02_end_to_end_machine_learning_project

February 21, 2023

Chapter 2 – End-to-end Machine Learning project

Welcome to Machine Learning Housing Corp.! Your task is to predict median house values in Californian districts, given a number of features from these districts.

This notebook contains all the sample code and solutions to the exercices in chapter 2.

Run in Google Colab

Warning: this is the code for the 1st edition of the book. Please visit https://github.com/ageron/handson-ml2 for the 2nd edition code, with up-to-date notebooks using the latest library versions.

Note: You may find little differences between the code outputs in the book and in these Jupyter notebooks: these slight differences are mostly due to the random nature of many training algorithms: although I have tried to make these notebooks' outputs as constant as possible, it is impossible to guarantee that they will produce the exact same output on every platform. Also, some data structures (such as dictionaries) do not preserve the item order. Finally, I fixed a few minor bugs (I added notes next to the concerned cells) which lead to slightly different results, without changing the ideas presented in the book.

1 Setup

First, let's make sure this notebook works well in both python 2 and 3, import a few common modules, ensure MatplotLib plots figures inline and prepare a function to save the figures:

[1]:
```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# 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)
```

2 Get the data

[2]:
```
import os
import tarfile
import urllib.request

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

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

[3]:
```
fetch_housing_data()
```

[4]:
```
import pandas as pd

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

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

[5]:
```
   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0    -122.23     37.88                41.0        880.0           129.0
1    -122.22     37.86                21.0       7099.0          1106.0
2    -122.24     37.85                52.0       1467.0           190.0
3    -122.25     37.85                52.0       1274.0           235.0
4    -122.25     37.85                52.0       1627.0           280.0

   population  households  median_income  median_house_value ocean_proximity
0       322.0       126.0         8.3252            452600.0        NEAR BAY
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
```

[6]:
```python
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
```

[7]:
```python
housing["ocean_proximity"].value_counts()
```

[7]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

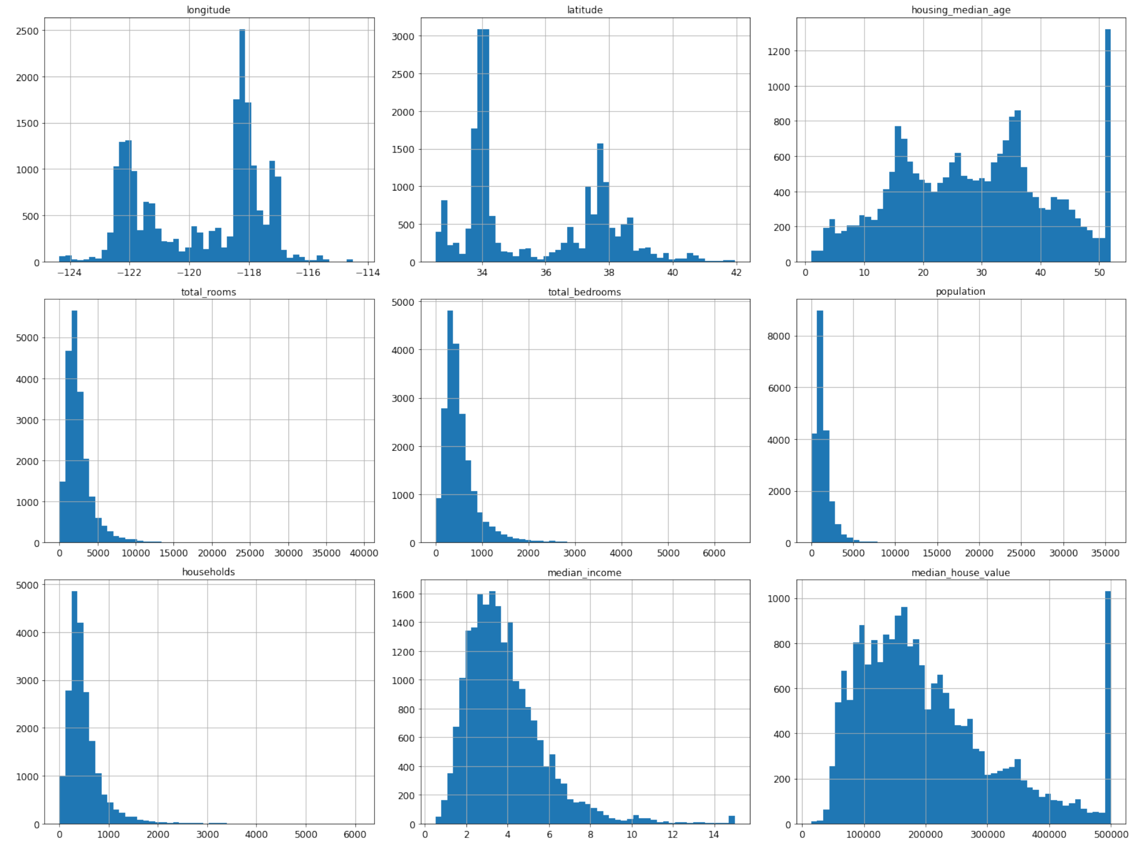
[8]:
```python
housing.describe()
```

[8]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
```

```
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.800000     33.930000           18.000000   1447.750000
50%     -118.490000     34.260000           29.000000   2127.000000
75%     -118.010000     37.710000           37.000000   3148.000000
max     -114.310000     41.950000           52.000000  39320.000000

       total_bedrooms    population    households  median_income  \
count    20433.000000  20640.000000  20640.000000   20640.000000
mean       537.870553   1425.476744    499.539680       3.870671
std        421.385070   1132.462122    382.329753       1.899822
min          1.000000      3.000000      1.000000       0.499900
25%        296.000000    787.000000    280.000000       2.563400
50%        435.000000   1166.000000    409.000000       3.534800
75%        647.000000   1725.000000    605.000000       4.743250
max       6445.000000  35682.000000   6082.000000      15.000100

       median_house_value
count        20640.000000
mean        206855.816909
std         115395.615874
min          14999.000000
25%         119600.000000
50%         179700.000000
75%         264725.000000
max         500001.000000
```

[9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
```

```
Saving figure attribute_histogram_plots
```



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

[11]:
```
import numpy as np

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

[12]:
```
train_set, test_set = split_train_test(housing, 0.2)
print(len(train_set), "train +", len(test_set), "test")
```

```
16512 train + 4128 test
```

[13]:
```
from zlib import crc32

def test_set_check(identifier, test_ratio):
```

```
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

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

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

[14]:
```
import hashlib

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

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

[15]:
```
def test_set_check(identifier, test_ratio, hash=hashlib.md5):
    return bytearray(hash(np.int64(identifier)).digest())[-1] < 256 * test_ratio
```

[16]:
```
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")
```

[17]:
```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

[18]:
```
test_set.head()
```

[18]:
```
    index  longitude  latitude  housing_median_age  total_rooms  \
8       8    -122.26     37.84                42.0       2555.0
10     10    -122.26     37.85                52.0       2202.0
11     11    -122.26     37.85                52.0       3503.0
12     12    -122.26     37.85                52.0       2491.0
13     13    -122.26     37.84                52.0        696.0

    total_bedrooms  population  households  median_income  median_house_value  \
8            665.0      1206.0       595.0         2.0804            226700.0
10           434.0       910.0       402.0         3.2031            281500.0
11           752.0      1504.0       734.0         3.2705            241800.0
12           474.0      1098.0       468.0         3.0750            213500.0
13           191.0       345.0       174.0         2.6736            191300.0

   ocean_proximity          id
8         NEAR BAY -122222.16
10        NEAR BAY -122222.15
```

```
11         NEAR BAY -122222.15
12         NEAR BAY -122222.15
13         NEAR BAY -122222.16
```

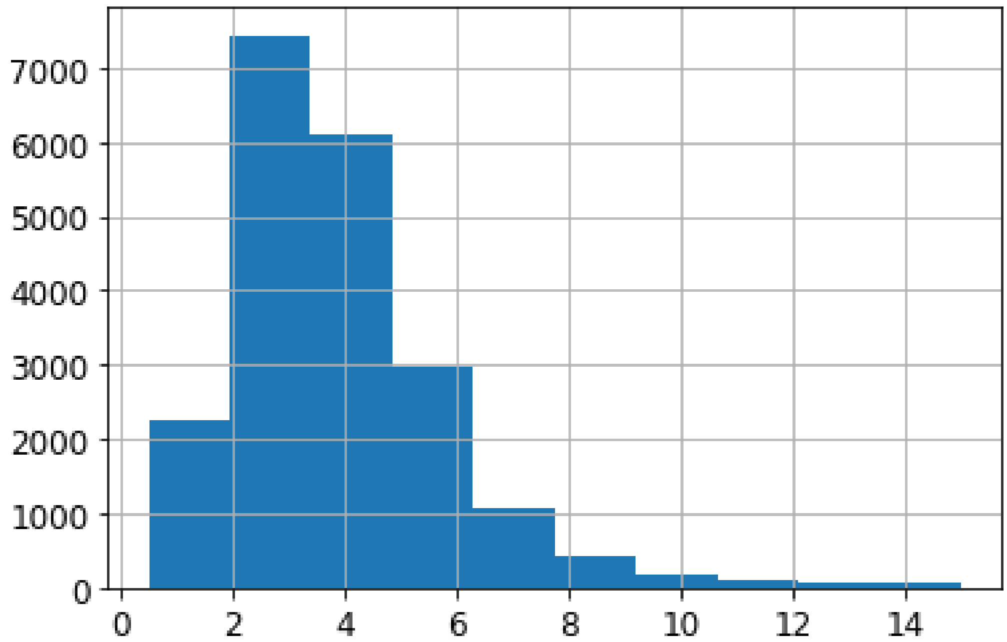
[19]:
```
from sklearn.model_selection import train_test_split

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

[20]:
```
test_set.head()
```

[20]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
20046    -119.01     36.06                25.0       1505.0             NaN
3024     -119.46     35.14                30.0       2943.0             NaN
15663    -122.44     37.80                52.0       3830.0             NaN
20484    -118.72     34.28                17.0       3051.0             NaN
9814     -121.93     36.62                34.0       2351.0             NaN

       population  households  median_income  median_house_value  \
20046      1392.0       359.0         1.6812             47700.0
3024       1565.0       584.0         2.5313             45800.0
15663      1310.0       963.0         3.4801            500001.0
20484      1705.0       495.0         5.7376            218600.0
9814       1063.0       428.0         3.7250            278000.0

      ocean_proximity
20046          INLAND
3024           INLAND
15663        NEAR BAY
20484       <1H OCEAN
9814       NEAR OCEAN
```

[21]:
```
housing["median_income"].hist()
```

[21]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3be9fcb730>
```



Warning: in the book, I did not use `pd.cut()`, instead I used the code below. The `pd.cut()` solution gives the same result (except the labels are integers instead of floats), but it is simpler to understand:

```
# Divide by 1.5 to limit the number of income categories
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
# Label those above 5 as 5
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

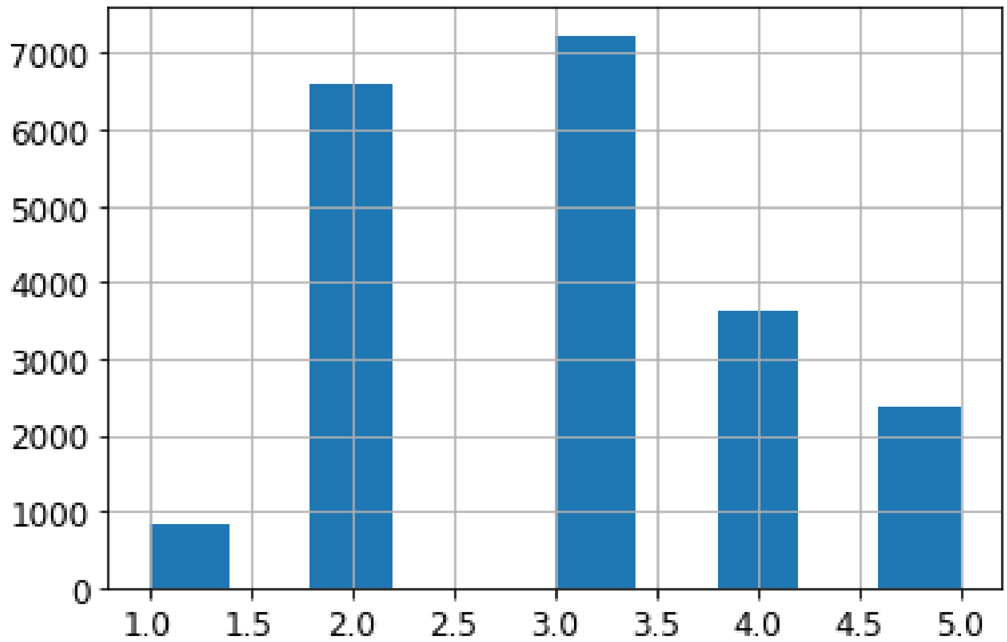
[22]:
```
housing["income_cat"] = pd.cut(housing["median_income"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
```

[23]:
```
housing["income_cat"].value_counts()
```

[23]:
```
3    7236
2    6581
4    3639
5    2362
1     822
Name: income_cat, dtype: int64
```

[24]:
```
housing["income_cat"].hist()
```

[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3be9f289d0>
```



[25]:
```
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]
```

[26]:
```
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

[26]:
```
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: income_cat, dtype: float64
```

[27]:
```
housing["income_cat"].value_counts() / len(housing)
```

[27]:
```
3    0.350581
2    0.318847
4    0.176308
5    0.114438
1    0.039826
Name: income_cat, dtype: float64
```

[28]:
```python
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
```

[29]:
```python
compare_props
```

[29]:

	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

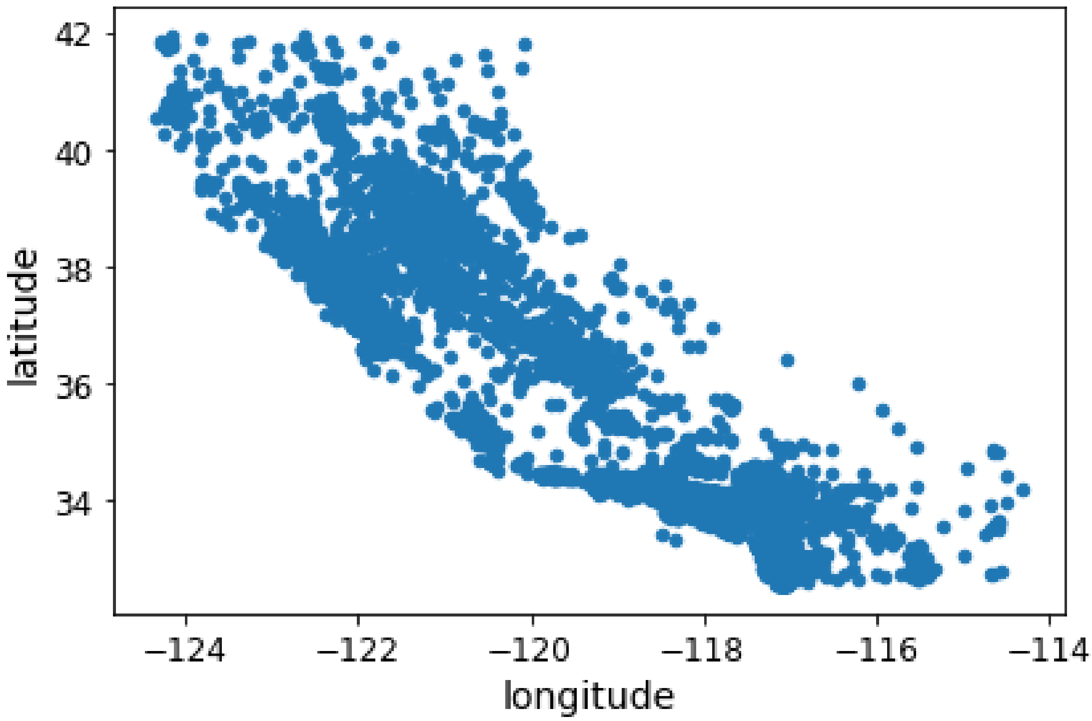
[30]:
```python
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

3 Discover and visualize the data to gain insights

[31]:
```python
housing = strat_train_set.copy()
```

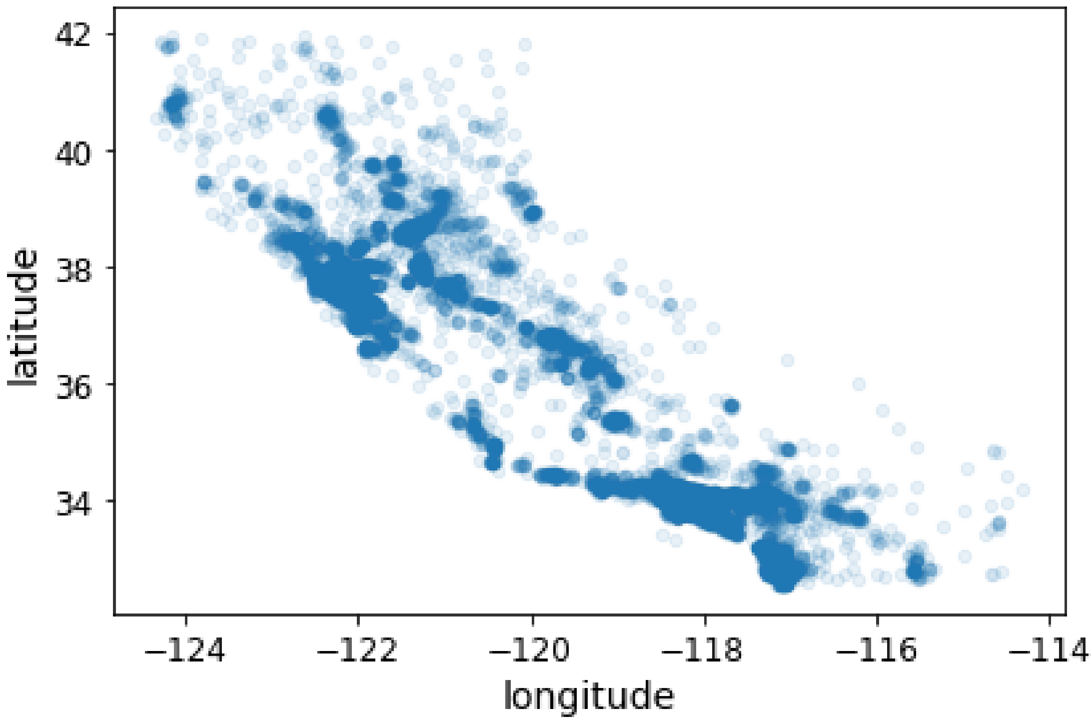
[32]:
```python
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

```
Saving figure bad_visualization_plot
```



[33]:
```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

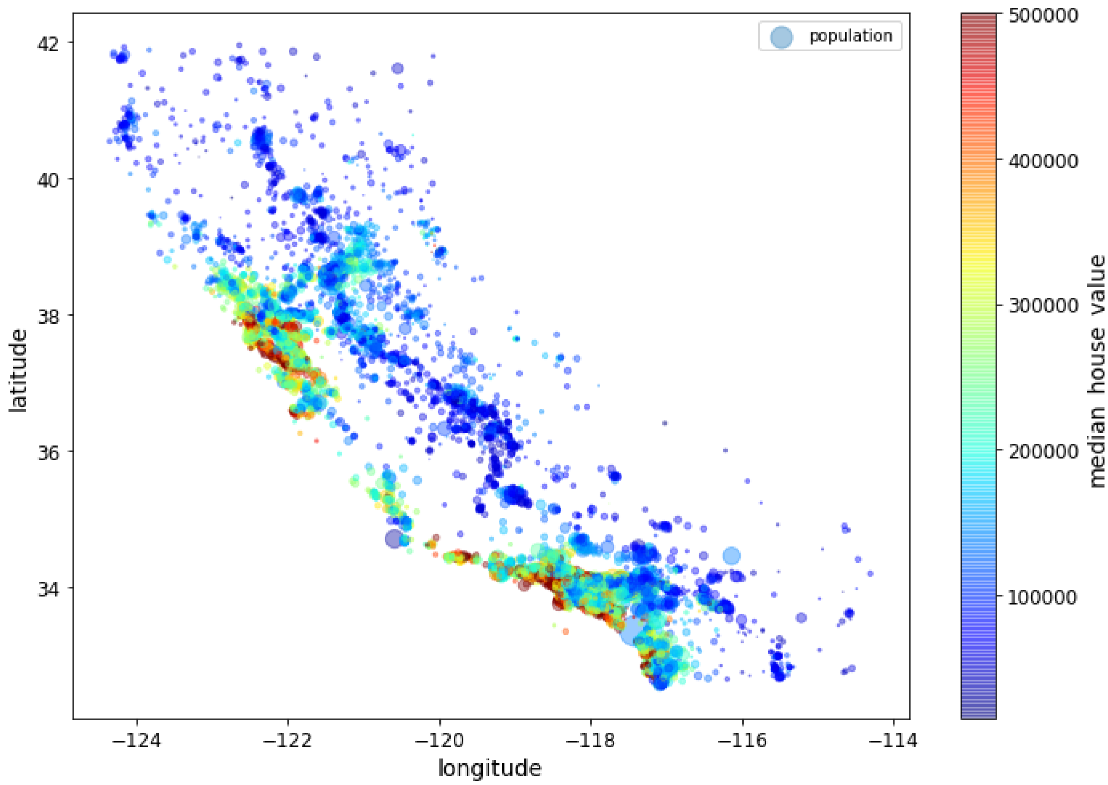
```
Saving figure better_visualization_plot
```



The argument `sharex=False` fixes a display bug (the x-axis values and legend were not displayed). This is a temporary fix (see: https://github.com/pandas-dev/pandas/issues/10611). Thanks to Wilmer Arellano for pointing it out.

[34]:
```
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
```



[35]:
```
# Download the California image
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "end_to_end_project")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
filename = "california.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/end_to_end_project/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))
```

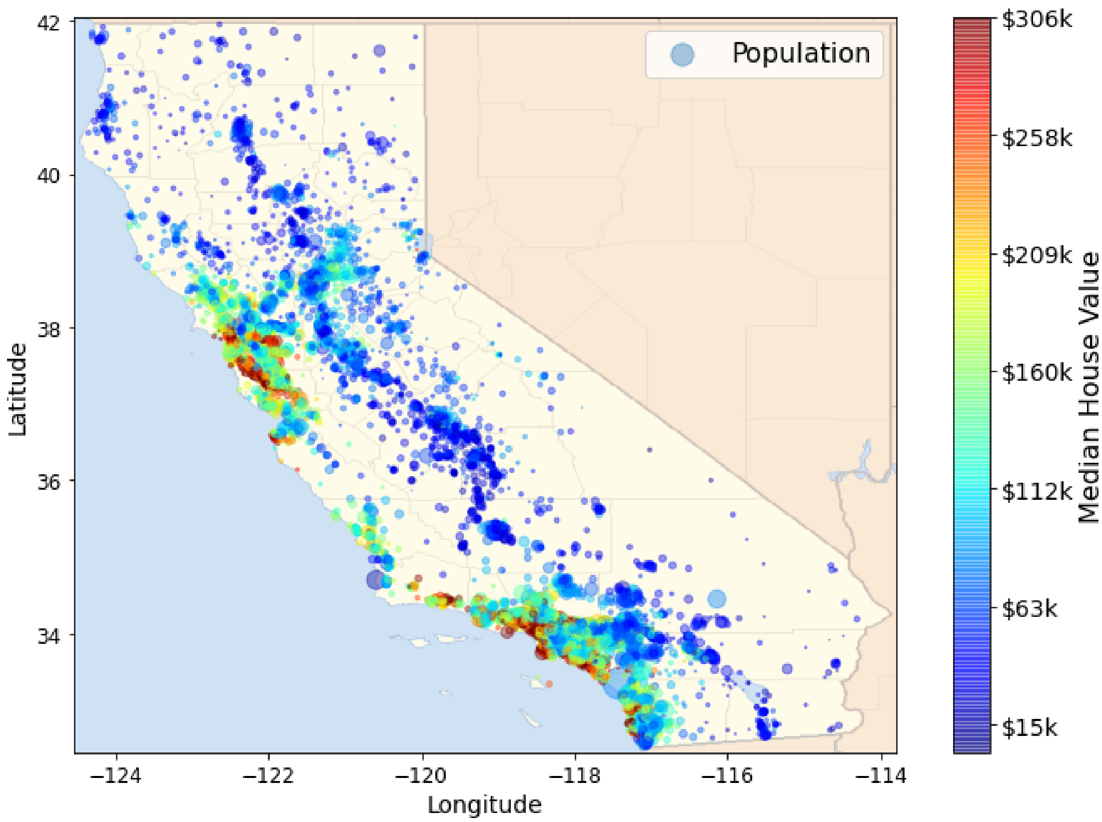
```
Downloading california.png
```

[35]:
```
('./images/end_to_end_project/california.png',
 <http.client.HTTPMessage at 0x7f3be9e50400>)
```

[36]:
```
import matplotlib.image as mpimg
california_img=mpimg.imread(PROJECT_ROOT_DIR + '/images/end_to_end_project/
↪california.png')
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                       s=housing['population']/100, label="Population",
                       c="median_house_value", cmap=plt.get_cmap("jet"),
                       colorbar=False, alpha=0.4,
                      )
```

```
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5,
           cmap=plt.get_cmap("jet"))
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar()
cbar.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values],␣
 ↪fontsize=14)
cbar.set_label('Median House Value', fontsize=16)

plt.legend(fontsize=16)
save_fig("california_housing_prices_plot")
plt.show()
```

```
Saving figure california_housing_prices_plot
```



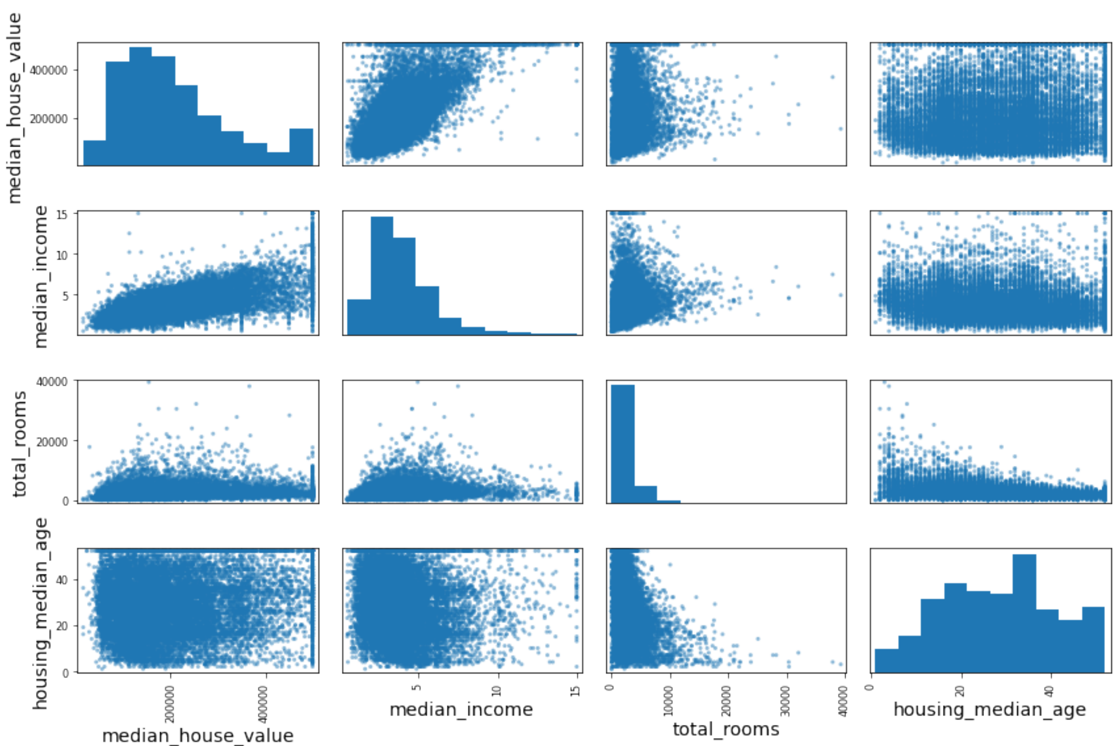
[37]:
```
corr_matrix = housing.corr()
```

[38]:
```
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[38]:
```
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
```

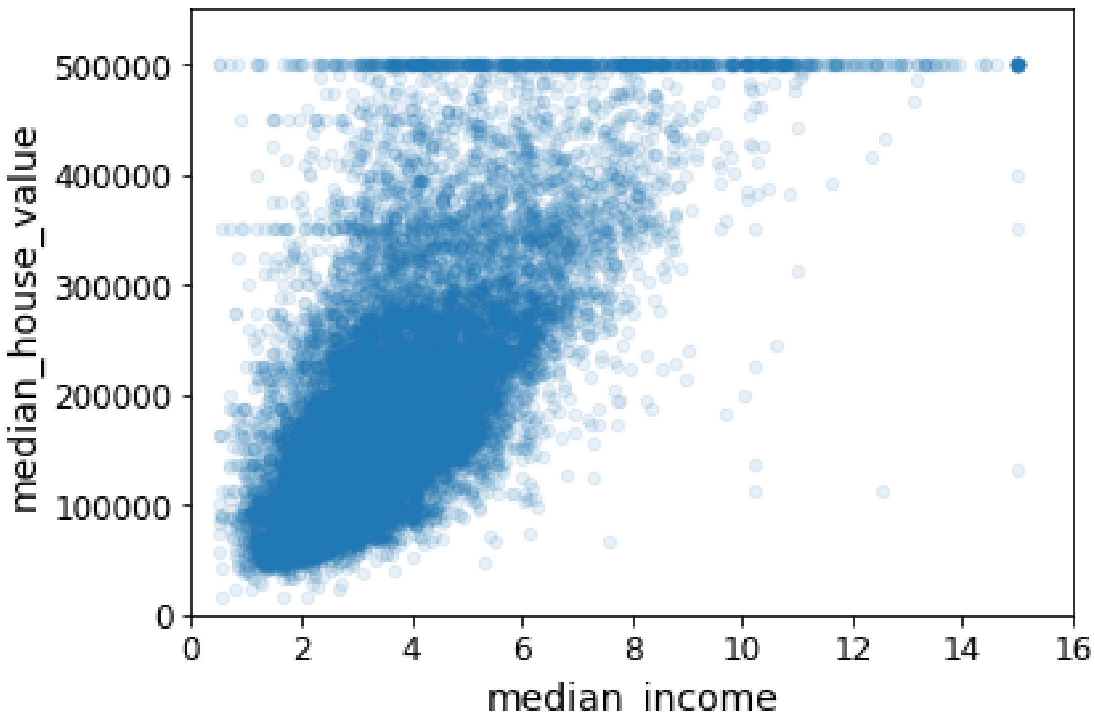
[39]:
```python
# from pandas.tools.plotting import scatter_matrix # For older versions of Pandas
from pandas.plotting import scatter_matrix

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
```



[40]:
```
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



[41]:
```
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"]
```

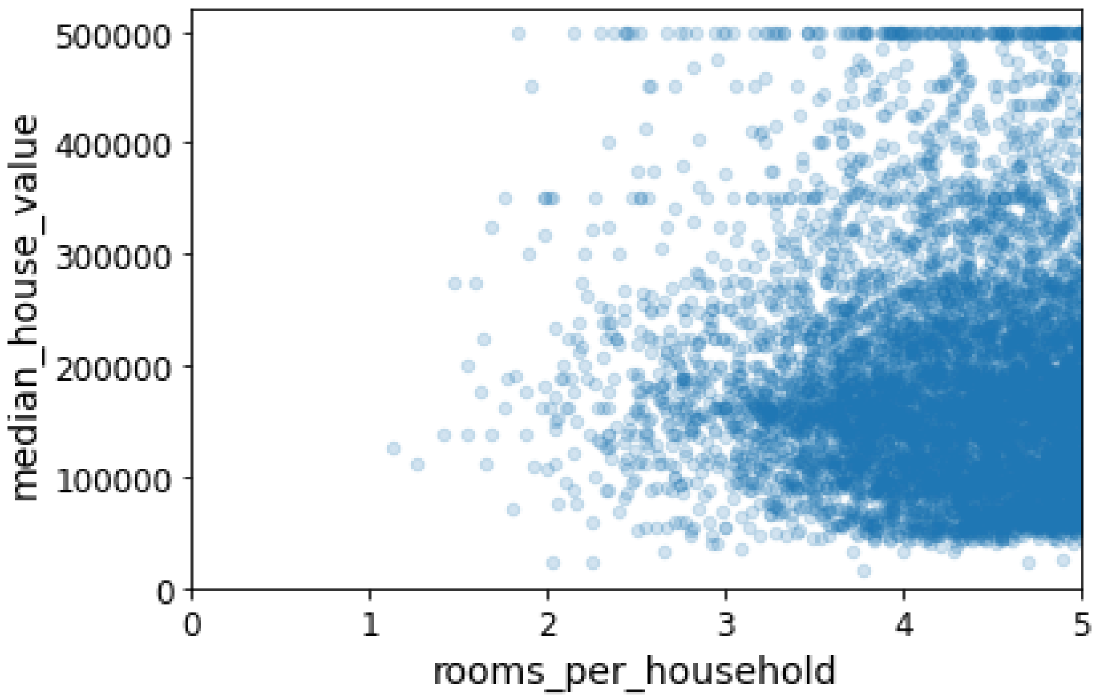
Note: there was a bug in the previous cell, in the definition of the `rooms_per_household` attribute. This explains why the correlation value below differs slightly from the value in the book (unless you are reading the latest version).

[42]:
```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[42]:
```
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
```

[43]:
```python
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



[44]:
```python
housing.describe()
```

[44]:
```
          longitude      latitude  housing_median_age   total_rooms  \
count  16512.000000  16512.000000        16512.000000  16512.000000
mean    -119.575635     35.639314           28.653404   2622.539789
std        2.001828      2.137963           12.574819   2138.417080
min     -124.350000     32.540000            1.000000      6.000000
25%     -121.800000     33.940000           18.000000   1443.000000
50%     -118.510000     34.260000           29.000000   2119.000000
75%     -118.010000     37.720000           37.000000   3141.000000
max     -114.310000     41.950000           52.000000  39320.000000
```

```
       total_bedrooms    population   households  median_income  \
count    16354.000000  16512.000000  16512.000000  16512.000000
mean       534.914639   1419.687379    497.011810      3.875884
std        412.665649   1115.663036    375.696156      1.904931
min          2.000000      3.000000      2.000000      0.499900
25%        295.000000    784.000000    279.000000      2.566950
50%        433.000000   1164.000000    408.000000      3.541550
75%        644.000000   1719.000000    602.000000      4.745325
max       6210.000000  35682.000000   5358.000000     15.000100

       median_house_value  rooms_per_household  bedrooms_per_room  \
count        16512.000000         16512.000000       16354.000000
mean        207005.322372             5.440406           0.212873
std         115701.297250             2.611696           0.057378
min          14999.000000             1.130435           0.100000
25%         119800.000000             4.442168           0.175304
50%         179500.000000             5.232342           0.203027
75%         263900.000000             6.056361           0.239816
max         500001.000000           141.909091           1.000000

       population_per_household
count              16512.000000
mean                   3.096469
std                   11.584825
min                    0.692308
25%                    2.431352
50%                    2.817661
75%                    3.281420
max                 1243.333333
```

4 Prepare the data for Machine Learning algorithms

[45]:
```
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels for␣
↪training set
housing_labels = strat_train_set["median_house_value"].copy()
```

[46]:
```
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
```

[46]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
1606     -122.08     37.88                26.0       2947.0             NaN
10915    -117.87     33.73                45.0       2264.0             NaN
19150    -122.70     38.35                14.0       2313.0             NaN
4186     -118.23     34.13                48.0       1308.0             NaN
16885    -122.40     37.58                26.0       3281.0             NaN
```

```
       population  households  median_income ocean_proximity
1606        825.0       626.0         2.9330        NEAR BAY
10915      1970.0       499.0         3.4193       <1H OCEAN
19150       954.0       397.0         3.7813       <1H OCEAN
4186        835.0       294.0         4.2891       <1H OCEAN
16885      1145.0       480.0         6.3580      NEAR OCEAN
```

[47]:
```python
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

[47]:
```
Empty DataFrame
Columns: [longitude, latitude, housing_median_age, total_rooms, total_bedrooms,
population, households, median_income, ocean_proximity]
Index: []
```

[48]:
```python
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

[48]:
```
       longitude  latitude  housing_median_age  total_rooms  population  \
1606     -122.08     37.88                26.0       2947.0       825.0
10915    -117.87     33.73                45.0       2264.0      1970.0
19150    -122.70     38.35                14.0       2313.0       954.0
4186     -118.23     34.13                48.0       1308.0       835.0
16885    -122.40     37.58                26.0       3281.0      1145.0

       households  median_income ocean_proximity
1606        626.0         2.9330        NEAR BAY
10915       499.0         3.4193       <1H OCEAN
19150       397.0         3.7813       <1H OCEAN
4186        294.0         4.2891       <1H OCEAN
16885       480.0         6.3580      NEAR OCEAN
```

[49]:
```python
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
sample_incomplete_rows
```

[49]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
1606     -122.08     37.88                26.0       2947.0           433.0
10915    -117.87     33.73                45.0       2264.0           433.0
19150    -122.70     38.35                14.0       2313.0           433.0
4186     -118.23     34.13                48.0       1308.0           433.0
16885    -122.40     37.58                26.0       3281.0           433.0

       population  households  median_income ocean_proximity
1606        825.0       626.0         2.9330        NEAR BAY
10915      1970.0       499.0         3.4193       <1H OCEAN
19150       954.0       397.0         3.7813       <1H OCEAN
4186        835.0       294.0         4.2891       <1H OCEAN
```

```
16885       1145.0       480.0          6.3580      NEAR OCEAN
```

Warning: Since Scikit-Learn 0.20, the `sklearn.preprocessing.Imputer` class was replaced by the `sklearn.impute.SimpleImputer` class.

[50]:
```
try:
    from sklearn.impute import SimpleImputer # Scikit-Learn 0.20+
except ImportError:
    from sklearn.preprocessing import Imputer as SimpleImputer

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

Remove the text attribute because median can only be calculated on numerical attributes:

[51]:
```
housing_num = housing.drop('ocean_proximity', axis=1)
# alternatively: housing_num = housing.select_dtypes(include=[np.number])
```

[52]:
```
imputer.fit(housing_num)
```

[52]:
```
SimpleImputer(strategy='median')
```

[53]:
```
imputer.statistics_
```

[53]:
```
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:

[54]:
```
housing_num.median().values
```

[54]:
```
array([-118.51    ,   34.26    ,   29.     , 2119.      ,  433.      ,
       1164.     ,  408.      ,    3.54155])
```

Transform the training set:

[55]:
```
X = imputer.transform(housing_num)
```

[56]:
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing.index)
```

[57]:
```
housing_tr.loc[sample_incomplete_rows.index.values]
```

[57]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
1606     -122.08     37.88                26.0       2947.0           433.0
10915    -117.87     33.73                45.0       2264.0           433.0
19150    -122.70     38.35                14.0       2313.0           433.0
4186     -118.23     34.13                48.0       1308.0           433.0
16885    -122.40     37.58                26.0       3281.0           433.0
```

```
       population  households  median_income
1606        825.0       626.0         2.9330
10915      1970.0       499.0         3.4193
19150       954.0       397.0         3.7813
4186        835.0       294.0         4.2891
16885      1145.0       480.0         6.3580
```

[58]:
```
imputer.strategy
```

[58]:
```
'median'
```

[59]:
```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
housing_tr.head()
```

[59]:
```
       longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
12655    -121.46     38.52                29.0       3873.0           797.0
15502    -117.23     33.09                 7.0       5320.0           855.0
2908     -119.04     35.37                44.0       1618.0           310.0
14053    -117.13     32.75                24.0       1877.0           519.0
20496    -118.70     34.28                27.0       3536.0           646.0

       population  households  median_income
12655      2237.0       706.0         2.1736
15502      2015.0       768.0         6.3373
2908        667.0       300.0         2.8750
14053       898.0       483.0         2.2264
20496      1837.0       580.0         4.4964
```

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

[60]:
```
housing_cat = housing[['ocean_proximity']]
housing_cat.head(10)
```

[60]:
```
      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
```

Warning: earlier versions of the book used the `LabelEncoder` class or Pandas'

Series.factorize() method to encode string categorical attributes as integers. However, the OrdinalEncoder class that was introduced in Scikit-Learn 0.20 (see PR #10521) is preferable since it is designed for input features (X instead of labels y) and it plays well with pipelines (introduced later in this notebook). If you are using an older version of Scikit-Learn (<0.20), then you can import it from future_encoders.py instead.

[61]:
```
try:
    from sklearn.preprocessing import OrdinalEncoder
except ImportError:
    from future_encoders import OrdinalEncoder # Scikit-Learn < 0.20
```

[62]:
```
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded[:10]
```

[62]:
```
array([[1.],
       [4.],
       [1.],
       [4.],
       [0.],
       [3.],
       [0.],
       [0.],
       [0.],
       [0.]])
```

[63]:
```
ordinal_encoder.categories_
```

[63]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

Warning: earlier versions of the book used the LabelBinarizer or CategoricalEncoder classes to convert each categorical value to a one-hot vector. It is now preferable to use the OneHotEncoder class. Since Scikit-Learn 0.20 it can handle string categorical inputs (see PR #10521), not just integer categorical inputs. If you are using an older version of Scikit-Learn, you can import the new version from future_encoders.py:

[64]:
```
try:
    from sklearn.preprocessing import OrdinalEncoder # just to raise an␣
 ↪ImportError if Scikit-Learn < 0.20
    from sklearn.preprocessing import OneHotEncoder
except ImportError:
    from future_encoders import OneHotEncoder # Scikit-Learn < 0.20

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

[64]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

By default, the `OneHotEncoder` class returns a sparse array, but we can convert it to a dense array if needed by calling the `toarray()` method:

[65]:
```
housing_cat_1hot.toarray()
```

[65]:
```
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`:

[66]:
```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

[66]:
```
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.]])
```

[67]:
```
cat_encoder.categories_
```

[67]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

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

[68]:
```
housing.columns
```

[68]:
```
Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income',
       'ocean_proximity'],
      dtype='object')
```

[69]:
```
from sklearn.base import BaseEstimator, TransformerMixin

# get the right column indices: safer than hard-coding indices 3, 4, 5, 6
rooms_ix, bedrooms_ix, population_ix, household_ix = [
```

```
    list(housing.columns).index(col)
    for col in ("total_rooms", "total_bedrooms", "population", "households")]

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kwargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_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)
```

Alternatively, you can use Scikit-Learn's `FunctionTransformer` class that lets you easily create a transformer based on a transformation function (thanks to Hanmin Qin for suggesting this code). Note that we need to set `validate=False` because the data contains non-float values (`validate` will default to `False` in Scikit-Learn 0.22).

[70]:
```
from sklearn.preprocessing import FunctionTransformer

def add_extra_features(X, add_bedrooms_per_room=True):
    rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
    population_per_household = X[:, population_ix] / X[:, household_ix]
    if 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 = FunctionTransformer(add_extra_features, validate=False,
                                 kw_args={"add_bedrooms_per_room": False})
housing_extra_attribs = attr_adder.fit_transform(housing.values)
```

[71]:
```
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()
```

```
[71]:        longitude latitude housing_median_age total_rooms total_bedrooms  \
12655   -121.46    38.52                29.0      3873.0           797.0
15502   -117.23    33.09                 7.0      5320.0           855.0
2908    -119.04    35.37                44.0      1618.0           310.0
14053   -117.13    32.75                24.0      1877.0           519.0
20496    -118.7    34.28                27.0      3536.0           646.0

       population households median_income ocean_proximity rooms_per_household  \
12655      2237.0      706.0        2.1736          INLAND            5.485836
15502      2015.0      768.0        6.3373      NEAR OCEAN            6.927083
2908        667.0      300.0         2.875          INLAND            5.393333
14053       898.0      483.0        2.2264      NEAR OCEAN            3.886128
20496      1837.0      580.0        4.4964       <1H OCEAN            6.096552

       population_per_household
12655                  3.168555
15502                  2.623698
2908                   2.223333
14053                  1.859213
20496                  3.167241
```

Now let's build a pipeline for preprocessing the numerical attributes (note that we could use `CombinedAttributesAdder()` instead of `FunctionTransformer(...)` if we preferred):

```
[72]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', FunctionTransformer(add_extra_features,␣
 ↪validate=False)),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

```
[73]: housing_num_tr
```

```
[73]: 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]])
```

Warning: earlier versions of the book applied different transformations to different columns using a solution based on a `DataFrameSelector` transformer and a `FeatureUnion` (see below). It is now preferable to use the `ColumnTransformer` class that was introduced in Scikit-Learn 0.20. If you are using an older version of Scikit-Learn, you can import it from `future_encoders.py`:

[74]:
```
try:
    from sklearn.compose import ColumnTransformer
except ImportError:
    from future_encoders import ColumnTransformer # Scikit-Learn < 0.20
```

[75]:
```
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)
```

[76]:
```
housing_prepared
```

[76]:
```
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.        ]])
```

[77]:
```
housing_prepared.shape
```

[77]:
```
(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:

[78]:
```
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 (again, we could use CombinedAttributesAdder() instead of FunctionTransformer(...) if we preferred):

[79]:
```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

old_num_pipeline = Pipeline([
        ('selector', OldDataFrameSelector(num_attribs)),
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', FunctionTransformer(add_extra_features,␣
 ↪validate=False)),
        ('std_scaler', StandardScaler()),
    ])

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

[80]:
```
from sklearn.pipeline import FeatureUnion

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

[81]:
```
old_housing_prepared = old_full_pipeline.fit_transform(housing)
old_housing_prepared
```

[81]:
```
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:

[82]:
```
np.allclose(housing_prepared, old_housing_prepared)
```

[82]:
```
True
```

5 Select and train a model

[83]:
```
from sklearn.linear_model import LinearRegression

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

[83]:
```
LinearRegression()
```

[84]:
```
# 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:

[85]:
```
print("Labels:", list(some_labels))
```

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

[86]:
```
some_data_prepared
```

[86]:
```
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.        ]])
```

[87]:
```
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
```

[87]:
```
68627.87390018745
```

[88]:
```
from sklearn.metrics import mean_absolute_error

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

[88]:
```
49438.66860915802
```

[89]:
```
from sklearn.tree import DecisionTreeRegressor

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

[89]:
```
DecisionTreeRegressor(random_state=42)
```

[90]:
```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

[90]:
```
0.0
```

6 Fine-tune your model

[91]:
```python
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)
```

[92]:
```python
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
```

[93]:
```python
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
```

Note: we specify `n_estimators=10` to avoid a warning about the fact that the default value is going to change to 100 in Scikit-Learn 0.22.

[94]:
```python
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=10, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

[94]:
```
RandomForestRegressor(n_estimators=10, random_state=42)
```

[95]:
```python
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
```

[95]:
```
22413.454658589766
```

[96]:
```
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: [53519.05518628 50467.33817051 48924.16513902 53771.72056856
 50810.90996358 54876.09682033 56012.79985518 52256.88927227
 51527.73185039 55762.56008531]
Mean: 52792.92669114079
Standard deviation: 2262.8151900582
```

[97]:
```
scores = cross_val_score(lin_reg, housing_prepared, housing_labels,␣
 ↪scoring="neg_mean_squared_error", cv=10)
pd.Series(np.sqrt(-scores)).describe()
```

[97]:
```
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
```

[98]:
```
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
```

[98]:
```
111095.06635291968
```

[99]:
```
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]},
  ]
```

```python
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)
```

[99]:
```
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')
```

The best hyperparameter combination found:

[100]:
```python
grid_search.best_params_
```

[100]:
```
{'max_features': 8, 'n_estimators': 30}
```

[101]:
```python
grid_search.best_estimator_
```

[101]:
```
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:

[102]:
```python
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}
```

[103]:
```
pd.DataFrame(grid_search.cv_results_)
```

[103]:
```
    mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0        0.072480      0.002602         0.004944        0.000461
1        0.229180      0.004078         0.013138        0.000674
2        0.672476      0.001871         0.035402        0.000784
3        0.128708      0.021488         0.005058        0.000651
4        0.518286      0.063770         0.014502        0.000863
5        1.108341      0.002866         0.036428        0.002273
6        0.150071      0.001643         0.005004        0.000607
7        0.526446      0.036520         0.013911        0.001839
8        1.700406      0.299065         0.037289        0.003874
9        0.200003      0.003256         0.004759        0.000206
10       0.794997      0.154712         0.014967        0.002049
11       2.572026      0.575364         0.056408        0.038263
12       0.109426      0.005002         0.005440        0.000106
13       0.355580      0.006526         0.015074        0.000322
14       0.141472      0.001391         0.005395        0.000078
15       0.472482      0.005227         0.014986        0.000172
16       0.181207      0.003012         0.005575        0.000345
17       0.690830      0.128061         0.016427        0.001862

   param_max_features param_n_estimators param_bootstrap  \
0                   2                  3             NaN
1                   2                 10             NaN
2                   2                 30             NaN
3                   4                  3             NaN
4                   4                 10             NaN
5                   4                 30             NaN
6                   6                  3             NaN
7                   6                 10             NaN
8                   6                 30             NaN
9                   8                  3             NaN
10                  8                 10             NaN
11                  8                 30             NaN
12                  2                  3           False
13                  2                 10           False
14                  3                  3           False
15                  3                 10           False
16                  4                  3           False
17                  4                 10           False

                                      params  split0_test_score  \
0    {'max_features': 2, 'n_estimators': 3}      -4.119912e+09
1   {'max_features': 2, 'n_estimators': 10}      -2.973521e+09
2   {'max_features': 2, 'n_estimators': 30}      -2.801229e+09
3    {'max_features': 4, 'n_estimators': 3}      -3.528743e+09
```

```
4           {'max_features': 4, 'n_estimators': 10}      -2.742620e+09
5           {'max_features': 4, 'n_estimators': 30}      -2.522176e+09
6            {'max_features': 6, 'n_estimators': 3}      -3.362127e+09
7           {'max_features': 6, 'n_estimators': 10}      -2.622099e+09
8           {'max_features': 6, 'n_estimators': 30}      -2.446142e+09
9            {'max_features': 8, 'n_estimators': 3}      -3.590333e+09
10          {'max_features': 8, 'n_estimators': 10}      -2.721311e+09
11          {'max_features': 8, 'n_estimators': 30}      -2.492636e+09
12  {'bootstrap': False, 'max_features': 2, 'n_est…      -4.020842e+09
13  {'bootstrap': False, 'max_features': 2, 'n_est…      -2.901352e+09
14  {'bootstrap': False, 'max_features': 3, 'n_est…      -3.687132e+09
15  {'bootstrap': False, 'max_features': 3, 'n_est…      -2.837028e+09
16  {'bootstrap': False, 'max_features': 4, 'n_est…      -3.549428e+09
17  {'bootstrap': False, 'max_features': 4, 'n_est…      -2.692499e+09

    split1_test_score  …  mean_test_score  std_test_score  rank_test_score  \
0       -3.723465e+09  …    -4.082592e+09    1.867375e+08               18
1       -2.810319e+09  …    -3.015803e+09    1.139808e+08               11
2       -2.671474e+09  …    -2.796915e+09    7.980892e+07                9
3       -3.490303e+09  …    -3.609050e+09    1.375683e+08               16
4       -2.609311e+09  …    -2.755726e+09    1.182604e+08                7
5       -2.440241e+09  …    -2.518759e+09    8.488084e+07                3
6       -3.311863e+09  …    -3.371513e+09    1.378086e+08               13
7       -2.669655e+09  …    -2.654240e+09    6.967978e+07                5
8       -2.446594e+09  …    -2.496981e+09    7.357046e+07                2
9       -3.232664e+09  …    -3.468718e+09    1.293758e+08               14
10      -2.675886e+09  …    -2.752030e+09    6.258030e+07                6
11      -2.444818e+09  …    -2.489909e+09    7.086483e+07                1
12      -3.951861e+09  …    -3.891485e+09    8.648595e+07               17
13      -3.036875e+09  …    -2.967697e+09    4.582448e+07               10
14      -3.446245e+09  …    -3.596953e+09    8.011960e+07               15
15      -2.619558e+09  …    -2.783044e+09    8.862580e+07                8
16      -3.318176e+09  …    -3.344440e+09    1.099355e+08               12
17      -2.542704e+09  …    -2.629472e+09    8.510266e+07                4

    split0_train_score  split1_train_score  split2_train_score  \
0        -1.155630e+09       -1.089726e+09       -1.153843e+09
1        -5.982947e+08       -5.904781e+08       -6.123850e+08
2        -4.412567e+08       -4.326398e+08       -4.553722e+08
3        -9.782368e+08       -9.806455e+08       -1.003780e+09
4        -5.063215e+08       -5.257983e+08       -5.081984e+08
5        -3.776568e+08       -3.902106e+08       -3.885042e+08
6        -8.909397e+08       -9.583733e+08       -9.000201e+08
7        -4.939906e+08       -5.145996e+08       -5.023512e+08
8        -3.760968e+08       -3.876636e+08       -3.875307e+08
9        -9.505012e+08       -9.166119e+08       -9.033910e+08
10       -4.998373e+08       -4.997970e+08       -5.099880e+08
```

```
11      -3.801679e+08       -3.832972e+08       -3.823818e+08
12      -0.000000e+00       -4.306828e+01       -1.051392e+04
13      -0.000000e+00       -3.876145e+00       -9.462528e+02
14      -0.000000e+00       -0.000000e+00       -0.000000e+00
15      -0.000000e+00       -0.000000e+00       -0.000000e+00
16      -0.000000e+00       -0.000000e+00       -0.000000e+00
17      -0.000000e+00       -0.000000e+00       -0.000000e+00

    split3_train_score  split4_train_score  mean_train_score  std_train_score
0         -1.118149e+09       -1.093446e+09     -1.122159e+09     2.834288e+07
1         -5.727681e+08       -5.905210e+08     -5.928894e+08     1.284978e+07
2         -4.320746e+08       -4.311606e+08     -4.385008e+08     9.184397e+06
3         -1.016515e+09       -1.011270e+09     -9.980896e+08     1.577372e+07
4         -5.174405e+08       -5.282066e+08     -5.171931e+08     8.882622e+06
5         -3.830866e+08       -3.894779e+08     -3.857872e+08     4.774229e+06
6         -8.964731e+08       -9.151927e+08     -9.121998e+08     2.444837e+07
7         -4.959467e+08       -5.147087e+08     -5.043194e+08     8.880106e+06
8         -3.760938e+08       -3.861056e+08     -3.826981e+08     5.418747e+06
9         -9.070642e+08       -9.459386e+08     -9.247014e+08     1.973471e+07
10        -5.047868e+08       -5.348043e+08     -5.098427e+08     1.303601e+07
11        -3.778452e+08       -3.817589e+08     -3.810902e+08     1.916605e+06
12        -0.000000e+00       -0.000000e+00     -2.111398e+03     4.201294e+03
13        -0.000000e+00       -0.000000e+00     -1.900258e+02     3.781165e+02
14        -0.000000e+00       -0.000000e+00      0.000000e+00     0.000000e+00
15        -0.000000e+00       -0.000000e+00      0.000000e+00     0.000000e+00
16        -0.000000e+00       -0.000000e+00      0.000000e+00     0.000000e+00
17        -0.000000e+00       -0.000000e+00      0.000000e+00     0.000000e+00

[18 rows x 23 columns]
```

[104]:
```python
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)
```

[104]:
```
RandomizedSearchCV(cv=5, estimator=RandomForestRegressor(random_state=42),
                   param_distributions={'max_features':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f3be989d4c0>,
```

```
                                                'n_estimators':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f3be99fa490>},
                   random_state=42, scoring='neg_mean_squared_error')
```

[105]:
```python
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}
```

[106]:
```python
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

[106]:
```
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])
```

[107]:
```python
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)
```

[107]:
```
[(0.3790092248170967, 'median_income'),
 (0.16570630316895876, 'INLAND'),
 (0.10703132208204354, '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'),
 (0.014934655161887776, 'households'),
 (0.006792660074259966, '<1H OCEAN'),
```

```
 (0.0030281610628962747, 'NEAR OCEAN'),
 (0.0015247327555504937, 'NEAR BAY'),
 (7.834806602687504e-05, 'ISLAND')]
```

[108]:
```
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)
```

[109]:
```
final_rmse
```

[109]:
```
47873.26095812988
```

We can compute a 95% confidence interval for the test RMSE:

[110]:
```
from scipy import stats
```

[111]:
```
confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
mean = squared_errors.mean()
m = len(squared_errors)

np.sqrt(stats.t.interval(confidence, m - 1,
                         loc=np.mean(squared_errors),
                         scale=stats.sem(squared_errors)))
```

[111]:
```
array([45893.36082829, 49774.46796717])
```

We could compute the interval manually like this:

[112]:
```
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)
```

[112]:
```
(45893.360828285535, 49774.46796717361)
```

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

[113]:
```
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)
```

[113]: (45893.9540110131, 49773.921030650374)

7 Extra material

7.1 A full pipeline with both preparation and prediction

[114]:
```
full_pipeline_with_predictor = Pipeline([
        ("preparation", full_pipeline),
        ("linear", LinearRegression())
    ])

full_pipeline_with_predictor.fit(housing, housing_labels)
full_pipeline_with_predictor.predict(some_data)
```

[114]:
```
array([ 85657.90192014, 305492.60737488, 152056.46122456, 186095.70946094,
       244550.67966089])
```

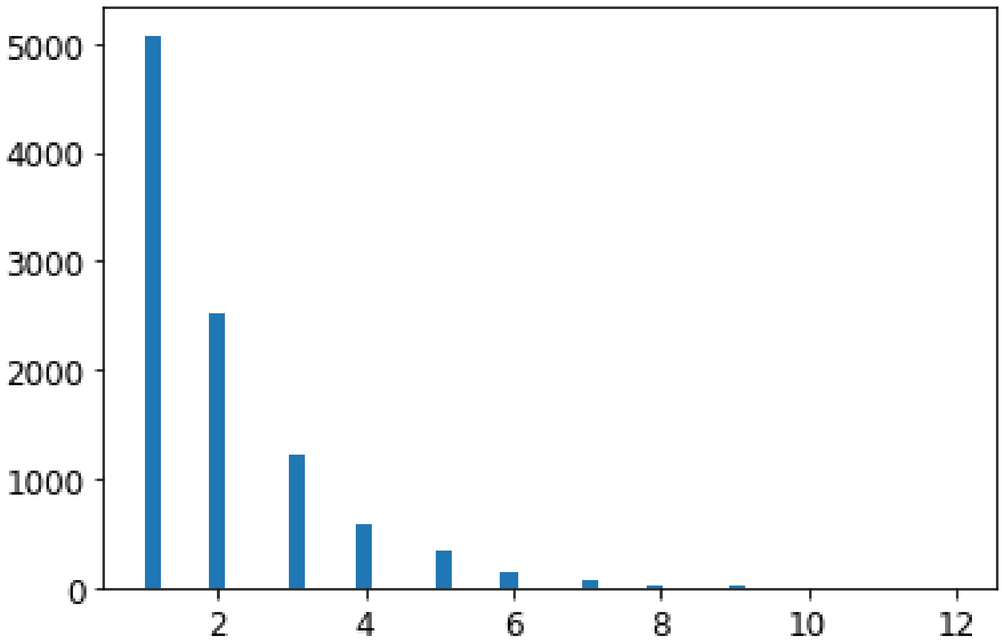
7.2 Model persistence using joblib

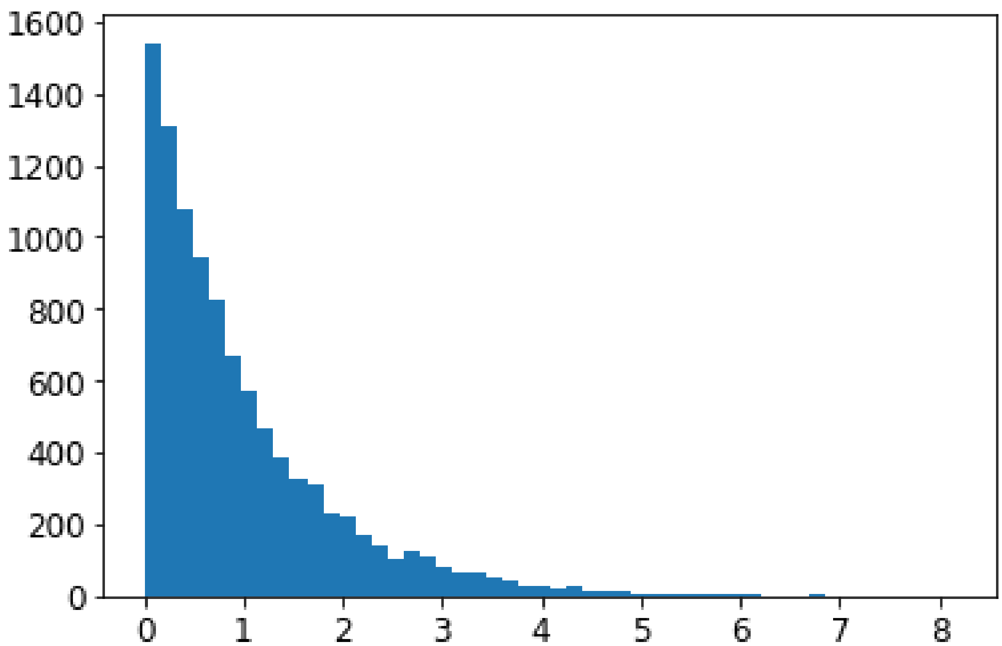
[115]:
```
my_model = full_pipeline_with_predictor
```

[116]:
```
#from sklearn.externals import joblib # deprecated, use import joblib instead
import joblib

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

7.3 Example SciPy distributions for `RandomizedSearchCV`

[117]:
```
from scipy.stats import geom, expon
geom_distrib=geom(0.5).rvs(10000, random_state=42)
expon_distrib=expon(scale=1).rvs(10000, random_state=42)
plt.hist(geom_distrib, bins=50)
plt.show()
plt.hist(expon_distrib, bins=50)
plt.show()
```


5000
4000
3000
2000
1000
0
2
4
6
8
10
12


1600
1400
1200
1000
800
600
400
200
0
0
1
2
3
4
5
6
7
8

8 Exercise solutions

8.1 1.

Question: Try a Support Vector Machine regressor (`sklearn.svm.SVR`), with various hyperparameters such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Don't worry about what these hyperparameters mean for now. How does the best `SVR` predictor perform?

[118]:
```
from sklearn.model_selection import GridSearchCV

param_grid = [
        {'kernel': ['linear'], 'C': [10., 30., 100., 300., 1000., 3000., 10000.
 ↪, 30000.0]},
        {'kernel': ['rbf'], 'C': [1.0, 3.0, 10., 30., 100., 300., 1000.0],
         'gamma': [0.01, 0.03, 0.1, 0.3, 1.0, 3.0]},
    ]

svm_reg = SVR()
grid_search = GridSearchCV(svm_reg, param_grid, cv=5,␣
 ↪scoring='neg_mean_squared_error', verbose=2, n_jobs=4)
grid_search.fit(housing_prepared, housing_labels)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

[118]:
```
GridSearchCV(cv=5, estimator=SVR(), n_jobs=4,
             param_grid=[{'C': [10.0, 30.0, 100.0, 300.0, 1000.0, 3000.0,
                                10000.0, 30000.0],
                          'kernel': ['linear']},
                         {'C': [1.0, 3.0, 10.0, 30.0, 100.0, 300.0, 1000.0],
                          'gamma': [0.01, 0.03, 0.1, 0.3, 1.0, 3.0],
                          'kernel': ['rbf']}],
             scoring='neg_mean_squared_error', verbose=2)
```

The best model achieves the following score (evaluated using 5-fold cross validation):

[119]:
```
negative_mse = grid_search.best_score_
rmse = np.sqrt(-negative_mse)
rmse
```

[119]:
```
70286.61835383571
```

That's much worse than the `RandomForestRegressor`. Let's check the best hyperparameters found:

[120]:
```
grid_search.best_params_
```

[120]:
```
{'C': 30000.0, 'kernel': 'linear'}
```

The linear kernel seems better than the RBF kernel. Notice that the value of `C` is the maximum tested value. When this happens you definitely want to launch the grid search again with higher values for C (removing the smallest values), because it is likely that higher values of C will be better.

8.2 2.

Question: Try replacing GridSearchCV with RandomizedSearchCV.

[121]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import expon, reciprocal

# see https://docs.scipy.org/doc/scipy/reference/stats.html
# for `expon()` and `reciprocal()` documentation and more probability␣
↪distribution functions.

# Note: gamma is ignored when kernel is "linear"
param_distribs = {
        'kernel': ['linear', 'rbf'],
        'C': reciprocal(20, 200000),
        'gamma': expon(scale=1.0),
    }

svm_reg = SVR()
rnd_search = RandomizedSearchCV(svm_reg, param_distributions=param_distribs,
                                n_iter=50, cv=5,␣
↪scoring='neg_mean_squared_error',
                                verbose=2, n_jobs=4, random_state=42)
rnd_search.fit(housing_prepared, housing_labels)
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
```

[121]:
```
RandomizedSearchCV(cv=5, estimator=SVR(), n_iter=50, n_jobs=4,
                   param_distributions={'C':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f3be9c1b880>,
                                        'gamma':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f3be9b55640>,
                                        'kernel': ['linear', 'rbf']},
                   random_state=42, scoring='neg_mean_squared_error',
                   verbose=2)
```

The best model achieves the following score (evaluated using 5-fold cross validation):

[122]:
```
negative_mse = rnd_search.best_score_
rmse = np.sqrt(-negative_mse)
rmse
```

[122]:
```
54751.69009488048
```

Now this is much closer to the performance of the RandomForestRegressor (but not quite there yet). Let's check the best hyperparameters found:

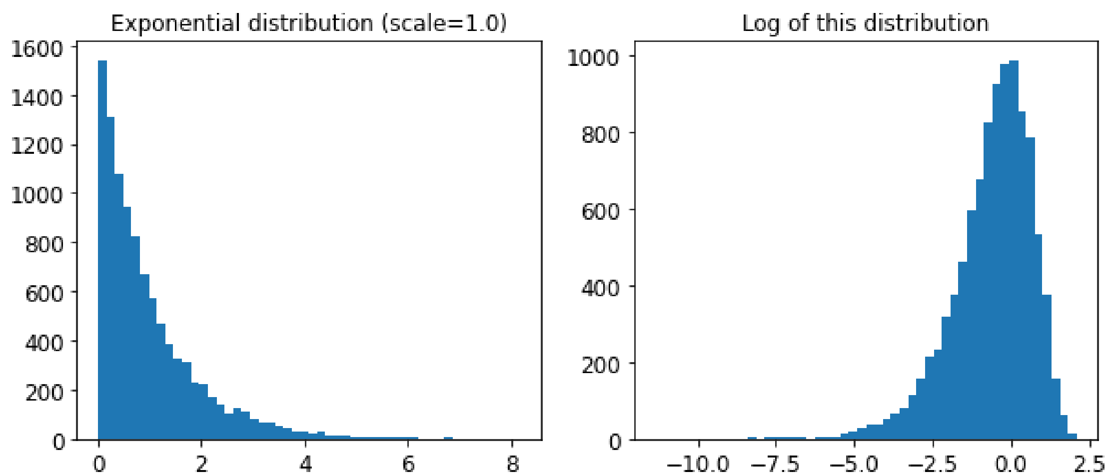
[123]:
```
rnd_search.best_params_
```

[123]: {'C': 157055.10989448498, 'gamma': 0.26497040005002437, 'kernel': 'rbf'}

This time the search found a good set of hyperparameters for the RBF kernel. Randomized search tends to find better hyperparameters than grid search in the same amount of time.

Let's look at the exponential distribution we used, with `scale=1.0`. Note that some samples are much larger or smaller than 1.0, but when you look at the log of the distribution, you can see that most values are actually concentrated roughly in the range of exp(-2) to exp(+2), which is about 0.1 to 7.4.

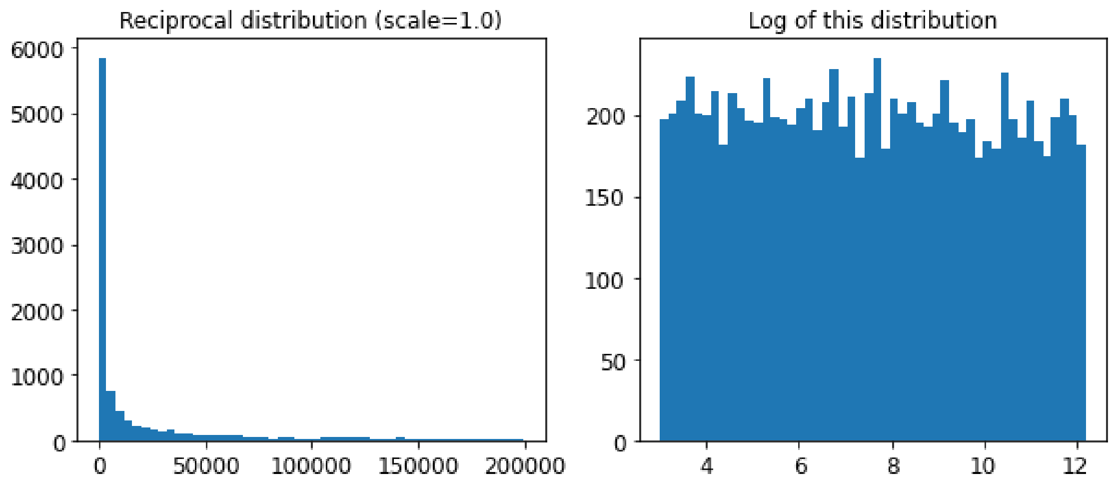
[124]:
```
expon_distrib = expon(scale=1.)
samples = expon_distrib.rvs(10000, random_state=42)
plt.figure(figsize=(10, 4))
plt.subplot(121)
plt.title("Exponential distribution (scale=1.0)")
plt.hist(samples, bins=50)
plt.subplot(122)
plt.title("Log of this distribution")
plt.hist(np.log(samples), bins=50)
plt.show()
```



The distribution we used for `C` looks quite different: the scale of the samples is picked from a uniform distribution within a given range, which is why the right graph, which represents the log of the samples, looks roughly constant. This distribution is useful when you don't have a clue of what the target scale is:

[125]:
```
reciprocal_distrib = reciprocal(20, 200000)
samples = reciprocal_distrib.rvs(10000, random_state=42)
plt.figure(figsize=(10, 4))
plt.subplot(121)
```

```
plt.title("Reciprocal distribution (scale=1.0)")
plt.hist(samples, bins=50)
plt.subplot(122)
plt.title("Log of this distribution")
plt.hist(np.log(samples), bins=50)
plt.show()
```



The reciprocal distribution is useful when you have no idea what the scale of the hyperparameter should be (indeed, as you can see on the figure on the right, all scales are equally likely, within the given range), whereas the exponential distribution is best when you know (more or less) what the scale of the hyperparameter should be.

8.3 3.

Question: Try adding a transformer in the preparation pipeline to select only the most important attributes.

[126]:
```
from sklearn.base import BaseEstimator, TransformerMixin

def indices_of_top_k(arr, k):
    return np.sort(np.argpartition(np.array(arr), -k)[-k:])

class TopFeatureSelector(BaseEstimator, TransformerMixin):
    def __init__(self, feature_importances, k):
        self.feature_importances = feature_importances
        self.k = k
    def fit(self, X, y=None):
        self.feature_indices_ = indices_of_top_k(self.feature_importances, self.
 ↪k)
        return self
    def transform(self, X):
```

```
        return X[:, self.feature_indices_]
```

Note: this feature selector assumes that you have already computed the feature importances somehow (for example using a `RandomForestRegressor`). You may be tempted to compute them directly in the `TopFeatureSelector`'s `fit()` method, however this would likely slow down grid/randomized search since the feature importances would have to be computed for every hyperparameter combination (unless you implement some sort of cache).

Let's define the number of top features we want to keep:

[127]:
```
k = 5
```

Now let's look for the indices of the top k features:

[128]:
```
top_k_feature_indices = indices_of_top_k(feature_importances, k)
top_k_feature_indices
```

[128]:
```
array([ 0,  1,  7,  9, 12])
```

[129]:
```
np.array(attributes)[top_k_feature_indices]
```

[129]:
```
array(['longitude', 'latitude', 'median_income', 'pop_per_hhold',
       'INLAND'], dtype='<U18')
```

Let's double check that these are indeed the top k features:

[130]:
```
sorted(zip(feature_importances, attributes), reverse=True)[:k]
```

[130]:
```
[(0.3790092248170967, 'median_income'),
 (0.16570630316895876, 'INLAND'),
 (0.10703132208204354, 'pop_per_hhold'),
 (0.06965425227942929, 'longitude'),
 (0.0604213840080722, 'latitude')]
```

Looking good… Now let's create a new pipeline that runs the previously defined preparation pipeline, and adds top k feature selection:

[131]:
```
preparation_and_feature_selection_pipeline = Pipeline([
    ('preparation', full_pipeline),
    ('feature_selection', TopFeatureSelector(feature_importances, k))
])
```

[132]:
```
housing_prepared_top_k_features = preparation_and_feature_selection_pipeline.
 ↪fit_transform(housing)
```

Let's look at the features of the first 3 instances:

[133]:
```
housing_prepared_top_k_features[0:3]
```

[133]:
```
array([[-0.94135046,  1.34743822, -0.8936472 ,  0.00622264,  1.        ],
       [ 1.17178212, -1.19243966,  1.292168  , -0.04081077,  0.        ],
       [ 0.26758118, -0.1259716 , -0.52543365, -0.07537122,  1.        ]])
```

Now let's double check that these are indeed the top k features:

[134]:
```
housing_prepared[0:3, top_k_feature_indices]
```

[134]:
```
array([[-0.94135046,  1.34743822, -0.8936472 ,  0.00622264,  1.        ],
       [ 1.17178212, -1.19243966,  1.292168  , -0.04081077,  0.        ],
       [ 0.26758118, -0.1259716 , -0.52543365, -0.07537122,  1.        ]])
```

Works great! :)

8.4 4.

Question: Try creating a single pipeline that does the full data preparation plus the final prediction.

[135]:
```
prepare_select_and_predict_pipeline = Pipeline([
    ('preparation', full_pipeline),
    ('feature_selection', TopFeatureSelector(feature_importances, k)),
    ('svm_reg', SVR(**rnd_search.best_params_))
])
```

[136]:
```
prepare_select_and_predict_pipeline.fit(housing, housing_labels)
```

[136]:
```
Pipeline(steps=[('preparation',
                 ColumnTransformer(transformers=[('num',
                                                  Pipeline(steps=[('imputer',
SimpleImputer(strategy='median')),
('attribs_adder',
FunctionTransformer(func=<function add_extra_features at 0x7f3be92d8040>)),
                                                                  ('std_scaler',
StandardScaler())]),
                                                  ['longitude', 'latitude',
                                                   'housing_median_age',
                                                   'total_rooms',
                                                   'total_bedrooms',
                                                   'population', 'househ…
                 TopFeatureSelector(feature_importances=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]),
                                    k=5)),
                ('svm_reg',
                 SVR(C=157055.10989448498, gamma=0.26497040005002437))])
```

Let's try the full pipeline on a few instances:

[137]:
```
some_data = housing.iloc[:4]
some_labels = housing_labels.iloc[:4]

print("Predictions:\t", prepare_select_and_predict_pipeline.predict(some_data))
print("Labels:\t\t", list(some_labels))
```

```
Predictions:     [ 83384.49158095 299407.90439234  92272.03345144
150173.16199041]
Labels:          [72100.0, 279600.0, 82700.0, 112500.0]
```

Well, the full pipeline seems to work fine. Of course, the predictions are not fantastic: they would be better if we used the best `RandomForestRegressor` that we found earlier, rather than the best `SVR`.

8.5 5.

Question: Automatically explore some preparation options using `GridSearchCV`.

[138]:
```
param_grid = [{
    'preparation__num__imputer__strategy': ['mean', 'median', 'most_frequent'],
    'feature_selection__k': list(range(1, len(feature_importances) + 1))
}]

grid_search_prep = GridSearchCV(prepare_select_and_predict_pipeline,␣
↪param_grid, cv=5,
                                scoring='neg_mean_squared_error', verbose=2,␣
↪n_jobs=4)
grid_search_prep.fit(housing, housing_labels)
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits

/usr/local/lib/python3.8/dist-
packages/sklearn/model_selection/_validation.py:372: FitFailedWarning:
9 fits failed out of a total of 240.
The score on these train-test partitions for these parameters will be set to
nan.
If these failures are not expected, you can try to debug them by setting
error_score='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
9 fits failed with the following error:
Traceback (most recent call last):
  File "/usr/local/lib/python3.8/dist-
packages/sklearn/model_selection/_validation.py", line 680, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "/usr/local/lib/python3.8/dist-packages/sklearn/pipeline.py", line 390,
in fit
    Xt = self._fit(X, y, **fit_params_steps)
```

```
  File "/usr/local/lib/python3.8/dist-packages/sklearn/pipeline.py", line 348,
in _fit
    X, fitted_transformer = fit_transform_one_cached(
  File "/usr/local/lib/python3.8/dist-packages/joblib/memory.py", line 349, in
__call__
    return self.func(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/sklearn/pipeline.py", line 893,
in _fit_transform_one
    res = transformer.fit_transform(X, y, **fit_params)
  File "/usr/local/lib/python3.8/dist-packages/sklearn/base.py", line 855, in
fit_transform
    return self.fit(X, y, **fit_params).transform(X)
  File "<ipython-input-126-6a801ecaa128>", line 14, in transform
IndexError: index 15 is out of bounds for axis 1 with size 15

  warnings.warn(some_fits_failed_message, FitFailedWarning)
/usr/local/lib/python3.8/dist-packages/sklearn/model_selection/_search.py:969:
UserWarning: One or more of the test scores are non-finite: [nan nan nan nan nan
nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan nan nan nan nan]
  warnings.warn(
```

[138]:

```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('preparation',
                                        ColumnTransformer(transformers=[('num',
Pipeline(steps=[('imputer',
          SimpleImputer(strategy='median')),
         ('attribs_adder',
          FunctionTransformer(func=<function add_extra_features at
0x7f3be92d8040>)),
         ('std_scaler',
          StandardScaler())]),
['longitude',
'latitude',
'housing_median_age',
'total_rooms',
'total_be…
       5.47789150e-02, 1.07031322e-01, 4.82031213e-02, 6.79266007e-03,
       1.65706303e-01, 7.83480660e-05, 1.52473276e-03, 3.02816106e-03]),
                                                         k=5)),
                                       ('svm_reg',
                                        SVR(C=157055.10989448498,
                                            gamma=0.26497040005002437))]),
             n_jobs=4,
             param_grid=[{'feature_selection__k': [1, 2, 3, 4, 5, 6, 7, 8, 9,
                                                   10, 11, 12, 13, 14, 15, 16],
```

```
                             'preparation__num__imputer__strategy': ['mean',
                                                                     'median',
'most_frequent']}],
             scoring='neg_mean_squared_error', verbose=2)
```

[139]:
```
grid_search_prep.best_params_
```

[139]:
```
{'feature_selection__k': 1, 'preparation__num__imputer__strategy': 'mean'}
```

The best imputer strategy is `most_frequent` and apparently almost all features are useful (15 out of 16). The last one (`ISLAND`) seems to just add some noise.

Congratulations! You already know quite a lot about Machine Learning. :)