Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

Concepts, Tools, and Techniques to Build Intelligent Systems

Stephen CUI[1]

October 30, 2022

[1]cuixuanStephen@gmail.com

Contents

1 End-to-End Machine Learning Project **1**

1.1 Working with Real Data . . . 1

1.2 Look at the Big Picture . . . 1

1.2.1 Frame the Problem . . . 1

1.2.2 Select a Performance Measure . . . 2

1.2.3 Check the Assumptions . . . 4

1.3 Get the Data . . . 4

1.3.1 Create the Workspace . . . 4

1.3.2 Download the Data . . . 5

1.3.3 Take a Quick Look at the Data Structure . . . 5

1.3.4 Create a Test Set . . . 7

1.4 Discover and Visualize the Data to Gain Insights . . . 10

1.4.1 Visualizing Geographical Data . . . 10

1.4.2 Looking for Correlations . . . 12

1.4.3 Experimenting with Attribute Combinations . . . 14

1.4.4 Handling Text and Categorical Attributes . . . 14

1.4.5 Feature Scaling and Transformation . . . 15

1.4.6 Custom Transformers . . . 18

1.4.7 Transformation Pipelines . . . 19

1.5 Select and Train a Model . . . 23

1.5.1 Train and Evaluate on the Training Set . . . 23

1.5.2 Better Evaluation Using Cross-Validation . . . 24

1.6 Fine-Tune Your Model . . . 26

1.6.1 Grid Search . . . 26

1.6.2 Randomized Search . . . 27

1.6.3 Ensemble Methods . . . 29

1.6.4 Analyzing the Best Models and Their Errors . . . 29

1.6.5 Evaluate Your System on the Test Set . . . 30

1.7 Launch, Monitor, and Maintain Your System . . . 31

1.8 Try It Out! . . . 32

2 Classification **33**

2.1 MNIST . . . 33
2.2 Training a Binary Classifier . . . 35
2.3 Performance Measures . . . 36
2.3.1 Measuring Accuracy Using Cross-Validation . . . 36
2.3.2 Confusion Matrices . . . 38
2.3.3 Precision and Recall . . . 38
2.3.4 The Precision/Recall Trade-off . . . 39
2.3.5 The ROC Curve . . . 42
2.4 Multiclass Classification . . . 42
2.4.1 . . . 42
2.4.2 . . . 42
2.5 . . . 42
2.5.1 . . . 42
2.5.2 . . . 42
2.6 . . . 42
2.7 . . . 42
2.8 . . . 42

3 Decision Trees **43**

4 Ensemble Learning and Random Forests **44**

5 Training and Deploying TensorFlow Models at Scale **45**

Chapter 1

End-to-End Machine Learning Project

1.1 Working with Real Data

When you are learning about Machine Learning, it is best to experiment with realworld data, not artificial datasets. Here are a few places you can look to get data:

- Popular open data repositories
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- 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

1.2 Look at the Big Picture

1.2.1 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

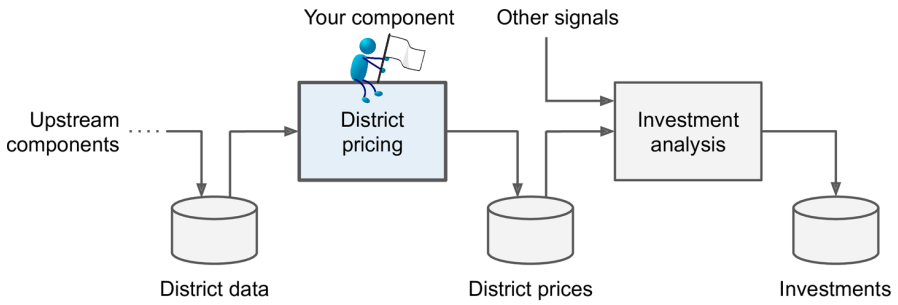

Figure 1.1: A Machine Learning pipeline for real estate investments

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.

Your boss answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system (see Figure 1.1), along with many other signals[1].

Pipelines

A sequence of data processing components is called a data pipeline. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply. Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.
On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system's performance drops.

The next question to ask your boss is what the current solution looks like (if any). The current situation will often give you a reference for performance

Tips: If the data were huge, you could either split your batch learning work across multiple servers (using the MapReduce technique) or use an online learning technique.

1.2.2 Select a Performance Measure

Your next step is to 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 1.1 shows the mathematical formula to compute the

[1] A piece of information fed to a Machine Learning system is often called a signal, in reference to Claude Shannon's information theory, which he developed at Bell Labs to improve telecommunications. His theory: you want a high signal-to-noise ratio.

RMSE.

$$RMSE(\mathbf{X},h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2} \tag{1.1}$$

Notations

This equation introduces several very common Machine Learning notations that we will use throughout this book:

- m is the number of instances in the dataset you are measuring the RMSE on.
- $\mathbf{x}^{(i)}$ 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). e.g.,

$$\mathbf{x}^{(i)} = \begin{bmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{bmatrix}$$

- $\mathbf{X}$ 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 $(\mathbf{x}^{(i)})^T$. e.g.,

$$\mathbf{x}^{(i)} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(m-1)})^T \\ (\mathbf{x}^{(m)})^T \end{bmatrix} = \begin{bmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

- 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.
- $RMSE(\mathbf{X},h)$ is the cost function measured on the set of examples using your hypothesis h.

We use lowercase italic font for scalar values and function names, lowercase bold font for vectors, and uppercase bold font for matrices.

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; see Equation 1.2):

$$MAE(\mathbf{X},h) = \frac{1}{m}\sum_{i=1}^{m}|h(x^{(i)}) - y^{(i)}| \tag{1.2}$$

Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values. Various distance measures, or *norms*, are possible:

- 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 $||v||_k = (|v_0|^k + |v_1|^k + \cdots + |v_n|^k)^{1/k}$. 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.

1.2.3 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. For example, the district prices that your system outputs are going to be fed into a downstream Machine Learning system, and you assume that these prices are going to be used as such. 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 pri- ces themselves? In this case, getting the price perfectly right is not important at all; your system just needs to get the category right. If that's so, then the problem should have been framed as a classification task, not a regression task. You don't want to find this out after working on a regression system for months.

1.3 Get the Data

The full Jupyter notebook is available at https://github.com/JPL-JUNO/HOML.

1.3.1 Create the Workspace

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/.

If you already have Jupyter running with all these modules installed, you can safely skip to Download the Data.

1.3.2 Download the Data

Having a function that downloads the data is useful in particular if the data changes regularly: you can write a small script that uses the function to fetch the latest data (or you can set up a scheduled job to do that automatically at regular intervals). Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

1.3.3 Take a Quick Look at the Data Structure

Let's take a look at the five rows using the DataFrame's `sample(n=5)` method instead of `head(n=5)`.

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. 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.

You can find out what cate- gories exist and how many districts belong to each category by using the `value_counts()` method:

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

The `describe()` method shows a summary of the numerical attributes. When with many columns, you can use `transpose()` for best information.

```
# housing.describe()
housing.describe().transpose()
```

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). You can either plot this one attribute at a time, or you can call the `hist()` method on the whole dataset, and it will plot a histogram for each numerical attribute (see Figure 1.2):

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


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

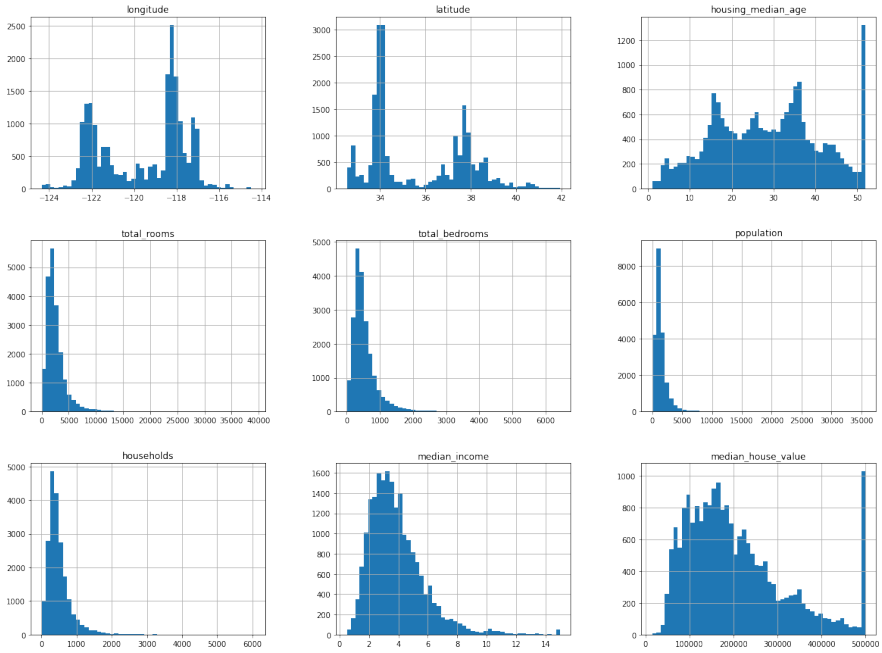

Figure 1.2: A histogram for each numerical attribute

Notes

The `hist()` method relies on Matplotlib, which in turn relies on a user-specified graphical backend to draw on your screen. So before you can plot anything, you need to specify which backend Matplotlib should use. The simplest option is to use Jupyter's magic command `%matplotlib inline`. This tells Jupyter to set up Matplotlib so it uses Jupyter's own backend. Plots are then rendered within the notebook itself. Note that calling `show()` is optional in a Jupyter notebook, as Jupyter will automatically display plots when a cell is executed.

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

1. 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. Working with preprocessed attributes is common in Machine Learning, and it is not necessarily a problem, but you should try to understand how the data was computed.

2. 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. 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).

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

4. 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.

Warnings

Wait! Before you look at the data any further, you need to create a test set, put it aside, and never look at it.

1.3.4 Create a Test Set

It may sound strange to voluntarily set aside part of the data at this stage. After all, you have only taken a quick glance at the data, and surely you should learn a whole lot more about it before you decide what algorithms to use, right? This is true, but your brain is an amazing pattern detection system, which means that it is highly prone to overfitting: 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.

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]

train_set, test_set = split_train_test(housing, .2)
len(train_set), len(test_set)
# (16512, 4128)
```

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(42)`)[2] before calling `np.random.permutation()` so that it always generates the same shuffled indices.

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). For example, you could 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(more about crc32):

```
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]

housing_with_id = housing.reset_index()
train_set, test_set = split_train_test_by_id(housing_with_id, .2, 'index')
len(train_set), len(test_set)
# (16512, 4128)
```

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:

```
housing_with_id['id'] = housing['longitude'] * 1000 + housing['latitude']
train_set, test_set = split_train_test_by_id(housing_with_id, .2, 'id')
len(train_set), len(test_set)
# (16322, 4318)
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is `train_test_split()`. First, there is a `random_state` parameter that allows you to set the random

[2] You will often see people set the random seed to 42. This number has no special property, other than to be the Answer to the Ultimate Question of Life, the Universe, and Everything.

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=.2, random_state=42)
len(train_set), len(test_set)
# (16512, 4128)
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you **run the risk of introducing a significant sampling bias**. When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population. For example, the US population is 51.3% females and 48.7% males, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male. This is called 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. If the people running the survey used purely random sampling, there would be about a 11.29% chance of sampling a skewed test set that was either less than 49% female or more than 54% female(Why see Notebook chapter2). Either way, the survey results would be significantly biased.

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

```
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()
```

Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn's `StratifiedShuffleSplit`(more about StratifiedShuffleSplit) class:

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=.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]
strat_test_set['income_cat'].value_counts() / len(strat_test_set)
```

As you can see, the test set generated using stratified sampling has income category proportions almost

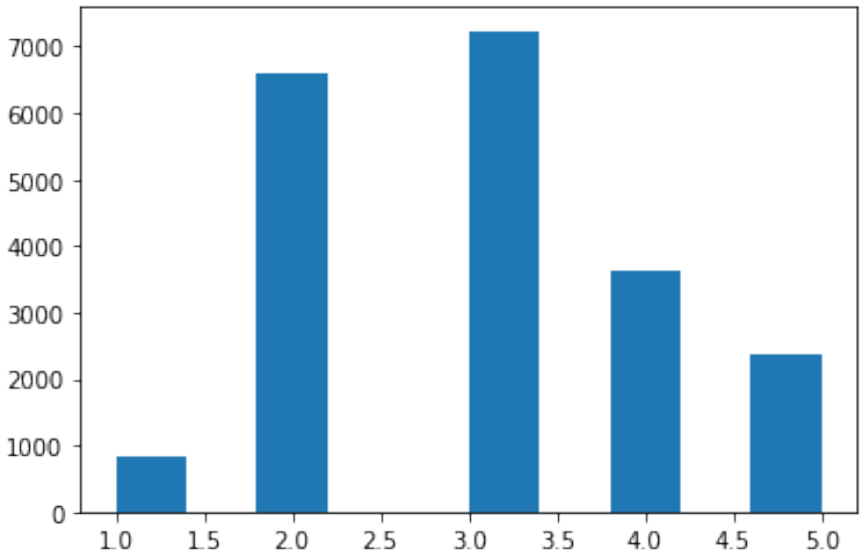

Figure 1.3: Histogram of income categories

identical to those in the full dataset, whereas the test set generated using purely random sampling is skewed(for more see Notebook chapter2).

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='columns', 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.

1.4 Discover and Visualize the Data to Gain Insights

First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast.

1.4.1 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 1.4):

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 1.5).

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.

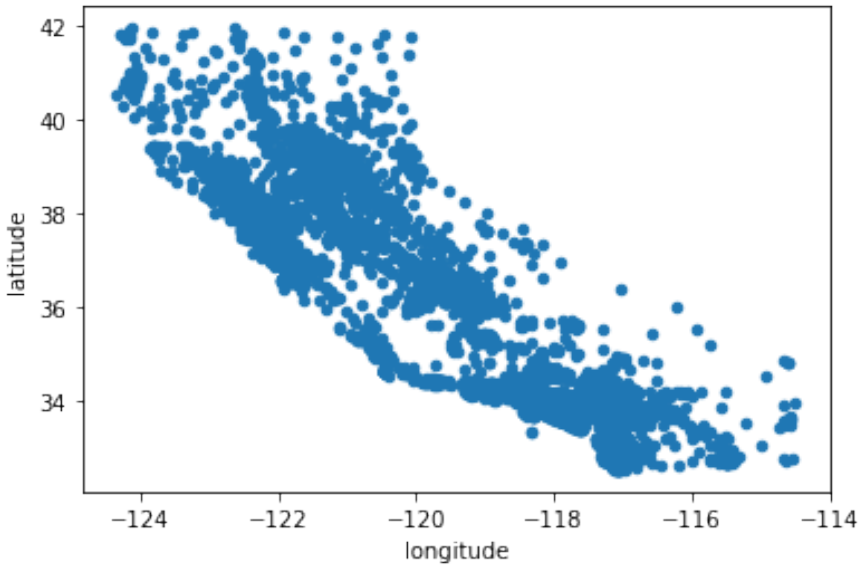

Figure 1.4: A geographical scatterplot of the data

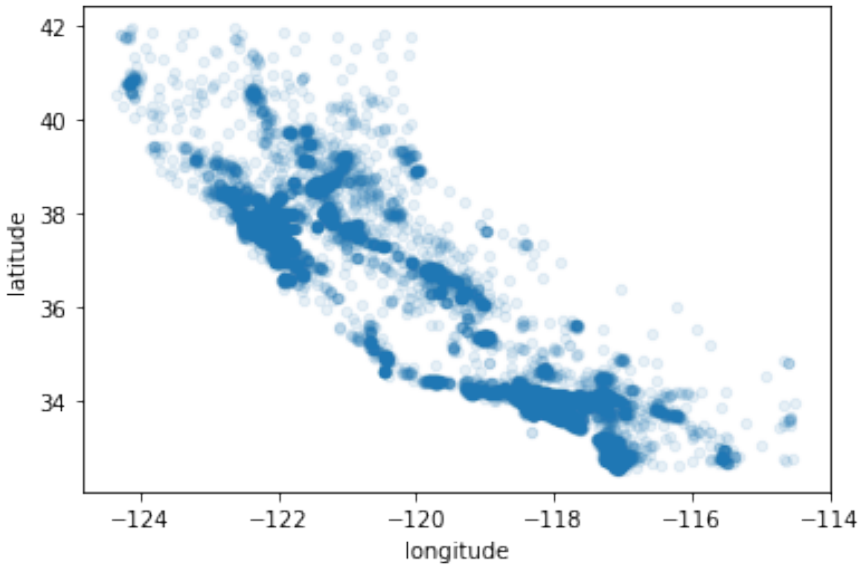

Figure 1.5: A better visualization that highlights high-density areas

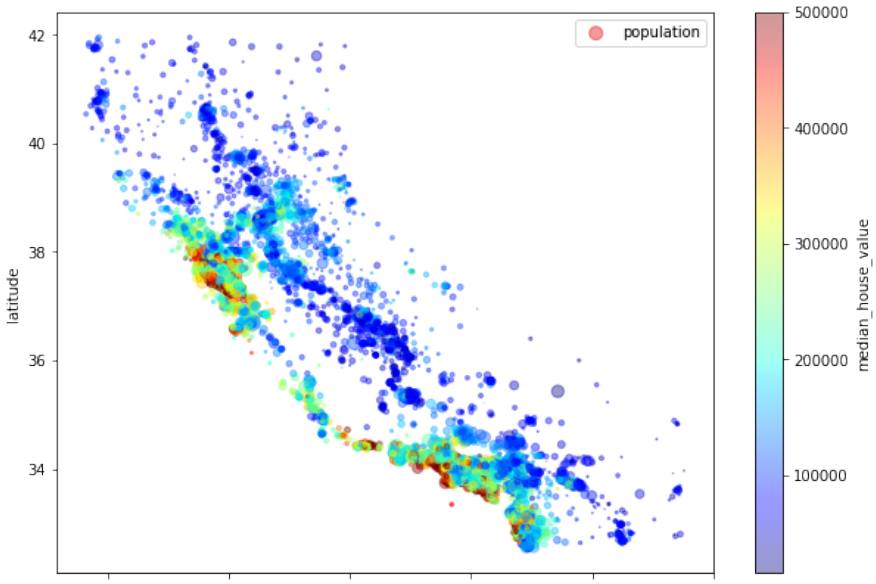

Figure 1.6: California housing prices

Now let's look at the housing prices (Figure 1.6).

```
housing.plot(kind='scatter', x='longitude', y='latitude', alpha=.4, s=housing['population']/100,
             label='population', figsize=(10, 7),
             c='median_house_value', cmap=plt.get_cmap('jet'), colorbar=True)
plt.legend()
plt.show()
```

This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers. The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.

1.4.2 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)
```

Figure 1.7 shows various plots along with the correlation coefficient between their horizontal and vertical axes.

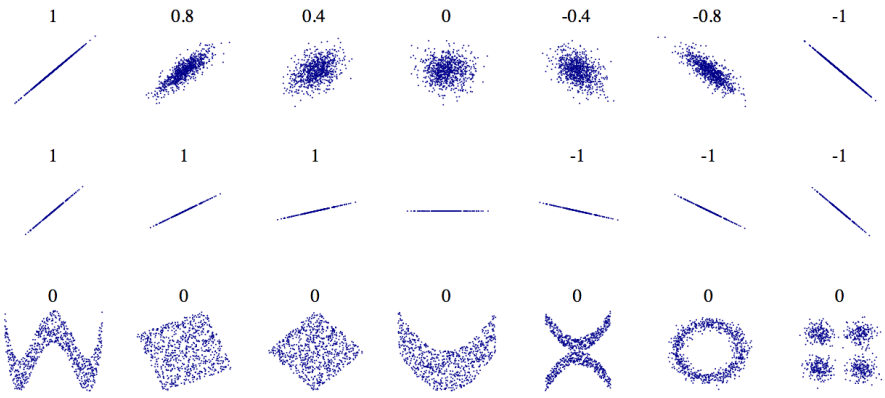

Figure 1.7: Standard correlation coefficient of various datasets

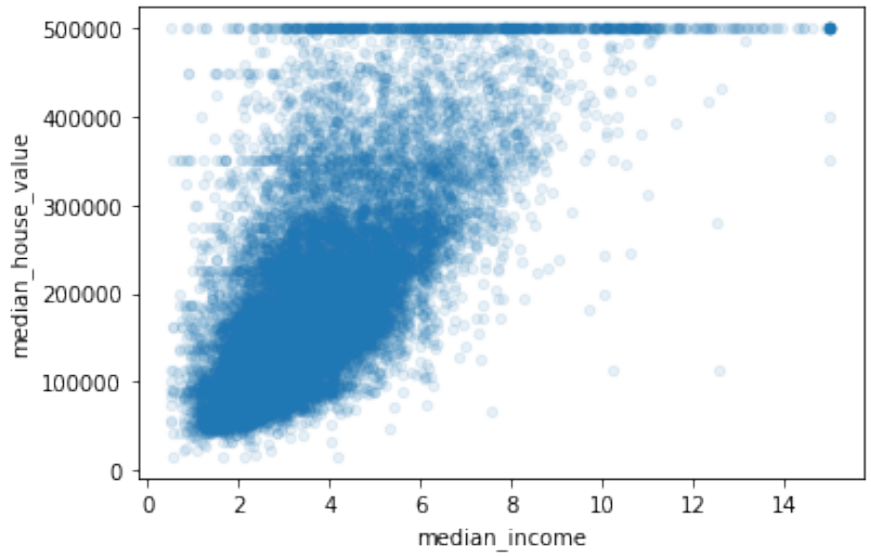

Figure 1.8: Median income versus median house value

Warnings

The correlation coefficient only measures linear correlations ("if x goes up, then y generally goes up/-down"). It may completely miss out on nonlinear relationships (e.g., "if x is close to 0, then y generally goes up").

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(You can see here).

The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot (Figure 1.8):

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.

1.4.3 Experimenting with Attribute Combinations

1.4.4 Handling Text and Categorical Attributes

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:

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

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 (in this case, a list containing a single array since there is just one categorical attribute):

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). To 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), and so on. This is called one-hot encoding, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are sometimes called dummy attributes. Scikit-Learn provides a OneHotEncoder class to convert categorical values into one-hot vectors:

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

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. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row. 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. 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.

Suggestions

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the ocean_proximity feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you could replace each category with a learnable, low-dimensional vector called an embedding. Each category's representation would be learned during training. This is an example of representation learning (see Chapters **??** and **??** for more details).

1.4.5 Feature Scaling and Transformation

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

Warnings

As with all estimators, it is important to fit the scalers to the training data only: never use fit() or fit_transform() for anything else than the training set. Once you have a trained scaler, you can then use it to transform() any other set, including the validation set, the test set, and new data. Note that while the training set values will always be scaled to the specified range, if new data contains outliers, these may end up scaled outside the range. If you want to avoid this, just set the clip hyperparameter to True.

Min-max scaling (many people call this normalization) is the simplest: for each attribute, the values are shifted and rescaled so that they end up ranging from 0 to 1. This is performed by subtracting the min value and dividing by the difference between the min and the max. 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 (e.g., neural networks work best with zero-mean inputs, so a range of –1 to 1 is preferable).

Standardization is different: first it subtracts the mean value (so standardized values have a zero mean), then it divides the result by the standard deviation (so standardized values have a standard deviation equal to 1). Unlike min-max scaling, standardization does not restrict values to a specific range. However, standardization is much less affected by outliers. For example, suppose a district has a median income equal to 100 (by mistake), instead of the usual 0–15. Min-max scaling to the 0–1 range would map this outlier down to 1 and it would crush all the other values down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called StandardScaler for standardization.

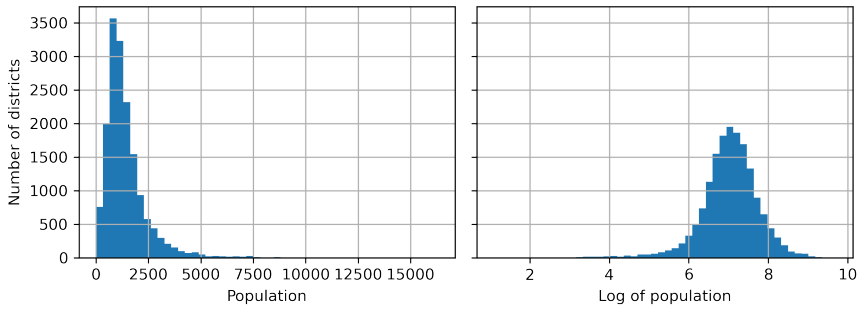

Figure 1.9: Transforming a feature to make it closer to a Gaussian distribution

Suggestions

If you want to scale a sparse matrix without converting it to a dense matrix first, you can use a StandardScaler with its with_mean hyperparameter set to False: it will only divide the data by the standard deviation, without subtracting the mean (as this would break sparsity).

When a feature's distribution has a heavy tail (i.e., when values far from the mean are not exponentially rare), both min-max scaling and standardization will squash most values into a small range. Machine learning models generally don't like this at all, as you will see in **??**. So **before you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the distribution roughly symmetrical**. For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its square root (or raise the feature to a power between 0 and 1). If the feature has a really long and heavy tail, such as a power law distribution, then replacing the feature with its logarithm may help. For example, the population feature roughly follows a power law: districts with 10,000 inhabitants are only 10 times less frequent than districts with 1,000 inhabitants, not exponentially less frequent. Figure 1.9 shows how much better this feature looks when you compute its log: it's very close to a Gaussian distribution (i.e., bell-shaped).

Another approach to handle heavy-tailed features consists in *bucketizing* the feature. （对数据进行分箱处理）。 This means chopping its distribution into roughly equal-sized buckets, and replacing each feature value with the index of the bucket it belongs to.

When a feature has a multimodal distribution (i.e., with two or more clear peaks, called modes), it can also be helpful to bucketize it, but this time treating the bucket IDs as categories, rather than as numerical values.

Another approach to transforming multimodal distributions is to add a feature for each of the modes (at least the main ones), representing the similarity between the data and that particular mode. The similarity measure is typically computed using a *radial basis function (RBF, 径向基函数)*—any function that depends only on the distance between the input value and a fixed point. The most commonly used RBF is the Gaussian RBF, whose output value decays exponentially as the input value moves away from the fixed point. Figure 1.10 shows this new feature as a function of the housing median age (solid line).

So far we've only looked at the input features, but the target values may also need to be transformed. For example, if the target distribution has a heavy tail, you may choose to replace the target with its logarithm.

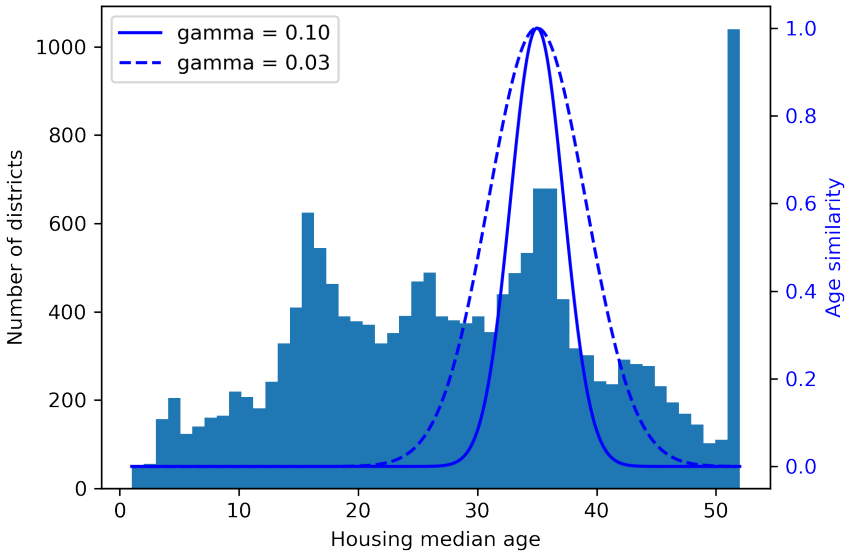

Figure 1.10: Gaussian RBF feature measuring the similarity

Luckily, most of Scikit-Learn's transformers have an `inverse_transform()` method, making it easy to compute the inverse of their transformations.

```
from sklearn.linear_model import LinearRegression

target_scalar = StandardScaler()
scaled_labels = target_scalar.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[['median_income']], scaled_labels)
# pretend this is new data
some_new_data = housing[['median_income']].iloc[:5]

scaled_predictions = model.predict(some_new_data)
predictions = target_scalar.inverse_transform(scaled_predictions)
# predictions.flatten()
```

This works fine, but a simpler option is to use a `TransformedTargetRegressor`. We just need to construct it, giving it the regression model and the label transformer, then fit it on the training set, using the original unscaled labels. It will automatically use the transformer to scale the labels and train the regression model on the resulting scaled labels. Then, when we want to make a prediction, it will call the regression model's `predict()` method and use the scaler's `inverse_transform()` method to produce the prediction.

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer=StandardScaler())
model.fit(housing[['median_income']], housing_labels)
predictions = model.predict(some_new_data)
```

1.4.6 Custom Transformers

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

For transformations that don't require any training, you can just write a function that takes a NumPy array as input and outputs the transformed array.

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[['population']])
```

The `inverse_func` argument is optional. Your transformation function can take hyperparameters as additional arguments. Custom transformers are also useful to combine features.

FunctionTransformer is very handy, but what if you would like your transformer to be trainable, learning some parameters in the fit() method and using them later in the transform() method? And you will want your transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines), and since Scikit-Learn relies on duck typing (not inheritance), all you need to do is create a class and implement three methods: `fit()`(returning `self`), `transform()`, and `fit_transform()`.

You can get `fit_transform()` for free by simply adding TransformerMixin as a base class: the default implementation will just call `fit()` and then `transform()`. If you add BaseEstimator as a base class (and avoid using *args and **kwargs in your constructor), you will also get two extra methods: `get_params()` and `set_params()`. These will be useful for automatic hyperparameter tuning.

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScaler(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True):
        self.with_mean = with_mean

    def fit(self, X, y=None):
```

```
        X = check_array(X)
        self.mean_ = X.mean(axis='index')
        self.scale_ = X.std(axis='index')
        self.n_features_in_ = X.shape[1]
        return self

    def transform(self, X):
        check_is_fitted(self)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

Here are a few things to note:

- The sklearn.utils.validation package contains several functions we can use to validate the inputs.
- Scikit-Learn pipelines require the fit() method to have two arguments X and y, which is why we need the y=None argument even though we don't use y.
- All Scikit-Learn estimators set `n_features_in_` in the fit() method, and they ensure that the data passed to transform() or predict() has this number of features.
- The fit() method must return self.
- This implementation is not 100% complete: all estimators should set `feature_names_in_` in the fit() method when they are passed a DataFrame. Moreover, all transformers should provide a `get_feature_names_out()` method, as well as an `inverse_transform()` method when their transformation can be reversed.

A custom transformer can (and often does) use other estimators in its implementation.

1.4.7 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.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([
    ('impute', SimpleImputer(strategy='median')),
    ('standardize', StandardScaler())
])
```

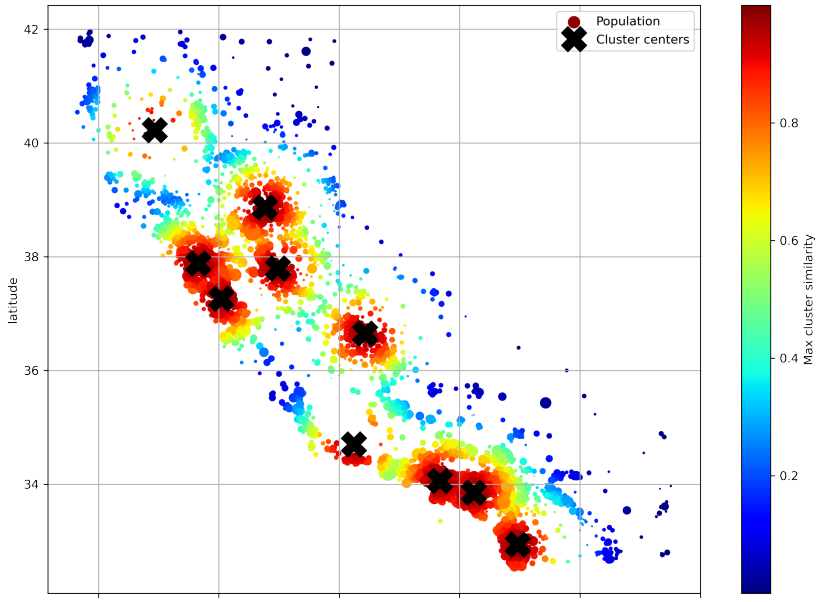

Figure 1.11: Gaussian RBF similarity to the nearest cluster center

The Pipeline constructor takes a list of name/estimator pairs (2-tuples) defining a sequence of steps. The names can be anything you like, as long as they are unique and don't contain double underscores (__). The estimators must all be transformers (i.e., they must have a `fit_transform()` method), except for the last one, which can be anything: a transformer, a predictor, or any other type of estimator.

If you don't want to name the transformers, you can use the `make_pipeline()` function instead; it takes transformers as positional arguments and creates a Pipeline using the names of the transformers' classes, in lowercase and without underscores.

```
from sklearn.pipeline import make_pipeline
num_pipeline = make_pipeline(SimpleImputer(strategy='median'), StandardScaler())
```

If multiple transformers have the same name, an index is appended to their names.

When you call the pipeline's fit() method, it calls `fit_transform()` sequentially on all the transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it just calls the fit() method.

If the last estimator were a predictor instead of a transformer, then the pipeline would have a predict() method rather than a transform() method. Calling it would sequentially apply all the transformations to the data and pass the result to the predictor's predict() method.

```
housing_num_prepared = num_pipeline.fit_transform(housing_num)
housing_num_prepared[: 2].round(2)
```

It would be more convenient to have a single transformer capable of handling all columns, applying the appropriate transformations to each column. For this, you can use a `ColumnTransformer`. Its constructor requires

a list of triplets (3-tuples), each containing a name (which must be unique and not contain double underscores), a transformer, and a list of names (or indices) of columns that the transformer should be applied to.

```
from sklearn.compose import ColumnTransformer

num_attribs = ['longitude', 'latitude', 'housing_median_age', 'total_rooms',
               'total_bedrooms', 'population', 'households', 'median_income']

cat_attribs = ['ocean_proximity']

cat_pipeline = make_pipeline(
    SimpleImputer(strategy='most_frequent'),
    OneHotEncoder(handle_unknown='ignore')
)

preprocessing = ColumnTransformer([
    ('num', num_pipeline, num_attribs),
    ('cat', cat_pipeline, cat_attribs)
])
```

Suggestions

Instead of using a transformer, you can specify the string ”drop” if you want the columns to be dropped, or you can specify ”passthrough” if you want the columns to be left untouched. By default, the remaining columns (i.e., the ones that were not listed) will be dropped, but you can set the remainder hyperparameter to any transformer (or to ”passthrough”) if you want these columns to be handled differently.

Since listing all the column names is not very convenient, Scikit-Learn provides a `make_column_selector()` function that returns a selector function you can use to automatically select all the features of a given type, such as numerical or categorical. You can pass this selector function to the ColumnTransformer instead of column names or indices. Moreover, if you don’t care about naming the transformers, you can use `make_column_transformer()`, which chooses the names for you, just like `make_pipeline()` does.

Once again this returns a NumPy array, but you can get the column names using `preprocessing.get_feature_names_out` and wrap the data in a nice DataFrame as we did before.

Warnings

The OneHotEncoder returns a sparse matrix and the num_pipeline returns a dense matrix. When there is such a mix of sparse and dense matrices, the ColumnTransformer estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold

(by default, `sparse_threshold=0.3`). In this example, it returns a dense matrix.

Let's recap what the pipeline will do and why:

- 大多数机器学习算法都不太喜欢缺失值，数值型的变量使用中位数进行impute，分类型变量使用频率最高的进行impute；
- 因为大多数机器学习算法只接收数值变量，因此分类变量会被进行one-hot编码；
- A few ratio features will be computed and added: `bedrooms_ratio`, `rooms_per_house`, and `people_per_house`. Hopefully these will better correlate with the median house value, and thereby help the ML models.
- A few cluster similarity features will also be added. These will likely be more useful to the model than latitude and longitude.
- Features with a long tail will be replaced by their logarithm, as most models prefer features with roughly uniform or Gaussian distributions.
- All numerical features will be standardized, as most ML algorithms prefer when all features have roughly the same scale.

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ['ratio']

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy='median'),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler()
    )

log_pipeline = make_pipeline(
    SimpleImputer(strategy='median'),
    FunctionTransformer(np.log, feature_names_out='one-to-one'),
    StandardScaler())

cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1, random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy='median'),
```

```
                                        StandardScaler())

preprocessing = ColumnTransformer([
    ('bedrooms', ratio_pipeline(), ['total_bedrooms', 'total_rooms']),
    ('rooms_per_house', ratio_pipeline(), ['total_rooms', 'households']),
    ('people_per_house', ratio_pipeline(), ['population', 'households']),
    ('log', log_pipeline, ['total_bedrooms', 'total_rooms', 'population',
                           'households', 'median_income']),
    ('geo', cluster_simil, ['latitude', 'longitude']),
    ('cat', cat_pipeline, make_column_selector(dtype_include=object))
],
                                    remainder=default_num_pipeline)
```

If you run this ColumnTransformer, it performs all the transformations and outputs a NumPy array with 24 features.

1.5 Select and Train a Model

1.5.1 Train and Evaluate on the Training Set

让我们从最基础的线性模型开始：

```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
```

```
housing_prediction = lin_reg.predict(housing)
print(housing_prediction[:5].round(-2))
# -2 = rounded to the nearest hundred
print(housing_labels.iloc[: 5].values)
```

Remember that you chose to use the RMSE as your performance measure, so you want to measure this regression model's RMSE on the whole training set using Scikit-Learn's `mean_squared_error()` function, with the squared argument set to False.

```
from sklearn.metrics import mean_squared_error
line_rmse = mean_squared_error(housing_labels,
                               housing_prediction,
```

```
                                    squared=False)
line_rmse
```

This is better than nothing, but clearly not a great score: the `median_housing_values` of most districts range between $120,000 and $265,000, so a typical prediction error of $68,628 is really 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. As we saw in the previous chapter, **the main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model**. This model is not regularized, which rules out the last option. You could try to add more features, but first you want to try a more complex model to see how it does.

我们决定使用更加复杂的决策树回归模型，这个模型更加强大，具有发现数据中非线性关系的的能力，更多见Chapter 3.

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)

housing_predictions = tree_reg.predict(housing)
tree_rmse = mean_squared_error(
    housing_labels,
    housing_predictions,
    squared=False
)
# 0
```

It is much more likely that the model has badly overfit the data. How can you be sure? As you 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**.

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

A great alternative is to use Scikit-Learn's *k-fold cross-validation* feature.(非常消耗算力)

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring='neg_root_mean_squared_error',
                              cv=10)
pd.Series(tree_rmses).describe()
```

Warnings

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 RMSE. It's a negative value, so you need to switch the sign of the output to get the RMSE scores.

In fact, it seems to perform almost as poorly as the linear regression model! 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). But cross-validation comes at the cost of training the model several times, so it is not always feasible.

Let's try one last model now: the RandomForestRegressor. As you will see in Chapter 4, random forests work by training many decision trees on random subsets of the features, then averaging out their predictions. Such models composed of many other models are called ensembles: they are capable of boosting the performance of the underlying model (in this case, decision trees).

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = - cross_val_score(forest_reg,
                                 housing,
                                 housing_labels,
                                 scoring='neg_root_mean_squared_error',
                                 cv=10)
pd.Series(forest_rmses).describe()
```

If you train a RandomForest and measure the RMSE on the training set, you will find roughly 17,474: that's much lower, meaning that there's still quite a lot of overfitting going on.(The training error is much lower than the validation error, which usually means that the model has overfit the training set.) Possible solutions are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into random forests, however, you should try out many other models from various categories of machine learning algorithms (e.g., several support vector machines with different kernels, and possibly a neural network), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising

models.(在进行调参之前，应该试试更多的模型，而不是一头扎进调参)

1.6 Fine-Tune Your Model

假设你已经有了一些想要建立的模型清单，你现在需要调整优化参数，来看一下怎么做。

1.6.1 Grid Search

最最不智能的方法就是在参数可行集上慢慢的测试，但这几乎是不可能的。Instead, you can use Scikit-Learn's `GridSearchCV` class to search for you. All you need to do is 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.

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ('preprocessing', preprocessing),
    ('random_forest', RandomForestRegressor(random_state=42))
])

param_grid = [
    {'preprocessing__geo__n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo__n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]

grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

Notice that you can refer to any hyperparameter of any estimator in a pipeline, even if this estimator is nested deep inside several pipelines and column transformers.

For example, when Scikit-Learn sees "`preprocessing__geo__n_clusters`", it splits this string at the double underscores, then it looks for an estimator named "`preprocessing`" in the pipeline and finds the preprocessing ColumnTransformer. Next, it looks for a transformer named "geo" inside this ColumnTransformer and finds the ClusterSimilarity transformer we used on the latitude and longitude attributes. Then it finds this transformer's `n_clusters` hyperparameter. Similarly, `random_forest__max_features` refers to the `max_features` hyperparameter of the estimator named "`random_forest`", which is of course the RandomForest model (the `max_features` hyperparameter will be explained in Chapter 4 Ensemble Learning and Random Forests).

Suggestions

Wrapping preprocessing steps in a Scikit-Learn pipeline allows you to tune the preprocessing hyperparameters along with the model hyperparameters. This is a good thing since they often interact. For example, perhaps increasing n_clusters requires increasing max_features as well. If fitting the pipeline transformers is computationally expensive, you can set the pipeline's memory hyperparameter to the path of a caching directory: when you first fit the pipeline, Scikit-Learn will save the fitted transformers to this directory. If you then fit the pipeline again with the same hyperparameters, Scikit-Learn will just load the cached transformers.

整个网格搜索会在两个字典指定的参数上共训练15（3×3和2×3）次，每次进行3次交叉验证。The best model is obtained by setting `n_clusters` to 15 and setting `max_features` to 6. 但是需要注意的是`n_clusters`达到了指定的最大值，你可能需要尝试更大的以获取更好的模型性能

```
grid_search.best_params_
```

You can access the best estimator using `grid_search.best_estimator_`. If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

The evaluation scores are available using `grid_search.cv_results_`, which is a dictionary.

```
cv_res = pd.DataFrame(grid_search.cv_results_)
cv_res.sort_values(by='mean_test_score', ascending=False, inplace=True)
cv_res = cv_res[['param_preprocessing__geo__n_clusters',
                 'param_random_forest__max_features',
                 'split0_test_score',
                 'split1_test_score',
                 'split2_test_score',
                 'mean_test_score']]
score_cols = ['split0', 'split1', 'split2', 'mean_test_rmse']
cv_res.columns = ['n_clusters', 'max_features'] + score_cols
cv_res[score_cols] = -cv_res[score_cols].round().astype(np.int64)
cv_res.head()
```

1.6.2 Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but `RandomizedSearchCV` is often preferable, especially when the hyperparameter search space is large. This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible

combinations it evaluates a fixed number of combinations, selecting a random value for each hyperparameter at every iteration. This may sound surprising, but this approach has several benefits:

- If some of your hyperparameters are continuous (or discrete but with many possible values), and you let randomized search run for, say, 1,000 iterations, then it will explore 1,000 different values for each of these hyperparameters, whereas grid search would only explore the few values you listed for each one.
- Suppose a hyperparameter does not actually make much difference, but you don't know it yet. If it has 10 possible values and you add it to your grid search, then training will take 10 times longer. But if you add it to a random search, it will not make any difference.(这个理由我似乎不太明白在讲什么)
- If there are 6 hyperparameters to explore, each with 10 possible values, then grid search offers no other choice than training the model a million times, whereas random search can always run for any number of iterations you choose.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {
    'preprocessing__geo__n_clusters': randint(low=3, high=50),
    'random_forest__max_features': randint(low=2, high=20)
}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distribs, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42
)

rnd_search.fit(housing, housing_labels)
```

Scikit-Learn also has `HalvingRandomSearchCV` and `HalvingGridSearchCV` hyperparameter search classes. Their goal is to use the computational resources more efficiently, either to train faster or to explore a larger hyperparameter space. Here's how they work: in the first round, many hyperparameter combinations (called "candidates") are generated using either the grid approach or the random approach. These candidates are then used to train models that are evaluated using cross-validation, as usual. However, training uses limited resources, which speeds up this first round considerably. By default, "limited resources" means that the models are trained on a small part of the training set. However, other limitations are possible, such as reducing the number of training iterations if the model has a hyperparameter to set it. Once every candidate has been evaluated, only the best ones go on to the second round, where they are allowed more resources to compete. After several rounds, the final candidates are evaluated using full resources. This may save you some time tuning hyperparameters.

在随机网格中，如何去选择超参数的采样分布？

- `scipy.stats.randint(a, b+1)`（随机整数）: for hyperparameters with discrete values that range from a to b, and all values in that range seem equally likely.
- `scipy.stats.uniform(a, b)`（均匀分布）: this is very similar, but for continuous hyperparameters.
- `scipy.stats.geom(1 / scale)`（几何分布）: for discrete values, when you want to sample roughly in a given scale. E.g., with scale=1000 most samples will be in this ballpark, but about 10% of all samples will be <100 and about 10% will be >2300. 就是几何分布中的参数p。
- `scipy.stats.expon(scale)`（指数分布）: this is the continuous equivalent of geom. Just set scale to the most likely value. 指数分布中的参数λ。
- `scipy.stats.reciprocal(a, b)`（倒数分布）: when you have almost no idea what the optimal hyperparameter value's scale is. If you set a=0.01 and b=100, then you're just as likely to sample a value between 0.01 and 0.1 as a value between 10 and 100.

$$f(x,a,b) = \frac{1}{xlog(b/a)}$$

Theorem 1 (密度变换公式) 设随机变量X有概率密度函数 $f(x), x \in (a,b)(a, b$ 可以为 $\infty)$, 而 $y = g(x)$ 在 $x \in (a,b)$ 上是严格单调的连续函数，存在唯一的反函数$x = y \in (\alpha,\beta)$ 并且 $h'(y)$ 存在且连续，那么 $Y = g(X)$ 也是连续型随机变量且有概率密度函数

$$p(y) = f(h(y))|h'(y)|,\ y \in (\alpha,\beta).$$

1.6.3 Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model. For example, you could train and fine-tune a k-nearest neighbors model, then create an ensemble model that just predicts the mean of the random forest prediction and that model's prediction. We will cover this topic in more detail in Chapter 4 Ensemble Learning and Random Forests.

1.6.4 Analyzing 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 the relative importance of each attribute for making accurate predictions:

```
final_model = rnd_search.best_estimator_
feature_importances = final_model['random_forest'].feature_importances_
feature_importances.round(2)

sorted(zip(feature_importances,
           final_model['preprocessing'].get_feature_names_out()),
       reverse=True)
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).

Suggestions

The sklearn.feature_selection.SelectFromModel transformer can automatically drop the least useful features for you: when you fit it, it trains a model (typically a random forest), looks at its fea ture_importances_ attribute, and selects the most useful features. Then when you call transform(), it drops the other features.

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem: adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.

1.6.5 Evaluate Your System on the Test Set

```
X_test = strat_test_set.drop('median_house_value', axis='columns')
y_test = strat_test_set['median_house_value'].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions, squared=False)
final_rmse
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch. 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 = .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)))
```

If you did a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation. That's because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets. When it happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

Now comes the project prelaunch phase: you need to present your solution (highlighting what you have learned, what worked and what did not, what assumptions were made, and what your system's limitations are),

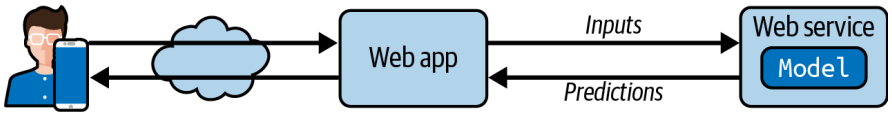

Figure 1.12: A model deployed as a web service and used by a web application

document everything, and create nice presentations with clear visualizations and easy-to-remember statements (e.g., “the median income is the number one predictor of housing prices”).

1.7 Launch, Monitor, and Maintain Your System

The most basic way to deploy your model to your production environment is just to save the best model you trained, transfer the file to your production environment, and load it. To save the model, you can use the joblib library like this:

Suggestions

It’s often a good idea to save every model you experiment with so that you can come back easily to any model you want. You may also save the cross-validation scores and perhaps the actual predictions on the validation set. This will allow you to easily compare scores across model types, and compare the types of errors they make.

Once your model is transferred to production, you can load it and use it. You must first import any custom classes and functions the model relies on (which means transferring the code to production), then load the model using joblib and use it to make predictions.

You can wrap the model within a dedicated web service that your web application can query through a REST API13[3] (see Figure 1.12).

But deployment is not the end of the story. You also need to write monitoring code to check your system’s live performance at regular intervals and trigger alerts when it drops.

So, you need to monitor your model’s live performance. But how do you do that? Well, it depends. In some cases, the model’s performance can be inferred from downstream metrics.

However, you may also need human analysis to assess the model’s performance.

Either way, you need to put in place a monitoring system (with or without human raters to evaluate the live model), as well as all the relevant processes to define what to do in case of failures and how to prepare for them. Unfortunately, this can be a lot of work. In fact, it is often much more work than building and training a model. 无论哪种方法，监控模型似乎要比训练模型更花费功夫。

If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should probably automate the whole process as much as possible. Here are a few things you can automate:

[3] In a nutshell, a REST (or RESTful) API is an HTTP-based API that follows some conventions, such as using standard HTTP verbs to read, update, create, or delete resources (GET, POST, PUT, and DELETE) and using JSON for the inputs and outputs.

- Collect fresh data regularly and label it (e.g., using human raters).
- Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.
- Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why). The script should probably test the performance of your model on various subsets of the test set, such as poor or rich districts, rural or urban districts, etc.

You should also make sure you evaluate the model's input data quality. Sometimes performance will degrade slightly because of a poor-quality signal (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale), but it may take a while before your system's performance degrades enough to trigger an alert. If you monitor your model's inputs, you may catch this earlier. 低质量的数据输入可能也会导致模型的性能变差。

Finally, make sure you keep backups of every model you create and have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly for some reason. Similarly, you should keep backups of every version of your datasets so that you can roll back to a previous dataset if the new one ever gets corrupted. Having backups of your datasets also allows you to evaluate any model against any previous dataset. 记得备份模型以及相关数据。

As you can see, machine learning involves quite a lot of infrastructure. Chapter 5 Training and Deploying TensorFlow Models at Scale discusses some aspects of this, but it's a very broad topic called *ML Operations(MLOps)*, which deserves its own book. 机器学习设计很多的基础设施，这里可能涉及到ML Operations。

1.8 Try It Out!

Hopefully this chapter gave you a good idea of what a machine learning project looks like as well as showing you some of the tools you can use to train a great system. As you can see, much of the work is in the data preparation step: building monitoring tools, setting up human evaluation pipelines, and automating regular model training. The machine learning algorithms are important, of course, but it is probably prefera ble to be comfortable with the overall process and know three or four algorithms well rather than to spend all your time exploring advanced algorithms.

A good place to start is on a competition website such as Kaggle: you will have a dataset to play with, a clear goal, and people to share the experience with. Have fun!

Chapter 2

Classification

Now we will turn our attention to classification systems.

2.1 MNIST

In this chapter we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the "hello world" of machine learning: whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST, and anyone who learns machine learning tackles this dataset sooner or later.

Scikit-Learn provides many helper functions to download popular datasets. The following code fetches the MNIST dataset from OpenML.org:

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', as_frame=False)
```

The sklearn.datasets package contains mostly three types of functions:

- `fetch_*` functions such as `fetch_openml()` to download real-life datasets;
- `load_*` functions to load small toy datasets bundled with Scikit-Learn (so they don't need to be downloaded over the internet);
- `make_*` functions to generate fake datasets, useful for tests;

Generated datasets are usually returned as an (X, y) tuple containing the input data and the targets, both as NumPy arrays. Other datasets are returned as sklearn.utils.Bunch objects, which are dictionaries whose entries can also be accessed as attributes. They generally contain the following entries:

- `DESCR`: A description of the dataset
- `data`: The input data, usually as a 2D NumPy array

Figure 2.1: Example of an MNIST image

- `target`: The labels, usually as a 1D NumPy array

The `fetch_openml()` function is a bit unusual since by default it returns the inputs as a Pandas DataFrame and the labels as a Pandas Series (unless the dataset is sparse). But the MNIST dataset contains images, and DataFrames aren't ideal for that, so it's preferable to set `as_frame=False` to get the data as NumPy arrays instead. Let's look at these arrays:

```
X, y = mnist.data, mnist.target
X.shape, y.shape
```

There are 70,000 images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). Let's take a peek at one digit from the dataset (Figure 2.1).

```
import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap='binary')
    plt.axis('off')

some_digit = X[0]
plot_digit(some_digit)
```

Figure 2.2: Digits from the MNIST dataset

To give you a feel for the complexity of the classification task, Figure 2.2 shows a few more images from the MNIST dataset.

在你进一步了解数据的时候，你应该将数据切分为训练集和测试集。The MNIST dataset returned by `fetch_openml()` is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images)[1]:

```
X_train, X_test, y_train, y_test = X[:60_000], X[60_000:], y[:60_000], y[60_000:]
```

2.2 Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5.

[1] Datasets returned by `fetch_openml()` are not always shuffled or split.

```
y_train_5 = (y_train == '5')
y_test_5 = (y_test == '5')
```

Now let's pick a classifier and train it. A good place to start is with a *stochastic gradient descent* (SGD, or stochastic GD) classifier, using Scikit-Learn's SGDClassifier class. This classifier is capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time, which also makes SGD well suited for online learning, as you will see later.

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Now we can use it to detect images of the number 5:

```
sgd_clf.predict([some_digit])
```

2.3 Performance Measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn a bunch of new concepts and acronyms!

2.3.1 Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in Chapter 1 End-to-End Machine Learning Project.

```
from sklearn.model_selection import cross_val_score

cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring='accuracy')
```

Wow! Above 95% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at **a dummy classifier that just classifies every single image in the most frequent class**, which in this case is the negative class (i.e., non 5):

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train)))
```

```
cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring='accuracy')
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is not a 5, you will be right about 90% of the time.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with skewed datasets (i.e., when some classes are much more frequent than others). A much better way to evaluate the performance of a classifier is to look at the *confusion matrix* (CM). 当数据是不平衡时，使用准确率accuracy来评价模型时不合理的。

Implementing Cross-Validation

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off the shelf. In these cases, you can implement cross-validation yourself. The following code does roughly the same thing as Scikit-Learn's `cross_val_score()` function, and it prints the same result.
The StratifiedKFold class performs stratified sampling (分层抽样) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))
```

2.3.2 Confusion Matrices

The general idea of a confusion matrix is to count the number of times instances of class A are classified as class B, for all A/B pairs.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets. You could make predictions on the test set, but it's best to keep that untouched for now. Instead, you can use the `cross_val_predict()` function:

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Just like the `cross_val_score()` function, `cross_val_predict()` performs k-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set (by "clean" I mean "out-of-sample": the model makes predictions on data that it never saw during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_train_5, y_train_pred)
cm
```

Each row in a confusion matrix represents an actual class, while each column represents a predicted class. The first row of this matrix considers non-5 images (the *negative class*): 53,892 of them were correctly classified as non-5s (they are called *true negatives*, TN), while the remaining 687 were wrongly classified as 5s (*false positives*, FP, also called *type I errors*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*, FN, also called *type II errors*), while the remaining 3,530 were correctly classified as 5s (*true positives*, TP).

Sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (Equation 2.1).

$$precision = \frac{TP}{TP + FP} \tag{2.1}$$

Precision is typically used along with another metric named *recall*, also called *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 2.2).

$$recall = \frac{TP}{TP + FN} \tag{2.2}$$

2.3.3 Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
from sklearn.metrics import precision_score, recall_score

print(precision_score(y_train_5, y_train_pred))
print(recall_score(y_train_5, y_train_pred))
```

Now our 5-detector does not look as shiny as it did when we looked at its accuracy. When it claims an image represents a 5, it is correct only 83.7% of the time. Moreover, it only detects 65.1% of the 5s.

It is often convenient to combine precision and recall into a single metric called the $F_1 score$, especially when you need a single metric to compare two classifiers. The $F_1 score$ is the harmonic mean(调和平均数，倒数平均数的倒数) of precision and recall (Equation 2.3). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{2.3}$$

To compute the $F_1 score$, simply call the `f1_score()` function:

```
from sklearn.metrics import f1_score

print(f1_score(y_train_5, y_train_pred))
```

The $F_1 score$ favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall.

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall trade-off*.

2.3.4 The Precision/Recall Trade-off

To understand this trade-off, let's look at how the SGDClassifier makes its classification decisions. For each instance, it computes a score based on a decision function. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class.

Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions. Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance, and then use any threshold you want to make predictions based on those scores:

```
y_scores = sgd_clf.decision_function([some_digit])
print(y_scores)
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

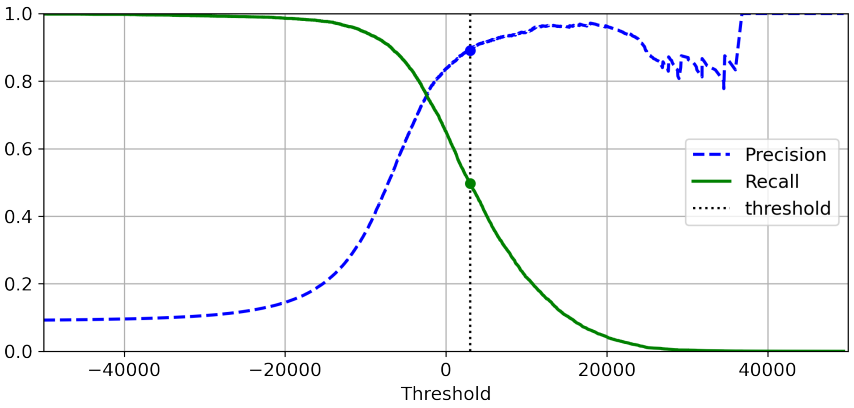

Figure 2.3: Precision and recall versus the decision threshold

The SGDClassifier uses a threshold equal to 0, so the preceding code returns the same result as the `predict()` method (i.e., True).(有些样本的得分确实会小于0) Let's raise the threshold:

```
threshold = 3_000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

This confirms that raising the threshold decreases recall.

那么如何确定阈值呢？First, use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf,
                             X_train,
                             y_train_5,
                             cv=3,
                             method='decision_function')
```

With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds (the function adds a last precision of 0 and a last recall of 1, corresponding to an infinite threshold):

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Finally, use Matplotlib to plot precision and recall as functions of the threshold value Figure 2.3.

Another way to select a good precision/recall trade-off is to plot precision directly against recall, as shown in Figure 2.4 (the same threshold is shown):

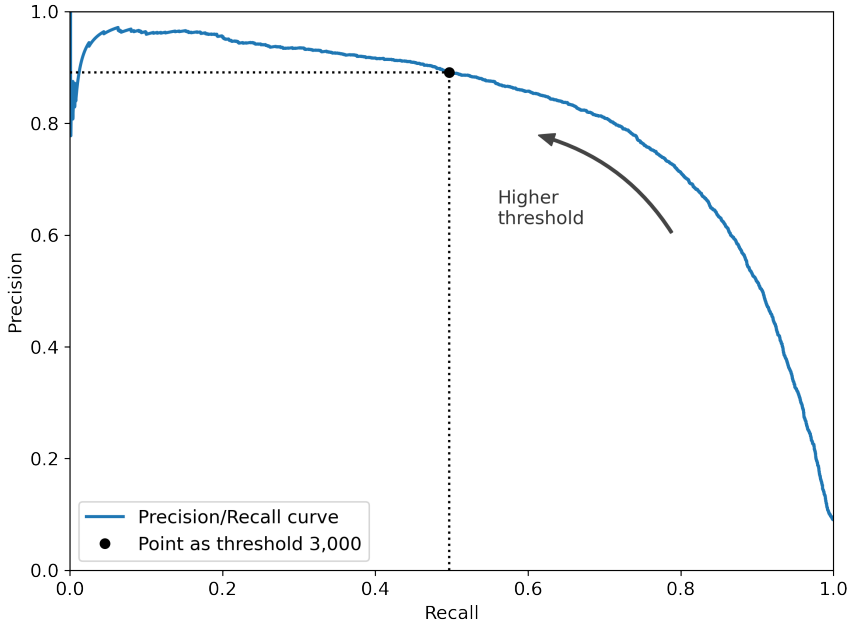

Figure 2.4: Precision versus recall

You can see that precision really starts to fall sharply at around 80% recall. You will probably want to select a precision/recall trade-off just before that drop—for example, at around 60% recall. But of course, the choice depends on your project.

Suppose you decide to aim for 90% precision. You could use the first plot to find the threshold you need to use, but that's not very precise. Alternatively, you can search for the lowest threshold that gives you at least 90% precision. For this, you can use the NumPy array's `argmax()` method. This returns the first index of the maximum value, which in this case means the first True value:

```
idx_for_90_precision = (precisions >= .9).argmax()
threshold_for_90_precision = thresholds[idx_for_90_precision]
threshold_for_90_precision # 3370.0194991439594
```

To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)

precision_score(y_train_5, y_train_pred_90) # 0.9000345901072293

recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)
recall_at_90_precision # 0.4799852425751706
```

As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high

enough threshold, and you're done. But wait, not so fast–a high-precision classifier is not very useful if its recall is too low!

Suggestions

If someone says, "Let's reach 99% precision", you should ask, "At what recall?"

2.3.5 The ROC Curve

2.4 Multiclass Classification

2.4.1

2.4.2

2.5

2.5.1

2.5.2

2.6

2.7

2.8

Chapter 3

Decision Trees

Chapter 4

Ensemble Learning and Random Forests

Chapter 5

Training and Deploying TensorFlow Models at Scale