California Houses Report

Martino Fusai
Collaborators: Chadi Tawbi, Alexandre Amate, Thomas Rajji, Shiyao Gu

October 2023

Index

Disclaimer

Given the constraints imposed and the context of our studies, we used Chat GPT to untangle our code and save time on certain methods. However, it was only a tool to help us, and the thinking behind the code, as well as the majority of the code, was not generated by Chat GPT.

Summary

Our study focuses on the exploration of a dataset named `California_Houses.csv` which covers real estate activity at the individual level in California. Our aim was to make the most of this dataset and produce predictions using a machine learning model based on a variable we determined.

0 Setup

First of all, we imported all the necessary packages and modules.

```
from __future__ import division, print_function, unicode_literals          [1]

import numpy as np
import os
import pandas as pd
import seaborn as sb

pd.set_option("display.max_rows", 999)
np.random.seed(42)

%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

After having set the working directory, we then proceeded to import the dataset and to do a preliminary analysis of its characteristics. In particular, the dataset has 14 columns and 20639 rows, where each row represents one district in California.

In order to create a new categorical variable `Closest_city` based on the closest California city and dropped the distance to each city, we created a dictionary called `rename_dict` to map the existing column names to more concise names, replacing the `Distance_to_` prefix with the city names. We then used the `rename` method to update the column names in the DataFrame. We calculated the closest city for each row by creating a new column `Closest_city` and using the `idxmin` method with `axis=1`. This method returned the name of the column (city) with the smallest distance for each row. Finally, we dropped the unnecessary columns, including `Distance_to_coast` and the individual city distance columns (*LA, SanDiego, SanJose, SanFrancisco*), since we had already determined the closest city and no longer need these distance values.

```
rename_dict = {'Distance_to_LA':'LA',                                    [5]
               'Distance_to_SanDiego': 'SanDiego',
              'Distance_to_SanJose': 'SanJose',
              'Distance_to_SanFrancisco' :'SanFrancisco'}

df.rename(columns=rename_dict, inplace=True)
df['Closest_city'] = df[['LA', 'SanDiego',
                         'SanJose', 'SanFrancisco']].idxmin(axis=1)
df.drop(['Distance_to_coast', 'LA', 'SanDiego', 'SanJose', 'SanFrancisco'],
        axis=1, inplace=True)
```

We then applied the `describe()` method on the transformed dataset to see all the relevant statistics of the quantitative variables (table 1).

	Median_House_Value	Median_Income	Median_Age	Tot_Rooms	Tot_Bedrooms	Population
count	20654.000000	20654.000000	20654.000000	20654.000000	20654.000000	20654.000000
mean	206859.008521	3.870671	28.638230	2635.554472	537.748338	1425.302411
std	115395.615874	1.899822	12.591488	2180.774851	421.068876	1132.221567
min	14999.000000	0.499900	1.000000	2.000000	1.000000	3.000000
25%	119600.000000	2.563400	18.000000	1447.500000	295.000000	787.000000
50%	179700.000000	3.534800	29.000000	2127.000000	435.000000	1166.000000
75%	264725.000000	4.743250	37.000000	3147.500000	647.000000	1725.000000
max	500001.000000	15.000100	52.000000	39320.000000	6445.000000	35682.000000

	Households	Latitude	Longitude	Distance_to_coast	Tot_No_Bedrooms	Max_Age
count	20654.000000	20654.000000	20654.000000	20654.000000	20654.000000	20654.000000
mean	499.375666	35.631504	-119.569465	40499.168256	2097.530551	43.153239
std	382.344011	2.135876	2.003603	49130.495010	1796.213255	12.947424
min	-98.000000	32.540000	-124.350000	120.676447	0.000000	9.000000
25%	280.000000	33.930000	-121.800000	9079.756762	1126.000000	33.000000
50%	409.000000	34.260000	-118.490000	20521.624925	1681.000000	43.000000
75%	605.000000	37.710000	-118.010000	49824.747685	2509.000000	52.000000
max	6082.000000	41.950000	-114.310000	333804.686371	33110.000000	76.000000

Table 1: `housing.describe()` output [11]

1 Data Exploration

In this second phase, we started to analyze our data to find interesting insights or any possible problem.

1.1 Preliminary Analysis

The discrete quantitative variables are *Median Age, Tot Rooms, Tot Bedrooms, Population, Households,* while continuous quantitative variables are *Median House Value, Median Income, Tot Bedrooms, Latitude, Longitude*. Only *Closest City* is a nominal categorical variable. In order to find out what categories exist in `Closest_city` column and how many districts belong to each category, we used the `value_counts()` method, that counts the occurrences of unique values within a series. It returned a new series with the unique values as the index and their respective counts as the values (table 2).

Closest City	Count
LA	9823
San Francisco	5054
San Jose	3764
San Diego	1999

Table 2: `housing['Closest_city'].value_counts()` output [11]

After, we selected only the quantitative variables and created a summary table of them (table 3), showing how many non-null values they have (20640) and the datatype (*int* or *float*).

#	Non-Null Count	Dtype
Median House Value	20640	float64
Median Income	20640	float64
Median Age	20640	int64
Tot Rooms	20640	int64
Tot Bedrooms	20640	int64
Population	20640	int64
Households	20640	int64
Latitude	20640	float64
Longitude	20640	float64

Table 3: `housing.select_dtypes(['int','float']).info()` [13]

In the following code, first, we set the Seaborn style to `whitegrid` to improve the overall appearance of our plots, as, according to us, a book can actually be judged by its cover. We then determined the number of histograms to create. To start creating the plots, we set up a figure and a set of subplots using the `plt.subplots()` function, where the `num_cols` value determines how many subplots (histograms) we need. Next, we used a loop to iterate through these subplots, pairing each subplot with a specific column from the DataFrame. We used the `zip` function to achieve this pairing. For each subplot, we created a histogram with 30 bins, a specific color from the Seaborn color palette (green water) and a title that describes the data it represents (e.g., "Distribution of `[column_name]`"). Additionally, we labeled the x-axis with the name of the column and the y-axis with "Frequency" to provide context for the plot.

```
sb.set_style('whitegrid')                                                    [16]

num_cols = len(housing.columns)
fig, axes = plt.subplots(nrows=num_cols, ncols=1, figsize=(15,100))

for ax, column in zip(axes, housing.columns):
    housing[column].hist(ax=ax, bins=30, color=sb.color_palette('Set2')[0],
    edgecolor='black')
    ax.set_title(f"Distribution of {column}", fontsize=14)
    ax.set_xlabel(column, fontsize=12)
    ax.set_ylabel("Frequency", fontsize=12)

plt.tight_layout()
plt.show()
```

Histograms provide us with a visual overview of data distribution, helping us in pattern recognition and outlier detection. We decided to focus in particular on the distribution of the median income by increasing the number of bins to 50 and plotting a boxplot (figure 1), in order to have a more precise representation of

the situation. This distribution is characterized by a positive skew, with a rightward tail that extends over a wide range of values. The presence of outliers on the higher end is evident due to the substantial difference between the mean and median, and the notably large standard deviation.

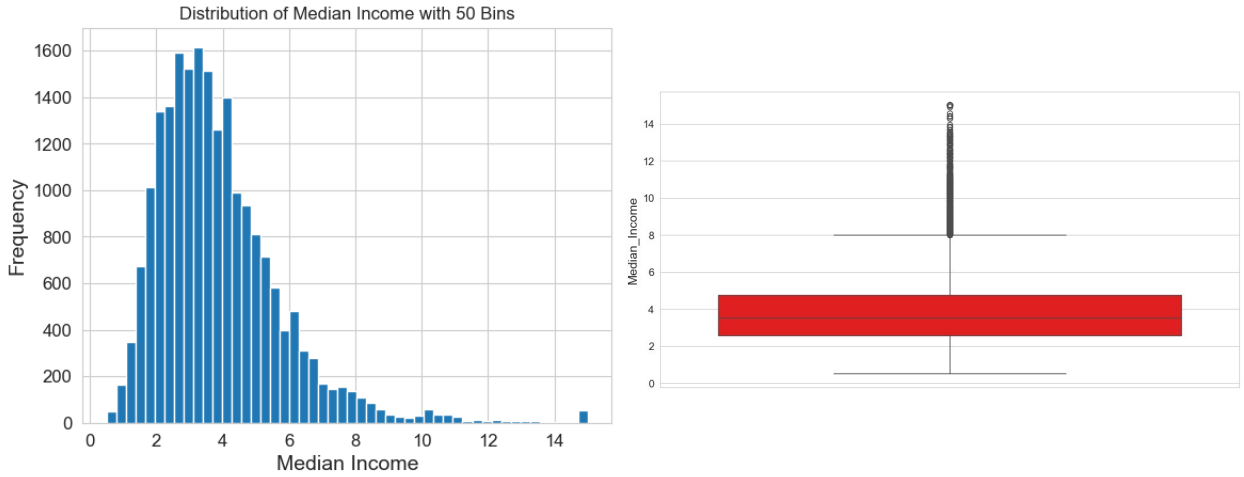

Figure 1: Median Income Histogram and Boxplot [17, 18]

After that, we firstly determined specific percentiles (quantiles) by creating a list called `cat`. We included the minimum and maximum values along with four quantiles (20th, 40th, 60th, and 80th percentiles) to divide the income data into five categories.

```
cat=[np.min(housing["Median_Income"])]                                    [19]
for i in [0.20, 0.40, 0.60, 0.80]:
    cat.append(housing["Median_Income"].quantile(i))
cat.append(np.max(housing["Median_Income"]))
print(cat)
```

Next, we used the `pd.cut()` function to create a new categorical column `income_cat` which contains 5 percentile ranges of the Median Income column. The `include_lowest` parameter ensures that the lowest category includes values equal to the minimum Median Income.

```
housing["income_cat"]=pd.cut(housing["Median_Income"], bins=cat,          [20]
                             labels = [1,2,3,4,5], include_lowest=True)
```

Subsequently, we counted the number of values in each income category to understand the distribution of the data among these categories, so that we finally created a histogram to visualize the distribution of the `income_cat` variable, which now represents Median Income divided into these distinct income categories. This process helped us better understand the distribution of income in the dataset.

The following code compares how different methods of creating test sets can reproduce the income category distribution of the complete dataset. After defining a function, `income_cat_proportions`, which calculates the proportion of each income category in a dataset, we divided the housing dataset into a training set and a test set using random sampling. Next, we built a dataframe, `compare_props`, to compare the distribution of income categories in the overall dataset, a stratified sample and a random sample. Finally, the code calculates the percentage error for each sampling method compared with the original distribution (table 4). This allowed us to see if one of the methods is more accurate in reproducing the original distribution of income categories.

```
def income_cat_proportions(data):                                    [26]
    return data["income_cat"].value_counts() / len(data)

train_set, test_set = train_test_split(housing, test_size=0.2,
                                       random_state=42)

compare_props = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(test_strat),
    "Random": income_cat_proportions(test_random),
}).sort_index()

compare_props["Rand. %error"] = 100 * compare_props["Random"]
                                  / compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"]
                                   / compare_props["Overall"] - 100
```

To assess the performance of machine learning models, a test set is crucial. It not only checks how well a model fits training data but also assesses its ability to generalize, preventing overfitting. Additionally, it aids in model optimization, allows model comparisons, and ensures consistent, reproducible results by keeping the test set separate and untouched during model development.
Before generating a test set, stratifying data on the income variable makes ensuring that the test set's income distribution is representative of the full dataset. This ensures that the test set is accurately representative and guards against biases brought on by unequal income distributions. Additionally, stratification offers consistent and repeatable train-test splits when used with a fixed seed, allowing accurate comparisons between models and settings.

income_cat	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.200097	0.200097	0.202762	1.331719	-1.421085e-14
2	0.200145	0.200097	0.203973	1.912370	-2.420721e-02
3	0.199758	0.199612	0.205184	2.716469	-7.276255e-02
4	0.200000	0.200097	0.198886	-0.557171	4.844961e-02
5	0.200000	0.200097	0.189196	-5.402132	4.844961e-02

Table 4: `compare_props` output [27]

Then, for safety, we copied the stratified train set to be used for modeling.

1.2 Data Visualization

We then created a scatter plot to visually represent each house in the dataset based on the geographical location (figure 2). `Longitude` and `Latitude` columns from the `houses_df` DataFrame were used as coordinates for each house in this 10x10 inches sized plot. Regarding the density of houses in particular areas, when dots overlap due to proximity of houses, the area appears darker on the plot. The resulting scatter plot provides insights into the spatial distribution of houses across different geographical zones. Given the latitude and longitude values provided, the scatter plot created from these coordinates resembles a map of California, albeit in a simplified form using points instead of detailed geographical boundaries. Densely populated areas, or areas with a higher concentration of houses, will be visually represented by darker clusters due to the overlapping of semi-transparent points. In contrast, lighter areas indicate regions with fewer houses. By looking at the plot, we can gain some insights into housing trends, socio-economic distributions, and potential urban development patterns relative to major cities.

```
plt.figure(figsize=(10,10))                                                    [29]
plt.scatter(houses_df['Longitude'],houses_df['Latitude'],alpha =0.2)
plt.title("Geographical Plot of Houses")
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.show()
```

The overlay of the scatter plot on the state's map allows for an intuitive understanding of housing trends across California (figure 3). The `california.png` image serves as a backdrop, allowing viewers to relate plotted data points directly with their actual geographical locations within the state of California. Each plotted point (or house) varies in size and color based on two distinct attributes: `Population` and `Median_House_Value`. The size of the points represents the population, making populous areas easily identifiable with larger dots. The color gradient gives a sense of house values, with the color spectrum representing a range from the minimum to the maximum median house value. The color bar on the side acts as a legend for house prices, and it's customized to display prices in thousands for better readability.

```
import warnings                                                                [30]
warnings.filterwarnings("ignore")

import matplotlib.image as mpimg
california_img=mpimg.imread("california.png")
ax = houses_df.plot(kind="scatter", x="Longitude", y="Latitude",
                        figsize=(14,10),
                        s=houses_df['Population']/100, label="Population",
                        c="Median_House_Value", cmap=plt.get_cmap("jet"),
                        colorbar=False, alpha=0.4)
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05],
           alpha=0.5,cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

prices = houses_df["Median_House_Value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar()
cbar.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values],
                        fontsize=14)
cbar.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
plt.show()
```

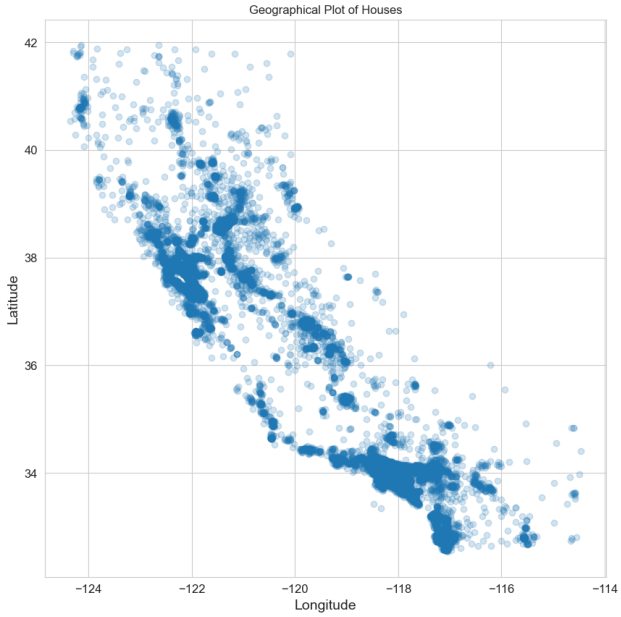

Figure 2: Scatterplot of California houses [29]

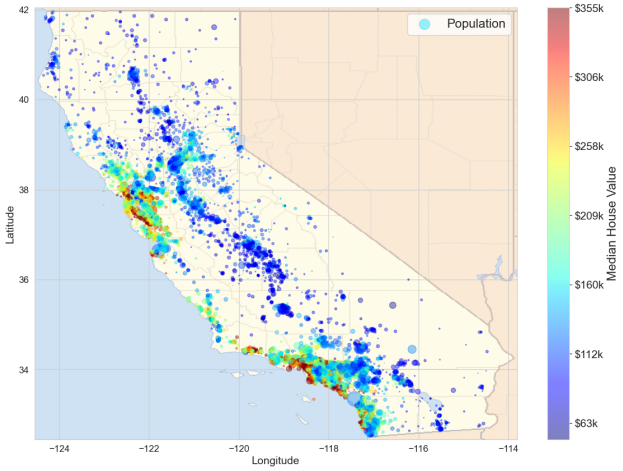

Figure 3: Scatterplot on map of California houses (with density) [30]

1.3 Bivariate Analysis

We first computed the correlation matrix of all the quantitative variables (table 5), which quantifies the linear relationship between variables, and named it as `houses_cor`.

	Median_House_Value	Median_Income	Median_Age	Tot_Rooms	Tot_Bedrooms
Median_House_Value	1.000000	0.689597	0.105611	0.136635	0.052198
Median_Income	0.689597	1.000000	-0.120037	0.202029	-0.007776
Median_Age	0.105611	-0.120037	1.000000	-0.359979	-0.318223
Tot_Rooms	0.136635	0.202029	-0.359979	1.000000	0.927990
Tot_Bedrooms	0.052198	-0.007776	-0.318223	0.927990	1.000000
Population	-0.023350	0.005136	-0.295396	0.852313	0.875293
Households	0.066550	0.013197	-0.300398	0.916679	0.980579
Latitude	-0.141518	-0.081302	0.013917	-0.033090	-0.063797
Longitude	-0.046849	-0.013227	-0.109430	0.040912	0.064818
Distance_to_coast	-0.466959	-0.243499	-0.221713	-0.001871	-0.023576

	Population	Households	Latitude	Longitude	Distance_to_coast
Median_House_Value	-0.023350	0.066550	-0.141518	-0.046849	-0.466959
Median_Income	0.005136	0.013197	-0.081302	-0.013227	-0.243499
Median_Age	-0.295396	-0.300398	0.013917	-0.109430	-0.221713
Tot_Rooms	0.852313	0.916679	-0.033090	0.040912	-0.001871
Tot_Bedrooms	0.875293	0.980579	-0.063797	0.064818	-0.023576
Population	1.000000	0.904064	-0.109085	0.099487	-0.041550
Households	0.904064	1.000000	-0.069316	0.052827	-0.063321
Latitude	-0.109085	-0.069316	1.000000	-0.925593	0.300766
Longitude	0.099487	0.052827	-0.925593	1.000000	0.007656
Distance_to_coast	-0.041550	-0.063321	0.300766	0.007656	1.000000

Table 5: `houses_cor` output [33]

The table above provided a comprehensive view of the linear relationship between all the quantitative variables. The values in this matrix range between -1 and 1, with 1 representing a perfect positive linear relationship, -1 a perfect negative linear relationship, and 0 showing the absence of linear correlation.

We then chose `Median_House_Value` as a pertinent outcome because `Median_Income` has a strong positive correlation (0.689597) with the target, indicating that areas with higher median incomes likely have higher house values. The correlations of all variables with the `Median_House_Value` (table 6) are extracted and sorted in descending order.

Variable	Correlation
Median_House_Value	1.000000
Median_Income	0.689597
Tot_Rooms	0.136635
Median_Age	0.105611
Households	0.066550
Tot_Bedrooms	0.052198
Population	-0.023350
Longitude	-0.046849
Latitude	-0.141518
Distance_to_coast	-0.466959

Table 6: `correlation_with_target` output [34]

`Median_House_Value` is a relevant output for a linear regression model, as it has significant correlations with certain input variables.
Subsequently, we generated the scatter plots for each couple of variable of the DataFrame (figure 4).

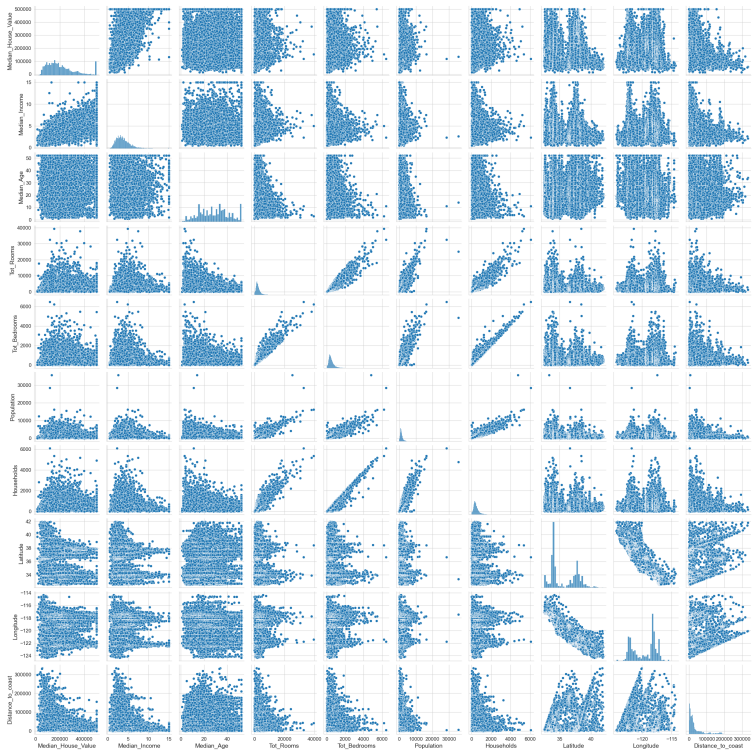

Figure 4: Scatter plots of each couple of variables [36]

There's a strong positive correlation between `Median_Income` and `Median_House_Value` and properties closer

to the coast tend to have higher values. `Tot_Rooms` and `Tot_Bedrooms` are also highly correlated, as districts with more rooms have more bedrooms. Regarding total number of rooms or bedrooms per district, they provide insights into the housing type and density in a district. However, both of the variables' correlation with the target variable is relatively weak, indicating that they might not be the primary drivers of house value. Thus, instead of just looking at the total number of rooms or bedrooms, metrics like rooms per household or bedrooms per household would offer a clearer picture of living conditions in each district. Hence, we created three new features into the dataset: `rooms_per_household`, `bedrooms_per_room` and `people_per_household`. With the new variables, we computed the correlation matrix again to evaluate how these newly formed variables correlate with existing ones, especially the target variable.

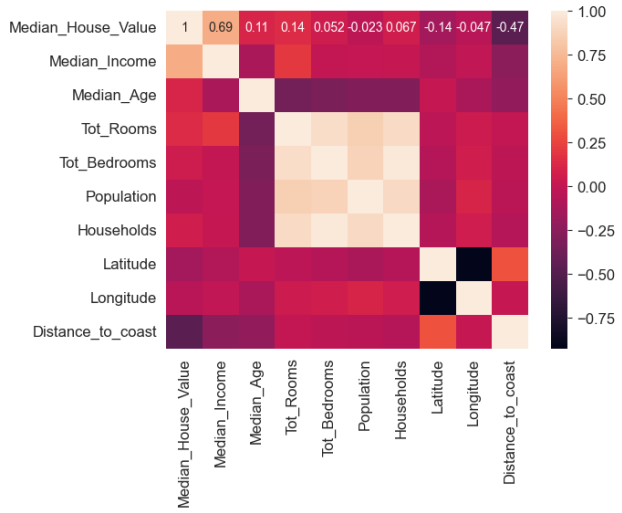

Figure 5: `sb.heatmap(houses_cor,annot=True)` output [39]

From the correlation table and the heatmap (5), we derived the following insights for the added variables:

- Rooms per Household: has a positive correlation of 0.146569 with `Median_House_Value`, which suggested that districts with more rooms per household might have a higher median house value.
- Bedrooms per Room: has a negative correlation of -0.253952 with the target variable. Districts with a higher ratio of bedrooms to rooms tended to have lower median house values.
- People per Household: the correlation of this feature with the target is very weak (-0.022265), which implied that the average household size in terms of population might not be a strong predictor of median house value on its own.

The predictors of `Median_House_Value` worthy to be paid attention to are `Median_Income` and `rooms_per_household`, as they are positively correlated with the target, implying they could be potential predictors. Conversely, `bedrooms_per_room` has a notable negative correlation, suggesting it might be a useful predictor as well. The total number of rooms `Tot_Rooms` has a weak positive correlation with the target. However, the `rooms_per_household` and `bedrooms_per_room` provided a more contextual and meaningful information, which captured the average living conditions in a district better than raw totals. We used Seaborn's heatmap function to visually represent the correlation matrix and the `annot=True` argument ensures that the correlation coefficients are displayed on the heatmap.

```
sb.heatmap(houses_cor,annot=True)                                    [39]
```

Looking at the heatmap, darker colors indicate strong negative correlations, while light color indicate strong positive correlations.

2 Preparing Data for Machine Learning

In this third phase, we proceeded to make sure that our data were ready and usable for the process of machine learning.

2.1 Missing Values

To create a dataset where there are 10% of missing values in one variable, we've firstly randomly selected 1651 indices from `houses_df`.

```
import random                                                    [40]
random.seed(42)
miss = np.random.choice(houses_df.index, 1651)
```

We then made a copy to make sure the original DataFrame doesn't get affected by changes, and, for all the rows that were randomly chosen, we set all the values to `None` in the `Tot_Bedroom` column. As expected, the column has missing values (table 7).

#	Column	Non-Null Count	Dtype	#	Column	Non-Null Count	Dtype
0	Median_House_Value	16512 non-null	float64	8	Longitude	16512 non-null	float64
1	Median_Income	16512 non-null	float64	9	Distance_to_coast	16512 non-null	float64
2	Median_Age	16512 non-null	int64	10	Closest_city	16512 non-null	object
3	Tot_Rooms	16512 non-null	int64	11	income_cat	16512 non-null	category
4	Tot_Bedrooms	14950 non-null	float64	12	rooms_per_household	16512 non-null	float64
5	Population	16512 non-null	int64	13	bedrooms_per_room	16512 non-null	float64
6	Households	16512 non-null	int64	14	people_per_household	16512 non-null	float64
7	Latitude	16512 non-null	float64				

Table 7: `houses_miss.info()` output [44]

When we have missing values, there are two main possibilities:
- dropping the rows associated to the missing values;
- estimating the missing values through an imputation method. In particular, the simplest and safest way is to use the median.

We firstly dropped all the rows associated with missing values by using dropna.

```
houses_drop = houses_miss.dropna(subset=["Tot_Bedrooms"])        [45]
```

We then computed the median of the `Tot_Bedrooms` column and filled the missing values in the original `houses_miss` DataFrame with this median value.

```
Bed_med = houses_miss["Tot_Bedrooms"].median()                   [47]
houses_miss["Tot_Bedrooms"].fillna(Bed_med, inplace=True)
```

After a quick check, the rows are confirmed to be back in `Tot_Bedrooms`.

A second possibility is to build a dataset with multiple missing values. We started by defining a function generating missing values in a chosen column of a DataFrame.

```
def col_miss (df, col, max_miss):                                    [47]

    random.seed(42)
    miss = np.random.choice(df.index, max_miss)
    df.loc[miss,col] = None

    return df
```

The `col_miss` function takes the DataFrame, column name, and the maximum number of missing values as input. The function uses a random process to select row indices and sets the corresponding values in the chosen column to `None`, representing missing data. The modified DataFrame is then returned, allowing for the introduction of random missing values in the specified column.
Again, we created a copy of the DataFrame to avoid complications, and then generated some missing values with our function in the first 10 predictors (from column index 0 included, to column index 10, excluded) of `housing_miss`. Having done this, we got a dataset with missing values in all the quantitative predictors. To find how many missing values there are in each variable we used the following functions.

```
housing_miss.isna().sum()                                            [53]
```

To do a multiple imputation we then used existing modules of scikit-learn. We tried firstly to use `SimpleImputer` to handle missing values in the DataFrame. It imputes the missing values by replacing them with the mean of their respective columns for all integer and float data types. The imputed data is then stored in a new DataFrame `X` with the same columns and index as the original `housing_miss` DataFrame, ensuring data consistency.

```
from sklearn.impute import SimpleImputer                             [55]

imputer = SimpleImputer(strategy='mean')

X = pd.DataFrame(imputer
                 .fit_transform(housing_miss.select_dtypes(['int','float'])))

selected_columns = housing_miss.select_dtypes(['int','float']).columns
X.columns = selected_columns
X.index = housing_miss.index
```

To check our results, we verified the correct imputation of the missing values by creating a DataFrame `X_df` from `X`.

Secondly, we used `KNNInputer` to impute missing data in each variable. It creates an imputer instance with a default of 5 nearest neighbors, applies it to the selected integer and float columns in the `housing_miss` DataFrame, and then assigns the same columns and index back to the imputed data, resulting in a DataFrame with missing values filled using a k-nearest neighbors approach.

```
from sklearn.impute import KNNImputer                                    [56]

knn_imputer = KNNImputer(n_neighbors=5)

housing_knn_imputed = pd.DataFrame(knn_imputer
                .fit_transform(housing_miss.select_dtypes(['int','float'])))

selected_columns = housing_miss.select_dtypes(['int','float']).columns
housing_knn_imputed.columns = selected_columns
housing_knn_imputed.index = housing_miss.index
```

Same as before, to check our results, we verified the correct imputation of the missing values by creating a DataFrame `h` from `X`.

`SimpleImputer` fills in missing values using a basic dataset statistic, such as mean, median or mode. It's fast, easy to understand, and works well when missing data are random or when their absence is not related to other variables. `KNNImputer`, on the other hand, uses the k-nearest neighbor algorithm to replace missing values. It takes into account relationships between features to estimate the best value for a missing input. This is often more accurate than the other, especially if the data have complex relationships or non-random missing value patterns. However, it is generally slower and requires more computational resources, so the choice depends on the dataset itself.

2.2 Categorical Variables

`Closest_city` is nominal, `income_cat` is ordinal. Nominal variables are like labels with no specific order, while ordinal variables are labels that have a clear hierarchy or sequence. In our case, `Closest_city` is nominal because San Francisco isn't "greater" or "less" than Los Angeles, they're just different names. `Income_cat` is ordinal because we have categories like "low income", "medium income", and "high income", there's a clear order from low to high. Hence, the two categorical variables are `Closest_city`, which is nominal and `income_cat`, which is ordinal.

2.2.1 One-Hot Encoding

One-hot encoding is a process where each category or modality of a categorical variable is converted into a new binary column. For each category, the new column will have a value of 1 if the original value matches the category and 0 otherwise.

```
from sklearn.preprocessing import OneHotEncoder as OHE                   [62]
onehot = OHE(sparse=False)
houses_onehot = onehot.fit_transform(houses_cat)
houses_onehot
```

```
array([[0., 0., 1., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.],
       [0., 0., 0., ..., 1., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 1.],
       [1., 0., 0., ..., 0., 0., 1.]])
```

The `OneHotEncoder` from the sklearn.preprocessing module is imported and aliased as OHE. An instance of `OneHotEncoder` is created with the parameter `sparse=False`, ensuring that the output is a dense array

rather than a sparse matrix. The `fit_transform` method is used to apply one-hot encoding to `houses_cat`, and the result is stored in `houses_onehot`.
The `onehot.categories_` attribute [64] is used to retrieve the categories that have been identified and encoded, and it gives the following output.

```
[array(['LA', 'SanDiego', 'SanFrancisco', 'SanJose'], dtype=object),
 array([1, 2, 3, 4, 5], dtype=int64)]
```

We can see that `Closest_city` has four modalities (*'LA', 'SanDiego', 'SanFrancisco', 'SanJose'*), while `income_cat` has five modalities (the five intervals that we have labeled 1,2,3,4,5). However, in the original values (`cat = [0.4999, 2.3523, 3.1406, 3.9669399999999997, 5.10972, 15.0001]`) these intervals are not equidistant, so we cannot really add nor substract them meaningfully. In short, both categorical variables should be considered nominal.

2.2.2 Building a Pipeline

A machine learning pipeline is a sequence of data processing and modeling steps that are designed to automate and streamline the process of developing, training, and deploying machine learning models.
Before building the pipeline, we needed to understand how it operates to help with the design, execution, and maintenance. We created a custom transformer to be used to add attributes. We did this to better integrate data in our pipeline and to make more complex machine learning tasks easier to process later. For this code we used two classes from the scikit-learn library: `BaseEstimator` to provide core functionality for all estimators in Scikit-Learn, like methods to set and get parameters, and `TransformerMixin` to provide the `fit transform` method, which combines the `fit` and `transform` operations. Here, sklearn used duck typing, not inheritance, which means that it does not strictly require objects to inherit from the mentioned base classes.

```
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6          [66]

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
```

The first line initializes four variables with integer values to represent column indices for the following attributes in the dataset: *rooms, bedrooms, population, households*. Then, we initiated our custom transformer `CombinedAttributesAdder`. The `add_bedrooms_per_room` boolean determines if the attribute `bedrooms_per_room` should be added, and the initializer sets the instance variable `self.add_bedrooms_per_room` based on the passed value or default. The `fit` method here does nothing but return the instance `self`. We used the `transform` method to check the `self.add_bedrooms_per_room`: if `True`, it calculates `bedrooms_per_room` (average number of bedrooms per room for each district) and then returns the original dataset (`X`) augmented with the three new attributes; if `False`, it returns the original dataset with only the first two

new attributes added. Then, an object `attr_adder` of the `CombinedAttributesAdder` class was created. The parameter `add_bedrooms_per_room` is set to `False`, which means that when this transformer is applied, it won't add the `bedrooms_per_room` attribute. Finally, we applied the transformation defined in the `CombinedAttributesAdder` to the data in `housing_df`. Specifically, it will add the attributes `rooms_per_household` and `population_per_household` to the dataset. The result of the transformation is stored in the `houses_plus` variable.

```
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)          [67]
houses_plus = attr_adder.transform(housing_df.values)
```

In short, for this step, we created a new copy of the original dataset, set up indices for specific columns for easy reference, defined a custom transformer which can add new calculated attributes, instantiated the transformer, and finally applied it to the dataset resulting in a new dataset with added attributes, stored in `houses_plus`. These codes enhanced our dataset by providing a more detailed representation of our data leading to better insights & data analysis and improved machine learning model performance.
Finally, we checked our answer by converting the data back into a structured DataFrame format for easier processing using Pandas. We transformed the data, assigned columns by appending new attribute names and using the index from the original `housing_df`, and finally we checked the new `houses_plus_df` to make sure that it was done properly.

For quantitative variables, we generated a pipeline by imputing missing values with the "median" method, adding two new attributes (rooms per household and population per household), and, in the end, standardizing the training set.
So, after isolating the quantitative attributes in the dataset, we broke down the `train` dataset into quantitative and categorical using `train_quanti = train[quanti_features]` to select only the quantitative attributes from the train dataset and store them in a new DataFrame. The two strings `closest_city` and `income_cat` are contained in the `cat_features` list.
We then initialized a new pipeline `quanti_pipeline`.

```
quanti_pipeline = Pipeline([                                               [73]
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

houses_quanti = quanti_pipeline.fit_transform(train_quanti)
```

The `SimpleImputer` class handles missing values in the dataset and the `strategy="median"` argument means it will replace the missing values with the median of the respective column. Then we added 3 attributes (see table 8) to the dataset (`rooms_per_household`, `population_per_household`, and `bedrooms_per_room`) and used the previously defined custom transformer `CombinedAttributeAdder` to compute them. Next, the `StandardScaler` scales the features to have a mean of 0 and a variance of 1. Finally, we used the `fit_transform` method to fit the pipeline to the data and then transform the data according to the learned parameters. The output here is stored in a new variable named `houses_quanti`.

#	0	1	2	3	4	5	6	7	8	9	10	11
0	0.054192	-1.509423	2.134192	2.806699	1.380344	2.680253	1.114801	-1.268340	-0.623034	2.494013	2.377987	-0.094520
1	1.826391	-1.671350	-0.289097	-0.146889	0.339490	-0.110597	0.813703	-1.120784	-0.183179	-0.203300	-0.171058	0.066281
2	-0.103194	0.028882	-0.099688	-0.239112	-0.250029	-0.288511	0.330990	0.125808	2.798050	-0.262003	-0.310642	-0.026174
3	-1.337175	-0.052081	0.183860	0.396746	-0.197986	0.340825	-0.763478	1.280813	0.993133	0.458445	0.402124	-0.096930
4	0.716497	0.190809	0.342173	0.207445	0.164518	0.197432	0.818482	-1.237811	-0.490040	0.127551	0.116761	-0.028703

Table 8: `houses_quanti_df.head()` output [74]

In summary, the pipeline `quanti_pipeline` took quantitative data, filled in any missing values with the median, added 3 computed attributes, and then standardized the resulting attributes, then the quantitative attributes in `train_quanti` is processed through the pipeline and the results are stored in `houses_quanti`.

Subsequently, we started by including the categorical variables in order to build a pipeline that can handle both quantitative and categorical variables of the dataset. We created a new transforamtion pipeline called `full_pipeline` using the `ColumnTransformer` class from the sklearn.compose library. `ColumnTransformer` applied `quanti_pipeline` to the columns listed in `quanti_features` and `OHE()` (one-hot-coding) to the columns listed in `cat_features`. The last line of code applied the entire `full_pipeline` to the train dataset and the processed data is then stored in the `houses_ready` variable that we just created.

```
from sklearn.compose import ColumnTransformer                    [76]

full_pipeline = ColumnTransformer([
        ("num", quanti_pipeline, quanti_features),
        ("cat", OHE(), cat_features),
    ])

houses_ready = full_pipeline.fit_transform(train)
```

To recap, this code allowed us to create a unified preprocessing pipeline to preprocess both quantitative and categorical attributes of the train dataset, and then we applied it to the dataset, resulting in `houses_ready`, which is a fully processed version.

3 Machine Learning

In this section we implemented, analyzed and tested various different machine learning models, in order to find the most suitable one for this dataset.

3.1 Training Set Only

We started from learning and evaluating with the training set alone.

3.1.1 Linear Regression

We started by training and testing a simple Linear Regression model. First, we initialized the Linear Regression model from the sklearn library and assisgned it to the variable `lr`. Then, we used the `train_set_split` function to split the dataset into training and testing sets. `X` represents the feature matrix and `y` is the target vector. We used `test_size=0.2` to specify that 20% of the data will be reserved for testing, while the remaining 80% will be used for training, while `random_state=42` ensures that the split is reproducible. By setting a constant seed (42), we are getting the same train-test split every time the code runs, which helps in consistent evaluations. Next, we trained the Linear Regression model using the training data `X_train` for features and `y_train` for the target. Once trained, we predicted the target values for the best set and the predictions are stored in the variable `y_pred`.

```
from sklearn.model_selection import cross_val_score                    [84]
from sklearn.linear_model import LinearRegression

lr  = LinearRegression()
lr.fit(X, y)

y_pred = lr.predict(X)
```

We then needed to estimate the performance, so we calculated the mean squared error, mean absolute error, root mean squared error and R^2. MSE measures the average squared difference between predictions and actual values, giving more weight to larger errors, making it sensitive to outliers. MAE calculates the average absolute difference between predictions and actual values, treating all errors equally and being less sensitive to outliers. RMSE measures the square root of the average of the squared differences between the predicted values and the actual values. Finally, R^2 measures the proportion of the variance in the dependent variable (the outcome) that can be explained by the independent variables (predictors) in the model.

```
mse = mean_squared_error(y, y_pred)                                    [86]
mae = mean_absolute_error(y, y_pred)
rmse = np.sqrt(mse)
```

3.1.2 Decision Tree Regression

Here, we trained a Decision Tree Regression model using the sklearn library. We started by initializing the *DecisionTreeRegressor* class and, as usual, by using the *random_state=42* parameter to ensure that the tree structure remains consistent every time we run the code. Next, we trained the model on the training data, and, finally, the predictions on the test set are stored in *y_pred_dt*.

```
dt_reg = DecisionTreeRegressor(max_depth=5)                            [89]
dt_reg.fit(X, y)

y_pred_dt = dt_reg.predict(X)
```

Similarly to what we did to the Linear Regression model, we calculated the mean squared error and mean absolute error to evaluate the performance of the model. Both the Linear Regression and Decision Tree Regression models yielded similar MSE values, with the Decision Tree having a slightly higher MSE. However, the Decision Tree Regression model outperformed Linear Regression in terms of MAE, suggesting that on average, the Decision Tree's predictions were closer to the actual values compared to those of the Linear Regression model.

3.2 Estimating with Cross Validation

First, we created a utility function named `display_scores` that takes in an array of scores and prints out scores, mean, and the standard deviation.

```
def display_scores(scores):                                            [92]
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())
```

The scores here are the raw scores from each fold of the cross-validation, the mean is the average of these scores, giving a central value that represents the overall performance across all the folds, and the standard deviation measures the amount of variation or dispersion in the scores.

3.2.1 Linear Regression

Next, we performed cross-validation on the Linear Regression model using the negative MSE as the scoring metric and then computed the RMSE for each fold. The RMSE values give a more interpretable measure of the model's performance on each fold since they are in the same unit as the target variable.

```
from sklearn.model_selection import cross_val_score                    [94]

lr_scores = cross_val_score(lr, X,y, scoring="neg_mean_squared_error", cv=10)
lr_rmse = np.sqrt(-lr_scores)
display_scores(lr_rmse)
```

Here, 10-fold cross validation is performed on the Linear Regression model. The model is trained and tested 10 times, each time on different subsets of `X` and `y` and we used the negative mean squared error to evaluate the performance. Then, we created `lr_rmse` to hold the RMSE values. To compute RMSE, the negative MSE values in `lr_scores` are made positive (`-lr_scores`) and the square root is taken for each value, giving the RMSE for each fold. Finally, we displayed the values to show the raw values, the mean, and the standard deviation of the Linear Regression model's performance across the 10 folds.

3.2.2 Penalized Linear Regression (Elastic Net)

First, we imported `ElasticNet` from sklearn.linear_model library and we instantiated an `ElasticNet` model object with a `random_state` parameter set to 42 to ensure reproducibility. This seed will be used in any random processes inside the algorithm and if we run the code multiple times, we will get the same results.

```
elastic_net = ElasticNet(random_state=42)                              [96]
```

Then, we performed cross validation on the `ElasticNet` using `cross_val_score` from the sklearn library. It basically splits the dataset into 10 folds and evaluates the model's performance on each fold. `X` represents the features and `y` represents the target variable of the dataset. Next, we evaluated the model using the negative mean squared error. Here, the mean squared error is negative because `cross_val_score` is designed to think that higher scores are better. However, for error metrics, lower is better. Finally, since the scores returned are negative, the scores were negated with `-en_scores` to get the positive mean squared error values for each fold, then the square root of those values are used to calculate the root mean squared error RMSE for each fold and the result is stored in `en_rmse`.

```
en_scores = cross_val_score(elastic_net, X, y,                         [97]
                            scoring="neg_mean_squared_error", cv=10)
en_rmse = np.sqrt(-en_scores)
display_scores(en_rmse)
```

3.2.3 Decision Tree

For the decision tree, we followed the same steps as the Elastic Net earlier.

```
dt_scores = cross_val_score(dt_reg, X, y,                                    [98]
                            scoring="neg_mean_squared_error", cv=10)
dt_rmse = np.sqrt(-dt_scores)
display_scores(dt_rmse)
```

Here, the scores are stored in `dt_rmse`.

3.2.4 Evaluation of Each Model

We started by fitting and training the 3 models on the dataset represented by `X` for features and `y` for target. After being trained, we made predictions on the same training set and then computed the RMSE for training predictions. For each model's predictions, the MSE between the predicted values and the actual target values is calculated, and the root of these MSE values is used to calculate the RMSE for each model (table 9).

```
lr.fit(X, y)                                                                 [99]
elastic_net.fit(X, y)
dt_reg.fit(X, y)

y_pred_train_lr = lr.predict(X)
y_pred_train_en = elastic_net.predict(X)
y_pred_train_dt = dt_reg.predict(X)

mse_train_lr = mean_squared_error(y, y_pred_train_lr)
mse_train_en = mean_squared_error(y, y_pred_train_en)
mse_train_dt = mean_squared_error(y, y_pred_train_dt)

rmse_train_lr = np.sqrt(mse_train_lr)
rmse_train_en = np.sqrt(mse_train_en)
rmse_train_dt = np.sqrt(mse_train_dt)

rmse_train_lr, rmse_train_en, rmse_train_dt
```

Model	Cross Validation RMSE	Training RMSE
Linear Regression	71,960.86	71,728.68
ElasticNet	77,326.83	77,328.36
Decision Tree	67,741.63	68,933.68

Table 9: Evaluation results [94, 97, 98, 99]

The training RMSE for the Linear Regression model is very close to the mean RMSE from cross-validation, indicating that the model isn't significantly overfitting and is fairly consistent on various subsets of the data. The training RMSE and the cross-validation RMSE for the Elastic Net model are nearly identical. This suggests that the Elastic Net model performs consistently both across various subsets of the data and across the entire dataset. In both cross-validation and training, the Decision Tree model has the lowest RMSE of the models tested. However, compared to the other models, its training RMSE and cross-validation RMSE difference is greater, indicating a little bit more variation in its performance. Although the Decision Tree regressor appears to have the lowest RMSE, we should be careful because decision trees are prone to overfitting. This may be suggested by the slight variation between the RMSEs for training and cross-validation. Elastic Net, a penalized linear regression, performs consistently but is less accurate on this dataset than the other models due to its higher RMSE. The performance of the standard linear regression

is in the middle; it is neither the best nor the worst of the three. Given these observations, even though the Decision Tree offers the lowest RMSE, it would be prudent to further examine its performance using a different test dataset. Overfitting may be present if the RMSE on the test data significantly increases. However, if its performance continues to be reliable, it might be the best model for this task.

3.2.5 Random Forest

We started by importing the proper library to initialize a `RandomForestRegressor`: we specified the number of trees in the forest to 100 to be created and each one will be trained on a different subset of the data and make its own predictions. Then, the random forest algorithm will average the predictions of all tress for regression tasks to create the final result.

```
from sklearn.ensemble import RandomForestRegressor                    [101]

rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)
```

Similarly to what we did for the previous evaluation of the models with cross validation, we used again the `cross_val_score` from the sklearn library.

```
rf_scores = cross_val_score(rf_reg, X, y,                             [102]
                            scoring="neg_mean_squared_error", cv=10)
rf_rmse = np.sqrt(-rf_scores)
display_scores(rf_rmse)
```

The Random Forest regressor's RMSE scores, as determined by 10-fold cross-validation, fall between 52,088 and 60,945. The Random Forests' average (mean) RMSE over the 10 folds is roughly 56,330. The RMSE scores for the Random Forests have a standard deviation of roughly 2,467, which represents the range of the RMSE scores over the 10 folds. The scores are more consistent across different subsets of the data when the standard deviation is lower, which indicates that the scores are closer to the mean.

3.2.6 Support Vector Machine

Then, we did the same for the Support Vector Machines using the `SVR` (support machine regression) Class from the sklearn's svm module.

```
svr_reg = SVR()                                                       [104]

svr_scores = cross_val_score(svr_reg, X, y,
                             scoring="neg_mean_squared_error", cv=10)
svr_rmse = np.sqrt(-svr_scores)
display_scores(svr_rmse)
```

The Support Vector Regression's RMSE scores, as determined by 10-fold cross-validation, fall between approximately 109,712 and 120,183. Approximately 115,462 is the average (mean) RMSE for the Support Vector Machines across the 10 folds. The Support Vector Machines' RMSE scores have a standard deviation of about 3,280, which indicates a reasonable degree of consistency across the various subsets but still a wider range than the Random Forests.

The Random Forest regressor outperforms the Support Vector Regression in terms of RMSE. This is clear from the fact that the Random Forest's mean RMSE is significantly lower than the Support Vector Regression's. The Random Forest model appears to perform more consistently across various subsets of

the data because its performance variability, as shown by the standard deviation, is lower than that of the Support Vector Machine. Overall, the Random Forest seems to be a better option for this dataset based on our metrics. However, thoughts like interpretability, training time, and other particular use-case requirements should also be considered when choosing a final model.

3.3 Tuning with Grid Search and Randomized Search

We've implemented 3 model examples: Random Forest, ElasticNet, and Decision Tree. For the three models, we followed the same approach. First, we created a dictionary to define the grid of hyperparameters to search over. Then, we set the parameters, used either grid or randomized search method, and set the number of folds for the cross-validation. The `scoring='r2'` here sets the metric to R^2 to evaluate the performance of the model. The `n_jobs=-1` tells the grid search to use all available cores on the machine to speed up the search process. Finally, we train the model with every possible combination of hyperparameters on the dataset using the `.fit()` method. Once done, the second line of the code (`grid_search`, `grid_search_en`, `random_search_cart`) will contain information about the best hyperparameters, the performance of each combination, and other important details.

3.3.1 Random Forest

```
rf_grid = {'n_estimators': [30,60,100],                                    [107]
           'max_features': [8,10,15]}
grid_search = GridSearchCV(rf_reg, rf_grid, cv=5, scoring='r2',
                           return_train_score=True,n_jobs=-1)

grid_search.fit(X, y)
```

3.3.2 Elastic Net

```
en_grid = {'alpha': np.logspace(-3, 4, 10),                                [110]
           'l1_ratio':np.linspace(0,1,11) }
elastic_net
grid_search_en = GridSearchCV(elastic_net, en_grid, cv=5, scoring='r2',
                              return_train_score=True,n_jobs=-1)

grid_search_en.fit(X, y
```

3.3.3 Decision Tree

```
cart_grid = {"min_samples_split": range(1,10),                             [114]
             "min_samples_leaf": range(1,60)}
random_search_cart = RandomizedSearchCV(dt_reg,param_distributions=cart_grid,
                                        n_iter=100, cv=5, scoring='r2',
                                        random_state=42,n_jobs=-1)

random_search_cart.fit(X, y)
```

3.3.4 Results of the Tuned Models

	Best Hyperparameters	Score of the Best Model
Random Forest	{'max_features': 10, 'n_estimators': 100}	0.751
Elastic Net	{'alpha': 0.036, 'l1_ratio': 0.8}	0.594
Decision Tree	{'min_samples_split': 8, 'min_samples_leaf': 8}	0.612

Table 10: Hyperparameters and model scores [108, 109, 111, 112, 115, 116]

3.4 Predicting Unseen Data

We first started by checking the structure of the test set, then we prepared our test set to be evaluated on the tuned models by dividing the datase into training and testing sets. `X_train` and `y_train` will be features and targets for the training set, respectively. `X_test` and `y_test` will be the features and targets for the testing set, respectively. The `train_test_split` function calls `X` and `y` as its primary arguments, where `X` is the feature matrix and `y` the target vector with a `test_size` of 0.2, which means that 20% of the data will be reserved for the test set and 80% for the training set.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)
```
[118]

Next, we estimated the performance of the data set on the best cross-validated model for the four following models: Random Forest, Elastic Net, Decision Tree, and Linear Regression.

For the following codes, we used `GridSearchCv` for the Random Forest and the Elastic Net and the `RandomizedSearchCv` for the Decision Tree, so the best model can be accessed via the `best_estimator` attribute. The best performing model is assigned to the variable we created here. Then, we made predictions on the test set and store them in the created variable `y_pred`.
The objective here was to compute the following evaluation metrics for each model then display them for comparison and analysis: MSE (`mean_squared_error`), MAE (`mean_absolute_error`), and R^2 (`r2_score`).

3.4.1 Random Forest

```
best_rf = grid_search.best_estimator_

y_pred = best_rf.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```
[120]

3.4.2 Elastic Net

```
best_en = grid_search_en.best_estimator_                    [121]
y_pred = best_en.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

3.4.3 Decision Tree

```
best_dt = random_search_cart.best_estimator_                [122]
y_pred = best_dt.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

3.4.4 Linear Regression

```
y_pred = lr.predict(X_test)                                 [123]

mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

3.4.5 Results

We then printed out our long-awaited results (11). Random Forest has the least MSE, or smallest average squared difference between predicted and actual values. It also has the lowest MAE, which means that its predictions deviate from the absolute values the least. The target variable's variability can be explained by the model to an extent of about 97.2%, according to the very high R^2 score (which is very close to 1). Compared to Random Forest, Elastic Net has a relatively high MSE and MAE. The target variable's variability is only explained by 62.6%, according to the R^2 score. In the Decision Tree model, MSE and MAE are greater than Random Forest but substantially lower than Elastic Net. According to the R^2 value, the model can account for roughly 66.6% of the variability. Regarding Linear Regression, he MSE and MAE are nearly identical to Elastic Net, indicating extremely similar predictive performance. Additionally, the R^2 score is very similar to that of Elastic Net as well.

	MSE	MAE	R^2 Score
Random Forest	360,067,718.86	12,743.93	0.9719
ElasticNet	4,788,859,732.76	50,298.24	0.6261
Decision Tree	4,276,865,036.76	47,161.47	0.6661
Linear Regression	4,789,827,626.57	50,246.59	0.6260

Table 11: Model metrics [120, 121, 122, 123]

In conclusion, the Random Forest model performs the best at predicting unobserved data based on the metrics offered. It has the best R^2 score and the lowest errors (both MSE and MAE), indicating good

accuracy and the capacity to explain a significant amount of the variance in the target variable. Both the ElasticNet and the Linear Regression models perform similarly poorly compared to the Random Forest model. With an R^2 score of 66.6%, the Decision Tree falls somewhere between the Random Forest and the linear models in terms of performance.

Although, in reality, it is also essentials to weigh other practical considerations, such as computational resources, model interpretability, the nature of data, etc., given these metrics above, the Random Forest model would be the best option regarding the prediction accuracy.

4 Conclusion

In this project, we embarked on a journey to predict housing prices using a variety of machine learning models. Our primary goal was to assess and compare the performance of these models on unseen data.

We began by meticulously preparing our dataset, addressing missing values and transforming features as necessary. Subsequent to this, we trained various models, namely Random Forest, ElasticNet, Decision Tree, and Linear Regression. To fine-tune these models and extract the most from their capabilities, we utilized techniques like Grid Search and Randomized Search. This helped in identifying the optimal set of hyperparameters for each model.

Upon predicting values with the unseen test data, the Random Forest model emerged as the clear winner in terms of prediction accuracy. Comparatively, both Elastic Net and Linear Regression exhibited similar performances, which were markedly lesser than that of Random Forest. Decision Tree, while not as proficient as Random Forest, still outperformed the linear models.

From a practical standpoint, while Random Forest's accuracy and ability to explain the variability in the housing prices make it a compelling choice, other considerations, such as computational efficiency, interpretability, and specific use cases, might lead to the selection of simpler models like Linear Regression or Decision Tree in certain scenarios.

To wrap up, this project underscores the importance of meticulous data preparation, model selection, and hyperparameter tuning in predictive analytics. It also emphasizes that while metrics can guide model selection, practical considerations play an equally pivotal role in determining the best fit for any given scenario. The Random Forest model's exemplary performance on this dataset sets a benchmark for predictive accuracy in housing price prediction. However, in real-world applications, a balanced approach considering both performance metrics and practicality is essential.