Algorithms and Data Analysis

-演算法與資料分析-

-Introduction to Artificial Intelligence-

機器學習專案

授課教師：張珀銀 老師

簡報大綱

1. Look at the big picture.

2. Get the data.

3. Discover and visualize the data to gain insights.

4. Prepare the data for Machine Learning algorithms.

5. Select a model and train it.

6. Fine-tune your model.

7. Present your solution.

8. Launch, monitor, and maintain your system.

6 Working with Real Data and SciPy

Working with Real Data and SciPy

- When you are learning about Machine Learning, it is best to experiment with real-world data, not artificial datasets.
- Here are a few places you can look to get data:
- Popular open data repositories
  - UCI Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets

Working with Real Data and SciPy

- Meta portals (they list open data repositories)
  - Data Portals
  - OpenDataMonitor
  - Quandl
- Other pages listing many popular open data repositories
  - Wikipedia's list of Machine Learning datasets
  - Quora.com
  - The datasets subreddit

Working with Real Data and SciPy

- We'll use the California Housing Prices dataset from the StatLib repository

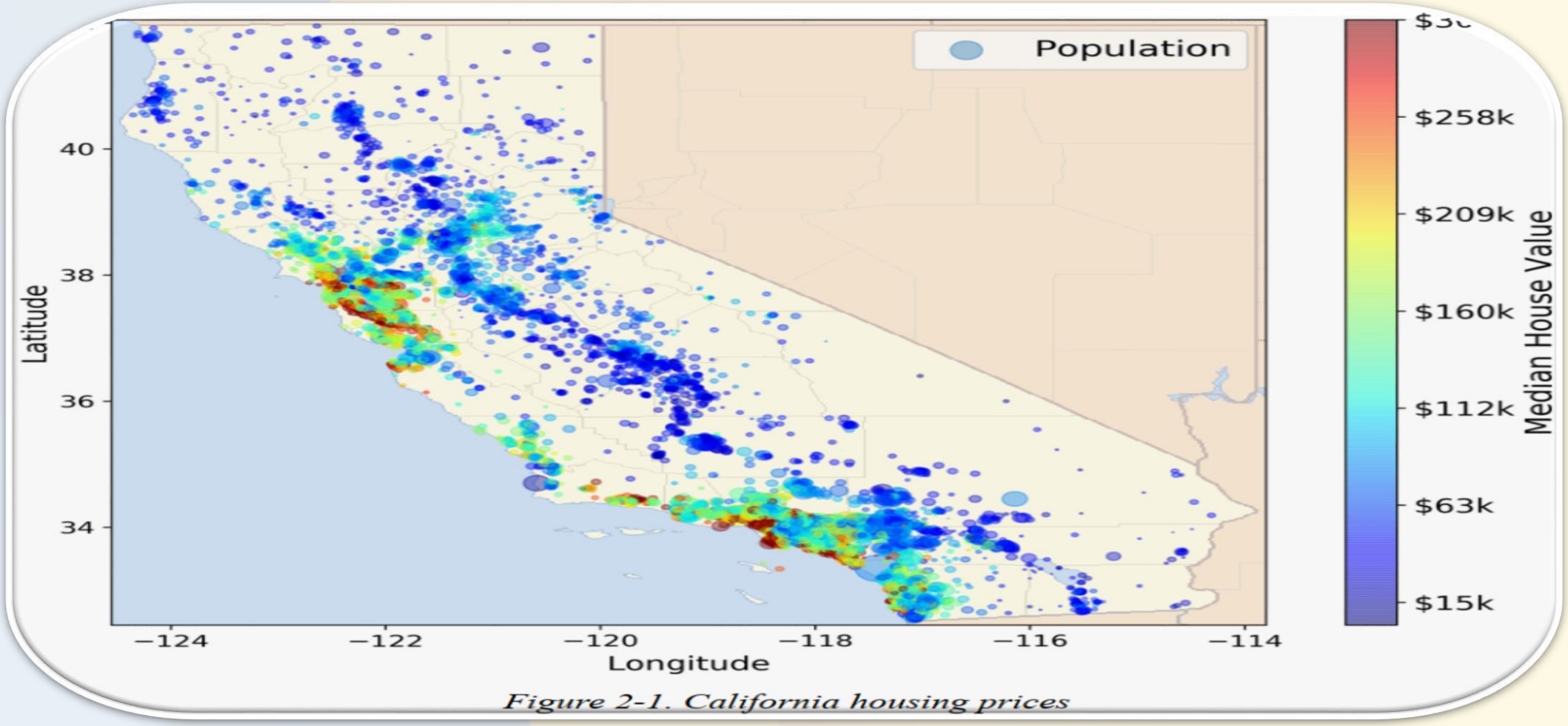


Figure 2-1. California housing prices

1 Look at the Big Picture

Look at the Big Picture

- This data includes metrics such as the population, median income, and median housing price 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).
- Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

Frame the Problem

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

Frame the Problem

- Your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system along with many other signals.
- This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.

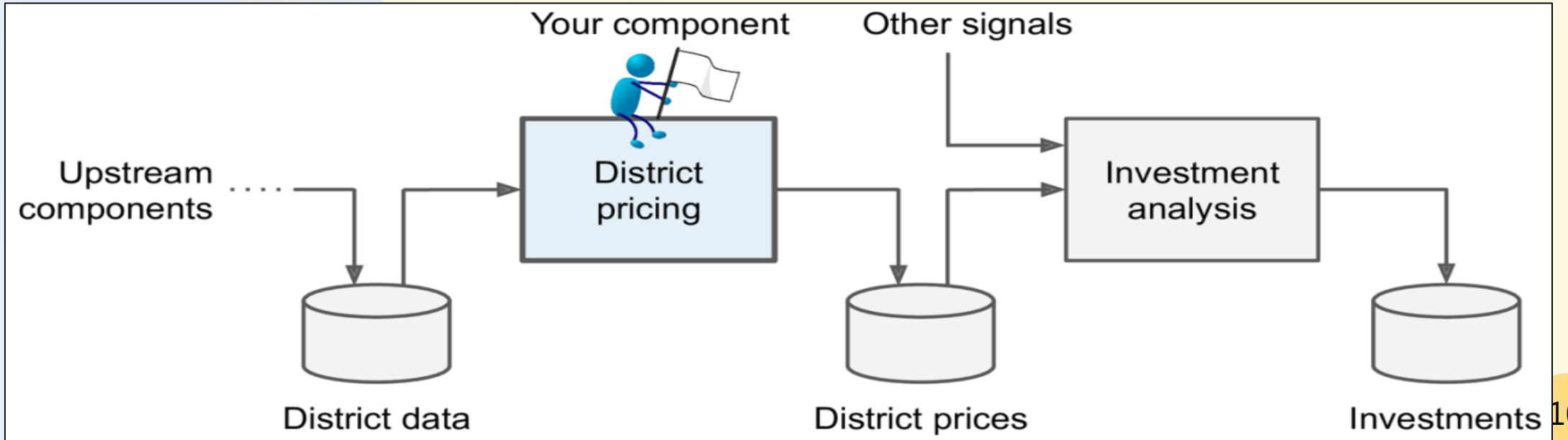


Figure 2-2. A Machine Learning pipeline for real estate investments

Frame the Problem

- The current situation will often give you a reference for performance, as well as insights on how to solve the problem.
- Current situation : 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%**.

Frame the Problem

- First, you need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning?
- Is it a classification task, a regression task, or something else?
- Should you use batch learning or online learning techniques?
- Before you read on, pause and try to answer these questions for yourself.

Frame the Problem

- it is clearly a typical **supervised learning task**, since you are given labeled training examples (each instance comes with the expected output, i.e., the district's median housing price).
- It is also a typical **regression task**, since you are asked to predict a value.
- This is a **multiple regression problem**, since the system will **use multiple features** to make a prediction (it will use the district's population, the median income, etc.).
- It is also a **univariate regression problem**, since we are only trying to predict a single value for each district.
- If we were trying to predict multiple values per district, it would be a **multivariate regression problem.**

Frame the Problem

1

- Finally, there is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly
- the data is small enough to fit in memory, so **plain batch learning** should do just fine.

Select a Performance Measure

- A typical performance measure for regression problems is the Root Mean Square Error (RMSE).
- It gives an idea of how much error the system typically makes in its predictions, with **a higher weight for large errors.**
- Equation 2-1 shows the mathematical formula to compute the RMSE:

Equation 2-1. Root Mean Square Error (RMSE)

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Select a Performance Measure

- In some contexts you may prefer to use another function. For example, suppose that there are many outlier districts.
- In that case, you may consider using the mean absolute error (MAE, also called the average absolute deviation; see Equation 2-2):

Equation 2-2. Mean absolute error (MAE)

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

Check the Assumptions

- Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others); this can help you catch serious issues early on.
- But what if the downstream system converts the prices into categories (e.g., “cheap,” “medium,” or “expensive”) and then uses those categories instead of the prices themselves?
- In this case, getting the price perfectly right is not important at all; your system just needs to get the category right
- You don’t want to find this out after working on a regression system for months.

2 Get the Data

Download the Data

- This function returns a pandas DataFrame object containing all the data.

```
import pandas as pd

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

Take a Quick Look at the Data Structure

- Let's take a look at the top five rows using the DataFrame's head() method (see Figure 2-5).

- There are 10 attributes (you can see the first 6 in the screenshot): longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value, and ocean_proximity.

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
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

Figure 2-5. Top five rows in the dataset

Take a Quick Look at the Data Structure

- The info() method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of nonnull values (see Figure 2-6).

```
In [6]: housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 2-6. Housing info

Take a Quick Look at the Data Structure

- There are 20,640 instances in the dataset, which means that it is fairly small by Machine Learning standards
- but it's perfect to get started. Notice that the total_bedrooms attribute has only 20,433 nonnull values, meaning that 207 districts are missing this feature. We will need to take care of this later.

```
In [6]: housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude           20640 non-null float64
latitude            20640 non-null float64
housing_median_age  20640 non-null float64
total_rooms         20640 non-null float64
total_bedrooms      20433 non-null float64
population          20640 non-null float64
households          20640 non-null float64
median_income       20640 non-null float64
median_house_value  20640 non-null float64
ocean_proximity     20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 2-6. Housing info

Take a Quick Look at the Data Structure

- All attributes are numerical, except the ocean_proximity field.
- Its type is object, so it could hold any kind of Python object. But since you loaded this data from a CSV file, you know that it must be a text attribute.
- When you looked at the top five rows, you probably noticed that the values in the ocean_proximity column were repetitive, which means that it is probably a categorical attribute.

```
households                  20640 non-null float64
median_income               20640 non-null float64
median_house_value          20640 non-null float64
ocean_proximity             20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 2-6. Housing info

Take a Quick Look at the Data Structure

- You can find out what categories exist and how many districts belong to each category 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
```

Take a Quick Look at the Data Structure

- Let's look at the other fields. The describe() method shows a summary of the numerical attributes (Figure 2-7).

```
In [8]: housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.38507(
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000(
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000(
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000(
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000(

Figure 2-7. Summary of each numerical attribute

Take a Quick Look at the Data Structure

- The count, mean, min, and max rows are self-explanatory. Note that the null values are ignored (so, for example, the count of total_bedrooms is 20,433, not 20,640).
- The std row shows the standard deviation, which measures how dispersed the values are
- The 25%, 50%, and 75% rows show the corresponding percentiles: a percentile indicates the value below which a given percentage of observations in a group of observations fall.

Take a Quick Look at the Data Structure

- A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis).
- You can either plot this one attribute at a time, or you can call the hist() method on the whole dataset (as shown in the following code example), and it will plot a histogram for each numerical attribute

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

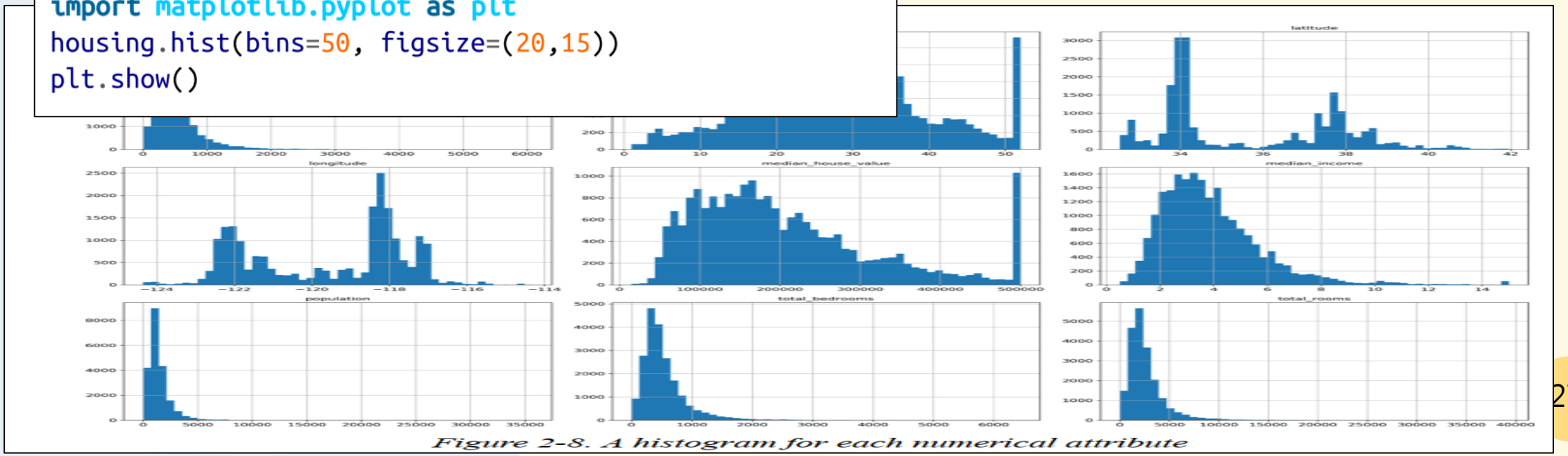


Figure 2-8. A histogram for each numerical attribute

Take a Quick Look at the Data Structure

- There are a few things you might notice in these histograms:
- First, the **median income** attribute **does not look like** it is expressed in US dollars (USD).
- After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes.
- The numbers represent roughly **tens of thousands** of dollars (e.g., 3 actually means about $30,000).

Take a Quick Look at the Data Structure

The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels).

- Your Machine Learning algorithms may learn that prices never go beyond that limit.
- You need to check with your client team (the team that will use your system's output) to see if this is a problem or not.

Take a Quick Look at the Data Structure

If they tell you that they need precise predictions even beyond $500,000, then you have two options:

- a. Collect proper labels for the districts whose labels were capped.
- b. Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond $500,000).

Take a Quick Look at the Data Structure

Finally, many histograms are tail-heavy:

- they extend much farther to the right of the median than to the left. This may make it a bit harder for some Machine Learning algorithms to detect patterns.
- We will try transforming these attributes later on to have more bell-shaped distributions.

Create a Test Set

- If you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model.
- When you estimate the generalization error using the test set, your estimate will be too optimistic, and you will launch a system that will not perform as well as expected.
- This is called data snooping bias.

Create a Test Set

- Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset (or less if your dataset is very large), and set them aside:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

- You can then use this function like this

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

Create a Test Set

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set!

Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid.

- One solution is to save the test set on the first run and then load it in subsequent runs.
- Another option is to set the random number generator's seed (e.g., with np.random.seed before calling np.random.permutation() so that it always generates the same shuffled indices.

Create a Test Set

But both these solutions will break the next time you fetch an updated dataset.

- To have a stable train/test split even after updating the dataset, a common solution is to use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier).

Create a Test Set

- Compute a hash of each instance's identifier and put that instance in the test set if the hash is lower than or equal to 20% of the maximum hash value. (This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset.)
- The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set.
- Here is a possible implementation:

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Create a Test Set

Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted.

- If this is not possible, then you can try to use the most stable features to build a unique identifier.
- For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so

Create a Test Set

- For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

Create a Test Set

- Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways.
- The simplest function is train_test_split(), which does pretty much the same thing as the function split_train_test(), with a couple of additional features.
- There is a **random_state** parameter that allows you to set the random generator seed. Second, **you can pass it multiple datasets** with an identical number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

```
from sklearn.model_selection import train_test_split

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

Create a Test Set

- 目前為止我們考慮的是純隨機抽樣方法。這一般是如果你的資料集足夠大，就OK（尤其是相對於你的資料集的屬性數量），但如果不是這樣，您就會面臨引入大量抽樣的偏差風險。

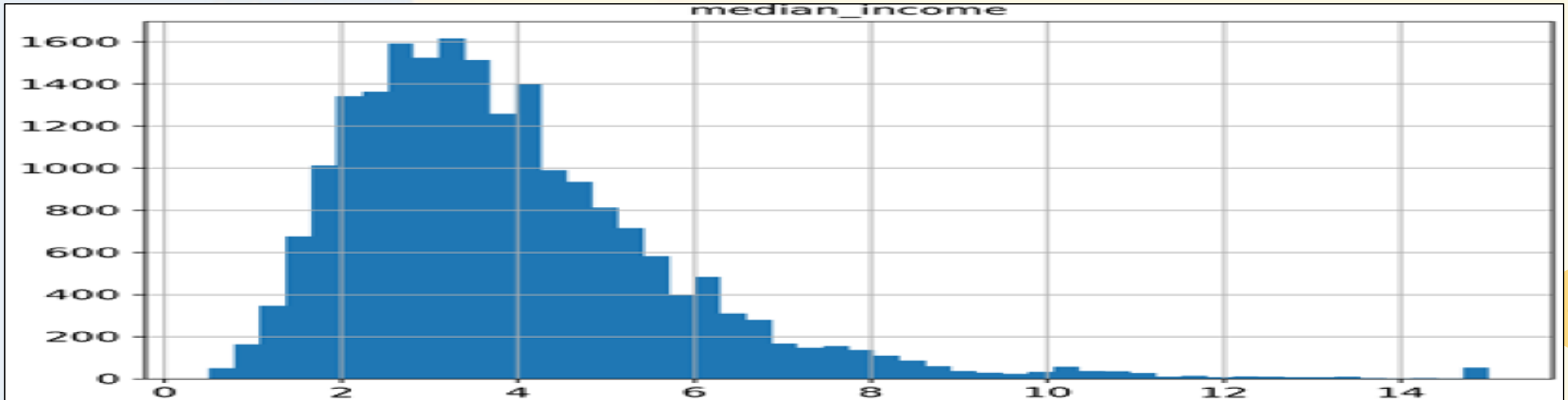
- 當一家調查公司決定電話訪查1000人，他們不只是隨機在電話簿挑選1000人，並嘗試確保這1,000人能夠代表整個人口。例如，美國人口中女性占51.3%，男性占48.7%，因此，要獲得良好調查品質的電話訪查會儘量保持這個比例在美國進行，即513 女性和487名男性。

- **分層抽樣(strata)**：總體被劃分為稱為子層的同質子組，並從每個層中採樣了正確數量的實例，以確保測試集可以代表總體。

- 如果調查使用完全隨機抽樣，將有大約12%的幾率抽到傾斜的測試集，女性比例低於49%或高於54%。皆會產生極大偏差。

Create a Test Set

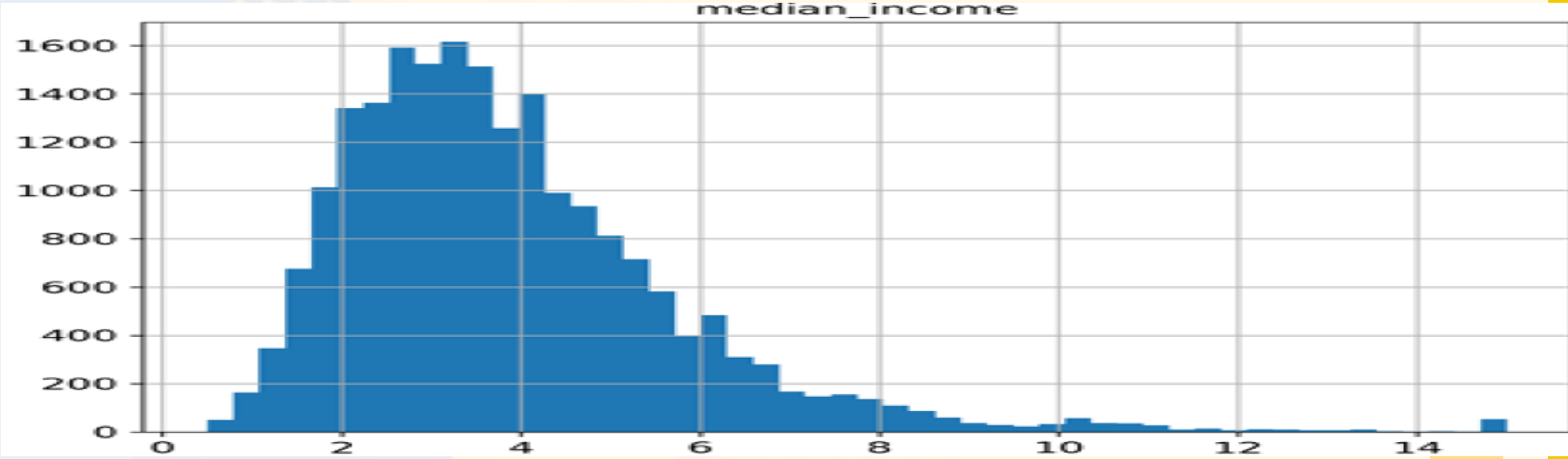
- Suppose you chatted with experts who told you that the median income is a very important attribute to predict median housing prices.
- You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset.
- Since the median income is a continuous numerical attribute, you first need to create an income category attribute. Let's look at the median income histogram more closely (back in Figure 2-8)



Create a Test Set

- most median income values are clustered around **1.5 to 6** (i.e., $15,000–$60,000), but some median incomes go far beyond 6.

- It is important to **have a sufficient number of instances in your dataset for each stratum**, or **else the estimate of a stratum's importance may be biased**.



Hands-on Machine Learning with Scikit-Learn, Keras and TensorFlow:

Create a Test Set

This means that you should not have too many strata, and each stratum should be large enough.

- The following code uses the pd.cut() function to create an income category attribute with five categories (labeled from 1 to 5): category 1 ranges from 0 to 1.5 (i.e., less than $15,000), category 2 from 1.5 to 3, and so on:

```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

Create a Test Set

These income categories are represented in Figure 2-9:

```
housing["income_cat"].hist()
```

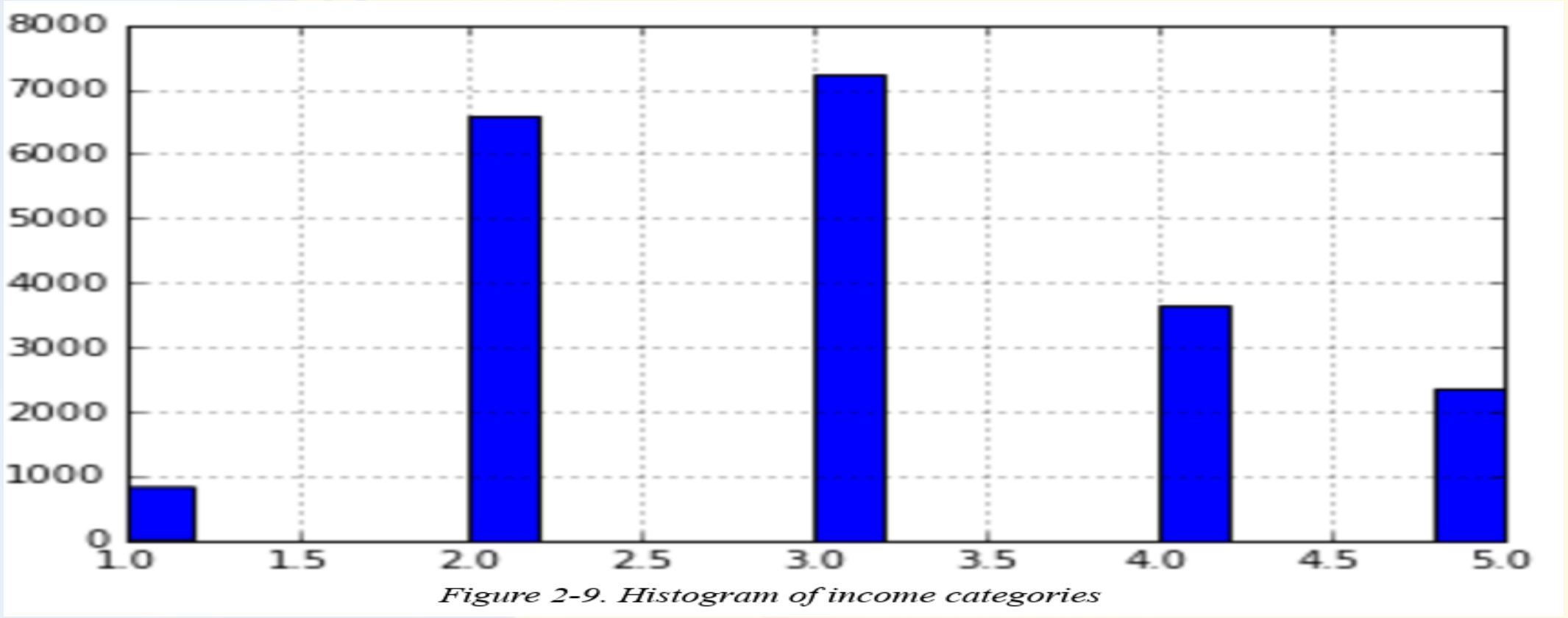


Figure 2-9. Histogram of income categories

Create a Test Set

Now you are ready to do stratified sampling based on the income category.

- For this you can use Scikit-Learn's StratifiedShuffleSplit class:

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Create a Test Set

Let's see if this worked as expected. You can start by looking at the income category proportions in the test set:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

Create a Test Set

With similar code you can measure the income category proportions in the full dataset. Figure 2-10 compares the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling.

- As you can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is skewed

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

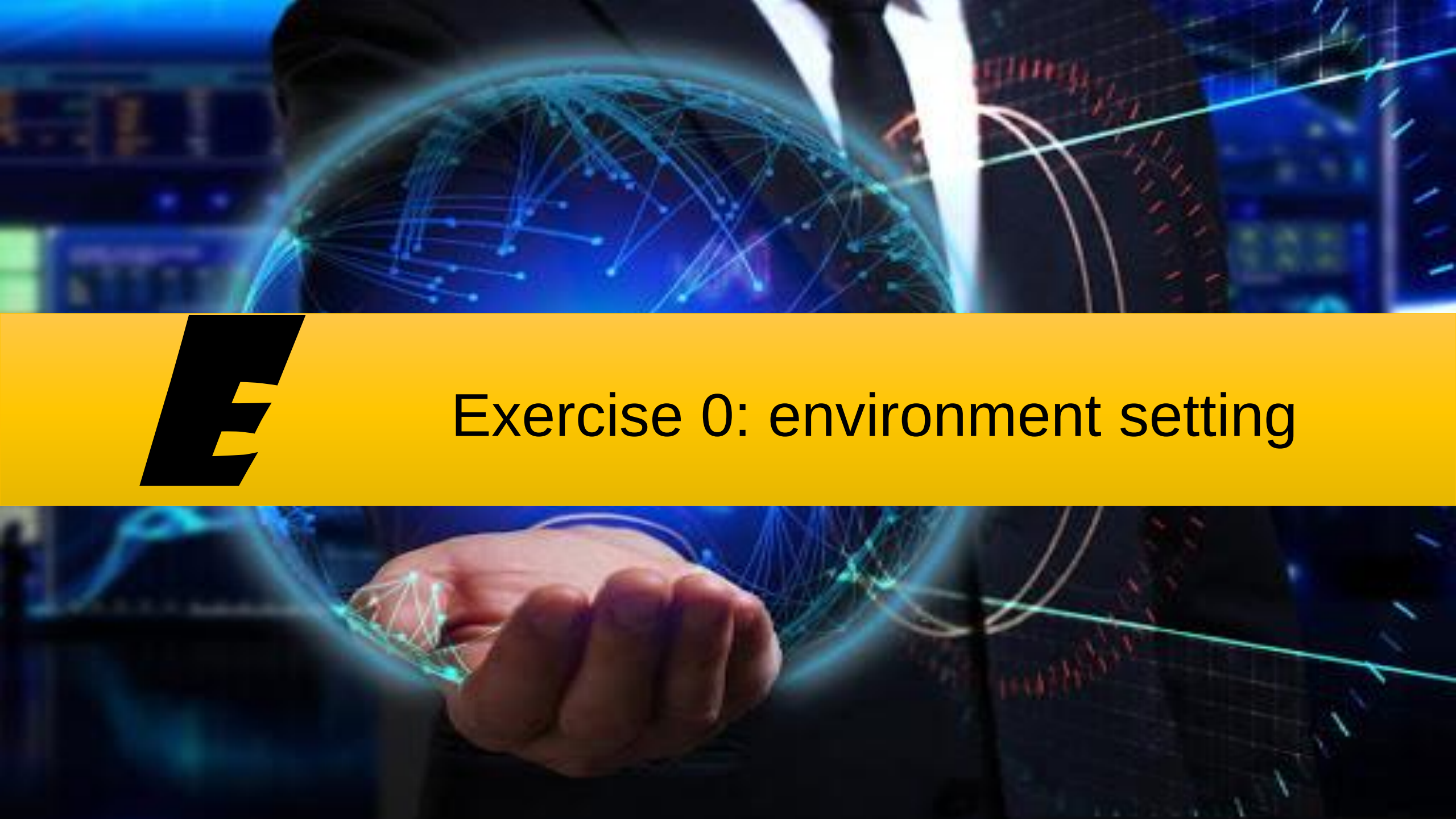
Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

Create a Test Set

Now you should remove the income_cat attribute so the data is back to its original state:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

- We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a Machine Learning project. Moreover, many of these ideas will be useful later when we discuss cross-validation. Now it's time to move on to the next stage: exploring the data.



Exercise 0: environment setting

In [1]:

```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

# Where to save the figures
PROJECT_ROOT_DIR = "."
CHAPTER_ID = "end_to_end_project"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "images", CHAPTER_ID)
os.makedirs(IMAGES_PATH, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = os.path.join(IMAGES_PATH, fig_id + "." + fig_extension)
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```


v:

Exercise 1: Get the Data

In [4]:

```
import pandas as pd

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

Take a Quick Look at the Data Structure

In [5]:
```
housing = load_housing_data()
housing.head()
```

Out[5]:

	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

- housing.info()
- housing["ocean_proximity"].value_counts()

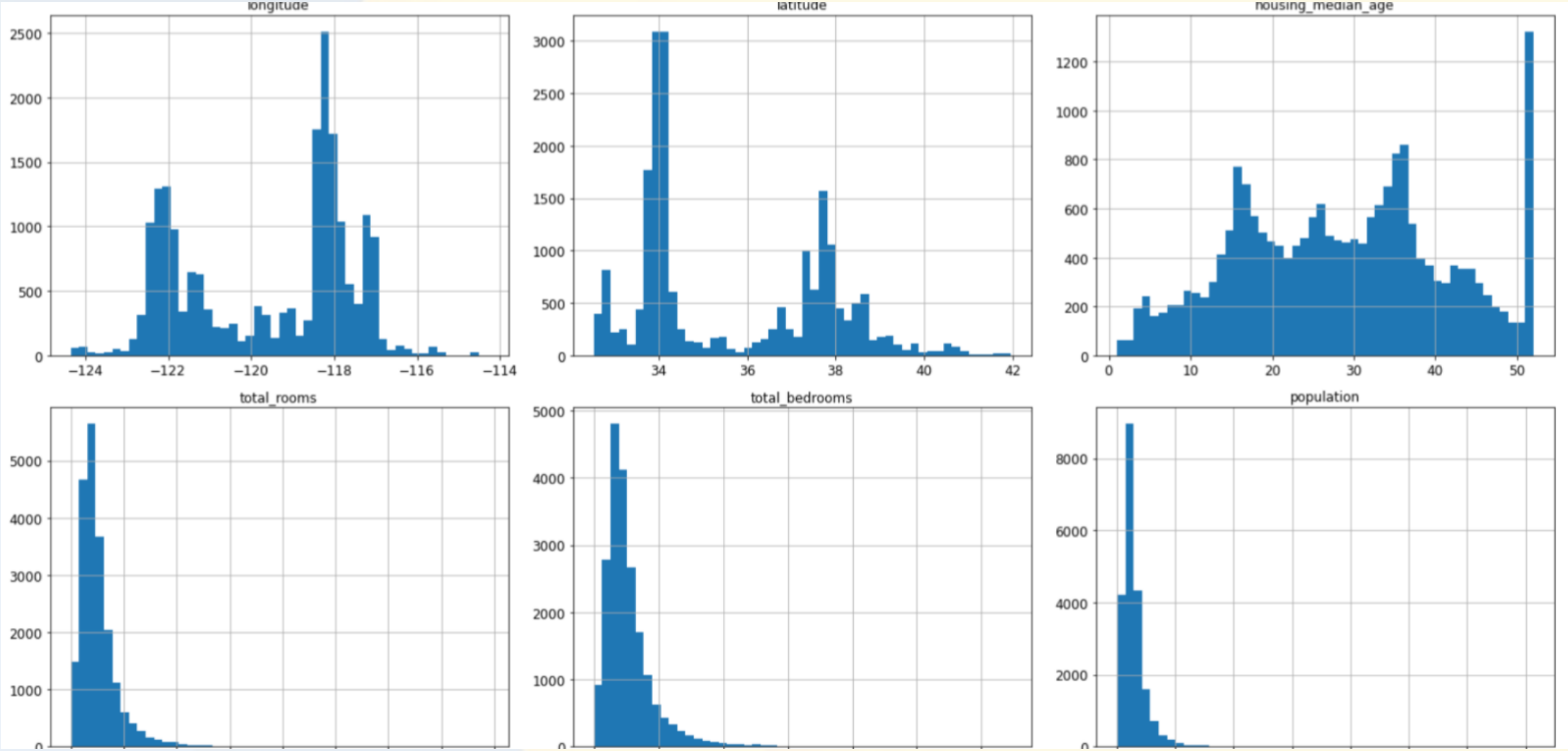
In [8]:
```
housing.describe()
```

Out[8]:

	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

n [9]:

```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
```


longitude
2500
2000
1500
1000
500
0
−124
−122
−120
−118
−116
−114
latitude
3000
2500
2000
1500
1000
500
0
34
36
38
40
42
housing_median_age
1200
1000
800
600
400
200
0
0
10
20
30
40
50
total_rooms
5000
4000
3000
2000
1000
0
total_bedrooms
5000
4000
3000
2000
1000
0
population
8000
6000
4000
2000
0

Exercise 2: Create a Test Set

n [9]:

```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
```

In [10]:
```
# to make this notebook's output identical at every run
np.random.seed(42)
```

In [11]:
```
import numpy as np

# For illustration only. Sklearn has train_test_split()
def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

In [12]:
```
train_set, test_set = split_train_test(housing, 0.2)
len(train_set)
```

Out[12]:

In [13]:
```
len(test_set)
```

In [14]:
```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

The implementation of `test_set_check()` above works fine in both Python 2 and Python 3. In earlier releases, the following implementation was proposed, which supported any hash function, but was much slower and did not support Python 2:

In [15]:
```
import hashlib

def test_set_check(identifier, test_ratio, hash=hashlib.md5):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio
```

If you want an implementation that supports any hash function and is compatible with both Python 2 and Python 3, here is one:

In [16]:
```
def test_set_check(identifier, test_ratio, hash=hashlib.md5):
    return bytearray(hash(np.int64(identifier)).digest())[-1] < 256 * test_ratio
```

In [17]:
```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

In [18]:
```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

In [19]:
```
test_set.head()
```

Out[19]:

	index	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity	id
8	8	-122.26	37.84	42.0	2555.0	665.0	1206.0	595.0	2.0804	226700.0	NEAR BAY	-122222.16
10	10	-122.26	37.85	52.0	2202.0	434.0	910.0	402.0	3.2031	281500.0	NEAR BAY	-122222.15
11	11	-122.26	37.85	52.0	3503.0	752.0	1504.0	734.0	3.2705	241800.0	NEAR BAY	-122222.15
12	12	-122.26	37.85	52.0	2491.0	474.0	1098.0	468.0	3.0750	213500.0	NEAR BAY	-122222.15
13	13	-122.26	37.84	52.0	696.0	191.0	345.0	174.0	2.6736	191300.0	NEAR BAY	-122222.16

In [20]:
```
from sklearn.model_selection import train_test_split

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

In [21]:
```
test_set.head()
```

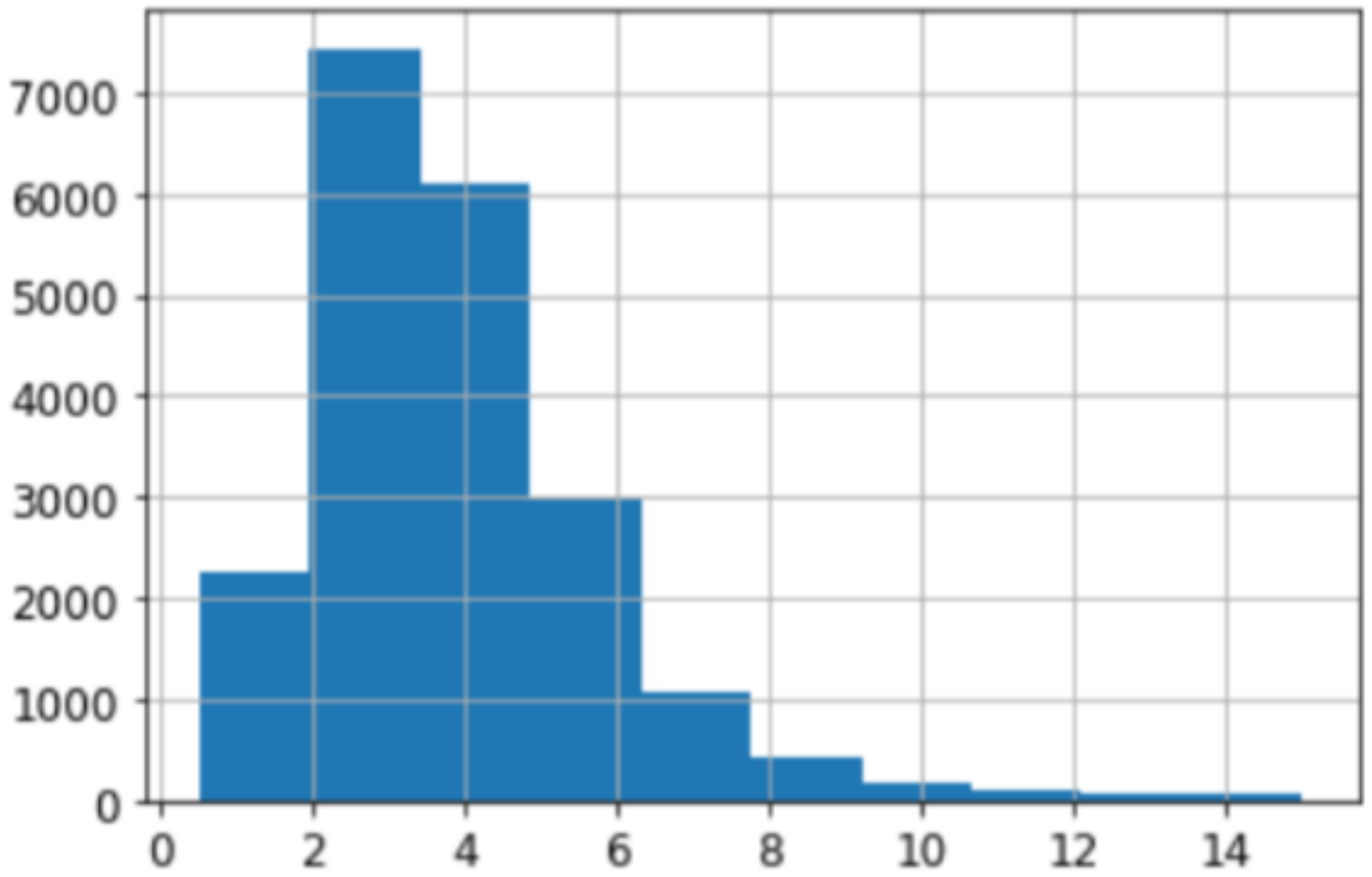
Out[21]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
20046	-119.01	36.06	25.0	1505.0	NaN	1392.0	359.0	1.6812	47700.0	INLAND
3024	-119.46	35.14	30.0	2943.0	NaN	1565.0	584.0	2.5313	45800.0	INLAND
15663	-122.44	37.80	52.0	3830.0	NaN	1310.0	963.0	3.4801	500001.0	NEAR BAY
20484	-118.72	34.28	17.0	3051.0	NaN	1705.0	495.0	5.7376	218600.0	<1H OCEAN
9814	-121.93	36.62	34.0	2351.0	NaN	1063.0	428.0	3.7250	278000.0	NEAR OCEAN

Exercise 3: Histogram

In [22]:
```
housing["median_income"].hist()
```

Out[22]:



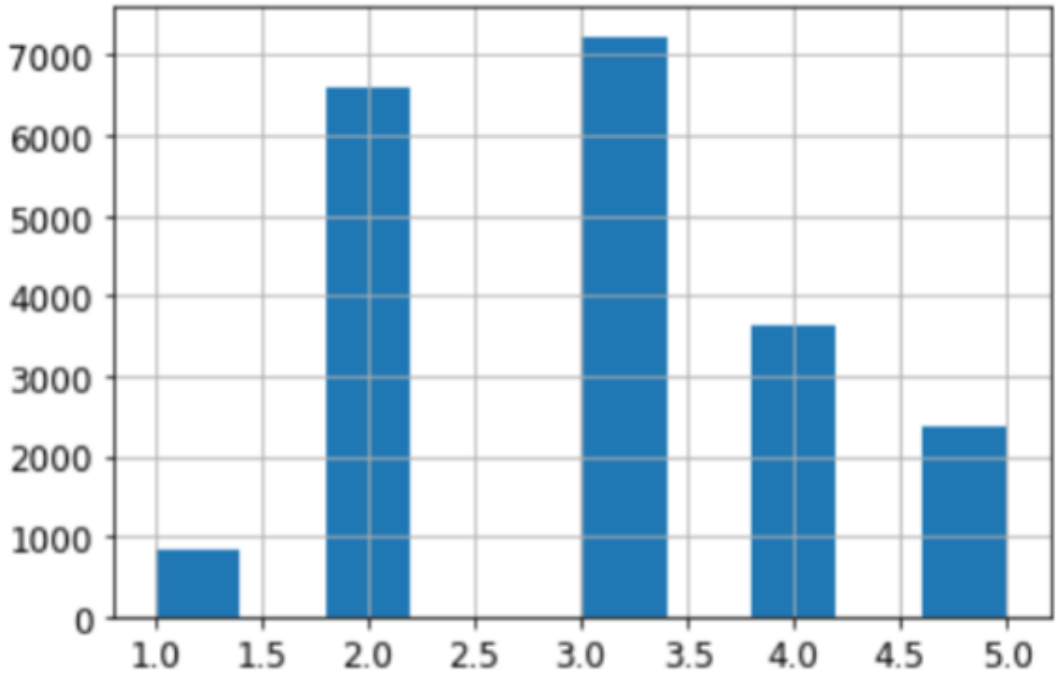
In [23]:
```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

In [24]:
```
housing["income_cat"].value_counts()
```

Out[24]:

In [25]:
```
housing["income_cat"].hist()
```

Out[25]:



and TensorFlow:

3 Discover and Visualize the Data to Gain Insights

Create a Test Set

First, make sure you have put the test set aside and you are only exploring the training set.

If the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small, so you can just work directly on the full set.

- Let's create a copy so that you can play with it without harming the training set:

```
housing = strat_train_set.copy()
```

Visualizing Geographical Data

- Since there is geographical information (latitude and longitude), it is a good idea to create a scatterplot of all districts to visualize the data (Figure 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

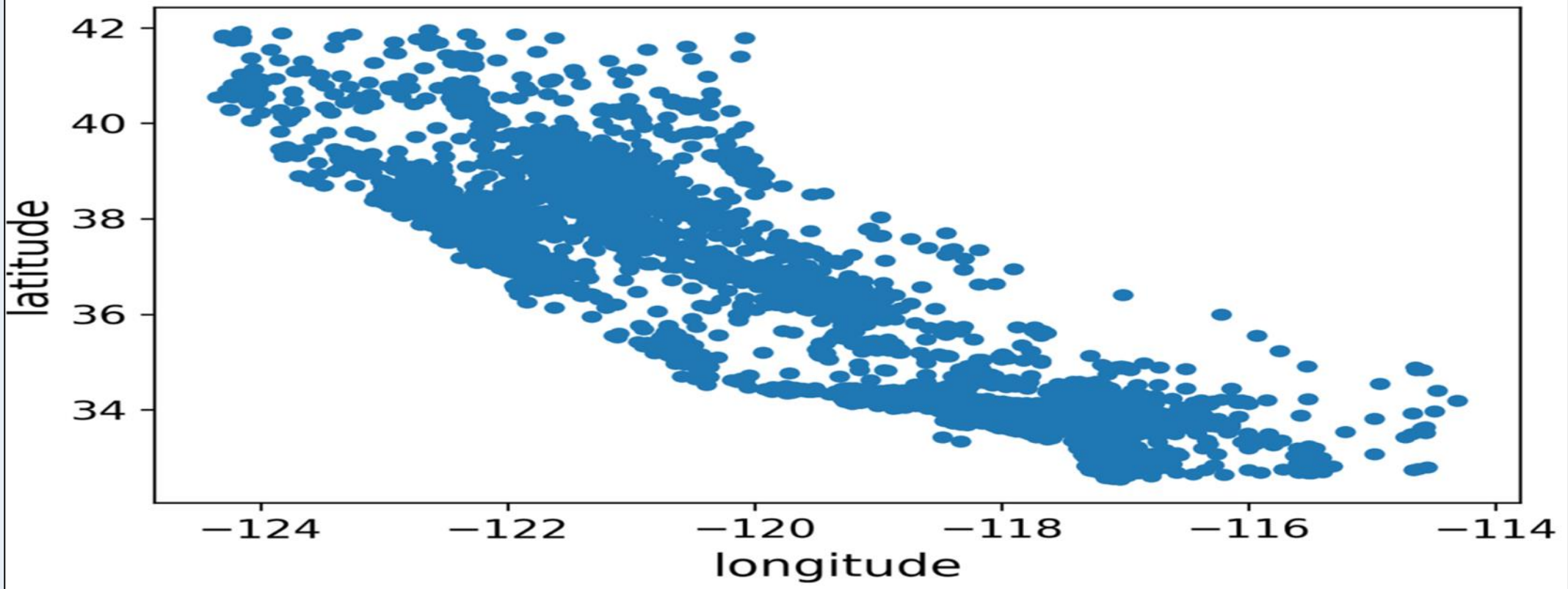


Figure 2-11. A geographical scatterplot of the data

Visualizing Geographical Data

- This looks like California all right, but other than that it is hard to see any particular pattern. Setting the alpha option to 0.1 makes it much easier to visualize **the places where there is a high density of data points** (Figure 2-12):

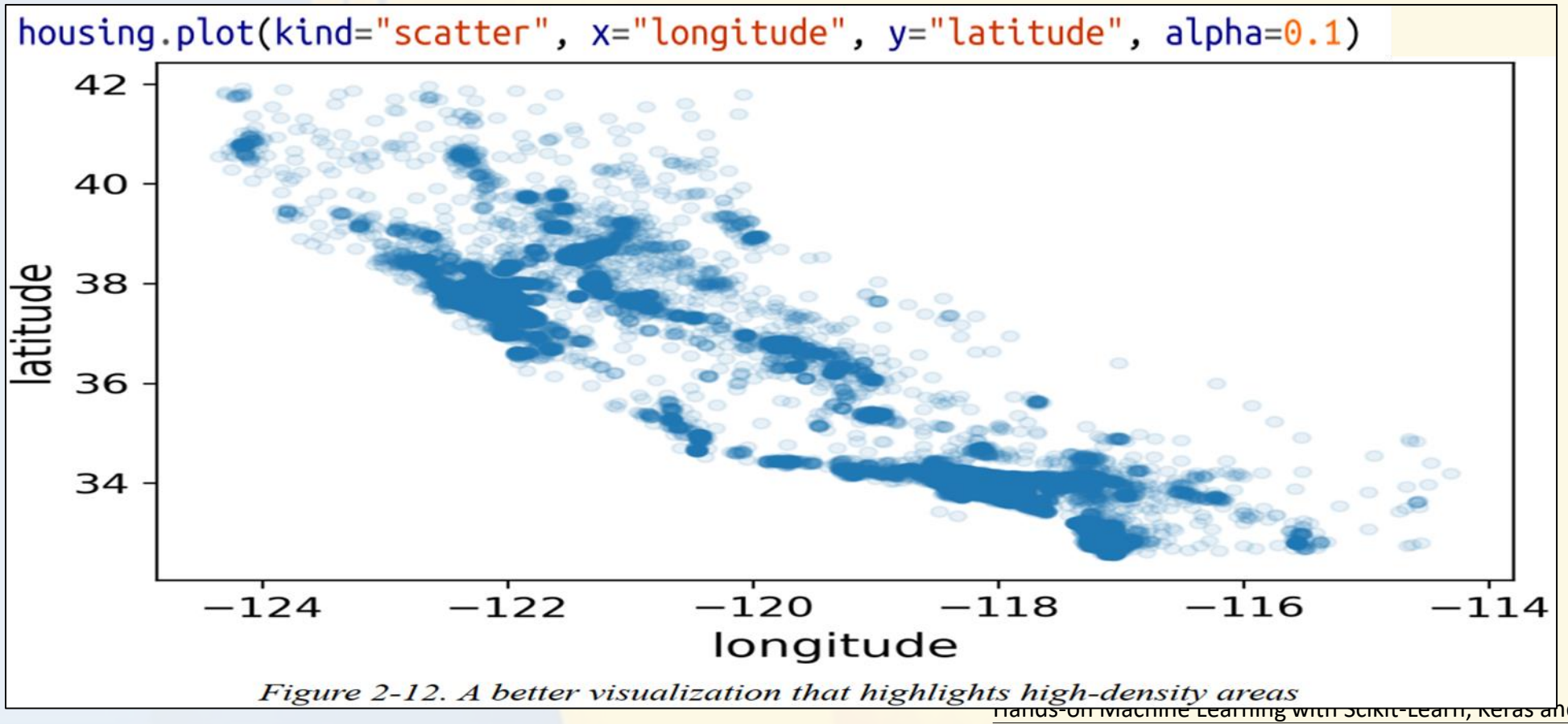


Figure 2-12. A better visualization that highlights high-density areas

Visualizing Geographical Data

- Now that's much better: you can clearly see the high-density areas, namely the Bay Area and around Los Angeles and San Diego, plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno.
- Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.

Visualizing Geographical Data

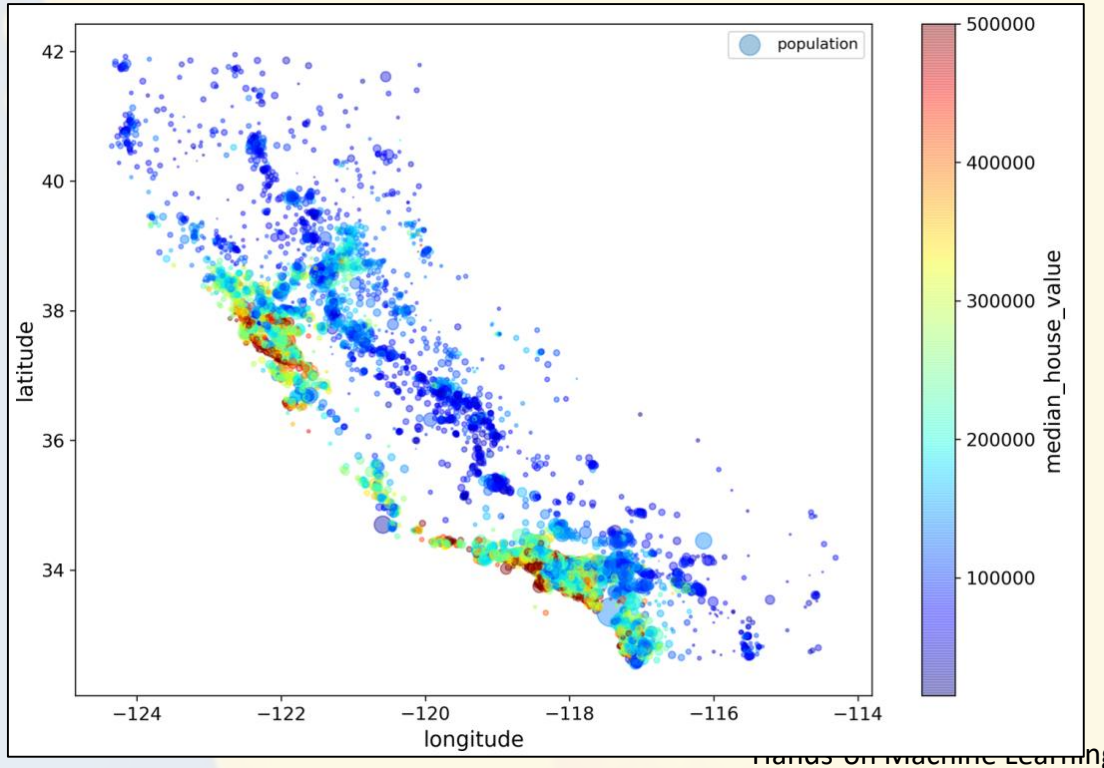
- 房屋價格（圖2-13）。 每個圓圈的半徑代表地區的人口（選項s），顏色代表價格（選項c）。
- 使用一種稱為jet的預定義顏色圖（選項cmap），範圍從藍色（低值）到紅色（高價）

```
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=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

Visualizing Geographical Data

- 這張圖片顯示房價與地理位置（例如，靠近海洋）和人口密度密切相關

- 這樣的情況可能顯示**分群演算法**對於檢測主聚類和添加新功能以測量與聚類中心的接近度有效，而海洋鄰近屬性也可能有用，儘管在北加利福尼亞州沿海地區的房價不是太高，因此背後規則仍有待分析。



Looking for Correlations

- Since the dataset is not too large, you can easily compute the standard correlation coefficient (also called Pearson's r) between every pair of attributes using the corr() method:

```
corr_matrix = housing.corr()
```

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude            -0.047279
latitude             -0.142826
Name: median_house_value, dtype: float64
```

Looking for Correlations

- The correlation coefficient ranges from –1 to 1.
- When it is close to 1, it means that there is a strong positive correlation

The median house value tends to go up when the median income goes up.

- When the coefficient is close to –1, it means that there is a strong negative correlation

You can see a small negative correlation between the latitude and the median house value

Looking for Correlations

- various plots along with the correlation coefficient between their horizontal and vertical axes.

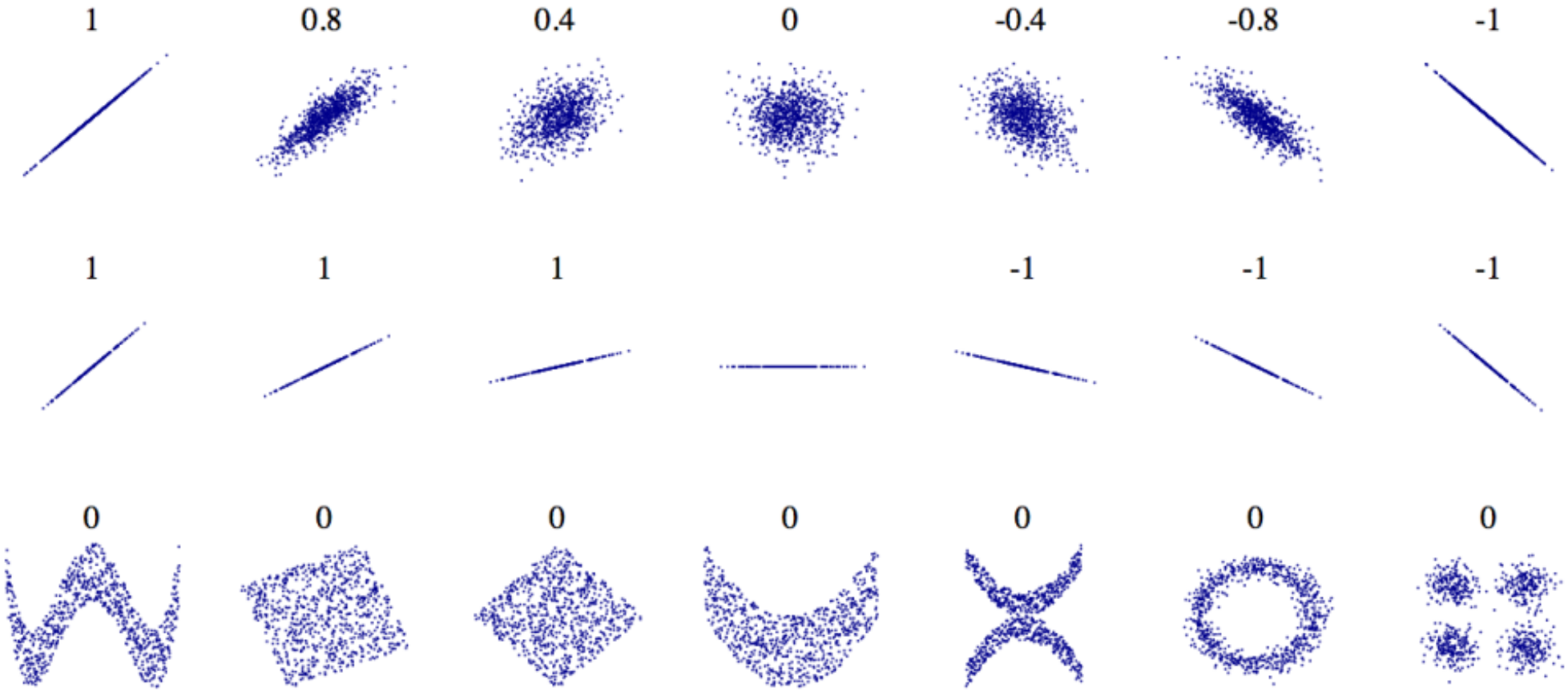


Figure 2-14. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

Looking for Correlations

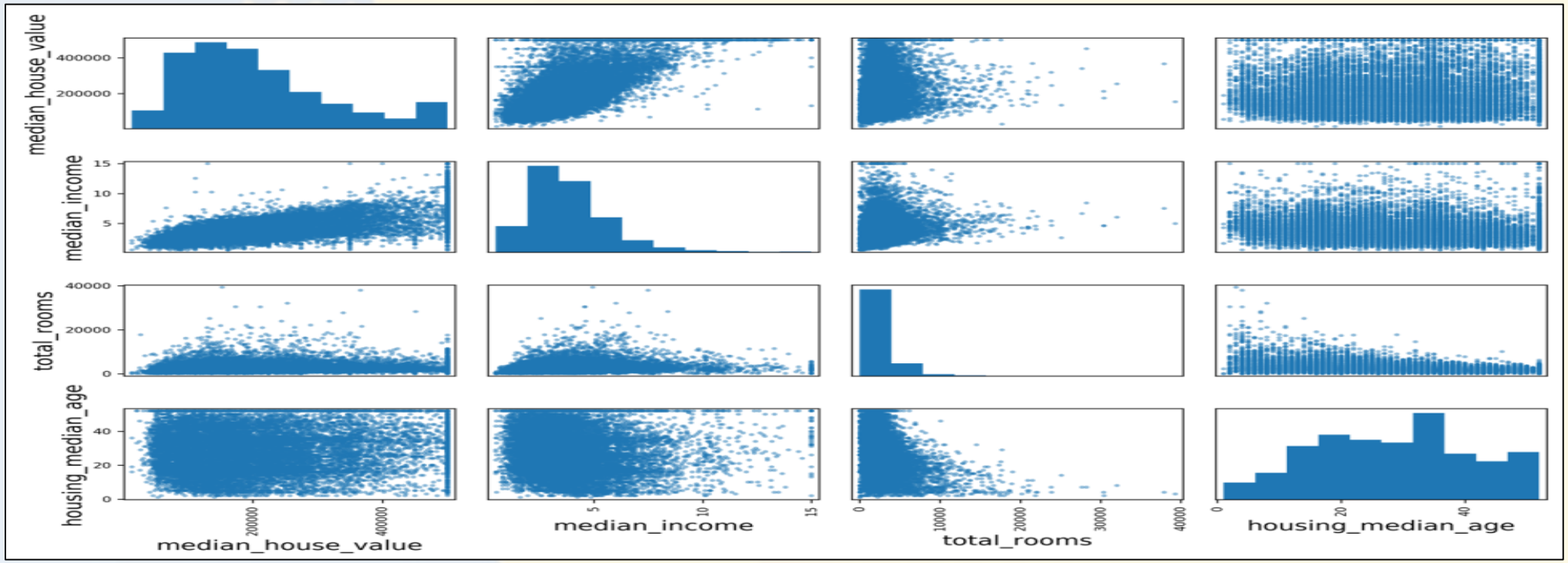
- Another way to check for correlation between attributes is to use the pandas

```
from pandas.plotting import scatter_matrix

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

Looking for Correlations

- The main diagonal (top left to bottom right) would be full of straight lines

- Pandas plotted each variable against itself, which would not be very useful. So instead pandas displays a histogram of each attribute (other options are available; see the pandas documentation for more details).

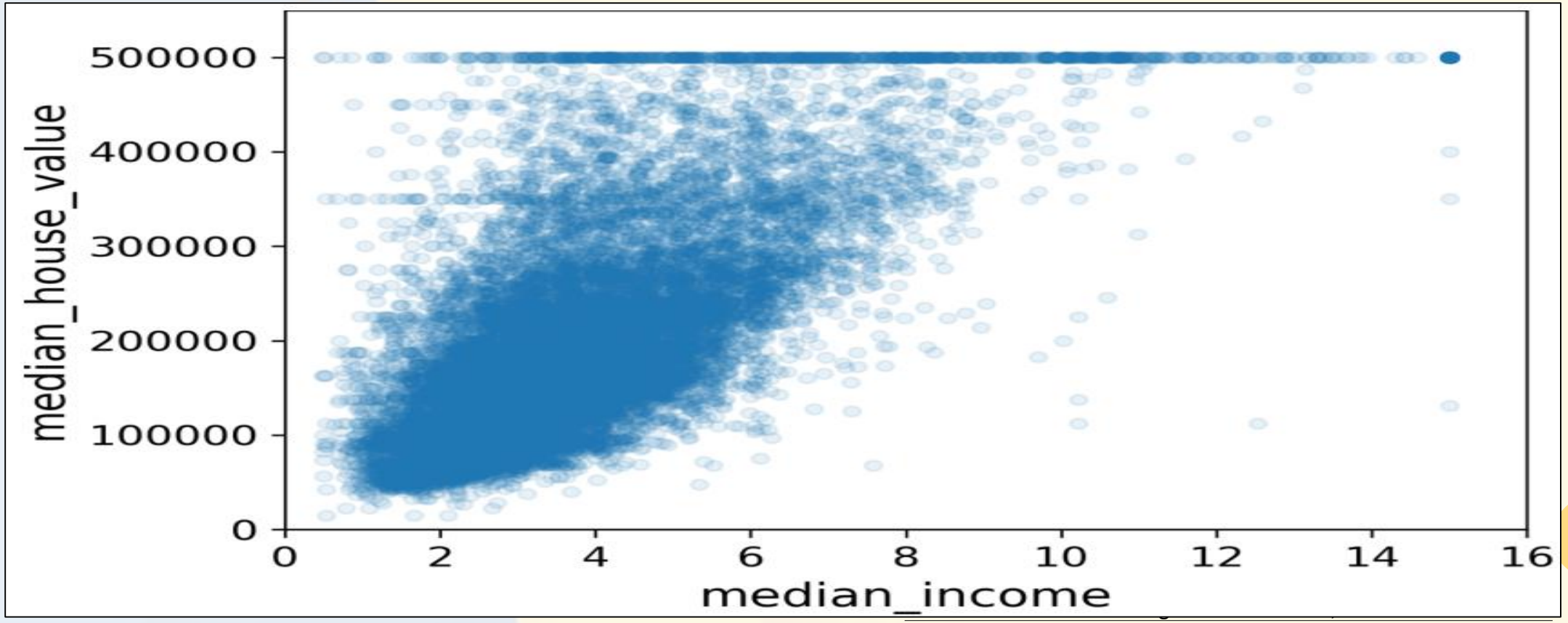


Looking for Correlations

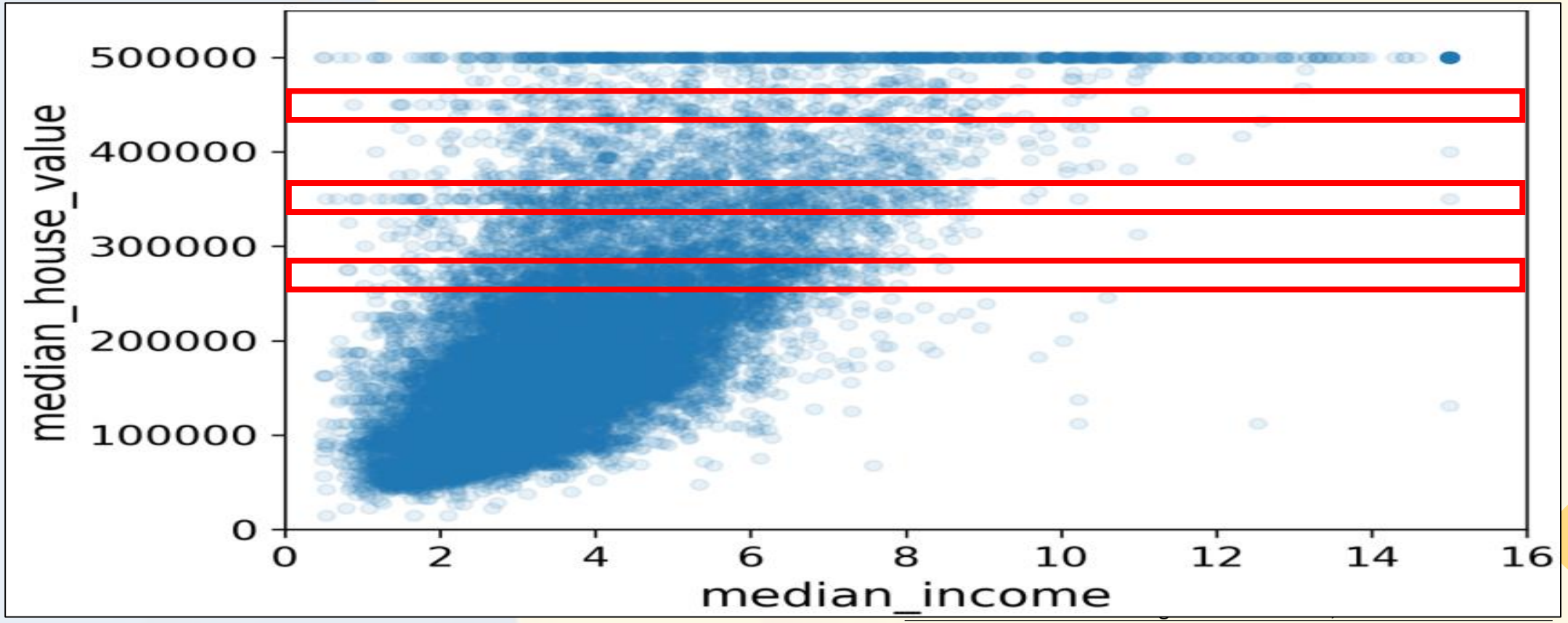
- This plot reveals a few things. First, the correlation is indeed very strong; you can clearly see the upward trend, and the points are not too dispersed. Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000. But this plot reveals other less obvious straight lines: a horizontal line around $450,000, another around $350,000, perhaps one around $280,000, and a few more below that. You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1)
```

- First, the correlation is indeed very strong; you can clearly see the upward trend, and the points are not too dispersed.
- Second, the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000.



- But this plot reveals other less obvious straight lines: a horizontal line around $450,000, $350,000, $280,000, …etc.
- You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.



Experimenting with Attribute Combinations

- You identified a few data quirks that you may want to clean up before feeding the data to a Machine Learning algorithm.
- One last thing you may want to do before preparing the data for Machin Learning algorithms is to try out various attribute combinations.
- For example, what you really want is the number of rooms per household

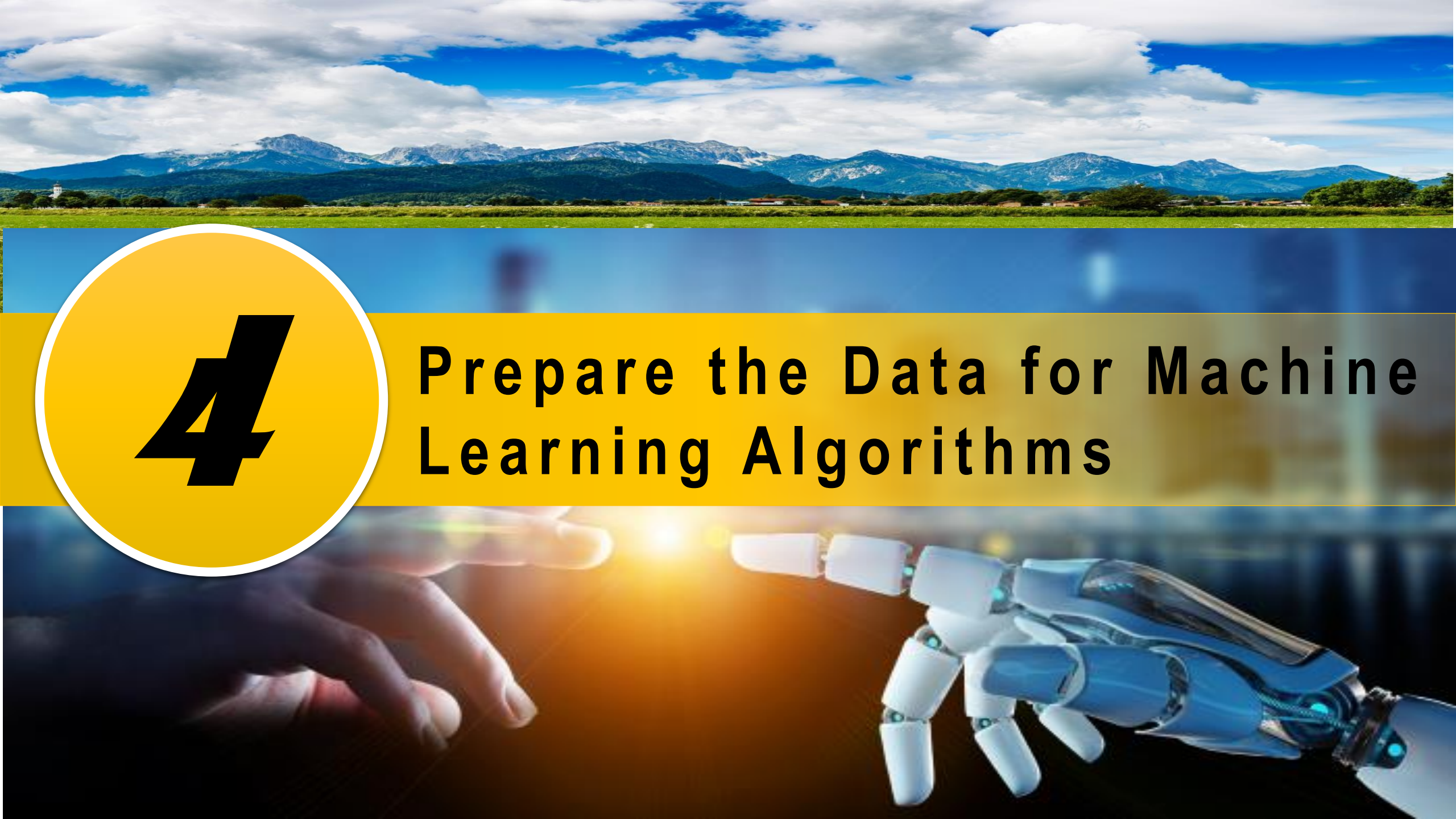
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

Experimenting with Attribute Combinations

- And now let's look at the correlation matrix again:

- The new bedrooms_per_room attribute is much more correlated with the median house value than the total number of rooms or bedrooms.

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```



4 Prepare the Data for Machine Learning Algorithms

Prepare the data for your Machine Learning algorithms.

You should write functions for this purpose, for several good reasons:

- This will allow you to reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset).
- You will gradually build a library of transformation functions that you can reuse in future projects.
- You can use these functions in your live system to transform the new data before feeding it to your algorithms.
- This will make it possible for you to easily try various transformations and see which combination of transformations works best.

Revert to a clean training set


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

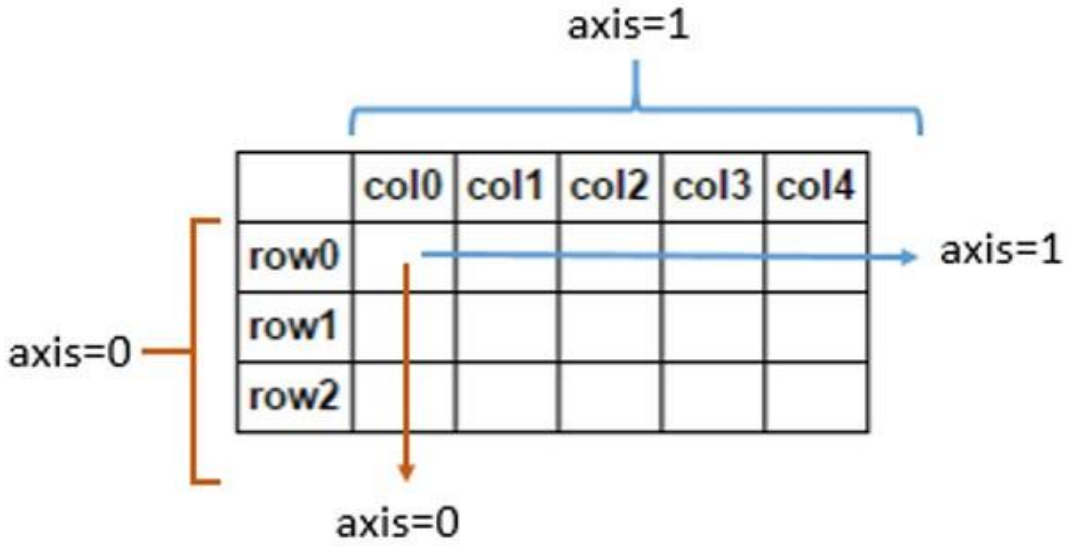
Data Cleaning

- Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them.
- We saw earlier that the total_bedrooms attribute has some missing values, so let's fix this. 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.).

Data Cleaning

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




```
housing.dropna(subset=["total_bedrooms"])    # option 1
housing.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()  # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Simple Imputer

- Scikit-Learn provides a handy class to take care of missing values:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Simple Imputer

- Since the median can only be computed on numerical attributes, you need to create a copy of the data without the text attribute ocean_proximity:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

- Now you can fit the imputer instance to the training data using the fit() method:

```
imputer.fit(housing_num)
```

Simple Imputer

- The imputer has simply computed the median of each attribute and stored the result in its statistics_ instance variable.
- Only the total_bedrooms attribute had missing values
- But we cannot be sure that there won't be any missing values in new data after the system goes live,
- It is safer to apply the imputer to all the numerical attributes

```
>>> imputer.statistics_
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
>>> housing_num.median().values
array([ -118.51 , 34.26 , 29. , 2119.5 , 433. , 1164. , 408. , 3.5409])
```

Simple Imputer

- Now you can use this "trained" imputer to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

- The result is a plain NumPy array containing the transformed features. If you want to put it back into a pandas DataFrame, it's simple:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

Handling Text and Categorical Attributes

- So far we have only dealt with numerical attributes, but now let's look at text attributes

- In this dataset, there is just one: the ocean_proximity attribute. Let's look at its value for the first 10 instances:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
       ocean_proximity
17606        <1H OCEAN
18632        <1H OCEAN
14650       NEAR OCEAN
3230            INLAND
3555         <1H OCEAN
19480           INLAND
8879         <1H OCEAN
13685           INLAND
4937         <1H OCEAN
4861         <1H OCEAN
```

Handling Text and Categorical Attributes

- It’s not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute.
- Most Machine Learning algorithms prefer to work with numbers, so let’s convert these categories from text to numbers.
- Use Scikit-Learn’s OrdinalEncoder class

Handling Text and Categorical Attributes

- Use Scikit-Learn's OrdinalEncoder class

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Handling Text and Categorical Attributes

- You can get the list of categories using the categories_ instance variable.
- It is a list containing a 1D array of categories for each categorical attribute

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Handling Text and Categorical Attributes

- One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values.
- This may be fine in some cases (e.g., for ordered categories such as "bad," "average," "good," and "excellent"), but it is obviously not the case for the ocean_proximity column

(for example, categories 0 and 4 are clearly more similar than categories 0 and 1).

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Fix this issue

- A common solution is to create one binary attribute per category:
- One attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise), another attribute equal to 1 when the category is "INLAND" (and 0 otherwise),
- This is called one-hot encoding
- Only one attribute will be equal to 1 (hot), while the others will be 0 (cold).
- The new attributes are sometimes called dummy attributes.

One Hot Encoding

- Scikit-Learn provides a OneHotEncoder class to convert categorical values into one-hot vector

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
```

One Hot Encoding

- Notice that the output is a SciPy sparse matrix, instead of a NumPy array.
- This is very useful when you have categorical attributes with thousands of categories
- Using up tons of memory mostly to store zeros would be very wasteful
- so instead a sparse matrix only stores the location of the nonzero elements

One Hot Encoding

- You can use it mostly like a normal 2D array, but if you really want to convert it to a (dense)
- NumPy array, just call the toarray() method:

```
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

One Hot Encoding

- Once again, you can get the list of categories using the encoder's categories_ instance variable:

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Custom Transformers

- Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]

        else:
            return np.c_[X, rooms_per_household, population_per_household]
```

Custom Transformers

- Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes

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

Custom Transformers

- add_bedrooms_per_room, set to True by default (it is often helpful to provide sensible defaults).
- This hyperparameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not.
- More generally, you can add a hyperparameter to gate any data preparation step that you are not 100% sure about.

Feature Scaling

- One of the most important transformations is feature scaling.
- Machine Learning algorithms don't perform well when the input numerical attributes have very different scales.
- The housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15.
- Scaling the target values is generally not required.

Feature Scaling

There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

- Min-max scaling (many people call this normalization) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1.

We do this by subtracting the min value and dividing by the max minus the min.

- Scikit-Learn provides a transformer called MinMaxScaler for this.

It has a feature_range hyperparameter that lets you change the range if, for some reason, you don't want 0–1.

Feature Scaling

- Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance

- Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1).

Feature Scaling

- standardization is much less affected by outliers.
- For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected.
- Scikit-Learn provides a transformer called StandardScaler for standardization.

Feature Scaling

Transformation Pipelines

- As you can see, there are many data transformation steps that need to be executed in the right order.
- Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations. Here is a small pipeline for the numerical attributes:

Feature Scaling

- The Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps.

- All but the last estimator must be transformers (i.e., they must have a fit_transform() method). The names can be anything you like (as long as they are unique and don't contain double underscores, __); they will come in handy later for hyperparameter tuning.

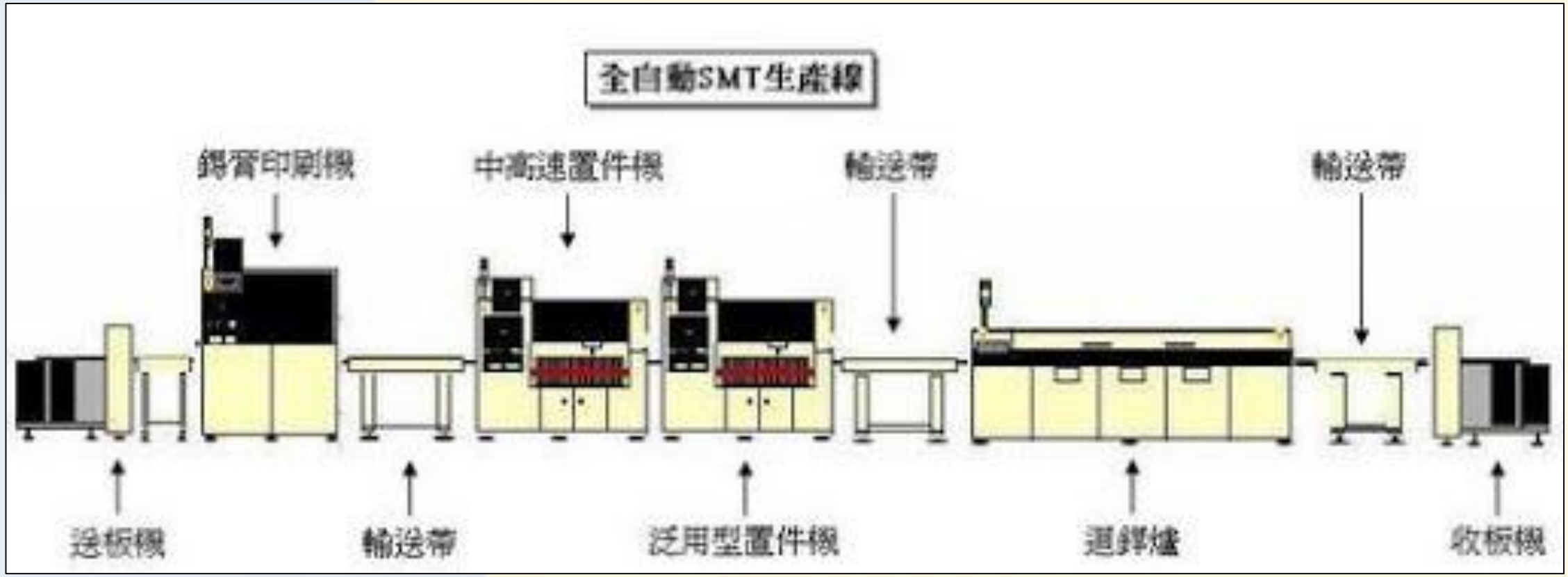
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Feature Scaling

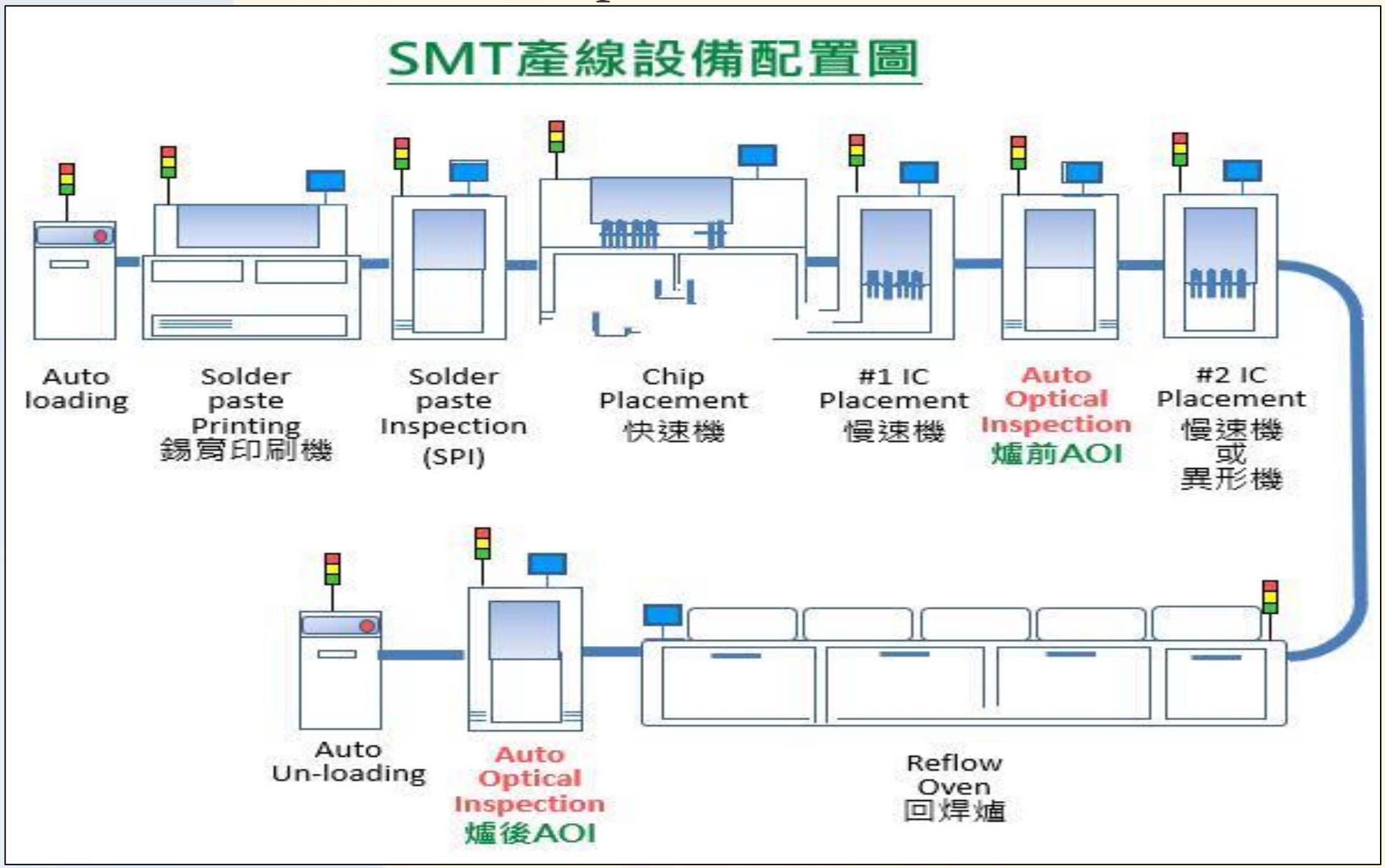
- tt would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. In version 0.20, Scikit-Learn introduced the ColumnTransformer for this purpose,

```
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),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

Pipeline



Pipeline


SMT產線設備配置圖
Auto loading
Solder paste Printing 錫膏印刷機
Solder paste Inspection (SPI)
Chip Placement 快速機
#1 IC Placement 慢速機
Auto Optical Inspection 爐前AOI
#2 IC Placement 慢速機 或 異形機
Auto Un-loading
Auto Optical Inspection 爐後AOI
Reflow Oven 回焊爐

Pipeline

5 Select a model and train it

Training and Evaluating on the Training Set

- The good news is that thanks to all these previous steps, things are now going to be much simpler than you might think. Let's first train a Linear Regression model, like we did in the previous chapter:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

- You now have a working Linear Regression model. Let's try it out on a few instances from the training set:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Training and Evaluating on the Training Set

- It works, although the predictions are not exactly accurate (e.g., the first prediction is off by close to 40%!).

```
>>> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]
>>> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

- Let's measure this regression model's RMSE on the whole training set using Scikit-Learn's mean_squared_error() function:

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.19819848922
```

Training and Evaluating on the Training Set

This is better than nothing, but clearly not a great score:

- Most districts' median_housing_values range between $120,000 and $265,000, so a typical prediction error of $68,628 is not very satisfying.

This is an example of a model underfitting the training data.

- When this happens it can mean that the features do not provide enough information to make good predictions.
- or that the model is not powerful enough.

Training and Evaluating on the Training Set

The main ways to fix underfitting are

- to select a more powerful model, to feed the training algorithm.
- to feed the training algorithm with better features
- to reduce the constraints on the model.

Let’s try a more complex model to see how it does.

- Let’s train a DecisionTreeRegressor.

Training and Evaluating on the Training Set

This is a powerful model, capable of finding complex nonlinear relationships in the data

- (Decision Trees are presented in more detail in Chapter 6).

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

- Now that the model is trained, let's evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Training and Evaluating on the Training Set

Wait, what!? No error at all? Could this model really be absolutely perfect?

- Of course, it is much more likely that the model has badly overfit the data.

How can you be sure? As we saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use

- part of the training set for training and part of it for model validation.

Better Evaluation Using Cross-Validation

One way to evaluate the Decision Tree model would be to use the

train_test_split() function to

- split the training set into a smaller training and a validation set
- train your models against the smaller training set and evaluate them against the validation set.
- It's a bit of work, but nothing too difficult, and it would work fairly well.

Better Evaluation Using Cross-Validation

A great alternative is to use Scikit-Learn's K-fold cross-validation feature

- The following code randomly splits the training set into 10 distinct subsets called folds
- Then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds.
- The result is an array containing the 10 evaluation scores:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

Better Evaluation Using Cross-Validation

WARNING

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

- Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better)
- So the scoring function is actually the opposite of the MSE (i.e., a negative value)
- It is why the preceding code computes -scores before calculating the square root.

Better Evaluation Using Cross-Validation

Now the Decision Tree doesn't look as good as it did earlier. In fact, it seems to perform worse than the Linear Regression model!

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
 71115.88230639 75585.14172901 70262.86139133 70273.6325285
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

Better Evaluation Using Cross-Validation

Notice that cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation).

- The Decision Tree has a score of approximately 71,407, generally ±2,439.
- You would not have this information if you just used one validation set.
- But cross-validation comes at the cost of training the model several times, so it is not always possible.

Better Evaluation Using Cross-Validation

Let's compute the same scores for the Linear Regression model just to be sure:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                              scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [66782.73843989 66960.118071   70347.95244419 74739.57052552
 68031.13388938 71193.84183426 64969.63056405 68281.61137997
 71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

That's right: the Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

Better Evaluation Using Cross-Validation

Let’s try one last model now: the RandomForestRegressor.

- Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions.
- Building a model on top of many other models is called Ensemble Learning, and it is often a great way to push ML algorithms even further.

Better Evaluation Using Cross-Validation


```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
18603.515021376355
>>> display_scores(forest_rmse_scores)
Scores: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

Better Evaluation Using Cross-Validation

Wow, this is much better: Random Forests look very promising.

- Note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set.

Possible solutions for overfitting are

- to simplify the model, constrain it (i.e., regularize it)
- get a lot more training data.

TIP: Save your ML model

- Save every model you experiment with so that you can come back easily to any model you want.
- Save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well.
- This will allow you to easily compare scores across model types, and compare the types of errors they make.

```
import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

TIP: Save your ML model


```
1 import  joblib
2
3 joblib.dump(forest_reg,  "forest_reg.pkl")
4 #  and  later...
5 my_model_loaded  =  joblib.load("forest_reg.pkl")
```

```
1 !ls
```

```
data  forest_reg.pkl  sample_data
```

```
1 my_model_loaded
```

```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```


6
Fine-tune your model

Grid Search

One option would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values.

- This would be very tedious work, and you may not have time to explore many combinations.
- You should get Scikit-Learn's GridSearchCV to search for you.
- Tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values.

Grid Search

For example, the following code searches for the best combination of hyperparameter values for the RandomForestRegressor:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

TIP: 找超參數的初始值設定

When you have no idea what value a hyperparameter should have, a simple approach is:

- To try out consecutive powers of 10.
- To set a smaller number if you want a more fine-grained search, as shown in this example with the n_estimators hyperparameter.

Grid Search

This param_grid tells Scikit-Learn to first evaluate all 3 × 4 = 12 combinations of n_estimators and max_features hyperparameter values specified in the first

- Then try all 2 × 3 = 6 combinations of hyperparameter values in the second dict, but this time with the bootstrap hyperparameter set to False

- The grid search will explore 12 + 6 = 18 combinations of RandomForestRegressor hyperparameter values, and it will train each model 5 times (since we are using five-fold cross validation).

- There will be 18 × 5 = 90 rounds of training! It may take quite a long time, but when it is done you can get the best combination of parameters like this:

Grid Search


```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```

You can also get the best estimator directly:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=None, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

Grid Search

And of course the evaluation scores are also available:

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Randomized Search

- When the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead.
- This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible combinations
- If you let the randomized search run for, say, 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter

Analyze the Best Models and Their Errors

- You will often gain good insights on the problem by inspecting the best models.
- For example, the RandomForestRegressor can indicate of each attribute for making accurate predictions:

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```

Analyze the Best Models and Their Errors

- Let's display these importance scores next to their corresponding attribute names:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = full_pipeline.named_transformers_["cat"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.3661589806181342, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.07334423551601242, 'longitude'),
 (0.0629090704826203, 'latitude'),
 (0.05641917918195401, 'rooms_per_hhold'),
 (0.05335107734767581, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.01487428089040276 7, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.00285647462732015 8, 'NEAR OCEAN')]
```

Let's display these importance

- Let's display these importance scores next to their corresponding attribute names:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = full_pipeline.named_transformers_["cat"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.3661589806181342, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.1087929567755157 3, 'pop_per_hhold'),
 (0.07334423551601242, 'longitude'),
 (0.062909070482620 3, 'latitude'),
 (0.0564191791819 5401, 'rooms_per_hhold'),
 (0.0533510773476 7581, 'bedrooms_per_room'),
 (0.0411437984787 29635, 'housing_median_age'),
 (0.0148742808904 02767, 'population'),
 (0.0146726854205 43237, 'total_rooms'),
 (0.0142575993234 07807, 'households'),
 (0.0141064834535 84102, 'total_bedrooms'),
 (0.0103114883263 03787, '<1H OCEAN'),
 (0.0028564746373 20158, 'NEAR OCEAN'),
 (0.0019604155994 7807, 'NEAR BAY'),
 (6.028038672736599e-05, 'ISLAND')]
```

Evaluate Your System on the Test Set

- Just get the predictors and the labels from your test set, run your full_pipeline to transform the data (call transform(), not fit_transform()—you do not want to fit the test set!), and evaluate the final model on the test set:

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,730.2
```

Compute a 95% Confidence Interval

- Youmight want to have an idea of how precise this estimate is. For this, you can compute a 95% confidence interval for the generalization error

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                          loc=squared_errors.mean(),
...                          scale=stats.sem(squared_errors)))
...
array([45685.10470776, 49691.25001878])
```


7
Present your solution