2025-02-27 11:53

tags :

Machine Learning
Hands on ML - book

Chapter 2 - Hands on

- This chapter guides you through an end-to-end machine learning project by simulating the role of a newly hired data scientist at a real estate company. Here are the main steps we will walk through:
    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

- 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

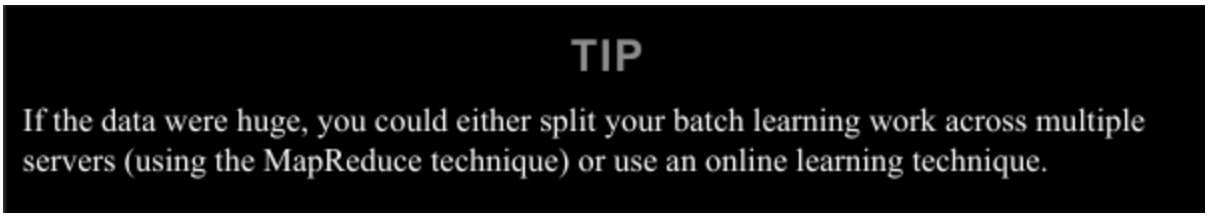
Look at the Big Picture

- Your first task is to use California census data to build a model of housing prices in the state.
- The dataset contains metrics like population, median income, and median housing price for each block group in California (referred to as "districts"). The goal is to train a model that can predict the median housing price of a district based on these features.

Frame the Problem

- Understanding the business objective is crucial, as building a model is not the final goal. Knowing how the company plans to use the model helps define the problem, select algorithms, choose evaluation metrics, and determine optimization efforts.
- The predicted median housing price from your model will be used as an input for another machine learning system, along with other signals. This downstream system will analyze investment opportunities, making accurate predictions essential for maximizing revenue.
- The current solution relies on experts who manually estimate district housing prices by gathering data and applying complex rules when exact prices are unavailable. This provides a reference for evaluating model performance.
- The current manual estimation process is costly, time-consuming, and often inaccurate, with errors exceeding 30%. To improve accuracy and efficiency, the company aims to train a model using census data, which includes median housing prices and other relevant district metrics.

- **PIPELINES**
  - A data pipeline is a sequence of processing components used to handle large amounts of data and apply multiple transformations, making it essential in machine learning systems.
  - Data pipeline components run asynchronously, processing large datasets and storing results for the next component. Each component operates independently, ensuring modularity and efficiency.
  - Components in a data pipeline interact through a data store, simplifying the system and allowing teams to focus on individual parts. This design enhances robustness, as downstream components can continue running using previous outputs if one component fails.
  - Without proper monitoring, a broken component may go unnoticed, leading to stale data and a decline in the system's overall performance.

- **now you are ready to design the system**
  - The housing price prediction task is a **supervised learning** problem since it uses labeled data. It is a **regression task**, specifically**multiple regression**, as it relies on multiple features (e.g., population, median income). Since it predicts a single value per district, it is **univariate regression**. Given that the data is static and fits in memory, **batch learning** is suitable rather than online learning.
  - 

Select a Performance Measure

- The **Root Mean Square Error (RMSE)** is a common regression metric that measures the typical prediction error, giving higher weight to large errors.

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

- **NOTATIONS**
  - m is the number of instances in the dataset you are measuring the RMSE on.
  - $x^{(i)}$ represents the feature values of an instance, while $y^{(i)}$ is the corresponding label or target value of the same instance.

- For example, if the first district in the dataset is located at longitude –118.29°, latitude 33.91°, and it has 1,416 inhabitants with a median income of $38,372, and the median house value is $156,400 (ignoring other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1{,}416 \\ 38{,}372 \end{pmatrix}$$

and:

$$y^{(1)} = 156{,}400$$

  - X is a matrix of feature values for all instances, with each row representing an instance and corresponding to the transpose of $x^{(i)}$
  - h is the prediction function (hypothesis) that outputs $\hat{y} = h(x)$, the predicted value for a given feature
  - $RMSE(X, h)$ is the cost function measured on the set of examples using your hypothesis h .
- While **Root Mean Squared Error (RMSE)** is commonly used for regression tasks, **Mean Absolute Error (MAE)** may be preferable in cases with many outliers. It introduces MAE as an alternative, defining it mathematically as:

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

Check the Assumptions

- It's important to list and verify assumptions early in a project to avoid serious issues later. There is an example where a machine learning system predicts district prices, assuming they will be used directly. However, if a downstream system categorizes them instead (e.g., "cheap," "medium," "expensive"), the task should be framed as **classification** rather than **regression**. By communicating with the downstream team, the assumption is confirmed: actual prices are needed. With this clarity, development can proceed confidently.

Get the Data

Download the Data

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

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

housing = load_housing_data()
```

Take a Quick Look at the Data Structure

- You start by looking at the top five rows of data using the DataFrame's `head()` method
  - There are 10 attributes : `longitude, latitude, housing_median_age, total_rooms, total_bedrooms, population, households, median_income, median_house_value, and ocean_proximity.`
- The `info()` method provides a summary of the dataset, including the number of rows, data types, and non-null values for each attribute.
  - The dataset contains 20,640 instances, making it small by ML standards but suitable for starting. The `total_bedrooms` attribute has 207 missing values, which need to be handled later.
  - All attributes are numerical except `ocean_proximity`, which is a categorical text attribute. Using `value_counts()`, you can determine the unique categories and the number of districts in each.

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

- The `describe()` method provides a summary of numerical attributes, including count, mean, standard deviation, minimum, and maximum values, as well as quartiles.

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

- The `describe()` method ignores null values and provides key statistics:
  - **Count, mean, min, and max** are straightforward.
  - **Standard deviation (std)** measures value dispersion.
  - **Percentiles (25%, 50%, 75%)** show data distribution:
    - 25% of values are below the 1st quartile.
    - 50% (median) divides the data in half.
    - 75% are below the 3rd quartile.
- Histograms help visualize data distribution by showing the number of instances per value range. Using the `hist()` method on the dataset plots histograms for all numerical attributes at once.

```
import matplotlib.pyplot as plt

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

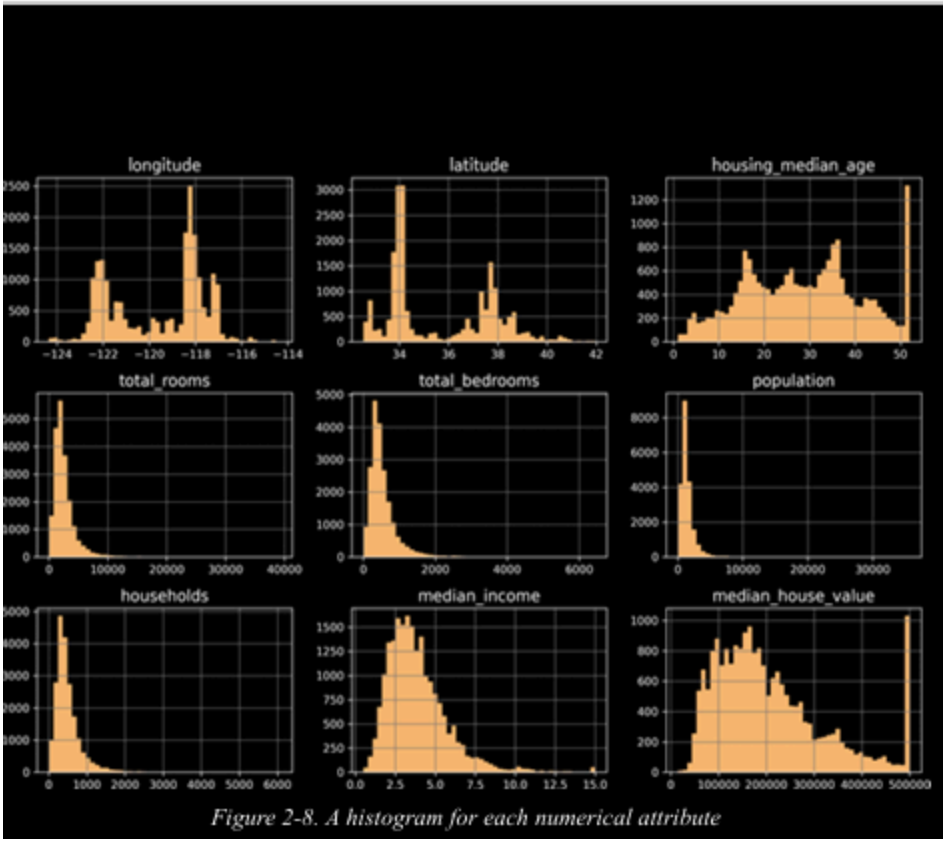


Figure 2-8. A histogram for each numerical attribute

- The `median_income` attribute is scaled and capped between 0.5 and 15 (representing tens of thousands of dollars). It is not in USD but a transformed value. Understanding such preprocessing steps is important in machine learning.
- The housing median age and median house value are capped, which may affect predictions. Since the median house value is the target variable, the model may learn an artificial limit. If precise predictions beyond $500,000 are needed, consider collecting uncapped labels or removing capped districts from the dataset.
- The attributes have different scales, which will be addressed later through feature scaling.
- Many histograms are right-skewed, which may affect pattern detection in some machine learning algorithms. Later, these attributes will be transformed for more symmetrical distributions.

Create a Test Set

- Setting aside a test set early prevents data snooping bias, where unintended patterns in test data influence model selection, leading to overly optimistic error estimates. Typically, 20% or less of the dataset is reserved for testing.

```
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]

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

  - To ensure the test set remains consistent across runs, you can either save the test set after the first run or set a random seed (e.g., `np.random.seed(42)`) before shuffling the data. This guarantees reproducibility by generating the same shuffled indices every time.
  - To maintain a stable train/test split even after updating the dataset, you can use each instance's unique identifier. By computing a hash of the identifier and placing instances in the test set if their hash falls below a threshold (e.g., 20% of the maximum hash value), you ensure that the test set remains consistent across multiple runs while incorporating new data.

```
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]

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")
```

  - If using the row index as a unique identifier, ensure that new data is only appended and no rows are deleted. If this is not feasible, create a unique identifier using stable

features, such as combining a district's latitude and longitude, which remain constant over time.

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

- Scikit-Learn's `train_test_split()` function simplifies dataset splitting by handling shuffling and test size specification. It includes a `random_state` parameter for reproducibility and allows splitting multiple datasets simultaneously based on the same indices.

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

- Random sampling works well for large datasets, but for smaller datasets, it can introduce sampling bias. To ensure representativeness, **stratified sampling** is used, where the population is divided into homogeneous subgroups (**strata**) based on relevant attributes. A proportional number of instances are sampled from each stratum to maintain the overall population distribution, reducing the risk of biased results.
- Suppose that **median income** is a key factor in predicting housing prices, it's important to ensure the test set represents different income levels. However, median income is a **continuous variable**, so it must be **categorized** first. Using `pd.cut()`, we 'll divide the dataset into **five income categories**, where each category represents a specific income range (e.g., 1.5–3 corresponds to $15,000–$30,000). This ensures that each stratum has enough instances, preventing bias in model training.

```
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()
```

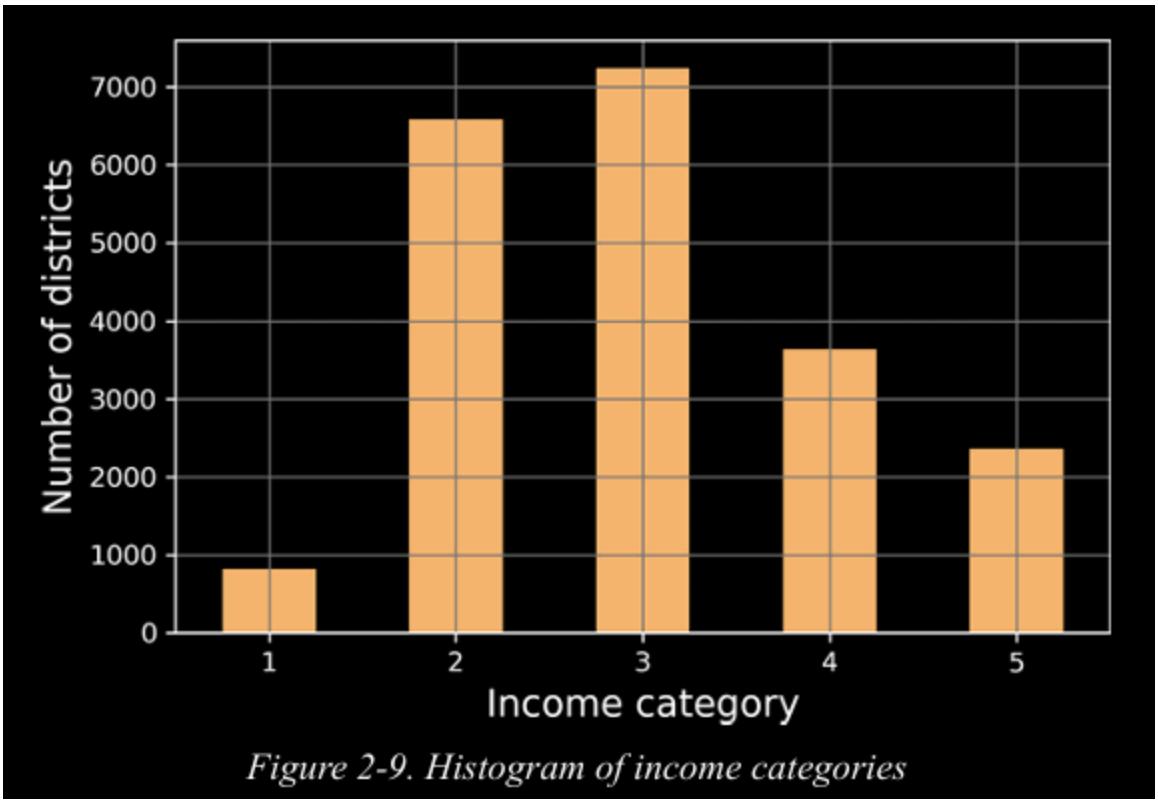


Figure 2-9. Histogram of income categories

- The `split()` method in **StratifiedShuffleSplit** returns **indices**, not the data itself. Using multiple splits (e.g., **10 stratified splits**) helps estimate model performance better, especially in **cross-validation**. The code creates **10 different stratified splits** of the dataset based on **income categories** and stores them in a list.

```
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] # you can use any split
```

- Instead of using **StratifiedShuffleSplit**, you can achieve **stratified sampling** more easily with **train_test_split()** by setting the `stratify` argument to the **income category** column. This ensures the **test set maintains the same income distribution** as the overall dataset. You can verify this by checking the **proportions** of income categories in the test set

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

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

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

- By measuring the **income category proportions** in the full dataset, you can compare different sampling methods. the photo shows that **stratified sampling** maintains proportions **very close** to the overall dataset, while **purely random sampling** introduces noticeable biases. This highlights why **stratified sampling** is preferred when ensuring a representative test set.

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

Explore and Visualize the Data to Gain Insights

- you should make a copy of the original so you can revert to it afterwards:

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

Visualizing Geographical Data

- The dataset contains geographical information (latitude and longitude), making it useful to visualize using scatterplots.
  - A simple scatterplot of the districts in California using `housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)`. This confirms the dataset's coverage

but does not reveal clear patterns.

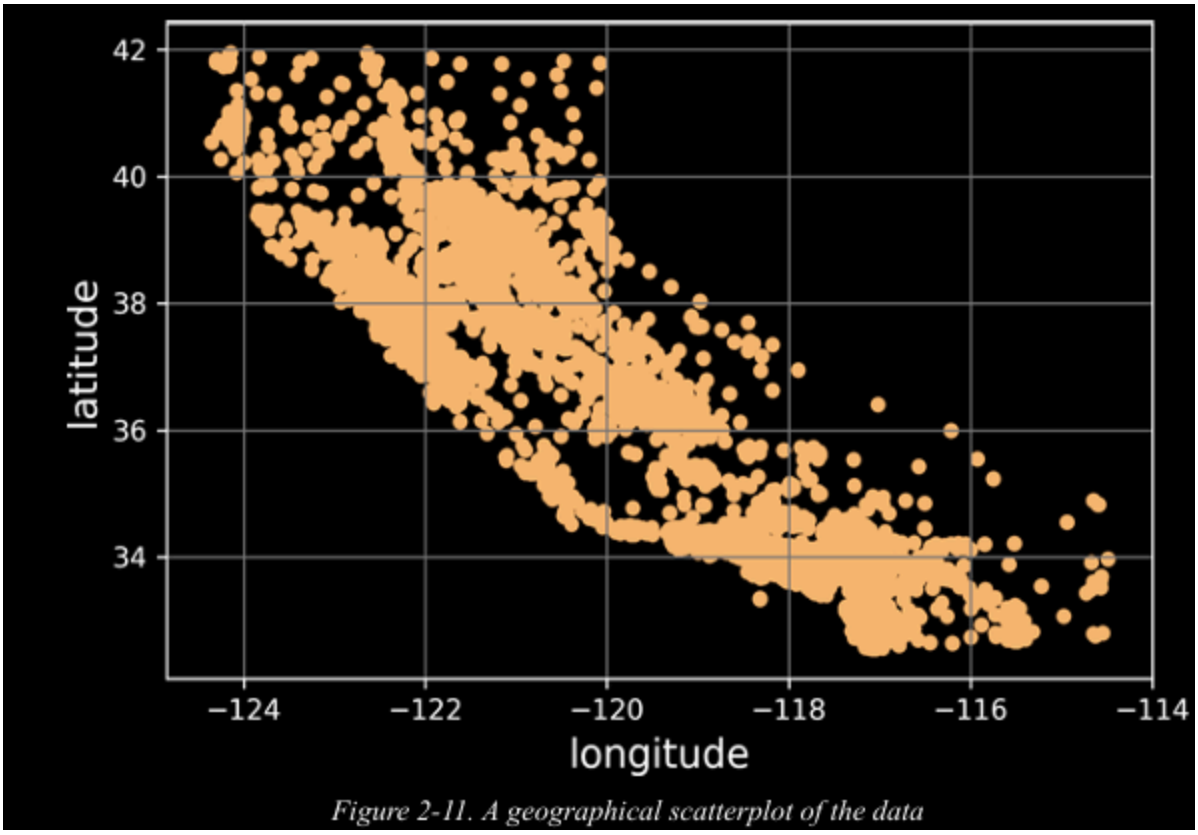


Figure 2-11. A geographical scatterplot of the data

- By setting `alpha=0.2`, the visualization improves by highlighting high-density areas, making regions like the Bay Area, Los Angeles, San Diego, and the Central Valley (Sacramento, Fresno) stand out.

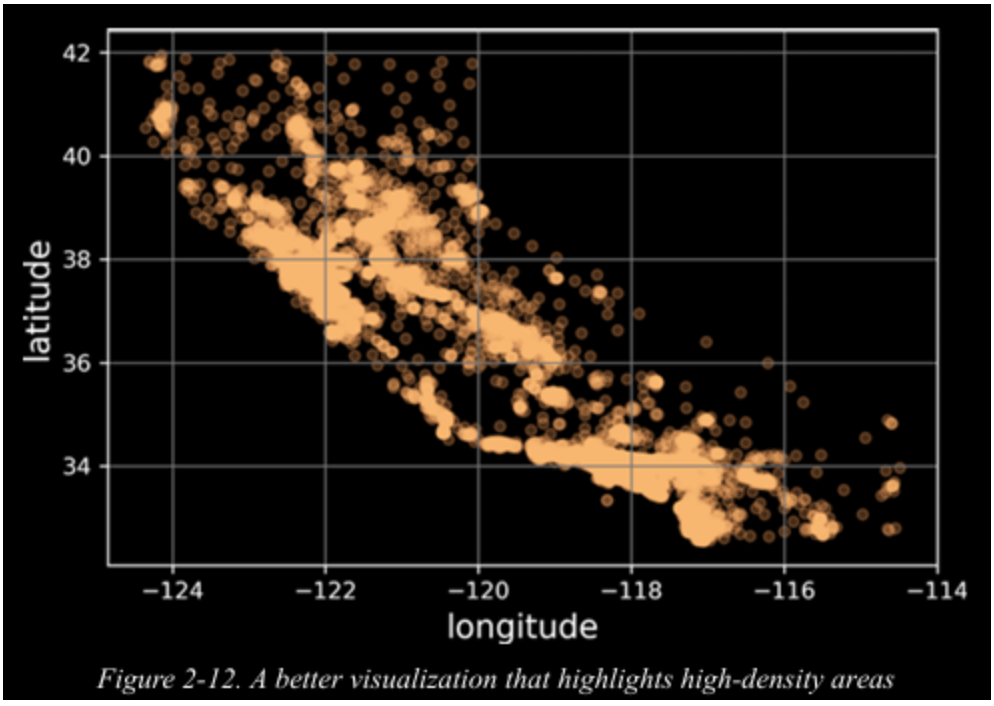


Figure 2-12. A better visualization that highlights high-density areas

- The scatter plot in the photo visualizes **California housing prices**, where:

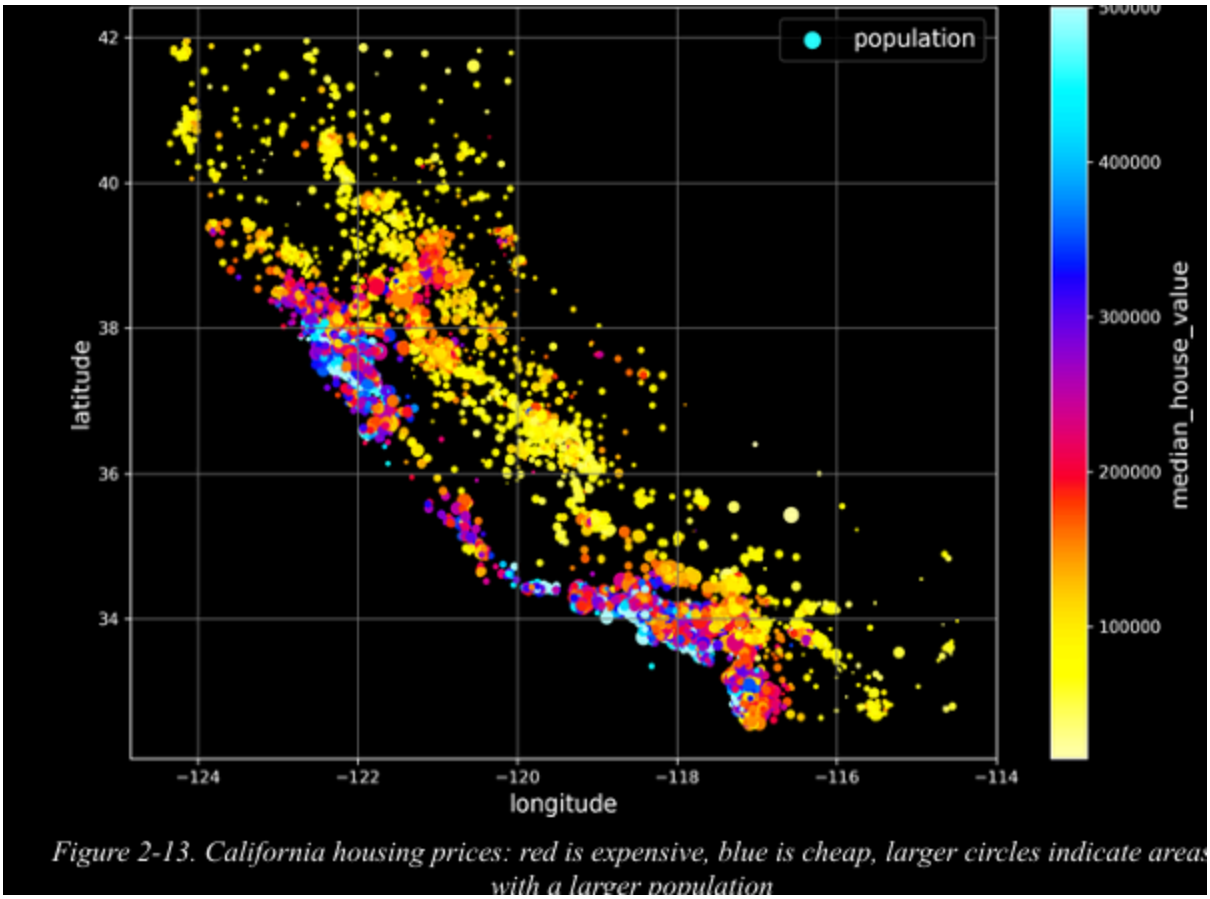


Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

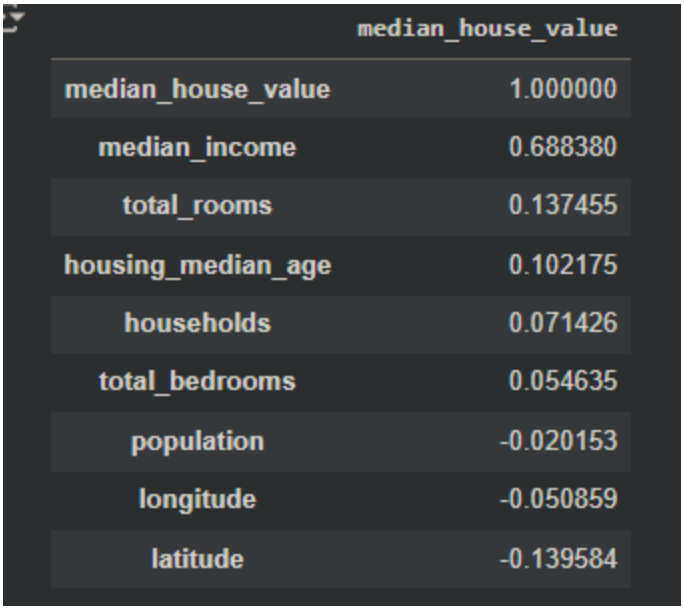
- **Circle size** represents **district population**.
- **Color** (using the `jet` colormap) represents **median house value** (blue = cheaper, red = expensive).
- Housing prices are **highly influenced by location**, especially near the **ocean** and in **high-population areas**.
- **Clustering algorithms** could help identify housing price patterns and improve analysis.
- **Ocean proximity** is a factor, but in **Northern California**, coastal prices are not always high.

```
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()
```

Look for Correlations

- You can calculate the **Pearson correlation coefficient** between attributes in the dataset using the `.corr()` method:

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

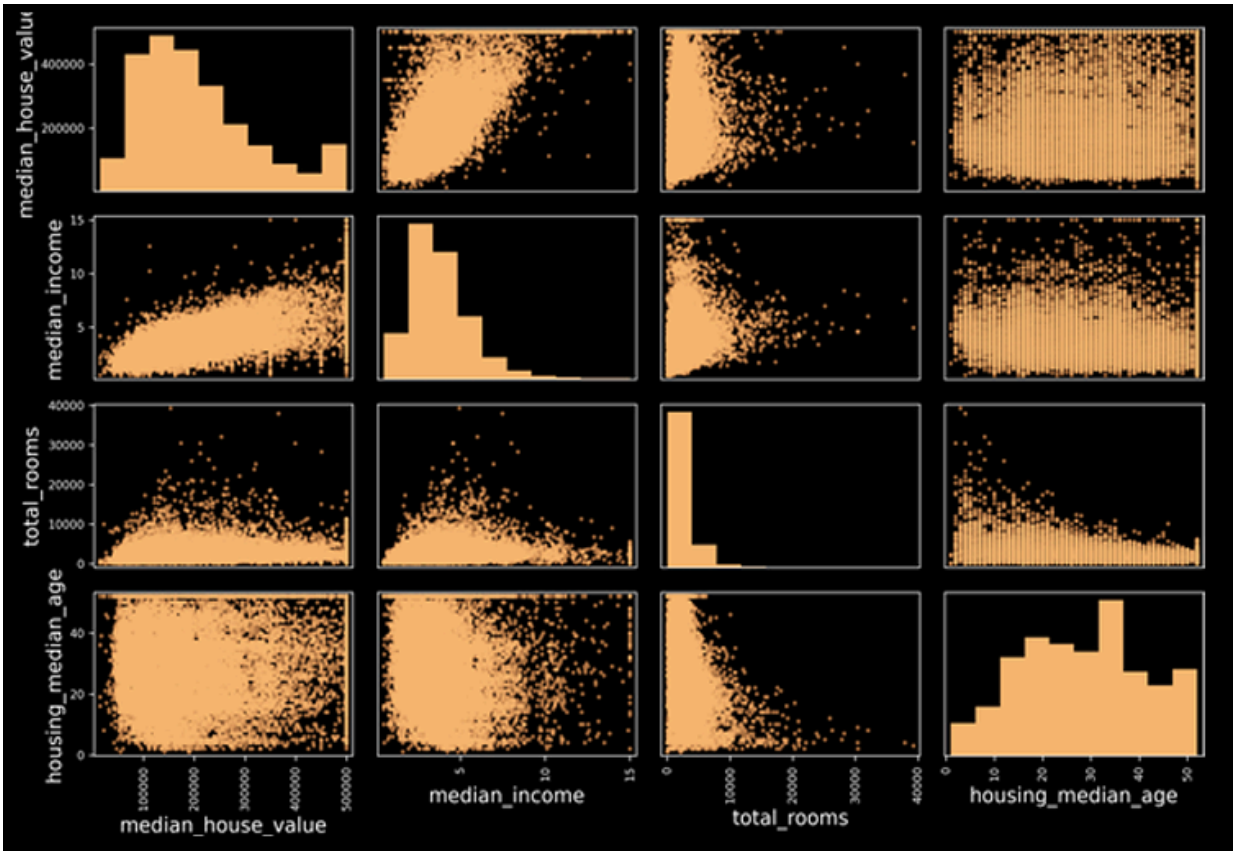


	median_house_value
median_house_value	1.000000
median_income	0.688380
total_rooms	0.137455
housing_median_age	0.102175
households	0.071426
total_bedrooms	0.054635
population	-0.020153
longitude	-0.050859
latitude	-0.139584

- The **correlation coefficient** ranges from **-1 to 1**:
  - **Close to 1** → Strong **positive** correlation (e.g., **higher median income → higher house value**).
  - **Close to -1** → Strong **negative** correlation (e.g., **latitude vs. house value**, where prices tend to decrease as you move north).
  - **Close to 0** → **No linear correlation** between the attributes.

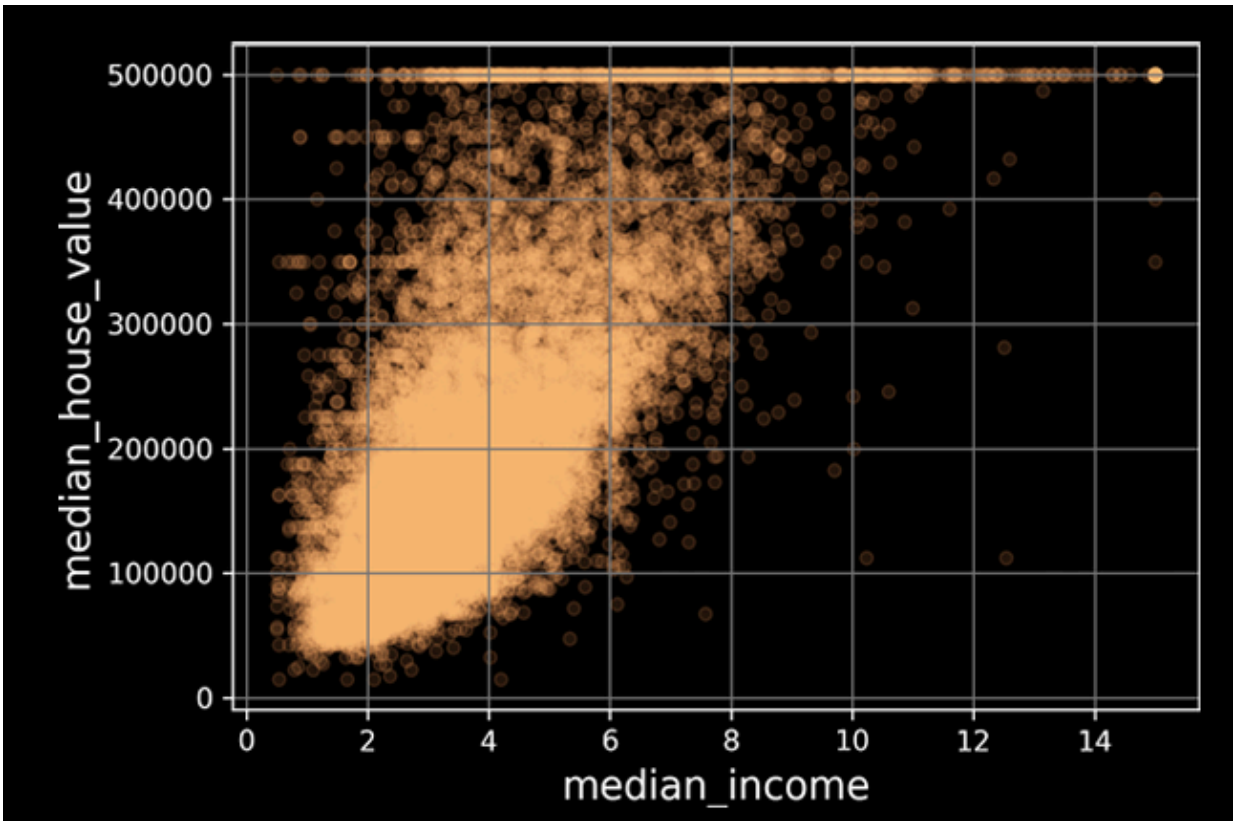
- You can use **Pandas'** `scatter_matrix()` to visualize correlations between numerical attributes. Since there are **11 numerical attributes**, plotting all **121 combinations** is impractical. Instead, you focus on the **most relevant attributes** that show strong correlations with **median house value**.

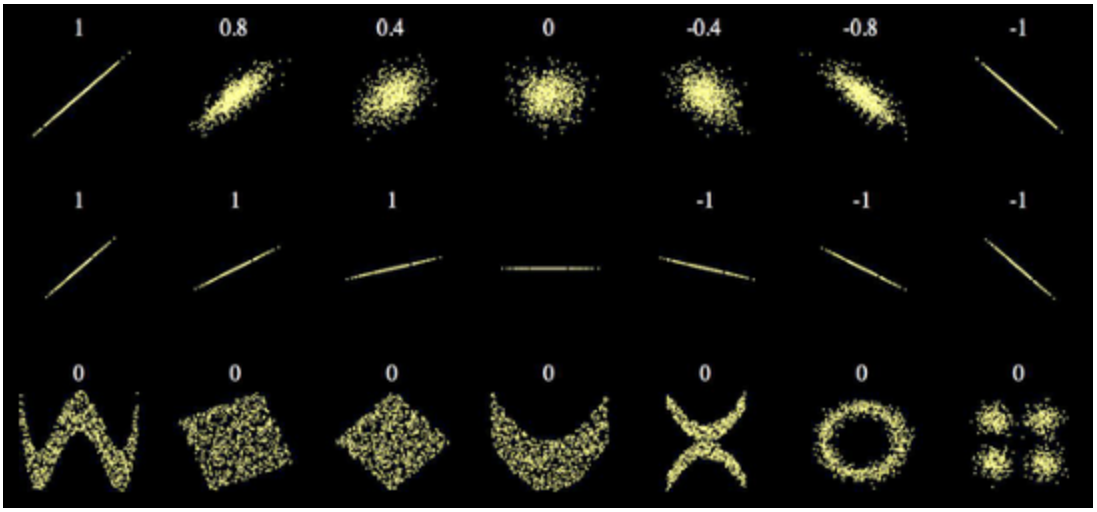
```
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
"housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```



- When using **Pandas'** `scatter_matrix()`, the **main diagonal** would display straight lines if each variable were plotted against itself. Instead, **histograms** of each attribute are shown. From the scatterplots, **median income** appears to be the most promising predictor of **median house value**. To explore this further, a scatter plot of **median income vs. median house value** is created with **low alpha (0.1) to reveal density patterns**.

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

- 

  - The scatter plot confirms a **strong correlation** between **median income** and **median house value**, showing an **upward trend with low dispersion**. It also reveals a **price cap at $500,000** and several **artificial-looking horizontal lines** at **$450,000, $350,000, and $280,000**, suggesting possible data quirks. Removing these districts may help prevent the model from learning **unrealistic patterns**.
- The correlation coefficient **only measures linear relationships** and may miss **nonlinear patterns**. the photo demonstrates datasets with various correlation values. Some datasets with a **correlation of 0** still show clear **nonlinear relationships**. Additionally, datasets with **correlations of 1 or -1** do not necessarily indicate a particular slope, as seen in cases like **height measurements in different units**.
  - 

Experiment with Attribute Combinations

- The previous sections demonstrated various ways to explore data and extract insights. Key takeaways include identifying data quirks that may need cleaning, discovering important correlations (especially with the target variable), and recognizing skewed-right distributions that may require transformations (e.g., logarithm or square root). While each project differs, the general principles of data exploration remain consistent.
- Before preparing data for machine learning, creating meaningful attribute combinations can improve insights. For example, instead of using total rooms alone, calculating **rooms per household** is more informative. Similarly, the **bedrooms-to-rooms ratio** and **population per household** provide better context. These new attributes can help models capture relationships more effectively.

```
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"]
```

  - And then you look at the correlation matrix again:

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

```
median_house_value    1.000000
median_income         0.688380
rooms_per_house       0.143663
total_rooms           0.137455
housing_median_age    0.102175
households            0.071426
total_bedrooms        0.054635
population           -0.020153
people_per_house     -0.038224
longitude            -0.050859
latitude             -0.139584
bedrooms_ratio       -0.256397
Name: median_house_value, dtype: float64
```

- Creating new attributes like **bedrooms_ratio** and **rooms per household** improves correlation with median house value. Homes with a lower bedroom-to-room ratio tend to be more expensive, and larger houses generally cost more. This exploratory step helps build a

strong initial prototype, but the process is iterative—further insights from the model's output can refine data exploration.

Prepare the Data for Machine Learning Algorithms

- Automating data preparation with functions ensures reproducibility, reusability, and flexibility for future datasets and live systems. Before applying transformations, it's essential to revert to a clean training set and separate predictors from target labels. This allows independent preprocessing for each and facilitates experimentation with different transformations.

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

Clean the Data

- Most machine learning algorithms cannot work with missing features
- To handle missing values in the `total_bedrooms` attribute, you have three options:
  1. remove rows with missing values,
  2. drop the entire attribute,
  3. fill missing values using imputation (e.g., median).
- These can be easily implemented using Pandas methods like `dropna()`, `drop()`, and `fillna()`. Imputation is often preferred to preserve valuable data.

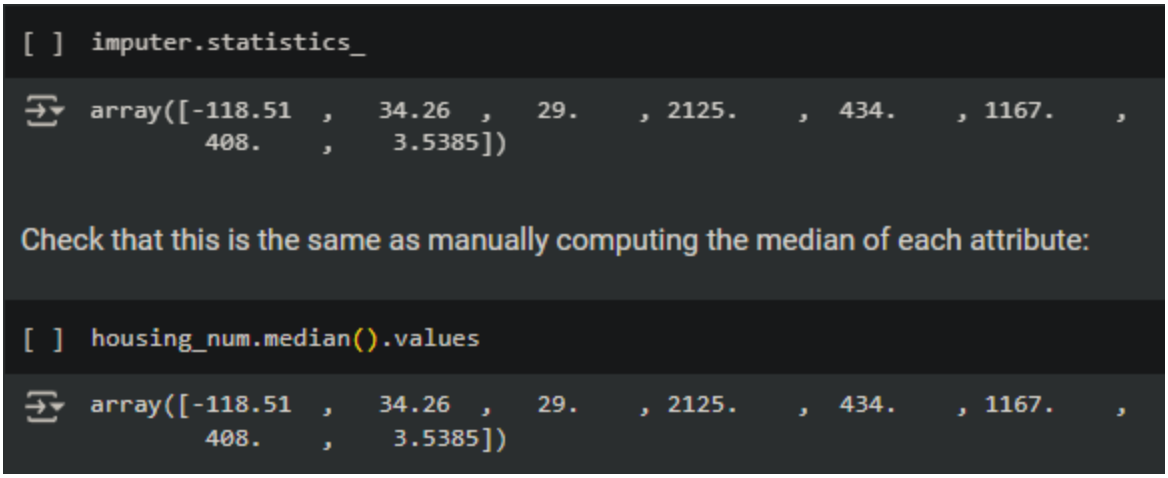
```
housing.dropna(subset=["total_bedrooms"], inplace=True) # option 1
housing.drop("total_bedrooms", axis=1) # option 2
median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

- Instead of manually filling missing values, you use Scikit-Learn's `SimpleImputer` to automatically impute missing values with the median. This approach ensures consistency across the training, validation, test sets, and future data.

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

- Since the median is only applicable to numerical attributes, you first create a copy of the data containing only numerical features. Then, you fit the `SimpleImputer` instance to the training data using the `fit()` method.

```
housing_num = housing.select_dtypes(include=[np.number])
imputer.fit(housing_num) # fit function here calculate the median of each
feature and store it to `statistics_` variable we call that the fit
function train the imputer
```

- The imputer calculated the median for each numerical attribute and stored these values in its `statistics_` variable
  - 

- 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)
```

  - 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.

TIP

There are also more powerful imputers available in the `sklearn.impute` package (both for numerical features only):

- `KNNImputer` replaces each missing value with the mean of the *k*-nearest neighbors’ 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 [1]
- Scikit-Learn transformers output NumPy arrays (or sometimes SciPy sparse matrices) even when given Pandas DataFrames. As a result, `imputer.transform(housing_num)` returns a NumPy array without column names or an index. To restore these, wrap the output in a DataFrame using:

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

Handling Text and Categorical Attributes

- The dataset contains a categorical attribute, `ocean_proximity`, with a limited number of possible values representing categories. Since machine learning models prefer numerical data, this attribute needs to be converted into numerical form. Scikit-Learn's `OrdinalEncoder` can be used for this transformation.

```
from sklearn.preprocessing import OrdinalEncoder
housing_cat = housing[["ocean_proximity"]] # separate categorical columns
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

- Here's what the first few encoded values in `housing_cat_encoded` look like:

```
>>> housing_cat_encoded[:8]
array([[3.],
       [0.],
       [1.],
       [1.],
       [4.],
       [1.],
       [0.],
       [3.]])
```

- The `categories_` attribute of `OrdinalEncoder` stores a list of unique categories for each categorical attribute. However, ordinal encoding assumes numerical order, which may mislead ML models when categories have no inherent ranking (e.g., `ocean_proximity`). To address this, one-hot encoding creates a binary column for each category, ensuring no unintended relationships between values. Scikit-Learn's `OneHotEncoder` is used to perform this transformation.

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

- By default, `OneHotEncoder` outputs a SciPy sparse matrix to efficiently store mostly zero values, saving memory and speeding up computations. This is useful when encoding categorical attributes with many categories. You can convert the sparse matrix to a dense NumPy array using `.toarray()`, or set `sparse_output=False` when initializing `OneHotEncoder` to directly get a NumPy array. The `categories_` attribute stores the unique categories for each encoded feature.
  -
```
>>> 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.]])
```
- Pandas' `get_dummies()` and Scikit-Learn's `OneHotEncoder` both convert categorical data into a one-hot encoded format. However, `OneHotEncoder` is more suitable for machine learning pipelines because it remembers the categories seen during training, ensuring consistency when transforming new data. In contrast, `get_dummies()` dynamically creates columns based on the input data, which can lead to inconsistencies if unseen categories appear.
  - `OneHotEncoder` remembers which categories it was trained on

```
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

```
cat_encoder.transform(df_test)
```

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

  - `get_dummies` generate a column for unknown category but `OneHotEncoder` raising an exception or ignoring them by Setting `handle_unknown="ignore"`

```
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

```
cat_encoder.handle_unknown = "ignore"
cat_encoder.transform(df_test_unknown)
```

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

- **TIP**
  If a categorical attribute has many categories, one-hot encoding can lead to high-dimensional data, slowing training and reducing performance. To address this, consider:
  1. **Feature Engineering**: Replace categories with meaningful numerical features (e.g., distance to the ocean instead of `ocean_proximity`).
  2. **Alternative Encoders**: Use encoders from the `category_encoders` package.
  3. **Embeddings (Neural Networks)**: Represent categories with learnable low-dimensional vectors for efficient learning.
- Scikit-Learn estimators store column names in the `feature_names_in_` attribute when fitted with a DataFrame. This ensures consistency in later transformations or predictions. Additionally, transformers provide `get_feature_names_out()` to retrieve output feature names, which can help reconstruct a DataFrame from transformed data.

```
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)
```

```
df_output
```

	ocean_proximity_<1H OCEAN	ocean_proximity_INLAND	ocean_proximity_ISLAND	ocean_proximity_NEAR BAY	ocean_proximity_NEAR OCEAN
0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	1.0	0.0	0.0

Feature Scaling and Transformation

- Feature scaling ensures machine learning models handle numerical attributes with different scales properly. Common methods include:

1. **Min-max scaling**: Rescales features to a fixed range (e.g., 0 to 1).
2. **Standardization**: Centers data around zero with unit variance.
    - This prevents models from giving undue importance to larger numerical values.

- Always fit scalers using only the training data. Use the trained scaler to transform validation, test, and new data. If new data has outliers, scaled values may exceed the intended range. To prevent this, set the `clip` hyperparameter to `True`.
    - When using `MinMaxScaler`, setting `clip=True` ensures that any new data values outside the training range are clipped to stay within the target range (e.g., [0, 1]). This prevents unexpected scaled values and improves model stability, especially when handling outliers or unseen extreme inputs.
- Min-max scaling (normalization) shifts and rescales values to a specified range, typically 0 to 1. It is done by **subtracting the minimum value and dividing by the range.** Scikit-Learn's `MinMaxScaler` performs this transformation, and its `feature_range` parameter allows customization (e.g., -1 to 1 for neural networks).

```
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)
```

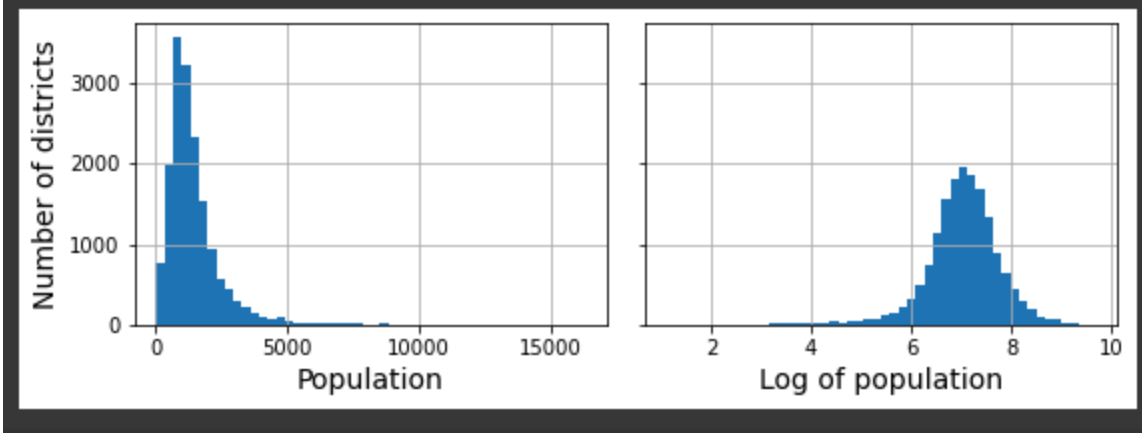
- Standardization transforms data by subtracting the mean and dividing by the standard deviation, resulting in a **zero mean and unit variance**(and unit standard deviation). Unlike min-max scaling, it does not restrict values to a fixed range and is less affected by outliers. Scikit-Learn provides `StandardScaler` for this transformation.

```
from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

- To scale a sparse matrix without converting it to a dense matrix, use `StandardScaler` with `with_mean=False`. This ensures that only division by the standard deviation is applied, preserving sparsity by avoiding mean subtraction.
    - To scale a **sparse matrix** without converting it to dense, use `StandardScaler(with_mean=False)`. This avoids subtracting the mean (which would break sparsity by changing zeros by substract mean from them) and only scales by standard deviation, keeping the matrix memory-efficient.
- When a feature has a heavy tail, min-max scaling and standardization can squash most values into a small range, which is problematic for machine learning models. To address this, transform the feature to reduce the heavy tail and make the distribution more symmetrical. For positive features with a right-heavy tail, using a square root or a power transformation (between 0 and 1) helps. If the tail is very long, applying a logarithm can be

effective. For example, the population feature follows a power law, and taking its log makes it resemble a Gaussian (bell-shaped) distribution

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



- the normalization and standarization helps to make the scale of all the data is near to each other or the same **But**: They do **not fix skewness**. but the log and sqrt work on eliminate the tail and skewness

Skew Type	Fix with:
Right-skewed	Log, square root, or power (0 < p < 1)
Left-skewed	Square or cube

- Bucketizing transforms a **heavy-tailed feature** by dividing its values into **equal-sized buckets** (e.g., percentiles). Each value is replaced with the **index of the bucket it belongs to**, ensuring a **more ==uniform distribution**==. Since the transformed feature is already balanced, **further scaling is unnecessary**. Optionally, dividing by the number of buckets normalizes values to the **0–1 range**.

```
# extra code – just shows that we get a uniform distribution

percentiles = [np.percentile(housing["median_income"], p) for p in range(1,
100)]

flattened_median_income = pd.cut(housing["median_income"], bins=[-np.inf] +
percentiles + [np.inf], labels=range(1, 100 + 1))
```

```
flattened_median_income.hist(bins=50)
plt.xlabel("Median income percentile")
plt.ylabel("Number of districts")
plt.show()

# Note: incomes below the 1st percentile are labeled 1, and incomes above
the
# 99th percentile are labeled 100. This is why the distribution below
ranges
# from 1 to 100 (not 0 to 100).
```

- Bucketizing multimodal[2] features like `housing_median_age` helps models learn distinct patterns. Instead of treating bucket indices as numbers, they should be categorized using **OneHotEncoding** to avoid misleading numerical relationships. This allows the model to capture unique trends, such as lower prices for houses built 35 years ago due to outdated styles.
- To handle **multimodal distributions**, another approach is to create features representing similarity to key modes. This can be done using **Radial Basis Functions (RBF)**, like the **Gaussian RBF**, which measures similarity based on distance. The similarity to a mode (e.g., age 35) is computed using **$\exp(-\gamma(x - 35)^2)$**, where **γ (gamma)** controls how fast similarity decreases with distance. In **Scikit-Learn**, this can be implemented using `rbf_kernel()`, creating a new feature that helps models learn patterns related to specific age groups.

```
from sklearn.metrics.pairwise import rbf_kernel
age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]],
gamma=0.1)
```

- The **Gaussian RBF similarity feature** measures how close housing median age values are to **35**, with a peak at this value. A **higher gamma (0.10)** results in a narrow, sharp peak, while a **lower gamma (0.03)** creates a broader similarity range. If houses around age **35** have a strong correlation with lower prices, this feature could enhance model performance.

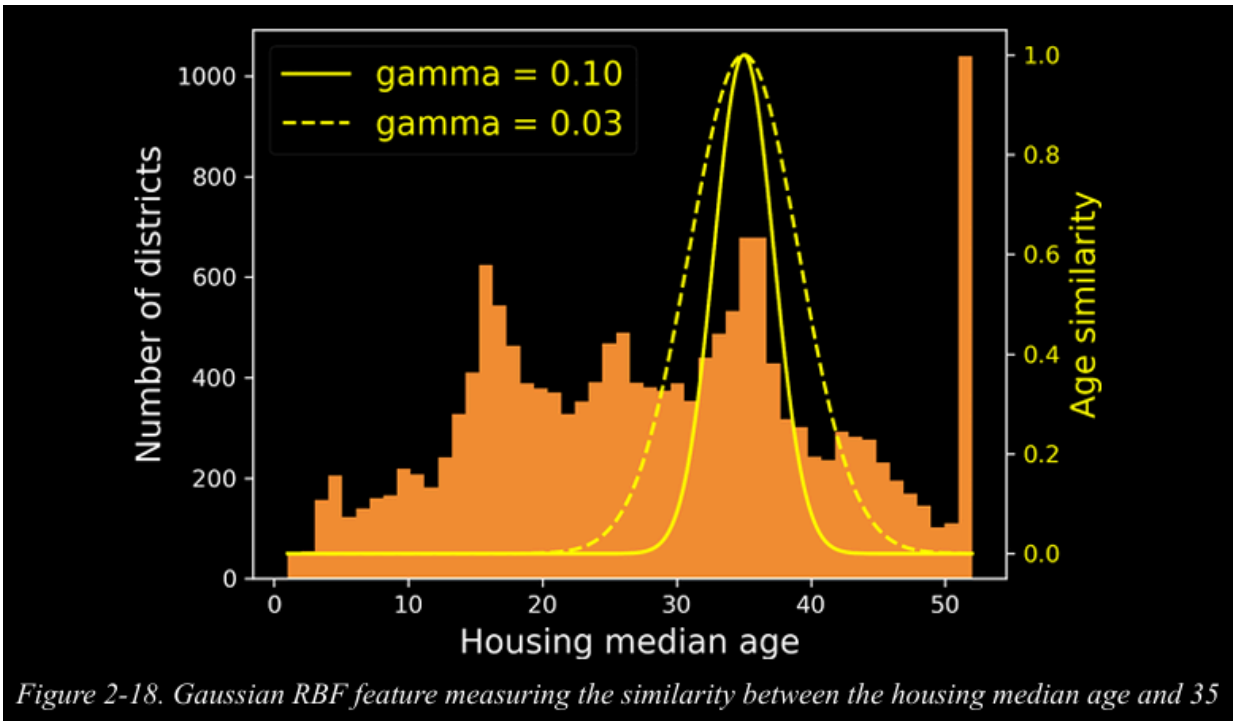


Figure 2-18. Gaussian RBF feature measuring the similarity between the housing median age and 35

- If the target variable has a **heavy-tailed distribution**, it may need to be **transformed**, such as applying a **log transformation**. However, the model will then predict the transformed values, requiring an **inverse transformation** to return to the original scale.Scikit-Learn's **inverse_transform()** method simplifies this process. The example demonstrates using **StandardScaler** to scale the target values before training a **Linear Regression** model on **median income**. After making predictions, the results are transformed back to the original scale using **inverse_transform()**.

```
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)
```

- The `TransformedTargetRegressor` simplifies label scaling by automatically applying a transformation (e.g., `StandardScaler`) to the target variable before training and then reversing it after prediction. This eliminates the need for manual scaling and inverse transformations. **it only scales the label (target) data, not the input (train) data.**

```
from sklearn.compose import TransformedTargetRegressor
model = TransformedTargetRegressor(LinearRegression(), ransformer =
StandardScaler())
```

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

Custom Transformers

- Scikit-Learn's `FunctionTransformer` allows you to apply custom transformations to features without needing to define a full transformer class. You can use it for tasks like log transformation (e.g., for heavy-tailed distributions) or computing similarity measures (e.g., Gaussian RBF). It also supports an inverse function for transformations that need to be reversed, such as when used in `TransformedTargetRegressor`.

```
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. and Your transformation function can take hyperparameters as additional arguments.

```
rbf_transformer = FunctionTransformer(rbf_kernel, kw_args=dict(Y=[[35.]],
gamma=0.1))
age_
simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

- Use `FunctionTransformer` for mathematical functions like `log`, `sqrt`, or custom transformations.
- 🧠 For built-in scalers like `StandardScaler` or `MinMaxScaler`, just use them directly — they already handle fitting and inverse transformation efficiently.
- FunctionTransformer has fit function but do nothig there is no any benefit from it. it's just for compatibility with pipelines.
- Scikit-Learn's `FunctionTransformer` can be used for various custom transformations, such as computing geographic similarity using the RBF kernel or creating feature combinations like ratios. However, the RBF kernel lacks an inverse function since distances are symmetric. Additionally, when applied to multiple features, it measures Euclidean distance rather than treating each feature separately.

```
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. For example, here's 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]])
    ```
- To create a trainable custom transformer in Scikit-Learn, you need a class with `fit()`, `transform()`, and optionally `fit_transform()`. Using `TransformerMixin` provides `fit_transform()` automatically, and inheriting `BaseEstimator` enables hyperparameter tuning.
  The example defines `StandardScalerClone`, mimicking `StandardScaler`. It:
  - Initializes with an optional `with_mean` parameter.
  - Uses `fit()` to compute the mean and standard deviation of `X`.
  - Uses `transform()` to normalize `X` using the learned parameters.
  - Ensures input validation with `check_array()` and `check_is_fitted()`.
    This structure ensures compatibility with Scikit-Learn pipelines and hyperparameter tuning.

```
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_
```

- Key points for implementing a custom Scikit-Learn transformer: (very important)
  - Use `sklearn.utils.validation` for input validation (important in production code).
  - The `fit()` method must have `X` and `y=None` for pipeline compatibility.
  - `n_features_in_` should be set in `fit()` to ensure input consistency.
  - `fit()` must return `self`.
  - For completeness, transformers should:
    - Set `feature_names_in_` when using DataFrames.
    - Implement `get_feature_names_out()`.
    - Provide `inverse_transform()` if the transformation is reversible.
- This code defines a **custom transformer** called `ClusterSimilarity`, which uses **KMeans clustering** to identify key clusters in the training data and then applies an **RBF kernel** to measure similarity to the cluster centers.
  Key Components:
  - `fit(X, y=None, sample_weight=None)`:
    - Uses `KMeans` to find `n_clusters` in the data.
  - `transform(X)`:
    - Computes the similarity of each sample to the cluster centers using `rbf_kernel()`.
  - `get_feature_names_out()`:
    - Returns descriptive names for the generated features.
    Purpose:
  - Creates features based on cluster similarity, which can improve model performance in tasks like **classification** or **regression**.

```
from sklearn.cluster import KMeans
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)]
```

- The **K-Means** algorithm is a **clustering technique** that finds a specified number of clusters (`n_clusters`) in data. It uses randomness, so setting `random_state` ensures reproducibility. The `sample_weight` parameter allows assigning different weights to samples during training.
  Custom Transformer: `ClusterSimilarity`
  - This transformer **clusters** geographic locations (`latitude` , `longitude`) and computes **Gaussian RBF similarity** between each district and the identified clusters.
  - **Key Steps:**
    1. **Fit**: Uses K-Means to find `n_clusters=10` .
    2. **Transform**: Computes similarity of each district to each cluster center.
    3. **Result**: A matrix where each row represents a district, and each column represents its similarity to a cluster.

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

```
>>> similarities[:3].round(2)
array([[0.  , 0.14, 0.  , 0.  , 0.  , 0.08, 0.  , 0.99, 0.  , 0.6
],
       [0.63, 0.  , 0.99, 0.  , 0.  , 0.  , 0.04, 0.  , 0.11, 0.
],
       [0.  , 0.29, 0.  , 0.  , 0.01, 0.44, 0.  , 0.7 , 0.  , 0.3
]])
```

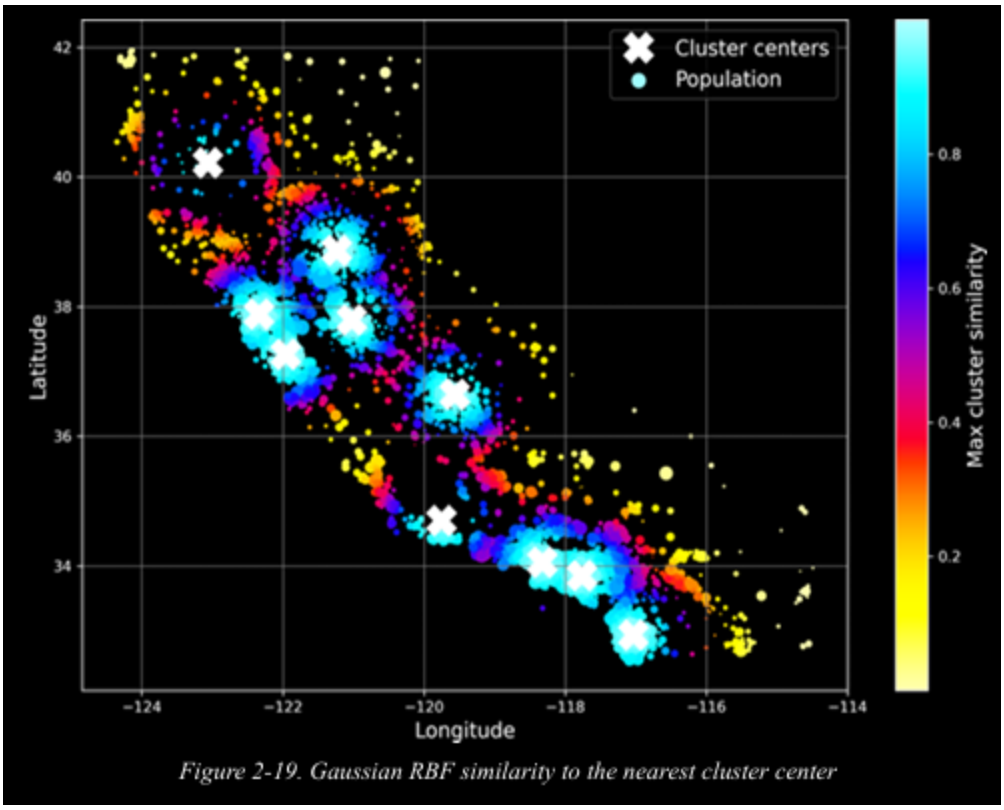


Figure 2-19. Gaussian RBF similarity to the nearest cluster center

---

Transformation Pipelines

- Scikit-Learn's `Pipeline` automates sequential data transformations. In the example, a pipeline is created to **impute missing values** using `SimpleImputer` and **scale features** with `StandardScaler`. This ensures transformations are applied in the correct order.

```
from sklearn.pipeline import Pipeline
num_pipeline = Pipeline([
        ("impute", SimpleImputer(strategy="median")),
        ("standardize", StandardScaler()),
        ])
```

- The `Pipeline` constructor takes a list of **name/estimator pairs**, ensuring *unique names without double underscores.* All steps must be **transformers** (having `fit_transform()`), except the last one, which can be any estimator.
  - **Tip:** In Jupyter Notebook, use `sklearn.set_config(display="diagram")` to visualize pipelines interactively.
- You can use `make_pipeline()` instead of naming transformers manually. It automatically assigns names based on the transformer class names in lowercase. If multiple transformers have the same name, an index is appended (e.g., `"foo-1"`, `"foo-2"`).

```
from sklearn.pipeline import make_pipeline
num_pipeline = make_pipeline(
        SimpleImputer(strategy="median"),
        StandardScaler()
        )
```

- When calling `fit()` on a pipeline, it sequentially applies `fit_transform()` on all transformers and `fit()` on the final estimator.
  - The pipeline behaves like its last estimator:
    - If it's a **transformer** (e.g., `StandardScaler`), the pipeline has a `transform()` method.
    - If it's a **predictor**, the pipeline has a `predict()` method instead.
  - The `get_feature_names_out()` method helps recover a structured DataFrame from transformed data.
  -
```
>>> 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]])
```

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

- **Pipeline Indexing and ColumnTransformer**
  - **Pipeline Indexing:**
    - `pipeline[1]` → Returns the second estimator in the pipeline.
    - `pipeline[:-1]` → Returns a new pipeline excluding the last estimator.
    - `pipeline["step_name"]` → Accesses an estimator by its name.
    - `pipeline.steps` → A list of (name, estimator) pairs.
    - `pipeline.named_steps` → A dictionary mapping names to estimators.
  - **Handling Different Column Types:**
    - Instead of separate pipelines for numerical and categorical data, we can combine them using `ColumnTransformer`.

```
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), ])
```

- To use `ColumnTransformer`:
  1. **Import** `ColumnTransformer`.
  2. **Define numerical & categorical column names**.
  3. **Create pipelines** for categorical and numerical attributes.
  4. **Construct** `ColumnTransformer` using a list of triplets:
     - `(name, transformer, columns)`
     - Names must be **unique** and **avoid double underscores**.

- TIP : You can control how `ColumnTransformer` handles unspecified columns:
  - **"drop"** → Exclude them from the output.
  - **"passthrough"** → Keep them unchanged.
  - `remainder` (default: "drop") → Can be set to a transformer (e.g., `StandardScaler()`) or `"passthrough"` to process or retain them.
- Scikit-Learn provides `make_column_selector()` to automatically select columns based on data type (e.g., numerical or categorical). Instead of listing column names, you can pass this selector to `ColumnTransformer`. Additionally, `make_column_transformer()` simplifies transformer creation by auto-naming the steps.
  Example:
  - `make_column_selector(dtype_include=np.number)` selects numerical columns.
  - `make_column_transformer()` creates a `ColumnTransformer` with auto-generated names.
    This approach makes preprocessing more convenient and adaptable.

```
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)
```

- `ColumnTransformer` decides whether to return a sparse or dense matrix based on the density of nonzero values. `OneHotEncoder` produces a sparse matrix, while `num_pipeline` returns a dense matrix. If the final matrix has a density below 30%, it remains sparse; otherwise, it is converted to a dense matrix. In this case, the density is above the threshold, so a dense matrix is returned.
- This pipeline performs a series of preprocessing steps to prepare the data for machine learning models. Here's a breakdown of what it does:
  1. **Handling Missing Values**
     - Numerical features: Missing values are replaced with the **median**.
     - Categorical features: Missing values are replaced with the **most frequent category**.
  2. **Feature Engineering**
     - **Ratio Features:** Computes new features:
       - `bedrooms_ratio = total_bedrooms / total_rooms`
       - `rooms_per_house = total_rooms / households`
       - `people_per_house = population / households`
     - **Cluster Similarity Features:** Uses `KMeans` clustering to transform latitude & longitude into similarity features.
     - **Log Transformation:** Applies `log()` to skewed features for better model performance.
  3. **Encoding Categorical Data**
     - Uses **one-hot encoding** to convert categorical variables into numerical format.
  4. **Standardization**
     - **All numerical features** are standardized to have a mean of 0 and unit variance.
  5. **ColumnTransformer Usage**
     - Combines all the above transformations into a single preprocessing step.
     - Uses a **default pipeline** for unprocessed numerical columns.
       Final Output:
       The transformed dataset includes engineered ratio features, standardized numerical features, cluster similarity scores, and one-hot encoded categorical variables—fully ready for training ML models

```
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


housing_prepared = preprocessing.fit_transform(housing)
housing_prepared.shape     # (16512, 24)
```

Select and Train a Model

Train and Evaluate on the Training Set

- After completing the preprocessing steps, you train a simple linear regression model using `LinearRegression` from `sklearn.linear_model`, combined with a preprocessing pipeline.

```
from sklearn.linear_model import LinearRegression
lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_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.])
```

- The linear regression model works but has significant prediction errors, with some predictions being off by over $200,000. To evaluate its performance, you calculate the RMSE using `mean_squared_error` with `squared=False`, resulting in an RMSE of approximately $68,687. Given that median housing values range from $120,000 to $265,000, this error is quite large, indicating underfitting. This suggests that the model is **too simple or the features are insufficient.** To address underfitting, potential solutions include using a more complex model, improving feature selection, or reducing constraints—though regularization is not an issue here. The next step is to try a more powerful model.

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

print(lin_rmse )  # 68687.89176589991
```

- You decide to try a DecisionTreeRegressor, as this is a fairly powerful model capable of finding complex nonlinear relationships

```
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
```

- If a model shows no error, it is likely overfitting rather than being perfect. To confirm this, you should avoid using the test set prematurely. Instead, you need to split the training data into a training set and a validation set to properly assess the model's performance before final evaluation.

Better Evaluation Using Cross-Validation

- Scikit-Learn's cross-validation expects scores where **higher is better**, but RMSE is a **cost function** where **lower is better**. To match this convention, RMSE is negated, returning **negative values**. To get the actual RMSE scores, simply **switch the sign** of the output.
- To evaluate the decision tree model, you can either manually split the dataset using `train_test_split()` or use **k-fold cross-validation** for a more reliable assessment. With **10-fold cross-validation**, the training set is split into **10 folds**, and the model is trained on **9 folds** while being evaluated on **the remaining fold**. This process repeats **10 times**, producing an array of RMSE 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)
```

```
>>> 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
```

- Cross-validation reveals that the **decision tree model** performs worse than expected, with an **RMSE of 66,868** and a **standard deviation of 2,061**, showing slight variability across folds. While it **slightly outperforms linear regression** (which has an **RMSE of 69,858** and

a **standard deviation of 4,182**), the difference is minimal.
The **decision tree is overfitting**—it has **low training error (almost zero) but high validation error**. Cross-validation provides both performance estimates and their precision, but it requires multiple training runs, making it computationally expensive.

- The **RandomForestRegressor** improves performance by training multiple **decision trees** on random subsets of features and averaging their predictions. This **ensemble approach** reduces overfitting and enhances accuracy.

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

```
>>> pd.Series(forest_rmses).describe()
count       10.000000
mean     47019.561281
std       1033.957120
min      45458.112527
25%      46464.031184
50%      46967.596354
75%      47325.694987
max      49243.765795
dtype: float64
```

- **Random forests** perform much better, but they still **overfit** with a low training RMSE (**17,474**) compared to validation errors. To address this, you can **simplify the model, regularize it, or get more data**.
Before fine-tuning, it's best to explore **other ML models** like **SVMs and neural networks** to identify **2–5 promising candidates** for further optimization.

```
forest_reg.fit(housing, housing_labels)
housing_predictions = forest_reg.predict(housing)
forest_rmse = root_mean_squared_error(housing_labels, housing_predictions)
forest_rmse
```

```
17474.619286483998
```

Fine-Tune Your Model

- After shortlisting **promising models**, the next step is to **fine-tune** them for better performance. There are several techniques to achieve this, which will be explored next.

Grid Search

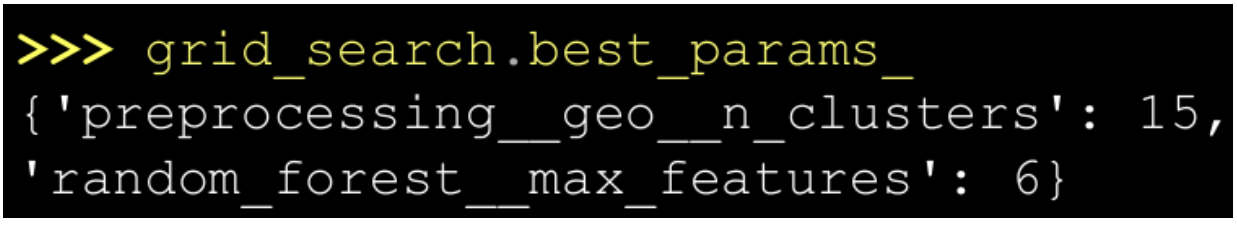
- Manually tuning **hyperparameters** is tedious, so **GridSearchCV** automates the process by testing all possible combinations and using **cross-validation** to find the best one.
  In the example, **GridSearchCV** is used to fine-tune a **RandomForestRegressor** by searching over different values for `max_features` and `n_clusters`. It evaluates each combination using **3-fold cross-validation** to determine the best settings.

```
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)
```

- Wrapping **preprocessing steps** in a **pipeline** allows tuning both **preprocessing and model hyperparameters** together, as they may interact. To speed up expensive computations, **Scikit-Learn** can cache fitted transformers, avoiding redundant computations when rerunning with the same hyperparameters.
- **GridSearchCV** explores **15 combinations** of hyperparameters, with **3-fold cross-validation**, leading to **45 training rounds**. After completion, the best parameters can be retrieved using `grid_search.best_params_`, which in this case are **n_clusters = 15** and **max_features = 6**.
  - 

    - TIP : Since 15 is the maximum value that was evaluated for n_clusters, you should probably try searching again with higher values; the score may continue to improve.
- The **best estimator** can be accessed using `grid_search.best_estimator_`. If `refit=True` (default), the model is **retrained on the full training set** with the best hyperparameters, improving performance.
  The **evaluation scores** from `grid_search.cv_results_` can be wrapped in a DataFrame for better readability. This provides:
    - Test scores for each **hyperparameter combination**.
    - Scores for **each cross-validation split**.
    - The **mean test RMSE** across all splits.
      The **best model** achieved a **mean test RMSE of 44,042**, which is an improvement over the **default hyperparameters' RMSE of 47,019**. **Fine-tuning successfully improved model performance!**

Randomized Search

- **RandomizedSearchCV** is an alternative to **GridSearchCV**, especially useful when the search space is large. Instead of testing all possible combinations, it randomly selects values for each hyperparameter over a fixed number of iterations.
  Advantages of Randomized Search:
  ✅ **Explores more values:** If a hyperparameter has many possible values, random search can test thousands, whereas grid search is limited to predefined options.
  ✅ **Handles irrelevant hyperparameters efficiently:** If a parameter has little impact, grid search significantly increases training time, while random search reduces unnecessary computations.
  ✅ **Scales better with more hyperparameters:** Grid search struggles with many hyperparameters, while random search allows control over the number of iterations, keeping training feasible.
  For each hyperparameter, you can specify a list of values or a probability distribution to sample from.

```
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)
```

- **HalvingGridSearchCV & HalvingRandomSearchCV Explained:**
  - 🔹 **Step 1: Initial Candidates** → Many hyperparameter combinations are generated and trained using **limited resources** (e.g., small dataset or fewer iterations).
  - 🔹 **Step 2: Evaluation & Elimination** → Weak models are **discarded**, and only the best move to the next round.
  - 🔹 **Step 3: Increased Resources** → Remaining models are trained on **more data** or **more iterations** to refine performance.
  - 🔹 **Step 4: Final Selection** → The top models are trained using **full resources**, and the best one is chosen.
    - **Why use it? Saves time** by eliminating poor models early, focusing resources on promising ones, and allowing wider hyperparameter exploration efficiently.

Ensemble Methods

- **Ensemble Learning for Fine-Tuning Models**
  - 🔹 **Idea:** Combine multiple well-performing models to improve overall performance.
  - 🔹 **Why?** Different models make different errors, so combining them reduces mistakes.
  - 🔹 **Example:** Train a **k-nearest neighbors (KNN) model** and a **random forest**, then create an **ensemble** that averages their predictions.
  - 🔹 **Benefit:** Often outperforms any single model, similar to how **random forests** outperform individual decision trees.

Analyzing the Best Models and Their Errors

- You can gain valuable insights by analyzing the best models. For example, `RandomForestRegressor` provides feature importance scores, helping identify which attributes contribute most to 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, 0.04, 0.01, 0.  ,
            0.01, 0.01, 0.01, 0.01, 0.01, 0.  , 0.01, 0.01, 0.01, 0.  , 0.01,
            0.01, 0.01, 0.01, 0.01, 0.  , 0.  , 0.02, 0.01, 0.01, 0.01, 0.02,
            0.01, 0.  , 0.02, 0.03, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01,
            0.01, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01, 0.02, 0.01, 0.  , 0.07,
            0.  , 0.  , 0.  , 0.01])
```

- By sorting these scores, you can determine which features are most relevant and consider removing less useful ones, such as certain categories of `ocean_proximity`.
  ```
  python 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')]
  ```
- `SelectFromModel` in `sklearn` selects the most important features based on a model's `feature_importances_`, automatically dropping less useful ones during transformation.
- Analyze your model's errors to identify improvements, such as adding relevant features, removing uninformative ones, or handling outliers. Ensure the model performs well across all categories (e.g., rural vs. urban, rich vs. poor) by evaluating subsets of your validation data. If performance is poor for a specific group, avoid deploying it for that category until the issue is resolved.

Evaluate Your System on the Test Set

- Once your model is fine-tuned, evaluate it on the test set by making predictions and calculating the error. Extract the features and labels, run the model, and compute metrics like RMSE to assess final performance.

```python
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
```

  - ==note that here in test set if we need to use the `full_pipeline` we will use `transform` only not `fit_transform` ==
- Hyperparameter tuning often **leads to slightly worse performance on new data than on validation data due to overfitting**. Avoid tweaking parameters to improve test results

artificially, as this won't generalize well. Before launch, present findings, document assumptions, highlight successes and limitations, and create clear visualizations. Even if the model's performance is only slightly better than experts' estimates, it may still be valuable if it saves experts time for more critical tasks.

Launch, Monitor, and Maintain Your System

- After getting approval to launch, prepare your model for production by polishing the code, writing documentation, and adding tests. The simplest way to deploy is by saving the trained model using `joblib.dump()`, then loading it in the production environment. It's also useful to save all experimented models, cross-validation scores, and validation predictions for easy comparison and analysis.

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

- Once your model is in production, follow these steps to use it:
  1. **Import All Dependencies**
     Reload any custom functions or classes used in the model, such as:
     - `KMeans`, `BaseEstimator`, `TransformerMixin`
     - Custom functions like `column_ratio`, `ratio_name`
     - Custom classes like `ClusterSimilarity`
  2. **Load the Model**
     Use `joblib` to load the saved model:

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

  3. **Make Predictions**
     Prepare new input data and make predictions:

```
predictions = final_model_reloaded.predict(new_data)
```

  Note: All supporting code (custom transformers/functions) must be transferred and available in the production environment.
- **Summary: Deploying a Model in a Web Application (with Figure 2-20)**
  - The model is integrated into a **web-based system** where:
    1. The **user enters data** and clicks a button (e.g., "Estimate Price").
    2. The input is sent to a **web server**, which forwards it to a **web app**.

3. The web app **calls** `model.predict()` to generate the result.

- ⚠️ **Best practice:** Load the model **once at server startup**, not on every prediction.

- **Alternative: REST API Web Service (Figure 2-20)**

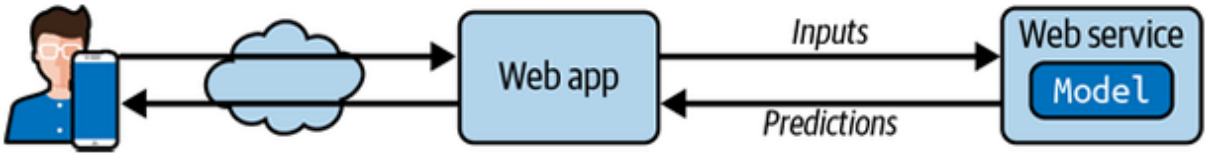


Figure 2-20. A model deployed as a web service and used by a web application

- The model is **deployed as a separate web service** (REST API).
- The web app communicates with it over HTTP.

✅ **Benefits:**

- 🎯 Easy to **upgrade** the model without touching the web app
- ⚖️ Can **scale easily** by running multiple model services and load balancing
- 🌍 Allows the web app to be written in **any language**, not just Python

- Deploying Models to the Cloud with Google Vertex AI
  - Save your trained model using `joblib`.
  - Upload the model file to **Google Cloud Storage (GCS)**.
  - In **Google Vertex AI**, create a **new model version** pointing to that file.
  - Vertex AI automatically:
    - Hosts the model as a **web service**
    - Handles **scaling** and **load balancing**
  - You send **JSON requests** with input data, and receive **JSON predictions** in response.
  - Works for **Scikit-Learn and TensorFlow** models.

📌 Use this web service directly in your website or app for production deployment.

- Monitoring Deployed Models (Plain Version)
  - Deployment is not the final step; ongoing monitoring of model performance is essential.
  - Performance can degrade:
    - Suddenly due to infrastructure issues
    - Gradually due to model rot (when the model no longer fits current data)
  - It's important to track the model's real-world effectiveness over time.
  - In some cases, you can monitor performance using downstream business metrics.
    - Example: In a recommendation system, track sales from recommended products.
    - A drop in such metrics may indicate a problem with the model or data pipeline.
  - If performance drops, the model may need to be retrained with updated data.
- Human-in-the-Loop Monitoring and Automation

- **Automated monitoring isn't always enough.**
  Some cases (like defect detection in images) may require **human analysis** to catch failures early.
- **Human raters** can help by reviewing:
  - Samples of classified data
  - Especially cases where the model is uncertain
  - Raters may be experts, crowdworkers (e.g., via Amazon Mechanical Turk), or even users (e.g., through surveys)
- **Monitoring systems** must be in place to:
  - Continuously evaluate model performance
  - Define actions in case of failure
  - Prevent large-scale errors (like shipping defective products)

- Automate the ML Lifecycle When Possible:
  1. **Collect and label fresh data** regularly (automate if possible)
  2. **Automate model training and hyperparameter tuning**
  3. **Automatically compare** the new model with the current one:
     - On the full test set
     - On key subgroups (e.g., rich/poor, urban/rural)
  4. **Deploy automatically** if performance improves or remains stable
     - Investigate issues if performance drops
       Building a model is just the beginning — maintaining it is often **much more work**.
- Ensuring Input Quality, Backups, and MLOps
  1. **Monitor Input Data Quality**
     - Model performance can slowly degrade due to bad inputs (e.g., broken sensors, outdated data from other teams).
     - To catch issues early, monitor:
       - Missing values
       - Unexpected changes in feature statistics (mean, std)
       - New/unseen categories in categorical features
  2. **Use Backups for Models and Datasets**
     - Keep backups of **every model version** to:
       - Roll back quickly if a new model fails
       - Compare performance across versions
     - Keep backups of **every dataset version** to:
       - Recover from corruption or outliers in newly added data
       - Re-evaluate models on previous data
  3. **Build ML Infrastructure (MLOps)**

- Machine learning in production needs solid infrastructure and processes (MLOps).
- The **first project may take time and effort**, but once infrastructure is ready, future deployments will be much faster.

Resources :

-

Related notes :

-

References :

- ## Internal :
  -
  -
  -
- ## External :
  - the notebook of the chapter
  - hegab videos
  - the book
  -

---

1. Scikit-Learn's API follows a well-structured design based on these principles:
   - **Consistency**: All objects use a simple interface with methods like `fit()`, `transform()`, and `predict()`.
   - **Estimators**: Objects that estimate parameters from data using `fit()`. Hyperparameters are set via instance variables.
   - **Transformers**: Estimators that transform data using `transform()`. They also have `fit_transform()` for efficiency.

- **Predictors**: Estimators that make predictions using `predict()` and evaluate performance with `score()`.
- **Inspection**: Hyperparameters are accessible as instance variables, and learned parameters have an underscore suffix (e.g., `imputer.statistics_`).
- **Nonproliferation of classes**: Uses standard NumPy arrays or SciPy sparse matrices instead of custom classes.
- **Composition**: Reuses building blocks, allowing for Pipelines that chain transformers and estimators.
- **Sensible defaults**: Provides reasonable default values for most parameters to simplify model development.

↩

2. A **multimodal feature** is one where the data has **multiple peaks or "modes"** — not a smooth or bell-shaped distribution. ↩