Get the Data

Monday, December 16, 2019 10:13 AM

Note Written by: Zulfadli Zainal

Now, lets start practicing.

Get your data and Jupyter Notebook from: https://github.com/ageron/handson-ml

Creating your Workspace

Install Python:

Few method to get python installed in your system;

1. Get python for using PIP: https://www.python.org/
2. Get python via Jupyter Notebook (Browser based): https://jupyter.org/
3. Get python via Conda Distribution (with Spyder IDE): https://www.anaconda.com/distribution/

#My personal preferences, I recommend to use Conda distribution - using PIP installation only if you are a pro.

Usually library/module needed is Jupyter, NumPy, Pandas, Matplotlib, and Scikit-Learn.

Update latest Python:

How to check whether python is already installed in your system (PIP):

```
$ pip3 --version
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

If PIP is installed, you need to check whether your PIP is up to date:

```
$ pip3 install --upgrade pip
Collecting pip
[...]
Successfully installed pip-9.0.1
```

Create Isolated Environment (Virtual Environment): Recommended so that your project will not conflict with library versions.

Creating virtual environment (via PIP):

```
$ pip3 install --user --upgrade virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

Now you can create an isolated Python environment by typing:

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix '[...]'
New python executable in [...]/ml/env/bin/python3.5
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

Now every time you want to activate this environment, just open a terminal and type:

```
$ cd $ML_PATH
$ source env/bin/activate
```

Important - Read This!!
While the environment is active, any package you install using pip will be installed in this isolated environment, and Python will only have access to these packages (if you also want access to the system's site packages, you should create the environment. using virtualenv's --system-site-packages option). Check out virtualenv's documentation for more information.

Install Library:

Now you can install all the required modules and their dependencies using this simple pip command:

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
[...]
```

To check your installation, try to import every module like this:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

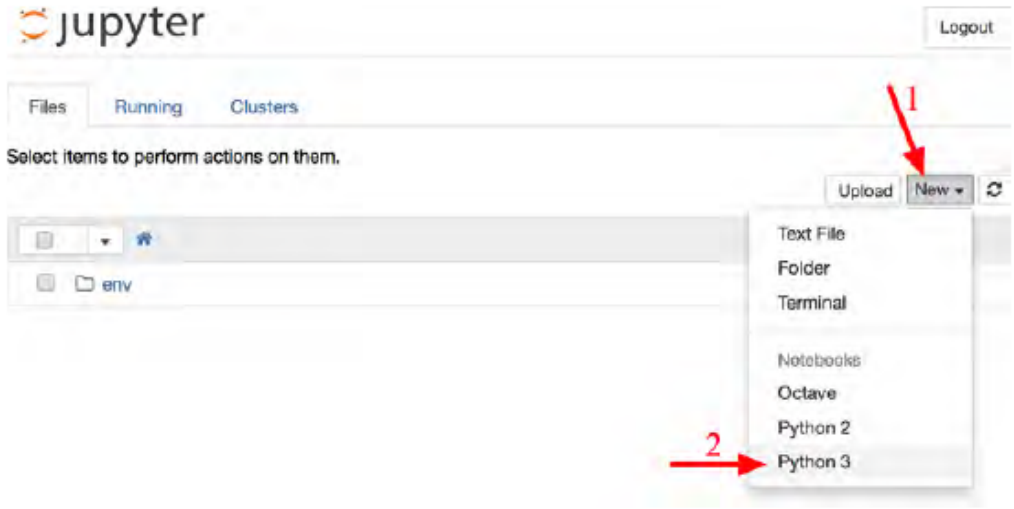
Running Jupyter Notebook:

There should be no output and no error. Now you can fire up Jupyter by typing:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

A Jupyter server is now running in your terminal, listening to port 8888. You can visit this server by opening your web browser to http://localhost:8888/ (this usually happens automatically when the server starts). You should see your empty workspace directory (containing only the env directory if you followed the preceding virtualenv instructions).

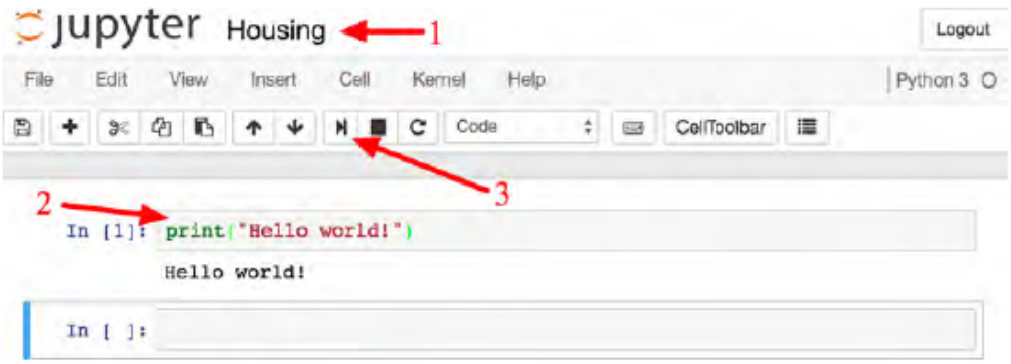
You will see something like this:



By clicking this, it will do 3 things:

1. Create 1 notebook called Untitled.ipynb
2. It starts Jupyter Python Kernel to run this notebook
3. It opens this notebook in new tab.

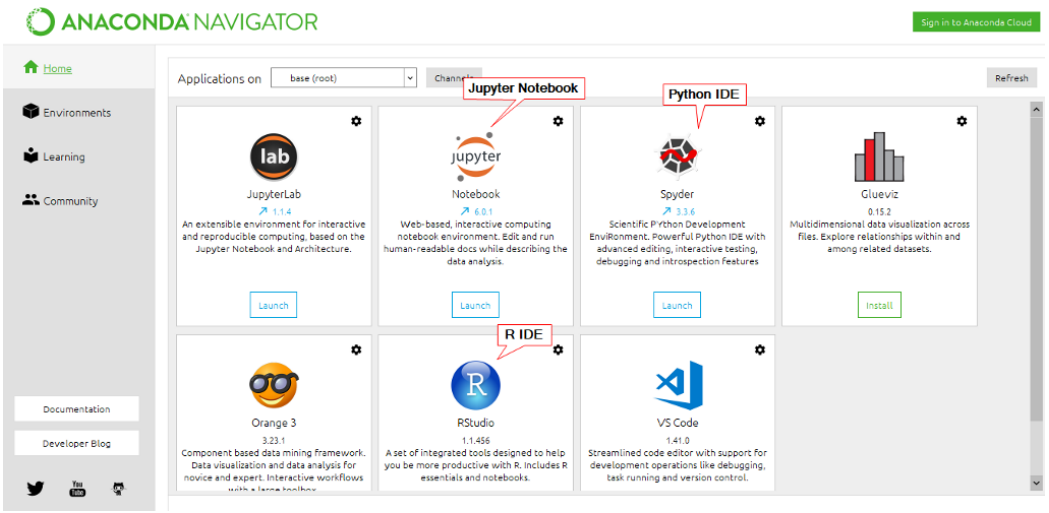
Start to code:



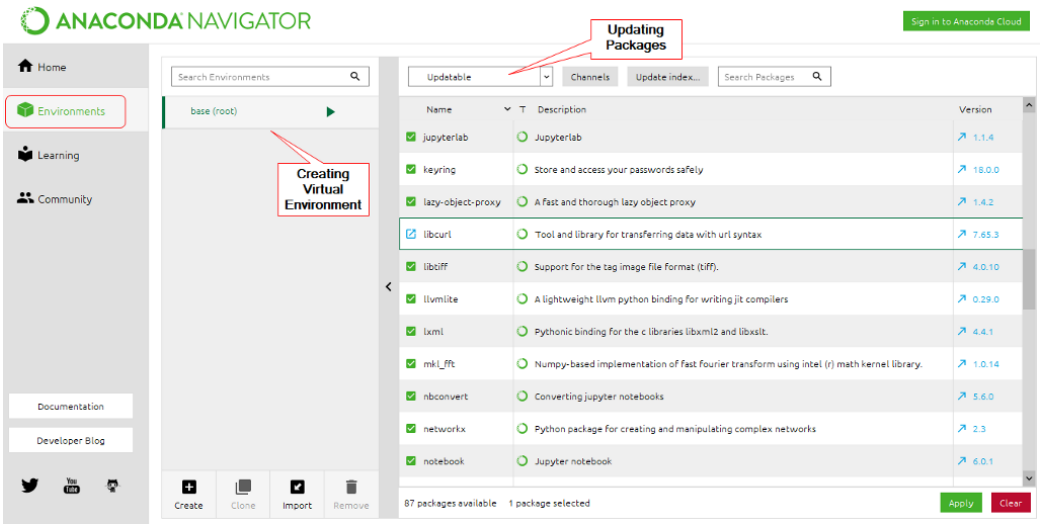
That's great, however..

I have encountered many problems before when using PIP, my codes cannot run because of library conflict. And it takes so much time to troubleshoot before it can work again. Hence this is the Reason I recommended Anaconda Distribution.

1. It's a 1 stop center for Data Science and Machine Learning (Python, R, and more)



2. Creating virtual environment is easy. Managing it is also easy (Update, Install, Remove)



Download the Data

Download the data:

Download your data from: https://github.com/ageron/handson-ml

What this code do:

1. Create *dataset/housing* directory in your workspace
2. Download *housing.tgz* file
3. Extract the file

Or you can just download in manually.

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

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

Loading the Data (Using Pandas):

Import the data using pandas module;

```
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)
```

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)*

Take a Quick Look

Checking the Data:

Take a quick look of the Data:

>>> housing.head() #Top 5 Rows view

```
In [5]: housing.head()
Out[5]:
   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
```

Get some info about the Data:

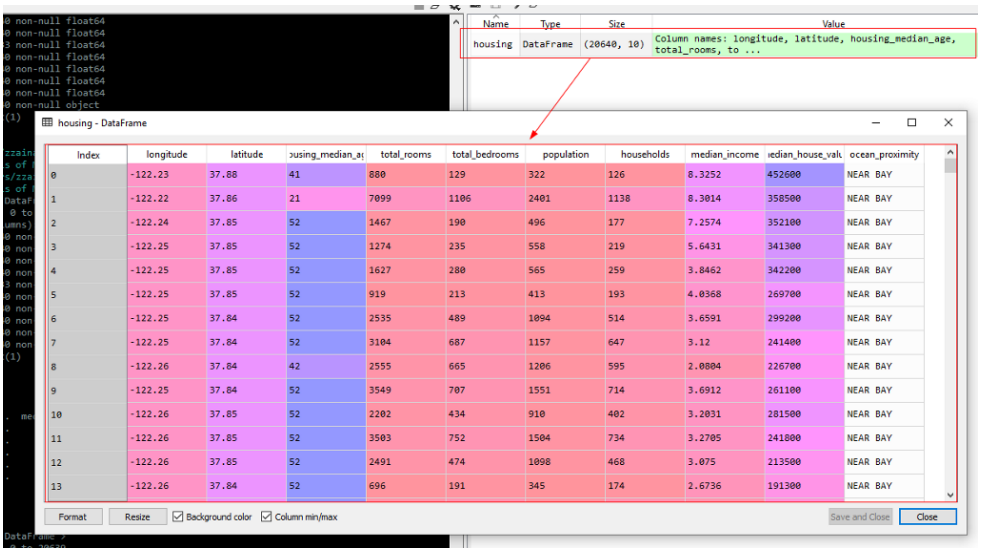
>>> housing.info()

```
In [6]: housing.info()
<class 'pandas.core.frame.Dat
RangeIndex: 20640 entries, 0
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-
households            20640 non-
median_income         20640 non-
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Very Small Row for ML. But Its Good for Practice.

Not enough data problem

Overall Data in Pandas (view in Spyder IDE):



All attributes are numerical, except the ocean_proximity field. Its type is object, so it could hold any kind of Python object, but since you loaded this data from a CSV file you know that it must be a text attribute.

It looks like the data has been categorized based on ocean_proximity field.

To get better overview, we can se how the data is being categorized:

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

```
In [7]: housing["ocean_proximity"].value_counts()
Out[7]:
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

To get better overview (on numerical Attributes), we can se how the data is being categorized.
(Note that: NULL is ignored)

>>> housing.describe()

```
In [8]: housing.describe()
Out[8]:
          longitude      latitude  ...  median_income  median_house_value
count  20640.000000  20640.000000  ...   20640.000000        20640.000000
mean    -119.569704     35.631861  ...       3.870671       206855.816909
std        2.003532      2.135952  ...       1.899822       115395.615874
min     -124.350000     32.540000  ...       0.499900        14999.000000
25%     -121.800000     33.930000  ...       2.563400       119600.000000
50%     -118.490000     34.260000  ...       3.534800       179700.000000
75%     -118.010000     37.710000  ...       4.743250       264725.000000
max     -114.310000     41.950000  ...      15.000100       500001.000000

[8 rows x
```

Statistics

Note:

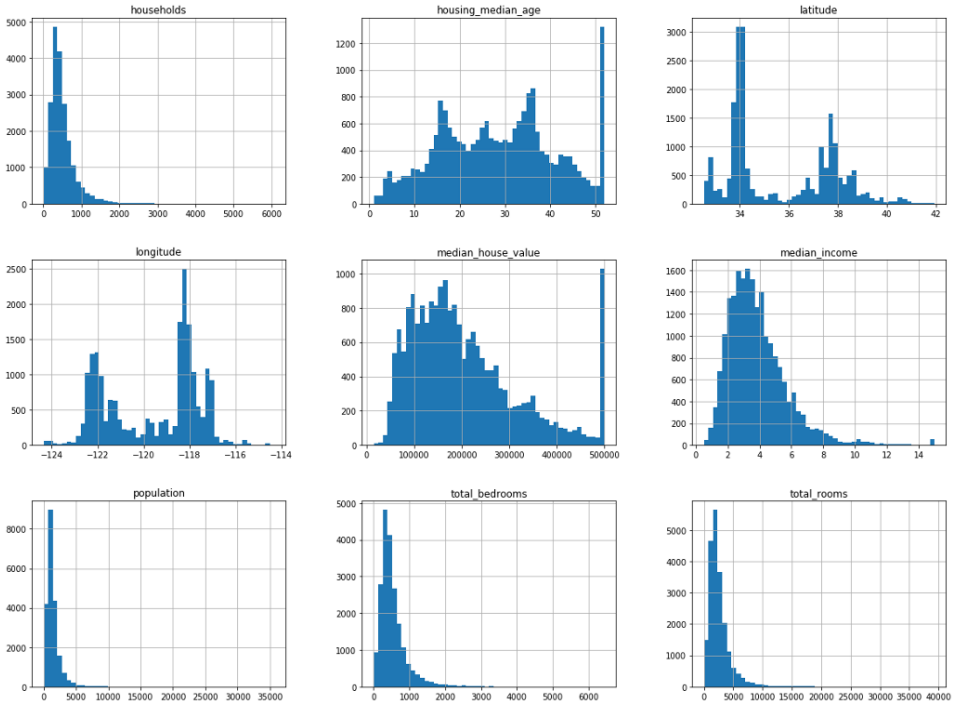
1. *NULL is ignored*
2. 25%, 50%, and 75% represent percentile.
   percentiles: a percentile indicates the value below which a given percentage of observations in a group of observations falls.
3. For Eg: districts have a housing_median_age lower than 18, while 50% are lower than 29 and 75% are lower than 37.

Index	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	iedian_house_valu
count	20640	20640	20640	20640	20433	20640	20640	20640	20640
mean	-119.57	35.6319	28.6395	2635.76	537.871	1425.48	499.54	3.87067	206856
std	2.00353	2.13595	12.5856	2181.62	421.385	1132.46	382.33	1.89982	115396
min	-124.35	32.54	1	2	1	3	1	0.4999	14999
25%	-121.8	33.93	18	1447.75	296	787	280	2.5634	119600
50%	-118.49	34.26	29	2127	435	1166	409	3.5348	179700
75%	-118.01	37.71	37	3148	647	1725	605	4.74325	264725
max	-114.31	41.95	52	39320	6445	35682	6082	15.0001	500001

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute.

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

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



Analyzing the Data:

1. First, the median income attribute does not look like it is expressed in US dollars (USD). After checking with the team that collected the data, you are told that the data 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. Working with preprocessed attributes is common in Machine Learning, and it is not necessarily a problem, but you should try to understand how the data was computed.

2. The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your Machine Learning algorithms may learn that prices never go beyond that limit. You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond $500,000, then you have mainly two options:
    a. Collect proper labels for the districts whose labels were capped.
    b. Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond $500,000).

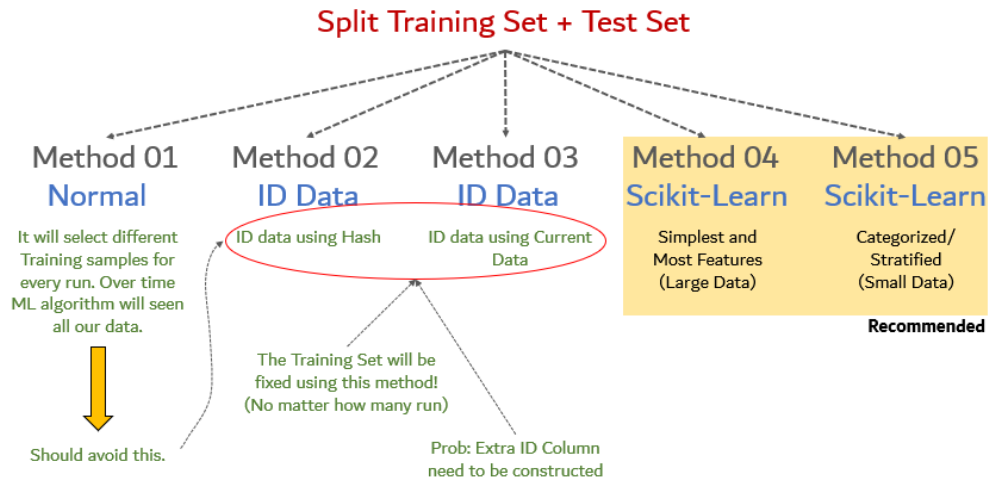
3. These attributes have very different scales. We will discuss this later in this chapter when we explore feature scaling.

4. Finally, many histograms are tail heavy: they extend much farther to the right of the median than to the left. This may make it a bit harder for some Machine Learning algorithms to detect patterns. We will try transforming these attributes later on to have more bell-shaped distributions.

Create Test Set

Creating a test set is theoretically quite simple.

Method to Define -> Split Training + Test Set Function (Based on Test Ratio)



Method 01

This code will split it Randomly

Ideal Rule: 80% Training Samples, 20% Test Sample

```
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)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

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

Result:

Name	Type	Size	Value
housing	DataFrame	(20640, 10)	Column names: longitude, latitu…
test_set	DataFrame	(4128, 10)	Column names: longitude, latitu…
train_set	DataFrame	(16512, 10)	Column names: longitude, latitu…

PROBLEM!!! (Read this, important)

1. This splitting method works, but not perfect.
2. If you run this program again, it will generate a different test set.
3. Because of this, over time, your ML algorithm will get to see your overall data (We need to avoid this)

Solution:

1. Save the test set on the first run and then load it in subsequent runs.
2. Another method is set the random number generator's seed before calling rand function. This will always generates the same shuffled indices.

   Eg: *np.random.seed(42)*

   Before calling *np.random.permutation()*

Problem with this Solution: Every time you update this data set, this solution will break.

Method 02

Solution: Use Unique Identifier

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    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")


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

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    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")
```

Method 03

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

Method 04

Simplest Ways to Split Test & Training Data: Use Scikit-Learn

1. One liner..
2. Benefit: Can input multiple data set, can input random_state (So the training set will not change)

```
from sklearn.model_selection import train_test_split

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

```
from sklearn.model_selection import train_test_split

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

Sampling Bias

Currently: We use RAND method to determine our training samples (This is OK if we have Large samples!)

However, if we only have small samples: Sampling Bias could happen

Eg:

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, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population. If they used purely random sampling, there would be about 12% chance of sampling a skewed test set with either less than 49% female or more than 54% female. Either way, the survey results would be significantly biased.

How to avoid sampling bias:

Create a categorization for the mot important features.

Use that features to segregate Test + Training samples.

Method 05

Categorize Feature:

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

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

Based on Categorize Feature, Stratified (Split using strata method)

```
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]
```

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]*

Check outcome: Strata Method shows less error in Training samples selection

```
>>> housing["income_cat"].value_counts() / len(housing)
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 [27]: housing["income_cat"].value_counts() / len(housing)
Out[27]:
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
```

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

Random: More Error (If Sample Bias

After done, remove categorize attributes so data can become like original:

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

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

Why Test Data Validation is important:

It is important part of ML -> But often neglected. The accuracy of the training dependent on how the training samples is being selected.