# Module 1: The Machine Learning Landscape

**Video: 670_mod1_vid1**

## What is Machine Learning?

Machine learning is the science, and art, of programming computers so they can *learn from data*. Naturally, the next question is: What does it mean for a computer to *learn from data*? In my opinion, a machine can learn from data when it is able to find patterns in that data, and build a model that describes those fundamental patterns underlying the data. (This is actually a poor definition, because it completely neglects a whole field of ML called instance-based learning, such as the K-Nearest Neighbors Algorithm.)

A more engineering-oriented definition given by Tom Mitchell states, "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."

Consider an email spam filter. In this case, the task T is to flag spam for new emails that are coming in, the experience E is the training data - that is a set of actual spam emails, and the performance measure P could be the ratio of correctly classified emails. The ratio of correctly classified entities to the number of all classified entities is called ***accuracy***, and is a very popular performance measure.

The example data that the system learns from are called the ***training set***. An individual datum point in this training set is called a ***training instance***, ***training sample***, or an ***observation***.

**Video: 670_mod1_vid2**

## Why use Machine Learning?

Some of the most frequently used machine learning techniques are nothing more than high-level statistical modeling tools, such as linear and logistic regression. These tools are widely used to make predictions based on data.

For example, consider the Snapshot program by the Progressive auto insurance company. Snapshot is a device that you plug into your vehicle and it collects data on your driving habits - how often you drive, where you drive, what times of day you drive, your speed, how hard you

brake, how fast you accelerate, and probably other data. You mail this device back to Progressive, and are possibly offered a discounted rate based on your driving habits. Machine learning algorithms that we discuss in this class could certainly be used to build a model that computes a car insurance rate based on thousands of observations of people's driving habits.

Machine learning is used for a wide variety of applications, but where machine learning algorithms are commonly employed are for problems that are either too complex for traditional approaches or for problems that have no known algorithm. Machine learning is often used to help humans learn by elucidating underlying patterns and behaviours in the data - this allows humans to have a better understanding of the problem. Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent - this is called ***data mining***.

# Machine Learning Applications

*Analyzing images of products on a production line to automatically classify them*

This is image classification, typically performed using convolutional neural networks (CNNs; see Chapter 14).

*Detecting tumors in brain scans*

This is semantic segmentation, where each pixel in the image is classified (as we want to determine the exact location and shape of tumors), typically using CNNs as well.

*Automatically classifying news articles*

This is natural language processing (NLP), and more specifically text classification, which can be tackled using recurrent neural networks (RNNs), CNNs, or Transformers (see Chapter 16).

*Creating a chatbot or a personal assistant*

This involves many NLP components, including natural language understanding (NLU) and question-answering modules.

*Forecasting your company's revenue next year, based on many performance metrics*

This is a regression task (i.e., predicting values) that may be tackled using any regression model, such as a Linear Regression or Polynomial Regression model (see Chapter 4), a regression SVM (see Chapter 5), a regression Random Forest (see Chapter 7), or an artificial neural network (see Chapter 10). If you want to take into account sequences of past performance metrics, you may want to use RNNs, CNNs, or Transformers (see Chapters 15 and 16).

*Detecting credit card fraud*

This is anomaly detection (see Chapter 9).

*Segmenting clients based on their purchases so that you can design a different marketing strategy for each segment*

This is clustering (see Chapter 9).

*Recommending a product that a client may be interested in, based on past purchases*

This is a recommender system. One approach is to feed past purchases (and other information about the client) to an artificial neural network (see Chapter 10), and get it to output the most likely next purchase. This neural net would typically be trained on past sequences of purchases across all clients.

**Video: 670_mod1_vid3**

# Types of Machine Learning Systems

There are so many different types of Machine Learning systems that it is useful to classify them in broad categories, based on the following criteria:

- Whether or not they are trained with human supervision (***supervised***, ***unsupervised***, ***semisupervised***, and ***Reinforcement Learning***)
- Whether or not they can learn incrementally on the fly (***online*** versus ***batch learning***)
- Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (***instance-based*** versus ***model-based learning***)

These criteria are not exclusive; you can combine them in any way you like. Let's take a closer look at each of these criteria more closely.

**Video: 670_mod1_vid4**

# Supervised/Unsupervised Learning

## Supervised Learning

Machine Learning systems can be classified according to the amount and type of supervision they get during training. In ***supervised learning***, the training set you feed to the algorithm includes the desired solutions, called ***labels*** or ***responses***.

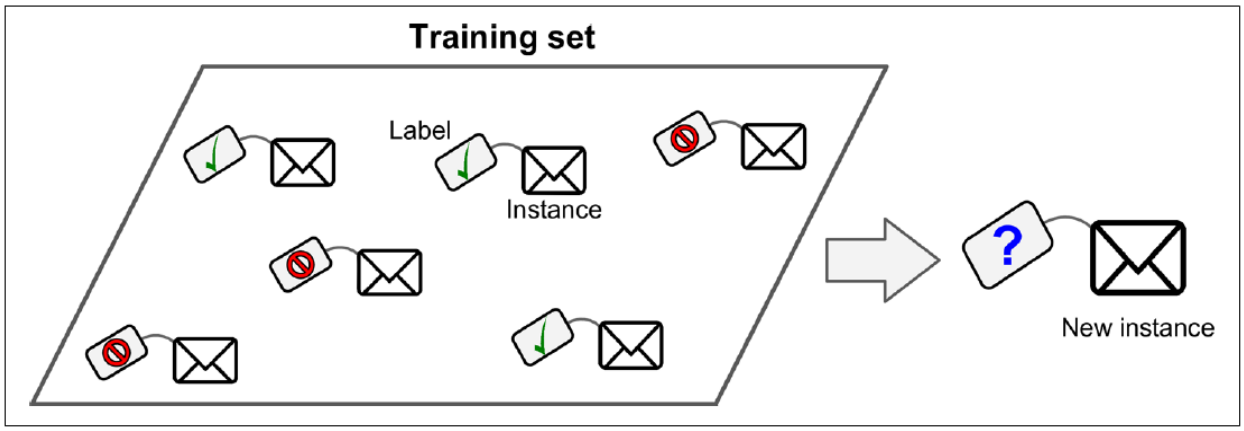


**Figure:** An example of a classification problem

A typical supervised learning task is ***classification***. The spam filter is a good example of this: it is trained with many example emails along with their ***class*** (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a target numeric value, such as the price of a car, given a set of ***features*** or ***predictors*** (i.e. mileage, age, brand). This sort of task is called ***regression***. To train the system, you need to give it many examples of cars, including both their predictors and their labels (i.e., their prices).


**Figure:** An example of a regression problem

Note that some regression algorithms can be used for classification as well, and vice versa. For example, ***logistic regression*** is commonly used for classification, as it can output a value that corresponds to the probability of belonging to a given class (e.g., 20% chance of being spam).

Here are some of the most important supervised learning algorithms (covered in our textbook):

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural Networks

**Video: 670_mod1_vid5**

## Unsupervised Learning

In unsupervised learning, as you might guess, the training data is unlabeled. The system tries to learn without a teacher.

Here are some of the most important unsupervised learning algorithms:

- Clustering
    - K-Means

    - DBSCAN
    - Hierarchical Cluster Analysis (HCA)
- Anomaly detection and novelty detection
    - One-class SVM
    - Isolation Forest
- Dimensionality reduction
    - Principal Component Analysis (PCA)
    - Kernel PCA

As an example of an unsupervised learning problem, say you have a lot of data about your blog's visitors. You may want to run a ***clustering*** algorithm to try to detect groups of similar visitors. At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are males who love comic books and generally read your blog in the evening, while 20% are young sci-fi lovers who visit during the weekends. If you use a hierarchical clustering algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.

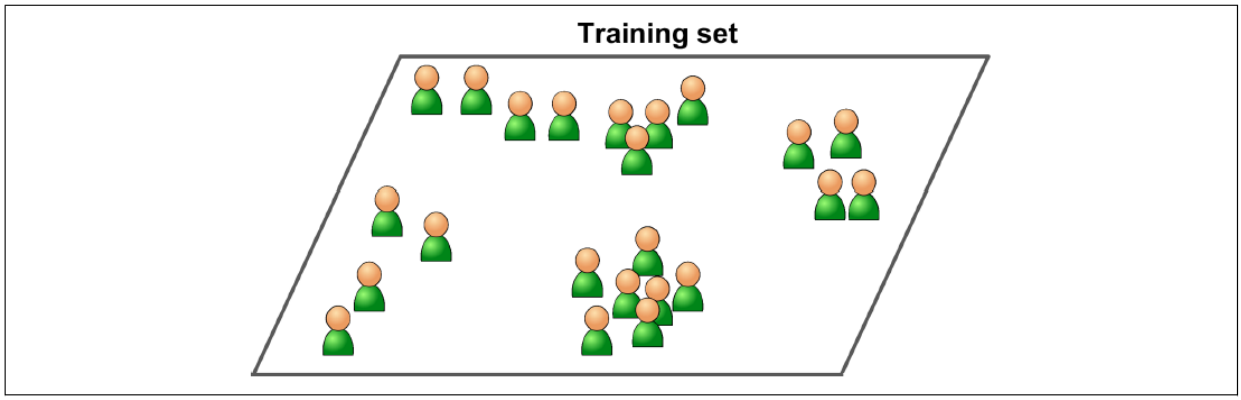


**Figure:** An example of a clustering problem (before clustering)

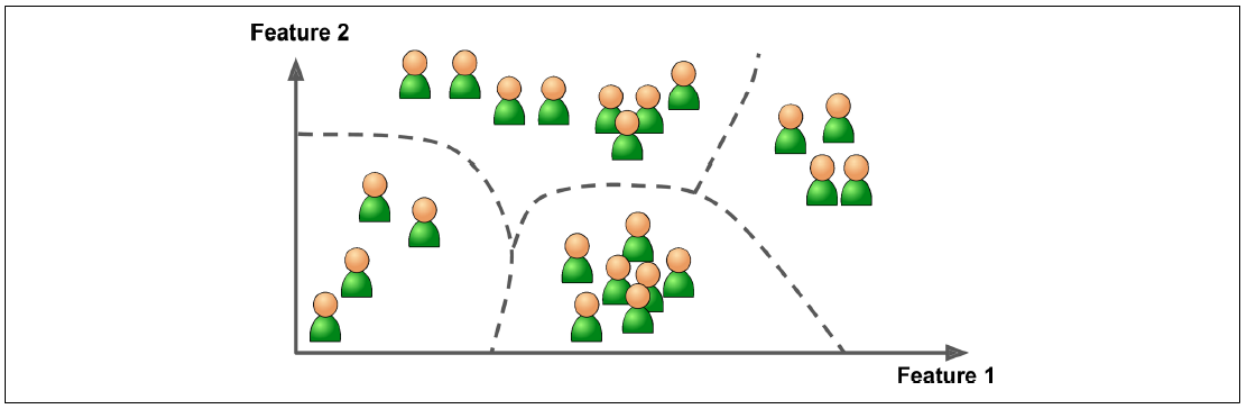


**Figure:** An example of a clustering problem (after clustering)

A related task is ***dimensionality reduction***, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called ***feature extraction***.

It is often a good idea to try to reduce the dimension of your training data using a dimensionality reduction algorithm before you feed it to another Machine Learning algorithm (such as a supervised learning algorithm). It will run much faster, the data will take up less disk and memory space, and in some cases it may also perform better.

Yet another important unsupervised task is ***anomaly detection*** - for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly.

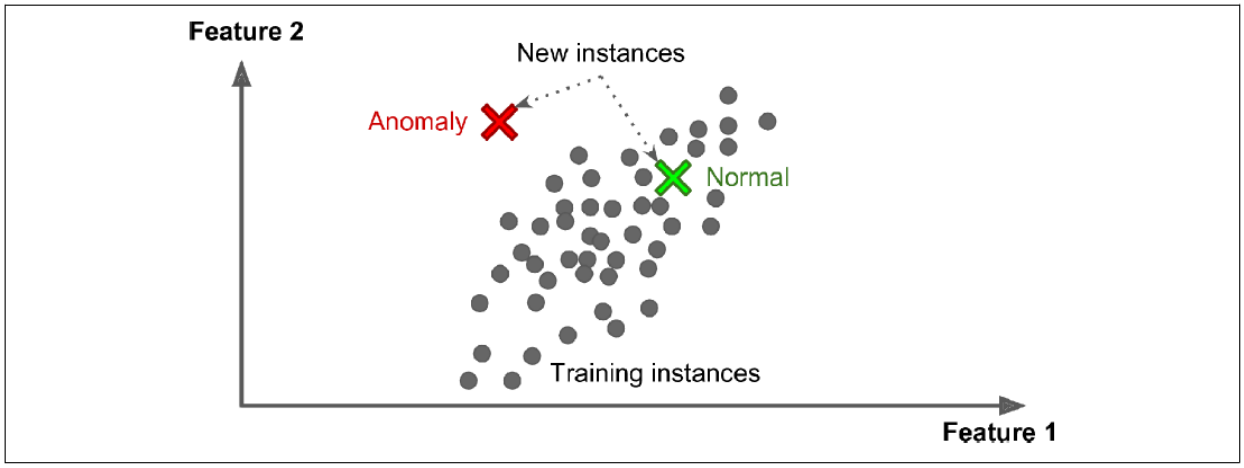


**Figure:** An example of anomaly detection

A very similar task is ***novelty detection***: it aims to detect new instances that look different from all instances in the training set. This requires having a very "clean" training set, devoid of any instance that you would like the algorithm to detect. For example, if you have thousands of pictures of dogs, and 1% of these pictures represent Chihuahuas, then a novelty detection algorithm should not treat new pictures of Chihuahuas as novelties. On the other hand, anomaly detection algorithms may consider these dogs as so rare and so different from other dogs that they would likely classify them as anomalies (no offense to Chihuahuas).

**Video: 670_mod1_vid6**

## Semisupervised Learning

Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that's partially labeled. This is called ***semisupervised learning***.

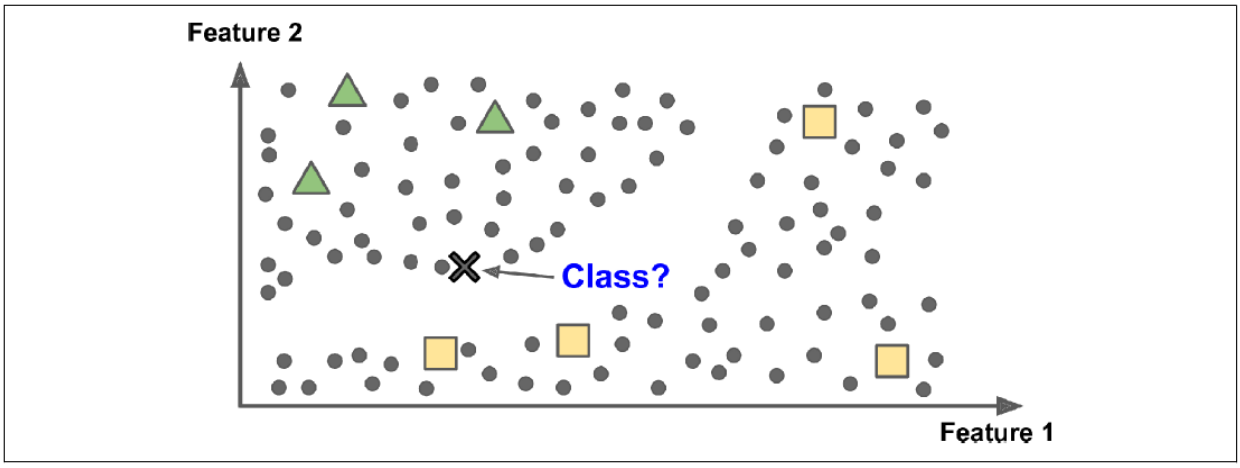


**Figure:** Semisupervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person and it is able to name everyone in every photo, which is useful for searching photos.

## Reinforcement Learning

***Reinforcement learning*** is a very different beast. 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.

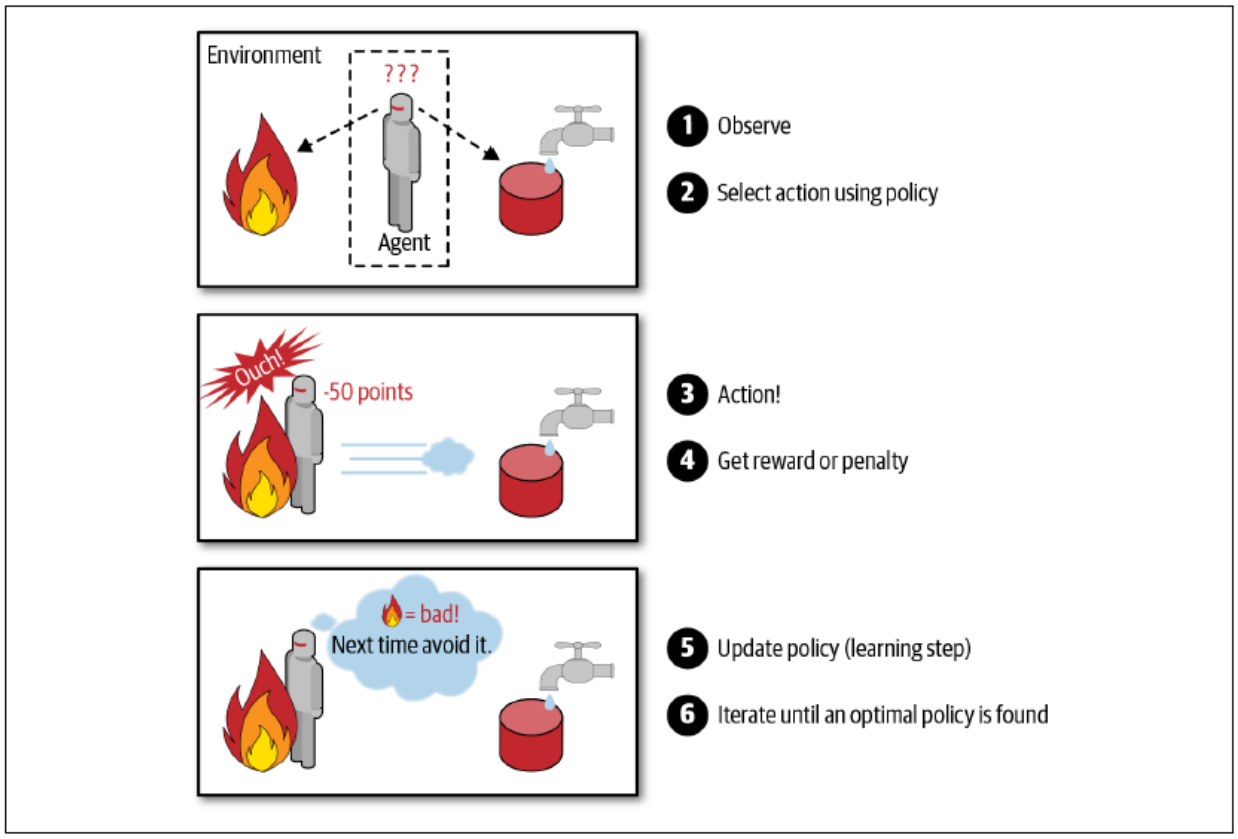


**Figure:** An example of reinforcement learning

For example, many robots implement Reinforcement Learning algorithms to learn how to walk. DeepMind's AlphaGo program is also a good example of Reinforcement Learning: it made the headlines in May 2017 when it beat the world champion Ke Jie at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned.

There is a great movie/documentary on YouTube about the AlphaGo project, which can be watched here: https://youtu.be/WXuK6gekU1Y. It is a good one!

**Video: 670_mod1_vid7**

# Batch and Online Learning

Another criterion used to classify Machine Learning systems is whether or not the system can learn incrementally from a stream of incoming data.

## Batch 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 you 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.

If your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper. Fortunately, a better option in these cases is to use algorithms that are capable of learning incrementally.

## Online Learning

***Online learning*** is great for systems that receive data as a continuous flow (e.g., stock prices) and need to adapt to change rapidly or autonomously. It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not need them anymore, so you can discard them (unless you want to be able to roll back to a previous state and "replay" the data). This can save a huge amount of space.

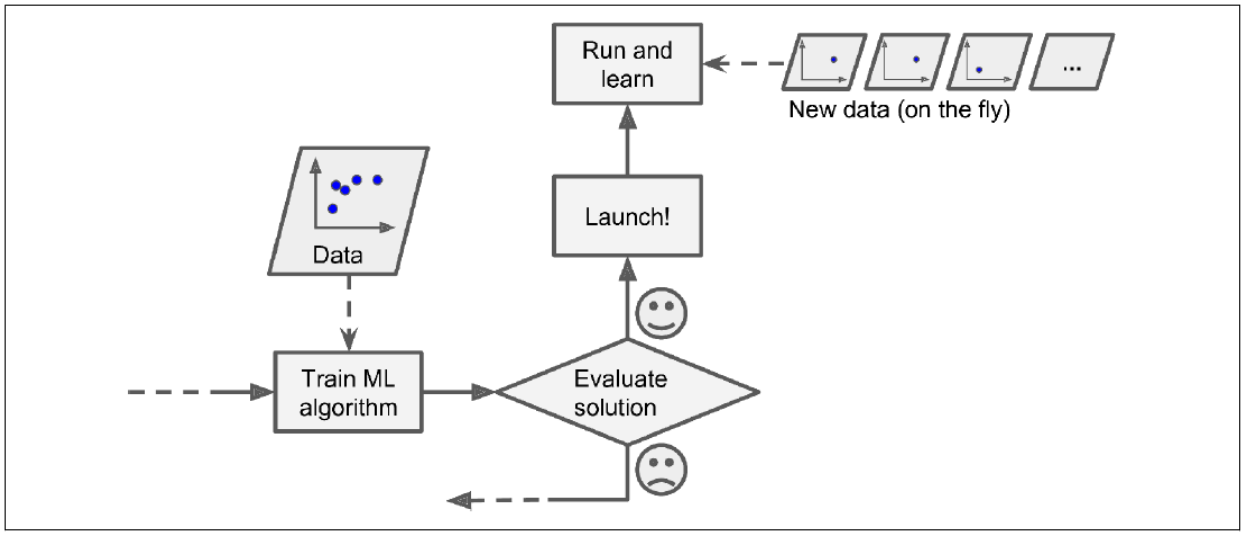


**Figure:** In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine's main memory (this is called ***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.

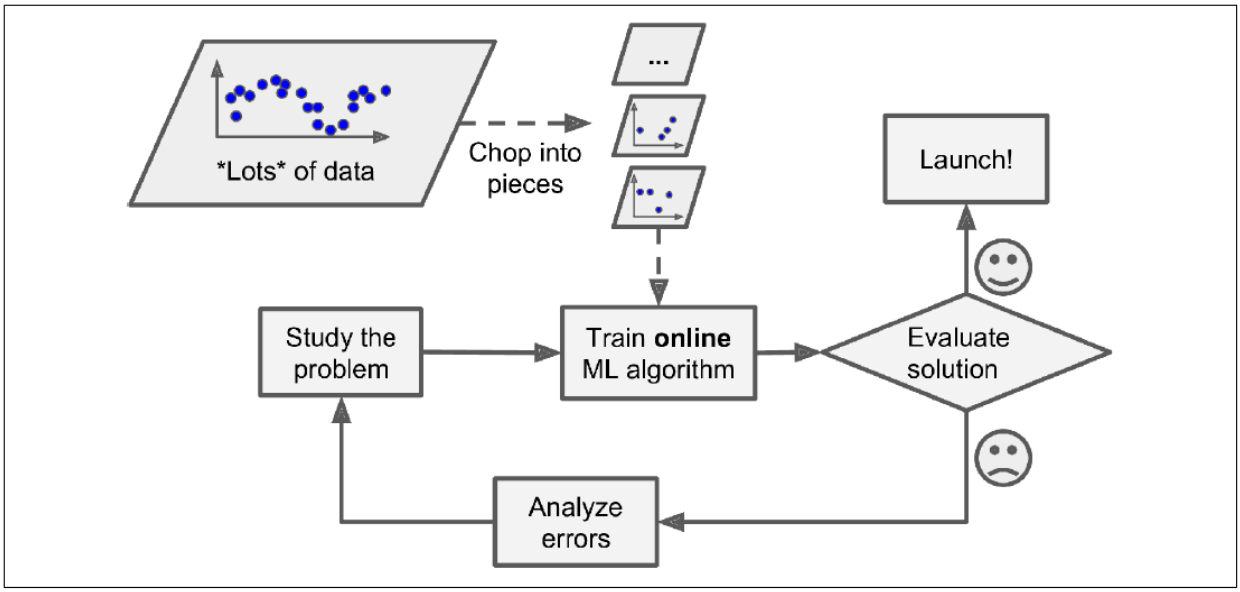


**Figure:** Using online learning to handle huge datasets

A word of caution: 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 online learning as ***incremental learning***.

One important parameter of online learning systems is how fast they should adapt to changing data: this is called the ***learning rate***. If you set a high learning rate, then your system will rapidly adapt to new data, but it will also tend to quickly forget the old data (you don't want a spam filter to flag only the latest kinds of spam it was shown). Conversely, if you set a low learning rate, the 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).

**Video: 670_mod1_vid8**

# Instance-Based Versus Model-Based Learning

One more way to categorize ML systems is by how they ***generalize***. Most ML tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before. Having a good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances.

There are two main approaches to generalization: ***instance-based learning*** and ***model-based learning***.

## Instance-based Learning

Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails that have already been flagged by users - not the worst solution, but certainly not the best.

Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed 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.

This is called ***instance-based learning***: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in the below figure, the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.

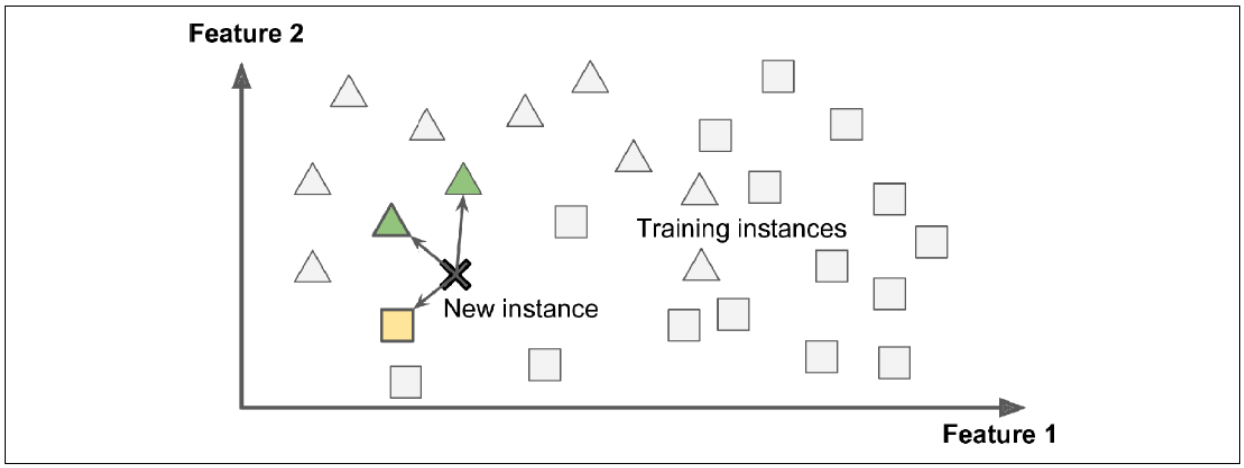


**Figure:** Example of instance-based learning

## Model-based Learning

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make predictions. This is called ***model-based learning***.

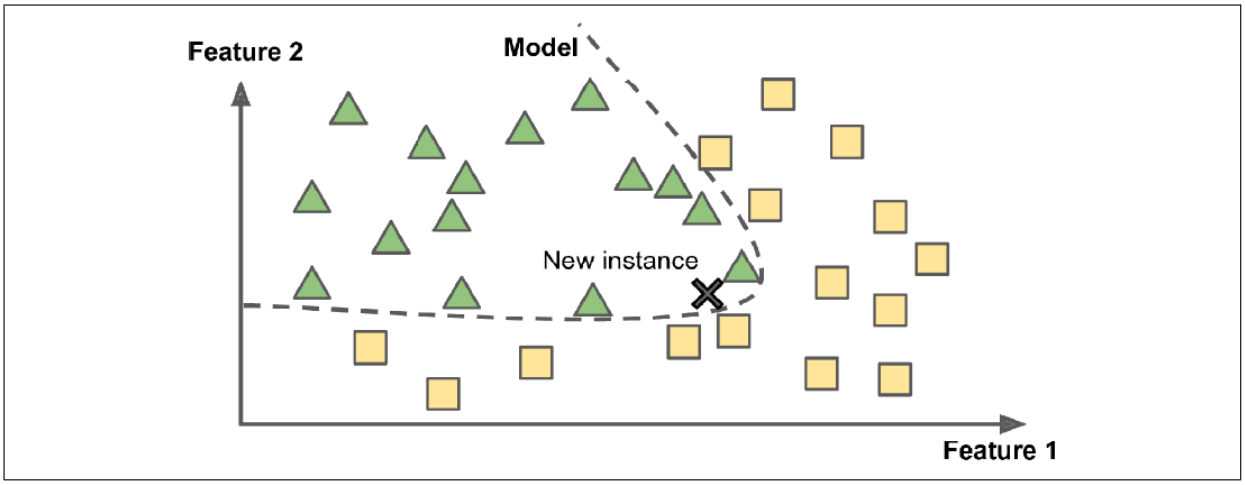


**Figure:** Example of model-based learning

**Video: 670_mod1_vid9**

Next, let's take a look at an example of a model-based learning problem. Suppose you want to know if money makes people happy, so you download the Better Life Index data from the OECD's (Organisation for Economic Cooperation and Development) website and stats about gross domestic product (GDP) per capita from the IMF's (International Monetary Fund) website. Then you join the tables and sort by GDP per capita. The following table shows an excerpt of what you get.

| Country | GDP per capita (USD) | Life satisfaction |
|---|---|---|
| Hungary | 12,240 | 4.9 |
| Korea | 27,195 | 5.8 |
| France | 37,675 | 6.5 |
| Australia | 50,962 | 7.3 |
| United States | 55,805 | 7.2 |

Let's plot the data for these countries and others:

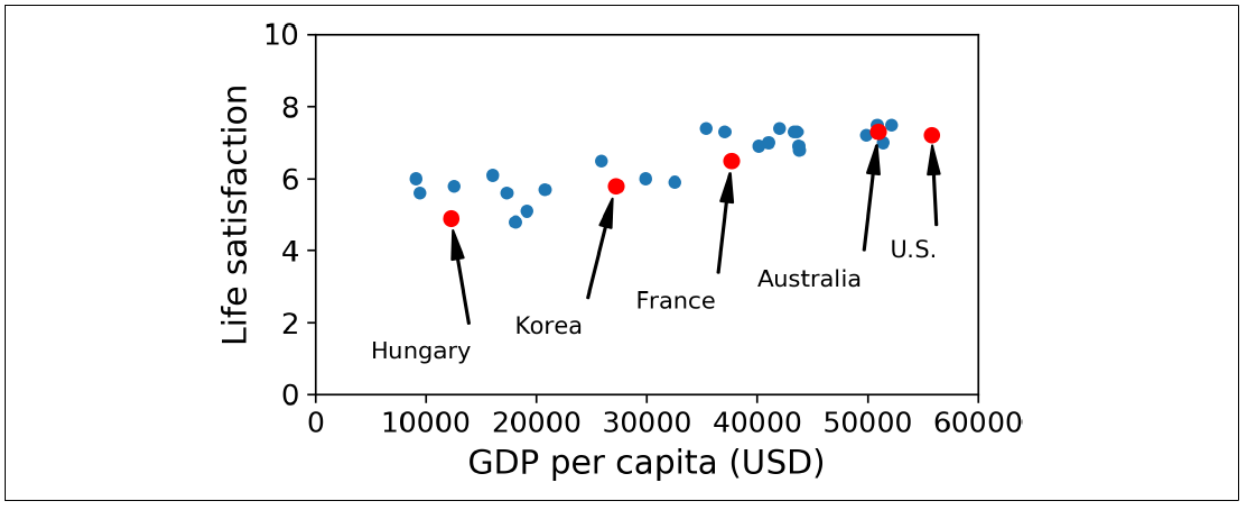


**Figure:** Do you see a linear trend among the data?

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.

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

This model has two model parameters, θ_0 and θ_1. By tweaking these parameters, you can make your model represent any linear function, as shown in the figure below.

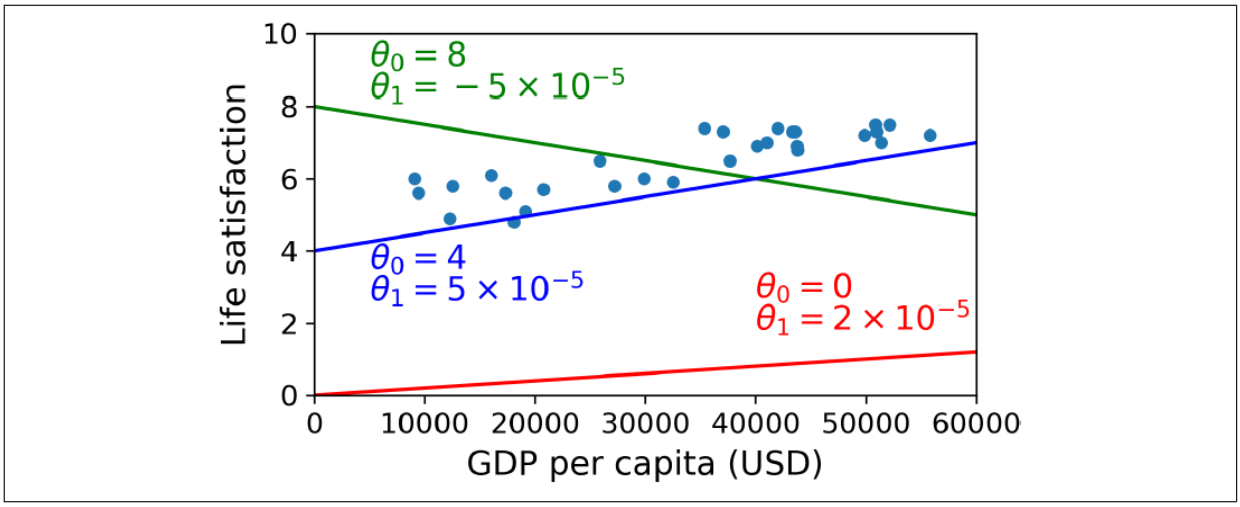


**Figure:** A few possible linear models

Before you can use your model, you need to define the parameter values θ_0 and θ_1. How can you know which values will make your model perform best? To answer this question, you need

to specify a performance measure. You can either define a ***utility function*** (or ***fitness function*** or ***objective 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 θ_0 = 4.85 and θ_1 = 4.91 × 10–5.

Model selection consists in choosing the type of model and fully specifying its architecture. Training a model means running an algorithm to find the model parameters that will make it best fit the training data (and hopefully make good predictions on new data).

With the optimal parameter values of θ_0 = 4.85 and θ_1 = 4.91 × 10–5, the model fits the training data as closely as possible (for a linear model), shown below:

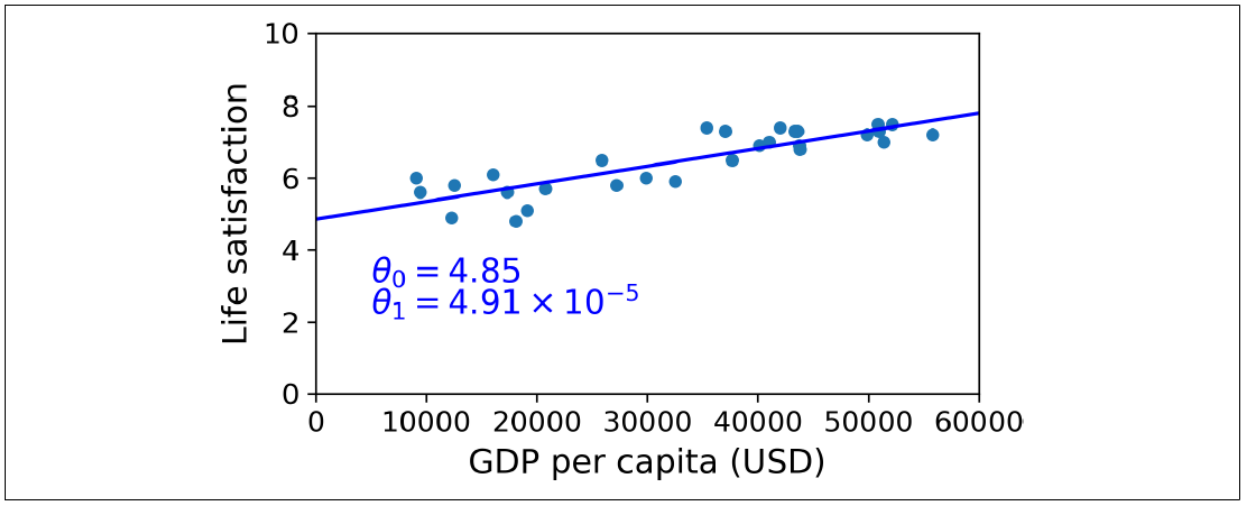


**Figure:** The linear model that fits the training data best

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Cyprus's GDP per capita, find $22,587, and then apply your model and find that life satisfaction is likely to be somewhere around 4.85 + 22,587 × 4.91 × 10-5 = 5.96.

(Now, if you would like, you can get your hands dirty with some Python code using Scikit-Learn. See the Jupyter notebook called module_1_moneyAndHappiness.ipynb. You do not need to understand how this code works at this point in time - we will get there soon!)

If you had used an instance-based learning algorithm instead, you 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, you would have predicted a life satisfaction of 5.7 for Cyprus. If you zoom out a bit and look at the two next-closest countries, you 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). Replacing the Linear Regression model with k-Nearest Neighbors regression in the previous code is as simple as replacing two lines of code.

```
import sklearn.linear_model
model = sklearn.linear_model.LinearRegression()
```

The above two lines of code should be replaced with the following two lines of code.

```
import sklearn.neighbors
model = sklearn.neighbors.KNeighborsRegressor(
    n_neighbors=3)
```

If all went well, your model will make good predictions. If not, you may need to use more attributes (employment rate, health, air pollution, etc.), get more or better-quality training data, or perhaps select a more powerful model (e.g., a Polynomial Regression model).

Let's summarize the process we just went through, which how a typical machine learning project will go:

- You studied the data.
- You selected a model.
- You trained it on the training data (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you applied the model to make predictions on new cases (this is called inference), hoping that this model will generalize well. This is what a typical Machine Learning project looks like

We have covered a lot of ground so far: you now know what Machine Learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like. Now let's look at what can go wrong in learning and prevent you from making accurate predictions.

**Video: 670_mod1_vid10**

# Main Challenges of Machine Learning

In short, since your main task is to select a learning algorithm and train it on some data, the two things that can go wrong are "bad algorithm" and "bad data." Let's start with examples of bad data.

## Insufficient Quantity of Training Data

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say "apple" (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.

Machine Learning is not quite there yet; it takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model).

## Nonrepresentative Training Data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

For example, the set of countries we used earlier for training the linear model was not perfectly representative; a few countries were missing. The figure below shows what the data looks like when you add the missing countries.

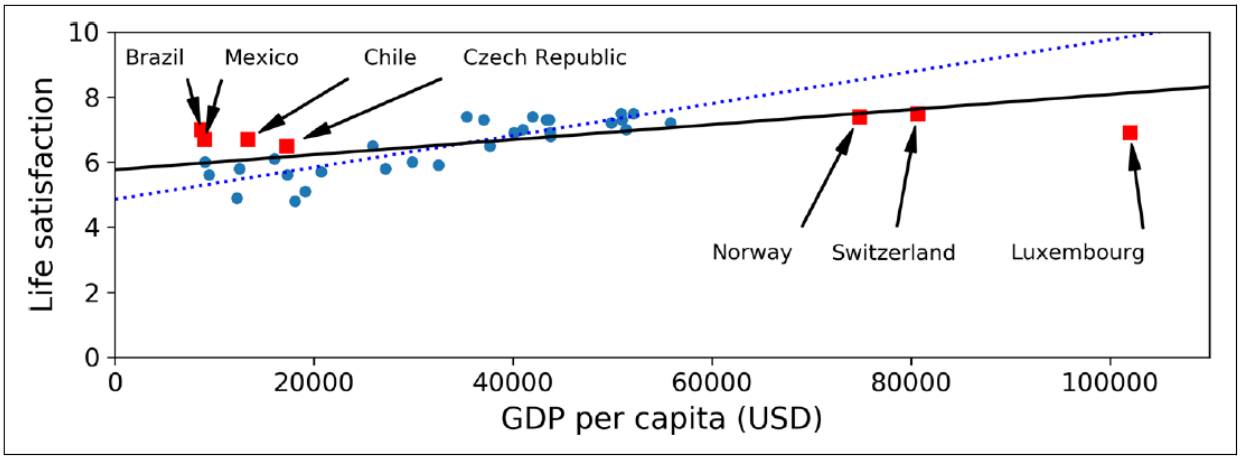

**Figure:** A more representative training sample

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem unhappier), and conversely some poor countries seem happier than many rich countries.

By using a nonrepresentative training set, we trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have ***sampling noise*** (i.e., nonrepresentative data as a result of chance), but even very large samples can be nonrepresentative if the sampling method is flawed. This is called ***sampling bias***.

## Examples of Sampling Bias

Perhaps the most famous example of sampling bias happened during the US presidential election in 1936, which pitted Landon against Roosevelt: the Literary Digest conducted a very large poll, sending mail to about 10 million people. It got 2.4 million answers, and predicted with high confidence that Landon would get 57% of the votes. Instead, Roosevelt won with 62% of the votes. The flaw was in the Literary Digest's sampling method:

- First, to obtain the addresses to send the polls to, the Literary Digest used telephone directories, lists of magazine subscribers, club membership lists, and the like. All of these lists tended to favor wealthier people, who were more likely to vote Republican (hence Landon).
- Second, less than 25% of the people who were polled answered. Again this introduced a sampling bias, by potentially ruling out people who didn't care much about politics, people who didn't like the Literary Digest, and other key groups. This is a special type of sampling bias called ***nonresponse bias***.

Here is another example: say you want to build a system to recognize funk music videos. One way to build your training set is to search for "funk music" on YouTube and use the resulting videos. But this assumes that YouTube's search engine returns a set of videos that are representative of all the funk music videos on YouTube. In reality, the search results are likely to be biased toward popular artists (and if you live in Brazil you will get a lot of "funk carioca" videos, which sound nothing like James Brown). On the other hand, how else can you get a large training set?

**Video: 670_mod1_vid11**

## Poor-Quality Data

Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple of examples of when you'd want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

## Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a Machine Learning project is coming up with a good set of features to train on. This process, called ***feature engineering***, involves the following steps:

- ***Feature selection*** (selecting the most useful features to train on among existing features)
- ***Feature extraction*** (combining existing features to produce a more useful one - as we saw earlier, dimensionality reduction algorithms can help)
- Creating new features by gathering new data

Now that we have looked at many examples of bad data, let's look at a couple of examples of bad algorithms.

**Video: 670_mod1_vid12**

## Overfitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that all taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In Machine Learning this is called ***overfitting***: it means that the model performs well on the training data, but it does not generalize well.

The below figure shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

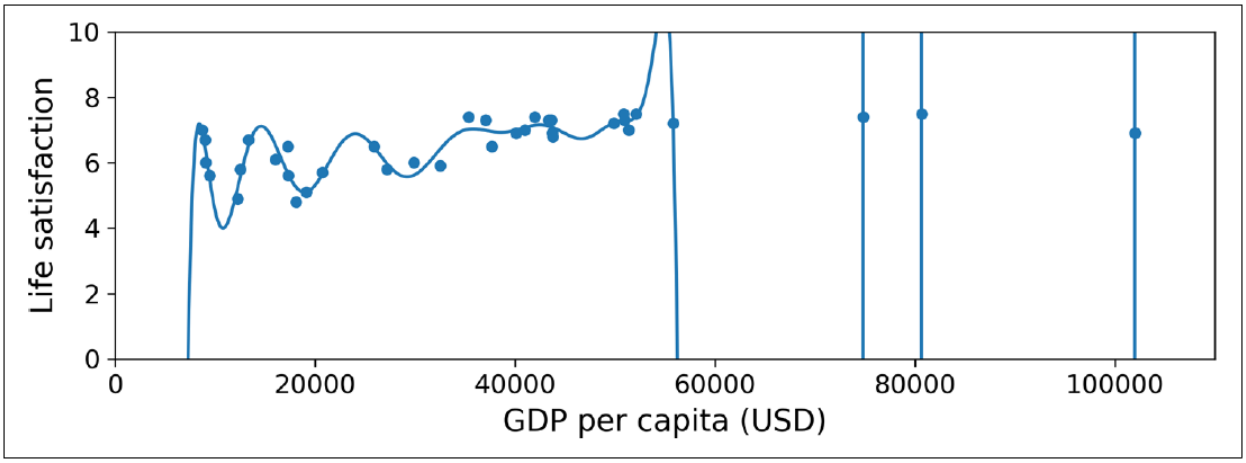


**Figure:** Overfitting the training data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small (which introduces sampling noise), then the model is likely to detect patterns in the noise itself. Obviously these patterns will not generalize to new instances.

Constraining a model to make it simpler also helps reduce the risk of overfitting - this is called ***regularization***. The amount of regularization to apply during learning can be controlled by a hyperparameter. A hyperparameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training.

## Underfitting the Training Data

As you might guess, underfitting is the opposite of overfitting: it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples. Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (e.g., reduce the regularization hyperparameter).

**Video: 670_mod1_vid13**

## A Short Summary So Far

By now you know a lot about Machine Learning. However, we went through so many concepts that you may be feeling a little lost, so let's step back and look at the big picture:

- Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules.
- There are many different types of ML systems: supervised or not, batch or online, instance-based or model-based.
- In an ML project you gather data in a training set, and you feed the training set to a learning algorithm. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances.
- The system will not perform well if your training set is too small, or if the data is not representative, is noisy, or is polluted with irrelevant features (garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit).

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases. You want to evaluate it and finetune it if necessary. Let's see how to do that.

**Video: 670_mod1_vid14**

# Testing and Validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. To do this, you should split your data into two sets before training any models - these sets are called the ***training set*** and the ***test set***. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the ***generalization error***, and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.

It is common to use 80% of the data for training and hold out 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.

## Hyperparameter Tuning and Model Selection

Evaluating a model is simple enough: just use a test set. But suppose you are hesitating between two types of models (say, a linear model and a polynomial model): how can you decide between them? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better, but you want to apply some regularization to avoid overfitting. The question is, how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error—say, just 5% error. You launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model *for that particular set*. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is called ***holdout validation***: you simply hold out part of the training set to evaluate several candidate models and select the best one. The new held-out set is called the ***validation set*** (or sometimes the ***development set***, or ***dev set***). More specifically, you train multiple models with various hyperparameters on the reduced training set (i.e., the full training set minus the validation set), and you select the model that performs best on the validation set. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

This solution usually works quite well. However, if the validation set is too small, then model evaluations will be imprecise: you may end up selecting a suboptimal model by mistake. Conversely, if the validation set is too large, then the remaining training set will be much smaller than the full training set. Why is this bad? Well, since the final model will be trained on the full training set, it is not ideal to compare candidate models trained on a much smaller training set. It would be like selecting the fastest sprinter to participate in a marathon. One way to solve this problem is to perform repeated ***cross-validation***, using many small validation sets. Each model is evaluated once per validation set after it is trained on the rest of the data. By averaging out all the

evaluations of a model, you get a much more accurate measure of its performance. There is a drawback, however: the training time is multiplied by the number of validation sets.