Springboard

Capstone Project 2

Menara App

First and Only Web App for

House Price Estimation, Forecast and GreatSchools Search

Final Report

June 2021

Aktham Momani – Data Science: Advanced Machine Learning

Mentor: Kenneth Gil-Pasquel

Table of Contents

1. Introduction

Whether you want to buy, sell, refinance, or even remodel a home, there's an increase need to have a platform or an application where it offers all the tools in one place to help buyers, sellers and landlords to make the most informed decision of investing money in a real estate.

That's why, here we're building an interactive web App called **Menara** that offers:

- The lowest 8.5% margin off-error for off-market homes in North California **(Competitive to the most known Home Estimate Sites e.g., Redfin)** by using the most sophisticated Machine learning algorithms.
- A golden opportunity to give you a sneak peek into the future; Up to 14 Months of house price forecast per zipcode by using a new model called **Neural Prophet**.
- A unique access to **GreatSchools**, the most trusted source of schools rating for many buyers and not just buyers with children because "location, location, location," means "schools, schools, schools."

2. Objective

Build an interactive web App called **Menara** using a Python library called **Streamlit**. It's an open-source app framework which turns data scripts into shareable web apps in minutes where we just write some python code and it will handle everything.

Menara App will be divided into three sections:

- **House Price Estimation:** This is the heart of the app which houses the final Machine Learning Model.
- **Median House Price Forecast Per Zip Code:** This is where we have a Neural Network based Time-Series Model.
- **GreatSchools Search:** In this section we'll request all Schools rating and information from **GreatSchools** REST API for specific cities/Zip codes in North California.

3. Data

Since the beginning, we made our mind to be unique in approaching this typical supervised machine learning problem, so we Built our dataset from scratch, utilizing multiple reliable sources as illustrated below:

a) Main Dataset: Source **redfin**:
   - Obtained sold houses (Off-Market) from Dec 2019-Dec 2020.
   - The sold houses located in North California distributed between 49 cities and 94 Zip Codes.
   - Data has 8,790 Observations and 27 Variables.

b) Median Income per Zip code:
   - Source: **USA.com**

c) Hotness score (0-100) to reflect the demand and supply per zip code:
   - Source: **realtor**

d) Public School per zip code:
   - Source: **Homeland Infrastructure Foundation-Level Data**

e) GreatSchools Rating to reflect School's rating in all the selected 47 cities in North CA:
   - Source: **GreatSchools.org API**

f) Shopping and Mall centers in CA per city:
   - Source: **Wikipedia**

g) Universities and colleges list in CA per city:
   - Source: **CA Colleges**

h) BART (Bay Area Rapid Transit) stations with parking (Y/N) per zip code in CA:
   - Source: **BART**

4. Data Wrangling

4.1 Introduction

The Data wrangling step focuses on collecting our data, organizing it, and making sure it's well defined. For our project we have collected below datasets to have a good foundation so we can build a Machine Learning model with the best performance possible:

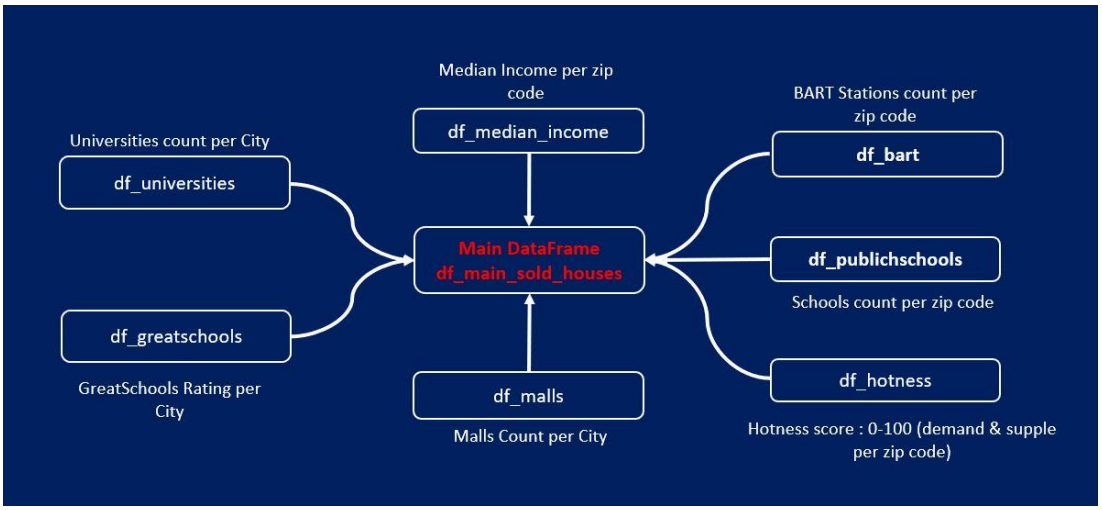


Figure 1: Data Wrangling-8 Datasets

4.2 Objectives

We have 8 Datasets to support this project as shown in Figure 1, so we'll focus in below:

- Clean missing values, duplicate values, wrong values and removing insignificant columns.
- Renaming Column labels.
- Correct datatypes.
- Merging and concatenation will be needed.
- The GreatSchools API is a REST-based web service, so will be using Python Packages:
  - requests: To request the data from GreatSchools API.
  - xml.etree.ElementTree module: to implement a simple and efficient API for parsing and creating XML data.
  - glob: to concatenate all the API output in one final DataFrame.
- Data Wrangling layout and the Initial DataFrame structure: at this stage we'll not be overzealous in our cleaning because we need to explore the data to better understand it in the exploratory data analysis Stage (EDA) where we'll provide the final DataFrame Structure.

4.3 Merging DataFrames

After completing the data wrangling as mentioned in section 4.2, now we need to merge below DataFrames into just one main DataFrame:

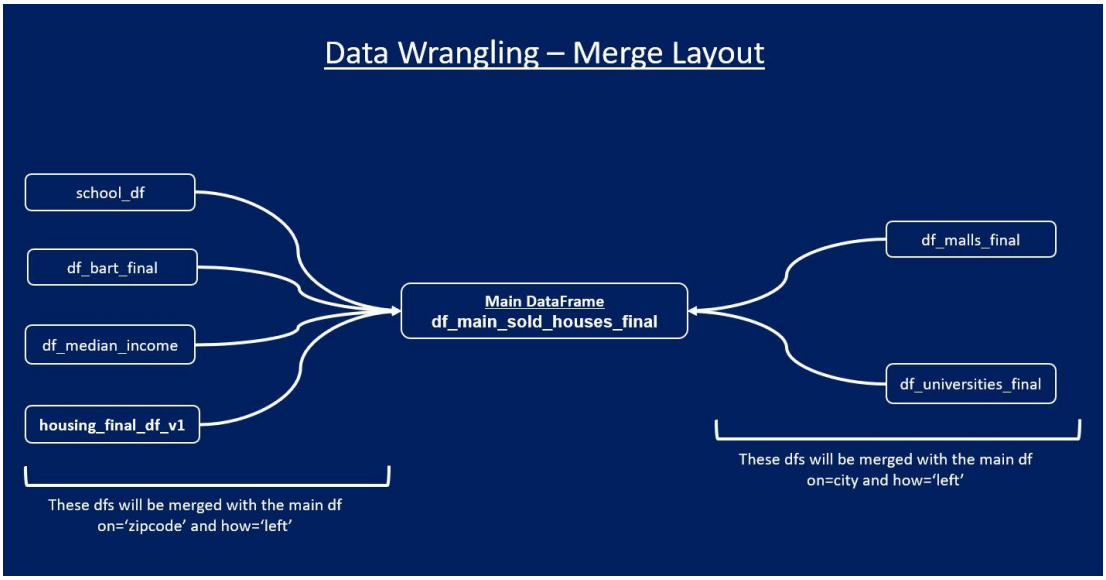


Figure 2: Merging DataFrames

4.4 Target Features

25 input features against the dependent feature "price" were the result of the data Wrangling part as shown below:

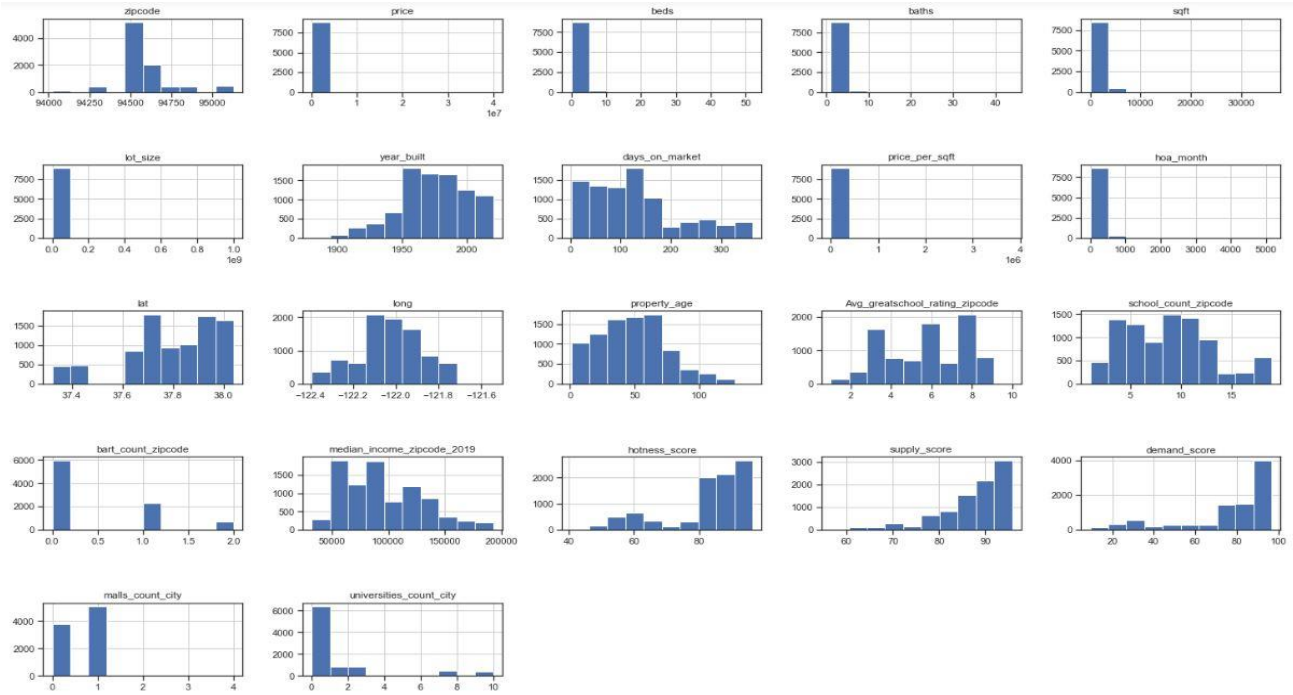


Figure 3: Target Features

5. Exploratory Data Analysis

5.1 Introduction

Now that we've obtained, cleaned, and wrangled our dataset into a form that's ready for analysis, it's time to perform exploratory data analysis (EDA).

5.2 Objectives

- To get familiar with the features in our final DataFrame.
- Generally, understand the core characteristics of our cleaned DataFrame.
- Explore the data relationships of all the features and understand how the features compare to the response variable.
- Let's be creative and think about interesting figures and all the plots that can be created to help deepen our understanding of the data.

5.3 Initial Statistics Summary

Stats	Summary
Avg Houses Price	1,058,497 USD
Std	948,060.35
Min House Price	1.600 USD
Q1	607,000 USD
Median House Price	835,000 USD
Q3	1,250,000 USD
Max House Price	39,988,000 USD
House Count	8918

Table 1:Initial Statistics Summary

5.4 DataFrame Final Modifications Summary

- Concatenate columns: Some of the columns need to be merged together for better visibility.
- Insignificant Columns: Check if all columns are significant, otherwise let's drop them.
- Duplicates: Explore the columns to make sure no duplicates.
- Renaming Columns: Some of the column's Labels need to be modified to correctly represent the underlined information.
- Wrong Values: Check if all our data is valid, either by correcting the values or dropping the rows.
- price: Check for any outliers so we can eliminate to better analyze the data.

5.5 Main Features

- Numerical Variables: 'price', 'sqft', 'price_per_sqft', 'lot_size', 'hoa_month' and 'property_age'
- Categorical Variables: 'beds', 'baths', 'school_rating', 'school_count', 'median_income', 'malls_count' and 'universities_count'
- Please note we'll be using a scatterplot to investigate Numerical Variables and a combination of scatterplot, boxplot and stripplot. to investigate Categorical Variables
- The main features before and after dealing with outliers in the Dataset (Outliers were eliminated by imputing with correct values or using mean() or median() or dropping them):

Figure 4: 'price' with Outliers

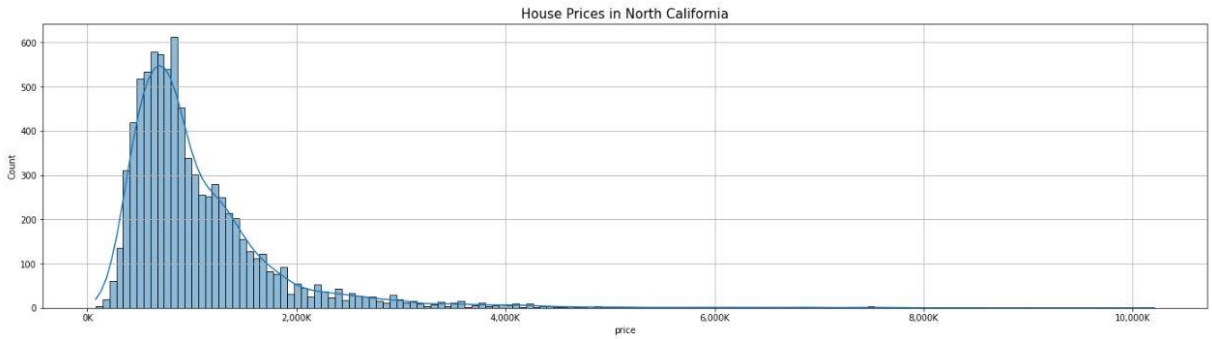


Figure 5: 'price' without Outliers

Figure 6: 'sqft', No outliers, positive correlation between 'price' & 'sqft'

Figure 7: 'price_per_sqft' with Outliers

Figure 8: 'price_per_sqft' without Outliers

Figure 9: 'lot_size' with Outliers

Figure 10: 'lot_size' without Outliers

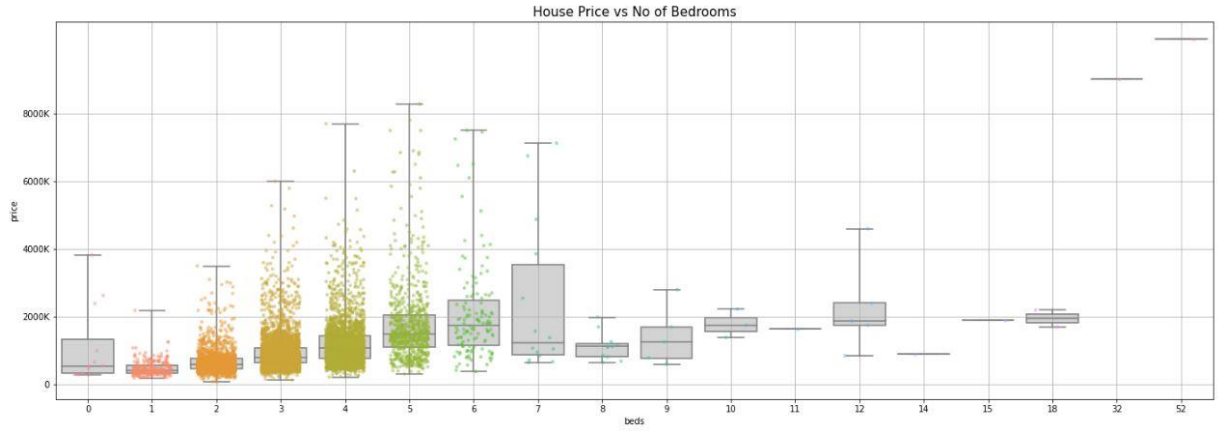


Figure 11: 'beds' with Outliers

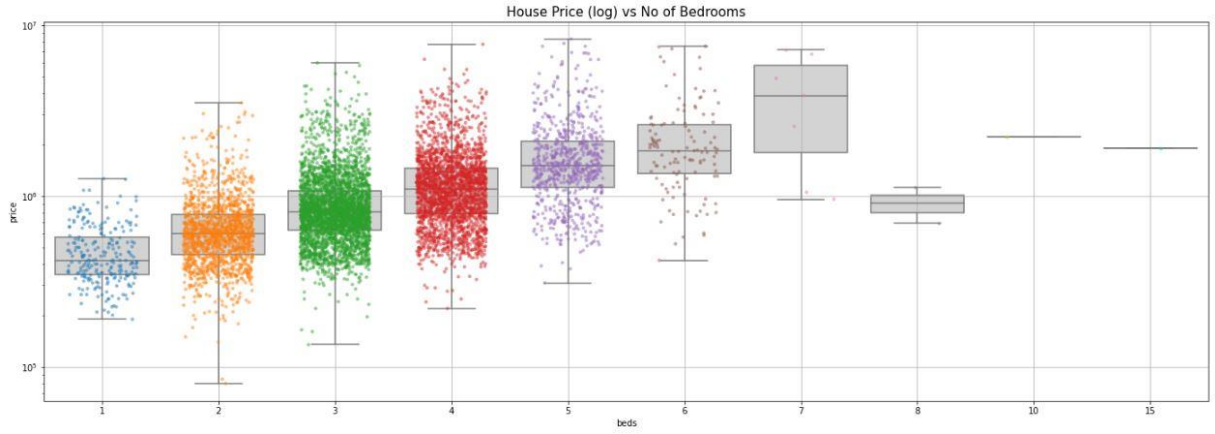


Figure 12: 'beds' without Outliers

Figure 13: 'school_rating': No Outliers, positive correlation between 'price' & 'school_rating'

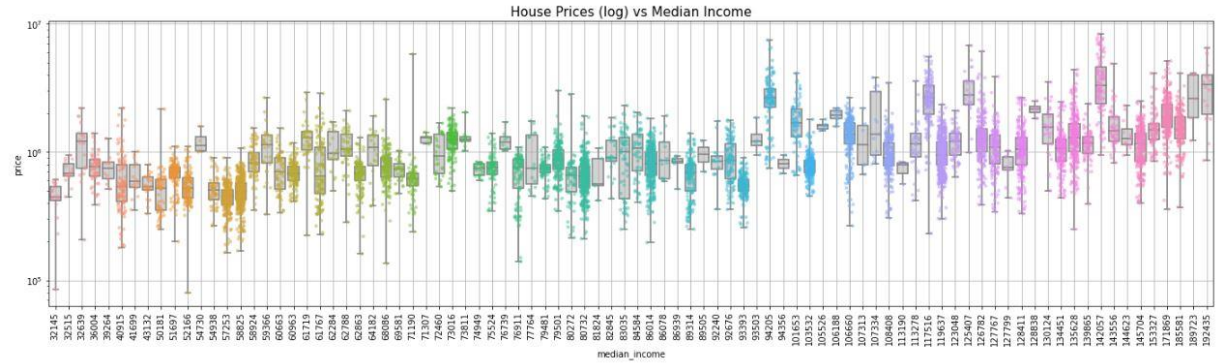


Figure 14: 'median_income': No Outliers, positive correlation between 'price' & 'median_income'

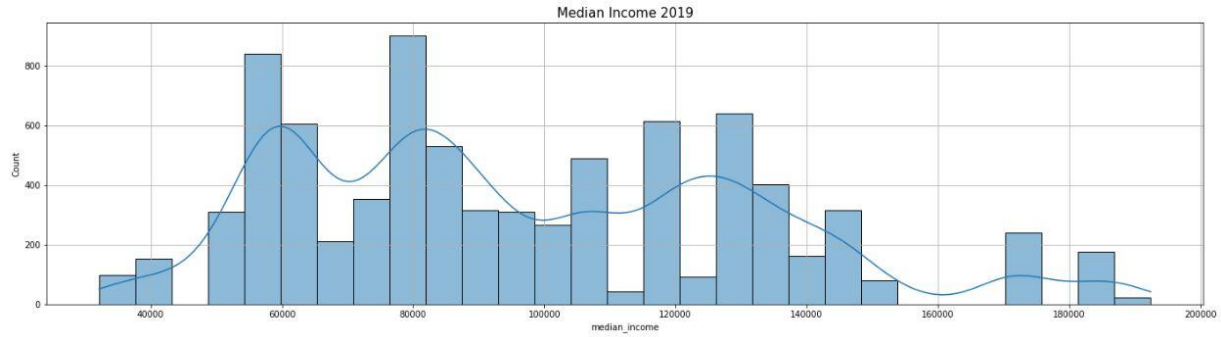


Figure 15: 'median_income' Histogram

5.6 Features Visualization using PairPlot and Heatmap

- PairPlot: let's use PairPlot from seaborn to plot our dependent variable 'price' against the main features: 'beds', 'property_type', 'baths', 'sqft', 'school_rating', 'school_count', **'bart_count', ''median_income', ''lot_size', ''hoa_month', 'malls_count', 'university_count' per 'property_type'.

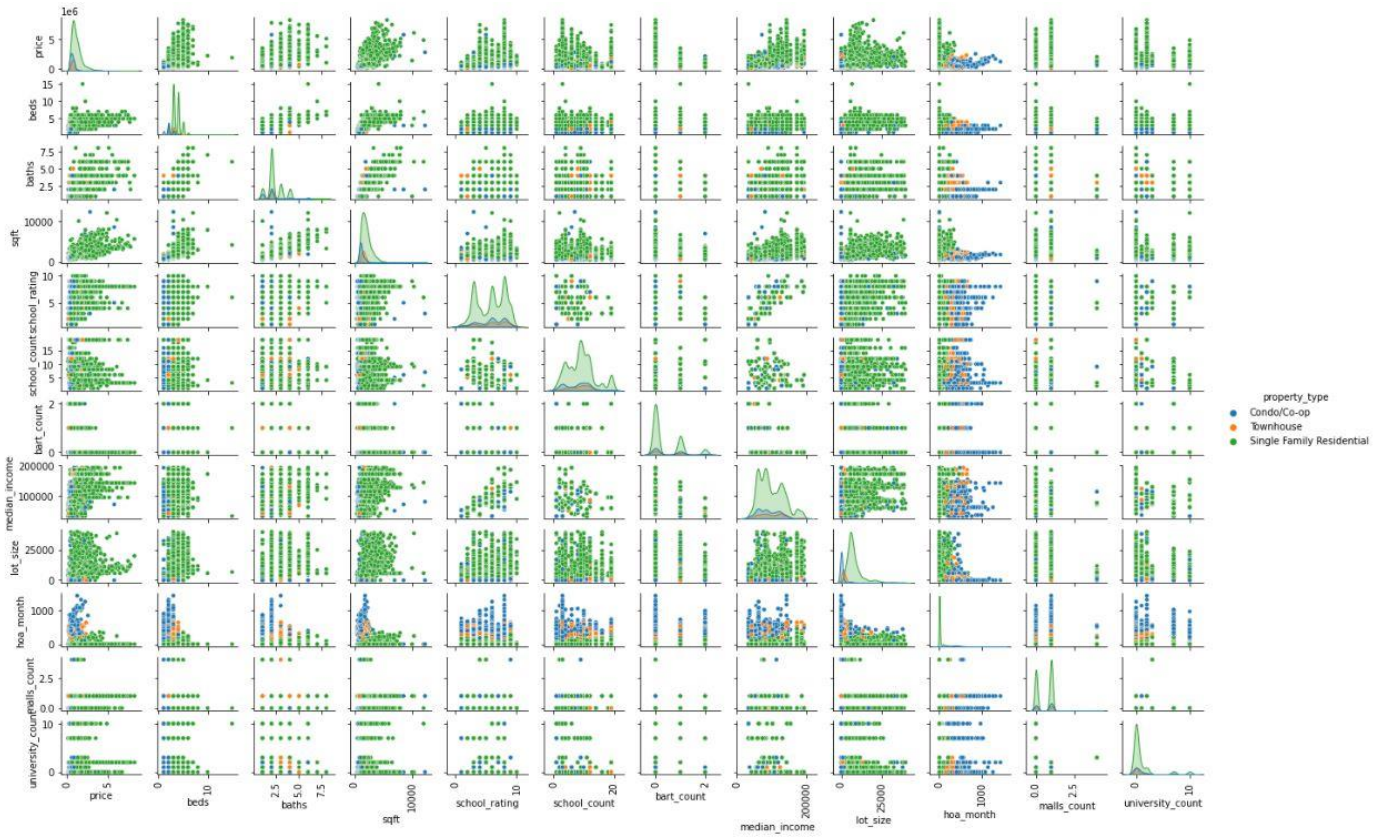


Figure 16: PairPlot Features against property type

- **Heatmap**: let's use heatmap from seaborn to Check the correlation between our dependent variable 'price' against the main features: 'beds', 'property_type', 'baths', 'sqft', 'school_rating', 'school_count', **'bart_count', ''median_income', ''lot_size', ''hoa_month', 'malls_count', 'university_count' per 'property_type'.

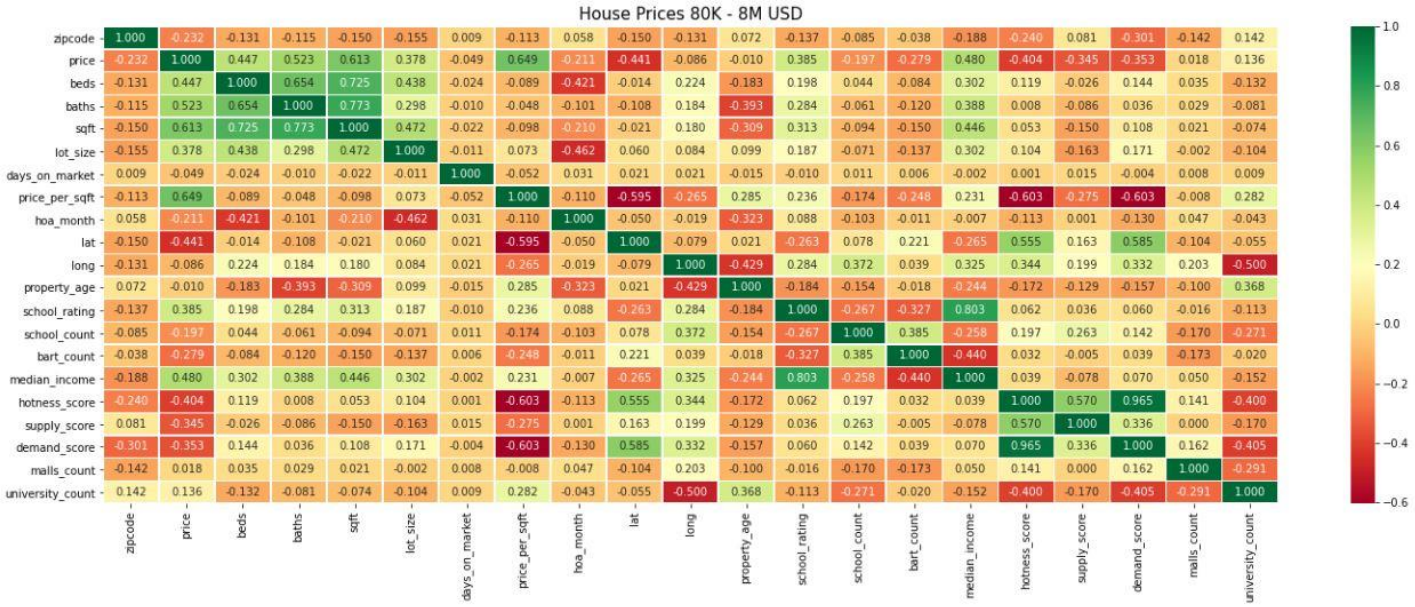


Figure 17: Heatmap

5.7 EDA Summary

- Original Data was having 8917 Observations and 26 variables (Output from Data Wrangling Part).
- There were modifications in the DataFrame which focused mainly in eliminating the outliers as much as possible:
  - Missing Values (NANs): There's no NANs in the df.
  - Concatenate columns: For better visibility, we merged 'address', 'city', 'state', and 'zipcode' into one column called 'address'.
  - Insignificant Columns: We dropped 'address', 'city', 'state', 'year_built', 'address_m' and 'year_built' column was transformed into column 'property_age'.
  - Duplicates: We removed 567 complete duplicates (Observations having the same variables) from 'address'
  - Renaming Columns: We renamed all columns to improve visibility.
- Wrong Values: We divided the variables into 2 categories: Numerical and categorical as below:
  - Numerical Variables: 'price', 'sqft', 'price_per_sqft', 'lot_size', 'hoa_month' and 'property_age'
  - Categorical Variables: 'beds', 'baths', 'school_rating', 'school_count', 'median_income', 'malls_count' and 'universities_count'

    - We used scatterplot to investigate Numerical Variables and a combination of scatterplot, boxplot and stripplot. to investigate Categorical Variables
    - Main target of this section was to eliminate outliers and impute and fix wrong values.
    - We reduced 'property_type' from 5 to 3 and we focused mainly in Single Family Residential, Condo/Co-op and Townhouse anything else was dropped (165 rows).
    - We dropped 11 rows associated with 'Beds' >30 and 'Beds'=0 (wrong values). also, we fixed 5 houses after we imputed the correct values (source: http://www.zillow.com).
    - We found 1 outlier at 'price_per_sqft'' and we imputed the correct values (source: http://www.zillow.com) 593 USD instead of 592,692 USD.
    - We found 163 outliers at 'lot_size' and we imputed the median values=5,760 Sq. Ft. (all values at these rows initially were wrong >40,000 Sq. Ft).
    - We found 7 outliers at 'hoa_month' and we imputed the mean values=115.92 USD (all values at these rows initially were wrong >1,500 USD).
- 'price': This was the main focus in imputing wrong values and eliminating outliers:
    - We dropped 7 houses > 14M USD
    - We fixed one wrong house price from 1,600 USD to 794,126 USD
    - PairPlot and heatmap revealed that we have strong correlation between 'price' and 'beds', 'baths', 'sqft', 'lot_size', 'price_per_sqft', 'school_rating', 'median_income' and 'university_count'.
- Further fine tuning in term of house price ranges revealed that when we keep only houses with price tag between 300K USD and 2M USD we see the highest correlation between ‘price' and the above features (775 Houses were dropped) as shown below:
    - House Prices between 80K USD - 8M USD: 8171 Houses
    - House Prices between 80K USD - 4M USD: 8081 Houses (dropped 90 houses)
    - House Prices between 300K USD - 4M USD: 7964 Houses (dropped 117 houses)
    - House Prices between 300K USD - 2M USD: 7396 Houses (dropped 568 houses)
- We're highlighting the correlation between difference house price ranges against all the possible features (by applying a conditional formatting using correlation() function (defined) and df.style property):
    - Green: Correlation > 0.4
    - Blue: 0 < Correlation < 0.4
    - Red>Red: Correlation < 0

	house_price_80K_8M	house_price_80K_4M	house_price_300K_4M	house_price_300K_2M
zipcode	-0.232481	-0.152358	-0.154542	0.126938
beds	0.446901	0.452340	0.441095	0.470590
baths	0.522976	0.505904	0.497517	0.477544
sqft	0.613164	0.609072	0.602328	0.628571
lot_size	0.378243	0.413851	0.407787	0.410909
days_on_market	-0.049492	-0.060262	-0.057761	-0.077175
price_per_sqft	0.648649	0.623998	0.617716	0.477801
hoa_month	-0.210898	-0.224408	-0.214820	-0.215245
lat	-0.440943	-0.436983	-0.429446	-0.394447
long	-0.085858	-0.056451	-0.057514	0.036530
property_age	-0.009635	-0.001237	0.002500	-0.060223
school_rating	0.384551	0.435499	0.427206	0.514370
school_count	-0.196507	-0.206063	-0.201268	-0.273568
bart_count	-0.278580	-0.299027	-0.292656	-0.283708
median_income	0.479796	0.520984	0.514408	0.543510
hotness_score	-0.403704	-0.371385	-0.375008	-0.244618
supply_score	-0.345309	-0.279392	-0.288473	-0.123786
demand_score	-0.353043	-0.335887	-0.336982	-0.240048
malls_count	0.018225	-0.007075	-0.005790	-0.063246
university_count	0.136250	0.150935	0.148901	0.142359

Table 2: Correlations Benchmarking between 4 different DataFrames

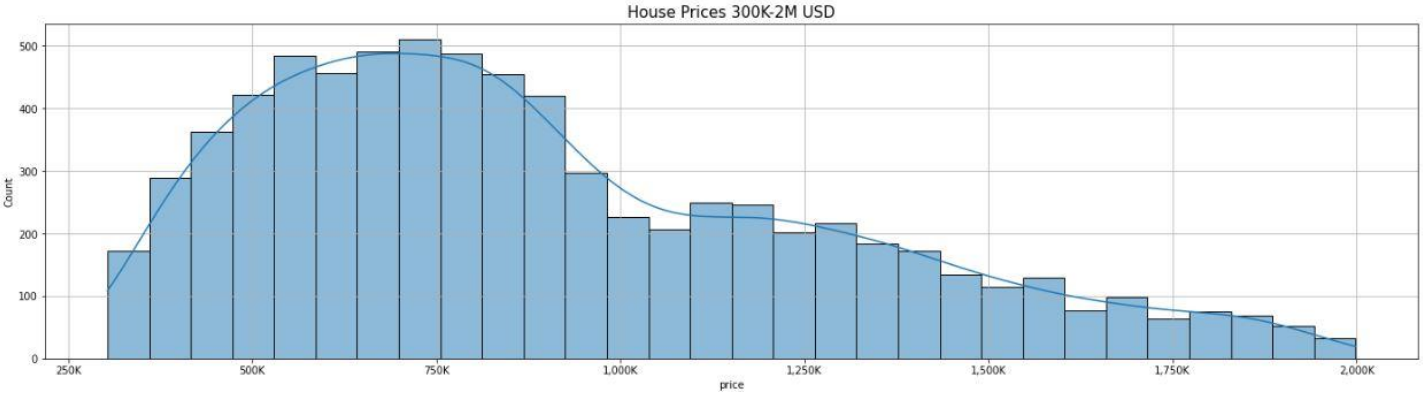


Figure 18: 'price' DataFrame with houses between $300K-2M USD

6. Pre-processing and Training Data Development

6.1 Introduction

The pre-processing and Training data is considered as the final step for data manipulation, where we can make sure that:

- No Outliers & No missing data
- Features Engineering:
  - Engineer New Features
  - Drop Features
  - Change feature's structure
  - Encode Features

6.2 Removing Outliers

Outliers are data points that exist far away from the majority of your data. This can happen due to several reasons, such as incorrect data recording to genuine rare occurrences. Either way you will often want to remove these values as they can negatively impact your models. An example of the negative effect can be seen here where an outlier is causing almost all of the scaled data to be squashed to the lower bound.

6.2.1 "price" Visualization with Outliers boundaries using Quantile and Standard Deviation

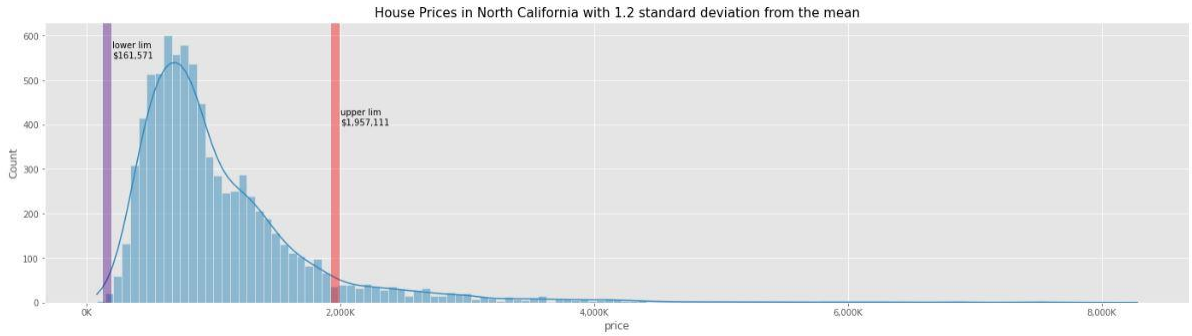


Figure 19: 'price' Boundaries with 1.2 Standard Deviation

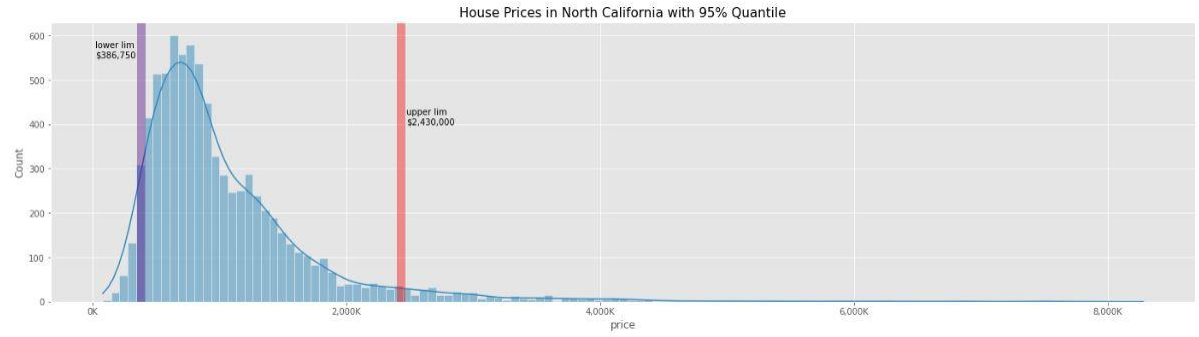


Figure 20: 'price' Boundaries with 95% Quantile

6.2.2 Visualization all difference datasets based on multiple outliers' method: Target Variable "price" Visualization using different datasets

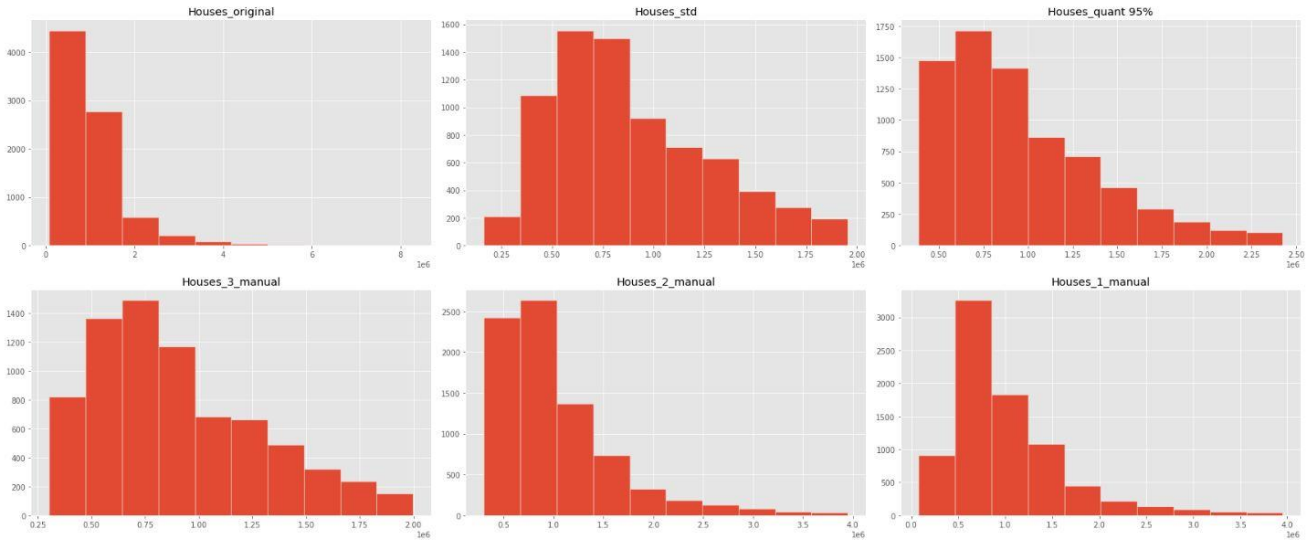


Figure 21: 'price' Visualization with different DataFrames

6.2.3 Datasets comparison using statistics summary

According to the Table 3 & 4, std_df is showing the best, mean, std and Correlations by eliminating all data greater than 1.2 standard deviation from the mean and at the same time we managed only to lose 687 outliers.

	df_main	std_df	quant_df	man_price_1	man_price_2	man_price_3
count	8171	7484	7352	8081	7964	7396
mean	1.05934e+06	887769	966399	1.01363e+06	1.0249e+06	900685
std	748142	387091	437422	600975	598064	386134
min	80000	165000	387000	80000	303482	303482
25%	610000	590000	635000	610000	620000	603248
50%	847000	808000	847000	840000	850000	815000
75%	1.275e+06	1.15e+06	1.21625e+06	1.255e+06	1.26262e+06	1.155e+06
max	8.28e+06	1.955e+06	2.425e+06	3.95e+06	3.95e+06	1.99889e+06

Table 3: Statistics Summary Benchmark between different DataFrames

	df_main_corr	std_df_corr	quant_df_corr	man_price_1_corr	man_price_2_corr	man_price_3_corr
zipcode	-0.232481	0.128117	0.044806	-0.152358	-0.154542	0.126938
beds	0.446901	0.481437	0.442365	0.452340	0.441095	0.470590
baths	0.522976	0.482788	0.478359	0.505904	0.497517	0.477544
sqft	0.613164	0.630884	0.624378	0.609072	0.602328	0.628571
lot_size	0.378243	0.413622	0.424785	0.413851	0.407787	0.410909
days_on_market	-0.049492	-0.077447	-0.069514	-0.060262	-0.057761	-0.077175
price_per_sqft	0.648649	0.494352	0.490266	0.623998	0.617716	0.477801
hoa_month	-0.210898	-0.229683	-0.187127	-0.224408	-0.214820	-0.215245
lat	-0.440943	-0.407198	-0.367912	-0.436983	-0.429446	-0.394447
long	-0.085858	0.033185	0.018411	-0.056451	-0.057514	0.036530
property_age	-0.009635	-0.058721	-0.037758	-0.001237	0.002500	-0.060223
school_rating	0.384551	0.520341	0.494630	0.435499	0.427206	0.514370
school_count	-0.196507	-0.275615	-0.270057	-0.206063	-0.201268	-0.273568
bart_count	-0.278580	-0.290867	-0.284661	-0.299027	-0.292656	-0.283708
median_income	0.479796	0.546868	0.546316	0.520984	0.514408	0.543510
hotness_score	-0.403704	-0.241393	-0.280435	-0.371385	-0.375008	-0.244618
supply_score	-0.345309	-0.107441	-0.201706	-0.279392	-0.288473	-0.123786
demand_score	-0.353043	-0.241754	-0.255160	-0.335887	-0.336982	-0.240048
malls_count	0.018225	-0.064539	-0.051074	-0.007075	-0.005790	-0.063246
university_count	0.136250	0.148016	0.137360	0.150935	0.148901	0.142359

Table 4: Dataset's comparison using Correlation and Applying conditional formatting

6.3 Features Engineering

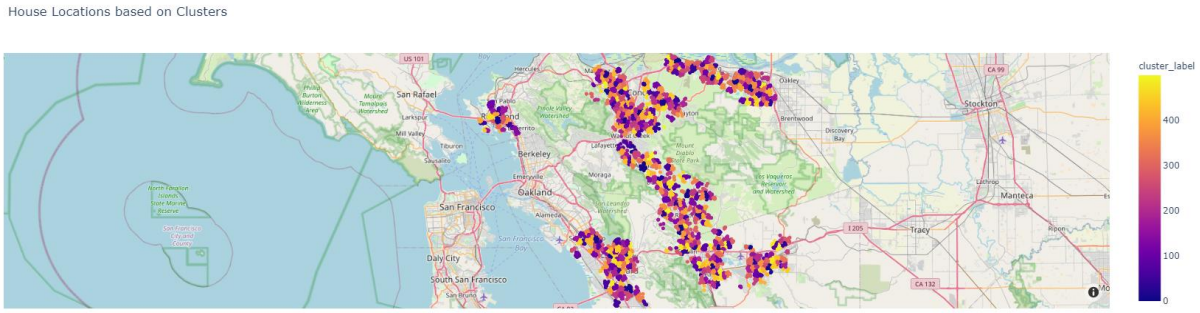
6.3.1 Introducing New Features

- Let's create New 3 categorical features (Boolean variables):
  - barts: 1 yes barts and 0 no barts
  - malls: 1 yes malls and 0 no malls
  - universities: 1 yes universities and 0 no Universities
- Beds_Baths_tradeoff (bedBath) = beds * baths
- AvgRoomSize = sqft / (beds + baths)

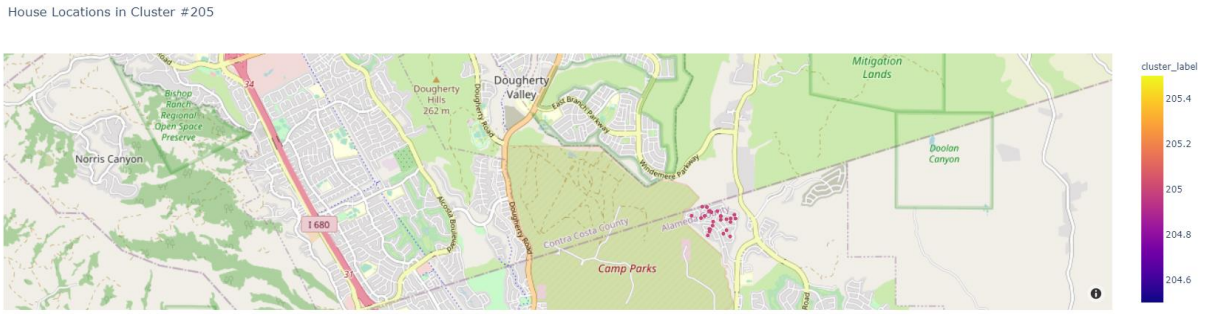
6.3.2 Unsupervised Machine Learning – Clustering using K-Means.

Currently, the lowest aggregation level in the DataFrame is zipcode, so we'll be utilizing Unsupervised Machine Learning by using Clustering (K-Means) to cluster all our houses based on Longitude and Latitude.

The Idea is to have clusters similar to a neighborhood and then we'll create a feature called **median_price_sqft_cluster** based on existing column called **price_per_sqft**.



Map 1: Houses in Clusters using K-Means



Map 2: Cluster # 205

As shown above, we can see that choosing cluster # 500 is the best, as illustrated in Map 2, Cluster# 205 is a good example because this is a new community in Dublin, CA called Wallis Ranch and K-Means managed to create the boundaries perfectly.

6.3.3 Calculating Distance between houses and Schools using Haversine Formula

The Haversine formula will help us to find all schools within 3 miles of every house and then we can calculate the average GreatSchool Ratings for all schools within 3 miles of every house in our database. Haversine formula is perhaps the first equation to consider when understanding how to calculate distances on a sphere. The word "Haversine" comes from the function:

$$\theta = \frac{d}{r}$$

Where:

- d is the distance between the two points along a great circle of the sphere (see spherical distance),
- r is the radius of the sphere.

The *haversine formula* allows the haversine of θ (that is, $\operatorname{hav}(\theta)$) to be computed directly from the latitude (represented by φ) and longitude (represented by λ) of the two points:

$$\operatorname{hav}(\theta) = \operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)$$

Where:

- φ_1, φ_2 are the latitude of point 1 and latitude of point 2 (in radians),
- λ_1, λ_2 are the longitude of point 1 and longitude of point 2 (in radians).

Finally, the haversine function $\operatorname{hav}(\theta)$, applied above to both the central angle θ and the differences in latitude and longitude, is

$$\operatorname{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) = \frac{1 - \cos(\theta)}{2}$$

The haversine function computes half a versine of the angle θ.

To solve for the distance d, apply the arc haversine (inverse haversine) to $h = \operatorname{hav}(\theta)$ or use the arcsine (inverse sine) function:

$$d = r\operatorname{archav}(h) = 2r\arcsin\left(\sqrt{h}\right)$$

or more explicitly:

$$\begin{aligned} d &= 2r\arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\operatorname{hav}(\lambda_2 - \lambda_1)}\right) \\ &= 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \end{aligned}$$

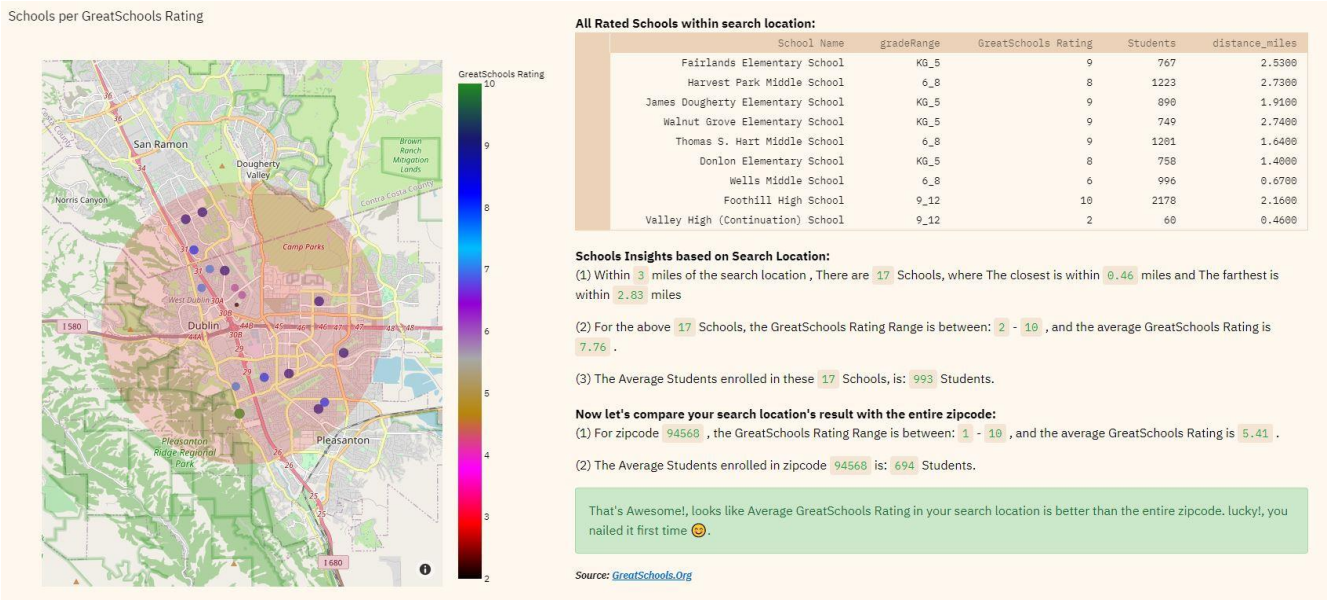


Figure 22: Menara App showing GreatSchools search using Haversine Formula

6.4 Final DataFrame Summary

Alright, as shown in Table 5, all our features are numerical, datatypes are correct and no missing data:

	unique_count	dtypes	na_count	na_%
lat	5865	float64	0	0.0
long	5267	float64	0	0.0
distance_miles	4737	float64	0	0.0
AvgRoomSize	2799	float64	0	0.0
lot_size	2654	int64	0	0.0
sqft	2179	int64	0	0.0
price	1439	int64	0	0.0
price_per_sqft	827	int64	0	0.0
gsRating	633	float64	0	0.0
cluster_label	500	int32	0	0.0
hoa_month	492	float64	0	0.0
median_price_sqft_cluster	362	float64	0	0.0
days_on_market	310	int64	0	0.0
property_age	121	int64	0	0.0
zipcode	31	int64	0	0.0
median_income	31	int64	0	0.0
hotness_score	31	float64	0	0.0
supply_score	31	float64	0	0.0
demand_score	31	float64	0	0.0
bedBath	20	int64	0	0.0
school_count	12	int64	0	0.0
school_rating	9	int64	0	0.0
beds	8	int64	0	0.0
baths	7	int64	0	0.0
university_count	3	int64	0	0.0
property_type	3	int64	0	0.0
malls	2	int64	0	0.0
universities	2	int64	0	0.0
barts	2	int64	0	0.0

Table 5: Final DataFrame Summary

6.5 Final Visualization of our Target Variable ‘price’

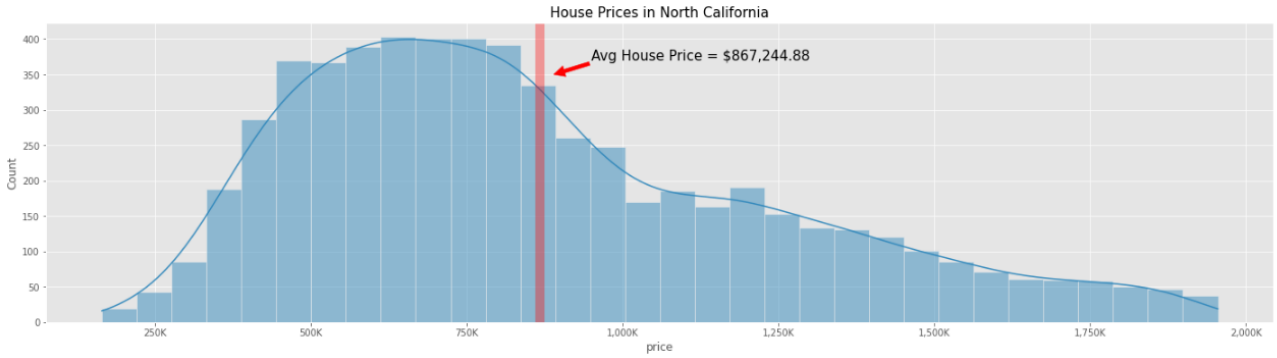


Figure 23: Final Vis of Target Variable 'price'

7. Machine Learning Modeling

7.1 Introduction

Here comes the really fun step: modeling! For this step, we'll be:

- Training multiple Regression algorithms.
- Apply hyperparameters tuning where applicable to ensure every algorithm will result in best prediction possible.
- Finally, evaluate these Models.

Regression Models:

- Dummy Regression (Baseline Model)
- Linear Regression
- Ridge Regression (L2 Regularization)
- Lasso Regression (L1 Regularization)
- Decision Trees
- Random Forests
- GB
- XGBoost
- Light GBM
- Artificial Neural networks (ANN).
- Stacking

7.2 Features Selection

Below features will be defined as our input features (X) against our Target feature 'price' (y)

'property_type', 'zipcode', 'beds', 'baths', 'sqft', 'lot_size', 'days_on_market', 'hoa_month', 'lat', 'long','property_age','school_rating','school_count','median_income','hotness_score', 'supply_score', 'demand_score', 'university_count', 'barts', 'malls', 'universities', 'bedBath', 'AvgRoomSize', 'cluster_label', 'median_price_sqft_cluster', 'gsRating', 'distance_miles'.

Also, we'll define only house & Neighborhood related features and then we'll compare results: 'property_type', 'beds', 'baths', 'sqft', 'zipcode', 'lot_size', 'days_on_market', 'hoa_month', 'lat', 'long', 'property_age', 'gsRating', 'median_income' and 'university_count', 'malls', 'median_price_sqft_cluster'.

7.3 Train/Test Split

We'll be splitting out data (X) into training and testing splits, without letting a model learn anything about the test split by having 75/25 Train/Test split.

7.4 Scaling and Transforming: Standardization

Standardization finds the mean of our data and centers the distribution around it, calculating the number of standard deviations away from the mean each point is. The number of standard deviations is then used as our new values. This centers the data around 0 but technically has no limit to the maximum and minimum values.

7.5 Dummy Regression (Baseline) and Linear Regression

7.5.1 Dummy Regressor

Dummy Regressor is a regressor that makes predictions using simple rules.
This regressor is useful as a simple baseline to compare with other (real) regressors.

The main role of strategy is to predict target values without any influence of the training data. There are namely four types of strategies that are used by the Dummy Regressor:

- Mean: This is the default strategy used by the Dummy Regressor. It always predicts the mean of the training target values.
- Median: This is used to predict the median of the training target values.
- Quantile: It is used to predict a particular quantile of training target values provided the quantile parameter is used along with it.
- Constant: This is generally used to predict a specific custom value that is provided and the constant parameter must be mentioned.

Dummy Regressor always predict the R-Squared as 0, since it is always predicting a constant without having an insight of the output. (In general, best R-Squared is 1 and Constant R-Squared is 0).

7.5.2 Linear Regression

Making a Linear Regression model: our first model. Sklearn has a LinearRegression() function built into the linear model module. We'll be using that to make our first regression model.

Figure 24: Linear Regression Coefficients

Linear Regression: intercept & Coefficients Summary:

- From above Linear Regression coefficients, we can see that Regularization is needed: As we know Linear regression minmizes a loss function by choosing a coefficient for each feature variable. If we allow these coefficients or parameters to be super large, we can get overfitting easily. so, it is common practice to alter the loss function so that it penalizes for large coefficients using Regularization. So next, we'll be using both Ridge Regression and Lasso Regression.
- Also, we can see that there're few features showing very low Coefficients i.e., 'malls', 'university_count', 'property_age', so if we drop them definitely, we'll improve our predictions.

7.5.3 Dummy Regression and Linear regression performance summary

Dummy Regression and Linear regression performance summary with Features tuning:

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Dummy Regression - (all features)	0 %	0 %	301148.2119	376306.4570	0 %
Linear Regression - Baseline (all features)	87.3432 %	87.1112 %	95909.6995	133875.4717	87.3781 %
Linear Regression - Tuned(High Important Features)	87.0594 %	86.9199 %	96522.4628	135367.8835	87.0805 %

Table 6: Dummy & Linear Regression Performance Summary

As we can see in Table 6, for the Linear Regression, the value of **root mean squared error (RMSE) is 133,875.4717**, which is slightly larger than 15% of the mean value of the Sales Price i.e., $867,244.88.

7.5.4 Linear Regression Actual vs Prediction Visualization

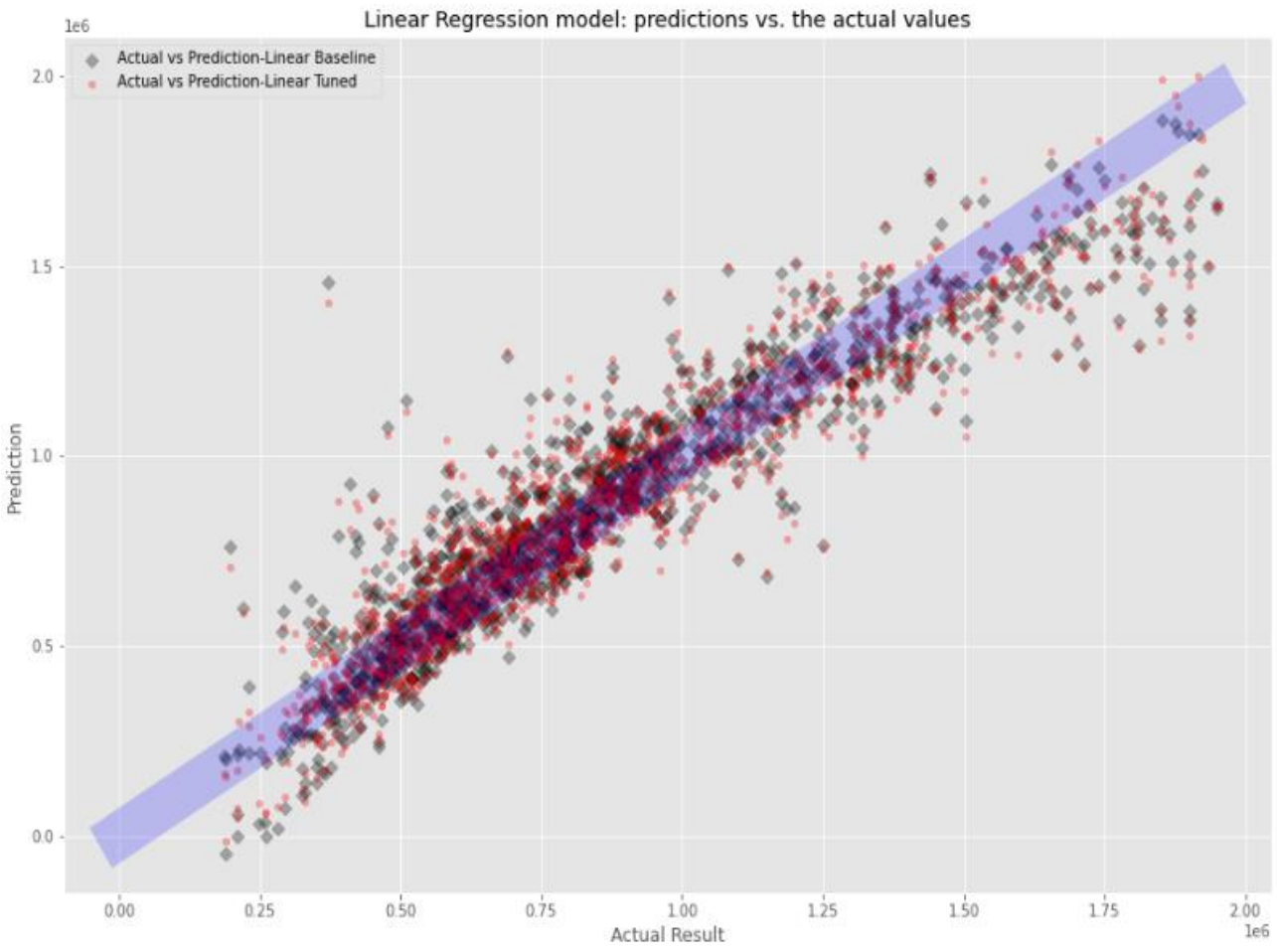


Figure 25: Linear Regression Actual vs Prediction Visualization

7.6 Ridge Regression (L2 Regularization)

The first type of regularized regression that we'll train is called Ridge Regression. in which our loss function is the standard OLS loss function plus the squared value of each coefficient multiplied by some constant alpha. Thus, when minimizing the loss function to fit to our data, models are penalized for coefficients with a large magnitude: large positive and large negative coefficients, that is!

Note that alpha is a parameter we need to choose in order to fit and predict. Essentially, we can select the alpha for which our model performs best. Picking alpha for ridge regression is similar to picking k in KNN. Before using Ridge regressor it is necessary to scale the inputs, because this model is sensitive to scaling of inputs. So, performing the scaling through sklearn's StandardScalar will be beneficial.

This alpha, which you may also see called lambda in the wild, can be thought of as a parameter that controls model complexity:

- When alpha is equal to zero, we get back OLS (Linear Regression). Large coefficients in this case are not penalized and the overfitting problem is not accounted for.
- A very high alpha means that large coefficients are significantly penalized, which can lead to a model that is too simple and ends up underfitting the data.

7.6.1 Ridge Regression Tuning Summary

Ridge Regression Model Tunning Summary:

- Keeping the high important features.
- Using Regularization to determine best Alpha=15.20975162616059 and solver=sparse_cg.
- Variance Score has degraded from 87.3653 % (Ridge - baseline) to 87.0970 %

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	87.3345 %	87.1024 %	95239.0802	133921.3033	87.3653 %
Tuned(High Important Features)	87.0600 %	86.9205 %	96520.6475	135365.0654	87.0810 %
Tuned(High Important Features) + Regularization {'alpha': 15.20975162616059, 'solver': 'sparse_cg'}	87.0763 %	86.9369 %	96489.2726	135279.9288	87.0970 %

Table 7: Ridge Regression Tuning Summary

As we can see in Table 7, for the Ridge Regression, the value of **root mean squared error (RMSE) is 135,365.0654**, which is slightly larger than 15% of the mean value of the Sales Price i.e., $867,244.88.

7.6.2 Ridge Regression Actual vs Prediction Visualization

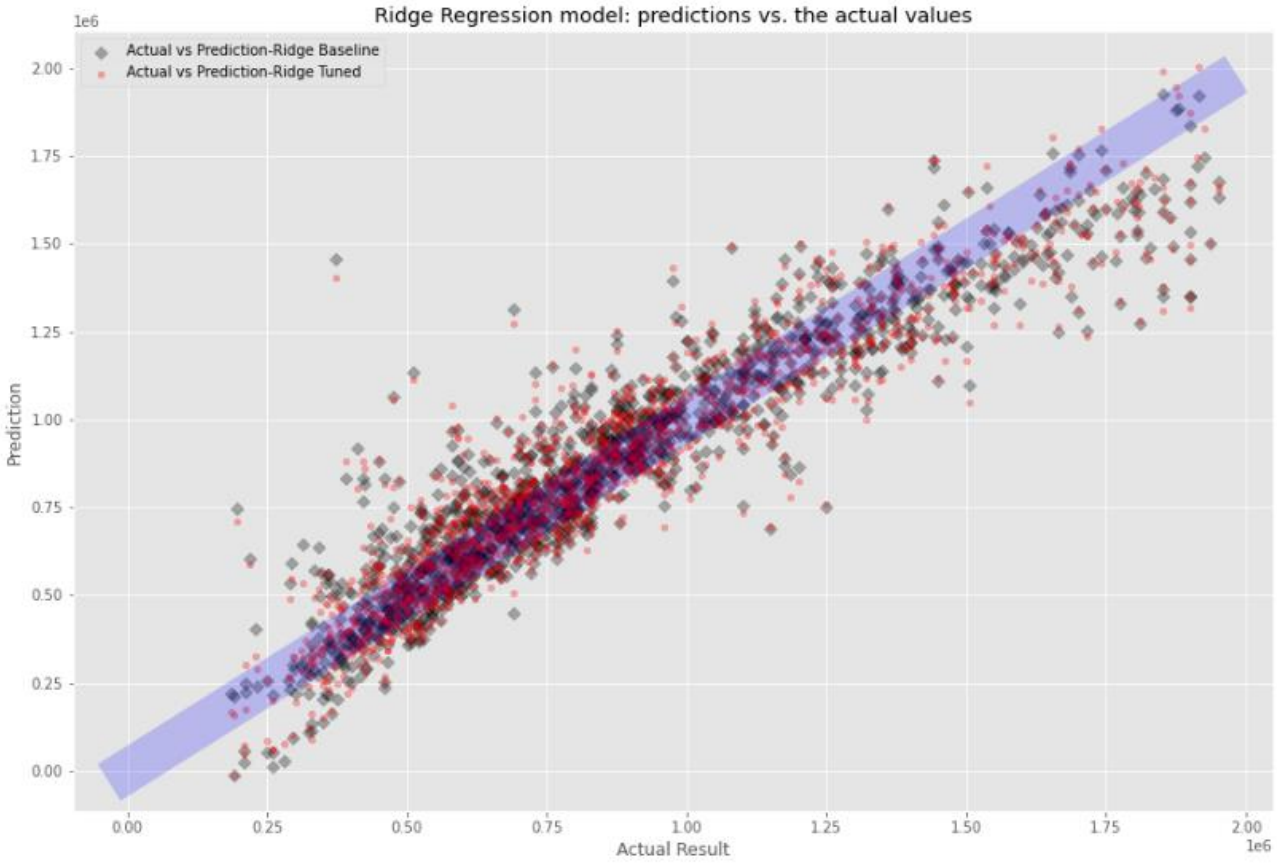


Figure 26: Ridge Regression Actual vs Prediction Visualization

7.7 Lasso Regression (L1 Regularization)

There is another type of regularized regression called lasso regression. In which our loss function is the standard OLS loss function plus the absolute value of each coefficient multiplied by some constant alpha. The method of performing lasso regression in scikit-learn mirrors ridge regression.

7.7.1 Lasso Regression Tuning Summary

Lasso Regression Model Tunning Summary:

- Keeping the high important features.
- Using Regularization to determine best Alpha=10.
- Variance Score has degraded from 87.30 % (Lasso - baseline) to 87.0806 %

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	87.27 %	87.0356 %	96077.5947	134267.8161	87.30 %
Tuned(High Important Features)	87.0595 %	86.9199 %	96522.2991	135367.8360	87.0805 %
Tuned(High Important Features) + Regularization {'alpha': 10'}	87.0595 %	86.9200 %	96520.8254	135367.3967	87.0806 %

Table 8: Lasso Regression Tuning Summary

As we can see in Table 8, for the Lasso Regression, the value of **root mean squared error (RMSE) is 135,367.8360**, which is slightly larger than 15% of the mean value of the Sales Price i.e., $867,244.88

7.7.2 Lasso Regression Actual vs Prediction Visualization

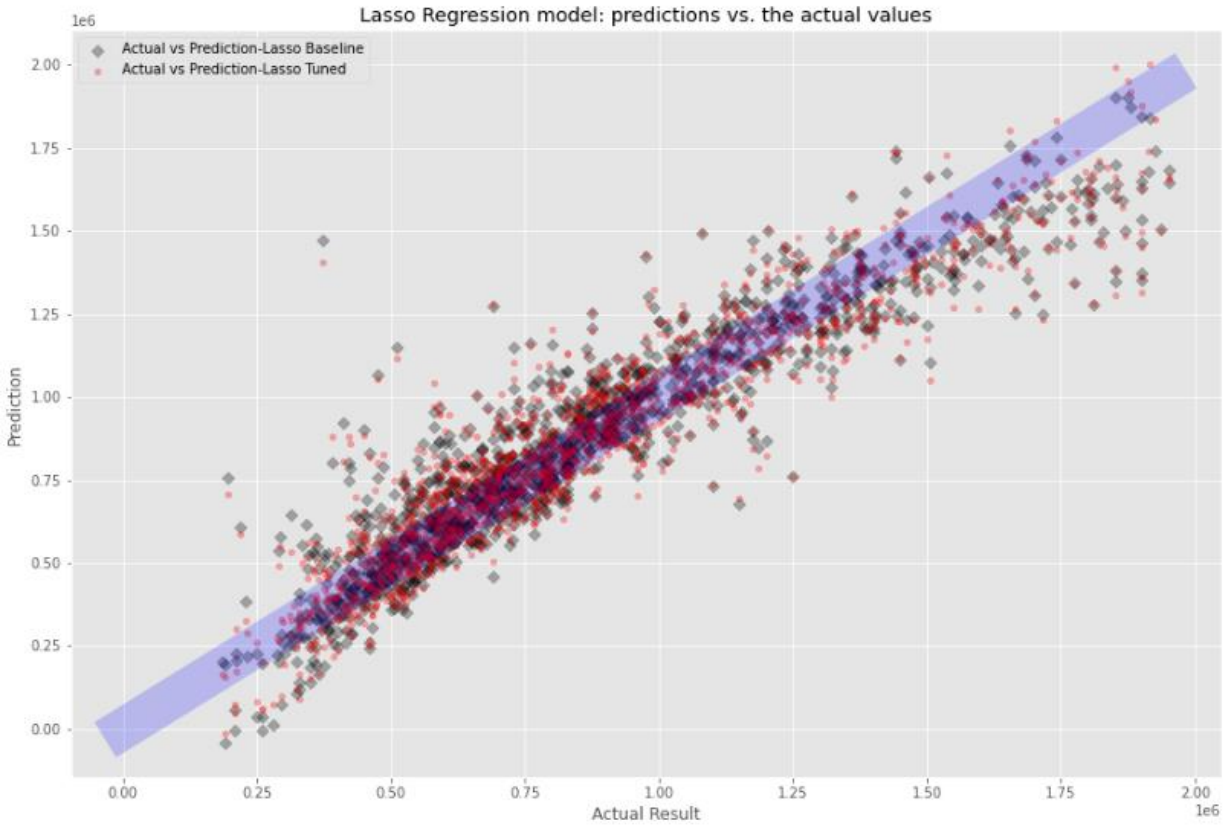


Figure 27: Lasso Regression Actual vs Prediction Visualization

7.8 Decision Tree Regression

Decision Tree is a decision-making tool that uses a flowchart-like tree structure or is a model of decisions and all of their possible results, including outcomes, input costs and utility.

Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables.

The branches/edges represent the result of the node and the nodes have either:

- Conditions (Decision Nodes).
- Result (End Nodes)

7.8.1 Decision Tree Regression: Features Importance

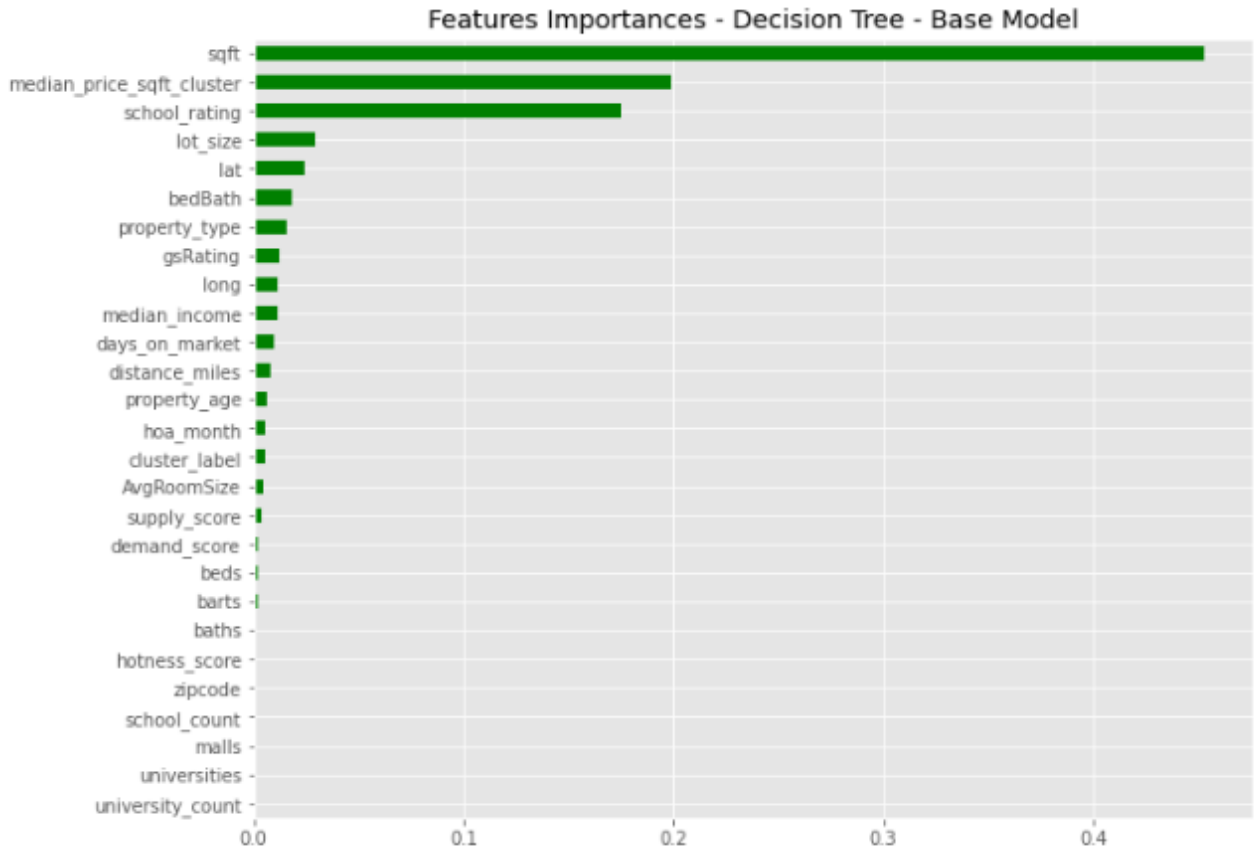


Figure 28: Decision Tree Regression: Features Importance

7.8.2 Decision Tree Regression Tuning Summary

As we can see below, we managed to improve our Decision Tree Regression Model by:

- keeping only the most important features: and
- Hyperparameters Tuning using RandomizedSearchCV to determine best:
  - {'min_samples_split': 28, 'min_samples_leaf': 11, 'max_features': 'auto'} when considering all features.
  - {'min_samples_split': 28, 'min_samples_leaf': 16, 'max_features': 'auto', 'max_depth': 15} when considering only the most important features.
- Variance Score has improved from 84.5661 % (Decision Tree - baseline) to 87.4864 %.

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	84.5454 %	84.3787 %	99336.2012	147934.2861	84.5661 %
Baseline (Keeping high important Features)	85.1269 %	84.9665 %	96266.4337	145124.2649	85.1990 %
All Features + RandomizedSearchCV {'min_samples_split': 28, 'min_samples_leaf': 11, 'max_features': 'auto'}	87.1429 %	86.9073 %	92447.3932	134930.5062	87.2065 %
High Important Features + RandomizedSearchCV {'min_samples_split': 28, 'min_samples_leaf': 16, 'max_features': 'auto', 'max_depth': 15}	87.4464 %	87.3110 %	92510.0434	133328.6903	87.4864 %

Table 9: Decision Tree Regression Tuning Summary

As we can see in Table 9, for the Decision Tree Regression, the value of **root mean squared error (RMSE) is 133,328.6903** which is slightly larger than 15% of the mean value of the Sales Price i.e., $867,244.88.

7.8.3 Decision Tree Regression Actual vs Prediction Visualization

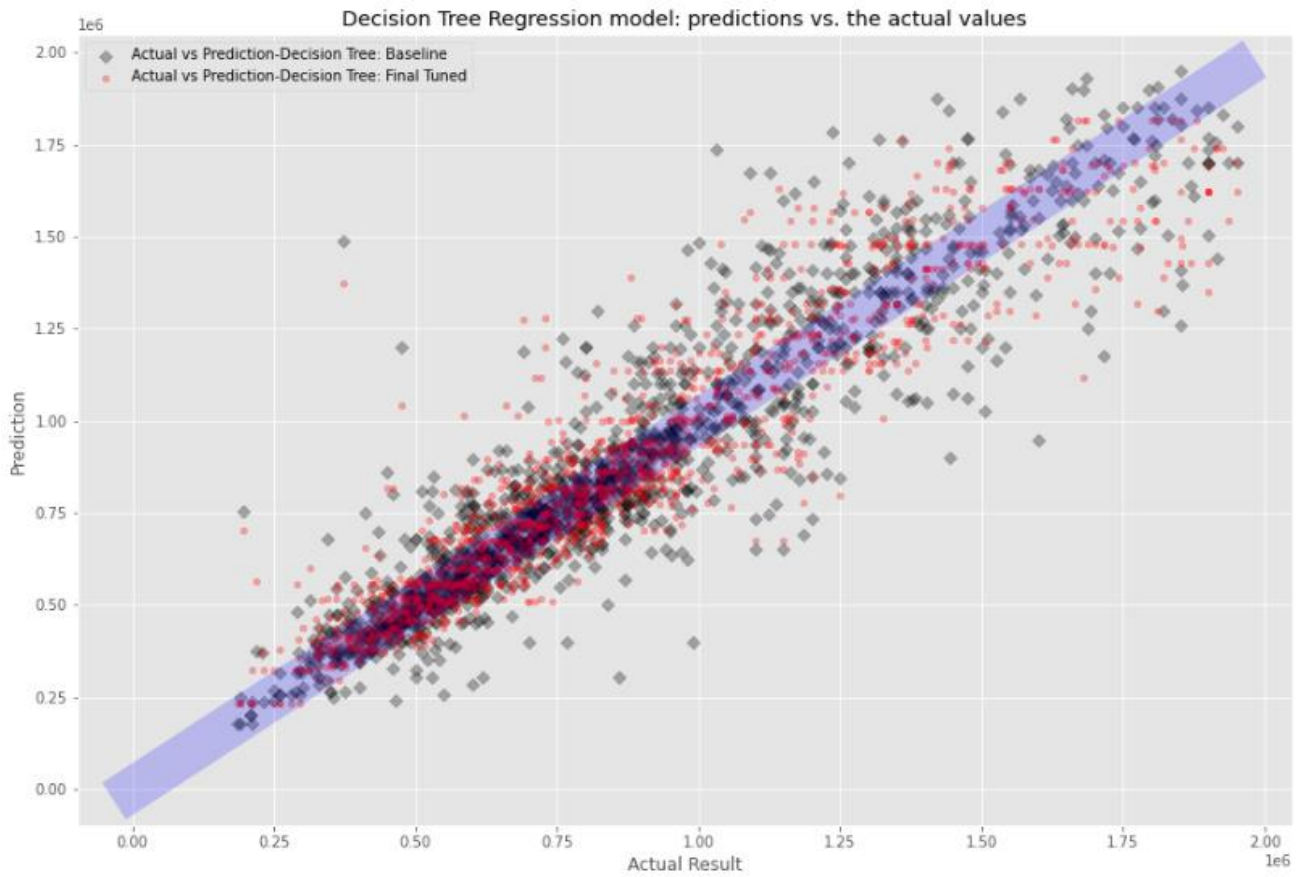


Figure 29: Decision Tree Regression Actual vs Prediction Visualization

7.9 Random Forests Regression

Random Forest is an ensemble technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called Bootstrap and Aggregation, commonly known as bagging. The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees. Random Forest has multiple decision trees as base learning models. We randomly perform row sampling and feature sampling from the dataset forming sample datasets for every model. This part is called Bootstrap.

A RandomForestRegressor has all the hyperparameters of:

- **DecisionTreeRegressor** (to control how trees are grown).
- **BaggingRegressor** to control the ensemble itself.

The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features. This results in a greater tree diversity, which (once again) trades a higher bias for a lower variance, generally yielding an overall better model.

Further Diversity with Random Forests:

- Random Forests is an ensemble method that uses a decision tree as a base estimator.
- Each estimator is trained on a different bootstrap sample having the same size as the training set.
- RF introduces further randomization than bagging when training each of the base estimators.
- When each tree is trained, only d features can be sampled at each node without replacement, where d is a number smaller than the total number of features.

7.9.1 Random Forests Regression: Features Importance

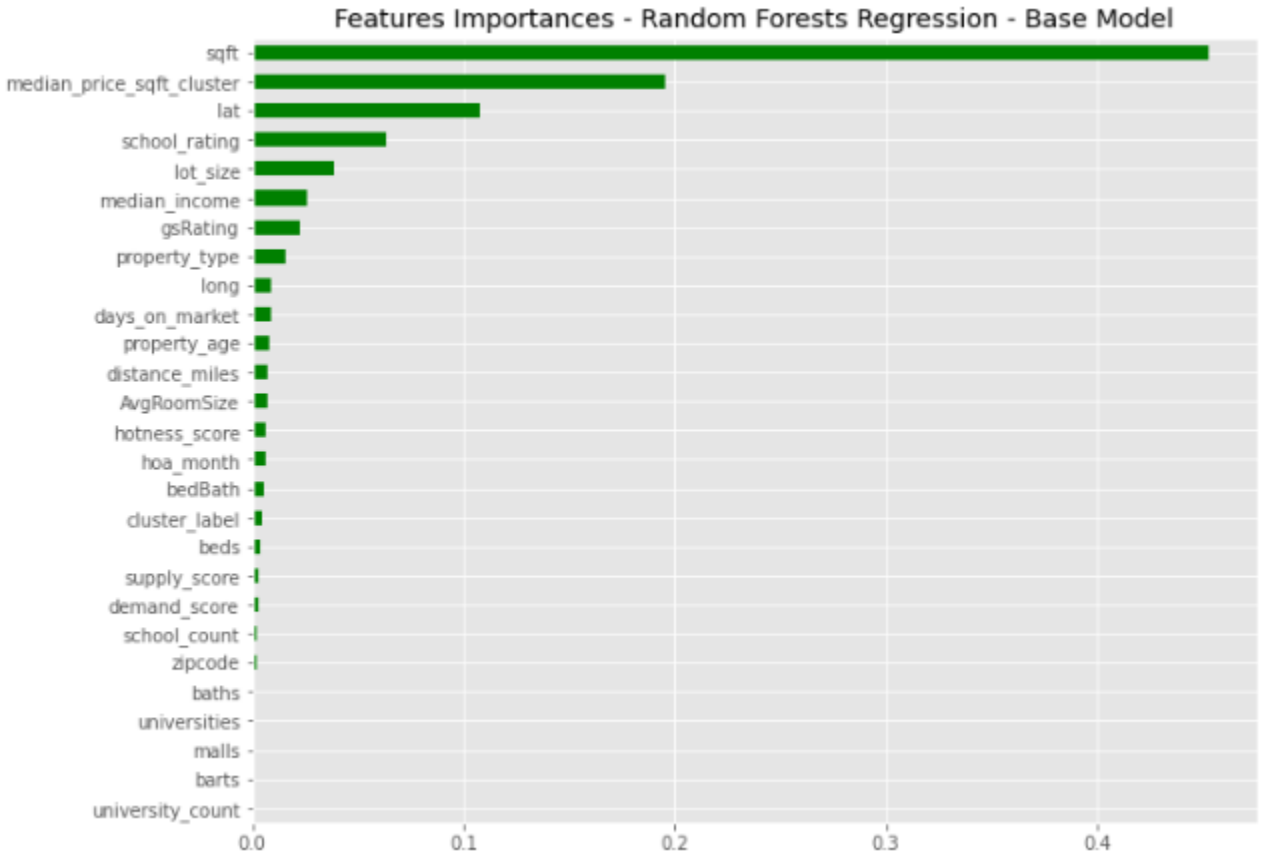


Figure 30: Random Forests Regression: Features Importance

7.9.2 Random Forests Regression Tuning Summary

As we can see below, we managed to improve our Random Forests Regression Model by:

- keeping only the high important features: and
- Hyperparameters Tuning using RandomizedSearchCV to determine best:
  - {'n_estimators': 300, 'max_features': 'sqrt', 'max_depth': 20} when considering all features.
  - {'n_estimators': 600, 'min_samples_split': 6, 'max_features': 'log2'} when considering only the high important features
- Variance Score has improved from 91.4562 % (Random Forests - baseline) to 92.2170 %.

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	91.4367 %	91.2797 %	71881.2010	110118.6979	91.4562 %
Baseline (Keeping high important Features)	91.4910 %	91.3993 %	72011.8407	109768.5543	91.5134 %
All Features + RandomizedSearchCV {'n_estimators': 300, 'max_features': 'sqrt', 'max_depth': 20}	91.9673 %	91.8201 %	68595.6250	106652.1579	91.9863 %
High Important Features + RandomizedSearchCV {'n_estimators': 600, 'min_samples_split': 6, 'max_features': 'log2'}	92.1982 %	92.1141 %	67886.0897	105107.8895	92.2170 %

Table 10: Random Forests Regression Tuning Summary

As we can see in Table 10, for the Random Forests Regression, the value of **root mean squared error (RMSE) is 105,107.8895** which is slightly larger than 12% of the mean value of the Sales Price i.e., $867,244.88.

7.9.3 Random Forests Regression Actual vs Prediction Visualization

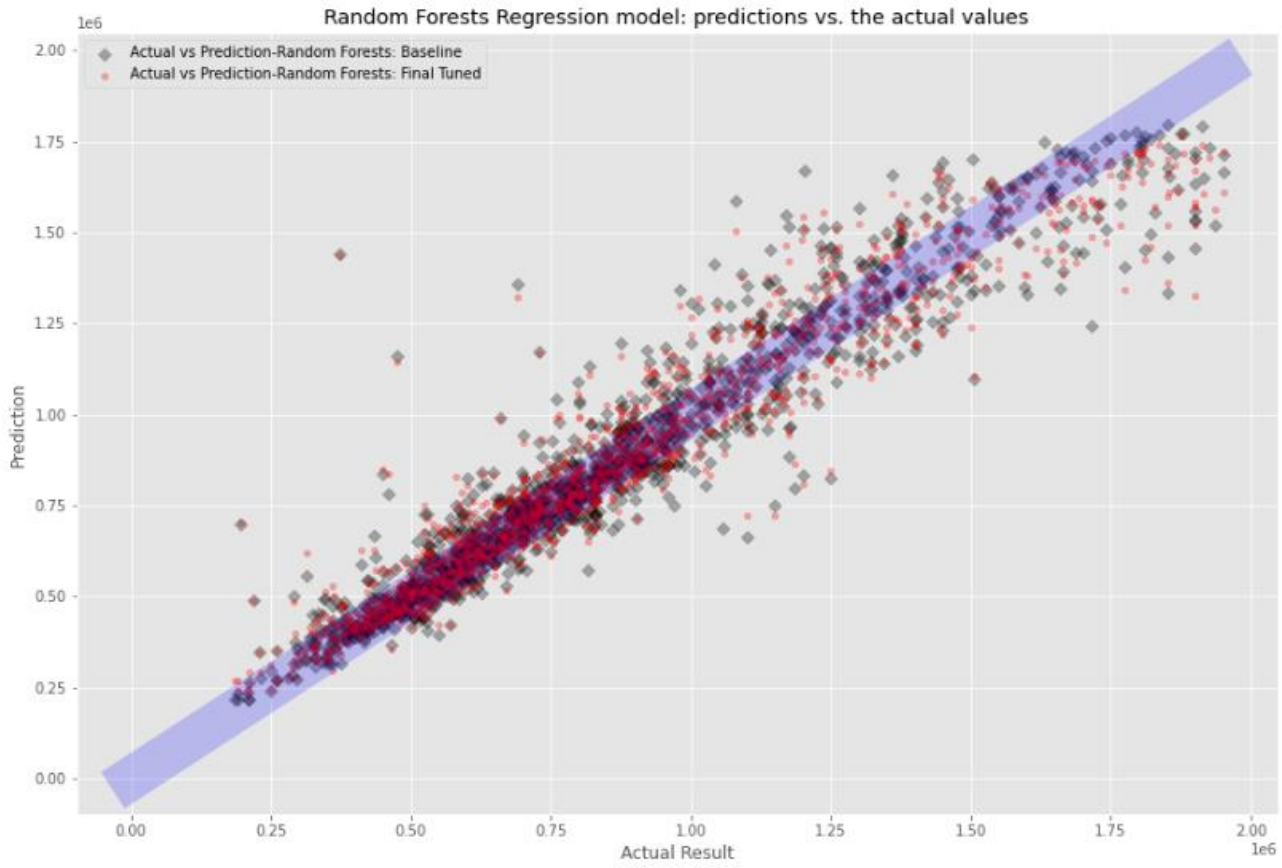


Figure 31: Random Forests Regression Actual vs Prediction Visualization

7.10 Gradient Boosting Regression

Gradient Boosting is an ensemble technique that takes an iterative approach to combining weak learners to create a strong learner by focusing on mistakes of prior iterations.

Trade-Offs of Gradient Boosting:

- **Pros**:
  - Extremely powerful on both classification and regression tasks.
  - More accurate predictions compared to random forests.
  - Accepts various types of inputs
  - Outputs feature importance

- **Cons**:
  - Longer to train (Cannot parallelize).
  - May overfit if too many trees are used (n_estimators)
  - More difficult to properly tune (Requires careful tuning of hyperparameters)
  - Sensitive to outliers

7.10.1 Gradient Boosting Regression: Features Importance

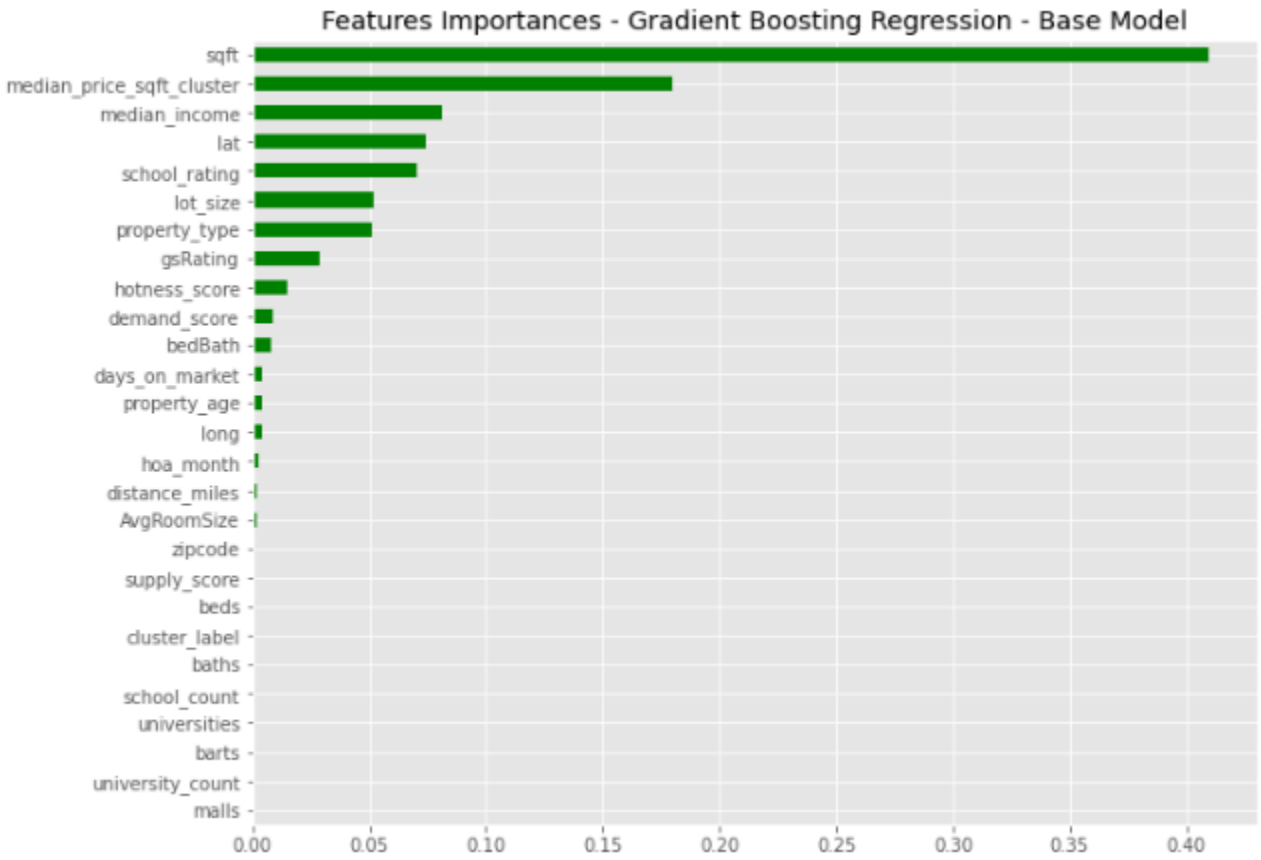


Figure 32: Gradient Boosting Regression: Features Importance

7.10.2 Gradient Boosting Regression Tuning Summary

As we can see below, we managed to improve our Gradient Boosting Regression Model by:

- keeping only the most important features: and
- Hyperparameters Tuning using RandomizedSearchCV to determine best:

- {'n_estimators': 1500, 'max_features': 'sqrt', 'max_depth': 8, 'learning_rate': 0.01} when considering all features.
- {'n_estimators': 800, 'max_features': 'log2', 'max_depth': 6, 'learning_rate': 0.05} when considering the high important features.

- Variance Score has improved from 91.1607 % (Gradient Boosting - baseline) to 92.4863 %.

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	91.1470 %	90.9847 %	77082.4736	111965.8849	91.1607 %
Baseline (Keeping high important Features)	90.8521 %	90.7535 %	78212.3780	113814.8578	90.8621 %
All Features + RandomizedSearchCV {'n_estimators': 1500, 'max_features': 'sqrt', 'max_depth': 8, 'learning_rate': 0.01}	92.4545 %	92.3162 %	66219.4133	103367.2310	92.4616 %
High Important Features + RandomizedSearchCV {'n_estimators': 800, 'max_features': 'log2', 'max_depth': 6, 'learning_rate': 0.05}	92.4799 %	92.3988 %	66910.7055	103193.3274	92.4863 %

Table 11: Gradient Boosting Regression Tuning Summary

As we can see in Table 11, for the Gradient Boosting Regression, the value of **root mean squared error (RMSE) is 103,193.3274** which is slightly less than 12% of the mean value of the Sales Price i.e., $867,244.88.

Learning rate Summary:

- The lower the learning rate, the slower the model learns.
- The advantage of slower learning rate is that the model becomes more robust and generalized (In statistical learning, models that learn slowly perform better).
- Learning slowly comes at a cost. It takes more time to train the model which brings us to the other significant hyperparameter.

n_estimator Summary:

- n_estimator is the number of trees used in the model.
- If the learning rate is low, we need more trees to train the model.
- We need to be very careful at selecting the number of trees. It creates a high risk of overfitting to use too many trees.

Random Forests vs Gradient Boosting:

- One key difference between random forests and gradient boosting decision trees is the number of trees used in the model.
  - Increasing the number of trees in random forests does not cause overfitting.
  - The number of trees in gradient boosting decision trees is very critical in terms of overfitting. Adding too many trees will cause overfitting so it is important to stop adding trees at some point.

7.10.3 Gradient Boosting Regression Actual vs Prediction Visualization

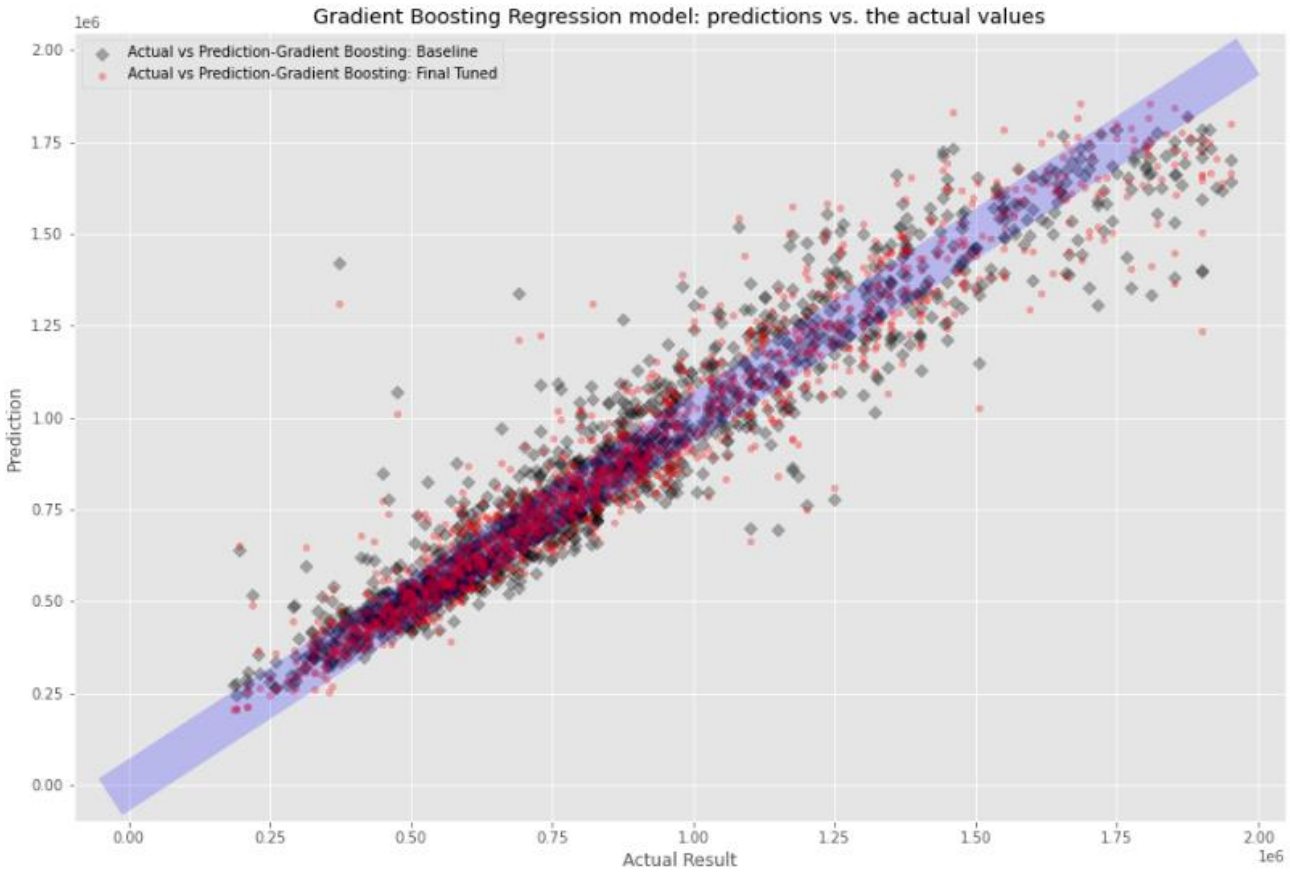


Figure 33: Gradient Boosting Regression Actual vs Prediction Visualization

7.11 XGBoost Regression

XGBoost an incredibly popular machine learning library for good reason. It was developed originally as a C++ command-line application. After winning a popular machine learning competition, the package started being adopted within the ML community. As a result, bindings, or functions that tapped into the core C++ code, started appearing in a variety of other languages, including Python, R, Scala, and Julia. We will cover the Python API in this course.

What makes XGBoost so popular?

- **Its speed**: Because the core XGBoost algorithm is parallelizable:
  - It can harness all of the processing power of modern multi-core computers.
  - It is parallelizable onto GPU's and across networks of computers, making it feasible to train models on very large datasets on the order of hundreds of millions of training examples.

- **It's performance**:
  - It consistently outperforms almost all other single-algorithm methods in machine learning competitions and has been shown to achieve state-of-the-art performance on a variety of benchmark machine learning datasets.

7.11.1 XGBoost Regression: Features Importance

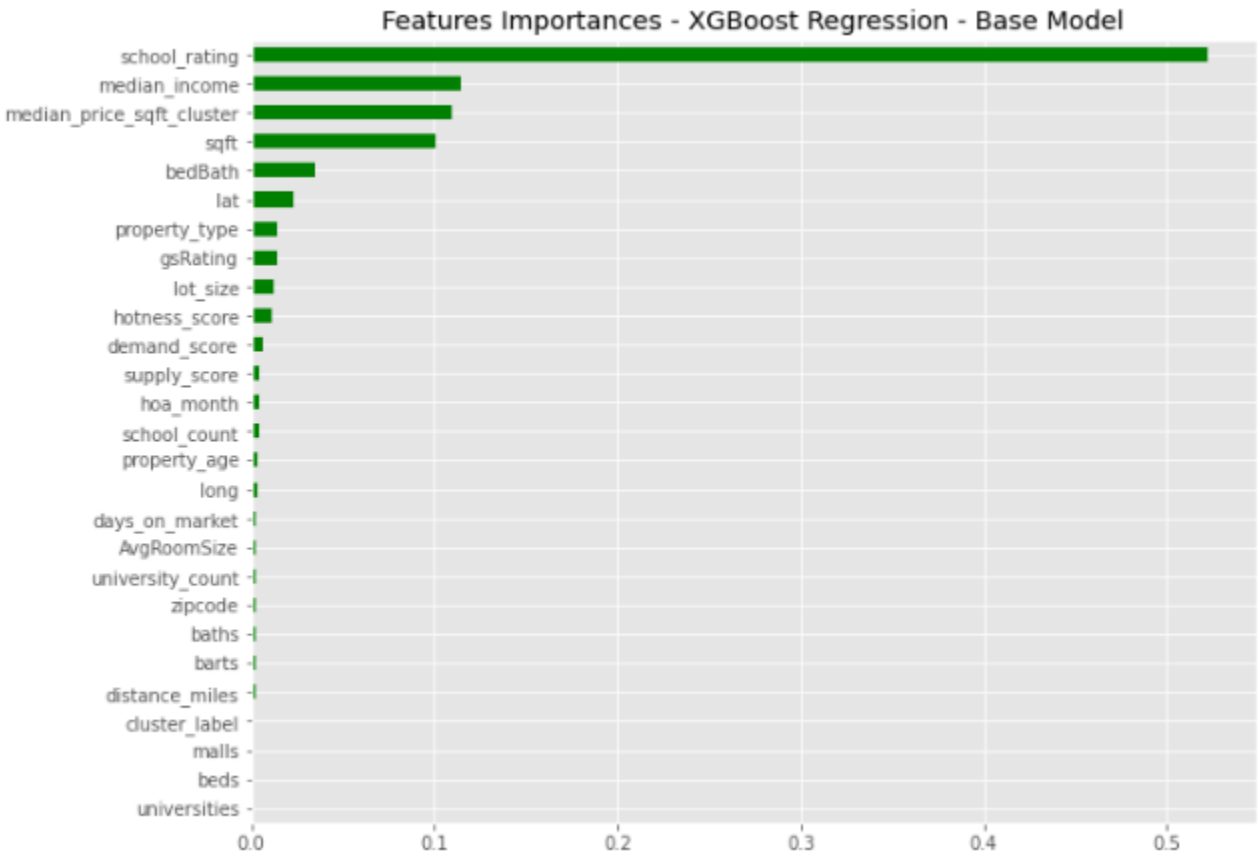


Figure 34: XGBoost Regression: Features Importance

7.11.2 XGBoost Regression: Tuning Summary

As we can see below, we managed to improve our XGBoost Regression Model by:

- keeping only the high important features: and
- Hyperparameters Tuning using RandomizedSearchCV to determine best:
  - {'n_estimators': 900, 'min_child_weight': 11, 'max_depth': 10, 'learning_rate': 0.01, 'gamma': 0.3, 'colsample_bytree': 0.4} when considering all features.
  - {'n_estimators': 600, 'min_child_weight': 12, 'max_depth': 16, 'learning_rate': 0.01, 'gamma': 0.3, 'colsample_bytree': 0.6} when considering only the high important features.
- Variance Score has improved from 90.9428 % (XGBoost - baseline) to 92.5786 %.

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	90.9334 %	90.7672 %	74685.8334	113308.0568	90.9428 %
Baseline (Keeping high important Features)	91.5418 %	91.4506 %	72408.9447	109440.3965	91.5428 %
All Features + RandomizedSearchCV {'n_estimators': 900, 'min_child_weight': 11, 'max_depth': 10, 'learning_rate': 0.01, 'gamma': 0.3, 'colsample_bytree': 0.4}	92.5617 %	92.3162 %	66304.8201	102630.7106	92.5737 %
High Important Features + RandomizedSearchCV {'n_estimators': 600, 'min_child_weight': 12, 'max_depth': 16, 'learning_rate': 0.01, 'gamma': 0.3, 'colsample_bytree': 0.6}	92.5777 %	92.4977 %	67043.7283	102520.0032	92.5786 %

Table 12: XGBoost Regression: Tuning Summary

As we can see in Table 12, for the XGBoost Regression, the value of **root mean squared error (RMSE) is 102,520.0032** which is slightly less than 12% of the mean value of the Sales Price i.e., $867,244.88.

7.11.3 XGBoost Regression Actual vs Prediction Visualization

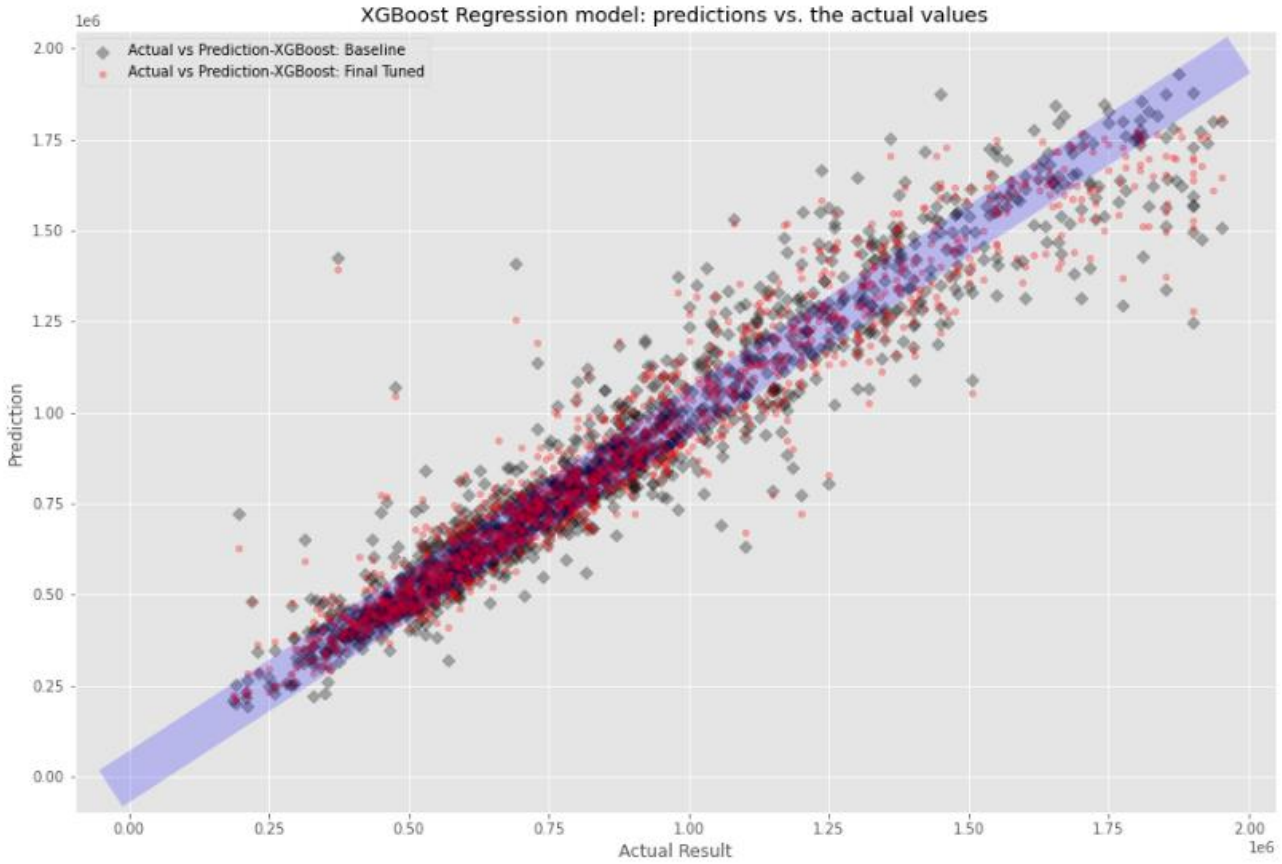


Figure 35: XGBoost Regression Actual vs Prediction Visualization

7.12 LightGBM Regression

LightGBM (Light Gradient Boosting Machine) is a gradient boosting framework based on decision trees to increases the efficiency of the model and reduces memory usage.
It uses two novel techniques: ***Gradient-based One Side Sampling and Exclusive Feature Bundling (EFB)*** which fulfills the limitations of histogram-based algorithm that is primarily used in all GBDT (Gradient Boosting Decision Tree) frameworks. The two techniques of GOSS and EFB described below form the characteristics of LightGBM Algorithm. They comprise together to make the model work efficiently and provide it a cutting edge over other GBDT frameworks.

Light GBM grows tree vertically while other algorithm grows trees horizontally meaning that Light GBM grows tree leaf-wise while another algorithm grows level-wise. It will choose the leaf with max delta loss to grow. When growing the same leaf, Leaf-wise algorithm can reduce more loss than a level-wise algorithm.

The size of data is increasing day by day and it is becoming difficult for traditional data science algorithms to give faster results. Light GBM is prefixed as 'Light' because of its high speed. Light GBM can handle the large size of data and takes lower memory to run. Another reason of why Light GBM is popular is because it focuses on accuracy of results. LGBM also supports GPU learning and thus data scientists are widely using LGBM for data science application development.

7.12.1 LightGBM Regression: Features Importance

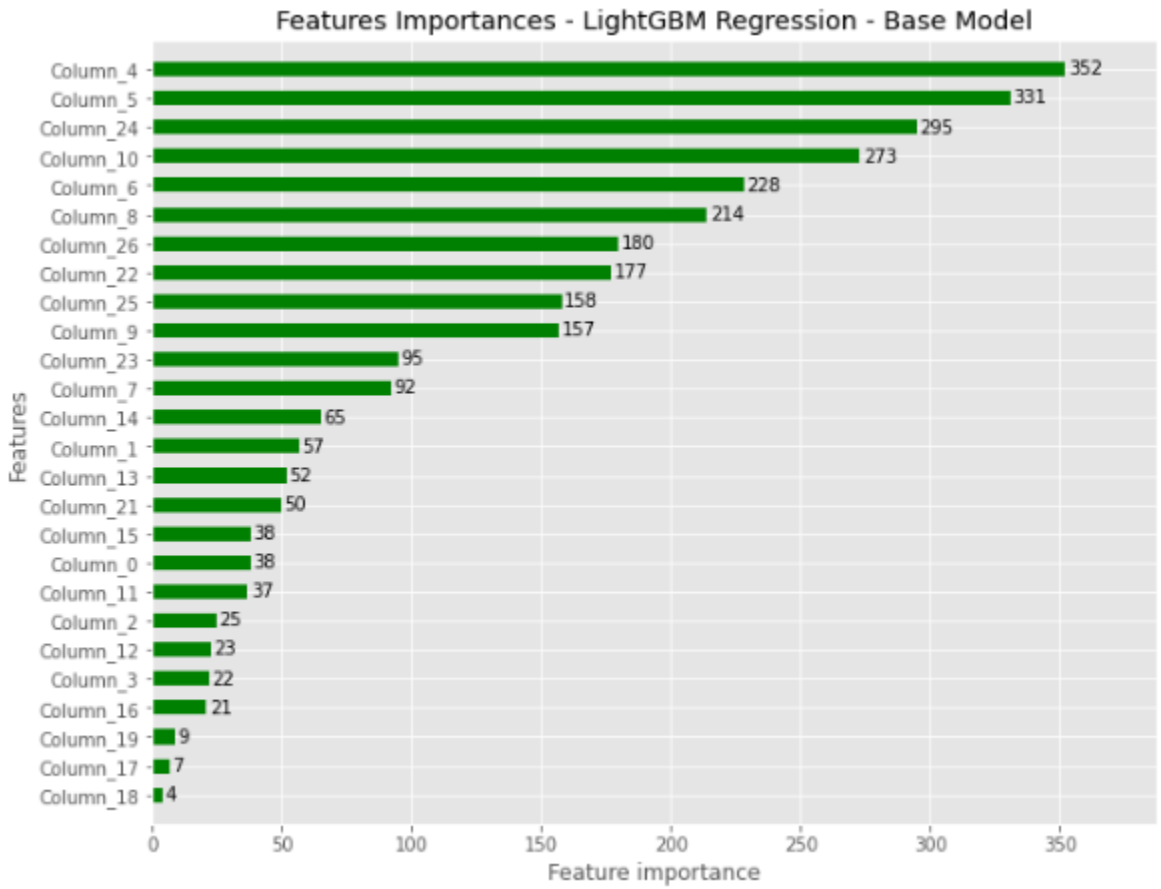


Figure 36: LightGBM Regression: Features Importance

7.12.2 LightGBM Regression Tuning Summary

- As we can see below, we managed to improve our LightGBM Regression Model by:
- keeping only the high important features: and
- Hyperparameters Tuning using RandomizedSearchCV to determine best:
  - {'reg_lambda': 0.4, 'reg_alpha': 0.6, 'n_estimators': 1700, 'learning_rate': 0.01, 'colsample_bytree': 0.6, 'max_depth':15, 'num_leaves'= 32767} when considering all features.

- {'reg_lambda': 0.4, 'reg_alpha': 0.8, 'n_estimators': 1000, 'learning_rate': 0.01, 'colsample_bytree': 0.6, 'max_depth':15, 'num_leaves'= 32767} when considering only the high important features.

- Variance Score has improved from 91.9935 % (LightGBM - baseline) to 92.6406 %.

Features Selection	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline (all features)	91.9853 %	91.8384 %	70644.4765	106532.6116	91.9935 %
Baseline (Keeping high important Features)	92.2960 %	92.2129 %	70316.6721	104447.5214	92.3067 %
All Features + RandomizedSearchCV	92.5526 %	92.4160 %	66041.6874	102693.4526	92.5621 %
High Important Features + RandomizedSearchCV	92.6304 %	92.5510 %	66235.3892	102155.1713	92.6406 %

Table 13: LightGBM Regression Tuning Summary

As we can see in Table 13, for the LightGBM Regression, the value of **root mean squared error (RMSE) is 102,155.1713** which is slightly less than 12% of the mean value of the Sales Price i.e., $867,244.88 and so far, this model is the best one we trained.

7.12.3 LightGBM Regression Actual vs Prediction Visualization

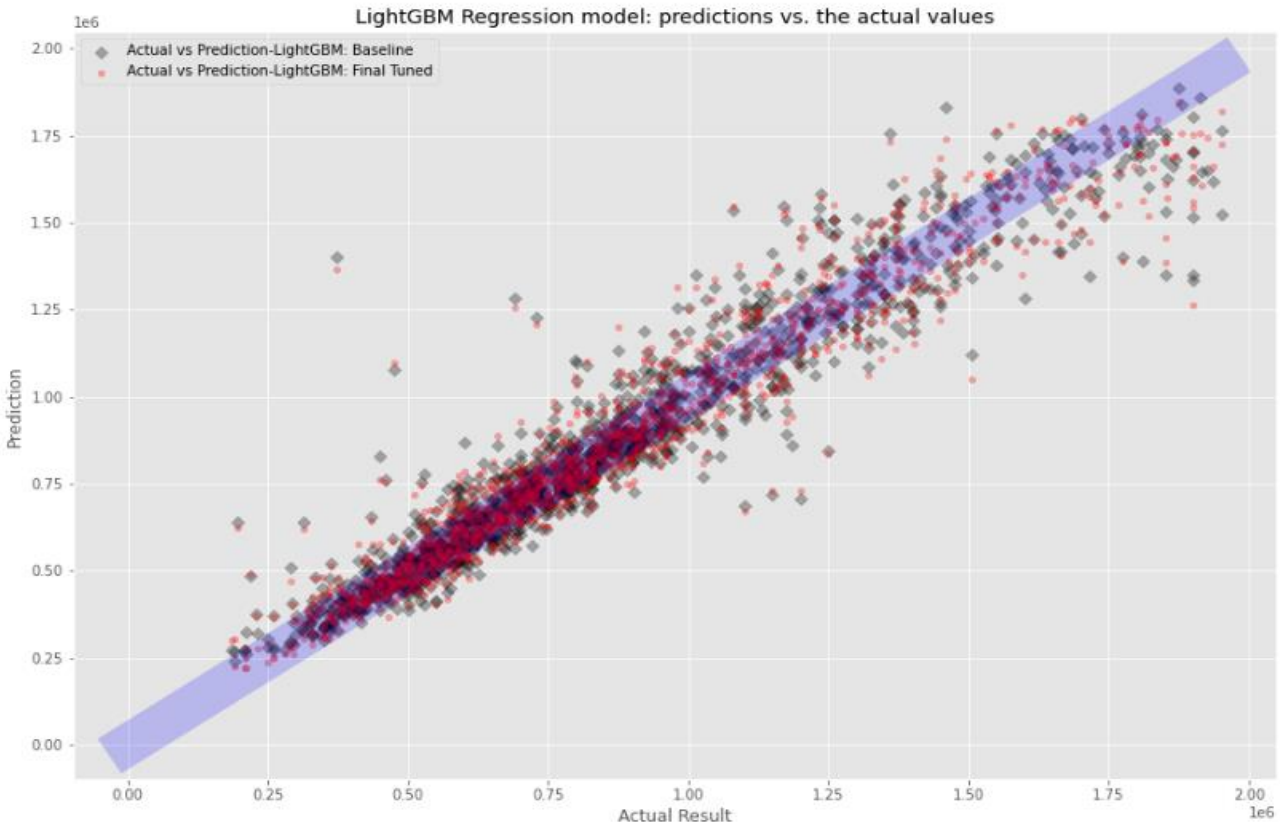


Figure 37: LightGBM Regression Actual vs Prediction Visualization

7.13 Artificial Neural Networks (ANN) using keras and Tensorflow

Neural networks consist of simple input/output units called neurons (inspired by neurons of the human brain). These input/output units are interconnected and each connection has a weight associated with it. Neural networks are flexible and can be used for both classification and regression. In this article, we will see how neural networks can be applied to regression problems.
Regression helps in establishing a relationship between a dependent variable and one or more independent variables. Regression models work well only when the regression equation is a good fit for the data. Most regression models will not fit the data perfectly. Although neural networks are complex and computationally expensive, they are flexible and can dynamically pick the best type of regression, and if that is not enough, ***hidden layers*** can be added to improve prediction.

7.13.1 Artificial Neural Network Regression Tuning Summary

As we can see below, we managed to improve our Neural Network Regression Model by:

- Making the Neural Network deeper by using more hidden Layers, and
- Making the Neural Network wider by using more neurons
  - NN-Model 1 we used 1 Hidden layer with 16 Neurons.
  - NN-Model 2 we used 3 Hidden layers with 16 Neurons.
  - NN-Model 3 we used 5 Hidden layers with 35 Neurons.
- **Variance Score has improved from 67.0386 % (Neural Network - baseline) to 91.1308 %.**

Neural Network Architecture	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Baseline: NN-Model 1 we used 1 Hidden layer with 16 Neurons	65.2787 %	64.9043 %	169664.7498	221736.5273	67.0386 %
RandomizedSearchCV{'epochs': 600, 'batch_size': 32}: NN-Model 1 we used 1 Hidden layer with 16 Neurons	89.0462 %	88.9281 %	86728.7935	124543.6689	89.0688 %
RandomizedSearchCV{'epochs': 600, 'batch_size': 32, 'learning_rate': 0.01}: NN-Model 1 we used 1 Hidden layer with 16 Neurons	91.0938 %	90.9978 %	76641.2342	112301.3342	91.1308 %
RandomizedSearchCV{'epochs': 600, 'batch_size': 32, 'learning_rate': 0.001}: NN-Model 2 we used 3 Hidden layer with 16 Neurons	90.6292 %	90.5281 %	78263.6255	115193.6831	90.6817 %
RandomizedSearchCV{'epochs': 600, 'batch_size': 32, 'learning_rate': 0.001}: NN-Model 2 we used 5 Hidden layer with 35 Neurons	90.5898 %	90.4884 %	77239.6853	115435.2556	90.6127 %

Table 14: Artificial Neural Network Regression Tuning Summary

As we can see in Table 14, for the ANN Regression, the value of **root mean squared error (RMSE) is 112,301.3342** which is slightly less than 13% of the mean value of the Sales Price i.e., $867,244.88

7.13.2 Neural Network Regression Actual vs Prediction Visualization

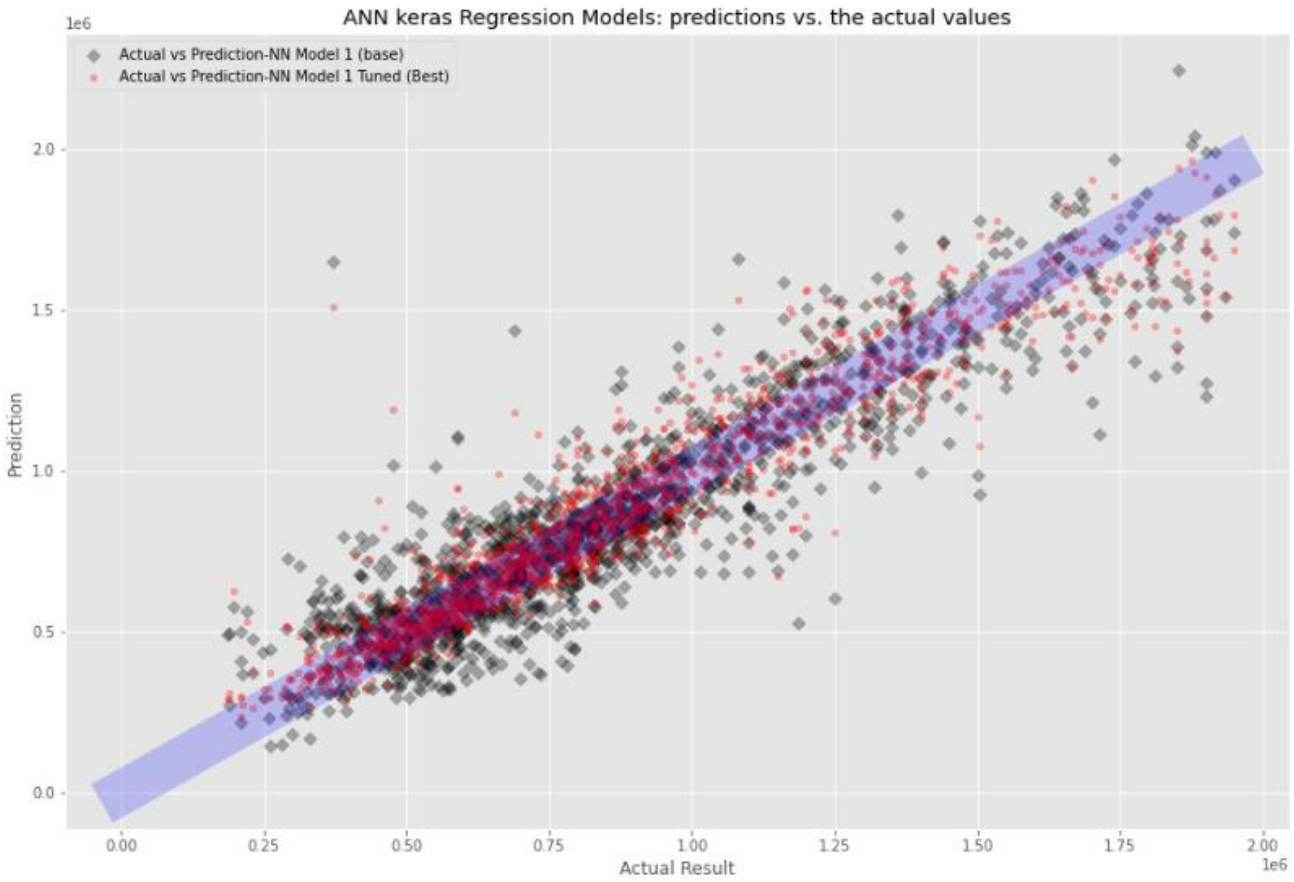


Figure 38: Neural Network Regression Actual vs Prediction Visualization

8. Final Model: Stacking Regressor

Stacking for short is an ensemble machine learning algorithm: It involves combining the predictions from multiple machine learning models on the same dataset, like bagging and boosting.
Stacking addresses the question:

Given multiple machine learning models that are skillful on a problem, but in different ways, how do you choose which model to use (trust)? The approach to this question is to use another machine learning model that learns when to use or trust each model in the ensemble.

Unlike bagging, in stacking, the models are typically different (e.g., not all decision trees) and fit on the same dataset (e.g., instead of samples of the training dataset). Unlike boosting, in stacking, a single model is used to learn how to best combine the predictions from the contributing models (e.g., instead of a sequence of models that correct the predictions of prior models). The architecture of a stacking model involves two or more base models, often referred to as level-0 models, and a meta-model that combines the predictions of the base models, referred to as a level-1 model:

- Level-0 Models (Base-Models): Models fit on the training data and whose predictions are compiled.
- Level-1 Model (Meta-Model): Model that learns how to best combine the predictions of the base models.

The meta-model is trained on the predictions made by base models on out-of-sample data. That is, data not used to train the base models is fed to the base models, predictions are made, and these predictions, along with the expected outputs, provide the input and output pairs of the training dataset used to fit the meta-model.

The outputs from the base models used as input to the meta-model may be real value in the case of regression, and probability values, probability like values, or class labels in the case of classification.

In our case we'll do as illustrated in Figure 37:

1. Base Models:
   - Random Forests
   - GB
   - XGBoost
   - Light GBM
2. Meta Model:
   - Ridge

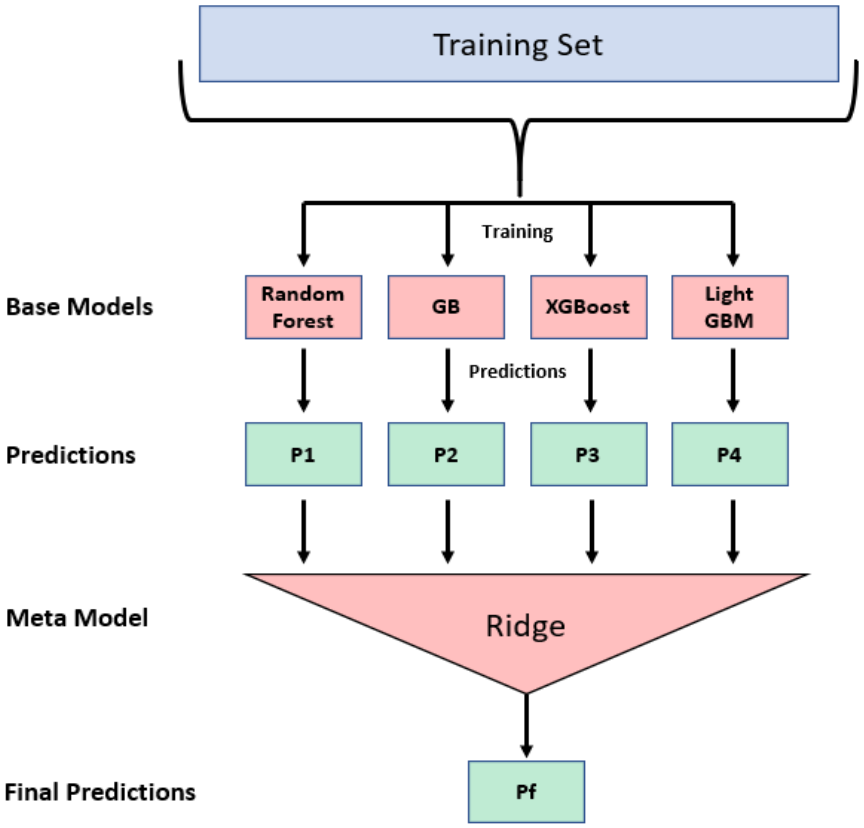


Figure 39: Stacking Regressor

8.1 Stacking Regressor: Hyperparameters Tunings

We have tuned all base models separately using RandomizedSearchCV and below are the best parameters where we achieved the best performance:

- **Random Forests:**
  - max_depth = None # this is the default
  - n_estimators = 600
  - min_samples_split = 6
  - min_samples_leaf = 1 # this is the default
  - max_features = 'log2'

- **Gradient Boosting:**
  - n_estimators = 800
  - max_depth = 6
  - learning_rate = 0.05

    - max_features = 'log2'
- **XGBoost:**
    - n_estimators = 600
    - learning_rate = 0.01
    - max_depth = 16
    - min_child_weight = 12
    - gamma = 0.3
    - colsample_bytree = 0.6

- **LightGBM:**
    - n_estimators = 1000
    - learning_rate = 0.01
    - reg_lambda = 0.4
    - reg_alpha = 0.8
    - boosting_type = 'gbdt'
    - colsample_bytree = 0.6
    - max_depth = 15
    - num_leaves= 32767 # Always num_leaves = 2^max_depth-1 AND <= (131072)

8.2 Stacking Regressor: Features Importance

StackingRegressor doesn't offer a straightforward way to extract Features Importance, but the good thing there're many Python Libraries that can provide such functionality (e.g., eli5, LIME and SHAP) and for this project we decided to use SHAP:

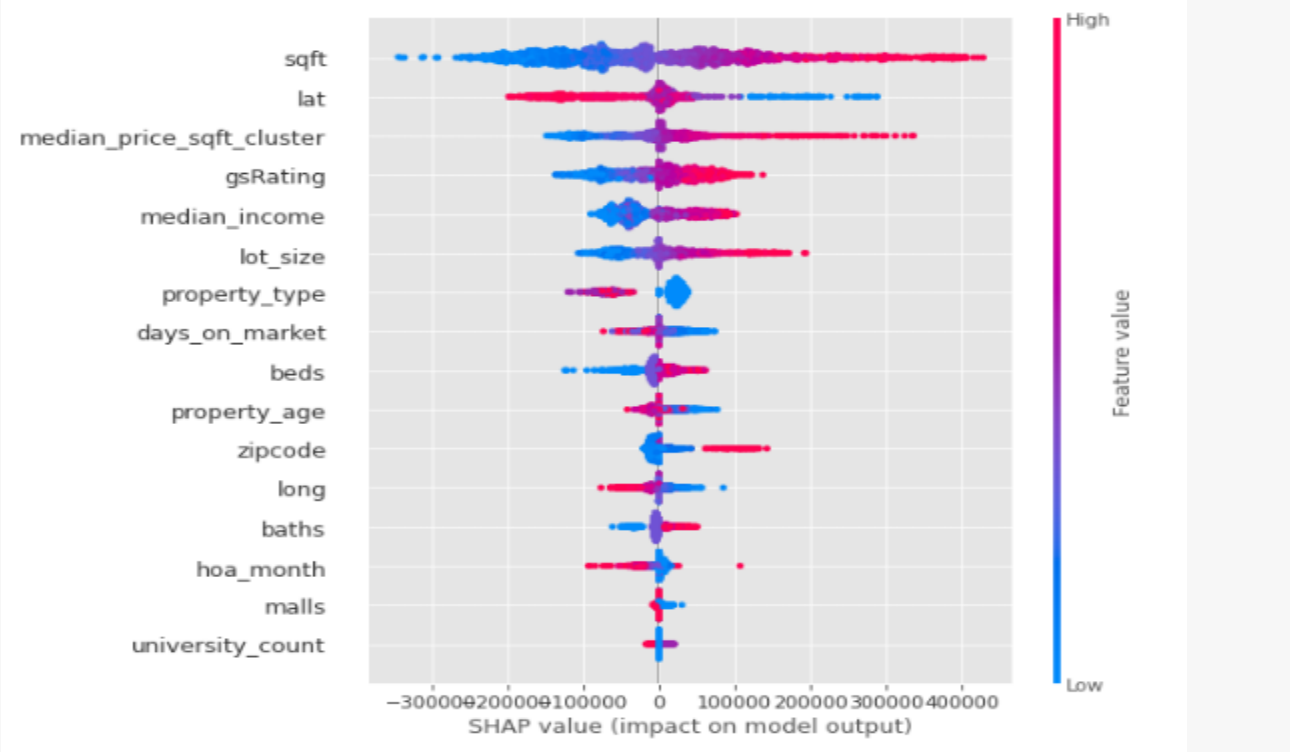


Figure 40: Stacking Regressor Features Importance Using SHAP

8.3 Stacking Regression Tuning Summary

As we can see below, we managed to improve our Stacking Regression Model by **stacking**:

- Random Forests.
- Gradient Boosting.
- LightGBM
- XGBoost

Variance Score has improved from 92.4168 % (Stacking Regression - baseline) to 92.7760 % and also, we managed to get better performance from all base (Tuned) Models as shown below:

Models	R^2 Score	Adjusted R^2 Score	MAE	RMSE	Variance Score
Tuned Random Forests	92.1982 %	92.1141 %	67886.0897	105107.8895	92.2170 %
Tuned Gradient Boosting	92.4799 %	92.3988 %	66910.7055	103193.3274	92.4863 %
Tuned XGBoost	92.5777 %	92.4977 %	67043.7283	102520.0032	92.5786 %
Tuned LightGBM	92.6304 %	92.5510 %	66235.3892	102155.1713	92.6406 %
Baseline Stacking	92.4033 %	92.3214 %	68855.7449	103717.3961	92.4168 %
Tuned Stacking	92.7615 %	92.6835 %	65381.7572	101242.5730	92.7760 %

Table 15: Stacking Regression Tuning Summary

As we can see in Table 15, for the Tuned Stacking Regression, the value of **root mean squared error (RMSE) is 101,242.5730** which is slightly larger than 11.6% of the mean value of the Sales Price i.e., $867,244.88. This makes Tuned Regressor the best and final model for our Menara App.

8.4 Stacking Regression Actual vs Prediction Visualization

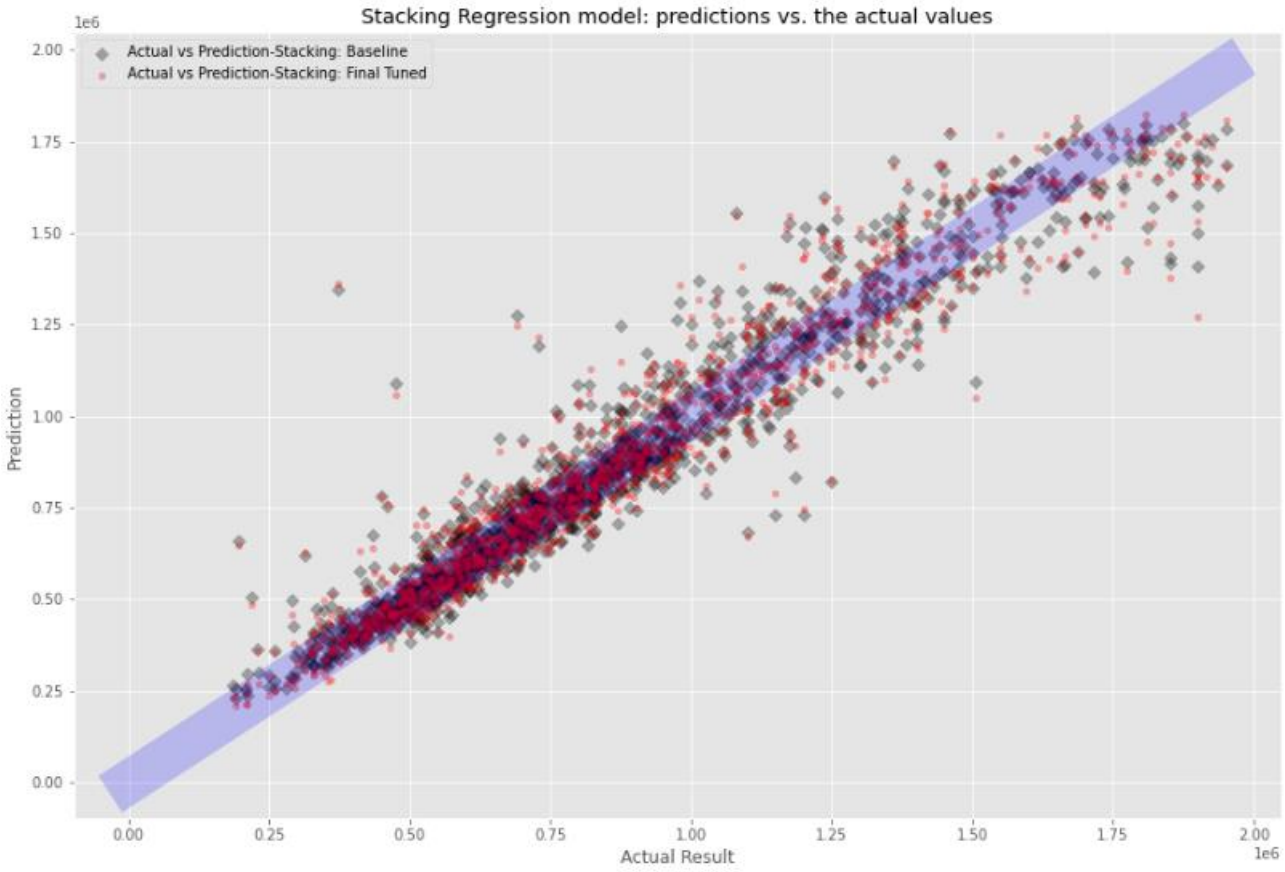


Figure 41: Stacking Regression Actual vs Prediction Visualization

9. Neural Prophet

9.1 Introduction

NeuralProphet is a Neural Network based **PyTorch** implementation of a user-friendly time series forecasting tool for practitioners. This is heavily inspired by Prophet, which is the popular forecasting tool developed by Facebook. **NeuralProphet** is developed in a fully modular architecture which makes it scalable to add any additional components in the future. The vision was to develop a simple to use forecasting tool for users while retaining the original objectives of Prophet such as interpretability, configurability and providing much more such as the automatic differencing capabilities by using **PyTorch** as the backend.

9.2 NeuralProphet vs Prophet

NeuralProphet has a number of added features with respect to original Prophet. They are as follows:

- Gradient Descent for optimization via using PyTorch as the backend.
- Modelling autocorrelation of time series using AR-Net
- Modelling lagged regressors using a separate Feed-Forward Neural Network.
- Configurable non-linear deep layers of the FFNNs.
- Tunable to specific forecast horizons (greater than 1).
- Custom losses and metrics.

9.3 Forecast Using NeuralProphet

Below Forecast is from Menara App:

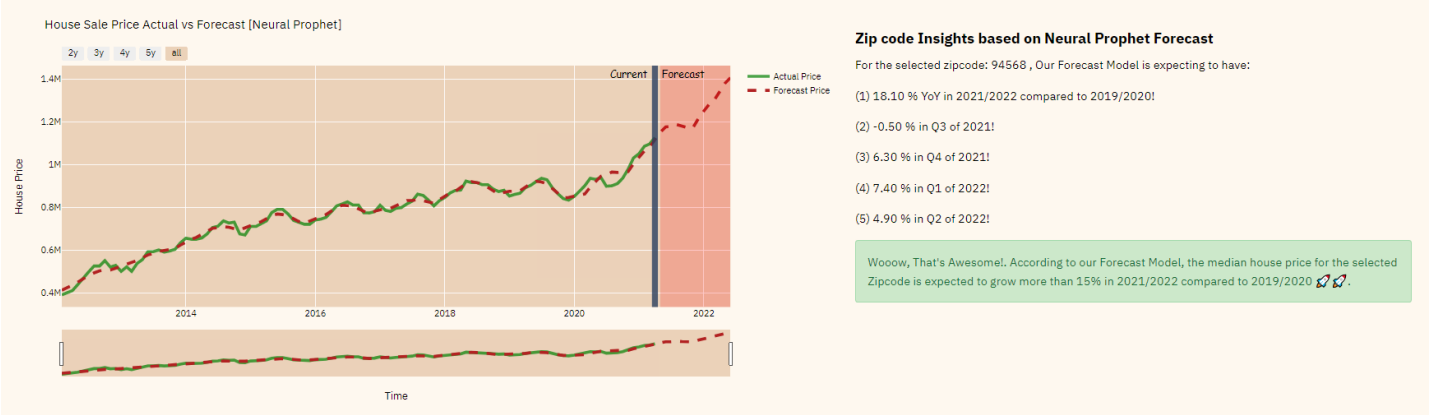


Figure 42: Forecast Using NeuralProphet

10. Menara App

First and Only Web App for House Price Estimation, Forecast and GreatSchools Search.

10.1 Introduction

Whether you want to buy, sell, refinance, or even remodel a home, **MENARA** offers many resources, estimates and forecasts to help you make the most informed decision. With a user-friendly interface, and offering many resources for buyers, sellers, and landlords alike. **MENARA** offers:

- The lowest 8.5% margin off-error for off-market homes in North California **(Competitive to the most known Home Estimate Sites e.g., Redfin)** by using the most sophisticated Machine learning algorithms.
- A golden opportunity to give you a sneak peek into the future; Up to 14 Months of house price forecast per zipcode by using a new model called **Neural Prophet**.
- A unique access to **GreatSchools**, the most trusted source of schools rating for many buyers and not just buyers with children because “location, location, location,” means “schools, schools, schools."

10.2 House Price Estimation

At **MENARA**, since the beginning, we made our mind to be unique in approaching this typical supervised machine learning problem, so here are the main reasons why you should trust our app:

- Built our dataset from scratch, utilizing multiple reliable sources (e.g., redfin, realtor, GreatSchools API,.. etc.). For more details, check Data Wrangling.
- Applied state of the art techniques in feature engineering: integrated Unsupervised Machine Learning - Clustering using K-Means and used Haversine Formula with Python to create crucial features in order to improve our final ML Model.
- Selected Stacking Regressor as our final Model because it managed to make predictions that have better performance than any single models we trained. So, in Stacking, we used a meta-learning algorithm (Ridge) to learn how to best combine the predictions from Four ML Algorithms (Random Forest, GB, XGBoost and LightGBM).

10.3 Median House Price Forecast Per Zip Code

If you're interested in buying or selling a house in 2021/2022, then, **MENARA** is offering a golden opportunity to give you a sneak peek into the future; Up to 14 Months of house price forecast per zipcode by using a new model called **Neural Prophet**. This model is a Neural Network based Time-Series Model, built on top of PyTorch and is heavily inspired by Facebook Prophet and AR-Net libraries Neural Prophet Site.

For the Data used in **Neural Prophet** we're utilizing multiple reliable sources (e.g., redfin, realtor) , because they have direct access to data from local multiple listing services, as well as insight from their real estate agents across the country.

10.4 GreatSchools Search

Most buyers understand that they may not be able to find a home that covers every single item on their wish list, but new survey data from realtor shows that school districts are an area where many buyers aren't willing to compromise. For many buyers and not just buyers with children, "location, location, location," means "schools, schools, schools."

Good schools desire by **78%** of buyers makes **GreatSchools** the trusted source of schools rating for many parents and the partner of choice for so many leading real estate websites (e.g., redfin, Zillow, realtor) simply because **GreatSchools** are the nation's leading source of school performance information and offer the most comprehensive set of school data available. Last year, **GreatSchools** had more than 55 million unique visitors, including over half of American families with school-age children.

Special Thanks to **GreatSchools** in particular Lindsay Zavala - Partnership Manager for providing a **free API trial key.** REST API access was essential to request GreatSchools Rating of all schools in specific zip codes/Cities in North California.

10. Future Work

- Currently, **Menara** App only supports specific Cities and Zip Codes in North California, for new versions of the App, we'll do our best to add more cities to our dataset in case we managed to get some positive feedback, so please stay Tuned!
- **Greykite Time Series Library** from LinkedIn was our first choice but due to many issues with the library, at this time we could not complete our forecast. we'll keep checking any updates in the library and will try it once applicable.