Chapter 2: Data handling and Linear Regression models

Werner Bauer

Outline

Loading data files with Pandas

Pandas provides useful functions to import various file types:

- .csv (comma-separated values) file

```
data = pd.read_csv('titanic.csv')
# Output:
     PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0              1         0       3  ...   7.2500   NaN         S
1              2         1       1  ...  71.2833   C85         C
2              3         1       3  ...   7.9250   NaN         S
3              4         1       1  ...  53.1000  C123         S
4              5         0       3  ...   8.0500   NaN         S
```

- .txt files (without column names and separated by spaces):

```
data = pd.read_csv('file.txt', sep = "\s+", names = ['column1_name','column2_name']) # use '\s+'
       if spacing is not known, assign names to columns, ....)
```

- excel (.xlsx) files

```
banking = pd.read_excel('banking.xlsx', sheet_name='bank-additional-full')
# Output:
       age          job  marital  ... euribor3m nr_employed    y
0       56    housemaid  married  ...     4.857      5191.0   no
1       57     services  married  ...     4.857      5191.0   no
2       37     services  married  ...     4.857      5191.0   no
```

- Loading files from internet sources via url's

Example: loading the housing.txt file (used later)

We want to read in the following txt file:

```
0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30 396.90   4.98  24.00
0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80 396.90   9.14  21.60
0.02729   0.00   7.070  0  0.4690  7.1850  61.10  4.9671   2  242.0  17.80 392.83   4.03  34.70
0.03237   0.00   2.180  0  0.4580  6.9980  45.80  6.0622   3  222.0  18.70 394.63   2.94  33.40
   .       .      .    .     .       .      .      .      .   .        .    .        .     .
   .       .      .    .     .       .      .      .      .   .        .    .        .     .
```

We can use `.read_csv()` with options as shown next:

```
# as file has no header and values separated by whitespaces
df = pd.read_csv('housing.txt', delimiter= '\s+', header = None) # OR
df = pd.read_csv('housing.txt', delim_whitespace=True, header = None) # OR
df = pd.read_fwf('housing.txt', header = None) # fwf for 'fixed-width formatted'
# Output:
          0     1      2   3      4   ...     9     10      11    12    13
0    0.00632  18.0   2.31   0  0.538  ...  296.0  15.3  396.90  4.98  24.0
1    0.02731   0.0   7.07   0  0.469  ...  242.0  17.8  396.90  9.14  21.6

# To set names rather then indices for the COLUMNS:
df.columns = ['CRIM','ZN','INDUS','CHAS','NOX','RM','AGE','DIS','RAD','TAX','PTRATIO','B','LSTAT', '
      MEDV'] # OR
df = pd.read_csv('housing.txt', delim_whitespace=True, header = None, names = ['CRIM',..., 'MEDV'])
# Output:
        CRIM    ZN  INDUS  CHAS    NOX  ...    TAX  PTRATIO       B  LSTAT  MEDV
0    0.00632  18.0   2.31     0  0.538  ...  296.0     15.3  396.90   4.98  24.0
1    0.02731   0.0   7.07     0  0.469  ...  242.0     17.8  396.90   9.14  21.6
```

Example (Hands-On ML book 46-47): loading (another) housing data

- We write functions to download and decompress the file:

```
import os
import tarfile
import urllib
import pandas as pd

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

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

fetch_housing_data() # fetch the data
housing = load_housing_data()
```

```
housing.head() # Outputs top 5 lines:
   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
```

Setting and modifying the column types (e.g. boolean, category)

Types are not aways correctly recognised:

```
print(titanic) # Output:
     PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0              1         0       3  ...   7.2500   NaN         S
1              2         1       1  ...  71.2833   C85         C
2              3         1       3  ...   7.9250   NaN         S
print(titanic.dtypes) # Output:
PassengerId      int64 # OK
Survived         int64 # should be categorical
...

titanic = titanic.astype({'Survived' : 'category'}, copy=True) # DEFAULT: copy=True
print(titanic.dtypes) # Output:
PassengerId       int64 # OK
Survived       category # OK
```

```
print(housing)
       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
print(housing.dtypes)
longitude             float64 # OK
...
ocean_proximity        object # Could be any Python object, but repetitive entries suggest it is
       categorical!
housing['ocean_proximity'].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
# THEN: modify type as suggested above! Discussed later again!
```

Looking at the data file (housing)

- Use `.head()` to look at top 5 lines
- `.describe()` gives overview of each numerical attribute

```
          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
```

- count, mean, min, max are self-explanatory
- std stands for (standard deviation)
- 25%, 50%, 75% for percentile (e.g. 25% of the districts have a `median_income` lower than 2.563400)

Looking at the data file (housing)

- The `.info()` method gives quick description of data

```
housing = load_housing_data()
print(housing.info()) # Output:
<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
None
```

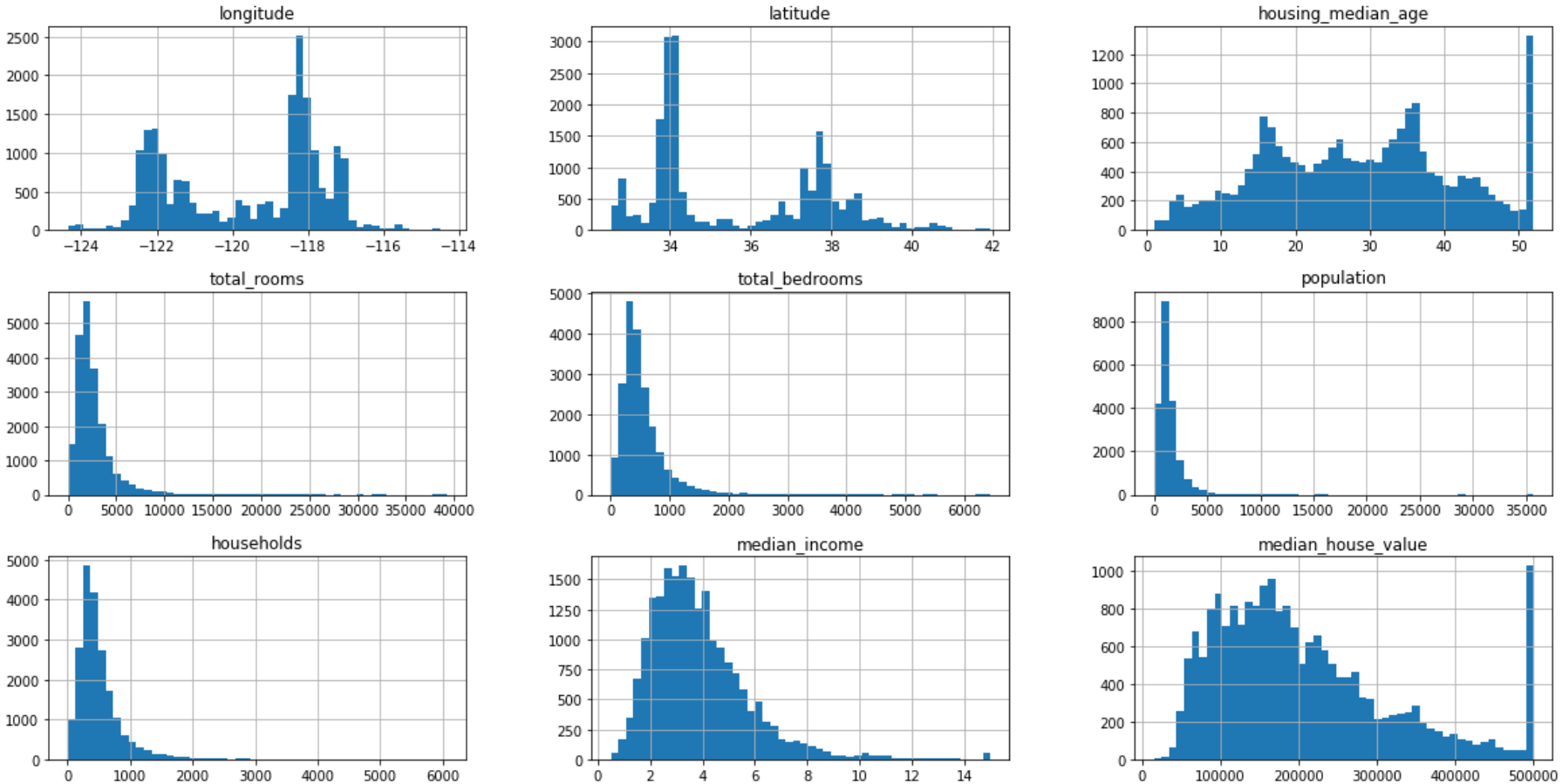
Gives total number of rows and columns, the attribute types and number of non-null values.

- Note: `total_bedrooms` is missing some values (we will need to address this)

Plotting the data

- Using a histogram plot to view numerical attributes:

```
1  import matplotlib.pyplot as plt
2  housing.hist(bins=50,figsize=(20,15))
```



Data set preparation: creating a test set

At this stage is is time to set aside part of the data. Why?

- Having test data is important to avoid overfitting of the ML model
- If your brain detects pattern when looking at the test data, you might realize pattern that influences your choice of ML model
- When you estimate the generalization error using the test set, your estimate will be too optimistic and it does not generalize this well to new data (data snooping bias)

Creating a test set (cf. Hands-On ML book on pages 51-55)

We can use a scikit-learn function to split datasets.

```
from sklearn.model_selection import train_test_split

# purely random sampling method:
train_set, test_set = train_test_split(housing,test_size=0.2,random_state=42)
```

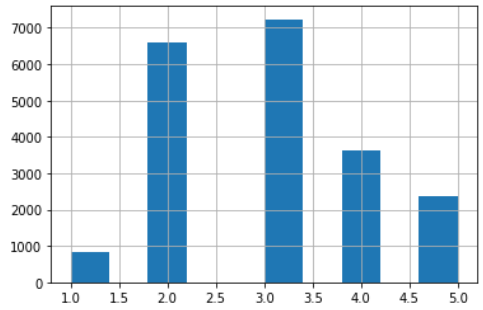
- Potentially with smaller datasets, random sampling introduces bias! Hence:
  we must make sure that the test set is representitive for the various categories. E.g. consider the `median_income` attribute:

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

from sklearn.model_selection import StratifiedShuffleSplit

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

# remove income_cut to bring data back to original state:
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```



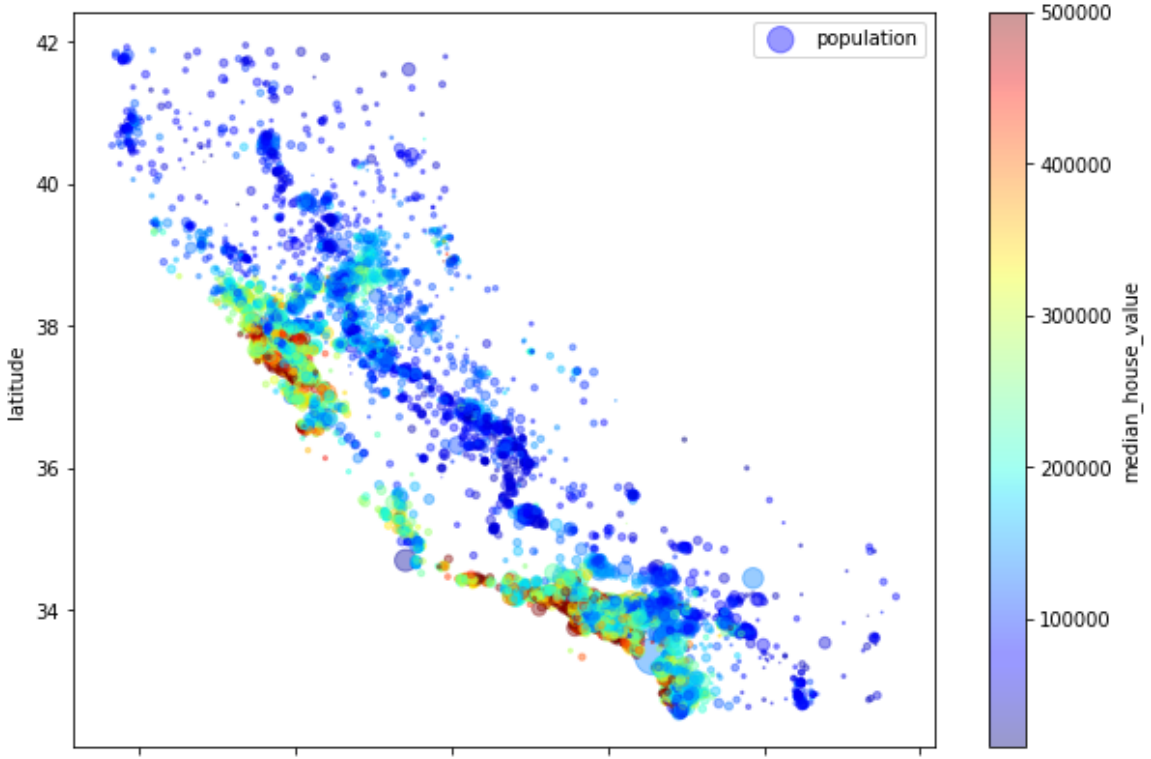

```
??_train_set['income_cat'].
value_counts()/len(??_train_set)
```

	Overall	Stratified	Random	Rand. %error	Strat. %error
1	0.039826	0.039729	0.040213	0.973236	-0.243309
2	0.318847	0.318798	0.324370	1.732260	-0.015195
3	0.350581	0.350533	0.358527	2.266446	-0.013820
4	0.176308	0.176357	0.167393	-5.056334	0.027480
5	0.114438	0.114583	0.109496	-4.318374	0.127011

Outline

Example: scatter plot of housing dataset

```
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)
plt.legend()
```



California housing prices: red is expensive, blue is cheap;
large circles for areas with large population

- Housing prices are related very much to location!

Looking for correlations: *standard correlation coefficient*

- If dataset is not too large: calculate standard correlation coefficient (Pearson's r):

```
corr_matrix = housing.corr()
print(corr_matrix) # Output:
                    longitude  latitude  ...  median_income  median_house_value
longitude            1.000000 -0.924478  ...      -0.019615           -0.047466
latitude            -0.924478  1.000000  ...      -0.075146           -0.142673
housing_median_age  -0.105823  0.005737  ...      -0.111315            0.114146
...
households           0.063146 -0.077765  ...       0.010869            0.064590
median_income       -0.019615 -0.075146  ...       1.000000            0.687151
median_house_value  -0.047466 -0.142673  ...       0.687151            1.000000
```

- select an attribute of interest and sort values:

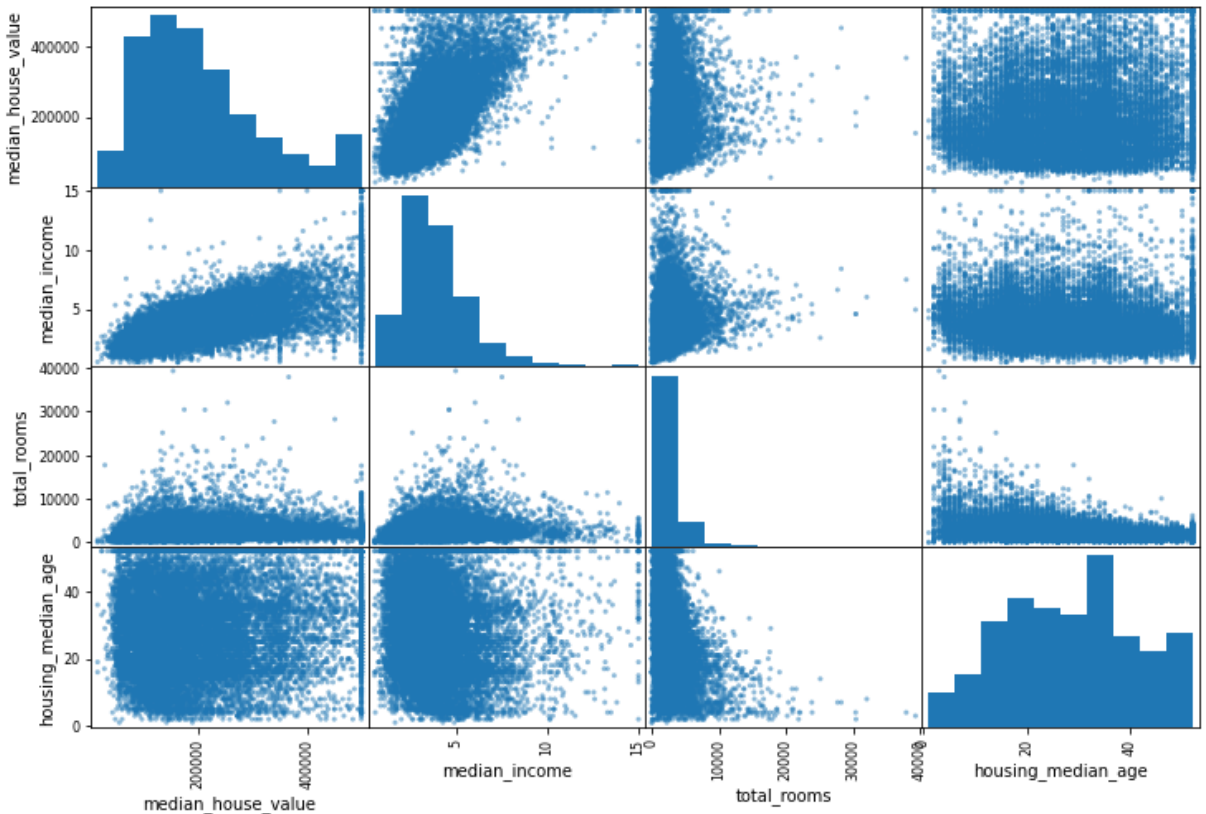
```
corr_matrix['median_house_value'].sort_values(ascending=False)
# Output:
median_house_value    1.000000
median_income         0.687151
total_rooms           0.135140
housing_median_age    0.114146
households            0.064590
total_bedrooms        0.047781
population           -0.026882
longitude            -0.047466
latitude             -0.142673
```

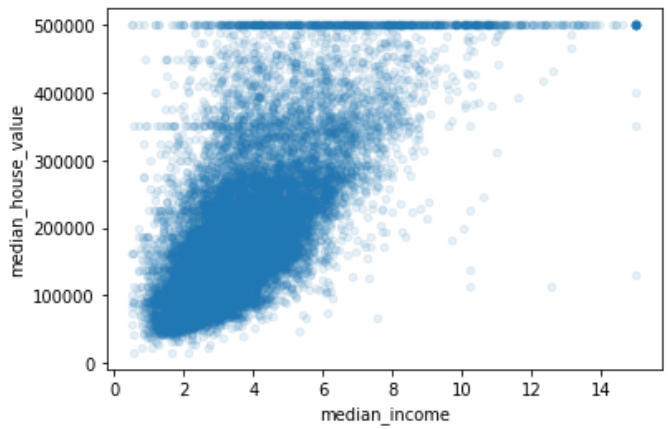
- Range [-1,1]: -1 negative, +1 positive, 0 no correlation
- works only for lin. correlation: see examples in Hands-On ML bookpage 59

Looking for correlations: pandas *scatter matrix*

- Plot attributes most correlated to `median_house_value`

```
1  from pandas.plotting import scatter_matrix
2
3  attributes = ["median_house_value", "median_income", "total_rooms", "housing_median_age"]
4  scatter_matrix(housing[attributes], figsize=(12, 8)) # LEFT
5
6  housing.plot(kind="scatter", x="median_income", y="median_house_value", alpha=0.1) # RIGHT
```





- Note: diagonal shows histograms rather than correlation

Testing out attribute combinations

- Try out some attribute combinations to increase correlation with the target attribute, here `median_house_value`.
- Example: total number of rooms in a district not very useful without knowing how many households there are!
- Following attribute combinations seem promising:

```
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"]

corr_matrix = housing.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
# Output:
median_house_value          1.000000
median_income               0.687151
rooms_per_household         0.146255
total_rooms                 0.135140
housing_median_age          0.114146
households                  0.064590
total_bedrooms              0.047781
population_per_household   -0.021991
population                 -0.026882
longitude                  -0.047466
latitude                   -0.142673
bedrooms_per_room          -0.259952
```

We found via `bedrooms_per_room`: houses with lower bedroom to room ratio are more expensive!

Outline

Outline

Outline