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

目录

Part I

The Fundamentals of Machine Learning

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

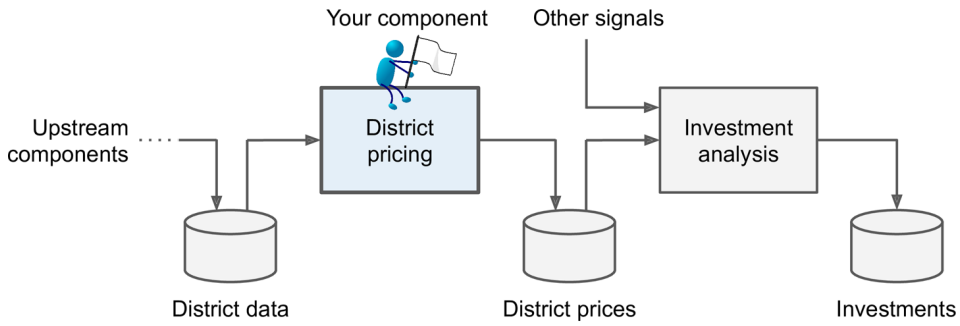

图 1.1: A Machine Learning pipeline for real estate investments

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

[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.

predictions, with a higher weight for large errors. Equation 1.1 shows the mathematical formula to compute the 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()
```

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

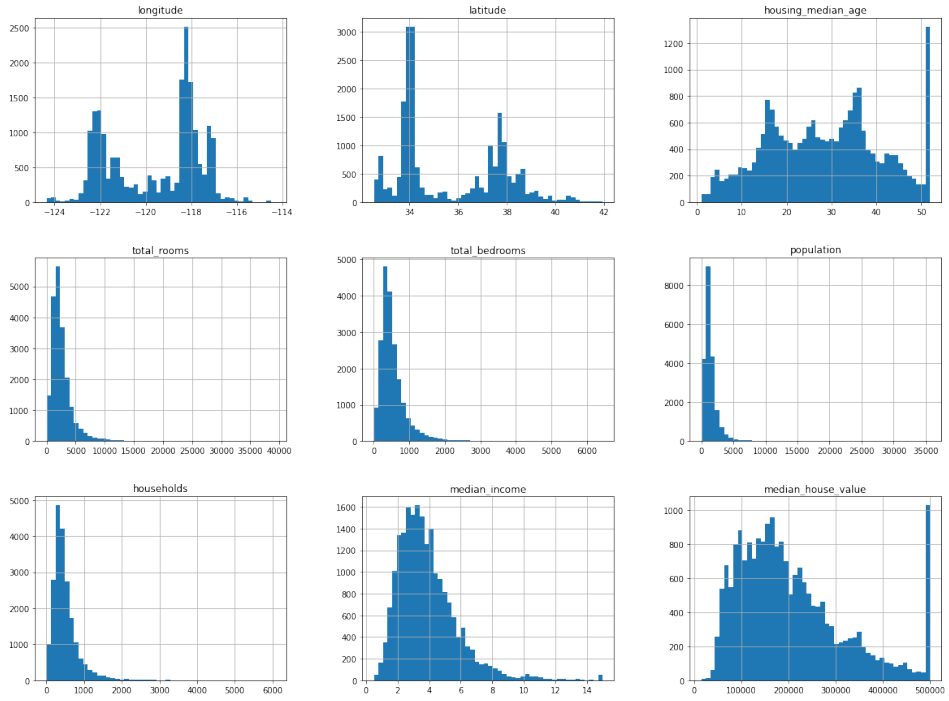

图 1.2: A histogram for each numerical attribute

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

[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.

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 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 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

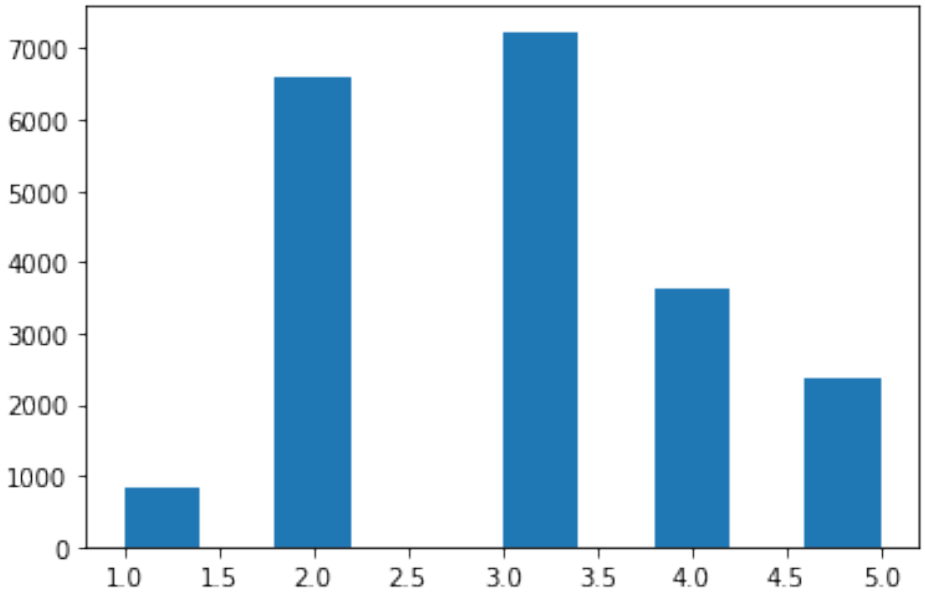

图 1.3: Histogram of income categories

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.

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()
```

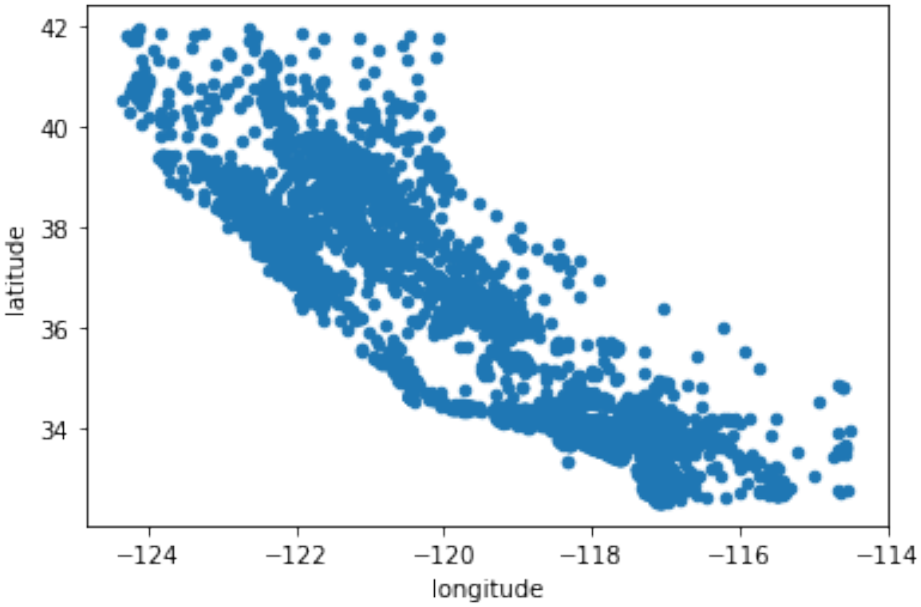

图 1.4: A geographical scatterplot of the data

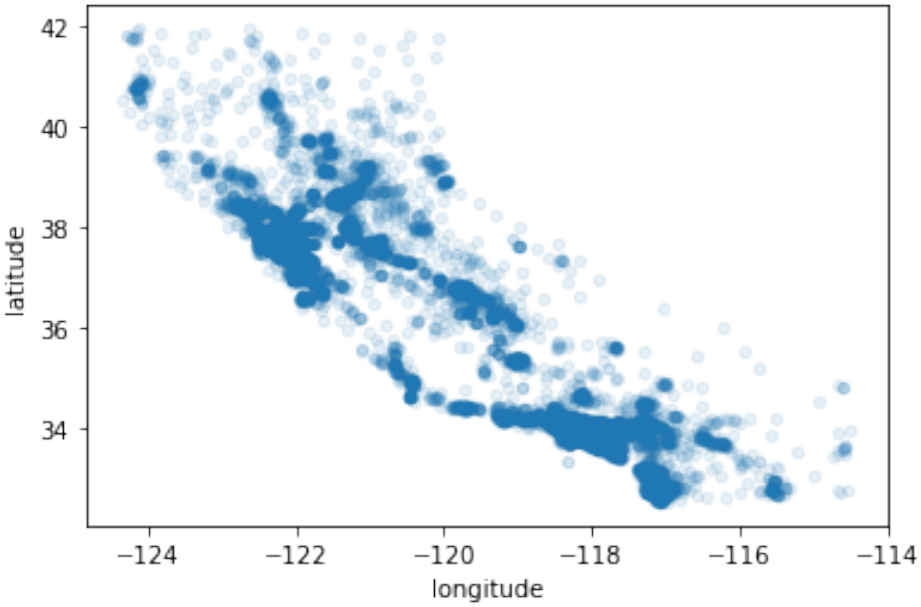

图 1.5: A better visualization that highlights high-density areas

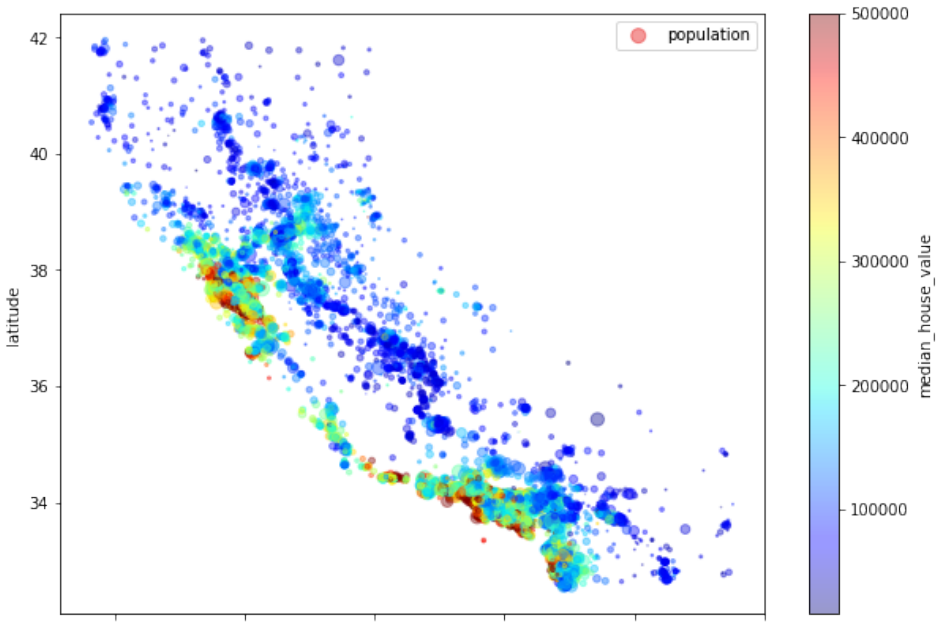

图 1.6: California housing prices

```
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.

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).

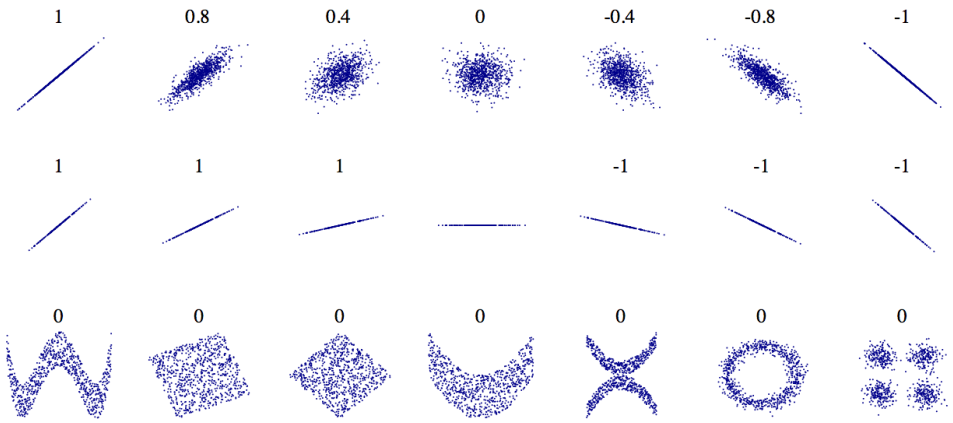

图 1.7: Standard correlation coefficient of various datasets

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

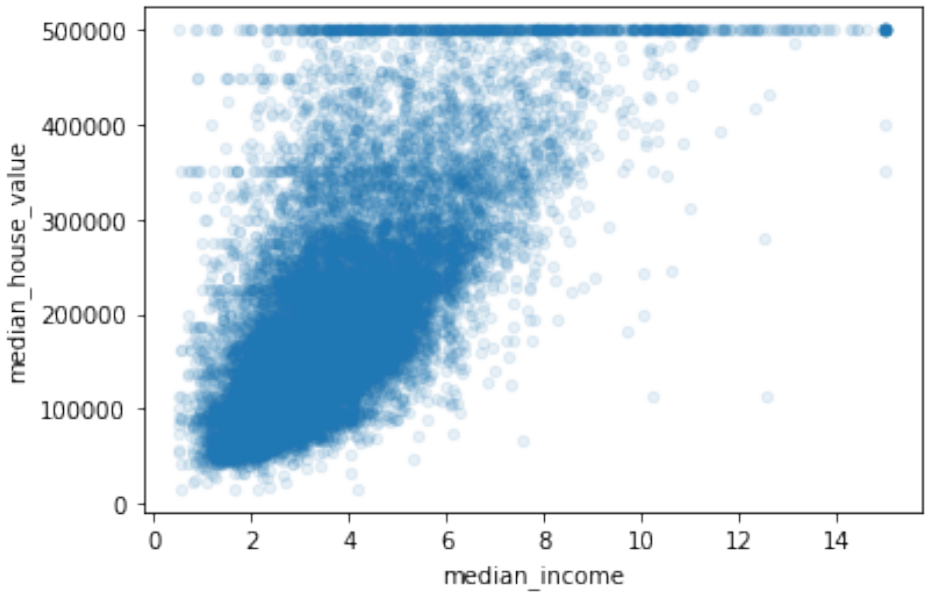

图 1.8: Median income versus median house value

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.

Tips: 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.

Tips: 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

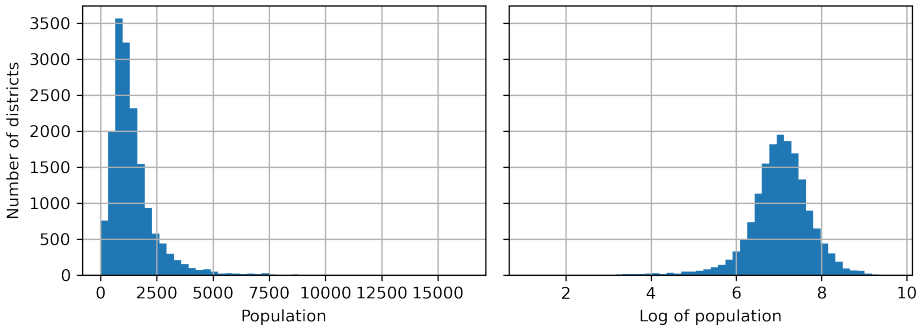

图 1.9: Transforming a feature to make it closer to a Gaussian distribution

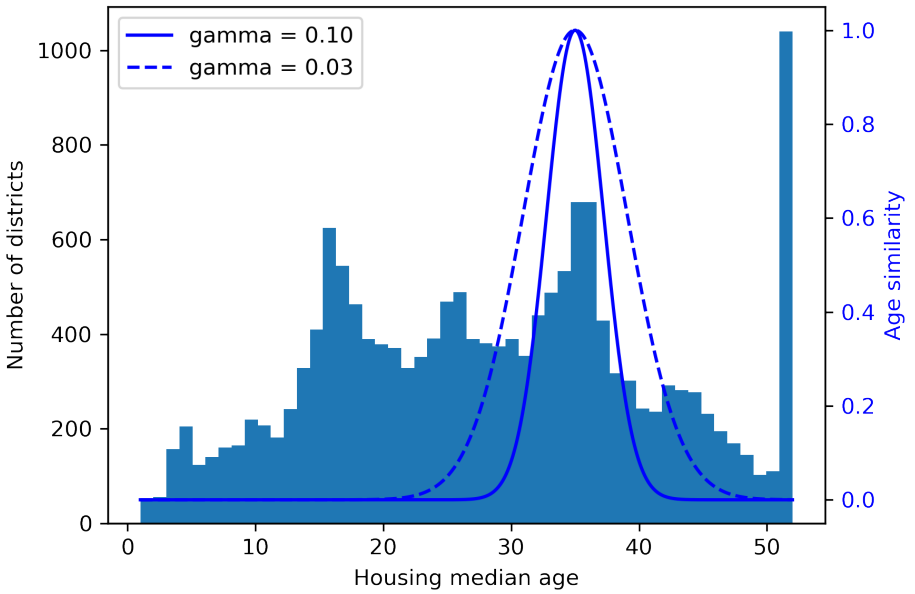

图 1.10: Gaussian RBF feature measuring the similarity

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.

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.

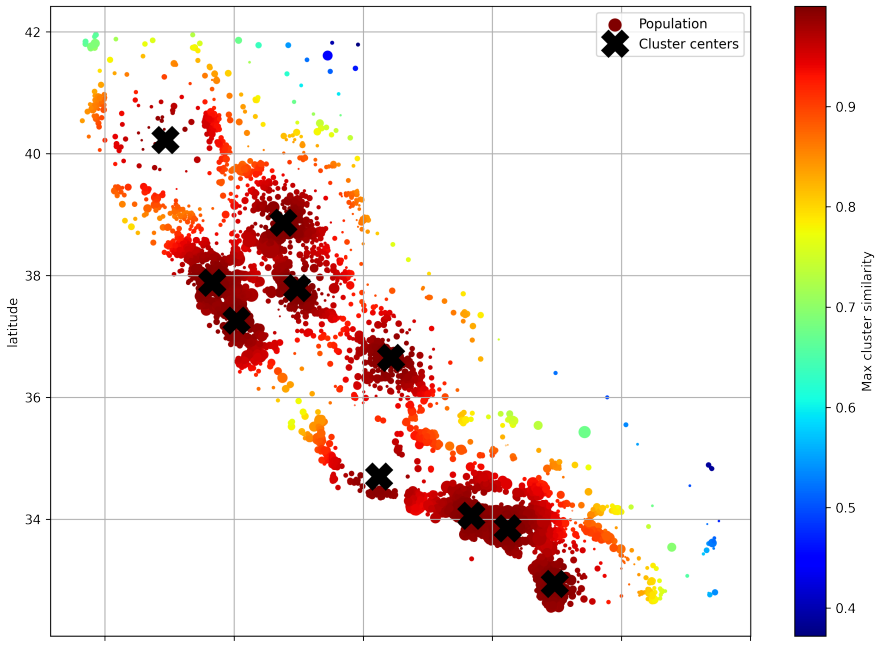

图 1.11: Gaussian RBF similarity to the nearest cluster center

- 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.