In [97]:
```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

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

# Common imports
import numpy as np
import os
import pandas as pd

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

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

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

In [98]:
```
# # Download the data
# import tarfile
# import urllib.request

# DOWNLOAD_ROOT = "https://github.com/ageron/handson-ml2/tree/master/"
# HOUSING_PATH = os.path.join("datasets", "housing")
# HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.csv"
```

```
# def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
#     if not os.path.isdir(housing_path):
#         os.makedirs(housing_path)
#     csv_path = os.path.join(HOUSING_PATH, "housing.csv")
#     urllib.request.urlretrieve(housing_url, csv_path)
# #     housing_tgz = tarfile.open(tgz_path)
# #     housing_tgz.extractall(path=housing_path)
# #     housing_tgz.close()
```

In [99]:
```
# Read Data
HOUSING_PATH = os.path.join("datasets", "housing")
csv_path = os.path.join(HOUSING_PATH, "housing.csv")
housing = pd.read_csv(csv_path)
housing.head()
```

Out[99]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | median_house_value | ocean_proximity |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | -122.23 | 37.88 | 41.0 | 880.0 | 129.0 | 322.0 | 126.0 | 8.3252 | 452600.0 | NEAR BAY |
| **1** | -122.22 | 37.86 | 21.0 | 7099.0 | 1106.0 | 2401.0 | 1138.0 | 8.3014 | 358500.0 | NEAR BAY |
| **2** | -122.24 | 37.85 | 52.0 | 1467.0 | 190.0 | 496.0 | 177.0 | 7.2574 | 352100.0 | NEAR BAY |
| **3** | -122.25 | 37.85 | 52.0 | 1274.0 | 235.0 | 558.0 | 219.0 | 5.6431 | 341300.0 | NEAR BAY |
| **4** | -122.25 | 37.85 | 52.0 | 1627.0 | 280.0 | 565.0 | 259.0 | 3.8462 | 342200.0 | NEAR BAY |

# Take a Quick Look at the Data Structure

In [100…
```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

In [101…

```
# Each row represents unique district. There are 20,640 districts.
# Notice that total_bedrooms attribute has only 20,433 non-null values, meaning that 207 districts are missing in this feature.
# There is once categorical variable. Let's look at what caegories exist and how many disticts belongs to eah category.
housing["ocean_proximity"].value_counts()
```

Out[101]:

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

In [102…

```
# Let's look at the summary of numerical attributes.
housing.describe().transpose()
```

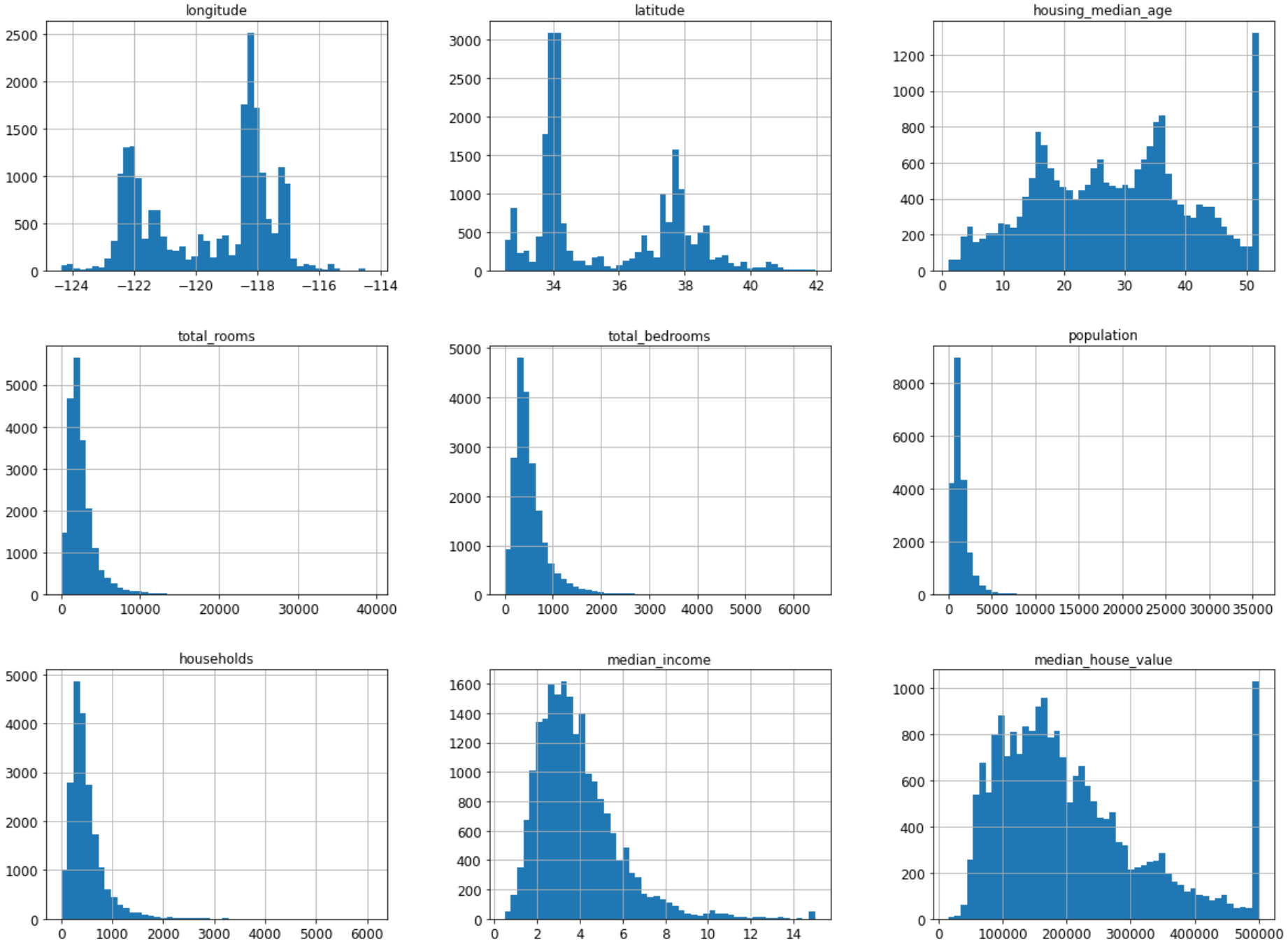
Out[102]:

| | count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|---|
| **longitude** | 20640.0 | -119.569704 | 2.003532 | -124.3500 | -121.8000 | -118.4900 | -118.01000 | -114.3100 |
| **latitude** | 20640.0 | 35.631861 | 2.135952 | 32.5400 | 33.9300 | 34.2600 | 37.71000 | 41.9500 |
| **housing_median_age** | 20640.0 | 28.639486 | 12.585558 | 1.0000 | 18.0000 | 29.0000 | 37.00000 | 52.0000 |
| **total_rooms** | 20640.0 | 2635.763081 | 2181.615252 | 2.0000 | 1447.7500 | 2127.0000 | 3148.00000 | 39320.0000 |
| **total_bedrooms** | 20433.0 | 537.870553 | 421.385070 | 1.0000 | 296.0000 | 435.0000 | 647.00000 | 6445.0000 |
| **population** | 20640.0 | 1425.476744 | 1132.462122 | 3.0000 | 787.0000 | 1166.0000 | 1725.00000 | 35682.0000 |
| **households** | 20640.0 | 499.539680 | 382.329753 | 1.0000 | 280.0000 | 409.0000 | 605.00000 | 6082.0000 |
| **median_income** | 20640.0 | 3.870671 | 1.899822 | 0.4999 | 2.5634 | 3.5348 | 4.74325 | 15.0001 |
| **median_house_value** | 20640.0 | 206855.816909 | 115395.615874 | 14999.0000 | 119600.0000 | 179700.0000 | 264725.00000 | 500001.0000 |

The 25%, 50%, and 75% rows show the corresponding percentiles: a percentile indi- cates the value below which a given percentage of observations in a group of observa- tions falls. For example, 25% of the districts have a housing_median_age lower than 18, while 50% are lower than 29 and 75% are lower than 37. These are often called the 25th percentile (or 1st quartile), the median, and the 75th percentile (or 3rd quartile).

In [103…

```
# Let's look at the number of instances for a given value range, inshort historgram.
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
housing.hist(bins=50,figsize=(20,15))
plt.show()
```


longitude
latitude
housing_median_age
total_rooms
total_bedrooms
population
households
median_income
median_house_value

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.

For example, You can see that slightly over 1000 districts have a median_house_value equal to about $500,000

1. First, the median income attribute does not look like it is expressed in US dollars (USD). Data has been scaled here and the number represents tens of thousands of dollars.

1. The housing median age and the median house value You re also capped.

1. 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. You will try transforming these attributes later on to have more bell-shaped distributions.

# Data Snooping Bias

It may sound strange to voluntarily set aside part of the data at this stage. But 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 notperform as You ll as expected. This is called data snooping bias.

# Select the Test Set

In [104…
```
# to make this notebook's output identical at every run
np.random.seed(42)
```

In [105…
```
import numpy as np

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

```
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

In [106…
```
train_set, test_set = split_train_test(housing, 0.2)
len(train_set)
```

Out[106]:
```
16512
```

In [107…
```
len(test_set)
```

Out[107]:
```
4128
```

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:

In [108…
```
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]
```

In [109…
```
housing_with_id = housing.reset_index()   # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

In [110…
```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

In [111…
```
test_set.head()
```

Out[111]:

| | index | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | median_house_value | ocean_ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **59** | 59 | -122.29 | 37.82 | 2.0 | 158.0 | 43.0 | 94.0 | 57.0 | 2.5625 | 60000.0 | |
| **60** | 60 | -122.29 | 37.83 | 52.0 | 1121.0 | 211.0 | 554.0 | 187.0 | 3.3929 | 75700.0 | |
| **61** | 61 | -122.29 | 37.82 | 49.0 | 135.0 | 29.0 | 86.0 | 23.0 | 6.1183 | 75000.0 | |
| **62** | 62 | -122.29 | 37.81 | 50.0 | 760.0 | 190.0 | 377.0 | 122.0 | 0.9011 | 86100.0 | |
| **67** | 67 | -122.29 | 37.80 | 52.0 | 1027.0 | 244.0 | 492.0 | 147.0 | 2.6094 | 81300.0 | |

In [112…
```
len(test_set)
```

Out[112]:
```
4318
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is train_test_split, which does pretty much the same thing as the function split_train_test defined earlier, with a couple of additional features. First there is a random_state parameter that allows You to set the random generator seed as explained previously, and second You can pass it multiple datasets with an identical number of rows, and it will split them on the same indices.

In [113…
```
from sklearn.model_selection import train_test_split

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

In [114…
```
test_set.head()
```

Out[114]:

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

In [115…
```
len(test_set)
```

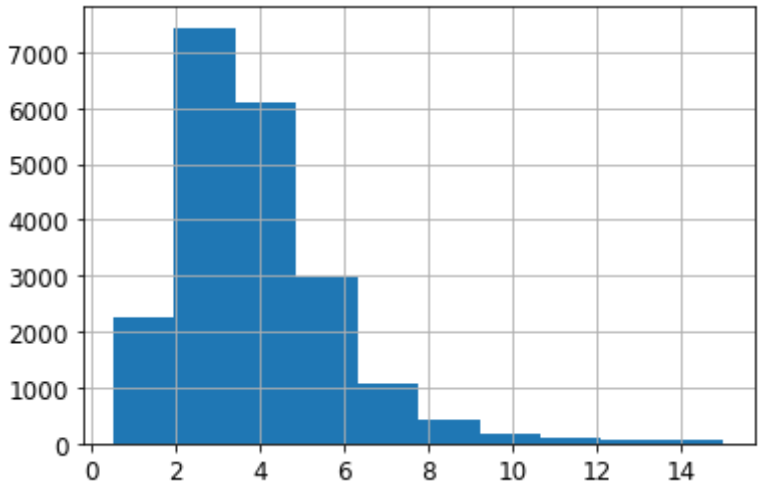
Out[115]:
```
4128
```

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.

# Avoid Sampling Bias

In [116…
```
housing["median_income"].hist()
```

Out[116]:
```
<AxesSubplot:>
```




most median income values are clustered around 1.5 to 6 (i.e., $15,000–60,000), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum

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

In [117…
```
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])
```

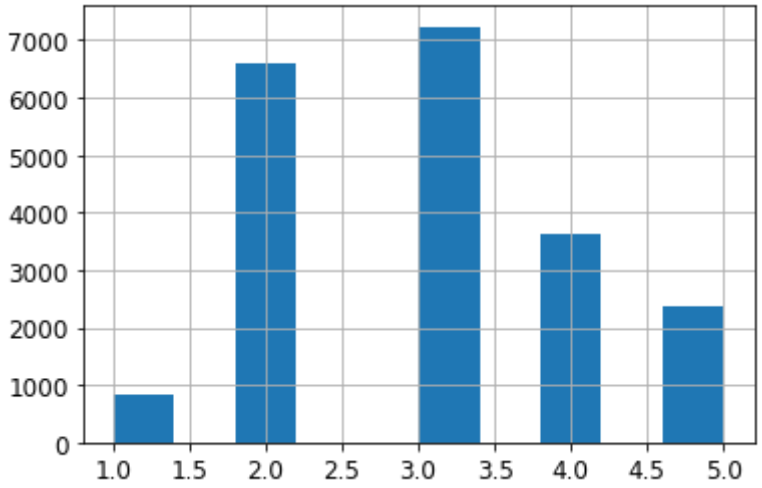
This means that You should not have too many strata, and each stratum should be large enough.

In [118…
```
housing["income_cat"].value_counts()
```

Out[118]:
```
3    7236
2    6581
4    3639
5    2362
1     822
Name: income_cat, dtype: int64
```

In [119…
```
housing["income_cat"].hist()
```

Out[119]:
```
<AxesSubplot:>
```




Now You are ready to do stratified sampling based on the income category. For this You can use Scikit-Learn's StratifiedShuffleSplit class:

In [120…
```
from sklearn.model_selection import StratifiedShuffleSplit

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

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

In [121…
```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

Out[121]:
```
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: income_cat, dtype: float64
```

In [122…
```
housing["income_cat"].value_counts() / len(housing)
```

Out[122]:
```
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64
```

In [123…
```
def income_cat_proportions(data):
    return data["income_cat"].value_counts() / len(data)

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

compare_props = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_cat_proportions(test_set),
}).sort_index()
compare_props["Rand. %error"] = 100 * compare_props["Random"] / compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] / compare_props["Overall"] - 100
```

Let's compare the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling. As You can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is quite skeYou d.

In [124…
```
compare_props
```

Out[124]:

| | Overall | Stratified | Random | Rand. %error | Strat. %error |
|---|---|---|---|---|---|
| **1** | 0.039826 | 0.039971 | 0.040213 | 0.973236 | 0.364964 |
| **2** | 0.318847 | 0.318798 | 0.324370 | 1.732260 | -0.015195 |
| **3** | 0.350581 | 0.350533 | 0.358527 | 2.266446 | -0.013820 |
| **4** | 0.176308 | 0.176357 | 0.167393 | -5.056334 | 0.027480 |
| **5** | 0.114438 | 0.114341 | 0.109496 | -4.318374 | -0.084674 |

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

In [125…
```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

You 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.

# Discover and Visualize the Data to Gain Insights

Let's create a copy so You can play with it without harming the training set

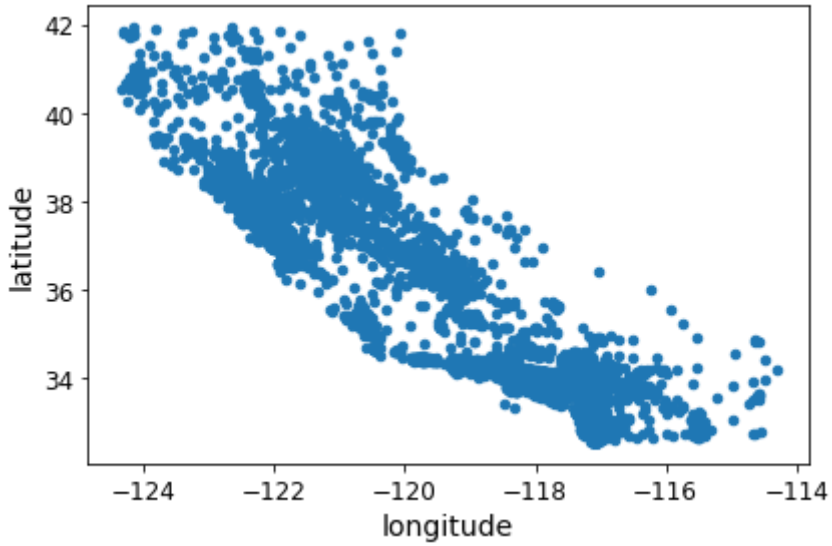
In [126…
```
housing = strat_train_set.copy()
```

# Visualize 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

In [127…
```
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

```
Saving figure bad_visualization_plot
```

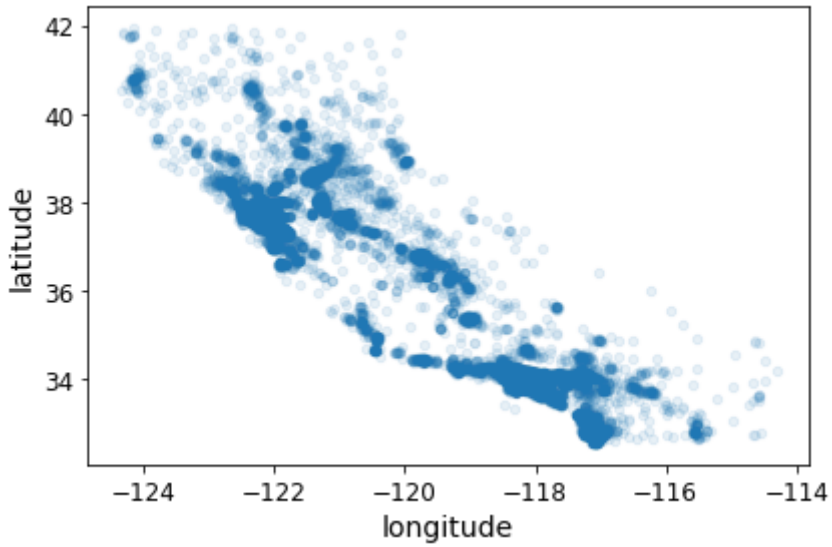



This looks like California all right, but other than that it is hard to see any particular pattern. Setting the alpha option to 0.1 makes it much easier to visualize the places where there is a high density of data points

In [128…

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

```
Saving figure better_visualization_plot
```



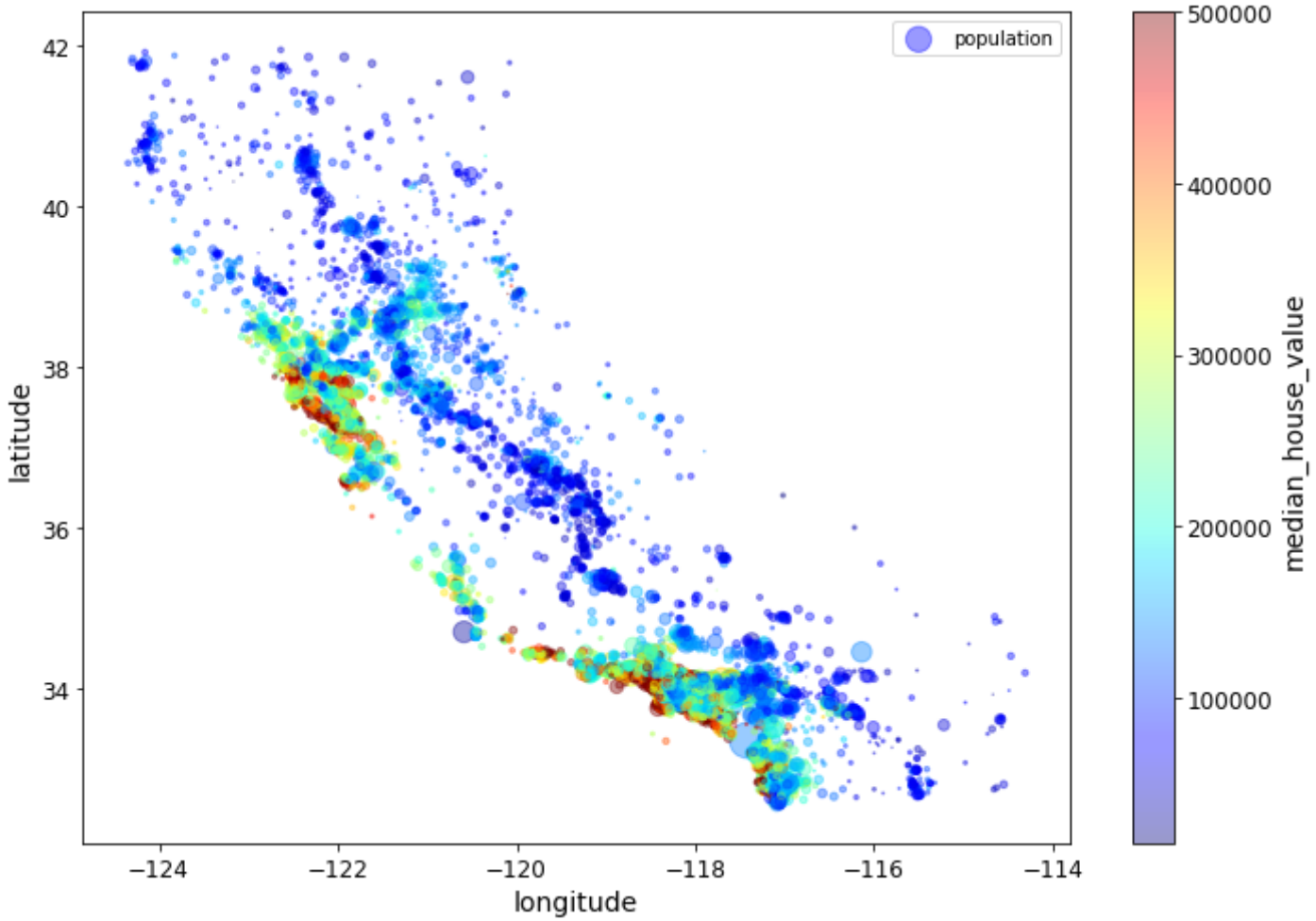
Now that's much better: You can clearly see the high-density areas, namely the Bay Area and around Los Angeles and San Diego, plus a long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno

Now let's look at the housing prices. The radius of each circle represents the district's population (option s), and the color represents the price (option c). You will use a predefined color map (option cmap) called jet, which ranges from blue (low values) to red (high prices)

In [129…

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
             s=housing["population"]/100, label="population", figsize=(10,7),
             c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
             sharex=False)
plt.legend()
save_fig("housing_prices_scatterplot")
```

```
Saving figure housing_prices_scatterplot
```




# Let's look at the correlations

Since the dataset is not too large, You can easily compute the standard correlation coe□cient (also called Pearson's r) betYou en every pair of attributes using the corr() method:

In [130…

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

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

In [131…
```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[131]:
```
median_house_value    1.000000
median_income         0.687151
total_rooms           0.135140
housing_median_age    0.114146
households            0.064590
total_bedrooms        0.047781
population           -0.026882
longitude            -0.047466
latitude             -0.142673
Name: median_house_value, dtype: float64
```

The correlation coefficient ranges from –1 to 1. When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up. When the coefficient is close to –1, it means that there is a strong negative correlation; You can see a small negative correlation betYou en the latitude and the median house value (i.e., prices have a slight tendency to go down when You go north). Finally, coefficients close to zero mean that there is no linear correlation

Another way to check for correlation betYou en attributes is to use Pandas' scatter_matrix function, which plots every numerical attribute against every other

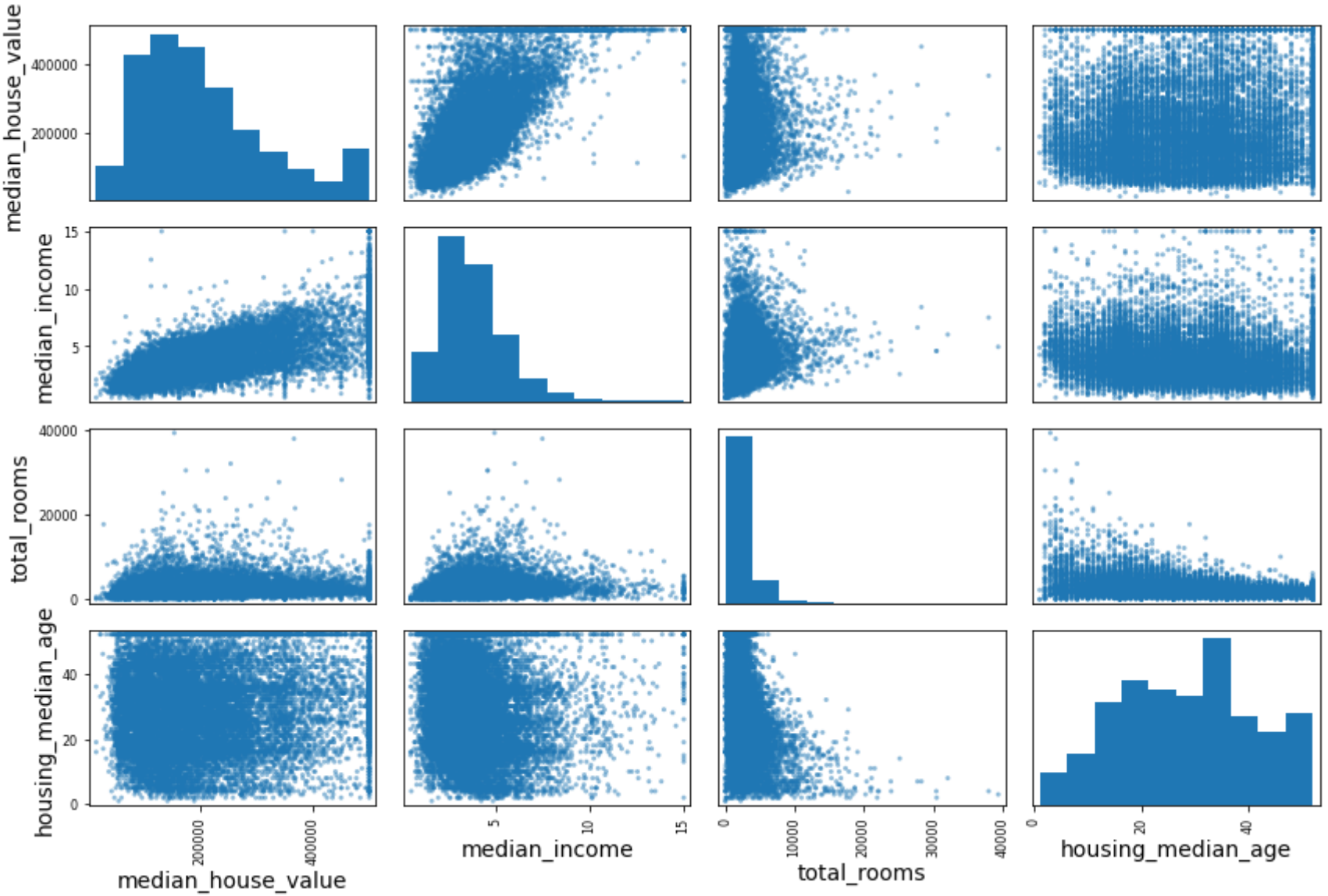
# numerical attribute. Since there are now 11 numerical attributes, You would get 11^2

121 plots, which would not fit on a page, so let's just focus on a few promising attributes that seem most correlated with the median housing value

In [132…
```
# from pandas.tools.plotting import scatter_matrix # For older versions of Pandas
from pandas.plotting import scatter_matrix

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

```
Saving figure scatter_matrix_plot
```



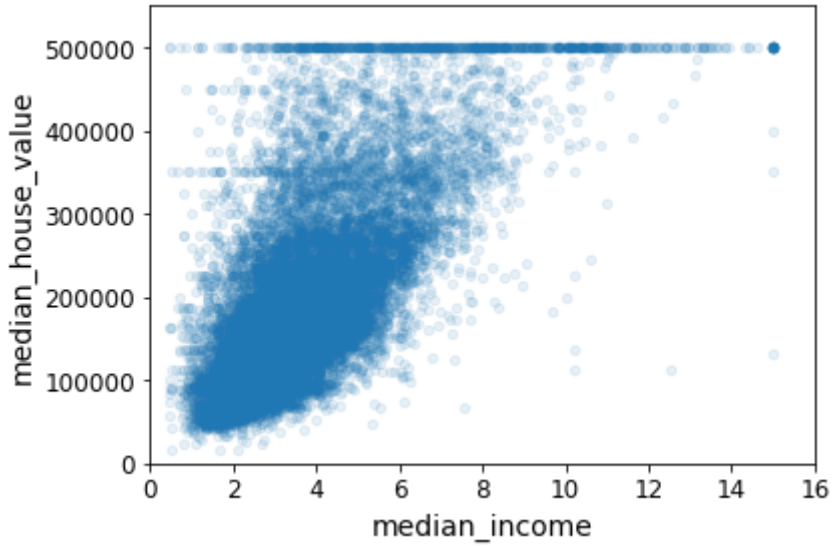

Let's zoom in on the most promising attribute's correlation

In [133…

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

```
plt.axis([0, 16, 0, 550000])
save_fig("income_vs_house_value_scatterplot")
```

```
Saving figure income_vs_house_value_scatterplot
```




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 You 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

# Learning So far...

1. You identified a few data quirks that You may want to clean up before feeding the data to a Machine Learning algorithm
2. You found interesting correlations betYou en attributes, in particular with the target attribute.
3. You also noticed that some attributes have a tail-heavy distribution, so You may want to trans- form them (e.g., by computing their logarithm).

One last thing you may want to do before actually preparing the data for Machine Learning algorithms is to try out various attribute combinations.

# Experimenting with Attribute Combinations

The total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. Let's create these new attributes:

In [134…
```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

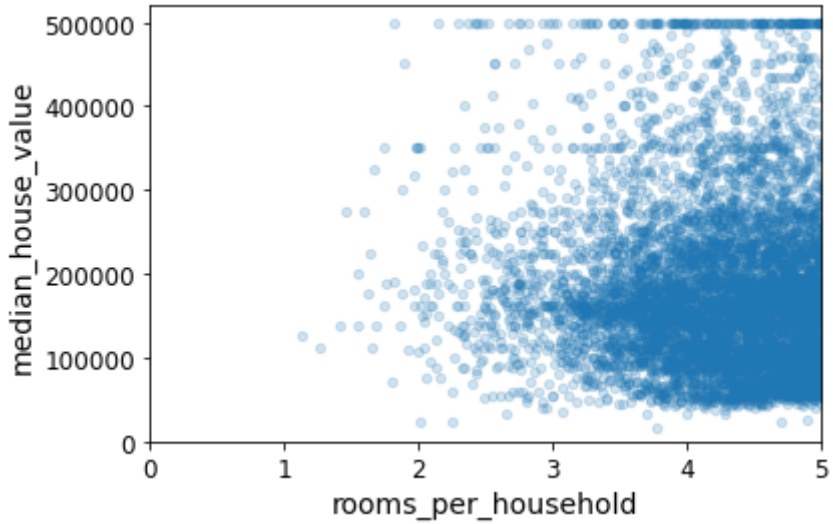
And now let's look at the correlation matrix again:

In [135…
```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

Out[135]:
```
median_house_value          1.000000
median_income               0.687151
rooms_per_household         0.146255
total_rooms                 0.135140
housing_median_age          0.114146
households                  0.064590
total_bedrooms              0.047781
population_per_household   -0.021991
population                 -0.026882
longitude                  -0.047466
latitude                   -0.142673
bedrooms_per_room          -0.259952
Name: median_house_value, dtype: float64
```

Hey, not bad! The new bedrooms_per_room attribute is much more correlated with the median house value than the total number of rooms or bedrooms. Apparently houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

In [136…
```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```




# Prepare the Data for Machine Learning Algorithms

First let's revert to a clean training set (by copying strat_train_set once again), and let's separate the predictors and the labels since we don't necessarily want to apply the same transformations to the predictors and the target values (note that drop() creates a copy of the data and does not affect strat_train_set):irst let's revert to a clean training set (by copying strat_train_set once again), and let's separate the predictors and the labels since we don't necessarily want to apply the same transformations to the predictors and the target values (note that drop() creates a copy of the data and does not affect strat_train_set):

In [137…
```
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels for training set
housing_labels = strat_train_set["median_house_value"].copy()
```

# Data Cleaning

Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them. You noticed earlier that the total_bedrooms attribute has some missing values, so let's fix this. You have three options:

• Get rid of the corresponding districts.

• Get rid of the whole attribute.

• Set the values to some value (zero, the mean, the median, etc.)

To demonstrate each of them, let's create a copy of the housing dataset, but keeping only the rows that contain at least one null. Then it will be easier to visualize exactly what each option does:

In [138…
```
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

Out[138]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | ocean_proximity |
|---|---|---|---|---|---|---|---|---|---|
| **1606** | -122.08 | 37.88 | 26.0 | 2947.0 | NaN | 825.0 | 626.0 | 2.9330 | NEAR BAY |
| **10915** | -117.87 | 33.73 | 45.0 | 2264.0 | NaN | 1970.0 | 499.0 | 3.4193 | <1H OCEAN |
| **19150** | -122.70 | 38.35 | 14.0 | 2313.0 | NaN | 954.0 | 397.0 | 3.7813 | <1H OCEAN |
| **4186** | -118.23 | 34.13 | 48.0 | 1308.0 | NaN | 835.0 | 294.0 | 4.2891 | <1H OCEAN |
| **16885** | -122.40 | 37.58 | 26.0 | 3281.0 | NaN | 1145.0 | 480.0 | 6.3580 | NEAR OCEAN |

In [139…
```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

Out[139]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | ocean_proximity |
|---|---|---|---|---|---|---|---|---|---|

In [140…
```
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

Out[140]:

| | longitude | latitude | housing_median_age | total_rooms | population | households | median_income | ocean_proximity |
|---|---|---|---|---|---|---|---|---|
| **1606** | -122.08 | 37.88 | 26.0 | 2947.0 | 825.0 | 626.0 | 2.9330 | NEAR BAY |
| **10915** | -117.87 | 33.73 | 45.0 | 2264.0 | 1970.0 | 499.0 | 3.4193 | <1H OCEAN |
| **19150** | -122.70 | 38.35 | 14.0 | 2313.0 | 954.0 | 397.0 | 3.7813 | <1H OCEAN |
| **4186** | -118.23 | 34.13 | 48.0 | 1308.0 | 835.0 | 294.0 | 4.2891 | <1H OCEAN |
| **16885** | -122.40 | 37.58 | 26.0 | 3281.0 | 1145.0 | 480.0 | 6.3580 | NEAR OCEAN |

In [141…
```
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
```

In [142…
```
sample_incomplete_rows
```

Out[142]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | ocean_proximity |
|---|---|---|---|---|---|---|---|---|---|
| **1606** | -122.08 | 37.88 | 26.0 | 2947.0 | 433.0 | 825.0 | 626.0 | 2.9330 | NEAR BAY |
| **10915** | -117.87 | 33.73 | 45.0 | 2264.0 | 433.0 | 1970.0 | 499.0 | 3.4193 | <1H OCEAN |
| **19150** | -122.70 | 38.35 | 14.0 | 2313.0 | 433.0 | 954.0 | 397.0 | 3.7813 | <1H OCEAN |
| **4186** | -118.23 | 34.13 | 48.0 | 1308.0 | 433.0 | 835.0 | 294.0 | 4.2891 | <1H OCEAN |
| **16885** | -122.40 | 37.58 | 26.0 | 3281.0 | 433.0 | 1145.0 | 480.0 | 6.3580 | NEAR OCEAN |

If you choose option 3, you should compute the median value on the training set, and use it to fill the missing values in the training set, but also don't forget to save the median value that you have computed. You will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data

Scikit-Learn provides a handy class to take care of missing values: SimpleImputer. Here is how to use it. First, you need to create a SimpleImputer instance, specifying that you want to replace each attribute's missing values with the median of that attribute

In [143…
```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

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

In [144…
```
housing_num = housing.drop("ocean_proximity", axis=1)
```

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

In [145…
```
imputer.fit(housing_num)
```

Out[145]:
```
SimpleImputer(strategy='median')
```

The imputer has simply computed the median of each attribute and stored the result in its statistics_ instance variable.

Only the total_bedrooms attribute had missing values, but we cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes:

In [146…
```
imputer.statistics_
```

Out[146]:
```
array([-118.51   ,   34.26   ,   29.     , 2119.     ,  433.     ,
       1164.     ,  408.     ,    3.54155])
```

Check that this is the same as manually computing the median of each attribute:

In [147…
```
housing_num.median().values
```

Out[147]:
```
array([-118.51   ,   34.26   ,   29.     , 2119.     ,  433.     ,
       1164.     ,  408.     ,    3.54155])
```

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

In [148…
```
X = imputer.transform(housing_num)
```

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

In [149…
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing.index)
```

In [150…
```
housing_tr.loc[sample_incomplete_rows.index.values]
```

Out[150]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income |
|---|---|---|---|---|---|---|---|---|
| **1606** | -122.08 | 37.88 | 26.0 | 2947.0 | 433.0 | 825.0 | 626.0 | 2.9330 |
| **10915** | -117.87 | 33.73 | 45.0 | 2264.0 | 433.0 | 1970.0 | 499.0 | 3.4193 |
| **19150** | -122.70 | 38.35 | 14.0 | 2313.0 | 433.0 | 954.0 | 397.0 | 3.7813 |
| **4186** | -118.23 | 34.13 | 48.0 | 1308.0 | 433.0 | 835.0 | 294.0 | 4.2891 |
| **16885** | -122.40 | 37.58 | 26.0 | 3281.0 | 433.0 | 1145.0 | 480.0 | 6.3580 |

# Handling Text and Categorical Attributes

Earlier we left out the categorical attribute ocean_proximity because it is a text attribute so we cannot compute its median:

Now let's preprocess the categorical input feature, ocean_proximity:

In [151…
```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(10)
```

Out[151]:

| | ocean_proximity |
|---|---|
| **12655** | INLAND |
| **15502** | NEAR OCEAN |
| **2908** | INLAND |
| **14053** | NEAR OCEAN |
| **20496** | <1H OCEAN |
| **1481** | NEAR BAY |
| **18125** | <1H OCEAN |
| **5830** | <1H OCEAN |
| **17989** | <1H OCEAN |
| **4861** | <1H OCEAN |

In [152…
```
housing["ocean_proximity"].unique()
```

Out[152]:
```
array(['INLAND', 'NEAR OCEAN', '<1H OCEAN', 'NEAR BAY', 'ISLAND'],
      dtype=object)
```

Most Machine Learning algorithms prefer to work with numbers anyway, so let’s convert these categories from text to numbers. For this, we can use Scikit-Learn’s Ordina lEncoder class

In [153…
```
from sklearn.preprocessing import OrdinalEncoder
```

```
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

Out[153]:
```
array([[1.],
       [4.],
       [1.],
       [4.],
       [0.],
       [3.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

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):

In [154…
```
ordinal_encoder.categories_
```

Out[154]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

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”, “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 OneHotEn coder class to convert categorical values into one-hot vectors

In [155…
```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[155]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

Notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one hot encoding we get a matrix with thousands of columns, and the matrix is full of zeros 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 non-zero 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:

In [156…
```
housing_cat_1hot.toarray()
```

Out[156]:
```
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0.],
       ...,
       [1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])
```

Alternatively, you can set sparse=False when creating the OneHotEncoder

In [157…
```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[157]:
```
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0.],
       ...,
       [1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])
```

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

In [158…
```
cat_encoder.categories_
```

Out[158]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

# Custom Transformers

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

Let's create a custom transformer to add extra attributes:

In [159…

```
from sklearn.base import BaseEstimator, TransformerMixin

# column index
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

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

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

In this example the transformer has one hyperparameter, add_bedrooms_per_room, set to True by default (it is often helpful to provide sensible defaults). This hyperparameter will allow you to easily find out whether adding this attribute helps the Machine Learning algorithms or not. More generally, you can add a hyperparameter to gate any data preparation step that you are not 100% sure about. The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely that you will find a great combination (and saving you a lot of time).

Note that I hard coded the indices (3, 4, 5, 6) for concision and clarity, but it would be much cleaner to get them dynamically, like this:

In [160…
```
col_names = "total_rooms", "total_bedrooms", "population", "households"
rooms_ix, bedrooms_ix, population_ix, households_ix = [
    housing.columns.get_loc(c) for c in col_names] # get the column indices
```

Also, housing_extra_attribs is a NumPy array, we've lost the column names (unfortunately, that's a problem with Scikit-Learn). To recover a DataFrame, you could run this:

In [161…
```
housing_extra_attribs = pd.DataFrame(
    housing_extra_attribs,
    columns=list(housing.columns)+["rooms_per_household", "population_per_household"],
    index=housing.index)
housing_extra_attribs.head()
```

Out[161]:

| | longitude | latitude | housing_median_age | total_rooms | total_bedrooms | population | households | median_income | ocean_proximity | rooms_per_hou |
|---|---|---|---|---|---|---|---|---|---|---|
| **12655** | -121.46 | 38.52 | 29.0 | 3873.0 | 797.0 | 2237.0 | 706.0 | 2.1736 | INLAND | 5. |
| **15502** | -117.23 | 33.09 | 7.0 | 5320.0 | 855.0 | 2015.0 | 768.0 | 6.3373 | NEAR OCEAN | 6. |
| **2908** | -119.04 | 35.37 | 44.0 | 1618.0 | 310.0 | 667.0 | 300.0 | 2.875 | INLAND | 5. |
| **14053** | -117.13 | 32.75 | 24.0 | 1877.0 | 519.0 | 898.0 | 483.0 | 2.2264 | NEAR OCEAN | 3. |
| **20496** | -118.7 | 34.28 | 27.0 | 3536.0 | 646.0 | 1837.0 | 580.0 | 4.4964 | <1H OCEAN | 6. |

# Transformation Pipelines

When you call the pipeline's fit() method, it calls fit_transform() sequentially on all 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.

In [162…
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

```
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

In [163…
```
housing_num_tr
```

Out[163]:
```
array([[-0.94135046,  1.34743822,  0.02756357, ...,  0.01739526,
         0.00622264, -0.12112176],
       [ 1.17178212, -1.19243966, -1.72201763, ...,  0.56925554,
        -0.04081077, -0.81086696],
       [ 0.26758118, -0.1259716 ,  1.22045984, ..., -0.01802432,
        -0.07537122, -0.33827252],
       ...,
       [-1.5707942 ,  1.31001828,  1.53856552, ..., -0.5092404 ,
        -0.03743619,  0.32286937],
       [-1.56080303,  1.2492109 , -1.1653327 , ...,  0.32814891,
        -0.05915604, -0.45702273],
       [-1.28105026,  2.02567448, -0.13148926, ...,  0.01407228,
         0.00657083, -0.12169672]])
```

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. In version 0.20, Scikit-Learn introduced the ColumnTransformer for this purpose, and the good news is that it works great with Pandas DataFrames.

In [164…
```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

In [165…
```
housing_prepared
```

Out[165]:
```
array([[-0.94135046,  1.34743822,  0.02756357, ...,  0.        ,
         0.        ,  0.        ],
       [ 1.17178212, -1.19243966, -1.72201763, ...,  0.        ,
         0.        ,  1.        ],
       [ 0.26758118, -0.1259716 ,  1.22045984, ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [-1.5707942 ,  1.31001828,  1.53856552, ...,  0.        ,
         0.        ,  0.        ],
       [-1.56080303,  1.2492109 , -1.1653327 , ...,  0.        ,
         0.        ,  0.        ],
       [-1.28105026,  2.02567448, -0.13148926, ...,  0.        ,
         0.        ,  0.        ]])
```

In [166…
```
housing_prepared.shape
```

Out[166]:
```
(16512, 16)
```

For reference, here is the old solution based on a DataFrameSelector transformer (to just select a subset of the Pandas DataFrame columns), and a FeatureUnion:

In [167…
```
from sklearn.base import BaseEstimator, TransformerMixin

# Create a class to select numerical or categorical columns
class OldDataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Now let's join all these components into a big pipeline that will preprocess both the numerical and the categorical features:

In [168…
```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

old_num_pipeline = Pipeline([
        ('selector', OldDataFrameSelector(num_attribs)),
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
```

```
        ('std_scaler', StandardScaler()),
    ])

old_cat_pipeline = Pipeline([
        ('selector', OldDataFrameSelector(cat_attribs)),
        ('cat_encoder', OneHotEncoder(sparse=False)),
    ])
```

In [169…
```
from sklearn.pipeline import FeatureUnion

old_full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", old_num_pipeline),
        ("cat_pipeline", old_cat_pipeline),
    ])
```

In [170…
```
old_housing_prepared = old_full_pipeline.fit_transform(housing)
old_housing_prepared
```

Out[170]:
```
array([[-0.94135046,  1.34743822,  0.02756357, ...,  0.        ,
         0.        ,  0.        ],
       [ 1.17178212, -1.19243966, -1.72201763, ...,  0.        ,
         0.        ,  1.        ],
       [ 0.26758118, -0.1259716 ,  1.22045984, ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [-1.5707942 ,  1.31001828,  1.53856552, ...,  0.        ,
         0.        ,  0.        ],
       [-1.56080303,  1.2492109 , -1.1653327 , ...,  0.        ,
         0.        ,  0.        ],
       [-1.28105026,  2.02567448, -0.13148926, ...,  0.        ,
         0.        ,  0.        ]])
```

The result is the same as with the ColumnTransformer:

In [171…
```
np.allclose(housing_prepared, old_housing_prepared)
```

Out[171]:
```
True
```

# Select and Train Model

# Training and Evaluating on the Training Set

Let's first train a Linear Regression model

In [172…
```
from sklearn.linear_model import LinearRegression

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

Out[172]:
```
LinearRegression()
```

In [173…
```
# let's try the full preprocessing pipeline on a few training instances
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)

print("Predictions:", lin_reg.predict(some_data_prepared))
```

```
Predictions: [ 85657.90192014 305492.60737488 152056.46122456 186095.70946094
 244550.67966089]
```

Compare against the actual values:

In [174…
```
print("Labels:", list(some_labels))
```

```
Labels: [72100.0, 279600.0, 82700.0, 112500.0, 238300.0]
```

In [175…
```
some_data_prepared
```

Out[175]:
```
array([[-0.94135046,  1.34743822,  0.02756357,  0.58477745,  0.64037127,
         0.73260236,  0.55628602, -0.8936472 ,  0.01739526,  0.00622264,
        -0.12112176,  0.        ,  1.        ,  0.        ,  0.        ,
         0.        ],
       [ 1.17178212, -1.19243966, -1.72201763,  1.26146668,  0.78156132,
         0.53361152,  0.72131799,  1.292168  ,  0.56925554, -0.04081077,
        -0.81086696,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ],
       [ 0.26758118, -0.1259716 ,  1.22045984, -0.46977281, -0.54513828,
        -0.67467519, -0.52440722, -0.52543365, -0.01802432, -0.07537122,
        -0.33827252,  0.        ,  1.        ,  0.        ,  0.        ,
         0.        ],
       [ 1.22173797, -1.35147437, -0.37006852, -0.34865152, -0.03636724,
        -0.46761716, -0.03729672, -0.86592882, -0.59513997, -0.10680295,
         0.96120521,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ],
       [ 0.43743108, -0.63581817, -0.13148926,  0.42717947,  0.27279028,
         0.37406031,  0.22089846,  0.32575178,  0.2512412 ,  0.00610923,
        -0.47451338,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ]])
```

In [176…
```
from sklearn.metrics import mean_squared_error

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

Out[176]:
```
68627.87390018745
```

In [177…
```
from sklearn.metrics import mean_absolute_error

lin_mae = mean_absolute_error(housing_labels, housing_predictions)
lin_mae
```

Out[177]:
```
49438.66860915802
```

Okay, this is better than nothing but clearly not a great score: most districts' median_housing_values range between 120,000 and 265,000, so a typical predic- tion error of 68,628 is not very satisfying.

This is an example of a model underfitting the training data. When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough.

Let's try a more complex model to see how it does.Let's train a DecisionTreeRegressor. This is a powerful model, capable of finding complex nonlinear relationships in the data

In [178…
```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)
```

Out[178]:
```
DecisionTreeRegressor(random_state=42)
```

In [179…
```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

Out[179]:
```
0.0
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data.

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 for model validation

# Better Evaluation Using Cross-Validation

A great alternative is to use Scikit-Learn's K-fold cross-validation feature. The following code randomly splits the training set into 10 distinct subsets called folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

In [180…
```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
```

```
                            scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

In [181…

```
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

```
Scores: [72831.45749112 69973.18438322 69528.56551415 72517.78229792
 69145.50006909 79094.74123727 68960.045444   73344.50225684
 69826.02473916 71077.09753998]
Mean: 71629.89009727491
Standard deviation: 2914.035468468928
```

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

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)

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

In [182…

```
lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                             scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

```
Scores: [71762.76364394 64114.99166359 67771.17124356 68635.19072082
 66846.14089488 72528.03725385 73997.08050233 68802.33629334
 66443.28836884 70139.79923956]
Mean: 69104.07998247063
Standard deviation: 2880.3282098180634
```

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

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

Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions.

In [183…
```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

Out[183]:
```
RandomForestRegressor(random_state=42)
```

In [184…
```
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

Out[184]:
```
18650.698705770003
```

cross validation for random forest

In [185…
```
from sklearn.model_selection import cross_val_score

forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
Scores: [51559.63379638 48737.57100062 47210.51269766 51875.21247297
 47577.50470123 51863.27467888 52746.34645573 50065.1762751
 48664.66818196 54055.90894609]
Mean: 50435.58092066179
Standard deviation: 2203.3381412764606
```

Wow, this is much better: Random Forests look very promising. However, note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. Possible solutions for overfitting are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. However, before you dive much deeper in Random Forests, you should try out many other models from various categories of Machine Learning algorithms (several Support Vector Machines with different kernels, possibly a neural network, etc.), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising models.

In [186…
```
scores = cross_val_score(lin_reg, housing_prepared, housing_labels, scoring="neg_mean_squared_error", cv=10)
pd.Series(np.sqrt(-scores)).describe()
```

Out[186]:
```
count       10.000000
mean     69104.079982
std       3036.132517
min      64114.991664
25%      67077.398482
50%      68718.763507
75%      71357.022543
max      73997.080502
dtype: float64
```

In [187…
```
from sklearn.svm import SVR

svm_reg = SVR(kernel="linear")
svm_reg.fit(housing_prepared, housing_labels)
housing_predictions = svm_reg.predict(housing_prepared)
svm_mse = mean_squared_error(housing_labels, housing_predictions)
svm_rmse = np.sqrt(svm_mse)
svm_rmse
```

Out[187]:
```
111095.06635291966
```

# Fine-Tune Your Model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them. Let's look at a few ways you can do that.

# Grid Search

One way to do that would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations. Instead you should get Scikit-Learn's GridSearchCV 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 evaluate all the possible combinations of hyperparameter values, using cross-validation.

The following code searches for the best combination of hyperparameter values for the RandomForestRegressor:

In [188…
```
from sklearn.model_selection import GridSearchCV
```

```
param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
```

Out[188]:
```
GridSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
                         {'bootstrap': [False], 'max_features': [2, 3, 4],
                          'n_estimators': [3, 10]}],
             return_train_score=True, scoring='neg_mean_squared_error')
```

This param_grid tells Scikit-Learn to first evaluate all 3 × 4 = 12 combinations of n_estimators and max_features hyperparameter values specified in the first dict, then try all 2 × 3 = 6 combinations of hyperparameter values in the second dict, but this time with the bootstrap hyperparameter set to False instead of True (which is the default value for this hyperparameter).

All in all, the grid search will explore 12 + 6 = 18 combinations of RandomForestRe gressor hyperparameter values, and it will train each model five times (since we are using five-fold cross validation). In other words, all in all, there will be 18 × 5 = 90 rounds of training! It may take quite a long time, but when it is done you can get the best combination of parameters like this:

In [189…
```
grid_search.best_params_
```

Out[189]:
```
{'max_features': 8, 'n_estimators': 30}
```

In [190…
```
grid_search.best_estimator_
```

Out[190]:
```
RandomForestRegressor(max_features=8, n_estimators=30, random_state=42)
```

Let's look at the score of each hyperparameter combination tested during the grid search:

In [191…
```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
63895.161577951665 {'max_features': 2, 'n_estimators': 3}
54916.32386349543 {'max_features': 2, 'n_estimators': 10}
52885.86715332332 {'max_features': 2, 'n_estimators': 30}
60075.3680329983 {'max_features': 4, 'n_estimators': 3}
52495.01284985185 {'max_features': 4, 'n_estimators': 10}
50187.24324926565 {'max_features': 4, 'n_estimators': 30}
58064.73529982314 {'max_features': 6, 'n_estimators': 3}
51519.32062366315 {'max_features': 6, 'n_estimators': 10}
49969.80441627874 {'max_features': 6, 'n_estimators': 30}
58895.824998155826 {'max_features': 8, 'n_estimators': 3}
52459.79624724529 {'max_features': 8, 'n_estimators': 10}
49898.98913455217 {'max_features': 8, 'n_estimators': 30}
62381.765106921855 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54476.57050944266 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59974.60028085155 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52754.5632813202 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57831.136061214274 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51278.37877140253 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

In this example, we obtain the best solution by setting the max_features hyperparameter to 8, and the n_estimators hyperparameter to 30. The RMSE score for this combination is 49,898, which is slightly better than the score you got earlier using the default hyperparameter values.

In [192…
```
pd.DataFrame(grid_search.cv_results_)
```

Out[192]:

| | mean_fit_time | std_fit_time | mean_score_time | std_score_time | param_max_features | param_n_estimators | param_bootstrap | params | split0_test_s |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0.099929 | 0.007401 | 0.007848 | 0.006987 | 2 | 3 | NaN | {'max_features': 2, 'n_estimators': 3} | -4.119912 |
| **1** | 0.334135 | 0.007459 | 0.015624 | 0.000004 | 2 | 10 | NaN | {'max_features': 2, 'n_estimators': 10} | -2.973521 |
| **2** | 0.996656 | 0.006258 | 0.043743 | 0.006248 | 2 | 30 | NaN | {'max_features': 2, 'n_estimators': 30} | -2.801229 |
| **3** | 0.159328 | 0.006252 | 0.009373 | 0.007653 | 4 | 3 | NaN | {'max_features': 4, 'n_estimators': 3} | -3.528743 |
| **4** | 0.527978 | 0.006219 | 0.015622 | 0.000003 | 4 | 10 | NaN | {'max_features': 4, 'n_estimators': 10} | -2.742620 |
| **5** | 1.574634 | 0.006253 | 0.046858 | 0.000017 | 4 | 30 | NaN | {'max_features': 4, 'n_estimators': 30} | -2.522176 |
| **6** | 0.206192 | 0.006251 | 0.009374 | 0.007654 | 6 | 3 | NaN | {'max_features': 6, 'n_estimators': 3} | -3.362127 |
| **7** | 0.706092 | 0.006242 | 0.015623 | 0.000002 | 6 | 10 | NaN | {'max_features': 6, 'n_estimators': 10} | -2.622099 |
| **8** | 2.180739 | 0.067447 | 0.043749 | 0.006232 | 6 | 30 | NaN | {'max_features': 6, 'n_estimators': 30} | -2.446142 |

| | mean_fit_time | std_fit_time | mean_score_time | std_score_time | param_max_features | param_n_estimators | param_bootstrap | params | split0_test_s |
|---|---|---|---|---|---|---|---|---|---|
| **9** | 0.321803 | 0.025392 | 0.009372 | 0.007652 | 8 | 3 | NaN | {'max_features': 8, 'n_estimators': 3} | -3.590333 |
| **10** | 0.940400 | 0.024977 | 0.015623 | 0.000004 | 8 | 10 | NaN | {'max_features': 8, 'n_estimators': 10} | -2.721311 |
| **11** | 2.774722 | 0.032238 | 0.046865 | 0.000004 | 8 | 30 | NaN | {'max_features': 8, 'n_estimators': 30} | -2.492636 |
| **12** | 0.161038 | 0.006324 | 0.001633 | 0.003265 | 2 | 3 | False | {'bootstrap': False, 'max_features': 2, 'n_est... | -4.020842 |
| **13** | 0.515505 | 0.009895 | 0.018739 | 0.006253 | 2 | 10 | False | {'bootstrap': False, 'max_features': 2, 'n_est... | -2.901352 |
| **14** | 0.203085 | 0.000017 | 0.003123 | 0.006246 | 3 | 3 | False | {'bootstrap': False, 'max_features': 3, 'n_est... | -3.687132 |
| **15** | 0.758967 | 0.060646 | 0.025000 | 0.012493 | 3 | 10 | False | {'bootstrap': False, 'max_features': 3, 'n_est... | -2.837028 |
| **16** | 0.249933 | 0.000014 | 0.009373 | 0.007653 | 4 | 3 | False | {'bootstrap': False, 'max_features': 4, 'n_est... | -3.549428 |
| **17** | 0.881882 | 0.086379 | 0.016662 | 0.002081 | 4 | 10 | False | {'bootstrap': False, 'max_features': 4, 'n_est... | -2.692499 |

18 rows x 23 columns

# Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead. This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration. This approach has two main benefits

• If you let the randomized search run for, say, 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter (instead of just a few values per hyperparameter with the grid search approach). • You have more control over the computing budget you want to allocate to hyperparameter search, simply by setting the number of iterations.

In [194…
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_error', random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

Out[194]:
```
RandomizedSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
                   param_distributions={'max_features': <scipy.stats._distn_infrastructure.rv_frozen object at 0x00000174599D7820>,
                                        'n_estimators': <scipy.stats._distn_infrastructure.rv_frozen object at 0x00000174599D7040>},
                   random_state=42, scoring='neg_mean_squared_error')
```

In [195…
```
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
49117.55344336652 {'max_features': 7, 'n_estimators': 180}
51450.63202856348 {'max_features': 5, 'n_estimators': 15}
50692.53588182537 {'max_features': 3, 'n_estimators': 72}
50783.614493515 {'max_features': 5, 'n_estimators': 21}
49162.89877456354 {'max_features': 7, 'n_estimators': 122}
50655.798471042704 {'max_features': 3, 'n_estimators': 75}
50513.856319990606 {'max_features': 3, 'n_estimators': 88}
49521.17201976928 {'max_features': 5, 'n_estimators': 100}
50302.90440763418 {'max_features': 3, 'n_estimators': 150}
65167.02018649492 {'max_features': 5, 'n_estimators': 2}
```

# Analyze the Best Models and Their Errors

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

In [196…

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

Out[196]:

```
array([6.96542523e-02, 6.04213840e-02, 4.21882202e-02, 1.52450557e-02,
       1.55545295e-02, 1.58491147e-02, 1.49346552e-02, 3.79009225e-01,
       5.47789150e-02, 1.07031322e-01, 4.82031213e-02, 6.79266007e-03,
       1.65706303e-01, 7.83480660e-05, 1.52473276e-03, 3.02816106e-03])
```

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

In [197…

```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

Out[197]:
```
[(0.3790092248170967, 'median_income'),
 (0.16570630316895876, 'INLAND'),
 (0.10703132208204355, 'pop_per_hhold'),
 (0.06965425227942929, 'longitude'),
 (0.0604213840080722, 'latitude'),
 (0.054778915018283726, 'rooms_per_hhold'),
 (0.048203121338269206, 'bedrooms_per_room'),
 (0.04218822024391753, 'housing_median_age'),
 (0.015849114744428634, 'population'),
 (0.015554529490469328, 'total_bedrooms'),
 (0.01524505568840977, 'total_rooms'),
 (0.014934655161887772, 'households'),
 (0.006792660074259966, '<1H OCEAN'),
 (0.0030281610628962747, 'NEAR OCEAN'),
 (0.0015247327555504937, 'NEAR BAY'),
 (7.834806602687504e-05, 'ISLAND')]
```

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). 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, on the contrary, getting rid of uninformative ones, cleaning up outliers, etc.)

# Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set, run your full_pipeline to transform the data (call transform(), not fit_transform(), you do not want to fit the test set!), and evaluate the final model on the test set.

In [198…
```
final_model = grid_search.best_estimator_

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

X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

In [199…
```
final_rmse
```

Out[199]:
```
47873.26095812988
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% confidence interval for the generalization error using scipy.stats.t.interval():

In [200…
```
from scipy import stats

confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

Out[200]:
```
array([45893.36082829, 49774.46796717])
```

In [201…
```
m = len(squared_errors)
mean = squared_errors.mean()
tscore = stats.t.ppf((1 + confidence) / 2, df=m - 1)
tmargin = tscore * squared_errors.std(ddof=1) / np.sqrt(m)
np.sqrt(mean - tmargin), np.sqrt(mean + tmargin)
```

Out[201]:
```
(45893.3608282853, 49774.46796717339)
```

Alternatively, we could use a z-scores rather than t-scores:

In [202…
```
zscore = stats.norm.ppf((1 + confidence) / 2)
zmargin = zscore * squared_errors.std(ddof=1) / np.sqrt(m)
np.sqrt(mean - zmargin), np.sqrt(mean + zmargin)
```

Out[202]:
```
(45893.954011012866, 49773.92103065016)
```

The performance will usually be slightly worse than what you measured using cross validation if you did a lot of hyperparameter tuning (because your system ends up fine-tuned to perform well on the validation data, and will likely not perform as well on unknown datasets). It is not the case in this example, but when this 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), 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"). In this California housing example, the final performance of the system is not better than the experts', but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks.

In [ ]: