Multifamily Housing Market Analysis

December 5, 2023

Gwen Martin, Ethan Crawford, Eliza Powell, Benjamin McMullin

1 Introduction

With growing interest rates and fluctuating market conditions, price prediction in real estate development is imperative to investments in multimillion dollar projects. PEG companies is a local real estate development company that focuses on large scale development projects ranging from hotels to multifamily housing across the country. The typical process for analyzing and predicting property values is often tedious and time consuming. Methods of machine learning and Deep learning have been utilized in correlating visual property appearance with value, analyzing market conditions as well as identifying investment opportunities ("Artificial Intelligence (AI) in Real Estate") .

Working with PEG, we received a dataset with information regarding multi-family units in 46 US states. This data represents a random subset of Costar's national market inventory dataset, an important resource used by prominent real estate firms throughout the country. Properties were selected by randomly choosing various zip codes from states and pulling multifamily housing properties from each of these areas. While this dataset included a robust set of features, there is no specific measure of how location influences the value of a property which we recognize has a significant impact on the overall price. In order to mitigate the consequences of this, we include city and state population as well as location clusters of the data to represent competition in property pricing.

Our goal is to assist PEG in their data analysis to help mitigate risk and assess potential profits given specific historical data. Specifically, we will predict the sale price of a multifamily project based on key factors that influence the expense such as time, market, location and property features. We hope that our model and findings can accurately predict the success of a multifamily project as well as shed light on features affecting pricing.

2 Data

The dataset we received had 9173 rows and 31 columns. A broad range of unit attributes are included in the data, from Star Rating, to Units Per Acre, to Year Renovated.

We first identified potential indicators already included in the dataset that we believed would be helpful to be used in predicting the Sale Price. These include property city and state, number of units, star rating, actual cap rate, holding period, building class, year built, land area, number of floors, location type, building square footage, age, amenities, average unit square footage, building condition, flood zone, floor area ratio, number of bedrooms, vacancy and year renovated.

Our dataset originally had 16597 null values. The following processes were completed in order to minimize the missing data:

- Drop columns with excessive null values which we will not need. These include Property Address, Location Type, Building Materials, and Property Zip Code
- Fill empty 'Year Renovated' values with 0
- Fill 'Annual Cap Rate' and 'Vacancy' with their mean column values
- Fill 'Avg Unit SF' with the calculated ratio between 'Building SF' and 'Number of Units'
- Fill categorical null data with the mode data value.

We then dropped the remaining rows with missing values. Our clean dataset had 29 columns and 7450 rows.

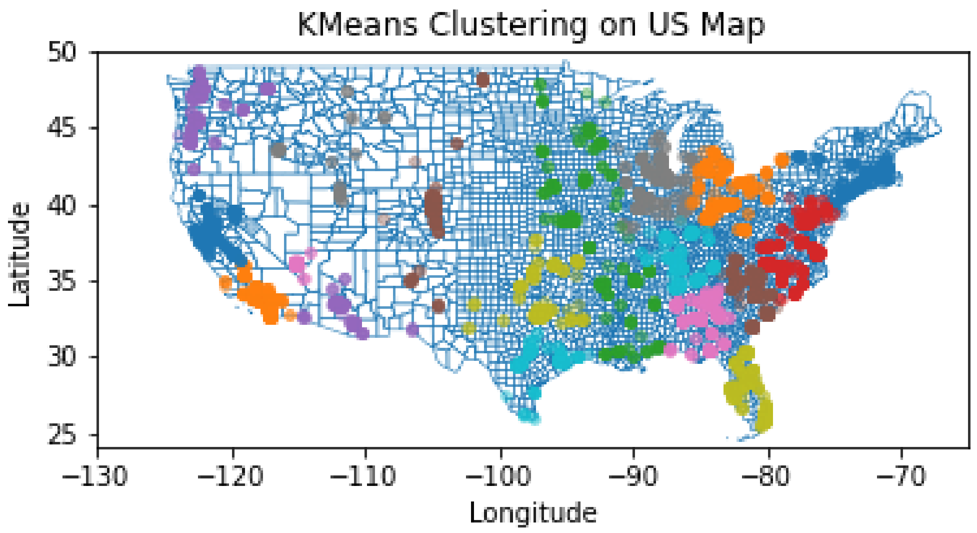
3 Feature Engineering

We undertook several measures to incorporate additional features aimed at enhancing the predictive accuracy of sale prices. Our primary objective was to integrate numerical metrics that effectively captured the variances in prices across different locations. As part of this effort, we introduced a column named 'Housing Avg by City,' linking property locations to the average sale prices within those locales. However, this metric proved to be too variable, influenced by diverse property sizes within the city. In response, we introduced an additional column, 'Avg price per SF per city,' which associates each property location with the average price per square footage, helping mitigate variability introduced by factors like property size.

In order to capture nuanced spatial patterns in housing prices, we incorporated the latitude and longitude features of our dataset by performing a KMeans Clustering algorithm. This added additional information to locations such as Los Angeles, Seattle and New York, which exhibited distinct pricing dynamics in their downtown areas compared to their rural counterparts. Each location cluster was assigned a label, which we then added as a new feature to our dataset.

Our final feature engineering involved one-hot-encoding our categorical variables. These included the 'Location Type', 'Building Class', 'Building Condition', 'Flood Zone' columns. This technique transforms categorical data into a format that machine learning models can effectively interpret, contributing to the overall robustness and accuracy of our predictive model.

```
[ ]: display(Image(filename='KMeans_housing_locations.png'))
```



4 Models

4.1 Random Forest

We begin our analysis by using a random forest regressor in order to predict sale price. Random forest is an ensemble of weak learners and thus gives a low generalization error. This method combines bootstrap aggregation as well as attribute bagging in order to provide efficient regression capabilities. For our dataset in particular a random forest provides a method which is resistant to outliers and robust despite missing data. We ran a grid search to determine the best parameters for our Random Forest model:

- Max Depth: 50
- Max Features: 'sqrt'
- Num Estimators: 500

```
[ ]: display(Image(filename='Tuned_Random_Forest.png'))
```

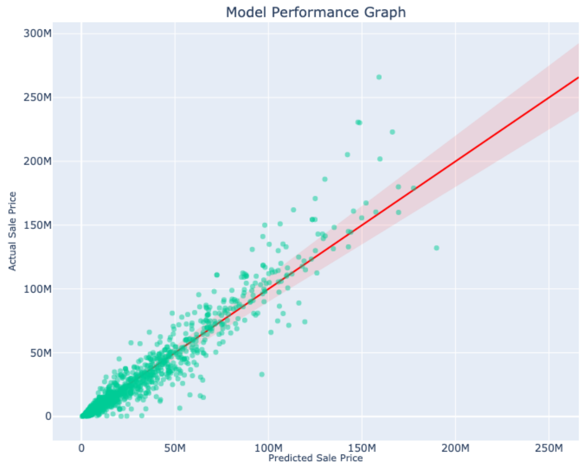
Tuned Random Forest

Model Statistics

R^2	MSE	RMSE	MAE	Std Dev
0.91457	102378077250030	10118205	5236879	34617321

Percent of Predictions Correct

Sale Price	Within 10%	Within 20%	Within 30%	Within 40%
$0M < br >$	27.14	46.29	58.97	68.92
$30M < br >$	38.46	61.99	80.09	90.05
$60M < br >$	50	76.67	90	95
$100M < br$	50	78.57	91.07	98.21
$150M < br$	36.84	42.11	78.95	100



4.2 XGBoost

XGBoost is another decision tree classifier. In comparison to Random Forest, this method utilizes base learners as opposed to larger decision trees. Even though XGBoost requires more time and hyper tuning of parameters, it often outperforms Random Forest regressors.

```
[ ]: display(Image(filename='XGBoost.png'))
```

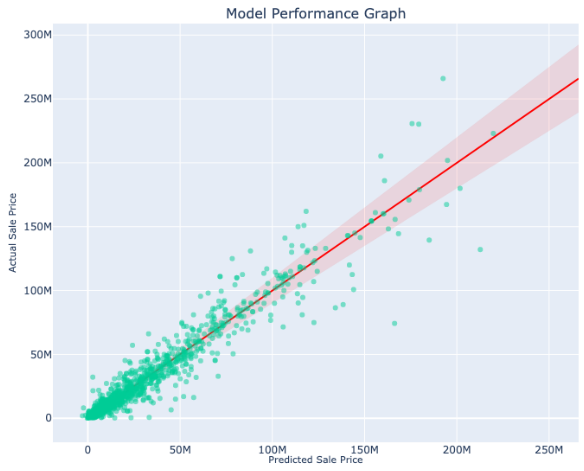
XGBoost

Model Statistics

R^2	MSE	RMSE	MAE	Std Dev
0.92921	84832792179184	9210472	4690034	34617321

Percent of Predictions Correct

Sale Price	Within 10%	Within 20%	Within 30%	Within 40%
$0M < br >$	26.01	47.7	65.45	73.9
$30M < br >$	42.53	69.23	81	90.95
$60M < br >$	50	79.17	90	94.17
$100M < br$	60.71	76.79	87.5	96.43
$150M < br$	57.89	73.68	100	100

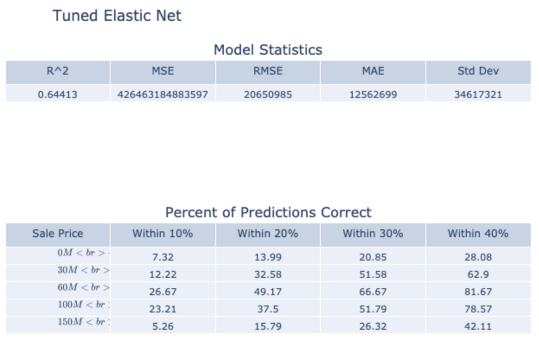


4.3 Elastic Net

Regularization techniques involve variations to the loss function through incorporating additional terms. We performed L1, L2 and Elastic Net Regularization. As anticipated, Elastic Net Regularization, a combination of the strengths of L1 and L2 Regularization, outperformed the models that included only one type of regularization. We used GridSearchCV to find the best hyperparameters:

- Alpha: 1000
- L1_ratio: 0.1
- Tol: 1000

```
[ ]: display(Image(filename='Tuned_Elastic_Net.png'))
```



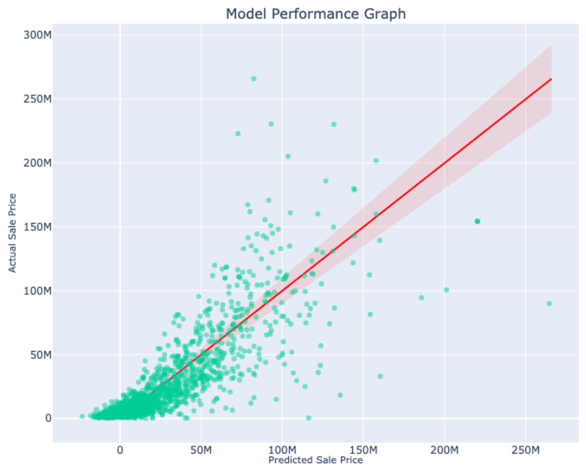
Tuned Elastic Net

Model Statistics

R^2	MSE	RMSE	MAE	Std Dev
0.64413	426463184883597	20650985	12562699	34617321

Percent of Predictions Correct

Sale Price	Within 10%	Within 20%	Within 30%	Within 40%
$0M < br >$	7.32	13.99	20.85	28.08
$30M < br >$	12.22	32.58	51.58	62.9
$60M < br >$	26.67	49.17	66.67	81.67
$100M < br$	23.21	37.5	51.79	78.57
$150M < br$	5.26	15.79	26.32	42.11



5 Analysis

5.1 Predicted vs. Actual Graph

In Figures 1-3, the x-axis represents the predicted values generated by our machine learning models, while the y-axis displays the corresponding actual values from our dataset. Each point on the graph represents an individual observation, illustrating how well our model aligns with the true outcomes.

Interpretation:

1. Y=X Line:

   The diagonal line y=x serves as a reference point, indicating perfect alignment between predicted and actual values. Points falling on or close to this line signify accurate predictions, where the model's estimated values closely match the true observed values.

2. Systematic Deviations:

   Observing any systematic deviations from the y=x line can reveal potential biases or trends in the model's predictions. For instance, consistently overestimating or underestimating values may point to areas for model refinement. We see that on average, as sale prices for properties increase, the error in prediction also tends to increase.

3. Outliers:

Identification of outliers, represented by data points significantly deviating from the y=x line, is critical. These instances may warrant further investigation, as they could be indicative of unique patterns or anomalies in the data. As stated previously, as sale prices increase, the error in prediction also increases. This could be an indication of outliers in the dataset, or it could be a failure in the feature engineering to accuratly capture trends in properties with large sale prices.

5.2 Percent of predictions within actual

A more quantitative assessment of our model's predictive accuracy is presented, detailing the percentage of predictions that fall within specified ranges of the actual sale prices. This analysis offers a nuanced understanding of how well our machine learning model captures the variability in real-world housing prices. Each table outlines the model's performance across various bins of actual sale prices, providing insights into the accuracy of our predictions at different price levels.

Interpretation:

1. 10%, 20%, 30%, 40% Accuracy:

   For each price range, the table quantifies the percentage of predictions that lie within 10%, 20%, 30%, and 40% of the actual sale price. These metrics serve as indicators of the model's precision and its ability to closely approximate the true market values.

2. Variability Across Price Ranges:

   By examining the accuracy metrics for different price brackets, we gain a nuanced perspective on the model's performance. Understanding how well the model performs across various segments of the housing market is essential for targeted improvements.

3. Implications for Model Reliability:

   High accuracy percentages within narrow price bands indicate a high level of precision, suggesting that our model excels at predicting prices in those specific ranges. Conversely, lower accuracy percentages may highlight areas where the model requires refinement or additional feature consideration.

We recognize the variability in error significance between different priced properties. For example, an estimating error of $5 million has a greater significance to a $30 million property than a $100 million property. Thus, it makes more sense to evaluate our model based on a percentage interval of accuracy. If our predicted price falls within 10% of our actual price, this implies that our predicted price falls within a range of either 10% of the actual price above or below the actual price. In comparing the accuracy among the three models, we see that XGBoost outperforms Regularization and Random Forest . It is also important to note that Elastic Net Regularization performed significantly worse in contrast to decision trees.

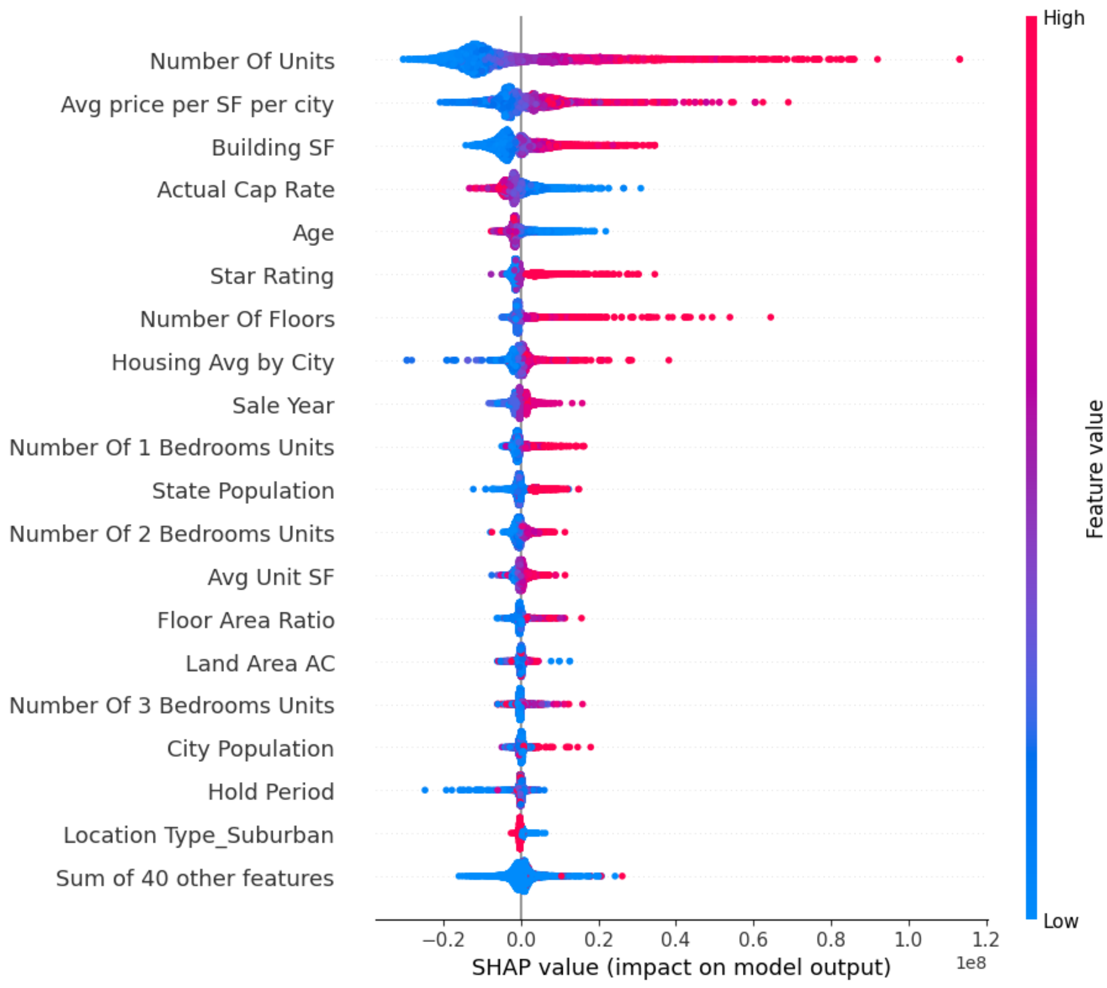
5.3 Shap

SHAP (Shapley Additive exPlanations) values provide a powerful and interpretable way to understand the impact of each feature on the predictions made by your machine learning model. Below is an explanation of SHAP values and how they can be used for analyzing your machine learning project:

1. Background: SHAP values are rooted in cooperative game theory and are designed to fairly distribute the contribution of each feature to the model's prediction across all possible feature combinations.
2. Definition: SHAP values represent the average contribution of a feature value to the prediction compared to the average prediction. Each SHAP value quantifies the impact of a specific feature for a particular instance.
3. Calculation: SHAP values are calculated by considering all possible combinations of features for a given prediction. The model's output is evaluated when each feature is included and excluded, and the difference is attributed to the feature in question.
4. Interpretation: A positive SHAP value for a feature indicates that the feature contributes positively to the model's prediction for a specific instance. Conversely, a negative SHAP value indicates a negative contribution.

The figure below is a beeswarm plot. Points in a beeswarm plot represent individual observations, and their clustering or spread offers insights into the underlying data distribution. The plot can highlight patterns, outliers, and concentration of values. The x-axis represent a SHAP values for individual instances of data. A positive SHAP value for a feature indicates that the presence of that feature increases the model's prediction compared to the average prediction. A negative SHAP value suggests the opposite. The color of a point indicates the value of that particular instance of data relative to others in the dataset. High and low values do not indicate the value of the feature, but rather the numerical value relative to the average over all data points.

```
[ ]: display(Image(filename='shap.png'))
```



6 Conclusion

In seeking to predict the value of multifamily housing properties across the country, we evaluated and analyzed three different models including Random Forest, XGBoost and Regularization techniques. Based on our findings, each of the models was moderately accurate with a particularly lower success rate for properties valued under \$30 million. As a result of these limitations, these results should not be used as an exclusive basis for analysis but should rather supplement additional findings. XGBoost clearly delivered the most accurate results among the three methods. Elastic Net Regularization however performed poorly in comparison to the other methods. We believe that XGBoost displayed better performance as a result of its robustness to outliers that exceeds the capabilities of Elastic Net Regularization. For example, outliers of properties in New York with high values relative to small square footage proved to be a better fit in the XGBoost model.

In terms of the ethical implications of this project, all of the data that was used is publicly available with purchase and does not include data about individuals. We recognize that these results do not imply causation and should be used to supplement further analysis.

7 References

- "Artificial Intelligence (AI) in Real Estate." Www.Nar.Realtor, 28 June 2023
- Oyler, B. (2017, December). US-state-populations.csv. Gist. https://gist.github.com/bradoyler/0fd473541083cfa9ea6b5da57b08461c
- Kruchten, N. (n.d.). us-cities-top-1k. GitHub. https://github.com/plotly/datasets/blob/master/us-cities-top-1k.csv

8 Appendix

[ ]:
```
import numpy as np
import pandas as pd
import os
from copy import deepcopy
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from xgboost import XGBRegressor
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import ElasticNet
import geopandas as gpd
from sklearn.cluster import KMeans
from IPython.display import display, Image
import shap
import plotly.subplots as sp
import plotly.graph_objects as go
import plotly.io as pio


DATA_DIR_PATH = os.path.join(os.getcwd(), 'data')
DATA_PATH = os.path.join(DATA_DIR_PATH, 'combined_data.csv')
CITY_POPULATION = os.path.join(DATA_DIR_PATH, 'us-cities-top-1k.csv')
STATE_POPULATION = os.path.join(DATA_DIR_PATH, 'us-state-populations.csv')
GPD_PATH = os.path.join(DATA_DIR_PATH, 'county_data.gpkg')

random_state = 42
```

[ ]:
```
# Read in data
data = pd.read_csv(DATA_PATH)
city_pop = pd.read_csv(CITY_POPULATION)
state_pop = pd.read_csv(STATE_POPULATION)
```

Data Cleaning

[ ]:
```
# Convert columns to numeric
data['Number Of Units'] = data['Number Of Units'].astype(float)
data['Sale Price'] = data['Sale Price'].astype(str).str.replace('$', '').str.
 ↪replace(',', '').astype(float)
```

```
data['Building SF'] = data['Building SF'].astype(str).str.replace(',', '').
 ↪astype(float)
data['Star Rating'] = data['Star Rating'].astype(str).str.extract('(\d+)').
 ↪astype(float)

# Get the month and the Year
data['Sale Date'] = data['Sale Date'].str.split('/')
data['Sale Month'] = data['Sale Date'].apply(lambda x: x[0]).astype(int)
data['Sale Year'] = data['Sale Date'].apply(lambda x: x[2][:4]).astype(int)
data.drop('Sale Date', axis=1, inplace=True)

# clean hold period
def clean_hold(x):
    if "Months" in x:
        clean_x = int(x[:-6])
    elif "Month" in x:
        clean_x = 1
    else:
        clean_x = 20*12
    return clean_x

data['Hold Period'] = data['Hold Period'].astype(str).apply(lambda x:␣
 ↪clean_hold(x))

# Drop the unwanted columns
data.drop(columns=['Property Address','Property Zip Code', 'Building␣
 ↪Materials', 'Year Built',
                   'Amenities'], axis=1, inplace=True)
columns_to_drop_city = ['lat', 'lon', 'State']
columns_to_drop_state = ['state']

# Drop the specified columns
city_pop = city_pop.drop(columns=columns_to_drop_city)
state_pop = state_pop.drop(columns=columns_to_drop_state)

# Rename the columns
city_pop = city_pop.rename(columns={'City': 'Property City', 'Population':␣
 ↪'City Population'})
state_pop = state_pop.rename(columns={'code': 'Property State', 'pop_2014':␣
 ↪'State Population'})

# Merge the city and state population data with the main data
data = pd.merge(data, city_pop, on='Property City')
data = pd.merge(data, state_pop, on='Property State')

### NULL VALS ###
```

```
# fill empty year renovated columns with 0
data['Year Renovated'].fillna(0, inplace=True)

# fill Actual Cap Rate with mean val
data['Actual Cap Rate'].fillna(data['Actual Cap Rate'].mean(), inplace=True)
# fill Vacancy with mean val
data['Vacancy'].fillna(data['Vacancy'].mean(), inplace=True)

# fill empty unit sf with building sf/number of units and drop remaining nans
data['Avg Unit SF'].fillna(data['Building SF'] / data['Number Of Units'],␣
 ↪inplace=True)

# fill categorical na with mode val
data['Building Condition'].fillna(data['Building Condition'].mode()[0],␣
 ↪inplace=True)
data['Flood Zone'].fillna(data['Flood Zone'].mode()[0], inplace=True)
data.isna().sum()

data.dropna(inplace=True)
```

Feature Engineering

[ ]:
```
# create new columns
data['Property City'] = data['Property City'].str.lower()
# housing avg by city
city_dict = data.groupby(by='Property City')['Sale Price'].mean().to_dict()
data['Housing Avg by City'] = data['Property City'].map(city_dict)

# price per square foot avg by city
data['Sale price per SF'] = data['Sale Price'] / data['Building SF']
city_sf_dict = data.groupby(by='Property City')['Sale price per SF'].mean().
 ↪to_dict()
data['Avg price per SF per city'] = data['Property City'].map(city_sf_dict)

# drop unnecessary cols
data.drop(columns=['Property City', 'Property State', 'Sale price per SF'],␣
 ↪axis=1, inplace=True)

# One hot encode the categorical variables
data = pd.get_dummies(data,
                      columns=['Location Type','Building Class','Building␣
 ↪Condition','Flood Zone'],
                      drop_first=True,
                      dtype=float)
```

[ ]:
```
##########
# KMeans #
```

```
##########

# Collect Geographic Data (Latitude and Longitude)
latitude = data['Latitude']
longitude = data['Longitude']

# Combine latitude and longitude
X = np.array(list(zip(latitude, longitude)))

# Do KMeans
n_clusters = 20  # Chosen from the elbow plot
kmeans = KMeans(n_clusters=n_clusters, random_state=0, n_init=10)
kmeans.fit(X)

# Plot the clusters on map
# county_data = gpd.read_file(GPD_PATH)

# Plot the county boundaries
# county_data.boundary.plot(linewidth=0.5)

# Plot the clustered data points
# for cluster in range(n_clusters):
#     cluster_points = X[kmeans.labels_ == cluster]
#     plt.scatter(cluster_points[:, 1], cluster_points[:, 0], s=20,␣
 ↪label=f'Cluster {cluster + 1}')

# fig = plt.gcf()
# fig.set_dpi(600)
# plt.title('KMeans Clustering on US Map')
# plt.xlabel('Longitude')
# plt.ylabel('Latitude')
# plt.xlim(-130, -65)
# plt.ylim(24, 50)
# plt.savefig('KMeans_housing_locations.png')
# plt.show()

# add the labels to the cleaned data
data['Label'] = kmeans.labels_

# one hot encode the labels
data = pd.get_dummies(data, columns=['Label'], dtype=float)

# drop lat and long
data.drop(columns=['Latitude', 'Longitude'], inplace=True)


###############################################
```

```
# Elbow plots to determine number of clusters #
################################################
# inertias = []
# for k in range(3, 30):
#     kmeans = KMeans(n_clusters=k, random_state=0, n_init=10).fit(X)
#     inertias.append(kmeans.inertia_)

# plt.plot(range(3, 75), inertias, marker='o')
# plt.xlabel('Number of clusters')
# plt.ylabel('Inertia')
# plt.show()
# X = np.array(list(zip(latitude, longitude)))
```

Modeling

[ ]:
```
def plot_results(y_pred, y_test, model_name):
    '''Plot the results of a model

    Parameters
    ----------

    y_pred : array-like
        The predicted values

    y_test : array-like
        The actual values

    model_name : str
        The name of the model used to make the predictions, used in the title␣
 ↪of the plot

    Returns
    -------

    None
    '''
    # Set the plot size
    plt.figure(figsize=(5, 5), dpi=100)

    def currency_formatter(x, pos):
        # Format the float as currency with commas for thousands separators and␣
 ↪two decimal places
        return "${:,.2f}".format(x)

    # Plot a line where x = y, dots above the line are overestimates, dots␣
 ↪below the line are underestimates
    percent_interval = 0.2 / 2
    max_val = max(y_test.max(), y_pred.max()) + 100
    plt.plot([0, max_val], [0, max_val], 'red')
    inverval = np.linspace(0, max_val, 1000)
    plt.fill_between(inverval, inverval + inverval*percent_interval, inverval -␣
 ↪inverval*percent_interval, color='red', alpha=0.1)

    # Plot the actual vs predicted
    plt.scatter(y_pred, y_test, label='Predicted', alpha=0.7)

    # Plot settings
    plt.xlabel('Predicted Sale Price')
    plt.ylabel('Actual Sale Price')
    plt.title(f'{model_name}: Actual Sale Price vs Predicted Sale Price')
    plt.legend()
    plt.gca().xaxis.set_major_formatter(currency_formatter)
    plt.gca().yaxis.set_major_formatter(currency_formatter)
    plt.xticks(rotation=45, ha='right')

    plt.show()

def train_test(data):
    # Set the X and y
    X = data.drop(columns=['Sale Price'])
    y = data['Sale Price']

    # Split the data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=random_state, shuffle=True)
    return X_train, X_test, y_train, y_test

def evaluate_percent_correct(y_pred, y_test, percent=0.2):
    '''Evaluate the percent of predictions that are within 20% of the actual␣
 ↪value

    Parameters
    ----------

    y_pred : array-like
        The predicted values

    y_test : array-like
        The actual values

    percent : float
        The percent of the actual value that the predicted value can be within␣
 ↪to be considered correct
        default: 0.2
    Returns
    -------

    float
        The percent of predictions that are within a percent of the actual value
    '''
    # Calculate the percent of predictions that are within 20% of the actual␣
↪value
    percent_correct = ((y_pred >= y_test * (1 - percent)) & (y_pred <= y_test *␣
↪(1 + percent))).sum() / len(y_pred)
    return percent_correct

def evaluate_predictions_with_intervals(y_pred, y_test, percents, prices,␣
↪verbose=False):
    # Create a df to store the results
    columns = [f'{percent * 100}%' for percent in percents]
    index = [f'${int(price/1e6)}M -> ${int(price2/1e6)}M' for price, price2 in␣
↪zip(prices[:-1], prices[1:])]
    results = pd.DataFrame(columns=columns, index=index)

    # For each percent
    for percent in percents:
        print(f'Within {percent*100}% of actual value:') if verbose else None

        for i in range(1, len(prices)): # For each price
            # set the price range
            price_range = [prices[i-1], prices[i]]

            # Get the percent of predictions that are within the percent of the␣
↪actual value
            y_test_range = y_test[(y_test > price_range[0]) & (y_test <␣
↪price_range[1])]
            y_pred_range = y_pred[(y_test > price_range[0]) & (y_test <␣
↪price_range[1])]

            percent_correct = round(evaluate_percent_correct(y_pred_range,␣
↪y_test_range, percent=percent)*100, 2)
            print(f'${int(price_range[0]/1e6)}M -> ${int(price_range[1]/1e6)}M:␣
↪{percent_correct}%') if verbose else None

            # Add the results to the df
            results.loc[f'${int(price_range[0]/1e6)}M -> ${int(price_range[1]/
↪1e6)}M', f'{percent*100}%'] = percent_correct
        print()
    return results

def model_pipeline(data, model, plot=False, plot_name='',␣
 ↪display_feature_importances=(False, 20), display_pred_interval=False,␣
 ↪save=False):
    # Train test split
    X_train, X_test, y_train, y_test = train_test(data)

    # Train the model
    model.fit(X_train, y_train)

    ########################
    # Performance Analysis #
    ########################
    # Predict
    y_pred = model.predict(X_test)

    # Get model statistics
    r_squared = model.score(X_test, y_test)
    mse = np.mean((y_pred - y_test)**2)
    rmse = np.sqrt(mse)
    mae = np.mean(np.abs(y_pred - y_test))
    std_dev = np.std(y_test)

    stats_df = pd.DataFrame({
        'R^2': round(r_squared, 5),
        'MSE': round(mse),
        'RMSE': round(rmse),
        'MAE': round(mae),
        'Std Dev': round(std_dev)
    }, index=[0])

    # Feature importances
    if hasattr(model, 'feature_importances_'):
        feature_importances = pd.DataFrame(model.feature_importances_,␣
 ↪index=X_train.columns, columns=['Importance'])
        feature_importances.sort_values(by='Importance', ascending=False)
    else:
        feature_importances = None

    if display_feature_importances[0]:
        print('Feature Importances:')
        display(feature_importances.head(display_feature_importances[1]))

    # How many predictions are within a percent of the actual value?
    prices = np.array([0, 3e7, 6e7, 1e8, 1.5e8, max(max(y_test), max(y_pred))])␣
 ↪   # Constants for model evaluation
```

```
    percents = np.array([0.1, 0.2, 0.3, 0.4])
    rf_results = evaluate_predictions_with_intervals(y_pred, y_test, percents,␣
 ↪prices)

    # Plot
    if plot:
        plot_data = {
        'X_train': X_train,
        'X_test': X_test,
        'y_train': y_train,
        'y_test': y_test,
        'y_pred': y_pred,
        'stats_df': stats_df,
        'rf_results': rf_results,
        'feature_importances': feature_importances,
        'plot_name': plot_name,
        'save':save
        }

        plot_individual_results_plotly(**plot_data)


    return rf_results, feature_importances

def plot_shap(data, model, bar=False, beeswarm=False):
    # Set the X and y
    X = data.drop(columns=['Sale Price'])
    y = data['Sale Price']

    # Train the model
    model.fit(X, y)

    # compute SHAP values
    explainer = shap.Explainer(xgb_default, X)
    shap_values = explainer(X)

    # Plots
    if bar:
        shap.plots.bar(shap_values)

    if beeswarm:
        shap.plots.beeswarm(shap_values, max_display=20, show=False)
        plt.savefig('shap.png', bbox_inches='tight')

def plot_individual_results_plotly(X_train, X_test, y_train, y_test, y_pred,␣
 ↪stats_df, rf_results, feature_importances, plot_name, save):
    max_val = max(y_test.max(), y_pred.max()) + 100
    ################
    # Plot Results #
    ################
    fig = sp.make_subplots(rows=3, cols=2,
                           subplot_titles=['temp' for _ in range(3)],
                           specs=[[{'type':'table'}, {"rowspan": 3, 'type':
↪'scatter'}],
                                  [{'rowspan':2, 'type':'table'}, None],
                                  [None, None]],
                           horizontal_spacing=0.05,
                          )

    # Scatter plot
    scatter = go.Scatter(x=y_pred, y=y_test, mode='markers',␣
↪marker=dict(opacity=0.5), name='')
    line = go.Scatter(x=[0, max_val], y=[0, max_val], mode='lines',␣
↪line=dict(color='red'), name='')

    # Cofidence interval
    interval = np.linspace(0, max_val, 1000)
    upper_bound = interval + interval * 0.1
    lower_bound = interval - interval * 0.1

    confidence_interval = go.Scatter(
        x=np.concatenate([interval, interval[::-1]]),
        y=np.concatenate([upper_bound, lower_bound[::-1]]),
        fill='toself',
        fillcolor='rgba(255, 0, 0, 0.1)',
        line=dict(color='rgba(255, 255, 255, 0)'),
        name=''
    )

    fig.add_trace(line, row=1, col=2)
    fig.add_trace(confidence_interval, row=1, col=2)
    fig.add_trace(scatter, row=1, col=2)
    fig.layout.annotations[1]['text'] = "Model Performance Graph"

    # Tables
    fig.add_trace(go.Table(
        header=dict(
            values=['Sale Price', 'Within 10%', 'Within 20%', 'Within 30%',␣
↪'Within 40%'],
            font=dict(size=12),
            align='center',
        ),
        cells=dict(
            values=[rf_results.index]+[rf_results[col] for col in rf_results.
↪columns],
            align = 'center',
            font=dict(size=11),
        ),

    ), row=2, col=1)
    fig.layout.annotations[2]['text'] = "Percent of Predictions Correct"


    fig.add_trace(go.Table(
        header=dict(
            values=['R^2', 'MSE', 'RMSE', 'MAE', 'Std Dev'],
            font=dict(size=12),
            align='center',
        ),
        cells=dict(
            values=[stats_df[col] for col in stats_df.columns],
            align = 'center',
            font=dict(size=11),
        ),

    ), row=1, col=1)
    fig.layout.annotations[0]['text'] = "Model Statistics"

    fig.update_xaxes(title_text='Predicted Sale Price', title_standoff=3,␣
↪title_font=dict(size=11), row=1, col=2)
    fig.update_yaxes(title_text='Actual Sale Price', title_standoff=1.5,␣
↪title_font=dict(size=11), row=1, col=2)
    fig.update_layout(
        title_text=plot_name,
        showlegend=False,
        height=600,
        width=1300,
        margin=dict(t=90, l=10, r=10, b=0)
    )

    fig.show()

    if save:
        pio.write_image(fig, f'{plot_name.replace(" ", "_")}.png')
```

[ ]:
```
#######################
# Tuned Random Forest #
#######################
rf_params = {
    'n_estimators': [50, 100, 200, 300, 400, 500],
    'max_depth': [5, 10, 20, 30, 40, 50],
    'max_features': ['sqrt', 'log2']
}

# Find the best parameters
rf_grid = GridSearchCV(RandomForestRegressor(random_state=random_state),␣
 ↪rf_params, cv=5, n_jobs=-1)
X_train, _, y_train, _ = train_test(data)
rf_grid.fit(X_train, y_train)
```

```
[ ]: GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42), n_jobs=-1,
                  param_grid={'max_depth': [5, 10, 20, 30, 40, 50],
                              'max_features': ['sqrt', 'log2'],
                              'n_estimators': [50, 100, 200, 300, 400, 500]})
```

```
[ ]: # Best random forest params
print(rf_grid.best_params_)

# Set up the model
rf_tuned = RandomForestRegressor(random_state=random_state, **rf_grid.
 ↪best_params_)

# Run the model
tuned_rf_results, feature_importances = model_pipeline(data, rf_tuned,␣
 ↪plot=True, plot_name='Tuned Random Forest',␣
 ↪display_feature_importances=(False, 15), display_pred_interval=False,␣
 ↪save=True)
```

```
{'max_depth': 50, 'max_features': 'sqrt', 'n_estimators': 500}
```

```
[ ]: ###################
# Default XGBoost #
###################
# Use data with nans (xgboost can handle them)
xgb_data = deepcopy(data)

# Set up the model
xgb_default = XGBRegressor(random_state=random_state)

# Run the model
default_xgb_results, feature_importances = model_pipeline(data, xgb_default,␣
 ↪plot=True, plot_name='XGBoost', display_feature_importances=(True, 15),␣
 ↪display_pred_interval=True, save=True)
```

```
Feature Importances:

                             Importance
Number Of Units                0.172131
Star Rating                    0.048845
Actual Cap Rate                0.017438
Hold Period                    0.005065
Land Area AC                   0.002461
Number Of Floors               0.094566
Building SF                    0.090569
Age                            0.016580
Avg Unit SF                    0.002380
Floor Area Ratio               0.015265
Number Of 1 Bedrooms Units     0.017920
Number Of 2 Bedrooms Units     0.004254
Number Of 3 Bedrooms Units     0.007850
Number Of Studios Units        0.006461
Vacancy                        0.002198
```

[ ]:
```
#################
# Tuned XGBoost #
#################
xgb_params = {
    'n_estimators': [100, 200, 300],
    'max_depth': [5, 10, 20],
    'learning_rate': [0.01, 0.05, 0.1]
}

# Find the best parameters
xgb_grid = GridSearchCV(XGBRegressor(random_state=random_state), xgb_params,␣
 ↪cv=5, n_jobs=-1)
X_train, _, y_train, _ = train_test(data)
xgb_grid.fit(X_train, y_train)
```

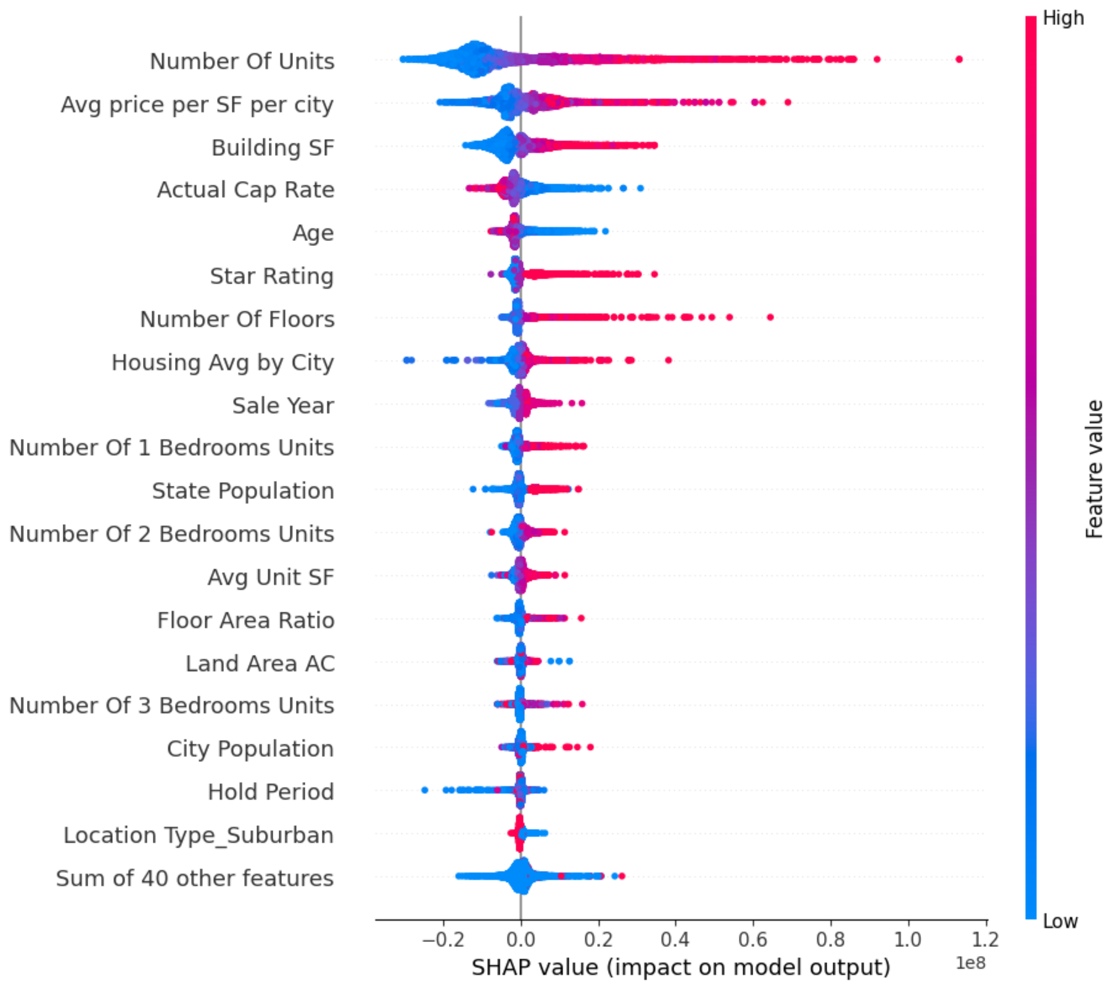
[ ]:
```
# Set up tuned model
xgb_tuned = XGBRegressor(random_state=random_state, **xgb_grid.best_params_)

# Run the model
tuned_xgb_results, feature_importances = model_pipeline(data, xgb_tuned,␣
 ↪plot=True, plot_name='Tuned XGBoost', display_feature_importances=(True,␣
 ↪15), display_pred_interval=True)
```

```
[ ]: #######################
     # Tuned Elastic Net #
     #######################
     X_train, X_test, y_train, y_test = train_test(data)

     # Elastic Net Grid Search
     elastic_net_param_grid = {'alpha': [0.0001, 0.001, 0.1, 1, 10, 100, 1000],␣
      ↪'l1_ratio': [0.1, 0.5, 0.7, 0.9, .99]}
     elastic_net_grid = GridSearchCV(ElasticNet(random_state=random_state,␣
      ↪max_iter=1000, tol=1000), elastic_net_param_grid,␣
      ↪scoring='neg_mean_squared_error', cv=5)
     elastic_net_grid.fit(X_train, y_train)
     # Best Elastic Net model
     best_elastic_net_model = elastic_net_grid.best_estimator_
```

```
[ ]: en_tuned = ElasticNet(**best_elastic_net_model.get_params())

     en_results, feature_importances = model_pipeline(data, en_tuned, plot=True,␣
      ↪plot_name='Tuned Elastic Net', display_feature_importances=(False, 15),␣
      ↪display_pred_interval=False, save=True)
```

```
[ ]: plot_shap(data, xgb_default, beeswarm=True)
```

```
/opt/homebrew/lib/python3.10/site-packages/xgboost/core.py:160: UserWarning:

[17:56:30] WARNING: /Users/runner/work/xgboost/xgboost/src/c_api/c_api.cc:1240:
Saving into deprecated binary model format, please consider using `json` or
`ubj`. Model format will default to JSON in XGBoost 2.2 if not specified.

 96%|===================  | 7149/7450 [00:17<00:00]
```


Number Of Units
Avg price per SF per city
Building SF
Actual Cap Rate
Age
Star Rating
Number Of Floors
Housing Avg by City
Sale Year
Number Of 1 Bedrooms Units
State Population
Number Of 2 Bedrooms Units
Avg Unit SF
Floor Area Ratio
Land Area AC
Number Of 3 Bedrooms Units
City Population
Hold Period
Location Type_Suburban
Sum of 40 other features
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e8
SHAP value (impact on model output)
High
Low
Feature value

```
[ ]:
```