** End-to-end Machine Learning project**

housing_Price_Prediction

Download the Data

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing = load_housing_data()
```

Take a Quick Look at the Data Structure

```
housing.head()
```

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

Next steps: Generate code with housing | View recommended plots | New interactive sheet

```
1 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
```

```
1 housing["ocean_proximity"].value_counts()
```

ocean_proximity	count
<1H OCEAN	9136
INLAND	6551
NEAR OCEAN	2658
NEAR BAY	2290
ISLAND	5

dtype: int64

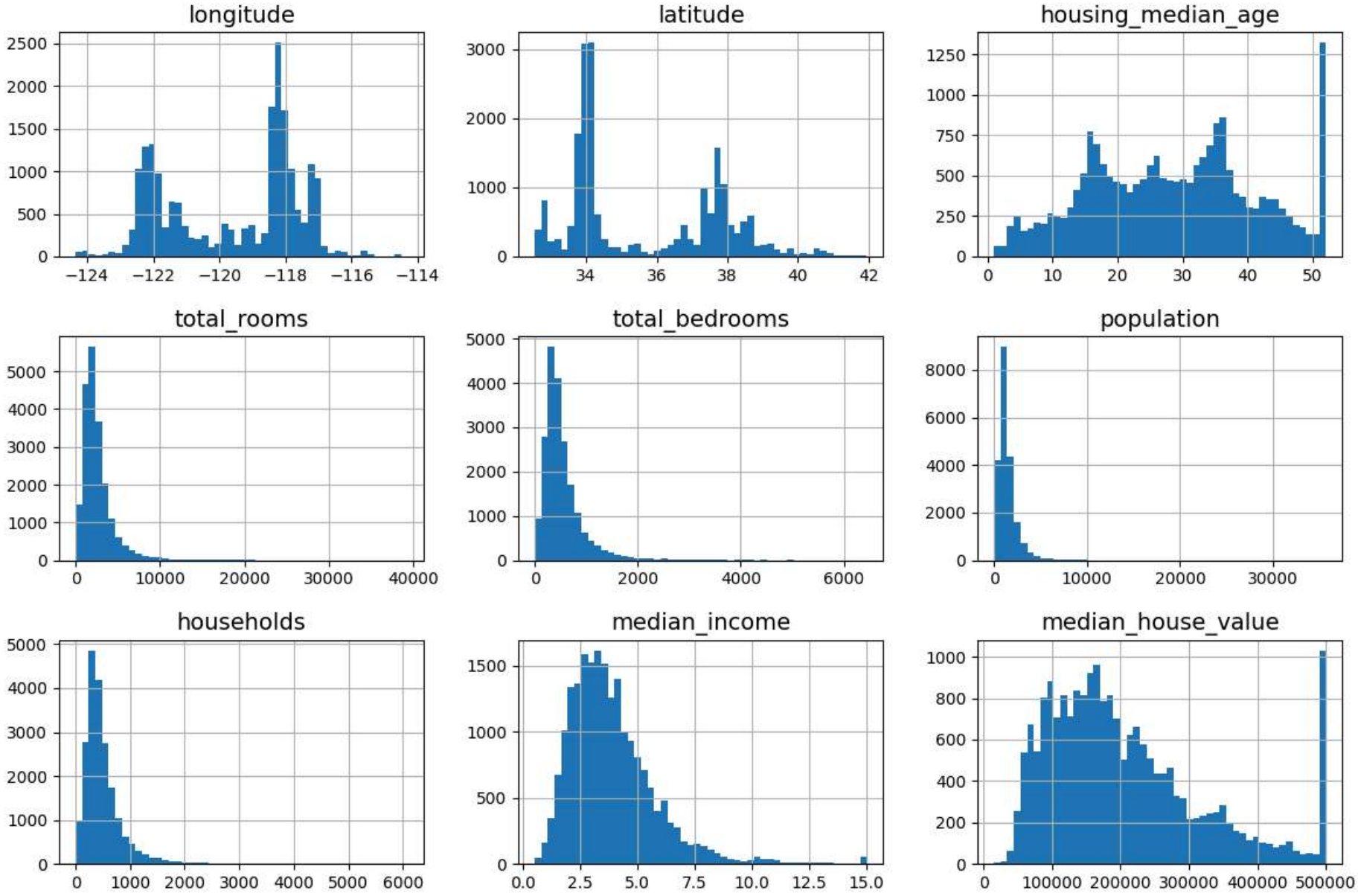
```
1 housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_hou
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	2064
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	20685
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	11539
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	1499
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	11960
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	17970
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	26472
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	50000

The following cell is not shown either in the book. It creates the `images/end_to_end_project` folder (if it doesn't already exist), and it defines the `save_fig()` function which is used through this notebook to save the figures in high-res for the book.

```
# extra code – code to save the figures as high-res PNGs for the book

IMAGES_PATH = Path() / "images" / "end_to_end_project"
IMAGES_PATH.mkdir(parents=True, exist_ok=True)

def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
    path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

```
import matplotlib.pyplot as plt

# extra code – the next 5 lines define the default font sizes
plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)

housing.hist(bins=50, figsize=(12, 8))
save_fig("attribute_histogram_plots")  # extra code
plt.show()
```



Create a Test Set

```
import numpy as np

def shuffle_and_split_data(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

```
train_set, test_set = shuffle_and_split_data(housing, 0.2)
len(train_set)
```

```
16512
```

```
len(test_set)
```

```
4128
```

To ensure that this notebook's outputs remain the same every time we run it, we need to set the random seed:

```
np.random.seed(42)
```

```
from zlib import crc32

def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32

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

```
housing_with_id = housing.reset_index()  # adds an `index` column
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "index")
```

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

```
from sklearn.model_selection import train_test_split

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

```
test_set["total_bedrooms"].isnull().sum()
```

```
44
```

To find the probability that a random sample of 1,000 people contains less than 48.5% female or more than 53.5% female when the population's female ratio is 51.1%, we use the binomial distribution. The `cdf()` method of the binomial distribution gives us the probability that the number of females will be equal or less than the given value.

```
# extra code – shows how to compute the 10.7% proba of getting a bad sample

from scipy.stats import binom

sample_size = 1000
ratio_female = 0.511
proba_too_small = binom(sample_size, ratio_female).cdf(485 - 1)
proba_too_large = 1 - binom(sample_size, ratio_female).cdf(535)
print(proba_too_small + proba_too_large)
```

```
0.10736798530929942
```

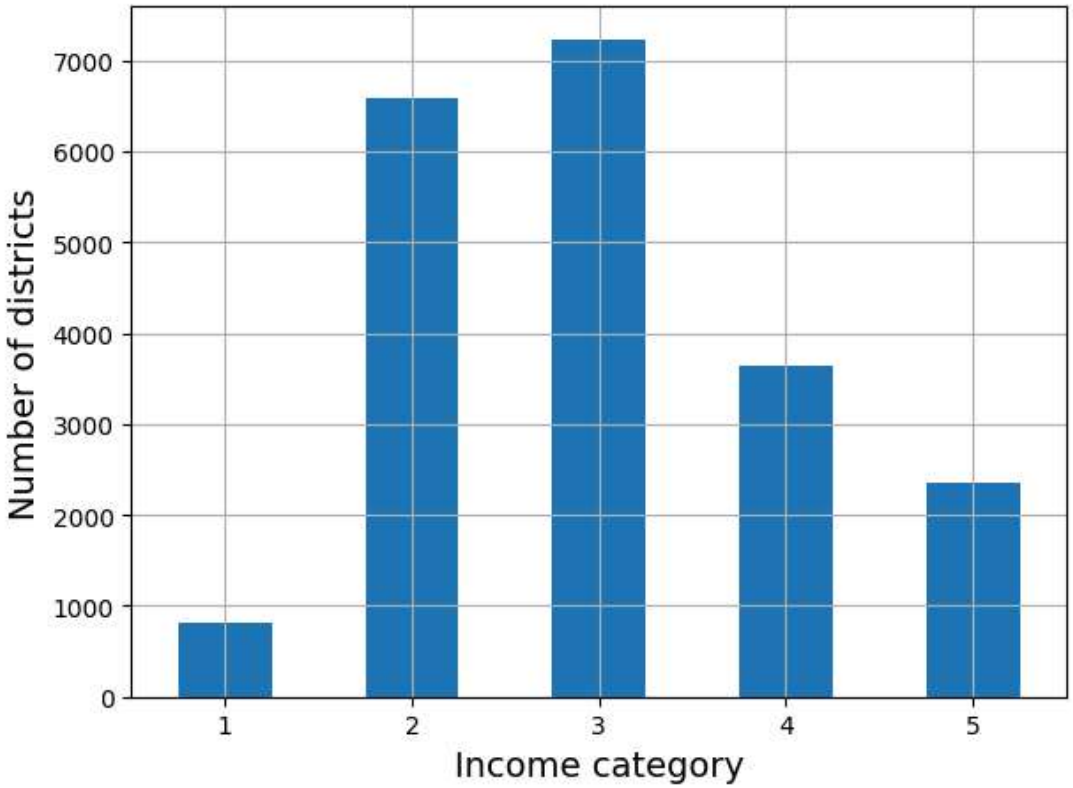
If you prefer simulations over maths, here's how you could get roughly the same result:

```
# extra code – shows another way to estimate the probability of bad sample

np.random.seed(42)

samples = (np.random.rand(100_000, sample_size) < ratio_female).sum(axis=1)
((samples < 485) | (samples > 535)).mean()
```

```
0.1071
```

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

```
housing["income_cat"].value_counts().sort_index().plot.bar(rot=0, grid=True)
plt.xlabel("Income category")
plt.ylabel("Number of districts")
save_fig("housing_income_cat_bar_plot")  # extra code
plt.show()
```




```
from sklearn.model_selection import StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
strat_splits = []
for train_index, test_index in splitter.split(housing, housing["income_cat"]):
    strat_train_set_n = housing.iloc[train_index]
    strat_test_set_n = housing.iloc[test_index]
    strat_splits.append([strat_train_set_n, strat_test_set_n])
```

```
strat_train_set, strat_test_set = strat_splits[0]
```

It's much shorter to get a single stratified split:

```
strat_train_set, strat_test_set = train_test_split(
    housing, test_size=0.2, stratify=housing["income_cat"], random_state=42)
```

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

income_cat	count
3	0.350533
2	0.318798
4	0.176357
5	0.114341
1	0.039971

dtype: float64

```
# extra code – computes the data for Figure 2–10

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.index.name = "Income Category"
compare_props["Strat. Error %"] = (compare_props["Stratified %"] /
                                   compare_props["Overall %"] - 1)
compare_props["Rand. Error %"] = (compare_props["Random %"] /
                                  compare_props["Overall %"] - 1)
(compare_props * 100).round(2)
```

Income Category	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

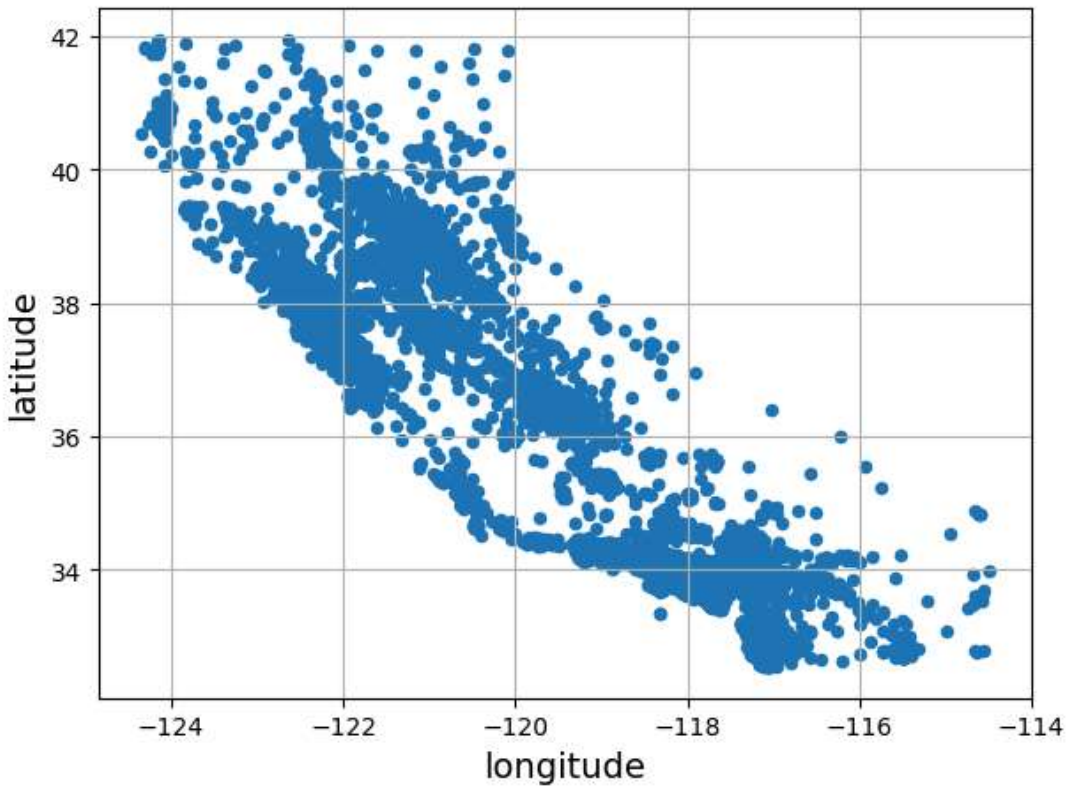
```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Discover and Visualize the Data to Gain Insights

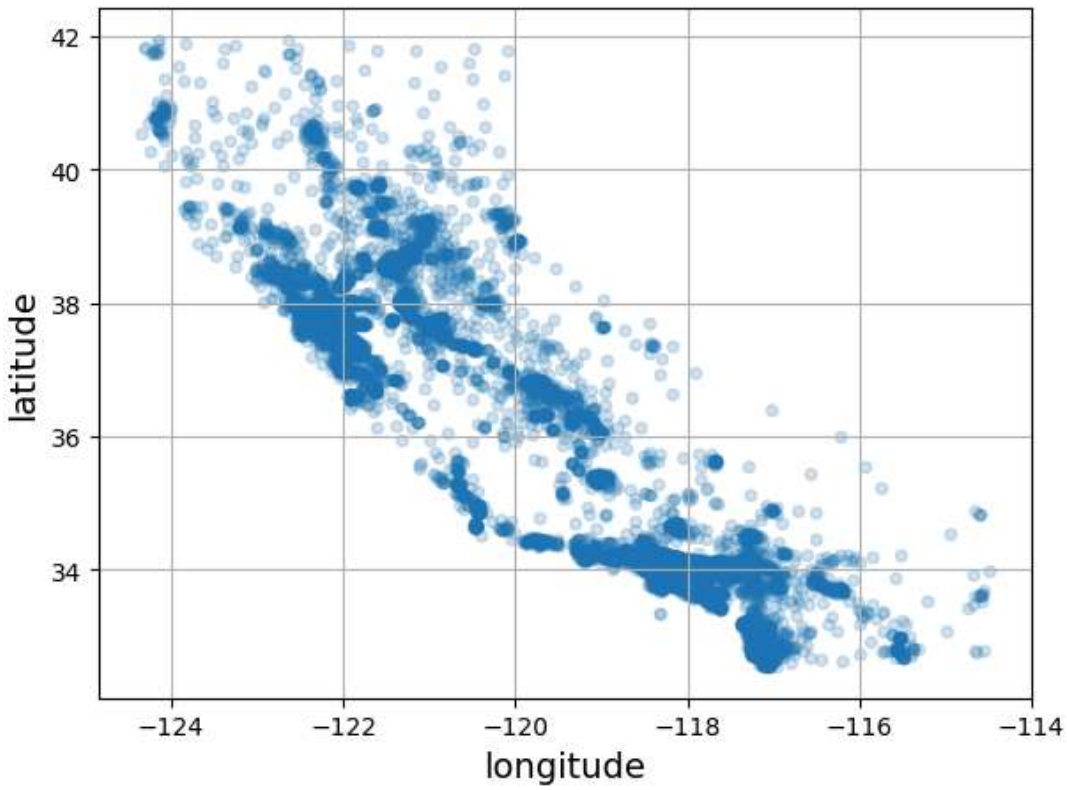
```
housing = strat_train_set.copy()
```

Visualizing Geographical Data

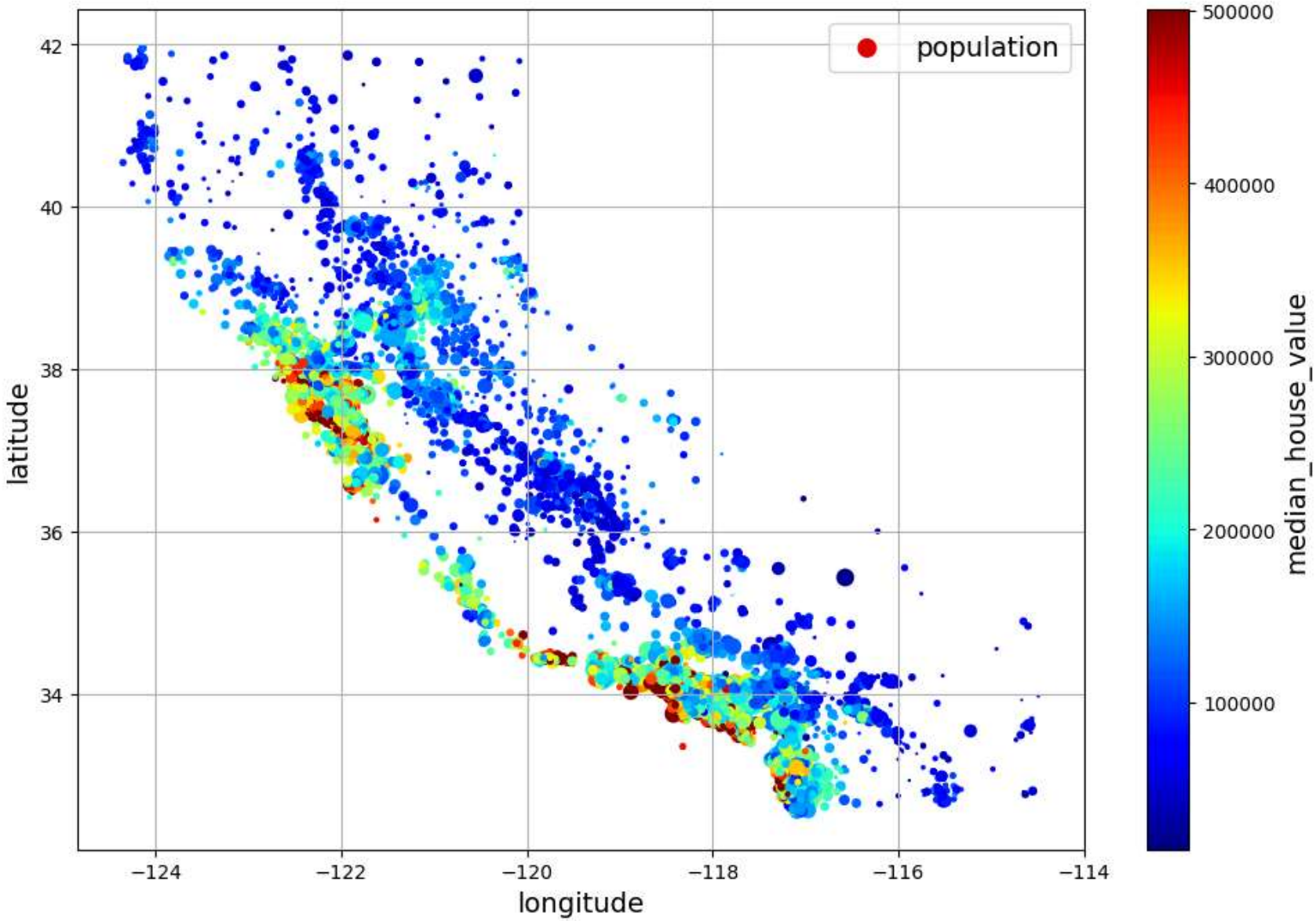
```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)
save_fig("bad_visualization_plot")  # extra code
plt.show()
```




```
1 housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)
2 save_fig("better_visualization_plot")  # extra code
3 plt.show()
```

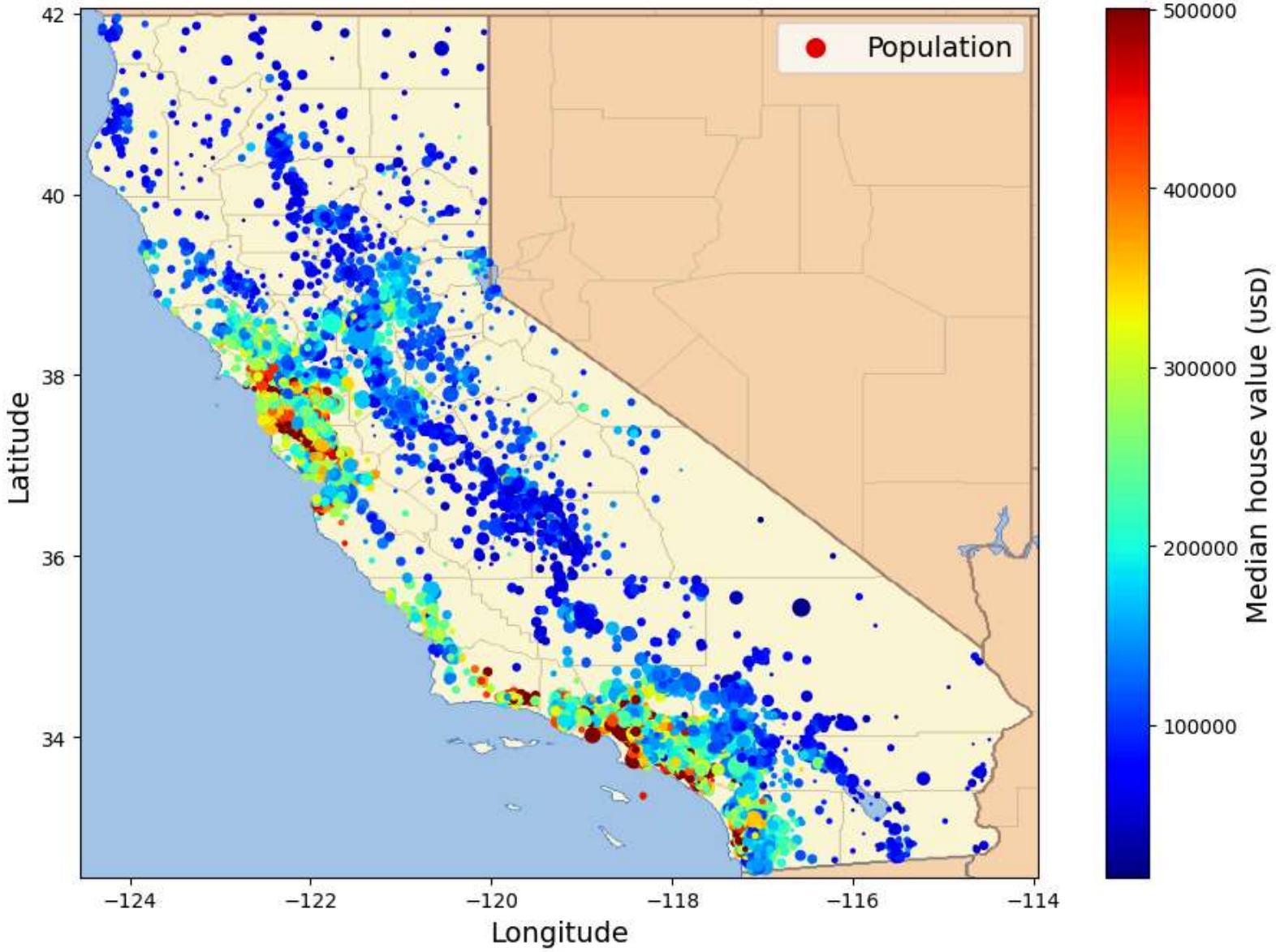



```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))
save_fig("housing_prices_scatterplot")  # extra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


population
latitude
longitude
median_house_value
42
40
38
36
34
−124
−122
−120
−118
−116
−114
500000
400000
300000
200000
100000

```
# extra code – this cell generates the first figure in the chapter

# Download the California image
filename = "california.png"
if not (IMAGES_PATH / filename).is_file():
    homl3_root = "https://github.com/ageron/handson-ml3/raw/main/"
    url = homl3_root + "images/end_to_end_project/" + filename
    print("Downloading", filename)
    urllib.request.urlretrieve(url, IMAGES_PATH / filename)

housing_renamed = housing.rename(columns={
    "latitude": "Latitude", "longitude": "Longitude",
    "population": "Population",
    "median_house_value": "Median house value (usd)"})
housing_renamed.plot(
             kind="scatter", x="Longitude", y="Latitude",
             s=housing_renamed["Population"] / 100, label="Population",
             c="Median house value (usd)", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))

california_img = plt.imread(IMAGES_PATH / filename)
axis = -124.55, -113.95, 32.45, 42.05
plt.axis(axis)
plt.imshow(california_img, extent=axis)

save_fig("california_housing_prices_plot")
plt.show()
```

```
Downloading california.png
```



Looking for Correlations

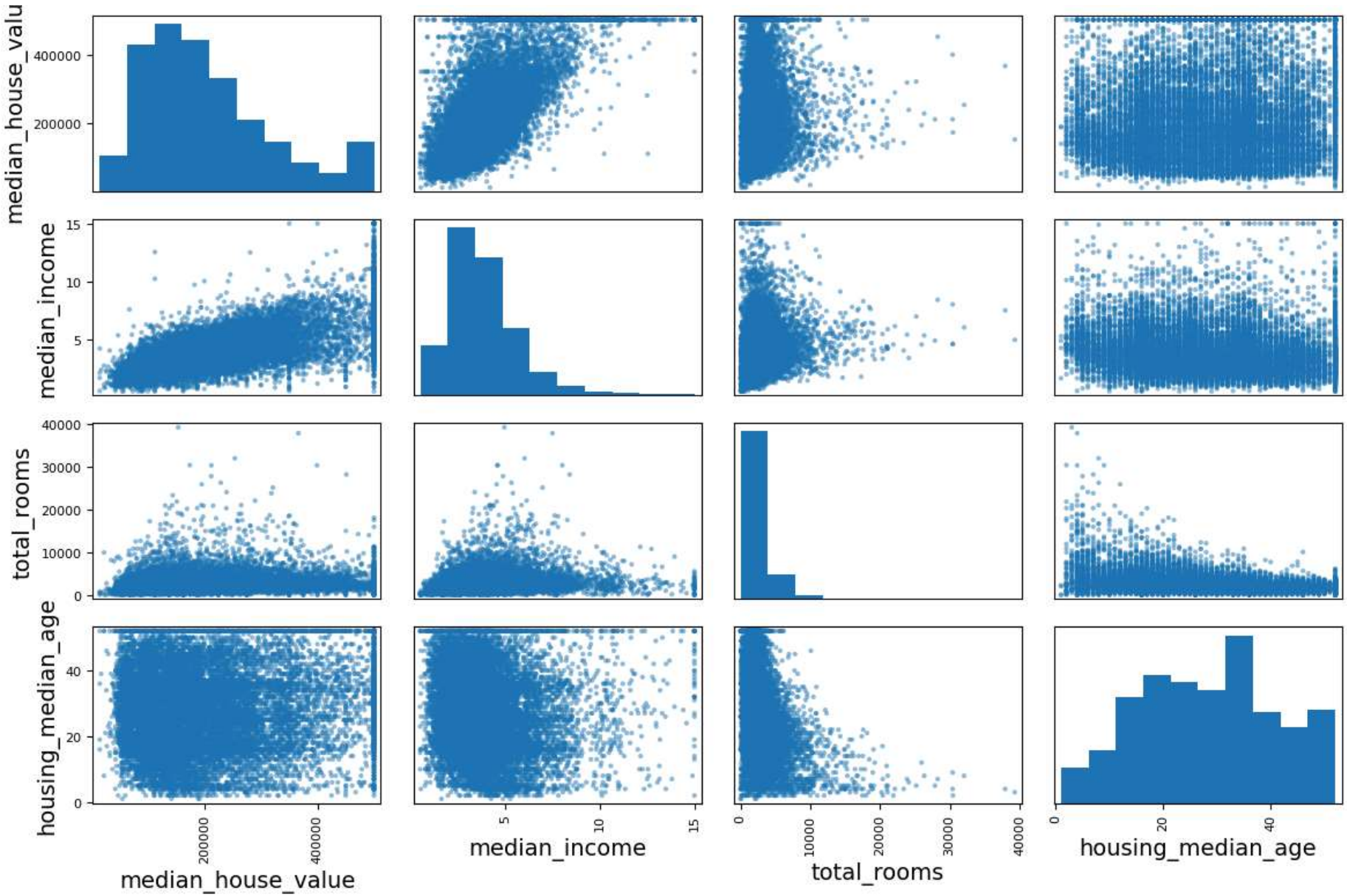
```
1 corr_matrix = housing.corr(numeric_only=True)
```

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

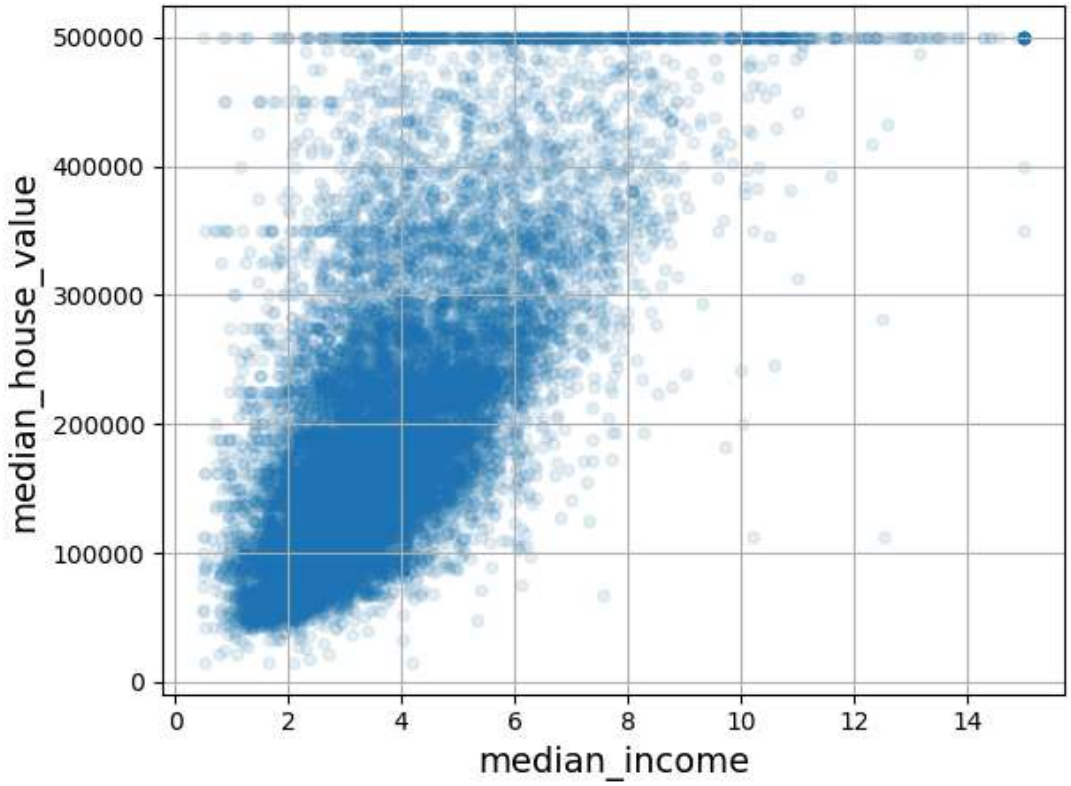
	median_house_value
median_house_value	1.000000
median_income	0.688380
total_rooms	0.137455
housing_median_age	0.102175
households	0.071426
total_bedrooms	0.054635
population	-0.020153
longitude	-0.050859
latitude	-0.139584

dtype: float64

```
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")  # extra code
plt.show()
```


median_house_valu
median_income
total_rooms
housing_median_age
median_house_value
median_income
total_rooms
housing_median_age

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1, grid=True)
save_fig("income_vs_house_value_scatterplot")  # extra code
plt.show()
```



Experimenting with Attribute Combinations

```
housing["rooms_per_house"] = housing["total_rooms"] / housing["households"]
housing["bedrooms_ratio"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["people_per_house"] = housing["population"] / housing["households"]
```

```
corr_matrix = housing.corr(numeric_only=True)
corr_matrix["median_house_value"].sort_values(ascending=False)
```

	median_house_value
median_house_value	1.000000
median_income	0.688380
rooms_per_house	0.143663
total_rooms	0.137455
housing_median_age	0.102175
households	0.071426
total_bedrooms	0.054635
population	-0.020153
people_per_house	-0.038224
longitude	-0.050859
latitude	-0.139584
bedrooms_ratio	-0.256397

dtype: float64

Prepare the Data for Machine Learning Algorithms

Let's revert to the original training set and separate the target (note that `strat_train_set.drop()` creates a copy of `strat_train_set` without the column, it doesn't actually modify `strat_train_set` itself, unless you pass `inplace=True`):

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

Data Cleaning

In the book 3 options are listed to handle the NaN values:

```
housing.dropna(subset=["total_bedrooms"], inplace=True)    # option 1

housing.drop("total_bedrooms", axis=1)       # option 2

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

For each option, we'll create a copy of `housing` and work on that copy to avoid breaking `housing`. We'll also show the output of each option, but filtering on the rows that originally contained a NaN value.

```
null_rows_idx = housing.isnull().any(axis=1)
housing.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	NaN	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	NaN	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	NaN	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	NaN	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	NaN	375.0	183.0	9.8020	<1H OCEAN

```
housing_option1 = housing.copy()

housing_option1.dropna(subset=["total_bedrooms"], inplace=True)  # option 1

housing_option1.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity

```
housing_option2 = housing.copy()

housing_option2.drop("total_bedrooms", axis=1, inplace=True)  # option 2

housing_option2.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	375.0	183.0	9.8020	<1H OCEAN

```
housing_option3 = housing.copy()

median = housing["total_bedrooms"].median()
housing_option3["total_bedrooms"].fillna(median, inplace=True)  # option 3

housing_option3.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	434.0	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	434.0	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	434.0	375.0	183.0	9.8020	<1H OCEAN

```
from sklearn.impute import SimpleImputer

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

Separating out the numerical attributes to use the `"median"` strategy (as it cannot be calculated on text attributes like `ocean_proximity`):

```
housing_num = housing.select_dtypes(include=[np.number])
```

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

```
SimpleImputer
SimpleImputer(strategy='median')
```

```
imputer.statistics_
```

```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

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

```
housing_num.median().values
```

```
array([-118.51  ,   34.26  ,   29.    , 2125.    ,  434.    , 1167.    ,
        408.    ,    3.5385])
```

Transform the training set:

```
X = imputer.transform(housing_num)
```

```
imputer.feature_names_in_
```

```
array(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income'],
      dtype=object)
```

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

```
housing_tr.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
14452	-120.67	40.50	15.0	5343.0	434.0	2503.0	902.0	3.5962
18217	-117.96	34.03	35.0	2093.0	434.0	1755.0	403.0	3.4115
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033
14360	-117.87	33.62	8.0	1266.0	434.0	375.0	183.0	9.8020

```
imputer.strategy
```

```
'median'
```

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

```
housing_tr.loc[null_rows_idx].head()  # not shown in the book
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
14452	-120.67	40.50	15.0	5343.0	434.0	2503.0	902.0	3.5962
18217	-117.96	34.03	35.0	2093.0	434.0	1755.0	403.0	3.4115
11889	-118.05	34.04	33.0	1348.0	434.0	1098.0	257.0	4.2917
20325	-118.88	34.17	15.0	4260.0	434.0	1701.0	669.0	5.1033
14360	-117.87	33.62	8.0	1266.0	434.0	375.0	183.0	9.8020

Now let's drop some outliers:

```
from sklearn.ensemble import IsolationForest

isolation_forest = IsolationForest(random_state=42)
outlier_pred = isolation_forest.fit_predict(X)
```

```
outlier_pred
```

```
array([-1,  1,  1, ...,  1,  1,  1])
```

If you wanted to drop outliers, you would run the following code:

```
#housing = housing.iloc[outlier_pred == 1]
#housing_labels = housing_labels.iloc[outlier_pred == 1]
```

Handling Text and Categorical Attributes

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

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(8)
```

	ocean_proximity
13096	NEAR BAY
14973	<1H OCEAN
3785	INLAND
14689	INLAND
20507	NEAR OCEAN
1286	INLAND
18078	<1H OCEAN
4396	NEAR BAY

Next steps: Generate code with `housing_cat` | View recommended plots | New interactive sheet

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

```
housing_cat_encoded[:8]
```

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

```
ordinal_encoder.categories_
```

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

```
from sklearn.preprocessing import OneHotEncoder

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

```
housing_cat_1hot
```

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

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:

```
housing_cat_1hot.toarray()
```

```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.]])
```

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

```
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       ...,
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.]])
```

```
cat_encoder.categories_
```

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

```
df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]})
pd.get_dummies(df_test)
```

	ocean_proximity_INLAND	ocean_proximity_NEAR BAY
0	True	False
1	False	True

```
cat_encoder.transform(df_test)
```

```
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

```
df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]})
pd.get_dummies(df_test_unknown)
```

	ocean_proximity_<2H OCEAN	ocean_proximity_ISLAND
0	True	False
1	False	True

```
cat_encoder.handle_unknown = "ignore"
cat_encoder.transform(df_test_unknown)
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

```
cat_encoder.feature_names_in_
```

```
array(['ocean_proximity'], dtype=object)
```

```
cat_encoder.get_feature_names_out()
```

```
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',
       'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
       'ocean_proximity_NEAR OCEAN'], dtype=object)
```

```
df_output = pd.DataFrame(cat_encoder.transform(df_test_unknown),
                         columns=cat_encoder.get_feature_names_out(),
                         index=df_test_unknown.index)
```

```
df_output
```

	ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEAR BAY	ocean_proximity_NEAR OCEAN
0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	1.0	0.0	0.0

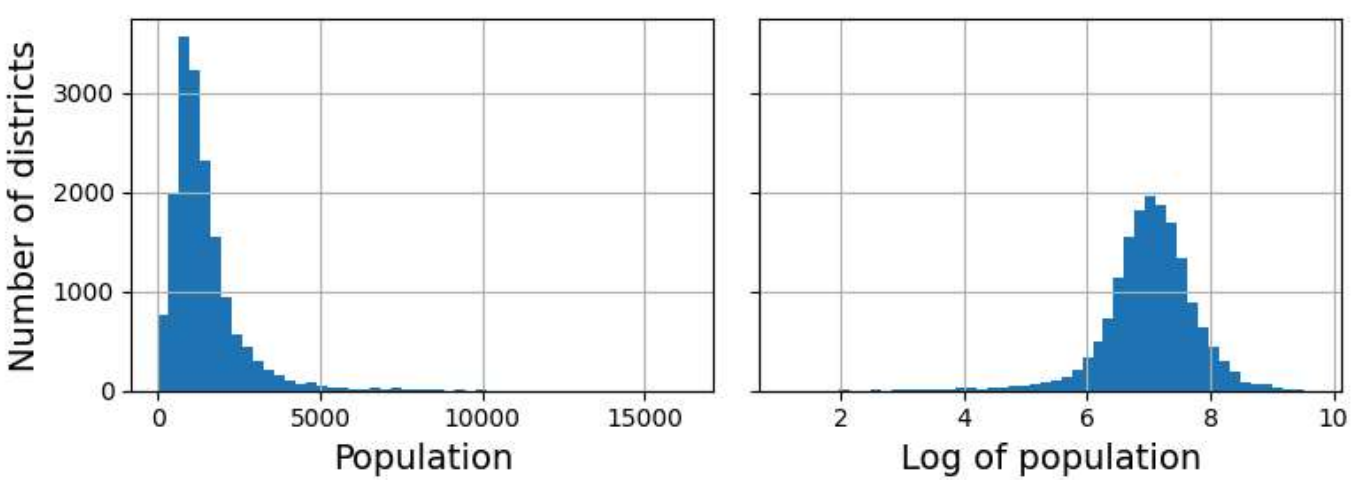
Next steps: Generate code with `df_output` | View recommended plots | New interactive sheet

Feature Scaling

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

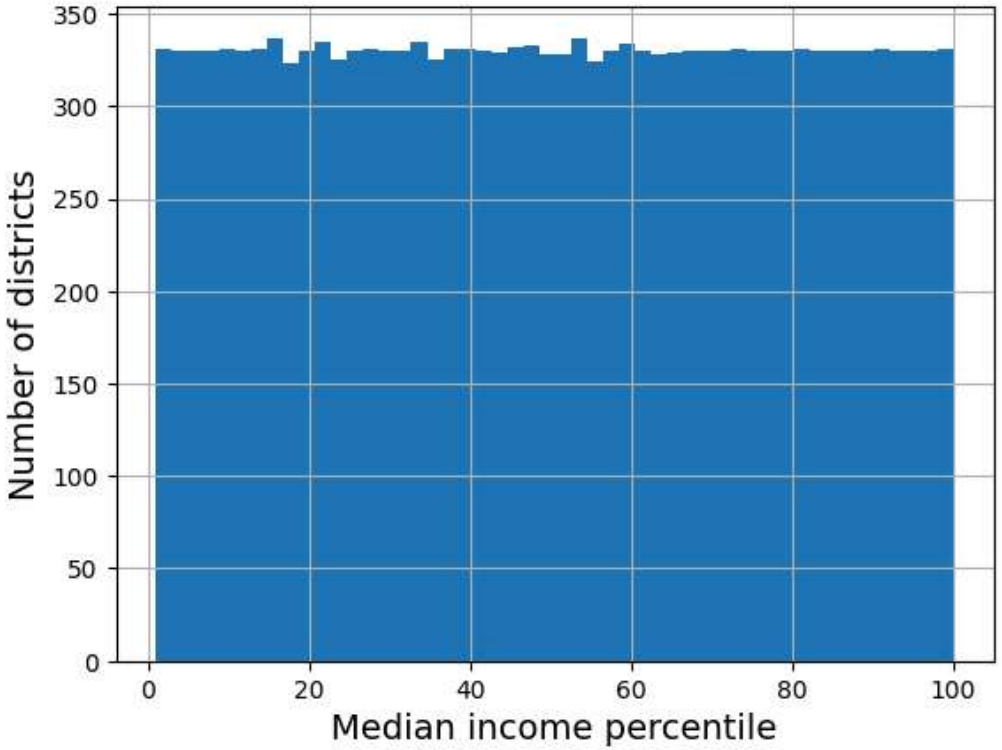
```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

```
# extra code – this cell generates Figure 2–17
fig, axs = plt.subplots(1, 2, figsize=(8, 3), sharey=True)
housing["population"].hist(ax=axs[0], bins=50)
housing["population"].apply(np.log).hist(ax=axs[1], bins=50)
axs[0].set_xlabel("Population")
axs[1].set_xlabel("Log of population")
axs[0].set_ylabel("Number of districts")
save_fig("long_tail_plot")
plt.show()
```



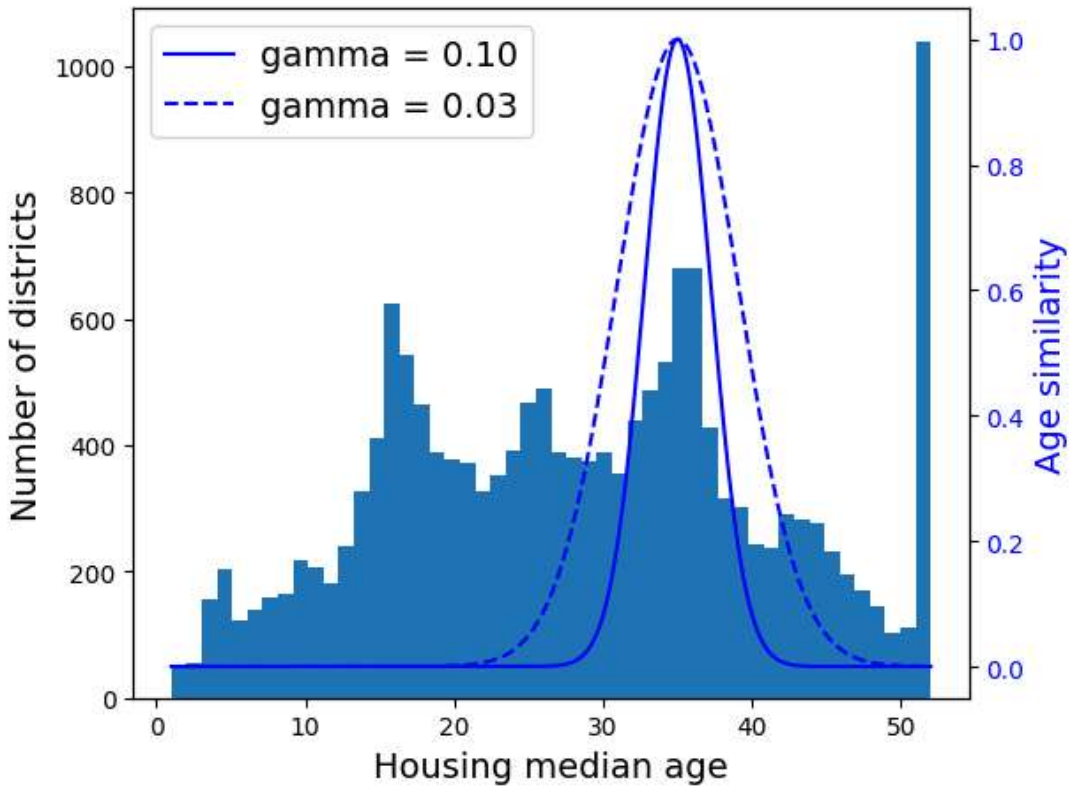
What if we replace each value with its percentile?

```
# extra code – just shows that we get a uniform distribution
percentiles = [np.percentile(housing["median_income"], p)
               for p in range(1, 100)]
flattened_median_income = pd.cut(housing["median_income"],
                                 bins=[-np.inf] + percentiles + [np.inf],
                                 labels=range(1, 100 + 1))
flattened_median_income.hist(bins=50)
plt.xlabel("Median income percentile")
plt.ylabel("Number of districts")
plt.show()
# Note: incomes below the 1st percentile are labeled 1, and incomes above the
# 99th percentile are labeled 100. This is why the distribution below ranges
# from 1 to 100 (not 0 to 100).
```




```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

```
# extra code – this cell generates Figure 2–18

ages = np.linspace(housing["housing_median_age"].min(),
                   housing["housing_median_age"].max(),
                   500).reshape(-1, 1)
gamma1 = 0.1
gamma2 = 0.03
rbf1 = rbf_kernel(ages, [[35]], gamma=gamma1)
rbf2 = rbf_kernel(ages, [[35]], gamma=gamma2)

fig, ax1 = plt.subplots()

ax1.set_xlabel("Housing median age")
ax1.set_ylabel("Number of districts")
ax1.hist(housing["housing_median_age"], bins=50)

ax2 = ax1.twinx()  # create a twin axis that shares the same x-axis
color = "blue"
ax2.plot(ages, rbf1, color=color, label="gamma = 0.10")
ax2.plot(ages, rbf2, color=color, label="gamma = 0.03", linestyle="--")
ax2.tick_params(axis='y', labelcolor=color)
ax2.set_ylabel("Age similarity", color=color)

plt.legend(loc="upper left")
save_fig("age_similarity_plot")
plt.show()
```




```
from sklearn.linear_model import LinearRegression

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

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

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

```
predictions
```

```
array([[131997.15275877],
       [299359.35844434],
       [146023.37185694],
```

```
           [138840.33653057],
           [192016.61557639]])
```

```
from sklearn.compose import TransformedTargetRegressor

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

```
predictions
```

```
array([131997.15275877, 299359.35844434, 146023.37185694, 138840.33653057,
       192016.61557639])
```

Custom Transformers

To create simple transformers:

```
from sklearn.preprocessing import FunctionTransformer

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

```
rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

```
age_simil_35
```

```
array([[2.81118530e-13],
       [8.20849986e-02],
       [6.70320046e-01],
       ...,
       [9.55316054e-22],
       [6.70320046e-01],
       [3.03539138e-04]])
```

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

```
sf_simil
```

```
array([[0.999927  ],
       [0.05258419],
       [0.94864161],
       ...,
       [0.00388525],
       [0.05038518],
       [0.99868067]])
```

```
ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
```

```
array([[0.5 ],
       [0.75]])
```

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

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True):  # no *args or **kwargs!
        self.with_mean = with_mean

    def fit(self, X, y=None):  # y is required even though we don't use it
        X = check_array(X)  # checks that X is an array with finite float values
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1]  # every estimator stores this in fit()
        return self  # always return self!

    def transform(self, X):
        check_is_fitted(self)  # looks for learned attributes (with trailing _)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

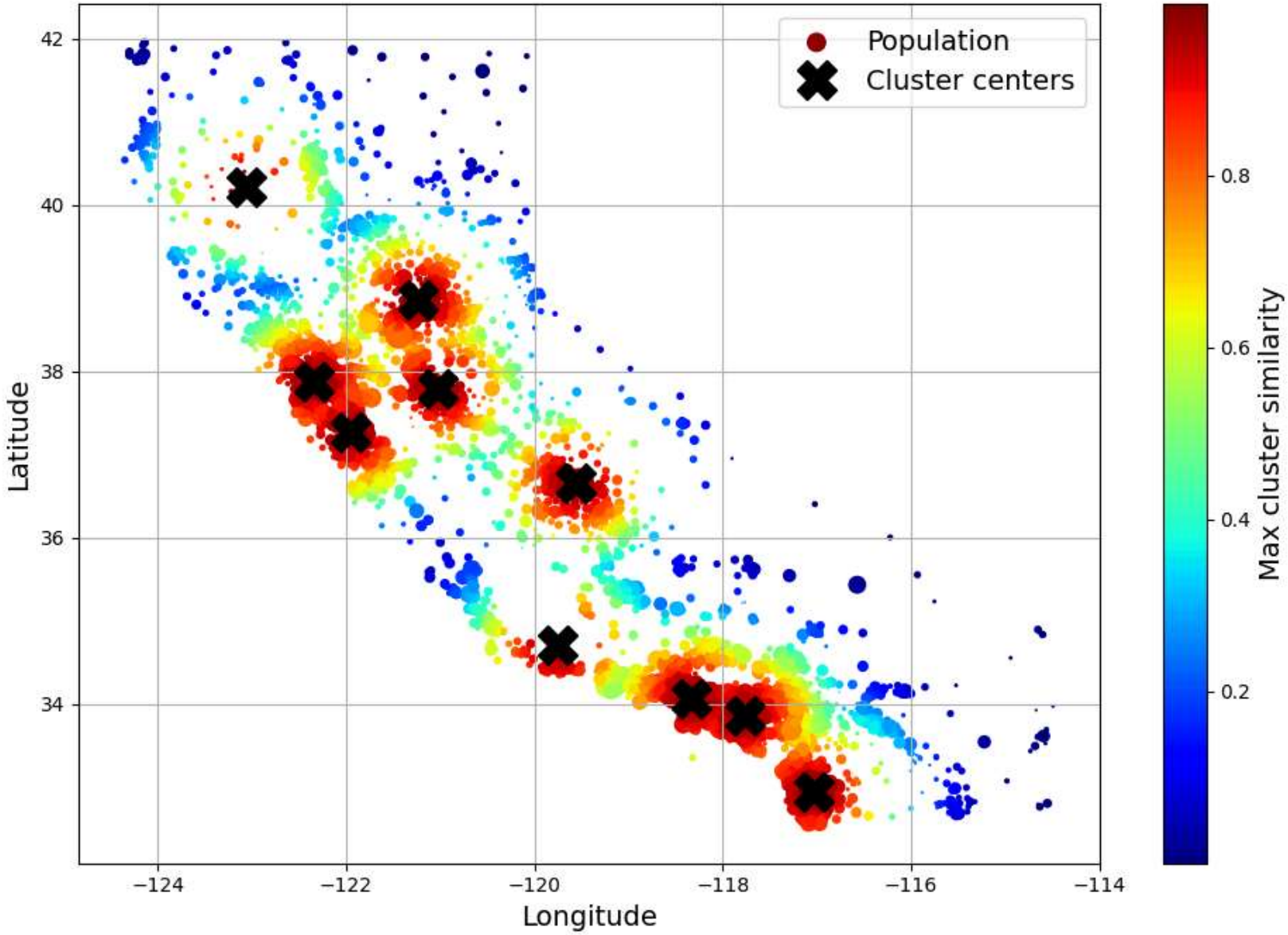
```
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, n_init=10,
                              random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]],
                                           sample_weight=housing_labels)
```

```
similarities[:3].round(2)
```

```
array([[0.08, 0.  , 0.6 , 0.  , 0.  , 0.99, 0.  , 0.  , 0.  , 0.14],
       [0.  , 0.99, 0.  , 0.04, 0.  , 0.  , 0.11, 0.  , 0.63, 0.  ],
       [0.44, 0.  , 0.3 , 0.  , 0.  , 0.7 , 0.  , 0.01, 0.  , 0.29]])
```

```
# extra code – this cell generates Figure 2–19

housing_renamed = housing.rename(columns={
    "latitude": "Latitude", "longitude": "Longitude",
    "population": "Population",
    "median_house_value": "Median house value (usd)"})
housing_renamed["Max cluster similarity"] = similarities.max(axis=1)

housing_renamed.plot(kind="scatter", x="Longitude", y="Latitude", grid=True,
                     s=housing_renamed["Population"] / 100, label="Population",
                     c="Max cluster similarity",
                     cmap="jet", colorbar=True,
                     legend=True, sharex=False, figsize=(10, 7))
plt.plot(cluster_simil.kmeans_.cluster_centers_[:, 1],
         cluster_simil.kmeans_.cluster_centers_[:, 0],
         linestyle="", color="black", marker="X", markersize=20,
         label="Cluster centers")
plt.legend(loc="upper right")
save_fig("district_cluster_plot")
plt.show()
```



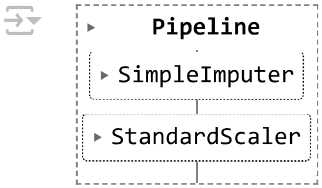
Transformation Pipelines

Now let's build a pipeline to preprocess the numerical attributes:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

```
from sklearn import set_config

set_config(display='diagram')

num_pipeline
```




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

```
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

```
def monkey_patch_get_signature_names_out():
    """Monkey patch some classes which did not handle get_feature_names_out()
       correctly in Scikit-Learn 1.0.*."""
    from inspect import Signature, signature, Parameter
    import pandas as pd
    from sklearn.impute import SimpleImputer
    from sklearn.pipeline import make_pipeline, Pipeline
    from sklearn.preprocessing import FunctionTransformer, StandardScaler

    default_get_feature_names_out = StandardScaler.get_feature_names_out

    if not hasattr(SimpleImputer, "get_feature_names_out"):
      print("Monkey-patching SimpleImputer.get_feature_names_out()")
      SimpleImputer.get_feature_names_out = default_get_feature_names_out

    if not hasattr(FunctionTransformer, "get_feature_names_out"):
        print("Monkey-patching FunctionTransformer.get_feature_names_out()")
        orig_init = FunctionTransformer.__init__
        orig_sig = signature(orig_init)

        def __init__(*args, feature_names_out=None, **kwargs):
            orig_sig.bind(*args, **kwargs)
            orig_init(*args, **kwargs)
            args[0].feature_names_out = feature_names_out

        __init__.__signature__ = Signature(
            list(signature(orig_init).parameters.values()) + [
                Parameter("feature_names_out", Parameter.KEYWORD_ONLY)])

        def get_feature_names_out(self, names=None):
            if callable(self.feature_names_out):
                return self.feature_names_out(self, names)
            assert self.feature_names_out == "one-to-one"
            return default_get_feature_names_out(self, names)

        FunctionTransformer.__init__ = __init__
        FunctionTransformer.get_feature_names_out = get_feature_names_out

monkey_patch_get_signature_names_out()
```

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

```
df_housing_num_prepared.head(2)  # extra code
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income
13096	-1.423037	1.013606	1.861119	0.311912	1.368167	0.137460	1.394812	-0.936491
14973	0.596394	-0.702103	0.907630	-0.308620	-0.435925	-0.693771	-0.373485	1.171942

Next steps: Generate code with df_housing_num_prepared | View recommended plots | New interactive sheet

```
num_pipeline.steps
```

```
[('simpleimputer', SimpleImputer(strategy='median')),
 ('standardscaler', StandardScaler())]
```

```
num_pipeline[1]
```

```
StandardScaler
StandardScaler()
```

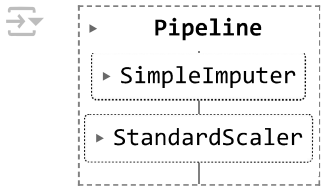
```
num_pipeline[:-1]
```

```
Pipeline
SimpleImputer
```

```
num_pipeline.named_steps["simpleimputer"]
```

```
SimpleImputer
SimpleImputer(strategy='median')
```

```
num_pipeline.set_params(simpleimputer__strategy="median")
```




```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

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

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

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

```
housing_prepared = preprocessing.fit_transform(housing)
```

```
# extra code – shows that we can get a DataFrame out if we want
housing_prepared_fr = pd.DataFrame(
    housing_prepared,
    columns=preprocessing.get_feature_names_out(),
    index=housing.index)
housing_prepared_fr.head(2)
```

	pipeline-1__longitude	pipeline-1__latitude	pipeline-1__housing_median_age	pipeline-1__total_rooms	pipeline-1__total_bedrooms	pipeline-1__population	pipeline-1__households	pipelin 1__median_inco
13096	-1.423037	1.013606	1.861119	0.311912	1.368167	0.137460	1.394812	-0.9364
14973	0.596394	-0.702103	0.907630	-0.308620	-0.435925	-0.693771	-0.373485	1.1719

Next steps: Generate code with housing_prepared_fr | View recommended plots | New interactive sheet

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

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"]  # feature names out

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

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())
preprocessing = ColumnTransformer([
        ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
        ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
        ("people_per_house", ratio_pipeline(), ["population", "households"]),
        ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                               "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
    ],
    remainder=default_num_pipeline)  # one column remaining: housing_median_age
```

```
housing_prepared = preprocessing.fit_transform(housing)
housing_prepared.shape
```

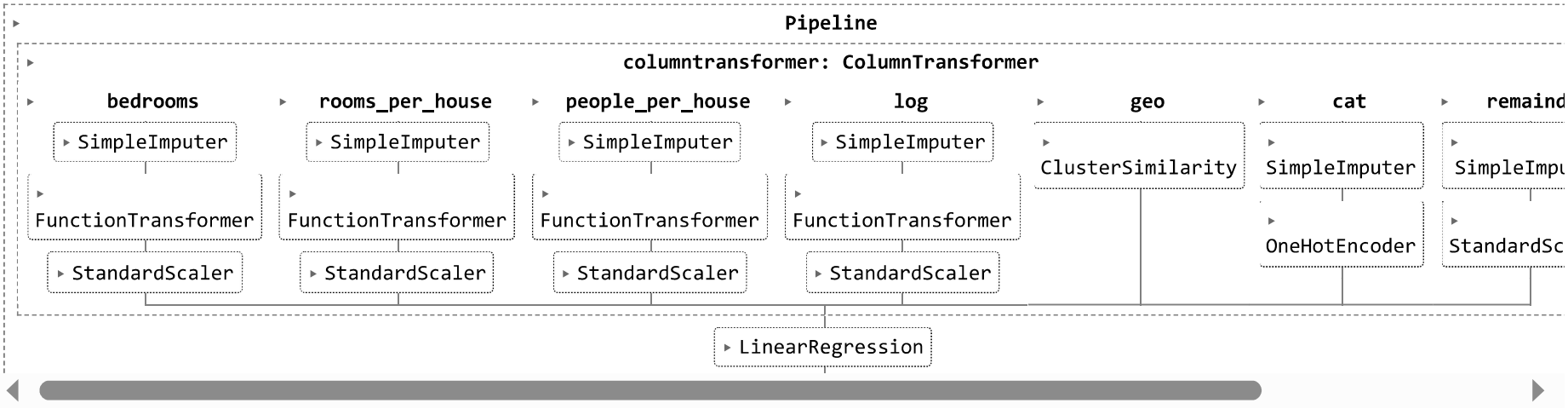
```
(16512, 24)
```

```
preprocessing.get_feature_names_out()
```

```
array(['bedrooms__ratio', 'rooms_per_house__ratio',
       'people_per_house__ratio', 'log__total_bedrooms',
       'log__total_rooms', 'log__population', 'log__households',
       'log__median_income', 'geo__Cluster 0 similarity',
       'geo__Cluster 1 similarity', 'geo__Cluster 2 similarity',
       'geo__Cluster 3 similarity', 'geo__Cluster 4 similarity',
       'geo__Cluster 5 similarity', 'geo__Cluster 6 similarity',
       'geo__Cluster 7 similarity', 'geo__Cluster 8 similarity',
       'geo__Cluster 9 similarity', 'cat__ocean_proximity_<1H OCEAN',
       'cat__ocean_proximity_INLAND', 'cat__ocean_proximity_ISLAND',
       'cat__ocean_proximity_NEAR BAY', 'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```

Select and Train a Model

```
from sklearn.linear_model import LinearRegression

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



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

```
housing_predictions = lin_reg.predict(housing)
housing_predictions[:5].round(-2)  # -2 = rounded to the nearest hundred
```

```
array([242800., 375900., 127500.,  99400., 324600.])
```

Compare against the actual values:

```
housing_labels.iloc[:5].values
```

```
array([458300., 483800., 101700.,  96100., 361800.])
```

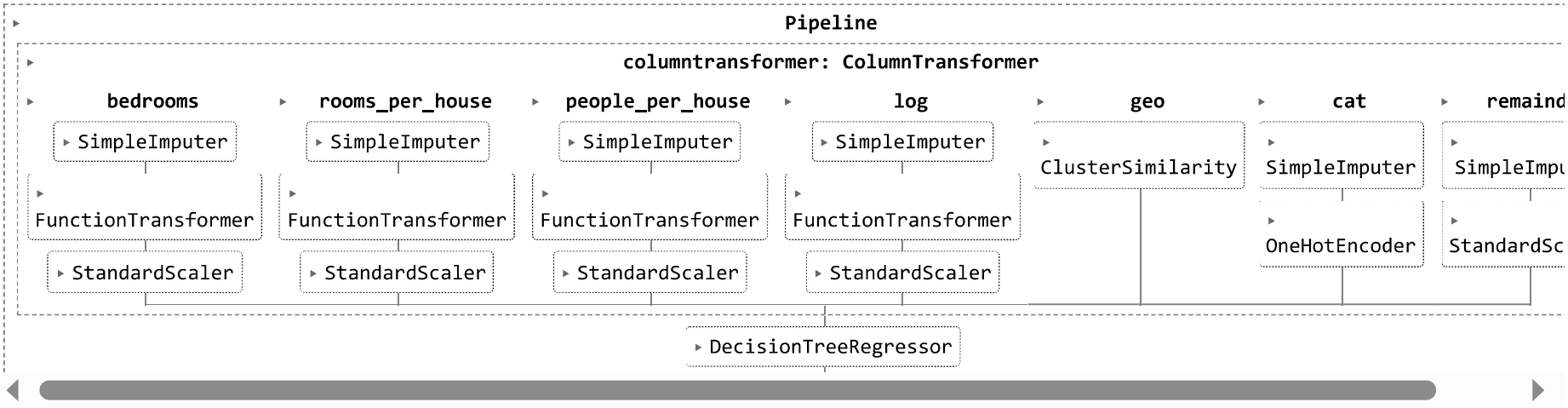
```
# extra code – computes the error ratios discussed in the book
error_ratios = housing_predictions[:5].round(-2) / housing_labels.iloc[:5].values - 1
print(", ".join([f"{100 * ratio:.1f}%" for ratio in error_ratios]))
```

```
-47.0%, -22.3%, 25.4%, 3.4%, -10.3%
```

```
from sklearn.metrics import mean_squared_error

lin_rmse = mean_squared_error(housing_labels, housing_predictions,
                              squared=False)
lin_rmse
```

```
68647.95686706658
```

```
from sklearn.tree import DecisionTreeRegressor

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




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

```
0.0
```

Better Evaluation Using Cross-Validation

```
from sklearn.model_selection import cross_val_score

tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring="neg_root_mean_squared_error", cv=10)
```

```
pd.Series(tree_rmses).describe()
```

	0
count	10.000000
mean	67153.318273
std	1963.580924
min	63925.253106
25%	66083.277180
50%	66795.829871
75%	68074.018403
max	70664.635833

dtype: float64

```
# extra code – computes the error stats for the linear model
lin_rmses = -cross_val_score(lin_reg, housing, housing_labels,
                             scoring="neg_root_mean_squared_error", cv=10)
pd.Series(lin_rmses).describe()
```

	0
count	10.000000
mean	69847.923224
std	4078.407329
min	65659.761079
25%	68088.799156
50%	68697.591463
75%	69800.966364
max	80685.254832

dtype: float64

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                scoring="neg_root_mean_squared_error", cv=10)
```

```
pd.Series(forest_rmses).describe()
```

	0
count	10.000000
mean	47002.931706
std	1048.451340
min	45667.064036
25%	46494.358345
50%	47093.173938
75%	47274.873814
max	49354.705514

dtype: float64

Let's compare this RMSE measured using cross-validation (the "validation error") with the RMSE measured on the training set (the "training error"):

```
forest_reg.fit(housing, housing_labels)
housing_predictions = forest_reg.predict(housing)
forest_rmse = mean_squared_error(housing_labels, housing_predictions,
                                 squared=False)
forest_rmse
```

```
17547.52124624957
```

The training error is much lower than the validation error, which usually means that the model has overfit the training set. Another possible explanation may be that there's a mismatch between the training data and the validation data, but it's not the case here, since both came from

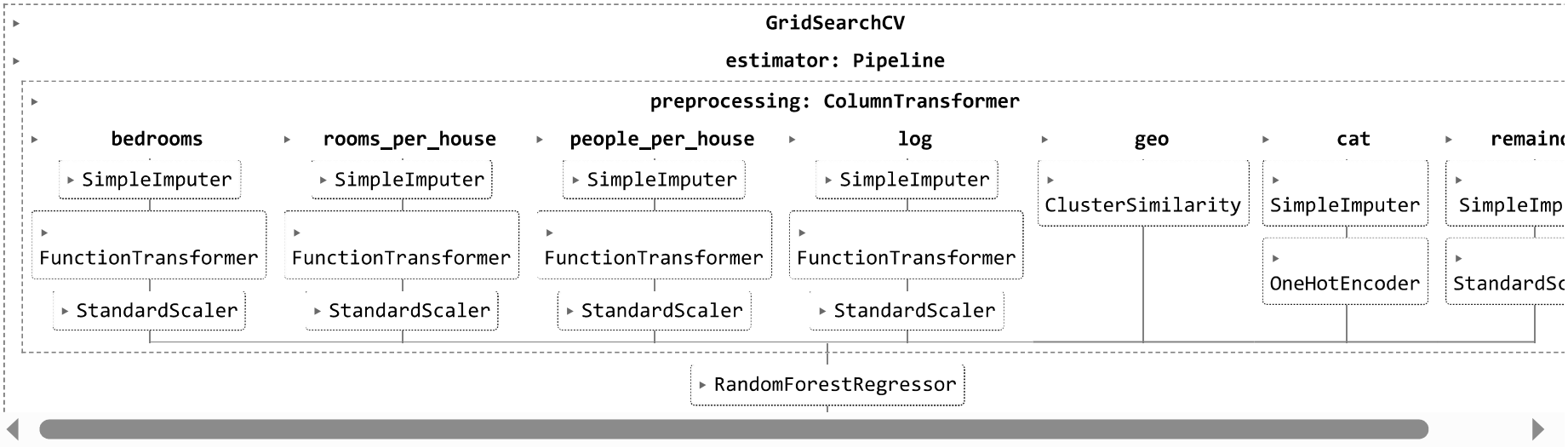
the same dataset that we shuffled and split in two parts.

Fine-Tune Your Model

Grid Search

Warning: the following cell may take a few minutes to run:

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [
    {'preprocessing__geo__n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo__n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```



You can get the full list of hyperparameters available for tuning by looking at `full_pipeline.get_params().keys()`:

```
# extra code – shows part of the output of get_params().keys()
print(str(full_pipeline.get_params().keys())[:1000] + "...")
```

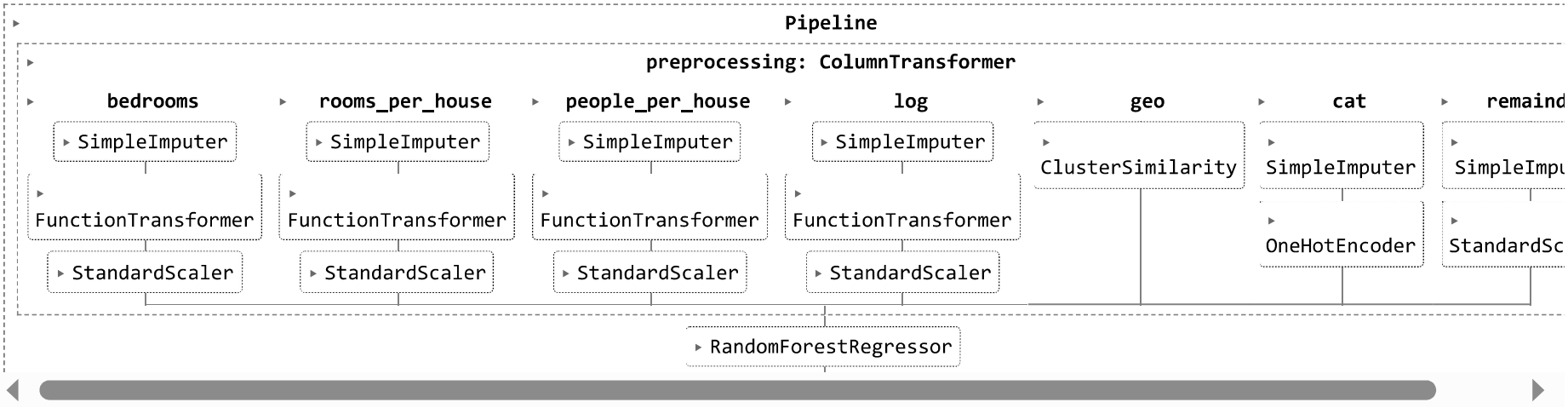
```
dict_keys(['memory', 'steps', 'verbose', 'preprocessing', 'random_forest', 'preprocessing__n_jobs', 'preprocessing__remainder__memory',
```

The best hyperparameter combination found:

```
grid_search.best_params_
```

```
{'preprocessing__geo__n_clusters': 15, 'random_forest__max_features': 6}
```

```
grid_search.best_estimator_
```



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

```
cv_res = pd.DataFrame(grid_search.cv_results_)
cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)

# extra code – these few lines of code just make the DataFrame look nicer
cv_res = cv_res[["param_preprocessing__geo__n_clusters",
                 "param_random_forest__max_features", "split0_test_score",
                 "split1_test_score", "split2_test_score", "mean_test_score"]]
score_cols = ["split0", "split1", "split2", "mean_test_rmse"]
cv_res.columns = ["n_clusters", "max_features"] + score_cols
cv_res[score_cols] = -cv_res[score_cols].round().astype(np.int64)

cv_res.head()
```

	n_clusters	max_features	split0	split1	split2	mean_test_rmse
12	15	6	43536	43753	44569	43953
13	15	8	44084	44205	44863	44384
14	15	10	44368	44496	45200	44688
7	10	6	44251	44628	45857	44912
9	10	6	44251	44628	45857	44912

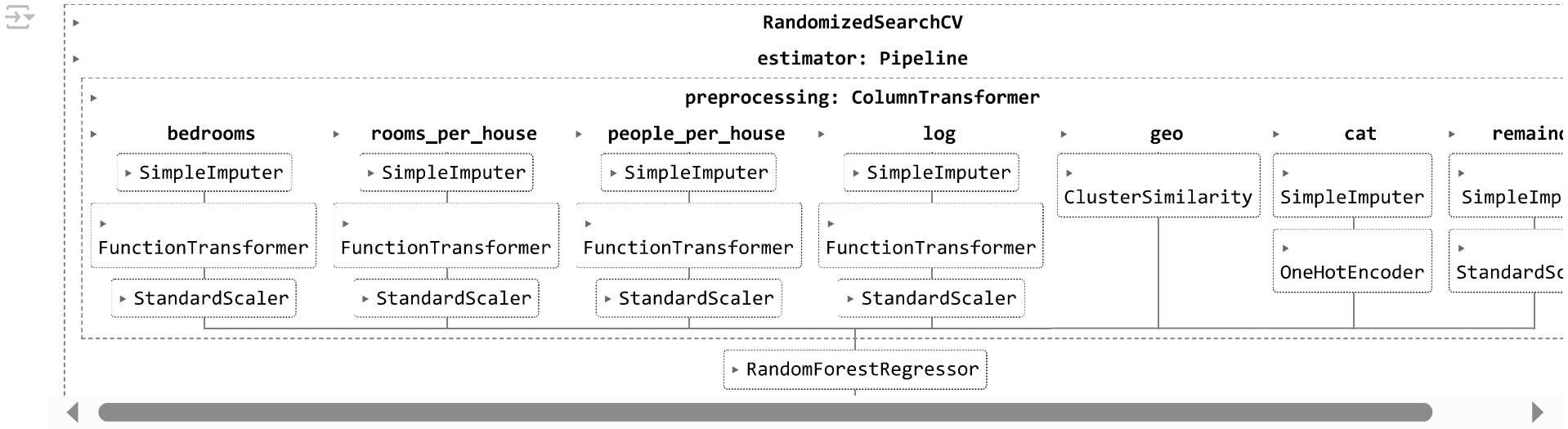
Randomized Search

```
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingRandomSearchCV
```

Try 30 (`n_iter × cv`) random combinations of hyperparameters:

Warning: the following cell may take a few minutes to run:

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

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

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

rnd_search.fit(housing, housing_labels)
```



```
# extra code – displays the random search results
cv_res = pd.DataFrame(rnd_search.cv_results_)
cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)
cv_res = cv_res[["param_preprocessing__geo__n_clusters",
                 "param_random_forest__max_features", "split0_test_score",
                 "split1_test_score", "split2_test_score", "mean_test_score"]]
cv_res.columns = ["n_clusters", "max_features"] + score_cols
cv_res[score_cols] = -cv_res[score_cols].round().astype(np.int64)
cv_res.head()
```

	n_clusters	max_features	split0	split1	split2	mean_test_rmse
1	45	9	41115	42151	42695	41987
8	32	7	41604	42200	43219	42341
0	41	16	42106	42743	43443	42764
5	42	4	41812	42925	43557	42765
2	23	8	42421	43094	43856	43124

Next steps: Generate code with `cv_res` | View recommended plots | New interactive sheet

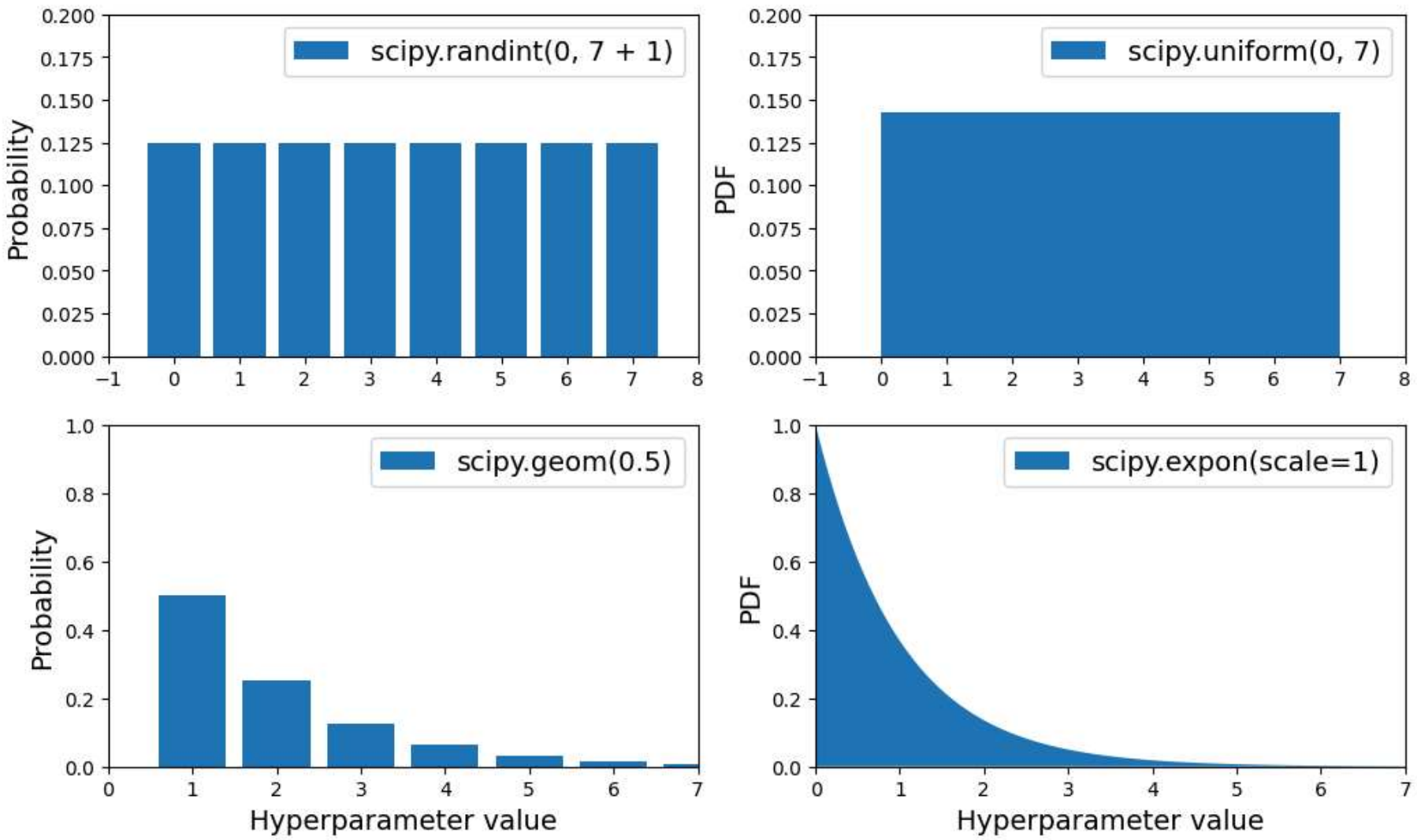
Bonus section: how to choose the sampling distribution for a hyperparameter

- `scipy.stats.randint(a, b+1)`: for hyperparameters with *discrete* values that range from a to b, and all values in that range seem equally likely.
- `scipy.stats.uniform(a, b)`: this is very similar, but for *continuous* hyperparameters.
- `scipy.stats.geom(1 / scale)`: for discrete values, when you want to sample roughly in a given scale. E.g., with scale=1000 most samples will be in this ballpark, but ~10% of all samples will be <100 and ~10% will be >2300.
- `scipy.stats.expon(scale)`: this is the continuous equivalent of `geom`. Just set `scale` to the most likely value.
- `scipy.stats.loguniform(a, b)`: when you have almost no idea what the optimal hyperparameter value's scale is. If you set a=0.01 and b=100, then you're just as likely to sample a value between 0.01 and 0.1 as a value between 10 and 100.

Here are plots of the probability mass functions (for discrete variables), and probability density functions (for continuous variables) for `randint()`, `uniform()`, `geom()` and `expon()`:

```
# extra code – plots a few distributions you can use in randomized search

from scipy.stats import randint, uniform, geom, expon

xs1 = np.arange(0, 7 + 1)
randint_distrib = randint(0, 7 + 1).pmf(xs1)

xs2 = np.linspace(0, 7, 500)
uniform_distrib = uniform(0, 7).pdf(xs2)

xs3 = np.arange(0, 7 + 1)
geom_distrib = geom(0.5).pmf(xs3)

xs4 = np.linspace(0, 7, 500)
expon_distrib = expon(scale=1).pdf(xs4)

plt.figure(figsize=(12, 7))

plt.subplot(2, 2, 1)
plt.bar(xs1, randint_distrib, label="scipy.randint(0, 7 + 1)")
plt.ylabel("Probability")
plt.legend()
plt.axis([-1, 8, 0, 0.2])

plt.subplot(2, 2, 2)
plt.fill_between(xs2, uniform_distrib, label="scipy.uniform(0, 7)")
plt.ylabel("PDF")
plt.legend()
plt.axis([-1, 8, 0, 0.2])

plt.subplot(2, 2, 3)
plt.bar(xs3, geom_distrib, label="scipy.geom(0.5)")
plt.xlabel("Hyperparameter value")
plt.ylabel("Probability")
plt.legend()
plt.axis([0, 7, 0, 1])

plt.subplot(2, 2, 4)
plt.fill_between(xs4, expon_distrib, label="scipy.expon(scale=1)")
plt.xlabel("Hyperparameter value")
plt.ylabel("PDF")
plt.legend()
plt.axis([0, 7, 0, 1])
```

```
44
45 plt.show()
```

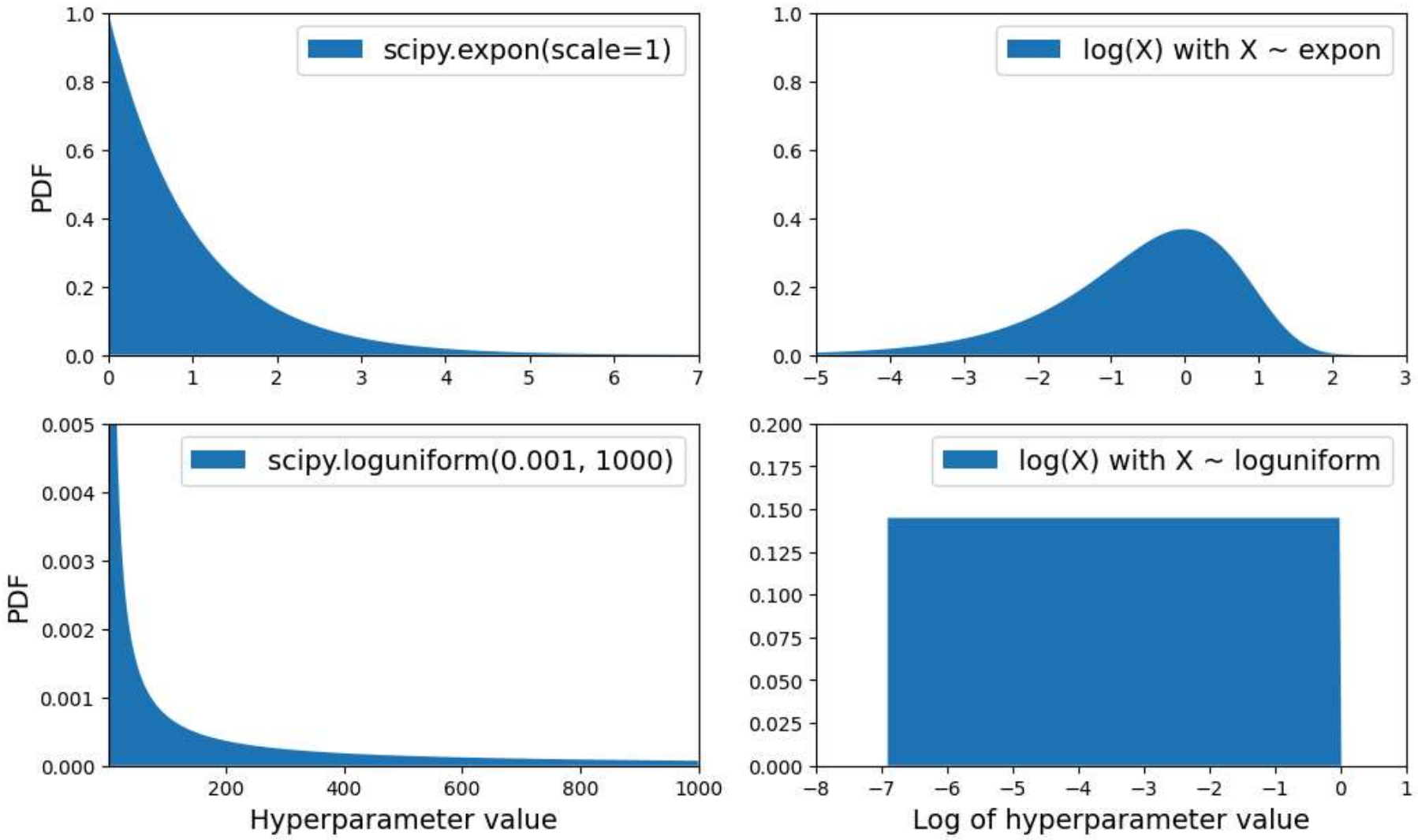


Here are the PDF for `expon()` and `loguniform()` (left column), as well as the PDF of log(X) (right column). The right column shows the distribution of hyperparameter *scales*. You can see that `expon()` favors hyperparameters with roughly the desired scale, with a longer tail towards the smaller scales. But `loguniform()` does not favor any scale, they are all equally likely:

```
1 # extra code – shows the difference between expon and loguniform
```

```
 2
 3 from scipy.stats import loguniform
 4
 5 xs1 = np.linspace(0, 7, 500)
 6 expon_distrib = expon(scale=1).pdf(xs1)
 7
 8 log_xs2 = np.linspace(-5, 3, 500)
 9 log_expon_distrib = np.exp(log_xs2 - np.exp(log_xs2))
10
11 xs3 = np.linspace(0.001, 1000, 500)
12 loguniform_distrib = loguniform(0.001, 1000).pdf(xs3)
13
14 log_xs4 = np.linspace(np.log(0.001), np.log(1000), 500)
15 log_loguniform_distrib = uniform(np.log(0.001), np.log(1000)).pdf(log_xs4)
16
17 plt.figure(figsize=(12, 7))
18
19 plt.subplot(2, 2, 1)
20 plt.fill_between(xs1, expon_distrib,
21                  label="scipy.expon(scale=1)")
22 plt.ylabel("PDF")
23 plt.legend()
24 plt.axis([0, 7, 0, 1])
25
26 plt.subplot(2, 2, 2)
27 plt.fill_between(log_xs2, log_expon_distrib,
28                  label="log(X) with X ~ expon")
29 plt.legend()
30 plt.axis([-5, 3, 0, 1])
31
32 plt.subplot(2, 2, 3)
33 plt.fill_between(xs3, loguniform_distrib,
34                  label="scipy.loguniform(0.001, 1000)")
35 plt.xlabel("Hyperparameter value")
36 plt.ylabel("PDF")
37 plt.legend()
38 plt.axis([0.001, 1000, 0, 0.005])
39
40 plt.subplot(2, 2, 4)
41 plt.fill_between(log_xs4, log_loguniform_distrib,
42                  label="log(X) with X ~ loguniform")
43 plt.xlabel("Log of hyperparameter value")
44 plt.legend()
45 plt.axis([-8, 1, 0, 0.2])
```

```
46
47 plt.show()
```



Analyze the Best Models and Their Errors

```
1 final_model = rnd_search.best_estimator_  # includes preprocessing
2 feature_importances = final_model["random_forest"].feature_importances_
3 feature_importances.round(2)
```

```
array([0.06, 0.06, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, 0.01, 0.01, 0.02,
       0.04, 0.01, 0.  , 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.  , 0.  ,
       0.01, 0.  , 0.01, 0.02, 0.02, 0.01, 0.01, 0.01, 0.03, 0.01, 0.01,
       0.01, 0.01, 0.01, 0.01, 0.  , 0.01, 0.01, 0.02, 0.01, 0.01, 0.01,
       0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.  , 0.08,
       0.  , 0.  , 0.  , 0.01])
```

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

```
(0.07709175866873944, 'cat__ocean_proximity_INLAND'),
(0.06455488601956336, 'bedrooms__ratio'),
(0.056936146643377976, 'rooms_per_house__ratio'),
(0.0490294770805355, 'people_per_house__ratio'),
(0.03807069074492323, 'geo__Cluster 3 similarity'),
(0.025643913400094476, 'geo__Cluster 22 similarity'),
(0.02179127543243723, 'geo__Cluster 17 similarity'),
(0.021575251507503695, 'geo__Cluster 6 similarity'),
(0.017868654556924362, 'geo__Cluster 2 similarity'),
(0.017431400050755975, 'geo__Cluster 32 similarity'),
(0.015981159400591683, 'geo__Cluster 18 similarity'),
(0.01488846425739688, 'geo__Cluster 40 similarity'),
(0.014488389218107143, 'geo__Cluster 43 similarity'),
(0.014252940099964142, 'geo__Cluster 7 similarity'),
(0.014038173319370725, 'geo__Cluster 21 similarity'),
(0.013846025114732157, 'geo__Cluster 38 similarity'),
(0.01362570996472274, 'geo__Cluster 34 similarity'),
(0.013547297167034428, 'geo__Cluster 41 similarity'),
(0.012900089026066918, 'geo__Cluster 24 similarity'),
(0.012620908145579916, 'geo__Cluster 10 similarity'),
(0.011621275372313349, 'remainder__housing_median_age'),
(0.01159076532124518, 'geo__Cluster 42 similarity'),
(0.011475471902949968, 'geo__Cluster 30 similarity'),
(0.011145862167873194, 'geo__Cluster 31 similarity'),
(0.011019562821385253, 'geo__Cluster 16 similarity'),
(0.010872453237288457, 'geo__Cluster 1 similarity'),
(0.010683824851366689, 'geo__Cluster 25 similarity'),
(0.01060046411739516, 'geo__Cluster 26 similarity'),
(0.010233275929521545, 'geo__Cluster 20 similarity'),
(0.01000904987677275, 'geo__Cluster 35 similarity'),
(0.009948648710662759, 'geo__Cluster 14 similarity'),
(0.009237024861905358, 'geo__Cluster 39 similarity'),
(0.009095509036677479, 'geo__Cluster 37 similarity'),
(0.009039692793107918, 'geo__Cluster 0 similarity'),
```

```
(0.008802851981011S,  geo__Cluster 9 similarity'),
(0.00865735444375151, 'geo__Cluster 36 similarity'),
(0.008468774649648216, 'geo__Cluster 28 similarity'),
(0.008132158692323862, 'geo__Cluster 44 similarity'),
(0.00806872019824202, 'geo__Cluster 4 similarity'),
(0.007900076467114424, 'geo__Cluster 11 similarity'),
(0.007654000251882609, 'log__total_rooms'),
(0.007117524274343567, 'log__population'),
(0.006828512079872657, 'log__total_bedrooms'),
(0.006436845508698041, 'log__households'),
(0.00606171524521791, 'geo__Cluster 23 similarity'),
(0.005534308195396936, 'geo__Cluster 19 similarity'),
(0.00549408919549185, 'geo__Cluster 27 similarity'),
(0.0052089915629445595, 'geo__Cluster 33 similarity'),
(0.004814425966149223, 'geo__Cluster 15 similarity'),
(0.004210239717026306, 'geo__Cluster 12 similarity'),
(0.004038656871498407, 'geo__Cluster 13 similarity'),
(0.00373031587904682, 'geo__Cluster 29 similarity'),
(0.003249851526492918, 'cat__ocean_proximity_<1H OCEAN'),
(0.001973755321832911, 'geo__Cluster 5 similarity'),
(0.0018976773551793616, 'cat__ocean_proximity_NEAR OCEAN'),
(0.0002358514810131377, 'cat__ocean_proximity_NEAR BAY'),
(6.124671466559031e-05, 'cat__ocean_proximity_ISLAND')]
```

Evaluate Your System on the Test Set

```
1 X_test = strat_test_set.drop("median_house_value", axis=1)
2 y_test = strat_test_set["median_house_value"].copy()
3
4 final_predictions = final_model.predict(X_test)
5
6 final_rmse = mean_squared_error(y_test, final_predictions, squared=False)
7 print(final_rmse)
```

```
41549.20158097943
```

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

```
from scipy import stats

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

```
array([39395.35475927, 43596.76969025])
```

We could compute the interval manually like this:

```
# extra code – shows how to compute a confidence interval for the RMSE
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)
```

```
(39395.35475926931, 43596.76969025394)
```

Alternatively, we could use a z-score rather than a t-score. Since the test set is not too small, it won't make a big difference:

```
# extra code – computes a confidence interval again using a z-score
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)
```

```
(39396.00369767951, 43596.18328117898)
```

Model persistence using joblib

Save the final model:

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")


```

Double-click (or enter) to edit

Now you can deploy this model to production. For example, the following code could be a script that would run in production:

```
import joblib

# extra code – excluded for conciseness
from sklearn.cluster import KMeans
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.metrics.pairwise import rbf_kernel

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

#class ClusterSimilarity(BaseEstimator, TransformerMixin):
#    [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = housing.iloc[:5]  # pretend these are new districts
predictions = final_model_reloaded.predict(new_data)
```

```
predictions
```

```
array([439808.14, 455211.06, 109492.  ,  98208.  , 340021.04])
```

You could use pickle instead, but joblib is more efficient.

Exercise solutions

1.

Exercise: *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). Note that SVMs don't scale well to large datasets, so you should probably train your model on just the first 5,000 instances of the training set and use only 3-fold cross-validation, or else it will take hours. Don't worry about what the hyperparameters mean for now (see the SVM notebook if you're interested). How does the best `SVR` predictor perform?*

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR

param_grid = [
        {'svr__kernel': ['linear'], 'svr__C': [10., 30., 100., 300., 1000.,
                                               3000., 10000., 30000.0]},
        {'svr__kernel': ['rbf'], 'svr__C': [1.0, 3.0, 10., 30., 100., 300.,
                                            1000.0],
         'svr__gamma': [0.01, 0.03, 0.1, 0.3, 1.0, 3.0]},
    ]

svr_pipeline = Pipeline([("preprocessing", preprocessing), ("svr", SVR())])
grid_search = GridSearchCV(svr_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing.iloc[:5000], housing_labels.iloc[:5000])
```

The best model achieves the following score (evaluated using 3-fold cross validation):

```
svr_grid_search_rmse = -grid_search.best_score_
svr_grid_search_rmse
```

```
69814.13889867254
```

That's much worse than the `RandomForestRegressor` (but to be fair, we trained the model on much less data). Let's check the best hyperparameters found:

```
grid_search.best_params_
```

```
{'svr__C': 10000.0, 'svr__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.

2.

Exercise: *Try replacing the `GridSearchCV` with a `RandomizedSearchCV`.*

Warning: the following cell will take several minutes to run. You can specify `verbose=2` when creating the `RandomizedSearchCV` if you want to see the training details.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import expon, loguniform

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

# Note: gamma is ignored when kernel is "linear"
param_distribs = {
        'svr__kernel': ['linear', 'rbf'],
        'svr__C': loguniform(20, 200_000),
        'svr__gamma': expon(scale=1.0),
    }

rnd_search = RandomizedSearchCV(svr_pipeline,
                                param_distributions=param_distribs,
                                n_iter=50, cv=3,
                                scoring='neg_root_mean_squared_error',
                                random_state=42)
rnd_search.fit(housing.iloc[:5000], housing_labels.iloc[:5000])
```

```
RandomizedSearchCV(cv=3,
                   estimator=Pipeline(steps=[('preprocessing',
                                              ColumnTransformer(remainder=Pipeline(steps=[('simpleimputer',
                                                                                           SimpleImputer(strategy='median')),
```

```
                                                                                                  ('standardscaler',
                                                                                                   StandardScaler())]),
                                                                                transformers=[('bedrooms',
                                                                                               Pipeline(steps=[('simpleimputer',
                                                                                                                SimpleImputer(strategy='median')),
                                                                                                               ('functiontransformer',
                                                                                                                FunctionTransformer(feature_names_...

    <sklearn.compose._column_transformer.make_column_selector object at 0x1a57e3a00>)])),
                                                                                                ('svr', SVR())]),
                                                n_iter=50,
                                                param_distributions={'svr__C': <scipy.stats._distn_infrastructure.rv_continuous_frozen object at 0x1a5d4c3a0>,
                                                                     'svr__gamma': <scipy.stats._distn_infrastructure.rv_continuous_frozen object at 0x1a5d9ca00>,
                                                                     'svr__kernel': ['linear', 'rbf']},
                                                random_state=42, scoring='neg_root_mean_squared_error')
```

The best model achieves the following score (evaluated using 3-fold cross validation):

```
1 svr_rnd_search_rmse = -rnd_search.best_score_
2 svr_rnd_search_rmse
```

```
55853.88100300133
```

Now that's really much better, but still far from the `RandomForestRegressor`'s performance. Let's check the best hyperparameters found:

```
1 rnd_search.best_params_
```

```
{'svr__C': 157055.10989448498,
 'svr__gamma': 0.26497040005002437,
 'svr__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.

Note that we used the `expon()` distribution for `gamma`, with a scale of 1, so `RandomSearch` mostly searched for values roughly of that scale: about 80% of the samples were between 0.1 and 2.3 (roughly 10% were smaller and 10% were larger):

```
np.random.seed(42)

s = expon(scale=1).rvs(100_000)  # get 100,000 samples
((s > 0.105) & (s < 2.29)).sum() / 100_000
```

```
0.80066
```

We used the `loguniform()` distribution for `C`, meaning we did not have a clue what the optimal scale of `C` was before running the random search. It explored the range from 20 to 200 just as much as the range from 2,000 to 20,000 or from 20,000 to 200,000.

3.

Exercise: *Try adding a `SelectFromModel` transformer in the preparation pipeline to select only the most important attributes.*

Let's create a new pipeline that runs the previously defined preparation pipeline, and adds a `SelectFromModel` transformer based on a `RandomForestRegressor` before the final regressor:

```
from sklearn.feature_selection import SelectFromModel

selector_pipeline = Pipeline([
    ('preprocessing', preprocessing),
    ('selector', SelectFromModel(RandomForestRegressor(random_state=42),
                                 threshold=0.005)),  # min feature importance
    ('svr', SVR(C=rnd_search.best_params_["svr__C"],
                gamma=rnd_search.best_params_["svr__gamma"],
                kernel=rnd_search.best_params_["svr__kernel"])),
])
```

```
selector_rmses = -cross_val_score(selector_pipeline,
                                  housing.iloc[:5000],
                                  housing_labels.iloc[:5000],
                                  scoring="neg_root_mean_squared_error",
                                  cv=3)
pd.Series(selector_rmses).describe()
```

```
count        3.000000
mean     56211.362086
```

```
std       1922.002802
min      54150.008629
25%      55339.929909
50%      56529.851189
75%      57242.038815
max      57954.226441
dtype: float64
```

Oh well, feature selection does not seem to help. But maybe that's just because the threshold we used was not optimal. Perhaps try tuning it using random search or grid search?

4.

Exercise: *Try creating a custom transformer that trains a k-Nearest Neighbors regressor (`sklearn.neighbors.KNeighborsRegressor`) in its `fit()` method, and outputs the model's predictions in its `transform()` method. Then add this feature to the preprocessing pipeline, using latitude and longitude as the inputs to this transformer. This will add a feature in the model that corresponds to the housing median price of the nearest districts.*

Rather than restrict ourselves to k-Nearest Neighbors regressors, let's create a transformer that accepts any regressor. For this, we can extend the `MetaEstimatorMixin` and have a required `estimator` argument in the constructor. The `fit()` method must work on a clone of this estimator, and it must also save `feature_names_in_`. The `MetaEstimatorMixin` will ensure that `estimator` is listed as a required parameters, and it will update `get_params()` and `set_params()` to make the estimator's hyperparameters available for tuning. Lastly, we create a `get_feature_names_out()` method: the output column name is the ...

```
1 from sklearn.neighbors import KNeighborsRegressor
2 from sklearn.base import MetaEstimatorMixin, clone
3
4 class FeatureFromRegressor(MetaEstimatorMixin, BaseEstimator, TransformerMixin):
5     def __init__(self, estimator):
6         self.estimator = estimator
7
8     def fit(self, X, y=None):
9         estimator_ = clone(self.estimator)
10        estimator_.fit(X, y)
11        self.estimator_ = estimator_
```