End-to-End
Machine Learning Project

Dr Ioanna Stamatopoulou

All material in these lecture notes is based on our textbook:
Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 3rd ed., O’Reilly, 2022

Coming up next:
What does an ML Project typically involve?

(more refined…)

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

Chapter 2 of our textbook takes you through these stages in detail for a particular Case Study

Example Case Study

Your first task is to use California census data to build a model of housing prices in the state.

This data includes metrics such as the population, median income, and median housing price for each district (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).

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

Chapter 2 of our textbook takes you through these stages in detail for the particular Case Study

1.
Look at the Big Picture

1. Look at the big picture
a. Frame the problem

- Define the business objective
  - The ML model is not a goal in itself
  - What does the client business need to achieve, to gain out of it?
  - How are they planning to use the outputs of your system?
- What does the current solution look like?
  - Gives you an insight on the solution and a reference to performance

1. Look at the big picture
a. Frame the problem (cont’d)

- Determine the type of ML model required
  - Type of task
    - — regression, classification, clustering, etc.
  - Training supervision
    - — supervised, unsupervised, etc.
  - Batch or online training

1. Look at the big picture
b. Select a Performance Measure

- Common for **Regression** tasks
  - Root mean square error (RMSE)
  - Mean absolute error (MAE) (if data contains many outliers)
- Common for **Classification** tasks
  - Accuracy
  - Confusion Matrix
  - Precision
  - Recall

1. Look at the big picture
b. Select a Performance Measure (cont'd)

- Common for **Clustering** tasks
  - Silhouette score
  - Calinski-Harabasz Index
  - Davies-Bouldin Index

- Is there a minimum performance that is required?

[time-out to sort out technicalities]

- This is where things start getting practical!
- Let’s take a break to select where/how we are going to work
- Options for you to choose:
  - **Google Colab**
  - Other platforms (e.g. Kaggle)
  - Locally on your machine (Python required)

Google Colab

Visit:

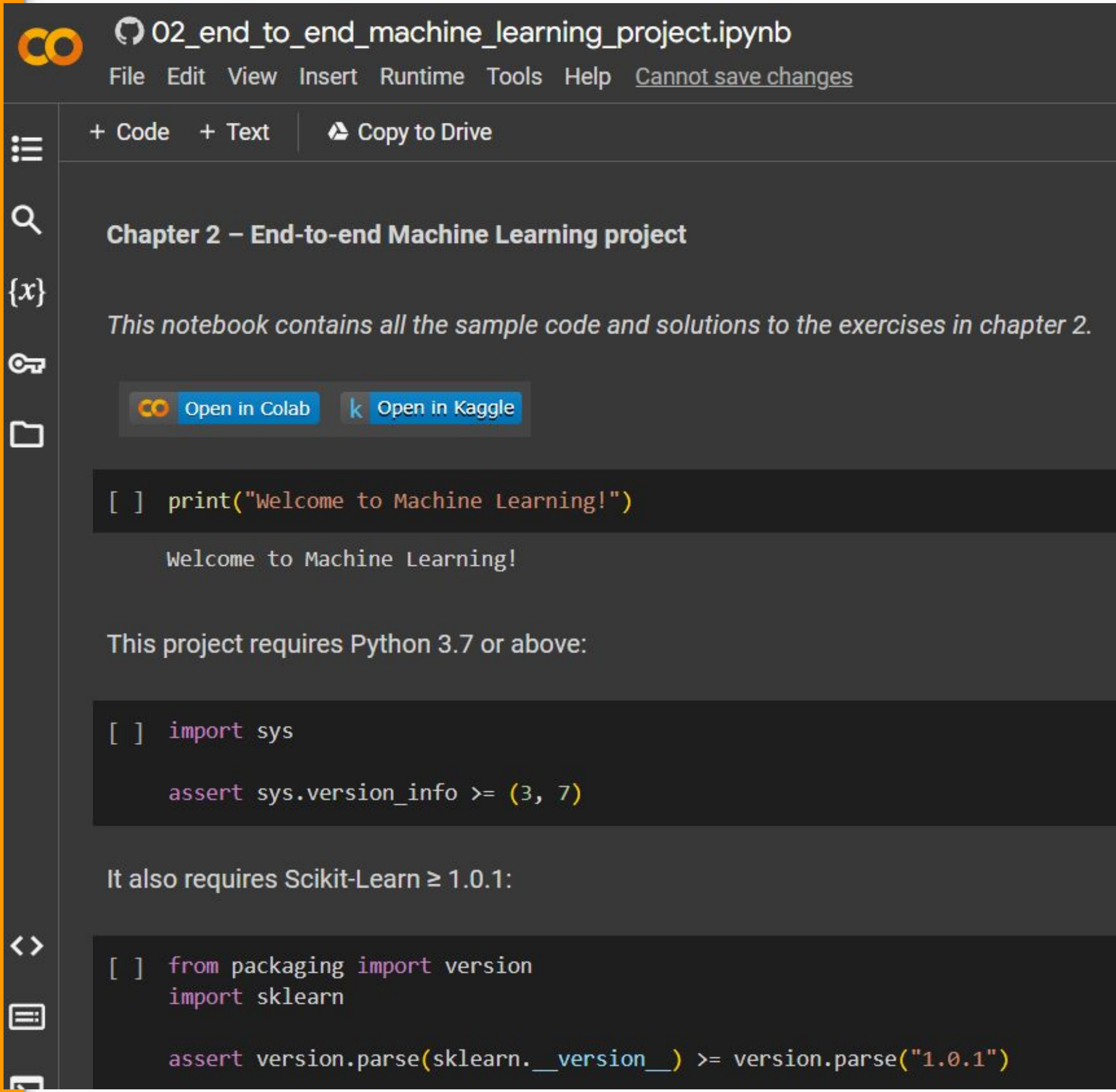
homl.info/colab3

for the California Housing project (select 02_end_to_end_machine_learning_project)

Or generally:

colab.research.google.com/

to create a new project



2.
Get the Data

2. Get the Data
a. Download the Data

- The Data may available in various formats:
  - Relational database
  - Comma-separated values file (CSV)
  - Compressed

Download the Data

California Housing project

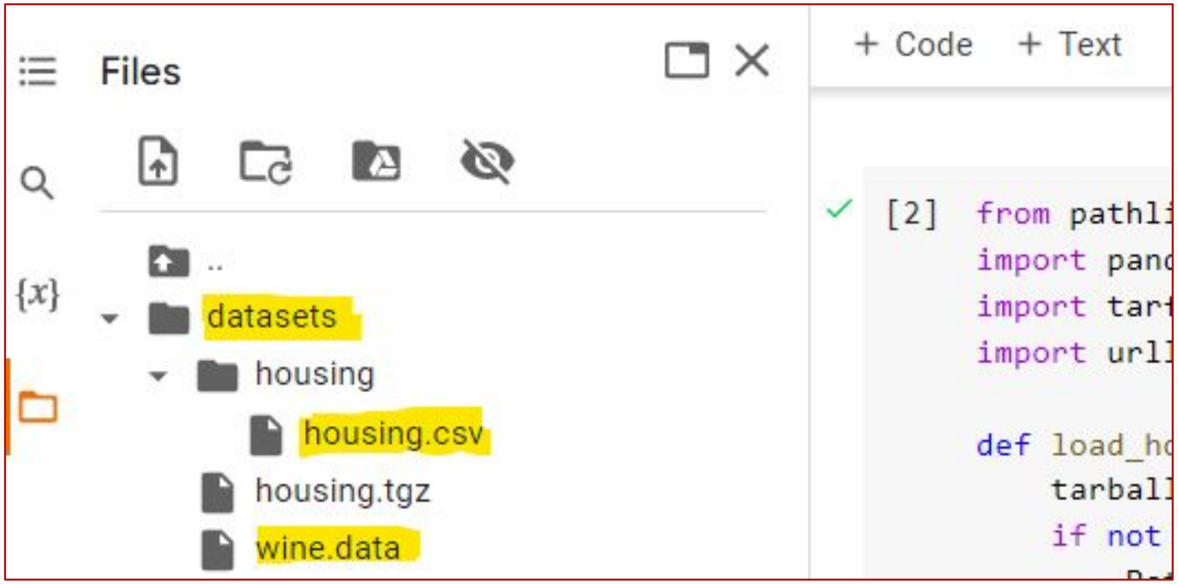
```python
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing = load_housing_data()
```

Download the Data

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_wine_data():
    tarball_path = Path("datasets/wine.data")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "http://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"
        urllib.request.urlretrieve(url, tarball_path)

    return pd.read_csv(Path("datasets/wine.data"))

winedata = load_wine_data()
```

Results of a chemical analysis of **wines** grown in the same region in Italy

.data file. No need for decompression



2. Get the Data
b. Take a Quick Look at the Data Structure

- Fundamentals of Data Visualisation
- Throwback Thursday to the Data Science module!

2b Take a Quick Look at the Data Structure

Attributes overall

```
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
```

```
housing.head()
```

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

2b Take a Quick Look at the Data Structure

Attributes overall

```
housing.describe()
```

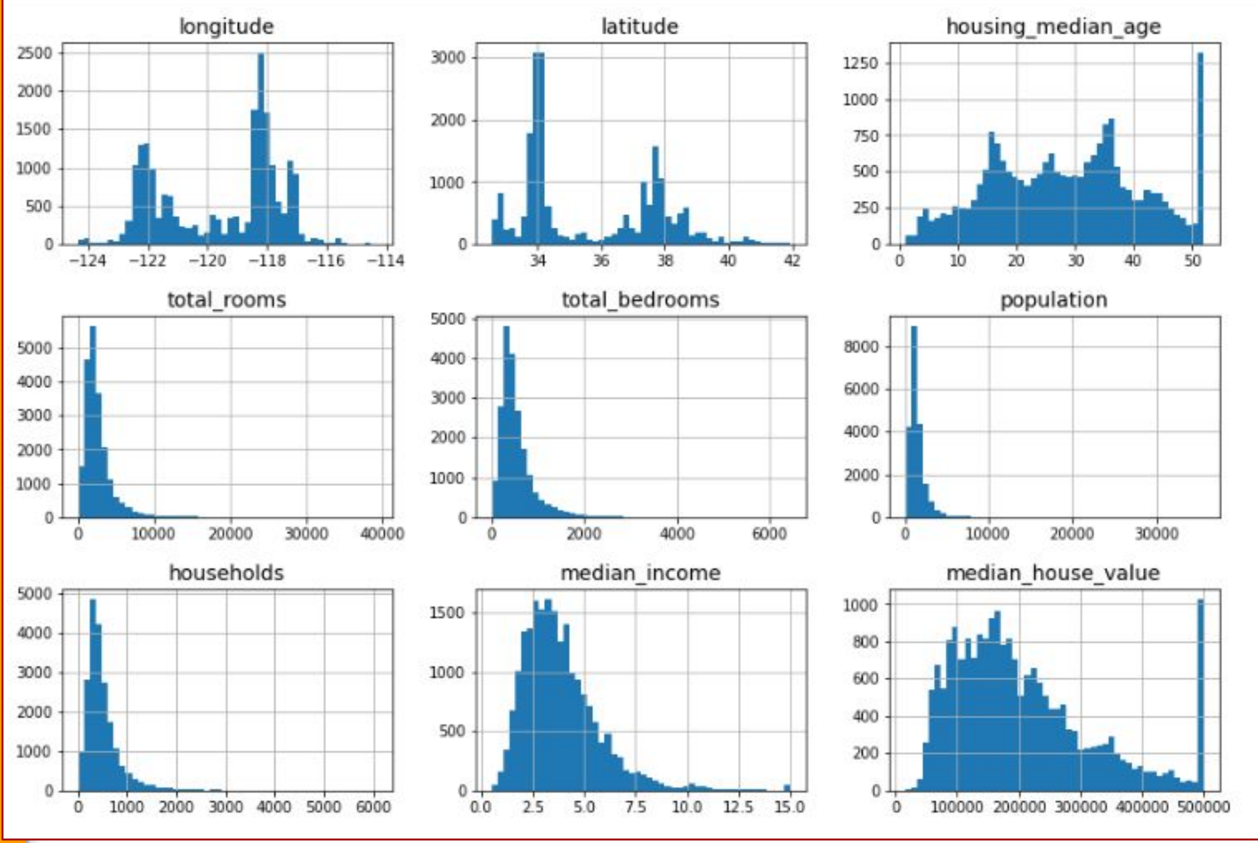
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	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

2b Take a Quick Look at the Data Structure

For individual attributes:

numerical

```
import matplotlib.pyplot as plt

# extra code – the next 5 lines define the default font sizes
plt.rc('font', size=14)
plt.rc('axes', labelsize=14, titlesize=14)
plt.rc('legend', fontsize=14)
plt.rc('xtick', labelsize=10)
plt.rc('ytick', labelsize=10)

housing.hist(bins=50, figsize=(12, 8))
save_fig("attribute_histogram_plots")  # extra code
plt.show()
```



2b Take a Quick Look at the Data Structure

For individual attributes:

categorical

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

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

2. Get the Data
b. Take a Quick Look at the Data Structure

Keep your eyes open for the following (1/2):

- **Capped feature values**
  - `median_income` has been capped in the range [500-15,000]
  - `housing_median_age` has been capped in the range [1-52]
  - `median_house_value` has been capped in the range [15,000-500,000]
    - The latter may be a problem; it is your target feature: Your algorithm will learn that prices do not go beyond the limits
    - Either find the correct values or remove from training/test sets

2. Get the Data
b. Take a Quick Look at the Data Structure

Keep your eyes open for the following (2/2):

- Attributes have very **different scales**

  ⇒ **Feature Scaling**

  ⇒ **Normalisation**

- **Skewing** to the right or left of the median (aka **heavy tail**); makes it harder for the algorithm to detect patterns

  ⇒ **Transformation** to bell-shaped distributions

2. Get the Data
c. Create a Test Set

Random Sampling

- Put aside part of the available data and never look at it!
  - Your brain is an amazing pattern detection system and, therefore, prone to overfitting
  - You might identify patterns that will lead you to select a particular ML model

  ⇒ **data snooping bias**

- Remember the rule of thumb? Split the data randomly into:
  - **80%** for training
  - **20%** for test

2. Get the Data
c. Create a Test Set

Random Sampling (cont'd)

- This is very easy to do but there is a problem:
  - Everytime you run the test set generation, you will get a different Test Set
  - You do not want want your algorithm to be exposed to all instances during training!
- Possible Solutions:
  - Save the test set separately on the first run and use the same all times afterwards
  - Set a random number generator seed so it always generates the same shuffled indices (Common seed used: **42. Why?)**

And don't forget **Prolog**'s 42…

Best Trivia ever!!!!!!

According to the **Hitchhiker's Guide to the Galaxy** by **Douglas Adams** (one of the best and funniest books ever written):

42 is the Answer to the Ultimate Question of Life, Universe and Everything

www.youtube.com/watch?v=mPPeneo-jqM

www.youtube.com/watch?v=D6tINlNluuY

2. Get the Data
c. Create a Test Set

Random Sampling (cont’d)

- The previously suggested solutions to generating the test set is still not optimal

- It will “break”, if your dataset gets updated

2. Get the Data
c. Create a Test Set

Stratified Sampling

- Random Sampling is ok, if your dataset is large enough relative to the number features
- If not, you are in danger of introducing sampling bias
- Example: US population is 51.1% females and 48.9% males
  - This ratio should be kept in the test set, if people’s answers may vary across genders
- An expert will typically tell you which feature may have such an effect on the target feature

2. Get the Data
c. Create a Test Set

Stratified Sampling (cont'd)

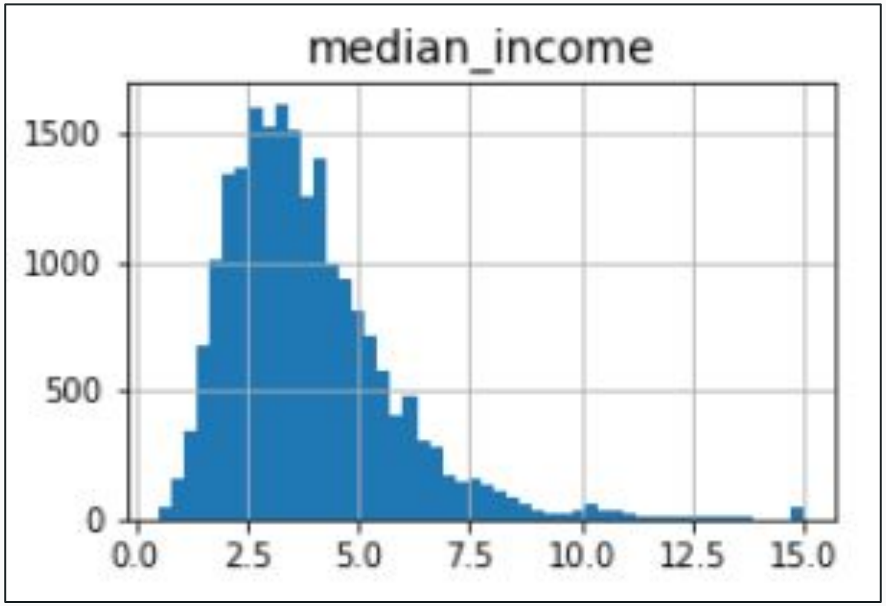
- Split your dataset into **strata**
- Sample the right number of instances from each stratum

Stratum
A homogeneous subgroup of the dataset

Stratified Sampling

An expert tells you that the median income is very important in predicting house prices

Median income is a continuous numerical attribute so we must create the strata, in this case called income categories

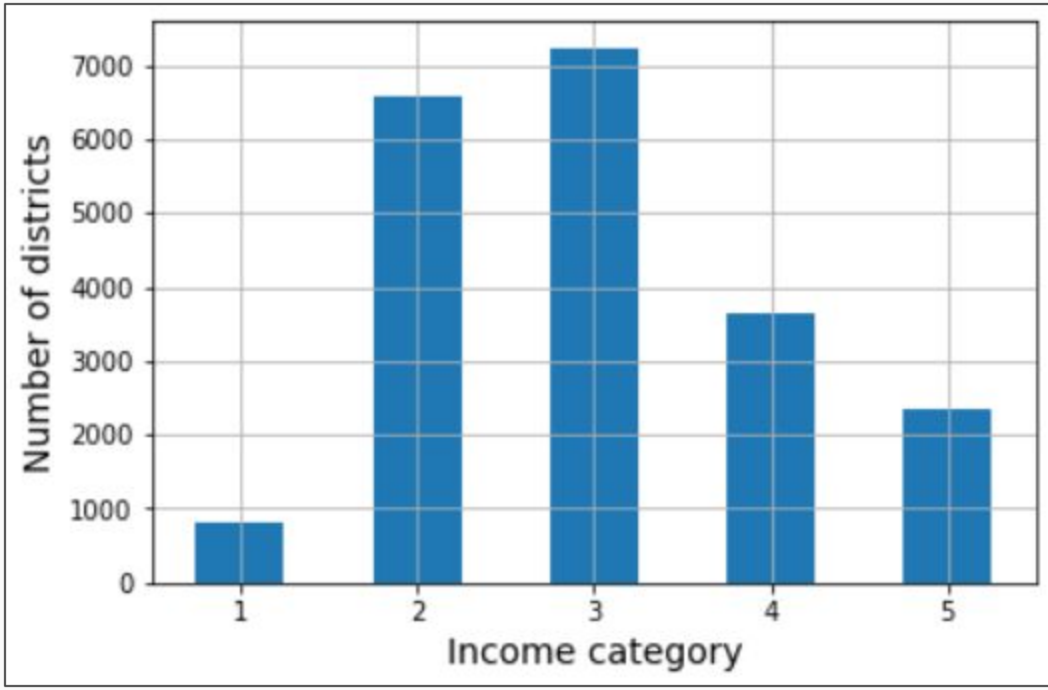


Stratified Sampling

```
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"].value_counts().sort_index().plot.bar(rot=0, grid=True)
plt.xlabel("Income category")
plt.ylabel("Number of districts")
save_fig("housing_income_cat_bar_plot")  # extra code
plt.show()
```

Avoid having too many strata so that each one of them has a sufficient number of instances



Stratified vs Random Sampling

Comparison of how each stratum is represented in the test set when using

- Stratified Sampling
- Random Sampling

Stratified:

```
strat_train_set, strat_test_set = train_test_split(
    housing, test_size=0.2, stratify=housing["income_cat"], random_state=42)
```

```
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
```

Random:

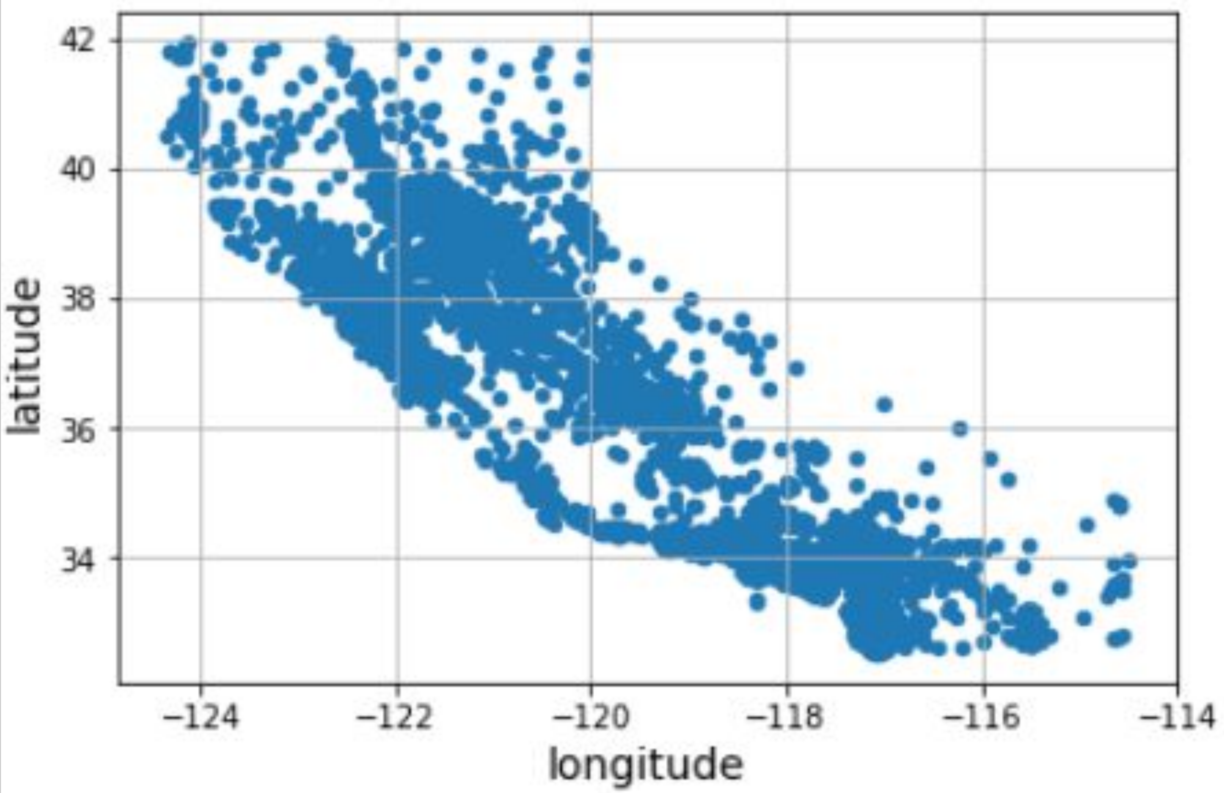
```
from sklearn.model_selection import train_test_split
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

Income Category	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

3.
Explore and Visualise the Data

2. Explore and Visualise Data

a. Visualising Geographical Data

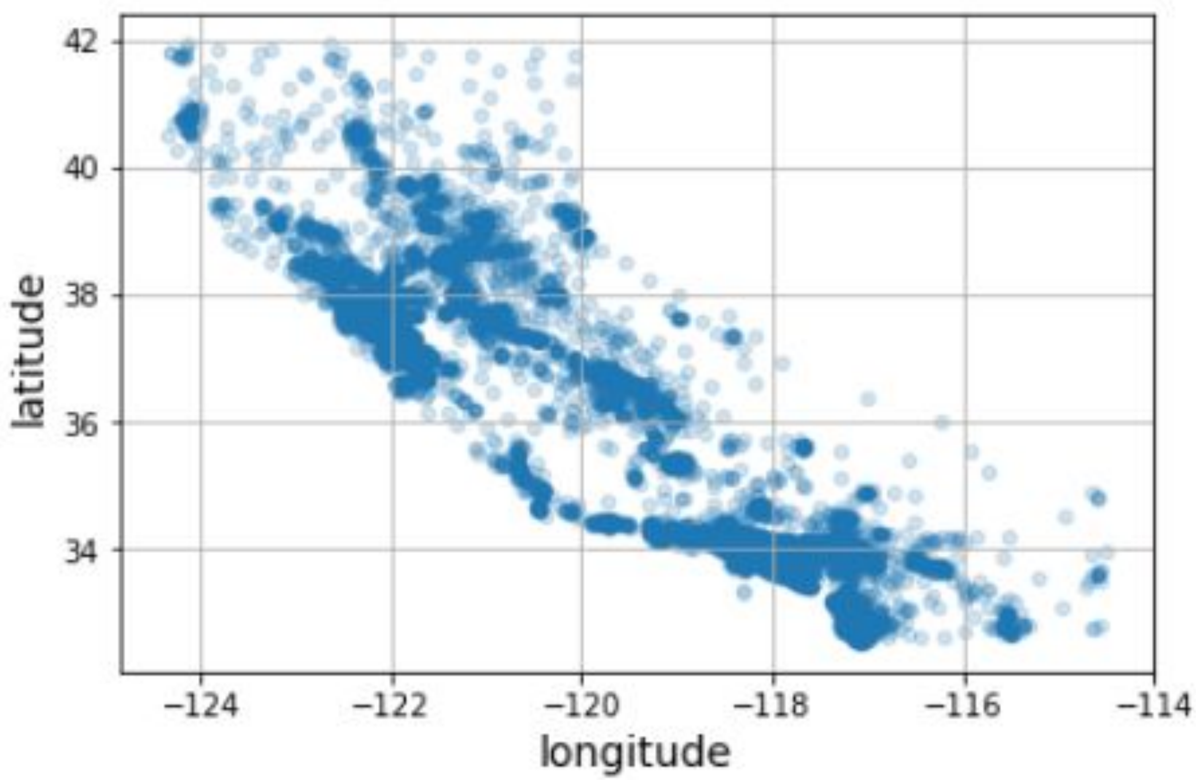



```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)
save_fig("bad_visualization_plot")  # extra code
plt.show()
```

2. Explore and Visualise Data

a. Visualising Geographical Data

With density

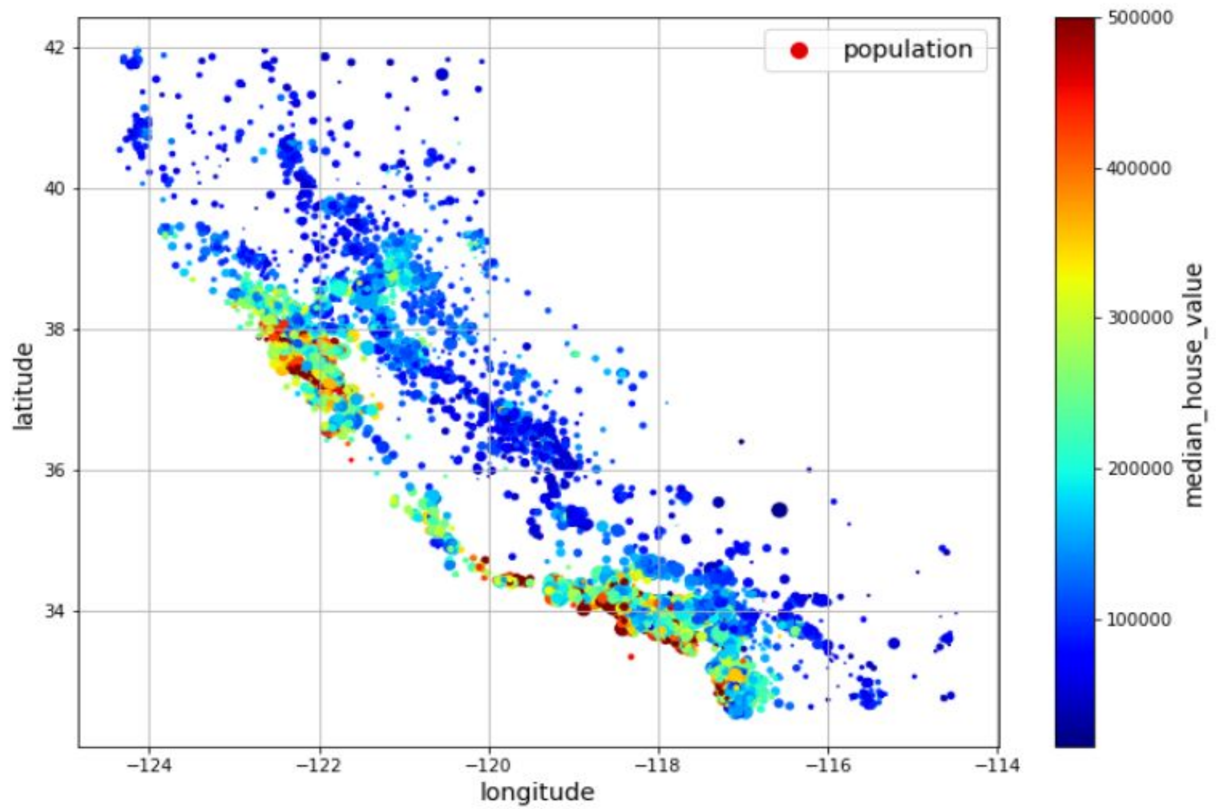



```python
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)
save_fig("better_visualization_plot")  # extra code
plt.show()
```

2. Explore and Visualise Data

a. Visualising Geographical Data

Against your target attribute




```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))
save_fig("housing_prices_scatterplot")  # extra code
plt.show()
```

3. Explore and Visualise the Data
b. Look for correlations

Using the Standard Correlation Coefficient

- Compute the **standard correlation coefficient**
  - Also called **Pearson's *r***
  - Ranges from **-1** to **1**
- Look into how much each of the attributes correlates with your target attribute
  - ~1 (strong positive correlation)
    $\Longrightarrow$ if x goes up, y goes up
  - ~-1 (strong negative correlation)
    $\Longrightarrow$ if x goes up, y goes down
  - ~0 means no linear correlation

Standard Correlation Coefficient

For the California Housing project

Pearson's *r* correlation matrix is computed

Correlations of all attributes with the target **`median_house_value`** are listed

```
corr_matrix = housing.corr(numeric_only=True)
```

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

```
median_house_value     1.000000
median_income          0.688380
total_rooms            0.137455
housing_median_age     0.102175
households             0.071426
total_bedrooms         0.054635
population            -0.020153
longitude             -0.050859
latitude              -0.139584
Name: median_house_value, dtype: float64
```

Median income seems to be the most promising attribute to predict the median house value

3. Explore and Visualise the Data
b. Look for correlations

Using Pandas Scatter Matrix

- Plots every numerical attribute against every other numerical attribute

- If you already have an insight about the correlations, you may choose to focus on fewer attributes

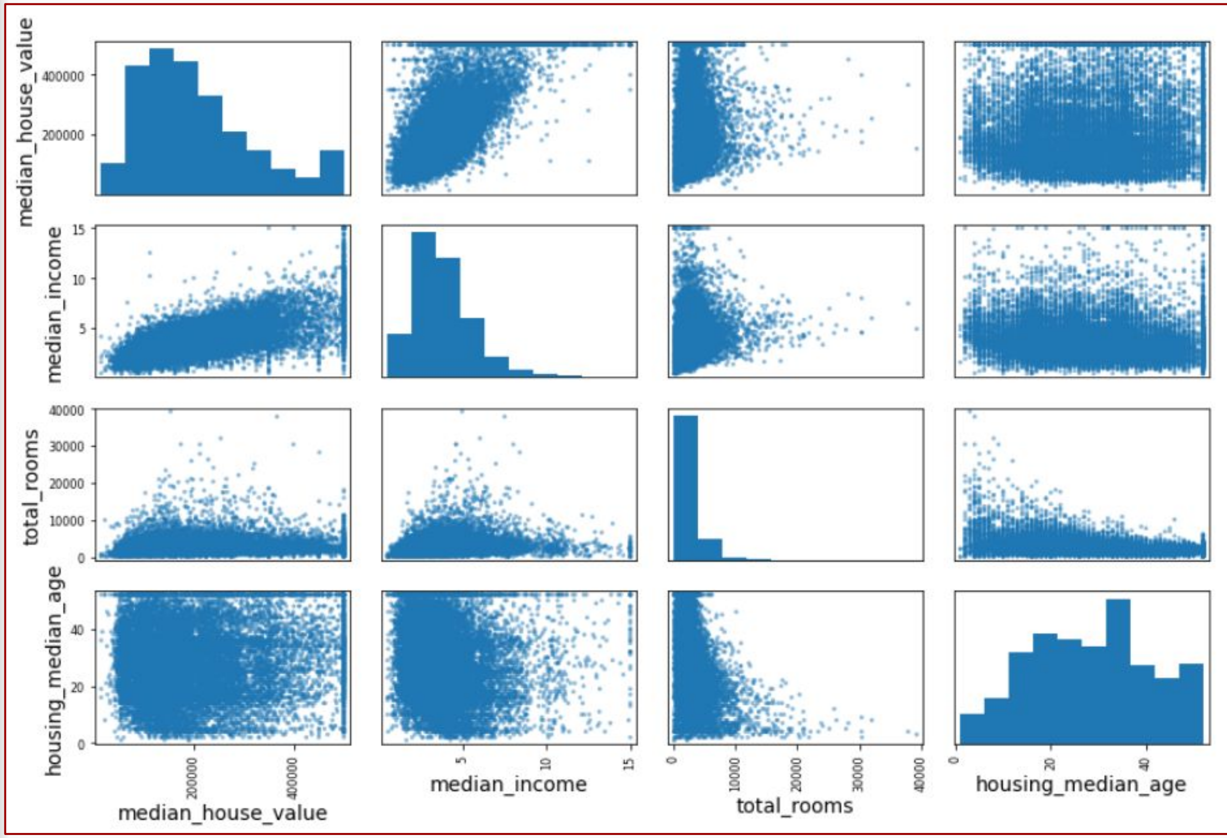
Pandas Scatter Matrix

California Housing project

Pearson's *r* is computed

Correlations of all attributes with the target **`median_house_value`** are listed

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
save_fig("scatter_matrix_plot")  # extra code
plt.show()
```



3. Explore and Visualise the Data
b. Look for correlations

- Pearson's *r* identifies only linear correlations
- If there is another correlation, Pearson's *r* will be zero

Figure 2-14. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

- Scatter Matrices can help identify non-linear correlation
- You may then use other additional techniques (indicatively, have a look here, if you wish)

3. Explore and Visualise the Data
c. Experiment with Attribute Combinations

- Explore whether the ratio of two attributes is more informative than the individual attributes
- Re-compute the correlation matrix
- This is a step you may come back to many times in the duration of the project as you will gain more and more insight progressively

Standard Correlation Coefficient

For the California Housing project

Pearson's *r* correlation matrix is re-computed to include attribute combinations

```
housing["rooms_per_house"] = housing["total_rooms"] / housing["households"]
housing["bedrooms_ratio"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["people_per_house"] = housing["population"] / housing["households"]
```

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

```
median_house_value     1.000000
median_income          0.688380
rooms_per_house        0.143663
total_rooms            0.137455
housing_median_age     0.102175
households             0.071426
total_bedrooms         0.054635
population            -0.020153
people_per_house      -0.038224
longitude             -0.050859
latitude              -0.139584
bedrooms_ratio        -0.256397
Name: median_house_value, dtype:
```

The bigger the house, the more expensive it is

The smaller the ratio, the more expensive the house is

4.
Prepare the Data for ML Algorithms

4. Prepare the Data for ML Algorithms
a. Data Cleaning

- Throwback to when we discussed the “poor-quality data” challenge
- Three options for the case of instances missing a feature value (or more):
  1. ignore the instances
  2. ignore the entire feature
  3. fill in the missing values (e.g. with zeros, the mean, the median value, etc.)
     ⇒ **imputation**

Data Cleaning: Imputing Numeric Values

For the California Housing project

The **`total_bedroom`** attribute has missing values

```
null_rows_idx = housing.isnull().any(axis=1)
housing.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	ocean_proximity
14452	-120.67	40.50	15.0	5343.0	NaN	2503.0	902.0	3.5962	INLAND
18217	-117.96	34.03	35.0	2093.0	NaN	1755.0	403.0	3.4115	<1H OCEAN
11889	-118.05	34.04	33.0	1348.0	NaN	1098.0	257.0	4.2917	<1H OCEAN
20325	-118.88	34.17	15.0	4260.0	NaN	1701.0	669.0	5.1033	<1H OCEAN
14360	-117.87	33.62	8.0	1266.0	NaN	375.0	183.0	9.8020	<1H OCEAN

Data Cleaning: Imputing Numeric Values

For the California Housing project

The **`total_bedroom`** attribute has missing values

1. ignore the instances
2. ignore the feature
3. impute with median

using **Pandas DataFrame**

```
# option 1
housing_option1 = housing.copy()
housing_option1.dropna(subset=["total_bedrooms"], inplace=True)
```

```
# option 2
housing_option2 = housing.copy()
housing_option2.drop("total_bedrooms", axis=1, inplace=True)

housing_option2.loc[null_rows_idx].head()
```

```
# option 3
housing_option3 = housing.copy()
median = housing["total_bedrooms"].median()
housing_option3["total_bedrooms"].fillna(median, inplace=True)

housing_option3.loc[null_rows_idx].head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	pop
14452	-120.67	40.50	15.0	5343.0	434.0	
18217	-117.96	34.03	35.0	2093.0	434.0	
11889	-118.05	34.04	33.0	1348.0	434.0	
20325	-118.88	34.17	15.0	4260.0	434.0	
14360	-117.87	33.62	8.0	1266.0	434.0	

4. Prepare the Data for ML Algorithms
a. Data Cleaning

Imputing Numeric Values with Scikit-Learn

- Filling in the missing data is the least destructive option
- Instead of Pandas, use Scikit-Learn's **`SimpleImputer`** class. Benefit:
  - It stores the median of each numerical attribute and can later be used to impute missing values in the validation set, test set, and any new data
- You need to create a copy of the data with only the numerical attributes

Data Cleaning

For the California Housing project

The **total_bedroom** attribute has missing values

Impute with median

using
Scikit Learn's
SimpleImputer

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

Separating out the numerical attributes:

```
housing_num = housing.select_dtypes(include=[np.number])
imputer.fit(housing_num)
```

Computed medians of all attributes are stored in the **statistics_** variable:

```
imputer.statistics_
```

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

Transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

X is a NumPy array (no column names, no indices)
To transform to a Pandas DataFrame:

```
imputer.feature_names_in_
```

```
array(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
       'total_bedrooms', 'population', 'households', 'median_income'],
      dtype=object)
```

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

Data Cleaning

Impute with Other Techniques

using
Scikit Learn's
`SimpleImputer`

```
imputer = SimpleImputer(strategy="mean")

imputer = SimpleImputer(strategy="most_frequent")

imputer = SimpleImputer(strategy="constant", fill_value=...)
```

4. Prepare the Data for ML Algorithms
a. Data Cleaning

Imputing Categorical Values

- A common technique is to use the most frequent value
- Other methods available as well
- For more info check here and here

4. Prepare the Data for ML Algorithms
b. Transforming Categorical Attributes

Vocabulary Mapping

- ML algorithms work better with numerical values
- When an attribute is categorical (value is text but there is a finite set of possible values), one solution is to encode each value/category with a number ⇒ **Vocabulary Mapping**
- This works well in cases when categories are ordered (bad, average, good, excellent) because the algorithm will understand by the numerical value of the categories when two of them are close (but not well in other cases)

4. Prepare the Data for ML Algorithms
b. Transforming Categorical Attributes

One-Hot Encoding

- When the categories are unordered, it is better to encode them as binary attributes ⇒ **One-Hot Encoding**
  - Length of value is the number of categories
  - For each category one value of the binary is **1 (hot)** while all the others are **0s (cold)**

```
[0., 0., 0., 1., 0.]
[1., 0., 0., 0., 0.]
[0., 1., 0., 0., 0.]
...,
[0., 0., 0., 0., 1.]
[1., 0., 0., 0., 0.]
[0., 0., 0., 0., 1.]
```

One-Hot Encoding

California Housing project

using
Scikit Learn's
`OneHotEncoder`

`OneHotEncoder` provides a number of very useful functionalities to deal also with cases that in the future new categories will appear in your dataset

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

```
housing_cat_1hot
```

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

```
housing_cat_1hot.toarray()
```

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

```
cat_encoder.categories_
```

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

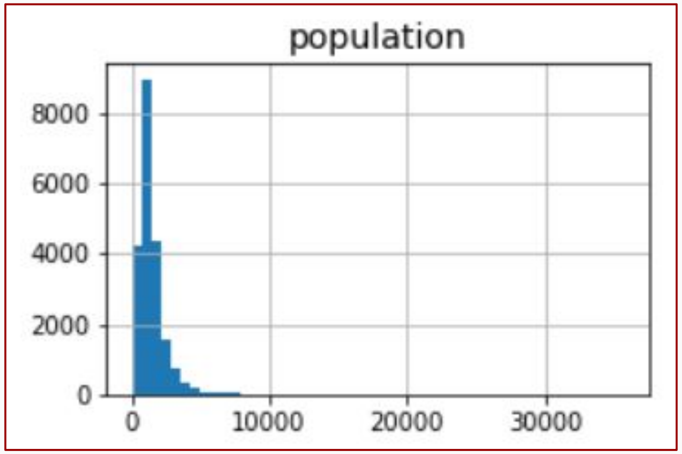
4. Prepare the Data for ML Algorithms
b. Transforming Categorical Attributes

Other techniques

- One-Hot Encoding will lead to a huge number of input features if the categories of an attribute are many
- As an alternative, considering replacing the attribute with an equivalent numerical one (e.g. in the case of ocean proximity with **`distance_from_the_ocean`**)
- A number of alternative techniques are available here: https://github.com/scikit-learn-contrib/category_encoders
- Additional alternative techniques are available when working with Neural Networks

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

- ML algorithms do not appreciate working with:
  - Numeric attributes whose values have very **different scales**
  - Numeric attributes with **skewed distributions** (i.e. heavy tails to the left or right):



Throwback to:
"Take a Quick Look at the Data Structure and Keep your eyes open for the following"

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

- **Normalisation** for scaling
  - min-max Scaling
  - z-score Standardisation
- **Transformations** for skewed distributions
  - **Log** Transformation
  - **Bucketisation** aka Bucketing aka Binning
    - Equally-spaced
    - Quantile *(outside the scope of this module)*

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

Normalisation: min-max Scaling

$$x' = (x - x_{min})/(x_{max} - x_{min})$$

- In **min-max Scaling** values are re-scaled so that they end up in the range:
  - from 0 to 1
  - from - 1 to 1 (zero-mean; better, especially for NNs)
- Simple but affected by outliers

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

Normalisation: min-max Scaling

- **min-max Scaling** is a good choice when:
  - You know the approximate upper and lower bounds on your data with few or no outliers
  - Your data is approximately uniformly distributed across that range
- Good example: age
- Bad example: income (few people with high income)

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

Normalisation: z-score Standardisation

$$x' = (x - \mu)/\sigma$$

- **z-score Standardisation**:
  - Subtracts mean value (i.e. zero mean)
  - Divides by the standard deviation (i.e. stdev = 1)
- Not affected by outliers
- Does not restrict to a specific range

min-max Scaling and Z-score Standardisation

California Housing project

using Scikit Learn

```python
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(-1, 1))
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

```python
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

Transformation: Log

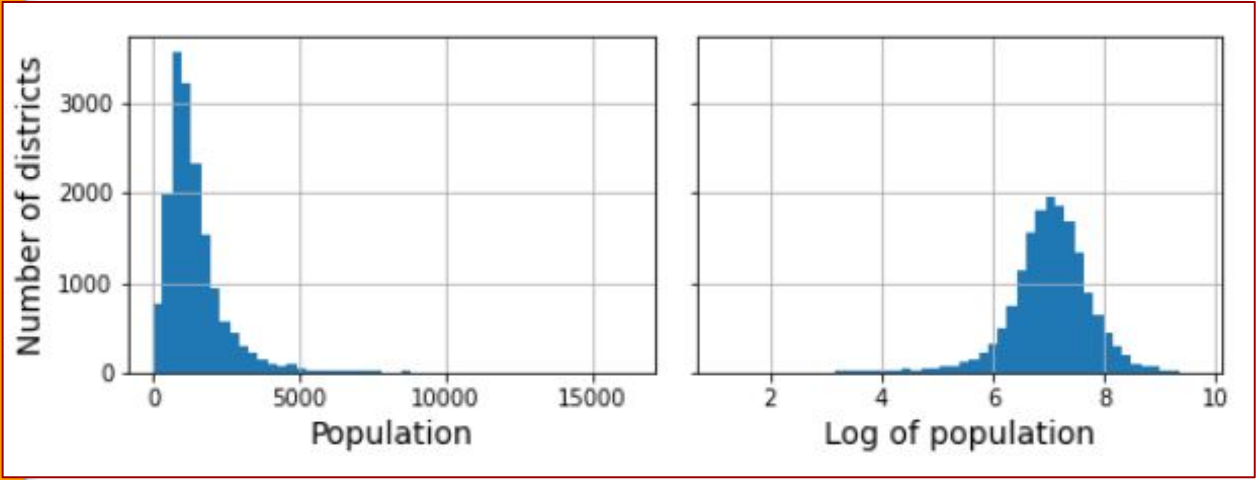
- A **skewed** or **heavy-tail** attribute is one whose data values trail off more sharply on one side than on the other
  - **Power Law Distribution**: one quantity varies as a power of another
- *Before scaling*, we want to make the distribution roughly symmetrical. Use:
  - Square root
  - **Log**

Log Transformation

California Housing project

using
Scikit Learn

```python
# extra code – this cell generates Figure 2–17
fig, axs = plt.subplots(1, 2, figsize=(8, 3), sharey=True)
housing["population"].hist(ax=axs[0], bins=50)
housing["population"].apply(np.log).hist(ax=axs[1], bins=50)
axs[0].set_xlabel("Population")
axs[1].set_xlabel("Log of population")
axs[0].set_ylabel("Number of districts")
save_fig("long_tail_plot")
plt.show()
```



Terminology

Normalisation
The process through which numeric features are transformed to be on a similar scale; it improves the performance and training stability of the model

min-max Scaling
The simplest normalisation method, which consists in rescaling the range of features to scale the range in $[0, 1]$ or $[-1, 1]$

z-score Standardisation
The process of normalizing every value in a dataset such that the mean of all of the values is 0 and the standard deviation is 1

Log Transformation
The process of replacing the value of an attribute by its log so that it is less/not skewed

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

Transformation: Equally-spaced Bucketisation

- **Equally-spaced Bucketisation** is about chopping the distribution into equal-sized buckets
  - Replace each value with the <u>index of the bucket</u> it belongs to
  - Rationale is similar to the one followed in *stratified sampling*
  - Typically results in uniform distribution, i.e. no need for further scaling
  - Divide by the number of buckets and values are in the range [0–1]

Terminology

Bucketisation
The process through which numeric features are transformed into categorical

Equally-spaced Bucketisation
The process through which numeric features are transformed into categorical, using a set of thresholds to set the buckets' size

4. Prepare the Data for ML Algorithms
c. Transforming Numeric Data

- Your target values may also need to be transformed
  - E.g. the target's distribution has a heavy tail on the right and you replace it with its logarithm
- In this case, remember that the predicted value of the model will be a transformed value that needs to be inversely transformed

FYI

4. Prepare the Data for ML Algorithms
d. Custom Transformers

- Instead of cleaning the data manually, write functions
- Writing your own custom transformers allows you to:
    - Reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset)
    - Gradually build a library of transformation functions that you can reuse in future projects
    - Use these functions in your live system to transform the new data before feeding it to your algorithms
    - Easily try various transformations and see which combination of transformations works best

FYI

4. Prepare the Data for ML Algorithms
e. Transformation Pipelines

- Many data transformation steps that need to be executed in the right order
- Scikit-Learn's **Pipeline** help with this

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
```

5.
Select and Train a Model

5. Select and Train a Model
a. Train and Evaluate on the Training Set

- Select your (first and simpler) model
- Run on an indicative number of instances, look at the predictions and compare them to the actual labels
- Run on the entire Training Set and use your selected **performance measure** to see what the success/error is
- **Identify and record** your observations

Trying out Linear Regression (1/2)

California Housing project

```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
housing_predictions = lin_reg.predict(housing)
```

Compare indicatively the predicted results:

```
housing_predictions[:5].round(-2)  # -2 = rounded to the nearest hundred
```

```
array([243700., 372400., 128800.,  94400., 328300.])
```

With the actual results:

```
housing_labels.iloc[:5].values
```

```
array([458300., 483800., 101700.,  96100., 361800.])
```

Trying out Linear Regression (2/2)

California Housing project

```
# extra code – computes the error ratios discussed in the book
error_ratios = housing_predictions[:5].round(-2) / housing_labels.iloc[:5].values - 1
print(", ".join([f"{100 * ratio:.1f}%" for ratio in error_ratios]))
```

```
-46.8%, -23.0%, 26.6%, -1.8%, -9.3%
```

This model is **underfitting**:

- features may not provide enough info
- model may not be powerful enough

```
from sklearn.metrics import mean_squared_error

lin_rmse = mean_squared_error(housing_labels, housing_predictions,
                              squared=False)
lin_rmse
```

```
68687.89176590038
```

If error very low
⇒ **overfitting**

5. Select and Train a Model
b. Cross-Validation

- **Select and run other alternative models!**
- The goal is to quickly shortlist a few of them as more promising
  - **Without spending time fine-tuning**
- Use **Cross-Validation** on all of them
- As we discussed, the typical way is by putting aside part of the Training Set as a Validation Set

5. Select and Train a Model

b. Cross-Validation

k-fold cross-validation

- Randomly splits the training set into ***k*** distinct, non-overlapping subsets called **folds**
- Trains the model ***k*** times. Each time:
  - One different fold is used for validation
  - The other ***k-1*** folds are used for training
- The result is an array containing ***k*** evaluation scores

5. Select and Train a Model
b. Cross-Validation

k-fold cross-validation

- Advantage: It is a very useful technique as it gives you not only the mean error but also its standard deviation (not feasible with one validation set)
- Disadvantage: training takes place *k* times; this is not always feasible

k-fold (***k*** = 10) cross-validation

California Housing project

ML Algorithm: Linear Regression

```
from sklearn.model_selection import cross_val_score
```

```
# extra code – computes the error stats for the linear model
lin_rmses = -cross_val_score(lin_reg, housing, housing_labels,
                             scoring="neg_root_mean_squared_error", cv=10)
pd.Series(lin_rmses).describe()
```

```
count       10.000000
mean     69858.018195
std       4182.205077
min      65397.780144
25%      68070.536263
50%      68619.737842
75%      69810.076342
max      80959.348171
dtype: float64
```

k-fold (***k*** = 10)
cross-validation

California Housing project

ML Algorithm: Decision Tree

```
from sklearn.model_selection import cross_val_score
```

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```

```
tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,
                              scoring="neg_root_mean_squared_error", cv=10)
```

```
pd.Series(tree_rmses).describe()
```

```
count       10.000000
mean     66868.027288
std       2060.966425
min      63649.536493
25%      65338.078316
50%      66801.953094
75%      68229.934454
max      70094.778246
dtype: float64
```

k-fold (***k*** = 10) cross-validation

California Housing project

ML Algorithm: Random Forest

```
from sklearn.model_selection import cross_val_score
```

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                scoring="neg_root_mean_squared_error", cv=10)
```

```
pd.Series(forest_rmses).describe()
```

```
count       10.000000
mean     47019.561281
std       1033.957120
min      45458.112527
25%      46464.031184
50%      46967.596354
75%      47325.694987
max      49243.765795
dtype: float64
```

6.
Fine-Tune your Model

6. Fine Tune your Model
a. Grid Search

- Instead of experimenting manually (!), use Scikit-Learn’s **`GridSearchCV`**
- Specify:
  - which hyperparameters to experiment with
  - what values to try out for each

  and it will use cross-validation to evaluate all possible combinations of hyperparameter values

Grid Search

California Housing project

ML Algorithm: Random Forest

`bootstrap`, **`n_estimators`** and **`max_features`** are hyperparameters of the model

Highlighted is the identified best combination of values

```
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',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

```
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

6. Fine Tune your Model
b. Randomised Search

- Better than Grid Search when the hyperparameter space is too big (too many combinations)
- Evaluates a fixed number of combinations (not all) selecting a random value for each parameter in each iteration
- For each hyperparameter you specify:
  - either a list of possible values
  - or a distribution

6. Fine Tune your Model
c. Analysing the Best Models and their Errors

- For each model, there are ways to identify which features are more “important”
  - This can give you an indication about features that can/should be dropped
- For each model, identify and look into the specific errors that it makes. Try to:
  - understand why
  - fix it (e.g. add a feature, drop a feature, etc.)

6. Fine Tune your Model

d. Evaluate your System on the Test Set

- After all of this (!), you know have a (best) model that performs sufficiently well

- Nothing new here
  - Apply the same preprocessing, and
  - Run your best model on the Test Set

6. Fine Tune your Model

d. Evaluate your System on the Test Set

- Performance will most likely be worse than what you got during cross-validation
  - Especially if you did a lot of hyperparameter tuning

    ⇒ your model was fine-tuned to perform well on the validation data
- This is normal
  - Resist the urge to go back and fiddle with the hyperparameters more in order to reduce the generalisation error

7.
Launch, Monitor, Maintain

7. Launch, Monitor, Maintain
a. Launch

- Technicalities on deploying are outside the scope of this module
- Use the `joblib` library to save
- When deployment time comes, check your options
  - Embedding in website
  - Web service that your website queries through a REST API
  - Cloud (e.g. Google's Vertex AI)

7. Launch, Monitor, Maintain
b. Monitor

- Live performance of the deployed system needs to be monitored frequently
  - **Model rot**: without retraining, your system may become outdated for the current data
- Two common ways:
  - Inferred from **calculated metrics** (e.g. number of recommended products sold)
  - **Human intervention**; use them to rate your system’s results
    - Experts
    - The users themselves

7. Launch, Monitor, Maintain
c. Maintain

If data keeps evolving:

- **Update datasets**

  ⇒ Collect fresh, labeled data (you may use the human raters to label it)

- **Retrain**

  ⇒ Write a script to retrain and fine-tune automatically and frequently (e.g. every week)

  ⇒ Write a script to evaluate old and updated model; deploy new model if performance is not decreased

7. Launch, Monitor, Maintain
c. Maintain

Evaluate the quality of the new data as well!

- Maybe data is collected through a sensor that malfunctions
- You can have alerts for when
  - New data is missing feature values
  - The **mean** and/or **standard deviation** has a **discrepancy** with those of the training set

Thank you!

Coming up next:
Supervised Learning:
Intro Classification