



Co-funded by the
Erasmus+ Programme
of the European Union



End-to-End Machine Learning Project

Prof. Gheith Abandah

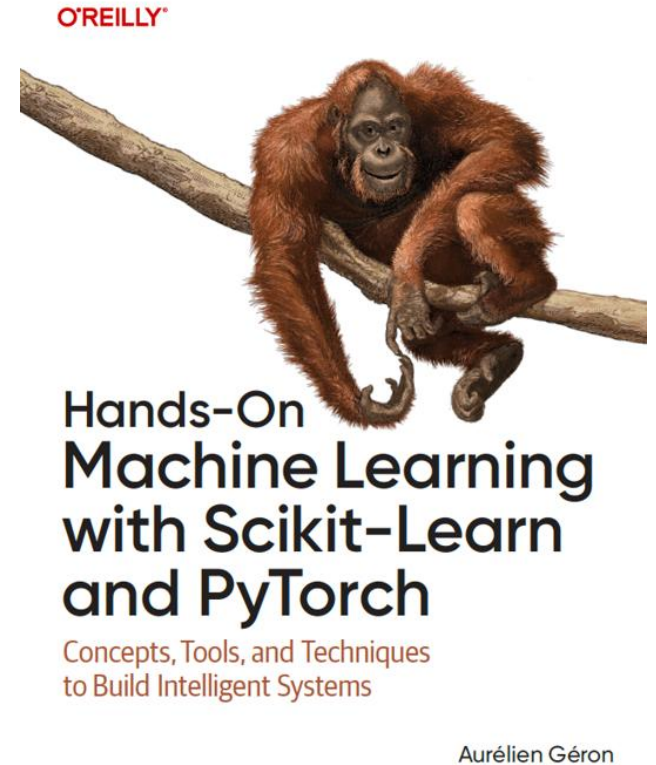
Updated 26/10/2026

Developing Curricula for Artificial Intelligence and Robotics (DeCAIR)
618535-EPP-1-2020-1-JO-EPPKA2-CBHE-JP

Reference

- Chapter 2: **End-to-End Machine Learning Project**

- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn and PyTorch**, O'Reilly, 2025
 - Material: <https://homl.info/colab-p>



The 7 Steps of Machine Learning

- YouTube Video: **The 7 Steps of Machine Learning** from Google Cloud Platform

<https://youtu.be/nKW8Ndu7Mjw>

Caution: *Alcohol is forbidden in the Islamic religion and causes addiction and has negative effects on health.*

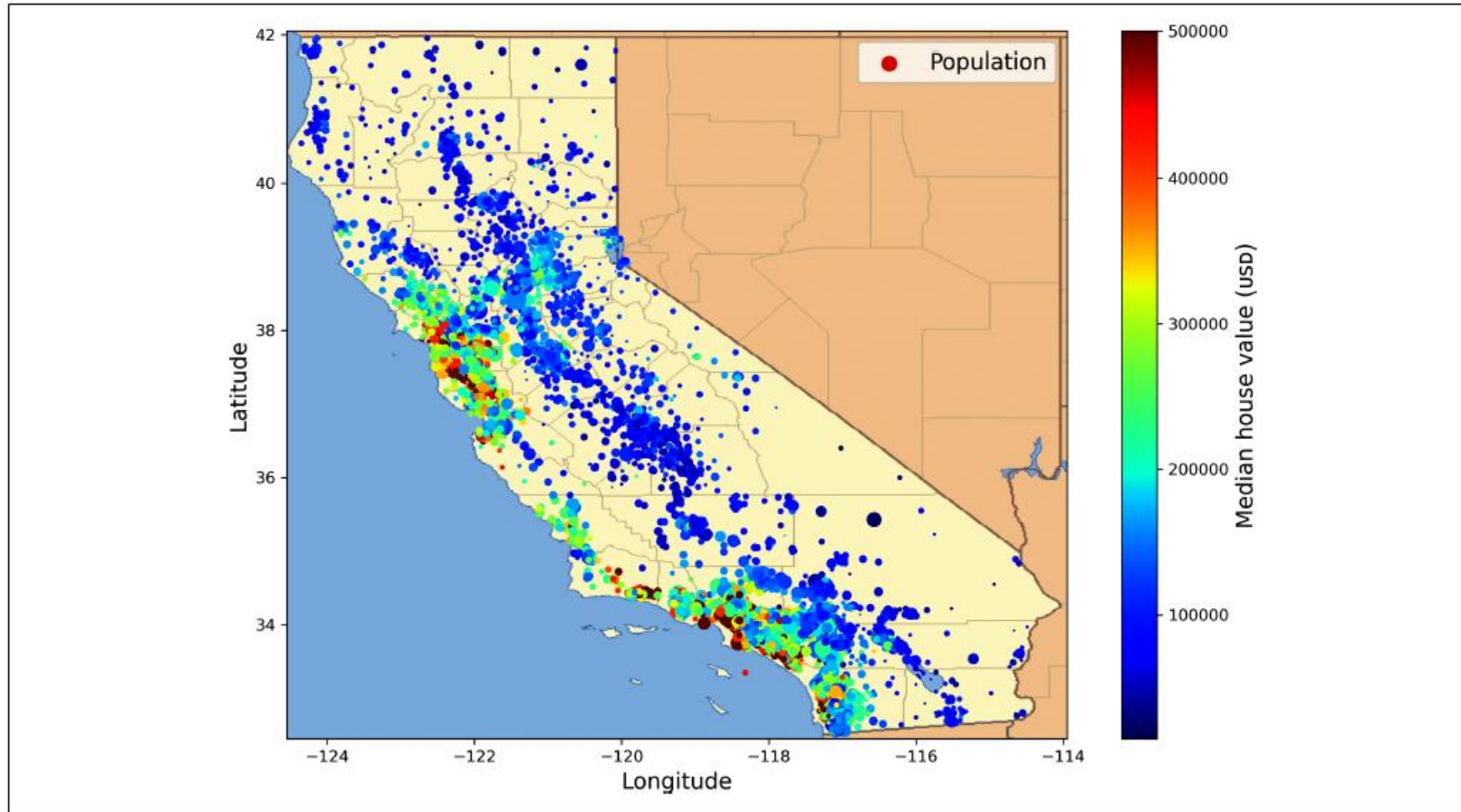
Outline

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

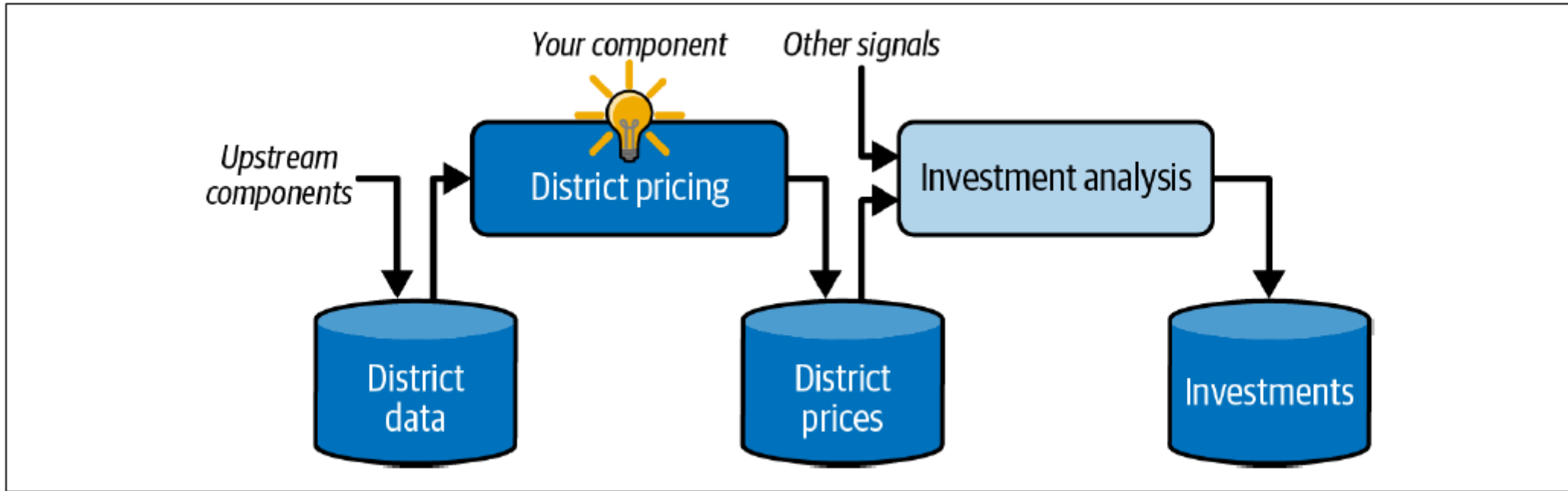
Working with Real Data

- **Popular open data repositories**
 - Google Datasets Search
 - Hugging Face Datasets
 - OpenML.org
 - Kaggle.com
 - UC Irvine Machine Learning Repository
 - Stanford Large Network Dataset Collection
 - Amazon's AWS datasets
 - U.S. Government's Open Data
 - DataPortals.org
 - Wikipedia's list of machine learning datasets

1. Look at the Big Picture: CA Housing Data



1.1. Frame the Problem

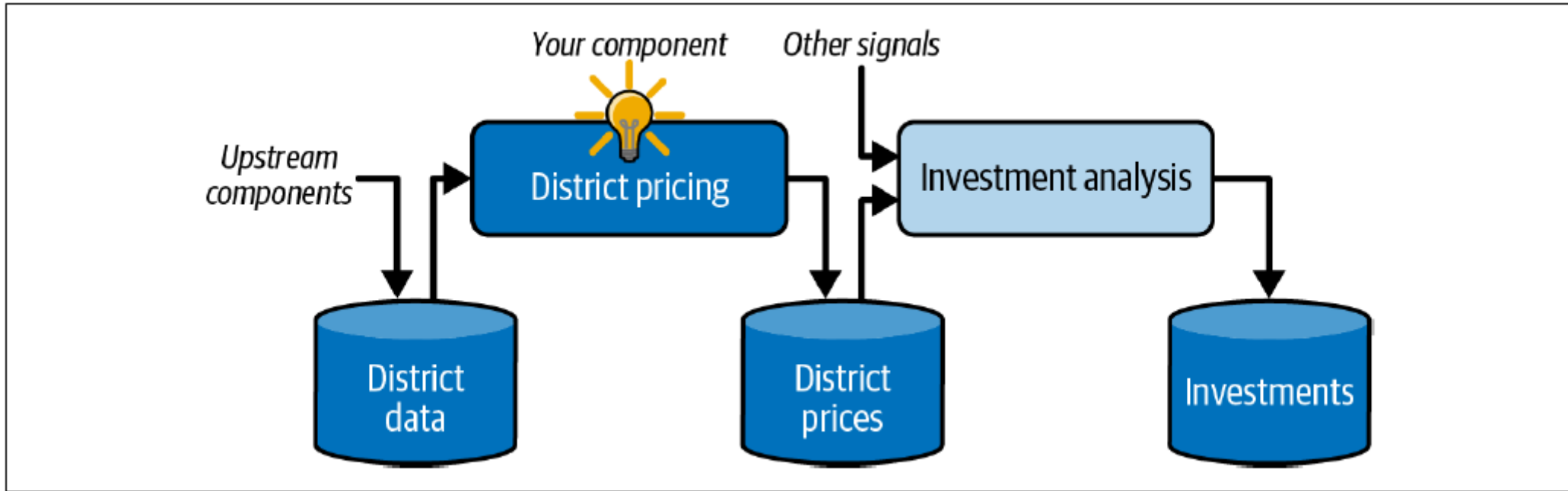


Is it supervised, unsupervised, or Reinforcement Learning?

Is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques?

Instance-based or Model-based learning?

1.1. Frame the Problem



Is it **supervised**, unsupervised, or Reinforcement Learning?

Is it a classification task, a **regression** task, or something else? Should you use **batch** learning or online learning techniques?

Instance-based or **Model-based** learning?

1.2. Select a Performance Measure

- **Root Mean Square Error (RMSE)**

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

- m is the number of samples
- $\mathbf{x}^{(i)}$ is the feature vector of Sample i
- $y^{(i)}$ is the label or desired output
- $\mathbf{X}$ is a matrix containing all the feature values

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

1.2. Select a Performance Measure

- **Mean Absolute Error**

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

- MAE is better than RMSE when there are outlier samples.

Outline

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

2. Get the Data

- The book notebooks can be run on Google Colab at <https://colab.research.google.com/github/ageron/handson-mlp/blob/main/>
- The following slides summarize the code used in Notebook 2.

2. Get the Data

1. Download the `housing.tgz` file from **Github** using `urllib.request.urlretrieve()` from the `urllib` package
2. Extract the data from this compressed tar file using `tarfile.open()` and `extractall()`. The data will be in the CSV file `housing.csv`
3. Read the CSV file into a Pandas DataFrame called `housing` using `pandas.read_csv()`

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", filter="data")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing_full = load_housing_data()
```

2.1. Take a Quick Look at the Data Structure

- Display the top five rows using the DataFrame's `head()` method
- The `info()` method is useful to get a quick description of the data
- To find categories and repetitions of some column use `housing['key'].value_counts()`
- The `describe()` method shows a summary of the numerical attributes.
- Show histogram using the `hist()` method and `matplotlib.pyplot.show()`

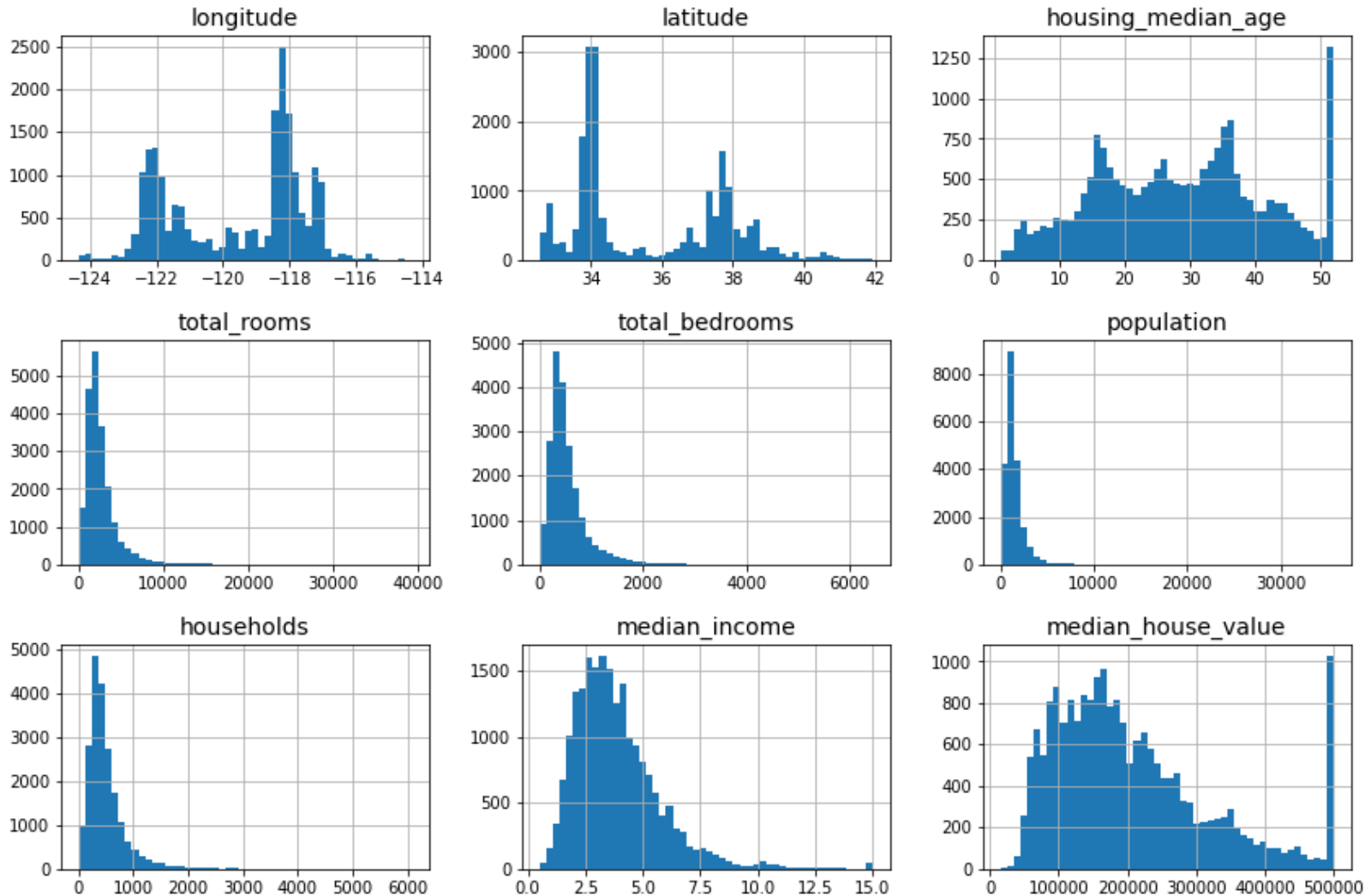
```
housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 20640 entries, 0 to 20639  
Data columns (total 10 columns):  
longitude                20640 non-null float64  
latitude                 20640 non-null float64  
housing_median_age       20640 non-null float64  
total_rooms              20640 non-null float64  
total_bedrooms           20433 non-null float64  
population               20640 non-null float64  
households               20640 non-null float64  
median_income            20640 non-null float64  
median_house_value       20640 non-null float64  
ocean_proximity          20640 non-null object  
dtypes: float64(9), object(1)  
memory usage: 1.6+ MB
```

207 missing
features

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

A histogram for each numerical attribute



2.2. Create a Test Set

- **Split** the available data randomly to:
 - Training set (80%)
 - Test set (20%)

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

- **Stratification** ensures that the test samples are representative of the target categories.

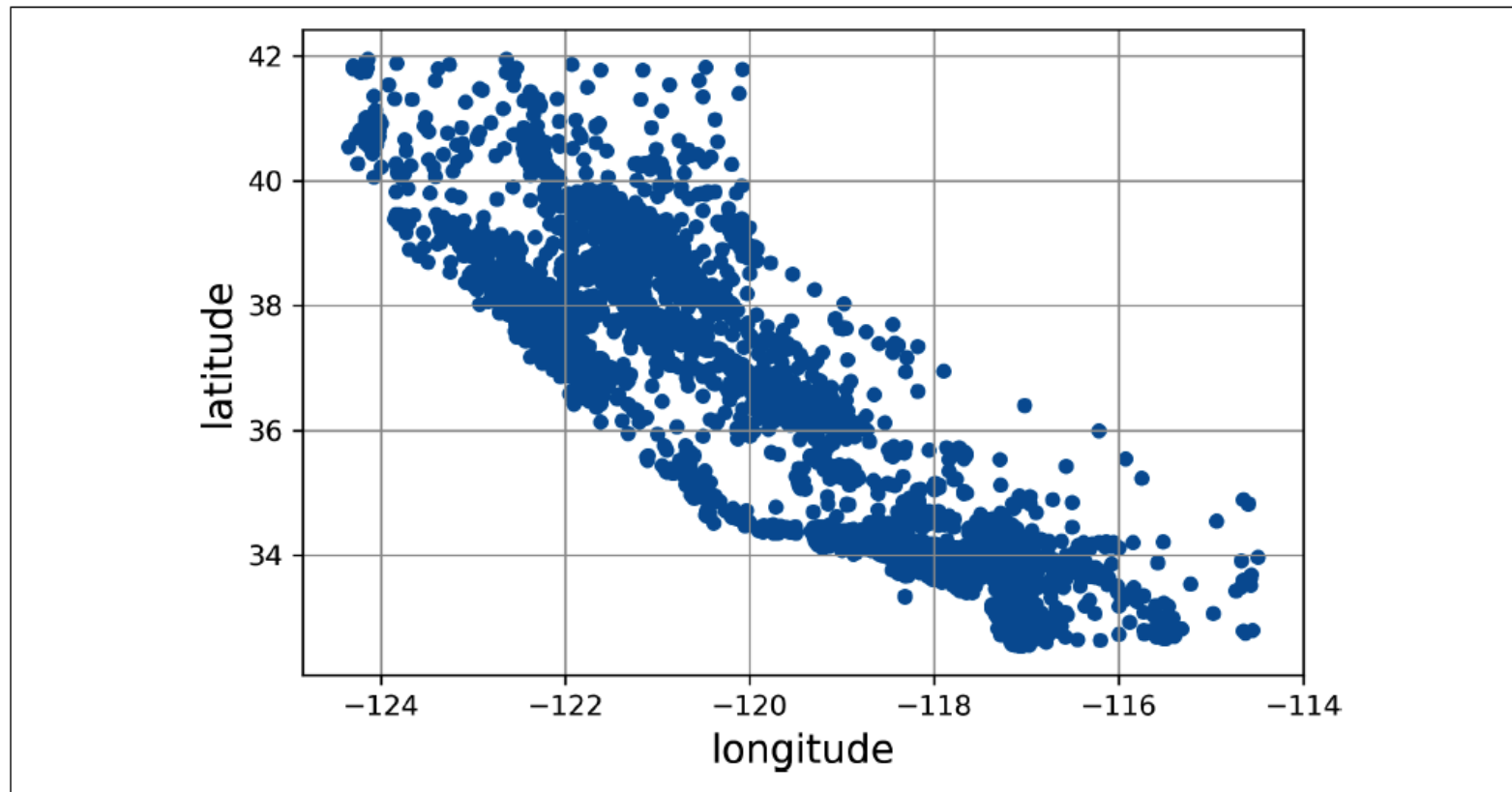
Outline

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

3. Discover and Visualize the Data to Gain Insights

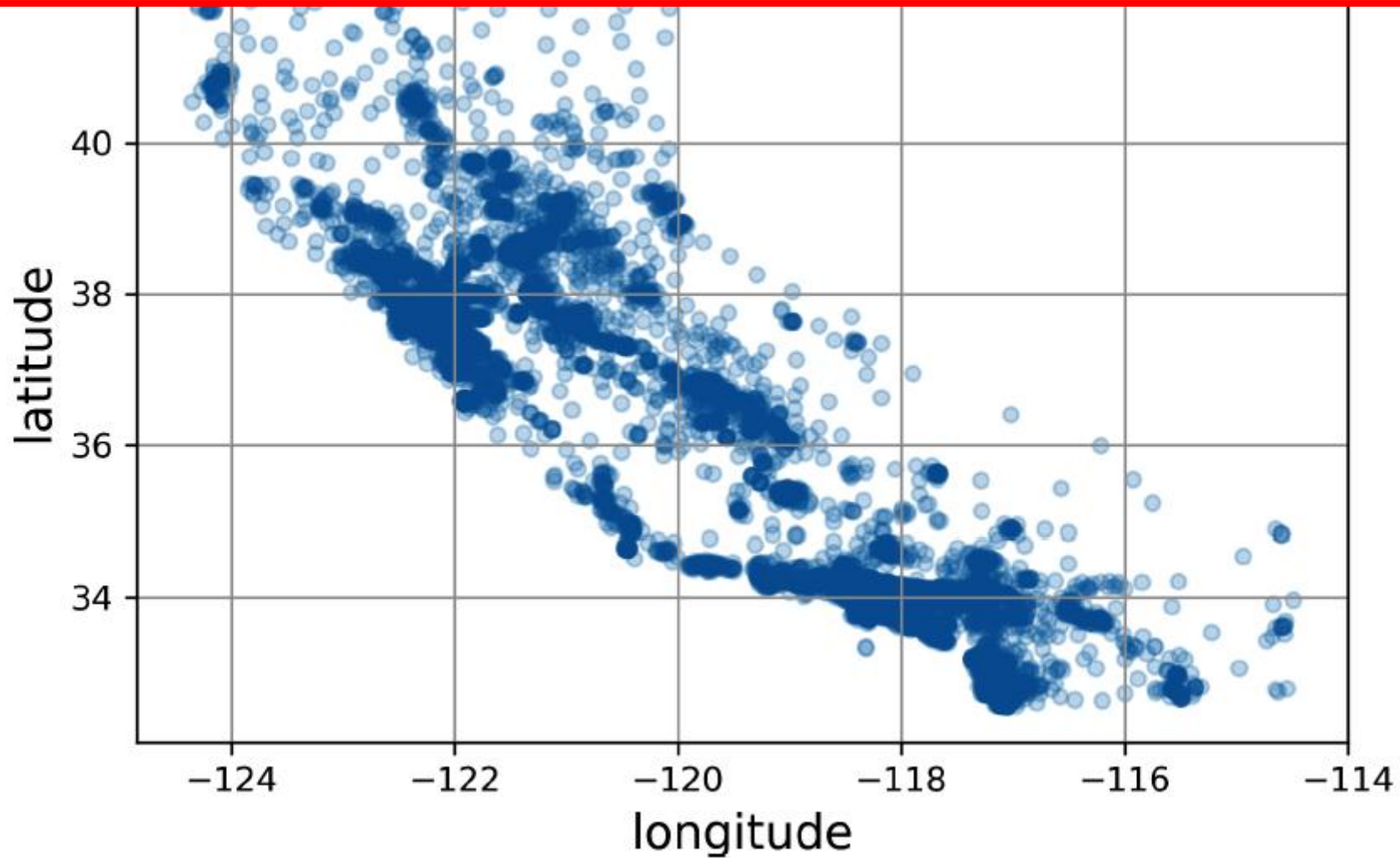
- **Visualize** geographical data using scatter plot:

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



- **Add** transparency.

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



3. Discover and Visualize the Data to Gain Insights

- **Add** size, color, and legend.

```
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()
```

s: size, **c**: color, **cmap**: color map blue to red

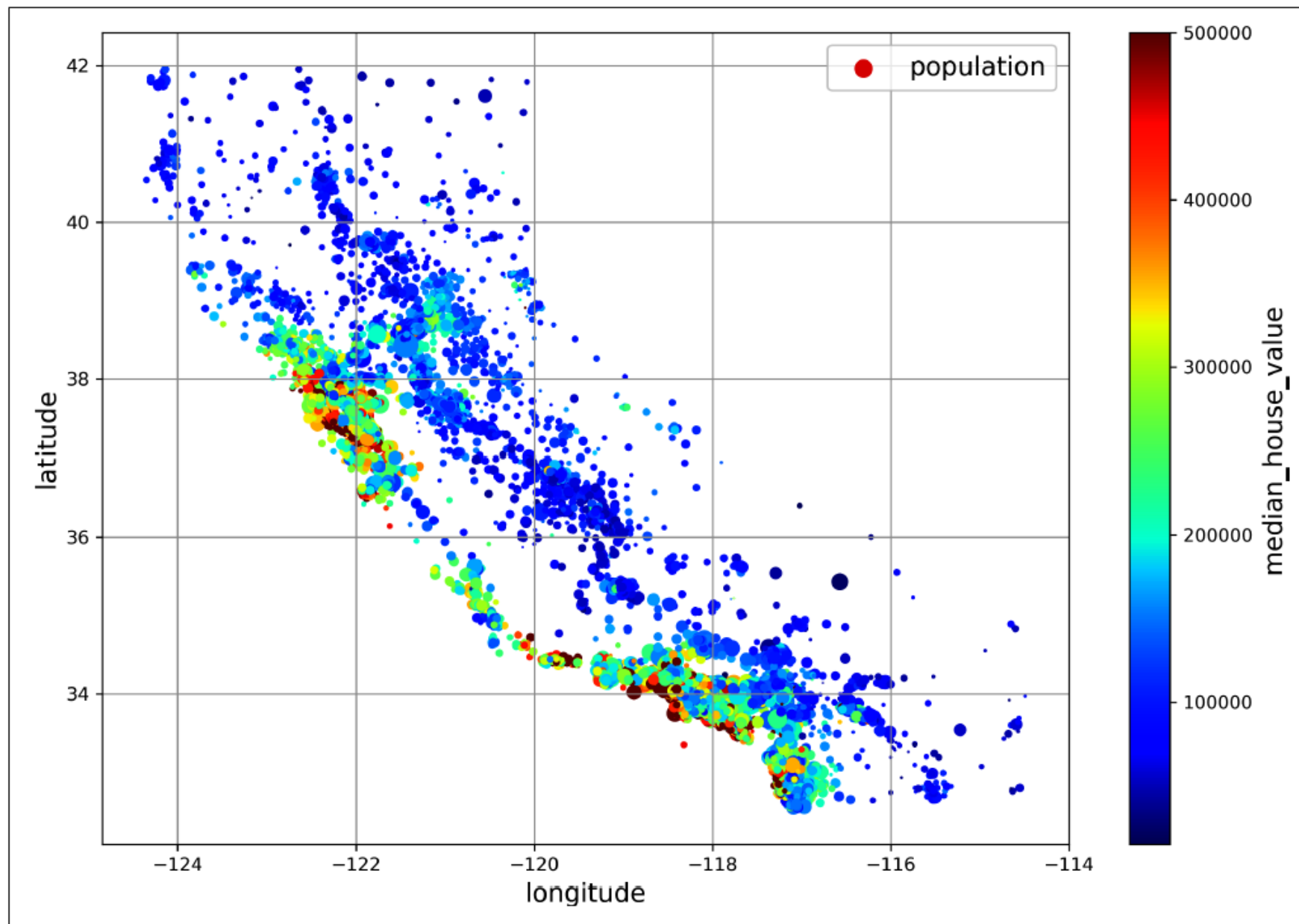


Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

3.1. Looking for Correlations

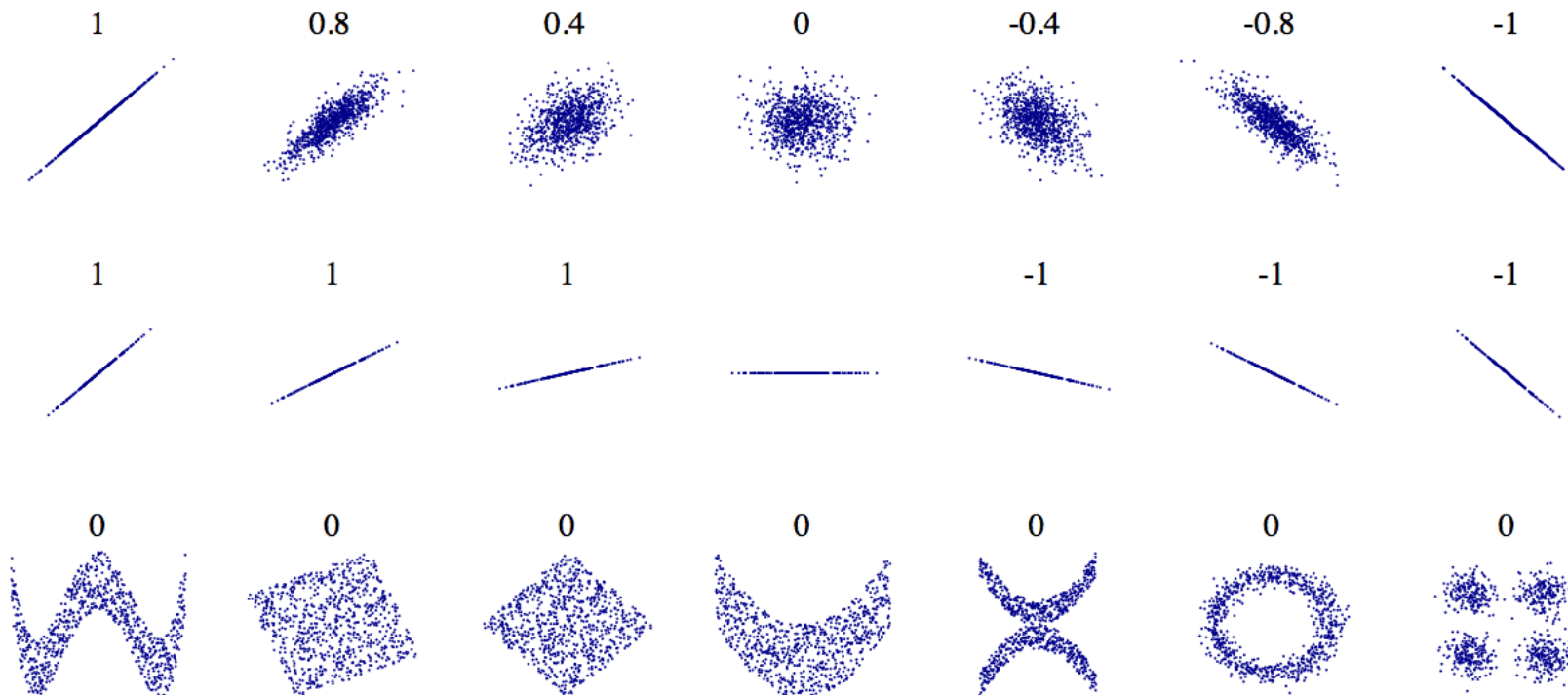
- Compute the **standard correlation coefficient** (also **called Pearson's r**) between every pair of attributes using **`corr_matrix = housing.corr()`**

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population            -0.026699
longitude             -0.047279
latitude              -0.142826
```

3.1. Looking for Correlations

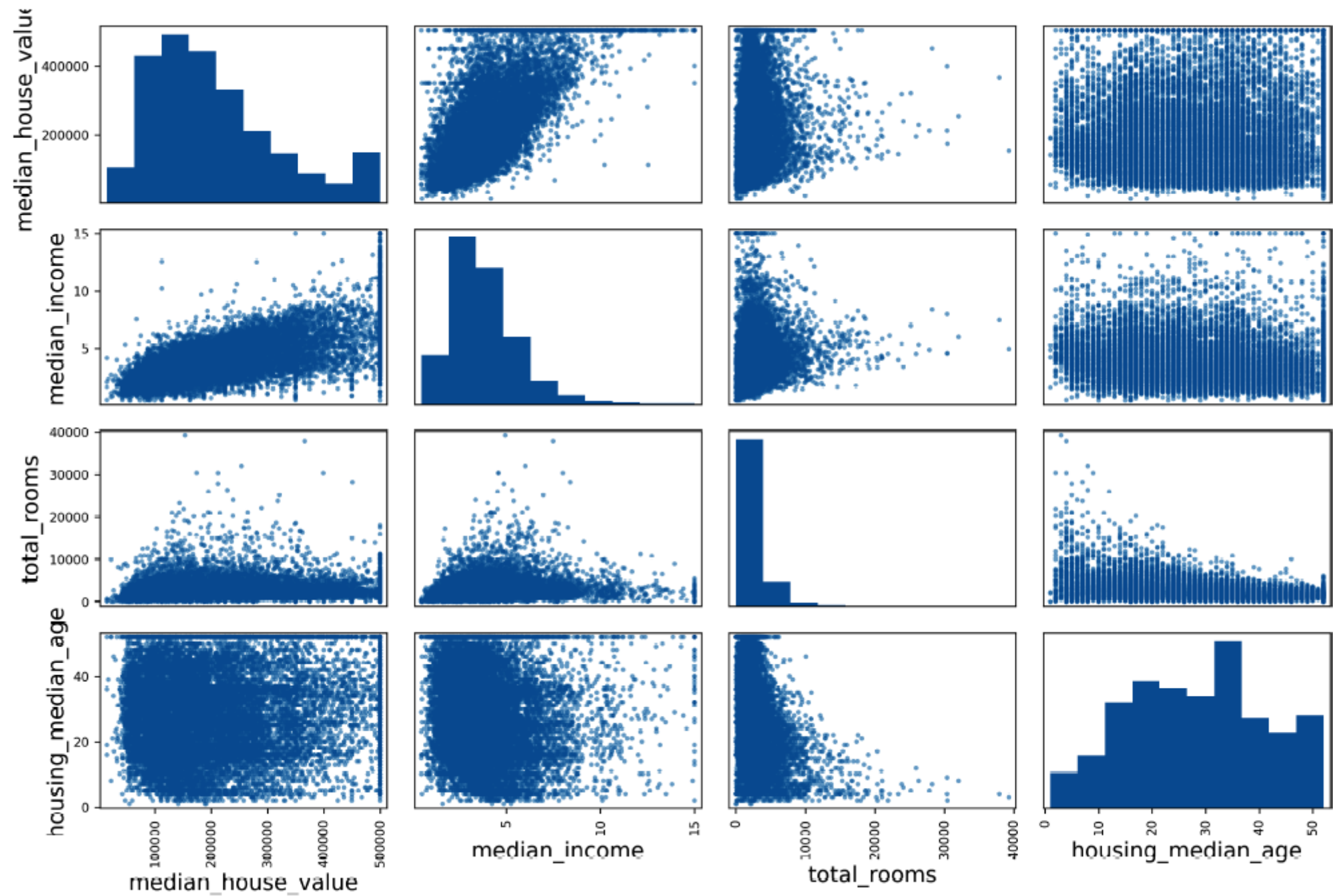
- Zero linear correlation ($r = 0$) does not guarantee **independence**.



3.2. Pandas Scatter Matrix

```
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()
```



3.3. Experimenting with Attribute Combinations

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

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

Outline

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

4. Prepare the Data for Machine Learning Algorithms

- **Split** to train and test (Done)
- **Separate** features from response
- Handle **missing** data
- Handle text and **categorical** features
- **Scale** (normalize) features
- Build preparation **pipeline**

4. Prepare the Data for Machine Learning Algorithms

- **Separate** the **features** from the **response**.

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

- **Options** of handling **missing features**:

1. **Get rid** of the corresponding **districts**
2. **Get rid** of the whole **attribute**
3. **Set the values** to some value (0, mean, median, etc.)

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

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

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

4.1. Handling Missing Features Using Scikit-Learn

- **Scikit-Learn** class design is consistent and simple
 - Estimators **.fit()**
 - Transformers **.transform()**
 - Predictors **.predict()**
 - Inspection **.statistics_**

4.1. Handling Missing Features Using Scikit-Learn

- Use **SimpleImputer** on the numerical features. Need to remove categorical variables before doing the fit. The attribute **statistics_** has the means.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
housing_num = housing.select_dtypes(include=[np.number])
imputer.fit(housing_num)
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])
```

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

Can be used multiple times.

NumPy array

4.2. Handling Text and Categorical Attributes

- **ocean_proximity** is categorical feature.

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
```

	ocean_proximity
17606	<1H OCEAN
18632	<1H OCEAN
14650	NEAR OCEAN
3230	INLAND
3555	<1H OCEAN
19480	INLAND
8879	<1H OCEAN
13685	INLAND
4937	<1H OCEAN
4861	<1H OCEAN

4.2. Handling Text and Categorical Attributes

- Most machine learning algorithms prefer to work with numbers.

Converting to numbers:

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
```

```
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

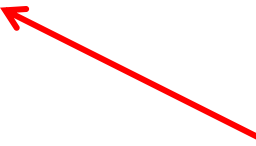
Numerical values
imply distances

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

4.2. Handling Text and Categorical Attributes

- To ensure encoding neutrality, we can use the one-hot encoding.

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> cat_encoder = OneHotEncoder()
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
  with 16512 stored elements in Compressed Sparse Row format>
>>> housing_cat_1hot.toarray()
array([[1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       ...,
       [0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0.]])
```



Converts sparse matrix
to dense matrix.

4.3. Feature Scaling

- ML algorithms generally **don't perform well** when the input numerical attributes have **very different scales**.
- Scaling techniques:
 - **Min-max scaling**
 - **Standardization**
 - **Shrink heavy tails**

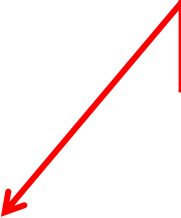
$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

$$x' = \frac{x - \bar{x}}{\sigma}$$

4.3. Feature Scaling

- **Min-max scaling**

```
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)
```



-1 to 1 scaling instead
of 0 to 1 scaling

- **Standardization**

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

4.3. Feature Scaling

- Shrink heavy tails

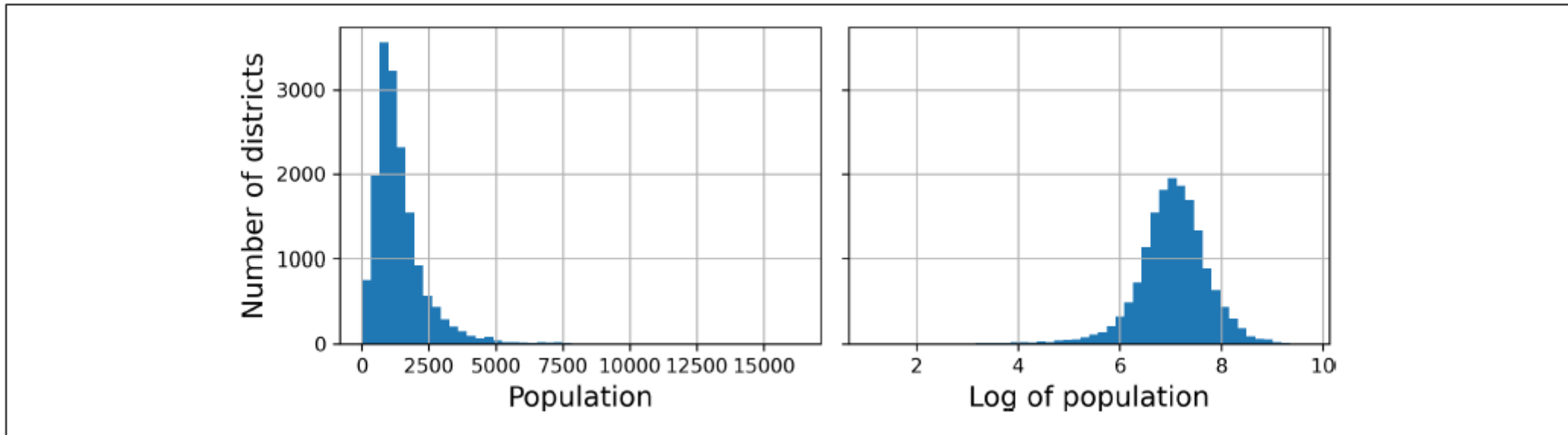


Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

4.4. Custom Transformers

- Scikit-Learn allows you to create your **own transformers**.
- For transformers that don't require any training, use **FunctionTransformer**.

```
from sklearn.preprocessing import FunctionTransformer

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

- Can also use it to **combine features**.

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

4.5. Transformation Pipelines



```
from sklearn.pipeline import Pipeline

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

4.5. Transformation Pipelines

- If you don't want to name the transformers, you can use the `make_pipeline()` function,

```
from sklearn.pipeline import make_pipeline
```

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

4.6. Full Pipeline

- Can combine multiple pipeline paths 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),
])
```

4.6. Full Pipeline

- Can automate attribute selection using `make_column_selector()` and `make_column_transformer()` to avoid naming the pipelines.

```
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)),
)
```

- Using the full pipeline is straight forward.

```
housing_prepared = preprocessing.fit_transform(housing)
```



Dense array

Full Pipeline Used in the Reference Book (1/3)

```
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())
```

Full Pipeline Used in the Reference Book (2/3)

```
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
```

Full Pipeline Used in the Reference Book (3/3)

```
>>> 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)
```

Outline

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

5. Select and Train a Model

- 1) **Linear regressor**
- 2) Using **RMSE** for evaluation
- 3) **Decision tree regressor**
- 4) **k-fold** cross validation
- 5) **Random forests regressor**

5. Select and Train a Model

- Let us start by training a simple **linear regressor**.

```
from sklearn.linear_model import LinearRegression

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

- Try it out on five instances from the training set.

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

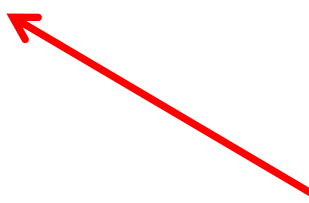
47% off

9% off

5.1. Evaluate the Model on the Entire Training Set

- Use RMSE

```
>>> from sklearn.metrics import root_mean_squared_error
>>> lin_rmse = root_mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse
68972.88910758484
```



This is not a satisfactory result as the **median_housing_values** range between \$120,000 and \$265,000.

5.2. Try the Decision Tree Regressor

```
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 = root_mean_squared_error(housing_labels, housing_predictions)
```

```
...
>>> tree_rmse
0.0
```



Overfitting: It has memorized
the entire training set!

5.3. Better Evaluation Using Cross-Validation

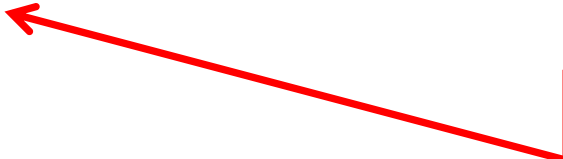
- Segment the training data into **10 sets** and repeat training and evaluation 10 times.

```
from sklearn.model_selection import cross_val_score

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

```
>>> pd.Series(tree_rmse).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
```



Not much better than the
Linear Regressor

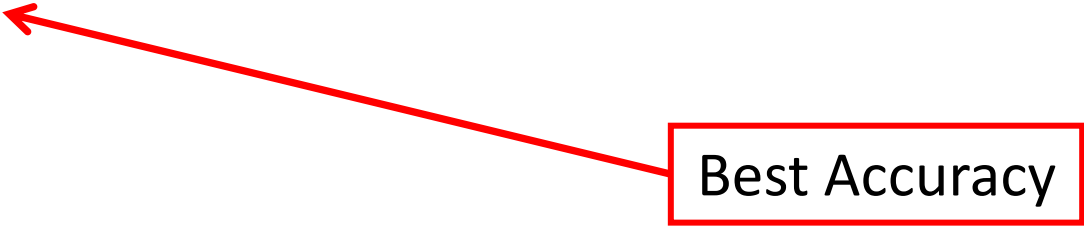
5.4. Try the Random Forests Regressor

- Repeating training and evaluation:

```
from sklearn.ensemble import RandomForestRegressor

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

>>> pd.Series(forest_rmse).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
```



Best Accuracy

Outline

1. Look at the big picture
2. Get the data
3. Discover 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**
- 9. Exercises**

6. Fine-Tune Your Model

- Fine-tune your system by fiddling with:
 - The hyperparameters
 - Removing and adding features
 - Changing feature preprocessing techniques
- Can experiment manually. But it is best to automate this process using Scikit-Learn:
 - **GridSearchCV**
 - or **RandomizedSearchCV**

6.1. Grid Search

- Can automate exploring a search space of $3 \times 3 + 2 \times 3 = 9 + 6 = 15$

```
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)
```

6.2 Examine the Results of Your Grid Search

- Can examine the best hyperparameters using:

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

- Can examine all search results using:

```
>>> 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
7	10	6	44683	44655	45657	44999
9	10	6	44683	44655	45657	44999

Dictionary

Best Tuned
Accuracy

6.2 Evaluate Your System on the Test Set

- The final model is the best estimator found by the grid search.

```
>>> final_model = rnd_search.best_estimator_ # includes preprocessing
```

- To evaluate it on the test set, transform the test features, predict using transformed features, and evaluate accuracy.

```
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 = root_mean_squared_error(y_test, final_predictions) #d=False)  
print(final_rmse) # prints 41445.533268606625
```

6.3 Save Your Best Model for the Production System

- Save your best model:

```
import joblib
```

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

- Load and use your model in the production system:

```
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)
```

Outline

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

7. Present Your Solution

- Present your solution highlighting:
 - What you have learned
 - What worked and what did not
 - What assumptions were made
 - What your system's limitations are
- Document everything, and create nice presentations with:
 - Clear visualizations
 - Easy-to-remember statements, e.g., “the median income is the number one predictor of housing prices”.

8. Launch, Monitor, and Maintain Your System

- Prepare your production program that uses your best trained model and launch it.
- Monitor the accuracy of your system. Also monitor the input data.
- Retrain your system periodically using fresh data.

Summary

1. Look at the big picture
2. Get the data
3. Discover 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
9. Exercises

Exercise

- Try a Support Vector Machine regressor (`sklearn.svm.SVR`), with various hyperparameters such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Note that support vector machines don't scale well to large datasets, so you should probably train your model on just the first 5,000 instances of the training set and use only 3-fold cross-validation, or else it will take hours. Don't worry about what the hyperparameters mean for now; we'll discuss them in Chapter 5. How does the best SVR predictor perform?