653	81244969260

Refining RF model by using cross validation with 5 folds:

```
rf_default_cv_results = cross_validate(RF_pipe10, X_train, y_train, cv=5)
```

The **'test_score'** array is shown below:

```
rf_cv_scores = rf_default_cv_results['test_score']
rf_cv_scores
```

```
array([0.71022963, 0.70564126, 0.74352067, 0.72791179, 0.65435751])
```

The 0.74 R_squared is a slight improvement from 0.73. seen before the cross validation processing.

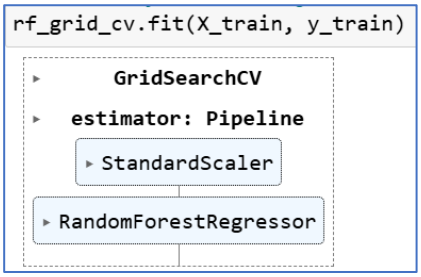
Fine tuning using GridSearchCV optimizer:

- Instantiate a gride search object:

```
rf_grid_cv = GridSearchCV(RF_pipe10, param_grid=grid_params, cv=5, n_jobs=-1)
```

- Fit the grid search object:

```
rf_grid_cv.fit(X_train, y_train)
```



- Find best hyperparameters parameters. There are 33

```
rf_grid_cv.best_params_
```

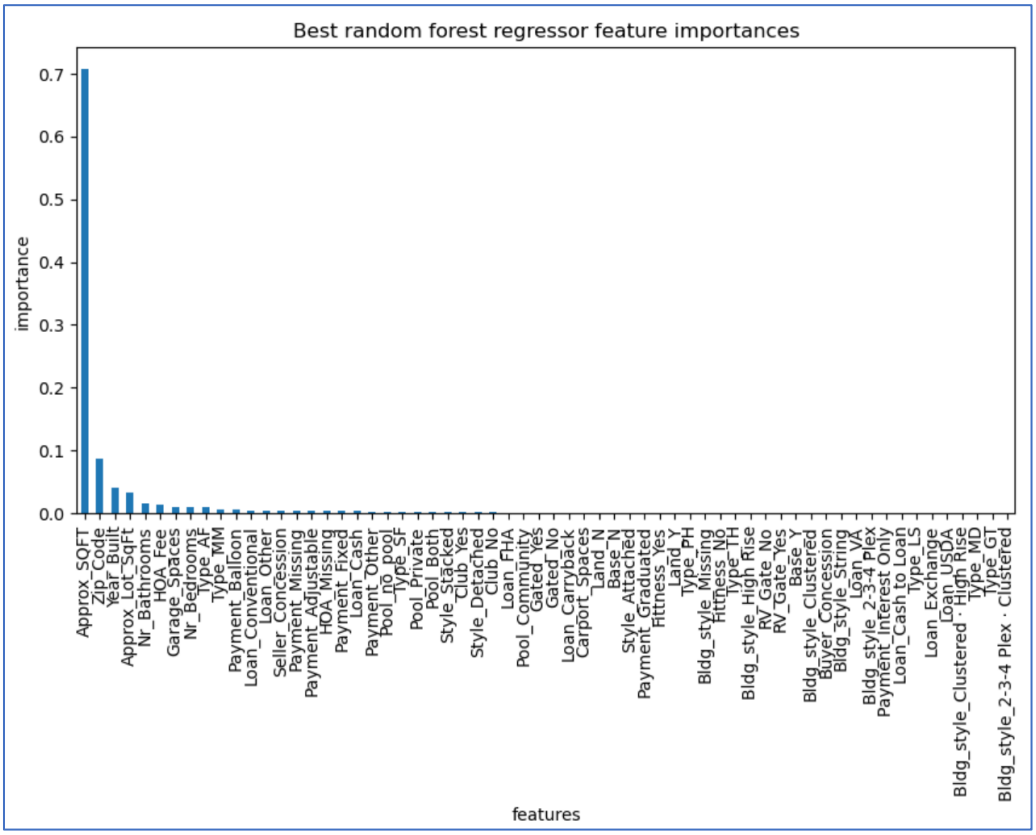
```
{'randomforestregressor__n_estimators': 33, 'standardscaler': StandardScaler()}
```

- Get best RF scores using a 5 fold cross validation. Best score is **0.75**

```
rf_best_cv_results = cross_validate(rf_grid_cv.best_estimator_, X_train, y_train, cv=5)
rf_best_scores = rf_best_cv_results['test_score']
rf_best_scores
```

```
array([0.66706201, 0.71585639, 0.75378211, 0.6983821 , 0.64691719])
```

- Visualization of Feature importance. The dominant feature is Approx. SqFt.



The dominant top 8 features in Random Forest:	The Linear model top 8 best estimators are:
• Approx_SqFt • Zip_Code • Year_Built • Approx_Lot_SqFt • Nr_Bathrooms • HOA_Fee • Garage_Spaces • Nr_Bedrooms	• Approx_SqFt • Nr_Bathrooms • Pool_Private • Pool_no_pool • Type_SF • Carport_Spaces • Garage_Spaces • Nr_Bedrooms

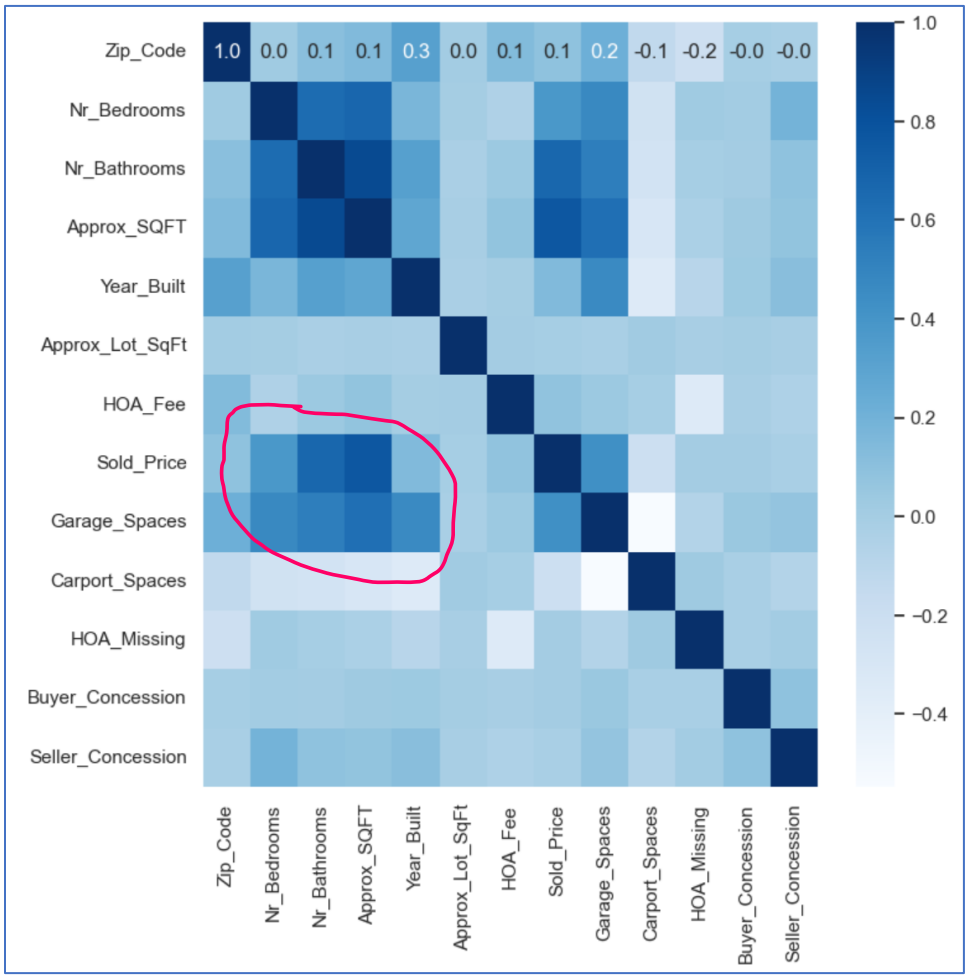
Model 3: XGB – Extreme Gradient Booster

For this model I exclude all object features from the original dataset. I keep only the numerical features as seen below:

```
df_xgb_num=df_xgb.select_dtypes(exclude=['object'])  # exclude object columns
df_xgb_num.dtypes
```

```
Zip_Code              int64
Nr_Bedrooms           int64
Nr_Bathrooms        float64
Approx_SQFT           int64
Year_Built            int64
Approx_Lot_SqFt       int64
HOA_Fee             float64
Sold_Price            int64
Garage_Spaces       float64
Carport_Spaces      float64
HOA_Missing           int64
Buyer_Concession    float64
Seller_Concession   float64
dtype: object
```

A correlation heatmap shows a visual on correlation between features, with area of higher correlation circled in red



- Instantiate an XGB model:

```
xgb_r = xg.XGBRegressor(objective ='reg:linear', n_estimators = 10, seed = 123)
```

- Fit the model

```
xgb = xgb_r.fit(X_train_xgb, y_train_xgb)
```

- Predict target value:

```
pred = xgb_r.predict(X_test_xgb)
```

- Assess the RF model performance using the 3 metrics

The R-squared is **0.942** for training data and **0.761** for test data. This is a better performance compared to linear regression model but not as good as RF model.

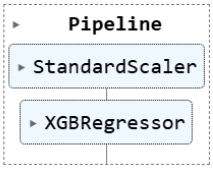
	MAE		MSE	
	Train	**Test**	**Train**	**Test**
XGB model	83327	135633	19353499953	111921124240

- Refining the XGB model by using a pipe and cross validation

```
xgb_pipe = make_pipeline(
    StandardScaler(),
    xgb
)
```

- Fit the xgb_pipe

```
xgb_pipe.fit(X_train_xgb, y_train_xgb)
```



- Refining RF model by using cross validation with 5 folds:

```
cv_results = cross_validate(xgb_pipe, X_train_xgb, y_train_xgb, cv=5)
```

- Getting test scores.

```
cv_scores = cv_results['test_score']
cv_scores
```

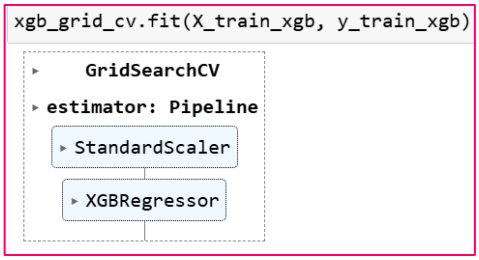
```
array([0.69911428, 0.78914762, 0.68304949, 0.51715063, 0.74668696])
```

- Grid Search

```
xgb_grid_cv = GridSearchCV(xgb_pipe, param_grid=grid_params, cv=5, n_jobs=-1)
```

- Grid Search Fit:

```
xgb_grid_cv.fit(X_train_xgb, y_train_xgb)
```



- Predict target values for train and test data

```
y_tr_pred = xgb_pipe.predict(X_train_xgb)
y_te_pred = xgb_pipe.predict(X_test_xgb)
```

- Assess performance using the 3 metrics

The R-squared is **0.939** for training data and **0.7603** for test data. This is very close to the previous numbers for XGB before the grid search processing. This is also better performance compared to linear regression model but not as good as RF model.

	MAE		MSE	
	Train	**Test**	**Train**	**Test**
XGB model	83327	135633	19353499953	111921124240
XGB Grid Search	82135	136774	20235191413	112400324530

Summary for the 3 models **LM**, **RF** and **XGB** and Final Model Selection using cross validation on the negative of mean absolute error.

```
lr_neg_mae = cross_validate(lr_grid_cv.best_estimator_, X_train, y_train,
                            scoring='neg_mean_absolute_error', cv=5, n_jobs=-1)
```

```
rf_neg_mae = cross_validate(rf_grid_cv.best_estimator_, X_train, y_train,
                            scoring='neg_mean_absolute_error', cv=5, n_jobs=-1)
```

```
xgb_neg_mae = cross_validate(xgb_grid_cv.best_estimator_, X_train_xgb, y_train_xgb,
                            scoring='neg_mean_absolute_error', cv=5, n_jobs=-1)
```

	MAE	Best model	Ranking
LM model	198864		3
RF model	130664	selected	1
XGB model	136774		2

The RF model was selected as best model

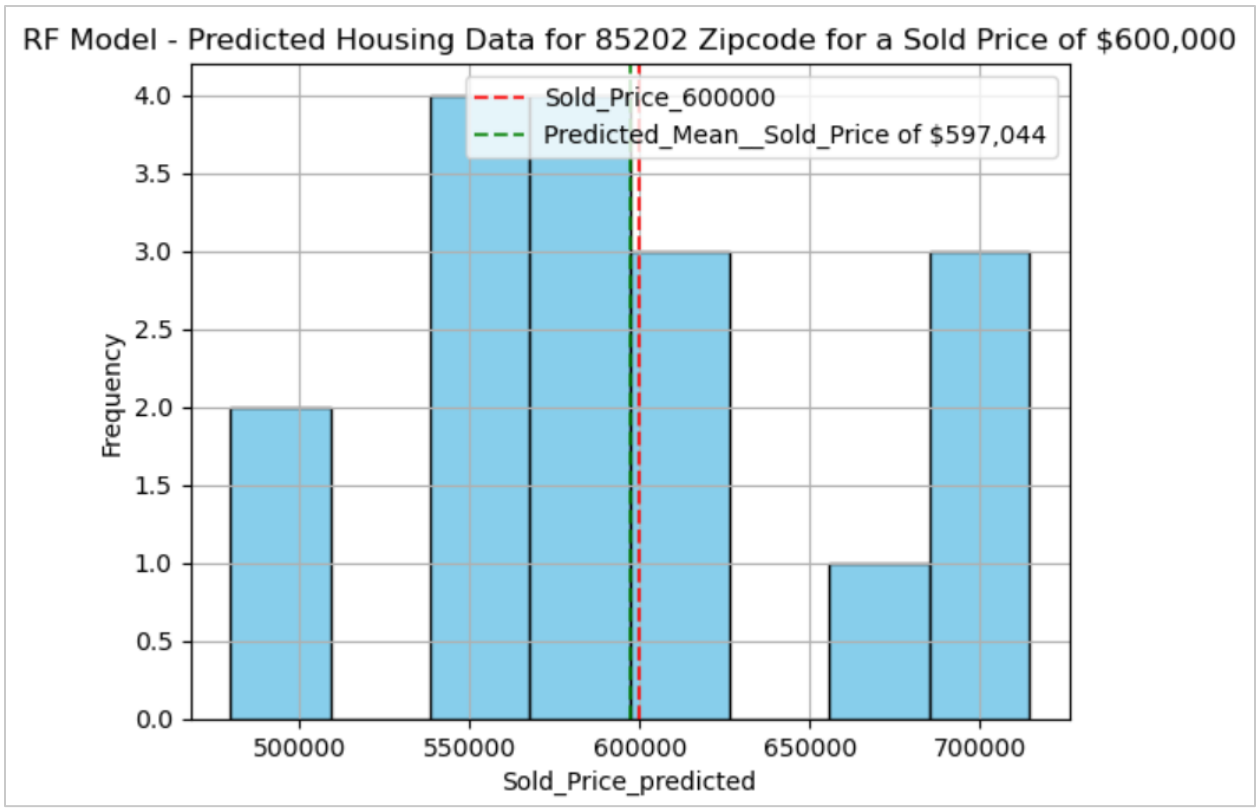
Model was used for a house valued at $600,000 in zipcode 85202 in Maricopa County, Arizona.

```
X_85202_rf = housing_data.loc[housing_data.Sold_Price == 600000, rf_model.X_columns]
y_85202_rf = housing_data.loc[housing_data.Sold_Price == 60000]
```

The mean predicted value was $597,044

```
zipcode_85202_pred_rf.mean()
```

```
597044.0303030303
```



The difference of $2,956 represents a 0.4% error from the actual sold Price of $600,000 which is less than the 1% threshold considered for criteria for success.

Future work

- Population income and demographic are variables to be included in a an improved prediction model
- 10 years Treasury bond rates, which is the proxy for mortgage rates, will be considered in the future model.
- The new buildings not listed in MLS are important for an expanded data set and for a more accurate predictions
- The outliers in exclusive zipcode may constitute the basis to build a model designed exclusively for high end markets where the variability is excessive.
- Influx of businesses, unemployment data and inflation are variables that would help improve the predictability of housing prices.