End-to-End Machine Learning Project

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

1. Look at the big picture.
2. Get the data.
3. Explore and visualize the data to gain insights.
4. Prepare the data for machine learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Working with Real Data

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

Look at the Big Picture

- Your task is to use California census data to build a model of housing prices in the state.
- This data includes metrics such as the population, median income, and median housing price for each block group in California.
- Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). I will call them "districts" for short.
- Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

Frame the Problem

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

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

Pipelines

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

Select a performance Measure

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

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

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

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

Equation 2-2. Mean absolute error (MAE)

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

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

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

Notations

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

Check the Assumptions

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

Get the Data

Create the Workspace

Download the Data

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

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

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

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

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

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

```
import pandas as pd

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

Take a Quick look at the Data Structure

- Take a look at the structure of the dataset using **head()** and **sample()** methods
- The **info()** method is useful to get a quick description of data, in particular the total no. of rows, each attribute's type, and the number of nonnull values.
- **Describe()** method shows a summary of the numerical attributes.
- Attribute of type **object** can hold any kind of Python object. But since you loaded this data from a CSV file, you know that it must be a text attribute.
- If we have a categorical attribute, **value_counts()** method can find out what categories exist and how many rows belong to each category
-

Create a Test Set

-

Discover and Visualize the Data to Gain Insights

-

Visualizing Geographical Data

-

Looking for Correlations

-

Experimenting with Attribute Combinations

-

Prepare the Data for Machine Learning Algorithms

-

Data Cleaning

-

Scikit-learn Design

-

Handling Text and Categorical Attributes

-

Custom Transformers

-

Feature Scaling

-

Transformation Pipelines

- Select and Train a Model
- Training and Evaluating on the Training Set
- Better Evaluation Using Cross-Validation
- Fine-Tune Your Model
- Grid Search
- Randomized Search
- Ensemble Methods
- Analyze the Best Models and Their Errors
- Evaluate Your System on the Test Set
- Launch, Monitor ,and Maintain Your System
- Exercises