CAP4770 – Intro to Data Science

End-to-End Machine Learning Project – Part 1

Prof. Ye Xia

Setting up Your Environment for ML/DS

If you are experienced with Python, including installing packages and working with virtual environments, you do not need to follow these steps.

If you are fairly new to Python, here are some guidelines...

Install Python Distribution and Conda

- Your choices are: Anaconda, Miniconda, or Miniforge.
- Conda is the package and environment manager for working with machine learning tools in the Python (or R) environment. It is a command-line tool.

  As a package manager, you use Conda to install, update and delete packages such as Python libraries and Jupyter.

  As an environment manager, you use Conda to create and activate (virtual) environments and switch between them.
- Anaconda and Miniconda are developed and maintained by the company Anaconda, Inc., and by default they connect to the Anaconda repository (aka channel) maintained by Anaconda, Inc.
- Anaconda installs Python, conda and a large number of packages. It is a bit bloated, consuming 3GB of disk space.

- Miniconda is a minimal installer for Conda, which includes - Conda, Python, the packages they depend on (e.g., openssl, ncurses, sqlite), and a limited number of other useful packages (including pip, zlib, etc). Its small base size and selectivity makes it a preferred choice for when space is a concern.
- Miniforge is a community-developed counterpart to miniconda. By default, it connects to the conda-forge repository (channel), which is community-maintained. Miniforge works better for Mac M1 chip, especially for some neural-network-related packages, such as Tensorflow.
- **Recommendation:** You should install either Miniconda or Miniforge. If you have a Mac with Apple Silicon, you should install Miniforge.

Install Miniforge

Go to this page: https://github.com/conda-forge/miniforge/. Follow the instruction there.

Example: for Mac with Apple silicon

- In the Download section, download the shell script: `Miniforge3-MacOSX-arm64.sh`.
- From a terminal, run the following:

```
chmod +x ~/Downloads/Miniforge3-MacOSX-arm64.sh
sh ~/Downloads/Miniforge3-MacOSX-arm64.sh
```

  At the end of the execution of the above scripts, you are prompted to run 'conda init'. Say 'yes'.

  Restart the terminal. You should see the shell prompt becomes something like:

```
(base) xia@Ye-2022-MacBook-Pro ~ %
```

It shows that you are in the base environment of Python. At this point, you have installed Miniforge in the directory ∼/miniforge3/.

- Possible complications:
  1. If in the execution of the above scripts you skipped 'conda init' at the end and if you use the bash shell, run the following to add ∼/miniforge3/bin to PATH (in .bash_profile).

     ```
     conda init bash
     ```

  2. If you use the bash shell, the script execution might switch you to the zsh shell.

     If you get switched from the bash shell to zsh, you may need to run the following to add ∼/miniforge3/bin to PATH (in .zshrc).

     ```
     conda init zsh
     ```

  3. If you still don't see the base environment shown in the command

prompt, try restarting the terminal. If that doesn't work, try

```
conda activate
```

This activates the base environment.

- You can check other Conda commands by typing on the terminal:

```
conda
```

You can use Conda to install, uninstall or update packages.

Also, and **importantly**, you can use Conda to create and work with virtual environments.

Why Working with Python Environment

- There is a large number of Python libraries and other software packages used for machine learning. There are complex dependencies between them.
- A typical machine learning project will use many of the packages and libraries.
- If you update one package/library to a new version in a system-wide installation point (maybe for new project), the new version might not work with other packages and your current project will not work any more.
- Portability issue: If you move your project to another machine, the versions of the packages are most likely different, and that tends to break your project.
- **Solution:** Each substantial project should live in its own (isolated) Python virtual environment. The required packages for that project are installed in that environment.

Install Jupyter in the Base Environment (Optional)

- After we install Miniforge, we have the base environment. You don't need to install anything in the base environment, expect perhaps Jupyter.
  - Jupyter Notebook is a web-based interactive computational environment in the form of a notebook. You can run Python commands interactively from within the notebook.
  - There is also JupyterLab, which is newer and offers a more flexible user interface and more features.
  - We will mostly use Jupyter Notebook for our examples.

- **To install Jupyter in the base environment:** While in the base environment (with `(base)` shown in the prompt), run the following:

```
conda install jupyter
```

Then, run the following to add Conda support to jupyter notebook.

```
conda install nb_conda
```

Now, if you run `jupyter notebook`, you will see a web page opening up. But, we will not proceed from there. Click ‘Quit’ stop Jupyter server and close the page.

Creating a New Python Environment

- We next create a new environment for basic machine learning projects in this course. I will call it 'env-basic'. You can name it anything you want.

```
conda deactivate
conda create -n env-basic
conda activate env-basic
```

Remarks: In the above, we first deactivated the base environment, then created a new environment 'env-basic', and activated the new environment.

You should be able to see the environments created using `conda env list`

You can activate/deactivate an environment by `conda activate env-name` or

`conda deactivate env-name`.

- **Install basic ML libraries to the env-basic environment:**

  While in the env-basic environment (i.e., while it is activated), run

  ```
  conda install matplotlib numpy pandas scipy scikit-learn
  ```

  Remarks:

  1. Some people also add 'jupyter' to the above list to install Jupyter to this environment. That will be fine. Alternatively, since we installed Jupyter to the base environment, we don't have to install Jupyter to the new environment. Instead, while in the env-basic environment, we run the following to enable Conda support in Jupyter (is this needed again?)

     ```
     conda install nb_conda
     ```

  2. We have installed the basic machine learning libraries to the env-basic environment so that it will work for most ML projects

in this course.

3. We will let most of the course projects share this environment. Although we said each project should have its own environment, our projects are mostly very small and they can be considered a single project.

Using Jupyter Notebook

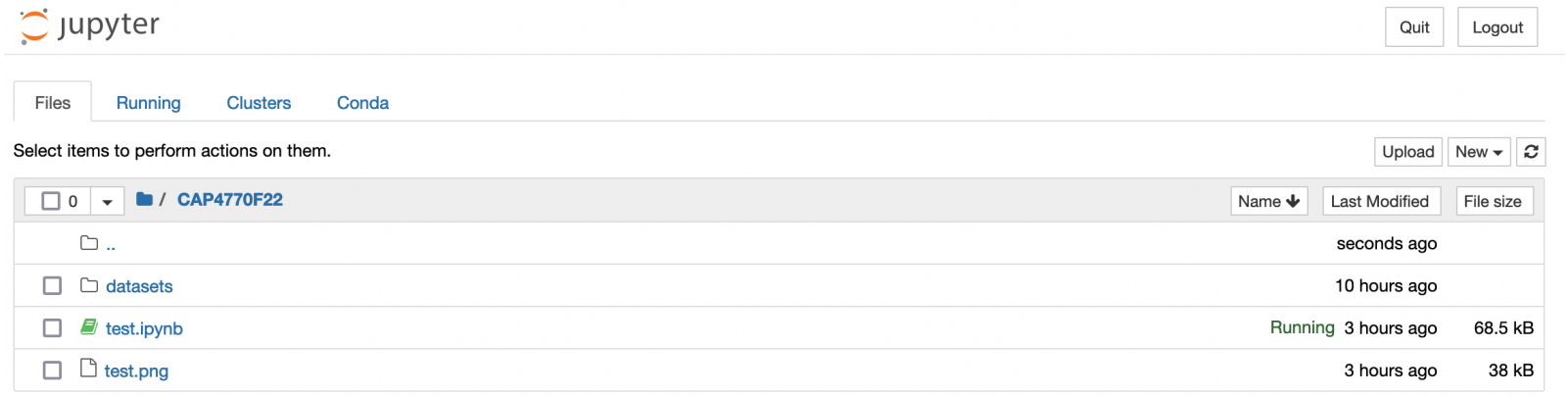
- Separately, you should create a directory for the projects in this course, such as ∼/CAP4770/. You can put various projects and datasets there.

  If you want to be more organized, you can create a sub-directory within ∼/CAP4770/ for each project, e.g., ∼/CAP4770/project1/.

- While the env-basic environment is activated, go to the directory ∼/CAP4770/, and start Jupyter Notebook from there:

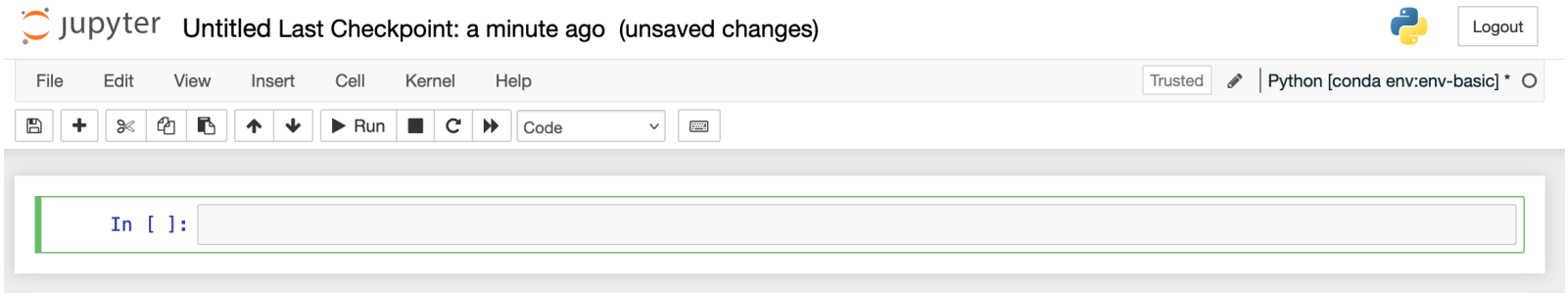
  ```
  jupyter notebook
  ```

  This brings up a web-page, something like

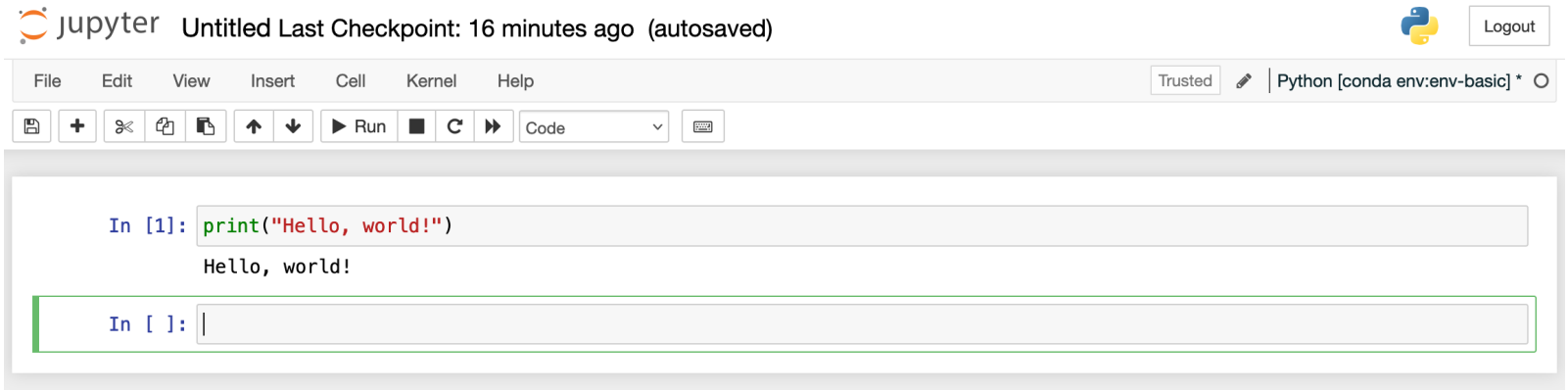


- You can create a new notebook by clicking 'New' on the page. Make sure you select the Python in your environment, e.g., `Python [conda env:env-basic]`.

  You will see something like the following:



- You will enter your Python code in the input cell marked by `In []:`.
- You evaluate an input cell by pressing 'Shift+Enter'. For instace, you will see something like the following:



- The new notebook will be auto-saved to the directory ~/CAP4770/. You can rename your notebook.

Basic Machine Learning Libraries/Packages

- NumPy: provides support for large, multi-dimensional arrays and matrices and a large collection of high-level mathematical functions to operate on these arrays.

```
import numpy as np
a = np.array([[1, 2, 3], [3, 4, 6.7], [5, 9.0, 5]])
print("Numpy array\n {}".format(a))
```

out:

```
Numpy array
 [[1.  2.  3. ]
 [3.  4.  6.7]
 [5.  9.  5. ]]
```

```
from numpy.linalg import solve, inv
inv(a)
```

out:

```
array([[-2.27683616,  0.96045198,  0.07909605],
       [ 1.04519774, -0.56497175,  0.1299435 ],
       [ 0.39548023,  0.05649718, -0.11299435]])
```

```
b =  np.array([3, 2, 1])
solve(a, b)  # solve the equation ax = b
```

out:

```
array([-4.83050847,  2.13559322,  1.18644068])
```

- SciPy: provides algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations, statistics and many other classes of problems.

SciPy also contains a linalg module. There is overlap in the functionality provided by the SciPy.linalg and NumPy.linalg modules.

- matplotlib: a comprehensive library for creating static, animated, and interactive visualizations in Python. Basically, it is a scientific plotting library for visualizing data.

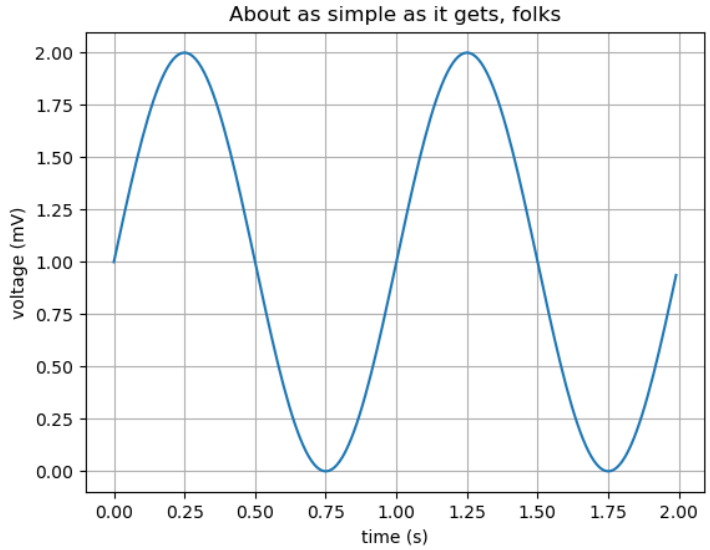
```
import matplotlib.pyplot as plt

# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)

ax.set(xlabel='time (s)', ylabel='voltage (mV)',
       title='About as simple as it gets, folks')
ax.grid()
```

```
fig.savefig("test.png")
plt.show()
```



- pandas: a Python library for data manipulation and analysis. It provides a key data structure: DataFrame, which can be thought as an Excel table. A DataFrame object is a multi-dimensional array in which the columns can be of different data types. It can ingest data from various data files and database such as SQL, Excel, CSV etc.

- scikit-learn: a machine learning library for Python. It features various classification, regression and clustering algorithms

Conda v.s. Pip

- It is possible that some Python libraries are not in the regular conda repositories, such as the conda-forge channel or the Conda channel maintained by Anaconda, Inc.
- Pip is another package manager for Python libraries (Python libraries only).
- The default library repository for pip is PyPI (pie-pea-eye). Packages on PyPI are typically uploaded by the authors of the packages.
- A version of pip is installed when you install Miniforge or Minconda.
- To use this version to install some package using pip (while in an environment):

```
pip install some-package-name
```

Overview of an End-to-End Project

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Working with Real Data

- When you are learning about Machine Learning, it is best to experiment with real- world data, not artificial datasets.
- There are thousands of open datasets to choose from, ranging across all sorts of domains.
- Popular open data repositories
  - UC Irvine Machine Learning Repository: http://archive.ics.uci.edu/ml/index.php
  - Kaggle datasets: https://www.kaggle.com/datasets
  - Amazon's AWS datasets: https://registry.opendata.aws/
- Meta portals (they list open data repositories)
  - Data Portals: http://dataportals.org/
  - OpenDataMonitor: https://opendatamonitor.eu

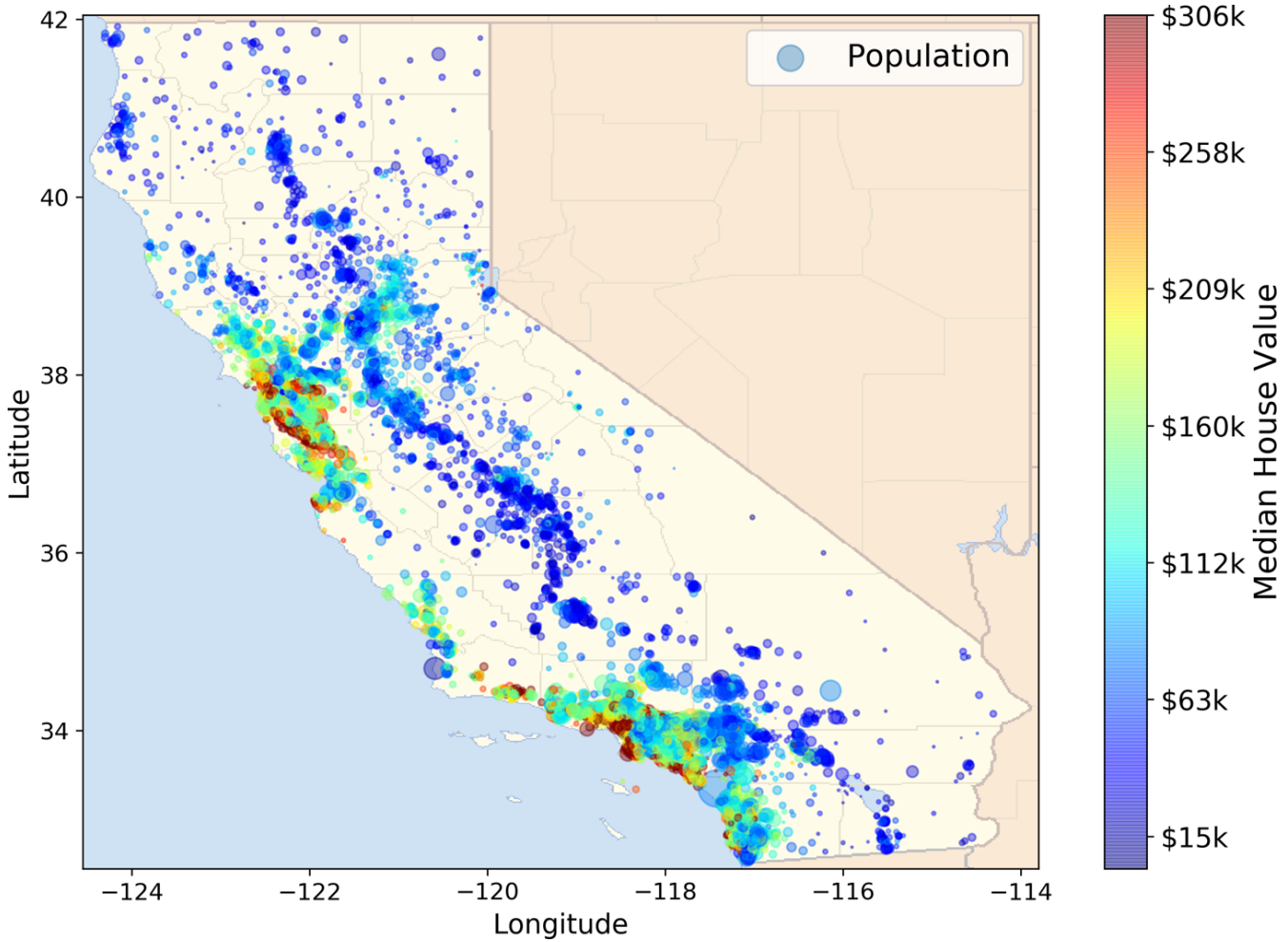
  - Nasdaq Data Link: https://data.nasdaq.com/

- Other pages listing many popular open data repositories
  - Wikipedia's list of Machine Learning datasets: Search 'List of datasets for machine-learning research wikipedia'
  - Curated dataset lists, e.g., Best Public Datasets for Machine Learning and Data Science by Stacy Stanford, Roberto Iriondo, Pratik Shukla (also see its references)

- Search on:
  - Google dataset search engine: https://datasetsearch.research.google.com/
  - Quora.com: Search 'Where can I find large datasets open to the public'
  - The datasets subreddit: https://www.reddit.com/r/datasets/

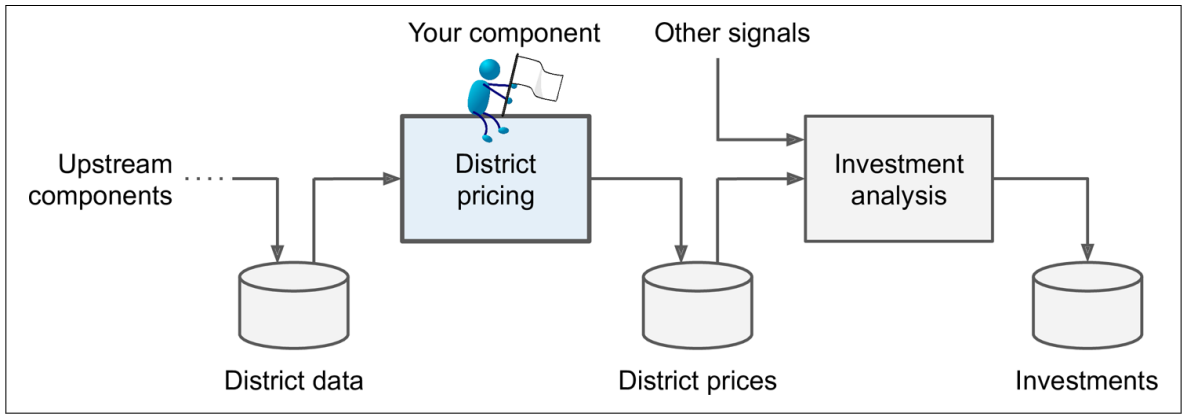
Data for This Project

- California Housing Prices dataset from the StatLib repository
- This dataset is based on data from the 1990 California census.
- The data includes metrics such as the population, median income, and median housing price 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 call them “districts” for short.


Population
Latitude
Longitude
42
40
38
36
34
−124
−122
−120
−118
−116
−114
$306k
$258k
$209k
$160k
$112k
$63k
$15k
Median House Value

The Big Picture

- Goal: Learn from this data so that the system can predict the median housing price in any district based on all the other (input) attributes.
- Know the (business) objective: How does the company expect to use and benefit from this model?
- For instance: Your model's output will be fed to another Machine Learning system, along with many other signals. This downstream system will determine whether it is worth investing in a given area or not.



- Think about the alternatives and compare. Maybe there are up-to-date surveys of the median housing price for some districts. Maybe there are other datasets based on which you can train a model that predicts the prices.

Get the Data

- For a general project, you may have to consider the kind of data you need for your project, and where and how to get the data.

  For instance, if you want to predict stock trend based on broad sentiment, you may crawl many web sites and get the text data where people write or discuss about the market.

- Sometimes, you stumble upon some dataset, and you want to figure out what you can do with it.
- In our case, we are given the dataset. It can be dowloaded from:
  https://github.com/ageron/handson-ml2/blob/master/datasets/housing/housing.tgz
- Load data using pandas and examine the data:

```
import pandas as pd
housing=pd.read_csv("datasets/housing/housing.csv")
```

Getting to Know the Data a Bit - But Not Too Much Yet

- **Warning:** Before you split the data into training/test sets, you shouldn't know too much about it (see later).

  After the split, you can look at your training data extensively.

- `housing` is a DataFrame, like a spreadsheet: "Two-dimensional, size-mutable, potentially heterogeneous tabular data". Pandas provides a great many methods to work with DataFrame.

- `housing.head()` gives the first five rows.

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

There are 10 input attributes/features/fields.

- The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of nonnull values.

```
In [37]: 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
```

Notice: the number of rows and columns, the missing data for `total_bedrooms` and the data types of each attribute.

- All the input features are numerical, except `ocean_proximity`. Its type is object, so it could hold any kind of Python object. Here, it is text and it is a categorical feature. You can find out more using the `value_counts()` method.

```
In [38]: housing["ocean_proximity"].value_counts()
Out[38]: <1H OCEAN     9136
         INLAND        6551
         NEAR OCEAN    2658
         NEAR BAY      2290
         ISLAND           5
         Name: ocean_proximity, dtype: int64
```

- The `describe()` method shows a summary of statistics for all the inputs.

```
In [39]: housing.describe()
```

Out[39]:

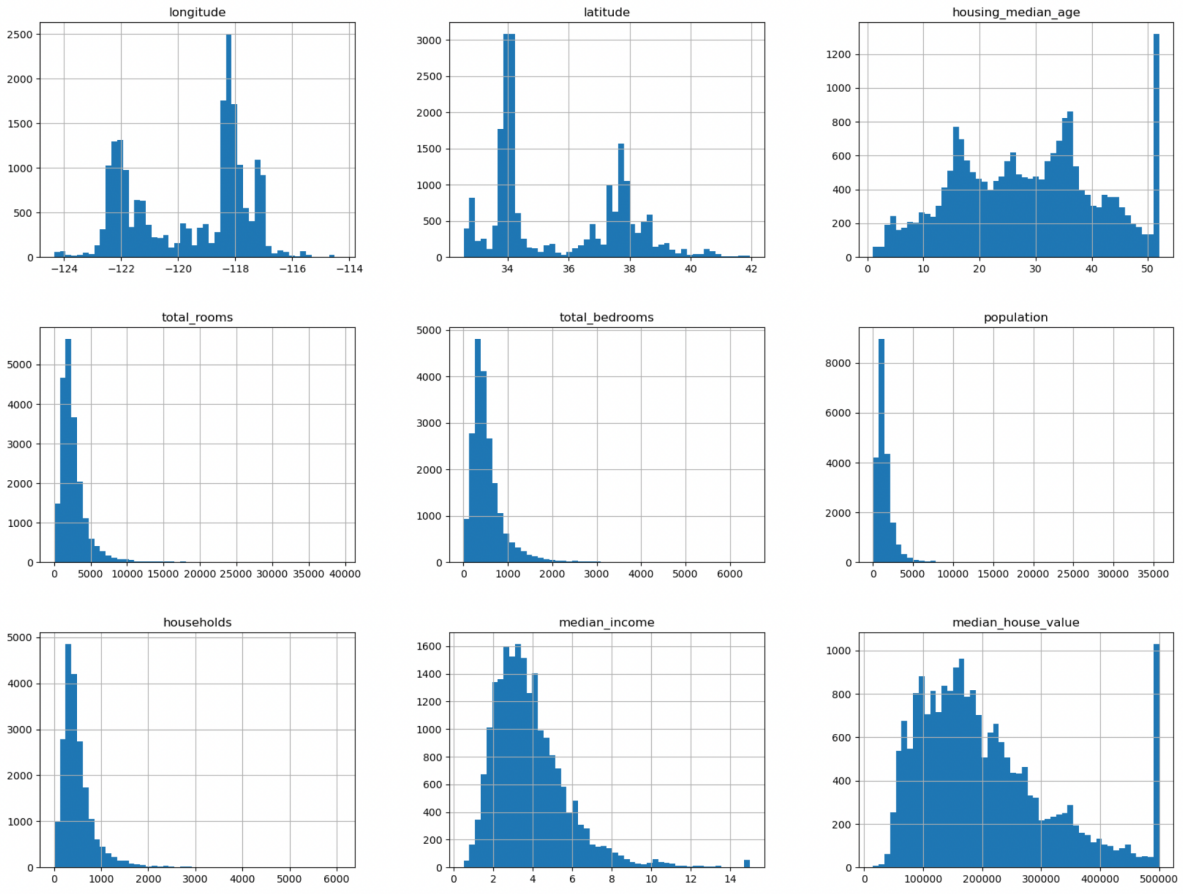
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000

- Histogram:

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

First line: No need to import if it has already been imported.

`plt.show()` is optional.



- These input attributes have very different scales. We discuss feature scaling.
- Some data has been pre-processed: scaled and capped.

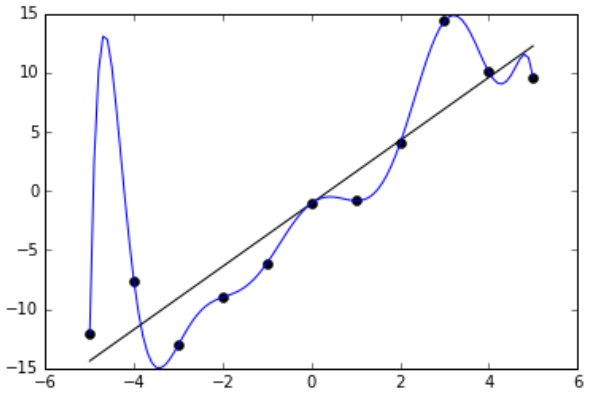
From outside knowledge: `median_income` has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. `housing_median_age` and `median_house_value` were also capped.

Capping of the data may affect your learning outcome. You may want to remove those capped data.

– Many histograms are long-tailed (skewed): they extend much farther to the right of the median than to the left. This may make it harder for some machine learning algorithms to detect patterns. One can try to transform these attributes to have more bell-shaped distributions.

Create a Test Set – Statistical Issues

- Problem of **overfitting:** Your model may be so powerful that it fits a given training dataset too well, even capturing the noise part, but fails to generalize to new instances.



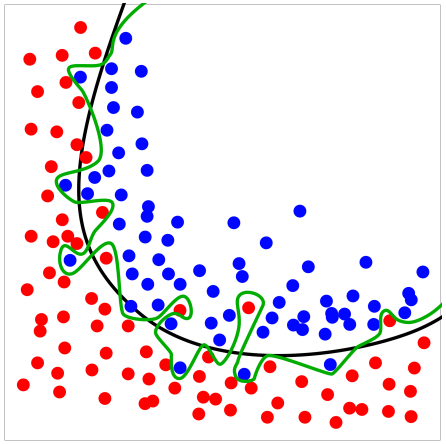


Figure 1: Left: The polynomial function fits the data perfectly; a likely overfit. Right: a classification example (From wikipedia).

- Solution: Split the given data into training/test sets. Set aside a part

of the data as the **test set**, typically 20% of the dataset. The rest forms the training set. You will later evaluate your algorithm's ability to generalize using the test set.

- Principles: You shouldn't know the data in your test set. Knowing too much may let you see some pattern in the test data, which may affect your model selection and lead to overfitting. This is known as **data snooping bias**. This implies that the instances used in the training set should not show up in the test set in future runs of the algorithm.

Test Set Creation Method 1 – Random Selection

- Randomly select 20% of the data based on their indices as the test set.

  Problem: If you run your program again, you will see a different test set. Overtime, you will see the whole dataset.

In [107]:
```
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]

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

In [108]:
```
len(train_set)
```

Out[108]: 16512

Test Set Creation Method 2 – Seeded Random Selection

Problem: If run the program again, you will have a different test set, which may include some data previously in a training set. Overtime, you will see the whole dataset. Not good!

Solution: Fix the random number generator seed. Then, use method 1.

- Add `np.random.seed(42)` to the beginning of the function `split_train_test()`.
- Problem: This method will also break if your dataset is updated: E.g., some instances are added or deleted, or the order of the instances is changed.

  You will have a different test set, which may include some data previously in a training set.

Test Set Creation Method 3 – Hash-Based Selection

- This is the preferred method.
- Use a (uniform) hash function. Compute the hash value of each instance's **persistent** identifier. Put an instance into the test set if its hash value is less than 20% of the maximum possible hash value. This way, even if the dataset is updated, the instances previously used in the training set will not show up in the new test set.
- A good uniform hash function maps the instances' identifiers to numbers on $[0, \max]$ uniformly, where max is a constant.

```
from zlib import crc32
def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32
```

- If the data instances do not have identifiers, you can use the row index

as the ID. But, you need to make sure that, in a data update, the new data will only be appended at the end, and no rows will be deleted. The following adds a column called "index", which contains the row indices.

```
housing_with_id = housing.reset_index()
```

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

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

`id.appy()`: applies a function to each entry of `id`. After applying the function, the complete result is a boolean array.

`pandas.DataFrame.loc`: accesses a group of rows and columns by label(s) or a boolean array. The label can be the row or column label or both.

- If the aforementioned updating rule cannot be ensured, you can use

the stable and relatively unique attribute(s) to build an identifier. In the housing data, those may be the longitude and latitude.

The following adds another column called “id”.

```
housing_with_id["id"] = housing["longitude"] * 1000 \
                        + housing["latitude"]
```

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

Unfortunately, the location information is quite coarse. The longitude and latitude together are still not distinct enough. There are collisions in the resulting id.

```
In [68]: housing_with_id["id"].value_counts()
Out[68]: -122372.20    15
         -122382.20    11
         -122402.22    11
         -122382.25    10
         -122402.20    10
                       ..
         -122311.70     1
         -122291.79     1
         -122361.66     1
         -122291.62     1
         -121200.63     1
         Name: id, Length: 12590, dtype: int64
```

You see that there are 15 instances with the same id -122372.20, and there are 12590 unique ids for 20640 rows.

This will introduce some sampling bias.

You could add more attributes for creating the id. For instance, the values of `median_income` seem quite distinct for different rows. If you do:

```
housing_with_id["id"] = housing["median_income"] * 10000000 \
                        + housing["longitude"] * 1000 \
                        + housing["latitude"]
```

You will get 20635 unique ids, which is much better. However, you need to make sure that in your future data update, the values of the median incomes will not change. This will work fine if future data updates will only delete or add rows.

Built-in Methods in Scikit-Learn

- There are different methods in Scikit-Learn.
- The simplest function is `train_test_split()`, which is similar to the function `split_train_test()` in method 1.
- You can set the random generator seed by setting the `random_state` parameter. This gives you method 2.

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

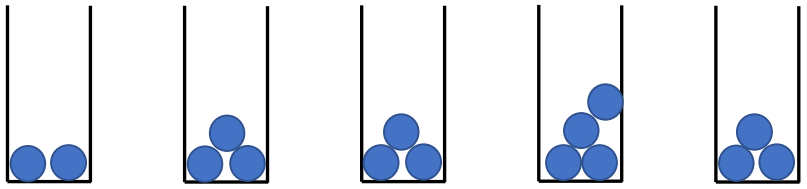
- You can pass it multiple datasets with an identical number of rows, and it will split them on the same indices. This is very useful if you have a separate DataFrame for labels (so that the inputs and outputs are split in the same way).

Stratified Sampling - Motivation

- We have assumed that the random generation and/or the hash function lead to uniformly random sampling of the dataset. In practice, good random generation methods and good hash functions come close to giving uniform sampling.
- If the dataset is large (thus, so is the test set), uniform sampling will work fine (because of the law of large numbers).
- When the dataset is small, even perfectly uniform sampling may not be enough because the variance is **relatively** large (compared to the mean-square of some quantity of interest).
- It is the same situation as in empirical average: When you do averaging over 1 million samples, the average will be close to the true mean, but when average with 5 samples, you will not likely get a good estimate of the mean.

- We can consider an abstract model. Imagine you put n balls in $k = 5$ bins uniformly at random (load-balancing). When n is very large, each bin should have $n/k + o(n)$ balls due to the law of large numbers, and therefore, each bin has a fraction very close to $1/k$ balls.
  - Take any bin. Suppose the random quantity of interest is the fraction of the balls falling into the bin. The true mean of the fraction is $1/k$. The deviation is bounded by $o(n)/n$. For large n, the deviation is small relative to the true mean because as $n \to \infty$, $o(n)/n \to 0$.

  When n is small, say $n = 15$, you will often see that some bin has more or less than n/k balls, like the following.



  After dividing by n, the fractions are: $\frac{2}{15}, \frac{3}{15}, \frac{3}{15}, \frac{4}{15}, \frac{3}{15}$. The

deviation from the mean fraction $3/15$ is not negligible for some bins.

- Voting example:

  Likely voters from different demographics vote differently. A small demographic group who strongly favors one candidate may sway an election.

  For election polling, the sample size is often small, e.g., 100 – 1000. The polling sample needs to reflect the true proportions of different demographic groups accurately (in addition to drawing enough instances from each group).

  To see why, suppose there are 5 demographic groups and suppose each group is 20 % of the total population. Suppose group 1 strongly favors candidate A versus B, group 2 very slightly prefers candidate B, and the other groups are 50-50 on either candidate. A uniform sampling of 100 likely voters may only pick up 16 voters from group 1. The election prediction based on the sample may be wrong.

  In reality, the polling agencies don't just randomly choose people from the phone book (this is uniformly random sampling).

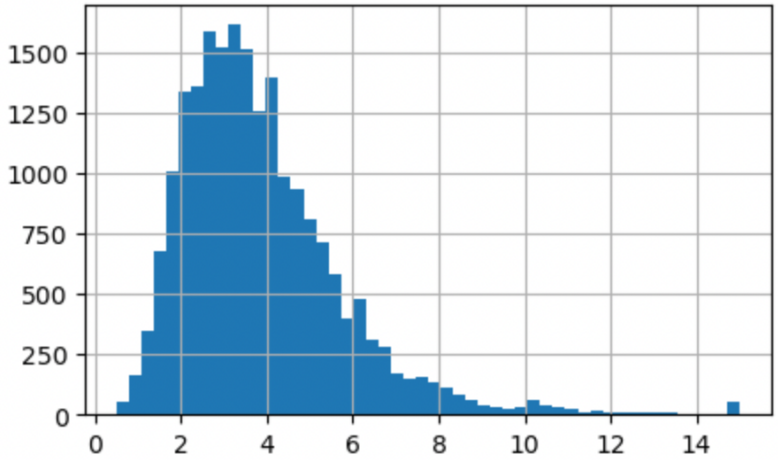
Solution: Stratified Sampling

- First, determine the number of instances drawn each stratum (i.e., a sub-popoulation), which should be in accordance with its proportion in the whole population. Then, draw instance randomly from each stratum until the pre-determined number is reached.
- Suppose you learned from experts that the median income is a very important attribute to predict median housing prices.You may want to ensure that the test set is representative of the various categories of median incomes in the whole dataset.
- Since the median income is a continuous numerical attribute, you first need to bin it, i.e., to create median income categories and make them an attribute. You can rely on the histogram to help.

  Principle: It is important to eventually have a sufficient number of instances for each stratum (i.e., category) in your test set, or else the

estimate of a stratum's importance may be biased (due to the relatively large fluctuation in a small sample size).

This in turn means that you should not have too many strata and each stratum should have a substantial weight (in terms of the fraction of the whole population).

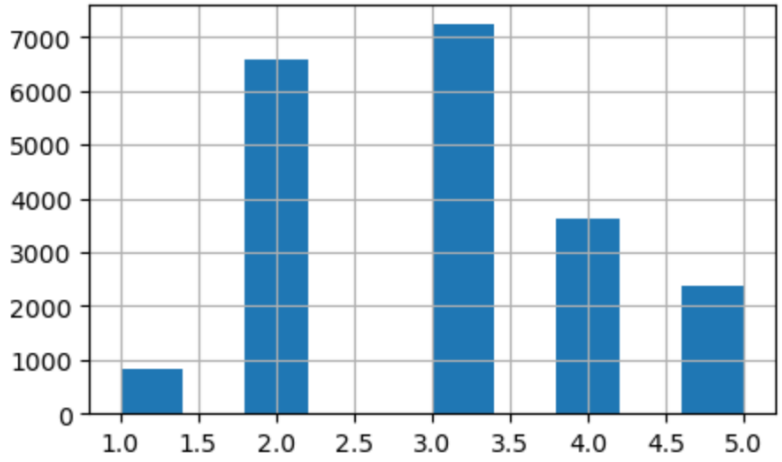


We see that most of the median income values fall in the range of 1.5 to 6, which can be divided into a few strata. Then, there is one stratum below 1.5 and one above 6.

The following code creates a new attribute called “income_cat”, the median income categories. There are five categories, as shown in the histogram that follows.

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

```
housing["income_cat"].hist(figsize=(5,3))
```



Stratified Sampling with Scikit-Learn

- Stratified sampling using Scikit-Learn's `StratifiedShuffleSplit` class.

```
from sklearn.model_selection import StratifiedShuffleSplit

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

`StratifiedShuffleSplit` is usually used in cross-validation, which needs more than one splits (into training/test sets)

Here, we only split into training/test sets once. The for-loop runs once.

The stratified sampling is based on the second argument of `sss.split()`. The first argument `housing` is not used in the

split call; it is there for compatibility.

The proportions of different strata in the test set look like the following:

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

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

The following is a comparison between the stratified sampling and the non-stratified random sampling using `train_test_split`.

bins	population	stratified	random
3	0.350581	0.350533	0.358527
2	0.318847	0.318798	0.324370
4	0.176308	0.176357	0.167393
5	0.114438	0.114341	0.109496
1	0.039826	0.039971	0.040213

Note: The discrepancy between the population-level proportions and those from stratified sampling is due to rounding.

Related – Simpson's Paradox

- Simpson's Paradox is the phenomenon that a trend seen at one or more sub-populations disappears or is reversed when the sub-populations are combined.
- Example of Berkeley graduate school admissions in 1973: Data from overall population showed bias against women.

	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
Total	12,763	41%	8,442	44%	4,321	35%

Data from the six largest departments showed no obvious bias.

Department	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
A	933	64%	**825**	62%	108	82%
B	585	63%	**560**	63%	25	68%
C	918	35%	325	37%	**593**	34%
D	792	34%	417	33%	375	35%
E	584	25%	191	28%	**393**	24%
F	714	6%	373	6%	341	7%
Total	**4526**	**39%**	**2691**	**45%**	**1835**	**30%**

Reason: “The proportion of women applicants tends to be high in departments that are hard to get into and low in those that are easy to get into.”

Bickel, PJ, Hammel, EA, O’Connell, JW: Sex Bias in Graduate Admissions: Data From Berkeley. *Science.* 187(4175), 398-404 (1975).

Simpson's Paradox - COVID-19 Vaccine Example

Jeffery Morris, "Israeli data: How can efficacy vs. severe disease be strong when 60% of hospitalized are vaccinated?"
https://www.covid-datascience.com/post/israeli-data-how-can-efficacy-vs-severe-disease-be-strong-when-60-of-hospitalized-are-vaccinated

- Israel hospital data as of Israel in Aug. 15, 2021: Out of 515 patients currently hospitalized with severe cases in Israel, 301 (58.4%) of these cases were fully vaccinated (with two doses of the Pfizer vaccine).

Age	Population (%)		Severe cases		Efficacy
	Not Vax %	Fully Vax %	Not Vax	Fully Vax	vs. severe disease
All ages			214	301	**Vax don't work!**

But, there are more vaccinated people than unvacinnated people at that time. Need to normalize against the sizes of the vaccinated and

unvaccinated sub-populations.

Age	Population (%)		Severe cases		Efficacy
	Not Vax %	Fully Vax %	Not Vax per 100k	Fully Vax per 100k	vs. severe disease
All ages	1,302,912 18.2%	5,634,634 78.7%	214 16.4	301 5.3	**67.5%**

Attack rate is defined as

$$AR = \frac{\text{number of new severe cases in the population at risk}}{\text{number of people in the population at risk}}.$$

$$ARU = \text{Attack rate of unvaccinated people}$$

$$ARV = \text{Attack rate of vaccinated people}$$

From the data in the table:

$$ARU = \frac{214}{1,302,912} = \frac{16.425}{100,000} \approx \frac{16.4}{100,000}$$

$$ARV = \frac{301}{5,634,634} = \frac{5.342}{100,000} \approx \frac{5.3}{100,000}$$

We see that ARU is much higher than ARV. Vaccine works!

Vaccine Efficacy (VE) (here, against severe cases) is the normalized reduction in attack rate (AR) from vaccination.

$$VE = \frac{ARU - ARV}{ARU}$$

$$VE = \frac{16.425 - 5.342}{16.425} \approx 67.5\%$$

- That's not even the main story. We further stratify by age:

Age	Population (%)		Severe cases/100k		Severe Case Risk	Efficacy
	% Not Vax	% Fully Vax	Not Vax	Fully Vax	Ratio w/ 30-39 UnVax	vs. severe disease
12-15	62.1%	29.9%	0.30	0.00	1/20x	**100%**
16-19	21.9%	73.5%	1.60	0.00	1/4x	**100%**
20-29	20.5%	76.2%	1.50	0.00	1/4x	**100%**
30-39	16.2%	80.9%	6.20	0.20	1	**96.8%**
40-49	13.2%	84.4%	16.50	1.00	2.7x	**93.9%**
50-59	10.0%	88.0%	40.20	2.90	6.5x	**92.8%**
60-69	8.8%	89.8%	76.60	8.70	12.4x	**88.7%**
70-79	4.2%	94.6%	190.10	19.80	30.7x	**89.6%**
80-89	5.6%	92.6%	252.30	47.90	40.7x	**81.1%**
90+	6.1%	90.5%	510.9	38.60	82.4x	**92.4%**

We see that VE in every vaccine-eligible age group is higher than VE of all age groups together (which is counter-intuitive).

The vaccine is highly effective!

Not separating the population into age groups gives the misleading result (that the vaccine is only moderately effectively).

Simpson's Paradox - Separate Learning May be Needed

- Sometimes even stratified sampling is not enough. One has to do regression on each sub-population separately.

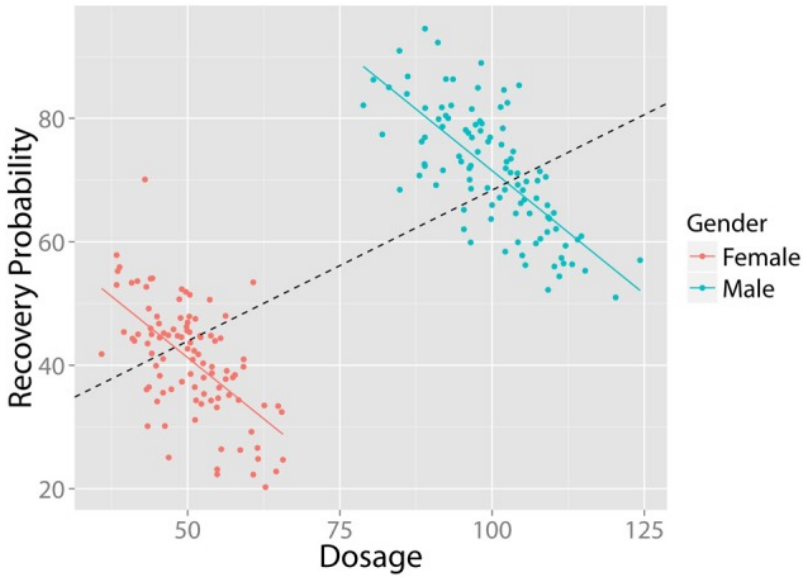


Figure 2: If the male and female sub-populations are not separated, one concludes (wrongly) that a higher drug dosage leads to a higher chance of recovery.

Kievit RA, Frankenhuis WE, Waldorp LJ, Borsboom D. Simpson's paradox in psychological science: a practical guide. *Frontiers in Psychology*. Aug., 2013; 4:513. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3740239/

Next: Discover and Visualize the Data to Gain Insights

- Now remove the `income_cat` attribute so the data is back to its original state.

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

`inplace=True`: Actually drop the column from the given DataFrame. Otherwise, the `drop` function returns a copy of the DataFrame with the column dropped.

- Put aside the test set. Only explore the training set.
- Make a copy of the training set and work with the copy.

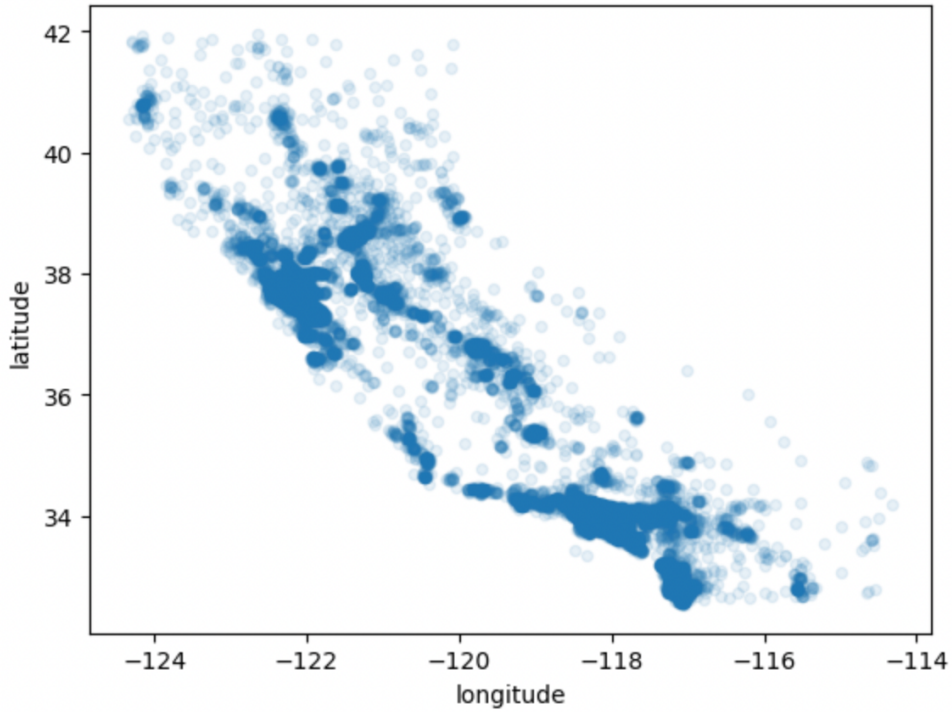
```
housing = strat_train_set.copy()
```

Visualizing Geographical Data

The following shows where the districts are located: more districts at population centers.

```
housing.plot(kind="scatter", x="longitude", \
        y="latitude", alpha=0.1)
```

`alpha` controls the amount of transparency.

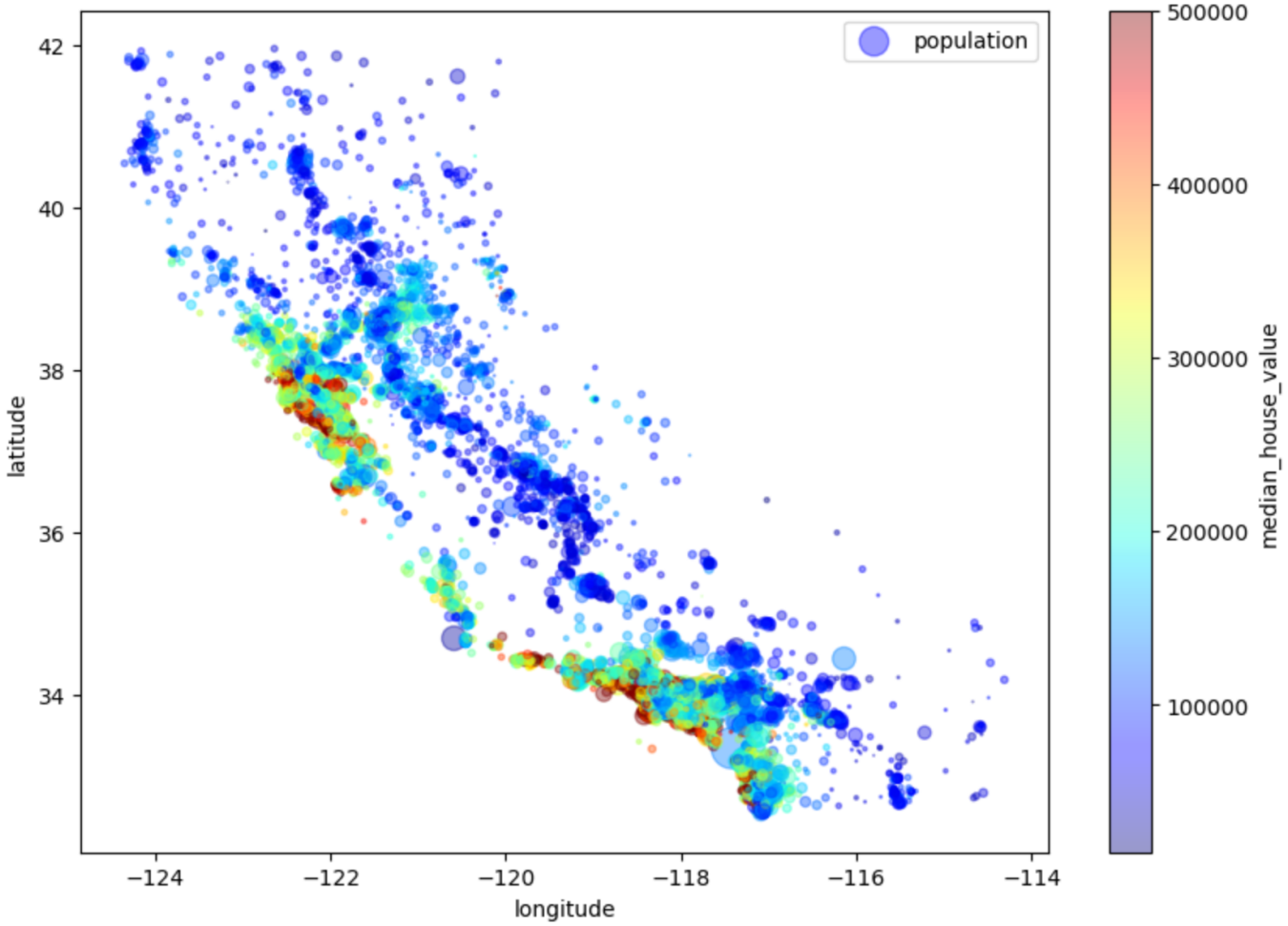


```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
        s=housing["population"]/100, label="population", figsize=(10,7),
        c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
    )
plt.legend()
```

s-option: circle size corresponds to population

c-option: color corresponds to median house value.

The housing prices are very much related to the location (e.g., close to the ocean) and to the population density.


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

Correlation

- Suppose X and Y are two random variables with joint distribution $P(X, Y)$. The **Pearson correlation coefficient** of X and Y, denoted by ρ_{XY}, is defined as

$$\rho_{XY} = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y}.$$

  $\text{cov}(X, Y)$: the covariance of X and Y defined by

$$\text{cov}(X, Y) = E[(X - EX)(Y - EY)]$$

  σ_X, σ_Y: standard deviations of X and Y.

$$\sigma_X^2 = E[(X - EX)^2] \qquad \sigma_Y^2 = E[(Y - EY)^2]$$

- ρ_{XY} measures how the fluctuations of X and Y are related.
- The product $\sigma_X \sigma_Y$ is for normalization so that $-1 \leq \rho_{XY} \leq 1$.

- $\rho_{XY} = 1$ iff $X = cY$ for some $c > 0$; $\rho_{XY} = -1$ iff $X = cY$ for some $c < 0$.
- If $\rho_{XY} = 0$, we say X and Y are uncorrelated.
- Empirical version: For a given data set $\{(x_1, y_1), \ldots, (x_N, y_N)\}$, where each x_i and y_i is a scalar, **Pearson correlation coefficient** (aka Pearson's r), denoted by r_{xy}, is defined as

$$r_{xy} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}},$$

where $\bar{x}$ and $\bar{y}$ are the sample means:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad\qquad \bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i.$$

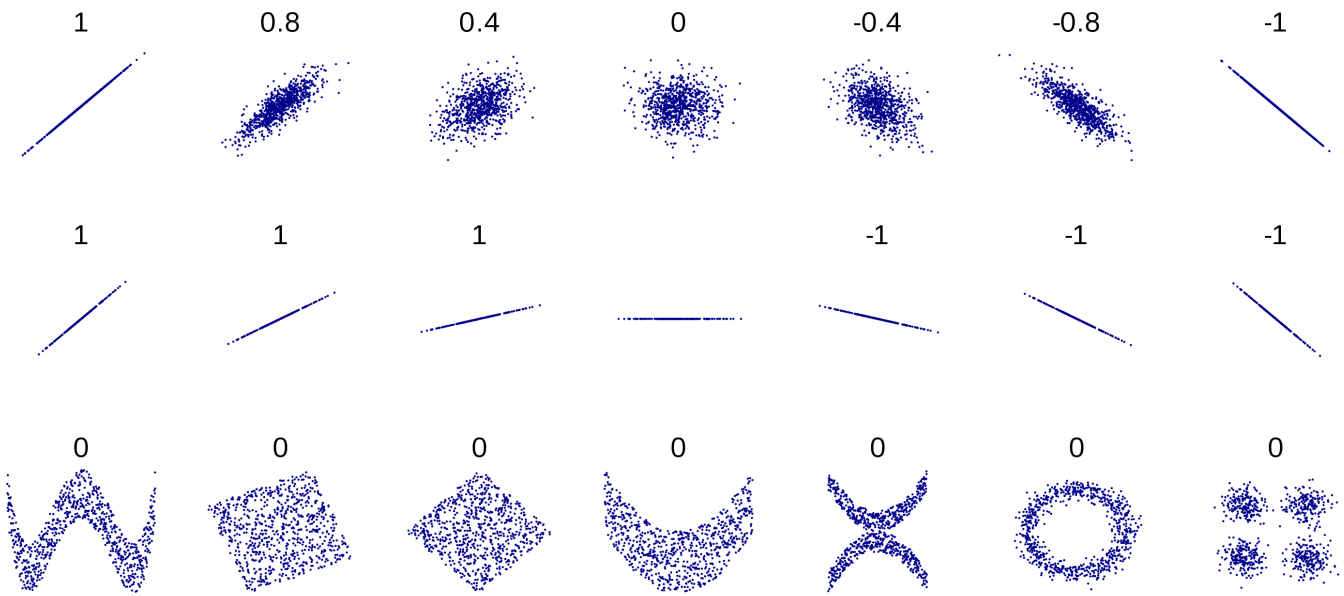


Figure 3: Some examples of Pearson's r.

Something closer to a linear line tends to have a large $|r_{xy}|$.

The magnitude of the slope of the line doesn't matter much; the sign matters.

$r_{xy} = 0$ doesn't mean x and y has no relationship (see the bottom row).

Correlations in the Housing Data

```
In [150]: corr_matrix = housing.corr()

In [151]: corr_matrix["median_house_value"].sort_values(ascending=False)

Out[151]: median_house_value    1.000000
          median_income         0.687151
          total_rooms           0.135140
          housing_median_age    0.114146
          households            0.064590
          total_bedrooms        0.047781
          population           -0.026882
          longitude            -0.047466
          latitude             -0.142673
          Name: median_house_value, dtype: float64
```

`corr_matrix` is also a data frame.

The median income is strongly correlated with the median housing value.

The further north is, the lower the median housing value.

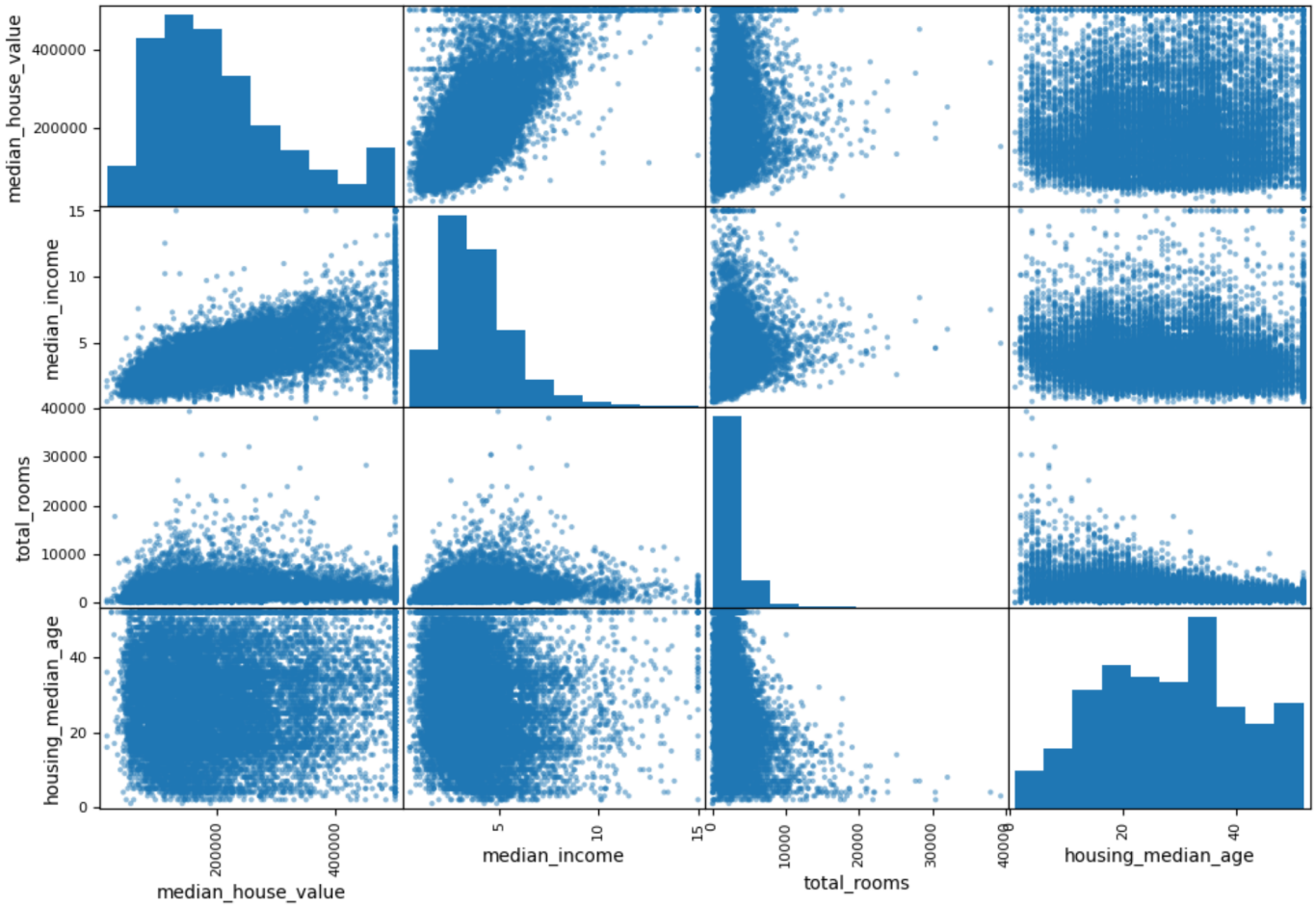
Some are not obvious, such as the housing median age or population.

One can also use pandas' `scatter_matrix`:

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

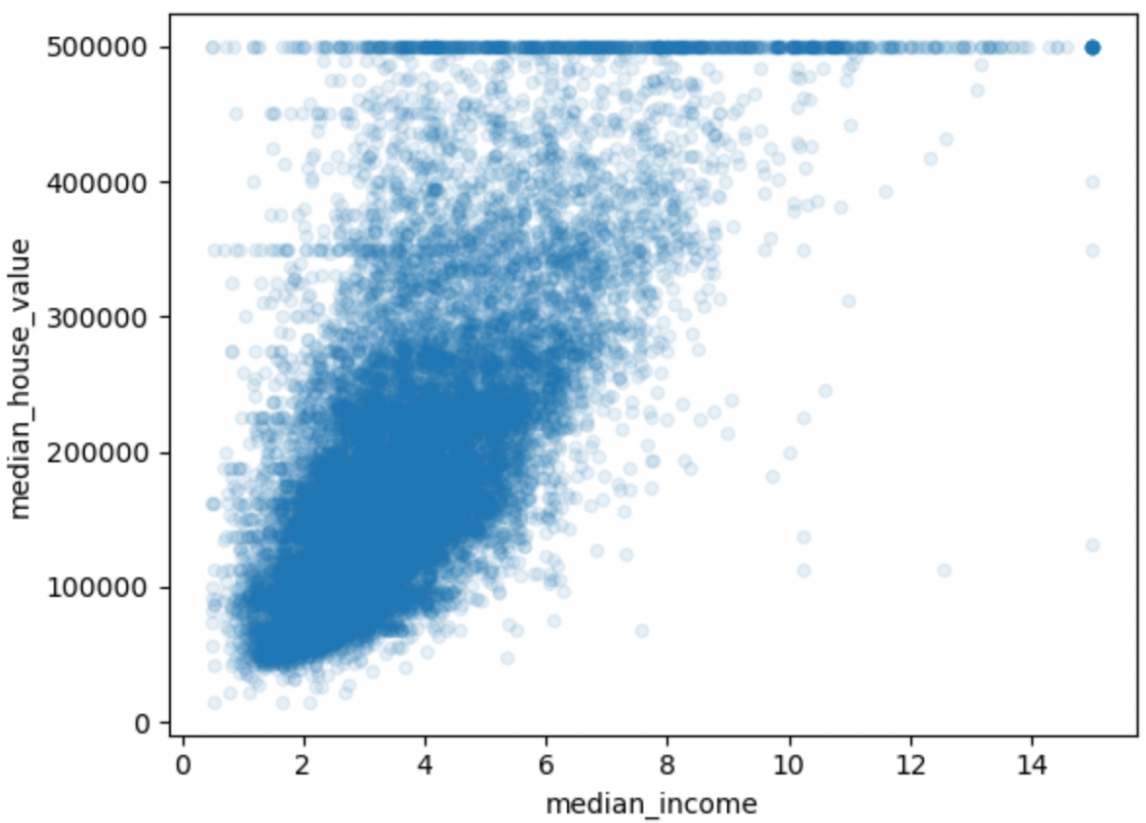
Here, we picked four attributes to look at their pair-wise correlation.

This gives:



Look more carefully at the correlation between the median income and the median housing value:

```
housing.plot(kind="scatter", x="median_income",
             y="median_house_value", alpha=0.1)
```



There are some unnatural lines at 500000, 450000 and 350000, etc. Those data may need to be removed before learning.

Experimenting with Attribute Combinations

- `total_rooms` seems to be not informative by itself. One can try normalizing them over the number of households.
- `total_bedrooms` also seems not informative by itself. One can try normalizing them over `total_rooms`. (Why?)
- The population per household also seems like an interesting attribute to look at.
- Create the following new attributes and look at the correlation again.
- `bedrooms_per_room` is much more correlated with the median housing value than `total_bedrooms` and `total_rooms`.

  The sign is negative: Apparently houses with a lower bedroom/room ratio tend to be more expensive.
- `rooms_per_household` is more correlated with the median

housing value than `total_rooms` of the district. Apparently, the larger the houses, the more expensive they are.

```
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)
```

```
median_house_value          1.000000
median_income               0.687151
rooms_per_household         0.146255
total_rooms                 0.135140
housing_median_age          0.114146
households                  0.064590
total_bedrooms              0.047781
population_per_household   -0.021991
population                 -0.026882
longitude                  -0.047466
latitude                   -0.142673
bedrooms_per_room          -0.259952
Name: median_house_value, dtype: float64
```