Fetch housing data

In [80]:

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = "datasets/housing"
HOUSING_URL = DOWNLOAD_ROOT + HOUSING_PATH + "/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        # If housing_path directory does not exist, create one.
        os.makedirs(housing_path)
    # Define compressed file path for downloading and storing it
    # where it's dir_name=housing path and base_name=housing.tgz
    tgz_path = os.path.join(housing_path, "housing.tgz")
    # Download the file from the url=housing_url and
    # target_directory=tgz_path
    urllib.request.urlretrieve(housing_url, tgz_path)
    # Open the file and create a tarfile object
    housing_tgz = tarfile.open(tgz_path)
    # Extract the tarfile in the housing directory
    housing_tgz.extractall(path=housing_path)
    # Close the file
    housing_tgz.close()
```

In [81]:

```
# fetch_housing_data()
# Data already downloaded
```

Load data from csv

In [82]:

```
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)
```

Observing the data to find visible patterns and outliers in the attributes

In [83]:

```
housing = load_housing_data()
scaling_factor = 1000
housing["median_house_value"] /= scaling_factor
housing.head()
```

Out[83]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	househ
0	-122.23	37.88	41.0	880.0	129.0	322.0	1
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	11
2	-122.24	37.85	52.0	1467.0	190.0	496.0	1
3	-122.25	37.85	52.0	1274.0	235.0	558.0	2
4	-122.25	37.85	52.0	1627.0	280.0	565.0	2

In [84]:

```
# Info on the data
housing.info()
```

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

In [85]:

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

Out[85]:

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

In [86]:

```
housing.groupby(['ocean_proximity']).mean()
# As we will see from the data, houses closer to ocean are costly
```

Out[86]:

ocean_proximity	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
<1H OCEAN	-118.847766	34.560577	29.279225	2628.343586	546.539185
INLAND	-119.732990	36.731829	24.271867	2717.742787	533.881619
ISLAND	-118.354000	33.358000	42.400000	1574.600000	420.400000
NEAR BAY	-122.260694	37.801057	37.730131	2493.589520	514.182819
NEAR OCEAN	-119.332555	34.738439	29.347254	2583.700903	538.615677

In [87]:

```
housing.describe()
```

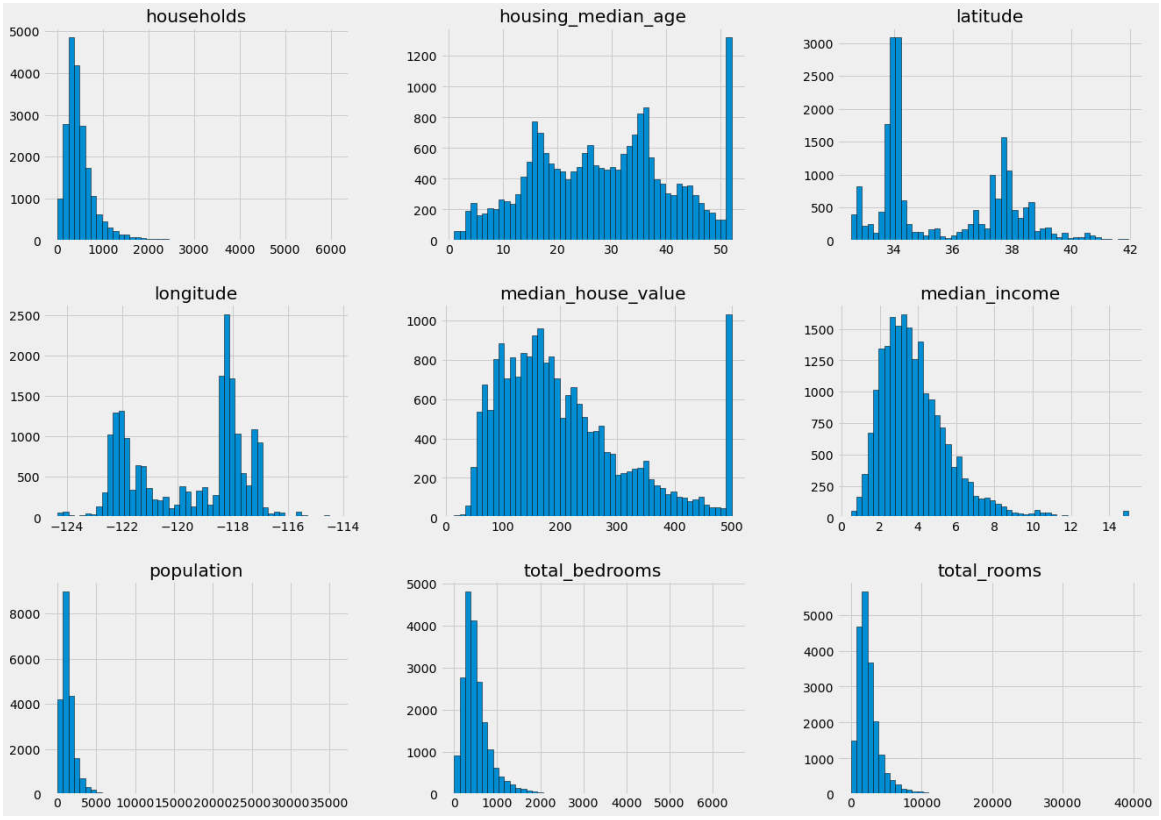
Out[87]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	p
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	2064
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	142
std	2.003532	2.135952	12.585558	2181.615252	421.385070	113
min	-124.350000	32.540000	1.000000	2.000000	1.000000	
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	78
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	116
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	172
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	3568

In [88]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')

housing.hist(bins=50, figsize=(20,15), edgecolor='black')
plt.show()
```



Splitting train and test data

1. Simple split

In [89]:

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

In [90]:

```
train_set, test_set = split_train_test(housing, 0.2)
```

In [91]:

```
print(len(train_set), "train +", len(test_set), "test")
```

```
16512 train + 4128 test
```

But this generates a different test set everytime the program is run again.

A common solution is to use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier). For example, you could compute a hash of each instance's identifier, keep only the last byte of the hash, and put the instance in the test set if this value is lower or equal to 51 (~20% of 256). This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset.

2. Split by id (which is a hash of index)

In [92]:

```
import hashlib
def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio
    # computes a hash of each row's id and keeps only the
    # last byte of the hash, and put the instance in the
    # test set if this value is lower or equal to 51
    # (~20% of 256)

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]    # Takes the index column from the dataset
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    # apply takes a function and applies it to all values of pandas series.
    return data.loc[~in_test_set], data.loc[in_test_set]
```

In [93]:

```
housing_with_id = housing.reset_index() # adds an 'index' column to dataframe
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset, and no row ever gets deleted. If this is not possible, then you can try to use the most stable features to build a unique identifier. For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID.

3. Split by id (which is a function of each blocks location)

In [94]:

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

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

4. Split using `train_test_split` of scikit-learn

In [95]:

```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant ***sampling bias***. When a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone booth. They try to ensure that these 1,000 people are representative of the whole population. For example, the US population is composed of 51.3% female and 48.7% male, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male. This is called ***stratified sampling***: the population is divided into homogeneous subgroups called ***strata***.

You should not have too many strata, and each stratum should be large enough. The following code creates an income category attribute by dividing the median income by 1.5 (to limit the number of income categories), and rounding up using ceil (to have discrete categories), and then merging all the categories greater than 5 into category 5:

In [96]:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

5. Stratified Shuffled splitting based on income category

In [97]:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):    # Strati
fication is done based on the 2nd arg
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

In [98]:

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

Out[98]:

```
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
```

In [99]:

```
strat_train_set.head()
```

Out[99]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	ho
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	
18632	-121.93	37.05	14.0	679.0	108.0	306.0	
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	

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

In [100]:

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

In [101]:

```
# Checking
strat_train_set.head()
```

Out[101]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	ho
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	
18632	-121.93	37.05	14.0	679.0	108.0	306.0	
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	

In [102]:

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

In [103]:

```
housing.info()
```

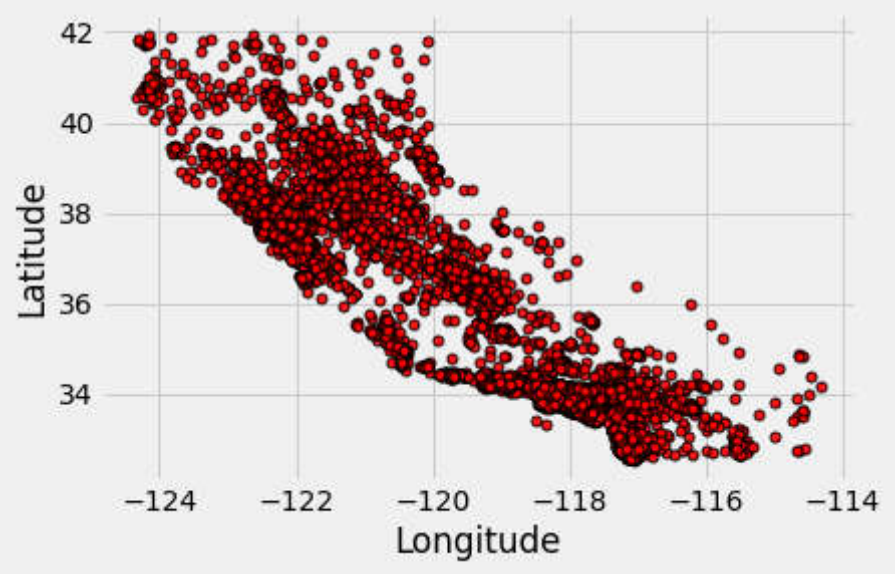
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16512 entries, 17606 to 15775
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16354 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
 8   median_house_value  16512 non-null  float64
 9   ocean_proximity     16512 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.4+ MB
```

Visualizing the data

In [104]:

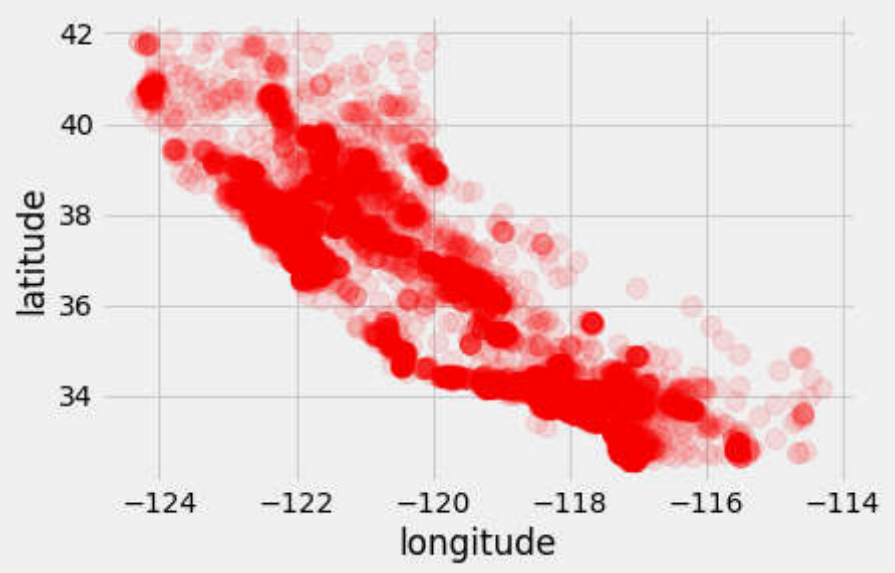
```
x = housing["longitude"]
y = housing["latitude"]
plt.scatter(x, y, s=100, c='red', marker='.',
            edgecolor='black', linewidth=1, alpha=0.9)
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.show()

# OR
# housing.plot(kind="scatter", x="longitude", y="latitude")
```



In [105]:

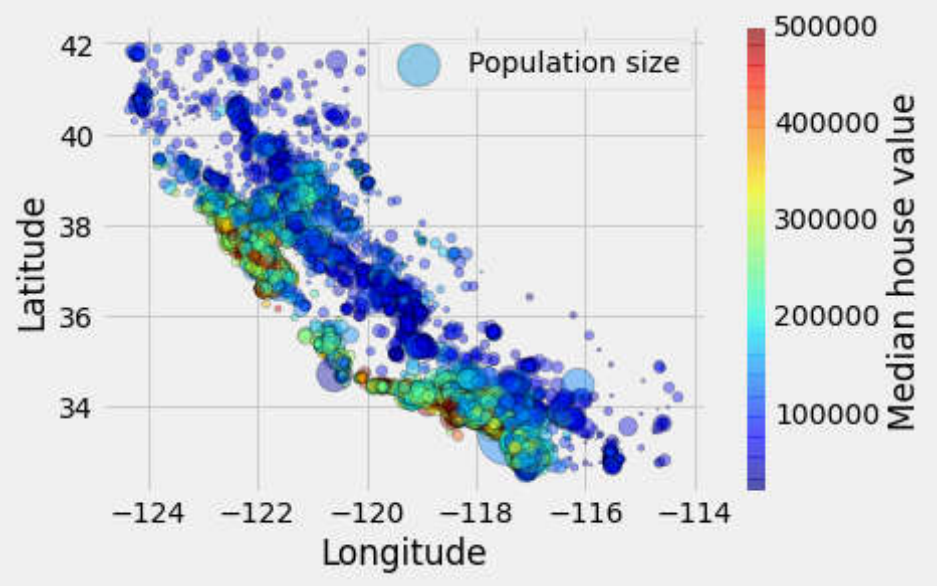
```
# Setting the alpha option to 0.1 and removing the edgecolor makes it much easier to vi
sualize
# the places where there is a high density of data points
housing.plot(kind="scatter", x="longitude", y="latitude", s=100, c='red', alpha=0.1, li
newidth=1)
plt.show()
```



In [106]:

```
x = housing["longitude"]
y = housing["latitude"]
size = housing["population"] / 10
color = housing["median_house_value"] * 1000
plt.scatter(x, y, s=size, c=color, label='Population size', marker='.',
            edgecolor='black', alpha=0.4,cmap='jet')
cbar = plt.colorbar()
plt.legend()
cbar.set_label('Median house value')
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.show()

# OR
# housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
            # s=housing["population"]/100, label="population",
            # c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
# )
# plt.legend()
```



Pearson's correlation coefficient

In [28]:

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

Out[28]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
longitude	1.000000	-0.924478	-0.105848	0.048871	0.076598
latitude	-0.924478	1.000000	0.005766	-0.039184	-0.072419
housing_median_age	-0.105848	0.005766	1.000000	-0.364509	-0.325047
total_rooms	0.048871	-0.039184	-0.364509	1.000000	0.929379
total_bedrooms	0.076598	-0.072419	-0.325047	0.929379	1.000000

In [29]:

```
# Now let's look at how much each attribute correlates with the median house value
corr_matrix["median_house_value"].sort_values(ascending=False)
```

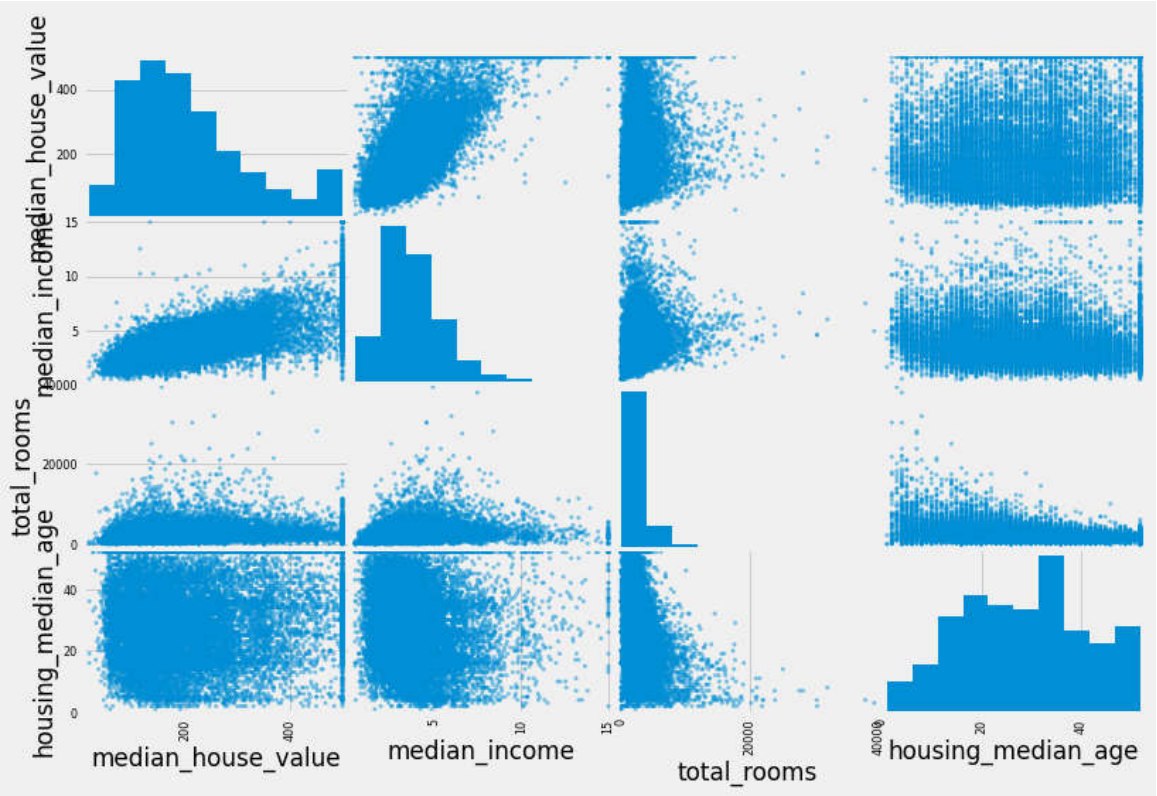
Out[29]:

```
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
```

Another way to check for correlation between attributes is to use Pandas' scatter_matrix function, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you would get 11^2 =121 plots. But we'll use the most prominent ones.

In [30]:

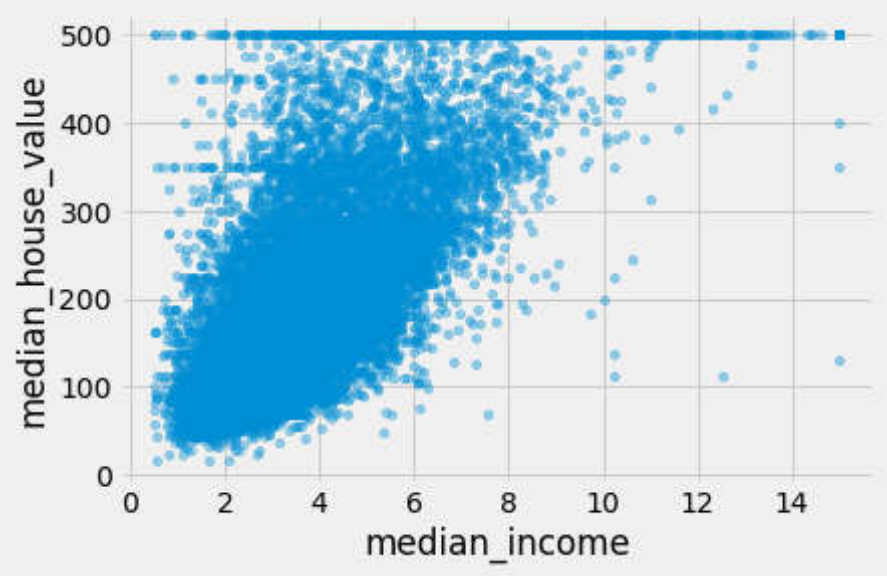
```
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms", "housing_median_ag
e"]
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```



The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot.

In [31]:

```
housing.plot(kind="scatter", x="median_income", y="median_house_value", alpha=0.4)
plt.show()
```



One last thing you may want to do before actually preparing the data for Machine Learning algorithms is to 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 also seems like an interesting attribute combination to look at.

In [32]:

```
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"]
```

In [33]:

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

Out[33]:

```
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
```

The new bedrooms_per_room attribute is much more correlated with the median house value than the total number of rooms or bedrooms. apparently houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

In [34]:

```
housing.head()
```

Out[34]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	ho
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	
18632	-121.93	37.05	14.0	679.0	108.0	306.0	
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	

Prepare the Data for Machine Learning Algorithms

Let's separate the predictors and the labels since we don't necessarily want to apply the same transformations to the predictors and the target values.

In [35]:

```
# Note that drop() creates a copy of the data and does not affect strat_train_set
housing = strat_train_set.drop("median_house_value", axis=1)    # Features
housing_labels = strat_train_set["median_house_value"].copy()   # Labels
```

Data Cleaning

Filling missing values

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

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the values to some value (zero, the mean, the median, etc.).

1. Using pandas methods

You can accomplish these easily using DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

In [36]:

```
# housing.dropna(subset=["total_bedrooms"])    # Option1
# housing.drop("total_bedrooms", axis=1)       # Option2
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median)     # Option3
```

Out[36]:

```
17606     351.0
18632     108.0
14650     471.0
3230      371.0
3555     1525.0
          ...
6563      236.0
12053     294.0
13908     872.0
11159     380.0
15775     682.0
Name: total_bedrooms, Length: 16512, dtype: float64
```

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

2. Using `Imputer` from Scikit-learn

Scikit-Learn provides a handy class to take care of missing values: `Imputer`.

In [37]:

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes(The attributes that do not take numerical values cannot be taken in this option. For those attributes, we may have to adopt option 1 or 2.), we need to create a copy of the data without the text attribute ocean_proximity:

In [38]:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

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

In [39]:

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

Out[39]:

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

In [40]:

```
housing_num.head()
```

Out[40]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	ho
17606	-121.89	37.29	38.0	1568.0	351.0	710.0	
18632	-121.93	37.05	14.0	679.0	108.0	306.0	
14650	-117.20	32.77	31.0	1952.0	471.0	936.0	
3230	-119.61	36.31	25.0	1847.0	371.0	1460.0	
3555	-118.59	34.23	17.0	6592.0	1525.0	4459.0	

The imputer has simply computed the median of each attribute and stored the result in its statistics_ instance variable. Only the total_bedrooms attribute had missing values, but we cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes.

In [41]:

```
imputer.statistics_
```

Out[41]:

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

In [42]:

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

Out[42]:

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

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

In [43]:

```
X = imputer.transform(housing_num)
X
# The result is a plain Numpy array containing the transformed features. If you want to
# put it back into a Pandas DataFrame, it’s simple:
```

Out[43]:

```
array([[-121.89  ,   37.29  ,   38.    , ...,  710.    ,  339.    ,
           2.7042],
       [-121.93  ,   37.05  ,   14.    , ...,  306.    ,  113.    ,
           6.4214],
       [-117.2   ,   32.77  ,   31.    , ...,  936.    ,  462.    ,
           2.8621],
       ...,
       [-116.4   ,   34.09  ,    9.    , ..., 2098.    ,  765.    ,
           3.2723],
       [-118.01  ,   33.82  ,   31.    , ..., 1356.    ,  356.    ,
           4.0625],
       [-122.45  ,   37.77  ,   52.    , ..., 1269.    ,  639.    ,
           3.575 ]])
```

In [44]:

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 16512 entries, 0 to 16511
Data columns (total 8 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           16512 non-null  float64
 1   latitude            16512 non-null  float64
 2   housing_median_age  16512 non-null  float64
 3   total_rooms         16512 non-null  float64
 4   total_bedrooms      16512 non-null  float64
 5   population          16512 non-null  float64
 6   households          16512 non-null  float64
 7   median_income       16512 non-null  float64
dtypes: float64(8)
memory usage: 1.0 MB
```

Handling texts and categorical attributes

In [45]:

```
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
housing_cat = housing["ocean_proximity"]
housing_cat_encoded = encoder.fit_transform(housing_cat)
housing_cat_encoded
```

Out[45]:

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

In [46]:

```
# You can look at the mapping that this encoder has learned using the classes_ attribut
e ("<1H
# OCEAN" is mapped to 0, "INLAND" is mapped to 1, etc.):
print(encoder.classes_)
```

```
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. Obviously this is not the case (for example, categories 0 and 4 are more similar than categories 0 and 1). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is "<1H OCEAN" (and 0 otherwise), another attribute equal to 1 when the category is "INLAND" (and 0 otherwise), and so on. This is called one-hot encoding, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). Scikit-Learn provides a **OneHotEncoder encoder to convert integer categorical values into one-hot vectors**. Let's encode the categories as one-hot vectors. Note that **`fit_transform()` expects a 2D array, but `housing_cat_encoded` is a 1D array, so we need to reshape it**:

In [47]:

```
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder()
housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
housing_cat_1hot
```

Out[47]:

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

Notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories.

In [48]:

```
#  If you really want to convert it to a (dense) NumPy array, just call the toarray() m
ethod:
housing_cat_1hot.toarray()
```

Out[48]:

```
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.]])
```

We can apply both transformations (from text categories to integer categories, then from integer categories to one-hot vectors) in one shot using the `LabelBinarizer` class:

In [49]:

```
from sklearn.preprocessing import LabelBinarizer
encoder = LabelBinarizer()
housing_cat_1hot = encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[49]:

```
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]])
```

Note that this returns a dense NumPy array by default. You can get a sparse matrix instead by passing sparse_output=True to the LabelBinarizer constructor:

In [50]:

```
from sklearn.preprocessing import LabelBinarizer
encoder = LabelBinarizer(sparse_output=True)
housing_cat_1hot = encoder.fit_transform(housing_cat)
housing_cat_1hot
```

Out[50]:

```
<16512x5 sparse matrix of type '<class 'numpy.int32'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

Custom Transformers

Although Scikit-Learn provides many useful transformers, you will need to write your own for tasks such as custom cleanup operations or combining specific attributes. To create custom transformers all you need is to create a class and implement three methods: `fit()` (returning self), `transform()`, and `fit_transform()`. You can get the last one for free by simply adding `TransformerMixin` as a base class. Also, if you add `BaseEstimator` as a base class (and avoid `*args` and `**kwargs` in your constructor) you will get two extra methods (`get_params()` and `set_params()`) that will be useful for automatic hyperparameter tuning. For example, here is a small transformer class that adds the combined attributes we discussed earlier:

In [51]:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_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, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

In [52]:

```
housing_extra_attribs
```

Out[52]:

```
array([[-121.89, 37.29, 38.0, ..., '<1H OCEAN', 4.625368731563422,
        2.094395280235988],
       [-121.93, 37.05, 14.0, ..., '<1H OCEAN', 6.008849557522124,
        2.7079646017699117],
       [-117.2, 32.77, 31.0, ..., 'NEAR OCEAN', 4.225108225108225,
        2.0259740259740258],
       ...,
       [-116.4, 34.09, 9.0, ..., 'INLAND', 6.34640522875817,
        2.742483660130719],
       [-118.01, 33.82, 31.0, ..., '<1H OCEAN', 5.50561797752809,
        3.808988764044944],
       [-122.45, 37.77, 52.0, ..., 'NEAR BAY', 4.843505477308295,
        1.9859154929577465]], dtype=object)
```

Feature Scaling

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

1. Min-max scaling (many people call this normalization) is quite simple: values are shifted and rescaled so that they end up ranging from 0 to 1. 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.
2. Standardization is quite different: first it 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. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms. However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Scikit-Learn provides a transformer called `StandardScaler` for standardization.

Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations.

Here is a small pipeline for the numerical attributes: from sklearn.pipeline import Pipeline from sklearn.preprocessing import StandardScaler num_pipeline = Pipeline([('imputer', Imputer(strategy="median")), # 1st arg: instance name, 2nd arg: Class name ('attribs_adder', CombinedAttributesAdder()), ('std_scaler', StandardScaler()),]) housing_num_tr = num_pipeline.fit_transform(housing_num)

The Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps. All but the last estimator must be transformers (i.e., they must have a `fit_transform()` method). The names can be anything you like. When you call the pipeline's `fit()` method, it calls `fit_transform()` sequentially on all transformers, passing the output of each call as the parameter to the next call, until it reaches the final estimator, for which it just calls the `fit()` method. The pipeline exposes the same methods as the final estimator. In this example, the last estimator is a `StandardScaler` , which is a transformer, so the pipeline has a `transform()` method that applies all the transforms to the data in sequence.

You now have a pipeline for numerical values, and you also need to apply the `LabelBinarizer` on the categorical values: how can you join these transformations into a single pipeline? Scikit-Learn provides a FeatureUnion class for this. You give it a list of transformers (which can be entire transformer pipelines), and when its `transform()` method is called it runs each transformer's `transform()` method in parallel, waits for their output, and then concatenates them and returns the result.

Each subpipeline starts with a selector transformer: it simply transforms the data by selecting the desired attributes (numerical or categorical), dropping the rest, and converting the resulting DataFrame to a NumPy array. There is nothing in Scikit-Learn to handle Pandas DataFrames,20 so we need to write a simple custom transformer for this task:

In [53]:

```
from sklearn.base import BaseEstimator, TransformerMixin
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
```

LabelBinarizer is not supposed to be used with X (Features), but is intended for labels only. Hence the fit and fit_transform methods are changed to include only single object y. But the Pipeline (which works on features) will try sending both X and y to it. Hence the error 'fit_transform() takes 2 positional arguments but 3 were given' will rise if we run the pipeline as suggested in the book. This can be solved by making a custom transformer that can handle 3 positional arguments:

In [54]:

```
from sklearn.base import TransformerMixin
class MyLabelBinarizer(TransformerMixin):
    def __init__(self, *args, **kwargs):
        self.encoder = LabelBinarizer(*args, **kwargs)
    def fit(self, x, y=0):
        self.encoder.fit(x)
        self.classes_, self.y_type_, self.sparse_input_ = self.encoder.classes_, self.e
ncoder.y_type_, self.encoder.sparse_input_
        return self
    def transform(self, x, y=0):
        return self.encoder.transform(x)
```

Keep your code the same only instead of using `LabelBinarizer()` , use the class we created : `MyLabelBinarizer()`

In [55]:

```
#  A full pipeline handling both numerical and categorical
# attributes may look like this:
from sklearn.pipeline import FeatureUnion    # For combining pipelines
from sklearn.pipeline import Pipeline    # For Creating Pipelines
from sklearn.preprocessing import StandardScaler

num_attribs = list(housing_num)    # Takes numerical features which are there is 'housi
ng_num'
cat_attribs = ["ocean_proximity"]    # Takes categorical feature 'ocean_proximity'
# Pipeline for numerical features
num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])
# Pipeline for textual features
cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', MyLabelBinarizer()),
    ])
# Combining the two pipelines
full_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])

# And you can run the whole pipeline simply:
housing_prepared = full_pipeline.fit_transform(housing)
housing_prepared
```

Out[55]:

```
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.        ]])
```

In [56]:

```
housing_prepared.shape
```

Out[56]:

```
(16512, 16)
```

Select and Train a Model

Training and Evaluating on the Training Set

Linear Regression model

In [57]:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Out[57]:

```
LinearRegression()
```

Done! You now have a working Linear Regression model.

In [58]:

```
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions:\t", lin_reg.predict(some_data_prepared))
```

```
Predictions:     [210.64460459 317.76880697 210.95643331  59.21898887 189.
7475585 ]
```

In [59]:

```
print("Labels:\t\t", list(some_labels))
```

```
Labels:          [286.6, 340.6, 196.9, 46.3, 254.5]
```

In [60]:

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

Out[60]:

```
68.62819819848923
```

In [61]:

```
lin_reg.score(some_data_prepared, some_labels)
```

Out[61]:

```
0.7862554884573294
```

The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Most districts' median_housing_values range between 120,000 dollars and 265,000 dollars, so a typical prediction error of $68,628 is not very satisfying. This is an example of a model underfitting the training data. When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough.

The main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model.

In [62]:

```
# Let's train a DecisionTreeRegressor
from sklearn.tree import DecisionTreeRegressor

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

Out[62]:

```
DecisionTreeRegressor()
```

In [63]:

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

Out[63]:

```
1.08182337217508e-13
```

In [64]:

```
tree_reg.score(some_data_prepared, some_labels)
```

Out[64]:

```
1.0
```

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

Better Evaluation Using Cross-Validation

One way to evaluate the Decision Tree model would be to use the train_test_split function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set.

The following code performs K-fold 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:

In [65]:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

In [66]:

```
# Let's look at the results:
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())
display_scores(tree_rmse_scores)
```

```
Scores: [69.67234801 67.19722739 71.47894357 68.3839494  70.95007765 72.95
020399
 70.64042783 72.22337397 75.20639154 69.60821878]
Mean: 70.83111621349558
Standard deviation: 2.1925815425361006
```

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

In [67]:

```
# Let's compute the same scores for the Linear Regression model just to be sure:
lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
                             scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

```
Scores: [66.78273844 66.96011807 70.34795244 74.73957053 68.03113389 71.19
384183
 64.96963056 68.28161138 71.55291567 67.66510082]
Mean: 69.05246136345082
Standard deviation: 2.7316740017983476
```

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

Let's try one last model now: the RandomForestRegressor. Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models is called Ensemble Learning, and it is often a great way to push ML algorithms even further.

In [68]:

```
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)
```

Out[68]:

```
RandomForestRegressor()
```

In [69]:

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

Out[69]:

```
18.65807458330976
```

In [70]:

```
forest_reg.score(some_data_prepared, some_labels)
```

Out[70]:

```
0.9690232051809806
```

In [71]:

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

```
Scores: [49.52310782 47.33199948 49.961667   52.13497475 49.6986289  53.37
039146
 48.87364571 48.11855857 52.83701843 50.43956387]
Mean: 50.228955599267906
Standard deviation: 1.8942487126511554
```

The score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. Possible solutions for overfitting are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data.

Saving Model

You should save every model you experiment with, so you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the crossvalidation scores and perhaps the actual predictions as well. You can easily save Scikit-Learn models by using Python's `pickle` module, or using `sklearn.externals.joblib` , which is more efficient at serializing large NumPy arrays:

from sklearn.externals import joblib joblib.dump(my_model, "my_model.pkl") # and later... my_model_loaded = joblib.load("my_model.pkl")

Fine-Tune Your Model

Grid Search

One way to fine tune the model would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, 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 crossvalidation. For example, the following code searches for the best combination of hyperparameter values for the `RandomForestRegressor`:

In [72]:

```
from sklearn.model_selection import GridSearchCV
param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')
grid_search.fit(housing_prepared, housing_labels)
```

Out[72]:

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

This param_grid tells Scikit-Learn to first evaluate all 3 × 4 = 12 combinations of `n_estimators` and `max_features` hyperparameter values specified in the first dict. Then try all 2 × 3 = 6 combinations of hyperparameter values in the second dict, but this time with the `bootstrap` hyperparameter set to `False` instead of `True` (which is the default value for this hyperparameter). It will train each model five times (since we are using five-fold cross validation). In other words, all in all, there will be 18 × 5 = 90 rounds of training!

In [73]:

```
grid_search.best_params_
```

Out[73]:

```
{'max_features': 8, 'n_estimators': 30}
```

Since 30 is the maximum value of n_estimators that was evalu- ated, you should probably evaluate higher values as well, since the score may continue to improve.

In [74]:

```
# You can also get the best estimator directly:
grid_search.best_estimator_
```

Out[74]:

```
RandomForestRegressor(max_features=8, n_estimators=30)
```

If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using crossvalidation, it retrains it on the whole training set. This is usually a good idea since feeding it more data will likely improve its performance.

In [75]:

```
# And of course the evaluation scores are also available:
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
64.94145074212032 {'max_features': 2, 'n_estimators': 3}
55.90918356237132 {'max_features': 2, 'n_estimators': 10}
53.08446330389762 {'max_features': 2, 'n_estimators': 30}
61.14022287188803 {'max_features': 4, 'n_estimators': 3}
52.421312810014136 {'max_features': 4, 'n_estimators': 10}
50.52567253095297 {'max_features': 4, 'n_estimators': 30}
58.958093298065805 {'max_features': 6, 'n_estimators': 3}
52.18223288624124 {'max_features': 6, 'n_estimators': 10}
49.92378676965928 {'max_features': 6, 'n_estimators': 30}
58.08940953974782 {'max_features': 8, 'n_estimators': 3}
51.962466362023875 {'max_features': 8, 'n_estimators': 10}
49.8415920592538 {'max_features': 8, 'n_estimators': 30}
62.74942671581503 {'bootstrap': False, 'max_features': 2, 'n_estimators':
3}
54.21337394989164 {'bootstrap': False, 'max_features': 2, 'n_estimators':
10}
59.83799048945298 {'bootstrap': False, 'max_features': 3, 'n_estimators':
3}
52.51205522589636 {'bootstrap': False, 'max_features': 3, 'n_estimators':
10}
58.97149616002273 {'bootstrap': False, 'max_features': 4, 'n_estimators':
3}
51.38352605475281 {'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. 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.

Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or “ensemble”) will often perform better than the best individual model.

Analyze the Best Models and Their Errors

In [76]:

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

Out[76]:

```
array([6.49887916e-02, 6.06008460e-02, 4.32688633e-02, 1.48530196e-02,
       1.55652932e-02, 1.59798246e-02, 1.33959405e-02, 3.70151861e-01,
       5.52209146e-02, 1.13890476e-01, 5.28223427e-02, 6.88526476e-03,
       1.66962032e-01, 1.66645461e-04, 2.10377187e-03, 3.14411315e-03])
```

In [77]:

```
# Let's display these importance scores next to their corresponding attribute names:
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
cat_one_hot_attribs = list(encoder.classes_)
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

Out[77]:

```
[(0.3701518605735106, 'median_income'),
 (0.16696203193357817, 'INLAND'),
 (0.11389047615453496, 'pop_per_hhold'),
 (0.06498879159538663, 'longitude'),
 (0.060600846002623526, 'latitude'),
 (0.0552209145584391, 'rooms_per_hhold'),
 (0.05282234267392159, 'bedrooms_per_room'),
 (0.04326886334676563, 'housing_median_age'),
 (0.015979824578558188, 'population'),
 (0.015565293220350891, 'total_bedrooms'),
 (0.01485301961875372, 'total_rooms'),
 (0.013395940496911729, 'households'),
 (0.006885264757911986, '<1H OCEAN'),
 (0.0031441131537800287, 'NEAR OCEAN'),
 (0.002103771874059563, 'NEAR BAY'),
 (0.00016664546091362418, 'ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one ocean_proximity category is really useful, so you could try dropping the others).

Evaluate Your System on the Test Set

There is nothing special about this process; just get the predictors and the labels from your test set, run your `full_pipeline` to transform the data (call `transform()`, not `fit_transform()`!), and evaluate the final model on the test set:

In [78]:

```
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)
final_rmse
```

Out[78]:

```
47.99396907462751
```

The performance will usually be slightly worse than what you measured using crossvalidation if you did a lot of hyperparameter tuning (because your system ends up fine-tuned to perform well on the validation data, and will likely not perform as well on unknown datasets).

In [ ]:

```

```