# MLPy Workshop 1

January 21, 2026

## 1 Week 1: Exploratory Data Analysis and Feature Engineering

**Darren Lim**

### 1.1 Aims

By the end of this notebook you will

- understand and play with the different aspects of data pre-processing
- be familiar with tools for exploratory data analysis and visualization
- understand the basics of feature engineering
- build your first pipeline

### 1.2 Topics and Instructions

1. Problem Definition and Setup
2. Exploratory Data Analysis
3. Data Preprocessing
4. Feature Engineering
5. Summary

In lecture this week, we reviewed the general **machine learning pipline**, which following the "Machine Learning Project Checklist" of Geron (2019) can be stuctured as:

> - Frame the problem and look at the big picture.
> - Get the data.
> - Explore the data and gain insights.
> - Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
> - Explore many different models and shortlist the best ones.
> - Fine-tune your models and combine them into a great solution.
> - Present your solution.
> - Launch, monitor, and mantain your system.

In this week's workshop, we will focus on the initial steps of this pipeline, that is on, data pre-processing, exploratory data analysis and feature engineering.

During workshops, you will complete the worksheets together in teams of 2-3, using **pair programming**. During the first few weeks, the worksheets will contain cues to switch roles between driver and navigator. When completing worksheets:

- You will have tasks tagged by (CORE) and (EXTRA).
- Your primary aim is to complete the (CORE) components during the WS session, afterwards you can try to complete the (EXTRA) tasks for your self-learning process.
- Look for the as cue to switch roles between driver and navigator.
- In some Exercises, you will see some beneficial hints at the bottom of questions.

Instructions for submitting your workshops can be found at the end of worksheet. As a reminder, you must submit a pdf of your notebook on Learn by 16:00 PM on the Friday of the week the workshop was given.

# 2 Problem Definition and Setup

## 2.1 Packages

Now lets load in some packages to get us started. The following are widely used libraries to start working with Python in general.

[1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
```

If you need to install any packages from scratch, you need to install the related library before calling it. For instance, feature-engine is a Python library for Feature Engineering and Selection, which:

- contains multiple transformers to engineer and select features to use in machine learning models.
- preserves scikit-learn functionality with methods fit() and transform() to learn parameters from and then transform the data (we will learn more about these throughout the course!).

[2]:
```
# To install the feature-engine library (if not already installed)
# !pip install feature-engine
```

In some cases, we may need only a component of the whole library. If this is the case, it is possible to import specific things from a module (library), using the following line of code:

[3]:
```
from feature_engine.imputation import EndTailImputer
```

## 2.2 Problem

Now, it is time move on to the next step.

> You are asked to build a model of housing prices in California using the California census data. This data has metrics such as the population, median income, median housing price, and so on for each block group in California. Block groups are the

smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will just call them "districts" for short.

**Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.**

The first question to ask your boss is what exactly is the business objective; building a model is probably not the end goal. **How does the company expect to use and benefit from this model?** This is important because it will determine how you frame the problem, what algorithms you will select, what performance measure you will use to evaluate your model, and how much effort you should spend tweaking it.

The next question to ask is **what the current solution looks like (if any)**. It will often give you a reference performance, as well as insights on how to solve the problem. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules.

This is costly and time-consuming, and their estimates are not great; in cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 20%. This is why the company thinks that it would be useful to train a model to predict a district's median housing price given other data about that district. The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

---

### 2.2.1 Exercise 1 (CORE)

Using the information above answer the following questions about how you may design your machine learning system.

a) Is this a supervised or unsupervised learning task?

This is a supervised learning task.

b) Is this a classification, regression, or some other task?

This is a regression task.

c) Suppose you are only required to predict if a district's median housing prices are "cheap," "medium," or "expensive". Will this be the same or a different task?

This will be a different task: Classification.

## 2.3 Data Download

The data we will be using this week is a modified version of the California Housing dataset. We can get the data a number of ways. The easiest is just to load it from the working directory that we are working on (where we have already downloaded it to).

[4]:
```python
housing = pd.read_csv("housing.csv")
display(housing.head(10))
display(housing.tail(10))
```

```
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -122.23     37.88                41.0        880.0           129.0
1    -122.22     37.86                21.0       7099.0          1106.0
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0
5    -122.25     37.85                52.0        919.0           213.0
6    -122.25     37.84                52.0       2535.0           489.0
7    -122.25     37.84                52.0       3104.0           687.0
8    -122.26     37.84                42.0       2555.0           665.0
9    -122.25     37.84                52.0       3549.0           707.0

   population  households  median_income  median_house_value ocean_proximity
0       322.0       126.0         8.3252            452600.0        NEAR BAY
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
5       413.0       193.0         4.0368            269700.0        NEAR BAY
6      1094.0       514.0         3.6591            299200.0        NEAR BAY
7      1157.0       647.0         3.1200            241400.0        NEAR BAY
8      1206.0       595.0         2.0804            226700.0        NEAR BAY
9      1551.0       714.0         3.6912            261100.0        NEAR BAY
```

```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
20630    -121.32     39.29                11.0       2640.0           505.0
20631    -121.40     39.33                15.0       2655.0           493.0
20632    -121.45     39.26                15.0       2319.0           416.0
20633    -121.53     39.19                27.0       2080.0           412.0
20634    -121.56     39.27                28.0       2332.0           395.0
20635    -121.09     39.48                25.0       1665.0           374.0
20636    -121.21     39.49                18.0        697.0           150.0
20637    -121.22     39.43                17.0       2254.0           485.0
20638    -121.32     39.43                18.0       1860.0           409.0
20639    -121.24     39.37                16.0       2785.0           616.0

       population  households  median_income  median_house_value  \
20630      1257.0       445.0         3.5673            112000.0
20631      1200.0       432.0         3.5179            107200.0
20632      1047.0       385.0         3.1250            115600.0
20633      1082.0       382.0         2.5495             98300.0
20634      1041.0       344.0         3.7125            116800.0
20635       845.0       330.0         1.5603             78100.0
20636       356.0       114.0         2.5568             77100.0
```

```
20637       1007.0       433.0          1.7000               92300.0
20638        741.0       349.0          1.8672               84700.0
20639       1387.0       530.0          2.3886               89400.0

      ocean_proximity
20630          INLAND
20631          INLAND
20632          INLAND
20633          INLAND
20634          INLAND
20635          INLAND
20636          INLAND
20637          INLAND
20638          INLAND
20639          INLAND
```

# 3 Exploratory Data Analysis

In this section we are going to start with exploring the California Housing data using methods that you will likely already be familiar with.

Data can come in a broad range of forms encompassing a collection of discrete objects, numbers, words, events, facts, measurements, observations, or even descriptions of things. Processing data using exploratory data analysis (EDA) can elicit useful information and knowledge by examining the available dataset to discover patterns, spot anomalies, test hypotheses, and check assumptions.

Let's start by examining the Data Dictionary and the variables available:

> `longitude`: A measure of how far west a house is; a higher value is farther west
>
> `latitude`: A measure of how far north a house is; a higher value is farther north
>
> `housingMedianAge`: Median age of a house within a block; a lower number is a newer building
>
> `totalRooms`: Total number of rooms within a block
>
> `totalBedrooms`: Total number of bedrooms within a block
>
> `population`: Total number of people residing within a block
>
> `households`: Total number of households, a group of people residing within a home unit, for a block
>
> `medianIncome`: Median income for households within a block of houses (measured in tens of thousands of US Dollars)
>
> `medianHouseValue`: Median house value for households within a block (measured in US Dollars)
>
> `oceanProximity`: Location of the house w.r.t ocean/sea

### 3.0.1 Exercise 2 (CORE)

a) Examine the datatypes for each column calling `info()`. What is the total number of observations and total number of variables? What is the type of each variable?

[5]:
```
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
```

Total number of observations = 20640

Total number of variables = 10

longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, and median_house_value are float data types, while ocean_proximity is a string data type.

b) From the information provided above, can you anticipate any data cleaning we may need to do?

We may need to do one-hot encoding of the variable ocean_proximity, as it is a categorical variable.

There are some missing values under the total_bedrooms variable, which may need to be dropped.

### 3.0.2 Exercise 3 (CORE)

a) Use descriptive statistics and histograms to examine the distributions of the numerical attributes.
Hint

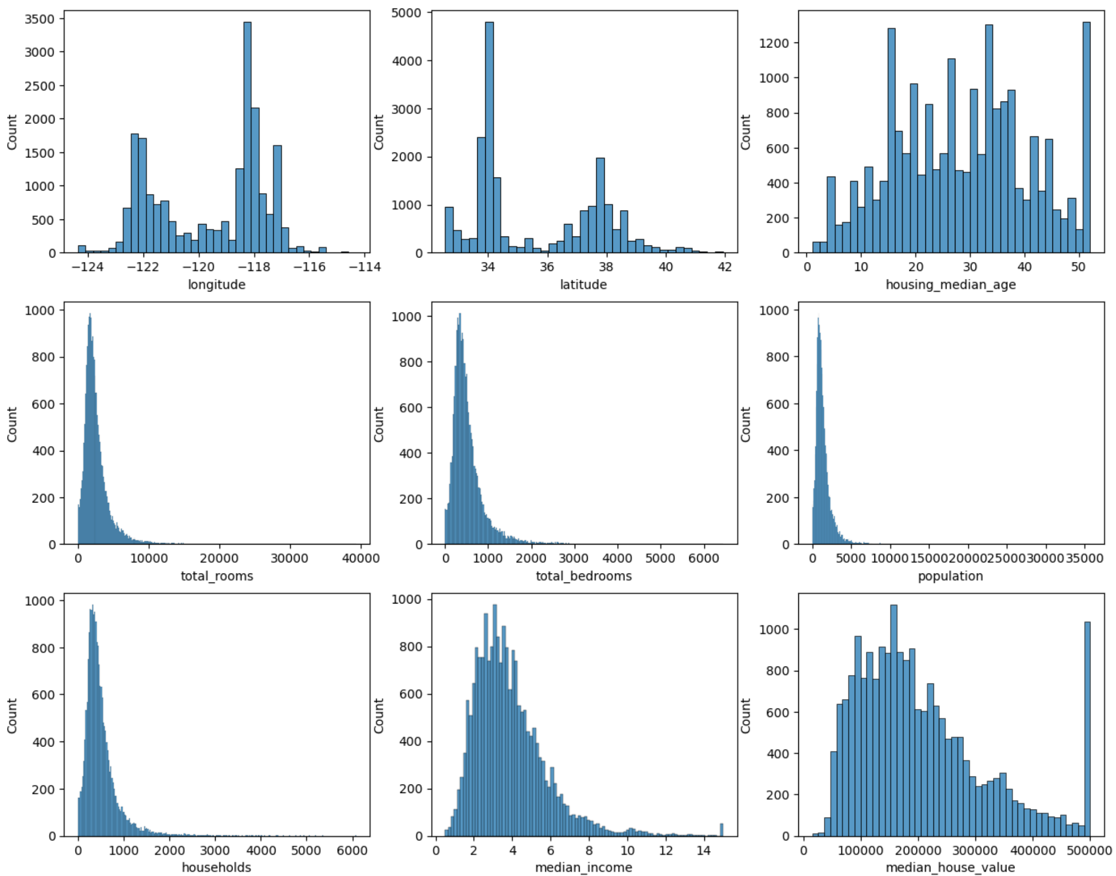
- .describe() can be used to create summary descriptive statistics on a pandas dataframe.
- You can use a sns.histplot to create histograms

[6]:
```
display(housing.describe())

plt.figure(figsize=(15, 12))
```

```
plt.subplot(331)
sns.histplot(housing["longitude"])
plt.subplot(332)
sns.histplot(housing["latitude"])
plt.subplot(333)
sns.histplot(housing["housing_median_age"])
plt.subplot(334)
sns.histplot(housing["total_rooms"])
plt.subplot(335)
sns.histplot(housing["total_bedrooms"])
plt.subplot(336)
sns.histplot(housing["population"])
plt.subplot(337)
sns.histplot(housing["households"])
plt.subplot(338)
sns.histplot(housing["median_income"])
plt.subplot(339)
sns.histplot(housing["median_house_value"])
display(plt.show())
```

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

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

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

```
max          500001.000000
```




```
None
```

b) Can you identify other pre-processing/feature engineering steps we may need to do? Which variables represent counts and how are they distributed?

total_rooms, total_bedrooms, population, and households represent counts. total_rooms, total_bedrooms, population, and households appears have a left-skewed, asymmetrical distribution. We may need to do a log(x+1) transformation of the data to give a more symmetrical distribution.

c) One thing you may have noticed from the histogram is that the median income, housing median age, and the median house value are capped. The median house value capping (this being our target value) may or may not be a problem depending on your client. If we needed precise predictions beyond $\$500,000$, we may need to either collect proper labels/outputs for the districts whose labels were capped or remove these districts from the data. Following the latter, remove all districts whose median house value is capped. How many observations are there now?

[7]:
```
# Remove the cases where median_house_value >= 500,000$
```

```
housing_uncapped = housing[housing.apply(lambda x: x["median_house_value"] <
    500000, axis=1)]
display(housing_uncapped.info())
# The number of observations has reduced from 20640 to 19648.
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 19648 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           19648 non-null  float64
 1   latitude            19648 non-null  float64
 2   housing_median_age  19648 non-null  float64
 3   total_rooms         19648 non-null  float64
 4   total_bedrooms      19448 non-null  float64
 5   population          19648 non-null  float64
 6   households          19648 non-null  float64
 7   median_income       19648 non-null  float64
 8   median_house_value  19648 non-null  float64
 9   ocean_proximity     19648 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
None
```

### 3.0.3 Exercise 4 (CORE)

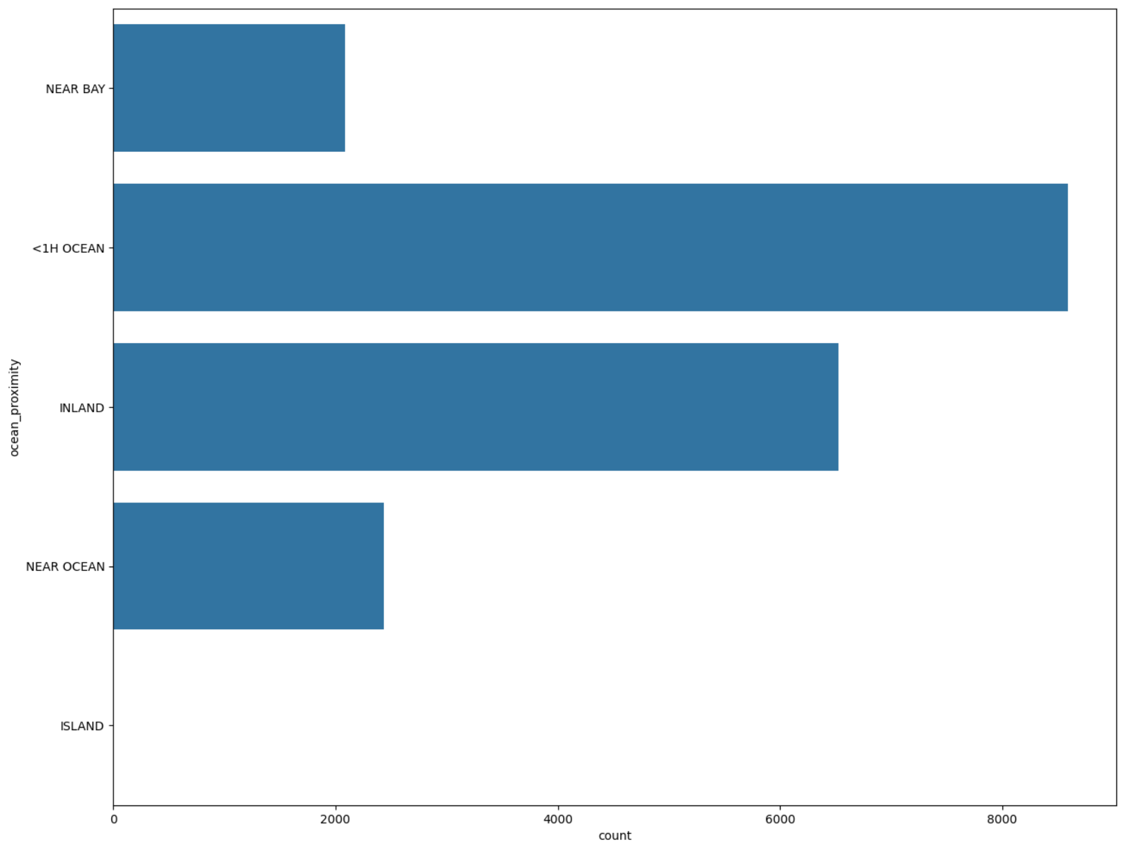
What are the possible categories for the `ocean_proximity` variable? Are the number of instances in each category similar?

Hint

- value_counts() can be used to count the values of the categories a pandas series.
- You can use a sns.countplot to create barplot with the number of instances of each category

[8]:
```
display(housing_uncapped["ocean_proximity"].value_counts())
plt.figure(figsize=(15,12))
sns.countplot(housing_uncapped["ocean_proximity"])
display(plt.show())
```

```
ocean_proximity
<1H OCEAN     8595
INLAND        6523
NEAR OCEAN    2437
NEAR BAY      2088
ISLAND           5
Name: count, dtype: int64
```




```
None
```

There are 5 possible categories under the ocean_proximity variable.

The number of instances varies among categories.

**Now, is a good point to switch driver and navigator**

### 3.0.4 Exercise 5 (CORE)

Examine if/which of the features are correlated to each other. Are any of the features correlated with our output (`median_house_value`) variable?

- Can you think of any reason why certain features may be correlated?
- How might we use this information in later steps of our model pipeline?

Hint

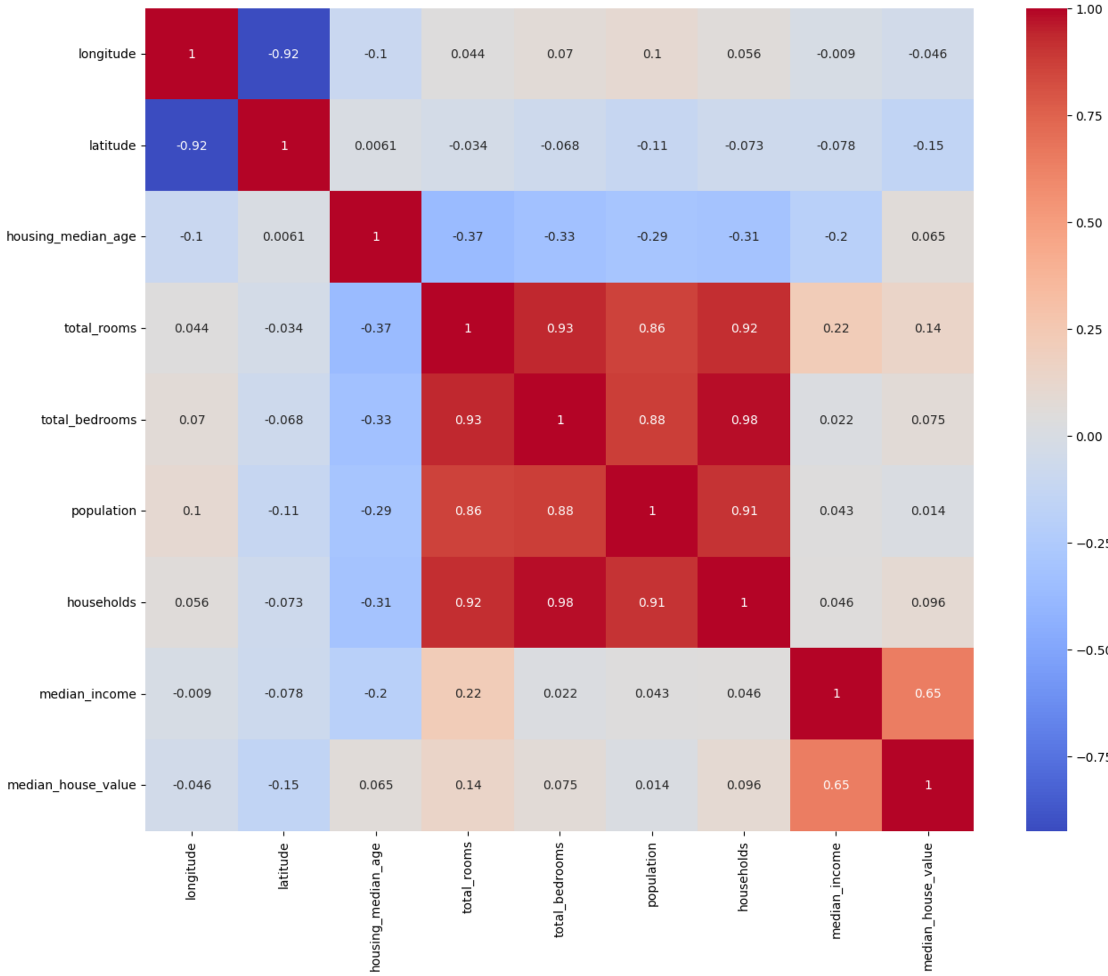
- .corr() can be used to compute the correlations.
- You can use a sns.heatmap to visualize the correlations

[9]:
```
housing_mt = housing_uncapped.select_dtypes(["number"]).corr()
display(housing_mt)
plt.figure(figsize=(15,12))
```

```python
sns.heatmap(housing_mt, annot=True, cmap="coolwarm")
display(plt.show)
```

```
                    longitude  latitude  housing_median_age  total_rooms  \
longitude            1.000000 -0.924139           -0.101740     0.044346
latitude            -0.924139  1.000000            0.006083    -0.033502
housing_median_age  -0.101740  0.006083            1.000000    -0.372174
total_rooms          0.044346 -0.033502           -0.372174     1.000000
total_bedrooms       0.069887 -0.067943           -0.327505     0.934233
population           0.100989 -0.113457           -0.294911     0.859642
households           0.055745 -0.072854           -0.309633     0.921177
median_income       -0.008992 -0.078135           -0.195542     0.224303
median_house_value  -0.045733 -0.149257            0.065139     0.144988

                    total_bedrooms  population  households  median_income  \
longitude                 0.069887    0.100989    0.055745      -0.008992
latitude                 -0.067943   -0.113457   -0.072854      -0.078135
housing_median_age       -0.327505   -0.294911   -0.309633      -0.195542
total_rooms               0.934233    0.859642    0.921177       0.224303
total_bedrooms            1.000000    0.879269    0.979137       0.022125
population                0.879269    1.000000    0.909090       0.042576
households                0.979137    0.909090    1.000000       0.046275
median_income             0.022125    0.042576    0.046275       1.000000
median_house_value        0.075219    0.013592    0.095634       0.646719

                    median_house_value
longitude                    -0.045733
latitude                     -0.149257
housing_median_age            0.065139
total_rooms                   0.144988
total_bedrooms                0.075219
population                    0.013592
households                    0.095634
median_income                 0.646719
median_house_value            1.000000
```

```
<function matplotlib.pyplot.show(close=None, block=None)>
```




total__rooms, total__bedrooms, population, and households are highly correlated with one another. The higher the population, the greater the number of households, which may require a greater number of rooms and therefore, a greater number of bedrooms. There may be too many dimensions and seeing that the these variables are highly and logically correlated, we may need to drop some of these variables (e.g. total_rooms and total__bedrooms may be irrelevant)

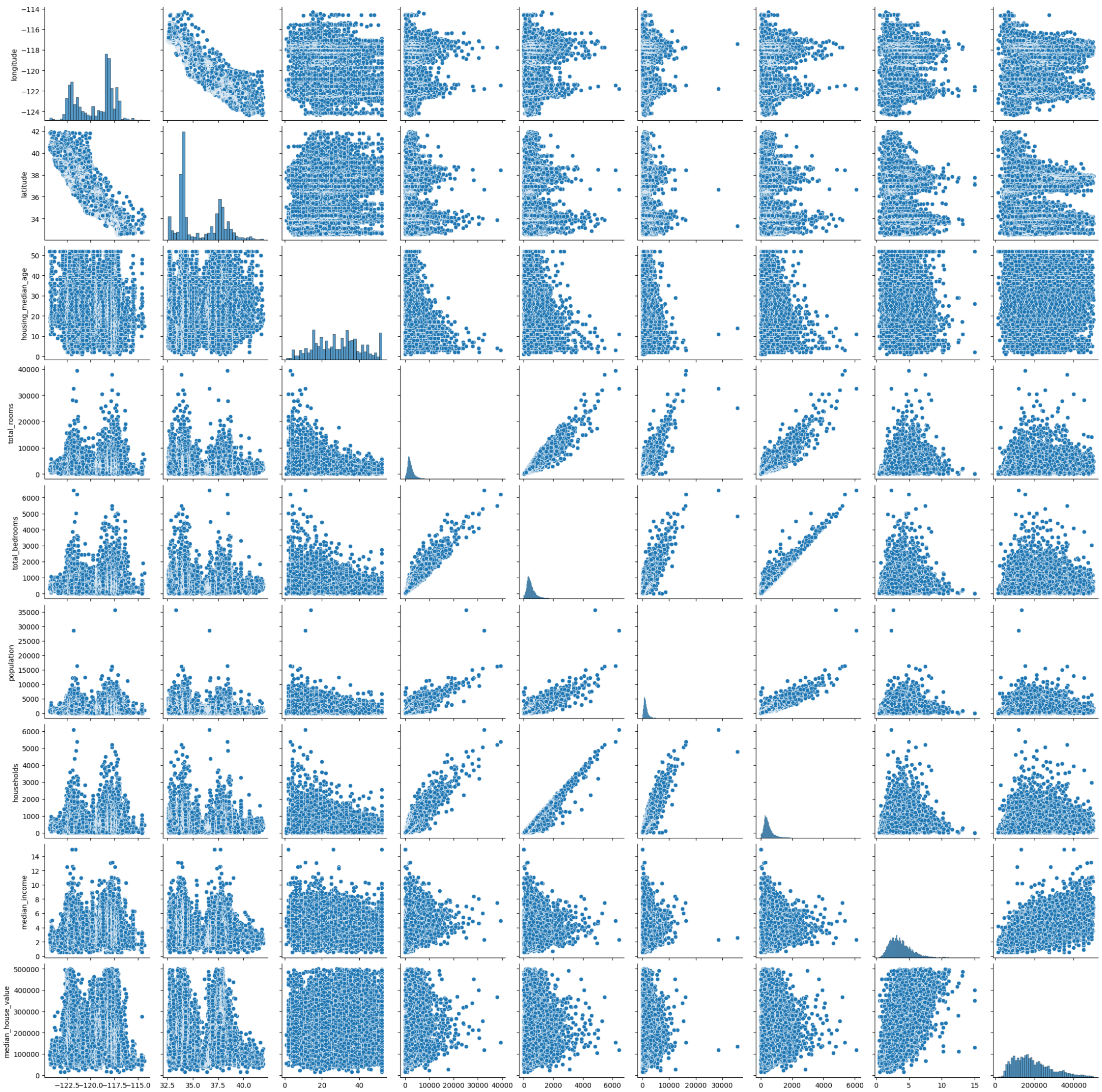
median__income and median__house__value may be correlated, with a moderate R value of 0.65. The higher the income, the greater the purchasing power of the individual, the greater the value of the house the individual is able to purchase.

### 3.0.5 Exercise 6 (CORE)

Use `sns.pairplot` to further investigate the joint relationship between each pair of variables. What insights into the data might this provide over looking only at the correlation?

[10]:
```
plt.figure(figsize=(15,12))
sns.pairplot(housing_uncapped.select_dtypes(["number"]))
display(plt.show())
```

<Figure size 1500x1200 with 0 Axes>



None

The pairplot provides a better visualisation of the correlation and distribution of data points, compared to the R value. Looking at only the R value may be insufficient in deriving correlation as the relationship between the two seemingly highly correlated variables may be non-linearly correlated or not correlated in the distribution of points in the pairplot.

# 4 Data Pre-Processing

Now we have some familiarity with the data though EDA, lets start preparing our data to be modelled.

## 4.1 Data Cleaning

Let's start with some basic data cleaning steps. For example, we may want to: - deal with duplicated, inconsitencies or typos in the data, - handle missing data, - remove uninformative features (e.g. subject identifiers), - fix variable types, - adjust data codes (e.g. missing variables may be coded as '999' instead NA), - optionally remove outliers.

Let's start with the former.

### 4.1.1 Data Duplication and Errors

We want to remove duplicates, that may have accidently been entered in the database twice, as they may bias our fitted model. In other words, we may potentially *overfit* to this subset of points. However, care should usually be taken to check they are not *real* data with identical values.

There a number of ways we could identify duplicates, the simplist (and the approach we'll focus on) is just to find observations with the same feature values. Of course this will not identify things such as spelling errors, missing values, address changes, use of aliases, etc. This may commonly happen with categorical or text data, and checking the unique values is recommended. In general for such errors, more complicated methods along with manual assessment may be needed.

### 4.1.2 Exercise 7 (CORE)

a) Are there any duplicated values in the data? If so how many?
   Hint
   With Pandas dataframes you can use `.duplicated()` to get a boolean of whether something is a duplicate and then use `.sum()` to count how many there are.
b) What are the unique values of the categorical variable? Are there any duplicated categories arising from misspellings?

[11]:
```
display(housing_uncapped.duplicated().value_counts())
display(housing_uncapped["ocean_proximity"].value_counts())
display(set(housing_uncapped["ocean_proximity"]))
```

```
False    19648
Name: count, dtype: int64
```

```
ocean_proximity
<1H OCEAN     8595
INLAND        6523
NEAR OCEAN    2437
NEAR BAY      2088
ISLAND           5
Name: count, dtype: int64
```

```
{'<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'}
```

There is no duplicated value in the data.

{'<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'} are the unique values of the categorical variable ocean_proximity. The categories INLAND and ISLAND may be duplicated due to misspellings.

#### 4.1.3 Outlier Detection

An **Outlier** is a data point that lies abnormally far from other observations and may distort the model fit and results. In general, it is a good idea to examine if any outliers are present during preprocessing. In some cases, you may want to drop these observations or cap their values (see https://feature-engine.trainindata.com/en/1.8.x/api_doc/outliers/index.html). However this may not be appropriate without explicit knowledge and testing if they are really outliers or not. In particular, when you drop or cap those observations you can discard important information unwittingly!

We will use basic statistics in order to try to identify outliers. A simple method of detecting outliers is to use the **inter-quartile range (IQR) proximity rule** (Tukey fences) which states that a value is an outlier if it falls outside these boundaries:

- Upper boundary = 75th quantile + (IQR * $k$)
- Lower boundary = 25th quantile - (IQR * $k$)

where IQR = 75th quantile - 25th quantile (the length of the box in the boxplot). This is used to construct the whiskers in the boxplot, where $k$ is a nonnegative constant which is typically set to 1.5 (the default value in `sns.boxplot`). However, it is also common practice to find extreme values by setting $k$ to 3.

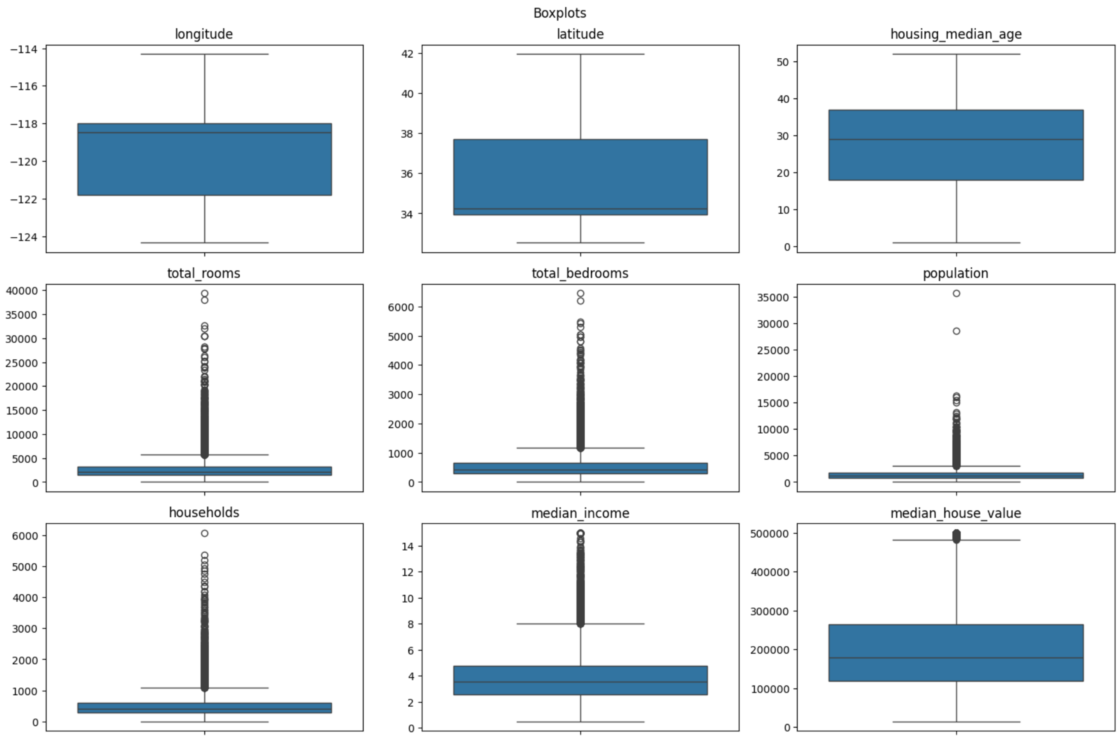
#### 4.1.4 Exercise 8 (EXTRA)

a) Can you identify any potential outliers using the generated boxplots below? Do you think any points should be removed?

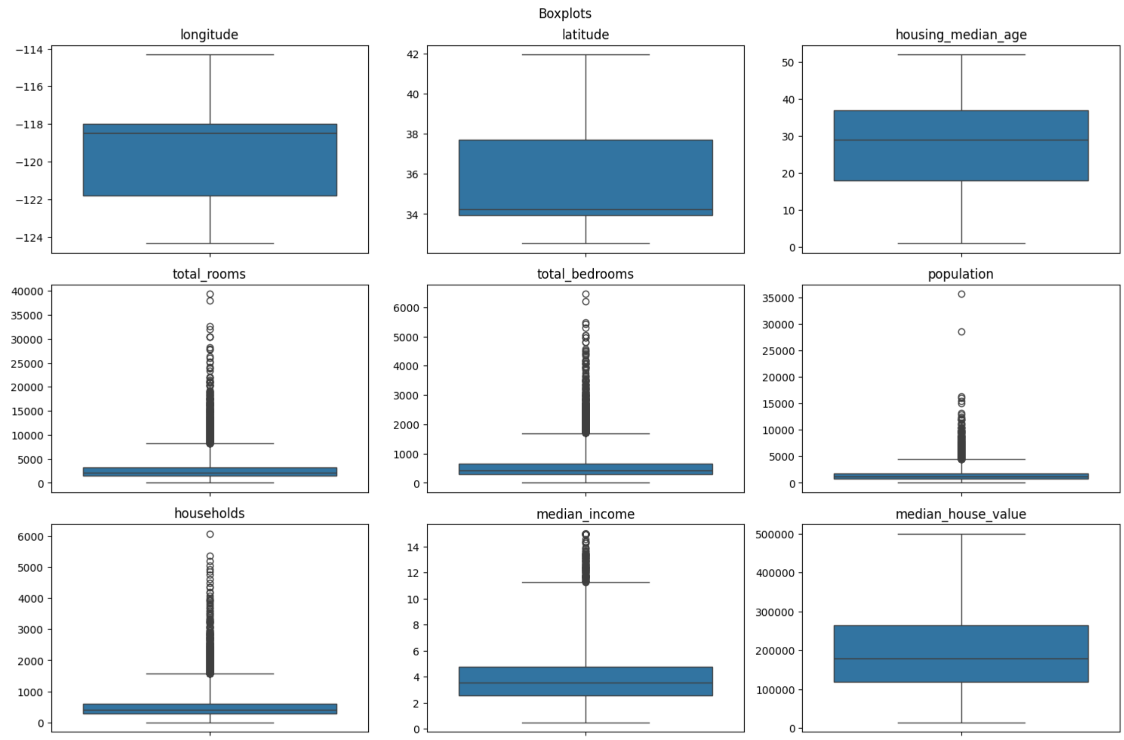
b) Try changing $k$, defining the length of the whiskers, to 3 in `sns.boxplot`. Can you still identify any potential outliers?

[12]:
```python
fig, axes = plt.subplots(figsize = (15,10), ncols = (housing.shape[1]-1)//3,␣
 ↪nrows = 3, sharex = True)
axes = axes.flatten()

for i, ax in enumerate(axes):
    sns.boxplot(y = housing.iloc[:,i], ax = ax)
    ax.set_title(housing.iloc[:,i].name)
    ax.set_ylabel("")

plt.suptitle("Boxplots")
plt.tight_layout()
display(plt.show())

fig, axes = plt.subplots(figsize = (15,10), ncols = (housing.shape[1]-1)//3,␣
 ↪nrows = 3, sharex = True)
axes = axes.flatten()

for i, ax in enumerate(axes):
    sns.boxplot(y = housing.iloc[:,i], ax = ax, whis = 3)
    ax.set_title(housing.iloc[:,i].name)
```

```
    ax.set_ylabel("")

plt.suptitle("Boxplots")
plt.tight_layout()
display(plt.show())
```




```
None
```




```
None
```

There are 2 potential outliers under the population variable, which may be removed.

As median_house_value is capped at $500000, it may be more beneficial to remove these data points as the high proportion of these data points skews the data.

After increasing k to 3, there are still 2 outliers under the population variable, which may be removed. The previous outliers in median_house_value are now captured within the range of values.

**Now, is a good point to switch driver and navigator**

### 4.1.5 Missing Data

Most ML models cannot handle missing values, and as we saw earlier, there are some present in `total_bedrooms`. We also saw that values of `median_house_value` are capped at $\$500,000$. This is another form of missingness, which is **informative** for missing values (i.e. the missing values are greater than $\$500,000$). However, we will focus on methods for dealing with missingness in our features and not the target variable.

As such, let's start by splitting our **features** from our **target** variable in the data set.

[13]:
```
# Extracting the features from the data
X = housing.drop("median_house_value", axis = 1)
features = list(X.columns)
print(features)
```

```
print(X.shape)
display(X.head())
```

```
['longitude', 'latitude', 'housing_median_age', 'total_rooms', 'total_bedrooms',
'population', 'households', 'median_income', 'ocean_proximity']
(20640, 9)

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

   population  households  median_income ocean_proximity
0       322.0       126.0         8.3252        NEAR BAY
1      2401.0      1138.0         8.3014        NEAR BAY
2       496.0       177.0         7.2574        NEAR BAY
3       558.0       219.0         5.6431        NEAR BAY
4       565.0       259.0         3.8462        NEAR BAY
```

[14]:
```
# Extracting the target features from the data
y = housing["median_house_value"].copy()
print(y.shape)
display(y.head())
```

```
(20640,)

0    452600.0
1    358500.0
2    352100.0
3    341300.0
4    342200.0
Name: median_house_value, dtype: float64
```

There are a number of ways we can deal with missing values. The simplest is to just **remove NA values**. We can do this in two ways by either:

1. Getting rid of the corresponding observations (deleting the corresponding rows).
2. Getting rid of the whole attribute (deleting the corresponding columns).

To is relatively straight forward by running `housing.dropna()` with either the `axis` set to `0` or `1` (depending if we want to remove rows or columns) before splitting our data into features (`X`) and outputs (`y`).

### 4.1.6 Exercise 9 (CORE)

Use `dropna()` to remove the missing observations. What is the shape of the feature matrix after dropping the missing observations?

**Notes**

- It may be tempting to overwrite `X` while working on our pre-processing steps. **Don't do this!** We will run these objects through our pipeline which combines missing data steps with other steps later, so if you want to test your function make sure to assign the output to tempory objects (e.g. `X_`).

[15]:
```
housing_uncap_drop = housing_uncapped.dropna()
display(housing_uncap_drop.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 19448 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           19448 non-null  float64
 1   latitude            19448 non-null  float64
 2   housing_median_age  19448 non-null  float64
 3   total_rooms         19448 non-null  float64
 4   total_bedrooms      19448 non-null  float64
 5   population          19448 non-null  float64
 6   households          19448 non-null  float64
 7   median_income       19448 non-null  float64
 8   median_house_value  19448 non-null  float64
 9   ocean_proximity     19448 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
None
```

Instead of simply dropping missing data, we may instead want to use other **imputation** methods. From here on in, we will be creating functions for our data transformations. Later, we will see why this is really useful to define our **model pipeline**, which allows us to chain together transformations and steps in a reproducible way.

In this course we are mostly going to be using `Scikit-learn`, with a little `Keras` at the end for neural networks. Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning (https://scikit-learn.org/stable/getting_started.html). It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

We will first focus on the *transformer* class within Scikit-learn, which provides functions for missing data imputation along with many others useful for data pre-processing and feature engineering.

**Transfomers** If we want to **alter the features** of our data, we need a *transfomer*.

- Transformers are classes that follow the scikit-learn API in Scikit-Learn clean, impute, reduce, expand, or generate feature representations.
- Transformers are classes with a `.fit()` method, which learn model parameters (e.g. mean for mean imputation) from a training set, and a `.transform()` method which applies this transformation model to data. To create a custom transformer, all you need is to create a class that implements three methods: `fit()`, `transform()`, and `fit_transform()`.

Therefore to transform a dataset, each sampler implements:

```
obj.fit(data)
data_transformed = obj.transform(data)
```

or simply…

```
data_transformed = obj.fit_transform(data)`
```

See more details: https://scikit-learn.org/stable/data_transforms.html. In the following subsections, we will see examples of *transformers* for categorical and numerical variables.

**Data Imputation** Instead of removing the missing data we can set it to some value. To do this, Scikit-Learn provides various transformers, including:

- `SimpleImputer` which provides simple strategies (e.g. `"mean"`, `"median"` for numerical features and `"most_frequent"` for categorical features).
- You can also add a missing indicator with the option `add_indicator=True` in `SimplerImputer`, or use the transfomer `MissingIndicator`. This may be useful in the case when missing features may be provide information for predicting the target (e.g. obese patients may prefer not to report bmi, thus, this missingness could be useful for estimating the risk of health conditions or diseases).
- Beyond simple imputation strategies, sklearn also provides more advanced imputation strategies in `IterativeImputer` and `KNNImputer`
- Other strategies are also available in `feature_engine.imputation`. Such as `EndTailImputer`, which is useful when missing values are located in the tails (e.g. capped values for privacy)

Let's start with the `SimpleImputer` to learn about transfomers and how to deal with missing data in sklearn.

[16]:
```
from sklearn.impute import SimpleImputer

# First create the imputer object/transformer
num_imputer = SimpleImputer(strategy="median")

# Now fit the object to the data
# num_imputer.fit(housing_uncapped)
```

Unfortunately, when we applied this to our data, we get the following error:

```
ValueError: Cannot use median strategy with non-numeric data:
could not convert string to float:
```

This is because the "median" strategy can only be used with numerical attributes so we need a way of only applying imputation to certain attributes. We could temporarily remove the categorical feature from our data to apply our function, or apply the function to a subset of the data and assign the output to the same subset.

However scikit-learn has a handy function to specify what column we want to apply a function to!

[17]:
```python
from sklearn.compose import ColumnTransformer

# Names of numerical columns
features = housing_uncapped.columns
numcols = features[:-1]
print(numcols)
catcols = [features[-1]]
print(catcols)

num_cols_imputer = ColumnTransformer(
    # apply the `num_imputer` to all columns apart from the last
    [("num", num_imputer, numcols)],
    # don't touch all other columns, instead concatenate it on the end of the
    # changed data.
    remainder = "passthrough"
)

num_cols_imputer.fit(housing_uncapped)

# Print the median values computed by calling fit
print("Computed median values for each numerical feature:")
print(num_cols_imputer["num"].statistics_)
```

```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value'],
      dtype='object')
['ocean_proximity']
Computed median values for each numerical feature:
[-1.18500e+02  3.42700e+01  2.80000e+01  2.11100e+03  4.36000e+02
  1.17900e+03  4.11000e+02  3.44905e+00  1.73600e+05]
```

After using `.fit`, our object now has a number of attributes, including `statistics_` which stores the median value for each numerical attribute on the training set. This value can be used when validating and testing the model as it will be used if there is missing data in the new data.

**Note** - The fitted `ColumnTransformer` contains a list of transformers, stored in the attribute `transformers_`. We named the first transformer in the list `num`. Try running `num_cols_imputer.transformers_` to see the names and types of the transformers in the list. - To access the fitted `num_imputer` in this case, `num_cols_imputer["num"]` is a shortcut to access the named transformer in the list.

Now, let's call `transform` to our fitted objected to impute the missing values.

[18]:
```python
X_ = num_cols_imputer.transform(housing_uncapped)
print("Number of Missing Values")
pd.DataFrame(X_, columns = features).isna().sum()
```

```
Number of Missing Values
```

[18]:
```
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
```

### 4.1.7 Exercise 10 (CORE)

In addition to median imputation, alter your transformer to also include a missing indicator. What is the shape of the transformed feature matrix? Use the method `.get_feature_names_out()` to print the names of the new features.

**Note:** You may want to add the option `verbose_feature_names_out = False` in your `ColumnTransformer` to reduce the length of the feature names.

[19]:
```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer

# First create the imputer object/transformer
num_imputer = SimpleImputer(strategy="median", add_indicator=True)

# Names of numerical columns
features = housing_uncapped.columns
numcols = features[:-1]
print(numcols)
catcols = [features[-1]]
print(catcols)

num_cols_imputer = ColumnTransformer(
    # apply the `num_imputer` to all columns apart from the last
    [("num", num_imputer, numcols)],
    # don't touch all other columns, instead concatenate it on the end of the
    # changed data.
    remainder = "passthrough",
    verbose_feature_names_out=False
)

num_cols_imputer.fit(housing_uncapped)

X_ = num_cols_imputer.transform(housing_uncapped)
display(X_.shape)
```

```
display(pd.DataFrame(X_, columns = num_cols_imputer.get_feature_names_out()).
 ↪head())
```

```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value'],
      dtype='object')
['ocean_proximity']
```

```
(19648, 11)
```

```
  longitude latitude housing_median_age total_rooms total_bedrooms population  \
0   -122.23    37.88               41.0       880.0          129.0      322.0
1   -122.22    37.86               21.0      7099.0         1106.0     2401.0
2   -122.24    37.85               52.0      1467.0          190.0      496.0
3   -122.25    37.85               52.0      1274.0          235.0      558.0
4   -122.25    37.85               52.0      1627.0          280.0      565.0

  households median_income median_house_value missingindicator_total_bedrooms  \
0      126.0        8.3252           452600.0                             0.0
1     1138.0        8.3014           358500.0                             0.0
2      177.0        7.2574           352100.0                             0.0
3      219.0        5.6431           341300.0                             0.0
4      259.0        3.8462           342200.0                             0.0

  ocean_proximity
0        NEAR BAY
1        NEAR BAY
2        NEAR BAY
3        NEAR BAY
4        NEAR BAY
```

**Now, is a good point to switch driver and navigator**

# 5 Feature Engineering

As discussed in the lectures, feature engineering is where we extract features from data and transform them into formats that are suitable for machine learning models. Today, we will have a look at two main cases that are present in our data: **categorical** and **numerical** values.

Feature engineering also requires a *transformer* class to **alter the features**.

## 5.1 Categorical Variables

- In the dataset, we have an text attribute (`ocean_proximity`) that we already had to treat differently when cleaning and visualizing the data. This extends to feature engineering as well, where we need to use separate methods than those used with numerical variables.
- If we look at the unique values of this attribute, we will see that there are a limited number of possible values which represent a category. We need a way of encoding this information into

our modeling framework **by converting our string/categorical variable into a numeric representation** that can be included in our models.

If we have a binary categorical variable (two levels) we could do this by picking one of the categorical levels and encode it as 1 and the other level as 0.

However, in this case as we have multiple categories, we would probably want to use another encoding method. To illustrate, we can try encoding the the categorical feature `ocean_proximity` using both the `OrdinalEncoder` and `OneHotEncoder` available in `sklearn.preprocessing`.

**Side Notes**

- The output of the `OneHotEncoder` provided in Scikit-Learn is a SciPy *sparse matrix*, instead of a NumPy array. These are useful when you have lots of categories as your matrix becomes mostly full of 0's. To store all these 0's takes up unneccesary memory, so instead a sparse matrix just stores the location of nonzero elements. The good news is that you can use a sparse matrix similar to a numpy matrix, but if you wanted to, you can convert it to a dense numpy matrix using `.toarray()`.
- The above does not seem to be the case if passed through a `ColumnTransformer`.

[20]:
```python
from sklearn.preprocessing import OrdinalEncoder
# Defining the OrdinalEncoder
ordinal_encoder = OrdinalEncoder()

encoder = ColumnTransformer(
    # apply the ordinal_encoder to the last column
    [("cat", ordinal_encoder, catcols)],
    remainder="passthrough",
    verbose_feature_names_out=False)

# fitting the encoder defined above
X_ = encoder.fit_transform(housing_uncapped)

# Accessing the fitted ordinal encoder (encoder["cat"]) to see how the␣
 ↪categories were mapped
display(dict(zip(list(encoder["cat"].categories_[0]), range(5))))

# Display the first few rows of the transformed data
display(pd.DataFrame(X_, columns = encoder.get_feature_names_out()).head())
```

```
{'<1H OCEAN': 0, 'INLAND': 1, 'ISLAND': 2, 'NEAR BAY': 3, 'NEAR OCEAN': 4}
```

```
   ocean_proximity  longitude  latitude  housing_median_age  total_rooms  \
0              3.0    -122.23     37.88                41.0        880.0
1              3.0    -122.22     37.86                21.0       7099.0
2              3.0    -122.24     37.85                52.0       1467.0
3              3.0    -122.25     37.85                52.0       1274.0
4              3.0    -122.25     37.85                52.0       1627.0

   total_bedrooms  population  households  median_income  median_house_value
```

| | | | | | |
|---|---|---|---|---|---|
| 0 | 129.0 | 322.0 | 126.0 | 8.3252 | 452600.0 |
| 1 | 1106.0 | 2401.0 | 1138.0 | 8.3014 | 358500.0 |
| 2 | 190.0 | 496.0 | 177.0 | 7.2574 | 352100.0 |
| 3 | 235.0 | 558.0 | 219.0 | 5.6431 | 341300.0 |
| 4 | 280.0 | 565.0 | 259.0 | 3.8462 | 342200.0 |

[21]:
```python
from sklearn.preprocessing import OneHotEncoder
# Defining the OneHotEncoder
onehot_encoder = OneHotEncoder()

encoder = ColumnTransformer(
    # apply the onehot_encoder to the last column
    [("cat", onehot_encoder, catcols)],
    remainder="passthrough",
    verbose_feature_names_out=False)

X_ = encoder.fit_transform(housing_uncapped)

# Display the first few rows of the transformed data
display(pd.DataFrame(X_, columns = encoder.get_feature_names_out()).head())
```

| | ocean_proximity_<1H OCEAN | ocean_proximity_INLAND | ocean_proximity_ISLAND \ |
|---|---|---|---|
| 0 | 0.0 | 0.0 | 0.0 |
| 1 | 0.0 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 | 0.0 |
| 4 | 0.0 | 0.0 | 0.0 |

| | ocean_proximity_NEAR BAY | ocean_proximity_NEAR OCEAN | longitude | latitude \ |
|---|---|---|---|---|
| 0 | 1.0 | 0.0 | -122.23 | 37.88 |
| 1 | 1.0 | 0.0 | -122.22 | 37.86 |
| 2 | 1.0 | 0.0 | -122.24 | 37.85 |
| 3 | 1.0 | 0.0 | -122.25 | 37.85 |
| 4 | 1.0 | 0.0 | -122.25 | 37.85 |

| | housing_median_age | total_rooms | total_bedrooms | population | households \ |
|---|---|---|---|---|---|
| 0 | 41.0 | 880.0 | 129.0 | 322.0 | 126.0 |
| 1 | 21.0 | 7099.0 | 1106.0 | 2401.0 | 1138.0 |
| 2 | 52.0 | 1467.0 | 190.0 | 496.0 | 177.0 |
| 3 | 52.0 | 1274.0 | 235.0 | 558.0 | 219.0 |
| 4 | 52.0 | 1627.0 | 280.0 | 565.0 | 259.0 |

| | median_income | median_house_value |
|---|---|---|
| 0 | 8.3252 | 452600.0 |
| 1 | 8.3014 | 358500.0 |
| 2 | 7.2574 | 352100.0 |
| 3 | 5.6431 | 341300.0 |
| 4 | 3.8462 | 342200.0 |

### 5.1.1 Exercise 11 (CORE)

a) What is the main difference between two methods regarding the obtained features? Which encoding method do you think is most appropriate for this variable and why?

b) How sensible is the default ordering of the ordinal encoder? Use the parameter `categories` of `OrdinalEncoder` to apply a different ordering.

One-hot encoding transforms the categories into binary, while ordinal encoding transforms the categories into orders/ranks. Ordinal encoding is more appropriate for the categorical variable ocean_proximity, as the categories are ordered by the location of the house with respect to the ocean/sea.

The default ordering of the ordinal encoder is by alphabetical order, which is not sensible.

[22]:
```
from sklearn.preprocessing import OrdinalEncoder
# Defining the OrdinalEncoder
ordinal_encoder = OrdinalEncoder(categories=[[
        'INLAND',
        '<1H OCEAN',
        'NEAR OCEAN',
        'NEAR BAY',
        'ISLAND'
    ]])

encoder = ColumnTransformer(
    # apply the ordinal_encoder to the last column
    [("cat", ordinal_encoder, catcols)],
    remainder="passthrough",
    verbose_feature_names_out=False)

# fitting the encoder defined above
X_ = encoder.fit_transform(housing_uncapped)

# Accessing the fitted ordinal encoder (encoder["cat"]) to see how the␣
↪categories were mapped
display(dict(zip(list(encoder["cat"].categories_[0]), range(5))))

# Display the first few rows of the transformed data
display(pd.DataFrame(X_, columns = encoder.get_feature_names_out()).head())
```

```
{'INLAND': 0, '<1H OCEAN': 1, 'NEAR OCEAN': 2, 'NEAR BAY': 3, 'ISLAND': 4}
```

```
   ocean_proximity  longitude  latitude  housing_median_age  total_rooms  \
0              3.0    -122.23     37.88                41.0        880.0
1              3.0    -122.22     37.86                21.0       7099.0
2              3.0    -122.24     37.85                52.0       1467.0
3              3.0    -122.25     37.85                52.0       1274.0
4              3.0    -122.25     37.85                52.0       1627.0
```

```
   total_bedrooms  population  households  median_income  median_house_value
0           129.0       322.0       126.0         8.3252            452600.0
1          1106.0      2401.0      1138.0         8.3014            358500.0
2           190.0       496.0       177.0         7.2574            352100.0
3           235.0       558.0       219.0         5.6431            341300.0
4           280.0       565.0       259.0         3.8462            342200.0
```

#### 5.1.2 Exercise 12 (EXTRA)

Another handy feature of `OneHotEncoder` and `OrdinalEncoder` is that infrequent categories can be aggregated into a single feature/value. The parameters to enable the gathering of infrequent categories are `min_frequency` and `max_categories`.

Use the `max_categories` attribute to set the maximum number of categories to 4. Use the `get_feature_names_out()` method of `OneHotEncoder` to print the new category names. Which two features have been combined?

[23]:
```
from sklearn.preprocessing import OneHotEncoder
# Defining the OneHotEncoder
onehot_encoder = OneHotEncoder(max_categories=4)

encoder = ColumnTransformer(
    # apply the onehot_encoder to the last column
    [("cat", onehot_encoder, catcols)],
    remainder="passthrough",
    verbose_feature_names_out=False)

X_ = encoder.fit_transform(housing_uncapped)

# Display the first few rows of the transformed data
display(pd.DataFrame(X_, columns = encoder.get_feature_names_out()).head())
```

```
   ocean_proximity_<1H OCEAN  ocean_proximity_INLAND  \
0                        0.0                     0.0
1                        0.0                     0.0
2                        0.0                     0.0
3                        0.0                     0.0
4                        0.0                     0.0

   ocean_proximity_NEAR OCEAN  ocean_proximity_infrequent_sklearn  longitude  \
0                         0.0                                 1.0    -122.23
1                         0.0                                 1.0    -122.22
2                         0.0                                 1.0    -122.24
3                         0.0                                 1.0    -122.25
4                         0.0                                 1.0    -122.25

   latitude  housing_median_age  total_rooms  total_bedrooms  population  \
0     37.88                41.0        880.0           129.0       322.0
1     37.86                21.0       7099.0          1106.0      2401.0
```

```
2     37.85                  52.0       1467.0           190.0        496.0
3     37.85                  52.0       1274.0           235.0        558.0
4     37.85                  52.0       1627.0           280.0        565.0

   households  median_income  median_house_value
0       126.0         8.3252            452600.0
1      1138.0         8.3014            358500.0
2       177.0         7.2574            352100.0
3       219.0         5.6431            341300.0
4       259.0         3.8462            342200.0
```

ISLAND and NEAR BAY fatures have been combined.

#### 5.1.3 Exercise 13 (EXTRA)

When there are many unordered categories, another useful encoding scheme is `TargetEncoder` which uses the target mean conditioned on the categorical feature for encoding unordered categories. Whereas one-hot encoding would greatly inflate the feature space if there are a very large number of categories (e.g. zip code or region), `TargetEncoder` is more parsimonious.

Use target encoding of ocean proximity. What are the numerical values assigned to the categories?

**Caution:** when using this transformer, be careful to avoid data leakage and overfitting by integrating it properly in your model pipeline! We will learn more about this later.

[24]:
```python
# Create a target encoder
from sklearn.preprocessing import TargetEncoder
t_encoder = TargetEncoder()

encoder = ColumnTransformer(
    # apply the t_encoder to the last column
    [("cat", t_encoder, catcols)],
    remainder="passthrough",
    verbose_feature_names_out=False)

X_ = encoder.fit_transform(
    housing_uncapped,
    y = housing_uncapped["median_house_value"]
)

# Display the first few rows of the transformed data
display(pd.DataFrame(X_, columns = encoder.get_feature_names_out()).head())
```

```
/home/darrenljs/MachineLearninginPython_UoE/.venv/lib/python3.12/site-
packages/sklearn/model_selection/_split.py:811: UserWarning: The least populated
class in y has only 1 members, which is less than n_splits=5.
  warnings.warn(

   ocean_proximity_14999.0  ocean_proximity_17500.0  ocean_proximity_22500.0  \
0                      0.0                      0.0                 0.000604
```

```
1                        0.0                        0.0                   0.000587
2                        0.0                        0.0                   0.000608
3                        0.0                        0.0                   0.000589
4                        0.0                        0.0                   0.000000

   ocean_proximity_25000.0  ocean_proximity_26600.0  ocean_proximity_26900.0  \
0                      0.0                      0.0                      0.0
1                      0.0                      0.0                      0.0
2                      0.0                      0.0                      0.0
3                      0.0                      0.0                      0.0
4                      0.0                      0.0                      0.0

   ocean_proximity_27500.0  ocean_proximity_28300.0  ocean_proximity_30000.0  \
0                      0.0                      0.0                      0.0
1                      0.0                      0.0                      0.0
2                      0.0                      0.0                      0.0
3                      0.0                      0.0                      0.0
4                      0.0                      0.0                      0.0

   ocean_proximity_32500.0  …  ocean_proximity_499100.0  longitude  \
0                 0.000604  …                       0.0    -122.23
1                 0.000587  …                       0.0    -122.22
2                 0.000000  …                       0.0    -122.24
3                 0.000589  …                       0.0    -122.25
4                 0.000603  …                       0.0    -122.25

   latitude  housing_median_age  total_rooms  total_bedrooms  population  \
0     37.88                41.0        880.0           129.0       322.0
1     37.86                21.0       7099.0          1106.0      2401.0
2     37.85                52.0       1467.0           190.0       496.0
3     37.85                52.0       1274.0           235.0       558.0
4     37.85                52.0       1627.0           280.0       565.0

   households  median_income  median_house_value
0       126.0         8.3252            452600.0
1      1138.0         8.3014            358500.0
2       177.0         7.2574            352100.0
3       219.0         5.6431            341300.0
4       259.0         3.8462            342200.0

[5 rows x 3849 columns]
```

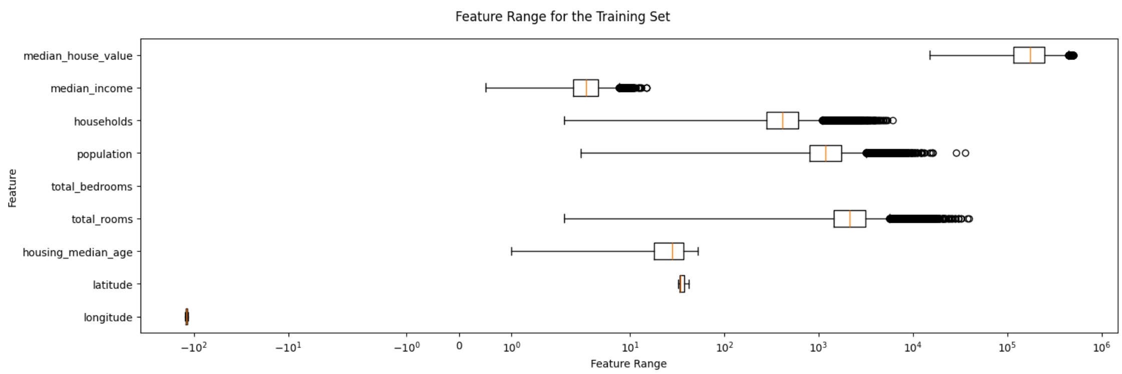
## 5.2 Numerical Variables

### 5.2.1 Feature Scaling

As we will discuss in later weeks, many machine learning algorithms are sensitive to the scale and magnitude of the features, and especially differences in scales across features. For these algorithms, feature scaling will improve performance.

For example, let's investigate the range of the features in our dataset:

[25]:
```
fig, ax = plt.subplots(figsize=(15,5))

plt.boxplot(housing_uncapped[numcols], vert = False)
plt.xscale("symlog")
plt.ylabel("Feature")
plt.xlabel("Feature Range")

ax.set_yticklabels(numcols)

plt.suptitle("Feature Range for the Training Set")
plt.tight_layout()
plt.show()
```




There are various options in scikit learn for feature scaling, including:

- Standardization (`preprocessing.StandardScaler`)
- Min-Max Scaling (`preprocessing.MinMaxScaler`)
- l2 Normalization (`preprocessing.normalize`)
- RobustScaler(`preprocessing.RobustScaler`)
- Scale with maximum absolute value (`preprocessing.MaxAbsScaler`)
  - As scaling generally improves the performance of most models when features cover a range of scales, it is probably a good idea to apply some sort of scaling to our data before fitting a model.
  - *Standardization* (or *variance scaling*), is the most common, but there are a number of other types, as listed above.

### 5.2.2 Exercise 14 (CORE)

Try implementing at least two different scalers for the `total_rooms` and `total_bedrooms` variables. Make a scatter plot of the original and transformed features to see the main differences.

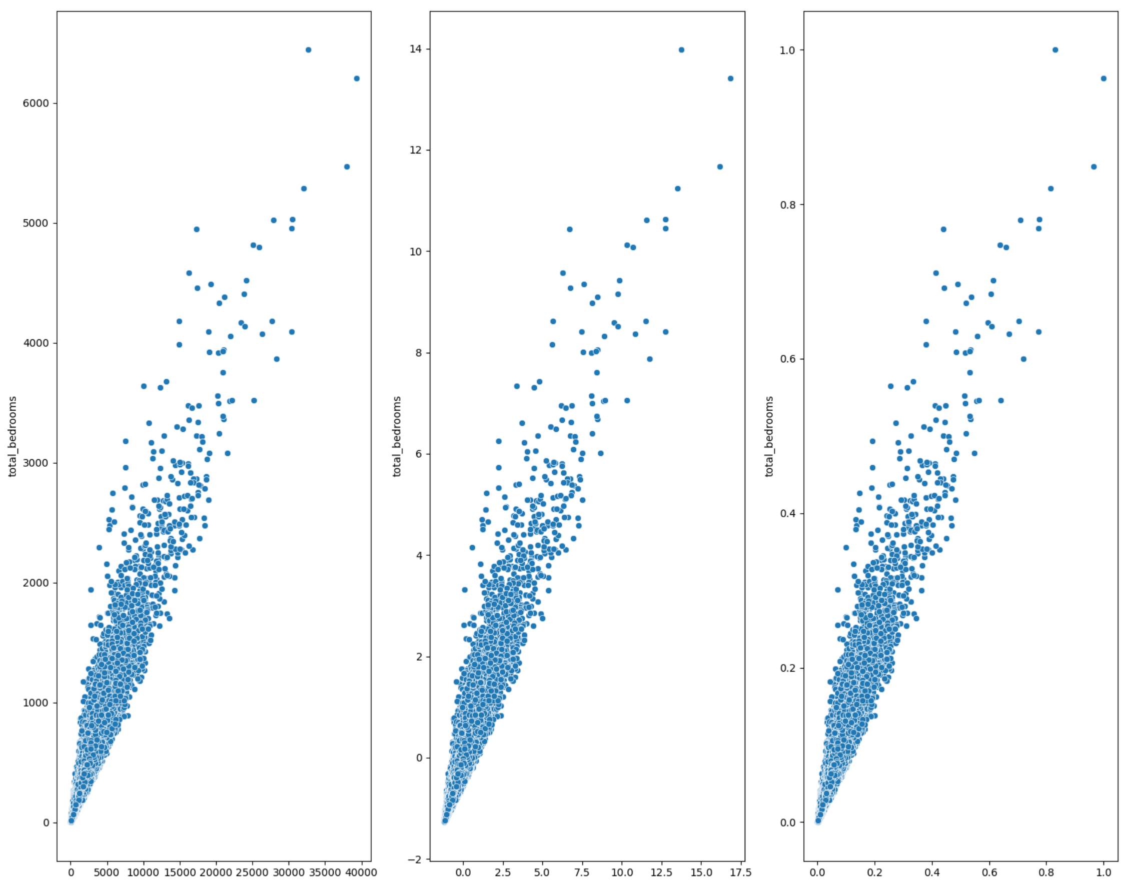
[26]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.compose import ColumnTransformer

std_scaler = StandardScaler()
mm_scaler = MinMaxScaler()

rmcols = ["total_rooms", "total_bedrooms"]

scaler_1 = ColumnTransformer(
    [("num", std_scaler, rmcols)],
    remainder = "passthrough",
    verbose_feature_names_out=False
)

scaler_2 = ColumnTransformer(
    [("num", mm_scaler, rmcols)],
    remainder = "passthrough",
    verbose_feature_names_out=False
)

X_1 = scaler_1.fit_transform(housing_uncapped)
X_2 = scaler_2.fit_transform(housing_uncapped)
X1 = pd.DataFrame(X_1, columns = scaler_1.get_feature_names_out())
X2 = pd.DataFrame(X_2, columns = scaler_2.get_feature_names_out())
display(X1.head())
display(X2.head())

fig, axes = plt.subplots(1, 3, figsize=(15, 12))
sns.scatterplot(data = housing_uncapped, x = "total_rooms", y =␣
↪"total_bedrooms", ax = axes[0])
sns.scatterplot(data = X1, x = "total_rooms", y = "total_bedrooms", ax =␣
↪axes[1])
sns.scatterplot(data = X2, x = "total_rooms", y = "total_bedrooms", ax =␣
↪axes[2])
plt.tight_layout()
display(plt.show())
```

```
  total_rooms total_bedrooms longitude latitude housing_median_age population  \
0   -0.797473      -0.971846   -122.23    37.88               41.0       322.0
1    2.052252       1.340349   -122.22    37.86               21.0      2401.0
2   -0.528492      -0.827481   -122.24    37.85               52.0       496.0
3   -0.616931      -0.720983   -122.25    37.85               52.0       558.0
4   -0.455176      -0.614485   -122.25    37.85               52.0       565.0
```

| | households | median_income | median_house_value | ocean_proximity |
|---|---|---|---|---|
| 0 | 126.0 | 8.3252 | 452600.0 | NEAR BAY |
| 1 | 1138.0 | 8.3014 | 358500.0 | NEAR BAY |
| 2 | 177.0 | 7.2574 | 352100.0 | NEAR BAY |
| 3 | 219.0 | 5.6431 | 341300.0 | NEAR BAY |
| 4 | 259.0 | 3.8462 | 342200.0 | NEAR BAY |

| | total_rooms | total_bedrooms | longitude | latitude | housing_median_age | population \ |
|---|---|---|---|---|---|---|
| 0 | 0.022331 | 0.019711 | -122.23 | 37.88 | 41.0 | 322.0 |
| 1 | 0.180503 | 0.171349 | -122.22 | 37.86 | 21.0 | 2401.0 |
| 2 | 0.03726 | 0.029179 | -122.24 | 37.85 | 52.0 | 496.0 |
| 3 | 0.032352 | 0.036163 | -122.25 | 37.85 | 52.0 | 558.0 |
| 4 | 0.04133 | 0.043148 | -122.25 | 37.85 | 52.0 | 565.0 |

| | households | median_income | median_house_value | ocean_proximity |
|---|---|---|---|---|
| 0 | 126.0 | 8.3252 | 452600.0 | NEAR BAY |
| 1 | 1138.0 | 8.3014 | 358500.0 | NEAR BAY |
| 2 | 177.0 | 7.2574 | 352100.0 | NEAR BAY |
| 3 | 219.0 | 5.6431 | 341300.0 | NEAR BAY |
| 4 | 259.0 | 3.8462 | 342200.0 | NEAR BAY |



```
None
```

### 5.2.3 Power Transformation

In some cases, we may wish to apply transformations to our data, so that they have a more Gaussian distribution. For example, log transformations are useful for altering count data to have a more normal distribution as they pull in the more extreme high values relative to the median, while stretching back extreme low values away from the median. You can use a log transformation with either the pre-made `LogTransformer()` from `feature_engine.transformation`, or a custom function and `sklearn.preprocessing.FunctionTransformer`.

More generally, the natural logarithm, square root, and inverse transformations are special cases of the **Box-Cox** family of transformations (Box and Cox 1964). The question is **why do we need such a transformation and when?**

- Note that, the method is typically used to transform the outcome, but can also be used to transform predictors.
- The method assumes that the variable takes only positive values. If there are any zero or negative values, we can 1) shift the distribution towards positive values by adding a constant, or 2) use the **Yeo-Johnson transformation** (Yeo and Johnson 2000).
- In general, transormations can make interpretations more difficult, thus **you should think carefully if they are needed**, particularly if they only result in modest improvements in model performance. Moreover, finding a suitable transformation is typically a trial-and-error process.
- Moreover, if you are transforming the features, you should also consider how this alters the relationship with the target variable.

The Yeo-Johnson transformation is defined as:

$$\tilde{y} = \begin{cases} \frac{(y+1)^\lambda - 1}{\lambda}, & \lambda \neq 0 \text{ and } y \geq 0 \\ \log(y+1), & \lambda = 0 \text{ and } y \geq 0 \\ -\frac{(1-y)^{2-\lambda} - 1}{2-\lambda}, & \lambda \neq 2 \text{ and } y < 0 \\ -\log(1-y), & \lambda = 2 \text{ and } y < 0 \end{cases},$$

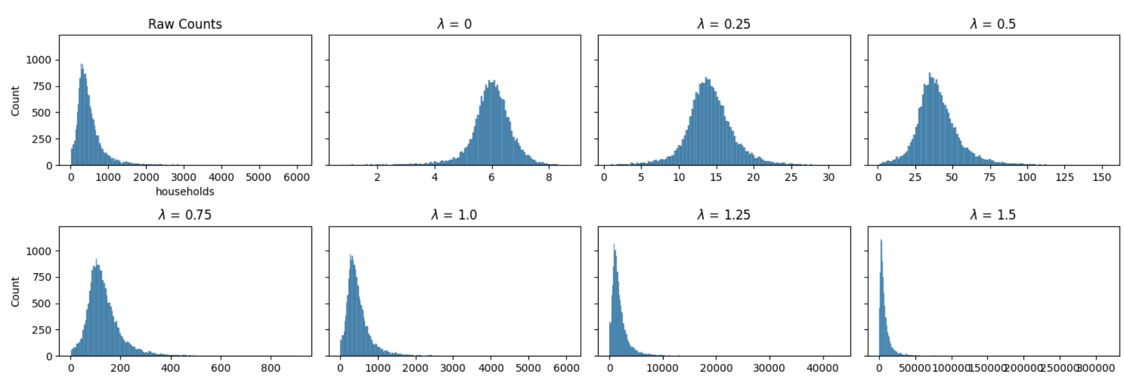
with the Box-Cox transformation as a special case (applied to $y - 1$).

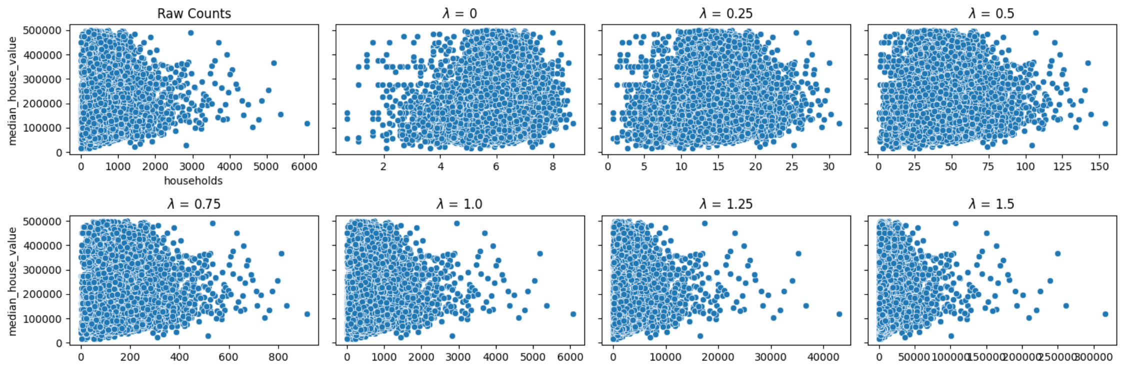
Because the parameter of interest is in the exponent, this type of transformation is called a **power transformation** and is implemented in sklearn's `PowerTransformer`. The parameter $\lambda$ is estimated from the data, and some values of $\lambda$ relate to common transformations, such as (for $y \geq 0$):

- $\lambda = 1$ (no transformation)
- $\lambda = 0$ (log)
- $\lambda = 0.5$ (square root)
- $\lambda = -1$ (inverse)

- Using the code below, if `lmbda=None` then the function will "find the lambda that maximizes the log-likelihood function and return it as the second output argument"
- Notice that we can not use `lambda` directly since it conflicts with the available object called `lambda`, this is the reason we preferred the indicator name as `lmbda`

[27]:
```
fig, axes = plt.subplots(figsize = (15,5), ncols = 4, nrows=2, sharey = True)
axes = axes.flatten()
sns.histplot(data = housing_uncapped['households'], ax = axes[0])
axes[0].set_title("Raw Counts")

for i, lmbda in enumerate([0, 0.25, 0.5, 0.75, 1., 1.25, 1.5]):

    house_box_ = stats.boxcox(housing_uncapped['households'].astype(float),␣
↪lmbda = lmbda)
    sns.histplot(data = house_box_, ax = axes[i + 1])
    axes[i + 1].set_title(r"$\lambda$ = {}".format(lmbda))

plt.tight_layout()
plt.show()

fig, axes = plt.subplots(figsize = (15,5), ncols = 4, nrows=2, sharey = True)
axes = axes.flatten()
sns.scatterplot(x = housing_uncapped['households'], y =␣
↪housing_uncapped["median_house_value"], ax = axes[0])
axes[0].set_title("Raw Counts")

for i, lmbda in enumerate([0, 0.25, 0.5, 0.75, 1., 1.25, 1.5]):

    house_box_ = stats.boxcox(housing_uncapped['households'].astype(float),␣
↪lmbda = lmbda)
    sns.scatterplot(x = house_box_, y = housing_uncapped["median_house_value"],␣
↪ax = axes[i + 1])
    axes[i + 1].set_title(r"$\lambda$ = {}".format(lmbda))

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






We can find the $\lambda$ that maximizes the log-likelihood function using scipy's `boxcox` function or sklearn's `PowerTransformer`.

[28]:

```python
# Find the MLE for lambda (using scipy's boxcox function)
house_box_, bc_params = stats.boxcox(housing_uncapped['households'].
 ↪astype(float), lmbda = None)
print(round(bc_params, 2))

# Find the MLE for lambda (using sklearn's PowerTransformer)
from sklearn.preprocessing import PowerTransformer
power_transformer = PowerTransformer(method='box-cox', standardize=False)
X_boxcox = power_transformer.fit_transform(housing_uncapped[['households']])
print(round(power_transformer.lambdas_[0], 2))
```

```
0.24
0.24
```

### 5.2.4 Exercise 15 (EXTRA)

- For the variable `households`, based on the `boxcox` transform shown above, do you think any of the values of $\lambda$ may be useful?
- Apply a similar code snippet to `median_house_value`. Would any values of $\lambda$ be useful?

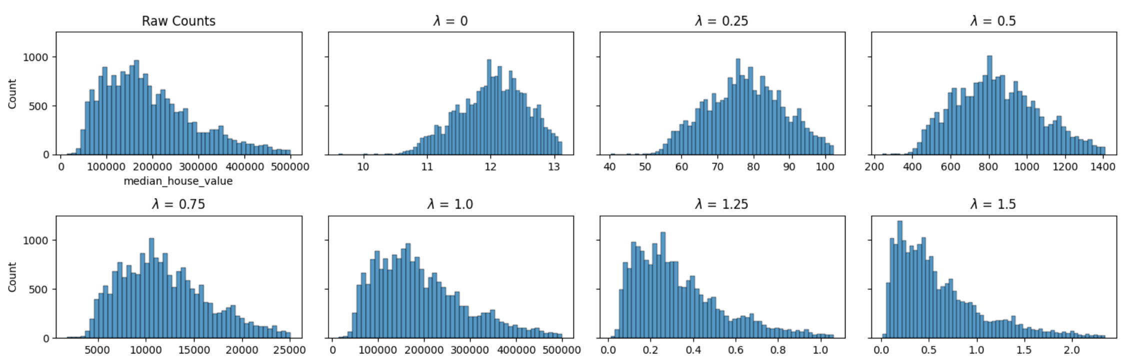
For households, lambda values of 0.24-0.25 are useful as it seems to give a more symmetrical distribution of data points.

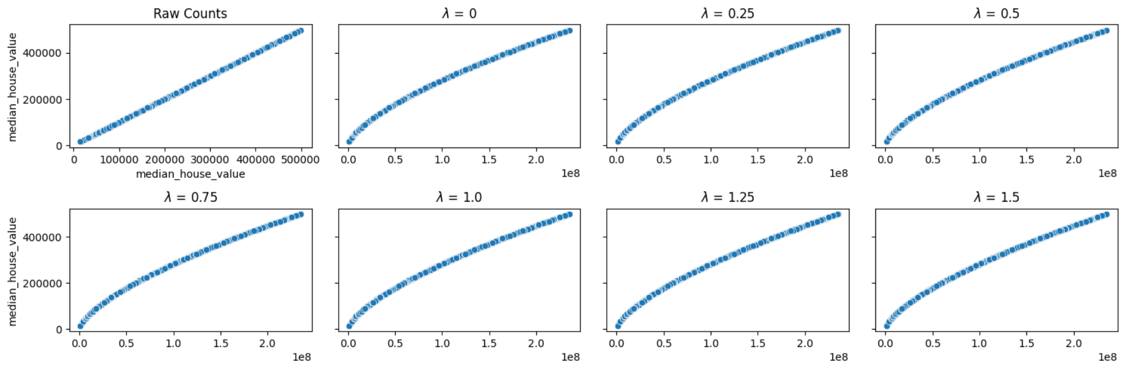
For median_house_value, a boxcox transformation may not be useful as there is little impact on the symmetry of distribution, and the transformation just adds complexity to data interpretation.

[29]:

```python
fig, axes = plt.subplots(figsize = (15,5), ncols = 4, nrows=2, sharey = True)
axes = axes.flatten()
sns.histplot(data = housing_uncapped['median_house_value'], ax = axes[0])
axes[0].set_title("Raw Counts")
```

```python
for i, lmbda in enumerate([0, 0.25, 0.5, 0.75, 1., 1.25, 1.5]):

    value_box_ = stats.boxcox(housing_uncapped['median_house_value'].astype(float), lmbda = lmbda)
    sns.histplot(data = value_box_, ax = axes[i + 1])
    axes[i + 1].set_title(r"$\lambda$ = {}".format(lmbda))

plt.tight_layout()
plt.show()

fig, axes = plt.subplots(figsize = (15,5), ncols = 4, nrows=2, sharey = True)
axes = axes.flatten()
sns.scatterplot(x = housing_uncapped['median_house_value'], y = housing_uncapped["median_house_value"], ax = axes[0])
axes[0].set_title("Raw Counts")

for i, lmbda in enumerate([0, 0.25, 0.5, 0.75, 1., 1.25, 1.5]):

    house_box_ = stats.boxcox(housing_uncapped['median_house_value'].astype(float), lmbda = lmbda)
    sns.scatterplot(x = value_box_, y = housing_uncapped["median_house_value"], ax = axes[i + 1])
    axes[i + 1].set_title(r"$\lambda$ = {}".format(lmbda))

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






## 5.3 Feature Combinations

- Looking at the datas attributes we may also want to manually combine them into features that are either a more meaningful representation of the data or have better properties.
- For example, we know **the number of rooms** in a district, but this may be more useful to combine with the **number of households** so that we have **a measure of rooms per household**.

[30]:
```
rooms_per_household = housing_uncapped['total_rooms'] /␣
↪housing_uncapped['households']
rooms_per_household.describe()
```

[30]:
```
count    19648.000000
mean         5.361708
std          2.293321
min          0.846154
25%          4.416667
50%          5.185730
75%          5.971083
max        132.533333
dtype: float64
```

### 5.3.1 Exercise 16 (EXTRA)

- Can you think of other combinations that may be useful?
- Create a custom transformer that creates these new combinations of features using the `FunctionTransformer`.

Hint

What about the following?

- `population_per_household`
- `bedrooms_per_room`

```
[31]: population_per_household = housing_uncapped['population'] /␣
       ↪housing_uncapped['households']
      bedrooms_per_room = housing_uncapped['total_bedrooms'] /␣
       ↪housing_uncapped['total_rooms']
      display(population_per_household.describe())
      display(bedrooms_per_room.describe())
```

```
count    19648.000000
mean         3.096560
std         10.639195
min          0.692308
25%          2.446614
50%          2.837779
75%          3.306021
max       1243.333333
dtype: float64
```

```
count    19448.000000
mean         0.214933
std          0.056922
min          0.100000
25%          0.177648
50%          0.204545
75%          0.240879
max          1.000000
dtype: float64
```

### 5.4 Other feature types

Feature engineering for other feature types beyond numerical categorical are also available in sklearn (e.g. for text and images) and feature engine (e.g. for Datetime and for time series).

**Now, is a good point to switch driver and navigator**

### 5.5 Combining into a Pipeline

Now, that we are familar with transformers, we are finally ready to create our first model pipeline!

Pipelines are very useful when we want to run data through our pipeline in the future; rather than having to copy and paste a load of code, we can just use our pipeline which combines all the steps. Later on the course, we will see this is important when we split our data into training, validation, and test sets, but this would also be required if you deploy your model in a "live" environment. In particular, pipelines help prevent you from **data leakage**, i.e. when information from your testing data leaks into your training or model selection. **Data leakage** is a common reason why many ML models fail to generalize to real world data. Furthermore, when refining a model, pipelines makes it easier for us to add or remove steps of our pipeline to see what works and what doesn't.

Its also worth examining what is meant by a **"Pipeline"**. A general definition is that it is just a sequence of data preparation operations that is ensured to be **reproducible**. Specifically, in sklearn, `Pipeline` can contains a sequence of *transformer* or *estimator* classes, or, if we use an imbalanced-learn `Pipeline` instead, also *resamplers*. This week we have focused on *transformers*, but later on in the course we will learn about *estimators* and *resamplers*. All three of these objects (*resamplers*, *transformers*, and *estimator*) all typically have a `.fit()` method. We have already seen examples of calling `.fit()` on *transformers*. The method works similarly on other classes and is used to - validate and interpret any parameters, - validate the input data, - estimate and store attributes from the parameters and provided data, - return the fitted estimator to facilitate method chaining in a pipeline.

Along with other sample properties (e.g. `sample_weight`), the `.fit()` method usually takes two inputs:

> - The input matrix (or design matrix) **X**. The size of **X** is typically (n_samples, n_features), which means that samples are represented as rows and features are represented as columns.
> - The target values **y** which are real numbers for regression tasks, or integers for classification (or any other discrete set of values). For unsupervised learning tasks, **y** does not need to be specified.
>
> https://scikit-learn.org/stable/getting_started.html

Other methods available for these objects other than `.fit()` will depend on what they are, e.g. `.transform()` for transformers, so we will learn about the methods for others objects later in the course.

This week, our focus is combining different feature engineering steps together to make different model pipelines.

- Remember we want to create a pipeline that treats the **numerical** and **categorical** attributes differently.
- We also need to supply the pipeline with an *estimator* (i.e. model). For now, let's use a linear regression model, which we will learn in more details in week 4.

[32]:
```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

numcols = features[:-1]
catcols = [features[-1]]

num_pre = Pipeline([
    ("num_impute", SimpleImputer(strategy="median")),
    ("num_scale", StandardScaler())])

cat_pre = Pipeline([
    ("cat_encode", OneHotEncoder(drop='first'))])

reg_pipe_1 = Pipeline([
    ("pre_processing", ColumnTransformer([("num_pre", num_pre, numcols),
                                          ("cat_pre", cat_pre, catcols)],
                                          verbose_feature_names_out=False)),
    ("model", LinearRegression())
```

```
])

# Alternative and equivalent model avoiding nested pipelines
# reg_pipe_1 = Pipeline([
#     ("impute", ColumnTransformer([("num_imp",␣
↪SimpleImputer(strategy="median"), numcols),
#                                   ("cat_imp",␣
↪SimpleImputer(strategy="constant"), catcols)])),
#     ("transform", ColumnTransformer([("num_trns", StandardScaler(), numcols),
#                                   ("cat_trns",␣
↪OneHotEncoder(drop='first'), catcols)])),                              ␣
↪
#     ("model", LinearRegression())
# ])

display(reg_pipe_1)
```

```
Pipeline(steps=[('pre_processing',
                 ColumnTransformer(transformers=[('num_pre',
                                                  Pipeline(steps=[('num_impute',
                                                                   ␣
↪SimpleImputer(strategy='median')),
                                                                  ('num_scale',
                                                                   ␣
↪StandardScaler())]),
                                                  Index(['longitude',␣
↪'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'median_house_value'],
      dtype='object')),
                                                 ('cat_pre',
                                                  Pipeline(steps=[('cat_encode',
                                                                   ␣
↪OneHotEncoder(drop='first'))]),
                                                  ['ocean_proximity'])],
                                   verbose_feature_names_out=False)),
                ('model', LinearRegression())])
```

[33]:
```
reg_pipe_1.fit(housing, housing["median_house_value"])
# Print the R squared (ranges 0 to 1, with higher values better)
print(round(reg_pipe_1.score(housing, housing["median_house_value"]), 3))
```

```
1.0
```

[34]:
```
# Print the coeffcients
coef_df = pd.DataFrame({'coef': reg_pipe_1['model'].coef_},
                       index = reg_pipe_1['pre_processing'].get_feature_names_out())
```

```
display(coef_df)
```

```
                                  coef
longitude                -1.038228e-10
latitude                 -4.638423e-11
housing_median_age        8.774570e-11
total_rooms              -4.661575e-11
total_bedrooms            1.573092e-11
population               -1.160053e-11
households                2.559008e-11
median_income             1.480201e-10
median_house_value        1.153928e+05
ocean_proximity_INLAND   -9.987967e-11
ocean_proximity_ISLAND   -5.357351e-11
ocean_proximity_NEAR BAY  4.429155e-11
ocean_proximity_NEAR OCEAN -5.156455e-11
```

Let's try some other combinations of the pre-processing and feature engineering steps that we have learned about this week.

[35]:
```
# Reg Pipe 2

# Define column indices
numcols = ['longitude', 'latitude', 'housing_median_age', 'median_income']
countcols = ['total_rooms', 'total_bedrooms', 'population', 'households']

# Reg Pipe 2
num_pre = Pipeline([
    ("num_scale", StandardScaler())])

count_pre = Pipeline([
    ("count_impute", SimpleImputer(strategy="median")),
    ("count_transform", PowerTransformer(method='box-cox', standardize=True))])

cat_pre = Pipeline([
    ("cat_encode", OneHotEncoder(drop='first'))])

# Overall ML pipeline inlcuding all
reg_pipe_2 = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("num_pre", num_pre, numcols),
        ("count_pre", count_pre, countcols),
        ("cat_pre", cat_pre, catcols)], verbose_feature_names_out=False)),
    ("model", LinearRegression())
])


display(reg_pipe_2)
```

```
Pipeline(steps=[('pre_processing',
                 ColumnTransformer(transformers=[('num_pre',
                                                  Pipeline(steps=[('num_scale',
                                                                   ␣
 ↪StandardScaler())]),
                                                  ['longitude', 'latitude',
                                                   'housing_median_age',
                                                   'median_income']),
                                                 ('count_pre',
                                                 ␣
 ↪Pipeline(steps=[('count_impute',
                                                                   ␣
 ↪SimpleImputer(strategy='median')),
                                                                  ␣
 ↪('count_transform',
                                                                   ␣
 ↪PowerTransformer(method='box-cox'))]),
                                                  ['total_rooms',
                                                   'total_bedrooms',
                                                   'population',
                                                   'households']),
                                                 ('cat_pre',
                                                  Pipeline(steps=[('cat_encode',
                                                                   ␣
 ↪OneHotEncoder(drop='first'))]),
                                                  ['ocean_proximity'])],
                                   verbose_feature_names_out=False)),
                ('model', LinearRegression())])
```

[36]:
```
reg_pipe_2.fit(housing, housing["median_house_value"])
# Print the R squared (ranges 0 to 1, with higher values better)
print(round(reg_pipe_2.score(housing, housing["median_house_value"]), 3))
```

```
0.668
```

[37]:
```
# Print the coeffcients
coef_df = pd.DataFrame({'coef': reg_pipe_2['model'].coef_},
             index = reg_pipe_2['pre_processing'].get_feature_names_out())
display(coef_df)
```

| | coef |
|---|---|
| longitude | -56830.218169 |
| latitude | -59129.677527 |
| housing_median_age | 12933.426473 |
| median_income | 76934.983259 |
| total_rooms | -22094.529133 |
| total_bedrooms | 45298.071656 |
| population | -65930.689885 |

```
households                       46349.287540
ocean_proximity_INLAND          -35349.967012
ocean_proximity_ISLAND          134076.721365
ocean_proximity_NEAR BAY         -7618.608513
ocean_proximity_NEAR OCEAN        -702.953510
```

[38]:

```
# Reg Pipe 3
from feature_engine.transformation import LogTransformer
from sklearn.compose import TransformedTargetRegressor

numcols = ['longitude', 'latitude', 'housing_median_age']
skewcols = ['total_rooms', 'total_bedrooms', 'population', 'households',␣
 ↪'median_income']

num_pre = Pipeline([
    ("num_scale", StandardScaler())])

skew_pre = Pipeline([
    ("skew_impute", SimpleImputer(strategy="median")),
    ("skew_transform", LogTransformer()),
    ("skew_scale", StandardScaler())])

cat_pre = Pipeline([
    ("cat_encode", OneHotEncoder(drop='first'))])

# Overall ML pipeline inlcuding all
reg_pipe_3 = Pipeline([
    ("pre_processing", ColumnTransformer([
        ("num_pre", num_pre, numcols),
        ("skew_pre", skew_pre, skewcols),
        ("cat_pre", cat_pre, catcols)], verbose_feature_names_out=False)),
    ("model", LinearRegression())
])
# Transform also the target variable
tt_reg_pipe_3 =TransformedTargetRegressor(regressor=reg_pipe_3,
                                  transformer=LogTransformer())

display(tt_reg_pipe_3)
```

```
TransformedTargetRegressor(regressor=Pipeline(steps=[('pre_processing',
                                                                    ␣
 ↪ColumnTransformer(transformers=[('num_pre',
                                                                             ␣
 ↪       Pipeline(steps=[('num_scale',
                                                                             ␣
 ↪                        StandardScaler())]),
                                                                             ␣
 ↪       ['longitude',
```

```
↪       'latitude',
↪       'housing_median_age']),
↪     ('skew_pre',
↪      Pipeline(steps=[('skew_impute',
↪                       SimpleImputer(strategy='median')),
↪                      ('skew_transform',
↪                       LogTransformer()),
↪                      ('skew_scale',
↪                       StandardScaler())]),
↪      ['total_rooms',
↪       'total_bedrooms',
↪       'population',
↪       'households',
↪       'median_income']),
↪     ('cat_pre',
↪      Pipeline(steps=[('cat_encode',
↪                       OneHotEncoder(drop='first'))]),
↪      ['ocean_proximity'])],
↪verbose_feature_names_out=False)),
                                                     ('model',
                                                      LinearRegression())]),
                           transformer=LogTransformer())
```

[39]:
```python
tt_reg_pipe_3.fit(housing, housing["median_house_value"])
# Print the R squared (ranges 0 to 1, with higher values better)
print(round(tt_reg_pipe_3.score(housing, housing["median_house_value"]), 3))
```

```
0.666
```

```
[40]: # Print the coeffcients
# Note: get_feature_names_out() does not work for LogTransformer
reg3_features = np.concatenate([tt_reg_pipe_3.
 ↪regressor_['pre_processing']['num_pre'].get_feature_names_out(),
               tt_reg_pipe_3.regressor_['pre_processing']['skew_pre'].
 ↪feature_names_in_,
               tt_reg_pipe_3.regressor_['pre_processing']['cat_pre'].
 ↪get_feature_names_out()]
)

coef_df = pd.DataFrame({'coef': tt_reg_pipe_3.regressor_['model'].coef_},
             index = reg3_features)
display(coef_df)
```

```
                              coef
longitude                -0.325886
latitude                 -0.346085
housing_median_age        0.032441
total_rooms              -0.095483
total_bedrooms            0.209126
population               -0.294461
households                0.188207
median_income             0.342995
ocean_proximity_INLAND   -0.280706
ocean_proximity_ISLAND    0.465280
ocean_proximity_NEAR BAY -0.044564
ocean_proximity_NEAR OCEAN -0.046772
```

### 5.5.1 Exercise 17 (CORE)

Explain in words what are the differences in pre-processing and/or feature engineering steps used across the three model pipelines above.

Pipeline 1: Basic approach

- Treats all numeric features the same way: imputes missing values with the median, then applies standard scaling
- Separates out only the categorical feature (the last column) for one-hot encoding
- No distinction between different types of numeric variables

Pipeline 2: Differentiated treatment by variable type

- Splits numeric features into two groups based on their nature: 1) Regular numeric features (longitude, latitude, housing_median_age, median_income): only scaled, no imputation, 2) Count features (total_rooms, total_bedrooms, population, households): imputed with median, then transformed using Box-Cox power transformation to handle skewness, which also standardises them
- Recognises that count variables often have skewed distributions and need special handling
- Assumes the regular numeric features don't have missing values (no imputation step)

Pipeline 3: Focus on skewness correction

- Also splits features into groups, but differently: 1) Standard numeric features (longitude, latitude, housing_median_age): only scaled, 2) Skewed features (total_rooms, total_bedrooms, population, households, median_income): imputed with median, log-transformed via a custom LogTransformer, then scaled
- Uses log transformation instead of Box-Cox to address skewness
- Includes median_income in the skewed group (unlike Pipeline 2, which treats it as a regular numeric feature)
- The log transformation followed by scaling is a more traditional approach compared to Box-Cox

#### 5.5.2 Exercise 18 (EXTRA)

Try to create your own pipeline by modifying at least one of the pre-processing and feature engineering steps above. What have you decided to change and why?

## 6 Summary

This week we covered a lot of ground!

- We've looked at some methods for pre-processing our data, cleaning and preparing it, as well as how to engineer some features and combine these steps into a reproducible pipeline.
- This is by **no means a complete collection of all the methods available** as covering more would go beyond the scope of this course (for those interested in learning more, have a look though the given companion readings).
- For example, we did not touch on handling text and dates/time much. These topics are quite complex and have enough materials to cover their own courses.

## 7 Competing the Worksheet

At this point you have hopefully been able to complete all the CORE exercises and attempted the EXTRA ones. Now is a good time to check the reproducibility of this document by restarting the notebook's kernel and rerunning all cells in order.

Before generating the PDF, please **change 'Student 1' and 'Student 2' at the top of the notebook to include your name(s)**.

Once that is done and you are happy with everything, you can then run the following cell to generate your PDF.

[41]:
```
# !jupyter nbconvert --to pdf mlp_week01.ipynb
```

Once generated, please submit this PDF on Learn page by 16:00 PM on the Friday of the week the workshop was given. Note that:

- You don't need to finish everything, but you should have had a substantial attempt at the bulk of the material, particularly the CORE tasks.
- If you are having trouble generating the pdf, please ask a tutor or post on piazza.

- As a back option, if you are having errors in converting to pdf, then a quick solution is to export to html and then convert to pdf in your browser.