Chapter 1: The Machine Learning Landscape

1.1. What is Machine Learning?

Machine learning is the science (and art) of programming computers so they can learn from data.

> *[Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.*
> —Arthur Samuel, 1959

> *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*
> —Tom Mitchell, 1997

1.2. Why use Machine Learning?

Machine Learning is great for:

- Problems for which existing solutions require a lot of hand-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better.
- Complex problems for which there is no good solution at all using a traditional approach: the best Machine Learning techniques can find a solution.
- Fluctuating environments: a Machine Learning system can adapt to new data.
- Getting insights about complex problems and large amounts of data.

Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called *data mining*.

1.3. Types of Machine Learning Systems

1.3.1. Supervised / Unsupervised Learning

Machine Learning systems can be classified according to the amount and type of supervision they get during training.

In *supervised learning*, the training data you feed to the algorithm includes the desired solutions, called labels. Some important supervised learning algorithms:

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression (*commonly used for classification*)
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural networks (*some NN architecture like autoencoders and restricted Boltzmann machines can be unsupervised, some architecture like deep belief networks and unsupervised pretraining can be semi-supervised*)

In *unsupervised learning*, the training data is unlabelled. The system tries to learn without a teacher. Some important unsupervised learning algorithms are:

- Clustering
    - K-Means
    - DBSCAN
    - Hierarchical Cluster Analysis (HCA)
- Anomaly detection and novelty detection
    - One-class SVM
    - Isolation Forest
- Visualization and dimensionality reduction
    - Principal Component Analysis (PCA)
    - Kernel PCA
    - Locally-Linear Embedding (LLE)
    - t-distributed Stochastic Neighbour Embedding (t-SNE)
- Association rule learning
    - Apriori
    - Eclat

Some algorithms can deal with partially labelled training data, usually a lot of unlabelled data and a little bit of labelled data. This is called *semi-supervised learning*. Example: Google Photos.

Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, deep belief networks (DBNs) are based on unsupervised components called restricted Boltzmann machines (RBMs) stacked on top of one another.

In *Reinforcement Learning*, the learning system, called an *agent* in this context, can observe the environment, select and perform actions, and get *rewards* in return (or *penalties* in the form of negative rewards). It must then learn by itself what is the best strategy, called a *policy*, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

1.3.2. Batch and Online Learning

In *batch learning*, the system is incapable of learning incrementally: it must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called *offline learning*.

If we want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then stop the old system and replace it with the new one.

Batch learning is not suitable in cases:

- System needs to adapt to rapidly changing data
- Training on the full set of data requires a lot of computing resources
- System needs to be able to learn autonomously and it has limited resources.

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or by small groups called mini-batches. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives.

Online learning is great for:

- Systems that receive data as a continuous flow and need to adapt to changes rapidly or autonomously.
- Limited computing resources (huge amount of memory can be saved by discarding the data on which the model is already trained and learned)
- Training systems on huge datasets that cannot fit in one machine's main memory (*out-of-core learning*). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data.

Note: Out-of-core learning is usually done offline (i.e., not on the live system), so online learning can be a confusing name. Think of it as *incremental learning*.

One important parameter of online learning system is *learning rate*: how fast they should adapt to changing data:

- *High learning rate*: System will rapidly adapt to new data, but it will also tend to quickly forget the old data.
- *Low learning rate*: System will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points (outliers).

A big challenge with online learning: if bad data is fed to the system, the system's performance will gradually decline. In case of a live system, users will notice the degrading performance and business will be affected.

1.3.3. Instance-based v/s Model-based Learning

One more way to categorize Machine Learning systems is by how they generalize. There are two main approaches to generalization: instance-based learning and model-based learning.

In *instance-based learning*, the system learns the examples by heart, then generalizes to new cases by comparing them to the learned examples (or a subset of them), using a similarity measure.

Take an example of a spam filter: instead of just flagging emails that are identical to known spam emails, we can program a spam filter to also flag emails that are very similar to known spam emails. This requires a *measure of similarity* between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.

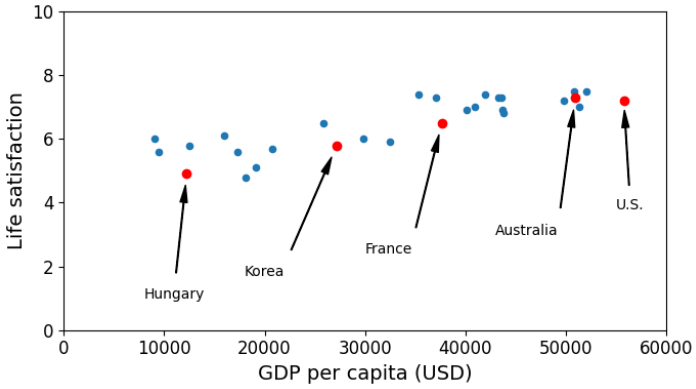
In a *model-based learning*, we build a model of these examples, and then we use that model to make *predictions*.

Let us take the below example.

We want to know if money makes people happy. We use the *Better Life Index* data from the OECD's website as well as stats about *GDP per capita* from the IMF's website.

The code is given in `01_the_machine_learning_landscape.ipynb`.

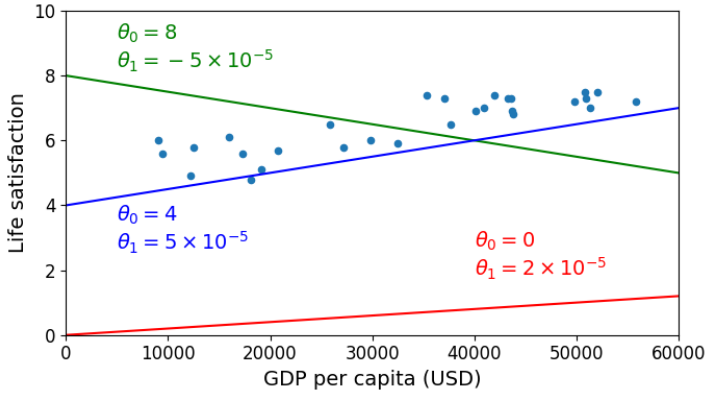
- Let's plot the data for a few random countries:



- There does seem to be a trend here! Although the data is noisy (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So, you decide to model life satisfaction as a linear function of GDP per capita. This step is called *model selection*: you selected a *linear model* of life satisfaction with just one attribute, GDP per capita.

$$life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$$

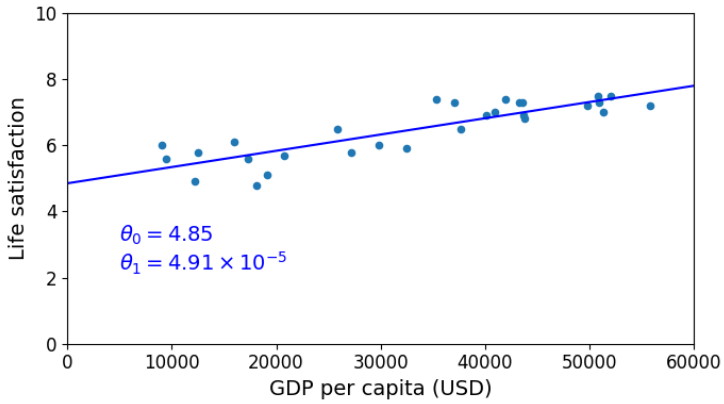
- This model has two model parameters, θ_0 and θ_1. By tweaking these parameters, we can make our model represent any linear function:



- We need to specify a performance measure to define the parameter values θ_0 and θ_1, to know for which values, the model will perform best. We can either define a *utility function* (or *fitness function*) that measures how good your model is, or you can define a *cost function* that measures how bad it is.
  For linear regression problems, people typically use a cost function that measures the distance between the linear model's predictions and the training examples; the objective is to minimize this distance.

- This is where the Linear Regression algorithm comes in: you feed it your training examples and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case the algorithm finds that the optimal parameter values are $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$.

  Now the model fits the training data as closely as possible (for a linear model).



- We are finally ready to run the model to make predictions. For example, if we want to know how happy Cypriots are, (OECD data does not have the answer), we can use our model to make a good prediction: we look up Cyprus's GDP per capita, find $22,587, and then apply your model and find that life satisfaction is likely to be around $4.85 + 22{,}587 \times 4.91 \times 10^{-5} = 5.96$.

Note: If we had used an *instance-based learning* algorithm instead, we would have found that Slovenia has the closest GDP per capita to that of Cyprus ($20,732), and since the OECD data tells us that Slovenians' life satisfaction is 5.7, we would have predicted a life satisfaction of 5.7 for Cyprus.

If we zoom out a bit and look at the two next closest countries, we will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. Averaging these three values, you get 5.77, which is pretty close to your model-based prediction.

This simple algorithm is called *k-Nearest Neighbors regression* (in this example, k = 3).