1
Getting Started with Python and Machine Learning

We kick off our Python and machine learning journey with the basic, yet important concepts of machine learning. We will start with what machine learning is about, why we need it, and its evolution over the last few decades. We will then discuss typical machine learning tasks and explore several essential techniques of working with data and working with models. It is a great starting point of the subject and we will learn it in a fun way. Trust me. At the end, we will also set up the software and tools needed in this book.

We will get into details for the topics mentioned:

- What is machine learning and why do we need it?
- A very high level overview of machine learning
- Generalizing with data
- Overfitting and the bias variance trade off
    - Cross validation
    - Regularization
- Dimensions and features
- Preprocessing, exploration, and feature engineering
    - Missing Values
    - Label encoding
    - One hot encoding
    - Scaling
    - Polynomial features
    - Power transformations
    - Binning

- Combining models
    - Bagging
    - Boosting
    - Stacking
    - Blending
    - Voting and averaging
- Installing software and setting up
- Troubleshooting and asking for help

What is machine learning and why do we need it?

Machine learning is a term coined around 1960 composed of two words—machine corresponding to a computer, robot, or other device, and learning an activity, or event patterns, which humans are good at.

So why do we need machine learning, why do we want a machine to learn as a human? There are many problems involving huge datasets, or complex calculations for instance, where it makes sense to let computers do all the work. In general, of course, computers and robots don't get tired, don't have to sleep, and may be cheaper. There is also an emerging school of thought called active learning or human-in-the-loop, which advocates combining the efforts of machine learners and humans. The idea is that there are routine boring tasks more suitable for computers, and creative tasks more suitable for humans. According to this philosophy, machines are able to learn, by following rules (or algorithms) designed by humans and to do repetitive and logic tasks desired by a human.

Machine learning does not involve the traditional type of programming that uses business rules. A popular myth says that the majority of the code in the world has to do with simple rules possibly programmed in Cobol, which covers the bulk of all the possible scenarios of client interactions. So why can't we just hire many software programmers and continue programming new rules?

One reason is that defining, maintaining, and updating rules becomes more and more expensive over time. The number of possible patterns for an activity or event could be enormous and therefore exhausting all enumeration is not practically feasible. It gets even more challenging to do so when it comes to events that are dynamic, ever-changing, or evolve in real-time. It is much easier and more efficient to develop learning rules or algorithms which command computers to learn and extract patterns, and to figure things out themselves from abundant data.

Another reason is that the volume of data is exponentially growing. Nowadays, the floods of textual, audio, image, and video data are hard to fathom. The **Internet of Things** (**IoT**) is a recent development of a new kind of Internet, which interconnects everyday devices. The Internet of Things will bring data from household appliances and autonomous cars to the forefront. The average company these days has mostly human clients, but, for instance, social media companies tend to have many bot accounts. This trend is likely to continue and we will have more machines talking to each other. Besides the quantity, the quality of data available has kept increasing over the past few years due to cheaper storage. These have empowered the evolution of machine learning algorithms and data-driven solutions.

Jack Ma from Alibaba explained in a speech that **Information Technology** (**IT**) was the focus over the past 20 years and now, for the next 30 years, we will be at the age of **Data Technology** (**DT**). During the age of IT, companies have grown larger and stronger thanks to computer software and infrastructure. Now that businesses in most industries have already gathered enormous amounts of data, it is presently the right time for exploiting DT to unlock insights, derive patterns, and to boost new business growth. Broadly speaking, machine learning technologies enable businesses to better understand customer behavior and engage with customers, also to optimize operations management. As for us individuals, machine learning technologies are already making our life better every day.

An application of machine learning that we all are familiar with is spam email filtering. Another is online advertising where ads are served automatically based on information advertisers have collected about us. Stay tuned for the next chapters where we will learn how to develop algorithms in solving these two problems. An application of machine learning we basically can not live without is search engines. Search engines involve information retrieval which parses what we look for and queries related records, and contextual ranking and personalized ranking which sorts pages by topical relevance and to the user's liking. E-commerce and media companies have been at the forefront of employing recommendation systems, which help customers find products, services, articles faster. The application of machine learning is boundless and we just keep hearing new examples everyday, credit card fraud detection, disease diagnosis, presidential election prediction, instant speech translation, robo-advisor, you name it!

In the 1983 *War Games* movie, a computer made life and death decisions, which could have resulted in Word War III. As far as we know, technology wasn't able to pull off such feats at the time. However, in 1997 the Deep Blue supercomputer did manage to beat a world chess champion. In 2005, a Stanford self-driving car drove by itself for more than 130 kilometers in a desert. In 2007, the car of another team drove through regular traffic for more than 50 kilometers. In 2011, the Watson computer won a quiz against human opponents. In 2016, the AlphaGo program beat one of the best Go players in the world. If we assume that computer hardware is the limiting factor, then we can try to extrapolate into the future. Ray Kurzweil did just that and according to him, we can expect human level intelligence around 2029. What's next?

A very high level overview of machine learning

Machine learning mimicking human intelligence is a subfield of artificial intelligence—a field of computer science concerned with creating systems. Software engineering is another field in computer science. Generally, we can label Python programming as a type of software engineering. Machine learning is also closely related to linear algebra, probability theory, statistics, and mathematical optimization. We usually build machine learning models based on statistics, probability theory, and linear algebra, then optimize the models using mathematical optimization. The majority of us reading this book should have at least sufficient knowledge of Python programming. Those who are not feeling confident about mathematical knowledge, might be wondering, how much time should be spent learning or brushing up the knowledge of the aforementioned subjects. Don't panic. We will get machine learning to work for us without going into any mathematical details in this book. It just requires some basic, 101 knowledge of probability theory and linear algebra, which helps us understand the mechanics of machine learning techniques and algorithms. And it gets easier as we will be building models both from scratch and with popular packages in Python, a language we like and are familiar with.

Those who want to study machine learning systematically can enroll into computer science, artificial intelligence, and more recently, data science master's programs. There are also various data science bootcamps. However the selection for bootcamps is usually stricter as they are more job oriented, and the program duration is often short ranging from 4 to 10 weeks. Another option is the free massive open online courses (MOOC), for example, the popular one is Andrew Ng's Machine Learning. Last but not least, industry blogs and websites are great resources for us to keep up with the latest development.

Machine learning is not only a skill, but also a bit of sport. We can compete in several machine learning competitions; sometimes for decent cash prizes, sometimes for joy, most of the time for playing to strengths. However, to win these competitions, we may need to utilize certain techniques, which are only useful in the context of competitions and not in the context of trying to solve a business problem. That's right, the "no free lunch" theorem applies here.

A machine learning system is fed with input data—this can be numerical, textual, visual, or audiovisual. The system usually has outputs—this can be a floating-point number, for instance, the acceleration of a self-driving car, can be an integer representing a category (also called a **class**), for example, a cat or tiger from image recognition.

The main task of machine learning is to explore and construct algorithms that can learn from historical data and make predictions on new input data. For a data-driven solution, we need to define (or have it defined for us by an algorithm) an evaluation function called **loss** or **cost function**, which measures how well the models are learning. In this setup, we create an optimization problem with the goal of learning in the most efficient and effective way.

Depending on the nature of the learning data, machine learning tasks can be broadly classified into three categories as follows:

- **Unsupervised learning**: when learning data contains only indicative signals without any description attached, it is up to us to find structure of the data underneath, to discover hidden information, or to determine how to describe the data. This kind of learning data is called **unlabeled** data. Unsupervised learning can be used to detect anomalies, such as fraud or defective equipment, or to group customers with similar online behaviors for a marketing campaign.
- **Supervised learning**: when learning data comes with description, targets or desired outputs besides indicative signals, the learning goal becomes to find a general rule that maps inputs to outputs. This kind of learning data is called **labeled** data. The learned rule is then used to label new data with unknown outputs. The labels are usually provided by event logging systems and human experts. Besides, if it is feasible, they may also be produced by members of the public through crowdsourcing for instance. Supervised learning is commonly used in daily applications, such as face and speech recognition, products or movie recommendations, and sales forecasting.
- We can further subdivide supervised learning into **regression** and **classification**. Regression trains on and predicts a continuous-valued response, for example predicting house prices, while classification attempts to find the appropriate class label, such as analyzing positive/negative sentiment and prediction loan defaults.

- If not all learning samples are labeled, but some are, we will have **semi-supervised learning**. It makes use of unlabeled data (typically a large amount) for training, besides a small amount of labeled. Semi-supervised learning is applied in cases where it is expensive to acquire a fully labeled dataset while more practical to label a small subset. For example, it often requires skilled experts to label hyperspectral remote sensing images, and lots of field experiments to locate oil at a particular location, while acquiring unlabeled data is relatively easy.
- **Reinforcement learning**: learning data provides feedback so that the system adapts to dynamic conditions in order to achieve a certain goal. The system evaluates its performance based on the feedback responses and reacts accordingly. The best known instances include self-driving cars and chess master AlphaGo.

Feeling a little bit confused by the abstract concepts? Don't worry. We will encounter many concrete examples of these types of machine learning tasks later in the book. In `Chapter 3`, *Spam Email Detection with Naive Bayes*, to `Chapter 6`, *Click-Through Prediction with Logistic Regression*, we will see some supervised learning tasks and several classification algorithms; in `Chapter 7`, *Stock Price Prediction with Regression Algorithms*, we will continue with another supervised learning task, regression, and assorted regression algorithms; while in `Chapter 2`, *Exploring the 20 Newsgroups Dataset with Text Analysis Algorithms*, we will be given an unsupervised task and explore various unsupervised techniques and algorithms.

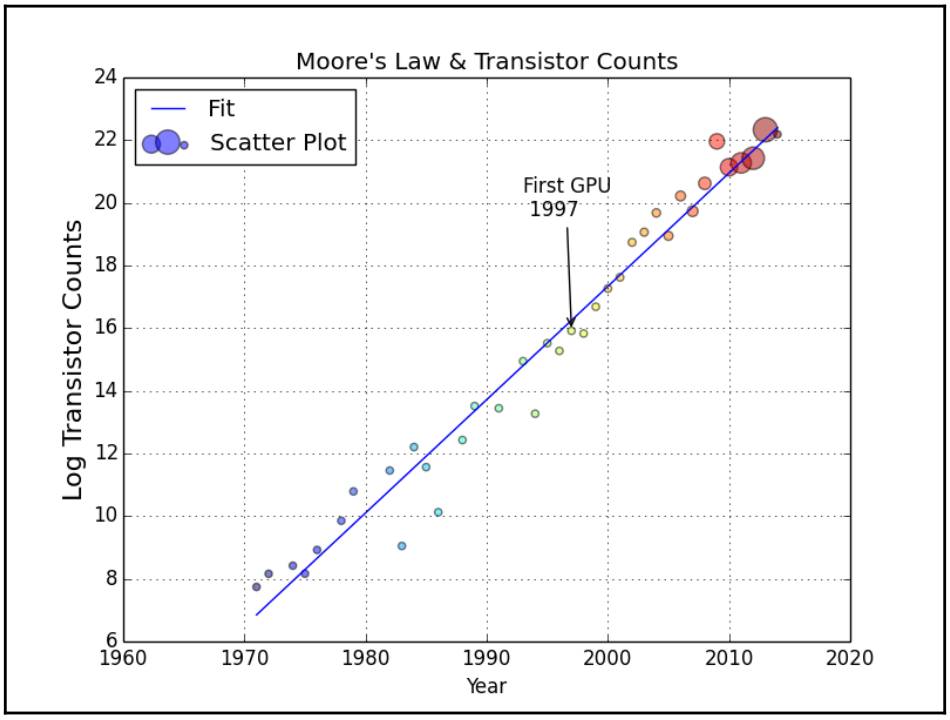
A brief history of the development of machine learning algorithms

In fact, we have a whole zoo of machine learning algorithms with popularity varying over time. We can roughly categorize them into four main approaches: **logic-based learning**, **statistical learning**, **artificial neural networks**, and **genetic algorithms**.

The logic-based systems were the first to be dominant. They used basic rules specified by human experts, and with these rules, systems tried to reason using formal logic, background knowledge, and hypotheses. In the mid-1980s, **artificial neural networks** (**ANN**) came to the foreground, to be then pushed aside by statistical learning systems in the 1990s. Artificial neural networks imitate animal brains, and consist of interconnected neurons that are also an imitation of biological neurons. They try to model complex relationships between inputs and outputs and to capture patterns in data. **Genetic algorithms** (**GA**) were popular in the 1990s. They mimic the biological process of evolution and try to find the optimal solutions using methods such as mutation and crossover.

We are currently (2017) seeing a revolution in **deep learning**, which we may consider to be a rebranding of neural networks. The term deep learning was coined around 2006, and refers to deep neural networks with many layers. The breakthrough in deep learning is amongst others caused by the integration and utilization of **graphical processing units** (**GPU**), which massively speed up computation. GPUs were originally developed to render video games, and are very good in parallel matrix and vector algebra. It is believed that deep learning resembles the way humans learn, therefore may be able to deliver on the promise of sentient machines.

Some of us may have heard of Moore's law-an empirical observation claiming that computer hardware improves exponentially with time. The law was first formulated by Gordon Moore, the co-founder of Intel, in 1965. According to the law, the number of transistors on a chip should double every two years. In the following graph, you can see that the law holds up nicely (the size of the bubbles corresponds to the average transistor count in GPUs):



The consensus seems to be that Moore's law should continue to be valid for a couple of decades. This gives some credibility to Ray Kurzweil's predictions of achieving true machine intelligence in 2029.

Generalizing with data

The good thing about data is that we have a lot of data in the world. The bad thing is that it is hard to process this data. The challenges stem from the diversity and noisiness of the data. We as humans, usually process data coming in our ears and eyes. These inputs are transformed into electrical or chemical signals. On a very basic level, computers and robots also work with electrical signals. These electrical signals are then translated into ones and zeroes. However, we program in Python in this book, and on that level normally we represent the data either as numbers, images, or text. Actually images and text are not very convenient, so we need to transform images and text into numerical values.

Especially in the context of supervised learning we have a scenario similar to studying for an exam. We have a set of practice questions and the actual exams. We should be able to answer exam questions without knowing the answers for them. This is called generalization—we learn something from our practice questions and hopefully are able to apply this knowledge to other similar questions. In machine learning, these practice questions are called **training sets** or **training samples**. They are where the models derive patterns from. And the actual exams are **testing sets** or **testing samples**. They are where the models are eventually applied and how compatible they are is what it's all about. Sometimes between practice questions and actual exams, we have mock exams to assess how well we will do in actual ones and to aid revision. These mock exams are called **validation sets** or **validation samples** in machine learning. They help us verify how well the models will perform in a simulated setting then we fine-tune the models accordingly in order to achieve greater hits.

An old-fashioned programmer would talk to a business analyst or other expert, then implement a rule that adds a certain value multiplied by another value corresponding, for instance, to tax rules. In a machine learning setting we give the computer example input values and example output values. Or if we are more ambitious, we can feed the program the actual tax texts and let the machine process the data further just like an autonomous car doesn't need a lot of human input.

This means implicitly that there is some function, for instance, a tax formula we are trying to figure out. In physics we have almost the same situation. We want to know how the universe works and formulate laws in a mathematical language. Since we don't know the actual function, all we can do is measure what error is produced, and try to minimize it. In supervised learning tasks we compare our results against the expected values. In unsupervised learning we measure our success with related metrics. For instance, we want clusters of data to be well defined, the metrics could be how similar the data points within one cluster are and how different the data points from two clusters are. In reinforcement learning, a program evaluates its moves, for example, in a chess game using some predefined function.

Overfitting, underfitting and the bias-variance tradeoff

Overfitting (one word) is such an important concept that I decided to start discussing it very early in the book.

If we go through many practice questions for an exam, we may start to find ways to answer questions which have nothing to do with the subject material. For instance, given only five practice questions, we find that if there are two potato and one tomato in a question, the answer is always *A*, if there are one potato and three tomato in a question, the answer is always *B*, then we conclude this is always true and apply such theory later on even though the subject or answer may not be relevant to potatoes or tomatoes. Or even worse, you may memorize the answers for each question verbatim. We can then score high on the practice questions; we do so with the hope that the questions in the actual exams will be the same as practice questions. However, in reality, we will score very low on the exam questions as it is rare that the exact same questions will occur in the actual exams.

The phenomenon of memorization can cause overfitting. We are over extracting too much information from the training sets and making our model just work well with them, which is called **low bias** in machine learning. However, at the same time, it will not help us generalize with data and derive patterns from them. The model as a result will perform poorly on datasets that were not seen before. We call this situation **high variance** in machine learning.

Overfitting occurs when we try to describe the learning rules based on a relatively small number of observations, instead of the underlying relationship, such the preceding potato and tomato example. Overfitting also takes place when we make the model excessively complex so that it fits every training sample, such as memorizing the answers for all questions as mentioned previously.

The opposite scenario is called **underfitting**. When a model is underfit, it does not perform well on the training sets, and will not so on the testing sets, which means it fails to capture the underlying trend of the data. Underfitting may occur if we are not using enough data to train the model, just like we will fail the exam if we did not review enough material; it may also happen if we are trying to fit a wrong model to the data, just like we will score low in any exercises or exams if we take the wrong approach and learn it the wrong way. We call any of these situations **high bias** in machine learning, although its variance is low as performance in training and test sets are pretty consistent, in a bad way.

We want to avoid both overfitting and underfitting. Recall bias is the error stemming from incorrect assumptions in the learning algorithm; high bias results in underfitting, and variance measures how sensitive the model prediction is to variations in the datasets. Hence, we need to avoid cases where any of bias or variance is getting high. So, does it mean we should always make both bias and variance as low as possible? The answer is yes, if we can. But in practice, there is an explicit trade-off between themselves, where decreasing one increases the other. This is the so-called **bias–variance tradeoff**. Does it sound abstract? Let's take a look at the following example.

We were asked to build a model to predict the probability of a candidate being the next president based on the phone poll data. The poll was conducted by zip codes. We randomly choose samples from one zip code, and from these, we estimate that there's a 61% chance the candidate will win. However, it turns out he loses the election. Where did our model go wrong? The first thing we think of is the small size of samples from only one zip code. It is the source of high bias, also because people in a geographic area tend to share similar demographics. However, it results in a low variance of estimates. So, can we fix it simply by using samples from a large number zip codes? Yes, but don't get happy so early. This might cause an increased variance of estimates at the same time. We need to find the optimal sample size, the best number of zip codes to achieve the lowest overall bias and variance. Minimizing the total error of a model requires a careful balancing of bias and variance. Given a set of training samples *x_1, x_2, …, x_n* and their targets *y_1, y_2, …, y_n*, we want to find a regression function, $\hat{y}(x)$, which estimates the true relation $y(x)$ as correctly as possible. We measure the error of estimation, how good (or bad) the regression model is by mean squared error (MSE):

$$MSE = E\left[\left(y(\boldsymbol{x}) - \hat{y}(\boldsymbol{x})\right)^2\right]$$

The *E* denotes the expectation. This error can be decomposed into bias and variance components following the analytical derivation as follows (although it requires a bit of basic probability theory to understand):

$$MSE = E[(y - \hat{y})^2]$$

$$= E[(y - E[\hat{y}] + E[\hat{y}] - \hat{y})^2]$$

$$= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + E[2(y - E[\hat{y}])(E[\hat{y}] - \hat{y})]$$

$$= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + 2(y - E[\hat{y}])(E[\hat{y}] - E[\hat{y}])$$

$$= (E[\hat{y} - y])^2 + E[\hat{y}^2] - E[\hat{y}]^2 = Bias[\hat{y}]^2 + Variance[\hat{y}]$$

The bias term measures the error of estimations, and the variance term describes how much the estimation $\hat{y}$ moves around its mean. The more complex the learning model $\hat{y}(x)$ and the larger the size of training samples, the lower the bias will be. However, these will also create more shift on the model in order to fit better the increased data points. As a result, the variance will be lifted.
We usually employ the cross-validation technique to find the optimal model balancing bias and variance and to diminish overfitting.

The last term is the irreducible error.

Avoid overfitting with cross-validation

Recall that between practice questions and actual exams, there are mock exams where we can assess how well we will perform in the actual ones and conduct necessary revision. In machine learning, the validation procedure helps evaluate how the models will generalize to independent or unseen datasets in a simulated setting. In a conventional validation setting, the original data is partitioned into three subsets, usually 60% for the training set, 20% for the validation set, and the rest 20% for the testing set. This setting suffices if we have enough training samples after partition and we only need a rough estimate of simulated performance. Otherwise, cross-validation is preferable.

In one round of cross-validation, the original data is divided into two subsets, for training and testing (or validation) respectively. The testing performance is recorded. Similarly, multiple rounds of cross-validation are performed under different partitions. Testing results from all rounds are finally averaged to generate a more accurate estimate of model prediction performance. Cross-validation helps reduce variability and therefore limit problems like overfitting.

There are mainly two cross-validation schemes in use, exhaustive and non-exhaustive. In the exhaustive scheme, we leave out a fixed number of observations in each round as testing (or validation) samples, the remaining observations as training samples. This process is repeated until all possible different subsets of samples are used for testing once. For instance, we can apply **leave-one-out-cross-validation** (**LOOCV**) and let each datum be in the testing set once. For a dataset of size n, LOOCV requires n rounds of cross-validation. This can be slow when n gets large.

On the other hand, the non-exhaustive scheme, as the name implies, does not try out all possible partitions. The most widely used type of this scheme is **k-fold cross-validation**. The original data first randomly splits the data into *k* equal-sized folds. In each trail, one of these folds becomes the testing set, and the rest of the data becomes the training set. We repeat this process *k* times with each fold being the designated testing set once. Finally, we average the *k* sets of test results for the purpose of evaluation. Common values for *k* are 3, 5, and 10. The following table illustrates the setup for five folds:

Iteration	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**
1	Testing	Training	Training	Training	Training
2	Training	Testing	Training	Training	Training
3	Training	Training	Testing	Training	Training
4	Training	Training	Training	Testing	Training
5	Training	Training	Training	Training	Testing

We can also randomly split the data into training and testing set numerous times. This is formally called **holdout** method. The problem with this algorithm is that some samples may never end up in the testing set, while some may be selected multiple times in the testing set. Last but not least, **nested cross-validation** is a combination of cross-validations. It consists of the following two phases:

- The inner cross-validation, which is conducted to to find the best fit, and can be implemented as a k-fold cross-validation
- The outer cross-validation, which is used for performance evaluation and statistical analysis

We will apply cross-validation very intensively from `Chapter 3`, *Spam Email Detection with Naive Bayes*, to `Chapter 7`, *Stock Price Prediction with Regression Algorithms*. Before that, let's see cross-validation through an analogy as follows, which will help us understand it better.

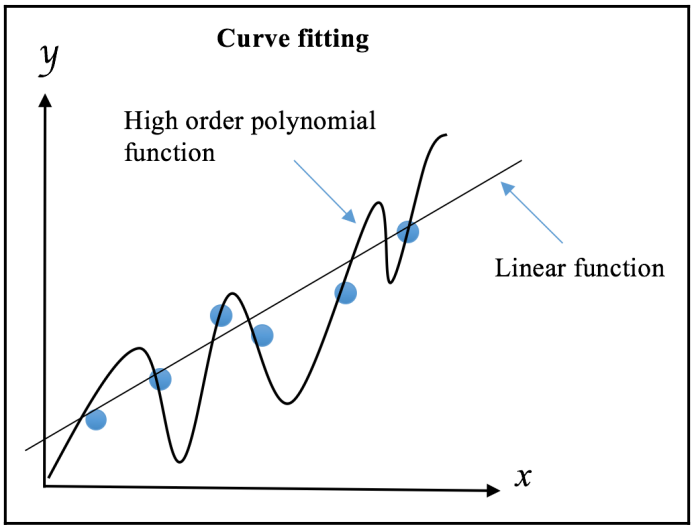
A data scientist plans to take his car to work, and his goal is to arrive before 9 am every day. He needs to decide the departure time and the route to take. He tries out different combinations of these two parameters on some Mondays, Tuesdays, and Wednesdays and records the arrival time for each trial. He then figures out the best schedule and applies it every day. However, it doesn't work quite well as expected. It turns out the scheduling model is overfit with data points gathered in the first three days and may work well on Thursdays and Fridays. A better solution would be to test the best combination of parameters derived from Mondays to Wednesdays on Thursdays and Fridays and similarly repeat this process based on different sets of learning days and testing days of the week. This analogized cross-validation ensures the selected schedule work for the whole week.

In summary, cross-validation derives a more accurate assessment of model performance by combining measures of prediction performance on different subsets of data. This technique not only reduces variances and avoids overfitting but also gives an insight into how the model will generally perform in practice.

Avoid overfitting with regularization

Another way of preventing overfitting is **regularization**. Recall that unnecessary complexity of the model is a source of overfitting just like cross-validation is a general technique to fight overfitting. Regularization adds extra parameters to the error function we are trying to minimize in order to penalize complex models.

According to the principle of Occam's Razor, simpler methods are to be favored. William Occam was a monk and philosopher who, around 1320, came up with the idea that the simplest hypothesis that fits data should be preferred. One justification is that we can invent fewer simple models than complex models. For instance, intuitively, we know that there are more high-polynomial models than linear ones. The reason is that a line ($y=ax+b$) is governed by only two parameters--the intercept b and slope a. The possible coefficients for a line span a two-dimensional space. A quadratic polynomial adds an extra coefficient to the quadratic term, and we can span a three-dimensional space with the coefficients. Therefore, it is much easier to find a model that perfectly captures all the training data points with a high order polynomial function as its search space is much larger than that of a linear model. However, these easily-obtained models generalize worse than linear models, which are more prompt to overfitting. And of course, simpler models require less computation time. The following figure displays how we try to fit a linear function and a high order polynomial function respectively to the data:



The linear model is preferable as it may generalize better to more data points drawn from the underlying distribution. We can use regularization to reduce the influence of the high orders of polynomial by imposing penalties on them. This will discourage complexity, even though a less accurate and less strict rule is learned from the training data.

We will employ regularization quite often staring from `Chapter 6`, *Click-Through Prediction with Logistic Regression*. For now, let's see the following analogy, which will help us understand it better:

A data scientist wants to equip his robotic guard dog the ability to identify strangers and his friends. He feeds it with the the following learning samples:

Male	**Young**	**Tall**	**With glasses**	**In grey**	**Friend**
Female	Middle	Average	Without glasses	In black	Stranger
Male	Young	Short	With glasses	In white	Friend
Male	Senior	Short	Without glasses	In black	Stranger
Female	Young	Average	With glasses	In white	Friend
Male	Young	Short	Without glasses	In red	Friend

The robot may quickly learn the following rules: any middle-aged female of average height without glasses and dressed in black is a stranger; any senior short male without glasses and dressed in black is a stranger; anyone else is his friend. Although these perfectly fit the training data, they seem too complicated and unlikely to generalize well to new visitors. In contrast, the data scientist limits the learning aspects. A loose rule that can work well for hundreds of other visitors could be: anyone without glasses dressed in black is a stranger.

Besides penalizing complexity, we can also stop a training procedure early as a form of regularization. If we limit the time a model spends in learning or set some internal stopping criteria, it is more likely to produce a simpler model. The model complexity will be controlled in this way, and hence, overfitting becomes less probable. This approach is called **early stopping** in machine learning.

Last but not least, it is worth noting that regularization should be kept on a moderate level, or to be more precise, fine-tuned to an optimal level. Regularization, when too small, does has make any impact; regularization, when too large, will result in underfitting as it moves the model away from the ground truth. We will explore how to achieve the optimal regularization mainly in `Chapter 6`, *Click-Through Prediction with Logistic Regression* and `Chapter 7`, *Stock Price Prediction with Regression Algorithms*.

Avoid overfitting with feature selection and dimensionality reduction

We typically represent the data as a grid of numbers (a matrix). Each column represents a variable, which we call a feature in machine learning. In supervised learning, one of the variables is actually not a feature but the label that we are trying to predict. And in supervised learning, each row is an example that we can use for training or testing. The number of features corresponds to the dimensionality of the data. Our machine learning approach depends on the number of dimensions versus the number of examples. For instance, text and image data are very high dimensional, while stock market data has relatively fewer dimensions. Fitting high dimensional data is computationally expensive and is also prone to overfitting due to high complexity. Higher dimensions are also impossible to visualize, and therefore, we can't use simple diagnostic methods.

Not all the features are useful, and they may only add randomness to our results. It is, therefore, often important to do good feature selection. Feature selection is the process of picking a subset of significant features for use in better model construction. In practice, not every feature in a dataset carries information useful for discriminating samples; some features are either redundant or irrelevant and hence can be discarded with little loss.

In principle, feature selection boils down to multiple binary decisions: whether to include a feature or not. For n features, we get 2^n feature sets, which can be a very large number for a large number of features. For example, for 10 features, we have 1,024 possible feature sets (for instance, if we are deciding what clothes to wear, the features can be temperature, rain, the weather forecast, where we are going, and so on). At a certain point, brute force evaluation becomes infeasible. We will discuss better methods in `Chapter 6`, *Click-Through Prediction with Logistic Regression,* in this book. Basically, we have two options: we either start with all the features and remove features iteratively or we start with a minimum set of features and add features iteratively. We then take the best feature sets for each iteration and then compare them.

Another common approach of reducing dimensionality reduction approach is to transform high-dimensional data in lower-dimensional space. This transformation leads to information loss, but we can keep the loss to a minimum. We will cover this in more detail later on.

Preprocessing, exploration, and feature engineering

Data mining, a buzzword in the 1990 is the predecessor of data science (the science of data). One of the methodologies popular in the data mining community is called **cross industry standard process for data mining** (**CRISP DM**). CRISP DM was created in 1996, and is still used today. I am not endorsing CRISP DM, however I like its general framework. The CRISP DM consists of the following phases, which are not mutually exclusive and can occur in parallel:

- **Business understanding**: This phase is often taken care of by specialized domain experts. Usually we have a business person formulate a business problem, such as selling more units of a certain product.
- **Data understanding**: This is also a phase, which may require input from domain experts, however, often a technical specialist needs to get involved more than in the business understanding phase. The domain expert may be proficient with spreadsheet programs, but have trouble with complicated data. In this book, I usually call this phase **exploration**.
- **Data preparation**: This is also a phase where a domain expert with only Excel know-how may not be able to help you. This is the phase where we create our training and test datasets. In this book I usually call this phase **preprocessing**.
- **Modeling**: This is the phase, which most people associate with machine learning. In this phase we formulate a model, and fit our data.
- **Evaluation**: In this phase, we evaluate our model, and our data to check whether we were able to solve our business problem.
- **Deployment**: This phase usually involves setting up the system in a production environment (it is considered good practice to have a separate production system). Typically this is done by a specialized team.

When we learn, we require high quality learning material. We can't learn from gibberish, so we automatically ignore anything that doesn't make sense. A machine learning system isn't able to recognize gibberish, so we need to help it by cleaning the input data. It is often claimed that cleaning the data forms a large part of machine learning. Sometimes cleaning is already done for us, but you shouldn't count on it. To decide how to clean the data, we need to be familiar with the data. There are some projects, which try to automatically explore the data, and do something intelligent, like producing a report. For now, unfortunately, we don't have a solid solution, so you need to do some manual work.

We can do two things, which are not mutually exclusive: first scan the data and second visualize the data. This also depends on the type of data we are dealing with; whether we have a grid of numbers, images, audio, text, or something else. At the end, a grid of numbers is the most convenient form, and we will always work towards having numerical features. I will pretend that we have a table of numbers in the rest of this section.

We want to know if features miss values, how the values are distributed, and what type of features we have. Values can approximately follow a normal distribution, a binomial distribution, a Poisson distribution, or another distribution altogether. Features can be binary: either yes or no, positive or negative, and so on. They can also be categorical: pertaining to a category, for instance continents (Africa, Asia, Europe, Latin America, North America, and so on). Categorical variables can also be ordered—for instance high, medium, and low. Features can also be quantitative, for example temperature in degrees or price in dollars.

Feature engineering is the process of creating or improving features. It's more of a dark art than a science. Features are often created based on common sense, domain knowledge, or prior experience. There are certain common techniques for feature creation, however there is no guarantee that creating new features will improve your results. We are sometimes able to use the clusters found by unsupervised learning as extra features. Deep neural networks are often able to create features automatically.

Missing values

Quite often we miss values for certain features. This could happen for various reasons. It can be inconvenient, expensive, or even impossible to always have a value. Maybe we were not able to measure a certain quantity in the past, because we didn't have the right equipment, or we just didn't know that the feature was relevant. However, we are stuck with missing values from the past. Sometimes it's easy to figure out that we miss values and we can discover this just by scanning the data, or counting the number of values we have for a feature and comparing to the number of values we expect based on the number of rows. Certain systems encode missing values with, for example, values such as 999999. This makes sense if the valid values are much smaller than 999999. If you are lucky, you will have information about the features provided by whoever created the data in the form of a data dictionary or metadata.

Once we know that we miss values the question arises of how to deal with them. The simplest answer is to just ignore them. However, some algorithms can't deal with missing values, and the program will just refuse to continue. In other circumstances, ignoring missing values will lead to inaccurate results. The second solution is to substitute missing values by a fixed value—this is called **imputing**.

We can impute the arithmetic mean, median or mode of the valid values of a certain feature. Ideally, we will have a relation between features or within a variable that is somewhat reliable. For instance, we may know the seasonal averages of temperature for a certain location and be able to impute guesses for missing temperature values given a date.

Label encoding

Humans are able to deal with various types of values. Machine learning algorithms with some exceptions need numerical values. If we offer a string such as `Ivan`, unless we are using specialized software the program will not know what to do. In this example, we are dealing with a categorical feature, names probably. We can consider each unique value to be a label. (In this particular example, we also need to decide what to do with the case-is `Ivan` the same as `ivan`). We can then replace each label by an **integer-label encoding**. This approach can be problematic, because the learner may conclude that there is an ordering.

One-hot-encoding

The **one-of-K** or **one-hot-encoding** scheme uses dummy variables to encode categorical features. Originally it was applied to digital circuits. The dummy variables have binary values like bits, so they take the values zero or one (equivalent to true or false). For instance, if we want to encode continents, we will have dummy variables, such as `is_asia`, which will be true if the continent is Asia and false otherwise. In general, we need as many dummy variables, as there are unique labels minus one. We can determine one of the labels automatically from the dummy variables, because the dummy variables are exclusive. If the dummy variables all have a false value, then the correct label is the label for which we don't have a dummy variable. The following table illustrates the encoding for continents:

	`Is_africa`	`Is_asia`	`Is_europe`	`Is_south_america`	`Is_north_america`
Africa	True	False	False	False	False
Asia	False	True	False	False	False
Europe	False	False	True	False	False
South America	False	False	False	True	False
North America	False	False	False	False	True
Other	False	False	False	False	False

The encoding produces a matrix (grid of numbers) with lots of zeroes (false values) and occasional ones (true values). This type of matrix is called a **sparse matrix**. The sparse matrix representation is handled well by the SciPy package, and shouldn't be an issue. We will discuss the SciPy package later in this chapter.

Scaling

Values of different features can differ by orders of magnitude. Sometimes this may mean that the larger values dominate the smaller values. This depends on the algorithm we are using. For certain algorithms to work properly we are required to scale the data. There are several common strategies that we can apply:

- **Standardization** removes the mean of a feature and divides by the standard deviation. If the feature values are normally distributed, we will get a Gaussian, which is centered around zero with a variance of one.
- If the feature values are not normally distributed, we can remove the median and divide by the interquartile range. The interquartile range is a range between the first and third quartile (or 25th and 75th percentile).
- **Scaling** features to a range is a common choice of range which is a range between zero and one.

Polynomial features

If we have two features *a* and *b*, we can suspect that there is a polynomial relation, such as *a2 + ab + b2*. We can consider each term in the sum to be a feature, in this example we have three features. The product *ab* in the middle is called an **interaction**. An interaction doesn't have to be a product, although this is the most common choice, it can also be a sum, a difference or a ratio. If we are using a ratio to avoid dividing by zero, we should add a small constant to the divisor and dividend. The number of features and the order of the polynomial for a polynomial relation are not limited. However, if we follow Occam's razor we should avoid higher order polynomials and interactions of many features. In practice, complex polynomial relations tend to be more difficult to compute and not add much value, but if you really need better results they may be worth considering.

Power transformations

Power transforms are functions that we can use to transform numerical features into a more convenient form, for instance to conform better to a normal distribution. A very common transform for values, which vary by orders of magnitude, is to take the logarithm. Taking the logarithm of a zero and negative values is not defined, so we may need to add a constant to all the values of the related feature before taking the logarithm. We can also take the square root for positive values, square the values, or compute any other power we like.

Another useful transform is the Box-Cox transform named after its creators. The Box-Cox transform attempts to find the best power need to transform the original data into data that is closer to the normal distribution. The transform is defined as follows:

$$y_i^{(\lambda)} = \begin{cases} \dfrac{y_i^{\lambda} - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(y_i) & \text{if } \lambda = 0, \end{cases} \tag{1.2}$$

Binning

Sometimes it's useful to separate feature values into several bins. For example, we may be only interested whether it rained on a particular day. Given the precipitation values, we can binarize the values, so that we get a true value if the precipitation value is not zero, and a false value otherwise. We can also use statistics to divide values into high, low, and medium bins.

The binning process inevitably leads to loss of information. However, depending on your goals this may not be an issue, and actually reduce the chance of overfitting. Certainly there will be improvements in speed and memory or storage requirements.

Combining models

In (high) school we sit together with other students, and learn together, but we are not supposed to work together during the exam. The reason is, of course, that teachers want to know what we have learned, and if we just copy exam answers from friends, we may have not learned anything. Later in life we discover that teamwork is important. For example, this book is the product of a whole team, or possibly a group of teams.

Clearly a team can produce better results than a single person. However, this goes against Occam's razor, since a single person can come up with simpler theories compared to what a team will produce. In machine learning we nevertheless prefer to have our models cooperate with the following schemes:

- Bagging
- Boosting
- Stacking
- Blending
- Voting and averaging

Bagging

Bootstrap aggregating or **bagging** is an algorithm introduced by Leo Breiman in 1994, which applies Bootstrapping to machine learning problems. Bootstrapping is a statistical procedure, which creates datasets from existing data by sampling with replacement. Bootstrapping can be used to analyze the possible values that arithmetic mean, variance, or another quantity can assume.

The algorithm aims to reduce the chance of overfitting with the following steps:

1. We generate new training sets from input train data by sampling with replacement.
2. Fit models to each generated training set.
3. Combine the results of the models by averaging or majority voting.

Boosting

In the context of supervised learning we define weak learners as learners that are just a little better than a baseline such as randomly assigning classes or average values. Although weak learners are weak individually like ants, together they can do amazing things just like ants can. It makes sense to take into account the strength of each individual learner using weights. This general idea is called **boosting**. There are many boosting algorithms; boosting algorithms differ mostly in their weighting scheme. If you have studied for an exam, you may have applied a similar technique by identifying the type of practice questions you had trouble with and focusing on the hard problems.

Face detection in images is based on a specialized framework, which also uses boosting. Detecting faces in images or videos is a supervised learning. We give the learner examples of regions containing faces. There is an imbalance, since we usually have far more regions (about ten thousand times more) that don't have faces. A cascade of classifiers progressively filters out negative image areas stage by stage. In each progressive stage, the classifiers use progressively more features on fewer image windows. The idea is to spend the most time on image patches, which contain faces. In this context, boosting is used to select features and combine results.

Stacking

Stacking takes the outputs of machine learning estimators and then uses those as inputs for another algorithm. You can, of course, feed the output of the higher-level algorithm to another predictor. It is possible to use any arbitrary topology, but for practical reasons you should try a simple setup first as also dictated by Occam's razor.

Blending

Blending was introduced by the winners of the one million dollar Netflix prize. Netflix organized a contest with the challenge of finding the best model to recommend movies to their users. Netflix users can rate a movie with a rating of one to five stars. Obviously each user wasn't able to rate each movie, so the user movie matrix is sparse. Netflix published an anonymized training and test set. Later researchers found a way to correlate the Netflix data to IMDB data. For privacy reasons, the Netflix data is no longer available. The competition was won in 2008 by a group of teams combining their models. Blending is a form of stacking. The final estimator in blending, however, trains only on a small portion of the train data.

Voting and averaging

We can arrive at our final answer through majority voting or averaging. It's also possible to assign different weights to each model in the ensemble. For averaging, we can also use the geometric mean or the harmonic mean instead of the arithmetic mean. Usually combining the results of models, which are highly correlated to each other doesn't lead to spectacular improvements. It's better to somehow diversify the models, by using different features or different algorithms. If we find that two models are strongly correlated, we may, for example, decide to remove one of them from the ensemble, and increase the weight of the other model proportionally.

Installing software and setting up

For most projects in this book we need scikit-learn (refer to, `http://scikit-learn.org/stable/install.html`) and matplotlib (refer to, `http://matplotlib.org/users/installing.html`). Both packages require NumPy, but we also need SciPy for sparse matrices as mentioned before. The scikit-learn library is a machine learning package, which is optimized for performance as a lot of the code runs almost as fast as equivalent C code. The same statement is true for NumPy and SciPy. There are various ways to speed up the code, however they are out of scope for this book, so if you want to know more, please consult the documentation.

matplotlib is a plotting and visualization package. We can also use the seaborn package for visualization. Seaborn uses matplotlib under the hood. There are several other Python visualization packages that cover different usage scenarios. matplotlib and seaborn are mostly useful for the visualization for small to medium datasets. The NumPy package offers the `ndarray` class and various useful array functions. The `ndarray` class is an array, that can be one or multi-dimensional. This class also has several subclasses representing matrices, masked arrays, and heterogeneous record arrays. In machine learning we mainly use NumPy arrays to store feature vectors or matrices composed of feature vectors. SciPy uses NumPy arrays and offers a variety of scientific and mathematical functions. We also require the pandas library for data wrangling.

In this book, we will use Python 3. As you may know, Python 2 will no longer be supported after 2020, so I strongly recommend switching to Python 3. If you are stuck with Python 2 you should still be able to modify the example code to work for you. In my opinion, the Anaconda Python 3 distribution is the best option. Anaconda is a free Python distribution for data analysis and scientific computing. It has its own package manager, conda. The distribution includes more than 200 Python packages, which makes it very convenient. For casual users, the Miniconda distribution may be the better choice. Miniconda contains the conda package manager and Python.

The procedures to install Anaconda and Miniconda are similar. Obviously, Anaconda takes more disk space. Follow the instructions from the Anaconda website at `http://conda.pydata.org/docs/install/quick.html`. First, you have to download the appropriate installer for your operating system and Python version. Sometimes you can choose between a GUI and a command line installer. I used the Python 3 installer, although my system Python version is 2.7. This is possible since Anaconda comes with its own Python. On my machine the Anaconda installer created an `anaconda` directory in my home directory and required about 900 MB. The Miniconda installer installs a `miniconda` directory in your home directory. Installation instructions for NumPy are at `http://docs.scipy.org/doc/numpy/user/install.html`.

Alternatively install NumPy with `pip` as follows:

```
$ [sudo] pip install numpy
```

The command for Anaconda users is:

```
$ conda install numpy
```

To install the other dependencies, substitute NumPy by the appropriate package. Please read the documentation carefully, not all options work equally well for each operating system. The pandas installation documentation is at `http://pandas.pydata.org/pandas-docs/dev/install.html`.

Troubleshooting and asking for help

Currently the best forum is at `http://stackoverflow.com`. You can also reach out on mailing lists or IRC chat. The following is a list of mailing lists:

- Scikit-learn: `https://lists.sourceforge.net/lists/listinfo/scikit-learn-general`.
- NumPy and SciPy mailing list: `https://www.scipy.org/scipylib/mailing-lists.html`.

IRC channels:

- #scikit-learn @ freenode
- #scipy @ freenode

Summary

We just finished our first mile in the Python and machine learning journey! Through this chapter, we got familiar with the basics of machine learning. We started with what machine learning is all about, the importance of machine learning (data technology era) and its brief history and recent development as well. We also learned typical machine learning tasks and explored several essential techniques of working with data and working with models. Now that we are equipped with basic machine learning knowledge, and also get software and tools set up, let's get ready for the real-world machine learning examples ahead.