Prediction of Median Home Values of the 1990 California Census

Zachariah Freitas, Nicholas Lee, and Payal Muni

Department of Engineering, University of San Diego

ADS 504: Machine Learning

August 15, 2022

Introduction

California experienced one of the worst housing market collapses during the 1990s (Myers & Park, 2002). A contributing factor that may have sparked the housing crisis was the 1990 to 1991 recession (Blanchard, 1993). However, the rest of the 1990s were uncharacterized by attributes associated with a housing collapse, as there were limited restrictions for housing development and low unemployment rates throughout the decade (Myers & Park, 2022). Another contributor for the housing crisis was the local government's unwillingness to further develop housing (Shanske, 2009). Lastly, there was an obvious change in wealth distribution that that likely contributed to the housing market collapse (Pfeffer et al., 2013), given that those in lower socioeconomic households experienced significantly steeper declines in wealth. Class-wealth differences were detected by longitudinal analysis, revealing regions of lower-income and regions of higher-income. Applying the findings by Pfeffer et al. (2013) to the 1990 recession, the distribution of wealth deviates from a normal distribution. Thus, the objective of this study was to create a model that could accurately capture the housing market based on housing information collected by the US Census in the 1990s. Five different regression models were built and optimized to predict the median house values for California Census data from the 1990s. They were evaluated using root mean squared error (RMSE), mean absolute error (MAE), mean squared error (MSE), and R-squared (r^2). The goal of this study was to predict median house pricing with an RMSE score that deviates less than 20%. Building an accurate model would contribute to broadening the general understanding of wealth distribution in the early 1990s.

Methods

Exploratory Data Analysis (EDA) and Pre-Processing

The dataset utilized for this study came from Kaggle and was a CSV file that contained 1990s US Census data on housing. The data included ten features of various attributes regarding housing and a total of 20,640 observations. The features were longitude, latitude, housingmedianage, total_rooms, total_bedrooms, population, households, median_income, medianhousevalue, and ocean_proximity. The feature, ocean_proximity, was nominal, while the other variables were numerical. The target feature was medianhousevalue. The programming language, Python, was used in Google Colab to conduct this study.

To understand the initial distribution of the data, histograms and boxplots were built. They revealed that many of the features were skewed right and have outliers above the upper threshold; in boxplots (Figure 2), the upper threshold for outliers was equivalent to the third quartile value plus 150% of the interquartile range (IQR). Aside from not being normally distributed, some features, such as total_rooms, total_bedrooms, population, and households. were also highly inter-correlated to one another ($r^2 \approx 0.9$) (Figure 3).

To comply with the assumption that the data was normally distributed, all features, besides longitude and latitude, were centered and scaled. Longitude and latitude were not centered and scaled as they represent a geographical location. Centering and scaling were used on the other features to reduce the number of outliers on the upper end. Principal component analysis (PCA) was performed on all the numerical features, excluding longitude, latitude and our target feature. Given that longitude and latitude were found to be highly correlated to one another, those features were extracted into a new feature, CountyNumber using K-means clustering. K was set to 58 and used to cluster the homes into home regions into a variable called

CountyNumbers (Figure 3). K-means clustering has been previously utilized in other research to create homogenous regions based on geographic features. A K of 58 was used to represent the 58 counties in San Diego (Senate Governance and Finance Committee, 2016).

The new feature, CountyNumber, and the other categorical feature, ocean_proximity, were encoded with a one-hot encoding method. 63 new, binary features were created through encoding (five for the five classes in ocean_proximity and 58 for the number of counties in CountyNumber). Two data sets were created. The first contained 63 binary encoded categorical features along with six centered and scaled numerical features. The second data set was created for models that were unable to handle multicollinearity. This data set contained the same 63 binary encoded categorical features along with two new features created from PCA using the original six numerical features as inputs. Both the dataset without PCA and the dataset with PCA were utilized for each model built. Each model was trained with the hyperparameter, alpha (α), equal to 0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, and 1000 and a CV of ten.

Modeling

Multiple Linear Regression

Multiple linear regression was used as it is "the extension of the simple linear regression model, to include more than one independent variable" (Sarkar, et al., 2017). Thus, a simple multiple linear regression model was utilized as the baseline model; a model that would be used to compare the performance of other models against. Two multiple linear regression models were trained. In both cases, the training numerical features were centered and scaled. However, one utilized PCA to resolve collinearity issues and reduce dimensionality, while the other did not.

Lasso (L1 Regularized) Regression

Given that the number of dimensions were increased as a product of encoding, regularization was deemed appropriate to counteract potential overfitting. Additionally, one-hot encoding resulted in a sparse matrix. Lasso regularization was known for its ability to reduce the impact of unimportant variables to zero. Thus, encoded categories deemed not important by regularization will not contribute to the predictions. This process acted as a form of feature selection. While, lasso regression inherently performs feature selection, PCA was utilized to prevent any potential issues with collinearity. Amongst the Lasso models, the model that performed the best without PCA had an alpha of one, whereas the model with PCA that performed the best had an alpha of 0.1.

Ridge (L2 Regularized) Regression.

Ridge regression was utilized given that it checks for "multicollinearity in multiple regression data" (CFI Team, 2021a). From EDA, it was apparent that a few categorical variables were highly correlated, such as population and household. For that reason, there is an expectation that ridge regression would be able to handle the data. Additionally, ridge regression is beneficial in "parameter estimation and prediction" (Arashi et al., 2021) as it "penalizes the sum square values of weights" (Koppert-Anisimova, 2021) with the cost function (Penn State, n.d.):

$$\sum_{i=1}^{n}(y_1 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \quad (1)$$

Both, the ridge regression model without PCA and with PCA, found the model performed the best with an alpha parameter of 0.1.

Elastic Net

Elastic net was used given that is uses "penalties from both lasso and ridge techniques to regularize regression models" (CFI Team, 2021b). By using elastic net, the risks of multicollinearity are reduced as it "provides the inclusion of 'n' number of variables until saturation … while [choosing] one variable [from highly correlated groups]" (CFI Team, 2021b). Additionally, the penalty in elastic net is similar to that of ridge regression (CTI Team b, 2021). The best performing elastic net models, with and without PCA, had alpha parameters of 0.0001 and a L1 ratios of zero. A L1 ratio of zero says that L2 models, Ridge Regression, performed better given the parameters and data used.

Boosted Regression

Boosted regression models were used given that the "trees incorporate advantages of tree-based methods, [can handle] different types of predictor variables, and … have no need for … elimination of outliers" (Elith et al., 2008). The model works by "combining decisions from a sequence of base models" where the class of models are (User Guide Overview, n.d.):

$$\sum_{i=1}^{n}(y_1 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \quad (2)$$

Amongst the boosted regression models, the model that performed best without PCA had an alpha of one, whereas the model with PCA that performed best had an alpha of 0.1.

Results

The metrics that were utilized to compare models and determine which performed the best in predicting housing prices in California from 1990 were MSE, MAE, RMSE, and r^2 (Table 1). These metrics resulted from the cross validation of the training data. MAE "represents the average of the absolute difference between the actual and predicted values in a dataset" (Chugh,

2020) and RMSE measures the "standard deviation of the residuals" (Chugh, 2020). Both RMSE and MSE "penalize large prediction errors" (Chugh, 2020); however, RMSE "returns the same units as the dependent variable" (Chugh, 2020). R-squared explains "how well the predictor variable can explain variation in the response variables" (Chugh, 2020). The Gradient Boosting Regression without PCA model demonstrated the highest model performance (RMSE = 44440.17, MAE = 31673.94, $r^2 = 0.85$) amongst all of the models. This shows that gradient boosting regression had the better fit in predicting California housing prices in comparison to the other models. Additionally, seeing that the median housing price was around $180,000, this shows that the predicted data differed from the actual data by around 25 percent. Following the gradient boosting model, the linear regression, lasso regression, ridge regression, and elastic net models all performed similarly (RMSE ≈ 61,659, MAE ≈ 43,162, $r^2 \approx 0.71$). The lowest model performance was seen with models where PCA was utilized (Table 1).

Discussion

The results of the model evaluation show that the gradient boosting regression model without PCA best predicted the median housing prices of California from the nine predictor variables. The goal for this study was to generate a model that would accurately represent the housing market of California during the 1990's recession. Given that historical data is available on the housing crisis and that it would be misleading to inaccurately model the housing market for historical understanding and for future reference, the gradient boosting regression model without PCA best supports the goal, as it shows the potential in predicting housing prices most accurately.

Although the sample size for this dataset was large and the data came from a spread throughout the state of California, there still runs the risk of confounding variables. These

include the fact that certain cities and regions in California have a much larger population than others. These areas would have a much higher housing price than those in newer developing cities. Additionally, the data came from the 1990 California Census. This implies that only those who were contacted and willing to provide information to the Census were able to communicate the information. This serves as a confounding variable as the data might not accurately represent the populations within California and can skew the dataset. With data that is provided via survey or interview, there also runs the risk of inaccurate data. For example, those who bought their home at a lower price than what it was valued during the 1990s might unintentionally or intentionally misinform the US Census given that they might not know the worth of their home during the recession or after inflation. Information provided and the questions asked to determine information such as proximity to the ocean could be subjective and also mislead outcomes.

Conclusion

The boosted gradient regression model with no PCA was recommended for predicting California housing prices in the 1900s. The model demonstrates the potential to show how US Census data could be used to explain the economic state in a historic timeline. After updating the dataset using future data and by tracking housing trends using this model over the years, future housing prices could be accurately predicted to help future homeowners and organizations plan for housing prices in California. By deploying the boosted gradient regression model, the goal of predicting median house pricing with an RMSE score that deviates less than 20% from the median housing prices was not supported. However, given the model's overall performance, it is recommended that future studies invest time in further tuning and training the model to improve performance. Additionally, it is recommended that a subject matter expert be involved with the process to help with determining how to utilize variables to improve model performance.

References

Arashi, M., Roozbeh, M., Hamzah, N.A., & Gasparini, M. (2021, April 8). Ridge regression and its applications in genetic studies. *PLoS ONE, 16*(4). doi: https://doi.org/10.1371/journal.pone.0245376

Arora, S. (2017). Analyzing mobile phone usage using clustering in Spark MLLib and Pig. *International Journal of Advanced Research in Computer Science, 8*(1). doi: https://doi.org/10.26483/ijarcs.v8i1.2869

Blanchard, O. (1993). Consumption and the Recession of 1990-1991. *The American Economic Review, 83*(2), 270-274. https://www.jstor.org/stable/2117676

CFI Team. (2021a, July 28). *Ridge.* CFI Education Inc. Retrieved on August 14, 2022, from https://corporatefinanceinstitute.com/resources/knowledge/other/ridge/

CFI Team. (2021b, September 1). *Elastic net.* CFI Education Inc. Retrieved August 14, 2022, from https://corporatefinanceinstitute.com/resources/knowledge/other/elastic-net/

Chugh, A. (2020, December 7). *MAE, mse, rmse, coefficient of determination, adjusted r squared – Which is better?* Towards Data Science. Retrieved August 14, 2022, from https://medium.com/analytics-vidhya/mae-mse-rmse-coefficient-of-determination-adjusted-r-squared-which-metric-is-better-cd0326a5697e

Elith, J., Leathwick, J.R., & Hastle, T. (2008, April 8). A working guide to boosted regression trees. *Journal of Animal Ecology, 77*(4), 802-813. doi: https://doi.org/10.1111/j.1365-2656.2008.01390.x

Koppert-Anisimova, I. (2021, January 18). Stay away from overfitting: L2-norm regularization, weight decay and L1-norm regularization techniques. *Towards Data Science.* Retrieved

August 14, 2022, from https://medium.com/unpackai/stay-away-from-overfitting-l2-norm-regularization-weight-decay-and-l1-norm-regularization-795bbc5cf958

Myers, D., & Park, J. (2002). *The great housing collapse in California*. Washington, DC: Fannie Mae Foundation. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.458.3983&rep=rep1&type=pdf

Novianti, P., Setyorini, D., & Rafflesia, U. (2017). K-Means cluster analysis in earthquake epicenter clustering. *International Journal of Advances in Intelligent Informatics, 3*(2), 81-89. doi: 10.26555/ijain.v3i2.100

Penn State. (n.d.). Ridge regression. The Pennsylvania State University Department of Statistics Online Programs. Retrieved August 14, 2022, from 5.1 - Ridge Regression | STAT 897D (psu.edu)

Pfeffer, F. T., Danziger, S., & Schoeni, R. F. (2013). Wealth disparities before and after the Great Recession. *The ANNALS of the American Academy of Political and Social Science, 650*(1), 98-123. doi: 10.1177/0002716213497452

Sarkar D., Bali R., & Sharma, T. (2017). *Practical machine learning with Python: A problem solver's GUIDE to building real-world intelligent systems*. Apress.

Senate Governance and Finance Committee. (2016). *COUNTY FACT SHEET*. https://sgf.senate.ca.gov/sites/sgf.senate.ca.gov/files/county_facts_2016.pdf

Shanske, D. (2009). Above all else stop digging: Local government law as a (partial) cause of (and solution to) the current housing crisis. *U. Mich. JL Reform, 43*, 663. https://heinonline.org/HOL/LandingPage?handle=hein.journals/umijlr43&div=25&id=&page=

User Guide Overview. (n.d.). Overview · GitBook (apple.github.io)

Table 1

Final Model Evaluation Metrics

Model Name	R2	MAE	MSE	RMSE
Linear Regression	0.7135	43162.6	3.80E+09	61659.4
Linear Regression with PCA	0.6533	48805.3	4.60E+09	67832.4
Lasso Regression	0.7135	41360.7	3.80E+09	61659.7
Lasso Regression with PCA	0.6533	48804.9	4.60E+09	67832.4
Ridge Regression	0.7135	43162.1	3.80E+09	61659.4
Ridge Regression with PCA	0.6533	48805.2	4.60E+09	67832.5
Elastic Net	0.7135	43162.2	3.80E+09	61659.5
Elastic Net with PCA	0.6533	48805.1	4.60E+09	67832.6
Gradient Boosting Regression	0.8512	31673.9	1.97E+09	44440.2
Gradient Boosting Regression with PCA	0.6802	46536.3	4.24E+09	65145.8

Figure 1

K-NN Graph for Latitude and Longitude

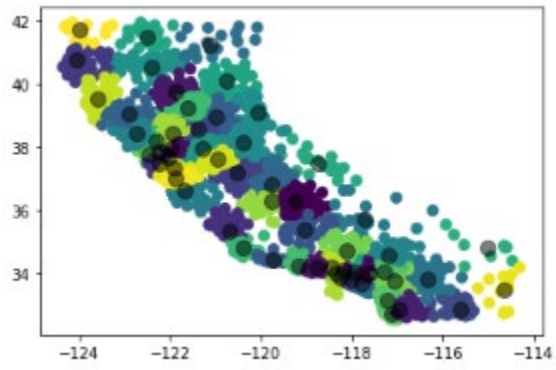


Figure 2

Boxplots showing data distribution

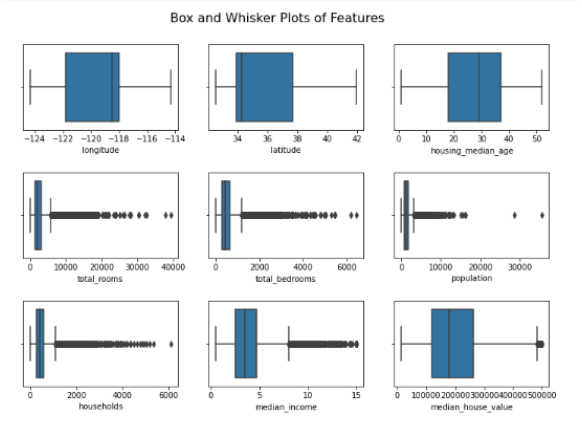
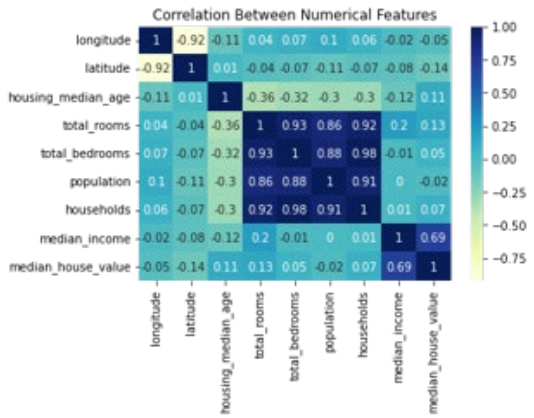


Figure 3

Correlation Matrix



ADS-504-Team-Project

August 14, 2022

1 ADS-504 Final Team Project

This project aims to construct a pipeline for accurately predicting the median value of a home in California, given census data from 1990. While the data set may be out of date, the method developed here can act as a foundation for modern home value estimations given up-to-date data.

`To save to GitHub, Click *File* and "Save a copy in GitHub", then save to the team repository`

[ ]:
```
## Load in Packages ##
from scipy.stats import skew
from math import sqrt
from google.colab import files
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.compose import ColumnTransformer
from sklearn.decomposition import PCA
from sklearn.feature_selection import VarianceThreshold
from sklearn.linear_model import ElasticNet, LinearRegression, Lasso, Ridge
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.model_selection import \
  train_test_split, cross_val_score, KFold, \
  GridSearchCV, cross_validate, RepeatedKFold
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.ensemble import GradientBoostingRegressor

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

1.1 Load in Data Set

[ ]:
```
# Load in Data #
## Upload housing.csv from device ##
#uploaded = files.upload()
```

[ ]:
```
## Convert housing.csv to a Data Frame ##
# I prefer to pull the data directly from github.
# To do this I need to go to the raw data screen in github and get the latest token value.
house_df = pd.read_csv("/content/housing.csv")
```

1.2 General Exploratory Data Analysis

Dimensions, first 5 rows, number of missing values, etc.

[ ]:
```
house_df.shape
```

[ ]:
```
(20640, 10)
```

[ ]:
```
house_df.head()
```

[ ]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	\
0	-122.23	37.88	41.0	880.0	129.0	
1	-122.22	37.86	21.0	7099.0	1106.0	
2	-122.24	37.85	52.0	1467.0	190.0	
3	-122.25	37.85	52.0	1274.0	235.0	
4	-122.25	37.85	52.0	1627.0	280.0	

	population	households	median_income	median_house_value	ocean_proximity
0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	496.0	177.0	7.2574	352100.0	NEAR BAY
3	558.0	219.0	5.6431	341300.0	NEAR BAY
4	565.0	259.0	3.8462	342200.0	NEAR BAY

Null Value Count

[ ]:
```
house_df.isnull().sum()
```

[ ]:
```
longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms      207
population            0
```

```
households              0
median_income           0
median_house_value      0
ocean_proximity         0
dtype: int64
```

Summary Statistics

[ ]:
```
house_df.describe()
```

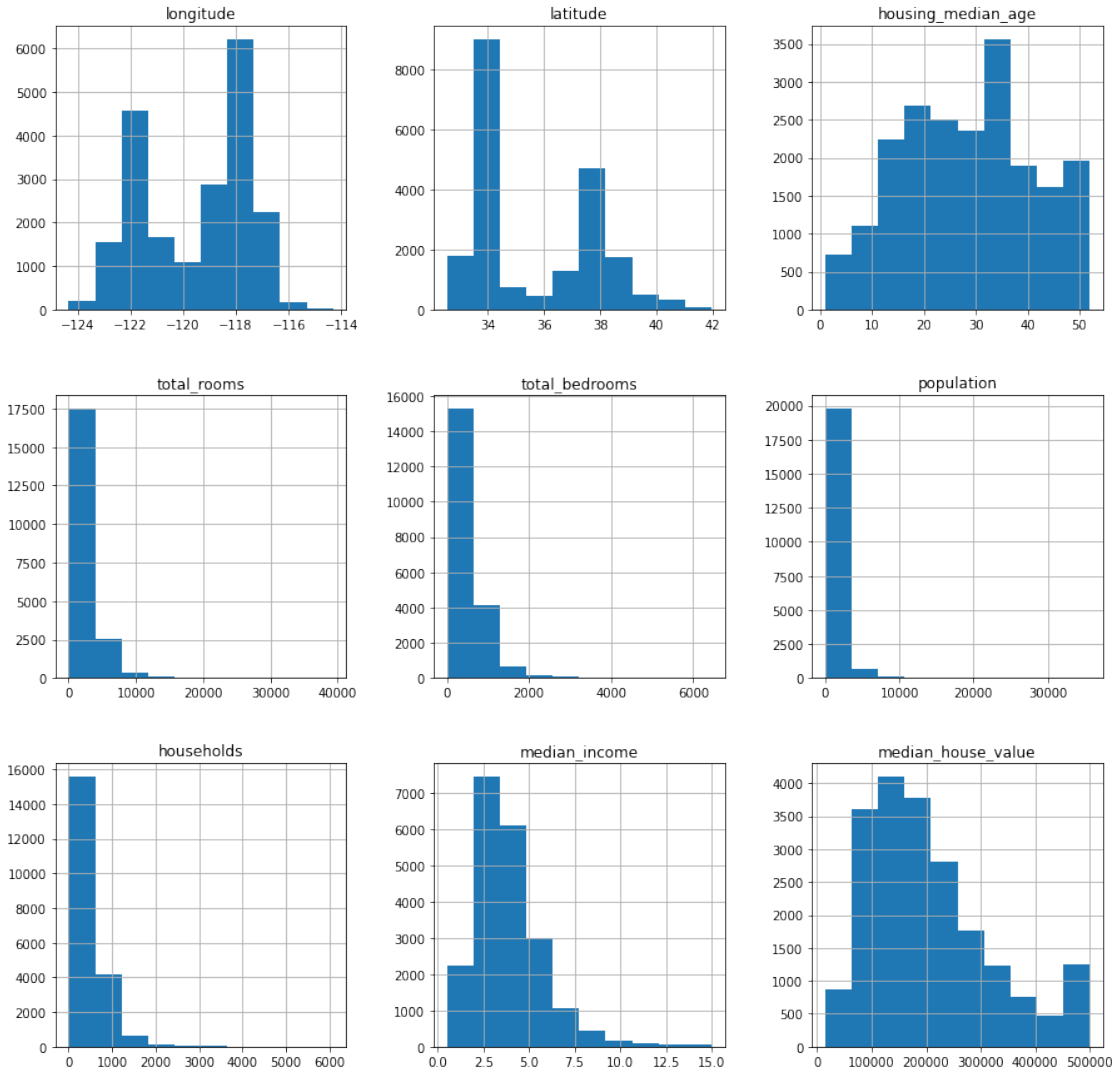
[ ]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.800000     33.930000           18.000000   1447.750000
50%     -118.490000     34.260000           29.000000   2127.000000
75%     -118.010000     37.710000           37.000000   3148.000000
max     -114.310000     41.950000           52.000000  39320.000000

       total_bedrooms    population    households  median_income  \
count    20433.000000  20640.000000  20640.000000   20640.000000
mean       537.870553   1425.476744    499.539680       3.870671
std        421.385070   1132.462122    382.329753       1.899822
min          1.000000      3.000000      1.000000       0.499900
25%        296.000000    787.000000    280.000000       2.563400
50%        435.000000   1166.000000    409.000000       3.534800
75%        647.000000   1725.000000    605.000000       4.743250
max       6445.000000  35682.000000   6082.000000      15.000100

       median_house_value
count        20640.000000
mean        206855.816909
std         115395.615874
min          14999.000000
25%         119600.000000
50%         179700.000000
75%         264725.000000
max         500001.000000
```

1.2.1 Distribution of Features

Histogram Distributions of Numerical Features

[ ]:
```
house_df.hist(figsize = (15, 15))
```

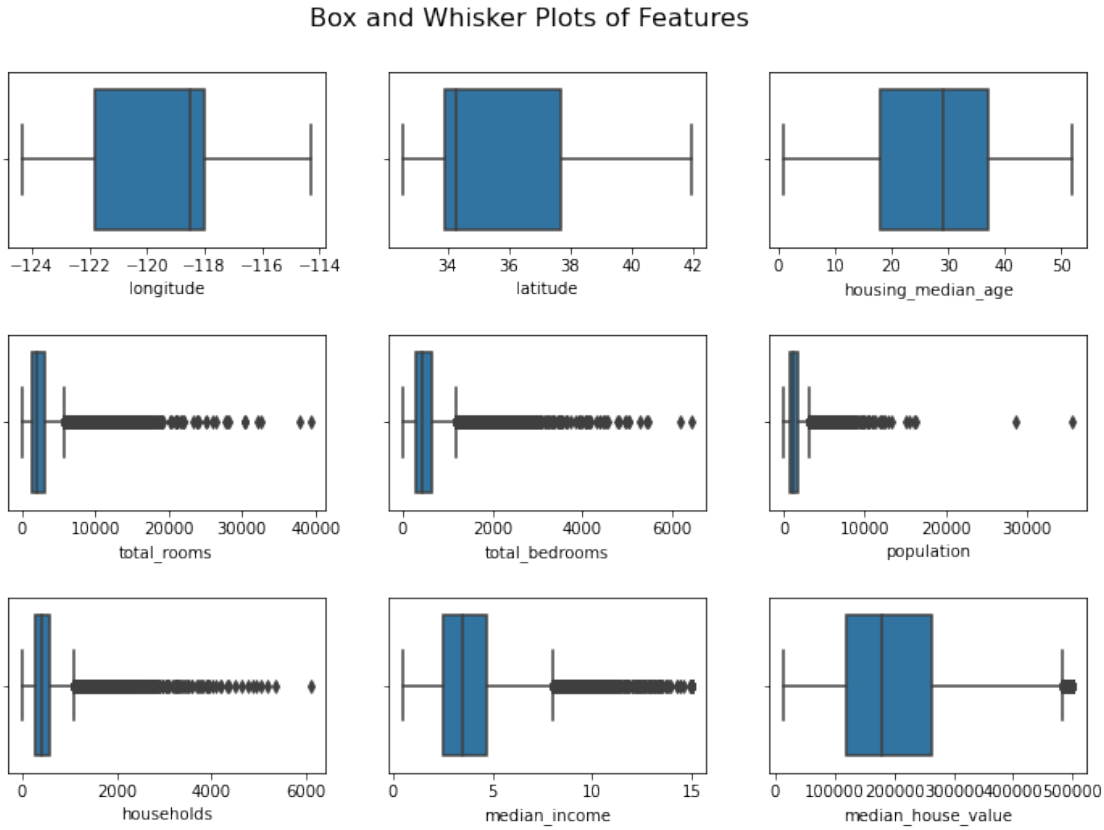
```
[ ]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81c913d0>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81be4110>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81c13550>],
           [<matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81bb8e90>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81b71390>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81b1f890>],
           [<matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81b4de10>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81b08290>,
            <matplotlib.axes._subplots.AxesSubplot object at 0x7f6d81b082d0>]],
          dtype=object)
```



The features latitude, total_rooms, total_bedrooms, population, households, median_income, and median_house_value are skewed right.

Box and Whisker Plots of Each Feature Plots will show if any outliers are present

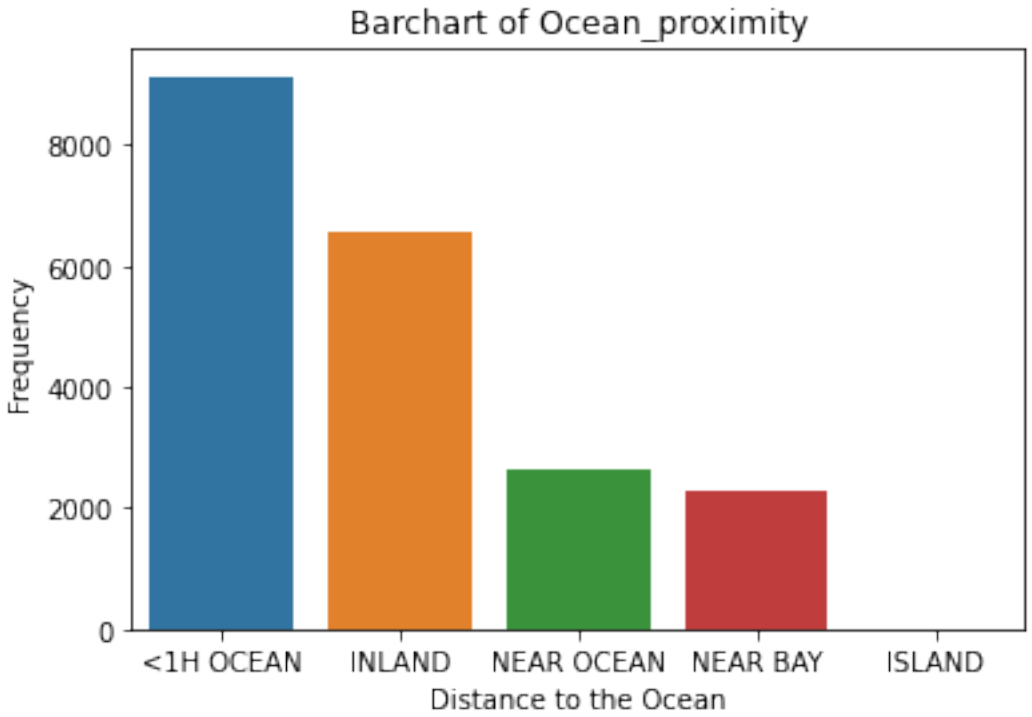
```
[ ]: plt.figure(figsize = (12, 8))
plt.subplots_adjust(hspace = 0.5)
plt.suptitle("Box and Whisker Plots of Features", fontsize = 16, y = 0.95)

feat = house_df.columns.tolist()
# Remove Categorical Feature #
feat.remove("ocean_proximity")

for n, feature in enumerate(feat):
  ax = plt.subplot(3, 3, n + 1)
  sns.boxplot(house_df[feature], ax = ax)
```



Distribution of Categorical Feature - ocean_proximity

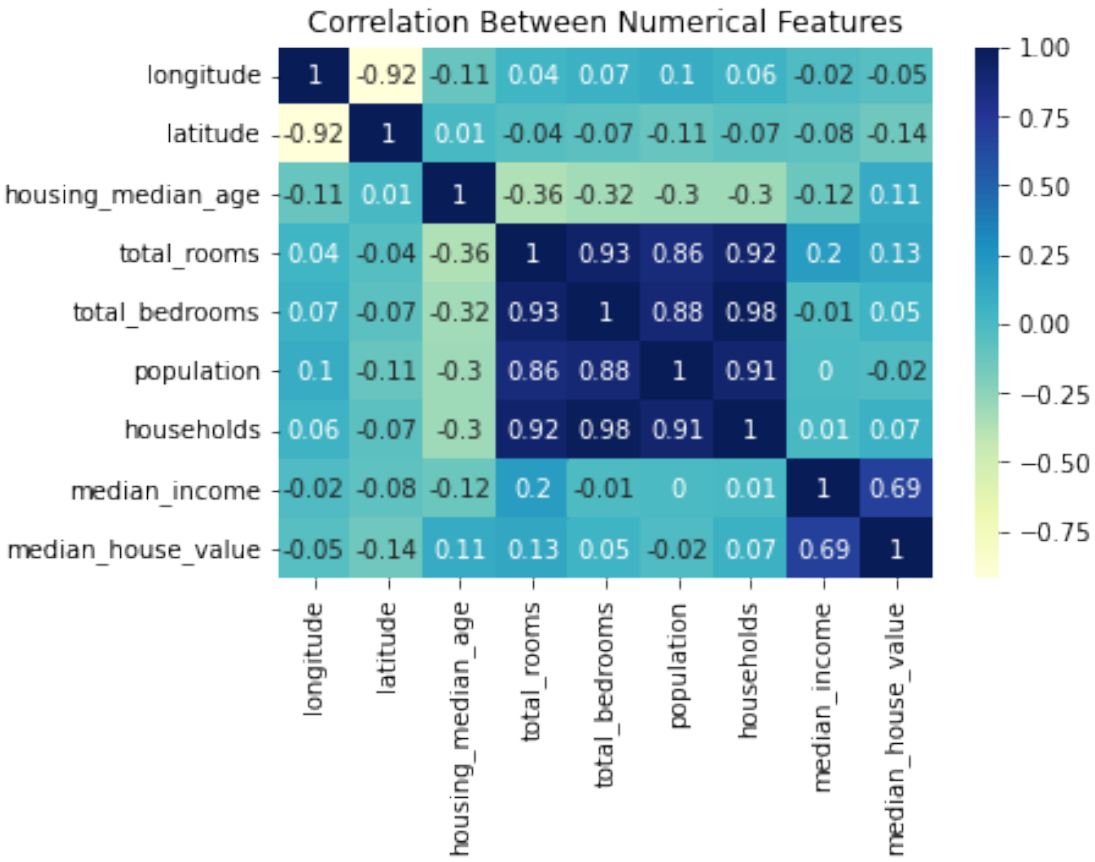
```
[ ]: ocean_prox = pd.DataFrame(house_df["ocean_proximity"].value_counts())
ocean_prox["Distance"] = ocean_prox.index
sns.barplot(ocean_prox["Distance"], ocean_prox["ocean_proximity"])
plt.xlabel("Distance to the Ocean")
plt.ylabel("Frequency")
```

```
plt.title("Barchart of Ocean_proximity")
plt.show()
```



Correlation Matrix

[ ]:
```
numerical_feat = house_df.iloc[:, :-1]
corr_values = round(numerical_feat.corr(),2)
sns.heatmap(corr_values, annot = True, cmap="YlGnBu")
plt.title("Correlation Between Numerical Features")
plt.show()
```



The four features that are strongly correlated with one another are total_rooms, total_bedrooms, population, and households.

There is also a strong correlation between latitude and longitude and a moderately strong correlation between median_income and the target feature, median_house_value.

1.3 Pre-Processing

1.3.1 Removing Missing Values

The 207 records missing values values for total_bedrooms will be removed. According to Mears et al. (n.d.), if missing value makes up less than 5% of the data, it is acceptable to remove the missing values.

Reference

Mears, K., Montelpare, W. J., Read, E., McComber, T., Mahar, A., & Ritchie, K. (n.d.). Working with missing data. Applied Statistics in Healthcare Research. Retrieved July 27, 2022, from https://pressbooks.library.upei.ca/montelpare/chapter/working-with-missing-data/#:~:text=How%20much%20data%20is%20missing%3F,to%20which%20data%20is%20missing.

[ ]:
```
house_df_completecases = house_df.dropna()
# house_df_completecases.isnull().sum()
```

1.3.2 K-Means Cluster Latitude/Longitude

K-means clustering has been used historically to create homogenous regions based on geographic features, such as longitude and latitude. Novianti et al. (2017) used K-means clustering to predict the magnitude of earthquakes, given a data set that contained longitude and latitude as predictors. Similarly, Arora (2017) used longitude and latitude to group users by location to predict for cellular activity.

K of 58 is used to represent the 58 counties in San Diego (Senate Governance and Finance Committee, 2016). Using K = 58 will mitigate the increase in dimensionality when encoding clusters to categories.

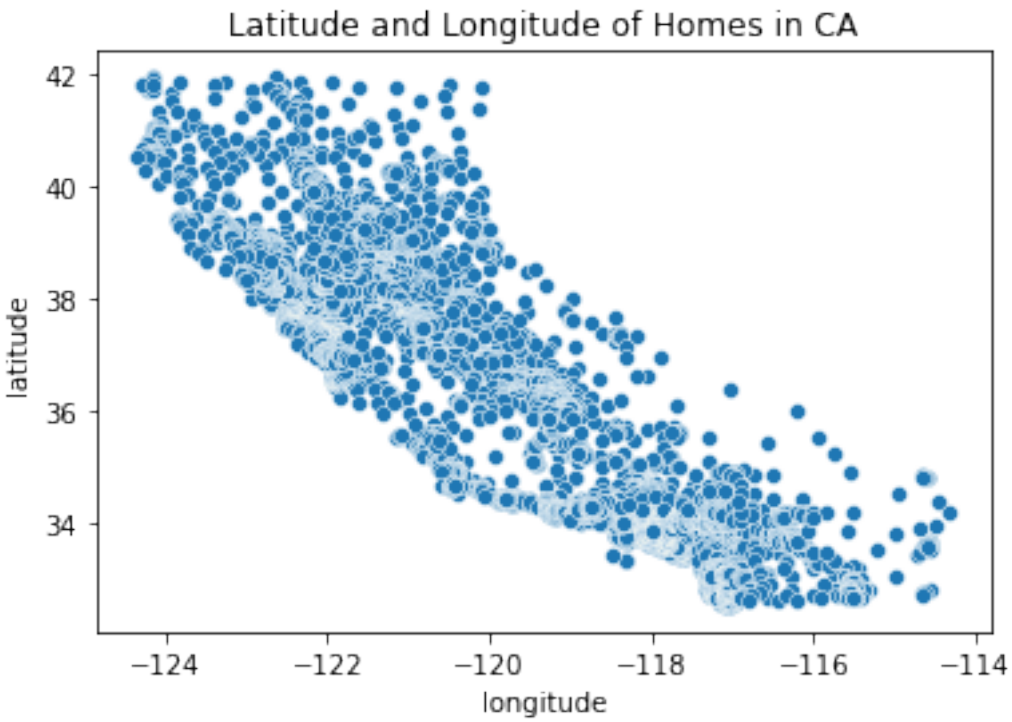
References

Arora, S. (2017). Analyzing mobile phone usage using clustering in Spark MLLib and Pig. *International Journal of Advanced Research in Computer Science, 8*(1).

Novianti, P., Setyorini, D., & Rafflesia, U. (2017). K-Means cluster analysis in earthquake epicenter clustering. *International Journal of Advances in Intelligent Informatics, 3*(2), 81-89.

Senate Governance and Finance Committee. (2016). *COUNTY FACT SHEET.* https://sgf.senate.ca.gov/sites/sgf.senate.ca.gov/files/county_facts_2016.pdf

[ ]:
```
# Plot of Latitude and Longitude #
## Latitude - Position North and South ##
## Longitude - Position East and West ##
sns.scatterplot(x = house_df["longitude"],
                y = house_df["latitude"])
plt.title("Latitude and Longitude of Homes in CA")
plt.show()
print("Correlation Between Latitude and Longitude: ",
      house_df["longitude"].corr(house_df["latitude"]))
```

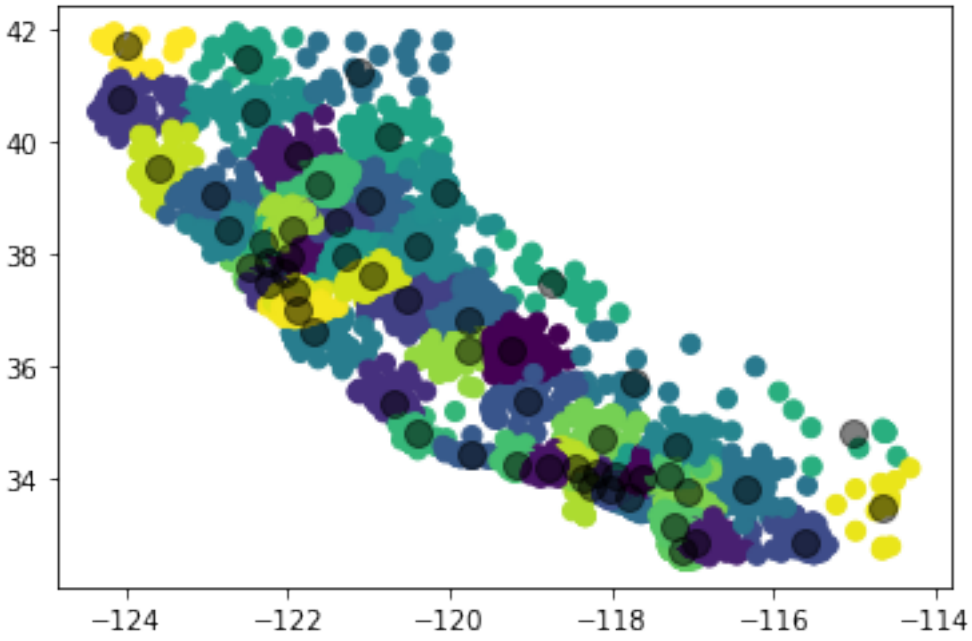



```
Correlation Between Latitude and Longitude:  -0.924664433915041
```

The variables latitude and longitude are strongly correlated. Because each pair of points represents a geographical location, clustering will be utilized to extract an (estimated) neighborhood/region feature.

[ ]:
```
lat_and_long = house_df_completecases[["longitude", "latitude"]]

from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=58, random_state = 50)
kmeans.fit(lat_and_long)
y_kmeans = kmeans.predict(lat_and_long)
```

[ ]:
```
plt.scatter(lat_and_long["longitude"], lat_and_long["latitude"],
            c = y_kmeans, s = 50, cmap='viridis')

centers = kmeans.cluster_centers_
# Centers: [longitude, latitude]
plt.scatter(centers[:, 0], centers[:, 1], c = 'black',
            s = 100, alpha=0.5);
```



Treat the assigned cluster number as a hypothetical county number. Also, because the county number is representative of longitude and latitude, longitude and latitude can be removed.

[ ]:
```
house_df_withcluster = house_df_completecases
house_df_withcluster["CountyNumber"] = y_kmeans
house_df_withcluster["CountyNumber"] = house_df_withcluster[
    "CountyNumber"].astype(str)
# house_df_withcluster.head(20)
```

[ ]:
```
# Drop Latitude and Longitude #
house_df_withcluster = house_df_withcluster.drop(
    ["longitude", "latitude"], axis = 1)
# Convert The Number to String - to treat as a category #
house_df_withcluster.head(3)
```

[ ]:

	housing_median_age	total_rooms	total_bedrooms	population	households	\
0	41.0	880.0	129.0	322.0	126.0	
1	21.0	7099.0	1106.0	2401.0	1138.0	
2	52.0	1467.0	190.0	496.0	177.0	

	median_income	median_house_value	ocean_proximity	CountyNumber
0	8.3252	452600.0	NEAR BAY	7
1	8.3014	358500.0	NEAR BAY	7
2	7.2574	352100.0	NEAR BAY	7

1.3.3 One Hot Encode

[ ]:
```
# Identify Categorical Features
categorical_feat = ["ocean_proximity", "CountyNumber"]

cat_encoder = ColumnTransformer(
    [('encoder', OneHotEncoder(), categorical_feat)])
encoded_matrix = cat_encoder.fit_transform(house_df_withcluster)

feat_names = cat_encoder.get_feature_names()

encoded_df = pd.DataFrame.sparse.from_spmatrix(encoded_matrix,
                columns = feat_names)
```

[ ]:
```
# Add Encoded Features to the DataFrame #
house_encoded_df = house_df_withcluster.reset_index(drop=True).merge(
    encoded_df.reset_index(drop=True),
    left_index=True, right_index=True)

# Drop Original Categorical Variables #
## Capital X for model ready train and test sets ##
house_df_modelready = house_encoded_df.drop(["ocean_proximity",␣
↪"CountyNumber"], axis = 1)
```

[ ]:
```
house_df_modelready
```

[ ]:

	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	encoder__x0_<1H OCEAN	encoder__x0_INLAND	encoder__x0_ISLAND	encoder__x0_NEAR BAY	encoder__x0_NEAR OCEAN	encoder__x1_1	encoder__x1_2	encoder__x1_3	encoder__x1_4	encoder__x1_5	encoder__x1_6	encoder__x1_7	encoder__x1_8	encoder__x1_9	encoder__x1_10
0	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
1	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
2	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
4	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
20428	25.0	1665.0	374.0	845.0	330.0	1.5603	78100.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
20429	18.0	697.0	150.0	356.0	114.0	2.5568	77100.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
20430	17.0	2254.0	485.0	1007.0	433.0	1.7000	92300.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
20431	18.0	1860.0	409.0	741.0	349.0	1.8672	84700.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
20432	16.0	2785.0	616.0	1387.0	530.0	2.3886	89400.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

```
20429      114.0         2.5568             77100.0                    0.0
20430      433.0         1.7000             92300.0                    0.0
20431      349.0         1.8672             84700.0                    0.0
20432      530.0         2.3886             89400.0                    0.0

       encoder__x0_INLAND  encoder__x0_ISLAND  …  encoder__x1_52  \
0                     0.0                 0.0  …             0.0
1                     0.0                 0.0  …             0.0
2                     0.0                 0.0  …             0.0
3                     0.0                 0.0  …             0.0
4                     0.0                 0.0  …             0.0
…                     …                   …    …             …
20428                 1.0                 0.0  …             0.0
20429                 1.0                 0.0  …             0.0
20430                 1.0                 0.0  …             0.0
20431                 1.0                 0.0  …             0.0
20432                 1.0                 0.0  …             0.0

       encoder__x1_53  encoder__x1_54  encoder__x1_55  encoder__x1_56  \
0                 0.0             0.0             0.0             0.0
1                 0.0             0.0             0.0             0.0
2                 0.0             0.0             0.0             0.0
3                 0.0             0.0             0.0             0.0
4                 0.0             0.0             0.0             0.0
…                 …               …               …               …
20428             0.0             0.0             0.0             0.0
20429             0.0             0.0             0.0             0.0
20430             0.0             0.0             0.0             0.0
20431             0.0             0.0             0.0             0.0
20432             0.0             0.0             0.0             0.0

       encoder__x1_57  encoder__x1_6  encoder__x1_7  encoder__x1_8  \
0                 0.0            0.0            1.0            0.0
1                 0.0            0.0            1.0            0.0
2                 0.0            0.0            1.0            0.0
3                 0.0            0.0            1.0            0.0
4                 0.0            0.0            1.0            0.0
…                 …              …              …              …
20428             0.0            0.0            0.0            0.0
20429             0.0            0.0            0.0            0.0
20430             0.0            0.0            0.0            0.0
20431             0.0            0.0            0.0            0.0
20432             0.0            0.0            0.0            0.0

       encoder__x1_9
0                0.0
1                0.0
```

```
2                0.0
3                0.0
4                0.0
…                …
20428            0.0
20429            0.0
20430            0.0
20431            0.0
20432            0.0

[20433 rows x 70 columns]
```

1.4 Train Test Split

[ ]:
```
# Split Predictors from Target Feature #
x_data = house_df_modelready.loc[
    :, house_df_modelready.columns != "median_house_value"]
y_data = house_df_modelready[["median_house_value"]]

# Distinguish Numerical Variables #
numerical_feat = house_df_modelready.iloc[:, :7].columns
numerical_feat = numerical_feat.drop("median_house_value")

# Split Data
xtrain, xtest, ytrain, ytest = train_test_split(
    x_data, y_data, test_size = 0.3, random_state = 50)
```

1.4.1 Pipeline Construction

One pipeline is built to remove near-zero variance predictors and center and scale numerical features.

Another pipeline utilizes one-hot encoding to encode categorical features, ocean_proximity and CountyNumber.

Feature Pipelines

[ ]:
```
# Numerical Pipeline:
## 1) Removes Near Zero Variance Predictors
## 2) Centers and Scales the Features
numerical_pipeline = Pipeline([("zerovariance", VarianceThreshold()),
                               ("scaler", StandardScaler())])

# PCA Pipeline for Numerical Features:
pca_pipeline = Pipeline([("pca", PCA(n_components = 2))])
```

```
# Categorical Pipeline
## Performs one-hot encoding on categorical features
# categorical_pipeline = Pipeline([("onehotencoder", OneHotEncoder())])
```

Model Pipelines

Linear Regression Since it is the most simple of the regression models, a multiple linear regression model will act as the baseline model for this study.

The below chunk is without PCA.

[ ]:
```
# Linear Regression Pipeline #
linear_reg_preprocess = make_pipeline(ColumnTransformer([
    ("num", numerical_pipeline, numerical_feat)], remainder = "passthrough"))
linear_reg_preprocess2 = make_pipeline(LinearRegression())

# Pre-Process Training Data #
linear_reg_train1 = linear_reg_preprocess.fit_transform(xtrain)
transformed_df = pd.DataFrame(linear_reg_train1)
#print(transformed_df.shape)

## Columns are ordered, thus principle components are the last columns 6 and 7␣
 ↪##
## Keep columns 6:end ##
#linear_reg_df = testdf.iloc[:, 6:]
# Train Linear Model #
linear_reg_model = linear_reg_preprocess2.fit(transformed_df, ytrain)
print("Training Metrics: ")
print("Coefficient of Determination (R^2): ",
      linear_reg_model.score(transformed_df, ytrain).round(4))
print("Mean Squared Error (MSE): ",
      mean_squared_error(
      ytrain, linear_reg_model.predict(transformed_df)).round(4))
print("Root Mean Square Error (RMSE): ",
      sqrt(mean_squared_error(
      ytrain, linear_reg_model.predict(transformed_df)).round(4)), "\n")
#print(transformed_df.shape)

# Preprocess Test Data #
linear_reg_test_preproc1 = linear_reg_preprocess.transform(xtest)

test_data_preproc_df = pd.DataFrame(linear_reg_test_preproc1)
#print(test_data_preproc_df.shape)
#linear_reg_test_df = test_data_preproc_df.iloc[:, 6:]
#print(linear_reg_test_df.shape)
```

```
#Predict Train Data
linear_reg_train_pred = linear_reg_model.predict(transformed_df)

# Fit trained model and predict test data #
linear_reg_pred = linear_reg_model.predict(test_data_preproc_df)
print("Test Metrics:")
print("Coefficient of Determination (R^2): ",
        r2_score(ytest, linear_reg_pred).round(4))
print("Mean Squared Error (MSE): ",
      mean_squared_error(
      ytest, linear_reg_pred).round(4))
print("Root Mean Square Error (RMSE): ",
      sqrt(mean_squared_error(
      ytest, linear_reg_pred)), "\n")
```

```
Training Metrics:
Coefficient of Determination (R^2):  0.7135
Mean Squared Error (MSE):  3801878023.2451
Root Mean Square Error (RMSE):  61659.37092806819

Test Metrics:
Coefficient of Determination (R^2):  0.7169
Mean Squared Error (MSE):  3805269979.068
Root Mean Square Error (RMSE):  61686.8703945015
```

The below chunk is with PCA and Cross-Validation.

[ ]:
```
# Linear Regression #
## Scaler (and Near-Zero Variance) Pipeline ##
scaler_pipe = make_pipeline(ColumnTransformer([
    ("num", numerical_pipeline, numerical_feat)], remainder = "passthrough"))

# Pre-Process Training Data #
xtrain_scaled = scaler_pipe.fit_transform(xtrain)
xtrain_scaled_df = pd.DataFrame(xtrain_scaled,
                  columns = scaler_pipe.get_feature_names_out())
# Apply Scaler Pipeline to Test Data #
xtest_scaled = scaler_pipe.transform(xtest)
xtest_scaled_df = pd.DataFrame(xtest_scaled,
                  columns = scaler_pipe.get_feature_names_out())

## PCA pipeline ##
num_features = list(xtrain_scaled_df.filter(regex="num_").columns)
pca_pipe = make_pipeline(ColumnTransformer([
    ("pca", pca_pipeline, num_features)], remainder = "passthrough"))
# Apply PCA to Training Data #
xtrain_pca = pca_pipe.fit_transform(xtrain_scaled_df)
```

```
xtrain_pca_df = pd.DataFrame(xtrain_pca)
# Apply PCA to Test Data #
xtest_pca = pca_pipe.fit_transform(xtest_scaled_df)
xtest_pca_df = pd.DataFrame(xtest_pca)

## Get Feature Names ##
feature_names = list(xtrain_scaled_df.columns)[6:]
all_features = ["PC1", "PC2"]
all_features.extend(feature_names)
# Re-name Columns #
xtrain_pca_df.columns = all_features
xtest_pca_df.columns = all_features

## Cross Validation ##
linear_reg_pipe = make_pipeline(LinearRegression())
cross_val = KFold(n_splits = 10)
cv_scores = cross_validate(linear_reg_pipe, xtrain_pca_df, ytrain,
            cv = cross_val, scoring = ["r2", "neg_mean_squared_error",
                                       "neg_root_mean_squared_error"])
print("Training Cross-Validated Metrics: ")
print("Avg. Cross-Validated R^2: ", cv_scores["test_r2"].mean().round(4))
print("Avg. Cross-Validated MSE: ", abs(cv_scores[
    "test_neg_mean_squared_error"]).mean().round(4))
print("Avg. Cross-Validated RMSE: ", abs(cv_scores[
    "test_neg_root_mean_squared_error"]).mean().round(4), "\n")

# Predict Testing Data #
linear_reg_model.fit(xtrain_pca_df, ytrain)

#PM train_pred
train_pred = linear_reg_model.predict(xtrain_pca_df)

test_pred = linear_reg_model.predict(xtest_pca_df)
print("Test Metrics: ")
print("R^2: ", r2_score(ytest, test_pred).round(4))
print("MSE: ", mean_squared_error(ytest, test_pred).round(4))
print("RMSE: ", sqrt(mean_squared_error(ytest, test_pred)))
```

```
Training Cross-Validated Metrics:
Avg. Cross-Validated R^2:  0.6436
Avg. Cross-Validated MSE:  4723667551.2431
Avg. Cross-Validated RMSE:  68673.9186

Test Metrics:
R^2:  0.6565
MSE:  4617745668.0938
RMSE:  67953.99670434257
```

Lasso Regression Model L1 Regularization

[ ]:
```
# Linear Model trained with L1 prior as regularizer (aka the Lasso).

# Lasso Regression Pipeline #
lasso_pipe = make_pipeline(Lasso())

# Parameter Grid Search #
# lasso_pipe.get_params().keys() # Use this to find available hyperparameter␣
 ↪names
lasso_params = {'lasso__alpha': [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100,␣
 ↪1000]}

# Cross-Validated Parameter Search #
lasso_gs = GridSearchCV(lasso_pipe,
                        param_grid = lasso_params,
                        scoring = "r2",
                        cv = 10,
                        return_train_score=True,)
```

[ ]:
```
# Using Non-PCA data
# Predict Testing Data #
lasso_fit = lasso_gs.fit(transformed_df, ytrain)

# Get Training Results
lasso_results = lasso_fit.cv_results_

#PM:
#Predict Training Set
lasso_train_pred = lasso_fit.predict(transformed_df)

# Predict Test Set
lasso_test_pred = lasso_fit.predict(test_data_preproc_df)

# Display training results
def training_results(model_fit):
  print(" Results from Grid Search: " )
  print("\n The best estimator across ALL searched params:\n",model_fit.
 ↪best_estimator_)
  print("\n The best score across ALL searched params:\n",model_fit.best_score_)
  print("\n The best parameters across ALL searched params:\n",model_fit.
 ↪best_params_)

training_results(lasso_fit)
```

```
## Test Prediction Results ##
def test_results(ypred):
  rsq = r2_score(ytest, ypred).round(4)
  mae = mean_absolute_error(ytest, ypred).round(4)
  mse = mean_squared_error(ytest, ypred).round(4)
  rmse = np.sqrt(mean_squared_error(ytest, ypred)).round(4)
  print("\n Test Metrics: ")
  print(f' Coefficient of Determination (R^2): {rsq:.2f}')
  print(f' Mean absolute error (MAE): {mae:.2f}')
  print(f' Mean squared error (MSE): {mse:.2f}')
  print(f' Root mean squared error (RMSE): {rmse:.2f}\n')

test_results(lasso_test_pred)

# Hyperparameter Results
def hp_results(cv_results):
  results_df = pd.DataFrame(cv_results)
  results_df = results_df.sort_values(by=["rank_test_score"])
  results_df = results_df.set_index(
      results_df["params"].apply(lambda x: "_".join(str(val) for val in x.
 ↪values()))
      ).rename_axis("param combo")
  return results_df[["params", "rank_test_score", "mean_test_score",␣
 ↪"std_test_score"]]

hp_results(lasso_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('lasso', Lasso(alpha=1))])

The best score across ALL searched params:
0.7088012816788359

The best parameters across ALL searched params:
{'lasso__alpha': 1}

Test Metrics:
Coefficient of Determination (R^2): 0.72
Mean absolute error (MAE): 43876.42
Mean squared error (MSE): 3805630466.51
Root mean squared error (RMSE): 61689.79
```

[ ]:

```
                              params  rank_test_score  mean_test_score  \
param combo
```

```
1                {'lasso__alpha': 1}              1         0.708801
0.1            {'lasso__alpha': 0.1}              2         0.708800
0.01          {'lasso__alpha': 0.01}              3         0.708799
0.001        {'lasso__alpha': 0.001}              4         0.708799
0.0001      {'lasso__alpha': 0.0001}              5         0.708799
1e-05        {'lasso__alpha': 1e-05}              6         0.708799
10              {'lasso__alpha': 10}              7         0.708689
100            {'lasso__alpha': 100}              8         0.701197
1000          {'lasso__alpha': 1000}              9         0.660635

             std_test_score
param combo
1                  0.018267
0.1                0.018261
0.01               0.018261
0.001              0.018261
0.0001             0.018261
1e-05              0.018261
10                 0.018378
100                0.018307
1000               0.018031
```

[ ]:
```
# Using PCA data

# Predict Testing Data #
lasso_pca_fit = lasso_gs.fit(xtrain_pca_df, ytrain)

# Get Training Results
lasso_pca_results = lasso_pca_fit.cv_results_

# Predict Train Set
lasso_pca_train_pred = lasso_pca_fit.predict(xtrain_pca_df)

# Predict Test Set
lasso_pca_test_pred = lasso_pca_fit.predict(xtest_pca_df)

# Display training results
training_results(lasso_pca_fit)

## Test Prediction Results ##
test_results(lasso_pca_test_pred)

# Hyperparameter Results
hp_results(lasso_pca_results)
```

```
Results from Grid Search:
```

```
The best estimator across ALL searched params:
Pipeline(steps=[('lasso', Lasso(alpha=0.1))])

The best score across ALL searched params:
0.6502119259290915

The best parameters across ALL searched params:
{'lasso__alpha': 0.1}

Test Metrics:
Coefficient of Determination (R^2): 0.66
Mean absolute error (MAE): 49354.53
Mean squared error (MSE): 4617727398.24
Root mean squared error (RMSE): 67953.86
```

[ ]:

param combo	params	rank_test_score	mean_test_score	std_test_score
0.1	{'lasso__alpha': 0.1}	1	0.650212	0.016611
0.01	{'lasso__alpha': 0.01}	2	0.650212	0.016609
0.001	{'lasso__alpha': 0.001}	3	0.650212	0.016609
0.0001	{'lasso__alpha': 0.0001}	4	0.650212	0.016609
1e-05	{'lasso__alpha': 1e-05}	5	0.650212	0.016609
1	{'lasso__alpha': 1}	6	0.650210	0.016628
10	{'lasso__alpha': 10}	7	0.650018	0.016838
100	{'lasso__alpha': 100}	8	0.641378	0.018420
1000	{'lasso__alpha': 1000}	9	0.585032	0.020291

Ridge Regression L2 Regularization

[ ]:
```
# Linear least squares with l2 regularization.

# Ridge Regression Pipeline #
```

```
ridge_pipe = make_pipeline(Ridge())

# Parameter Grid Search #
# ridge_pipe.get_params().keys() # Use this to find available hyperparameter␣
 ↪names
parameter_grid = {
    "ridge__alpha": [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# Cross-Validated Parameter Search #
ridge_gs = GridSearchCV(ridge_pipe,
                        param_grid = parameter_grid,
                        scoring = "r2",
                        cv = 10,
                        return_train_score=True,)
```

[ ]:
```
# Using Non-PCA data
# Predict Testing Data #
ridge_fit = ridge_gs.fit(transformed_df, ytrain)

# Get Training Results
ridge_results = ridge_fit.cv_results_

#Predict Train Set
ridge_train_pred = ridge_fit.predict(transformed_df)

# Predict Test Set
ridge_test_pred = ridge_fit.predict(test_data_preproc_df)

# Display training results
training_results(ridge_fit)

## Test Prediction Results ##
test_results(ridge_test_pred)

# Hyperparameter Results
hp_results(ridge_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('ridge', Ridge(alpha=0.1))])

The best score across ALL searched params:
0.7088051470141989

The best parameters across ALL searched params:
{'ridge__alpha': 0.1}
```

```
Test Metrics:
Coefficient of Determination (R^2): 0.72
Mean absolute error (MAE): 43876.66
Mean squared error (MSE): 3805289499.32
Root mean squared error (RMSE): 61687.03
```

[ ]:

param combo	params	rank_test_score	mean_test_score	std_test_score
0.1	{'ridge__alpha': 0.1}	1	0.708805	0.018271
0.01	{'ridge__alpha': 0.01}	2	0.708800	0.018262
0.001	{'ridge__alpha': 0.001}	3	0.708799	0.018261
0.0001	{'ridge__alpha': 0.0001}	4	0.708799	0.018261
1e-05	{'ridge__alpha': 1e-05}	5	0.708799	0.018261
1	{'ridge__alpha': 1}	6	0.708790	0.018363
10	{'ridge__alpha': 10}	7	0.707122	0.018697
100	{'ridge__alpha': 100}	8	0.694502	0.018112
1000	{'ridge__alpha': 1000}	9	0.652298	0.019623

[ ]:
```
# Using PCA data
# Predict Testing Data #
ridge_pca_fit = ridge_gs.fit(xtrain_pca_df, ytrain)

# Get Training Results
ridge_pca_results = ridge_pca_fit.cv_results_

#Predict Train Set
ridge_pca_train_pred = ridge_pca_fit.predict(xtrain_pca_df)

# Predict Test Set
ridge_pca_test_pred = ridge_pca_fit.predict(xtest_pca_df)

# Display training results
training_results(ridge_pca_fit)
```

```
## Test Prediction Results ##
test_results(ridge_pca_test_pred)

# Hyperparameter Results
hp_results(ridge_pca_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('ridge', Ridge(alpha=0.1))])

The best score across ALL searched params:
0.6502157786788072

The best parameters across ALL searched params:
{'ridge__alpha': 0.1}

Test Metrics:
Coefficient of Determination (R^2): 0.66
Mean absolute error (MAE): 49353.26
Mean squared error (MSE): 4617578914.53
Root mean squared error (RMSE): 67952.77
```

[ ]:

param combo	params	rank_test_score	mean_test_score	std_test_score
0.1	{'ridge__alpha': 0.1}	1	0.650216	0.016627
0.01	{'ridge__alpha': 0.01}	2	0.650212	0.016611
0.001	{'ridge__alpha': 0.001}	3	0.650212	0.016609
0.0001	{'ridge__alpha': 0.0001}	4	0.650212	0.016609
1e-05	{'ridge__alpha': 1e-05}	5	0.650212	0.016609
1	{'ridge__alpha': 1}	6	0.650165	0.016769
10	{'ridge__alpha': 10}	7	0.647661	0.017540
100	{'ridge__alpha': 100}	8	0.628123	0.018751
1000	{'ridge__alpha': 1000}	9	0.555309	

1000 0.021703

Elastic Net Model L1 and L2 Regularization

[ ]:
```
# Linear regression with combined L1 and L2 priors as regularizer.

# Elastic Net Regression Pipeline #
# enet Regression Pipeline #
enet_pipe = make_pipeline(ElasticNet())

# Parameter Grid Search #
# enet_pipe.get_params().keys() # Use this to find available hyperparameter␣
 ↪names
parameter_grid = {
    "elasticnet__alpha": [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000],
    "elasticnet__l1_ratio": [0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1]}

# For l1_ratio = 0 the penalty is an L2 penalty. L2 regularization penalizes␣
 ↪the sum of squares of the weights.
# For l1_ratio = 1 it is an L1 penalty. L1 regularization penalizes the sum of␣
 ↪absolute values of the weights.

# Cross-Validated Parameter Search #
enet_gs = GridSearchCV(enet_pipe,
                       param_grid = parameter_grid,
                       scoring = "r2",
                       cv = 10,
                       return_train_score=True,)
```

[ ]:
```
# Using Non-PCA data
# Predict Testing Data #
enet_fit = enet_gs.fit(transformed_df, ytrain)

# Get Training Results
enet_results = enet_fit.cv_results_

# Predict Train Set
enet_train_pred = enet_fit.predict(transformed_df)

# Predict Test Set
enet_test_pred = enet_fit.predict(test_data_preproc_df)

# Display training results
training_results(enet_fit)
```

```
## Test Prediction Results ##
test_results(enet_test_pred)

# Hyperparameter Results
hp_results(enet_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('elasticnet', ElasticNet(alpha=1e-05, l1_ratio=0))])

The best score across ALL searched params:
0.7088070645177229

The best parameters across ALL searched params:
{'elasticnet__alpha': 1e-05, 'elasticnet__l1_ratio': 0}

Test Metrics:
Coefficient of Determination (R^2): 0.72
Mean absolute error (MAE): 43876.59
Mean squared error (MSE): 3805315633.80
Root mean squared error (RMSE): 61687.24
```

[ ]:

param combo	params	rank_test_score	mean_test_score	std_test_score
1e-05_0	{'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…	1	0.708807	0.018276
1e-05_1e-05	{'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…	2	0.708807	0.018276
1e-05_0.0001	{'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…	3	0.708807	0.018276
1e-05_0.001	{'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…	4	0.708807	0.018276
1e-05_0.01	{'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…	5	0.708807	0.018276
…	…	…	…	…
1000_0.01	{'elasticnet__alpha': 1000, 'elasticnet__l1_ra…			
1000_0.001	{'elasticnet__alpha': 1000, 'elasticnet__l1_ra…			
1000_0.0001	{'elasticnet__alpha': 1000, 'elasticnet__l1_ra…			
1000_1e-05	{'elasticnet__alpha': 1000, 'elasticnet__l1_ra…			
1000_0	{'elasticnet__alpha': 1000, 'elasticnet__l1_ra…			

```
1000_0.01                   59        -0.000090        0.001530
1000_0.001                  60        -0.000101        0.001530
1000_0.0001                 61        -0.000102        0.001530
1000_1e-05                  62        -0.000102        0.001530
1000_0                      63        -0.000102        0.001530

[63 rows x 4 columns]
```

[ ]:
```
# Using PCA data
# Predict Testing Data #
enet_pca_fit = enet_gs.fit(xtrain_pca_df, ytrain)

# Get Training Results
enet_pca_results = enet_pca_fit.cv_results_

#Predict Train Set
ent_pca_train_pred = enet_pca_fit.predict(xtrain_pca_df)

# Predict Test Set
enet_pca_test_pred = enet_pca_fit.predict(xtest_pca_df)

# Display training results
training_results(enet_pca_fit)

## Test Prediction Results ##
test_results(enet_pca_test_pred)

# Hyperparameter Results
hp_results(enet_pca_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('elasticnet', ElasticNet(alpha=1e-05, l1_ratio=0))])

The best score across ALL searched params:
0.6502163755619608

The best parameters across ALL searched params:
{'elasticnet__alpha': 1e-05, 'elasticnet__l1_ratio': 0}

Test Metrics:
Coefficient of Determination (R^2): 0.66
Mean absolute error (MAE): 49352.43
Mean squared error (MSE): 4617475400.40
Root mean squared error (RMSE): 67952.01
```

[ ]:
```
                                                        params  \
param combo
1e-05_0       {'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…
1e-05_1e-05   {'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…
1e-05_0.0001  {'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…
1e-05_0.001   {'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…
1e-05_0.01    {'elasticnet__alpha': 1e-05, 'elasticnet__l1_r…
…                                                          …
1000_0.01     {'elasticnet__alpha': 1000, 'elasticnet__l1_ra…
1000_0.001    {'elasticnet__alpha': 1000, 'elasticnet__l1_ra…
1000_0.0001   {'elasticnet__alpha': 1000, 'elasticnet__l1_ra…
1000_1e-05    {'elasticnet__alpha': 1000, 'elasticnet__l1_ra…
1000_0        {'elasticnet__alpha': 1000, 'elasticnet__l1_ra…

              rank_test_score  mean_test_score  std_test_score
param combo
1e-05_0                     1         0.650216        0.016634
1e-05_1e-05                 2         0.650216        0.016634
1e-05_0.0001                3         0.650216        0.016634
1e-05_0.001                 4         0.650216        0.016634
1e-05_0.01                  5         0.650216        0.016633
…                         …             …               …
1000_0.01                  59        -0.000432        0.001535
1000_0.001                 60        -0.000440        0.001535
1000_0.0001                61        -0.000440        0.001535
1000_1e-05                 62        -0.000441        0.001535
1000_0                     63        -0.000441        0.001535

[63 rows x 4 columns]
```

Boosted Trees Model

[ ]:
```
# Gradient Boosting for regression.

# Gradient Boosting Regression Pipeline #
# gbr Regression Pipeline #
gbr_pipe = make_pipeline(GradientBoostingRegressor())

# Parameter Grid Search #
# gbr_pipe.get_params().keys() # Use this to find available hyperparameter names
parameter_grid = {'gradientboostingregressor__learning_rate': [0.00001, 0.0001,␣
 ↪0.001, 0.01, 0.1, 1, 10, 100, 1000],
                  # 'gradientboostingregressor__subsample': [0.9, 0.5, 0.2, 0.
 ↪1],
```

```
                  # 'gradientboostingregressor__n_estimators': [100, 500, 1000,␣
↪1500],
                  # 'gradientboostingregressor__max_depth': [4, 6, 8, 10]
                  }

# Cross-Validated Parameter Search #
gbr_gs = GridSearchCV(gbr_pipe,
                      param_grid = parameter_grid,
                      scoring = "r2",
                      cv = 10,
                      return_train_score=True,
                      n_jobs=-1)
```

[ ]:
```
# Using Non-PCA data
# Predict Testing Data #
gbr_fit = gbr_gs.fit(transformed_df, ytrain)

# Get Training Results
gbr_results = gbr_fit.cv_results_

# Predict Training Set
gbr_train_pred = gbr_fit.predict(transformed_df)

# Predict Test Set
gbr_test_pred = gbr_fit.predict(test_data_preproc_df)

# Display training results
training_results(gbr_fit)

## Test Prediction Results ##
test_results(gbr_test_pred)

# Hyperparameter Results
hp_results(gbr_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('gradientboostingregressor', GradientBoostingRegressor())])

The best score across ALL searched params:
0.7421541506416929

The best parameters across ALL searched params:
{'gradientboostingregressor__learning_rate': 0.1}

Test Metrics:
```

```
Coefficient of Determination (R^2): 0.75
Mean absolute error (MAE): 41359.87
Mean squared error (MSE): 3388518033.05
Root mean squared error (RMSE): 58210.98
```

[ ]:

```
                                                        params  \
param combo
0.1           {'gradientboostingregressor__learning_rate': 0.1}
1               {'gradientboostingregressor__learning_rate': 1}
0.01          {'gradientboostingregressor__learning_rate': 0…
0.001         {'gradientboostingregressor__learning_rate': 0…
0.0001        {'gradientboostingregressor__learning_rate': 0…
1e-05         {'gradientboostingregressor__learning_rate': 1…
10             {'gradientboostingregressor__learning_rate': 10}
100           {'gradientboostingregressor__learning_rate': 100}
1000          {'gradientboostingregressor__learning_rate': 1…

              rank_test_score  mean_test_score  std_test_score
param combo
0.1                         1     7.421542e-01        0.015085
1                           2     7.414322e-01        0.013176
0.01                        3     5.221351e-01        0.017461
0.001                       4     1.020082e-01        0.004312
0.0001                      5     9.995739e-03        0.001426
1e-05                       6    -1.477968e-04        0.001540
10                          7   -4.049996e+190             inf
100                         8             -inf             NaN
1000                        8             -inf             NaN
```

[ ]:

```
# Using PCA data
# Predict Testing Data #
gbr_pca_fit = gbr_gs.fit(xtrain_pca_df, ytrain)

# Get Training Results
gbr_pca_results = gbr_pca_fit.cv_results_

# Predict Training Set
gbr_pca_train_pred = gbr_pca_fit.predict(xtrain_pca_df)

# Predict Test Set
gbr_pca_test_pred = gbr_pca_fit.predict(xtest_pca_df)

# Display training results
training_results(gbr_pca_fit)

## Test Prediction Results ##
```

```
test_results(gbr_pca_test_pred)

# Hyperparameter Results
hp_results(gbr_pca_results)
```

```
Results from Grid Search:

The best estimator across ALL searched params:
Pipeline(steps=[('gradientboostingregressor', GradientBoostingRegressor())])

The best score across ALL searched params:
0.6638010189221191

The best parameters across ALL searched params:
{'gradientboostingregressor__learning_rate': 0.1}

Test Metrics:
Coefficient of Determination (R^2): 0.67
Mean absolute error (MAE): 48057.58
Mean squared error (MSE): 4427660552.05
Root mean squared error (RMSE): 66540.67
```

[ ]:

param combo	params	rank_test_score	mean_test_score	std_test_score
0.1	{'gradientboostingregressor__learning_rate': 0.1}	1	6.638010e-01	0.018157
1	{'gradientboostingregressor__learning_rate': 1}	2	6.431244e-01	0.019353
0.01	{'gradientboostingregressor__learning_rate': 0…	3	4.531262e-01	0.019687
0.001	{'gradientboostingregressor__learning_rate': 0…	4	8.624306e-02	0.004265
0.0001	{'gradientboostingregressor__learning_rate': 0…	5	8.282402e-03	0.001342
1e-05	{'gradientboostingregressor__learning_rate': 1…	6	-3.208031e-04	0.001531
10	{'gradientboostingregressor__learning_rate': 10}	7	-3.431532e+190	inf
100	{'gradientboostingregressor__learning_rate': 100}	8	-inf	NaN
1000	{'gradientboostingregressor__learning_rate': 1…	8	-inf	NaN

[ ]:

2 Cross Validation on the Training Data

[ ]:
```
# create a list of names
names = ['Linear Regression', 'Linear Regression w/PCA',
         'Lasso Regression', 'Lasso Regression w/PCA',
         'Ridge Regression', 'Ridge Regression w/PCA',
         'ElasticNet', 'ElasticNet w/PCA',
         'Gradient Boosting Regression', 'Gradient Boosting Regression w/PCA']


# create a list of subjects
ypred = [linear_reg_train_pred,train_pred,
 lasso_train_pred, lasso_pca_train_pred,
 ridge_train_pred, ridge_pca_train_pred,
 enet_train_pred,  ent_pca_train_pred,
 gbr_train_pred, gbr_pca_train_pred]

results = []
for i, (names, ypred) in enumerate(zip(names, ypred)):
  rsq = r2_score(ytrain, ypred).round(4)
  mae = mean_absolute_error(ytrain, ypred).round(4)
  mse = mean_squared_error(ytrain, ypred).round(4)
  rmse = np.sqrt(mean_squared_error(ytrain, ypred)).round(4)
  results.append({
      'model': names,
      'rsq': rsq,
      'mae': mae,
      'mse': mse,
      'rmse': rmse})


#Create a dataframe of results and store it here
reg_df = pd.DataFrame(results)
reg_df
```

[ ]:

	model	rsq	mae	mse	\
0	Linear Regression	0.7135	43162.6090	3.801878e+09	
1	Linear Regression w/PCA	0.6533	48804.0641	4.601239e+09	
2	Lasso Regression	0.7135	43160.6816	3.801916e+09	
3	Lasso Regression w/PCA	0.6533	48804.9007	4.601232e+09	
4	Ridge Regression	0.7135	43162.0650	3.801886e+09	
5	Ridge Regression w/PCA	0.6533	48805.1683	4.601242e+09	
6	ElasticNet	0.7135	43162.2297	3.801899e+09	

```
7                    ElasticNet w/PCA  0.6533  48805.0780  4.601259e+09
8         Gradient Boosting Regression  0.7630  39432.9865  3.145940e+09
9  Gradient Boosting Regression w/PCA  0.6802  46536.2666  4.243969e+09

          rmse
0  61659.3709
1  67832.4328
2  61659.6779
3  67832.3788
4  61659.4335
5  67832.4543
6  61659.5384
7  67832.5821
8  56088.6805
9  65145.7532
```

2.0.1 Cross Validation on the Testing

Metric - MSE, RMSE, R-squared?

[ ]:
```
# create a list of names
names = ['Linear Regression', 'Linear Regression w/PCA',
         'Lasso Regression', 'Lasso Regression w/PCA',
         'Ridge Regression', 'Ridge Regression w/PCA',
         'ElasticNet', 'ElasticNet w/PCA',
         'Gradient Boosting Regression', 'Gradient Boosting Regression w/PCA']


# create a list of subjects
ypred = [linear_reg_pred, test_pred,
         lasso_test_pred, lasso_pca_test_pred,
         ridge_test_pred, ridge_pca_test_pred,
         enet_test_pred, enet_pca_test_pred,
         gbr_test_pred, gbr_pca_test_pred]

results = []
for i, (names, ypred) in enumerate(zip(names, ypred)):
  rsq = r2_score(ytest, ypred).round(4)
  mae = mean_absolute_error(ytest, ypred).round(4)
  mse = mean_squared_error(ytest, ypred).round(4)
  rmse = np.sqrt(mean_squared_error(ytest, ypred)).round(4)
  results.append({
      'model': names,
      'rsq': rsq,
      'mae': mae,
      'mse': mse,
```

```
      'rmse': rmse})


#Create a dataframe of results and store it here
reg_df = pd.DataFrame(results)
reg_df
```

[ ]:

	model	rsq	mae	mse	rmse
0	Linear Regression	0.7169	43877.5206	3.805270e+09	61686.8704
1	Linear Regression w/PCA	0.6565	49353.7998	4.617746e+09	67953.9967
2	Lasso Regression	0.7169	43876.4228	3.805630e+09	61689.7922
3	Lasso Regression w/PCA	0.6565	49354.5263	4.617727e+09	67953.8623
4	Ridge Regression	0.7169	43876.6573	3.805289e+09	61687.0286
5	Ridge Regression w/PCA	0.6565	49353.2552	4.617579e+09	67952.7697
6	ElasticNet	0.7169	43876.5890	3.805316e+09	61687.2404
7	ElasticNet w/PCA	0.6565	49352.4350	4.617475e+09	67952.0081
8	Gradient Boosting Regression	0.7479	41359.8691	3.388518e+09	58210.9786
9	Gradient Boosting Regression w/PCA	0.6706	48057.5769	4.427661e+09	66540.6684

[ ]: