Second chapter - End to End Machine learning problem
Welcome to the Housing corporate machine learning!

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

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.

1. look at the big picture

- Framing the problem:

a. The first question to ask your boss is what exactly the business objective is.
The answer: Your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system

b. The next question to ask your boss is what the current solution looks like (if any).
The answer: district housing prices are currently estimated manually by experts and they often realize that their estimates were off by more than 20%.
c. You need to 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?
The answer: it is clearly a typical **supervised** learning task, since you are given *labeled* training examples . It is also a typical **regression** task, since you are asked to predict a value.Finally, there is no continuous flow of data coming into the system and the data is small enough to fit in memory, so **plain batch learning** should do just fine.

- Select a performance measure - (cost function)

a typical performance measure for regression problems: Root Mean Square Error (RMSE)

Suppose that there are many outlier districts, in that case, you may consider using the *mean absolute error* (MAE, also called the average absolute deviation;)

quiz:

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

2. Get the Data

The full Jupyter notebook is available at *https://github.com/ageron/handson-ml2*.
You could use your web browser to download the file and run tar xzf housing.tgz to decompress it and extract the CSV file, but it is preferable to create a small function to do that.
Fetch the data:

```
import os
import tarfile
import urllib

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

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
os.makedirs(housing_path, exist_ok=True)
tgz_path = os.path.join(housing_path, "housing.tgz")
urllib.request.urlretrieve(housing_url, tgz_path)
housing_tgz = tarfile.open(tgz_path)
housing_tgz.extractall(path=housing_path)
housing_tgz.close()
```

Now let's load the data using pandas. Once again, you should write a small function to load the data:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
csv_path = os.path.join(housing_path, "housing.csv")
return pd.read_csv(csv_path)
```

fetch_housing_data()

housing = load_housing_data()

3.Take a Quick Look at the Data Structure

take a look at the top five rows using the DataFrame's head() method

```
housing.head()
```

The info() method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of non null values

```
housing.info()
```

All attributes are numerical, except the ocean_proximity field. You can find out what categories exist and how many districts belong to each category by using the value_counts() method:

```
housing.ocean_proximity.value_counts()
```

Let's look at the other fields. The describe() method shows a summary of the numerical attributes.
Percentile is a value below which a given percentage of observations in a group of observations falls.For example, 25% of the districts have a housing_median_age lower than 18

```
housing.describe()
```

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis).

```
%matplotlib inline # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

4. Creat a test set

To avoid **DATA SNOOPING BIAS**, we put voluntarily set aside part of the data at this stage. Creating a test set is theoretically simple: pick some instances randomly, typically 20% of the dataset and set them aside

```
import numpy as np

def split_train_test(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]
```

Scikit learn also has a package for this purpose:

```
from sklearn.model_selection import train_test_split

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

Purely ***random sampling*** method may cause significant sampling bias in case the dataset is not large enough. When we want to the data to be representative we need to use ***stratified sampling.*** the population is divided into homogeneous subgroups called *strata*, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population.

Stratifies sampling for our district price predictions:

Suppose the median income is a very important attribute to predict median housing prices. To ensure that the test set is representative of the various categories of incomes in the whole dataset, we need to create an income category attribute first and do stratified sampling based on the income category.

The following code uses the pd.cut() function to create an income category attribute with five categories (labeled from 1 to 5)

```
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])
```

Scikit-Learn's StratifiedShuffleSplit class:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
strat_train_set = housing.loc[train_index]
strat_test_set = housing.loc[test_index]
```

removing income_cat from features because we don't need it any more:

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

5. Discover and Visualize the Data to Gain Insights

make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast. In our case, the set is quite small, so you can just work directly on the full set. Let's create a copy so that you can play with it without harming the training set:

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

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
s=housing["population"]/100, label="population", figsize=(10,7),
c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```

Looking for Correlations

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

```
>>> 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
Name: median_house_value, dtype: float64
```

Another way to check correlation between attributes:

to not lose the nonlinear correlations only based on the ratios

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
"housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

6. Experimenting with Attribute Combinations

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]
```

7. Prepare the Data for Machine Learning Algorithms

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

8. Data Cleaning

Scikit-Learn provides a handy class to take care of missing values: SimpleImputer

```
from sklearn.impute import SimpleImputer

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

Since the median can only be computed on numerical attributes, you need to create a copy of the data without the categorical attribute ocean_proximity:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

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

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

Then put it back to DataFrame format:

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

Estimators

Any object that can estimate some parameters based on a dataset is called an *estimator* (e.g., an imputer is an estimator). The estimation itself is performed by the fit() method, and it takes only a dataset as a parameter (or two for supervised learning algorithms; the second dataset contains the labels). Any other parameter needed to guide the estimation process is considered a hyperparameter (such as an imputer's strategy), and it must be set as an instance variable (generally via a constructor parameter).

Transformers

Some estimators (such as an imputer) can also transform a dataset; these are called *transformers*. Once again, the API is simple: the transformation is performed by the transform() method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for an imputer. All transformers also have a convenience method

called fit_transform() that is equivalent to calling fit() and then transform() (but sometimes fit_transform() is optimized and runs much faster).

Predictors

Finally, some estimators, given a dataset, are capable of making predictions; they are called *predictors*. For example, the LinearRegression model in the previous chapter was a predictor: given a country's GDP per capita, it predicted life satisfaction. A predictor has a predict() method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a score() method that measures the quality of the predictions, given a test set (and the corresponding labels, in the case of supervised learning algorithms).

Inspection

All the estimator's hyperparameters are accessible directly via public instance variables (e.g., imputer.strategy), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., imputer.statistics_).

Handling text and categorical variables

Most Machine Learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers:

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

TIP

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the ocean_proximity feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you could replace each category with a learnable, low-dimensional vector called an *embedding*. Each category's representation would be learned during training.

9. Feature scaling

With few exceptions, Machine Learning algorithms don't perform well when the input numerical attributes have very different scales. Note that scaling the target values is generally not required.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

Min-max scaling (many people call this ***normalization***) is the simplest:
values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus the min.
Scikit-Learn provides a transformer called MinMaxScaler for this.

Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem

for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called StandardScaler for standardization.

10. Transformation pipelines

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
('imputer', SimpleImputer(strategy="median")),
('attribs_adder', CombinedAttributesAdder()),
('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

and how to apply transformers pipeline to both cat an num column at one place:

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
("num", num_pipeline, num_attribs),
("cat", OneHotEncoder(), cat_attribs),
])

housing_prepared = full_pipeline.fit_transform(housing)
```

Note:

OneHotEncoder returns a sparse matrix, while the num_pipeline returns a dense matrix. When there is such a mix of sparse and dense matrices, the ColumnTransformer estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, sparse_threshold=0.3) otherwise it returns a dense matrix. In this example, it returns a dense matrix. And that's it! We have a preprocessing pipeline that takes the full housing data and applies the appropriate transformations to each column.

11. Select and train a model

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

what is the accuracy?

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.19819848922
```

Not very promising, no?! Books tries different algorithms like decision-tree, random forest and find the k fold validation score and compare them together(check codes GitHub or ask me to send them to you!) The final model was random forest:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
18603.515021376355
```

```
>>> display_scores(forest_rmse_scores)
Scores: [49519.80364233 47461.9115823 50029.02762854 52325.28068953
49308.39426421 53446.37892622 48634.8036574 47585.73832311
53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

12. Fine-Tune Your Model

Grid Search

Scikit-Learn's GridSearchCV is a good approach to search for the best hyper parameters. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the best combination of hyperparameter values for the RandomForestRegressor:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
{'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
{'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring='neg_mean_squared_error',
return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

TIP

When you have no idea what value a hyperparameter should have, a simple approach is to try out consecutive powers of 10 (or a smaller number if you want a more fine-grained search.

Randomized Search

The grid search approach is fine when you are exploring relatively few combinations,, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCVinstead. This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration. This approach has two main benefits:

- If you let the randomized search run for, say, 1,000 iterations, this approach will explore 1,000 different values for each hyperparameter (instead of just a few values per hyperparameter with the grid search approach).
- Simply by setting the number of iterations, you have more control over the computing budget you want to allocate to hyperparameter search.

Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model (just like Random Forests perform better than the individual Decision Trees they rely on), especially if the individual models make very different types of errors. We will cover this topic in more detail in Chapter 7.

13. Analyze the Best Models and Their Errors

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

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])
```

Let's display these importance scores next to their corresponding attribute names:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = full_pipeline.named_transformers_["cat"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.3661589806181342, 'median_income'),
(0.1647809935615905, 'INLAND'),
(0.10879295677551573, 'pop_per_hhold'),
(0.07334423551601242, 'longitude'),
(0.0629090704826203, 'latitude'),
(0.05641917918195401, 'rooms_per_hhold'),
(0.05335107734767581, 'bedrooms_per_room'),
(0.041143798478729635, 'housing_median_age'),
(0.014874280890402767, 'population'),
(0.014672685420543237, 'total_rooms'),
(0.014257599323407807, 'households'),
(0.014106483453584102, 'total_bedrooms'),
(0.010311488326303787, '<1H OCEAN'),
(0.002856474637320158, 'NEAR OCEAN'),
(0.00196041559947807, 'NEAR BAY'),
(6.028038672736599e-05, 'ISLAND')]
```

14. Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. Now is the time to evaluate the final model on the test set

```
final_model = grid_search.best_estimator_

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

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)
```

```
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse) # => evaluates to 47,730.2
```

15. Launch, Monitor, and Maintain Your System

polish the code, write documentation and tests
Then you can deploy your model to your production environment.
One way to do this is to save the trained Scikit-Learn model (e.g., using joblib), including the full preprocessing and prediction pipeline, then load this trained model within your production environment and use it to make predictions by calling its predict() method.
For example, perhaps the model will be used within a website: the user will type in some data about a new district and click the Estimate Price button. This will send a query containing the data to the web server, which will forward it to your web application, and finally your code will simply call the model's predict() method (you want to load the model upon server startup, rather than every time the model is used).
Alternatively, you can wrap the model within a dedicated web service that your web application can query through a REST API[23] (see Figure 2-17). This makes it easier to upgrade your model to new versions without interrupting the main application. It also simplifies scaling, since you can start as many web services as needed and load-balance the requests coming from your web application across these web services. Moreover, it allows your web application to use any language, not just Python.

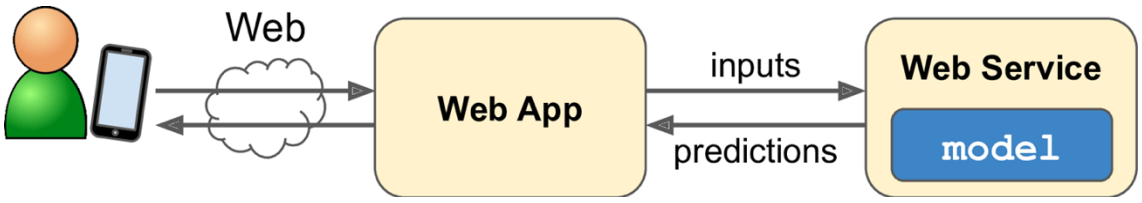

Figure 2-17. A model deployed as a web service and used by a web application

Another popular strategy is to deploy your model on the cloud, for example on Google Cloud AI Platform (formerly known as Google Cloud ML Engine): just save your model using joblib and upload it to Google Cloud Storage (GCS), then head over to Google Cloud AI Platform and create a new model version, pointing it to the GCS file. That's it! This gives you a simple web service that takes care of load balancing and scaling for you. It takes JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions. You can then use this web service in your website (or whatever production environment you are using). As we will see in Chapter 19, deploying TensorFlow models on AI Platform is not much different from deploying Scikit-Learn models.

But deployment is not the end of the story!
You also need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops. This could be a steep drop, likely due to a broken component in your infrastructure, but be aware that it could also be a gentle decay that could easily go unnoticed for a long time. This is quite common because models tend to "rot" over time: indeed, the world changes, so if the model was trained with last year's data, it may not be adapted to today's data.

So you need to monitor your model's live performance. But how do you that? Well, it depends. In some cases, the model's performance can be inferred from downstream metrics. For example, if your model is part of a recommender system and it suggests products that the users may be interested in, then it's easy to monitor the number of recommended products sold each day. If this number drops (compared to non-recommended products), then the prime suspect is the model. This may be because the data pipeline is broken, or perhaps the model needs to be retrained on fresh data (as we will discuss shortly).

However, it's not always possible to determine the model's performance without any human analysis. For example, suppose you trained an image classification model (see Chapter 3) to detect several product defects on a production line. How can you get an alert if the model's performance drops, before thousands of defective products get shipped to your clients? One solution is to send to human raters a sample of all the pictures that the model classified (especially pictures that the model wasn't so sure about). Depending on the task, the raters may need to be experts, or they could be nonspecialists, such as workers on a crowdsourcing platform (e.g., Amazon Mechanical Turk). In some applications they could even be the users themselves, responding for example via surveys or repurposed captchas.[24]

Either way, you need to put in place a monitoring system (with or without human raters to evaluate the live model), as well as all the relevant processes to define what to do in case of failures and how to prepare for them.

If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should probably automate the whole process as much as possible. Here are a few things you can automate:

- Collect fresh data regularly and label it (e.g., using human raters).
- Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.
- Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why).

You should also make sure you evaluate the model's input data quality. Sometimes performance will degrade slightly because of a poor-quality signal (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale)

Finally, make sure you keep backups of every model you create and have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly for some reason. Having backups also makes it possible to easily compare new models with previous ones. Similarly, you should keep backups of every version of your datasets so that you can roll back to a previous dataset if the new one ever gets corrupted (e.g., if the fresh data that gets added to it turns out to be full of outliers). Having backups of your datasets also allows you to evaluate any model against any previous dataset.

Try It Out!

Now is a good time to pick up a laptop, select a dataset that you are interested in, and try to go through the whole process from A to Z. A good place to start is on a competition website such as *http://kaggle.com/*: you will have a dataset to play with, a clear goal, and people to share the experience with. Have fun!