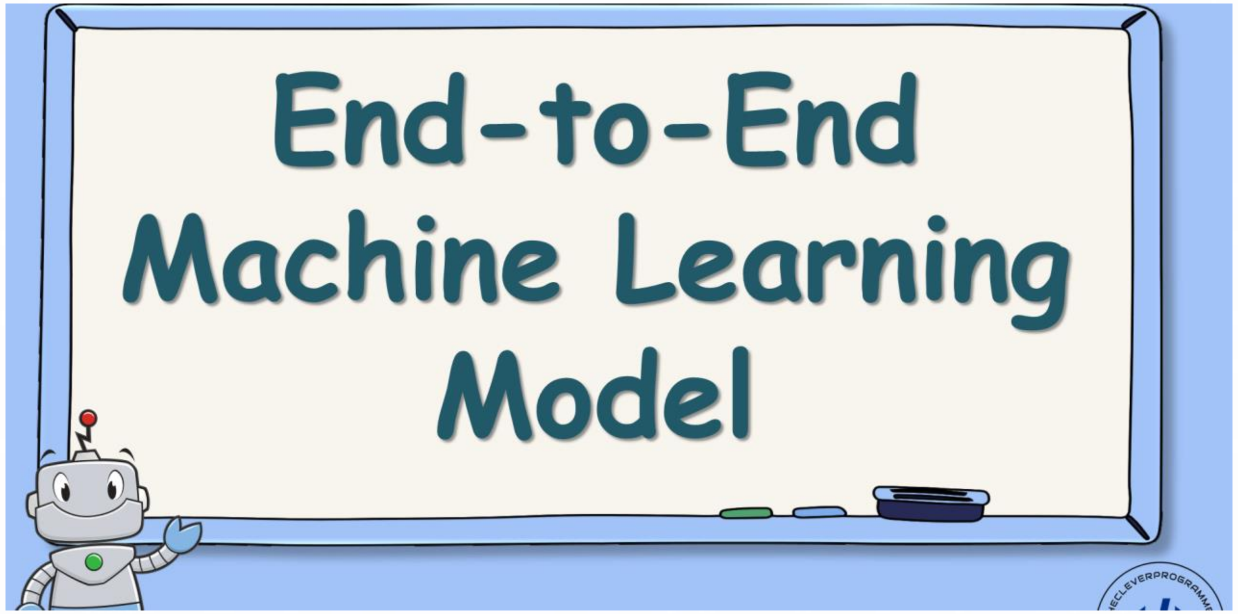
ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

دانشگاه شاهد

AI in Biomedical Data

Dr. M.B. Khodabakhshi
Amir Hossein Fouladi
Alireza Javadi

github.com/mbkhodabakhshi/AI_in_BiomedicalData



یادگیری ماشین در زیست پزشکی

Chapter 2. End-to-End Machine Learning Project

دکتر محمدباقر خدابخشی
mb.khodabakhshi@gmail.com

What we have in this chapter

The goal is to illustrate the main steps of a machine learning project, not to learn anything about the real-world example. Here are the main steps we will walk through:

1. Look at the big picture.
2. Get the data.
3. Explore 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

Working with Real Data

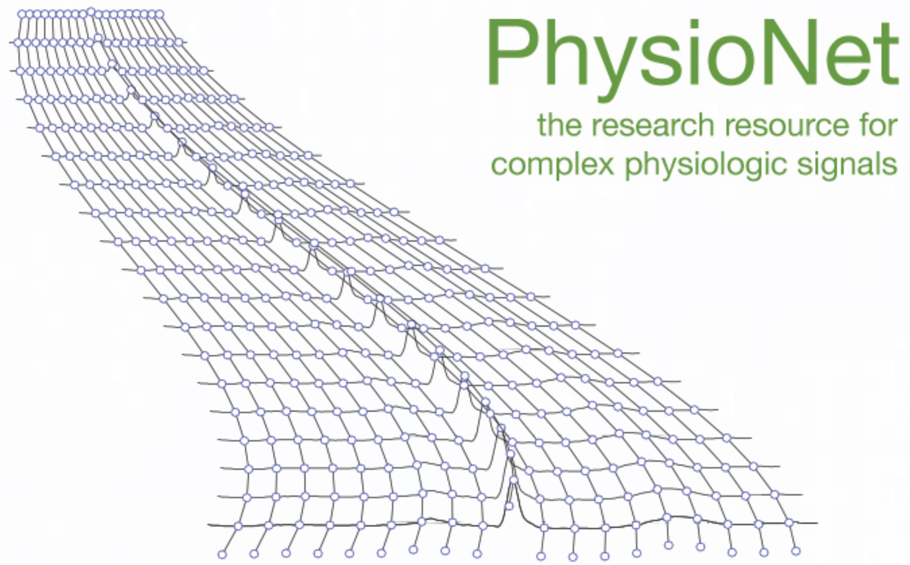
OpenML

Find Open Datasets and Machine Learning Projects | Kaggle

UCI Machine Learning Repository

Machine Learning Datasets | Papers With Code

PhysioNet



Look at the big picture: A practical ML example

A machine learning pipeline for real estate investments

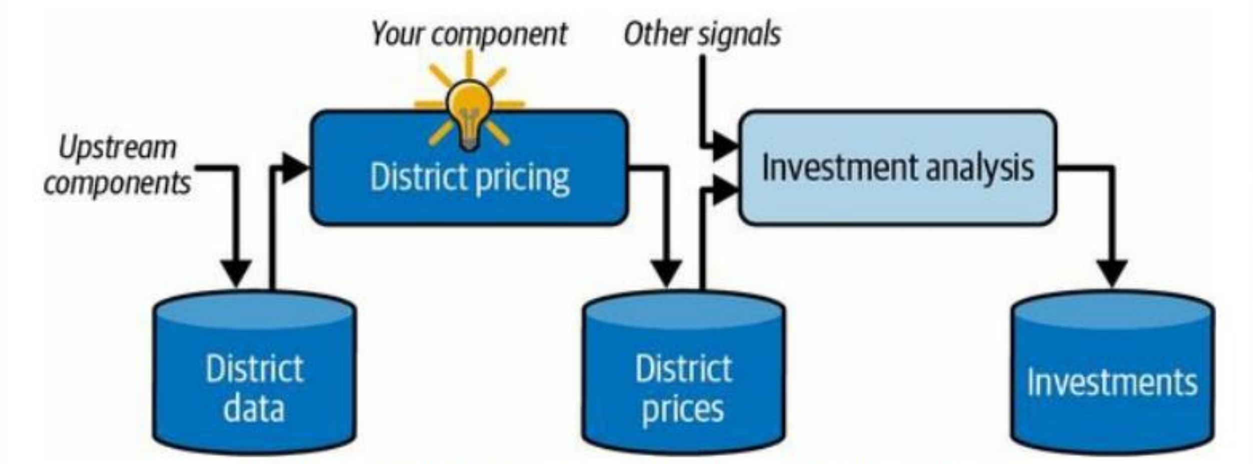


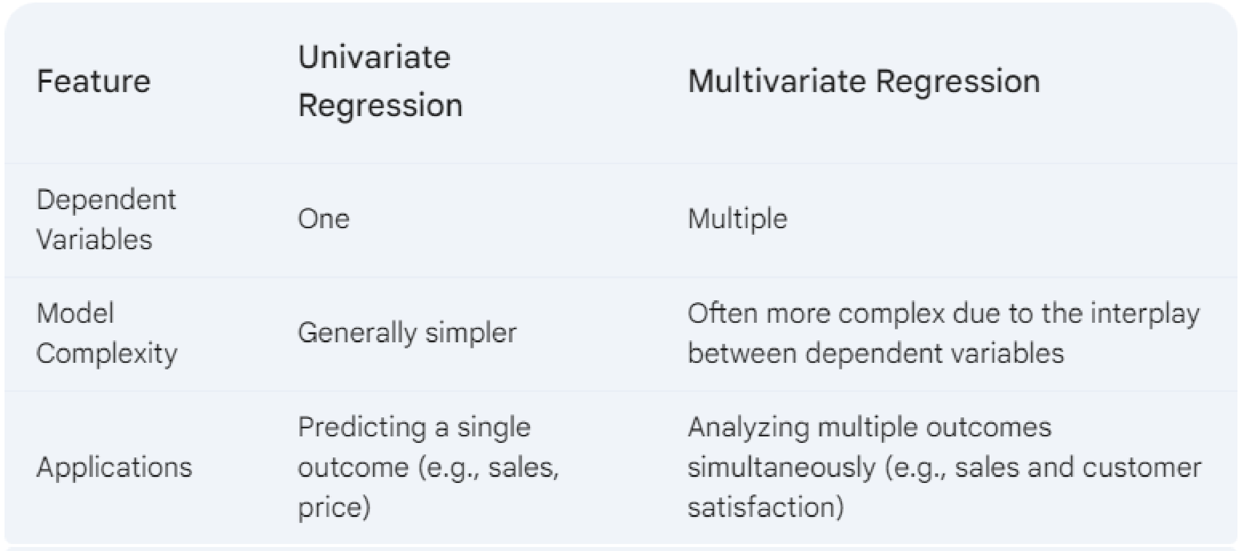
Figure 2-2. A machine learning pipeline for real estate investments

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 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.

First, **determine what kind of training supervision the model will need**: is it a supervised, unsupervised, semi-supervised, self-supervised, or reinforcement learning task?

Is it a classification task, a regression task, or something else?
Should you use batch learning or online learning techniques?

This is clearly **a typical supervised learning task**, since the model can be trained with labeled examples (each instance comes with the expected output, i.e., the district's median housing price).
It is **a typical regression task**, since the model will be asked to predict a value.
More specifically, this is a **multiple regression problem**, since the system will use multiple features to make a prediction (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**.



Feature	Univariate Regression	Multivariate Regression
Dependent Variables	One	Multiple
Model Complexity	Generally simpler	Often more complex due to the interplay between dependent variables
Applications	Predicting a single outcome (e.g., sales, price)	Analyzing multiple outcomes simultaneously (e.g., sales and customer satisfaction)

Select a Performance Measure

A typical performance measure for regression problems is the *root mean square error* (**RMSE**).

RMSE(X,h) is the cost function measured on the set of examples using your hypothesis *h*.

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

- m is the number of instances in the dataset you are measuring the RMSE on.
  - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2{,}000$.

- $\mathbf{x}^{(i)}$ is a vector of all the feature values (excluding the label) of the i^{th} instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance).
  - For example, if the first district in the dataset is located at longitude –118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of $38,372, and the median house value is $156,400 (ignoring the other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1{,}416 \\ 38{,}372 \end{pmatrix} \qquad y^{(1)} = 156{,}400$$

- **X** is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the i^{th} row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^\intercal$.[4]
  - For example, if the first district is as just described, then the matrix **X** looks like this:

$$\mathbf{X} = \begin{pmatrix} \left(\mathbf{x}^{(1)}\right)^\intercal \\ \left(\mathbf{x}^{(2)}\right)^\intercal \\ \vdots \\ \left(\mathbf{x}^{(1999)}\right)^\intercal \\ \left(\mathbf{x}^{(2000)}\right)^\intercal \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1{,}416 & 38{,}372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ for that instance ($\hat{y}$ is pronounced "y-hat").

— For example, if your system predicts that the median housing price in the first district is $158,400, then $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158{,}400$. The prediction error for this district is $\hat{y}^{(1)} - y^{(1)} = 2{,}000$.

Even though the RMSE is generally the preferred performance measure for regression tasks, 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*)

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

- ❖ Computing the root of a sum of squares (RMSE) corresponds to the *Euclidean norm*: this is the notion of distance you are familiar with. It is also called the ℓ_2 *norm*, noted $\|\cdot\|_2$(or just $\|\cdot\|$).

- ❖ Computing the sum of absolutes (MAE) corresponds to the ℓ_1 *norm*, noted $\|\cdot\|_1$. This is sometimes called the *Manhattan norm* because it measures the distance between two points in a city if you can only travel along orthogonal city blocks.

- ❖ More generally, the ℓ_k *norm* of a vector *v* containing *n* elements is defined as $\| v \|_k = (|v_0|^k + |v_1|^k + \ldots + |v_n|^k)^{\frac{1}{k}}$. ℓ_0 gives the number of nonzero elements in the vector, and ℓ_∞ gives the maximum absolute value in the vector.

The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE.

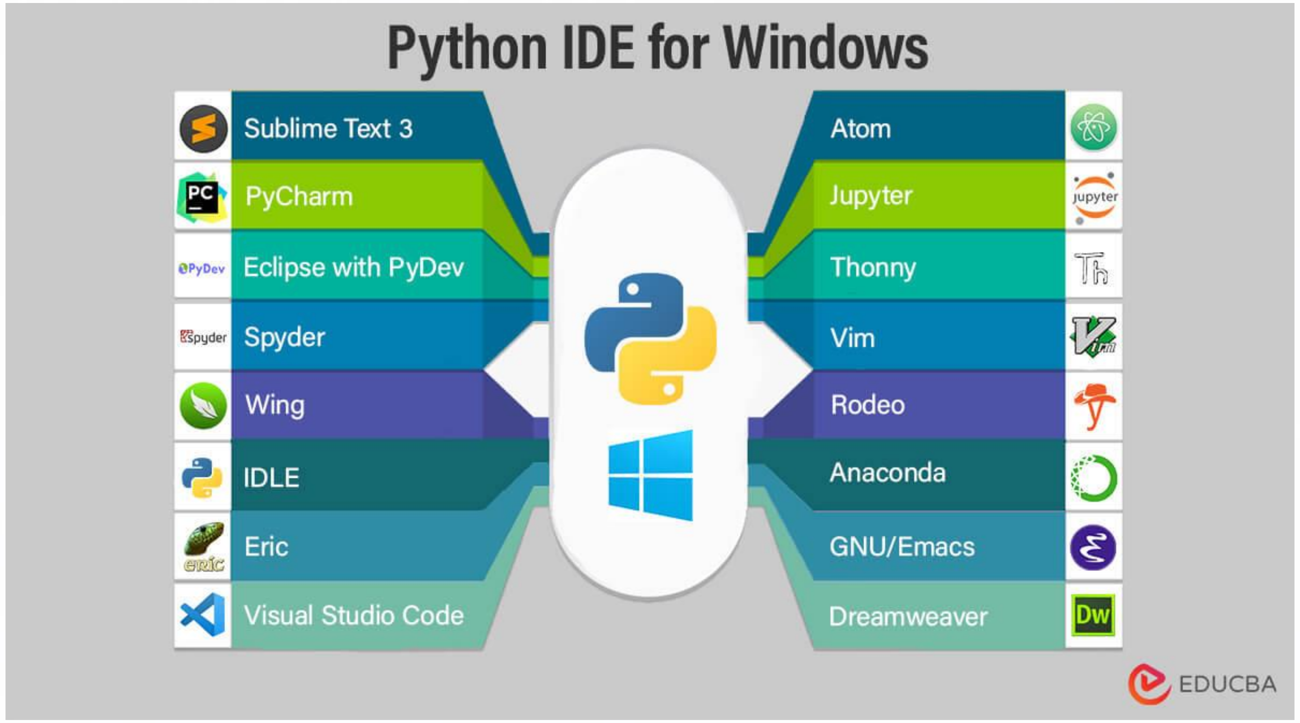
But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

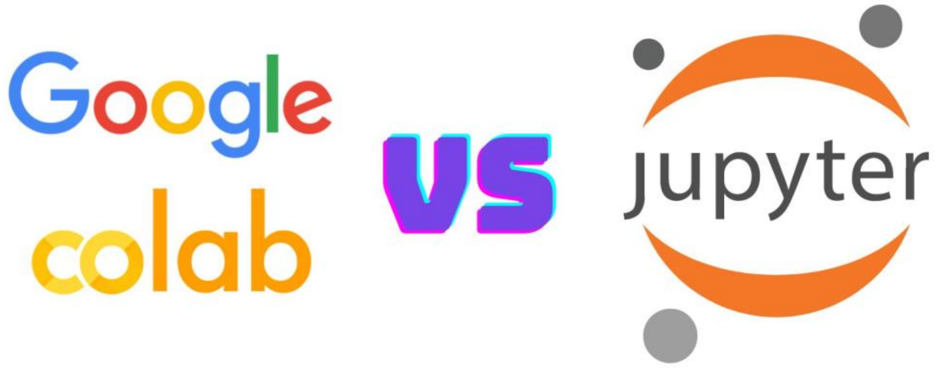
Think more about ℓ_0 *norm*.

When is it applicable?

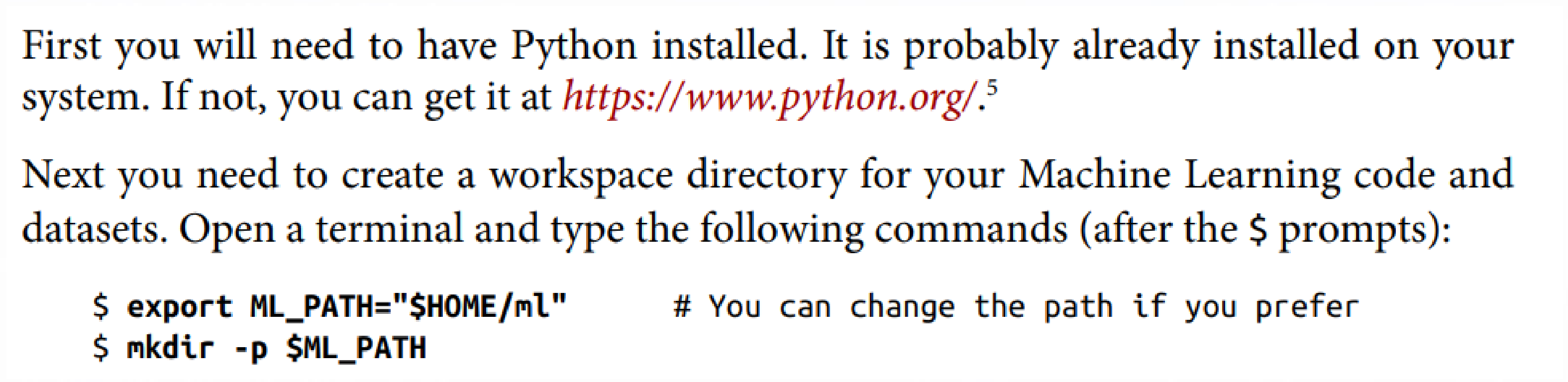
Get the Data

It's time to get your hands dirty. Don't hesitate to pick up your laptop and walk through the following code examples in a **Jupyter notebook**. The full Jupyter note- book is available a: *https://github.com/ageron/handson-ml2*





Create the Workspace



First you will need to have Python installed. It is probably already installed on your system. If not, you can get it at *https://www.python.org/*.[5]

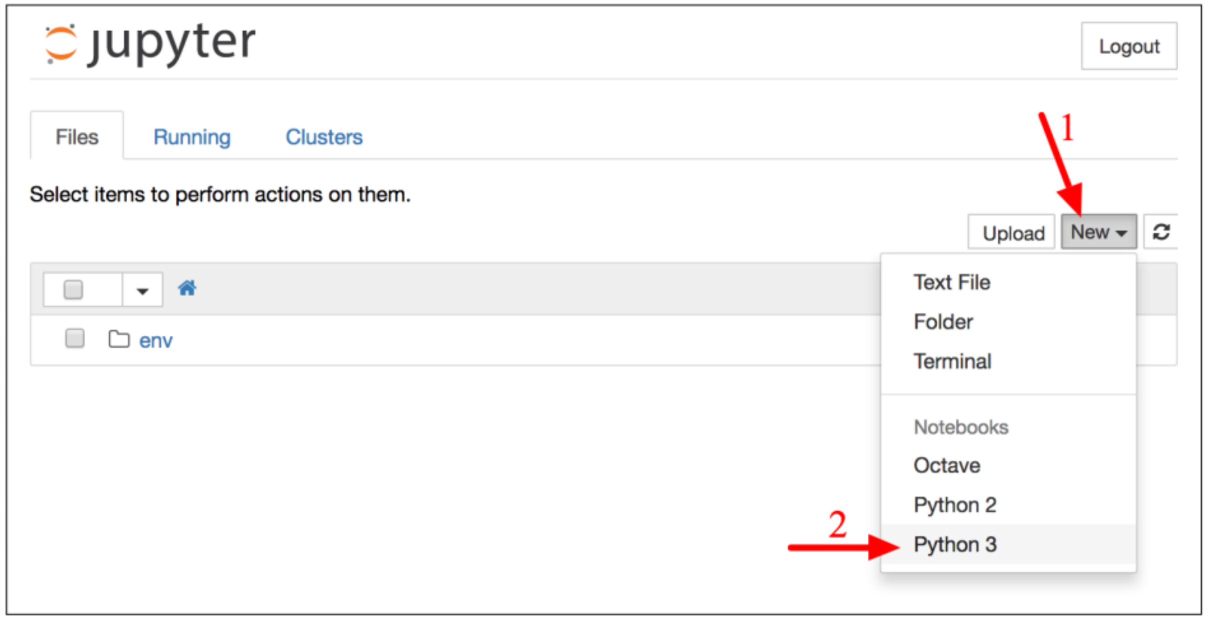
Next you need to create a workspace directory for your Machine Learning code and datasets. Open a terminal and type the following commands (after the $ prompts):

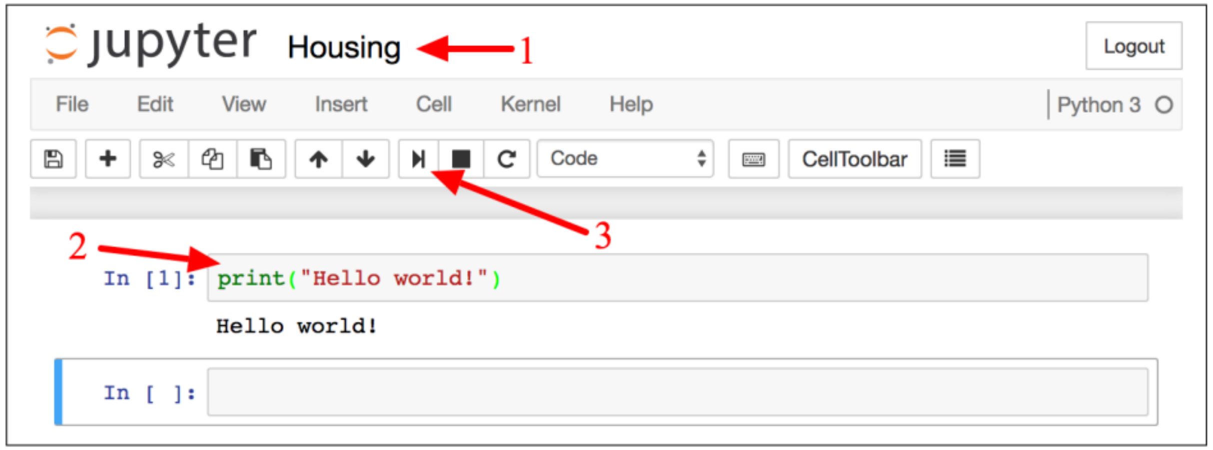
```
$ export ML_PATH="$HOME/ml"      # You can change the path if you prefer
$ mkdir -p $ML_PATH
```

Please read *pages*
42 – 45 of the reference book for detailed information about
installing python and making isolated environments

ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

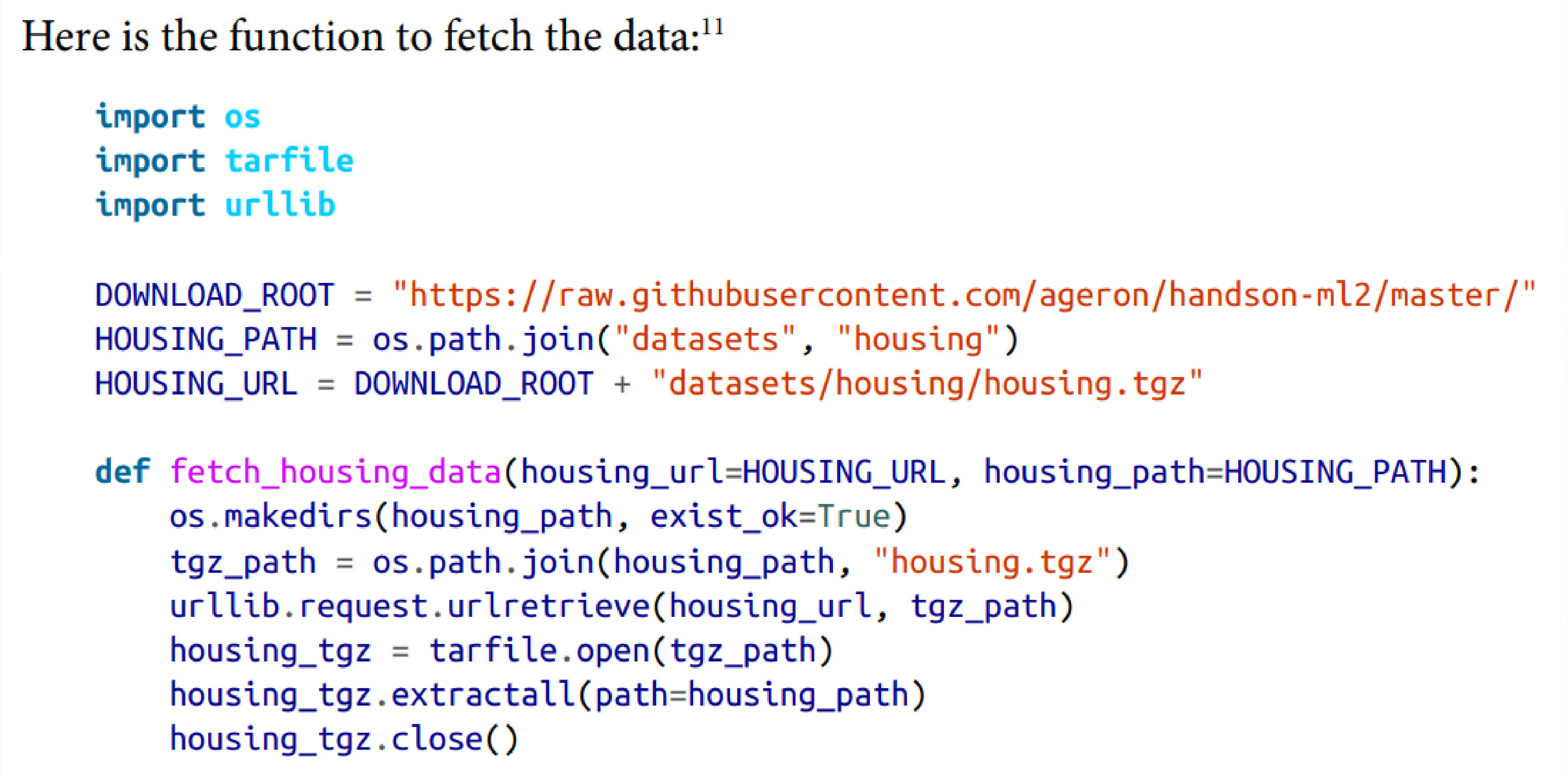
Jupyter notebook





```
In [1]: print("Hello world!")
Hello world!
In [ ]:
```

Download the Data



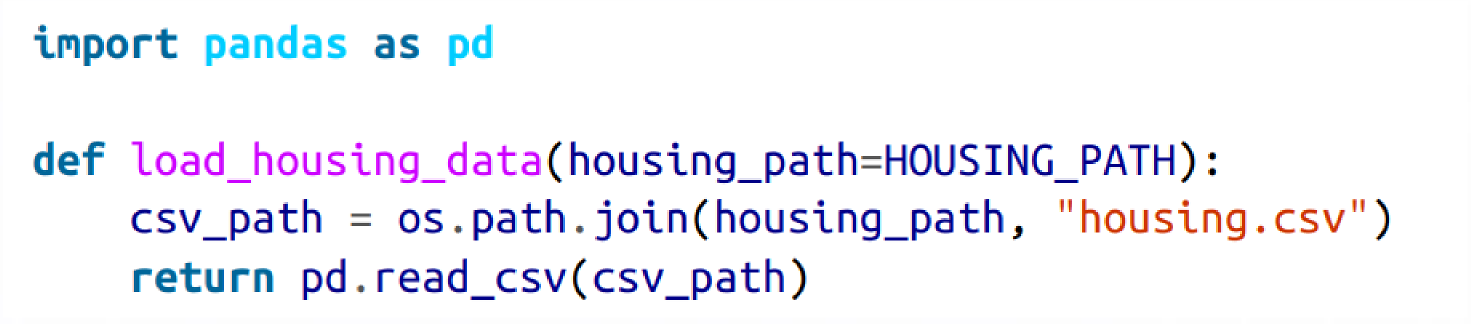
Here is the function to fetch the data:[11]

```
import os
import tarfile
import urllib

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

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

Now when you call *fetch_housing_data(),* it creates a *datasets/housing* directory in your workspace, downloads the *housing.tgz* file, and extracts the *housing.csv* file from it in this directory.

Download the Data

Now let’s load the data using pandas. Once again, you should write a small function to load the data

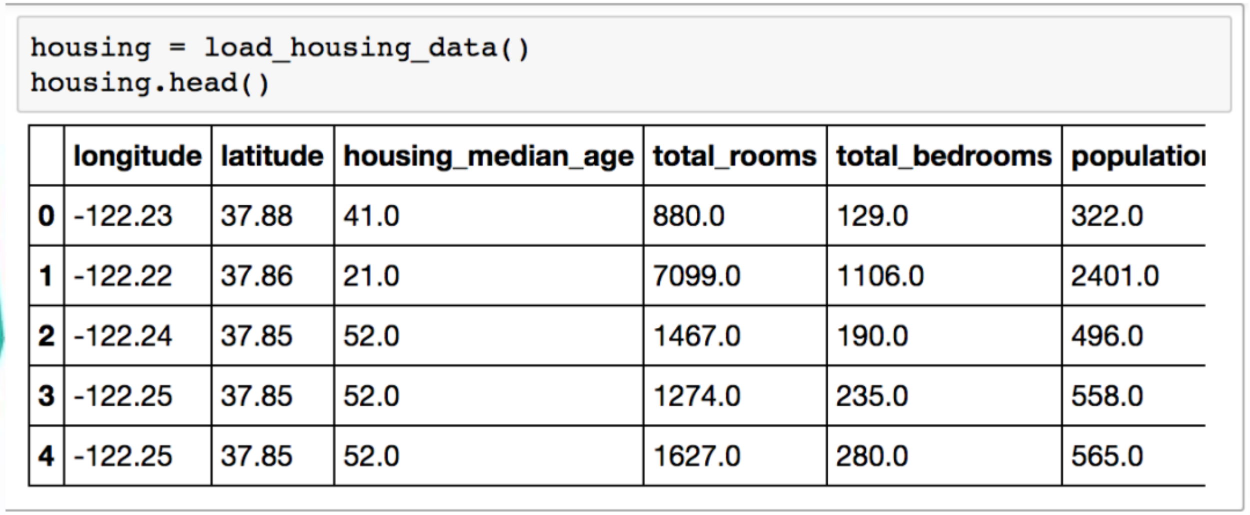


```
import pandas as pd

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

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

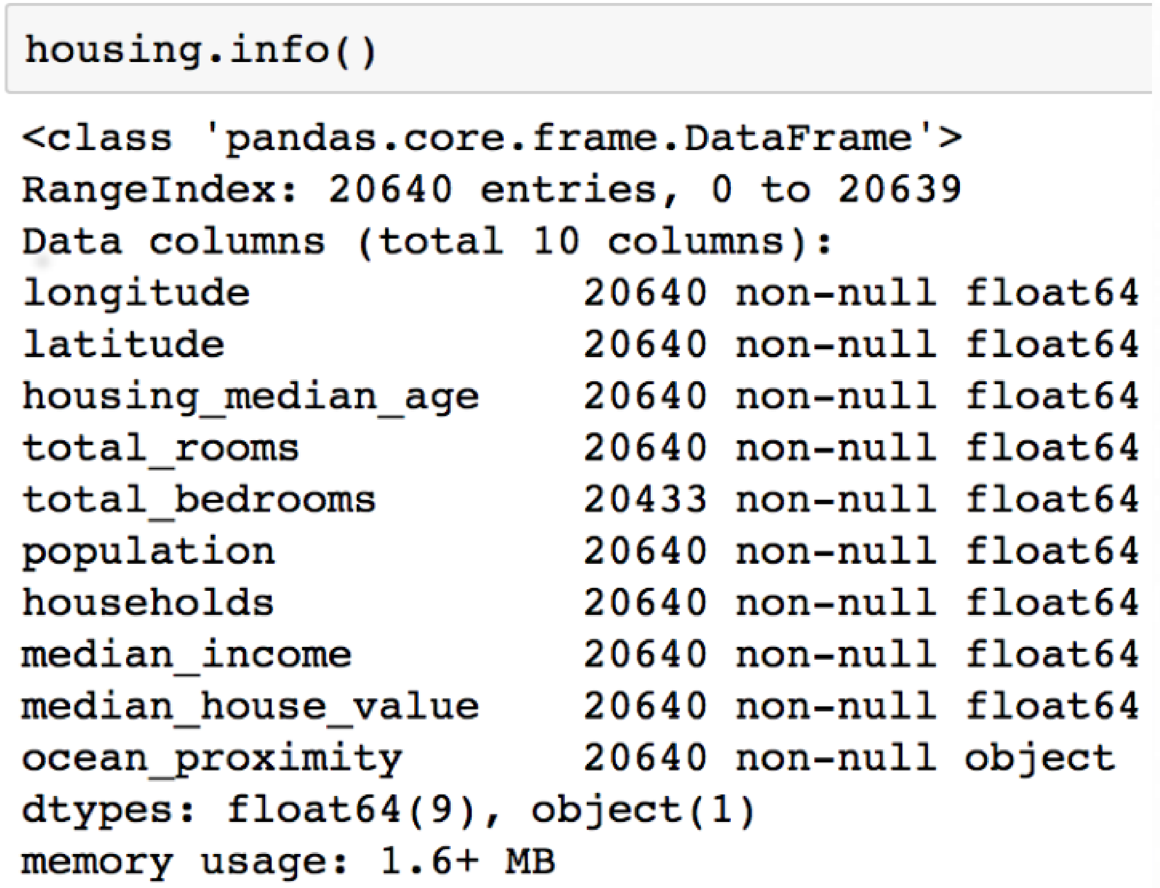
A *DataFrame* object in Pandas is a two-dimensional labeled data structure with columns that can hold different data types.

A quick look at the Data



```
housing = load_housing_data()
housing.head()
```

	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



```
housing.info()
```

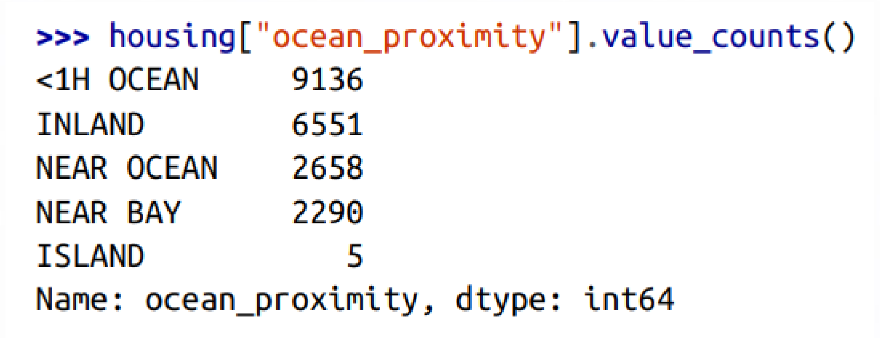
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        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
```

A quick look at the Data

All attributes are numerical, except the *ocean_proximity* field. Its type is object, so it could hold any kind of Python object.

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**.

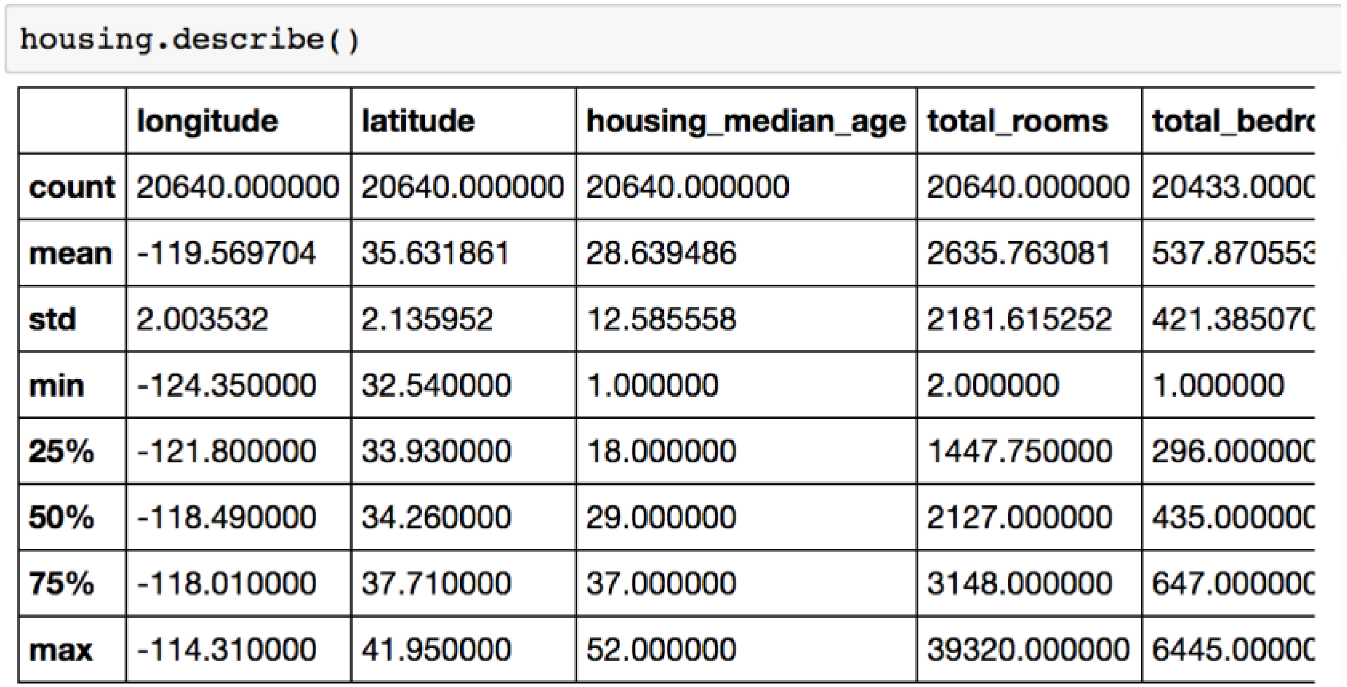
You can find out what categories exist and how many districts belong to each category by using the *value_counts()* method:



```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

A quick look at the Data

Let’s look at the other fields. The *describe()* method shows a summary of the **numerical attributes**

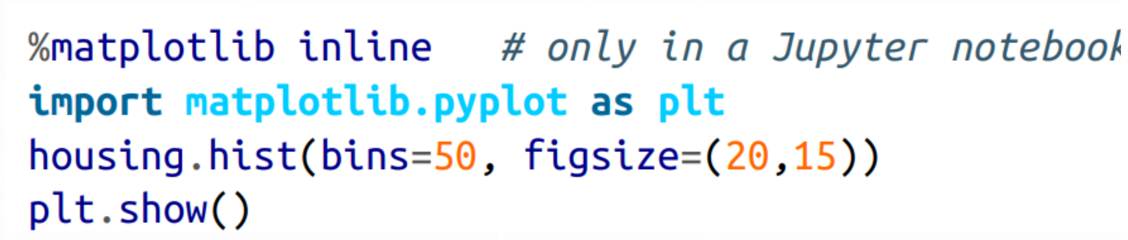


```
housing.describe()
```

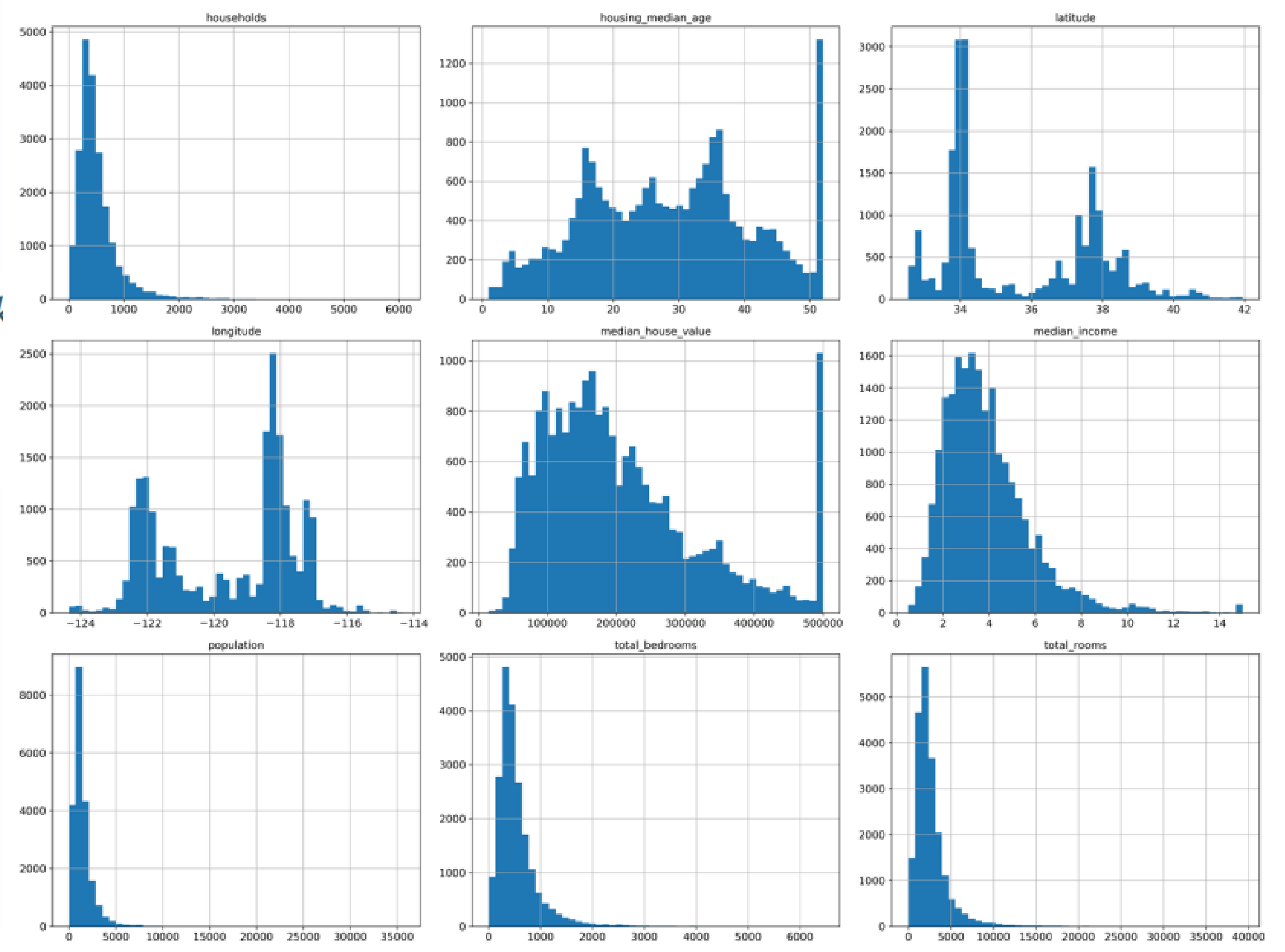
	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.0000
mean	-119.569704	35.631861	28.639486	2635.763081	537.87055
std	2.003532	2.135952	12.585558	2181.615252	421.38507
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	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

A quick look at the Data

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis).



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



A quick look at the Data

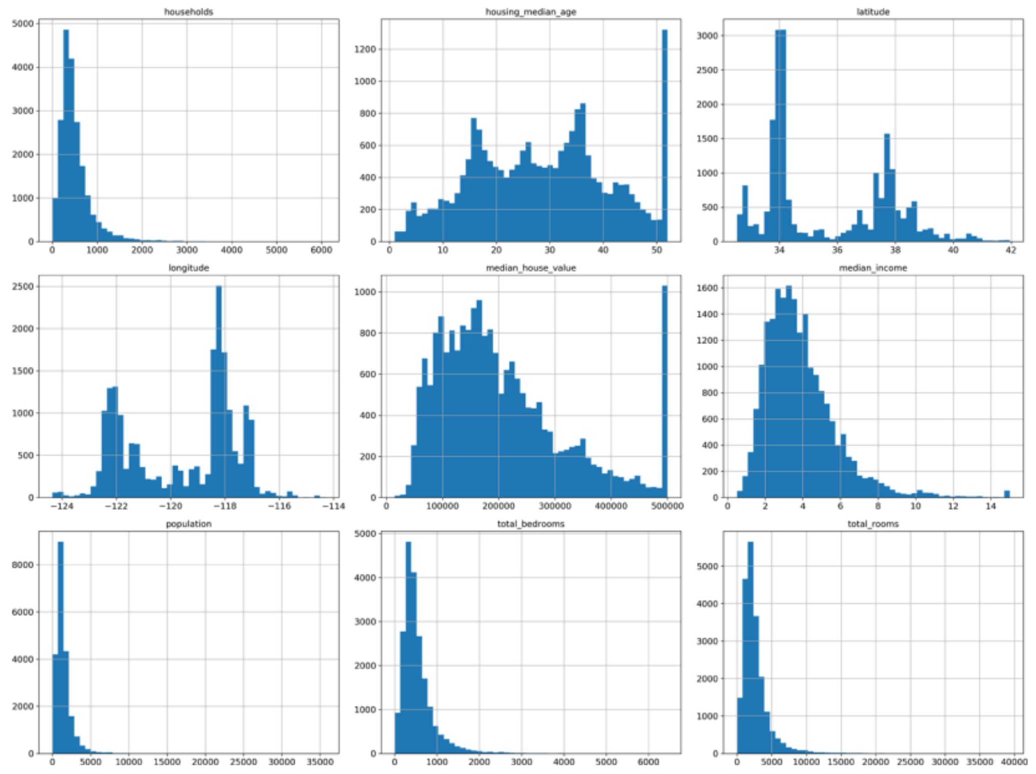
There are a few things you might notice in these histograms:

- the median income attribute does not look like it is expressed in US dollars (USD).

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

- These attributes have very different scales. We will discuss this later in this chapter, when we explore feature scaling.

- 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

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.

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***.

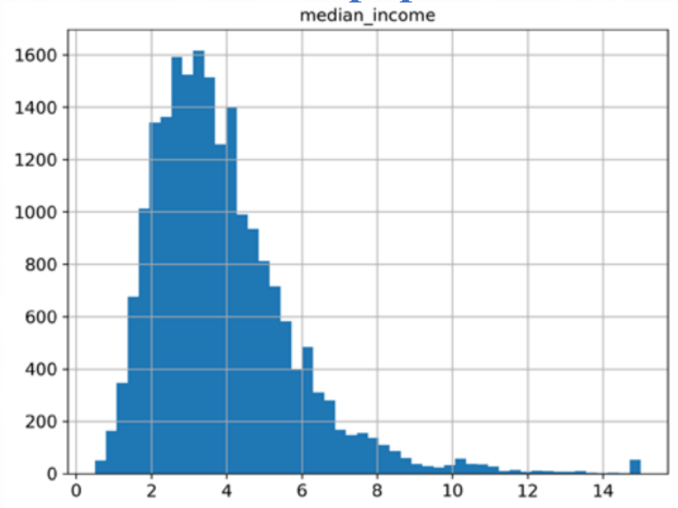
Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. First, there is a *random_state* parameter. If you set a specific *random_state*, you will get the same train-test split every time you run the code.

```
from sklearn.model_selection import train_test_split

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

Stratified sampling

The population is divided into homogeneous subgroups called *strata*, and the right number of instances are sampled from each stratum to guarantee that the test set is **representative of the overall population.**

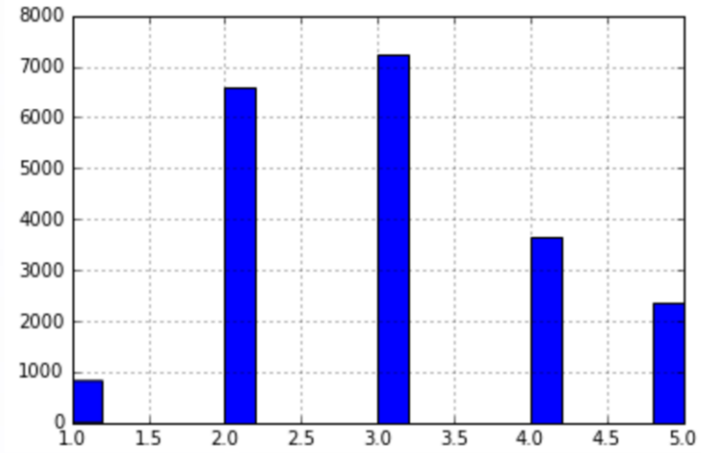
An expert told you that the *median income* is a very important attribute to predict *median housing prices*. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create an income category attribute.



The following code uses the *pd.cut()* function **to create an income category attribute** with five categories (labeled from 1 to 5)

```
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])
```

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



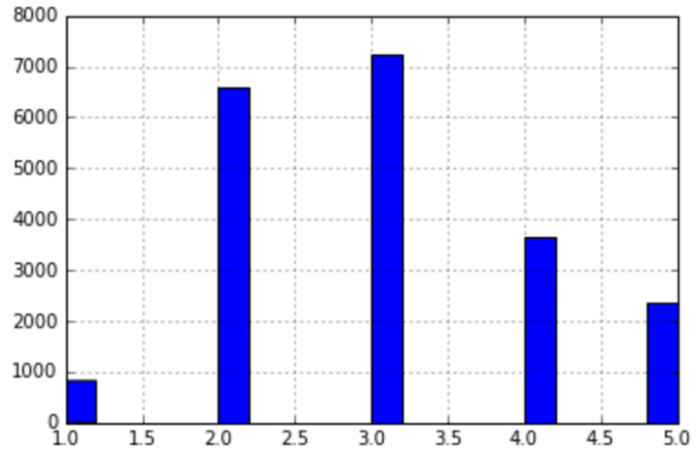
Stratified sampling

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]
```

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
```



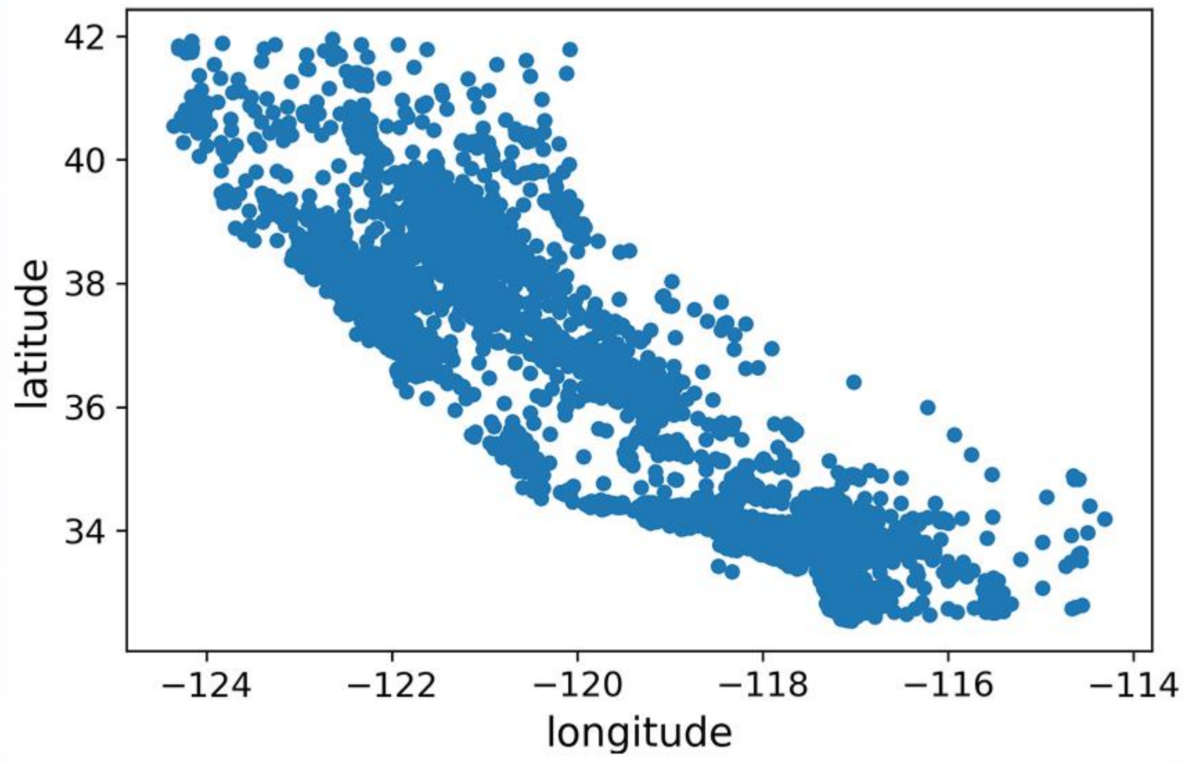
Explore and visualize the data

Put the test set aside and you are only **exploring the training set.**
Let’s create a copy so that you can play with it without harming the training set:

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

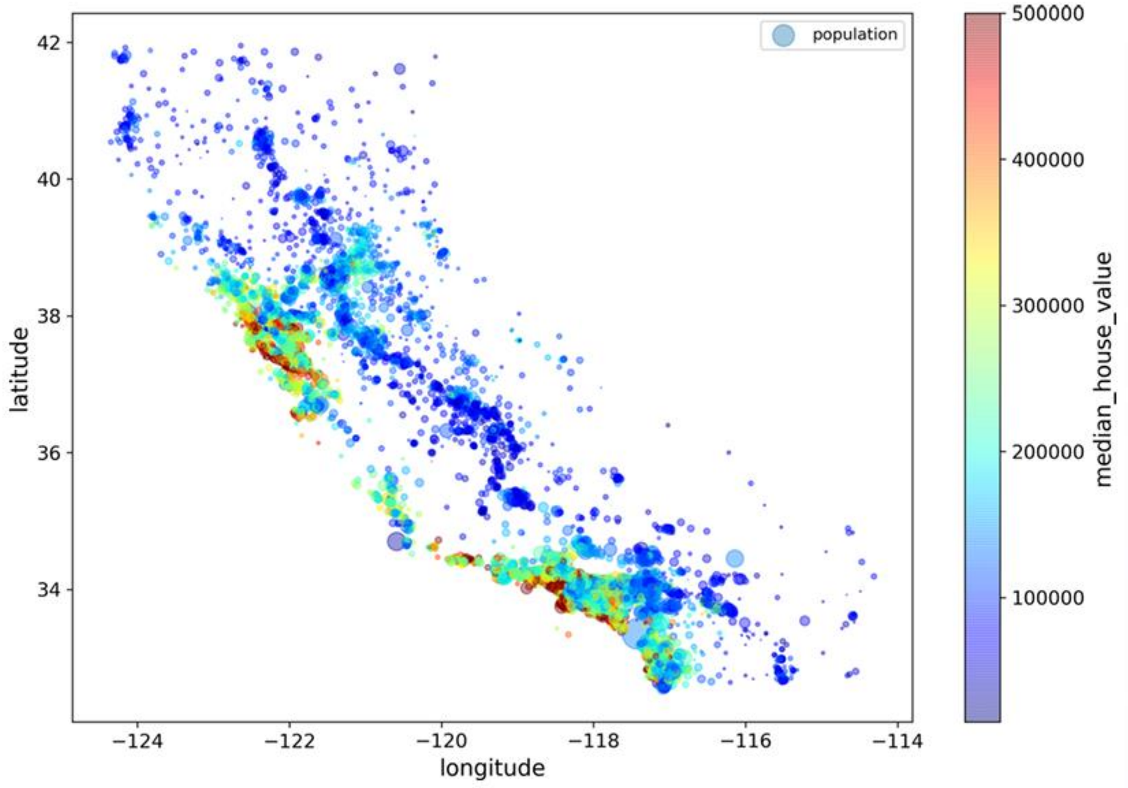
Create a scatterplot of all districts to visualize the data

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



Now let's look at the housing prices (Figure 2-13). The radius of each circle represents the district's population (option `s`), and the color represents the price (option `c`). We will use a predefined color map (option `cmap`) called `jet`, which ranges from blue (low values) to red (high prices):[16]

```
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()
```



Correlations between attributes

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()
```

Now let's look at how much each attribute correlates with the median house value:

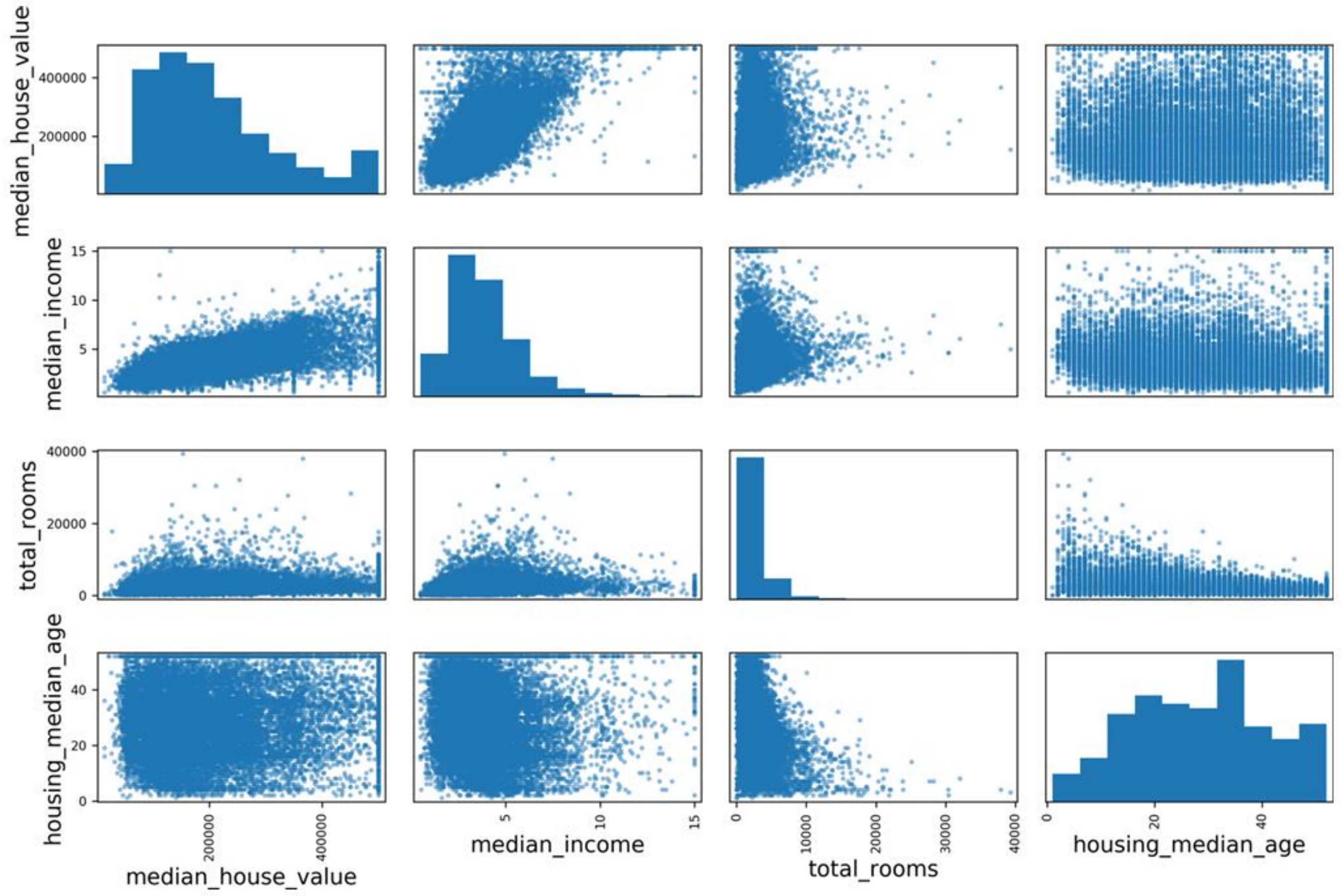
```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population           -0.026699
longitude   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

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

Another way to check for correlation between attributes is to use the pandas *scatter_matrix()* function, which plots every numerical attribute against every other numerical attribute.

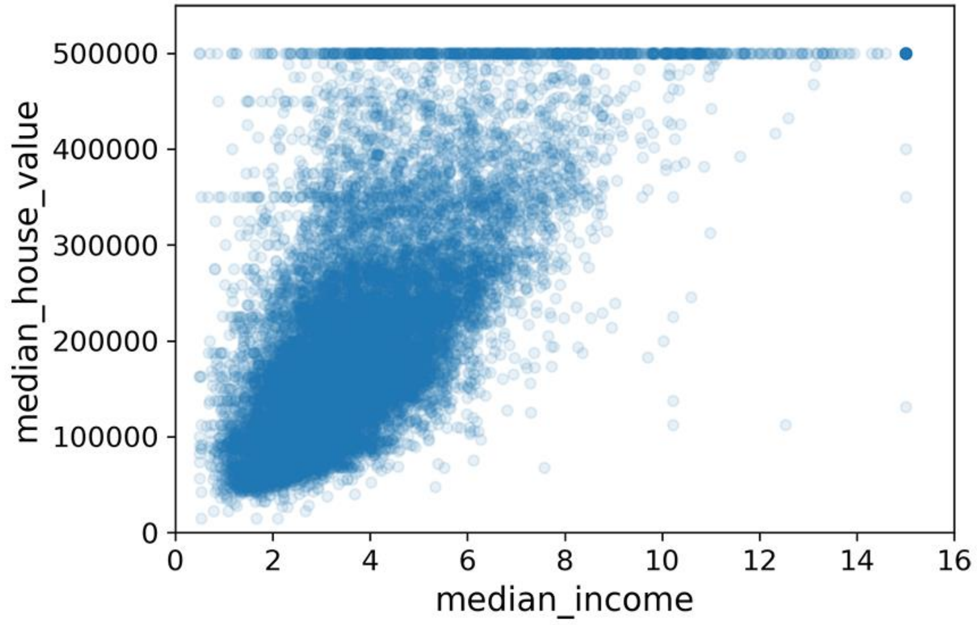
```
from pandas.plotting import scatter_matrix

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

The most promising attribute to predict the median house value is the ***median income.***



ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

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



The correlation is indeed very strong; you can clearly see the upward trend, and the points are not too dispersed.

the price cap that we noticed earlier is clearly visible as a horizontal line at $500,000.
You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

Experimenting with Attribute Combinations

```
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"]
```

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

```
>>> 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
```

Prepare the data for machine learning algorithms

It's time to prepare the data for your Machine Learning algorithms.

- 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.

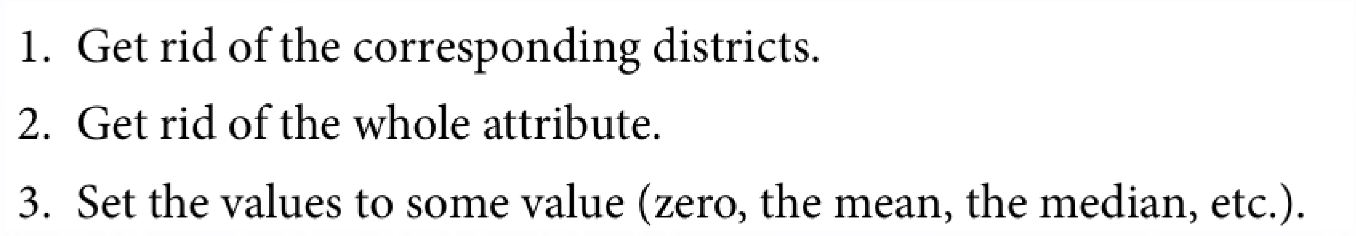
Let's separate the predictors and the labels, since we don't necessarily want to apply the same transformations to the predictors and the target values (note that *drop()* creates a copy of the data and does not affect *strat_train_set*):

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

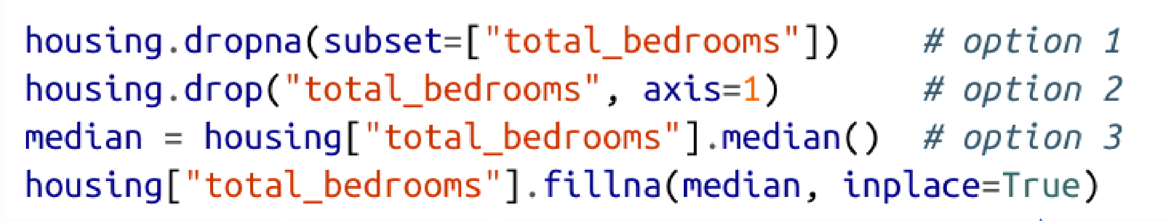
remove a column (axis=1) rather than a row (axis=0)

Data Cleaning

Most Machine Learning algorithms cannot work with ***missing features.*** 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.).



```
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)
```

The *inplace=True* argument ensures that
the modifications are made directly to the original housing *DataFrame*, rather than creating a copy.

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

```
from sklearn.impute import SimpleImputer

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

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)
```

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

The *transform* method uses the statistics learned during *fit* to fill in missing values in each column of *housing_num*.

```
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
```

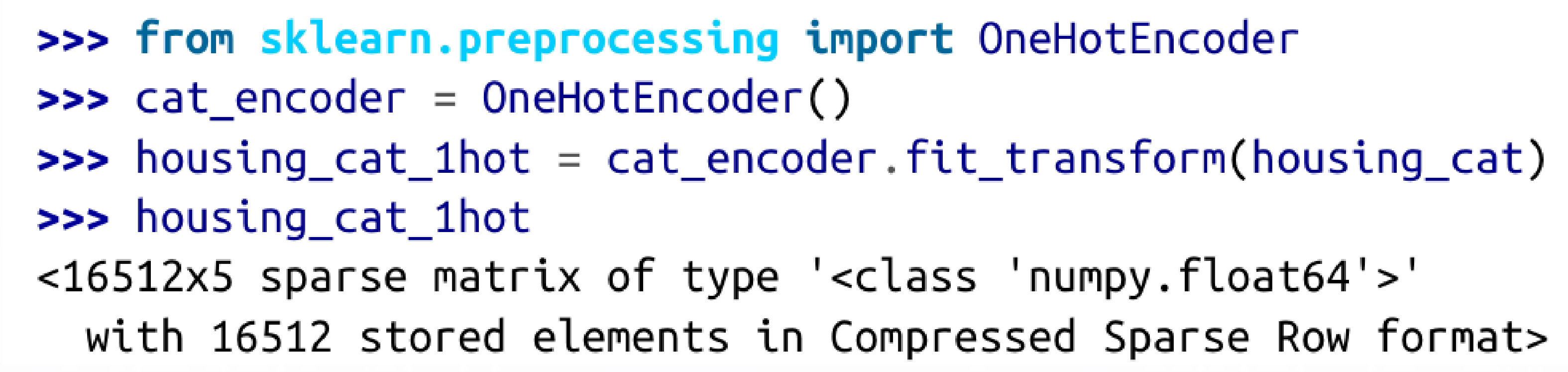
There are a limited number of possible values, each of which represents a category.
So this attribute is a **categorical attribute**.
Most Machine Learning algorithms prefer to work with numbers, so let’s convert these categories from text to numbers.
For this, we can use Scikit-Learn’s *OrdinalEncoder* class.

ستاد توسعه فناوری های
هوش مصنوعی و رباتیک

Ordinal Encoder

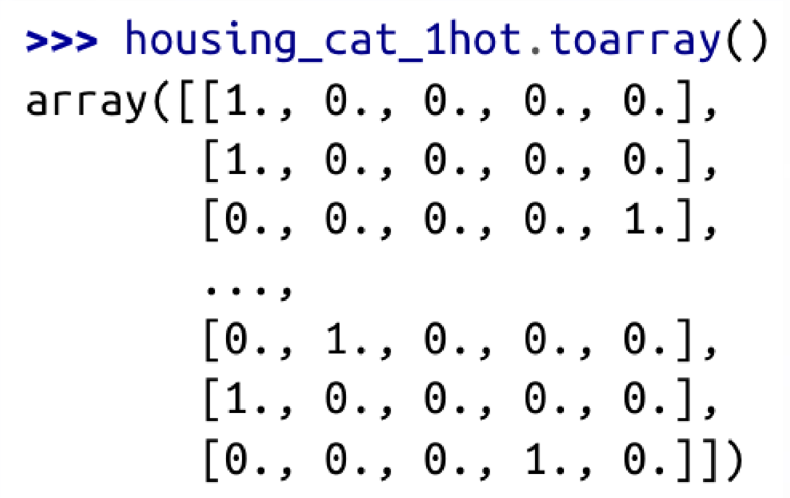
```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
  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

One-hot encoding



```
>>> 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>
```

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. 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.]])
```

Feature Scaling

One of the most important transformations you need to apply to your data is **feature scaling**. With few exceptions, **Machine Learning algorithms don't perform well when the input numerical attributes have very different scales**.
This is the case for 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. Note that **scaling the target values is generally not required**

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 trans former 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.

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- **Standardization**: 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). However, 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

Standardization:

$$z = \frac{x-\mu}{\sigma}$$

with mean:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} (x_i)$$

and standard deviation:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}$$

```
# Assuming your DataFrame is named 'housing'
scaler = MinMaxScaler() # or StandardScaler()

# Fit the scaler to your data
scaler.fit(housing)

# Transform the data
housing_scaled = scaler.transform(housing)

# Convert the scaled data back to a DataFrame if needed
housing_scaled = pd.DataFrame(housing_scaled,
columns=housing.columns)
```

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Standardization:

$$z = \frac{x-\mu}{\sigma}$$

with mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N}(x_i)$$

and standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

As with all the transformations, it is important to fit the scalers to the training data only, not to the full dataset (including the test set). Only then can you use them to transform the training set and the test set (and new data).

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.

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

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

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

So far, we have handled the categorical columns and the numerical columns separately. It 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, and the good news is that it works great with pandas *DataFrames*.

```
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)
```

Select and Train a Model

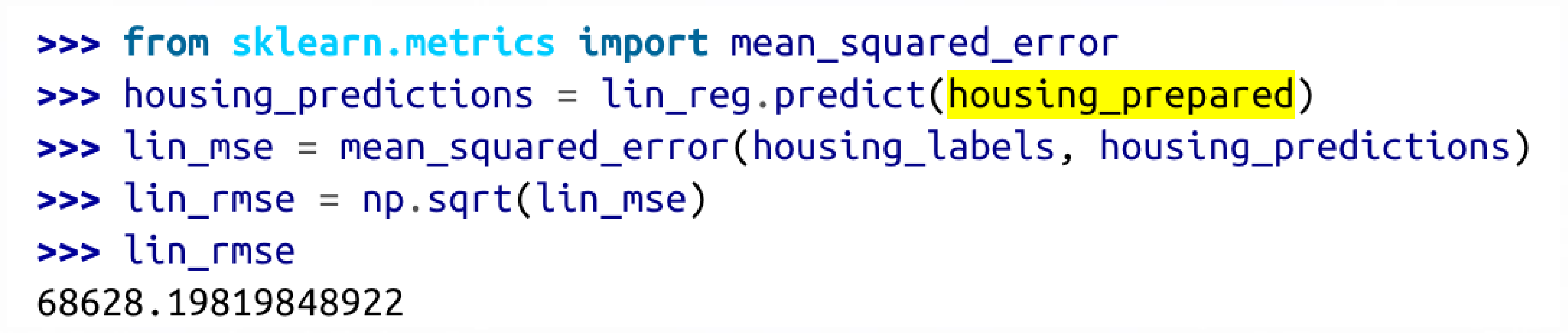
You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote transformation pipelines to clean up and prepare your data for Machine Learning algorithms automatically. You are now ready to select and train a Machine Learning model.

Let's first train a Linear Regression model

```
from sklearn.linear_model import LinearRegression

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

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]
```



```
>>> 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
```

most districts' *median_hous ing_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.

First let's try **a more complex model** to see how it does:

DecisionTreeRegressor

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
```

It is much more likely that the model has badly overfit the data.

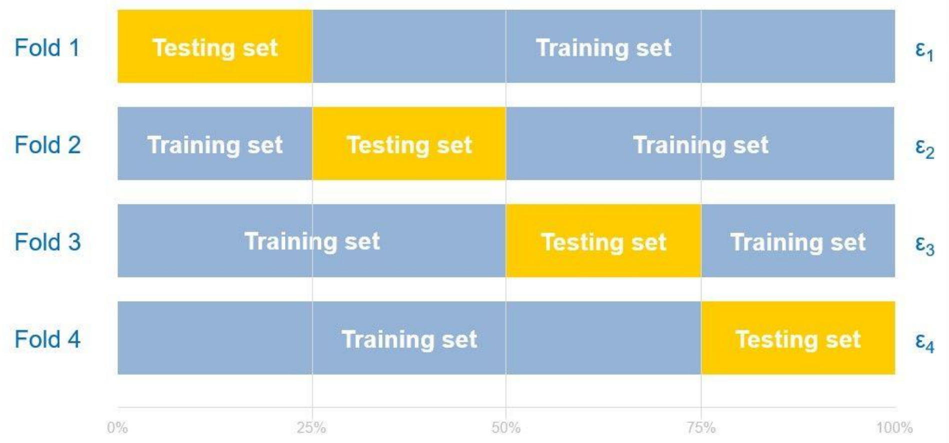
One way to evaluate the Decision Tree model would be to use the *train_test_split()* function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set.

A great alternative is to use **Scikit-Learn’s K-fold cross-validation feature.**

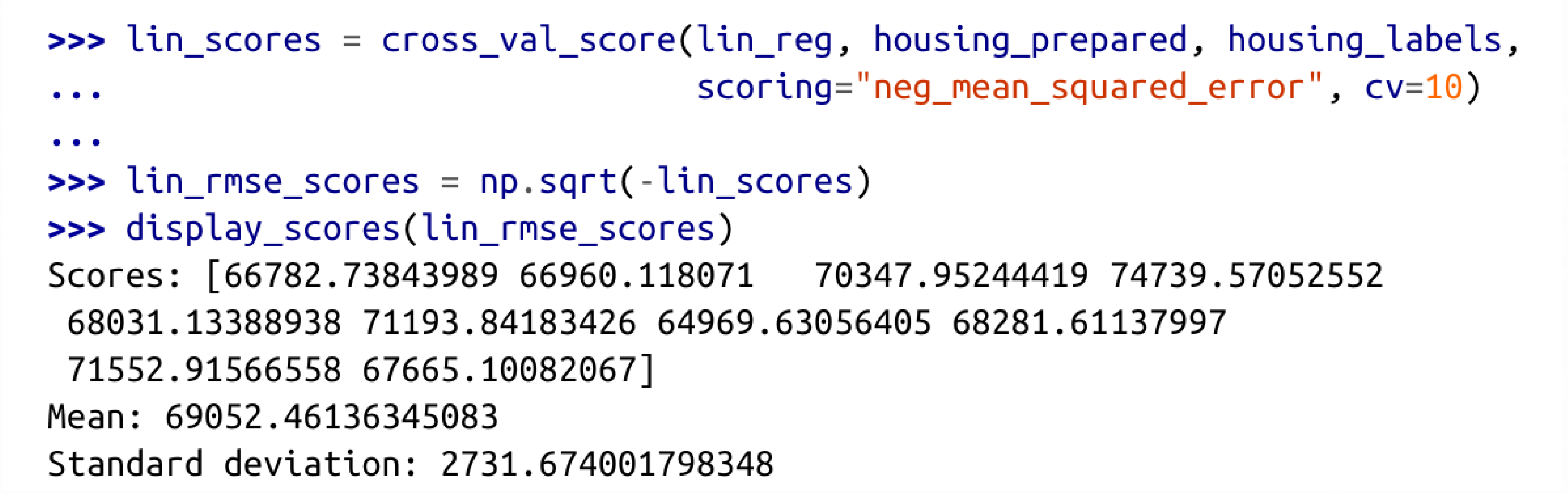
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

```
>>> 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
 71115.88230639 75585.14172901 70262.86139133 70273.6
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

4-fold validation (k=4)



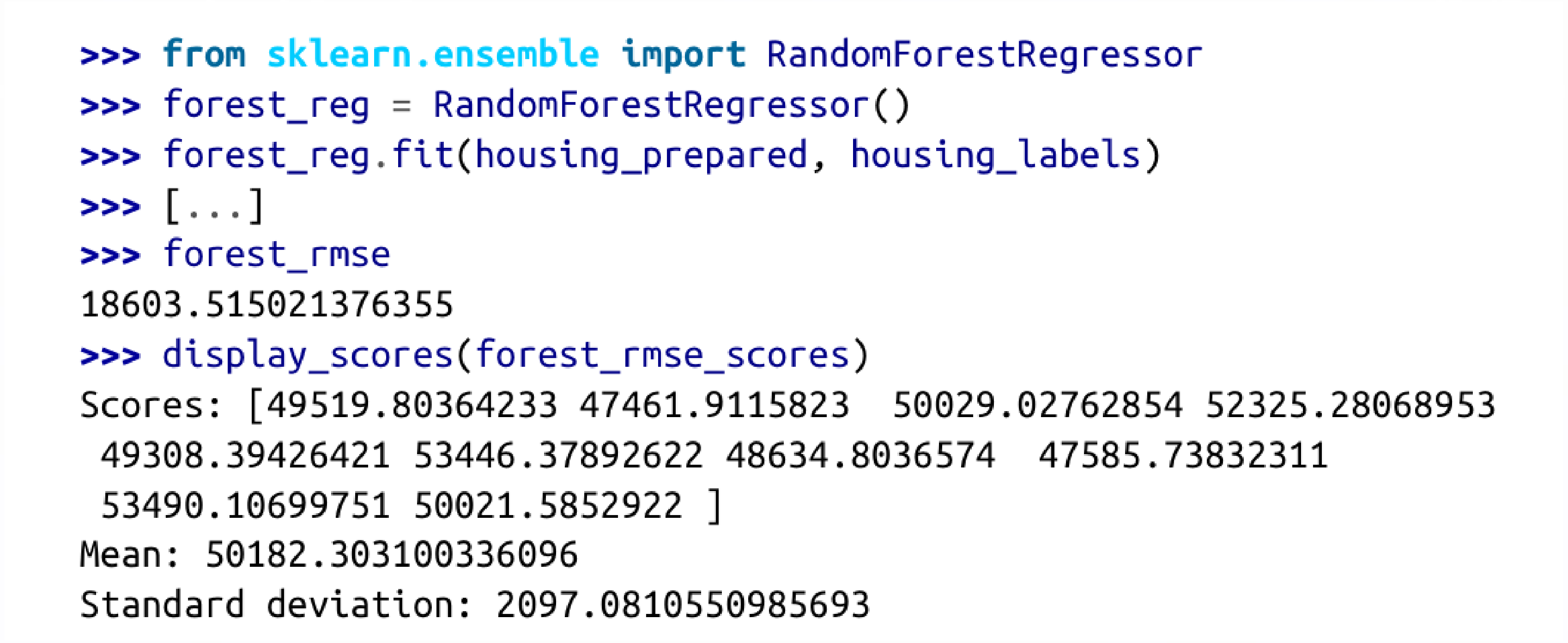
Using Cross validation in Linear Regression



```
>>> 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
```

The Decision Tree model is overfitting so badly that it performs worse than the Linear Regression model.

Let’s try one last model now: the *RandomForestRegressor*. As we will see in Chapter 7, 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.



```
>>> 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
```

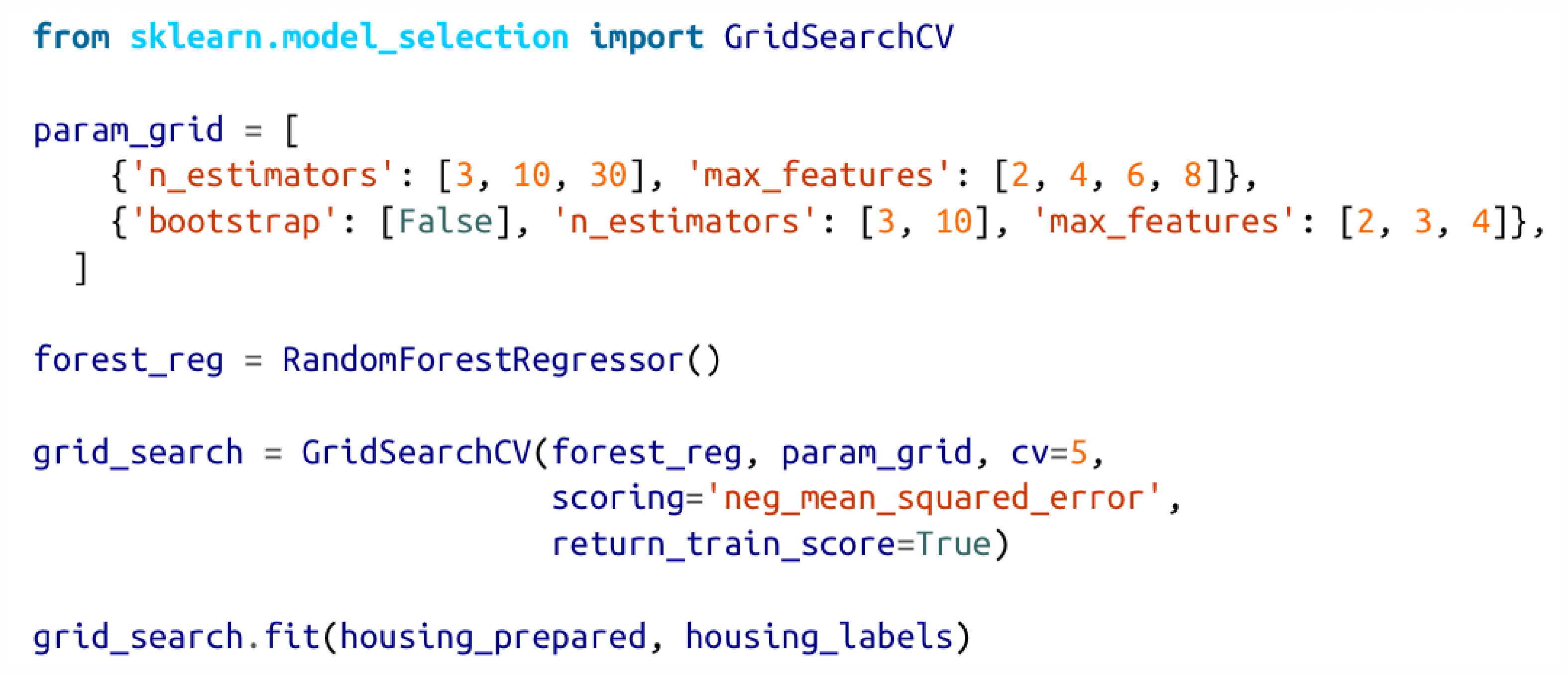
Fine Tune the model

Find a great combination of hyperparameter values.

You should get Scikit-Learn’s *GridSearchCV*

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

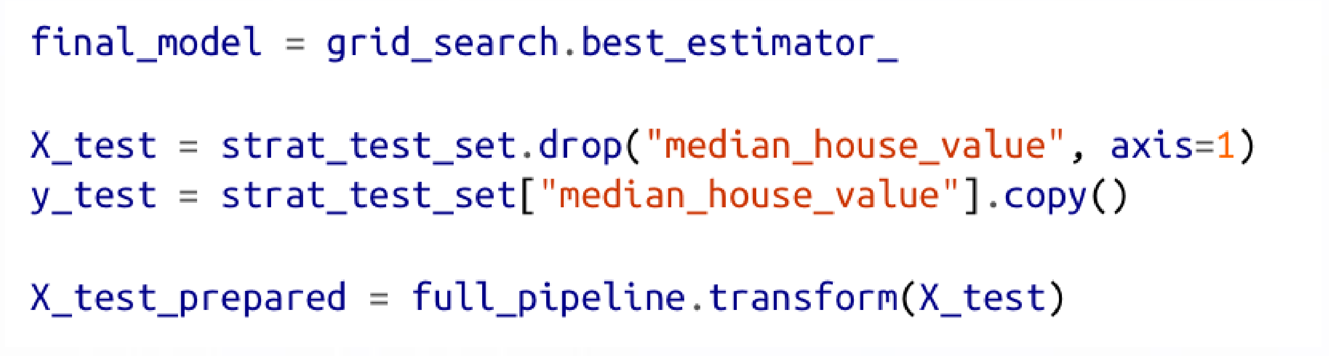
(don’t worry about what these hyperparameters mean for now; they will be explained in Chapter 7)



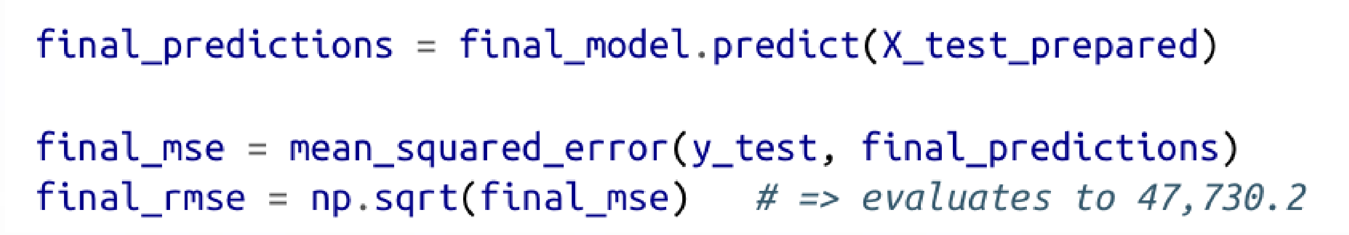
```
from sklearn.model_selection import GridSearchCV

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)
```

Evaluate Your System on the Test Set

Now is the time to evaluate the final model on the test set



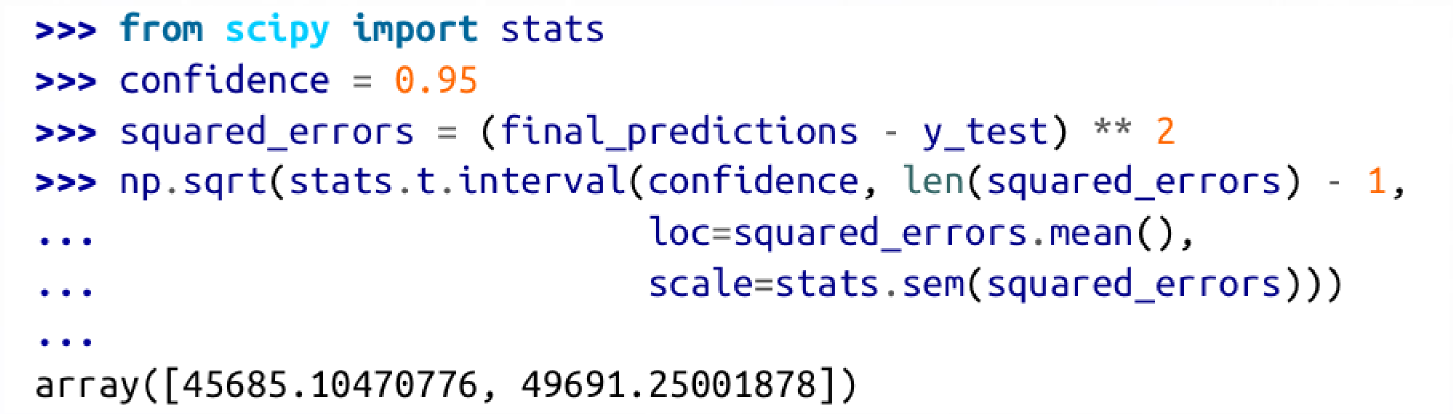
```
final_model = grid_search.best_estimator_

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
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is.

For this, you can compute a 95% confidence interval for the generalization error using *scipy.stats.t.interval()*:



```
>>> 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])
```