End-to-End Machine Learning Project

The Main steps of an end-to-end machine learning project:

1. Look at the big picture.
2. Get the data.
3. Explore 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

- Places you can look to get data:
  - Popular open data repositories:
    - OpenML.org
    - Kaggle.com
    - PapersWithCode.com
    - UC Irvine Machine Learning Repository
    - Amazon's AWS datasets
    - TensorFlow datasets
  - Meta portals (they list open data repositories):
    - DataPortals.org
    - OpenDataMonitor.eu
  - Other pages listing many popular open data repositories:
    - Wikipedia's list of machine learning datasets
    - Quora.com
    - The datasets subreddit

Look at the Big Picture

- Your 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 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). I will call them "districts" for short.
- Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

Frame the Problem

- The first question to ask is what the business objective is? Building a model is probably not the end goal.
- How does the company expect to use and benefit from this model?
- Knowing the objective is important because it will determine how you frame the problem, which algorithms you will select, which performance measure you will use to evaluate your model, and how much effort you will spend tweaking it.
- The next question to ask your boss is what the current solution looks like (if any). The current situation will often give you a reference for performance, as well as insights on how to solve the problem.
- With all this information, you are now ready to start designing your system.

    - Is it **supervised**, **unsupervised**, **semi-supervised**, **self-supervised**, or **reinforcement learning** task?
    - Is it a **classification** task, a **regression** task, or something else?
    - Should you use **batch learning** or **online learning** techniques?
- In our case it is clearly a typical **supervised** learning task, since the model can be trained with **labeled** examples
- It is a typical **regression** task since the model will be asked to predict a value.
- This is a **multiple regression** problem, since the system will use multiple features to make a prediction
- It is also a **univariate** regression problem since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a **multivariate** regression problem.
- There is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain **batch learning** should do just fine.
- If the data is huge, you could either split your **batch learning** across multiple servers(using **MapReduce** technique) or use an **online learning** technique.

Pipelines

- A sequence of data processing components is called a **data pipeline**.
- A piece of information fed to a Machine Learning system is called a **Signal**
- **Pipelines** are very common in machine learning systems, since there is a lot of data to manipulate and many data transformations to apply.
- **Components** typically run asynchronously. Each **component** pulls in a large amount of data, processes it, and spits out the result in another data store.
- Each **component** is fairly self-contained: the interface between components is simply the data store.
- if a **component** breaks down, the downstream **components** can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.

Select a performance Measure

- Your next step is to select a performance measure.
- A typical performance measure for regression problems is the **Root Mean Square Error(RMSE)**. It gives an idea of how much error the system typically makes in its predictions, with a higher weight for **large errors**.

Equation 2-1. Root mean square error (RMSE)

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

- **RMSE** is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function.
- Suppose your dataset has many outliers, in that case, you may consider using the **mean absolute error (MAE)**

Equation 2-2. Mean absolute error (MAE)

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

- **RMSE** corresponds to the **Euclidean norm**: this is the notion of distance we are all familiar with. It is also called the **ℓ_2 norm**,
- **MAE** corresponds to the **ℓ_1 norm,** called the **Manhattan norm.**

- The higher the **norm index**, the more it focuses on large values and neglects small ones. This is why the **RMSE** is more sensitive to outliers than the **MAE**.

Notations

- This equation introduces several very common machine learning notations :
  - **m** is the number of instances in the dataset you are measuring the **RMSE** on.
  - $\mathbf{x}^{(i)}$ is a vector of all the feature values (excluding the label) of the $\mathbf{i}^{th}$ instance in the dataset, and $\mathbf{y}^{(i)}$ is its label (the desired output value for that instance).
  - **X** is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the $\mathbf{i}^{th}$ row is equal to the transpose of $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$□.
  - **h** is your system's prediction function, also called a **hypothesis**. When your system is given an instance's feature vector $\mathbf{x}^{(i)}$, it outputs a predicted value $\hat{\mathbf{y}} = \mathbf{h}(\mathbf{x}^{(i)})$ for that instance ($\hat{\mathbf{y}}$ is pronounced **"y-hat"**).
  - **RMSE(X, h)** is the **cost function** measured on the set of examples using your **hypothesis h**.

Check the Assumptions

- It is good practice to list and verify the assumptions that have been made so far (by you or others), this can help you catch serious issues early on.

Get the Data

Create the Workspace

Download the Data

- In typical environments your data would be available in a relational database or some other common data store and spread across multiple tables/documents/files. To access it, you would first need to get your credentials and access authorizations□ and familiarize yourself with the data schema.
- In this project, however, things are much simpler: you will just download a single compressed file, housing.tgz, which contains a **comma separated values (CSV)** file called housing.csv with all the data.
- Rather than manually downloading and decompressing the data, it's usually preferable to write a function that does it for you.
- This is useful in particular if the data changes regularly: you can write a small script that uses the function to fetch the latest data (or you can set up a scheduled job to do that automatically at regular intervals).
- Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

Here is the function to fetch the data:[11]

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    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()
```

Once again you should write a small function to load the data:

```
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 at the Data Structure

- Take a look at the structure of the dataset using **head()** and **sample()** methods
- The **info()** method is useful to get a quick description of data, in particular the total no. of rows, each attribute's type, and the number of nonnull values.
- **Describe()** method shows a summary of the numerical attributes.
- Attribute of type **object** can 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.
- If we have a categorical attribute, **value_counts()** method can find out what categories exist and how many rows belong to each category
- Plotting a histogram for each numerical attribute using hist() method on the whole dataset.

```
import matplotlib.pyplot as plt

housing.hist(bins=50, figsize=(12, 8))
plt.show()
```

Create a Test Set

- You need to create a **test set** before diving into the data any further, put it aside, and never look at it .
- **Data leakage** is a scenario when ML model already has information of **test data** in **training data**, but this information would not be available at the time of prediction. It causes high performance while training set but perform poorly in deployment or production.
- Also, to avoid **data snooping bias** because your brain is an amazing pattern detection system, which means it is highly prone to **overfitting**, you may stumble upon some interesting pattern in test set that leads you to select a particular kind of ML model.
- How to create a test set?
  - Creating a test set is theoretically simple: picks some instances randomly, typically 20% of the dataset (or less if your dataset is very large) and set them aside.

1. Splitting your dataset using a defined function

```
import numpy as np

def shuffle_and_split_data(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]
```

You can then use this function like this:

```
>>> train_set, test_set = shuffle_and_split_data(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

> This way works, but it is not perfect, if you run the program again, it will generate a different test set!
>
> Overtime your ML algorithms will get to see the whole dataset, which is what you want to avoid.
>
> One solution is to save the test set on the first run or use **np.random.seed(42)**
>
> but both solutions will break next time you fetch an updated dataset.

2. To have a stable train/test set split even after updating the dataset, a common solution is to use each instance's identifier to decide whether or not it should go in the test set

> If your dataset doesn't have an identifier column, the simplest solution to use the row index as the ID, but you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted or use the most stable feature to build a unique identifier.

```python
from zlib import crc32

def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32

def split_data_with_id_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]
```

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

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

3. Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways, The simplest function is **train_test_split(),** it has **random_state** parameter, you can pass multiple datasets with an identical no. of rows, and it will split them on indices.

```python
from sklearn.model_selection import train_test_split

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

This is fine if your dataset is large enough(especially relative to the no. of attributes), but if not you risk introducing a **sampling bias.**

4. To solve the problem of **sampling bias** when splitting small datasets is to use **stratified sampling**, the population is divided into homogeneous subgroups called **strata**, and the right no. of instances are sampled from each **stratum** to guarantee that the **test set** is representative of the overall population.

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


from sklearn.model_selection import StratifiedShuffleSplit

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.iloc[train_index]
    strat_test_set_n = housing.iloc[test_index]
    strat_splits.append([strat_train_set_n, strat_test_set_n])

strat_train_set, strat_test_set = strat_splits[0]

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

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

1. You need to pick an important attribute to split on to make sure that the test set is representative of the various attributes in the whole dataset. Also, it is important to have a sufficient no. of instances in your dataset for each stratum, or else the estimate of a stratum's importance may be biased
2. You can use **StratifiedShuffleSplit** class, note that each splitter has a **split()** method that returns an iterator over different training/test splits of the same data, it returns indices not the data itself.
3. The code generates 10 different stratified splits of the same dataset, you can pick anyone.
4. since stratified sampling is fairly common, there's a shorter way to get a single split using the **train_test_split()** function with the **stratify** argument.
5. You should remove the attribute you split on, so the data is back to its original state.

Discover and Visualize the Data to Gain Insights

- Make sure you have put the **test set** aside and you are only exploring the **training set.**
- Also, if the training set is very large, you may want to sample an **exploration set**, to make manipulations easy and fast.
- Create a copy so that you can play with it without harming the training set.

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

Visualizing Geographical Data

- If your dataset contains geographical information(latitude and longitude), it is good idea to create a scatter plot of all districts to visualize the data
- Setting the **alpha** option makes it much easier to visualize the places where there is a high density of data points

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
alpha=0.2)
plt.show()
```

- You can also get the radius of each circle to represent the district's population (option **s**), and the color represents the price (option **c**), use a predefined color map (option **cmap**) called jet which ranges from blue (low values) to red (high prices).

```
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))
plt.show()
```

Looking for Correlations

- You can compute the **standard correlation coefficient** between every pair of attributes using the **corr()** method

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

```
>>> 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
```

- The correlation **coefficient** ranges from **-1** to **1**. When it is close to **1**, it means that there is a strong **positive correlation**. When the **coefficient** is close to **-1**, it means there is a strong **negative correlation**.
- **Coefficients** close to **0** mean that there is no linear correlation.
- The **correlation coefficient** only measures linear correlations, it may completely miss out on nonlinear relationships.
- You can also check for **correlation** between attributes is to use the pandas **scatter_matrix()** function, which plots every numerical attribute against every other numerical attribute.

Experimenting with Attribute Combinations

- You may want to try out various 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"]
```

- This round of exploration does not have to be absolutely thorough; the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype.

- But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

Prepare the Data for Machine Learning Algorithms

- When preparing the data for your ML algorithms, instead of doing it manually, you should write functions for this purpose:
  - This will allow you to reproduce these transformations on any dataset.
  - You will gradually build a library of transformation functions that you can reuse in future projects.
  - You can use these functions in your live system to transform the new data before feeding it to your algorithms.
  - This will make it possible for you to easily try various transformations and see which combination of transformations works best.
- Separate the predictors and labels since we don't necessarily want to apply the same transformations to the predictors and the target values

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Data Cleaning

- Most ML algorithms cannot work with missing features.
- How to deal with missing values?

  1. Get rid of the corresponding rows.

  2. Get rid of the whole attribute.

  3. Set the missing values to some value (zero, the mean, the median, etc.). This is called **imputation**.

- You can accomplish these easily using the Pandas DataFrame's **dropna(), drop(),** and **fillna()** methods.

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

housing.drop("total_bedrooms", axis=1)  # option 2

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

- For option 3, don't forget to save the median value that you have computed, you will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.
- Scikit-Learn provides **SimpleImputer** class. The benefit is that it will store the median value of each feature: this will make it possible to impute missing values not only on the training set, but also on the validation set, the test set, and any new data fed to the model.

```
from sklearn.impute import SimpleImputer

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

- Since the median can only be computed on numerical attributes, you then need to create a copy of the data with only the **numerical attributes.**

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

- Now you can fit the imputer instance to the training data using the **fit()** method:

```
imputer.fit(housing_num)
```

- The **imputer** has simply computed the median of each attribute and stored the result in its **statistics_** instance variable.

```
>>> imputer.statistics_
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
>>> housing_num.median().values
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
```

- Now you can use this "trained" imputer to transform the training set by replacing missing values with the learned medians.

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

- The result is a plain NumPy array containing the transformed features. If you want to put it back into a Pandas DataFrame, it's simple:

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

- Missing values can also be replaced with the mean value **(strategy="mean")**, or with the most frequent value **(strategy="most_frequent"),** or with a constant value **(strategy="constant", fill_value=…)**.
- The last two strategies support non-numerical data.
- **KNNImputer** replaces each missing value with the mean of the **k-nearestneighbors'** values for that feature. The distance is based on all the available features.
- **IterativeImputer** trains a regression model per feature to predict the missing values based on all the other available features. It then trains the model again on the updated data, and repeats the process several times, improving the models and the replacement values at each iteration.

Scikit-learn Design

- Scikit-Learn's API is remarkably well designed, These are the main design principles:
  - **Consistency**
    - All objects share a consistent and simple interface
    - **Estimators**
      - Any object that can estimate some parameters based on a dataset is called an **estimator**.
      - The estimation itself is performed by the **fit()** method.
      - Any other parameter needed to guide the estimation process is considered a **hyperparameter** (such as an **imputer's strategy**)
    - **Transformers**
      - Some estimators (such as an imputer) can also transform a dataset; these are called transformers.
      - The transformation is performed by the **transform()** method with the dataset, It returns the transformed dataset.
      - All transformers also have a convenience method called **fit_transform()** that is equivalent to calling **fit()** and then **transform()** (but sometimes **fit_transform()** is optimized and runs much faster).
    - **Predictors**
      - some estimators are capable of making predictions given a dataset; they are called **predictors**. For example, the **LinearRegression** model is a **predictor**

        - A **predictor** has a **predict()** method that takes a dataset of new instances and returns a dataset of corresponding predictions.
        - It also has a **score()** method that measures the quality of the predictions given in a test set.
    - **Inspection**
        - All the **estimator's hyperparameters** are accessible via public instance variables (e.g., **imputer.strategy**),
        - all the estimator's learned parameters are also accessible via public instance variables with an underscore suffix (e.g., **imputer.statistics_**).
    - **Nonproliferation of classes**
        - Datasets are represented as **NumPy arrays** or **SciPy sparse matrices**, instead of homemade classes.
        - **Hyperparameters** are just regular Python strings or numbers.
    - **Composition**
        - Existing building blocks are reused as much as possible.
        - For example, it is easy to create a Pipeline estimator from an arbitrary sequence of transformers followed by a final estimator.
    - **Sensible defaults**
        - Scikit-Learn provides reasonable default values for most parameters, making it easy to create a baseline working system quickly.

Handling Text and Categorical Attributes

- Most Machine Learning algorithms prefer to work with numbers, so how to convert categorical attributes from text to numbers.

1. Using **OrdinalEncoder** Class

```
from sklearn.preprocessing import OrdinalEncoder

ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:8]
array([[3.],
       [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.]])

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

    - The issue with **OrdinalEncoder** is that ML algorithms will assume that two nearby values are more similar than two distant values.
    - This may be fine in some cases (e.g., for ordered categories such as "bad", "average", "good", and "excellent").
2. To fix this issue, a common solution is to create one binary attribute per category, This is called **one-hot encoding**, because only one attribute will be equal to **1** (hot), while the others will be **0** (cold). The new attributes are sometimes called **dummy attributes**. Scikit-Learn provides a **OneHotEncoder** class to convert categorical values into one-hot vectors

```
>>> 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>
```

- A **sparse matrix** is a very efficient representation for matrices that contain mostly zeros. Indeed, internally it only stores the nonzero values and their positions.
- If you want to convert it to a (**dense**) NumPy array, just call the **toarray()** method

```
>>> 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.]])
```

- Alternatively, you can set **sparse=False** when creating the **OneHotEncoder**, in which case the transform() method will return a regular (**dense**) NumPy array directly.

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

- Pandas has a function called **get_dummies(),** which also converts each categorical feature into a one-hot representation, with one binary feature per category:

```
>>> df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]})
>>> pd.get_dummies(df_test)
   ocean_proximity_INLAND  ocean_proximity_NEAR BAY
0                       1                         0
1                       0                         1
```

- why not use **get_dummies()** instead of **OneHotEncoder**()?
  - the advantage of OneHotEncoder is that it remembers which categories it was trained on.
  - This is very important because once your model is in production, it should be fed exactly the same features as during training
  - Look what our trained **cat_encoder** outputs when we make it transform the same **df_test** (using **transform()**, not **fit_transform()**):

```
>>> cat_encoder.transform(df_test)
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

  - **get_dummies()** saw only two categories, so it output two columns, whereas **OneHotEncoder** output one column per learned category, in the right order.
  - If you feed **get_dummies()** a DataFrame containing an unknown category (e.g., "<2H OCEAN"), it will happily generate a column for it:

```
>>> df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN",
"ISLAND"]})
>>> pd.get_dummies(df_test_unknown)
   ocean_proximity_<2H OCEAN  ocean_proximity_ISLAND
0                          1                       0
1                          0                       1
```

- But **OneHotEncoder** is smarter: it will detect the unknown category and raise an exception. If you prefer, you can set the **handle_unknown** hyperparameter to **"ignore",** in which case it will just represent the unknown category with zeros:

```
>>> cat_encoder.handle_unknown = "ignore"
>>> cat_encoder.transform(df_test_unknown)
array([[0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

- If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features.
- This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories.
- For example, you could replace the **ocean_proximity** feature with the distance to the ocean (similarly, a **country code** could be replaced with the country's population and GDP per capita).
- Alternatively, you can use one of the encoders provided by the **category_encoders** package on GitHub. Or, when dealing with neural networks, you can replace each category with a learnable, low-dimensional vector called an **embedding**. This is an example of representation learning
- When you **fit** any Scikit-Learn **estimator** using a **DataFrame**, the **estimator** stores the column names in the **feature_names_in_** attribute.
- Scikit-Learn then ensures that any DataFrame fed to this estimator after that (e.g., to **transform()** or **predict()**) has the same column names.
- **Transformers** also provide a **get_feature_names_out()** method that you can use to build a DataFrame around the transformer's output:

```
>>> cat_encoder.feature_names_in_
array(['ocean_proximity'], dtype=object)
>>> cat_encoder.get_feature_names_out()
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',
       'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
       'ocean_proximity_NEAR OCEAN'], dtype=object)
>>> df_output = pd.DataFrame(cat_encoder.transform(df_test_unknown),
...                          columns=cat_encoder.get_feature_names_out(),
...                          index=df_test_unknown.index)
...
```

Feature Scaling and Transformation

- One of the most important transformations you need to apply to your data is **feature scaling**. With few exceptions, machine learning algorithms don't perform well when the input numerical attributes have very different scales.
- There are two common ways to get all attributes to have the same scale: **min-max scaling** and **standardization.**

1. **Min-max scaling** (**Normalization**) is the simplest: for each attribute, the values are shifted and rescaled so that they end up ranging from **0** to **1**.
   - This is performed by subtracting the **min value** and dividing by the **difference** between the **min** and the **max**
   - Scikit-Learn provides a **transformer** called **MinMaxScaler** for this.

- It has a **feature_range** hyperparameter that lets you change the range if, for some reason, you don't want **0–1** (e.g., **neural networks** work best with zero-mean inputs, so a range of **–1** to **1** is preferable).

2nd edition says neural networks expect an input of range between 0 and 1

```
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)
```

2. **Standardization** is different: first it **subtracts** the **mean** value (so **standardized** values have a **zero mean**), then it **divides** the result by the **standard deviation** (so **standardized** values have a **standard deviation** equal to **1**).
   - Unlike **min-max scaling**, **standardization** does not restrict values to a specific range. However, standardization is much less affected by **outliers**.
   - If you want to **scale** a **sparse matrix** without converting it to a **dense matrix** first, you can use a **StandardScaler** with its **with_mean** hyperparameter set to **False**: it will only **divide** the data by the **standard deviation**, without **subtracting** the **mean** (as this would break **sparsity**).
   - Scikit-Learn provides a **transformer** called **StandardScaler** for **standardization**:

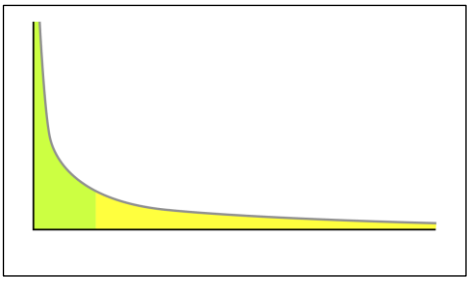
```
from sklearn.preprocessing import StandardScaler

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

- As with all **estimators**, it is important to **fit** the scalers to the **training data** only: never use **fit()** or **fit_transform()** for anything else than the **training set**.
- Once you have a trained scaler, you can then use it to **transform()** any other set, including the **validation**
- **set**, the **test set**, and new data.
- Note that while the **training set** values will always be scaled to the specified range, if new data contains outliers, these may end up scaled outside the range. If you want to avoid this, just set the **clip** hyperparameter to **True**.

Scaling Heavy Tailed Distributions

- When a feature's distribution has a **heavy tail** (i.e., when values far from the mean are not exponentially rare), both **min-max scaling** and **standardization** will **squash** most values into a small range.
- **Machine learning** models generally don't like this at all. So, before you **scale** the feature, you should first **transform** it to **shrink** the **heavy tail**, and if possible to make the distribution roughly **symmetrical**.
- For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its **square root** (or **raise** the feature to a **power** between **0** and **1**).
- If the **feature** has a really **long** and **heavy tail**, such as a
- **power law distribution**, then replacing the feature with its logarithm may help.



- For example, the **population feature** roughly follows a **power law**: districts with 10,000 inhabitants are only 10 times less frequent than districts with 1,000 inhabitants, not exponentially less frequent.
- Figure 2-17 shows how much better this **feature** looks when you compute its **log**: it's very close to a **Gaussian distribution** (i.e., **bell-shaped**).

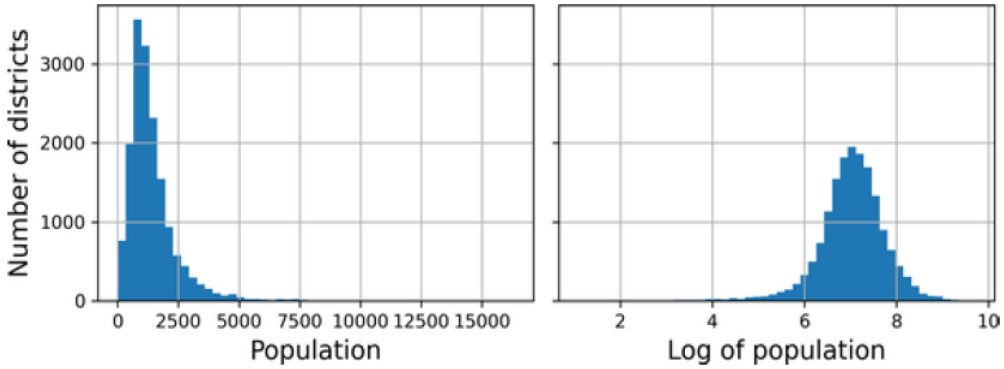
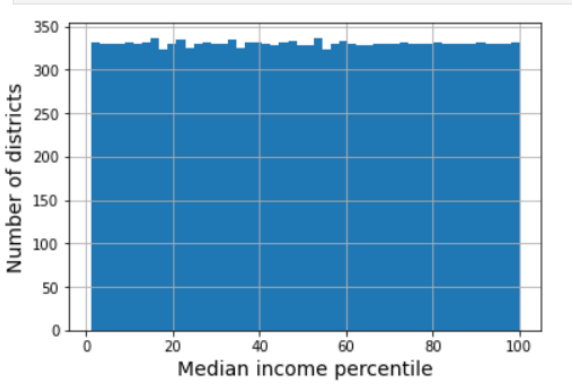


Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

- Another approach to **handle heavy-tailed features** consists in **bucketizing** the **feature**. This means **chopping** its **distribution** into roughly **equal-sized buckets**, and **replacing** each **feature** value with the **index** of the **bucket** it belongs to.
- you could replace each value with its **percentile**, **Bucketizing** with **equal-sized buckets** results in a feature with an almost **uniform distribution**, so there's no need for further scaling, or you can just **divide** by the number of buckets to force the values to the **0–1** range.



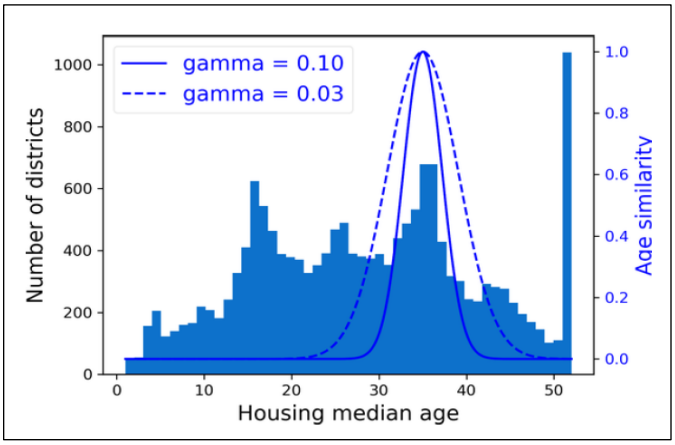
- When a feature has a **multimodal distribution** (i.e., with **two** or **more** clear **peaks**, called **modes**)It can also be helpful to bucketize it, but this time treating the **bucket IDs** as **categories**, rather than as numerical values.
- This means that the **bucket indices** must be **encoded**, for example using a **OneHotEncoder** (so you usually don't want to use too many buckets).
- This approach will allow the **regression model** to learn different rules more easily for different ranges of this feature value.
- For example, perhaps houses built around 35 years ago have a peculiar style that fell out of fashion, and therefore they're cheaper than their age alone would suggest.
- Another approach to transforming **multimodal distributions** is to add a **feature** for each of the **modes** (at least the main ones), representing the **similarity** between the **housing median age** and that particular **mode**.
- The **similarity measure** is typically computed using a **radial basis function (RBF)**—any function that depends only on the distance between the input value and a fixed point.
- The most commonly used **RBF** is the **Gaussian RBF**, whose **output** value **decays exponentially** as the input value moves away from the fixed point.
- For example, the **Gaussian RBF similarity** between the **housing age x** and **35** is given by the **equation $\exp(-\gamma(x - 35)^2)$**.
- The hyperparameter **γ (gamma)** determines how quickly the **similarity** measure decays as **x** moves away from **35**.

- Using Scikit-Learn's **rbf_kernel()** function, you can create a new **Gaussian RBF** feature measuring the **similarity** between the **housing median age** and **35**:

```
from sklearn.metrics.pairwise import rbf_kernel

age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

- Figure 2-18 shows this new feature as a **function** of the **housing median age** (solid line). It also shows what the feature would look like if you used a **smaller gamma** value.
- As the chart shows, the new age **similarity** feature peaks at **35**, right around the **spike** in the **housing median age** distribution: if this particular **age group** is well **correlated** with **lower prices**, there's a good chance that this new **feature** will help.



- The **target** values may also need to be transformed. if the target distribution has a **heavy tail**, you may choose to replace the target with its **logarithm**. But if you do, the **regression model** will now predict the **log** of the **median house** value, not the **median house** value itself.
- You will need to compute the **exponential** of the model's prediction if you want the predicted **median house value**.
- most of Scikit-Learn's **transformers** have an **inverse_transform()** method, making it easy to compute the **inverse** of their **transformations**.

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())

model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5]  # pretend this is new
data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

- Note that we convert the **labels** from a Pandas **Series** to a **DataFrame**, since the **StandardScaler** expects **2D** inputs.
- This works fine, but a simpler option is to use a **TransformedTargetRegressor**. We just need to construct it, giving it the **regression model** and the **label transformer**, then **fit** it on the **training set**, using the original **unscaled labels**.
- It will automatically use the **transformer** to scale the **labels** and train the **regression** model on the resulting **scaled labels**.
- Then, when we want to make a prediction, it will call the **regression model**'s **predict()** method and use the scaler's **inverse_transform()** method to produce the prediction:

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                   transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

Custom Transformers

- Although Scikit-Learn provides many useful **transformers**, you will need to write your own for tasks such as **custom transformations**, **cleanup operations**, or **combining specific attributes**.

1. For **transformations** that don't require any training, you can just write a function that takes a **NumPy array** as input and outputs the **transformed** array.
   - create a **log-transformer** and apply it to the population feature

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

   - The **inverse_func** argument is optional. It lets you specify an **inverse transform** function, e.g., if you plan to use your **transformer** in a **TransformedTargetRegressor**.
   - Your **transformation** function can take **hyperparameters** as additional arguments.
   - create a transformer that computes the same **Gaussian RBF similarity measure**

```
rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

   - Note that there's no **inverse function** for the **RBF kernel**, since there are always **two** values at a given distance from a fixed point (except at distance **0**).
   - Also note that **rbf_kernel()** does not treat the features **separately**. If you pass it an array with two features, it will measure the **2D distance** (**Euclidean**) to measure **similarity**.
   - Create a feature that will measure the **geographic similarity** between each **district** and **San Francisco**

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                     kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

   - **Custom transformers** are also useful to **combine** features.
   - Create a **FunctionTransformer** that computes the ratio between the input features **0** and **1**

```
>>> ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
>>> ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
array([[0.5 ],
       [0.75]])
```

2. **FunctionTransformer** is very handy, If you want your transformer to be trainable, learning some parameters in the **fit()** method and using them later in the **transform()** method

- You will want your **transformer** to work seamlessly with scikit-learn **functionalities**(such as **pipelines**), and since Scikit-Learn relies on **duck typing**, so this class does not have to inherit from any particular base class.
- All it needs is three methods: **fit()** (which must return self), **transform()**, and **fit_transform()**.
- You can get **fit_transform()** for free by simply adding **TransformerMixin** as a base class: the default implementation will just call fit() and then **transform().**
- If you add **BaseEstimator** as a base class (and avoid using ***args** and ****kwargs** in your constructor), you will also get two extra methods: **get_params()** and **set_params().** These will be useful for automatic **hyperparameter tuning**.
- Create custom transformer that acts much like the **StandardScaler**

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True):  # no *args or **kwargs!
        self.with_mean = with_mean

    def fit(self, X, y=None):  # y is required even though we don't use it
        X = check_array(X)  # checks that X is an array with finite float
values
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1]  # every estimator stores this in
fit()
        return self  # always return self!

    def transform(self, X):
        check_is_fitted(self)  # looks for learned attributes (with trailing
_)


        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

- The **sklearn.utils.validation** package contains several functions we can use to validate the inputs. Production code should have them.
- Scikit-Learn pipelines require the **fit()** method to have two arguments **X** and y**,** which is why we need the **y=None** argument even though we don't use **y**.
- All Scikit-Learn estimators set **n_features_in_** in the **fit()** method, and they ensure that the data passed to **transform()** or **predict()** has this number of features.
- The **fit()** method must return self.
- This implementation is not 100% complete: all **estimators** should set **feature_names_in_** in the **fit()** method when they are passed a DataFrame. Moreover, all **transformers** should provide a **get_feature_names_out()** method, as well as an **inverse_transform()** method when their transformation can be reversed.

3. A **custom transformer** can (and often does) use other **estimators** in its implementation.
   - The following code demonstrates **custom transformer** that uses a **KMeans clusterer** in the **fit()** method to identify the main clusters in the training data, and then uses **rbf_kernel()** in the **transform()** method to measure how similar each sample is to each cluster center

```
class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self  # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]


cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]],
                                           sample_weight=housing_labels)
```

- Figure 2-19 shows the 10 cluster centers found by k-means. The districts are colored according to their geographic similarity to their closest cluster center. As you can see, most clusters are located in highly populated and expensive areas.

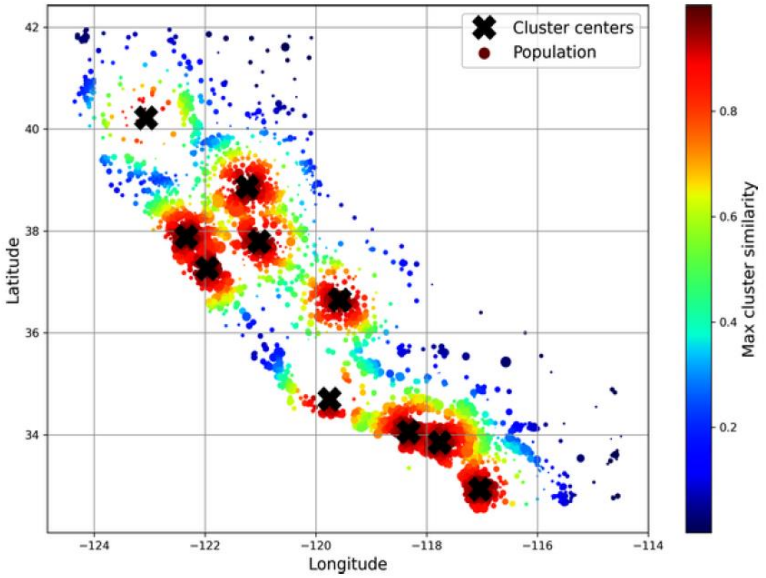


Figure 2-19. Gaussian RBF similarity to the nearest cluster center

- You can check whether your **custom estimator** respects Scikit-Learn's API by passing an instance to **check_estimator()** from the **sklearn.utils.estimator_checks package**. For the full API, check out https://scikit-learn.org/stable/developers.

Transformation Pipelines

- There are many data transformation steps that need to be executed in the right order.
- Fortunately, Scikit-Learn provides the **Pipeline class** to help with such sequences of **transformations**.
- Here is a small pipeline for numerical attributes, which will first **impute** then **scale** the input features

```
from sklearn.pipeline import Pipeline

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

- The **Pipeline** constructor takes a list of **name/estimator** pairs (**2-tuples**) defining a sequence of steps. The names can be anything you like, as long as they are unique and don't contain **double underscores** (__). They will be useful later when we discuss **hyperparameter tuning**.
- The **estimators** must all be **transformers** (i.e., they must have a **fit_transform()** method), except for the last one, which can be anything: a **transformer**, a **predictor**, or any other type of **estimator**.
- If you don't want to name the **transformers**, you can use the **make_pipeline()** function instead; it takes **transformers** as positional arguments and creates a **Pipeline** using the names of the **transformers' classes**, in **lowercase** and without underscores (e.g., "**simpleimputer**")
- If **multiple transformers** have the same name, an **index** is appended to their names (e.g., "foo-1", "foo-2", etc.).

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
StandardScaler())
```

- When you call the **pipeline's fit()** method, it calls **fit_transform()** sequentially on all the transformers, passing the output of each call as then parameter to the next call until it reaches the final estimator, for which it just calls the **fit()** method.
- The **pipeline** exposes the same methods as the final **estimator**. In this example the last estimator is a **StandardScaler**, which is a **transformer**, so the **pipeline** also acts like a **transformer**. If you call the pipeline's **transform()** method, it will sequentially apply all the transformations to the data.
- If the last estimator were a **predictor** instead of a **transformer**, then the **pipeline** would have a **predict()** method rather than a **transform()** method. Calling it would sequentially apply all the **transformations** to the data and pass the result to the predictor's **predict()** method.

```
>>> housing_num_prepared = num_pipeline.fit_transform(housing_num)
>>> housing_num_prepared[:2].round(2)
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

- if you want to recover a nice **DataFrame**, you can use the **pipeline's get_feature_names_out()** method

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

- **Pipelines** support **indexing**: for example, **pipeline[1]** returns the second estimator in the **pipeline**, and **pipeline[:-1]** returns a **Pipeline** object containing all but the last estimator.
- You can also access the **estimators** via the **steps** attribute, which is a list of **name/estimator** pairs, or via the **named_steps** dictionary attribute, which maps the names to the estimators.
- **num_pipeline["simpleimputer"]** returns the **estimator** named **"simpleimputer"**.
- So far, we have handled the **categorical** columns and the **numerical** columns separately. It would be more convenient to have a single **transformer** capable of handling all columns, applying the appropriate **transformations** to each column. For this, you can use a **ColumnTransformer**.
- The following **ColumnTransformer** will apply **num_pipeline** to the **numerical** attributes and **cat_pipeline** to the **categorical** attribute

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])
```

- First we **import** the **ColumnTransformer** class then we define the list of **numerical** and **categorical** column names and construct a simple **pipeline** for **categorical** attributes. Lastly, we construct a **ColumnTransformer**. Its constructor requires a list of triplets (**3-tuples**), each containing a name (which must be **unique** and not contain **double underscores**), a **transformer**, and a list of names (or indices) of **columns** that the **transformer** should be applied to.
- **Note** that the **OneHotEncoder** returns a **sparse matrix**, while the **num_pipeline** returns a **dense matrix**. When there is such a mix of **sparse** and **dense** matrices, the **ColumnTransformer** estimates the density of the final matrix (i.e., the ratio of non-zero cells), and it returns a **sparse matrix** if the density is lower than a given **threshold** (by default, **sparse_threshold=0.3**).
- **Tip**: Instead of using a **transformer**, you can specify the string "**drop**" if you want the columns to be dropped, or you can specify "**passthrough**" if you want the columns to be left untouched. By default, the remaining columns (i.e., the ones that were not listed) will be **dropped**, but you can set the **remainder** hyperparameter to any transformer (or to "**passthrough**") if you want these columns to be handled differently.
- Since listing all the column names is not very convenient, Scikit-Learn provides a **make_column_selector()** function that returns a **selector** function you can use to automatically select all the features of a given type, such as **numerical** or **categorical**. You can pass this **selector** function to the **ColumnTransformer** instead of column names or indices. Moreover, if you don't care about naming the transformers, you can use **make_column_transformer()**, which chooses the names for you, just like **make_pipeline()** does.
- For example, the following code creates the same **ColumnTransformer** as earlier, except the transformers are automatically named "**pipeline-1**" and "**pipeline-2**" instead of "**num**" and "**cat**":

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)

housing_prepared = preprocessing.fit_transform(housing)
```

- Great! We have a preprocessing pipeline that takes the entire training dataset and applies each **transformer** to the appropriate columns, then concatenates the transformed columns horizontally (**transformers** must never change the number of rows). Once again this **returns** a **NumPy** array, but you can get the column names using **preprocessing.get_feature_names_out()** and wrap the data in a nice **DataFrame** as we did before.
- **Note**: In a Jupyter notebook, if you import **sklearn** and run **sklearn**. **set_config(display="diagram")**, all Scikit-Learn **estimators** will be rendered as **interactive diagrams**. This is particularly useful for **visualizing pipelines**. To visualize num_pipeline, run a cell with **num_pipeline** as the last line. Clicking an **estimator** will show more details.

- Create a single **pipeline** that will perform all the **transformations** we have experimented with up to now.
  1. **Missing values** in **numerical** features will be **imputed** by replacing them with the **median**, as most ML algorithms don't expect missing values.
  2. In **categorical** features, missing values will be replaced by the **most frequent** category.
  3. The **categorical** feature will be **one-hot encoded**, as most ML algorithms only accept **numerical** inputs.
  4. A few ratio features will be computed and added: **bedrooms_ratio**, **rooms_per_house**, and **people_per_house**.
  5. A few **cluster similarity** features will also be added. These will likely be more useful to the model than latitude and longitude.
  6. **Features** with a **long tail** will be replaced by their **logarithm**, as most models prefer **features** with roughly **uniform** or **Gaussian** distributions.
  7. All **numerical** features will be **standardized**, as most ML algorithms prefer when all features have roughly the same scale.

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"]  # feature names out

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)

default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())
preprocessing = ColumnTransformer([
        ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
        ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
        ("people_per_house", ratio_pipeline(), ["population", "households"]),
        ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                               "households", "median_income"]),
        ("geo", cluster_simil, ["latitude", "longitude"]),
        ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
    ],
    remainder=default_num_pipeline)  # one column remaining:
housing_median_age
```

- If you run this ColumnTransformer, it performs all the transformations and outputs a NumPy array with 24 features:

```
>>> housing_prepared = preprocessing.fit_transform(housing)
>>> housing_prepared.shape
(16512, 24)
>>> preprocessing.get_feature_names_out()
array(['bedrooms__ratio', 'rooms_per_house__ratio',
       'people_per_house__ratio', 'log__total_bedrooms',
       'log__total_rooms', 'log__population', 'log__households',
       'log__median_income', 'geo__Cluster 0 similarity', [...],
       'geo__Cluster 9 similarity', 'cat__ocean_proximity_<1H OCEAN',
       'cat__ocean_proximity_INLAND', 'cat__ocean_proximity_ISLAND',
       'cat__ocean_proximity_NEAR BAY', 'cat__ocean_proximity_NEAR OCEAN',
       'remainder__housing_median_age'], dtype=object)
```

Select and Train a Model

- You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote a preprocessing pipeline to automatically clean up and prepare your data for machine learning algorithms. You are now ready to select and train a machine learning model.

Training and Evaluating on the Training Set

- Training a very basic Linear Regression model

```
from sklearn.linear_model import LinearRegression

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

- Look at the first five predictions and compare them to the labels

```
>>> housing_predictions = lin_reg.predict(housing)
>>> housing_predictions[:5].round(-2)  # -2 = rounded to the nearest hundred
array([243700., 372400., 128800.,  94400., 328300.])
>>> housing_labels.iloc[:5].values
array([458300., 483800., 101700.,  96100., 361800.])
```

- Measure this regression model's **RMSE** on the whole training set using Scikit-Learn's **mean_squared_error()** function, with the **squared argument** set to **False**:

```
>>> from sklearn.metrics import mean_squared_error
>>> lin_rmse = mean_squared_error(housing_labels, housing_predictions,
...                               squared=False)
...
>>> lin_rmse
68687.89176589991
```

- This is an example of a model **underfitting** the training data. When this happens, it can mean that the **features** do not provide enough information to make good predictions, or that the **model** is not powerful enough.
- The main ways to fix **underfitting** are to select a more powerful **model**, to feed the training algorithm with better **features**, or to reduce the **constraints** on the model.
- This model is not **regularized**, which rules out the last option. You could try to add more features, but first you want to try a more **complex model** to see how it does.
- You decide to try a **DecisionTreeRegressor**, as this is a fairly powerful model capable of finding complex nonlinear relationships in the data:

```
from sklearn.tree import DecisionTreeRegressor

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

```
>>> housing_predictions = tree_reg.predict(housing)
>>> tree_rmse = mean_squared_error(housing_labels, housing_predictions,
...                                squared=False)
...
>>> tree_rmse
0.0
```

- No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has **badly overfit** the data.
- you don't want to touch the **test set** until you are ready to launch a model you are confident about, so you need to use part of the **training set** for training and part of it for **model validation**.

Better Evaluation Using Cross-Validation

- You can use Scikit-Learn's **k-fold cross-validation** feature to better evaluate your models.
- It randomly splits the **training set** into **10** nonoverlapping subsets called **folds**, then it trains and evaluates the decision tree model **10** times, picking a different **fold** for evaluation every time and using the other **9** folds for **training**. The result is an array containing the **10** evaluation **scores**:

```
from sklearn.model_selection import cross_val_score

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

- **WARNING:** Scikit-Learn's **cross-validation** features expect a **utility** function (**greater** is better) rather than a **cost** function (**lower** is better), so the scoring function is actually the opposite of the **RMSE**. It's a **negative** value, so you need to switch the sign of the output to get the **RMSE** scores.

```
>>> 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
```

- The **decision tree** doesn't look as good as it did earlier. it seems to perform almost as poorly as the **linear regression** model!
- Notice that **cross-validation** allows you to get not only an estimate of the performance of your model, but also a measure of how **precise** this estimate is (i.e., its standard deviation).
- **cross-validation** comes at the cost of training the model several times, so it is not always **feasible**.
- We know there's an **overfitting** problem because the **training error** is low (actually zero) while the **validation error** is high.
- Create a **RandomForestRegressor** model, **Random forests** work by training many **decision trees** on random subsets of the features, then averaging out their predictions.
- Such models composed of many other models are called **ensembles**: they are capable of boosting the performance of the underlying model (in this case, **decision trees**).

```
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)
```

- **Random forests** look very promising for this task! However, if you train a **RandomForest** and measure the **RMSE** on the training set is **17,474**: that's much lower, meaning that there's still overfitting.
- Possible solutions are to **simplify** the **model**, **constrain** it (i.e., **regularize** it), or get a lot more **training data**.
- you should try out many other models from various categories of ML algorithms, without spending too much time tweaking **hyperparameters**. The goal is to shortlist a few (two to five)**promising models**.

```
>>> 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
```

Fine-Tune Your Model

- Assume that you now have a shortlist of promising **models**. You now need to **fine-tune** them.

Grid Search

- One option would be to fiddle with the **hyperparameters** manually, until you find a great combination of **hyperparameter** values. This would be very tedious work, and you may not have time to explore many combinations.
- Instead, you can use Scikit-Learn's **GridSearchCV** class to search for you. All you need to do is tell it which **hyperparameters** you want it to experiment with and what values to try out, and it will use **cross-validation** to evaluate all the possible combinations of **hyperparameter** values.

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [


    {'preprocessing__geo__n_clusters': [5, 8, 10],
     'random_forest__max_features': [4, 6, 8]},
    {'preprocessing__geo__n_clusters': [10, 15],
     'random_forest__max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

- Notice that you can refer to any **hyperparameter** of any estimator in a pipeline, even if this estimator is nested deep inside several **pipelines** and **column transformers**.
- For example, when Scikit-Learn sees "**preprocessing__geo__n_clusters**", it splits this string at the **double underscores**, then it looks for an **estimator** named "**preprocessing**" in the **pipeline** and finds the **preprocessing ColumnTransformer**. Next, it looks for a **transformer** named "**geo**" inside this **ColumnTransformer** and finds the **ClusterSimilarity transformer** we used on the **latitude** and **longitude attributes**. Then it finds this **transformer's n_clusters hyperparameter**.
- Similarly, **random_forest__max_features** refers to the **max_features hyperparameter** of the **estimator** named "**random_forest**", which is of course the **RandomForest** model
- **TIP:** Wrapping **preprocessing** steps in a Scikit-Learn **pipeline** allows you to tune the **preprocessing hyperparameters** along with the **model hyperparameters**. This is a good thing since they often interact.
- For example, perhaps increasing **n_clusters** requires increasing **max_features** as well. If fitting the **pipeline transformers** is computationally expensive, you can set the **pipeline's memory hyperparameter** to the path of a **caching** directory: when you first **fit** the **pipeline**, Scikit-Learn will save the fitted **transformers** to this directory. If you then **fit** the **pipeline** again with the same **hyperparameters**, Scikit- Learn will just load the **cached transformers**.
- There are **two** dictionaries in this **param_grid**, so **GridSearchCV** will first evaluate all **3 × 3 = 9** combinations of **n_clusters** and **max_features hyperparameter** values specified in the **first** dict, then it will try all **2 × 3 = 6** combinations of **hyperparameter** values in the **second** dict. So, in total the grid search will explore **9 + 6 = 15** combinations of **hyperparameter** values,

- Also, it will train the **pipeline 3** times per combination since we are using **3- fold cross validation.** This means there will be a grand total of **15 × 3 = 45** rounds of training! It may take a while, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'preprocessing__geo__n_clusters': 15, 'random_forest__max_features': 6}
```

- You can access the best **estimator** using **grid_search.best_estimator_,** If **GridSearchCV** is initialized with **refit=True** (which is the default), then once it finds the best **estimator** using **cross-validation**, it retrains it on the whole training set. This is usually a good idea, since feeding it more **data** will likely improve its **performance**.
- The evaluation **scores** are available using **grid_search.cv_results_**. This is a dictionary, but if you wrap it in a **DataFrame** you get a nice list of all the **test scores** for each combination of hyperparameters and for each crossvalidation split, as well as the mean test score across all splits:

```
>>> cv_res = pd.DataFrame(grid_search.cv_results_)
>>> cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)
>>> [...]  # change column names to fit on this page, and show rmse = -score
>>> cv_res.head()  # note: the 1st column is the row ID
   n_clusters max_features  split0  split1  split2  mean_test_rmse
12         15            6   43460   43919   44748           44042
13         15            8   44132   44075   45010           44406
14         15           10   44374   44286   45316           44659
```

- The **mean test RMSE** score for the best model is **44,042**, which is better than the score you got earlier using the default **hyperparameter** values (which was **47,019**). Congratulations, you have successfully **fine-tuned** your best model!

Randomized Search

- The **grid search** approach is fine when you are exploring relatively **few** combinations.
- **RandomizedSearchCV** is often preferable, especially when the **hyperparameter** search space is **large**.
- This class can be used in much the same way as the **GridSearchCV** class, but instead of trying out all possible combinations it evaluates a fixed number of combinations, selecting a random value for each hyperparameter at every iteration.
- This approach has several benefits: If some of your **hyperparameters** are **continuous** (or **discrete** but with many possible values), and you let **randomized search** run for, say, **1,000** iterations, then it will explore **1,000** different values for each of these **hyperparameters**, whereas **grid search** would only explore the few values you listed for each one.
- Suppose a **hyperparameter** does not actually make much difference, but you don't know it yet. If it has 10 possible values and you add it to your **grid search**, then training will take **10** times longer. But if you add it to a **random search**, it will not make any difference.
- If there are **6 hyperparameters** to explore, each with **10** possible values, then **grid search** offers no other choice than training the model a **million times**, whereas **random search** can always run for any number of iterations you choose. For each **hyperparameter**, you must provide either a **list** of possible values, or a **probability** distribution:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_distribs = {'preprocessing__geo__n_clusters': randint(low=3, high=50),
                  'random_forest__max_features': randint(low=2, high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distribs, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

- Scikit-Learn also has **HalvingRandomSearchCV** and **HalvingGridSearchCV** hyperparameter search classes. Their goal is to use the computational resources more efficiently, either to train faster or to explore a larger hyperparameter space.
- Here's how they work: in the first round, many **hyperparameter** combinations (called "**candidates**") are generated using either the **grid** approach or the **random** approach. These **candidates** are then used to **train models** that are evaluated using **crossvalidation**,
- However, training uses **limited resources**, which speeds up this first round considerably. By default, "**limited resources**" means that the models are trained on a small part of the **training set**. However, other limitations are possible, such as **reducing** the number of **training iterations** if the model has a **hyperparameter** to set it. Once every **candidate** has been evaluated, only the best ones go on to the **second** round, where they are allowed more resources to compete. After several rounds, the final candidates are evaluated using full resources. This may save you some time **tuning hyperparameters**.

Ensemble Methods

- Another way to **fine-tune** your system is to try to combine the models that perform best.
- The group (or "**ensemble**") will often perform better than the best individual model—just like random forests perform better than the individual decision trees they rely on—especially if the individual models make very different types of errors.
- For example, you could train and **finetune** a **k-nearest neighbors** model, then create an **ensemble** model that just predicts the **mean** of the **random forest** prediction and that model's prediction.

Analyze the Best Models and Their Errors

- You will often gain good insights on the problem by inspecting the **best models**.
- For example, the **RandomForestRegressor** can indicate the relative importance of each attribute for making accurate predictions:

```
>>> final_model = rnd_search.best_estimator_  # includes preprocessing
>>> feature_importances = final_model["random_forest"].feature_importances_
>>> feature_importances.round(2)
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0.01])
```

- Sort these importance scores in descending order and display them next to their corresponding attribute names:

```
>>> sorted(zip(feature_importances,
...            final_model["preprocessing"].get_feature_names_out()),
...            reverse=True)
...
[(0.18694559869103852, 'log__median_income'),
 (0.0748194905715524, 'cat__ocean_proximity_INLAND'),
 (0.06926417748515576, 'bedrooms__ratio'),
 (0.05446998753775219, 'rooms_per_house__ratio'),
 (0.05262301809680712, 'people_per_house__ratio'),
 (0.03819415873915732, 'geo__Cluster 0 similarity'),
 [...]
 (0.00015061247730531558, 'cat__ocean_proximity_NEAR BAY'),
 (7.301686597099842e-05, 'cat__ocean_proximity_ISLAND')]
```

- With this information, you may want to try dropping some of the less useful features (e.g., apparently only one **ocean_proximity** category is really useful, so you could try dropping the others).
- You should also look at the specific **errors** that your system makes, then try to understand why it makes them and what could fix the problem
- Adding extra **features** or getting rid of **uninformative** ones, cleaning up **outliers**, etc.
- Now is also a good time to ensure that your **model** not only works well on average, but also on all categories of districts, whether they're rural or urban, rich, or poor, northern, or southern, minority or not, etc.
- Creating subsets of your **validation set** for each category takes a bit of work, but it's important: if your **model** performs poorly on a whole category of districts, then it should probably not be deployed until the issue is solved, or at least it should not be used to make predictions for that category, as it may do more harm than good.
- **TIP:** The **sklearn.feature_selection.SelectFromModel** transformer can automatically drop the least useful features for you: when you fit it, it trains a model (typically a random forest), looks at its **feature_importances_** attribute, and selects the most useful features. Then when you call **transform()**, it drops the other features.

Evaluate Your System on the Test Set

- After **tweaking** your models for a while, you eventually have a system that performs sufficiently well.
- You are ready to **evaluate** the final **model** on the **test set**.
- There is nothing special about this process; just get the **predictors** and the **labels** from your **test set** and run your **final_model** to **transform** the data and make predictions, then evaluate these predictions:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions, squared=False)
print(final_rmse)  # prints 41424.40026462184
```

- In some cases, such a point estimate of the **generalization error** will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production?
- You might want to have an idea of how precise this estimate is. For this, you can compute a **95% confidence interval** for the **generalization error** using **scipy.stats.t.interval()**.

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
...                          loc=squared_errors.mean(),
...                          scale=stats.sem(squared_errors)))
...
array([39275.40861216, 43467.27680583])
```

- If you did a lot of **hyperparameter tuning**, the performance will usually be slightly worse than what you measured using **cross-validation**.
- That's because your system ends up **fine-tuned** to perform well on the **validation** data and will likely not perform as well on unknown datasets.
- when this happens, you must resist the temptation to **tweak** the **hyperparameters** to make the numbers look good on the **test set**; the improvements would be unlikely to generalize to new data.
- Now comes the project **prelaunch** phase: you need to present your solution (highlighting what you have learned, what worked and what did not, what assumptions were made, and what your system's limitations are), document everything, and create nice presentations with clear visualizations and easy-to-remember statements (e.g., "the median income is the number one predictor of housing prices").

Launch, Monitor ,and Maintain Your System

- now need to get your solution ready for **production** (e.g., polish the code, write documentation and tests, and so on).
- Then you can **deploy** your model to your production environment.
- One way to do this is to save the trained Scikit-Learn model, including the full preprocessing and production pipeline, then load this trained model within your production environment and use it to make predictions by calling its **predict()** method.

```
from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

- To save the model, you can use the **joblib** library like this:

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")
```

- Once your model is transferred to production, you can load it and use it. For this you must first import any custom classes and functions the model relies on (which means transferring the code to production), then load the model using **joblib** and use it to make predictions:

```
import joblib
[...]  # import KMeans, BaseEstimator, TransformerMixin, rbf_kernel, etc.

def column_ratio(X): [...]
def ratio_name(function_transformer, feature_names_in): [...]
class ClusterSimilarity(BaseEstimator, TransformerMixin): [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = [...]  # some new districts to make predictions for
predictions = final_model_reloaded.predict(new_data)
```

- **TIP**: You should **save** every **model** you experiment with, so you can come back easily to any **model** you want. Make sure you save both the **hyperparameters** and the **trained parameters**, as well as the **cross-validation** scores and perhaps the actual **predictions** as well.
- **TIP:** This will allow you to easily **compare scores** across model types and compare the **types** of **errors** they make. You can easily save Scikit-Learn models by using **Python's pickle** module, or using **sklearn.externals.joblib**, which is more efficient at serializing large NumPy arrays

- For example, perhaps the **model** will be used within a **website**: the user will type in some data about a new district and click the Estimate Price button.
- This will send a **query** containing the data to the **web server**, which will forward it to your web application, and finally your code will simply call the model's **predict()** method (you want to load the model upon server startup, rather than every time the model is used).
- Alternatively, you can wrap the model within a dedicated web service that your web application can query through a **REST API**□.
- This makes it easier to upgrade your model to new versions without interrupting the main application.
- It also simplifies scaling since you can start as many web services as needed and load-balance the requests coming from your web application across these web services. Moreover, it allows your web application to use any programming language, not just Python.

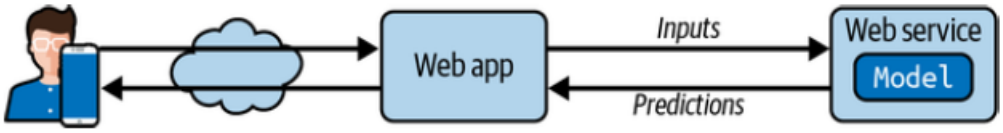


Figure 2-20. A model deployed as a web service and used by a web application

- Another popular strategy is to deploy your model to the cloud, for example on **Google's Vertex AI** (formerly known as Google Cloud AI Platform and Google Cloud ML Engine): just save your model using **joblib** and upload it to **Google Cloud Storage (GCS),** then head over to **Vertex AI** and create a new model version, pointing it to the **GCS** file.
- This gives you a simple web service that takes care of load balancing and scaling for you. It takes **JSON** requests containing the **input** data (e.g., of a district) and returns **JSON** responses containing the **predictions**.
- You can then use this web service in your website (or whatever production environment you are using).
- **Deployment** is not the end of the story. You also need to write **monitoring** code to check your system's live **performance** at regular intervals and trigger alerts when it drops.
- It may drop very quickly, for example if a component breaks in your infrastructure, but be aware that it could also decay very slowly, which can easily go unnoticed for a long time.
- This is quite common because of model **rot**: if the model was trained with last year's data, it may not be adapted to today's data.
- How to **monitor** your model's live **performance**?
- In some cases, the model's **performance** can be **inferred** from downstream metrics.
- For example, if your model is part of a **recommender system** and it **suggests** products that the users may be interested in, then it's easy to **monitor** the number of **recommended** products sold each day. If this number drops (compared to non-recommended products), then the prime suspect is the model.
- This may be because the **data pipeline** is broken, or perhaps the **model** needs to be retrained on fresh data (as we will discuss shortly).
- However, you may also need **human analysis** to assess the model's **performance**.
- For example, suppose you trained an **image classification** model to detect various product **defects** on a production line. How can you get an alert if the model's **performance** drops before thousands of **defective** products get shipped to your clients?
- One solution is to send to **human raters** a sample of all the pictures that the model classified (especially pictures that the model wasn't so sure about). Depending on the task, the raters may be experts, non-specialists or users via surveys or repurposed captchas.
- Also, you need to define all the **relevant processes** to do in case of failures and how to prepare for them.
- You should **automate** the whole process of **updating** datasets and **retraining** your model regularly:
  - **Collect** fresh data regularly and label it (e.g., using human raters).

    - Write a script to **train** the **model** and **fine-tune** the **hyperparameters** automatically. This script could run automatically, for example every day or every week, depending on your needs.
    - **Write** another script that will **evaluate** both the **new** model and the **previous** model on the updated **test set** and **deploy** the model to production if the **performance** has not decreased (if it did, make sure you investigate why).
    - The script should probably **test** the **performance** of your model on various **subsets** of the **test set**, such as poor or rich districts, rural or urban districts, etc.
- You should also make sure you **evaluate** the model's **input** data **quality**. Sometimes performance will degrade slightly because of a **poor-quality signal** (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale), but it may take a while before your system's performance degrades enough to trigger an alert.
- If you **monitor** your model's inputs, you may catch this earlier. For example, you could trigger an alert if more and more inputs are missing a **feature**, or the mean or **standard deviation** drifts too far from the **training set**, or a **categorical feature** starts containing new categories.
- Finally, make sure you keep **backups** of every **model** you create and have the process and tools in place to roll back to a previous model quickly in case the new model starts failing badly for some reason.
- Having **backups** also makes it possible to easily compare new models with previous ones.
- Similarly, you should keep **backups** of every version of your **datasets** so that you can roll back to a previous dataset if the new one ever gets corrupted(e.g., if the fresh data that gets added to it turns out to be full of outliers).
- Having **backups** of your **datasets** also allows you to **evaluate** any model against any previous **dataset**.