

# End-to-End Machine Learning Project

## Get the data

```
In [16]: from pathlib import Path
import pandas as pd
import tarfile
import urllib.request
import matplotlib.pyplot as plt
import numpy as np
from zlib import crc32
```

### Download and load the housing data

```
In [2]: 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()

    return pd.read_csv(Path("datasets/housing/housing.csv"))
housing = load_housing_data()
```

### Take a Quick Look at the Data Structure

Looking at the first few rows of the dataset:

```
In [3]: housing.head()
```

```
Out[3]:   longitude  latitude  housing_median_age  total_rooms  total_bedrooms  population  households
          0      -122.23     37.88             41.0       880.0           129.0      322.0         126
          1      -122.22     37.86             21.0      7099.0           1106.0     2401.0        1138
          2      -122.24     37.85             52.0      1467.0           190.0      496.0         177
          3      -122.25     37.85             52.0      1274.0           235.0      558.0         219
          4      -122.25     37.85             52.0      1627.0           280.0      565.0         259
```

- Each row represents one district.
- There are 10 attributes: longitude, latitude, housing\_median\_age, total\_rooms, total\_bedrooms, population, households, median\_income, median\_house\_value, and ocean\_proximity.

Description of the dataset:

```
In [4]: 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
```

- There are 20,640 instances in the dataset.
- total\_bedrooms attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature (take care of this later).
- All attributes are numerical, except for ocean\_proximity (text and categorical). Find out how many categories exist:

In [5]: `housing["ocean_proximity"].value_counts()`

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

The `describe()` method shows a summary of the numerical attributes:

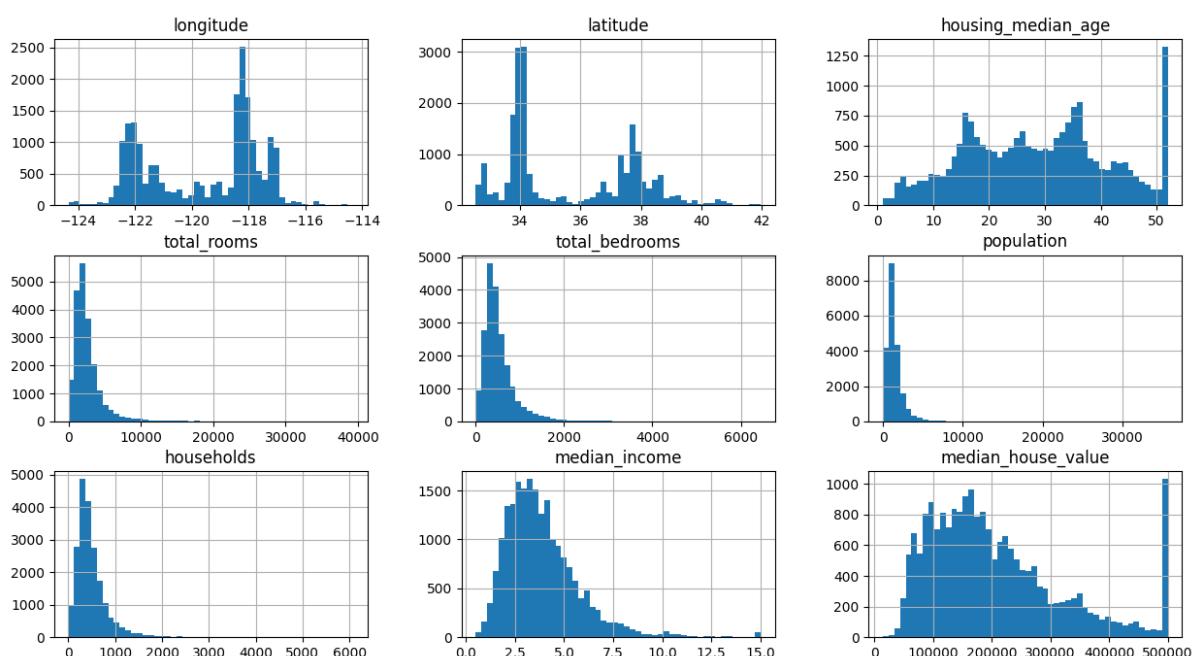
In [6]: `housing.describe()`

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	
<b>count</b>	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
<b>mean</b>	-119.569704	35.631861	28.639486	2635.763081	537.870553	1.000000
<b>std</b>	2.003532	2.135952	12.585558	2181.615252	421.385070	1.000000
<b>min</b>	-124.350000	32.540000	1.000000	2.000000	1.000000	1.000000
<b>25%</b>	-121.800000	33.930000	18.000000	1447.750000	296.000000	1.000000
<b>50%</b>	-118.490000	34.260000	29.000000	2127.000000	435.000000	1.000000
<b>75%</b>	-118.010000	37.710000	37.000000	3148.000000	647.000000	1.000000
<b>max</b>	-114.310000	41.950000	52.000000	39320.000000	6445.000000	356.000000

- The count, mean, min, and max rows are self-explanatory.
- Null values are ignored.
- The std row shows the standard deviation.
- The 25%, 50%, and 75% rows show the corresponding percentiles (the median is the 50th percentile).

**Histogram:** shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis).

```
In [7]: housing.hist(bins=50, figsize=(15,8))
plt.show()
```



- The median\_income attribute has been scaled and capped. The values represent tens of thousands of USD (e.g.,  $3 \approx \$30,000$ ), and the range is restricted between 0.5 (lower limit) and 15 (upper limit). This type of preprocessing is standard in Machine Learning.
- Both housing\_median\_age and the target variable, median\_house\_value, are capped (likely at \$500,000). This is risky because the model might learn that prices never exceed this limit. If the client requires accurate predictions for high-value homes, you must either collect the correct labels for the capped districts or remove these districts entirely from the dataset to avoid biasing the model.
- Many features exhibit a right-skewed distribution (long tail to the right), meaning they are not symmetrical. This asymmetry can make it difficult for Machine Learning algorithms to detect patterns. To improve performance, you will later apply transformations (like log scaling) to convert these distributions into more symmetrical, bell-shaped (Gaussian) forms.

## Create a Test Set

- You must set aside the test set immediately to avoid data snooping bias. Because the human brain is prone to overfitting, looking at the test data might subconsciously influence your choice of model based on incidental patterns in that set. This leads to overly optimistic error estimates and poor performance on new, real-world data.
- Creating a test set is theoretically simple; pick some instances randomly, typically 20% of the dataset:

```
In [14]: def shuffle_and_split_data(data, test_ratio):
    # Obtiene índices aleatorios para dividir el conjunto de datos
    shuffled_indices = np.random.permutation(len(data))
    # Calcula el tamaño del conjunto de prueba
    test_set_size = int(len(data) * test_ratio)
    # Divide los índices en conjuntos de entrenamiento y prueba
    test_indices = shuffled_indices[:test_set_size]
```

```
train_indices = shuffled_indices[test_set_size:]

# Devuelve los conjuntos de datos divididos
return data.iloc[train_indices], data.iloc[test_indices]

train_set, test_set = shuffle_and_split_data(housing, 0.2)
print(len(train_set))
print(len(test_set))
```

```
16512
4128
```

- Simple random splitting is problematic because generating a new test set every time leads to data leakage (the model eventually sees all data). Setting a random seed (np.random.seed(42)) fixes consistency for a static dataset but fails if the dataset is updated. The best solution is hash-based splitting: using a unique identifier for each instance to compute a hash. This ensures the test set remains stable and consistent across multiple runs, even when new data is added to the dataset. Possible implementation:

```
In [17]: def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32

def split_data_with_hash(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: is_id_in_test_set(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

- Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

```
In [18]: housing_with_id = housing.reset_index()
train_set, test_set = split_data_with_hash(housing_with_id, 0.2, "index")
print(len(train_set))
print(len(test_set))
```

```
16512
4128
```

- If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted.
- If this is not possible, then you can try to use the most stable features to build a unique identifier. For example:

```
In [19]: housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_data_with_hash(housing_with_id, 0.2, "id")
print(len(train_set))
print(len(test_set))
```

```
16322
4318
```

- Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is train\_test\_split(),
- There is a random\_state parameter that allows you to set the random generator seed.
- You can pass it multiple datasets with an identical number of rows, and it will split them on the same indices

```
In [20]: from sklearn.model_selection import train_test_split
```

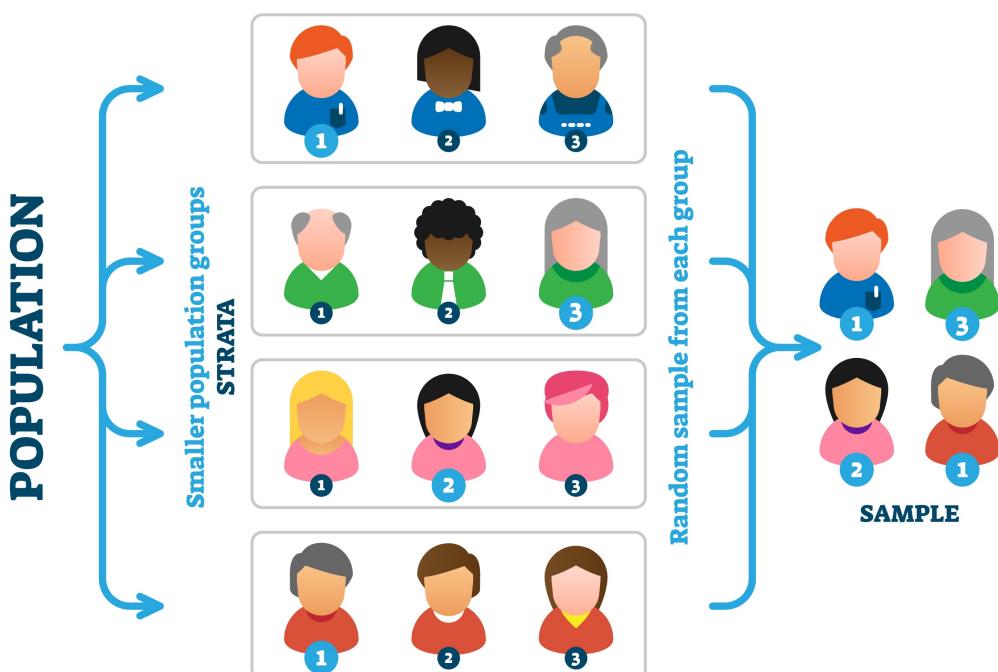
```
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
print(len(train_set))
print(len(test_set))
```

16512

4128

- Random sampling poses a risk of sampling bias, especially with smaller datasets. To ensure the test set is representative of the whole population, you should use stratified sampling. Since median\_income is a critical predictor, the text demonstrates how to bin this continuous variable into 5 discrete categories (strata) using pd.cut. This prepares the data to guarantee that both the training and test sets accurately reflect the overall income distribution.

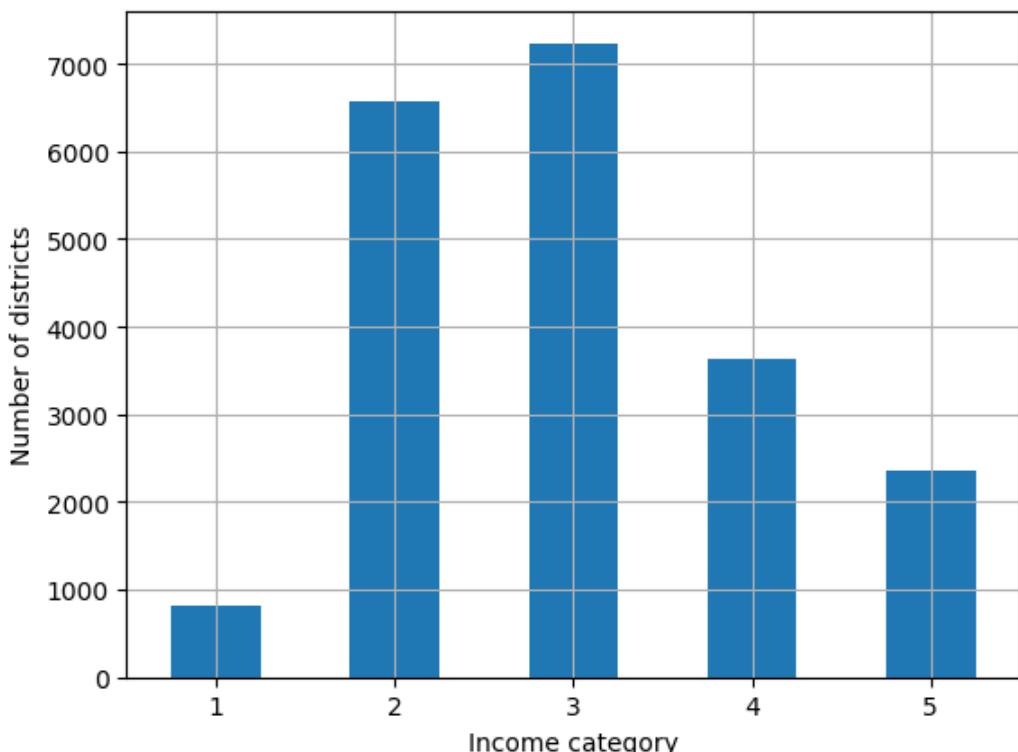
## STRATIFIED SAMPLING



- Uses the pd.cut() function to create an income category attribute with five categories (labeled from 1 to 5); category 1 ranges from 0 to 1.5 (i.e., less than \$15,000), category 2 from 1.5 to 3:

```
In [23]: 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")
plt.show()
```



- The text demonstrates two ways to implement stratified sampling using Scikit-Learn. The class StratifiedShuffleSplit allows generating multiple splits (yielding indices), which is useful for cross-validation. However, for a single split, it is more efficient to use the standard train\_test\_split() function with the stratify argument pointing to the income category. The final output verifies that the test set proportions match the overall population distribution.

```
In [28]: # With StratifiedShuffleSplit
from sklearn.model_selection import StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
strat_splits = []
for train_index, test_index in splitter.split(housing, housing["income_cat"]):
    strat_train_set_n = housing.loc[train_index]
    strat_test_set_n = housing.loc[test_index]
    strat_splits.append((strat_train_set_n, strat_test_set_n))

strat_train_set, strat_test_set = strat_splits[0]
print(strat_test_set["income_cat"].value_counts() / len(strat_test_set))

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

```
In [29]: # A shorter way with train_test_split
strat_train_set, strat_test_set = train_test_split(
    housing, test_size=0.2,
    stratify=housing["income_cat"],
    random_state=42
)
print(strat_test_set["income_cat"].value_counts() / len(strat_test_set))
```

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

- Validates the stratified sampling method by comparing it to purely random sampling. The results show that stratified sampling maintains the original dataset's income proportions almost perfectly, whereas random sampling introduces significant bias.

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

Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

- The temporary income\_cat column is removed (drop) from both sets to restore the data to its original state.

```
In [30]: for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

## Final code

```
In [33]: housing = pd.read_csv("./datasets/housing/housing.csv")

# Create income category attribute (that attribute will be used for 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]
)

# Stratified split using train_test_split
strat_train_set, strat_test_set = train_test_split(
    housing,
    test_size=0.2,
    stratify=housing["income_cat"],
    random_state=42
)

# Verify the proportions
print(strat_test_set["income_cat"].value_counts() / len(strat_test_set))

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

print(len(strat_train_set))
print(len(strat_test_set))
```

```
income_cat
3      0.350533
2      0.318798
4      0.176357
5      0.114341
1      0.039971
Name: count, dtype: float64
16512
4128
```