- **First**, the initial step is to look at the big picture: We have data about housing in California.
  - We have data regarding housing in California.
  - This data includes metrics such as population, median income, and median housing price.
  - Our task is to construct a model that predicts the median house price in a specific district based on certain metrics ("features").
- **Secondly**, we need to frame the problem:
  - This involves determining the objective business goal of the problem.
    - This step aids in choosing the algorithm, performance metrics, and the effort to be expended.
    - For example, this model can guide decisions on district-based company investments within a machine learning system.
  - We also need to investigate previous solutions for this problem.
    - This provides insights into potential problem-solving approaches.
    - Historically, house prices were estimated manually, which was both challenging and costly, often requiring complex formulas.
  - After addressing all of these, we can frame the problem:
    - It's supervised learning as we're using labeled data.
    - It's batch learning because the dataset is small, and there's no need for continuous learning from new data.
    - It's a regression problem as we're predicting a continuous value (price).
    - It's univariate regression since we're determining a single value.
    - It's multiple regression as we're using multiple metrics.
- **Thirdly**, we'll select the performance metric:
  - RMSE (Root Mean Square Error) is a suitable performance metric, commonly used with regression problems due to its emphasis on large errors.
  - However, MAE (Mean Absolute Error) is also effective as it handles outliers.

- **Fourthly**, we'll validate the assumptions made:
    - Reviewing assumptions, along with the big picture and our data, suggests that the assumptions are valid.

1. get data:

- I create a workspace that will operate using a Jupyter notebook, and I use Python 3.

First:

Download and extract the dataset in your local device from

https://raw.githubusercontent.com/ageron/handson-ml/master/datasets/housing/housing.tgz

then read it using pandas method read_csv

Read housing.csv as a dataframe called housing.

```
housing = pd.read_csv("housing.csv")
```

Second:

Write a simple function that gets the dataset directly from the website.

```
# import needed libraries
import os
import tarfile
import urllib
```

```
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

Use load_housing_data to create dataframe called housing.

```
fetch_housing_data()
housing = load_housing_data()
```

- I have read a dataset in two ways:
    - The first way involved downloading the data and then using **pd.read_csv()**.
    - The second way involved using the code from the reference to load data from GitHub.

2. Discover and visualize the data

A- Data discovery

Check the head of housing, and check out its info() and describe() methods.

1-Let's take a look at the top five rows using the DataFrame's head() method

```
housing.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

2-Use the info() method to get a quick description of the data

```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

3-Let's take a look at how many districts belong to "ocean_proximity" by using the value_counts() method

```
housing['ocean_proximity'].value_counts()
```

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

4-Let's look at the summary of the numerical attributes . Using the describe() method

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

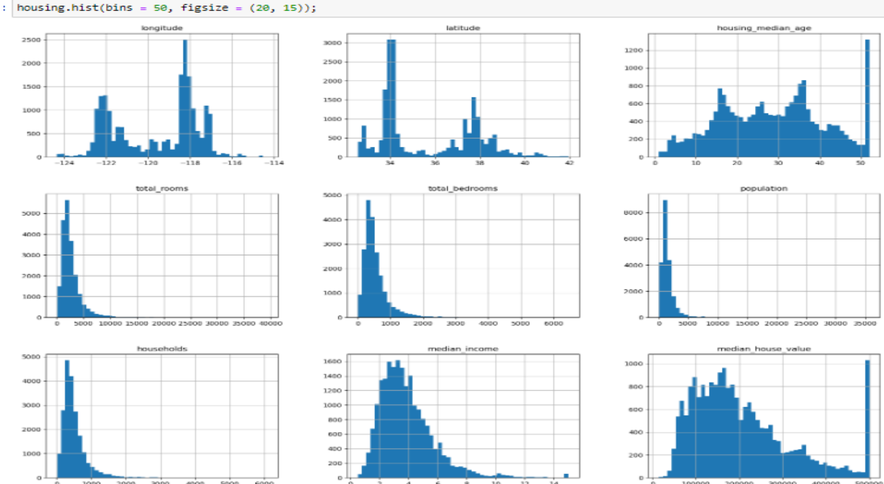
B- Data visualization

NOTE: ALL THE COMMANDS FOR PLOTTING A FIGURE SHOULD ALL GO IN THE SAME CELL. SEPARATING THEM OUT INTO MULTIPLE CELLS MAY CAUSE NOTHING TO SHOW UP.

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Create a hist plot for housing dataframe as shown down

```
housing.hist(bins = 50, figsize = (20, 15));
```



- I have explored the data structure using:
    - The **head** method, which allows me to view the first five instances of the data, inspect the metrics, and observe their values to understand what each metric represents.

    - The **info** method provides information such as the number of instances, the data type of each metric, the number of NaN values in each metric, and the overall size of the data. I also noted that all metrics are numerical except for **ocean_proximity**, which is an object; however, since I read it from a CSV file, I know that it represents text.

    - The **value_counts** method offers an overview of the frequency of each category in **ocean_proximity**.

    - The **describe** method provides statistical information about the data, such as count, mean, and standard deviation.

    - By utilizing histograms, I made several observations:
        - The **median_income** is capped between 0.5 and 15, scaled down in ten-thousandths. Actual values range from 5000 to 150000 dollars.

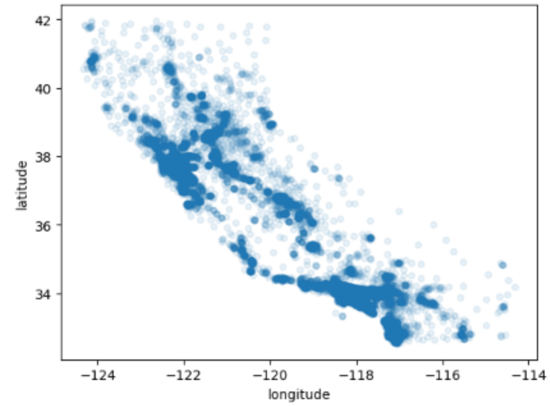
        - Similarly, **median_house_value** and **housing_median_age** are capped such that all values above 50 and 500000 are collected to 50 and 5000000, respectively.

        - There are disparities in scale among the data, which can be addressed through feature scaling.

        - Most attributes exhibit significant skewness, which can pose challenges for some models during training; therefore, we may need to transform them to achieve a normal distribution.

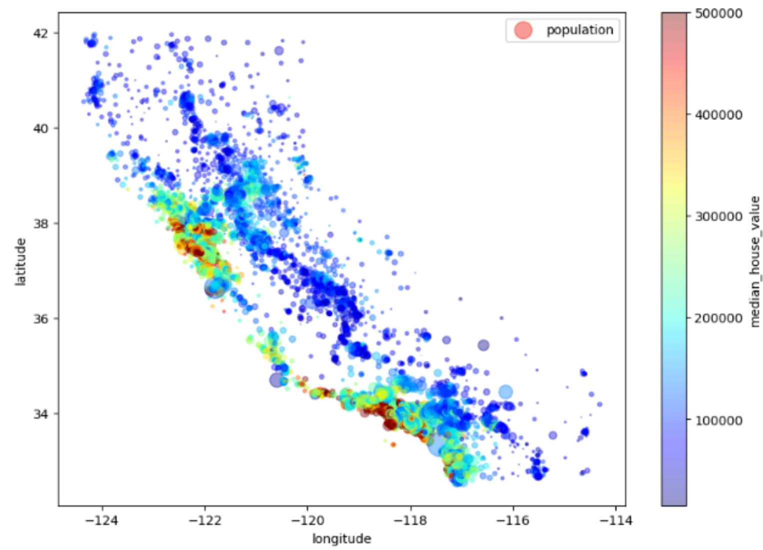
Create a scatter plot between "longitude" in x axis and "latitude" in y axis with alpha = 0.1

```
housing.plot(kind = 'scatter', x = 'longitude', y = 'latitude', alpha = .1);
```



Make The radius of each circle represent the district's population (option s), and the color represents the price (option c).

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4, s=housing["population"]/100, label="population",
            figsize = (10, 7), c="median_house_value", cmap="jet");
```



- Visualizing and exploring the data to gain insights:
  - Utilizing scatterplots, we can observe the locations with a high number of houses.

  - Employing scatterplots with two parameters, c and s:
    - The **c** parameter aids in visualizing the prices of houses within the scatterplot in a visually appealing manner using a colormap.

    - The **s** parameter assists in visualizing the distribution of people in different regions based on the size of each dot in the scatterplot.

Explore correlation between all continuous numeric variables using .corr() method.

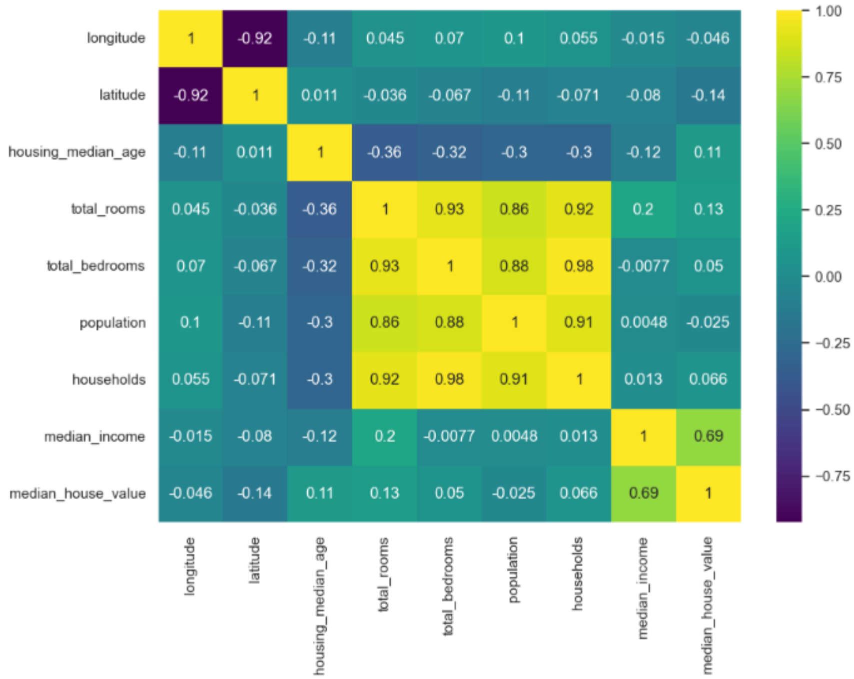
```
housing.corr()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
longitude	1.000000	-0.924664	-0.108197	0.044568	0.069608	0.099773	0.055310	-0.015176	-0.045967
latitude	-0.924664	1.000000	0.011173	-0.036100	-0.066983	-0.108785	-0.071035	-0.079809	-0.144160
housing_median_age	-0.108197	0.011173	1.000000	-0.361262	-0.320451	-0.296244	-0.302916	-0.119034	0.105623
total_rooms	0.044568	-0.036100	-0.361262	1.000000	0.930380	0.857126	0.918484	0.198050	0.134153
total_bedrooms	0.069608	-0.066983	-0.320451	0.930380	1.000000	0.877747	0.979728	-0.007723	0.049686
population	0.099773	-0.108785	-0.296244	0.857126	0.877747	1.000000	0.907222	0.004834	-0.024650
households	0.055310	-0.071035	-0.302916	0.918484	0.979728	0.907222	1.000000	0.013033	0.065843
median_income	-0.015176	-0.079809	-0.119034	0.198050	-0.007723	0.004834	0.013033	1.000000	0.688075
median_house_value	-0.045967	-0.144160	0.105623	0.134153	0.049686	-0.024650	0.065843	0.688075	1.000000

Use seaborn method to convert the correlation matrix to a heatmap plot

It's usually a better way to look for correlations among the features

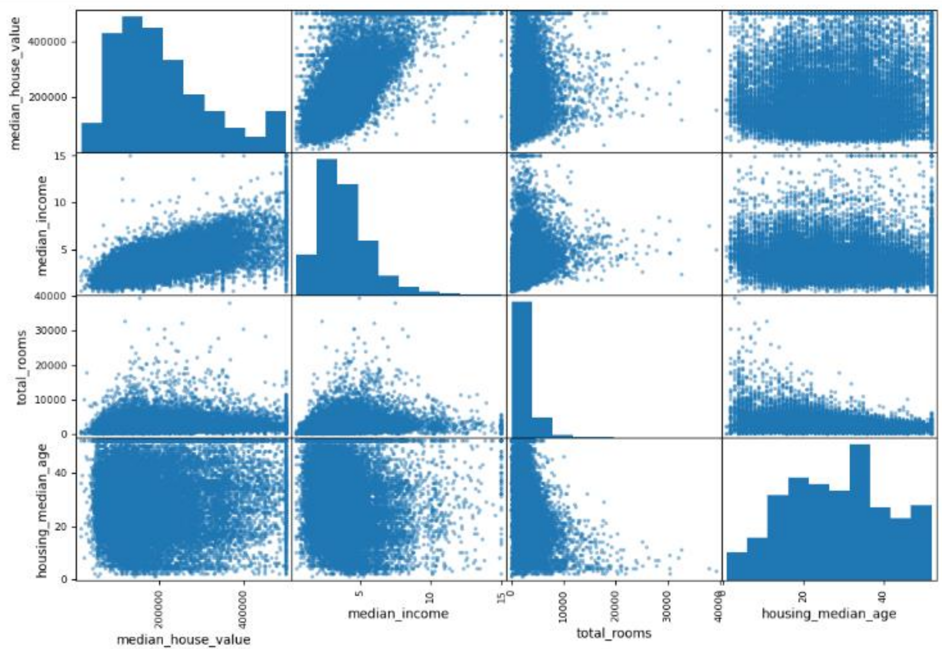
```
import seaborn as sb
sns.set(rc = {'figure.figsize':(10, 7)})
sb.heatmap(housing.corr(), annot = True, cmap = 'viridis');
```



Another way to check for correlation between attributes is to use the pandas scatter_matrix() function

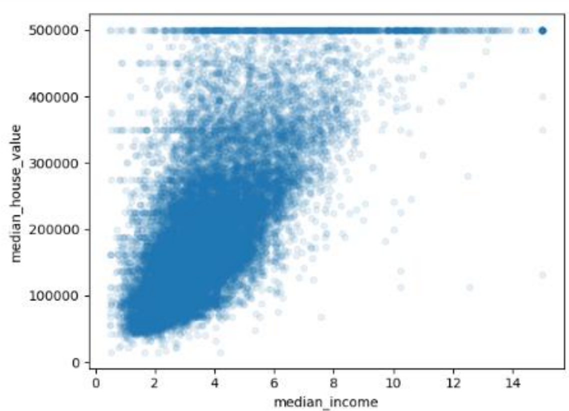
```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms","housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8));
```



Create a scatter plot between median_income and median_house_value

```
housing.plot(kind = "scatter", x = 'median_income', y = 'median_house_value', alpha = .1);
```



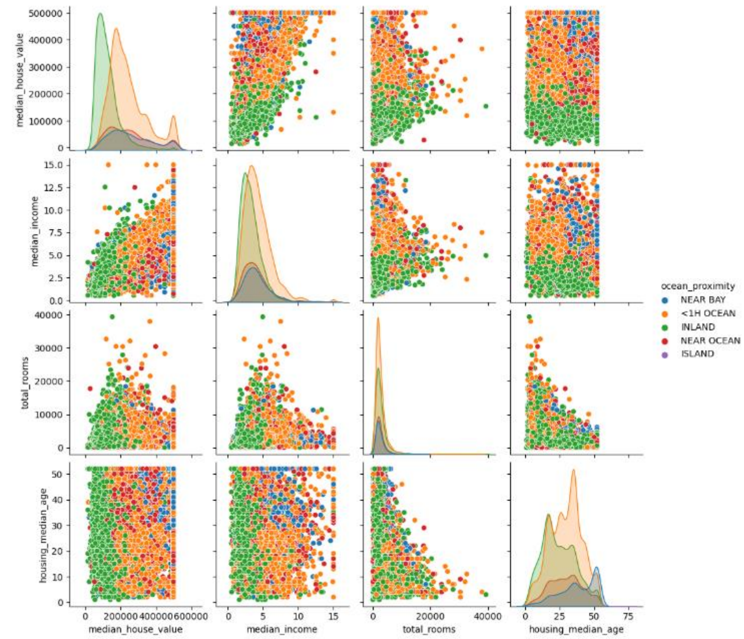
CHALLENGE

change the color of the plot based on ocean_proximity category

hint: use seaborn pairplot

```
attributes = ["median_house_value", "median_income", "total_rooms", "housing_median_age", "ocean_proximity"]
```

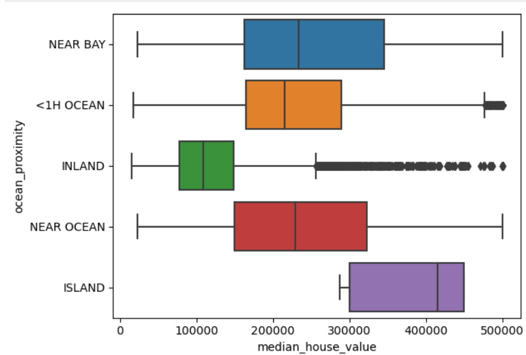
```
import seaborn as sb
sb.pairplot(housing[attributes], hue = 'ocean_proximity');
```



Create a boxplot to show the relation between median_house_value and the categorical feature ocean_proximity

```
import seaborn as sb

sb.boxplot(data = housing, x = "median_house_value", y = "ocean_proximity");
```



- I checked the correlation using different methods:
    - First, I created a correlation matrix, which displays the standard correlation coefficient "Pearson's r" between metrics.
        - Values close to 1 indicate a high positive correlation, values close to 0 indicate no correlation, and values close to -1 indicate a high negative correlation, implying that if one value is lower, the other tends to be higher.
        - Notably, there is a high correlation between **median_house_value** and **median_income**.
        - It's important to note that correlation measures linear relationships; there may be instances where the coefficient is 0, but there exists a quadratic or cubic relationship.
    - Second, I used a heatmap to visualize the correlation in a more aesthetically pleasing and clearer manner using a color map.
    - Third, I employed a scatter matrix, which illustrates scatter plots between specific attributes. From this scatter plot, it's evident that there's a strong correlation between **median_house_value** and **median_income**, indicated by a pronounced trend and minimal dispersion. Additionally, there are some horizontal lines at values of 500000, 430000, and 350000, which may introduce confusion during model training and should be addressed.
    - Fourth, I utilized a pairplot, which is similar to the previous method but separates scatter plot points based on **ocean_proximity**.
    - Finally box plot help us to view that there are many outilers in INLAND and few in < 1H OCEAN and no outliers in ISLAND and NEAR BAY with respect to median_house_value and it show us also the 75% of housing in ISLAND is less than or equal around 430000 which is the high cost and around 75% of houses in INLAND is less than or equal to 150000 which is the most cheaper.

3.Prepare the data

A- Data Cleaning

Create a Series that displays the total count of missing values per column.

```
housing.isna().sum()
```

```
longitude              0
latitude               0
housing_median_age     0
total_rooms            0
total_bedrooms       207
population             0
households             0
median_income          0
median_house_value     0
ocean_proximity        0
dtype: int64
```

the total_bedrooms attribute has some missing values. You have three options:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the values to some value (zero, the mean, the median, etc.).

You can accomplish these easily using DataFrame's dropna(), drop(), and fillna() methods:

Use the third option and fill the total_bedrooms null values with the median

```
# CODE HERE
median = housing['total_bedrooms'].median()
housing['total_bedrooms'].fillna(median, inplace = True)
```

Check if there any zeros in dataframe.

```
(housing == 0).sum(axis=0)
```

```
longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms        0
population            0
households            0
median_income         0
median_house_value    0
ocean_proximity       0
dtype: int64
```

Great there aren't any zeros in the data.

Zeros may sometimes be missing values so we need to take a closer look at them

B- Attribute Combinations

create new attribute called rooms_per_household between total_rooms and households

```
housing['rooms_per_household'] = housing['total_rooms'] / housing['households']
```

create new attribute called bedrooms_per_room between total_bedrooms and total_rooms

```
housing['bedrooms_per_room'] = housing['total_bedrooms'] / housing['total_rooms']
```

create new attribute called population_per_household between population and households

```
housing['population_per_household'] = housing['population'] / housing['households']
```

Now let's look for correlation again

```
housing.corr()['median_house_value'].sort_values(ascending = False)
```

```
median_house_value          1.000000
median_income               0.688075
rooms_per_household         0.151948
total_rooms                 0.134153
housing_median_age          0.105623
households                  0.065843
total_bedrooms              0.049457
population_per_household   -0.023737
population                 -0.024650
longitude                  -0.045967
latitude                   -0.144160
bedrooms_per_room          -0.233303
Name: median_house_value, dtype: float64
```

now let's remove old features("total_bedrooms", "total_rooms", "population", "households"), use drop method

Note: make inplace parameter True to save changes

```
new_housing = housing.copy()
new_housing.drop(["total_bedrooms", "total_rooms", "population", "households"], axis = 1, inplace = True)
```

```
new_housing.columns.value_counts()
```

```
longitude                   1
latitude                    1
housing_median_age          1
median_income               1
median_house_value          1
ocean_proximity             1
rooms_per_household         1
bedrooms_per_room           1
population_per_household    1
dtype: int64
```

- Now, after combining some attributes, I've observed that:
  - The correlation has improved. For instance, the correlation of **bedrooms_per_room** is greater than that of **bedrooms** alone. When the bedroom ratio is lower, the **median_house_value** tends to be higher, and the same holds true for all new features.
  - As a result, I no longer require the attributes "total_bedrooms", "total_rooms", "population", and "households", so I will remove them from the dataset.

C- Handling Text and Categorical Attributes

We have 5 classes in ocean_proximity.

To handle this categorical feature we create housing_cat that contain ocean_proximity.

```
print(housing["ocean_proximity"].unique())
```

```
['NEAR BAY' '<1H OCEAN' 'INLAND' 'NEAR OCEAN' 'ISLAND']
```

```
housing_cat = housing[["ocean_proximity"]]
```

Now use sklearn OneHotEncoder to fit and transform housing_cat.

```
from sklearn.preprocessing import OneHotEncoder

oneHotEncoder = OneHotEncoder()
housing_cat_1_hot_encoder = oneHotEncoder.fit_transform(housing_cat)
housing_cat_1_hot_encoder.toarray()
```

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

- I converted categorical attributes to numerical ones using the following rationale:
    - Machine learning algorithms inherently handle numerical values more effectively.
    - I employed the one-hot encoder technique, resulting in a sparse matrix (SciPy matrix). This matrix is 2D and stores a 1 for specific categories and 0 for others.
    - One-hot encoding was deemed suitable for this scenario over ordinal encoding. While ordinal encoding may struggle to distinguish between two numbers that are close in value, it can be effective in cases like "bad," "average," "good," "excellent," etc.

Custom Transformers

Here is a small transformer class that adds the combined attributes wediscussed earlier:

```python
from sklearn.base import BaseEstimator, TransformerMixin

# get the right column indices: safer than hard-coding indices 3, 4, 5, 6
rooms_ix, bedrooms_ix, population_ix, household_ix = [
    list(housing.columns).index(col)
    for col in ("total_rooms", "total_bedrooms", "population", "households")]

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True):
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_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]
```

Use the above class (CombinedAttributesAdder) to create instance called attr_reader, then transform housing values and save them in a variable called housing_extra_attribs.

```python
attr_adder            = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

```python
print(housing_extra_attribs)
```

```
[[-122.23 37.88 41.0 ... 2.5555555555555554 6.984126984126984
  2.5555555555555554]
 [-122.22 37.86 21.0 ... 2.109841827768014 6.238137082601054
  2.109841827768014]
 [-122.24 37.85 52.0 ... 2.8022598870056497 8.288135593220339
  2.8022598870056497]
 ...
 [-121.22 39.43 17.0 ... 2.325635103926097 5.20554272517321
  2.325635103926097]
 [-121.32 39.43 18.0 ... 2.1232091690544412 5.329512893982808
  2.1232091690544412]
 [-121.24 39.37 16.0 ... 2.616981132075472 5.254716981132075
  2.616981132075472]]
```

- Here, we've developed our own transformer instead of relying on a scikit-learn transformer:
    - This approach aids in data cleanup and attribute combination and is distinct from scikit-learn transformers.
    - At this stage, we're uncertain whether adding the **bedrooms_per_room** attribute will improve performance, so we've included a hyperparameter to toggle its addition or removal.

Transformation Pipelines

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
```

Create your own pipline for numerical attributes. It should contain SimpleImputer, CombinedAttributesAdder, and StandardScaler. call it num_pipeline.

```
num_pipeline = Pipeline([
               ("imputer", SimpleImputer(strategy = "median")),
               ("attr_adder", CombinedAttributesAdder()),
               ("std_scaler", StandardScaler())
               ])
```

now create a full pipeline called full_pipeline , use num_pipeline for numerical attributes and OneHotEncoder for catigorcal attributes .

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
```

```
full_pipeline = ColumnTransformer([
                ("num", num_pipeline, num_attribs),
                ("cat", OneHotEncoder(), cat_attribs)
                ])
```

fit and transform full_pipeline with housing data then saved it in housing_prepared

```
housing_prepared = full_pipeline.fit_transform(housing)
```

```
print(housing_prepared[:5])
```

```
[[-1.32783522  1.05254828  0.98214266 -0.8048191  -0.97247648 -0.9744286
  -0.97703285  2.34476576  2.12963148  0.62855945 -1.02998783 -0.04959654
   0.62855945 -0.04959654 -1.02998783  0.          0.          0.
   1.          0.        ]
 [-1.32284391  1.04318455 -0.60701891  2.0458901   1.35714343  0.86143887
   1.66996103  2.33223796  1.31415614  0.32704136 -0.8888972  -0.09251223
   0.32704136 -0.09251223 -0.8888972   0.          0.          0.
   1.          0.        ]
 [-1.33282653  1.03850269  1.85618152 -0.53574589 -0.82702426 -0.82077735
  -0.84363692  1.7826994   1.25869341  1.15562047 -1.29168566 -0.02584253
   1.15562047 -0.02584253 -1.29168566  0.          0.          0.
   1.          0.        ]
 [-1.33781784  1.03850269  1.85618152 -0.62421459 -0.71972345 -0.76602806
  -0.73378144  0.93296751  1.16510007  0.15696608 -0.4496128  -0.0503293
   0.15696608 -0.0503293  -0.4496128   0.          0.          0.
   1.          0.        ]
 [-1.33781784  1.03850269  1.85618152 -0.46240395 -0.61242263 -0.75984669
  -0.62915718 -0.012881    1.17289952  0.3447108  -0.63908657 -0.08561576
   0.3447108  -0.08561576 -0.63908657  0.          0.          0.
   1.          0.        ]]
```

- I've constructed my own pipeline, which streamlines the entire transformation process:
  - Pipelines are invaluable as they consolidate all transformations into a single location.
  - Specifically, I've created a **num_pipeline** that executes all the transformations we've previously defined.
  - To apply these transformations to all columns, including both numerical and categorical ones, I've utilized **ColumnTransformer**. For numerical columns, we apply the **num_pipeline** we've constructed earlier, while for categorical columns, we employ the one-hot encoder.
  - It's worth noting that while **num_pipeline** returns a dense matrix and the one-hot encoder returns a sparse one, we combine both and compare them with a threshold to determine whether the final matrix should be stored as sparse or dense. By default, the sparse threshold is set to 0.3, resulting in a dense matrix.

4. Create a Test Set and Train Set

4- Create a Test Set and Train Set

Use model_selection.train_test_split from sklearn to split the data into training and testing sets.

Note: Set random_state=42 to get the same result

```
from sklearn.model_selection import train_test_split

start_train_set, start_test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Let's also separate the predictors and the labels to housing and housing_labels

(note that drop() creates a copy of the data and does not affect strat_train_set):

```
housing_labels = start_train_set["median_house_value"].copy()
housing        = start_train_set.drop("median_house_value", axis = 1)
```

- Finally, I've generated a training and test set:
    - I've employed **train_test_split** from sklearn, adhering to the convention of splitting the data into 20% test and 80% training sets. Additionally, I've set the random state to 42 to ensure consistent splits if the cell is run again.
    - Subsequently, I created the labels by copying the **median_house_value** column from the housing dataset, and predictors were generated by dropping the **median_house_value** column.