Lecture_2b_Regression

August 29, 2020

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.

The first task you are asked to perform is to build amodel of housing prices in California using the California census data. This data has metrics such as the population, median income, median housing price, and so on for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). We will just call them "districts" for short. Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics. The project should identify the district worth investing for a company.

#Picture: Data -> Price Prediction -> Investment Decisions Currently the prices are estimated by experts with error of 15%. We need to find the right alhorithm. The outcomes are clearly labeled: hence this is a supervised problem. We need to prodict a continious number – hence the multivariate regression.

** Select a Performance Measure ** Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (RMSE). It measures the standard deviations of the errors the system makes in its predictions. (Example with probabilities, 68%, 95%). Formula:

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

m is the number of instances in the dataset you are measuring the RMSE on. For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then $m = 2,000$. $x(i)$ is a vector of all the feature values (excluding the label) of the ith instance in the dataset, and $y(i)$ is its label (the desired output value for that instance). (Example of prediction with multiple features).

RMSE is the Eucleadian norm l_2, instead of square one can use a Manhattan norm l_1 or powers higher than 2 l_n. The higher the power of the norm the more it is sensitive to outliers. RMSE is easy and differentiate and historically preferred.

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:

[135]:
```
# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)

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

# Common imports
import numpy as np
import os

# 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):
    path = os.path.join(fig_id + ".png")
    print("Saving figure", fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format='png', dpi=300)

# Ignore useless warnings (see SciPy issue #5998)
import warnings
warnings.filterwarnings(action="ignore", message="^internal gelsd")
```

2 Get the data

[137]:
```python
import os
import tarfile
import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/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):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    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()
```

[138]:
```python
fetch_housing_data()
```

[140]:
```python
#Convert data to pandas
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)
```

[141]:
```python
housing = load_housing_data()
housing.head()
```

[141]:
```
   longitude  latitude  …  median_house_value  ocean_proximity
0    -122.23     37.88  …            452600.0         NEAR BAY
1    -122.22     37.86  …            358500.0         NEAR BAY
2    -122.24     37.85  …            352100.0         NEAR BAY
3    -122.25     37.85  …            341300.0         NEAR BAY
4    -122.25     37.85  …            342200.0         NEAR BAY

[5 rows x 10 columns]
```

[142]:
```python
#describe the variables
housing.info()
# Object can contain any type of data. Here it is a string (text).
```

```
<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
```

[143]:
```python
housing["ocean_proximity"].value_counts()
# This is the distribution
```

[143]:
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

[144]:
```python
# Statistical description of the data
housing.describe()
# N = 20640 small, 207 blocks are missing bedroom. Unless we will impute our␣
 ↪final data will be 20433.
# Income is scaled to some scale between 0.5 and 15, and probably top-coded as␣
 ↪most income data is.
# Housing price was capped at 500K. We can eighter collect real data on these␣
 ↪areas or drop them, as we don't know the
# the distribution beyond 500K.
```

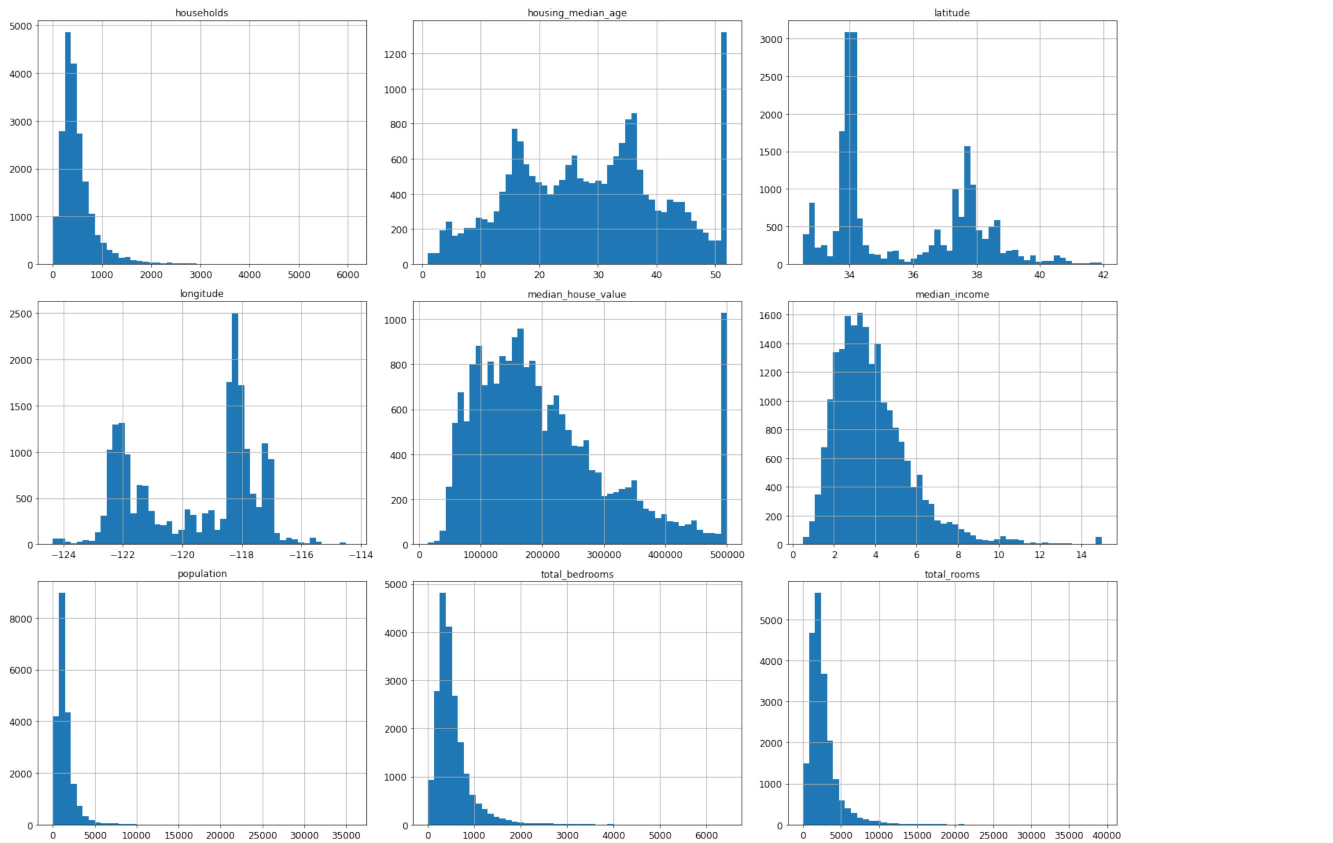
[144]:
```
          longitude      latitude  …  median_income  median_house_value
count  20640.000000  20640.000000  …   20640.000000        20640.000000
mean    -119.569704     35.631861  …       3.870671       206855.816909
std        2.003532      2.135952  …       1.899822       115395.615874
min     -124.350000     32.540000  …       0.499900        14999.000000
25%     -121.800000     33.930000  …       2.563400       119600.000000
50%     -118.490000     34.260000  …       3.534800       179700.000000
75%     -118.010000     37.710000  …       4.743250       264725.000000
max     -114.310000     41.950000  …      15.000100       500001.000000

[8 rows x 9 columns]
```

[146]:
```python
%matplotlib inline
#plot the data
```

```
import matplotlib.pyplot as plt
#create 50 bins (bars if n > 0)
# histogram of all vaiables
housing.hist(bins=50, figsize=(20,15))
save_fig("attribute_histogram_plots")
plt.show()
# Housing age was topcoded at 52, value at 500,000, median income at 15,
# ML algorithms works the best with the bell-shaped data. It is better to transform the tail-heavy data to bell-shape.
```

Saving figure attribute_histogram_plots



[147]:
```
# to make this notebook's output identical at every run
np.random.seed(42)
```

[148]:
```
#The book discuses several spliting methods. The good basic method is provided by sklearn.
#there is a random_state parameter for random generator seed
# You pass it multiple datasets with an identical number of rows, and it will split them on the same indices
from sklearn.model_selection import train_test_split
```

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

[149]:
```
test_set.head()
```

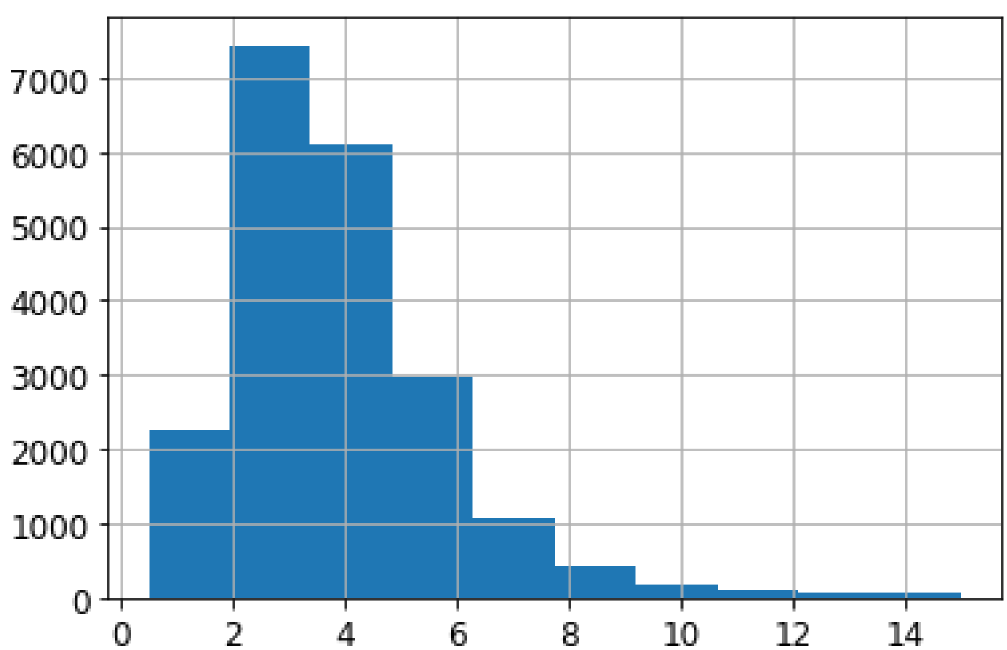
[149]:
```
       longitude  latitude  …  median_house_value  ocean_proximity
20046    -119.01     36.06  …             47700.0           INLAND
3024     -119.46     35.14  …             45800.0           INLAND
15663    -122.44     37.80  …            500001.0         NEAR BAY
20484    -118.72     34.28  …            218600.0        <1H OCEAN
9814     -121.93     36.62  …            278000.0       NEAR OCEAN

[5 rows x 10 columns]
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. If US population is 53.1% female and 48.7% male. A random draw of 1000 people may have very different share, like 60% male and 40% female. We can impose a restriction on sampling draws to maintain the desired ratio – stratified sampling. It is easy with discrete, for continious variable like income we need to break it into strata:

[150]:
```
housing["median_income"].hist()
```

[150]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4c55913c88>
```



Most median income values are clustered around 2–5 (tens of thousands of dollars), though there is a long right tail. The number of strata should be large enough to insure the similar income distribution and small enough not to dominate random selection.

We create an income category attribute by dividing the median income by 1.5, and rounding up using ceil (to have discrete categories), and then merging all the categories greater than 5 into category 5:

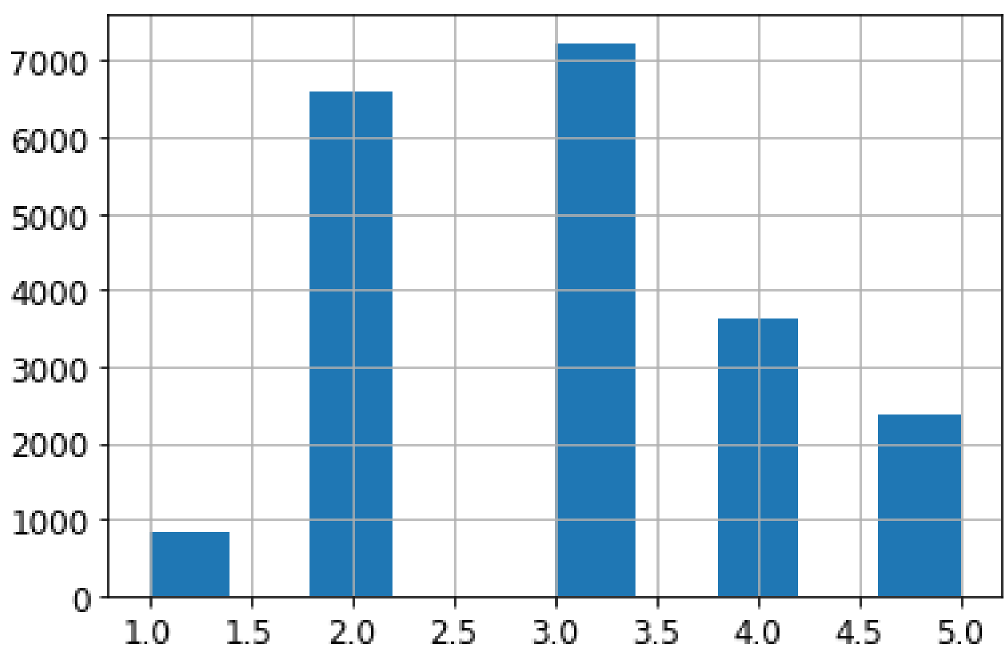
[151]:
```
# Divide by 1.5 to limit the number of income categories. Ceiling rounds up the␣
↪numbers
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
# Label those above 5 as 5
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

[152]:
```
housing["income_cat"].value_counts()
```

[152]:
```
3.0    7236
2.0    6581
4.0    3639
5.0    2362
1.0     822
Name: income_cat, dtype: int64
```

[153]:
```
# Income transformation
housing["income_cat"].hist()
```

[153]: <matplotlib.axes._subplots.AxesSubplot at 0x7f4c582e0c50>



[154]:
```python
# Not let's run a stratified sampling. Import standard command from sklearn
from sklearn.model_selection import StratifiedShuffleSplit

# Code a commande split: 80/20 with a single splint. Random generate is 42.
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
# Create two datasets by splitting the housing data stratified by␣
↪housing["income_cat"]
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]
```

[155]:
```python
train_index
```

[155]:
```
array([17606, 18632, 14650, …, 13908, 11159, 15775])
```

[156]:
```python
# Show distribution of income of the split data generated by the stratfication
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
```

[156]:
```
3.0    0.350533
2.0    0.318798
4.0    0.176357
5.0    0.114583
1.0    0.039729
Name: income_cat, dtype: float64
```

[157]:
```python
# Compare with overal sample. Almost identical
housing["income_cat"].value_counts() / len(housing)
```

[157]:
```
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

[158]:
```python
# Let's compare the randomization erros for stratified and non-stratified data
# create function that returns the distribution of income categories
def income_cat_proportions(data):
    return data["income_cat"].value_counts() / len(data)
# Split the data into training and test
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
# Create pandas data that merges three vectors and sorts data
compare_props = pd.DataFrame({
    # Overall sampl
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_cat_proportions(test_set),
```

```
}).sort_index()
# add columns for deivation rate.
compare_props["Rand. %error"] =  100 * compare_props["Random"] /␣
 ↪compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] /␣
 ↪compare_props["Overall"] - 100
```

[159]:
```
compare_props
# Stratified sampling performs much better
```

[159]:

	Overall	Stratified	Random	Rand. %error	Strat. %error
1.0	0.039826	0.039729	0.040213	0.973236	-0.243309
2.0	0.318847	0.318798	0.324370	1.732260	-0.015195
3.0	0.350581	0.350533	0.358527	2.266446	-0.013820
4.0	0.176308	0.176357	0.167393	-5.056334	0.027480
5.0	0.114438	0.114583	0.109496	-4.318374	0.127011

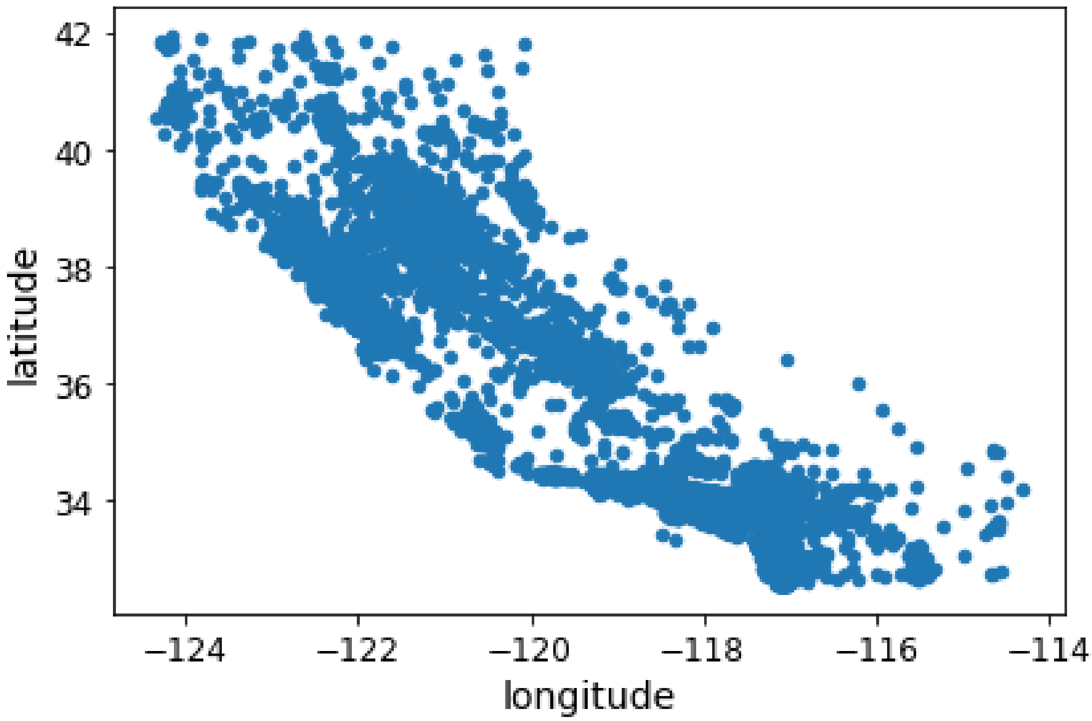
[160]:
```
# Next we drop the income_cat to return the data to the original state
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

3 Discover and visualize the data to gain insights

[161]:
```
# We create a reference to the strat_train_set. If we change strat_train_set␣
 ↪then the housing data will change as well
housing = strat_train_set.copy()
```

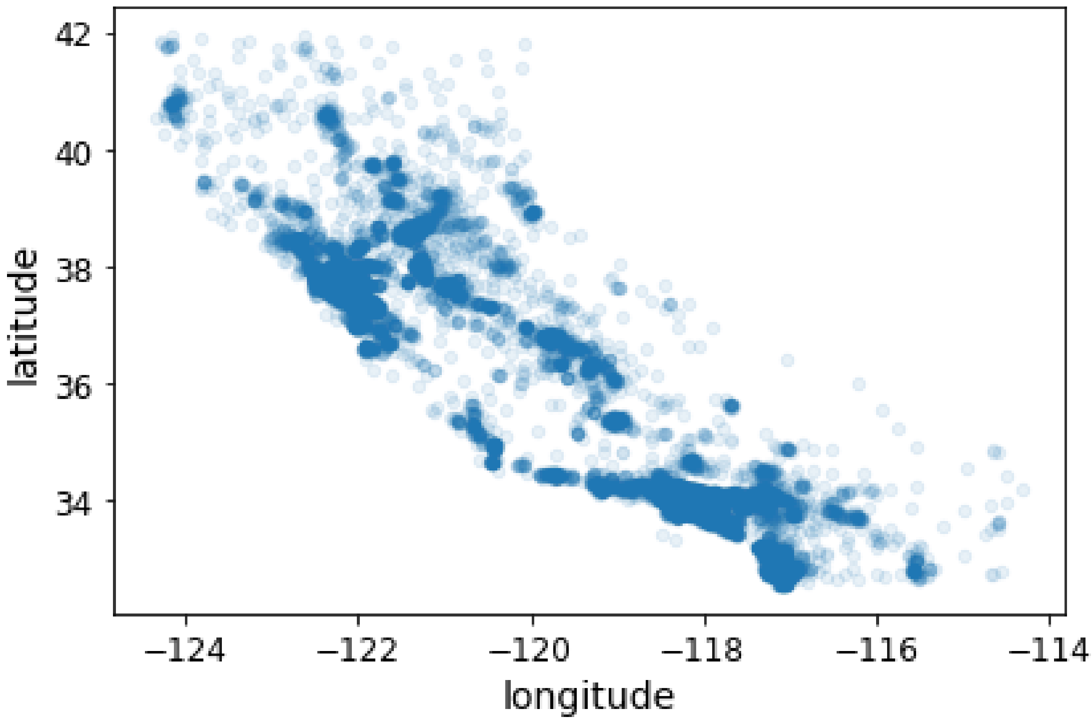
[162]:
```
# plot housing data
housing.plot(kind="scatter", x="longitude", y="latitude")
save_fig("bad_visualization_plot")
```

Saving figure bad_visualization_plot



[163]:
```
# Parameter alpha is the transparency of the dots, we see density.
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
save_fig("better_visualization_plot")
```

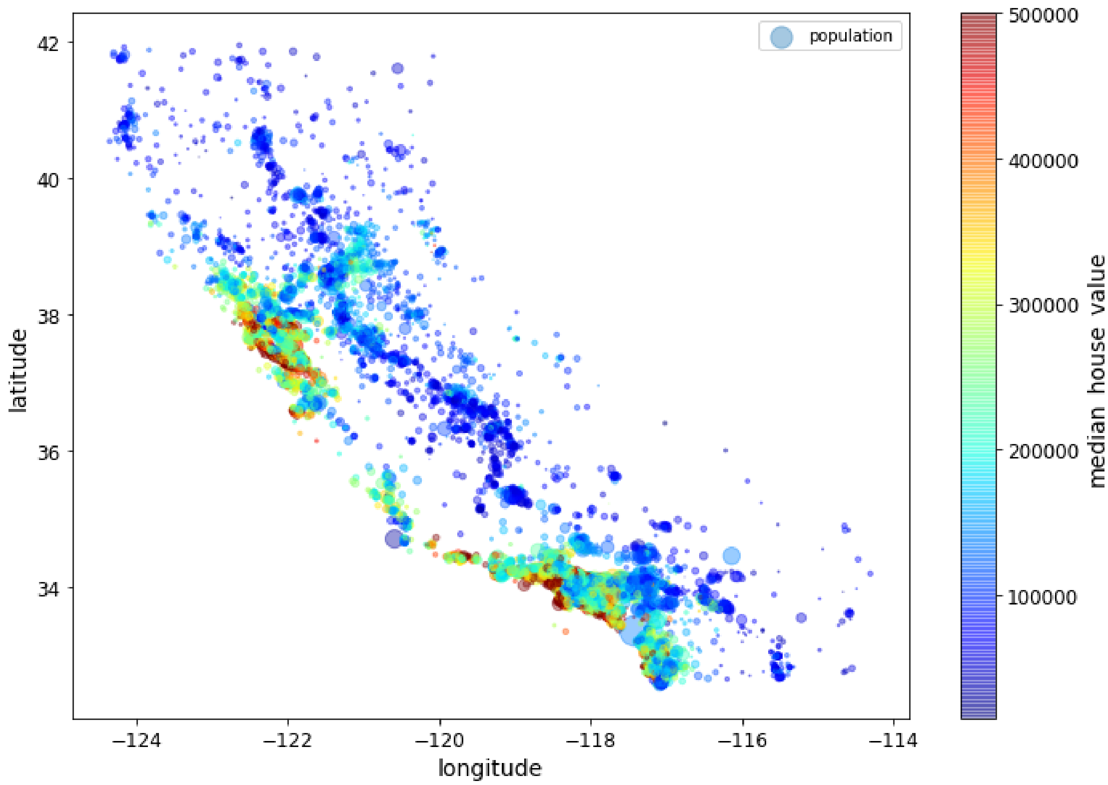
```
Saving figure better_visualization_plot
```



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.

[164]:
```
# Size of the dot is population/100, color is the housing value, jet shows␣
↪colors from red to blue.
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
```



[165]:
```python
# download California map
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "california.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/end_to_end_project/" + filename
urllib.request.urlretrieve(url, os.path.join(filename))
```

```
Downloading california.png
```

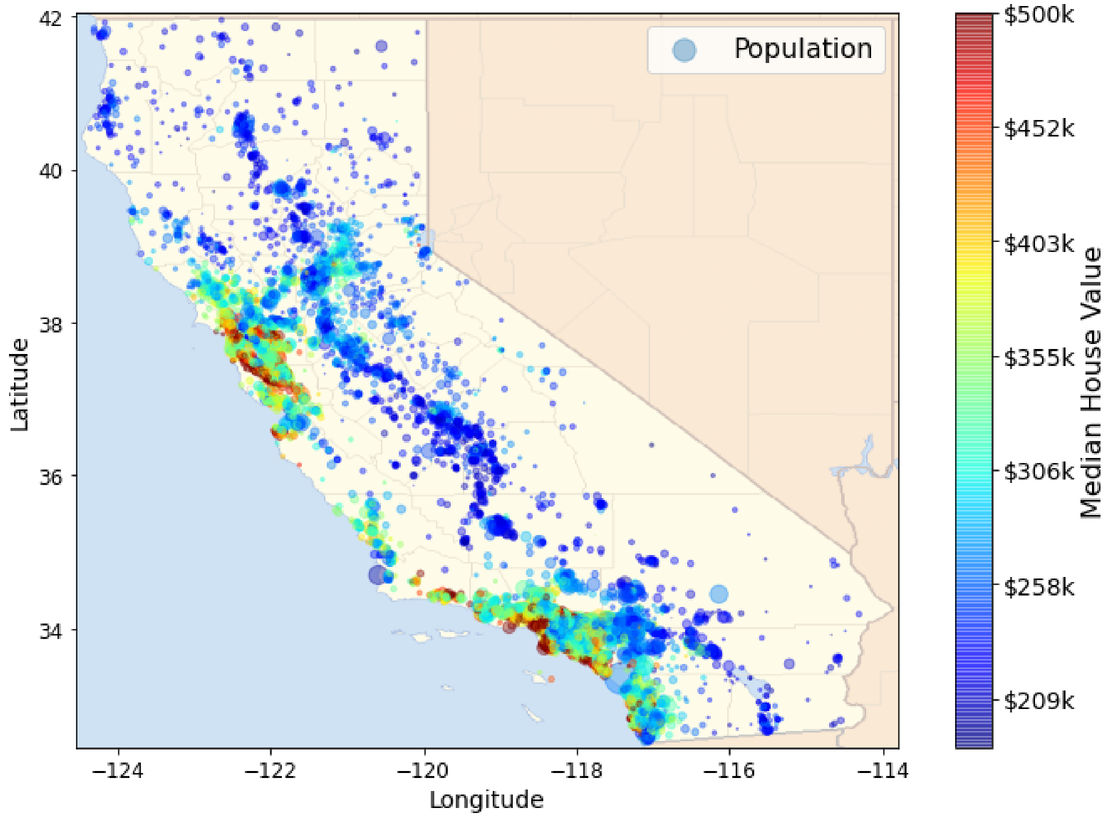
[165]:
```
('california.png', <http.client.HTTPMessage at 0x7f4c669a7630>)
```

[166]:
```python
import matplotlib.image as mpimg
california_img=mpimg.imread(filename)
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)
```

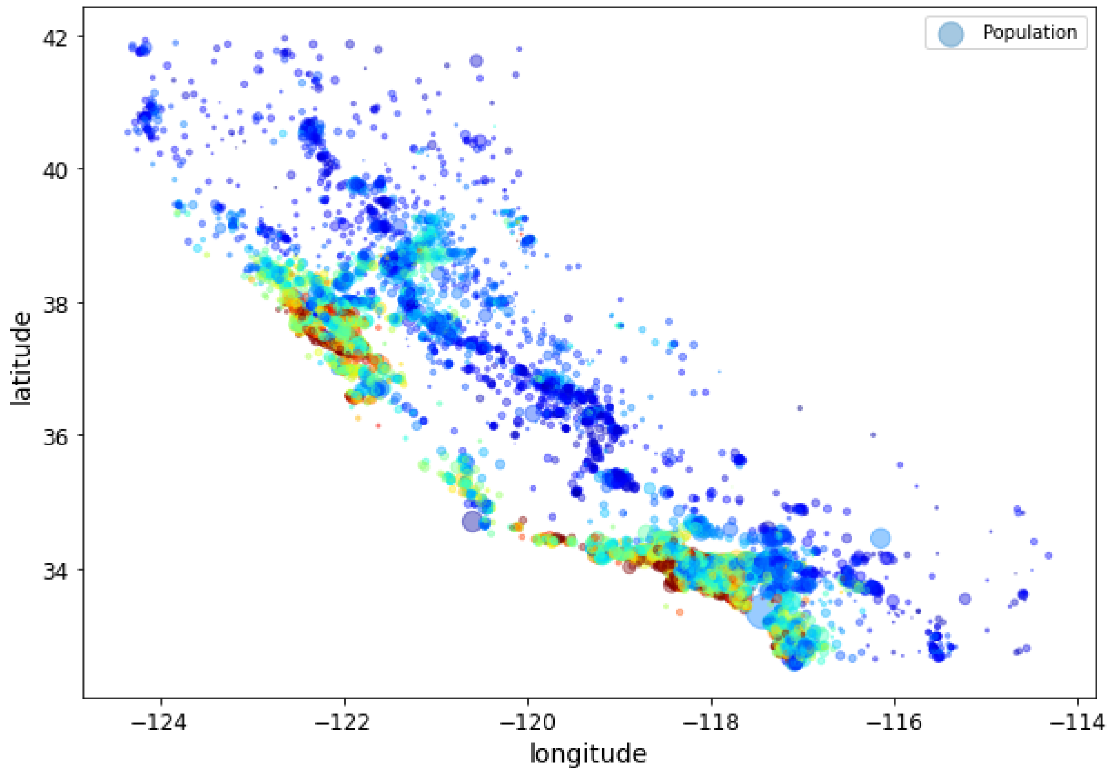
```
prices = housing["median_house_value"]
tick_values = np.linspace(prices.min(), prices.max(), 11)
cbar = plt.colorbar(ticks=tick_values/prices.max())
cbar.ax.set_yticklabels(["$%dk"%(round(v/1000)) for v in tick_values],␣
 ↪fontsize=14)
cbar.set_label('Median House Value', fontsize=16)

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

```
Saving figure california_housing_prices_plot
```



[167]:
```
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,
                      )
```



[168]:
```
# Create a matrix of correlations of all variables
corr_matrix = housing.corr()
```

[170]:
```
# Show the correlates of househing price.
corr_matrix["median_house_value"].sort_values(ascending=False)
```

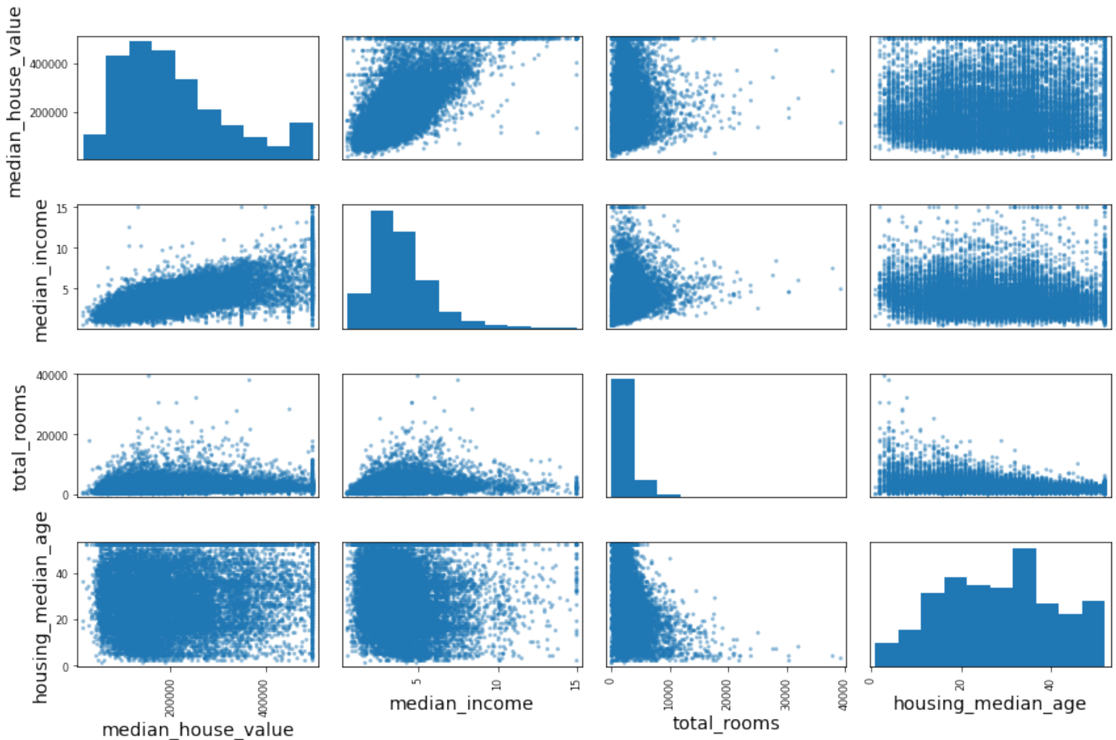
[170]:
```
median_house_value    1.000000
median_income         0.687160
total_rooms           0.135097
housing_median_age    0.114110
households            0.064506
total_bedrooms        0.047689
population           -0.026920
longitude            -0.047432
latitude             -0.142724
Name: median_house_value, dtype: float64
```

[171]:
```
# Another visualisation tool. Look at correlation matrix. Relatively strong␣
↪correlaion between house value and income
# Other things look weak

from pandas.plotting import scatter_matrix
```

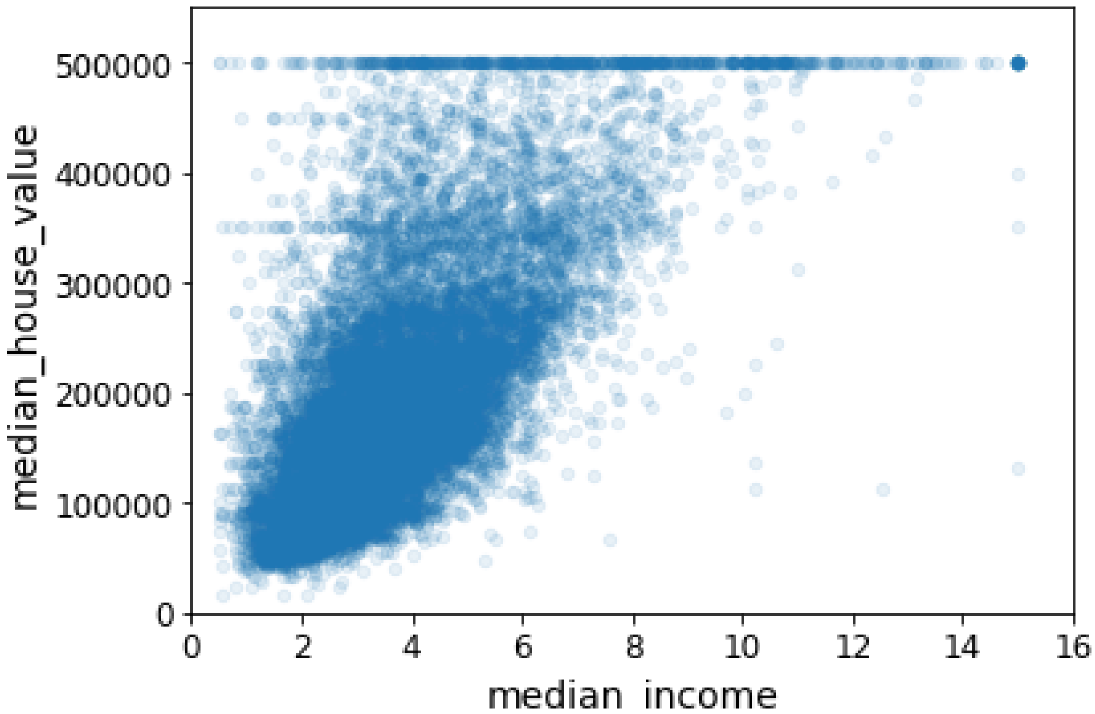
```
attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")
```

```
Saving figure scatter_matrix_plot
```



[175]:
```
# Look at price-income relationship in a greater detail.
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
```



[176]:
```
# Some attributes have a tail-heavy distribution, so you may want to transform␣
↪them (e.g., by computing their logarithm).
# We try out various attribute combinations. For example, the total number of␣
↪rooms in a district is not very
#useful if you don’t know how many households there are. What you really want␣
↪is the number of rooms
#per household. Similarly, the total number of bedrooms by itself is not very␣
↪useful: you probably want to
#compare it to the number of rooms. And the population per household


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"]
```

[177]:
```
corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

[177]:
```
median_house_value          1.000000
median_income               0.687160
rooms_per_household         0.146285
total_rooms                 0.135097
housing_median_age          0.114110
```

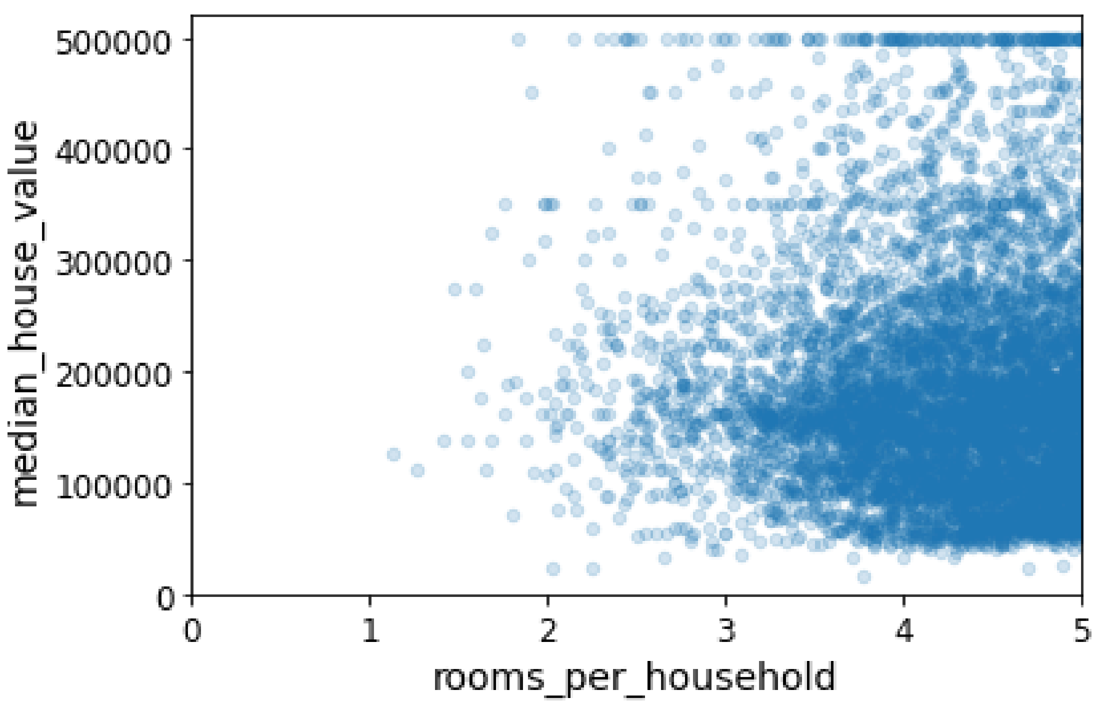
```
households                  0.064506
total_bedrooms              0.047689
population_per_household   -0.021985
population                 -0.026920
longitude                  -0.047432
latitude                   -0.142724
bedrooms_per_room          -0.259984
Name: median_house_value, dtype: float64
```

[178]:
```
housing.head()
```

[178]:
```
       longitude  latitude  …  bedrooms_per_room  population_per_household
17606    -121.89     37.29  …           0.223852                  2.094395
18632    -121.93     37.05  …           0.159057                  2.707965
14650    -117.20     32.77  …           0.241291                  2.025974
3230     -119.61     36.31  …           0.200866                  4.135977
3555     -118.59     34.23  …           0.231341                  3.047847

[5 rows x 13 columns]
```

[179]:
```
housing.plot(kind="scatter", x="rooms_per_household", y="median_house_value",
             alpha=0.2)
plt.axis([0, 5, 0, 520000])
plt.show()
```



[180]:
```
strat_train_set.describe()
```

[180]:
```
          longitude      latitude  …  median_income  median_house_value
count  16512.000000  16512.000000  …   16512.000000        16512.000000
mean    -119.575834     35.639577  …       3.875589       206990.920724
std        2.001860      2.138058  …       1.904950       115703.014830
min     -124.350000     32.540000  …       0.499900        14999.000000
25%     -121.800000     33.940000  …       2.566775       119800.000000
50%     -118.510000     34.260000  …       3.540900       179500.000000
75%     -118.010000     37.720000  …       4.744475       263900.000000
max     -114.310000     41.950000  …      15.000100       500001.000000

[8 rows x 9 columns]
```

4 Prepare the data for Machine Learning algorithms

[181]:
```
# It does not affect the training set, we just created a copy
housing = strat_train_set.drop("median_house_value", axis=1) # drop labels for␣
↪training set
# create separate label vector
housing_labels = strat_train_set["median_house_value"].copy()
```

[182]:
```
sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
sample_incomplete_rows
# WE have some incomeplete observations for houses with missing bedroom␣
↪information.
```

[182]:
```
       longitude  latitude  …  median_income  ocean_proximity
4629     -118.30     34.07  …         2.2708        <1H OCEAN
6068     -117.86     34.01  …         5.1762        <1H OCEAN
17923    -121.97     37.35  …         4.6328        <1H OCEAN
13656    -117.30     34.05  …         1.6675           INLAND
19252    -122.79     38.48  …         3.1662        <1H OCEAN

[5 rows x 9 columns]
```

[183]:
```
# We can eighter drop missing observations
sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
```

[183]:
```
Empty DataFrame
Columns: [longitude, latitude, housing_median_age, total_rooms, total_bedrooms,
population, households, median_income, ocean_proximity]
Index: []
```

[184]:
```
# Or drop the variable with missing observation.
sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
```

[184]:
```
       longitude  latitude  …  median_income  ocean_proximity
4629     -118.30     34.07  …         2.2708        <1H OCEAN
6068     -117.86     34.01  …         5.1762        <1H OCEAN
17923    -121.97     37.35  …         4.6328        <1H OCEAN
13656    -117.30     34.05  …         1.6675           INLAND
19252    -122.79     38.48  …         3.1662        <1H OCEAN

[5 rows x 8 columns]
```

[187]:
```
# Or replace the missing bedroom values with the median bedroom number
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
sample_incomplete_rows
```

[187]:
```
       longitude  latitude  …  median_income  ocean_proximity
4629     -118.30     34.07  …         2.2708        <1H OCEAN
6068     -117.86     34.01  …         5.1762        <1H OCEAN
17923    -121.97     37.35  …         4.6328        <1H OCEAN
13656    -117.30     34.05  …         1.6675           INLAND
19252    -122.79     38.48  …         3.1662        <1H OCEAN

[5 rows x 9 columns]
```

[188]:
```
# WE can also impute the missing data
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

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

[189]:
```
housing_num = housing.drop('ocean_proximity', axis=1)
# alternatively: housing_num = housing.select_dtypes(include=[np.number])
```

[192]:
```
imputer.fit(housing_num)
```

[192]:
```
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              missing_values=nan, strategy='median', verbose=0)
```

[193]:
```
imputer.statistics_
```

[193]:
```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

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

[194]:
```
housing_num.median().values
```

[194]:
```
array([-118.51  ,   34.26  ,   29.    , 2119.5   ,  433.    , 1164.    ,
        408.    ,    3.5409])
```

Transform the training set:

[195]:
```
# Imputer values. # imputer is the estimator. #See page 61 for more details.
X = imputer.transform(housing_num)
```

[196]:
```
# create Panda's file out of matrix created by the Imputer
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index = list(housing.index.values))
```

[197]:
```
# Look at imputed observations , total_bedrooms = 433
housing_tr.loc[sample_incomplete_rows.index.values]
```

[197]:
```
       longitude  latitude  …  households  median_income
4629     -118.30     34.07  …      1462.0         2.2708
6068     -117.86     34.01  …       727.0         5.1762
17923    -121.97     37.35  …       386.0         4.6328
13656    -117.30     34.05  …       391.0         1.6675
19252    -122.79     38.48  …      1405.0         3.1662

[5 rows x 8 columns]
```

[198]:
```
imputer.strategy
```

[198]:
```
'median'
```

[198]:
```

```

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

[199]:
```
housing_cat = housing[['ocean_proximity']]
# Look at top 10
housing_cat.head(10)
# Most estimators need to use numbers. So we need to convert them to codes.
```

[199]:
```
      ocean_proximity
17606       <1H OCEAN
18632       <1H OCEAN
14650      NEAR OCEAN
3230           INLAND
3555        <1H OCEAN
19480          INLAND
8879        <1H OCEAN
13685          INLAND
4937        <1H OCEAN
4861        <1H OCEAN
```

[200]:
```
# Load package for encoding
from sklearn.preprocessing import OrdinalEncoder
```

[201]:
```
ordinal_encoder = OrdinalEncoder()
# We transform housing categories into numeric codes. .fit_transform() fits␣
↪string labels into numeric codes and fit it into
# the data.
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
# show resulting data
housing_cat_encoded[:10]
# We converted text to codes
```

[201]:
```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

[114]:
```
# Look at the categories
# "<1HOCEAN" is mapped to 0, "INLAND" is mapped to 1, etc
ordinal_encoder.categories_
```

[114]:
```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

[202]:
```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

[202]:
```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

[203]:
```
housing_cat_1hot.toarray()
```

[203]:
```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       …,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

[204]:
```
cat_encoder = OneHotEncoder(sparse=False)
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

```
housing_cat_1hot
```

[204]:
```
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       …,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

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

[205]:
```
from sklearn.base import BaseEstimator, TransformerMixin

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

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

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

[ ]:

[206]:
```
# Our data is in the array form. Need to convert to pandas
housing_extra_attribs = pd.DataFrame(
    housing_extra_attribs,
    columns=list(housing.columns)+["rooms_per_household",␣
 ↪"population_per_household"])
housing_extra_attribs.head()
```

[206]:

	longitude	latitude	…	rooms_per_household	population_per_household
0	-121.89	37.29	…	4.62537	2.0944
1	-121.93	37.05	…	6.00885	2.70796

```
2    -117.2     32.77   …             4.22511                   2.02597
3   -119.61     36.31   …             5.23229                   4.13598
4   -118.59     34.23   …             4.50581                   3.04785

[5 rows x 11 columns]
```

Feature Scaling One of the most important transformations you need to apply to your data is feature scaling. With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Note that scaling the target values is generally not required.

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

Min-max scaling (many people call this normalization) : values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus the min. Scikit-Learn provides a transformer called MinMaxScaler for this. It has a feature_range hyperparameter that lets you change the range if you don't want 0–1 for some reason. $X_{norm} = \frac{X}{Max(X)-Min(X)}$ Standardization subtracts the mean value (so standardized values always have a zero mean), and then it divides by the variance so that the resulting distribution has unit variance (sometimes standard deviation). Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected.

$X_{std} = \frac{X-Mean(X)}{Var(X)}$ Scikit-Learn provides a transformer called StandardScaler for standardization

The more you automate these data preparation steps, the more combinations you can automatically try out, making it much more likely that you will find a great combination (and saving you a lot of time). Pipeline is the series of estimators and transformation applied together.

[207]:
```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

[208]:
```python
housing_num_tr
```

[208]:
```
array([[-1.15604281,  0.77194962,  0.74333089, …, -0.31205452,
        -0.08649871,  0.15531753],
```

```
       [-1.17602483,  0.6596948 , -1.1653172 , …,  0.21768338,
        -0.03353391, -0.83628902],
       [ 1.18684903, -1.34218285,  0.18664186, …, -0.46531516,
        -0.09240499,  0.4222004 ],
       …,
       [ 1.58648943, -0.72478134, -1.56295222, …,  0.3469342 ,
        -0.03055414, -0.52177644],
       [ 0.78221312, -0.85106801,  0.18664186, …,  0.02499488,
         0.06150916, -0.30340741],
       [-1.43579109,  0.99645926,  1.85670895, …, -0.22852947,
        -0.09586294,  0.10180567]])
```

And a transformer to just select a subset of the Pandas DataFrame columns:

[210]:
```
from sklearn.base import BaseEstimator, TransformerMixin

# Create a class to select numerical or categorical columns
# since Scikit-Learn doesn't handle DataFrames yet
class DataFrameSelector(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:

[210]:
```
```

[210]:
```
```

[211]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

[212]:
```
from sklearn.compose import ColumnTransformer

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

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

housing_prepared = full_pipeline.fit_transform(housing)
```

[213]:
```
housing_prepared = full_pipeline.fit_transform(housing)
housing_prepared
```

[213]:
```
array([[-1.15604281,  0.77194962,  0.74333089, …,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , …,  0.        ,
         0.        ,  0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, …,  0.        ,
         0.        ,  1.        ],
       …,
       [ 1.58648943, -0.72478134, -1.56295222, …,  0.        ,
         0.        ,  0.        ],
       [ 0.78221312, -0.85106801,  0.18664186, …,  0.        ,
         0.        ,  0.        ],
       [-1.43579109,  0.99645926,  1.85670895, …,  0.        ,
         1.        ,  0.        ]])
```

[214]:
```
housing_prepared.shape
```

[214]:
```
(16512, 16)
```

5 Select and train a model

[215]:
```
#Let’s first train a Linear Regression model, like we did in the previous␣
 ↪chapter:
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
# Fit linear regression
lin_reg.fit(housing_prepared, housing_labels)
```

[215]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

[216]:
```
# let's try the full pipeline on a few training instances
# Take first 5 observations
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
# Use cofficients estimates in the previous model to predict first five obs
```

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

```
Predictions: [210644.60459286 317768.80697211 210956.43331178  59218.98886849
 189747.55849879]
```

Compare against the actual values:

[217]:
```
# What we observe in data
print("Labels:", list(some_labels))
```

```
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

[218]:
```
# This is what the data looks like
some_data_prepared
```

[218]:
```
array([[-1.15604281,  0.77194962,  0.74333089, -0.49323393, -0.44543821,
        -0.63621141, -0.42069842, -0.61493744, -0.31205452, -0.08649871,
         0.15531753,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [-1.17602483,  0.6596948 , -1.1653172 , -0.90896655, -1.0369278 ,
        -0.99833135, -1.02222705,  1.33645936,  0.21768338, -0.03353391,
        -0.83628902,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [ 1.18684903, -1.34218285,  0.18664186, -0.31365989, -0.15334458,
        -0.43363936, -0.0933178 , -0.5320456 , -0.46531516, -0.09240499,
         0.4222004 ,  0.        ,  0.        ,  0.        ,  0.        ,
         1.        ],
       [-0.01706767,  0.31357576, -0.29052016, -0.36276217, -0.39675594,
         0.03604096, -0.38343559, -1.04556555, -0.07966124,  0.08973561,
        -0.19645314,  0.        ,  1.        ,  0.        ,  0.        ,
         0.        ],
       [ 0.49247384, -0.65929936, -0.92673619,  1.85619316,  2.41221109,
         2.72415407,  2.57097492, -0.44143679, -0.35783383, -0.00419445,
         0.2699277 ,  1.        ,  0.        ,  0.        ,  0.        ,
         0.        ]])
```

[220]:
```
# Import main metric to compare observation and prediction
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)
# Root mean squared error is the geometric average difference between␣
 ↪prediction and observation.
lin_rmse
# The average error of 68K is very large. We probably underfit the data. To fit␣
 ↪it better we could
# 1. add new/better features,
```

```
# 2. Use a more powerful model
# 3. Relax constraints such as regularization. (We don't use regularization in␣
↪the linear regrssion, so it does not apply)
```

[220]: 68628.19819848923

[221]:
```
from sklearn.metrics import mean_absolute_error

lin_mae = mean_absolute_error(housing_labels, housing_predictions)
lin_mae
# Average difference between observation and prediction
# RMSE is usually larger than MAE
```

[221]: 49439.89599001897

[222]:
```
# Train a very powere Decision Tree Regressor, we will cover it more in chaper 6
from sklearn.tree import DecisionTreeRegressor

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

[222]:
```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=42, splitter='best')
```

[223]:
```
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
tree_rmse
```

[223]: 0.0

[ ]:
```
# The model is too powerfull, our error is 0, we probably badly overfit the␣
↪data. Need to break data into training, testing
# and validation sets to test it.
```

6 Fine-tune your model

Powerful testing technique is cross-validation. it randomly splits the training set into 10 distinct subsets called folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

[224]:
```python
from sklearn.model_selection import cross_val_score
# Select cross validation with 10 folds
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
# The scores the negative MSE, so we use "minus" in the square root. This is␣
↪just a feature of cross-validation
tree_rmse_scores = np.sqrt(-scores)
```

[225]:
```python
# Create a function to display RMSE for all 10 folds, average and the standard␣
↪deviation of the scores
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
# Average RMSE is a whooping 71K, the regression tree model does not fit new␣
↪data so well.
```

```
Scores: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
 71115.88230639 75585.14172901 70262.86139133 70273.6325285
 75366.87952553 71231.65726027]
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

[226]:
```python
# Do the same for linear regression model
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)
# We have better RMSE on the testing data of 69K. Linear regression despite its␣
↪simplicity performs better than regression tree.
# Cross-validation allows you to see the stability of the RMSE estimates. The␣
↪average error of our MSE estimate
#is 2.7K in our prediction. If
```

```
Scores: [66782.73843989 66960.118071   70347.95244419 74739.57052552
 68031.13388938 71193.84183426 64969.63056405 68281.61137997
 71552.91566558 67665.10082067]
Mean: 69052.46136345083
Standard deviation: 2731.674001798344
```

We have better RMSE on the testing data of 69K. Linear regression despite its simplicity performs better than regression tree.Cross-validation allows you to see the stability of the RMSE estimates. The average error of our MSE estimate is 2.7K in our prediction. If the model takes too long, runnig it 10 times may be time-consuming.

Ensemble learing (learn later) allows to estimate many models and compare their perfomance. This particular command runs though a many types of random trees (Random forest) and averages the result. The result is much better than in using just one tree.

[227]:
```
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor(random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
```

[227]:
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=42, verbose=0, warm_start=False)
```

[228]:
```
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
forest_rmse
# RMSE is 18K in testing data. Maybe we overfit again.
```

[228]:
```
18603.515021376355
```

[229]:
```
#Let's check it
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)
# Forest regression has average RMSE of just 52K
```

```
Scores: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

[231]:
```
scores = cross_val_score(lin_reg, housing_prepared, housing_labels,␣
 ↪scoring="neg_mean_squared_error", cv=10)
pd.Series(np.sqrt(-scores)).describe()
```

[231]:
```
count       10.000000
mean     69052.461363
std       2879.437224
min      64969.630564
25%      67136.363758
50%      68156.372635
75%      70982.369487
```

```
max      74739.570526
dtype: float64
```

** Optimization ** Optimization is not only the choice is the best model, but mainly the choice of the hyperparameters and features inside a particular model. Doing it manually is very tedious.

Instead you should get Scikit-Learn's GridSearchCV to search for you. All you need to do is tell it which hyperparameters you want it to experiment with, and what values to try out, and it will evaluate all the possible combinations of hyperparameter values, using cross-validation. For example, the following code searches for the best combination of hyperparameter values for the RandomForestRegressor:

[232]:
```python
from sklearn.model_selection import GridSearchCV

# n_estimators is the number of trees in the forest. max_features is the␣
↪largest number of features in a forest,
# boostrap is the estimation of forest average by randomly dropping some trees.
param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

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

[232]:
```
GridSearchCV(cv=5, error_score=nan,
             estimator=RandomForestRegressor(bootstrap=True, ccp_alpha=0.0,
                                             criterion='mse', max_depth=None,
                                             max_features='auto',
                                             max_leaf_nodes=None,
                                             max_samples=None,
                                             min_impurity_decrease=0.0,
                                             min_impurity_split=None,
                                             min_samples_leaf=1,
                                             min_samples_split=2,
                                             min_weight_fraction_leaf=0.0,
                                             n_estimators=100, n_jobs=None,
                                             oob_score=False, random_state=42,
                                             verbose=0, warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'max_features': [2, 4, 6, 8],
                          'n_estimators': [3, 10, 30]},
```

```
                    {'bootstrap': [False], 'max_features': [2, 3, 4],
                     'n_estimators': [3, 10]}],
          pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
          scoring='neg_mean_squared_error', verbose=0)
```

The best hyperparameter combination found:

[233]:
```
grid_search.best_params_
```

[233]:
```
{'max_features': 8, 'n_estimators': 30}
```

[234]:
```
grid_search.best_estimator_
```

[234]:
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features=8, max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=30, n_jobs=None, oob_score=False,
                      random_state=42, verbose=0, warm_start=False)
```

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

[235]:
```
cvres = grid_search.cv_results_
# Zip merges two arrays in a 2 x n matrix
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
# Best testing RMSE if 49K.
```

```
63669.11631261028 {'max_features': 2, 'n_estimators': 3}
55627.099719926795 {'max_features': 2, 'n_estimators': 10}
53384.57275149205 {'max_features': 2, 'n_estimators': 30}
60965.950449450494 {'max_features': 4, 'n_estimators': 3}
52741.04704299915 {'max_features': 4, 'n_estimators': 10}
50377.40461678399 {'max_features': 4, 'n_estimators': 30}
58663.93866579625 {'max_features': 6, 'n_estimators': 3}
52006.19873526564 {'max_features': 6, 'n_estimators': 10}
50146.51167415009 {'max_features': 6, 'n_estimators': 30}
57869.25276169646 {'max_features': 8, 'n_estimators': 3}
51711.127883959234 {'max_features': 8, 'n_estimators': 10}
49682.273345071546 {'max_features': 8, 'n_estimators': 30}
62895.06951262424 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.176157539405 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.40652318466 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52724.9822587892 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.5691951261 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.495668875716 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

** Randomized Search ** The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead.

It is similar to GridSearchCV class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration.

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

[236]:
```
# Runs for 10 min!
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Set the range of parameters
param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

forest_reg = RandomForestRegressor(random_state=42)
# Set the number of iterations as 10
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)
```

[236]:
```
RandomizedSearchCV(cv=5, error_score=nan,
                   estimator=RandomForestRegressor(bootstrap=True,
                                                   ccp_alpha=0.0,
                                                   criterion='mse',
                                                   max_depth=None,
                                                   max_features='auto',
                                                   max_leaf_nodes=None,
                                                   max_samples=None,
                                                   min_impurity_decrease=0.0,
                                                   min_impurity_split=None,
                                                   min_samples_leaf=1,
                                                   min_samples_split=2,
                                                   min_weight_fraction_leaf=0.0,
                                                   n_estimators=100,
                                                   n_jobs=None,
oob_score=Fals…
                                                   warm_start=False),
                   iid='deprecated', n_iter=10, n_jobs=None,
                   param_distributions={'max_features':
```

```
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f4c66451e10>,
                                        'n_estimators':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f4c66451ba8>},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring='neg_mean_squared_error',
                   verbose=0)
```

[237]:

```
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
# Best model has RMSE = 49147, slightly lower than the previous grid search.␣
 ↪It's possible to improve result with
# more time/computing power
```

```
49150.70756927707 {'max_features': 7, 'n_estimators': 180}
51389.889203389284 {'max_features': 5, 'n_estimators': 15}
50796.155224308866 {'max_features': 3, 'n_estimators': 72}
50835.13360315349 {'max_features': 5, 'n_estimators': 21}
49280.9449827171 {'max_features': 7, 'n_estimators': 122}
50774.90662363929 {'max_features': 3, 'n_estimators': 75}
50682.78888164288 {'max_features': 3, 'n_estimators': 88}
49608.99608105296 {'max_features': 5, 'n_estimators': 100}
50473.61930350219 {'max_features': 3, 'n_estimators': 150}
64429.84143294435 {'max_features': 5, 'n_estimators': 2}
```

[238]:

```
# What are the most important features
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
```

[238]:

```
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```

[239]:

```
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
num_attribs = list(housing_num)
# List of categorical variables
cat_attribs = ["ocean_proximity"]
attributes = num_attribs + extra_attribs + cat_attribs
sorted(zip(feature_importances, attributes), reverse=True)
# By far the most important feature is median income
```

[239]:

```
[(0.36615898061813423, 'median_income'),
 (0.10879295677551575, 'pop_per_hhold'),
 (0.07334423551601243, 'longitude'),
 (0.06290907048262032, 'latitude'),
 (0.056419179181954014, 'rooms_per_hhold'),
```

```
 (0.053351077347675815, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402769, 'population'),
 (0.014672685420543239, 'total_rooms'),
 (0.014257599323407808, 'households'),
 (0.014106483453584104, 'total_bedrooms'),
 (0.010311488326303788, 'ocean_proximity')]
```

[240]:
```
# Use as an exersise
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)
```

[241]:
```
final_rmse
```

[241]:
```
47730.22690385927
```

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

[242]:
```
from scipy import stats
```

[243]:
```
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)))
```

[243]:
```
array([45685.10470776, 49691.25001878])
```

We could compute the interval manually like this:

[244]:
```
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)
```

[244]:
```
(45685.10470776, 49691.25001877858)
```

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

[245]:
```
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)
```

[245]:
```
(45685.717918136455, 49690.68623889413)
```

7 Extra material

7.1 A full pipeline with both preparation and prediction

[246]:
```
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)
```

[246]:
```
array([210644.60459286, 317768.80697211, 210956.43331178,  59218.98886849,
       189747.55849879])
```

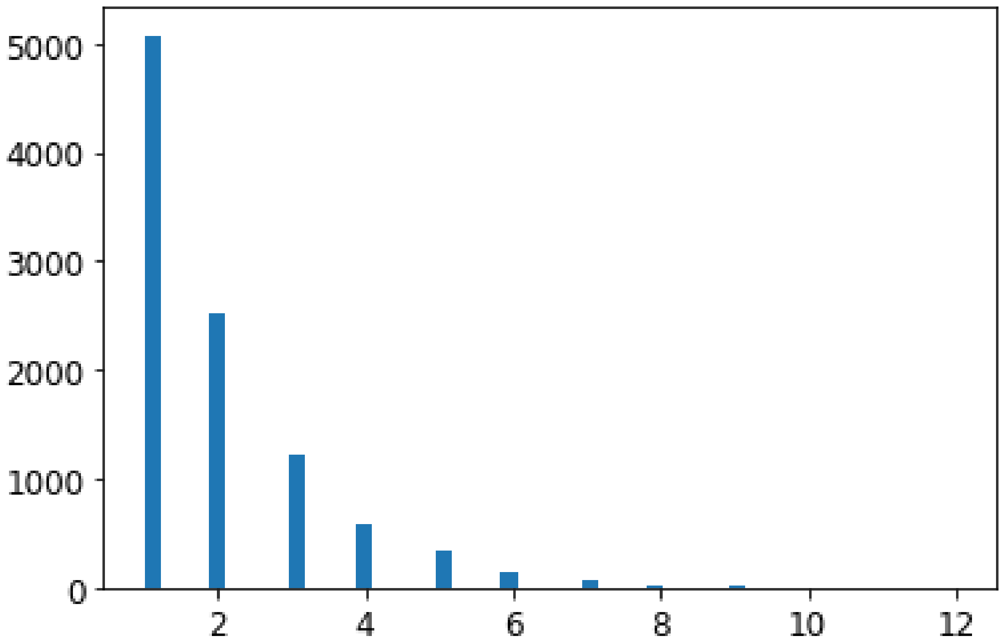
7.2 Model persistence using joblib

[247]:
```
my_model = full_pipeline_with_predictor
```

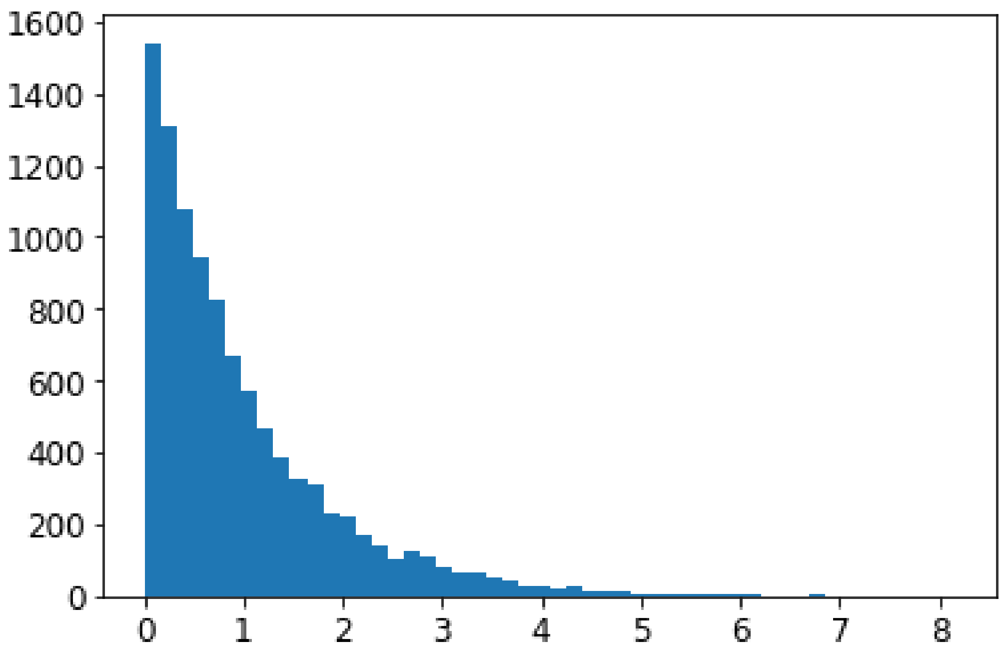
[248]:
```
import joblib
joblib.dump(my_model, "my_model.pkl") # DIFF
#...
my_model_loaded = joblib.load("my_model.pkl") # DIFF
```

7.3 Example SciPy distributions for `RandomizedSearchCV`

[249]:
```
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


```
[ ]:
```