Roberto Battiti · Mauro Brunato

The LION Way

Machine Learning *plus* Intelligent Optimization

LIONsolver

lionsolver.com

Edition: June 2013

ROBERTO BATTITI AND MAURO BRUNATO.
The LION way. Machine Learning plus Intelligent Optimization.
Lionsolver Inc., Los Angeles, June 2013.

For inquiries about this book and related materials, refer to the book's web page:
http://lionsolver.com/LIONbook

Chapter 1

Introduction

Fatti non foste a viver come bruti ma per seguir virtute e canoscenza
You were not made to live like beasts, but to follow virtue and knowledge.
(Dante Alighieri)

1.1 Learning and Intelligent Optimization: a prairie fire

Almost by definition, *optimization*, the **automated search for improving solutions**, is a source of a tremendous power for continually improving processes, decisions, prod-

Latest chapter revision: November 14, 2013

ucts and services. This is related to decision making (picking the best among a set of given possible solutions) but goes beyond that, by actively searching for and **creatively generating new solutions**.

Almost all business problems can be formulated as **finding an optimal decision $\boldsymbol{x}$ in order to maximize a measure** *goodness*$(\boldsymbol{x})$. For a concrete mental image, think of x as a collective variable $\boldsymbol{x} = (x_1, \ldots, x_n)$ describing the settings of one or more knobs to be rotated, choices to be made, parameters to be fixed. In marketing, $\boldsymbol{x}$ can be a vector of values specifying the budget allocation to different campaigns (TV, web, newspapers), *goodness*$(\boldsymbol{x})$ can be a count of the new customers generated by the campaign. In website optimization, $\boldsymbol{x}$ can be related to using images, links, topics, text of different size, *goodness*$(\boldsymbol{x})$ can be the conversion rate from a casual visitor to a customer. In engineering, $\boldsymbol{x}$ can be the set of all design choices of a car motor, *goodness*$(\boldsymbol{x})$ can be the miles per gallon travelled.

Formulating a problem as "optimize a goodness function" also helps by encouraging decision makers **to use quantitative goals, to understand intents in a measurable manner, to focus on policies more than on implementation details**. Getting stuck in implementations at the point of forgetting goals is a plague infecting businesses and limiting their speed of movement when external conditions change.

Automation is the key: after formulating the problem, deliver the goodness model to a computer which will search for one or more optimal choices. And when conditions or priorities change, just revise the goals and restart the optimization process, *et voilà*. To be honest, CPU time is an issue and globally optimal solutions are not always guaranteed, but for sure the speed and latitude of the search by computers surpass human capabilities by a huge and growing factor.

But the enormous power of optimization is still largely stifled in most real-world contexts. The main reason blocking its widespread adoption is that **standard mathematical optimization assumes the existence of a function** to be maximized, in other words, an explicitly defined model *goodness*$(\boldsymbol{x})$. Now, in most real-world business contexts this function does not exist or is extremely difficult and costly to build by hand. Try asking a CEO "Can you please tell me the mathematical formula that your business is optimizing?", probably this is not the best way to start a consultancy job. For sure, a manager has *some* ideas about objectives and tradeoffs, but these objectives are not specified as a mathematical model, they are dynamic, changing in time, fuzzy and subject to estimation errors and human learning processes. *Gut feelings* and intuition are assumed to substitute clearly specified, quantitative and data-driven decision processes.

If optimization is fuel, the match to light the fire is **machine learning**. Machine learning comes to the rescue by renouncing to a clearly specified goal *goodness*$(\boldsymbol{x})$: **the model can be built by machine learning from abundant data**.

Learning and Intelligent Optimization (LION) is the combination of learning from data and optimization applied to solve complex and dynamic problems. The LION way is about increasing the automation level and connecting data directly to decisions and actions. More **power is directly in the hands of decision makers in a self-service manner**, without resorting to intermediate layers of data scientists. LION is a complex array of mechanisms, like the engine in an automobile, but the user (driver) does not need to know the inner-workings of the engine in order to realize tremendous benefits. LION's adoption will create a prairie fire of innovation which will reach most businesses in the next decades. Businesses, like plants in wildfire-prone ecosystems, will survive and prosper by adapting and embracing LION techniques, or they risk being transformed from giant trees to ashes by the spreading competition.

The questions to be asked in the LION paradigm are not about mathematical goodness models but about abundant data, expert judgment of concrete options (examples of success cases), interactive definition of success criteria, at a level which makes a human person at ease with his mental models. For example, in marketing, relevant data can describe the money allocation and success of previous campaigns, in engineering they can describe experiments about motor designs (real or simulated) and corresponding fuel consumption.

1.2 Searching for gold and for partners

Machine learning for optimization needs data. Data can be created by **the previous history of the optimization process** or by **feedback by decision makers**.

Figure 1.1: Danie Gerhardus Krige, the inventor of *Kriging*.

To understand the two contexts, let's start with two concrete examples. Danie G. Krige, a South African Mining Engineer, had a problem to solve: how to identify the best coordinates $\boldsymbol{x}$ on a geographical map where to dig gold mines [25]. Around 1951 he began his pioneering work on applying insights in statistics to the valuation of new gold mines, using a limited number of boreholes. The function to be optimized was a glittering $Gold(\boldsymbol{x})$, the quantity of gold extracted from a mine at position $\boldsymbol{x}$. For sure, evaluating $Gold(\boldsymbol{x})$ at a new position $\boldsymbol{x}$ was very costly. As you can imagine, digging a new mine is not a quick and simple job. But after digging some exploratory mines, engineers accumulate **knowledge in the form of examples** relating coordinates $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$... and the corresponding gold quantities $Gold(\boldsymbol{x}_1)$, $Gold(\boldsymbol{x}_2)$, $Gold(\boldsymbol{x}_3)$. Krige's intuition was to use these examples (**data about the previous history of the optimization process**) to build a model of the function $Gold(\boldsymbol{x})$, let's call it $Model_{Gold}(\boldsymbol{x})$, which could generalize the experimental results (by giving output values for each position $\boldsymbol{x}$), and which could be used by an optimizer to identify the next point to dig, by finding the position $\boldsymbol{x}_{best}$ maximizing $Model_{Gold}(\boldsymbol{x})$.

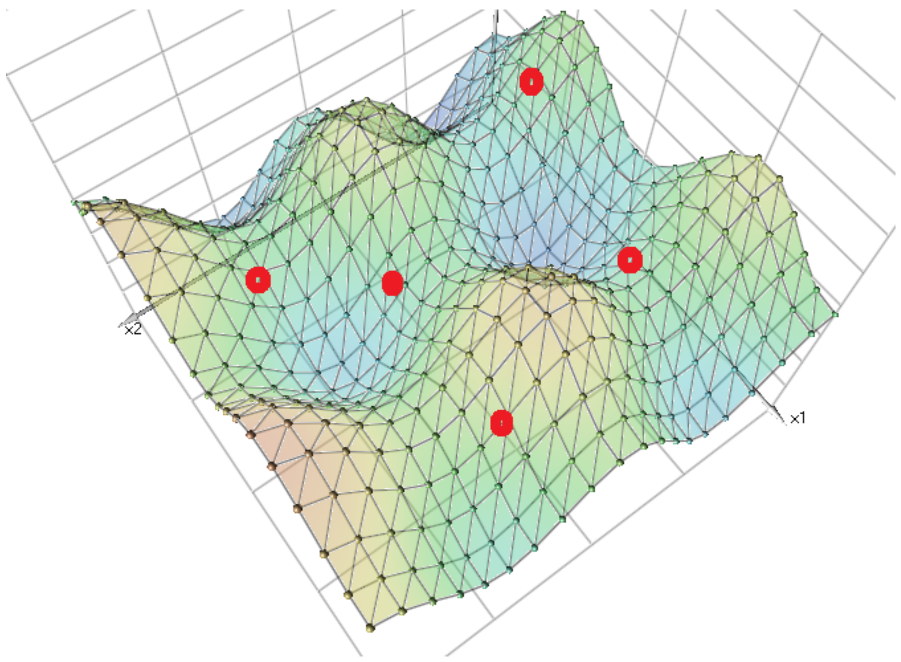


Figure 1.2: *Kriging* method constructing a model from samples. Some samples are shown as red dots. Height and color of the surface depend on gold content.

Think of this model as starting from "training" information given by pins at the boreholes locations, with height proportional to the gold content found, and building a complete surface over the geographical area, with height at a given position proportional to the estimated gold content (Fig. 1.2). Optimizing means now identifying the highest point on this model surface, and the corresponding next position where to dig the next mine.

This technique is now called *Kriging* and it is based on the idea that the value at an unknown point should be the average of the known values at its neighbors, weighted by the neighbors' distance to the unknown point. *Gaussian processes, Bayesian inference, splines* refer to related techniques.

For the second example about getting **feedback by decision makers**, let's imagine a dating service: you pay and you are delivered a contact with the best possible partner from millions of waiting candidates. In Kriging the function to be optimized exists, it is only extremely difficult to evaluate. In this case, it is difficult to assume that a function *CandidateEvaluation*$(\boldsymbol{x})$ exists, relating individual characteristics $\boldsymbol{x}$ like beauty, intelligence, etc. to your individual preference. If you are not convinced and you think that this function does exist, as a homework you are invited to define in precise mathematical terms the *CandidateEvaluation* for your ideal partner. Even assuming that you can identify some building blocks and make them precise, like *Beauty*$(\boldsymbol{x})$ and *Intelligence*$(\boldsymbol{x})$, you will have difficulties in combining the two objectives *a priori*, before starting the search for the optimal candidate. Questions like: "How many IQ points are you willing to sacrifice for one less beauty point?" or "Is beauty more important than intelligence for you? By how much?" will be difficult to answer. Even if you are tortured and deliver an initial answer, for sure you will not trust the optimization and you will probably like to give a look at the real candidate before paying the matching service and be satisfied. You will want to know the $\boldsymbol{x}$ and not only the value of the provisional function the system will optimize. Only after considering different candidates and giving feedback to the matching service you may hope to identify your best match.

In other words, some information about the function to be optimized is missing at the beginning, and only the decision maker will be able to fine-tune the search process. Solving many if not most real-world problems requires **iterative processes with learning involved**. The user will learn and adjust his preferences after knowing more and more cases, the system will build models of the user preferences from his feedback. The steps will continue until the user is satisfied or the time allocated for the decision is finished.

1.3 All you need is data

Let's continue with some motivation for business users. If this is not your case you can safely skip this part and continue with Section 1.6.

Enterprises are awash in big data. With the explosion in social network usage, rapidly expanding e-commerce, and the rise of the *internet of things*, the web is creating a tsunami of structured and unstructured data, driving more than $200 billion in spending on information technology through 2016. Recent evidence also indicates a decline in the use of standard business intelligence platforms as enterprises are forced to consider a mass of unstructured data that has uncertain real-world value. For example, social networks create vast amounts of data, most of which resists classification and the rigid hierarchies of traditional data. How do you measure the value of a Facebook *Like*? Moreover, unstructured data requires an adaptive approach to analysis. How does the value of a *Like* change over time? These questions are driving the adoption of advanced methodologies for data modeling, adaptive learning, and optimization.

LION tools bring a competitive edge to the enterprise by facilitating interaction between domain experts, decision makers, and the software itself. Programmed with "reactive" algorithms, the software is capable of self-improvement and rapid adaptation to new business objectives. The strength in this approach lies in abilities that are often associated with the human brain: learning from past experiences, **learning on the job**, coping with incomplete information, and quickly adapting to new situations.

This inherent flexibility is critical where decisions depend on factors and priorities that are not identifiable before starting the solution process. For example, what factors should be used and to what degree do they matter in scoring the value of a marketing lead? With the LION approach, the answer is: "It doesn't matter." The system will begin to train itself, and successive data plus feedback by the final user will rapidly improve performance. Experts —in this case marketing managers— can further refine the output by contributing their points of view.

1.4 Beyond traditional business intelligence

Every business has two fundamental needs from their data:

1. to **understand** their current business processes and related performance;
2. to **improve** profitability by making informed and rational decisions about critical business factors.

Traditional business intelligence excels at mapping and recording (or visualizing) historical performance. Building these maps meant hiring a top-level consultancy or on-boarding of personnel specifically trained in statistics, analysis, and databases. Experts had to design data extraction and manipulation processes and hand them over to programmers for the actual execution. This is a slow and cumbersome process when you consider the dynamic environment of most businesses.

As a result, enterprises relying heavily on BI are using snapshots of performance to try to understand and react to conditions and trends. Like driving a car by looking into the rearview mirror, it's most likely that you're going to hit something. For the enterprise, it appears that they already have hit a wall of massive, unexpected, semi-structured data. The projected IT spending for the next four years in big data reflects how hard the collision has been [1]. Enterprises are scrambling to replace antiquated technologies with those that can make sense of the present and provide guidance for the future.

Predictive analytics does better in trying to anticipate the effect of decisions, but the real power comes from the **integration of data-driven models with optimization**, the automated search for improving solutions. From the data directly to the best improving plan, **from actionable insight to actions**!

1.5 Implementing LION

The steps for fully adopting LION as a business practice will vary depending on the current business state, and most importantly, the state of the underlying data. It goes without saying that it is easier and less costly to introduce the paradigm if data capture has already been established. For some enterprises, legacy systems can prove quite costly to migrate, as there is extensive cleaning involved. This is where skilled service providers can play a valuable role.

Beyond cleaning and establishing the structure of the underlying data, the most important factor is to establish collaboration between the data analytics team and their business end-user customers. By its nature, LION presents a way to collectively discover the hidden potential of structured and semi-structured data stores. The key in having the data analytics team work effectively alongside the business end-user is to enable changing business objectives to quickly reflect into the models and outputs. The introduction of LION methods can help analytics teams generate radical changes in the value-creation chain, revealing hidden opportunities and increasing the speed by which their business counterparts can respond to customer requests and to changes in the market.

The LION way is a radically disruptive **intelligent approach to uncover hidden value, quickly adapt to changes and improve businesses**. Through proper planning and implementation, LION helps enterprises to lead the competition and avoid being burned by wild prairie fires.

[1]See for example reports by Gartner, a leading information technology research and advisory company.

1.6 A “hands on” approach

Because this book is about (machine) learning from examples we need to be coherent: most of the content follows the learning from examples and **learning by doing** principle. The different techniques are introduced by presenting the basic theory, and concluded by giving the “gist to take home”. Experimentation on real-world cases is encouraged with the examples and software in the book website. This is the best way to avoid the impression that LION techniques are only for experts and not for practitioners in different positions, interested in quick and measurable results.

Some of the theoretical parts can be skipped for a first adoption. But some knowledge of the theory is critical both for developing new and more advanced LION tools and for a more competent use of the technology. We tried to keep both developers and final users in mind in our effort.

Accompanying data, instructions and short tutorial movies will be posted on the book website `http://lionsolver.com/LIONbook`.

Wikipedia has been mined for some standard statistical and mathematical results, and for some explanatory images. A sign of gratitude goes to the countless contributors of this open encyclopedic effort (including the sons of the authors). Venice photo by LION4@VENICE 2010. Dante’s painting in the Introduction by Domenico di Michelino, Florence, 1465. Brain drawing in the Neural Networks chapter by Leonardo da Vinci (14521519). Photo of prof. Vapnik in the SVM chapter from Yann LeCun. Venice painting in the Democracy chapter by Canaletto, 1730. Painting in the Clustering chapter by Michelangelo Buonarroti (Sistine Chapel), 1541. Painting in Appendix B by Giacomo Balla, Form-Spirit Transformation, 1918, Appendix C Kandinsky, Several Circles, 1926, Appendix D Umberto Boccioni, The City Rises, 1910. A detailed list of references for the images will appear here in the final version of this free book.

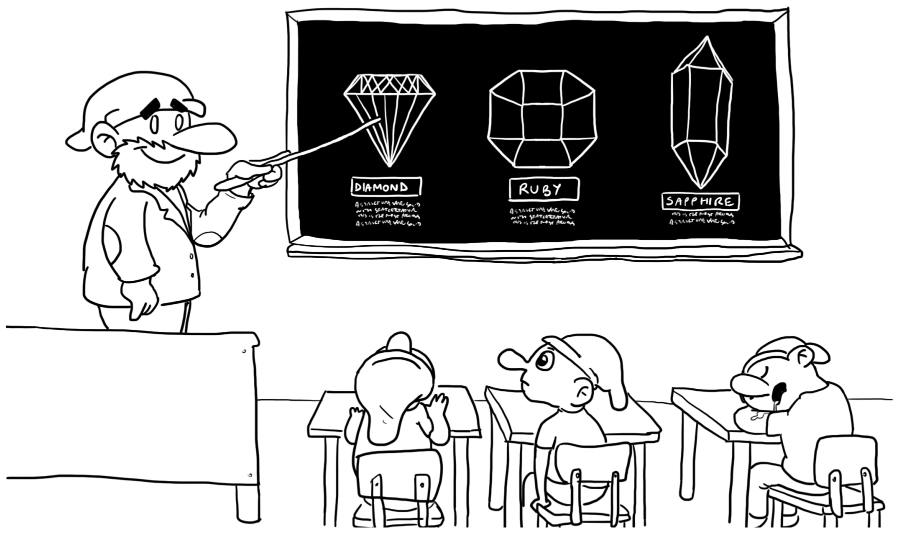
Last but not least, we are happy to acknowledge the growing contribution of our readers to the quality of this free book including Drake Pruitt, Dinara Mukhlisullina, Antonio Massaro, Fred Glover, Patrizia Nardon, Yaser Abu-Mostafa, Marco Dallariva, Enrico Sartori, Danilo Tomasoni, Nick Maravich, Jon Lehto, George Hart, Markus

Dreyer. This list is updated periodically, please email the authors if your name is missing.

Chapter 2

Lazy learning: nearest neighbors

Natura non facit saltus
Nature does not make jumps



If you still remember how you learnt to read, you have everything it takes to understand **learning from examples**, in particular supervised learning. Your parents and your teachers presented you with examples of written characters (“a”,“b”,“c”. . .) and told you: This is an “a”, this is a “b”, . . .

For sure, they did not specify mathematical formulas or precise rules for the geometry of “a” “b” “c”. . . They just presented **labeled examples**, in many different styles, forms, sizes, colors. From those examples, after some effort and some mistakes, your brain managed not only to recognize the examples in a correct manner, which you can do via memorization, but to extract the underlying patterns and regularities, to filter out

Latest chapter revision: November 14, 2013

Figure 2.1: Mushroom hunting requires classifying edible and poisonous species (public domain image by George Chernilevsky).

irrelevant “noise” (like the color) and to **generalize by recognizing new cases**, not seen during the training phase. A natural but remarkable result indeed. It did not require advanced theory, it did not require a PhD. Wouldn’t it be nice if you could solve business problems in the same natural and effortless way? The LION unification of learning from data and optimization is the way and we will start from this familiar context.

In **supervised learning** a system is trained by a supervisor (teacher) giving **labeled examples**. Each example is an array, a *vector* of input parameters x called **features** with an associated output label y.

The authors live in an area of mountains and forests and a very popular pastime is mushroom hunting. Popular and fun, but deadly if the wrong kind of mushroom is eaten. Kids are trained early to distinguish between edible and poisonous mushrooms. Tourists can buy books showing photos and characteristics of both classes, or they can bring mushrooms to the local Police and have them checked by experts for free.

Let’s consider a simplified example, and assume that two parameters, like the height and width of the mushrooms are sufficient to discriminate them, as in Fig. 2.2. In general, imagine many more input parameters, like color, geometrical characteristics, smell, etc., and a much more confused distribution of positive (edible) and negative cases.

Lazy beginners in mushroom picking follow a simple pattern. They do not study anything before picking; after all, they are in Trentino for vacation, not for work. When

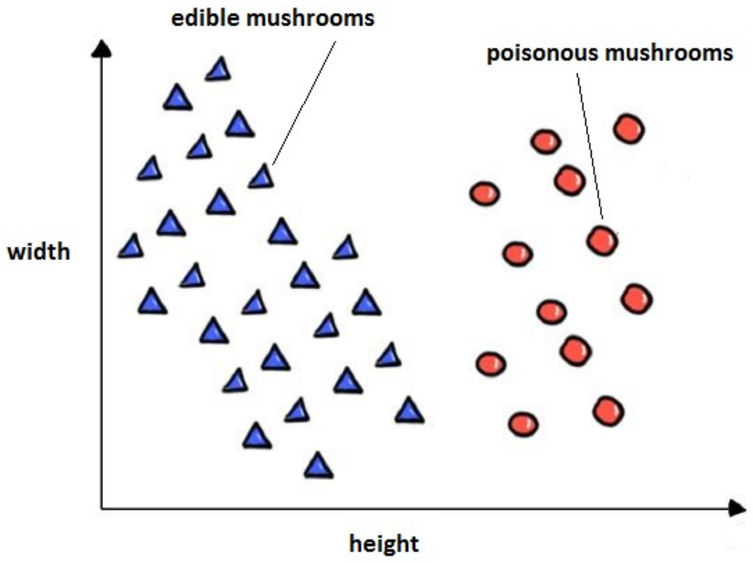


Figure 2.2: A toy example: two features (width and height) to classify edible and poisonous mushrooms.

they spot a mushroom they search in the book for images of similar ones and then double-check for similar characteristics listed in the details. This is a practical usage of the **lazy "nearest neighbor" method of machine learning**.

Why does this simple method work? The explanation is in the *Natura non facit saltus* (Latin for "nature does not make jumps") principle. Natural things and properties change gradually, rather than suddenly. If you consider a prototype edible mushroom in your book, and the one you are picking has very similar characteristics, you may assume that it is edible too.

Disclaimer: do not use this toy example to classify real mushrooms, every classifier has a probability of making mistakes and *false negative* classifications of mushrooms (classifying it as non-poisonous when in fact it is) can be very harmful to your health.

2.1 Nearest Neighbors Methods

The **nearest-neighbors** basic form of learning, also related to **instance-based learning**, **case-based** or **memory-based**, works as follows. The labeled examples (inputs and corresponding output label) are stored and no action is taken until a new input pattern demands an output value. These systems are called **lazy learners**: they do nothing

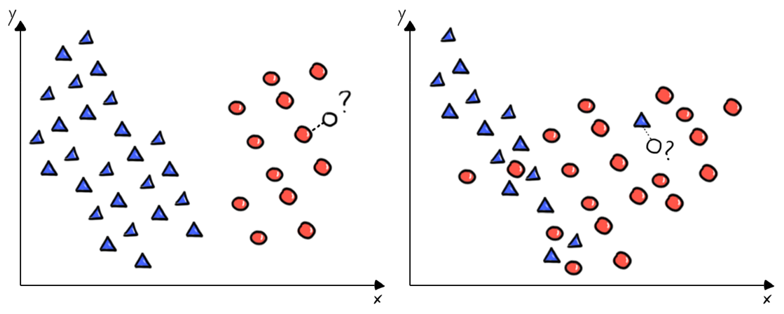


Figure 2.3: Nearest neighbor classification: a clear situation (left), a more confusing situation (right). In the second case the nearest neighbor to the query point with the question mark belongs to the wrong class although most other close neighbors belong to the right class.

but store the examples until the user interrogates them. When a new input pattern arrives, the memory is searched for examples which are *near* the new pattern, and the output is determined by retrieving the stored outputs of the close patterns, as shown in Fig. 2.3. Over a century old, this form of data mining is still being used very intensively by statisticians and machine learners alike, both for classification and for regression problems.

A simple version is that the output for the new input is simply that of the *closest* example in memory. If the task is to classify mushrooms as edible or poisonous, a new mushroom is classified in the same class as the most similar mushrooms in the memorized set.

Although very simple indeed, surprisingly this technique can be very effective in many cases. After all, it pays to be lazy! Unfortunately, the time to recognize a new case can grow in a manner proportional to the number of stored examples unless less lazy methods are used. Think about a student who is just collecting books, and then reading them when confronted with a problem to solve.

A more robust and flexible technique considers a set of k nearest neighbors instead of one, not surprisingly it is called **k-nearest-neighbors (KNN)**. The flexibility is given by different possible classification techniques. For example, the output can be that of the **majority** of the k neighbors outputs. If one wants to be on the safer side, one may decide to classify the new case only if all k outputs agree (**unanimity rule**), and to report "unknown" in the other cases (this can be suggested for classifying edible mushrooms: if "unknown" contact the local mushroom police for guidance).

If one considers **regression** (the prediction of a real number, like the content of poison in a mushroom), the output can be obtained as a simple average of the outputs corresponding to the k closest examples.

Of course, the vicinity of the k examples to the new case can be very different and in some cases it is reasonable that closer neighbors should have a bigger influence on the output. In the **weighted k-nearest-neighbors** technique (WKNN), the weights depend on the distance.

Let $k \leq \ell$ be a fixed positive integer (ℓ is the number of labeled examples), and consider a feature vector $\boldsymbol{x}$. A simple algorithm to estimate its corresponding outcome y consists of two steps:

1. Find within the training set the k indices $i_1, \ldots, i_k$ whose feature vectors $\boldsymbol{x}_{i_1}, \ldots, \boldsymbol{x}_{i_k}$ are nearest (according to a given feature-space metric) to the given $\boldsymbol{x}$ vector.

2. Calculate the estimated outcome y by the following average, weighted with the inverse of the distance between feature vectors:

$$y = \frac{\displaystyle\sum_{j=1}^{k} \frac{y_{i_j}}{d(\boldsymbol{x}_{i_j}, \boldsymbol{x}) + d_0}}{\displaystyle\sum_{j=1}^{k} \frac{1}{d(\boldsymbol{x}_{i_j}, \boldsymbol{x}) + d_0}}; \tag{2.1}$$

   where $d(\boldsymbol{x}_i, \boldsymbol{x})$ is the distance between the two vectors in the feature space (for example the Euclidean distance), and d_0 is a small constant offset used to avoid division by zero. The larger d_0, the larger the relative contribution of far away points to the estimated output. If d_0 goes to infinity, the predicted output tends to the *mean* output over all training examples.

The WKNN algorithm is simple to implement, and it often achieves low estimation errors. Its main drawbacks are the massive amounts of memory required, and the computational cost of the testing phase. A way to reduce the amount of memory required considers clustering the examples, grouping them into sets of similar cases, and then storing only the prototypes (centroids) of the identified clusters. More details in the chapter about clustering.

As we will see in the following part of this book the idea of considering distances between the current case and the cases stored in memory can be generalized. **Kernel methods and locally-weighted regression** can be seen as flexible and smooth generalizations of the nearest-neighbors idea; instead of applying a brute exclusion of the distant points, all points contribute to the output but with a significance ("weight") related to their distance from the query point.

Gist

KNN (K Nearest Neighbors) is a primitive and lazy form of machine learning: just store all training examples into memory.

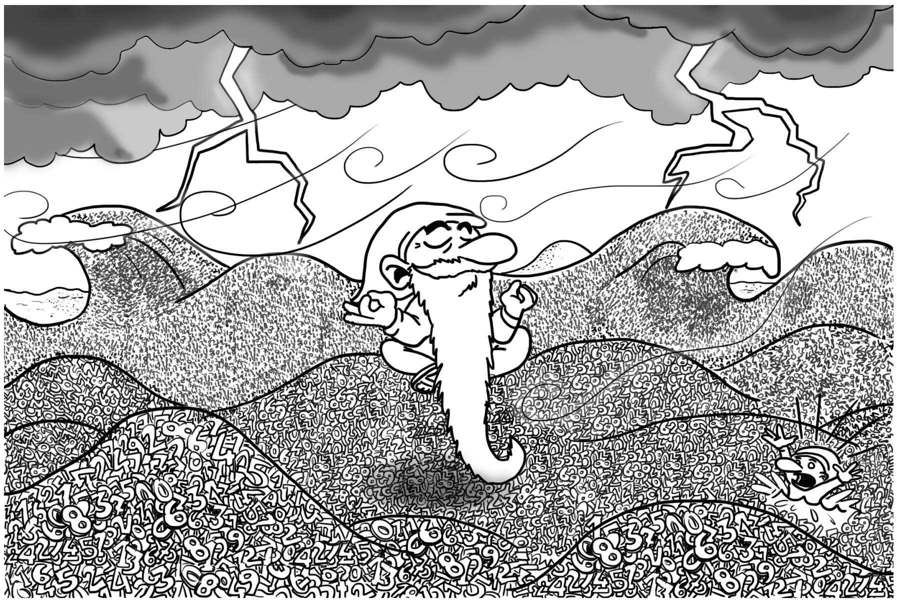
When a new input to be evaluated appears, search in memory for the K closest examples stored. Read their output and derive the output for the new input by majority or averaging. Laziness during training causes long response times when searching a very large memory storage.

KNN works in many real-world cases because similar inputs are usually related to similar outputs, a basic hypothesis in *machine learning*. It is similar to some "case-based" human reasoning processes. Although simple and brutal, it can be surprisingly effective in many cases.

Chapter 3

Learning requires a method

Data Mining, noun 1. Torturing the data until it confesses... and if you torture it long enough, you can get it to confess to anything.



Learning, both human and machine-based, is a powerful but subtle instrument. Real learning is associated with extracting the deep and basic relationships in a phenomenon, with summarizing with short models a wide range of events, with **unifying different cases by discovering the underlying explanatory laws**. We are mostly interested in **inductive inference**, a kind of reasoning that constructs general propositions derived from specific examples, in a bottom-up manner.

In other words, learning from examples is only a means to reach the real goal: **generalization, the capability of explaining new cases**, in the same area of application but not already encountered during the learning phase. On the contrary, learning by heart or

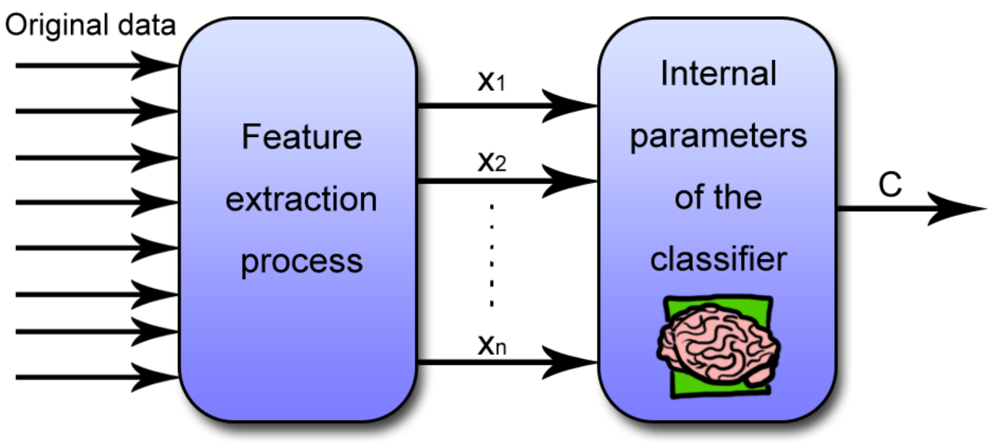

Original data
Feature extraction process
x1
x2
xn
Internal parameters of the classifier
C

Figure 3.1: Supervised learning architecture: feature extraction and classification.

memorizing are considered very poor forms of learning, useful for beginners but not for real experts. The KNN methods already encountered resemble this basic and lazy form of learning. If the goal is generalization, **estimating the performance has to be done with extreme care**, in order not to be misled by excessive optimism when observing the behavior of the model on the learning examples. After all, a student who is good at learning by heart will not always end up being the most successful individual in life!

Let's now define the machine learning context (ML for short) so that its full power can be switched on without injuries caused by improper usage of the tools and over-optimism.

Let's consider the input data. In many cases, before starting the machine learning process, it is useful to extract an informative subset of the original data by using the intuition and intelligence of the user. **Feature extraction** is the name of this preparatory phase (Fig. 3.1). Features (a.k.a. attributes) are the individual measurable properties of the phenomena being observed with useful information to derive the output value.

An input example can be the image of a letter of the alphabet, with the output corresponding to the letter symbol. Automated reading of zip codes for routing mail, or automated conversion of images of old book pages into the corresponding text are relevant applications, called optical character recognition. Intuition tells us that the absolute image brightness is not an informative feature (digits remain the same under more or less light). In this case, suitable features can be related to edges in the image, to histograms of gray values collected along rows and columns of an image, etc. More sophisticated techniques try to ensure that a translated or enlarged image is recognized as the original one, for example by extracting features measured with respect to the barycenter of the image (considering a gray value in a pixel as a mass value), or scaled with respect to the maximum extension of the black part of the image, etc. Extracting useful features often requires concentration, insight and knowledge about the problem, but doing so greatly simplifies the subsequent automated learning phase.

To fix the notation, a training set of ℓ *tuples* (ordered lists of elements) is considered, where each tuple is of the form $(\boldsymbol{x}_i, y_i)$, $i = 1, \dots, \ell$; $\boldsymbol{x}_i$ being a vector (array) of input parameter values in d dimensions ($\boldsymbol{x}_i \in \mathbb{R}^d$); y_i being the measured outcome to be learned by the algorithm. As mentioned before, we consider two possible problems: *regression problems*, where y_i is a real numeric value; and *classification problems*, where y_i belongs to a finite set.

In **classification** (recognition of the class of a specific object described by features $\boldsymbol{x}$), the output is a suitable code for the class. The output y belongs to a finite set, e.g., $y_i = \pm 1$ or $y_i \in \{1, \dots, N\}$. For example, think about classifying a mushroom as edible or poisonous.

In **regression**, the output is a real number from the beginning, and the objective is to model the relationship between a dependent variable (the output y) and one or more independent variables (the input features $\boldsymbol{x}$). For example, think about predicting the poison content of a mushroom from its characteristics.

In some applications, the classification is not always crisp and there are confused boundaries between the different classes. Think about classifying bald versus hairy people: no clear-cut distinction exists. In these cases there is a natural way to transform a classification into a regression problem. To take care of indecision, the output can be a real value ranging between zero and one, and it can be interpreted as the posterior **probability for a given class**, given the input values, or as a **fuzzy membership** value when probabilities cannot be used. For example, if a person has a few hair left, it makes little sense to talk about a probability of 0.2 of being hairy, in this case *fuzzy membership* value of 0.2 in the class of hairy persons can be more appropriate.

Having a continuous output value, for example from 0 to 1, gives additional flexibility for the practical usage of classification systems. Depending on a threshold, a human reader can be consulted in the more confused cases (for example the cases with output falling in the range from 0.4 to 0.6). The clear cases are handled automatically, the most difficult cases can be handled by a human person. This is the way in which optical character recognition is used in many contexts. Consider an image which may correspond to the digit 0 (zero) or the letter O (like in Old), it may be preferable to output 0.5 for each case instead of forcing a hard classification.

3.1 Learning from labeled examples: minimization and generalization

The task of a supervised learning method is to use the examples in order to build an association $y = \hat{f}(\boldsymbol{x})$ between input $\boldsymbol{x}$ and output y. The association is selected within a **flexible model** $\hat{f}(\boldsymbol{x}; \boldsymbol{w})$, where the flexibility is given by some **tunable parameters** (or **weights**) $\boldsymbol{w}$, and the best choice for the value $\boldsymbol{w}^*$ of the parameters depends on the examples analyzed. A scheme of the architecture is shown in Fig. 3.1, the two parts of feature extraction and identification of optimal internal weights of the classifier are distinguished. In many cases feature extraction requires more insight and intervention by human experts, while the **determination of the best parameters is fully automated**, this is why the method is called *machine* learning after all.

Think about a "multi-purpose box" waiting for input and producing the corresponding output depending on operations influenced by internal parameters. The information to be used for "customizing" the box has to be extracted from the given training examples. The basic idea is to fix the free parameters by **demanding that the learned model works correctly on the examples** in the *training set*.

Now, we are all believers in the power of optimization: we start by defining an *error measure* to be minimized[1], and we adopt an appropriate (automated) optimization process to determine optimal parameters. A suitable *error measure* is the sum of the errors between the correct answer (given by the example label) and the outcome predicted by the model (the output ejected by the multi-purpose box). The errors are considered as absolute values and often squared. The "**sum of squared errors**" is possibly the most widely used error measure in ML. If the error is zero, the model works 100% correctly on the given examples. The smaller the error, the better the average behavior on the examples.

Supervised learning therefore becomes **minimization of a specific error function**, depending on parameters $\boldsymbol{w}$. If you only care about the final result you may take the optimization part as a **"big red button"** to push on the multi-purpose box, to have it customized for your specific problem after feeding it with a set of labeled examples.

If you are interested in developing new LION tools, you will get more details about optimization techniques in the following chapters. The gist is the following: if the function is **smooth** (think about pleasant grassy California hills) one can discover points of low altitude (lakes?) by being blindfolded and parachuted to a random initial point, sampling neighboring points with his feet, and moving always in the direction of steepest descent. No "human vision" is available to the computer to "see" the lakes ("blindfolded"), only the possibility to sample one point at a time, and sampling in the neighborhood of the current point is often very effective.

[1]Minimization becomes maximization after multiplying the function to be optimized by minus one, this is why one often talks about "optimization", without specifying the direction min or max.

Let's reformulate the above mental image with a more mathematical language. If the function to be optimized is differentiable, a simple approach consists of using **gradient descent**. One iterates by calculating the gradient of the function w.r.t. the weights and by taking a small step in the direction of the negative gradient. This is in fact the popular technique in neural networks known as learning by **backpropagation** of the error [37, 38, 29].

Assuming a smooth function is not artificial: There is a basic **smoothness assumption** underlying supervised learning: if two input points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are close, the corresponding outputs y_1 and y_2 should also be close[2]. If the assumption is not true, it would be impossible to generalize from a finite set of examples to a set of possibly infinite new and unseen test cases. Let's note that the physical reality of signals and interactions in our brain tends to naturally satisfy the smoothness assumption. The activity of a neuron tends to depend smoothly on the neuron inputs, in turn caused by chemical and electrical interactions in their dendritic trees. It is not surprising that the term **neural networks** is used to denote some successful supervised learning techniques.

Now, minimization of an error function is a first critical component, but not the only one. If the **model complexity** (the flexibility, the number of tunable parameters) is too large, learning the examples with zero errors becomes trivial, but predicting outputs for new data may fail brutally. In the human metaphor, if learning becomes rigid memorization of the examples without grasping the underlying model, students have difficulties in generalizing to new cases. This is related to the *bias-variance* dilemma, and requires care in model selection, or minimization of a weighted combination of model error on the examples plus model complexity.

The **bias-variance dilemma** can be stated as follows.

- Models with too few parameters are inaccurate because of a large bias: they lack flexibility.
- Models with too many parameters are inaccurate because of a large variance: they are too sensitive to the sample details (changes in the details will produce huge variations).
- Identifying the best model requires identifying the proper "model complexity", i.e., the proper architecture and number of parameters.

The preference of simple models to avoid over-complicated models has also been given a fancy name: *Occam's razor*, referring to "shaving away" unnecessary complications in theories[3]. Optimization is still used, but the error measures become different, to take model complexity into account.

[2]Closeness between points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ can be measured by the Euclidean distance.

[3] Occam's razor is attributed to the 14th-century theologian and Franciscan friar Father William of Ockham who wrote "entities must not be multiplied beyond necessity" (*entia non sunt multiplicanda praeter necessitatem*). To quote Isaac Newton, "We are to admit no more causes of natural things than

It is also useful to distinguish between two families of methods for supervised classification. In one case one is interested in deriving a “constructive model” of how the output is generated from the input, in the other case one cares about the bottom line: obtaining a correct classification. The first case is more concerned with explaining the underlying mechanisms, the second with crude performance.

Among examples of the first class, **generative methods** try to model the process by which the measured data $\boldsymbol{x}$ are generated by the different classes y. Given a certain class, for example of a poisonous mushroom, what is the probability of obtaining real mushrooms of different geometrical forms? In mathematical terms, one learns a class-conditional density $p(\boldsymbol{x}|y)$, the probability of $\boldsymbol{x}$ given y. Then, given a fresh measurements, an assignment can be made by maximizing the posterior probability of a class given a measurement, obtained by Bayes’ theorem:

$$p(y|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|y)p(y)}{\sum_y p(\boldsymbol{x}|y)p(y)}; \tag{3.1}$$

where $p(\boldsymbol{x}|y)$ is known as the likelihood of the data, and $p(y)$ is the prior probability, which reflects the probability of the outcome before any measure is performed. The term at the denominator is the usual normalization term to make sure that probabilities sum up to one (a mnemonic way to remember Bayes’ rule is: “posterior = prior times likelihood”).

Discriminative algorithms do not attempt at modeling the data generation process, they just aim at directly estimating $p(y|\boldsymbol{x})$, a problem which is in some cases simpler than the two-steps process (first model $p(\boldsymbol{x}|y)$ and then derive $p(y|\boldsymbol{x})$) implied by generative methods. Multilayer perceptron neural networks, as well as Support Vector Machines (SVM) are examples of discriminative methods described in the following chapters.

The difference is profound, it tells practitioners that **accurate classifiers can be built without knowing or building a detailed model** of the process by which a certain class generates input examples. You do not need to be an expert mycologist in order to pick mushroom without risking death, you just need an abundant and representative set of example mushrooms, with correct classifications.

3.2 Learn, validate, test!

When learning from labeled examples we need to follow **careful experimental procedures** to measure the effectiveness of the learning process. In particular, it is a terrible mistake to measure the performance of the learning systems on the same examples used

such as are both true and sufficient to explain their appearances. Therefore, to the same natural effects we must, so far as possible, assign the same causes.”

for training. The objective of machine learning is to obtain a system capable of **generalizing** to new and previously unseen data. Otherwise the system is not learning, it is merely memorizing a set of known patterns, this must be why questions at university exams are changing from time to time…

Let's assume we have a supervisor (a software program) who can generate labeled examples from a given probability distribution. Ideally we should ask the supervisor for some examples during training, and then test the performance by asking for some fresh examples. Ideally, the number of examples used for training should be sufficiently large to permit convergence, and the number used for testing should be very large to ensure a statistically sound estimation. We strongly suggest you not to conclude that a machine learning system to identify edible from poisonous mushrooms is working, after testing it on three mushrooms.

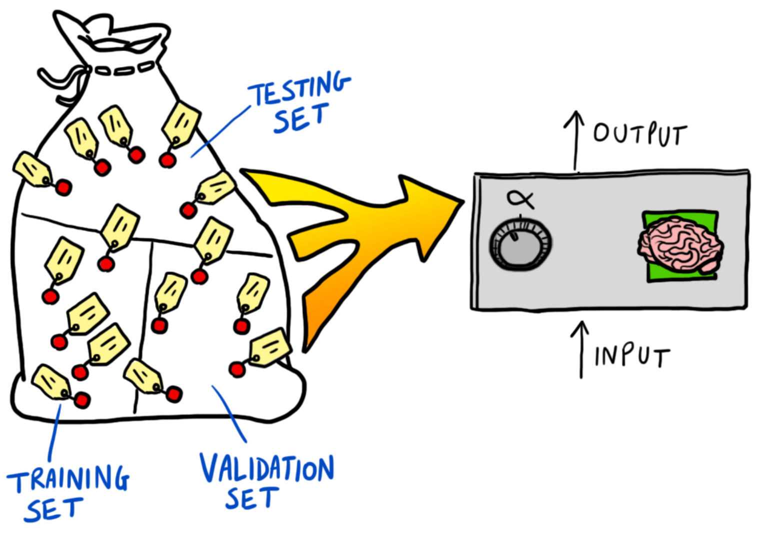


Figure 3.2: Labeled examples need to be split into training, validation and test sets.

This ideal situation may be far from reality. In some cases the set of examples is rather small, and has to be used in the best possible way *both* for training *and* for measuring performance. In this case the set has to be clearly partitioned between a **training set** and a **validation set**, the first used to train, the second to measure performance, as illustrated in Fig. 3.2. A typical performance measure is the **root mean square (abbreviated RMS)** error between the output of the system and the correct output given by the supervisor[4].

In general, the learning process optimizes the model parameters to make the model *reproduce* the training data as well as possible. If we then take an independent sample of validation data from the same population as the training data, it will generally turn out that the error on the validation data will be larger than the error on the training data. This discrepancy is likely to become severe if training is excessive, leading to **over-fitting** (**overtraining**), particularly likely to happen when the number of training examples is small, or when the number of parameters in the model is large.

If the examples are limited, we now face a problem: do we want to use more of them for training and risk a poor and noisy measurement of the performance, or to have a more robust measure but losing training examples? Think in concrete terms: if you have 50 mushroom examples, do you use 45 for training and 5 for testing, 30 for training and 20 for testing? ...Luckily, there is a way to jump over this embarrassing situation, just use **cross-validation**. Cross-validation is a generally applicable way to predict the performance of a model on a validation set by using repeated experiments instead of mathematical analysis.

The idea is to **repeat** many train-and-test experiments, by using different partitions of the original set of examples into two sets, one for training one for testing, and then averaging the test results. This general idea can be implemented as K**-fold cross-validation**: the original sample is randomly partitioned into K subsamples. A single subsample is used as the validation data for testing, and the remaining $K - 1$ subsamples are used as training data. The cross-validation process is then repeated K times (the folds), with each of the K subsamples used exactly once as the validation data. The results from the folds then can be collected to produce a single estimation. The advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once. If the example set is really small one can use the extreme case of **leave-one-out cross-validation**, using a single observation from the original sample as the validation data, and the remaining observations as the training data (just put K equal to the number of examples).

Stratified cross-validation is an additional improvement to avoid different class balances in the training and testing set. We do not want that, by chance, a class is more present in the training examples and therefore less present in the testing examples (with respect to its average presence in all examples). When stratification is used, the extraction of the ℓ/K testing patterns is done separately for examples of each class, to ensure a fair balance among the different classes (Fig. 3.3).

[4] The RMS value of a set of values is the *square root* of the arithmetic *mean* (average) of the *squares* of the original values. In our case of a set of errors e_i , the RMS value is given by this formula:

$$RMS = \sqrt{\frac{e_1^2 + e_2^2 + \cdots + e_\ell^2}{\ell}}$$

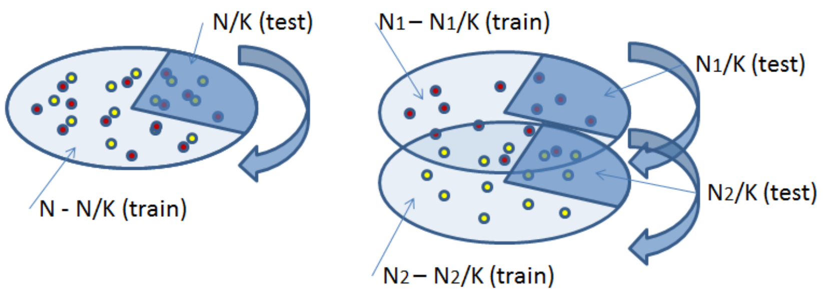


Figure 3.3: Stratified cross-validation, examples of two classes. In normal cross-validation $1/K$ of the examples are kept for testing, the slice is then "rotated" K times. In stratification, a separate slice is kept for each class to maintain the relative proportion of cases of the two classes.

If the machine learning method itself has some **parameters to be tuned**, this creates an additional problem. Let's call them **meta-parameters** in order to distinguish them from the basic parameters of the model, of the "multi-purpose box" to be customized. Think for example about deciding the termination criterion for an iterative minimization technique (when do we stop training), or the number of hidden neurons in a multilayer perceptron, or appropriate values for the crucial parameters of the Support Vector Machine (SVM) method. Finding optimal values for the meta-parameters implies **reusing** the validation examples many times. Reusing validation examples means that they also *become part of the training process*. We are in fact dealing with a kind of meta-learning, learning the best way to learn. The more you reuse the validation set, the more the danger that the measured performance will be optimistic, not corresponding to the real performance on new data. One is in practice "torturing the data until it confesses... and if you torture it long enough, you can get it to confess to anything."

In the previous context of a limited set of examples to be used for all needs, to proceed in a sound manner one has to split the data into **three sets: a training, a validation and a (final) testing one**. The test set is *used only once for a final measure of performance*.

Finally let's note that, to increase the confusion, in the standard case of a single train-validation cycle, the terms validation and testing are often used as synonyms.

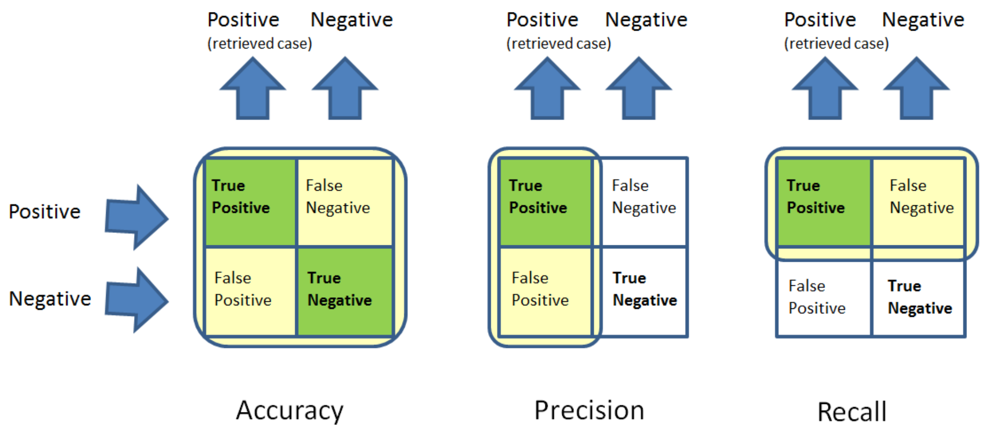


Figure 3.4: Each line in the matrix reports the different classifications for cases of a class. You can visualize cases entering at the left and being sorted out by the different columns to exit at the top. Accuracy is defined as the fraction of correct answers over the total, precision as the fraction of correct answers over the number of retrieved cases and recall is computed as the fraction of correct answers over the number of relevant cases. In the plot: divide the cases in the green part by the cases in the yellow area.

3.3 Errors of different kinds

When measuring the performance of a model, mistakes are not born to be equal. If you classify a poisonous mushroom as edible you are going to die, if you classify an edible mushroom as poisonous you are wasting a little time. Depending on the problem, the criterion for deciding the "best" classification changes. Let's consider a binary classification (with "yes" or "no" output). Some possible criteria are: **accuracy, precision, and recall**. The definitions are simple but require some concentration to avoid easy confusions (Fig. 3.4).

The **accuracy** is the proportion of true results given by the classifier (both true positives and true negatives). The other measures focus on the cases which are labeled as belonging to the class (the "positives"). The **precision** for a class is the number of true positives (i.e. the number of items correctly labeled as belonging to the positive class) divided by the total number of elements labeled as belonging to the positive class (i.e. the sum of true positives and false positives, which are incorrectly labeled as belonging to the class). The **recall** is the number of true positives divided by the total number of elements that actually belong to the positive class (i.e. the sum of true positives and false negatives, which are not labeled as belonging to the positive class but should have been). Precision answers the question: "How many of the cases labeled as positive are correct?" Recall answers the question: "How many of the truly positive cases are retrieved as positive?” Now, if you are picking mushrooms, are you more interested in high precision or high recall?

A **confusion matrix** explains how cases of the different classes are correctly classified or confused as members of wrong classes (Fig. 3.5).

Each row tells the story for a single class: the total number of cases considered and how the various cases are recognized as belonging to the correct class (cell on the diagonal) or confused as members of the other classes (cells corresponding to different columns).

Gist

The goal of machine learning is to use a set of training examples to realize a system which will correctly generalize to new cases, in the same context but not seen during learning.

ML learns, i.e., determines appropriate values for the free parameters of a flexible model, by automatically minimizing a measure of the error on the example set, possibly corrected to discourage complex models, and therefore improve the chances of correct generalization.

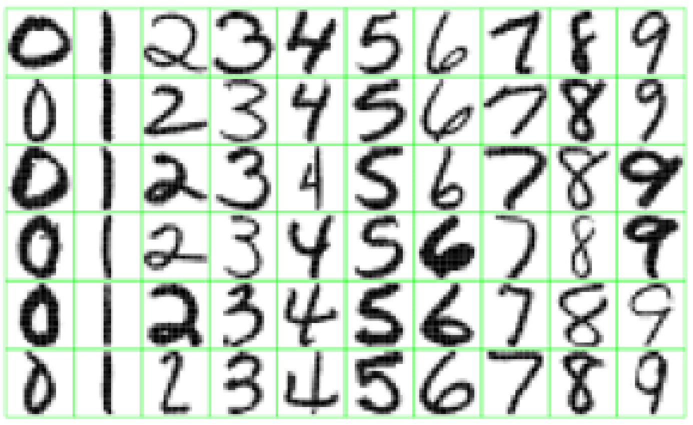
The output value of the system can be a class (classification), or a number (regression). In some cases having as output the probability for a class increases flexibility of usage.

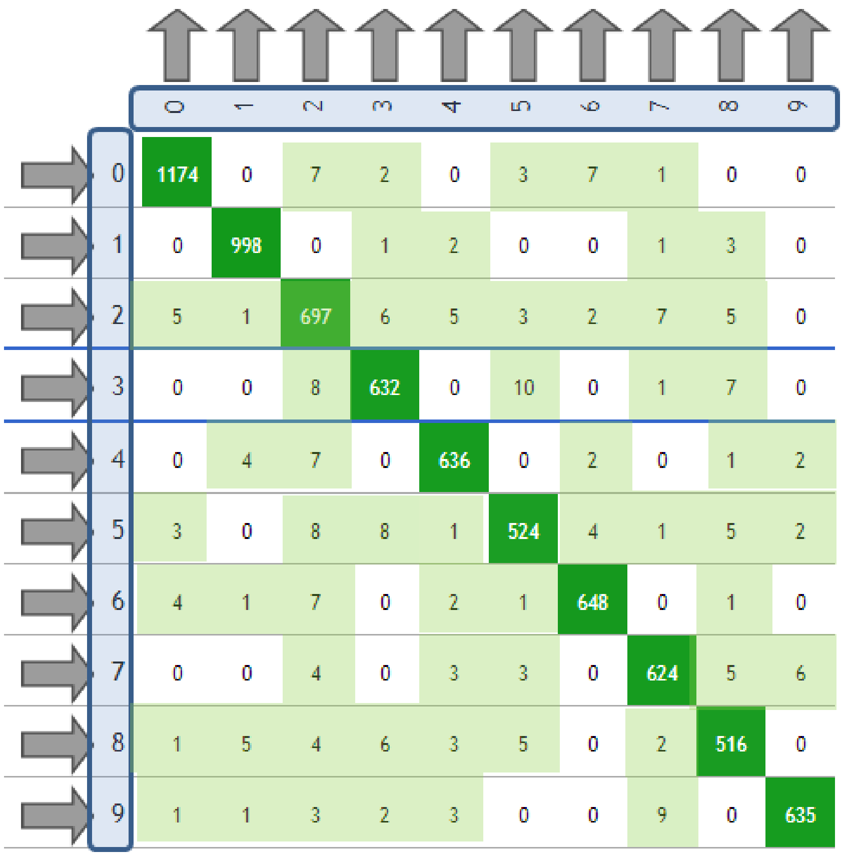
Accurate classifiers can be built without any knowledge elicitation phase, just starting from an abundant and representative set of example data. This is a dramatic paradigm change which enormously facilitates the adoption of ML in business contexts.

ML is very powerful but requires a strict method (a kind of “pedagogy” of ML). For sure, never estimate performance on the training set — this is a mortal sin: be aware that re-using validation data will create optimistic estimates. If examples are scarce, use cross-validation to show off that you are an expert ML user.

To be on the safe side and enter the ML paradise, set away some test examples and use them only once at the end to estimate performance.

The is no single way to measure the performance of a model, different kinds of mistakes can have very different costs. Accuracy, precision and recall are some possibilities for binary classification, a confusion matrix is giving the complete picture for more classes.





	0	1	2	3	4	5	6	7	8	9
0	1174	0	7	2	0	3	7	1	0	0
1	0	998	0	1	2	0	0	1	3	0
2	5	1	697	6	5	3	2	7	5	0
3	0	0	8	632	0	10	0	1	7	0
4	0	4	7	0	636	0	2	0	1	2
5	3	0	8	8	1	524	4	1	5	2
6	4	1	7	0	2	1	648	0	1	0
7	0	0	4	0	3	3	0	624	5	6
8	1	5	4	6	3	5	0	2	516	0
9	1	1	3	2	3	0	0	9	0	635

Figure 3.5: Confusion matrix for optical character recognition (handwritten ZIP code digits). The various confusions make sense. For example the digit "3" is recognized correctly in 632 cases, confused with "2" in 8 cases, with "5" in 10 cases, with "8" in 7 cases. "3" is never confused as "1" or "4", digits with a very different shape.

Part I

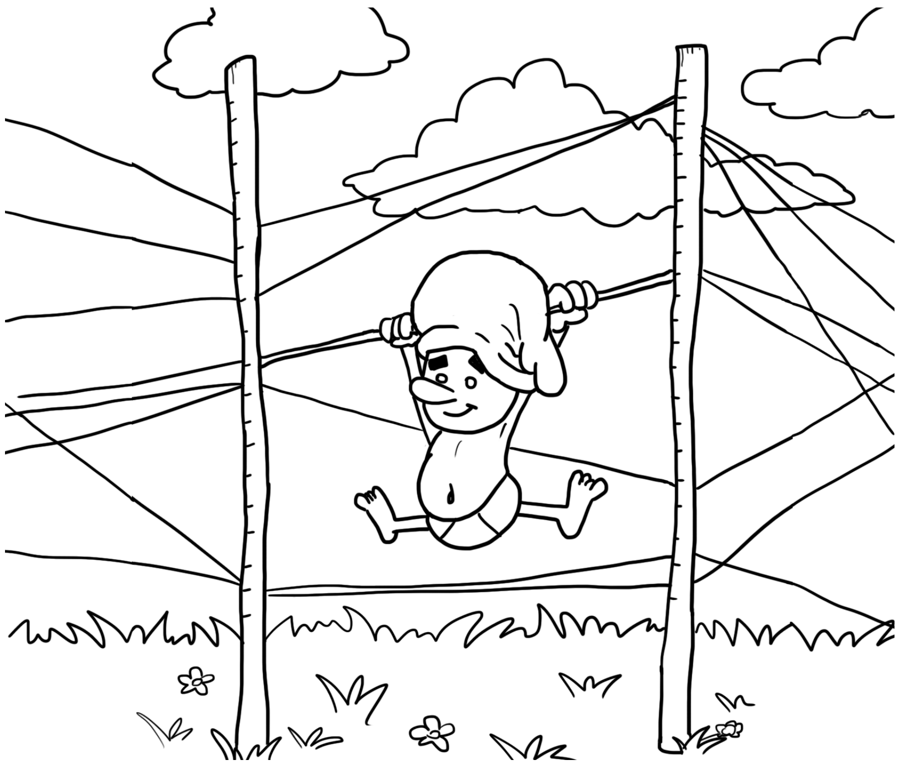
Supervised learning

Latest chapter revision: November 14, 2013

Chapter 4

Linear models

Most right-handed people are linear thinking, think inside the box.
(our readers are free to disbelieve our openings)



Just below the mighty power of optimization lies the awesome power of linear algebra. Do you remember your teacher at school: "Study linear algebra, you will need it in life"? Well, with more years on your shoulders you know he was right. Linear algebra is a "math survival kit," when confronted with a difficult problem, try with linear equa-

Latest chapter revision: November 14, 2013

tions first. In many cases you will either solve it or at least come up with a workable approximation. Not surprisingly, this is true also for models to explain data.

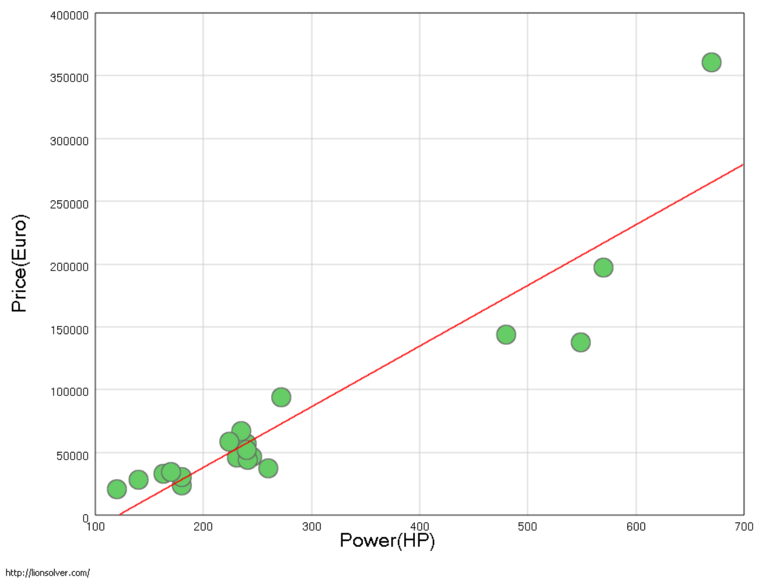


Figure 4.1: Data about price and power of different car models. A linear model (in red) is shown.

Fig. 4.1 plots the price of different car models as a function of their horse power. As you can expect, the bigger the power the bigger the price, car dealers are honest, and there is an approximate linear relationship between the two quantities. If we summarize the data with the linear model in red we lose some details but most of the trend is preserved. We are *fitting* the data with a line.

Of course, defining what we mean by "best fit" will immediately lead us to *optimizing* the corresponding goodness function.

4.1 Linear regression

A linear dependence of the output from the input features is a widely used model. The model is simple, it can be easily trained, and the computed *weights* in the linear summation provide a direct explanation of the *importance* of the various attributes: the bigger the absolute value of the weight, the bigger the effect of the corresponding attribute. Therefore, do not complicate your life and consider nonlinear models unless you are strongly motivated by your application.

Math people do not want to waste trees and paper[1], arrays of numbers (*vectors*) are denoted with a single variable, like $\boldsymbol{w}$. The vector $\boldsymbol{w}$ contains its components $(w_1, w_2, ..., w_d)$, d being the number of input attributes or *dimension*. Vectors "stand up" like columns, to make them lie down you can transpose them, getting $\boldsymbol{w}^T$. The scalar product between vector $\boldsymbol{w}$ and vector $\boldsymbol{x}$ is therefore $\boldsymbol{w}^T \cdot \boldsymbol{x}$, with the usual matrix-multiplication definition, equal to $w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$.

The hypothesis of a linear dependence of the outcomes on the input parameters can be expressed as

$$y_i = \boldsymbol{w}^T \cdot \boldsymbol{x}_i + \epsilon_i,$$

where $\boldsymbol{w} = (w_1, \ldots, w_d)$ is the vector of *weights* to be determined and ϵ_i is the error[2]. Even if a linear model correctly explains the phenomenon, errors arise during measurements: **every physical quantity can be measured only with a finite precision**. Approximations are not because of negligence but because measurements are imperfect.

One is now looking for the weight vector $\boldsymbol{w}$ so that the linear function

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{w}^T \cdot \boldsymbol{x} \tag{4.1}$$

approximates as closely as possible our experimental data. This goal can be achieved by finding the vector $\boldsymbol{w}^*$ that minimizes the sum of the squared errors (**least squares** approximation):

$$\text{ModelError}(\boldsymbol{w}) = \sum_{i=1}^{\ell} (\boldsymbol{w}^T \cdot \boldsymbol{x}_i - y_i)^2. \tag{4.2}$$

In the unrealistic case of zero measurement errors and a perfect linear model, one is left with a set of **linear equations** $\boldsymbol{w}^T \cdot \boldsymbol{x}_i = y_i$, one for each measurement, which can be solved by standard linear algebra if the system of linear equations is properly defined (d non-redundant equations in d unknowns). In all real-world cases measurement errors are present, and the number of measurements $(\boldsymbol{x}_i, y_i)$ can be much larger than the input dimension. Therefore one needs to search for an approximated solution, for weights $\boldsymbol{w}$ obtaining the lowest possible value of the above equation (4.2), typically larger than zero.

You do not need to know *how* the equation is minimized to use a linear model, true believers of optimization can safely trust its magic problem-solving hand for linear models. But if you are curious, masochistic, or dealing with very large and problematic cases you may consider reading Sections 4.6 and 4.7.

[1]We cannot go very deep into linear algebra in our book: we will give the basic definitions and motivations, it will be very easy for you to find more extended presentations in dedicated books or websites.

[2] In many cases the error ϵ_i is assumed to have a Gaussian distribution.

4.2 A trick for nonlinear dependencies

Your appetite as a true believer in the awesome power of linear algebra is now huge but unfortunately not every case can be solved with a linear model. In most cases, a function in the form $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$ is too restrictive to be useful. In particular, it assumes that $f(0) = 0$. It is possible to change from a *linear* to an *affine* model by inserting a constant term w_0, obtaining: $f(\boldsymbol{x}) = w_0 + \boldsymbol{w}^T \cdot \boldsymbol{x}$. The constant term can be incorporated into the dot product, by defining $\boldsymbol{x} = (1, x_1, \ldots, x_d)$, so that equation (4.1) remains valid.

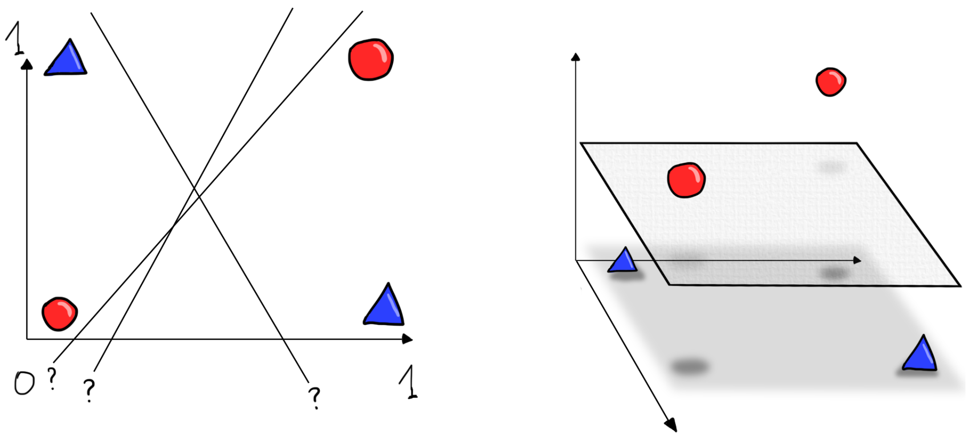


Figure 4.2: A case where a linear separation is impossible (XOR function, left). A linear separability with a hyperplane can be obtained by mapping the point in a nonlinear way to a higher-dimensional space.

The insertion of a constant term is a special case of a more general technique to model nonlinear dependencies while remaining in the easier context of linear least squares approximations. This apparent contradiction is solved by a trick: the model remains linear and it is applied to *nonlinear features* calculated from the raw input data instead of the original input $\boldsymbol{x}$, as shown in Fig. 4.2. It is possible to define a set of functions:

$$\phi_1, \ldots, \phi_n : \mathbb{R}^d \to \mathbb{R}^n$$

that map the input space into some more complex space, in order to apply the linear regression to the vector $\boldsymbol{\phi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), \ldots, \phi_n(\boldsymbol{x}))$ rather than to $\boldsymbol{x}$ directly.

For example if $d = 2$ and $\boldsymbol{x} = (x_1, x_2)$ is an input vector, a quadratic dependence of the outcome can be obtained by defining the following *basis functions*:

$$\phi_1(\boldsymbol{x}) = 1, \quad \phi_2(\boldsymbol{x}) = x_1, \quad \phi_3(\boldsymbol{x}) = x_2,$$

$$\phi_4(\boldsymbol{x}) = x_1x_2, \quad \phi_5(\boldsymbol{x}) = x_1^2, \quad \phi_6(\boldsymbol{x}) = x_2^2.$$

Note that $\phi_1(\boldsymbol{x})$ is defined in order to allow for a constant term in the dependence. The linear regression technique described above is then applied to the 6-dimensional vectors obtained by applying the basis functions, and not to the original 2-dimensional parameter vector.

More precisely, we look for a dependence given by a scalar product between a vector of weights $\boldsymbol{w}$ and a vector of features $\boldsymbol{\phi}(\boldsymbol{x})$, as follows:

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{w}^T \cdot \boldsymbol{\phi}(\boldsymbol{x}).$$

The output is a weighted sum of the features.

4.3 Linear models for classification

Section 4.1 considers a linear function that nearly matches the observed data, for example by minimizing the sum of squared errors. Some tasks, however, allow for a small set of possible outcomes. One is faced with a *classification* problem.

Let the outcome variable be two-valued (e.g., ± 1). In this case, linear functions can be used as discriminants, the idea is to have an hyperplane defined by a vector $\boldsymbol{w}$ separating the two classes. A plane generalizes a line, and a hyperplane generalizes a plane when the number of dimensions is more than three.

The goal of the training procedure is to find the best hyperplane so that the examples of one class are on a side of the hyperplane, examples of the other class are on the other side. Mathematically one finds the best coefficient vector $\boldsymbol{w}$ so that the decision procedure:

$$y = \begin{cases} +1 & \text{if } \boldsymbol{w}^T \cdot \boldsymbol{x} \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.3}$$

performs the best classification. The method for determining the **best separating linear function** (geometrically identifiable with a hyperplane) depend on the chosen classification criteria and error measures.

For what we know from this chapter about regression, we can ask that points of the first class are mapped to $+1$, and points of the second classed are mapped to -1. This is a stronger requirement than separability but permits us to use a technique for regression, like gradient descent or pseudo-inverse.

If the examples are not separable with a hyperplane, one can either live with some error rate, or try the trick suggested before and calculate some *nonlinear features* from the raw input data to see if the transformed inputs are now separable. An example is shown in Fig. 4.2, the two inputs have 0-1 coordinates, the output is the *exclusive OR* function (XOR function) of the two inputs (one or the other, but not both equal to 1).

The two classes (with output 1 or 0) cannot be separated by a line (a hyperplane of dimension one) in the original two-dimensional input space. But they can be separated by a plane in a three-dimensional transformed input space.

4.4 How does the brain work?

Our brains are a mess, at least those of the authors. For sure the system at work when summing two large numbers is very different from the system active while playing “shoot’em up” action games. The system at work when calculating or reasoning in logical terms is different from the system at work when recognizing the face of your mother. The first system is iterative, it works by a sequence of steps, it requires conscious effort and attention. The second system works in parallel, is very fast, often effortless, sub-symbolic (not using symbols and logic).

Different mechanisms in machine learning resemble the two systems. Linear discrimination, with iterative *gradient-descent* learning techniques for gradual improvements, resembles more our sub-symbolic system, while classification trees based on a sequence of “if-then-else” rules (we will encounter them in the following chapters) resemble more our logical part.

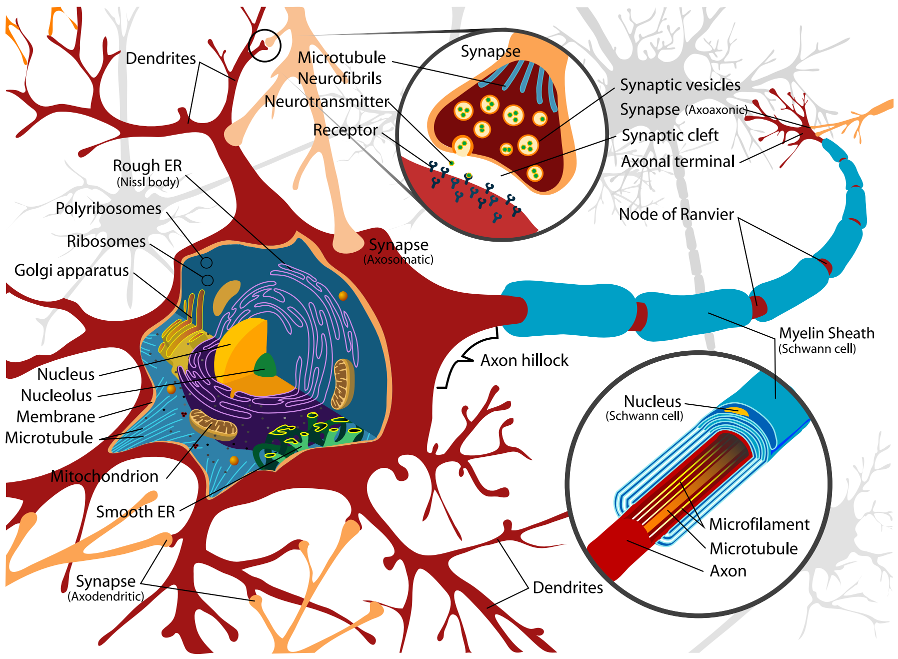


Figure 4.3: Neurons and synapses in the human brain (derived from Wikipedia commons).

Linear functions for classification have been known under many names, the historic one being *perceptron*, a name that stresses the analogy with biological neurons. Neurons (see Fig. 4.3) communicate via chemical synapses. Synapses [3] are essential to neuronal

[3]The term *synapse* has been coined from the Greek “syn-” (“together”) and “haptein” (“to clasp”).

function: neurons are cells that are specialized to pass signals to individual target cells, and synapses are the means by which they do so.

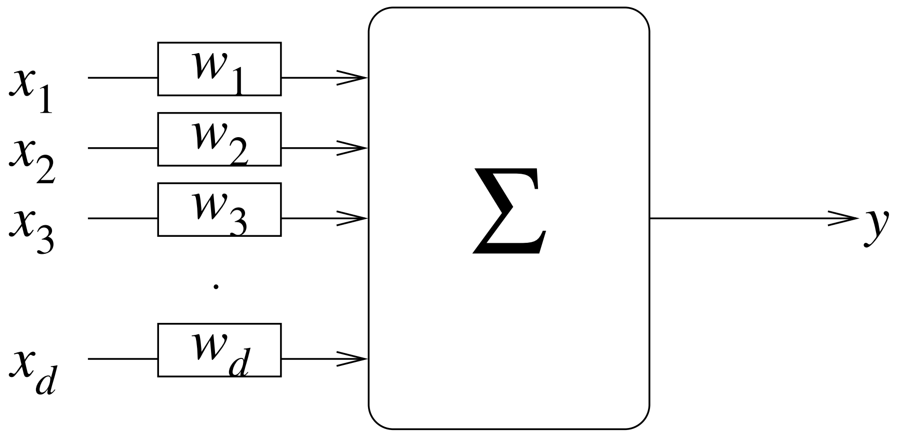


Figure 4.4: The Perceptron: the output is obtained by a weighted sum of the inputs passed through a final threshold function.

The fundamental process that triggers synaptic transmission is a propagating electrical signal that is generated by exploiting the electrically excitable membrane of the neuron. This signal is generated (the neuron output *fires*) if and only if the result of incoming signals combined with excitatory and inhibitory synapses and integrated surpasses a given threshold. Fig. 4.4 therefore can be seen as the abstract and functional representation of a single nerve cell. Variations of gradient-descent for improving classification performance can be related to biological learning systems.

4.5 Why are linear models popular and successful?

The deep reason why linear models are so popular is the **smoothness** underlying many if not most of the physical phenomena ("Nature does not make jumps"). An example is in Fig. 4.5, the average stature of kids grows gradually, without jumps, to slowly reach a saturating stature after adolescence.

Now, if you remember calculus, every smooth (differentiable) function can be approximated around an operating point x_c with its **Taylor series approximation**. The second term of the series is linear, given by a scalar product between the gradient $\nabla f(x_c)$ and the displacement vector, the additional terms go to zero in a quadratic manner:

$$f(x) = f(x_c) + \nabla f(x_c) \cdot (x - x_c) + O(\|x - x_c\|^2). \tag{4.4}$$

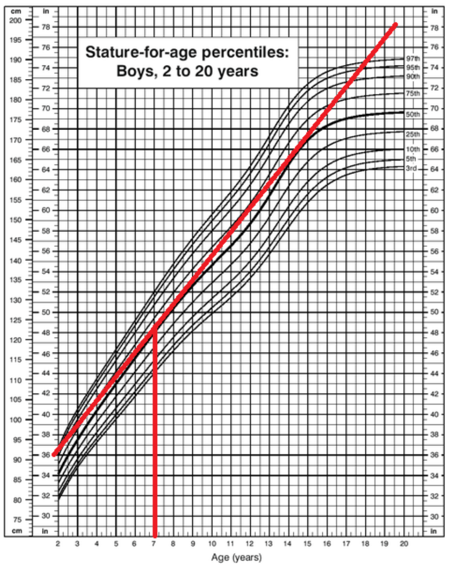


Figure 4.5: Functions describing physical phenomena tend to be *smooth*. The stature-for-age curve can be approximated well by a tangent line (red) from 2 to about 15 years.

Therefore, if the operating point of the smooth systems is close to a specific point x_c, a linear approximation is a reasonable place to start.

In general, the local model will work reasonably well only in the neighbourhood of a given operating point. A linear model for stature growth of children obtained by a tangent at the 7 years stature point will stop working at about 15 years, luckily for the size of our houses.

4.6 Minimizing the sum of squared errors

Linear models are identified by minimizing the sum of squared errors of equation (4.2). If you are not satisfied with a ”proof in the pudding” approach but want to go deeper into the matter, read on.

As mentioned, in the unrealistic case of zero measurement errors and of a perfect linear model, one is left with a set of **linear equations** $\boldsymbol{w}^T \cdot \boldsymbol{x}_i = y_i$, one for each measurement. If the system of linear equations is properly defined (d non-redundant equations in d unknowns) one can solve them by *inverting the matrix* containing the coefficients.

In practice, in real-world cases, reaching zero for the ModelError is impossible, errors are present and the number of data points can be be much larger than the number of parameters d. Furthermore, let’s remember that the goal of learning is generalization, we are interested in reducing the *future* prediction errors. We do not need to stress ourselves too much with reducing the error on reproducing the training examples with zero error, which can actually be counterproductive.

We need to **generalize the solution of a system of linear equations by allowing for errors**, and to generalize matrix inversion. We are lucky that equation (4.2) is quadratic, minimizing it leads again to a system of linear equations[4].

If you are familiar with analysis, finding the minimum is straightforward: calculate the gradient and demand that it is equal to zero. If you are not familiar, think that the bottom of the valleys (the points of minimum) are characterized by the fact that small movements keep you at the same altitude.

The following equations determine the optimal value for w:

$$\boldsymbol{w}^* = (X^T X)^{-1} X^T \boldsymbol{y}; \tag{4.5}$$

where $\boldsymbol{y} = (y_1, \ldots, y_\ell)$ and X is the matrix whose rows are the $\boldsymbol{x}_i$ vectors.

The matrix $(X^T X)^{-1} X^T$ is the **pseudo-inverse** and it is a natural generalization of a matrix inverse to the case in which the matrix is non-square. If the matrix is invertible and the problem can be solved with zero error, the pseudo-inverse is equal to the inverse, but in general, e.g., if the number of examples is larger than the number of weights, aiming at a *least-square* solution avoids the embarrassment of not having an exact solution and provides a statistically sound "compromise" solution. In the real world, exact models are not compatible with the noisy characteristics of nature and of physical measurements and it is not surprising that *least-square* and *pseudo-inverse* beasts are among the most popular tools.

The solution in equation (4.5) is "one shot:" calculate the pseudo-inverse from the experimental data and multiply to get the optimal weights. In some cases, if the number of examples is huge, **an iterative technique based on gradient descent** can be preferable: start from initial weights and keep moving them by executing small steps along the direction of the negative gradient, until the gradient becomes zero and the iterations reach a stable point. By the way, as you may anticipate, real neural systems like our brain do not work in a "one shot" manner with linear algebra but more in an iterative manner by gradually modifying weights, as we will see later. Maybe this is why linear algebra is not so popular at school?

Let us note that the minimization of squared errors has a physical analogy to the spring model presented in Fig. 4.6. Imagine that every sample point is connected by a vertical spring to a rigid bar, the physical realization of the best fit line. All springs have equal elastic constants and zero extension at rest. In this case, the potential energy of each spring is proportional to the square of its length, so that equation (4.2) describes the overall potential energy of the system up to a multiplicative constant. If one starts

[4]Actually, you may suspect that the success of the quadratic model is related precisely to the fact that, after calculating derivatives, one is left with a linear expression.

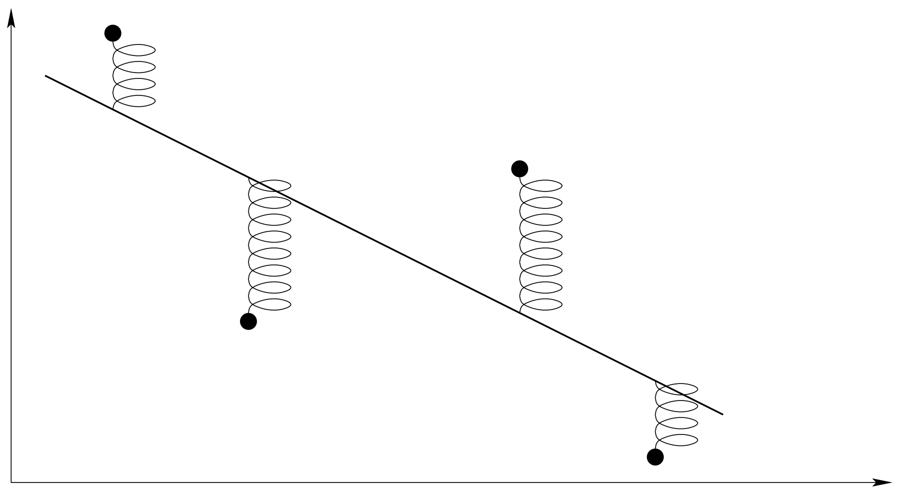


Figure 4.6: Physics gives an intuitive spring analogy for least squares fits. The best fit is the line that minimizes the overall potential energy of the system (proportional to the sum of the squares of the spring length).

and lets the physical system oscillate until equilibrium is reached, with some friction to damp the oscillations, the final position of the rigid bar can be read out to obtain the least square fit parameters; an analog computer for line fitting!

For sure you will forget the pseudo-inverse but you will never forget this physical system of damped oscillating springs attaching the best-fit line to the experimental data.

If features are transformed by some ϕ function (as a trick to consider nonlinear relationships), the solution is very similar. Let $\boldsymbol{x}'_i = \boldsymbol{\phi}(\boldsymbol{x}_i)$, $i = 1, \dots, \ell$, be the transformations of the training input tuples $\boldsymbol{x}_i$. If X' is the matrix whose rows are the $\boldsymbol{x}'_i$ vectors, then the optimal weights with respect to the least squares approximation are computed as:

$$\boldsymbol{w}^* = (X'^T X')^{-1} X'^T y. \tag{4.6}$$

4.7 Real numbers in computers are fake: numerical instabilities

Real numbers (like π and "most" numbers) cannot be represented in a digital computer. Each number is assigned a *fixed and limited* number of bits, no way to represent an infinite number of digits like in 3.14159265... Therefore real numbers represented in a computer are "fake", they can and most often will have mistakes. Mistakes will propagate during mathematical operations, in certain cases the results of a sequence of operations can be very different from the mathematical results. Get a matrix, find its inverse and multiply the two. You are assumed to get the identity but you end up with something different (maybe you should check which precision your bank is using).

When the number of examples is large, equation (4.6) is the solution of a linear system in the over-determined case (more linear equations than variables). In particular, matrix $X^T X$ must be non-singular, and this can only happen if the training set points $x_1, \dots, x_\ell$ do not lie in a proper subspace of $\mathbb{R}^d$, i.e., they are not "aligned." In many cases, even though $X^T X$ *is* invertible, the distribution of the training points is not generic enough to make it *stable*. **Stability** here means that small perturbations of the sample points lead to small changes in the results. An example is given in Fig. 4.7, where a bad choice of sample points (in the right plot, x_1 and x_2 are not independent) makes the system much more dependent on noise, or even to rounding errors.

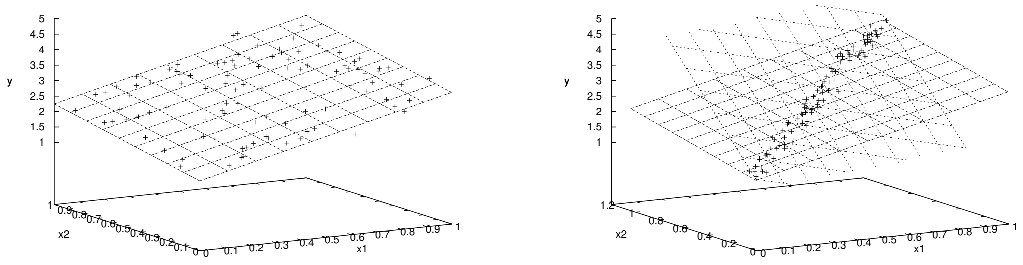


Figure 4.7: A well-spread training set (left) provides a stable numerical model, whereas a bad choice of sample points (right) may result in wildly changing planes, including very steep ones (adapted from [4]).

If there is no way to modify the choice of the training points, the standard mathematical tool to ensure numerical stability when sample points cannot be distributed at will is known as **ridge regression**. It consists of the addition of a **regularization** term to the (least squares) error function to be minimized:

$$\text{error}(\boldsymbol{w}; \lambda) = \sum_{i=1}^{\ell} (\boldsymbol{w}^T \cdot \boldsymbol{x}_i - y_i)^2 + \lambda \boldsymbol{w}^T \cdot \boldsymbol{w}. \qquad (4.7)$$

The minimization with respect to $\boldsymbol{w}$ leads to the following:

$$\boldsymbol{w}^* = (\lambda I + X^T X)^{-1} X^T \boldsymbol{y}.$$

The insertion of a small diagonal term makes the inversion more robust. Moreover, one is actually demanding that the solution takes the size of the weight vector into account, to avoid steep interpolating planes such as the one in the right plot of Fig. 4.7.

If you are interested, the theory justifying the approach is based on *Tichonov regularization*, which is the most commonly used method for curing *ill-posed* problems. A problem is ill-posed if no unique solution exists because there is not enough information specified in the problem, for example because the number of examples is limited. It is necessary to supply extra information or smoothness assumptions. By jointly minimizing the empirical error and penalty, one seeks a model that not only fits well and is also simple to avoid large variation which occurs in estimating complex models.

You do not need to know the theory to use machine learning but you need to be aware of the problem, this will raise your debugging capability if complex operations do not lead to the expected result. Avoiding very large or very small numbers is a pragmatic way to cure most problems, for example by scaling your input data before starting with machine learning.

Gist

Traditional linear models for regression (linear approximation of a set of input-output pairs) identify the best possible linear fit of experimental data by minimizing a sum the squared errors between the values predicted by the linear model and the training examples. Minimization can be "one shot" by generalizing matrix inversion in linear algebra, or iteratively, by gradually modifying the model parameters to lower the error. The pseudo-inverse method is possibly the most used technique for fitting experimental data.

In classification, linear models aim at separating examples with lines, planes and hyper-planes. To identify a separating plane one can require a mapping of the inputs to two distinct output values (like $+1$ and -1) and use regression. More advanced techniques to find robust separating hyper-planes when considering generalization will be the Support Vector Machines described in the future chapters.

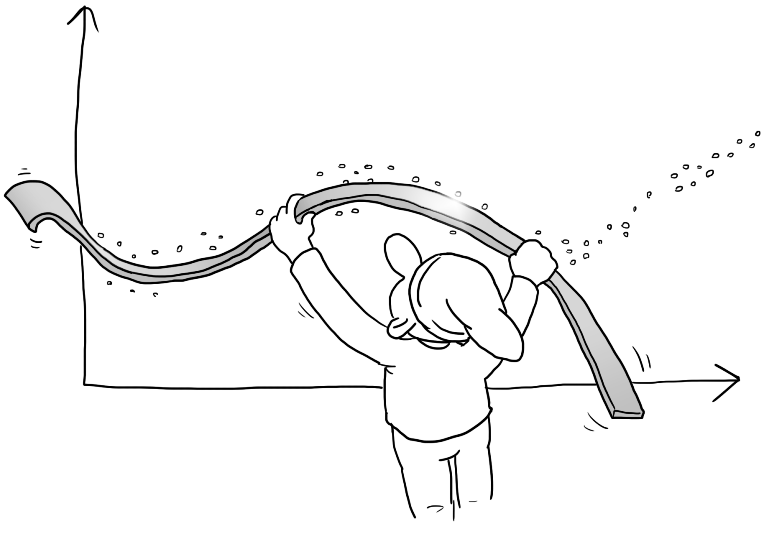
Real numbers in a computer do not exist and their approximation by limited-size binary numbers is a possible cause of mistakes and instability (small perturbations of the sample points leading to large changes in the results).

Some machine learning methods are loosely related to the way in which biological brains learn from experience and function. Learning to drive a bicycle is not a matter of symbolic logic and equations but a matter of gradual tuning and ... rapidly recovering from initial accidents.

Chapter 5

Mastering generalized linear least-squares

Entia non sunt multiplicanda praeter necessitatem.
Entities must not be multiplied beyond necessity.
(William of Ockham c. 1285 – 1349)



Some issues were left open in the previous chapter about linear models. The output of a serious modeling effort is not only a single “take it or leave it” model. Usually one is dealing with multiple modeling architectures, with **judging the quality of a model** (the goodness-of-fit in our case) and **selecting the best possible architecture**, with **determining confidence regions** (e.g., error bars) for the estimated model parameters,

Latest chapter revision: November 14, 2013

etc. After reading this chapter you are supposed to raise from the status of casual user to that of professional least-squares guru.

In the previous chapter we mentioned a trick to take care of some nonlinearities: mapping the original input by some nonlinear function ϕ and then considering a linear model in the transformed input space (see Section 4.2). While the topics discussed in this Chapter are valid in the general case, your intuition will be helped if you keep in mind the special case of **polynomial fits** in one input variable, in which the nonlinear functions consist of powers of the original inputs, like

$$\phi_0(x) = x^0 = 1, \quad \phi_1(x) = x^1 = x, \quad \phi_2(x) = x^2, \ldots$$

The case is of particular interest and widely used in practice to deserve being studied.

The task is to estimate the functional dependence of output (Y) values on input (X) values. Given raw data in the form of pairs of values:

$$(x_i, y_i), \qquad i \in 1, 2, \ldots, N,$$

the aim is to derive a function $f(x)$ which appropriately models the dependence of Y on X, so that it is then possible to evaluate the function on new and unknown x values.

Identifying significant patterns and relationships implies eliminating insignificant details. For example, in many cases values are obtained by physical measurements, and every measurement is accompanied by errors (measurement *noise*). Think about modeling how the height of a person changes with age[1].

Coming back to the model, we are *not* going to demand that the function plot passes exactly through the sample values (i.e., we don't require that $y_i = f(x_i)$ for all points). We are not dealing with **interpolation** but with **fitting** (being compatible, similar or consistent). Losing fidelity is not a weakness but a strength, providing an opportunity to create more powerful models by simplifying the analysis and permitting reasoning that isn't bogged down by trivia. A comparison between a function interpolating all the points in the dataset, and a much simpler one, like in Fig. 5.1, can show immediately how these functions can behave differently in modeling the data distribution. *Occam's razor* illustrates this basic principle that simpler models should be preferred over unnecessarily complicated ones.

The freedom to choose different models, e.g., by picking polynomials of different degrees, is accompanied by the responsibility of judging the goodness of the different models. A standard way for polynomial fits is by statistics from the resulting sum-of-squared-errors, henceforth the name of least-squares methods.

[1]If you repeat measuring your height with a high precision instrument, you are going to get different values for each measurement. A reflection of these noisy measurements is the simple fact that you are giving a limited number of digits to describe your height (no mentally sane person would answer 1823477 micrometers when asked about his height).

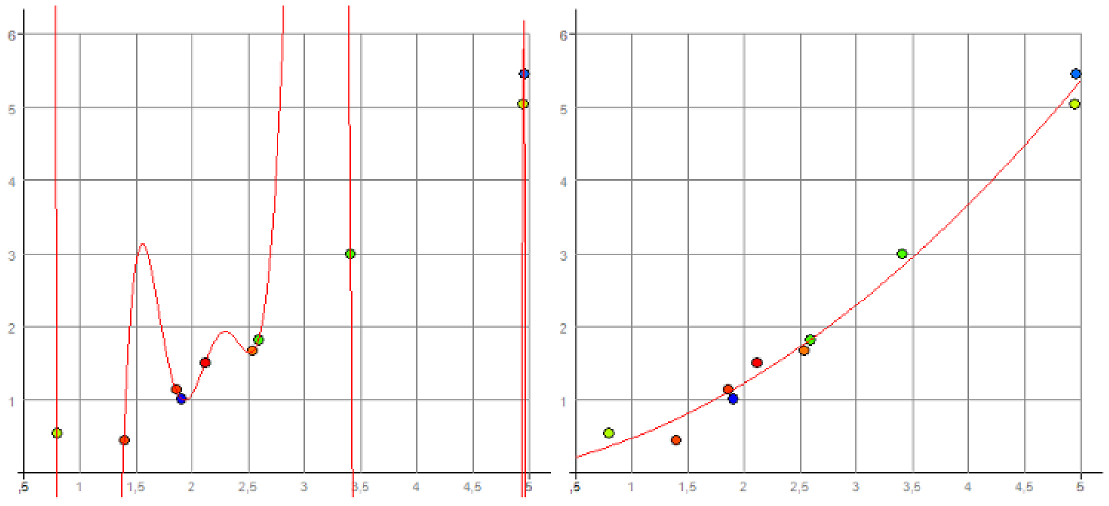


Figure 5.1: Comparison between interpolation and fitting. The number of free parameters of the polynomial (equal to its degree minus one) changes from the number of data points (left), to three (right).

5.1 Fitting as maximization of the goodness-of-fit

Let's get more concrete and consider the simple math behind fitting. Let's start with a polynomial of degree $M - 1$, where M is defined as the *degree bound*, equal to the degree plus one. M is also the number of free parameters (the constant term in the polynomial also counts). One searches for the polynomial of a suitable degree that best describes the data distribution:

$$f(x, \boldsymbol{c}) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{M-1} x^{M-1} = \sum_{k=0}^{M-1} c_k x^k. \tag{5.1}$$

When the dependence on parameters $\boldsymbol{c}$ is taken for granted, we will just use $f(x)$ for brevity. Because a polynomial is determined by its M parameters (collected in vector $\boldsymbol{c}$), we search for *optimal values* of these parameters. This is an example of what we call the *power of optimization*. The general recipe is: formulate the problem as one of minimizing a function and then resort to optimization.

For reasons having roots in statistics and maximum-likelihood estimation, described in the following Section 5.2, a widely used merit function to estimate the **goodness-of-fit** is given by the **chi-squared**, a term derived from the Greek letter used to identify a connected statistical distribution, χ^2:

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i)}{\sigma_i} \right)^2. \tag{5.2}$$

The explanation is simple if the parameters σ_i are all equal to one: in this case, χ^2 measures the sum of squared errors between the actual value y_i and the value obtained by the model $f(x_i)$, and it is precisely the ModelError($\boldsymbol{w}$) function described in the previous Chapter.

In some cases, however, the measurement processes may be different for different points and one has an estimate of the measurement error σ_i, assumed to be the standard deviation. Think about measurements done with instruments characterized by different degrees of precision, like a meter stick and a high-precision caliper.

The chi-square definition is a precise, mathematical method of expressing the obvious fact that an error of one millimeter is acceptable with a meterstick, much less so with a caliper: when computing χ^2, the errors have to be compared to the standard deviation (i.e., *normalized*), therefore the error is divided by σ_i. The result is a number that is independent from the actual error size, and whose meaning is standardized:

- A value $\chi^2 \approx 1$ is what we would expect if the σ_i are good estimates of the measurement noise and the model is good.
- Values of χ^2 that are too large mean that either you underestimated your source of errors, or that the model does not fit very well. If you trust your σ_i's, maybe increasing the polynomial degree will improve the result.
- Finally, if χ^2 is too small, then the agreement between the model $f(x)$ and the data (x_i, y_i) is suspiciously good; we are possibly in the situation shown in the left-hand side of Fig. 5.1 and we should reduce the polynomial degree[2].

Now that you have a precise way of measuring the quality of a polynomial model by the *chi-squared*, your problem becomes that of finding polynomial coefficients *minimizing* this error. An inspiring physical interpretation is illustrated in Fig. 5.2. Luckily, this problem is solvable with standard linear algebra techniques, as already explained in the previous chapter.

Here we complete the details of the analysis exercise as follows: take partial derivatives $\partial\chi^2/\partial c_k$ and require that they are equal to zero. Because the *chi-square* is quadratic in the coefficients c_k, we get a set of M linear equations to be solved:

$$0 = \frac{\partial \chi^2}{\partial c_k} = 2 \sum_{i=1}^{N} \frac{1}{{\sigma_i}^2} \left(y_i - \sum_{j=0}^{M-1} c_j {x_i}^j \right) {x_i}^k, \qquad \text{for } k = 0, 1, \ldots, M-1 \tag{5.3}$$

[2]Pearson's $chi - squared$ test provides objective thresholds for assessing the goodness-of-fit based on the value of χ^2, on the number of parameters and of data points, and on a desired confidence level, as explained in Section 5.2.

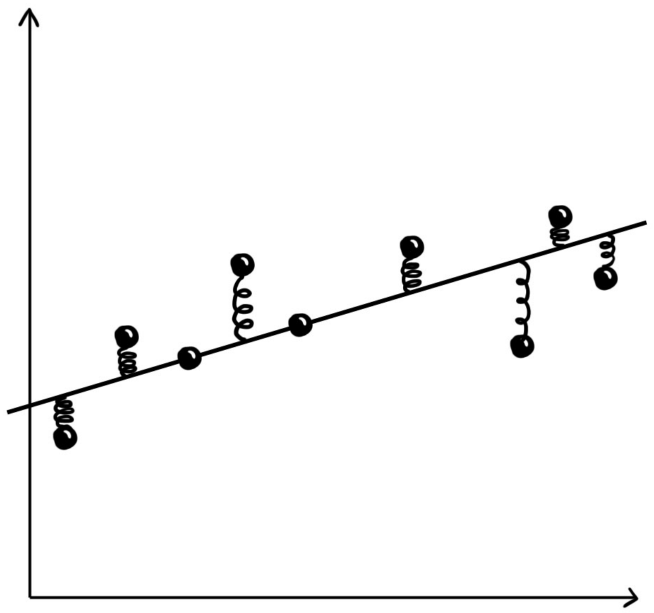


Figure 5.2: A fit by a line with a physical analogy: each data point is connected by a spring to the fitting line. The strength of the spring is proportional to $1/\sigma_i{}^2$. The minimum energy configuration corresponds to the minimum *chi-square*.

To shorten the math it is convenient to introduce the $N \times M$ matrix $A = (a_{ij})$ such that $a_{ij} = x_i{}^j/\sigma_i$, containing powers of the x_i coordinates normalized by σ_i, the vector $\boldsymbol{c}$ of the unknown coefficients, and the vector $\boldsymbol{b}$ such that $b_i = y_i/\sigma_i$.

It is easy to check that the linear system in equation (5.3) can be rewritten in a more compact form as:

$$(A^T \cdot A) \cdot \boldsymbol{c} = A^T \cdot \boldsymbol{b}, \tag{5.4}$$

which is called the *normal equation* of the least-squares problem.

The coefficients can be obtained by deriving the inverse matrix $C = (A^T \cdot A)^{-1}$, and using it to obtain $\boldsymbol{c} = C \cdot A^T \cdot \boldsymbol{b}$. Interestingly, C is the covariance matrix of the coefficients' vector $\boldsymbol{c}$ seen as a random variable: the diagonal elements of C are the variances (squared uncertainties) of the fitted parameters $c_{ii} = \sigma^2(c_i)$, and the off-diagonal elements are the covariances between pairs of parameters.

The matrix $(A^T \cdot A)^{-1} A^T$ is the **pseudo-inverse** already encountered in the previous chapter, generalizing the solutions of a system of linear equations in the **least-squared-errors** sense:

$$\min_{\boldsymbol{c} \in \mathbb{R}^M} \chi^2 = \|A \cdot \boldsymbol{c} - \boldsymbol{b}\|^2. \tag{5.5}$$

If an exact solution of equation (5.5) is possible the resulting *chi-squared* value is zero, and the line of fit passes exactly through all data points. This occurs if we have M parameters and M distinct pairs of points (x_i, y_i), leading to an invertible system of M linear equations in M unknowns. In this case we are not dealing with an approximated fit but with interpolation. If no exact solution is possible, as in the standard case of more pairs of points than parameters, the pseudo-inverse gives us the vector $\boldsymbol{c}$ such that $A \cdot \boldsymbol{c}$ is as close as possible to $\boldsymbol{b}$ in the Euclidean norm, a very intuitive way to interpret the approximated solution. Remember that good models of noisy data need to *summarize* the observed data, not to reproduce them exactly, so that the number of parameters has to be (much) less than the number of data points.

The above derivations are not limited to fitting a polynomial, we can now easily fit many other functions. In particular, if the function is given by a linear combination of basis functions $\phi_k(\boldsymbol{x})$, as

$$f(x) = \sum_{k=0}^{M-1} c_k \phi_k(\boldsymbol{x})$$

most of the work is already done. In fact, it is sufficient to substitute the basis function values in the matrix A, which now become $a_{ij} = \phi_j(\boldsymbol{x}_i)/\sigma_i$. We have therefore a powerful mechanism to fit more complex functions like, for example,

$$f(x) = c_0 + c_1 \cos x + c_2 \log x + c_3 \tanh x^3.$$

Let's only note that the unknown parameters must appear *linearly*, they cannot appear in the arguments of functions. For example, by this method we cannot fit directly $f(x) = \exp(-cx)$, or $f(x) = \tanh(x^3/c)$. At most, we can try to transform the problem to recover the case of a linear combination of basis functions. In the first case, we can for example fit the values $\hat{y}_i = \log y_i$ with a linear function $f(\hat{y}) = -cx$, but this trick will not be possible in the general case.

A polynomial fit is shown as a curve in the scatterplot of Fig. 5.3, which shows a fit with a polynomial of degree 2 (a parabola). A visual comparison of the red line and the data points can already give a visual feedback about the goodness-of-fit (the *chi-squared* value). This 'chi-by-eye" approach consists of looking at the plot and judging it to look nice or bad w.r.t. the scatterplot of the original measurements.

When the experimental data do not follow a polynomial law, fitting a polynomial is not very useful and can be misleading. As explained above, a low value of the *chi-squared* can still be reached by increasing the degree of the polynomial: this will give it a larger freedom of movement to pass closer and closer to the experimental data. The polynomial will actually *interpolate* the points with zero error if the number of parameters of the polynomial equals the number of points. But this reduction in the error will be paid by wild oscillations of the curve *between* the original points, as shown in Fig. 5.1 (left). The model is not summarizing the data and it has serious *difficulties in generalizing*. It fails to predict y values for x values which are different from the ones used for building the polynomial.

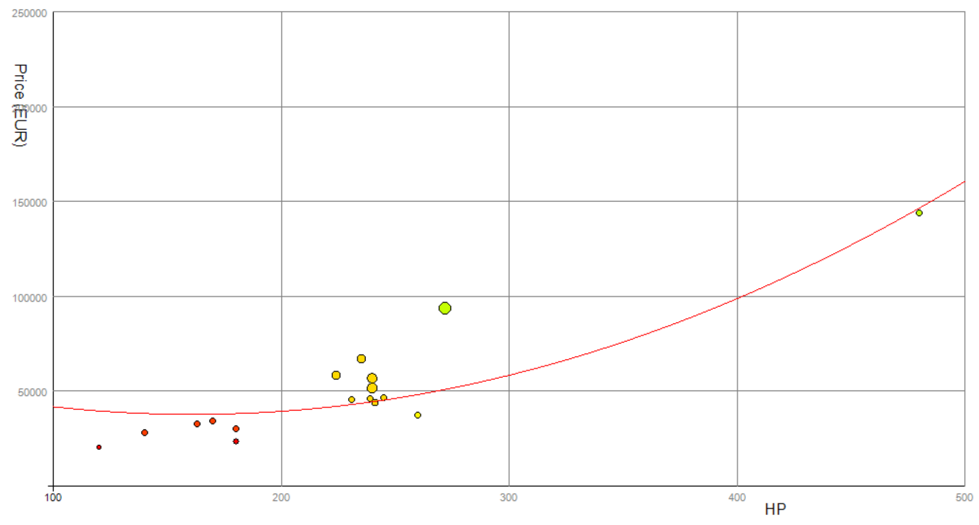


Figure 5.3: A polynomial fit: price of cars as a function of engine power.

The above problem is a very general one and related to the issue of **overfitting**. In statistics, overfitting occurs when a model tends to describe random error or noise instead of the underlying relationship. Overfitting occurs when a model is too complex, such as having too many degrees of freedom, in relation to the amount of data available (too many coefficients in the polynomial in our case).

An overfitted model will generally have a poor predictive performance. An analogy in human behavior can be found in teaching: if a student concentrates and memorizes only the details of the teacher's presentation (for example the details of a specific exercise in mathematics) without extracting and understanding the underlying rules and meaning he will only be able to vacuously repeat the teacher's words by heart, but not to generalize his knowledge to new cases.

5.2 Which fit is the fittest of them all?

Now that the basic technology for generalized least-squares fitting is known, let's pause to consider some additional motivation from statistics. In addition, given the freedom of selecting different models, for example different degrees for a fitting polynomial, instructions to identify the best model architecture are precious to go beyond the superficial "chi-by-eye" method. Good fits are precious because they identify potentially interesting relationships between the chosen Y and X coordinates.

The least-squares fitting process is as follows:

1. Assume that Mother Nature and the experimental procedure (including measurements) are generating independent experimental samples (x_i, y_i). Assume that the y_i values are affected by errors distributed according to a normal (i.e., Gaussian) distribution.

2. If the model parameters $\boldsymbol{c}$ are known, one can estimate the probability of our measured data, given the parameters. In statistical terms this is called **likelihood** of the data.

3. Least-squares fitting is equivalent to searching for the parameters which are maximizing the likelihood of our data. Least-squares is a **maximum likelihood estimator**. Intuitively, this maximizes the "agreement" of the selected model with the observed data.

The demonstration is straightforward. You may want to refresh Gaussian distributions in Section 5.3 before proceeding. The probability for a single data point to be in an interval of width $\mathrm{d}y$ around its measure value y_i is proportional to

$$\exp\left(-\frac{1}{2}\left(\frac{y_i - f(x_i, \boldsymbol{c})}{\sigma_i}\right)^2\right)\mathrm{d}y. \tag{5.6}$$

Because points are generated independently, the same probability for the entire experimental sequence (its *likelihood*) is obtained by multiplying individual probabilities:

$$\mathrm{d}P \propto \prod_{i=1}^{N} \exp\left(-\frac{1}{2}\left(\frac{y_i - f(x_i, \boldsymbol{c})}{\sigma_i}\right)^2\right)\mathrm{d}y. \tag{5.7}$$

One is maximizing over $\boldsymbol{c}$ and therefore constant factors like $(\mathrm{d}y)^N$ can be omitted. In addition, maximizing the likelihood is equivalent to maximizing its logarithm (the logarithm is in fact an increasing function of its argument). Well, because of basic properties of logarithms (namely they transform products into sums, powers into products, etc.), the logarithm of equation (5.7), when constant terms are omitted, is precisely the definition of *chi-squared* in equation (5.2). The connection between least-squares fitting and maximum likelihood estimation should be now clear.

5.2.1 Hypothesis testing

You can now proceed with statistics to **judge the quality** of your model. The fundamental question to ask is: considering the N experimental points and given the estimated M parameters, what is the probability that, by chance, values equal to or larger than our measured *chi-squared* are obtained? The above question translates the obvious question about our data ("**what is the likelihood of measuring the data that one actually did**

measure?") into a more precise statistical form, i.e.: "**what's the probability that another set of sample data fits the model even worse than our current set does?**" If this probability is high, the obtained discrepancies between y_i and $f(x_i, \boldsymbol{c})$ make sense from a statistical point of view. If this probability is very low, either you have been very unlucky, or something does not work in your model: errors are too large with respect to what can be expected by Mother Nature plus the measurement generation process.

Let $\hat{\chi}^2$ be the *chi-squared* computed on your chosen model for a given set of inputs and outputs. This value follows a probability distribution called, again, *chi-squared with ν degrees of freedom* (χ^2_ν), where the number of degrees of freedom ν determines how much the dataset is "larger" than the model. If we assume that errors are normally distributed with null mean and unit variance (remember, we already normalized them), then $\nu = N - M$[3]. Our desired goodness-of-fit measure is therefore expressed by the parameter Q as follows:

$$Q = Q_{\hat{\chi}^2,\nu} = \Pr(\chi^2_\nu \geq \hat{\chi}^2).$$

The value of Q for a given empirical value of $\hat{\chi}^2$ and the given number of degrees of freedom can be calculated or read from tables[4].

The importance of the *number of degrees of freedom ν*, which decreases as long as the number of parameters in the model increases, becomes apparent when models with different numbers of parameters are compared. As we mentioned, it is easy to get a low *chi-square* value by increasing the number of parameters. Using Q to measure the goodness-of-fit takes this effect into account. A model with a larger *chi-square* value (larger errors) can produce a higher Q value (i.e., be better) w.r.t. one with smaller errors but a larger number of parameters.

By using the goodness-of-fit Q measure you can rank different models and pick the most appropriate one. The process sounds now clear and quantitative. If you are fitting a polynomial, you can now repeat the process with different degrees, measure Q and select the best model architecture (the best polynomial degree).

But the devil is in the details: the machinery works *provided that the assumptions are correct*, provided that errors follow the correct distribution, that the σ_i are known and appropriate, that the measurements are independent. By the way, asking for σ_i values can be a puzzling question for non-sophisticated users. You need to proceed with caution: **statistics is a minefield if assumptions are wrong** and a single wrong assumption makes the entire chain of arguments explode.

[3]In the general case, the correct number of degrees of freedom also depends on the number of parameters needed to express the error distribution (e.g., skewness).

[4]The exact formula is

$$Q_{\hat{\chi}^2,\nu} = \Pr(\chi^2_\nu \geq \hat{\chi}^2) = \left(2^{\frac{\nu}{2}}\Gamma\left(\frac{\nu}{2}\right)\right)^{-1}\int_{\hat{\chi}^2}^{+\infty} t^{\frac{\nu}{2}-1}e^{-\frac{t}{2}}\,\mathrm{d}t,$$

which can be easily calculated in this era of cheap CPU power.

5.2.2 Non-parametric tests

What if you cannot provide our σ_i's? What if you cannot guess the error distribution? If you have no prior assumptions about the errors, the chi-squared test described above cannot be applied. This doesn't imply that least-squares models are useless, of course: you can either play the card of statistical non-parametric tests, or exit the comfortable realm of "historical" statistical results and estimate errors via a more brute-force approach based on Machine Learning.

Let us briefly consider the first option, non-parametric tests: consider the residues

$$\epsilon_i = y_i - f(x_i, \boldsymbol{c}), \qquad i = 1, \ldots, n,$$

i.e., the differences between the outcomes predicted by our model and the corresponding observed values. In many cases, we can define a model "good" if the residues, the errors it doesn't account for, follow a normal distribution. The underlying assumption is that if the residue distribution is not normal, while it is reasonable to assume that the measurement errors are, then the model is not considering some important factor, or it is not powerful enough (degree too low?), or it's too powerful.

Some statistical tests are precisely devoted to this kind of questions: are the residues distributed normally? A particularly useful test in this context (but all previous caveats about statistics being a minefield still apply here) is the **Anderson-Darling test**, which answers the question "Does the given sample set follow a given distribution?"[5] For the normal distribution, the A-D test amounts to the following steps:

1. set a significance value α — roughly speaking, the probability of being wrong if rejecting the normality hypothesis (we may want to be very conservative in this and ask for a low error here, usually $\alpha = .1$, .5, or .01);

2. sort the residues ϵ_i in increasing order;

3. compute the RMSE $\hat{\sigma}$ of the residues:

$$\hat{\sigma} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} {\epsilon_i}^2};$$

4. normalize the residues:

$$\epsilon_i' = \frac{\epsilon_i}{\hat{\sigma}};$$

[5]Many other tests are available, most notably D'Agostino's K-Squared test[15]. For didactical and historical reasons, one may also be interested in the Kolmogorov-Smirnov test.

5. compute the following statistic:

$$A = \left(-n - \sum_{i=1}^{N} (2i-1) \left(\ln \Phi(\epsilon'_i) + \ln \left(1 - \Phi(\epsilon'_{n-i+1}) \right) \right) \right)^{\frac{1}{2}},$$

   where $\Phi(x)$ is the Cumulative Distribution Function (CDF) of a normalized Gaussian;

6. compare A with a threshold θ computed on the basis of the distribution of the A statistic and depending on the chosen significance value α; generally accepted values are the following:

$$\theta = \begin{cases} .656 & \text{if } \alpha = .1 \\ .787 & \text{if } \alpha = .05 \\ 1.092 & \text{if } \alpha = .01; \end{cases}$$

   reject the normality hypothesis if $A > \theta$.

Beware — if some residues are very large or very small, A might be infinite because of numeric roundings in the computation of Φ. In such case, either we consider the offending residues as outliers and remove the corresponding samples from the model, or we “follow the rules” and reject the normality hypothesis.

5.2.3 Cross-validation

Up to now we presented “historical” results, statistics was born well before the advent of computers, when calculations were very costly. Luckily, the current abundance of computational power permits robust techniques to estimate error bars and gauge the confidence in your models and their predictions. These methods do not require advanced mathematics, they are normally easy to understand, and they tend to be robust w.r.t. different distributions of errors.

In particular the **cross-validation** method of Section 3.2 can be used to select the best model. As usual the basic idea is to keep some measurements in the pocket, use the other ones to identify the model, take them out of the pocket to estimate errors on new examples, repeat and average the results. These estimates of generalization can be used to identify the best model architecture in a robust manner, provided that data is abundant. The distribution of results by the different folds of cross-validation gives information about the stability of the estimates, and permits to assert that, with a given probability (confidence), expected generalization results will be in a given performance range. The issue of deriving **error bars** for performance estimates, or, in general, for quantities estimated from the data, is explored in the next section.

5.3 Bootstrapping your confidence (error bars)

Let's imagine that Mother Nature is producing data (input-output pairs) from a true polynomial characterized by parameters $\boldsymbol{c}$. Mother Nature picks all x_i's randomly, independently and from the same distribution and produces $y_i = f(x_i, \boldsymbol{c}) + \epsilon_i$, according to equation (5.1) plus error ϵ_i.

By using generalized linear least squares you determine the maximum likelihood value $\boldsymbol{c}^{(0)}$ for the (x_i, y_i) pairs that you have been provided. If the above generation by Mother Nature and the above estimation process are repeated, there is no guarantee that you will get the same value $\boldsymbol{c}^{(0)}$ again. On the contrary, most probably you will get a different $\boldsymbol{c}^{(1)}$, then $\boldsymbol{c}^{(2)}$, etc.

It is unfair to run the estimation once and just use the first $\boldsymbol{c}^{(0)}$ that you get. If you could run many processes you could find average values for the coefficients, estimate **error bars**, maybe even use many different models and average their results (*ensemble* or *democratic* methods will be considered in later chapters). Error bars allow quantifying your confidence level in the estimation, so that you can say: with probability 90% (or whatever confidence value you decide), the coefficient c_i will have a value between $c - B$ and $c + B$, B being the estimated error bar[6]. Or, "We are 99% confident that the true value of the parameter is in our confidence interval." When the model is used, similar error bars can be obtained on the predicted y values. For data generated by simulators, this kind of repeated and randomized process is called a **Monte Carlo experiment**[7].

On the other hand, Mother Nature, i.e. the process generating your data, can deliver just a single set of measurements, and repeated interrogations can be too costly to afford. How can you get the advantages of repeating different and randomized estimates by using just *one* series of measurements? At first, it looks like an absurdly impossible action. Similar absurdities occur in the "Surprising Adventures," when Baron Munchausen pulls himself and his horse out of a swamp by his hair (Fig. 5.4), and to imitate him one could try to "pull oneself over a fence by one's bootstraps," hence the modern meaning of the term *bootstrapping* as a description of a self-sustaining process.

Well, it turns out that there is indeed a way to use a single series of measurements to imitate a real Monte Carlo method. This can be implemented by constructing a number of *resamples* of the observed dataset (and of equal size). Each new sample is obtained by *random sampling with replacement*, so that the same case can be taken more than

[6]As a side observation, if you know that an error bar is 0.1, you will avoid putting too many digits after the decimal point. If you estimate your height, please do not write "182.326548435054cm": stopping at 182.3cm (plus or minus 0.1cm) will be fine.

[7]Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to obtain numerical results; i.e., by running simulations many times over just like actually playing and recording your results in a real casino situation: hence the name from the town of Monte Carlo in the Principality of Monaco, the European version of Las Vegas.

Figure 5.4: Baron Munchausen pulls himself out of a swamp by his hair.

once (Fig. 5.5). By simple math and for large numbers N of examples, about 37% of the examples[8] are not present in a sample, because they are replaced by multiple copies of the original cases[9].

For each new i-th resample the fit procedure is repeated, obtaining many estimates $\boldsymbol{c}_i$ of the model parameters. One can then analyze how the various estimates are distributed, using observed frequencies to estimate a probability distribution, and summarizing the distribution with confidence intervals. For example, after fixing a confidence level of 90% one can determine the region around the median value of $\boldsymbol{c}$ where an estimated $\boldsymbol{c}$ will fall with probability 0.9. Depending on the sophistication level, the confidence region in more than one dimension can be given by rectangular intervals or by more flexible regions, like ellipses. An example of a confidence interval in one dimension (a single parameter c to be estimated) is given in Fig. 5.6. Note that confidence intervals can be obtained for arbitrary distributions, not necessarily normal, and

[8]Actually approximately $1/e$ of the examples.

[9]In spite of its "brute force" quick-and-dirty look, bootstrapping enjoys a growing reputation also among statisticians. The basic idea is that the actual data set, viewed as a probability distribution consisting of a sum of *Dirac delta* functions on the measured values, is in most cases the best available estimator of the underlying probability distribution [27].

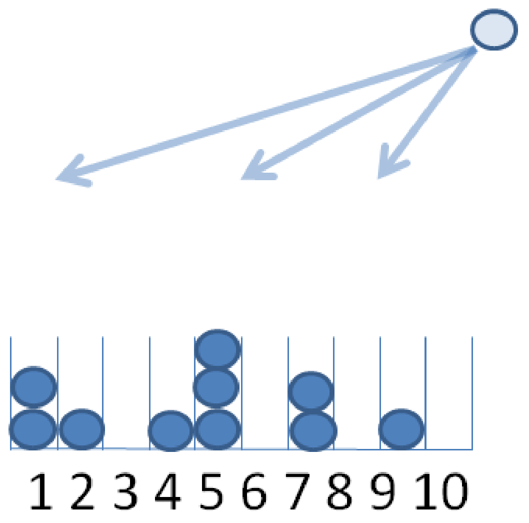


Figure 5.5: Bootstrapping: 10 balls are thrown with uniform probability to end up in the 10 boxes. They decide which cases and how many copies are present in the bootstrap sample (two copies of case 1, one copy of case 2, zero copies of case 3,. . .

confidence intervals do not need to be symmetric around the median.

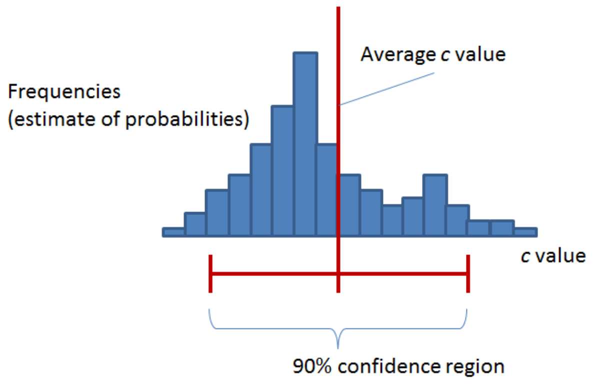


Figure 5.6: Confidence interval: from the histogram characterizing the distribution of the estimated c values one derives the region around the average value collecting 90% of the cases. Other confidence levels can be used, like 68.3%, 95.4%. etc. (the historical probability values corresponding to σ and 2σ in the case of normal distributions).

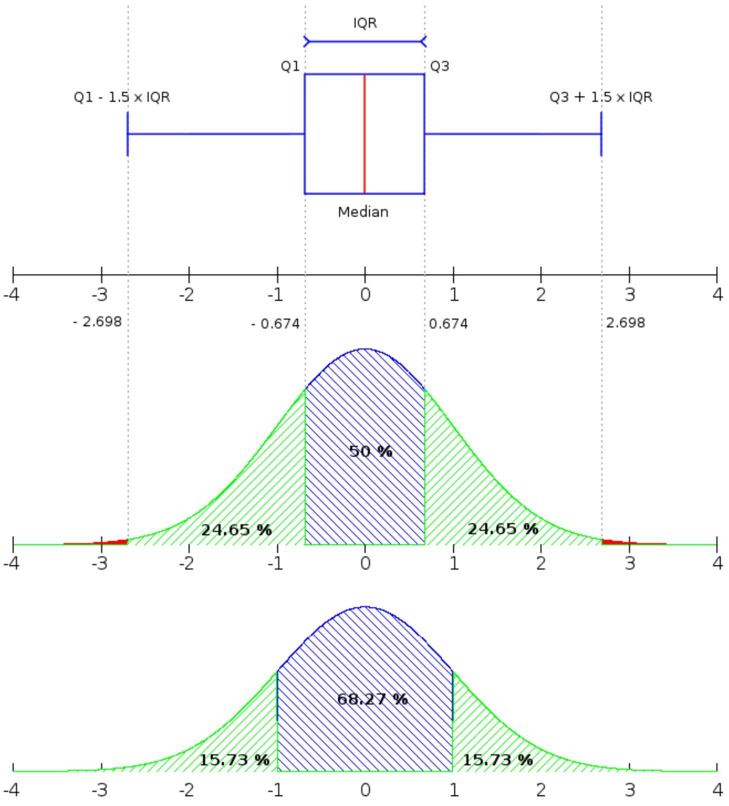


Figure 5.7: Comparison between box plot (above) and a normal distribution. The X axis shows positions in terms of the standard deviation σ. For example, in the bottom plot, 68.27% of the points fall within plus or minus one σ from the mean value.

Appendix: Plotting confidence regions (percentiles and box plots)

A quick-and-dirty way to analyze the distribution of estimated parameters is by histograms (counting frequencies for values occurring in a set of intervals). In some cases the histogram contains more information than what is needed, and the information is not easily interpreted. A very compact way to represent a distribution of values is by its **average value** μ. Given a set $\mathcal{X}$ of N values x_i, the average is

$$\mu(\mathcal{X}) = (\sum_{i=1}^{N} x_i)/N, \qquad x_{i,\ldots,N} \in \mathcal{X}. \tag{5.8}$$

The average value is also called the expected value, or mathematical expectation, or mean, or first moment, and denoted in different ways, for example as $\overline{x}$ or $E(x)$.

A related but different value is the **median**, defined as the value separating the higher half of a sample from the lower half. Given a finite list of values, it can be found by sorting all the observations from the lowest to the highest value and picking the middle one. If some **outliers** are present (data which are far from most of the other values), the median is a more robust measure than the average, which can be heavily influenced by outliers. On the contrary, if the data are clustered, like when they are produced by a normal distribution, the average tends to coincide with the median. The median can be generalized by considering the **percentile**, the value of a variable below which a certain percentage of observations fall. So the 10th percentile is the value below which 10 percent of the observations are found. **Quartiles** are a specific case, they are the lower quartile (25-th percentile), the median, and the upper quartile (75-th percentile). The interquartile range (IQR), also called the midspread or middle fifty, is a measure of statistical dispersion, being equal to the difference between the third and first quartiles.

A **box plot**, also known as a **box-and-whisker plot**, shows five-number summaries of the set of values: the smallest observation (sample minimum), the lower quartile (Q1), the median (Q2), the upper quartile (Q3), and the largest observation (sample maximum). A box plot may also indicate which observations, if any, might be considered outliers, usually shown by circles. In a box plot, the bottom and top of the box are always the lower and upper quartiles, and the band near the middle of the box is always the median. The ends of the whiskers can represent several possible alternative values, for example:

- the minimum and maximum of all the data;
- one standard deviation above and below the mean of the data;
- the 9th percentile and the 91st percentile;
- ...

Fig. 5.7 presents a box plot with 1.5 IQR whiskers, which is the usual (default) value, corresponding to about plus or minus 2.7σ and 99.3 coverage, if the data are normally distributed. In other words, for a Gaussian distribution, on average less than 1 percent of the data fall outside the box-plus-whiskers range, a useful indication to identify possible outliers. As mentioned, an outlier is one observation that appears to deviate markedly from other members of the sample in which it occurs. Outliers can occur by chance in any distribution, but they often indicate either measurement errors or that the population has a *heavy-tailed distribution*. In the former case one should discard them or use statistics that are *robust* to outliers, while in the latter case one should be cautious in relying on tools or intuitions that assume a normal distribution.

Gist

Polynomial fits are a specific way to use linear models to deal with nonlinear problems. The model consists of a linear sum of coefficients (to be determined) multiplying products of original input variables. The same technology works if products are substituted with arbitrary functions of the input variables, provided that the functions are fixed (no free parameters in the functions, only as multiplicative coefficients). Optimal coefficients are determined by minimizing a sum of squared errors, which leads to solving a set of linear equations. If the number of input-output examples is not larger than the number of coefficients, over-fitting appears and it is dangerous to use the model to derive outputs for novel input values.

The goodness of a polynomial model can be judged by evaluating the probability of getting the observed discrepancy between predicted and measured data (the likelihood of the data given the model parameters). If this probability is very low we should not trust the model too much. But wrong assumptions about how the errors are generated may easily lead us to overly optimistic or overly pessimistic conclusions. Statistics builds solid scientific constructions starting from assumptions. Even the most solid statistics construction will be shattered if built on a the sand of invalid assumptions. Luckily, approaches based on easily affordable massive computation like cross-validation are easy to understand and robust.

“Absurdities” like bootstrapping (re-sampling the same data and repeating the estimation process in a Monte Carlo fashion) can be used to obtain confidence intervals around estimated parameter values.

You just maximized the likelihood of being recognized as linear least-squares guru.

Chapter 6

Rules, decision trees, and forests

If a tree falls in the forest and there's no one there to hear it, does it make a sound?

Rules are a way to **condense nuggets of knowledge in a way amenable to human understanding**. If "customer is wealthy" then "he will buy my product." If "body temperature greater than 37 degrees Celsius" then "patient is sick." Decision rules are commonly used in the medical field, in banking and insurance, in specifying processes to deal with customers, etc.

In a rule one distinguishes the **antecedent, or precondition** (a series of tests), and the **consequent, or conclusion**. The conclusion gives the output class corresponding to inputs which make the precondition true (or a probability distribution over the classes if the class is not 100% clear). Usually, the preconditions are *AND*-ed together: all tests must succeed if the rule is to "fire," i.e., to lead to the conclusion. If "distance less than

2 miles” AND “sunny” then “walk.” A test can be on the value of a categorical variable (”sunny”), or on the result of a simple calculation on numerical variables (”distance less than 2 miles”). The calculation has to be simple if a human has to understand. A practical improvement is to unite in one statement also the classification when the antecedent is false. If “distance less than 3 kilometers” AND “no car” then “walk” else “take the bus”.

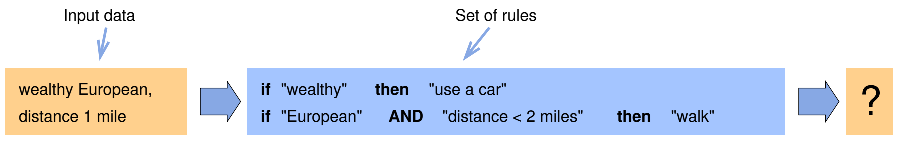

Input data
Set of rules
wealthy European, distance 1 mile
if "wealthy" then "use a car"
if "European" AND "distance < 2 miles" then "walk"
?

Figure 6.1: A set of unstructured rules can lead to contradictory classifications.

Extracting knowledge nuggets as a set of simple rules is enticing. But **designing and maintaining rules by hand is expensive** and difficult. When the set of rules gets large, complexities can appear, like rules leading to different and contradictory classifications (Fig. 6.1). In these cases, the classification may depend on the order with which the different rules are tested on the data and fire. **Automated ways to extract non-contradictory rules** from data are precious.

Instead of dealing with very long preconditions with many tests, breaking rules into a chain of simple questions has value. In a greedy manner, the most informative questions are better placed at the beginning of the sequence, leading to **a hierarchy of questions**, from the most informative to the least informative. The above motivations lead in a natural way to consider **decision trees**, an organized hierarchical arrangement of decision rules, without contradictions (Fig. 6.2, top).

Decision trees have been popular since the beginning of machine learning (ML). Now, it is true that only small and shallow trees can be “understood” by a human, but the popularity of decision trees is recently growing with the abundance of computing power and memory. Many, in some cases hundreds of trees, can be jointly used as **decision forests** to obtain robust classifiers. When considering forests, the care for human understanding falls in the background, the pragmatic search for robust top-quality performance without the risk of overtraining comes to the foreground.

6.1 Building decision trees

A decision tree is a set of questions organized in a hierarchical manner and represented graphically as a tree. Historically, trees in ML, as in all of Computer Science, tend to be drawn with their root upwards — imagine trees in Australia if you are in the Northern hemisphere. For a given input object, a decision tree estimates an unknown property of the object by asking successive questions about its known properties. Which question

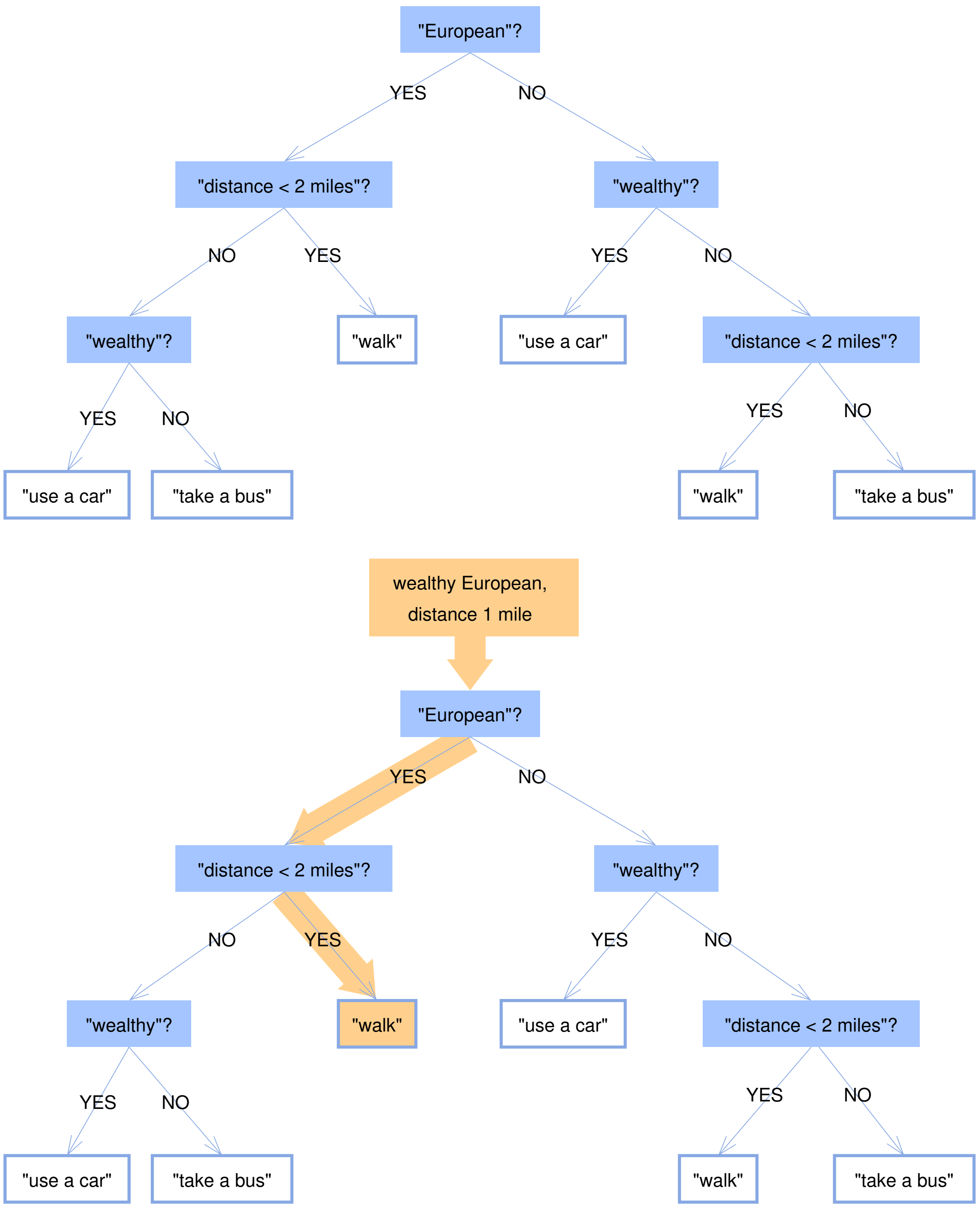


Figure 6.2: A decision tree (top), and the same tree working to reach a classification (bottom). The data point arriving at the split node is sent to its left or right child node according to the result of the test function.

to ask next depends on the answer of the previous question and this relationship is represented graphically as a path through the tree which the object follows, the orange thick path in the bottom part of Fig. 6.2. The decision is then made based on the terminal node on the path. Terminal nodes are called leaves. A decision tree can also be thought of as a technique for splitting complex problems into a set of simpler ones.

A basic way to build a decision tree from labeled examples proceeds in a greedy manner: the most informative questions are asked as soon as possible in the hierarchy. Imagine that one considers the initial set of labeled examples. A question with two possible outputs ("YES" or "NO") will divide this set into two subsets, containing the examples with answer "YES", and those with answer "NO", respectively. The initial situation is usually confused, and examples of different classes are present. When the leaves are reached after the chain of questions descending from the root, the final remaining set in the leaf should be almost "pure", i.e., consisting of examples of the same class. This class is returned as classification output for all cases trickling down to reach that leaf.

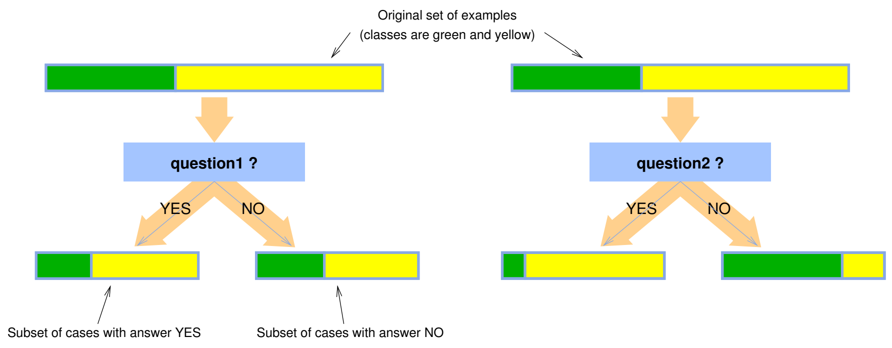

Original set of examples
(classes are green and yellow)
question1 ?
YES
NO
question2 ?
YES
NO
Subset of cases with answer YES
Subset of cases with answer NO

Figure 6.3: Purification of sets (examples of two classes): question2 produces purer subsets of examples at the children nodes.

We need to transition from an initial confused set to a final family of (nearly) pure sets. A **greedy** way to aim at this goal is to start with the "most informative" question. This will split the initial set into two subsets, corresponding to the "YES" or "NO" answer, the children sets of the initial root node (Fig. 6.3). A greedy algorithm will take a first step leading as close as possible to the final goal. In a greedy algorithm, the first question is designed in order to get the two children subsets as pure as possible. After the first subdivision is done, one proceeds in a **recursive** manner (Fig. 6.4), by using the *same* method for the left and right children sets, designing the appropriate questions, and so on and so forth until the remaining sets are sufficiently pure to stop the recursion. The complete decision tree is induced in a top-down process guided by the relative proportions of cases remaining in the created subsets.

The two main ingredients are a quantitative measure of purity and the kind of questions to ask at each node. We all agree that maximum purity is for subsets with cases

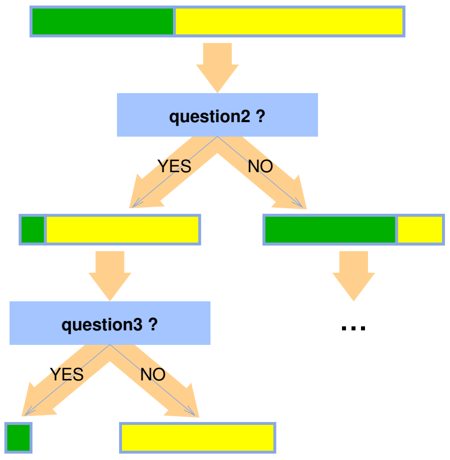


Figure 6.4: Recursive step in tree building: after the initial purification by *question2*, the same method is applied on the left and right example subsets. In this case *question3* is sufficient to completely purify the subsets. No additional recursive call is executed on the pure subsets.

of one class only, the different measures deal with measuring impure combinations. Additional spices have to do with termination criteria: let's remember that we aim at generalization so that we want to discourage very large trees with only one or two examples left in the leaves. In some cases one stops with slightly impure subsets, and an output probability for the different classes when cases reach a given leaf node.

For the following description, let's assume that all variables involved are categorical (names, like the "European" in the above example). The two widely used measures of purity of a subset are the *information gain* and the *Gini impurity*. Note that we are dealing with supervised classification, so that we know the correct output classification for the training examples.

Information gain Suppose that we sample from a set associated to an internal node or to a leaf of the tree. We are going to get examples of a class y with a probability $\Pr(y)$ proportional to the fraction of cases of the class present in the set. The statistical uncertainty in the obtained class is measured by Shannon's **entropy** of the label probability distribution:

$$H(Y) = -\sum_{y \in Y} \Pr(y) \log \Pr(y). \tag{6.1}$$

In information theory, entropy quantifies the average information needed to specify which event occurred (Fig. 6.5). If the logarithm's base is 2, information (hence entropy) is measured in bits. Entropy is maximum, $H(Y) = \log n$, when all n classes have the same share of a set, while it is minimum, $H(Y) = 0$, when all cases belong to

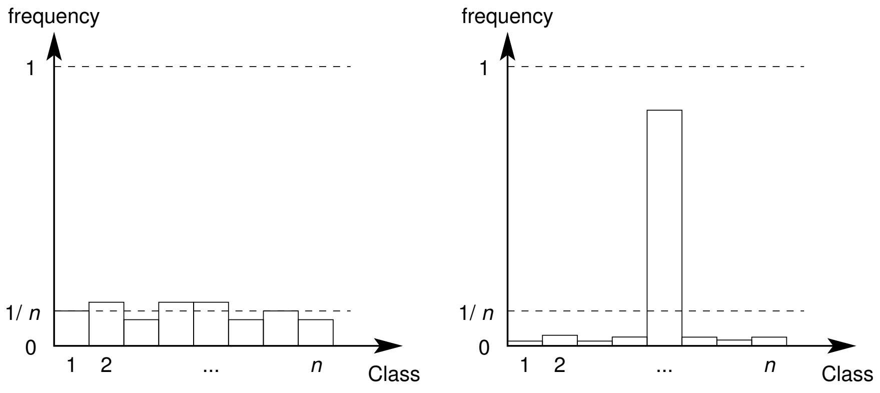


Figure 6.5: Two distributions with widely different entropy. High entropy (left): events have similar probabilities, the uncertainty is close to the maximum one ($\log n$). Low entropy (right): events have very different probabilities, the uncertainty is small and close to zero because one event collects most probability.

the same class (no information is needed in this case, we already know which class we are going to get).
In the information gain method **the impurity of a set is measured by the entropy** of the class probability distribution.
Knowledge of the answer to a question will decrease the entropy, or leave it equal only if the answer does not depend on the class. Let $\mathcal{S}$ be the current set of examples, and let $\mathcal{S} = \mathcal{S}_{\text{YES}} \cup \mathcal{S}_{\text{NO}}$ be the splitting obtained after answering a question about an attribute. The ideal question leaves no indecision: all elements in $\mathcal{S}_{\text{YES}}$ are cases of one class, while elements of $\mathcal{S}_{\text{NO}}$ belong to another class, therefore the entropy of the two resulting subsets is zero.
The average reduction in entropy after knowing the answer, also known as the "information gain", is the mutual information (MI) between the answer and the class variable. With this notation, the information gain (mutual information) is:

$$\text{IG} = H(\mathcal{S}) - \frac{|\mathcal{S}_{\text{YES}}|}{|\mathcal{S}|} H(\mathcal{S}_{\text{YES}}) - \frac{|\mathcal{S}_{\text{NO}}|}{|\mathcal{S}|} H(\mathcal{S}_{\text{NO}}). \tag{6.2}$$

Probabilities needed to compute entropies can be approximated with the corresponding frequencies of each class value within the sample subsets.
Information gain is used by the ID3, C4.5 and C5.0 methods pioneered by Quinlan [28]. Let's note that, because we are interested in generalization, the information gain is useful but not perfect. Suppose that we are building a decision tree for some data describing the customers of a business and that each node can have more than two children. One of the input attributes might be the customer's credit card number. This attribute has a high mutual information with respect to any classification, because it uniquely identifies each customer, but we do not want to include it in the decision tree: deciding how to

treat a customer based on their credit card number is unlikely to generalize to customers we haven't seen before (we are over-fitting).

Gini impurity Imagine that we extract a random element from a set and label it randomly, with probability proportional to the shares of the different classes in the set. Primitive as it looks, this quick and dirty method produces zero errors if the set is pure, and a small error rate if a single class gets the largest share of the set.
In general, the Gini impurity (GI for short) measures how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset[1]. It is computed as the expected value of the mistake probability; as usual, the expectation is given by adding, for each class, the probability of mistaking the classification of an item in that class (i.e., the probability of assigning it to any class but the correct one: $1 - p_i$) times the probability for an item to be in that class (p_i). Suppose that there are m classes, and let f_i be the fraction of items labeled with value i in the set. Then, by estimating probabilities with frequencies ($p_i \approx f_i$):

$$\mathrm{GI}(f) = \sum_{i=1}^{m} f_i(1 - f_i) = \sum_{i=1}^{m}(f_i - f_i{}^2) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} f_i{}^2 = 1 - \sum_{i=1}^{m} f_i{}^2. \qquad (6.3)$$

GI reaches its minimum (zero) when all cases in the node fall into a single target category. GI is used by the CART algorithm (Classification And Regression Tree) proposed by Breiman [10].

When one considers the kind of questions asked at each node, considering **questions with a binary output** is sufficient in practice. For a categorical variable, the test can be based on the variable having a subset of the possible values (for example, if day is "Saturday or Sunday" YES, otherwise NO). For real-valued variables, the tests at each node can be on a single variable (like: *distance* $<$ *2 miles*) or on simple combinations, like a linear function of a subset of variables compared with a threshold (like: *average of customer's spending on cars and motorbikes greater than 20K dollars*). The above concepts can be generalized for a numeric variable to be predicted, leading to **regression trees** [10]. Each leaf can contain either the average numeric output value for cases reaching the leaf, or a simple model derived from the contained cases, like a linear fit (in this last case one talks about **model trees**).

In real-world data, **Missing values** are abundant like snowflakes in winter. A missing value can have two possible meanings. In some cases the fact that a value *is* missing

[1]This definition is easy to understand, and this explains why a more general version, known as Gini Index, Coefficient, or Ratio, is widely used in econometrics to describe inequalities in resource distribution within a population; newspapers periodically publish rankings of countries based on their Gini index with respect to socio-economical variables — without any explanation for the layperson, of course.

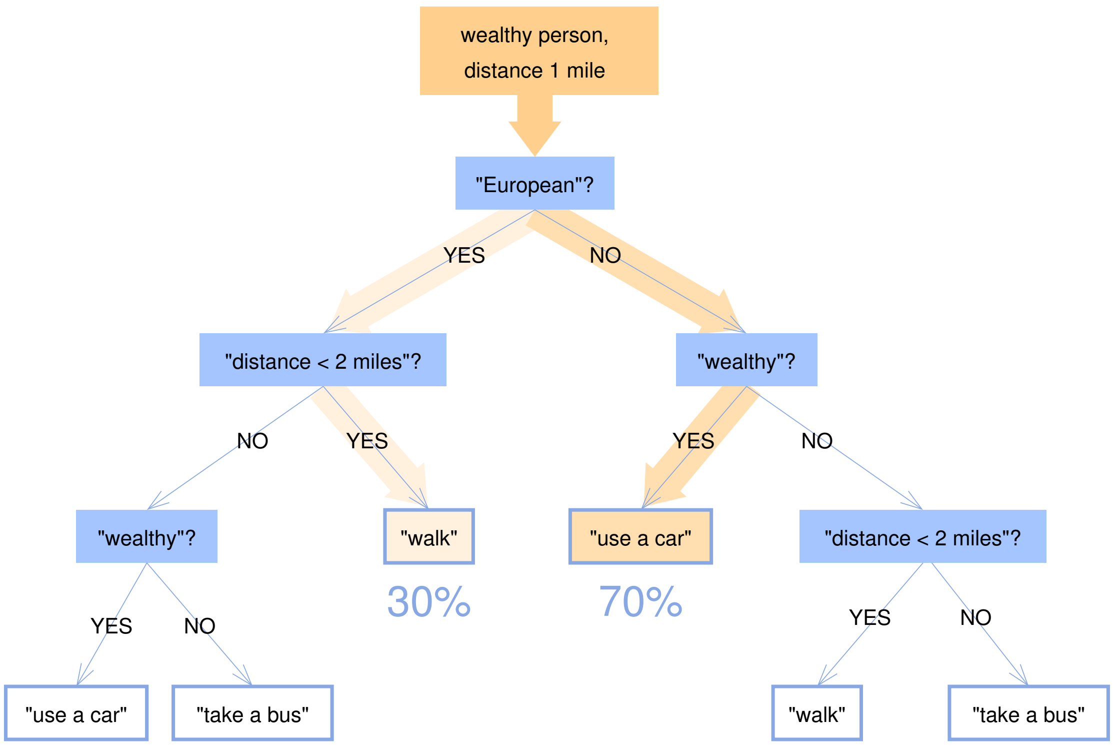


Figure 6.6: Missing nationality information. The data point arriving at the top node is sent both to its left and right child nodes with different weights depending on the frequency of "YES" and "NO" answers in the training set.

provides useful information —e.g., in marketing if a customer *does* not answer a question, we can assume that he is not very interested,— and "missing" can be treated as another value for a categorical variable. In other cases there is no significant information in the fact that a value is missing (e.g., if a sloppy salesman forgets to record some data about a customer). Decision trees provide a natural way to deal also with the second case. If an instance reaches a node and the question cannot be answered because data is lacking, one can ideally "split the instance into pieces", and send part of it down each branch in proportion to the number of training instances going down that branch. As Fig. 6.6 suggests, if 30% of training instances go left, an instance with a missing value at a decision node will be virtually split so that a portion of 0.3 will go left, and a portion of 0.7 will go right. When the different pieces of the instance eventually reach the leaves, the corresponding leaf output values can be averaged, or a distribution can be computed. The weights in the weighted average are proportional to the weights of the pieces reaching the leaves. In the example, the output value will be 0.3 times the output obtained by going left plus 0.7 times the output obtained by going right[2].

6.2 Democracy and decision forests

In the nineties, researchers discovered how using **democratic ensembles of learners** (e.g., generic "weak" classifiers with a performance slightly better than random guessing) yields greater accuracy and generalization[30, 5]. The analogy is with human society: in many cases setting up a **committee of experts** is a way to reach a decision with a better quality by either reaching consensus or by coming up with different proposals and voting (in other cases it is a way of postponing a decision, all analogies have their weaknesses). "Wisdom of crowds" [33] is a recent term to underline the positive effect of democratic decisions. For machine learning, outputs are combined by majority (in classification) or by averaging (in regression).

This seems particularly true for high dimensional data, with many irrelevant attributes, as often encountered in real-world applications. The topic is not as abstract as it looks: many relevant applications have been created, ranging from Optical Character Recognition with neural networks [5] to using ensembles of trees in input devices for game consoles[3] [12].

In order for a committee of experts to produce superior decisions you need them to be different (no *groupthink* effect, experts thinking in the same manner are useless) and of reasonable quality. Ways to obtain different trees are, for example, training them on different sets of examples, or training them with different randomized choices (in the

[2]Needless to say, a recursive call of the same routine with the left and right subtrees as argument is a way to obtain a very compact software implementation.

[3]Decision forests are used for human body tracking in the Microsoft Kinect sensor for the XBox gaming console.

human case, think about students following different courses on the same subject):

- Creating **different sets of training examples** from an initial set can be done with **bootstrap samples** (Section 5.3): given a standard training set D of size ℓ, bootstrap sampling generates new training sets by sampling from D uniformly and with replacement (some cases can be repeated). After ℓ samples, a training set is expected to have $1 - 1/e \approx 63.2\%$ of the unique examples of D, the rest being duplicates. Think about randomly throwing ℓ balls into ℓ boxes (recall Fig. 5.5): for large ℓ, only about 63.2% of the boxes will contain one or more balls. For each ball in the box, pick one version of the corresponding example. In each bootstrap training set, about one-third of the instances are left out. The application of bootstrapping to the creation of different sample sets is called **bagging** ("bootstrap aggregation"): different trees are built by using different random bootstrap samples and combined by averaging the output (for regression) or voting (for classification).

- **Different randomized choices** during training can be executed by limiting the choices when picking the optimal question at a node.

As an example of the differentiating methods just described, here's how they are implemented in **random decision forests** [21, 9]:

- each tree is trained on a bootstrap sample (i.e., with replacement) of the original data set;

- each time a leaf must be split, only a randomly chosen subset of the dimensions is considered for splitting. In the extreme case, only one random attribute (one dimension) is considered.

To be more precise, if d is the total number of input variables, each tree is constructed as follows: a small number d' of input variables, usually much smaller than d (in the extreme case just one), are used to determine the decision at a node. A bootstrap sample ("bag") is guiding the tree construction, while the cases which are not in the bag can be used to estimate the error of the tree. For each node of the tree, d' variables are randomly chosen on which to base the decision at that node. The best split based on these d' variables is calculated ("best" according to the chosen purity criterion, IG or GI). Each time a categorical variable is selected to split on at a node, one can select a random subset of the categories of the variable, and define a substitute variable equal to 1 when the categorical value of the variable is in the subset, and 0 outside. Each tree is fully grown and not pruned (as may be done in constructing a normal tree classifier).

By the above procedure we have actually populated a committee ("forest") where every expert ("tree") has received a different training, because they have seen a different set of examples (by bagging) and because they look at the problem from different points of view (they use different, randomly chosen criteria at each node). No expert is guaranteed to be very good at its job: the order at which each expert looks at the variables is far from being the greedy criterion that favors the most informative questions, thus an isolated tree is rather weak; however, as long as most experts are better than random classifiers, the majority (or weighted average) rule will provide reasonable answers.

Estimates of generalization when using bootstrap sampling can be obtained in a natural way during the training process: the **out-of-bag error** (error for cases not in the bootstrap sample) for each data point is recorded and averaged over the forest.

The relevance of the different variables (**feature or attribute ranking**) in decision forests can also be estimated in a simple manner. The idea is: if a categorical feature is important, randomly permuting its values should decrease the performance in a significant manner. After fitting a decision forest to the data, to derive the importance of the i-th feature after training, the values of the i-th feature are randomly permuted and the out-of-bag error is again computed on this perturbed data set. The difference in out-of-bag error before and after the permutation is averaged over all trees. The score is the percent increase in misclassification rate as compared to the out-of-bag rate with all variables intact. Features which produce large increase are ranked as more important than features which produce small increases.

The fact that many trees can be used (thousands are not unusual) implies that, for each instance to be classified or predicted, a very large number of output values of the individual trees are available. By collecting and analyzing the entire distribution of outputs of the many trees one can derive confidence bars for the regression or probabilities for classification. For example, if 300 trees predict “sun” and 700 trees predict “rain” we can come up with an estimate of a 70% probability of “rain.”

Gist

Simple “if-then” rules condense nuggets of information in a way which can be understood by human people. A simple way to avoid a confusing mess of possibly contradictory rules is to proceed with a hierarchy of questions (the most informative first) leading to an organized structure of simple successive questions called a **decision tree**.

Trees can be learned in a greedy and recursive manner, starting from the complete set of examples, picking a test to split it into two subsets which are as pure as possible, and repeating the procedure for the produced subsets. The recursive process terminates when the remaining subsets are sufficiently pure to conclude with a classification or an output value, associated with the leaf of the tree.

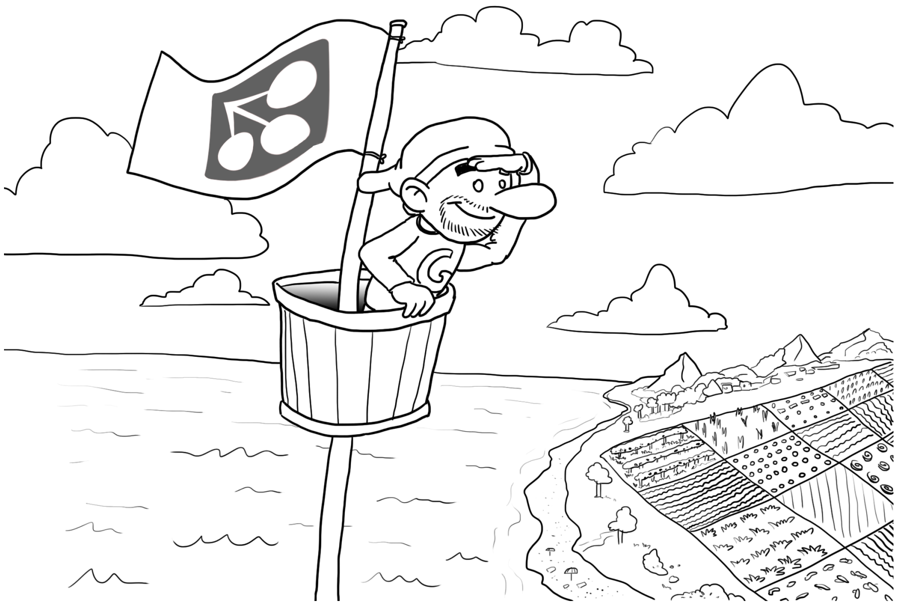
The abundance of memory and computing power permits training very large numbers of different trees. They can be fruitfully used as **decision forests** by collecting all outputs and averaging (regression) or voting (classification). Decision forests have various positive features: like all trees they naturally handle problems with more than two classes and with missing attributes, they provide a probabilistic output, probabilities and error bars, they generalize well to previously unseen data without risking over-training, they are fast and efficient thanks to their parallelism and reduced set of tests per data point.

A single tree casts a small shadow, hundreds of them can cool even the most torrid machine learning applications.

Chapter 7

Ranking and selecting features

I don't mind my eyebrows. They add... something to me. I wouldn't say they were my best feature, though. People tell me they like my eyes. They distract from the eyebrows.
(Nicholas Hoult)



Before starting to learn a model from the examples, one must be sure that the input data (input **attributes or features**) have sufficient information to predict the outputs. And after a model is built, one would like to get **insight** by identifying attributes which are influencing the output in a significant manner. If a bank investigates which customers are reliable enough to give them credit, knowing which factors are influencing the credit-worthiness in a positive or negative manner is of sure interest.

Feature selection, also known as attribute selection or variable subset selection, is the process of selecting a subset of relevant features to be used in model construction.

Feature selection is different from **feature extraction**, which considers the creation of new features as functions of the original features.

The issue of selecting and ranking features is far from trivial. Let's assume that a linear model is built:

$$y = w_1 x_1 + w_2 x_2 + ... + w_d x_d.$$

If a weight, say w_j, is zero you can easily conclude that the corresponding feature x_j does not influence the output. But let's remember that numbers are not exact in computers and that examples are "noisy" (affected by measurement errors) so that getting zero is a very low-probability event indeed. Considering the non-zero weights, can you conclude that the largest weights (in magnitude) are related to the most informative and significant features?

Unfortunately you cannot. This has to do with how inputs are "scaled". If weight w_j is very large when feature x_j is measured in kilometers, it will jump to a very small value when measuring the same feature in millimeters (if we want the same result, the multiplication $w_j \times x_j$ has to remain constant when units are changed). An aesthetic change of units for our measurements immediately changes the weights. The value of a feature cannot depend on your choice of units, and therefore we cannot use the weight magnitude to assess its importance.

Nonetheless, the weights of a linear model can give *some* robust information if the inputs are *normalized*, pre-multiplied by constant factors so that the range of typical values is the same, for example the approximate range is from 0 to 1 for all input variables. If selecting features is already complicated for linear models, expect an even bigger complication for nonlinear ones.

7.1 Selecting features: the context

Let's now come to some definitions for the case of a classification task (Fig. 3.1 on page 17) where the output variable c identifies one among N_c classes and the input variable $\boldsymbol{x}$ has a finite set of possible values. For example, think about predicting whether a mushroom is edible (class 1) or poisonous (class 0). Among the possible features extracted from the data one would like to obtain a highly informative set, so that the classification problem starts from sufficient information, and only the actual construction of the classifier is left.

At this point you may ask why one is not using the entire set of inputs instead of a subset of features. After all, some information *shall* be lost if we eliminate some input data. True, but the **curse of dimensionality** holds here: if the dimension of the input is too large, the learning task becomes unmanageable. Think for example about the difficulty of estimating probability distributions from samples in very high-dimensional spaces. This is the standard case in "big data" text and web mining applications, in which each document can be characterized by tens of thousands dimensions (a dimension for each possible word in the vocabulary), so that vectors corresponding to the documents can be very sparse in the vector space.

Heuristically, one aims at **a small subset of features, possibly close to the smallest possible, which contains sufficient information to predict the output**, with redundancy eliminated. In this way not only memory usage is reduced but generalization can be improved because irrelevant features and irrelevant parameters are eliminated. Last but not least, your human understanding of the model becomes easier for smaller models.

Think about recognizing digits from written text. If the text is written on colored paper, maintaining the color of the paper as a feature will make the learning task more difficult and worsen generalization if paper of different color is used to test the system.

Feature selection is a typical problem with many possible solutions and without formal guarantees of optimality. No simple recipe is available, although one identifies some classes of methods.

First, the **designer intuition and existing knowledge** should be applied. For example, if your problem is to recognize handwritten digits, images should be scaled and normalized (a "five" is still a five even if enlarged, reduced, stretched, rotated...), and clearly irrelevant features like the color should be eliminated from the beginning.

Second, one needs a way to **estimate the relevance or discrimination power of the individual features** and then one can proceed in a bottom-up or top-down manner, in some cases by directly testing the tentative feature set through repeated runs of the considered training model. The value of a feature is related to a model-construction method, and some evaluation techniques depend on the method. One identifies three classes of methods.

Wrapper methods are built "around" a specific predictive model. Each feature subset is used to train a model. The generalization performance of the trained model gives the score for that subset. Wrapper methods are computationally intensive, but usually provide the best performing feature set for the specific model.

Filter methods use a proxy measure instead of the error rate to score a feature subset. Common measures include the Mutual Information and the correlation coefficient. Many filters provide a feature ranking rather than an explicit best feature subset.

Embedded methods perform feature selection as part of the model construction process. An example of this approach is the LASSO method for constructing a linear model, which penalizes the regression coefficients, shrinking many of them to zero, so that the corresponding features can be eliminated. Another approach is Recursive Feature Elimination, commonly used with Support Vector Machines to repeatedly construct a model and remove features with low weights.

By combining filtering with a wrapper method one can proceed in a bottom-up or top-down manner. In a **bottom-up method of greedy inclusion** one gradually inserts

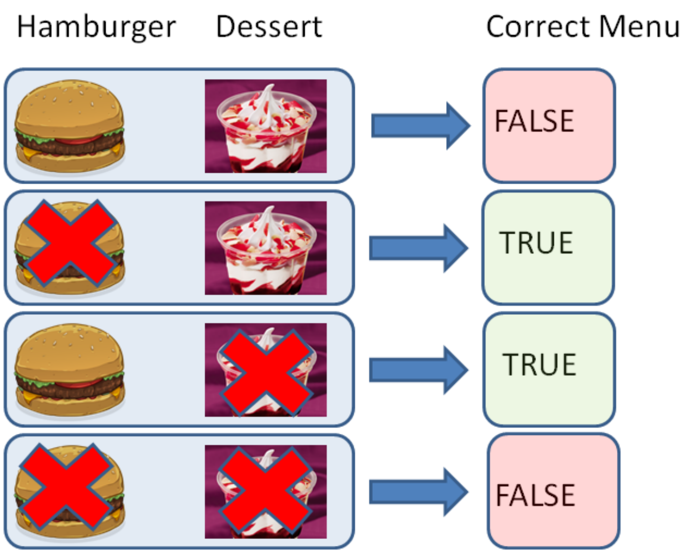


Figure 7.1: A classifier with two binary inputs and one output. Single features in isolation are not informative, both input features are needed and sufficient for a correct classification.

the ranked features in the order of their individual discrimination power and checks the effect on output error reduction though a validation set. The heuristically optimal number of features is determined when the output error measured on the validation set stops decreasing. In fact, if many more features are added beyond this point, the error may remain stable, or even gradually increase because of *over-fitting*.

In a **top-down truncation** method one starts with the complete set of features and progressively eliminates features while searching for the optimal performance point (always checking the error on a suitable validation set).

A word of caution for filter methods. Let's note that **measuring individual features in isolation will discard mutual relationships** and therefore the result will be approximated. It can happen that two features in isolation have no relevant information and are discarded, even if their *combination* would allow perfect prediction of the output, think about realizing an *exclusive OR* function of two inputs.

As a example of the exclusive OR, imagine that the class to recognize is *CorrectMenu(hamburger, dessert)*, where the two variables *hamburger* and *dessert* have value 1 if present in a menu, 0 otherwise (Fig. 7.1). To get a proper amount of calories in a fast food you need to get either a hamburger or a dessert but not both. The individual presence or absence of a hamburger (or of a dessert) in a menu will not be related to classifying a menu as correct or not. But it would not be wise to eliminate one or both inputs because their *individual* information is not related to the output classification. You need to keep and read both attributes to correctly classify your meal! The toy example can be generalized: any diet expert will tell you that what matters are not individual quantities but an overall balanced combination.

Now that the context is clear, let's consider some example **proxy measures of the discrimination power of individual features**.

7.2 Correlation coefficient

Let Y be the random variable associated with the output classification, let $\Pr(y)$ ($y \in Y$) be the probability of y being its outcome; X_i is the random variable associated with the input variable x_i, and X is the input vector random variable, whose values are x.

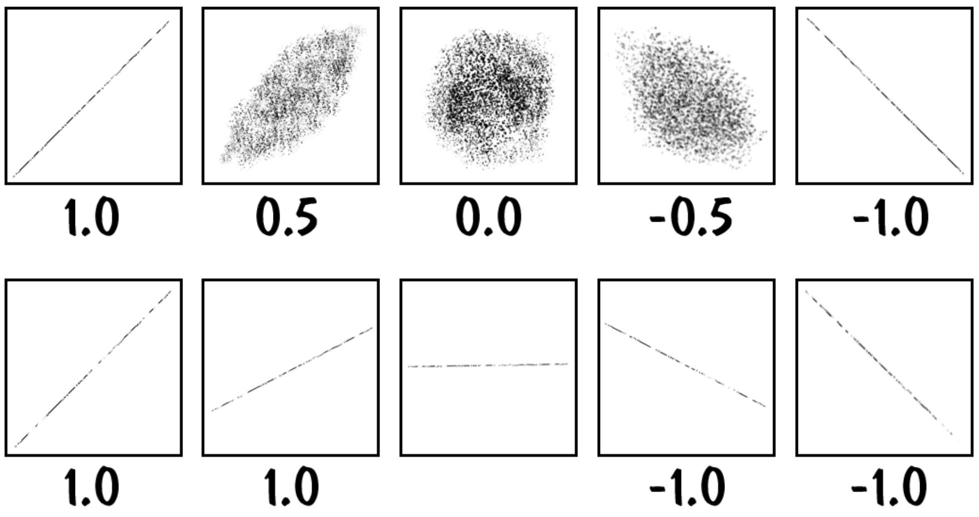


Figure 7.2: Examples of data distributions and corresponding correlation values. Remember that values are divided by the standard deviation, this is why the linear distributions of the bottom plots all have the same maximum correlation coefficient (positive 1 or negative -1).

The most widely used measure of **linear relationship between numeric variables** is the Pearson product-moment **correlation coefficient**, which is obtained by dividing the covariance of the two variables by the product of their standard deviations. In the above notation, the correlation coefficient $\rho_{X_i,Y}$ between the i-th input feature X_i and the classifier's outcome Y, with expected values μ_{X_i} and μ_Y and standard deviations σ_{X_i} and σ_Y, is defined as:

$$\rho_{X_i,Y} = \frac{\mathrm{cov}[X_i, Y]}{\sigma_{X_i}\sigma_Y} = \frac{E[(X_i - \mu_{X_i})(Y - \mu_Y)]}{\sigma_{X_i}\sigma_Y}; \tag{7.1}$$

where E is the expected value of the variable and cov is the covariance. After simple transformations one obtains the equivalent formula:

$$\rho_{X_i,Y} = \frac{E[X_iY] - E[X_i]E[Y]}{\sqrt{E[X_i^2] - E^2[X_i]}\,\sqrt{E[Y^2] - E^2[Y]}}. \tag{7.2}$$

The division by the standard deviations makes the correlation coefficient independent of units (e.g., measuring in kilometers or millimeters will produce the same result). The correlation value varies from -1 to 1. Correlation close to 1 means increasing linear relationship (an increase of the feature value x_i relative to the mean is usually accompanied by an increase of the outcome y), close to -1 means a decreasing linear relationship. The closer the coefficient is to zero, the weaker the correlation between the variables, for example the plot of (x_i, y) points looks like an isotropic cloud around the expected values, without an evident direction, as shown in Fig. 7.2.

As mentioned before, statistically independent variables have zero correlation, but zero correlation does *not* imply that the variables are independent. The correlation coefficient detects only *linear* dependencies between two variables: it may well be that a variable has full information and actually determines the value of the second, as in the case that $y = f(x_i)$, while still having zero correlation.

The normal suggestion for this and other feature ranking criteria is not to use them blindly, but supported by experimental results on classification performance on test (validation) data, as in wrapper techniques.

7.3 Correlation ratio

In many cases, the desired outcome of our learning algorithm is categorical (a “yes/no” answer or a limited set of choices). The correlation coefficient assumes that the output is numeric, thus it is not applicable to the categorical case. In order to sort out general dependencies, the *correlation ratio* method can be used.

The basic idea behind the correlation ratio is to partition the sample feature vectors into classes according to the observed outcome. If a feature is significant, then it should be possible to identify at least one outcome class where the feature’s average value is significantly different from the average on all classes, otherwise that component would not be useful to discriminate any outcome. We are therefore measuring a **relationship between a numeric input and a categorical output**.

Suppose that one has a set of ℓ sample feature vectors, possibly collected during previous stages of the algorithm that one is trying to measure. Let ℓ_y be the number of times that outcome $y \in Y$ appears, so that one can partition the sample feature vectors by their outcome:

$$\forall y \in Y \qquad S_y = \big((x_{jy}^{(1)}, \dots, x_{jy}^{(n)}); j = 1, \dots, \ell_y\big).$$

In other words, the element $x_{jy}^{(i)}$ is the i-th component (feature) of the j-th sample vector among the ℓ_y samples having outcome y. Let us concentrate on the i-th feature from all sample vectors, and compute its average within each outcome class:

$$\forall y \in Y \qquad \bar{x}_y^{(i)} = \frac{1}{\ell_y} \sum_{j=1}^{\ell_y} x_{jy}^{(i)},$$

and the overall average:

$$\bar{x}^{(i)} = \frac{1}{\ell} \sum_{y \in Y} \sum_{j=1}^{\ell_y} x_{jy}^{(i)} = \frac{1}{\ell} \sum_{y \in Y} \ell_y \bar{x}_y^{(i)}.$$

Finally, the **correlation ratio** between the i-th component of the feature vector and the outcome is given by

$$\eta_{X_i,Y}^2 = \frac{\sum_{y \in Y} \ell_y (\bar{x}_y^{(i)} - \bar{x}^{(i)})^2}{\sum_{y \in Y} \sum_{j=1}^{\ell_y} (x_{jy}^{(i)} - \bar{x}^{(i)})^2}.$$

If the relationship between values of the i-th feature component and values of the outcome is linear, then both the correlation coefficient and the correlation ratio are equal to the slope of the dependence:

$$\eta_{X_i,Y}^2 = \rho_{X_i,C}^2.$$

In all other cases, the correlation ratio can capture nonlinear dependencies.

7.4 Chi-square test

Let's again consider a two-way classification problem and a single feature with a binary value. For example, in text mining, the feature can express the presence/absence of a specific term (keyword) t in a document and the output can indicate if a document is about programming languages or not. We are therefore evaluating a **relationship between two categorical features**.

One can start by deriving four counters $\text{count}_{c,t}$, counting in how many cases one has (has not) the term t is a document which belongs (does not belong) to the given class. For example $\text{count}_{0,1}$ counts for class=0 and presence of term t, $\text{count}_{0,0}$ counts for class=0 and absence of term t... Then it is possible to estimate probabilities by dividing the counts by the total number of cases n.

In the *null hypothesis* that the two events "occurrence of term t" and "document of class c" are independent, the expected value of the above counts for joint events are obtained by *multiplying* probabilities of individual events. For example $E(\text{count}_{0,1}) = n \cdot \Pr(\text{class} = 0) \cdot \Pr(\text{term } t \text{ is present})$.

If the count deviates from the one expected for two independent events, one can conclude that the two events are *dependent*, and that therefore the feature *is* significant to predict the output. All one has to do is to check if the *deviation is sufficiently large that it cannot happen by chance*. A statistically sound manner to test is by **statistical hypothesis testing**.

A statistical hypothesis test is a method of making statistical decisions by using experimental data. In statistics, a result is called **statistically significant if it is unlikely to have occurred by chance**. The phrase "test of significance" was coined around 1925 by Ronald Fisher, a genius who created the foundations for modern statistical science.

Hypothesis testing is sometimes called **confirmatory data analysis**, in contrast to exploratory data analysis. Decisions are almost always made by using *null-hypothesis tests*; that is, ones that answer the question: Assuming that the null hypothesis is true, what is the probability of observing a value for the test statistic that is at least as extreme as the value that was actually observed?

In our case one measures the χ^2 value:

$$\chi^2 = \sum_{c,t} \frac{[\text{count}_{c,t} - n \cdot \Pr(\text{class} = c) \cdot \Pr(\text{term} = t)]^2}{n \cdot \Pr(\text{class} = c) \cdot \Pr(\text{term} = t)}. \tag{7.3}$$

The larger the χ^2 value, the lower the belief that the independence assumption is supported by the observed data. The probability of a specific value happening by chance can be calculated by standard statistical formulas if one desires a quantitative value.

For feature ranking no additional calculation is needed and one is satisfied with the crude value: the best features according to this criterion are the ones with larger χ^2 values. They are deviating more from the independence assumption and therefore probably dependent.

7.5 Entropy and mutual information

The qualitative criterion of "informative feature" can be made precise in a statistical way with the notion of **mutual information** (MI).

An output distribution is characterized by an *uncertainty* which can be measured from the probability distribution of the outputs. The theoretically sound way to measure the uncertainty is with the **entropy**, see below for the precise definition. Now, knowing a specific input value x, the uncertainty in the output can decrease. The amount by which the uncertainty in the output decreases after the input is known is termed **mutual information**.

If the mutual information between a feature and the output is zero, knowledge of the input does not reduce the uncertainty in the output. In other words, the selected feature cannot be used (in isolation) to predict the output - no matter how sophisticated our model. The MI measure between a vector of input features and the output (the desired prediction) is therefore very relevant to identify promising (informative) features. Its use in feature selection is pioneered in [3].

In information theory **entropy**, measuring the statistical uncertainty in the output class (a random variable), is defined as:

$$H(Y) = -\sum_{y \in Y} \Pr(y) \log \Pr(y). \tag{7.4}$$

Entropy quantifies the average information, measured in bits, used to specify which event occurred (Fig. 6.5). It is also used to quantify how much a message can be compressed without losing information[1].

Let us now evaluate the impact that the i-th input feature x_i has on the classifier's outcome y. The entropy of Y after knowing the input feature value ($X_i = x_i$) is:

$$H(Y|x_i) = -\sum_{y \in Y} \Pr(y|x_i) \log \Pr(y|x_i),$$

where $\Pr(y|x_i)$ is the conditional probability of being in class y given that the i-th feature has value x_i.

Finally, the *conditional entropy* of the variable Y is the expected value of $H(Y|x_i)$ over all values $x_i \in X_i$ that the i-th feature can have:

$$H(Y|X_i) = E_{x_i \in X_i}\big[H(Y|x_i)\big] = -\sum_{x_i \in X_i} \Pr(x_i) H(Y|x_i). \tag{7.5}$$

The conditional entropy $H(Y|X_i)$ is always less than or equal to the entropy $H(Y)$. It is equal if and only if the i-th input feature and the output class are *statistically independent*, i.e., the joint probability $\Pr(y, x_i)$ is equal to $\Pr(y)\Pr(x_i)$ for every $y \in Y$ and $x_i \in X_i$ (note: this definition does not talk about linear dependencies). The amount by which the uncertainty decreases is by definition the *mutual information* $I(X_i; Y)$ between variables X_i and Y:

$$I(X_i; Y) = I(Y; X_i) = H(Y) - H(Y|X_i). \tag{7.6}$$

An equivalent expression which makes the symmetry between X_i and Y evident is:

$$I(X_i; Y) = \sum_{y, x_i} \Pr(y, x_i) \log \frac{\Pr(y, x_i)}{\Pr(y)\Pr(x_i)}. \tag{7.7}$$

[1] Shannon's source coding theorem shows that, in the limit, the average length of the shortest possible representation to encode the messages in a binary alphabet is their entropy. If events have the same probability, no compression is possible. If the probabilities are different one can *assign shorter codes to the most probable events*, therefore reducing the overall message length. This is why "zip" tools successfully compress meaningful texts with different probabilities for words and phrases, but have difficulties to compress quasi-random sequences of digits like JPEG or other efficiently encoded image files.

Although very powerful in theory, estimating mutual information for a high-dimensional feature vector starting from labeled samples is not a trivial task. A heuristic method which uses only mutual information between individual features and the output is presented in [3].

Let's stress that the mutual information is different from the correlation. A feature can be informative even if not linearly correlated with the output, and the mutual information measure does not even require the two variables to be quantitative. Remember that a nominal categorical variable is one that has two or more categories, but there is no intrinsic ordering to the categories. For example, gender is a nominal variable with two categories (male and female) and no intrinsic ordering. Provided that you have sufficiently abundant data to estimate it, there is not a better way to measure information content than by Mutual Information.

Gist

Reducing the number of input attributes used by a model, while keeping roughly equivalent performance, has many advantages: smaller model size and better human understandability, faster training and running times, possible higher generalization.

It is difficult to rank individual features without considering the specific modeling method and their mutual relationships. Think about a detective (in this case the classification to reach is between "guilty" and "innocent") intelligently combining multiple clues and avoiding confusing evidence. Ranking and filtering is only a first heuristic step and needs to be validated by trying different feature sets with the chosen method, "wrapping" the method with a feature selection scheme.

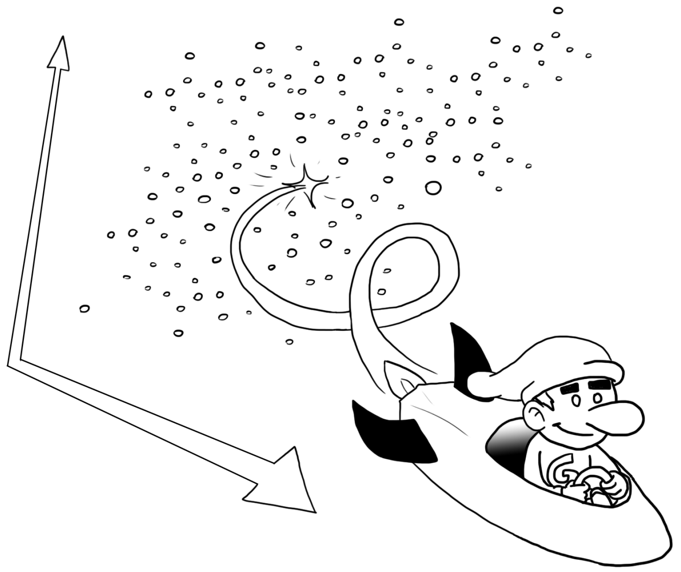
A short recipe is: trust the correlation coefficient only if you have reasons to suspect *linear* relationships, otherwise other correlation measures are possible, in particular the correlation ratio can be computed even if the outcome is not quantitative. Use chi-square to identify possible dependencies between inputs and outputs by estimating probabilities of individual and joint events. Finally, use the powerful mutual information to estimate arbitrary dependencies between qualitative or quantitative features, but be aware of possible overestimates when few examples are present.

As an exercise, pick your favorite Sherlock Holmes story and identify which feature (clue, evidence) selection technique he used to capture and expose a culprit and to impress his friend Watson.

Chapter 8

Specific nonlinear models

He who would learn to fly one day must first learn to stand and walk and run and climb and dance; one cannot fly into flying.
(Friedrich Nietzsche)



In this chapter we continue along our path from linear to nonlinear models. In order to avoid the vertigo caused by an abrupt introduction of the most general and powerful models, we start by gradual modifications of the linear model, first to make it suitable for predicting probabilities (**logistic regression**), then by making the linear models *local* and giving more emphasis to the closest examples, in a kind of smoothed version of K nearest neighbors (**locally-weighted linear regression**), finally by selecting a subset of inputs via appropriate constraints on the weights (**LASSO**).

Latest chapter revision: November 14, 2013

After this preparatory phase, in the next chapters we will be ready to enter the holy of holies of flexible nonlinear models for arbitrary smooth input-output relationships like Multi-Layer Perceptrons (MLP) and Support Vector Machines (SVM).

8.1 Logistic regression

In statistics, logistic regression is used for **predicting the probability of the outcome of a categorical variable** from a set of recorded past events. For example, one starts from data about patients which can have a heart disease (disease "yes" or "no" is the categorical output variable) and wants to predict the probability that a new patient has the heart disease. The name is somewhat misleading, it really is a technique for classification, not regression. But classification is obtained through an estimate of the probability, henceforth the term "regression." Frequently the output is binary, that is, the number of available categories is two.

The problem with using a linear model is that the output value is not bounded: we need to bound it to be between zero and one. In logistic regression most of the work is done by a linear model, but a **logistic function** (Fig. 8.1) is used to transform the output of a linear predictor to obtain a value between zero and one, which can be interpreted as a probability. The probability can then be compared with a threshold to reach a classification (e.g., classification "yes" if output probability greater than 0.5).

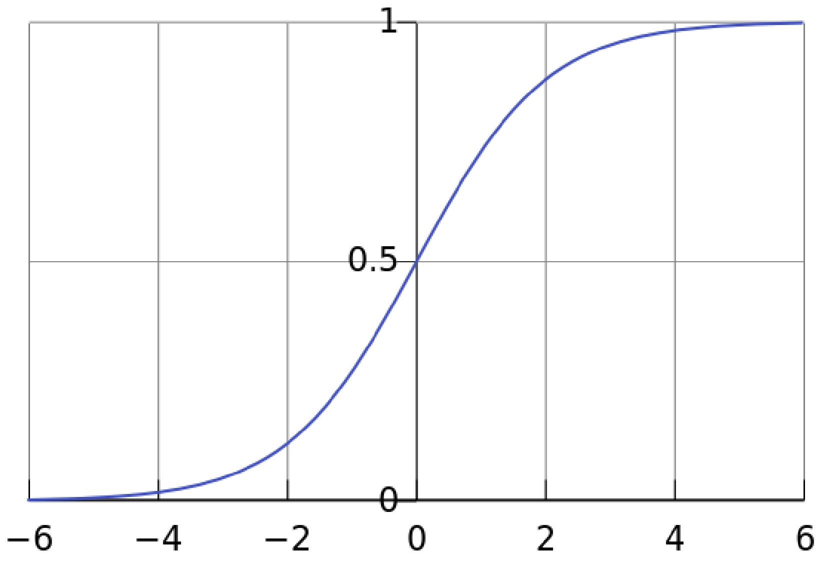


Figure 8.1: A logistic function transforms input values into an output value in the range 0-1, in a smooth manner. The output of the function can be interpreted as a probability.

A logistic function or logistic curve is a common **sigmoid function**[1]. A simple logistic function is defined by the formula

$$P(t) = \frac{1}{1+e^{-t}},$$

where e is Euler's number and the variable t might be thought of as time or, in our case, the output of the linear model, remember Equation (4.1) at page 31:

$$P(\boldsymbol{x}) = \frac{1}{1+e^{-(\boldsymbol{w}^T\boldsymbol{x})}}.$$

Remember that a constant value w_0 can also be included in the linear model $\boldsymbol{w}$, provided that an artificial input x_0 always equal to 1 is added to the list of input values.

Let's see which function is to be maximized in this case. The best values for the weights of the linear transformation are determined by **maximum likelihood estimation**, i.e., by maximizing the probability of getting the output values which were actually *obtained* on the given labeled examples. The probabilities of individual independent cases are multiplied. Let y_i be the observed output (1 or 0) for the corresponding input $\boldsymbol{x}_i$. If $\Pr(y = 1|\boldsymbol{x}_i)$ is the probability obtained by the model, the probability of obtaining the measured output value y_i is $\Pr(y = 1|\boldsymbol{x}_i)$ if the correct label is 1, $\Pr(y = 0|\boldsymbol{x}_i) = 1 - \Pr(y = 1|\boldsymbol{x}_i)$ if the correct label is 0. All factors need to be multiplied to get the overall probability of obtaining the complete series of examples. It is customary to work with logarithms, so that the multiplication of the factors (one per example) becomes a summation:

$$\text{LogLikelihood}(\boldsymbol{w}) = \sum_{i=1}^{\ell}\Big\{y_i \ln \Pr(y = y_i|\boldsymbol{x}_i, \boldsymbol{w}) + (1-y_i)\ln(1-\Pr(y = y_i|\boldsymbol{x}_i, \boldsymbol{w}))\Big\}.$$

The dependence of $\Pr$ on the coefficients (weights) $\boldsymbol{w}$ has been made explicit.

Given the nonlinearities in the above expression, it is not possible to find a closed-form expression for the weight values that maximize the likelihood function: an iterative process must be used instead, for example gradient descent. This process begins with a tentative solution $\boldsymbol{w}_{\text{start}}$, revises it slightly by moving in the direction of the negative gradient to see if it can be improved, and repeats this revision until improvement is minute, at which point the process is said to have converged.

As usual, in ML one is interested in maximizing the generalization. The above minimization process can - and should - be stopped early, when the estimated generalization results measured by a validation set are maximal.

[1]The term "logistic" was given when this function was introduced to study population growth. In a population, the rate of reproduction is proportional to both the existing population and the amount of available resources. The available resources decrease when the population grows and become zero when the population reaches the "carrying capacity" of the system. The initial stage of growth is approximately exponential; then, as saturation begins, the growth slows, and at maturity, growth stops.

8.2 Locally-Weighted Regression

In Section 4.1 we have seen how to determine the coefficients of a linear dependence. The K Nearest Neighbors method in Chapter 2 predicts the output for a new input by comparing the new input with the closest old (and labeled) ones, giving as output either the one of the closest stored input, or some simple combination of the outputs of a selected set of closest neighbors.

In this section we consider a similar approach to obtain the output from a *linear* combination of the outputs of the closest neighbors. But we are less cruel in eliminating all but the K closest neighbors. The situation is smoothed out: instead of selecting a set of K winners we **gradually reduce the role of examples on the prediction based on their distance** from the case to predict.

The method works if a relationship based on experimental data can be safely assumed to be **locally linear**, although the overall global dependence can be quite complex. If the model must be evaluated at different points, then we can still use linear regression, provided that training points *near* the evaluation point are considered "more important" than distant ones. We encounter a very general principle here: in learning (natural or automated) similar cases are usually deemed more relevant than very distant ones. For example, case-based reasoning solves new problems based on the solutions of similar past problems.

Locally Weighted Regression is a *lazy* memory-based technique, meaning that all points and evaluations are stored and a specific model is built *on-demand* only when a specified query point demands an output.

To predict the outcome of an evaluation at a point $\boldsymbol{q}$ (named a *query point*), linear regression is applied to the training points. To enforce locality in the determination of the regression parameters (near points are more relevant), to each sample point is assigned a *weight* that decreases with its distance from the query point. Note that, in the neural networks community, the term "weight" refers to the parameters of the model being computed by the training algorithm, while, in this case, it measures the importance of each training sample. To avoid confusion we use the term *significance* and the symbol s_i (and S for the diagonal matrix used below) for this different purpose.

In the following we shall assume, as explained in Section 4.1, that a constant 1 is attached as entry 0 to all input vectors $\boldsymbol{x}_i$ to include a constant term in the regression, so that the dimensionality of all equations is actually $d + 1$.

The weighted version of *least squares* fit aims at minimizing the following weighted error (compare with equation (4.2), where weights are implicitly uniform):

$$\text{error}(\boldsymbol{w}; s_1, \dots, s_n) = \sum_{i=1}^{\ell} s_i(\boldsymbol{w}^T \cdot \boldsymbol{x}_i - y_i)^2. \tag{8.1}$$

From the viewpoint of the spring analogy discussed in Section 4.1, the distribution of

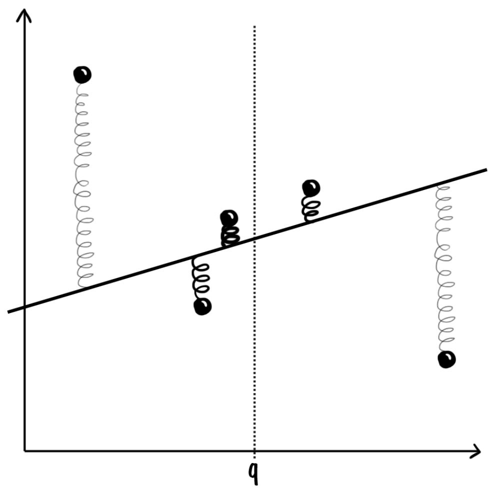


Figure 8.2: The spring analogy for the weighted least squares fit (compare with Fig. 4.6). Now springs have different elastic constants, thicker meaning harder, so that their contribution to the overall potential energy is weighted. In the above case, harder springs are for points closer to the query point $\boldsymbol{q}$.

different weights to sample points corresponds to using springs with a different elastic constant (strength), as shown in Fig. 8.2. Minimization of equation (8.1) is obtained by requiring its gradient with respect to $\boldsymbol{w}$ to be equal to zero, and we obtain the following solution:

$$\boldsymbol{w}^* = (X^T S^2 X)^{-1} X^T S^2 \boldsymbol{y}; \tag{8.2}$$

where $S = \text{diag}(s_1, \ldots, s_d)$, while X and $\boldsymbol{y}$ are defined as in Equation 4.5, page 37. Note that equation (8.2) reduces to equation (4.5) when all weights are equal.

It makes sense to assign significance values to the stored examples according to their distance from the query point (”nature does not make jumps”). A common function used to set the relationship between significance and distance is

$$s_i = \exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{q}\|^2}{W_K}\right);$$

where W_K is a parameter measuring the “kernel width,” i.e. the sensitivity to distant sample points; if the distance is much larger than W_K the significance rapidly goes to zero.

An example is given in Fig. 8.3 (top), where the model must be evaluated at query point $\boldsymbol{q}$. Sample points x_i are plotted as circles, and their significance s_i decreases with the distance from $\boldsymbol{q}$ and is represented by the interior shade, black meaning highest significance. The linear fit (solid line) is computed by considering the significance of the various points and is evaluated at $\boldsymbol{q}$ to provide the model's value at that point. The

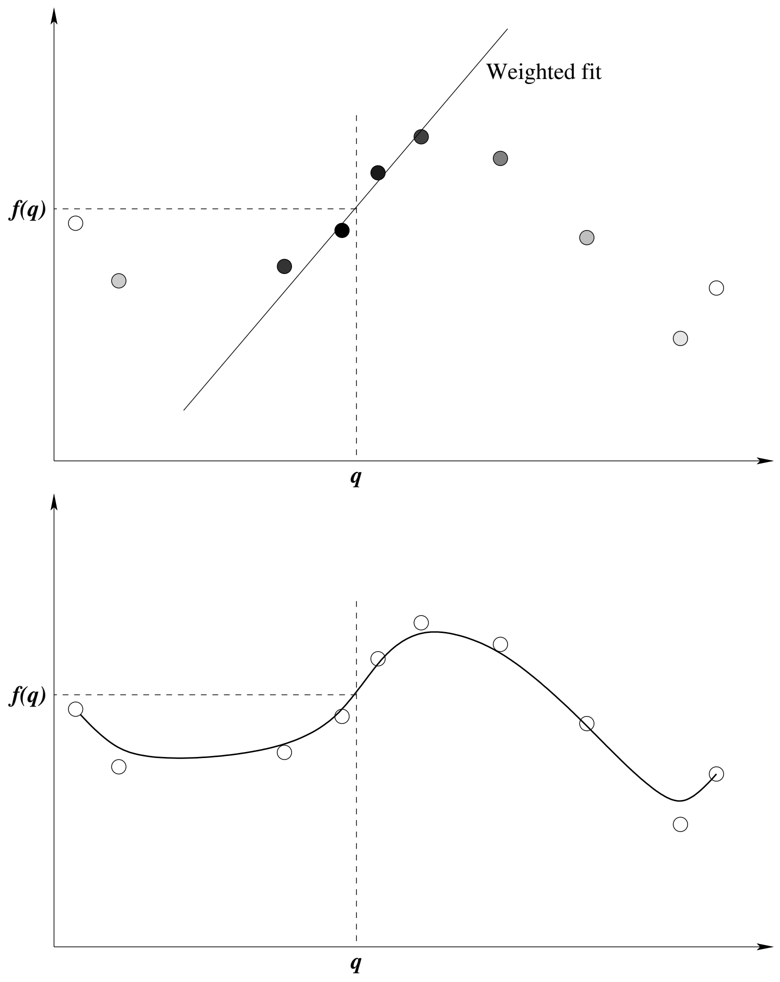


Figure 8.3: Top: evaluation of LWR model at query point $\boldsymbol{q}$, sample point significance is represented by the interior shade. Bottom: Evaluation over all points, each point requires a different linear fit.

significance of each sample point and the subsequent linear fit are recomputed for each query point, providing the curve shown in Fig. 8.3 (bottom).

8.2.1 Bayesian LWR

Up to this point, no assumption has been made on the *prior* probability distribution of coefficients to be determined. In some cases some more information is available about the task which can conveniently be added through a prior distribution.

Bayesian Locally Weighted Regression, denoted as B-LWR, is used if prior information about what values the coefficients should have can be specified when there is not enough data to determine them. The usual power of Bayesian techniques derives from the *explicit* specification of the modeling assumptions and parameters (for example, a *prior distribution* can model our initial knowledge about the function) and the possibility to model not only the expected values but entire probability distributions. For example *confidence intervals* can be derived to quantify the uncertainty in the expected values.

The prior assumption on the distribution of coefficients w, leading to Bayesian LWR, is that it is distributed according to a multivariate Gaussian with zero mean and covariance matrix Σ, and the prior assumption on σ is that $1/\sigma^2$ has a Gamma distribution with k and θ as the shape and scale parameters. Since one uses a weighted regression, each data point and the output response are weighted using a Gaussian weighing function. In matrix form, the weights for the data points are described in $\ell \times \ell$ diagonal matrix $S = \text{diag}(s_1, \ldots, s_\ell)$, while $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_\ell)$ contains the prior variance for the $\boldsymbol{w}$ distribution.

The local model for the query point $\boldsymbol{q}$ is predicted by using the marginal posterior distribution of $\boldsymbol{w}$ whose mean is estimated as

$$\bar{\boldsymbol{w}} = (\Sigma^{-1} + X^T S^2 X)^{-1} (X^T S^2 \boldsymbol{y}). \tag{8.3}$$

Note that the removal of prior knowledge corresponds to having infinite variance on the prior assumptions, therefore Σ^{-1} becomes null and equation (8.3) reduces to equation (8.2). The matrix $\Sigma^{-1} + X^T S^2 X$ is the weighted covariance matrix, supplemented by the effect of the $\boldsymbol{w}$ priors. Let's denote its inverse by $V_{\boldsymbol{w}}$. The variance of the Gaussian noise based on ℓ data points is estimated as

$$\sigma^2 = \frac{2\theta + (\boldsymbol{y}^T - \boldsymbol{w}^T X^T) S^2 \boldsymbol{y}}{2k + \sum_{i=1}^{\ell} s_i^2}.$$

The estimated covariance matrix of the $\boldsymbol{w}$ distribution is then calculated as

$$\sigma^2 V_{\boldsymbol{w}} = \frac{(2\theta + (\boldsymbol{y}^T - \boldsymbol{w}^T X^T) S^2 \boldsymbol{y})(\Sigma^{-1} + X^T S^2 X)}{2k + \sum_{i=1}^{\ell} s_i^2}.$$

The degrees of freedom are given by $k + \sum_{i=1}^{\ell} s_i^2$. Thus the predicted output response for the query point $\boldsymbol{q}$ is

$$\hat{y}(\boldsymbol{q}) = \boldsymbol{q}^T \bar{\boldsymbol{w}},$$

while the variance of the mean predicted output is calculated as:

$$Var(\hat{y}(\boldsymbol{q})) = \boldsymbol{q}^T V_{\boldsymbol{w}} \boldsymbol{q} \sigma^2. \tag{8.4}$$

8.3 LASSO to shrink and select inputs

When considering linear models, *ridge regression* was mentioned as a way to make the model more stable by penalizing large coefficients in a quadratic manner, as in equation (4.7).

Ordinary least squares estimates often have low bias but large variance; prediction accuracy can sometimes be improved by shrinking or setting to 0 some coefficients. By doing so we sacrifice a little bias to reduce the variance of the predicted values and hence may improve the overall prediction accuracy. The second reason is **interpretation**. With a large number of predictors (input variables), we often would like to determine a smaller subset that exhibits the strongest effects. The two standard techniques for improving the estimates, feature subset selection and ridge regression, have some drawbacks. Subset selection provides interpretable models but can be extremely variable because it is a discrete process - input variables (a.k.a. regressors) are either retained or dropped from the model. Small changes in the data can result in very different models being selected and this can reduce prediction accuracy. Ridge regression is a continuous process that shrinks coefficients and hence is more stable: however, it does not set any coefficients to 0 and hence does not give an easily interpretable model. The work in [34] proposes a new technique, called the **LASSO**, "least absolute shrinkage and selection operator." It shrinks some coefficients and sets others to 0, and hence tries to retain the good features of both subset selection and ridge regression.

LASSO uses the constraint that $\|\boldsymbol{w}\|_1$, the sum of the absolute values of the weights (the L_1 norm of the parameter vector), is no greater than a given value. LASSO minimizes the residual sum of squares subject to the sum of the absolute value of the coefficients being less than a constant. By a standard trick to transform a constrained optimization problem into an unconstrained one through Lagrange multipliers, this is equivalent to an unconstrained minimization of the *least squares* penalty with $\lambda\|\boldsymbol{w}\|_1$ added:

$$\text{LASSOerror}(\boldsymbol{w};\lambda) = \sum_{i=1}^{\ell} (\boldsymbol{w}^T \cdot \boldsymbol{x}_i - y_i)^2 + \lambda \sum_{j=0}^{d} |w_j|. \tag{8.5}$$

One of the prime differences between LASSO and ridge regression is that in ridge regression, as the penalty is increased, all parameters are reduced while still remaining *non-zero*, while in LASSO, increasing the penalty will cause more and more of the parameters to be driven to *zero*. The inputs corresponding to weights equal to zero can be eliminated, leading to models with fewer inputs (**sparsification** of inputs) and therefore more interpretable. Fewer nonzero parameter values effectively reduce the number of variables upon which the given solution is dependent. In other words, LASSO is an embedded method to perform **feature selection as part of the model construction**

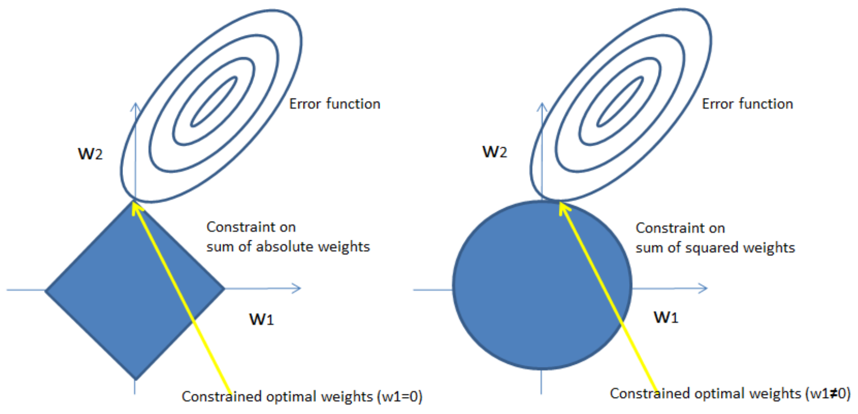

W2
Error function
Constraint on sum of absolute weights
W1
Constrained optimal weights (w1=0)
W2
Error function
Constraint on sum of squared weights
W1
Constrained optimal weights (w1≠0)

Figure 8.4: In LASSO, the best solution is where the contours of the quadratic error function touch the square, and this will sometimes occur at a corner, corresponding to some zero coefficients. On the contrary the quadratic constraint of ridge regression does not have corners for the contours to hit and hence zero values for the weights will rarely result.

process.

Let's note that the term that penalizes large weights in Equation (8.5) does not have a derivative when a weight is equal to zero (the partial derivative jumps from minus one for negative values to plus one for positive values). The "trick" of obtaining a linear system by calculating a derivative and setting it to zero cannot be used. The LASSO optimization problem may be solved by using **quadratic programming** with linear inequality constraints or more general convex optimization methods. The best value for the LASSO parameter λ can be estimated via cross-validation.

8.3.1 Lagrange multipliers to optimize problems with constraints

The above method of transforming an optimization problem with constraints into an unconstrained one is widely used and it deserves a small math digression for the most curious readers. In mathematical optimization, the method of **Lagrange multipliers** is a strategy for finding the local maxima and minima of a function subject to *constraints*. The problem is transformed into an unconstrained one by adding each constrained multiplied by a parameter (a Lagrange multiplier). Minimizing the transformed function yields a necessary condition for optimality.

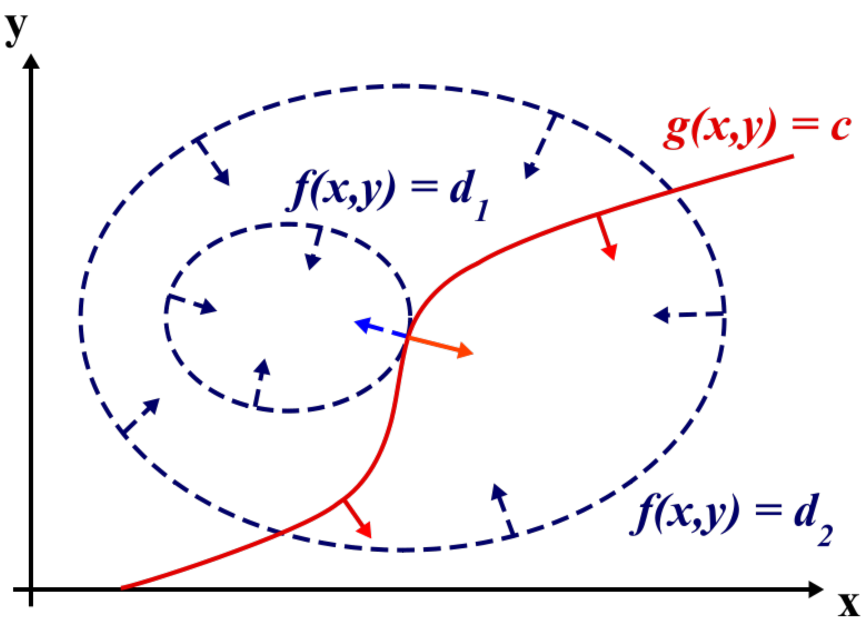


Figure 8.5: Lagrange multipliers.

Consider a two-dimensional problem:

$$\begin{array}{ll} \text{maximize} & f(x,y) \\ \text{subject to} & g(x,y) = c. \end{array}$$

We can visualize contours of f given by

$$f(x,y) = d$$

for various values of d and the contour of g given by $g(x,y) = c$, as shown in Fig. 8.5.

Suppose we walk along the contour line with $g = c$. In general the contour lines of f and g may be distinct, so the contour line for $g = c$ will intersect with or cross the contour lines of f. This is equivalent to saying that while moving along the contour line for $g = c$ the value of f can vary. Only when the contour line for $g = c$ meets contour lines of f tangentially, do we neither increase nor decrease the value of f – that is, when the contour lines touch but do not cross.

The contour lines of f and g touch when the **tangent vectors of the contour lines are parallel**. Since the gradient of a function is perpendicular to the contour lines, this is the same as saying that the gradients of f and g are parallel. Thus we want points (x,y) where $g(x,y) = c$ and

$$\nabla f(x,y) = \lambda \nabla g(x,y).$$

The Lagrange multiplier λ specifies how one gradient needs to be multiplied to obtain the other one!

Gist

Linear models are widely used but insufficient in many cases. Three examples of specific modifications have been considered in this chapter.

First, there can be reasons why the output needs to have a limited range of possible values. For example, if one needs to predict a probability, the output can range only between zero and one. Passing a linear combination of inputs through a "squashing" logistic function is a possibility. When the log-likelihood of the training events is maximized, one obtains the widely-used **logistic regression**.

Second, there can be cases when a linear model needs to be *localized*, by giving more significance to input points which are closer to a given input sample to be predicted. This is the case of **locally-weighted regression**.

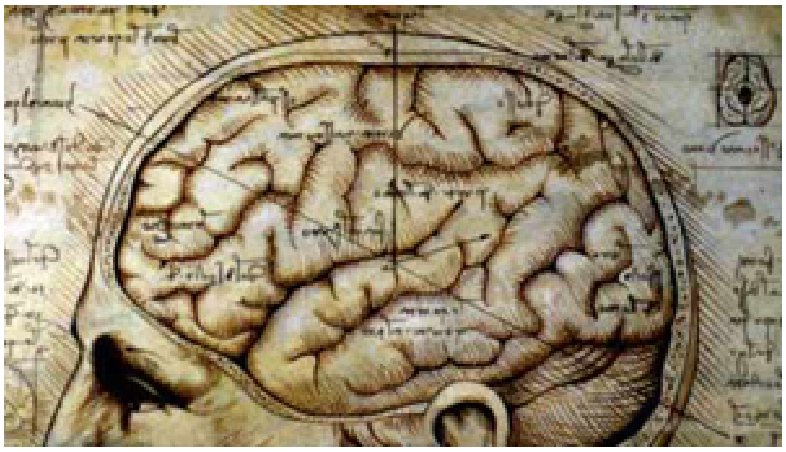
Last, the penalties for large weights added to the function to be optimized can be different from the sum of squared weights (the only case in which a linear equation is obtained after calculating derivatives). As an example, penalties given by a sum of absolute values can be useful to both reduce weight magnitude and *sparsify* the inputs. LASSO reduces the number of weights different from zero, and therefore the number of inputs which influence the output. This is the case of the **LASSO** technique to shrink and select inputs.

Before reading this chapter a lasso for you was a long rope with a running noose at one end used especially to catch horses and cattle. Now you can catch more meaningful models too.

Chapter 9

Neural networks, shallow and deep

Those who took other inspiration than from nature, master of masters,
were labouring in vain.
(Leonardo Da Vinci)



Our **wet neural system**, composed of about 100 billion (100,000,000,000) computing units and about 10^{15} (1,000,000,000,000,000) connections is capable of surprisingly intelligent behaviors. Actually, the capabilities of our brain *define* intelligence. The computing units are specialized cells called **neurons**, the connections are called **synapses**, and computation occurs at each neuron by currents generated by electrical signals at the synapses, integrated in the central part of the neuron, and leading to electrical spikes propagated to other neurons when a threshold of excitation is surpassed. Neurons and synapses have been presented in Chapter 4 (Fig. 4.3). A way to model a neuron is though a linear discrimination by a weighted sum of the inputs passed through a ”squashing” function (Fig. 4.4). The output level of the squashing function is intended to represent the spiking frequency of a neuron, from zero up to a maximum frequency.

Latest chapter revision: November 14, 2013

A single neuron is therefore **a simple computing unit**, a scalar product followed by a sigmoidal function. By the way, the computation is rather noisy and irregular, being based on electrical signals influenced by chemistry, temperature, blood supply, sugar levels, etc. The intelligence of the system is coded in the interconnection strengths, and **learning occurs by modifying connections**. The paradigm is very different from that of "standard" sequential computers, which operate in cycles, fetching items from memory, applying mathematical operations and writing results back to memory. Neural networks do not separate memory and processing but operate via the flow of signals through the network connections.

The main mystery to be solved is how a system composed of many simple interconnected units can give rise to such incredible activities as recognizing objects, speaking and understanding, drinking a cup of coffee, fighting for your career. **Emergence** is the way in which **complex systems arise out of a multiplicity of relatively simple interactions**. Similar emergent properties are observed in nature, think about snowflakes forming complex symmetrical patterns starting from simple water molecules.

The real brain is therefore an incredible source of inspiration for researchers, and **a proof that intelligent systems can emerge from very simple interconnected computing units**. Ever since the early days of computers, the biological metaphor has been irresistible ("electronic brains"), but only as a simplifying analogy rather than a blueprint for building intelligent systems. As Frederick Jelinek put it, "**airplanes don't flap their wings**." Yet, starting from the sixties, and then again the late eighties, the principles of biological brains gained ground as a tool in computing. The shift in thinking resulted in a paradigm change, from artificial intelligence based on symbolic rules and reasoning, to artificial neural systems where knowledge is encoded in system parameters (like synaptic interconnection weights) and learning occurs by gradually modifying these parameters under the influence of external stimuli.

Given that the function of a single neuron is rather simple, it subdivides the input space into two regions by a hyperplane, the complexity must come from having *more* layers of neurons involved in a complex action (like recognizing your grandmother in all possible situations). The "squashing" functions introduce critical nonlinearities in the system, without their presence multiple layers would still create linear functions. Organized layers are very visible in the human cerebral cortex, the part of our brain which plays a key role in memory, attention, perceptual awareness, thought, language, and consciousness (Fig. 9.1).

For more complex "sequential" calculations like those involved in logical reasoning, feedback loops are essential but more difficult to simulate via artificial neural networks. As you can expect, the "high-level", symbolic, and reasoning view of intelligence is complementary to the "low-level" sub-symbolic view of artificial neural networks. What is simple for a computer, like solving equations or reasoning, is difficult for our brain, what is simple for our brain, like recognizing our grandmother, is still

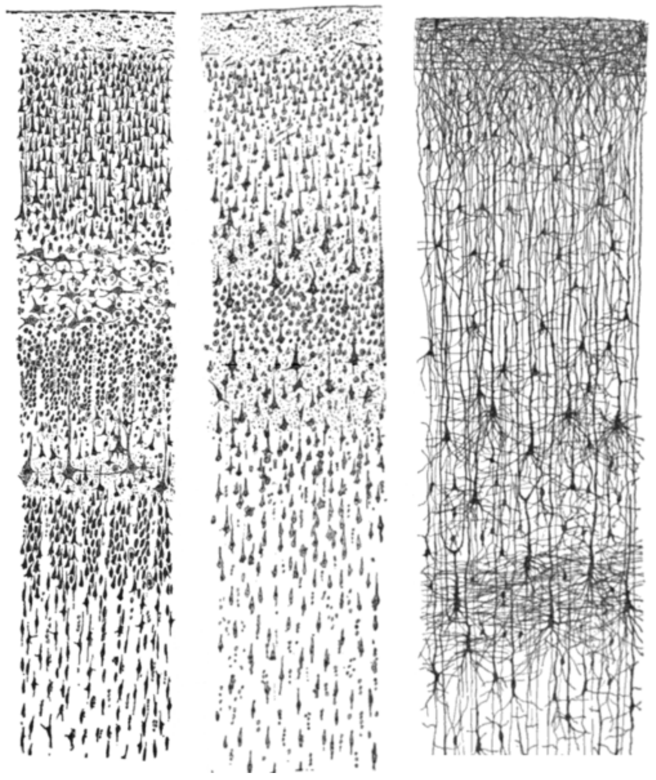


Figure 9.1: Three drawings of cortical lamination by Santiago Ramon y Cajal, each showing a vertical cross-section, with the surface of the cortex at the top. The different stains show the cell bodies of neurons and the dendrites and axons of a random subset of neurons.

difficult to simulate on a computer[1].

In any case, the ”airplanes don’t flap their wings” principle still holds. Even if real brains are a useful source of inspiration and a proof of feasibility, most artificial neural networks are actually run on standard computers, and the different areas of ”neural networks”, ”machine learning”, ”artificial intelligence” are actually converging so that the different terms are now umbrellas that cover a continuum of techniques to address different and often complementary aspects of intelligent systems.

This chapter is focused on **feedforward multi-layer perceptron neural networks** (without feedback loops).

[1]The two styles of intelligent behavior are now widely recognized, leading also to popular books about ”fast and slow thinking” [24].

9.1 Multilayer Perceptrons (MLP)

The logistic regression model of Section 8.1 in Chapter 8 was a simple way to add the ”minimal amount of nonlinearity” to obtain an output which can be interpreted as a probability, by applying a sigmoidal transfer function to the unlimited output of a linear model. Imagine this as transforming a crisp plane separating the input space (output 0 on one side, output 1 on the other side, based on a linear calculation compared with a threshold) into a smooth and gray transition area, black when far from the plane in one direction, white when far from the plane in the other direction, gray in between[2] (Fig. 9.2).

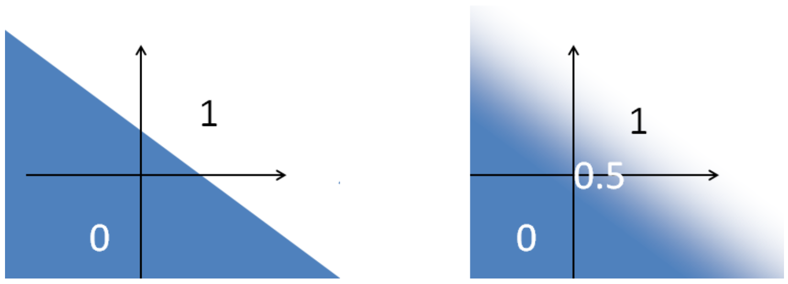


Figure 9.2: Effect of the logistic function. Linear model with threshold (left), smooth sigmoidal transition (right).

If one visualizes y as the elevation of a terrain, in many cases a mountain area presents too many hills, peaks and valleys to be modeled by planes plus maybe a single smooth transition region.

If linear transformations are composed, by applying one after another, the situation does not change: two linear transformation in a sequence still remain linear[3]. But if the output of the first linear transformation is transformed by a *nonlinear* sigmoid before applying the second linear transformation we get what we want: **flexible models capable of approximating all smooth functions**. The term **non-parametric models** is used to underline their flexibility and differentiate them from rigid models in which only the value of some parameters can be tuned to the data[4]. The first linear transformation will provide a first ”hidden layer” of outputs (hidden because internal and not directly visible as final ouputs), the second transformation will act to produce the visible outputs from the hidden layer.

[2]As an observation, let’s note that logistic regression and an MLP network with no hidden layer and a single output value are indeed the same architecture, what changes is the function being optimized, the sum of squared errors for MLP, the $LogLikelyhood$ for logistic regression.

[3]Let’s consider two linear transformations A and B. Applying B after A to get $B(A(\boldsymbol{x}))$ still maintains linearity. In fact, $B(A(a\boldsymbol{x} + b\boldsymbol{y})) = B(aA(\boldsymbol{x}) + bA(\boldsymbol{y})) = aB(A(\boldsymbol{x})) + bB(A(\boldsymbol{y}))$.

[4]An example of a parametric model could be an oscillation $\sin(\omega x)$, in which the parameter ω has to be determined from the experimental data.

A multilayer perceptron neural network (MLP) is composed of a large number of highly interconnected units (*neurons*) working in parallel to solve a specific problem and organized in layers with a feedforward information flow (no loops). The architecture is illustrated in Fig. 9.3.

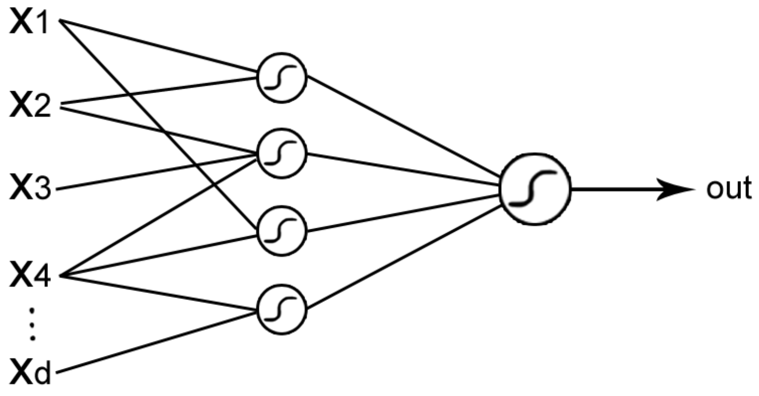


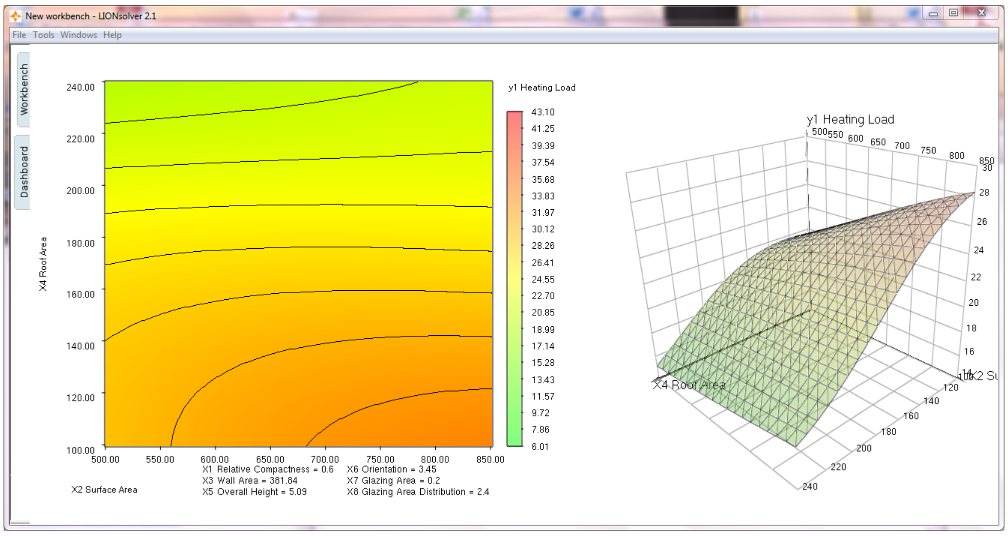
Figure 9.3: Multilayer perceptron: the nonlinearities introduced by the sigmoidal transfer functions at the intermediate (hidden) layers permit arbitrary continuous mappings to be created. A single hidden layer is present in this image.

The architecture of the multilayer perceptron is organized as follows: the signals flow sequentially through the different layers from the input to the output layer. The intermediate layers are called *hidden* layers because they are not visible at the input or at the output. For each layer, each unit first calculates a scalar product between a vector of weights and the vector given by the outputs of the previous layer. A *transfer function* is then applied to the result to produce the input for the next layer. A popular smooth and *saturating* transfer function (the output saturates to zero for large negative signals, to one for large positive signals) is the sigmoidal function, called sigmoidal because of the “S” shape of its plot. An example is the logistic transformation encountered before (Fig. 8.1):

$$f(x) = \frac{1}{1+e^{-x}}.$$

Other transfer functions can be used for the output layer; for example, the identity function can be used for unlimited output values, while a sigmoidal output function is more suitable for “yes/no” classification problems or to model probabilities.

A basic question about MLP is: **what is the flexibility of this architecture** to represent input-output mappings? In other words, given a function $f(\boldsymbol{x})$, is there a specific MLP network with specific values of the weights so that the MLP output closely approximates the function f? While perceptrons are limited in their modeling power to classification cases where the patterns (i.e., inputs) of the two different classes can be



X1 Relative Compactness = 0.6
X3 Wall Area = 381.84
X5 Overall Height = 5.09
X6 Orientation = 3.45
X7 Glazing Area = 0.2
X8 Glazing Area Distribution = 2.4

Figure 9.4: Analyzing a neural network output with LIONsolver Sweeper. The output value, the energy consumed to heat a house in winter, is shown as a function of input parameters. Color coded output (left), surface plot (right). Nonlinearities are visible.

separated by a hyperplane in input space, MLP's are **universal approximators** [22]: an MLP with one hidden layer can approximate any smooth function to any desired accuracy, subject to a sufficient number of hidden nodes.

This is an interesting results: neural-like architectures composed of simple units (linear summation and squashing sigmoidal transfer functions), organized in layers with at least a hidden layer are what we need to model arbitrary smooth input-output transformations.

For colleagues in mathematics this is a brilliant "existence" results. For more practical colleagues the next question is: given that there exist an MLP approximator, how can one find it in a fast manner by starting from labeled examples?

After reading the previous chapter you should already know at least a possible training technique. Think about it and then proceed to the next section.

As an example of an MLP input-output transformation, Fig. 9.4 shows the smooth and nonlinear evolution of the output value as a function of varying input parameters. By using sliders one fixes a subset of the input values, and the output is color-coded for a range of values of two selected input parameters.

9.2 Learning via backpropagation

As usual, take a "guiding" function to be optimized, like the traditional sum-of-squared errors on the training examples, make sure it is smooth (differentiable), and use **gradient descent**. Iterate by calculating the gradient of the function w.r.t. the weights and by taking a small step in the direction of the negative gradient[5].

The technical issue is now one of *calculating* partial derivatives by using the "chain rule" formula[6] in calculus for computing the derivative of the composition of two or more functions. In MLP the basic functions are: scalar products, then sigmoidal functions, then scalar products, and so on until the output layer is reached and the error is computed. For MLP network the gradient can be efficiently calculated, its calculation requires a number of operations proportional to the number of weights, and the actual calculation is done by simple formulas similar to the one used for the forward pass (from inputs to outputs) but now going in the contrary directions, from output errors backwards towards the inputs. The popular technique in neural networks known as learning by **backpropagation** of the error consists precisely in the above exercise: gradient calculation and small step in the direction of the negative gradient [37, 38, 29].

It is amazing how a straightforward application of gradient descent took so many years to reach wide applicability in the late eighties, and brought so much fame to the researchers who made it popular. A possible excuse is that gradient descent is normally considered a "vanilla" technique capable of reaching only locally-optimal points (with zero gradient) without guarantees of global optimality. Experimentation on different problems, after initializing the networks with small and randomized weights, was therefore needed to show its practical applicability for training MLPs. In addition, let's remember that ML aims at *generalization*, and for this goal reaching global optima is not necessary. It may actually be counterproductive and lead to overtraining!

The use of gradual adaptations with simple and local mechanisms permits a close link with neuroscience, although the detailed realization of gradient descent algorithms with real neurons is still a research topic.

Let's note that, after the network is trained, calculating the output from the inputs requires a number of simple operations proportional to the number of weights, and therefore the operation can be extremely fast if the number of weights is limited.

Let us briefly define the notation. We consider the "standard" multi-layer percep-

[5]If the gradient is different from zero, there exist a sufficiently small step in the direction of the negative gradient which will decrease the function value.

[6] If f is a function and g is a function, then the chain rule expresses the derivative of the composite function $f \circ g$ in terms of the derivatives of f and g. For example, the chain rule for $(f \circ g)(x)$ is:

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}.$$

tron (MLP) architecture, with weights connecting only nearby layers and the sum-of-squared-differences *energy* function defined as:

$$E(w) \; = \; \frac{1}{2}\sum_{p=1}^{P} E_p \; = \; \frac{1}{2}\sum_{p=1}^{P}(t_p \; - \; o_p(w))^2, \tag{9.1}$$

where t_p and o_p are the target and the current output values for pattern p, respectively. The sigmoidal transfer function is $f(x) = 1/(1 + e^{-x})$.

The initialization can be done by having the initial weights randomly distributed in a range[7]. In the following sections we present two 'gradient-based' techniques: standard *batch* backpropagation and a version with adaptive learning rate (*bold driver BP*, see [1]), and the on-line stochastic backpropagation of [29].

9.2.1 Batch and "Bold Driver" Backpropagation

The batch backpropagation update is a textbook form of gradient descent. After collecting all partial derivatives in the gradient, denoted as $g_k = \nabla E(w_k)$, the weights at the next iteration $k + 1$ are updated as follows:

$$w_{k+1} = w_k \; - \; \epsilon \; g_k. \tag{9.2}$$

The previous update, with a fixed *learning rate* ϵ, can be considered as a crude version of *steepest descent*, where the exact minimum along the gradient direction is searched at each iteration:

$$w_{k+1} = w_k - \epsilon_k g_k, \tag{9.3}$$

$$\text{where} \quad \epsilon_k \;\; \text{minimizes} \quad E(w_k - \epsilon g_k). \tag{9.4}$$

An application problem consists of picking a value of the learning rate which is appropriate for a specific learning task, not too small to avoid very long training times (caused by very small modifications of the weights at every iteration) and not too large to avoid oscillations leading to wild increases of the energy function (let's remember that a descent is *guaranteed* only if the step along the gradient becomes very small).

An heuristic proposal for avoiding the choice and for modifying the learning rate while the learning task runs is the *bold driver* (BD) method described in [1]. The learning rate increases exponentially if successive steps reduce the energy, and decreases rapidly if an "accident" is encountered (if E increases), until a suitable value is found.

[7]Choosing an initial range, like $(-.5, .5)$ is not trivial, if the weights are too large, the scalar products will be in the saturated areas of the sigmoidal function, leading to gradients close to zero and numerical problems.

After starting with a small learning rate, its modifications are described by the following equation:

$$\epsilon(t) = \begin{cases} \rho\, \epsilon(t-1), & \text{if } E(w(t)) < E(w(t-1)); \\ \sigma^l\, \epsilon(t-1), & \text{if } E(w(t)) > E(w(t-1)) \text{ using } \epsilon(t-1), \end{cases} \tag{9.5}$$

where ρ is close to one ($\rho = 1.1$) in order to avoid frequent "accidents" because the energy computation is wasted in these cases, σ is chosen to provide a rapid reduction ($\sigma = 0.5$), and l is the minimum integer such that the reduced rate $\sigma^l\epsilon(t-1)$ succeeds in diminishing the energy.

The performance of this "quick and dirty" form of backpropagation is close and usually better than the one obtained by appropriately choosing a *fixed* learning rate. Nonetheless, being a simple form of *steepest descent*, the technique suffers from the common limitation of techniques that use the gradient as a search direction.

9.2.2 On-Line or stochastic backpropagation

Because the energy function E is a sum of many terms, one for each pattern, the gradient will be a sum of the corresponding partial gradients $\nabla E_p(w_k)$, the gradient of the error for the p-th pattern: $(t_p \;-\; o_p(w))^2$.

If one has one million training examples, first the contributions $\nabla E_p(w_k)$ are summed, and the small step is taken.

An immediate option comes to mind: how about taking a small step along a single negative $\nabla E_p(w_k)$ immediately after calculating it? If the steps are very small, the weights will differ by small amounts w.r.t. the initial ones, and the successive gradients $\nabla E_p(w_{k+j})$ will be very similar to the original ones $\nabla E_p(w_k)$.

If the patterns are taken in a random order, one obtains what is called **stochastic gradient descent**, a.k.a. **online backpropagation**.

By the way, because biological neurons are not very good at complex and long calculations, online backpropagation has a more biological flavor. For example, if a kid is learning to recognize digits and a mistake is done, the correction effort will tend to happen immediately after the recognized mistake, not after waiting to collect thousands of mistaken digits.

The stochastic on-line backpropagation update is given by:

$$w_{k+1} = w_k \;-\; \epsilon\, \nabla E_p(w_k), \tag{9.6}$$

where the pattern p is chosen randomly from the training set at each iteration and ϵ is the learning rate. This form of backpropagation has been used with success in many contexts, provided that an appropriate learning rate is selected by the user. The main difficulties of the method are that the iterative procedure is not guaranteed to converge and that the use of the gradient as a search direction is very inefficient for some problems[8]. The competitive advantage with respect to *batch* backpropagation, where the complete gradient of E is used as a search direction, is that the partial gradient $\nabla E_p(w_k)$ requires only a single forward and backward pass, so that the inaccuracies of the method can be compensated by the low computation required by a single iteration, especially if the training set is large and composed of redundant patterns. In these cases, if the learning rate is appropriate, the actual CPU time for convergence can be small.

The learning rate must be chosen with care: if ϵ is too small the training time increases without producing better generalization results, while if ϵ grows beyond a certain point the oscillations become gradually wilder, and the uncertainty in the generalization obtained increases.

9.2.3 Advanced optimization for MLP training

As soon as the importance of optimization for machine learning was recognized, researchers began to use techniques derived from the optimization literature that use higher-order derivative information during the search, going *beyond gradient descent*. Examples are conjugate gradient and ”secant” methods, i.e., methods that update an approximation of the second derivatives (of the Hessian) in an iterative way by using only gradient information. In fact, it is well known that taking the gradient as the current search direction produces very slow convergence speed if the Hessian has a large *condition number* (in a pictorial way this corresponds to having ”narrow valleys” in the search space). Techniques based on second order information are of widespread use in the neural net community, their utility being recognized in particular for problems with a limited number of weights ($<$ 100) and requiring high precision in the output values. A partial bibliography and a description of the relationships between different second-order techniques has been presented in [2]. Two techniques that use second-derivatives information (in an indirect and fast way): the conjugate gradient technique and the one-step-secant method with fast line searches (OSS), are described in [2], [1]. More details will be described in the chapter of this book dedicated to optimization of continuous functions.

9.3 Deep neural networks

There is abundant evidence from neurological studies that our brain develops higher-level concepts in stages by first extracting **multiple layers of useful and gradually**

[8]The precise definition is that of *ill-conditioned* problems.

more complex representations. In order to recognize your grandmother, first simple elements are detected in the visual cortex, like image edges (abrupt changes of intensity), then progressively higher-level concepts like eyes, mouth, and complex geometrical features, independently on the specific position in the image, illumination, colors, etc.

The fact that one hidden layer is sufficient for the existence of a suitable approximation does not mean that building this approximation will be easy, requiring a small number of examples and a small amount of CPU time. In addition to the neurological evidence in our brain, there are theoretical arguments to demonstrate that some classes of input-output mappings are much easier to build when more hidden layers are considered [7].

The dream of ML research is to feed examples to an MLP with many hidden layers and have the MLP **automatically develop internal representations** (the activation patterns of the hidden-layer units). The training algorithm should determine the weights interconnecting the lower levels (the ones closer to the sensory input) so that the representations in the intermediate levels correspond to ”concepts” which will be useful for the final complex classification task. Think about the first layers developing ”nuggets” of useful regularities in the data.

This dream has some practical obstacles. When backpropagation is applied to an MLP network with many hidden layers, the partial derivatives associated to the weights of the first layers tend to be very small, and therefore subject to numerical estimation problems. This is easy to understand[9]: if one changes a weight in the first layers, the effect will be propagated upwards through many layers and it will tend to be confused among so many effects by hundreds of other units. Furthermore, saturated units (with output in the flat regions of the sigmoid) will squeeze the change so that the final effect on the output will be very small. The net effect can be that the internal representations developed by the first layers will not differ too much from what can be obtained by setting the corresponding first-layer weights randomly, and leaving only the topmost levels to do some ”useful” work. From another point of view, when the number of parameters is very large w.r.t. the number of examples (and this is the case of deep neural networks) overtraining becomes more dangerous, it will be too easy for the network to accommodate the training examples without being forced to extract the relevant regularities, those essential for generalizing.

In the nineties, these difficulties shifted the attention of many users towards ”simpler” models, based on linear systems with additional constraints, like the Support Vector Machines considered in the subsequent chapter.

More recently, a revival of **deep neural networks** (MLPs with many hidden layers)

[9]If you are not familiar with partial derivatives, think about changing a weight by a small amount Δw and calculating how the output changes Δf. The partial derivative is the limit of the ratio $\Delta f/\Delta w$ when the magnitude of the change goes to zero.

and more powerful training techniques brought deep learning to the front stage, leading to superior classification results in challenging areas like speech recognition, image processing, molecular activity for pharmaceutical applications. Deep learning **without any ad hoc feature engineering** (handcrafting of new features by using knowledge domain and preliminary experiments) lead to winning results and significant improvements over the state of the art [7].

The main scheme of the latest applications is as follows:

1. use unsupervised learning from many unlabeled examples to prepare the deep network in an initial state;

2. use backpropagation only for the final tuning with the set of labeled examples, after starting from the initial network trained in an unsupervised manner.

The scheme is very powerful for all those situations in which the number of unlabeled (unclassified) examples is much larger than the number of labeled examples, and the classification process is costly. For examples, collecting huge numbers of unlabeled images by crawling the web is now very simple. Labeling them by having humans to describe the image content is much more costly. The unsupervised system is in charge of extracting useful building blocks, like detectors for edges, for blobs, for textures of different kinds, in general, building blocks which appear in real images and not in random ”broken TV screen” patterns.

9.3.1 Auto-encoders

An effective way to build internal representations in an unsupervised manner is through auto-encoders. One builds a network with a hidden layer and demands that the output simply reproduces the input. It sounds silly and trivial at first, but interesting work gets done when one squeezes the hidden layer, and therefore demands that the original information in the input is compressed into **an encoding $c(\boldsymbol{x})$ with less variables than the original ones** (Fig. 9.5). For sure, this compression will not permit a faithful reconstruction of *all* possible inputs. But this is positive for our goals: the internal representation $c(\boldsymbol{x})$ will be forced to discover regularities in the specific input patterns shown to the system, to extract useful and significant information from the original input.

For example, if images of faces are presented, some internal units will specialize to detect edges, other maybe will specialize to detect eyes, and so on.

The auto-encoder can be trained by backpropagation or variations thereof. Classification labels are not necessary. If the original inputs are labeled for classification, the labels are simply forgotten by the system in this phase.

After the auto-encoder is built one can now transplant the hidden layer structure (weights and hidden units) to a second network intended for classification (Fig. 9.6), add

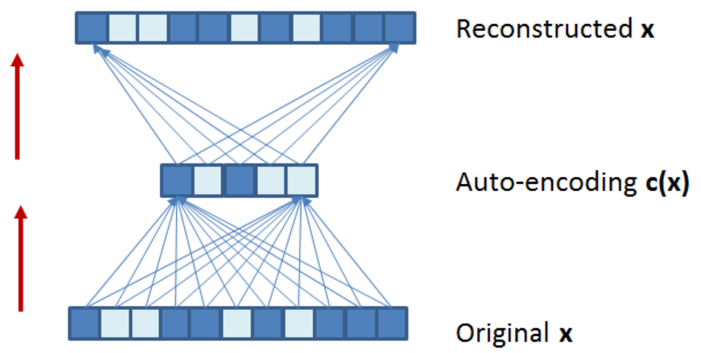


Figure 9.5: Auto-encoder.

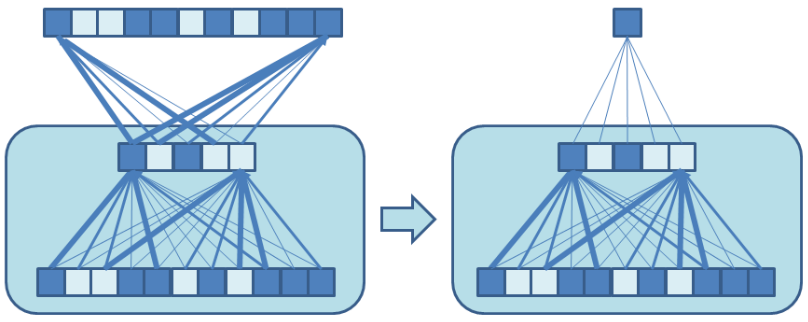


Figure 9.6: Using an auto-encoder trained from unlabeled data to initialize an MLP network.

an additional layer (initialized with small random weights), and consider this ”Frankenstein monster” network as the starting point for a final training phase intended to realize a classifier. In this final phase only a set of labeled pattern is used.

In many significant applications the final network has a better generalization performance than a network which could be obtained by initializing randomly all weights. Let’s note that the same properly-initialized network can be used for different but related supervised training tasks. The network is initialized in the same manner, but different labeled examples are used in the final tuning phase. **Transfer learning** is the term related to using knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize people faces could apply when recognizing monkeys.

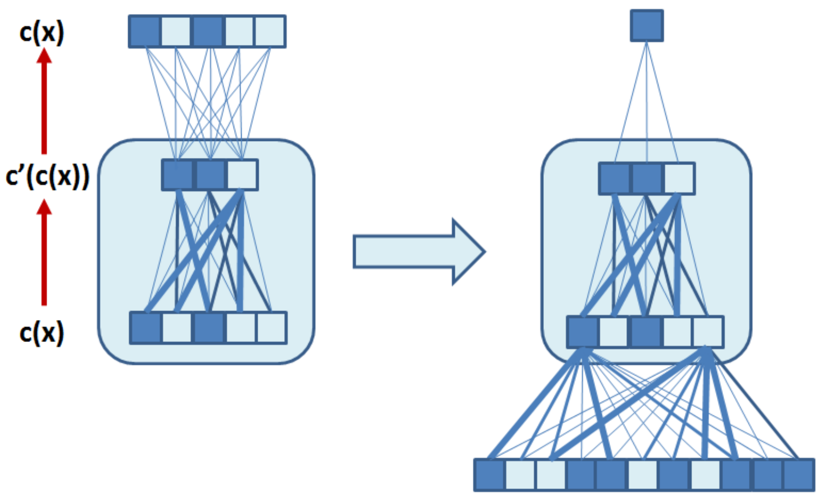


Figure 9.7: Recursive training of auto-encoders to build deeper networks.

The attentive reader may have noticed that up to now only one hidden layer has been created. But we can easily produce a chain of subsequent layers by iterating, compressing the first code $c(\boldsymbol{x})$, again by auto-encoding it, to develop a second more compressed code and internal representation $c'(c(\boldsymbol{x}))$. Again, the developed auto-encoding weights can be used to initialize the second layer of the network, and so on (Fig. 9.7).

In addition to being useful for pre-training neural networks, very **deep auto-encoders can be useful for visualization and clustering**. For example, if one considers handwritten digits, and manages to auto-encode them so that the bottleneck compressed layer contains only two units, the two-dimensional coordinates corresponding to each digit can be visualized on a two-dimensional plane. If done appropriately, different clusters approximately corresponding to the different digits are clearly visible in the two-dimensional space, the two (or more) coordinates can therefore be very good starting points for clustering objects.

The optimal number of layers and the optimal number of units in the ”pyramidal” structure is still a research topic, but appropriate numbers can be obtained pragmatically by using some form of cross-validation to select appropriate meta-parameters. More details in [7].

9.3.2 Random noise, dropout and curriculum

Now that you are getting excited about the idea of combining unsupervised pre-training with supervised final tuning to get deep and deeper network, so that less and less hand-made feature engineering will be required for state-of-the art performance, let's mention some more advanced possibilities which are now being transferred from pure research to the first real-world applications.

The first possibility has to do with injecting controlled amount of noise into the system [36] (**denoising auto-encoders**). The starting idea is very simple: corrupt each pattern $\boldsymbol{x}$ with random noisy (e.g., if the pattern is binary, flip the value of each bit with a given small probability) and ask the auto-encoding network to reconstruct the original noise-free pattern $\boldsymbol{x}$, to *denoise* the corrupted versions of its inputs. The task becomes more difficult, but asking the system to go the extra mile encourages it to extract even stronger and more significant regularities from the input patterns. This version bridges the performance gap with deep belief networks (DBN), another way to pre-train networks which is more difficult to explain [18, 19], and in several cases surpasses it. Biologically, there is indeed *a lot* of noise in the wet brain matter. These results demonstrate that noise can in fact have a positive impact on learning!

Another way to make the learning problem harder but to increase generalization (by reducing overfitting) is through **random dropout** [20]: during stochastic backpropagation training, after presenting each training case, each hidden unit is randomly omitted from the network with probability 0.5. In this manner, complex co-adaptation on training data is avoided. Each unit cannot rely on other hidden units being present and is encouraged to become a detector identifying useful information, independently on what the other units are doing[10]. In a way, randomly dropping some units is related to using different network architectures at different times during training, and then averaging their results during testing. Using ensembles of different networks is another way to reduce overtraining and increasing generalization, as it will be explained in future chapters. With random dropout the different networks are contained in the same complete MLP network (they are obtained by activating only selected parts of the complete network).

Another possibility to improve the final result when training MLP is though the use of **curriculum learning** [8]. As in the human case, training examples are not presented to the network at the same time but in stages, by starting from the easiest cases first and

[10] Interestingly, there is an intriguing similarity between dropout and the role of sex in evolution. One possible interpretation is that sex breaks up sets of co-adapted genes. Achieving a function by using a large set of co-adapted genes is not nearly as robust as achieving the same function, in multiple alternative ways, each of which only uses a small number of co-adapted genes. This allows evolution to avoid dead-ends in which improvements in fitness require co-ordinated changes to a large number of co-adapted genes. It also reduces the probability that small changes in the environment will cause large decreases in fitness a phenomenon similar to the overfitting in the field of ML [20].

then proceeding to the more complex ones. For example, when learning music, first the single notes are learnt, then more complex symphonies. Pre-training by auto-encoding can be considered a preliminary form of curriculum learning. The analogy with the learning of languages is that first the learner is exposed to a mass of spoken material in a language (for example by placing him in front of a foreign tv channel). After training the ear to be tuned to characteristic utterances of the given spoken language, the more formal phase of training by translating phrases is initiated.

Gist

Creating artificial intelligence based on the ”real thing” is the topic of artificial neural networks research. **Multi-layer perceptron neural networks** (MLPs) are a flexible (non-parametric) modeling architecture composed of layers of sigmoidal units interconnected in a feedforward manner only between adjacent layers. A unit recognizing the probability of your grandmother appearing in an image can be built with our neural hardware (no surprise) modeled as an MLP network. Effective training from labeled examples can occur via variations of gradient descent, made popular with the term ”error backpropagation.” The weakness of gradient descent as optimization method does not prevent successful practical results.

Deep neural networks composed of many layers are becoming effective (and superior to Support Vector Machines) through appropriate learning schemes, consisting of an unsupervised preparatory phase followed by a final tuning phase by using the scarce labeled examples.

Among the ways to improve generalization, the use of **controlled amounts of noise during training** is effective (noisy auto-encoders, random dropout). If you feel some noise and confusion in your brain, relax, it can be positive after all.

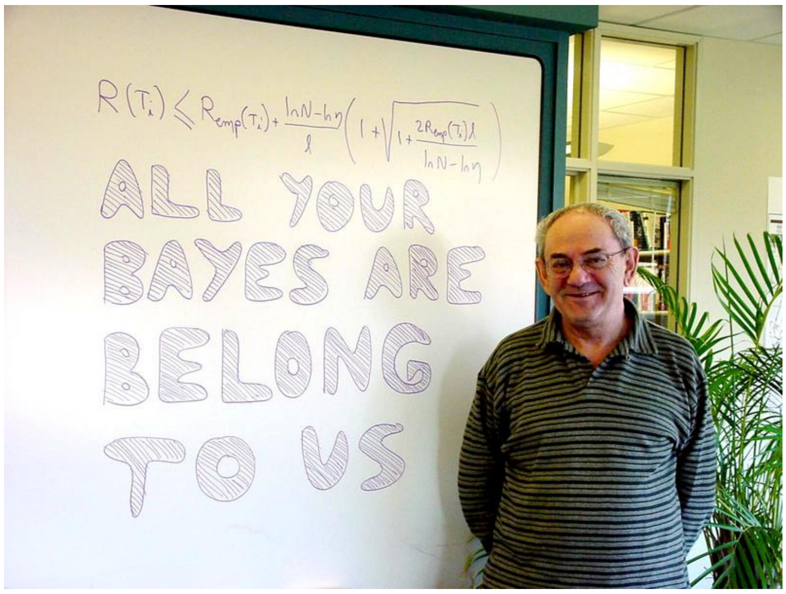
There are indeed striking analogies between human and artificial learning schemes. In particular, increasing the effort during training pays dividends in terms of improved generalization. The effort with a serious and demanding teacher (diversifying test questions, writing on the blackboard, asking you to take notes instead of delivering pre-digested material) can be a pain in the neck during training but increases the power of your mind at later stages of your life[a].

[a] The German philosopher Hegel was using the term *Anstrengung des Begriffs* (”effort to define the concept”) when defining the role of Philosophy.

Chapter 10

Statistical Learning Theory and Support Vector Machines (SVM)

Sembravano traversie ed eran in fatti opportunità.
They seemed hardships and were in fact opportunities.
(Giambattista Vico)


$R(T_i) \leq R_{emp}(T_i) + \frac{\ln N - \ln\eta}{l}\left(1 + \sqrt{1 + \frac{2R_{emp}(T_i)\,l}{\ln N - \ln\eta}}\right)$
ALL YOUR
BAYES ARE
BELONG
TO US

The order of chapters in this book has some connections with the history of machine learning[1]. Before 1980, most learning methods concentrated either on symbolic "rule-based" expert systems, or on simple sub-symbolic *linear discrimination* techniques, with clear theoretical properties. In the eighties, decision trees and neural networks

[1]The photo of prof. Vapnik is from Yann LeCun website (Vladimir Vapnik meets the video games sub-culture at `http://yann.lecun.com/ex/fun/index.html#allyourbayes`).

Latest chapter revision: November 14, 2013

paved the way to efficient learning of *nonlinear* models, but with little theoretical basis and naive optimization techniques (based on gradient descent).

In the nineties, efficient learning algorithms for nonlinear functions based on statistical learning theory developed, mostly through the seminal work by Vapnik and Chervonenkis. **Statistical learning theory** (SLT) deals with fundamental questions about *learning from data*. Under which conditions can a model learn from examples? How can the measured performance on a set of examples lead to bounds on the generalization performance?

These theoretical results are everlasting, although the conditions for the theorems to be valid are almost impossible to check for most practical problems. In another direction, the same researchers proposed a resurrection of **linear separability** methods, with additional ingredients intended to improve generalization, with the name of **Support Vectors Machines (SVM)**.

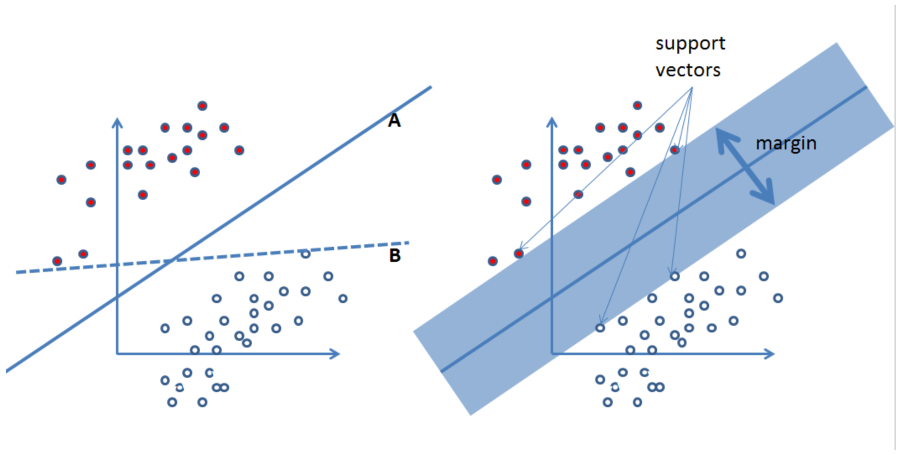


Figure 10.1: Explaining the basis of Support Vectors Machines. The *margin* of line A is larger than that of line B. A large margin increases the probability that new examples will fall in the right side of the separator line. *Support vectors* are the points touching the widest possible margin.

The term SVM sounds technical but the rationale is simple to grasp. Let's consider the two classes (dark and bright dots) in Fig. 10.1 (left) and the two possible lines A and B. They both linearly-separate the examples and can be two results of a generic ML scheme to separate the labeled training data. The difference between the two is clear when thinking about *generalization*. When the trained system will be used, new examples from the two classes will be generated with the same underlying probability distribution. Two clouds with a similar shape will be produced, but, for the case of line B, the probability that some of the new points will fall on the *wrong* side of the separator is bigger than for line A. Line B is passing very close to some training examples, it makes it just barely to separate them. Line A has the biggest possible distance from the examples of the two classes, it has the largest possible "safety area" around the boundary, a.k.a. *margin*. SVMs are **linear separators with the largest possible margin**, and the *support vectors* are the ones touching the safety *margin* region on both sides (Fig. 10.1, right).

Asking for the maximum-margin linear separator leads to standard **Quadratic Programming (QP)** problems, which can be solved to optimality for problems of reasonable size. QP is the problem of optimizing a quadratic function of several variables subject to linear constraints on these variables. The issue with local minima potentially dangerous for MLP[2] disappears and this makes users feel relaxed. As you may expect, there's no rose without a thorn, and complications arise if the classes are *not* linearly separable. In this case one first applies **a nonlinear transformation** ϕ to the points so that they become (approximately) linearly separable. Think of ϕ as building appropriate features so that the transformed points $\phi(\boldsymbol{x})$ of the two classes *are* linearly separable. The nonlinear transformation has to be handcrafted for the specific problem, no general-purpose transformation is available.

To discover the proper ϕ, are we back to feature extraction and feature engineering? In a way yes, and after transforming inputs with ϕ, the **features in SVM are all similarities between an example to be recognized and the training examples**[3]. A critical step of SVM, which has to be executed by hand through some form of cross-validation, is to identify which similarity measures are best to learn and generalize, an issue related to selecting the so-called **kernel functions**.

SVMs can be seen as a way to **separate two concerns**: that of identifying a proper way of **measuring similarities** between input vectors, the *kernel functions* $K(\boldsymbol{x}, \boldsymbol{y})$, and that of **learning a linear architecture** to combine the outputs on the training examples, weighted by the measured similarities with the new input example. As expected, more similar input examples contribute more to the output, as in the more primitive nearest-neighbors classifiers encountered in Chapter 2. This is the way to grasp formulas like:

$$\sum_{i=1}^{\ell} y_i \lambda_i^* K(\boldsymbol{x}, \boldsymbol{x}_i),$$

(ℓ is the number of training examples, y_i is the output on training example $\boldsymbol{x}_i$, $\boldsymbol{x}$ is the new example to be classified) that we will encounter in the following theoretical description. Kernels calculate *dot products* (scalar products) of data points mapped by a function $\phi(\boldsymbol{x})$ without actually calculating the mapping, this is called the "**kernel trick**" (Fig. 10.2):

$$K(\boldsymbol{x}, \boldsymbol{x}_i) = \boldsymbol{\varphi}(\boldsymbol{x}) \cdot \boldsymbol{\varphi}(\boldsymbol{x}_i).$$

A symmetric and positive semi-definite *Gram Matrix* containing the kernel values

[2] Because local minima can be very far from global optima.

[3] Actually only *support vectors* will give a non-zero contribution.

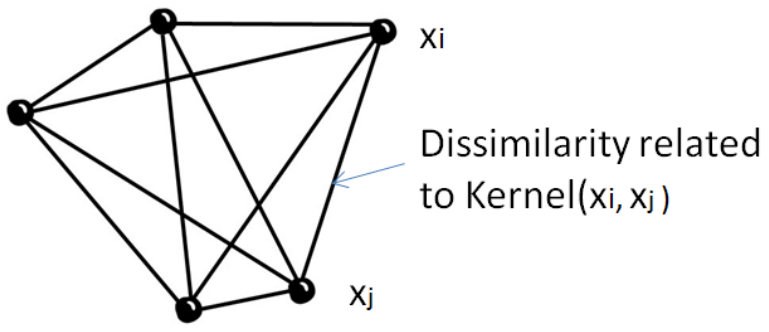


Figure 10.2: Initial information for SVM learning are similarity values between couples of input points $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where K is known as *kernel function*. These values, under some conditions, can be interpreted as scalar products obtained after mapping the initial inputs by a nonlinear function $\phi(\boldsymbol{x})$, but the actual mapping does not need to be computed, only the kernel values are needed ("kernel trick").

for couples of points fuses information about data and kernel[4]. Estimating a proper *kernel matrix* from available data, one that will maximize generalization results, is an ongoing research topic.

Now that the overall landscape is clear, let's plunge into the mathematical details. Some of these details are quite complex and difficult to grasp. Luckily, you will not need to know the demonstration of the theorems to use SVMs, although a knowledge of the main math results will help in selecting meta-parameters, kernels, etc.

10.1 Empirical risk minimization

We mentioned before that minimizing the error on a set of examples is not the only objective of a statistically sound learning algorithm, also the modeling architecture has to be considered. **Statistical Learning Theory** provides mathematical tools for *deriving unknown functional dependencies* on the basis of observations.

A **shift of paradigm** occurred in statistics starting from the sixties: previously, following Fisher's research in the 1920–30s, in order to derive a functional dependency from observations one had to know the detailed form of the desired dependency and to determine only the values of a finite number of *parameters* from the experimental

[4]Every similarity matrix can be used as kernel, if it satisfies Mercer's theorem criteria.

data. The new paradigm does not require the detailed knowledge, and proves that some general properties of the set of functions to which the unknown dependency belongs are sufficient to estimate the dependency from the data. **Nonparametric techniques** is a term used for these flexible models, which can be used even if one does not know a detailed form of the input-output function. The MLP model described before is an example.

A brief summary of the main methodological points of Statistical Learning Theory is useful to motivate the use of Support Vector Machines (SVM) as a learning mechanism. Let $P(\boldsymbol{x}, y)$ be the unknown probability distribution from which the examples are drawn. The learning task is to learn the mapping $\boldsymbol{x}_i \to y_i$ by determining the values of the parameters of a function $f(\boldsymbol{x}, \boldsymbol{w})$. The function $f(\boldsymbol{x}, \boldsymbol{w})$ is called *hypothesis*, the set $\{f(\boldsymbol{x}, \boldsymbol{w}) : \boldsymbol{w} \in \mathcal{W}\}$ is called the *hypothesis space* and denoted by $\mathcal{H}$, and $\mathcal{W}$ is the set of abstract parameters. A choice of the parameter $\boldsymbol{w} \in \mathcal{W}$, based on the labeled examples, determines a "trained machine."

The *expected test error* or *expected risk* of a trained machine for the classification case is:

$$R(\boldsymbol{w}) = \int \|y - f(\boldsymbol{x}, \boldsymbol{w})\| \, \mathrm{d}P(\boldsymbol{x}, y), \tag{10.1}$$

while the *empirical risk* $R_{\text{emp}}(\boldsymbol{w})$ is the mean error rate measured on the training set:

$$R_{\text{emp}}(\boldsymbol{w}) = \frac{1}{\ell} \sum_{i=1}^{\ell} \|y_i - f(\boldsymbol{x}_i, \boldsymbol{w})\|. \tag{10.2}$$

The classical learning method is based on the **empirical risk minimization** (ERM) inductive principle: one approximates the function $f(\boldsymbol{x}, \boldsymbol{w}^*)$ which minimizes the risk in (10.1) with the function $f(\boldsymbol{x}, \hat{\boldsymbol{w}})$ which minimizes the empirical risk in (10.2).

The rationale for the ERM principle is that, if R_{emp} converges to R in probability (as guaranteed by the law of large numbers), the minimum of R_{emp} may converge to the minimum of R. If this does not hold, the ERM principle is said to be *not consistent*.

As shown by Vapnik and Chervonenkis, consistency holds if and only if convergence in probability of R_{emp} to R is *uniform*, meaning that as the training set increases the probability that $R_{\text{emp}}(\boldsymbol{w})$ approximates $R(\boldsymbol{w})$ uniformly tends to 1 on the whole $\mathcal{W}$. Necessary and sufficient conditions for the consistency of the ERM principle is the finiteness of the **Vapnik-Chervonenkis dimension (VC-dimension)** of the hypothesis space $\mathcal{H}$.

The VC-dimension of the hypothesis space $\mathcal{H}$ is, loosely speaking, the largest number of examples that can be separated into two classes in all possible ways by the set of functions $f(\boldsymbol{x}, \boldsymbol{w})$. The VC-dimension h measures the **complexity and descriptive power of the hypothesis space** and is often proportional to the number of free parameters of the model $f(\boldsymbol{x}, \boldsymbol{w})$.

Vapnik and Chervonenkis provide **bounds on the deviation of the empirical risk from the expected risk**. A bound that holds with probability $1-p$ is the following:

$$R(\boldsymbol{w}) \leq R_{\text{emp}}(\boldsymbol{w}) + \sqrt{\frac{h\left(\ln\frac{2\ell}{h}+1\right) - \ln\frac{p}{4}}{\ell}} \quad \forall \boldsymbol{w} \in \mathcal{W}.$$

By analyzing the bound, and neglecting logarithmic factors, in order to obtain a small expected risk, both the empirical risk *and* the ratio h/ℓ between the VC-dimension of the hypothesis space and the number of training examples have to be small. In other words, a valid generalization after training is obtained if the hypothesis space is sufficiently powerful to allow reaching a small empirical risk, i.e., to learn correctly the training examples, but not too powerful to simply memorize the training examples without extracting the structure of the problem. For a larger model flexibility, a larger number of examples is required to achieve a similar level of generalization.

The choice of an appropriate value of the VC-dimension h is crucial to get good generalization performance, especially when the number of data points is limited.

The method of **structural risk minimization** (SRM) has been proposed by Vapnik based on the above bound, as an attempt to overcome the problem of choosing an appropriate value of h. For the SRM principle one starts from a nested structure of hypothesis spaces

$$\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots \subset \mathcal{H}_n \subset \cdots \tag{10.3}$$

with the property that the VC-dimension $h(n)$ of the set $\mathcal{H}_n$ is such that $h(n) \leq h(n+1)$. As the subset index n increases, the minima of the empirical risk decrease, but the term responsible for the confidence interval increases. The SRM principle chooses the subset $\mathcal{H}_n$ for which minimizing the empirical risk yields the best bound on the actual risk. Disregarding logarithmic factors, the following problem must be solved:

$$\min_{\mathcal{H}_n} \left(R_{\text{emp}}(\boldsymbol{w}) + \sqrt{\frac{h(n)}{\ell}} \right). \tag{10.4}$$

The SVM algorithm described in the following is based on the SRM principle, by minimizing a bound on the VC-dimension and the number of training errors at the same time.

The mathematical derivation of Support vector Machines is summarized first for the case of a linearly separable problem, also to build some intuition about the technique.

10.1.1 Linearly separable problems

Assume that the labeled examples are linearly separable, meaning that there exist a pair $(\boldsymbol{w}, b)$ such that:

$$\begin{array}{rcll} \boldsymbol{w}\cdot\boldsymbol{x}+b & \geq & 1 & \forall \boldsymbol{x} \in \text{Class}_1; \\ \boldsymbol{w}\cdot\boldsymbol{x}+b & \leq & -1 & \forall \boldsymbol{x} \in \text{Class}_2. \end{array}$$

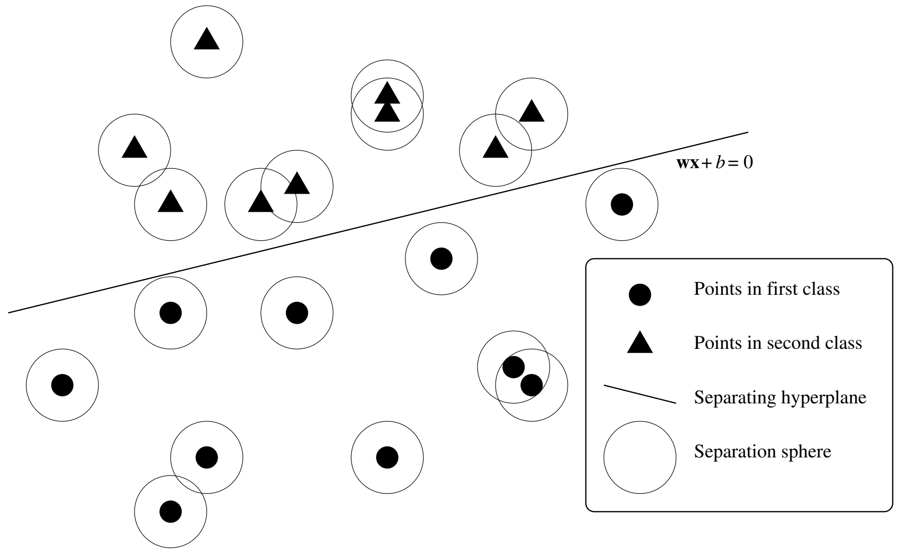


Figure 10.3: Hypothesis space constraint. The separating hyperplane must maximize the margin. Intuitively, no point has to be too close to the boundary so that some noise in the input data and future data generated by the same probability distribution will not ruin the classification.

The hypothesis space contains the functions:

$$f_{\boldsymbol{w},b} = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b).$$

Because scaling the parameters $(\boldsymbol{w}, b)$ by a constant value does not change the decision surface, the following constraint is used to identify a unique pair:

$$\min_{i=1,\ldots,\ell} |\boldsymbol{w} \cdot \boldsymbol{x}_i + b| = 1.$$

A structure on the hypothesis space can be introduced by limiting the norm of the vector $\boldsymbol{w}$. It has been demonstrated by Vapnik that if all examples lie in an n-dimensional sphere with radius R then the set of functions $f_{\boldsymbol{w},b} = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ with the bound $||\boldsymbol{w}|| \leq A$ has a VC-dimension h that satisfies

$$h \leq \min\{\lceil R^2 A^2 \rceil, n\} + 1.$$

The geometrical explanation of why bounding the norm of $\boldsymbol{w}$ constrains the hypothesis space is as follows (see Fig. 10.3): if $||\boldsymbol{w}|| \leq A$, then the distance from the hyperplane $(\boldsymbol{w}, b)$ to the closest data point has to be larger than $1/A$, because only the hyperplanes that do not intersect spheres of radius $1/A$ placed around each data point are considered. In the case of linear separability, minimizing $||\boldsymbol{w}||$ amounts to determining a separating hyperplane with the maximum *margin* (distance between the convex hulls of the two training classes measured along a line perpendicular to the hyperplane).

The problem can be formulated as:

$$\begin{array}{ll} \text{Minimize}_{\boldsymbol{w},b} & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{subject to} & y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 \quad i = 1,\ldots,\ell. \end{array}$$

The problem can be solved by using standard **quadratic programming (QP)** optimization tools.

The dual quadratic program, after introducing a vector $\boldsymbol{\Lambda} = (\lambda_1, \ldots, \lambda_\ell)$ of non-negative Lagrange multipliers corresponding to the constraints is as follows:

$$\begin{array}{ll} \text{Maximize}_{\boldsymbol{\Lambda}} & \boldsymbol{\Lambda}\cdot\mathbf{1} - \frac{1}{2}\boldsymbol{\Lambda}\cdot D\cdot\boldsymbol{\Lambda} \\ \text{subject to} & \begin{cases} \boldsymbol{\Lambda}\cdot\boldsymbol{y} = 0 \\ \boldsymbol{\Lambda} \geq 0 \end{cases} ; \end{array} \tag{10.5}$$

where $\boldsymbol{y}$ is the vector containing the example classification, and D is a symmetric $\ell \times \ell$ matrix with elements $D_{ij} = y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j$.

The vectors $\boldsymbol{x}_i$ for which $\lambda_i > 0$ are called **support vectors**. In other words, support vectors are the ones for which the constraints in (10.5) are active. If $\boldsymbol{w}^*$ is the optimal value of $\boldsymbol{w}$, the value of b at the optimal solution can be computed as $b^* = y_i - \boldsymbol{w}^* \cdot \boldsymbol{x}_i$ for any support vector $\boldsymbol{x}_i$, and the classification function can be written as

$$f(\boldsymbol{x}) = \text{sign}\left(\sum_{i=1}^{\ell} y_i \lambda_i^* \boldsymbol{x}\cdot\boldsymbol{x}_i + b^*\right).$$

Note that the summation index can as well be restricted to support vectors, because all other vectors have null λ_i^* coefficients. The classification is determined by a linear combination of the classifications obtained on the examples y_i weighted according to the scalar product between input pattern and example pattern (a measure of the "similarity" between the current pattern and example $\boldsymbol{x}_i$) and by parameter λ_i^*.

10.1.2 Non-separable problems

If the hypothesis set is unchanged but the examples are not linearly separable a penalty proportional to the constraint violation ξ_i (collected in vector Ξ) can be introduced, solving the following problem:

$$\begin{array}{ll} \text{Minimize}_{\boldsymbol{w},b,\Xi} & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_{i=1}^{\ell}\xi_i\right)^k \\ \text{subject to} & \begin{cases} y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 - \xi_i & i = 1,\ldots,\ell \\ \xi_i \geq 0 & i = 1,\ldots,\ell \\ \|\boldsymbol{w}\|^2 \leq c_r; \end{cases} \end{array} \tag{10.6}$$

where the parameters C and k determine the cost caused by constraint violation, while c_r limits the norm of the coefficient vector. In fact, the first term to be minimized is related to the VC-dimension, while the second is related to the empirical risk. (See the above described SRM principle.) In our case, k is set to 1.

10.1.3 Nonlinear hypotheses

Extending the above techniques to nonlinear classifiers is based on mapping the input data $\boldsymbol{x}$ into a higher-dimensional vector of *features* $\boldsymbol{\varphi}(\boldsymbol{x})$ and using *linear* classification in the transformed space, called the *feature space*. The SVM classifier becomes:

$$f(\boldsymbol{x}) = \text{sign}\left(\sum_{i=1}^{\ell} y_i \lambda_i^* \boldsymbol{\varphi}(\boldsymbol{x}) \cdot \boldsymbol{\varphi}(\boldsymbol{x}_i) + b^*\right).$$

After introducing the *kernel function* $K(\boldsymbol{x}, \boldsymbol{y}) \equiv \boldsymbol{\varphi}(\boldsymbol{x}) \cdot \boldsymbol{\varphi}(\boldsymbol{y})$, the SVM classifier becomes

$$f(\boldsymbol{x}) = \text{sign}\left(\sum_{i=1}^{\ell} y_i \lambda_i^* K(\boldsymbol{x}, \boldsymbol{x}_i) + b^*\right),$$

and the quadratic optimization problem becomes:

$$\begin{array}{ll} \text{Maximize}_{\boldsymbol{\Lambda}} & \boldsymbol{\Lambda} \cdot \mathbf{1} - \frac{1}{2} \boldsymbol{\Lambda} \cdot D \cdot \boldsymbol{\Lambda} \\ \text{subject to} & \begin{cases} \boldsymbol{\Lambda} \cdot \boldsymbol{y} = 0 \\ 0 \leq \boldsymbol{\Lambda} \leq C\mathbf{1}, \end{cases} \end{array},$$

where D is a symmetric $\ell \times \ell$ matrix with elements $D_{ij} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

An extension of the SVM method is obtained by weighing in a different way the errors in one class with respect to the error in the other class, for example when the number of samples in the two classes is not equal, or when an error for a pattern of a class is more expensive than an error on the other class. This can be obtained by setting two different penalties for the two classes: C^+ and C^-. The function to minimize becomes:

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C^+(\sum_{i:y_i=+1}^{\ell} \xi_i)^k + C^-(\sum_{i:y_i=-1}^{\ell} \xi_i)^k.$$

If the feature functions $\boldsymbol{\varphi}(\boldsymbol{x})$ are chosen with care one can **calculate the scalar products without actually computing all features**, therefore greatly reducing the computational complexity.

The method used to avoid the explicit mapping is also called **kernel trick**. One uses learning algorithms that only require dot products between the vectors in the original input space, and chooses the mapping such that these high-dimensional dot products can be computed within the original space, by means of a **kernel function**.

For example, in a one-dimensional space a reasonable choice can be to consider monomials in the variable x multiplied by appropriate coefficients a_n:

$$\boldsymbol{\varphi}(x) = (a_0 1, a_1 x, a_2 x^2, \ldots, a_d x^d),$$

so that $\boldsymbol{\varphi}(x) \cdot \boldsymbol{\varphi}(y) = (1 + xy)^d$. In more dimensions, it can be shown that if the features are monomials of degree $\leq d$ then one can always determine coefficients a_n so that:

$$K(\boldsymbol{x}, \boldsymbol{y}) = (1 + \boldsymbol{x} \cdot \boldsymbol{y})^d.$$

The kernel function $K(\cdot, \cdot)$ is a convolution of the canonical inner product in the feature space. Common kernels for use in a SVM are the following.

1. Dot product: $K(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} \cdot \boldsymbol{y}$; in this case no mapping is performed, and only the optimal separating hyperplane is calculated.
2. Polynomial functions: $K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y} + 1)^d$, where the *degree* d is given.
3. Radial basis functions (RBF), like Gaussians: $K(\boldsymbol{x}, \boldsymbol{y}) = e^{-\gamma \|\boldsymbol{x} - \boldsymbol{y}\|^2}$ with parameter γ.
4. Sigmoid (or neural) kernel: $K(\boldsymbol{x}, \boldsymbol{y}) = \tanh(a \boldsymbol{x} \cdot \boldsymbol{y} + b)$ with parameters a and b.
5. ANOVA kernel: $K(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} e^{-\gamma (x_i - y_i)} \right)^d$, with parameters γ and d.

When ℓ becomes large the quadratic optimization problem requires a $\ell \times \ell$ matrix for its formulation, so it rapidly becomes an unpractical approach as the training set size grows. A decomposition method where the optimization problem is split into an active and an inactive set is introduced in [26]. The work in [23] introduces efficient methods to select the working set and to reduce the problem by taking advantage of the small number of support vectors with respect to the total number of training points.

10.1.4 Support Vectors for regression

Support vector methods can be applied also for regression, i.e., to estimate a function $f(\boldsymbol{x})$ from a set of training data $\{(\boldsymbol{x}_i, y_i)\}$. As it was the case for classification, one starts from the case of linear functions and then preprocesses the input data $\boldsymbol{x}_i$ into an appropriate feature space to make the resulting model nonlinear.

In order to fix the terminology, the linear case for a function $f(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b$ can be summarized. The convex optimization problem to be solved becomes:

$$\begin{array}{ll} \text{Minimize}_{\boldsymbol{w}} & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{subject to} & \begin{cases} y_i - (\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \leq \varepsilon \\ (\boldsymbol{w} \cdot \boldsymbol{x}_i + b) - y_i \leq \varepsilon, \end{cases} \end{array}$$

assuming the existence of a function that approximates all pairs with ε precision.

If the problem is not feasible, a *soft margin* loss function with slack variables ξ_i, ξ_i^*, collected in vector $\boldsymbol{\Xi}$, is introduced in order to cope with the infeasible constraints, obtaining the following formulation:

$$\begin{array}{ll}\text{Minimize}_{\boldsymbol{w},b,\boldsymbol{\Xi}} & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_{i=1}^{\ell}\xi_i^* + \sum_{i=1}^{\ell}\xi_i\right) \\ \text{subject to} & \begin{cases} y_i - \boldsymbol{w}\cdot\boldsymbol{x}_i - b \le \varepsilon - \xi_i^* & i = 1,\dots,\ell \\ \boldsymbol{w}\cdot\boldsymbol{x}_i + b - y_i \le \varepsilon - \xi_i & i = 1,\dots,\ell \\ \xi_i^* \ge 0 & i = 1,\dots,\ell \\ \xi_i \ge 0 & i = 1,\dots,\ell \\ \|\boldsymbol{w}\|^2 \le c_r. \end{cases}\end{array} \tag{10.7}$$

As in the classification case, C determines the tradeoff between the flatness of the function and the tolerance for deviations larger than ε. Detailed information about support vector regression can be found also in [32].

Gist

Statistical Learning Theory (SLT) states the conditions so that learning from examples is successful, i.e., such that positive results on training data translate into effective generalization on new examples produced by the same underlying probability distribution. The **constancy of the distribution** is critical: a good human teacher will never train students on some examples just to give completely different examples in the tests. In other words, the examples have to be representative of the problem. The conditions for learnability mean that the hypothesis space (the "flexible machine with tunable parameters" that we use for learning) must be sufficiently powerful to allow reaching a good performance on the training examples (a small *empirical risk*), but not too powerful to simply memorize the examples without extracting the deep structure of the problem. The flexibility is quantified by the VC-dimension.

SLT demonstrates the existence of the Paradise of Learning from Data but, for most practical problems, it does not show the practical steps to enter it, and appropriate choices of kernel and parameters though intuition and cross-validation are critical to the success.

The latest results on deep learning and MLPs open new hopes that the "feature engineering" and kernel selection step can be fully automated. Research has not reached an end to the issue, there is still space for new disruptive techniques and for following the wild spirits of creativity more than the lazy spirits of hype and popular wisdom.

Chapter 11

Democracy in machine learning

While in every republic there are two conflicting factions, that of the people and that of the nobles, it is in this conflict that all laws favorable to freedom have their origin.
(Machiavelli)

This is the final chapter in the supervised learning part. As you discovered, there are *many* competing techniques for solving the problem, and each technique is characterized by choices and meta-parameters: when this flexibility is taken into account, one easily ends up with **a very large number of possible models for a given task**.

When confronted with this abundance one may just select the best model (and best meta-parameters) and throw away everything else, or recognize that there's never too much of a good thing and try to use all of them, or at least the best ones. One already spends effort and CPU time to select the best model and meta-parameters, producing many models as a byproduct. Are there sensible ways to recycle them so that the effort is not wasted? Relax, this chapter does not introduce radically new models but deals

Latest chapter revision: November 14, 2013

with **using many different models in flexible, creative and effective ways**. The advantage is in some cases so clear that using many models will make a difference between winning and losing a competition in ML.

The *leitmotif* of this book is that many ML principles resemble some form of human learning. Asking a **committee of experts** is a human way to make important decisions, and committees work well if the participants have different competencies and comparable levels of professionalism. Diversity of background, culture, sex is assumed to be a critical component in successful innovative businesses. Democracy itself can be considered as a pragmatic way to pool knowledge from citizens in order to reach workable decisions (well, maybe not always optimal but for sure better than decisions by a single dictator).

We already encountered a creative usage of many classification trees as **classification forests** in Chapter 6 (Sec. 6.2). In this chapter we review the main techniques to make effective use of more and different ML models with a focus on the architectural principles and a hint at the underlying math.

11.1 Stacking and blending

If you are participating in a ML competition (or if you want to win a contract or a solution to a critical need in a business), chances are you will experiment with different methods and come up with a large set of models. Like for good coffee, blending them can bring higher quality.

The two straightforward ways to combine the outputs of the various models are by **voting** and by **averaging**. Imagine that the task is to classify patterns into two classes. In voting, each trained model votes for a class, votes are collected and the final output class is the one getting more votes, exactly as in a basic democratic process based on **majority**. If each model has a probability of correct classification greater than $1/2$, **and if the errors of the different models are uncorrelated, then the probability that the majority of M models will be wrong goes to zero** as the number of models grows[1]. Unfortunately errors tend to be *correlated* in practical cases (if a pattern is difficult to recognize, it will be difficult for *many* models, the probability that many of them will be wrong will be higher than the product of individual mistake probabilities — think about stained digits in a zip code on a letter) and the advantage will be less dramatic.

If the task is to predict a probability (a posterior probability for a class given the input pattern), averaging individual probabilities is another option. By the way, averaging the results of experimental measures is the standard way to **reduce variance**[2].

[1]The demonstration is simple by measuring the area under the binomial distribution where more than $M/2$ models are wrong.

[2]The "law of large numbers" in statistics is about explaining why, under certain conditions, the average of the results obtained from a large number of trials should be close to the expected value, and why it

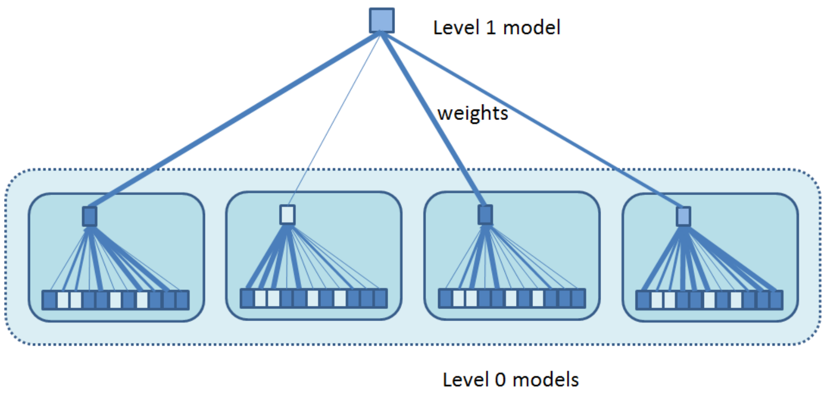


Figure 11.1: Blending different models by adding an additional model on top of them (stacking).

Although straightforward, averaging and voting share a weakness: they treat all models equally and the performance of the very best models can vanish amidst a mass of mediocre models. The more complex a decision, the more the different **experts have to be weighted, and often weighted in a manner which depends on the specific input**.

You already have a hammer for nailing also this issue of weighting experts: machine learning itself! Just add another linear model on top, connect the different outputs by the level-0 models (the experts) and let ML identify the optimal weights (Fig. 11.1). This is the basic idea of **stacked generalization** [39]. To avoid overtraining one must take care that the training examples used for training the stacked model (for determining the weights of the additional layer on top) were *never* used before for training the individual models. Training examples are like fish: they stink if you use them for too long!

Results in stacked generalization are as follows [35]:

- When you can, **use class probabilities** as outputs of the original level-0 models (instead of class predictions). Estimates of probabilities tell something about the *confidence*, and not just the prediction. Keeping them will give more information to the higher level.

- Ensure **non-negative weights** for the combination by adding constraints in the optimization task. They are necessary for stacked regression to improve accuracy. They are not necessary for classification task, but in both cases they increase the **interpretability** of the level-1 model (a zero weight means that the corresponding 0-level model is not used, the higher the weight, the more important the model).

tends to become closer as more trials are performed.

If your appetite is not satisfied, you can experiment with more than one level, or with more structured combination. For example you can stack a level on top of MLPs and decision forests, or combine a stacked model already done by a group with your model by adding yet another level (Fig. 11.2). The more models you manage, the more careful you have to be with the "stinking example" rule above. The higher-level models do not need to be linear: some interpretability will be lost, but the final results can be better with nonlinear combining models.

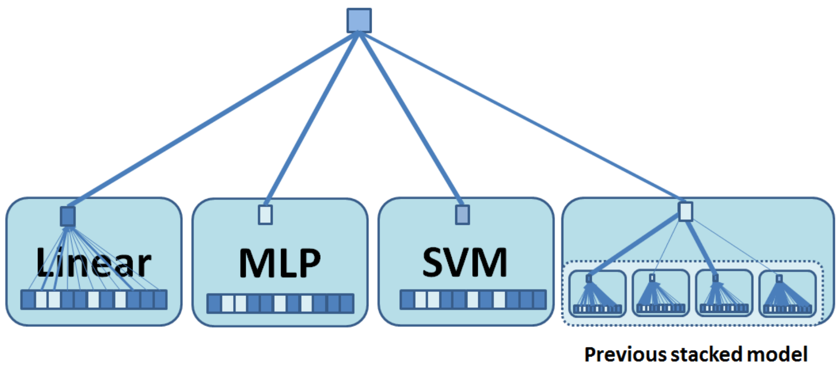


Figure 11.2: Stacking can be applied to different models, including previously stacked ones.

An interesting option is the **feature-weighted linear stacking** [31]. We mention it as an example of how a specific real-world application can lead to an elegant solution. In some cases, one has some additional information, called "**meta-features**" in addition to the raw input features. For example, if the application is to predict preferences for customers for various products (in a *collaborative filtering and recommendation* context), the reliability of a model can vary depending on the additional information. For example, a model A may be more reliable for users who rated many products (in this case, the number of products rated by the user is the "meta-feature"). To maintain linear regression while allowing for weights to *depend* on meta-features (so that model A can have a larger weight when used for a customer who rated more products), one can ask for weights to be *linear in the meta-features*. If $g_i(\boldsymbol{x})$ is the output for level-0 model i and $f_j(\boldsymbol{x})$ is the j-th meta-feature, weights will be

$$w_i(\boldsymbol{x}) = \sum_j v_{ij} f_j(\boldsymbol{x}),$$

where v_{ij} are the parameters to be learned by the stacked model. The level-1 output will be

$$b(\boldsymbol{x}) = \sum_{i,j} v_{ij} f_j(\boldsymbol{x}) g_i(\boldsymbol{x}),$$

leading to the following **feature-weighted linear stacking** problem:

$$\min_{(v_{ij})} \sum_{\boldsymbol{x}} \sum_{i,j} \big(v_{ij} f_j(\boldsymbol{x}) g_i(\boldsymbol{x}) - y(\boldsymbol{x})\big)^2.$$

Because the model is still linear in v, we can use standard linear regression to identify the optimal v. As usual, never underestimate the power of linear regression if used in proper creative ways.

11.2 Diversity by manipulating examples (bagging and boosting)

For a successful democratic systems in ML one needs **a set of accurate and diverse classifiers**, or regressors, also called **ensemble**, like a group of musicians who perform together. **Ensemble methods** is the traditional term for these techniques in the literature, **multiple-classifiers systems** is a synonym.

Different techniques can be organized according to the main way in which they create diversity [13].

Training models on **different subsets of training examples** is a possibility. In **bagging** (”bootstrap aggregation”), different subsets are created by random sampling *with replacement* (the same example can be extracted more than once). Each bootstrap replica contains about two thirds (actually $\approx 63.2\%$) of the original examples. The results of the different models are then aggregated, by averaging, or by majority rules. Bagging works well to improve *unstable* learning algorithms, whose results undergo major changes in response to small changes in the learning data. As described in Chapter 6 (Section 6.2), bagging is used to produce **classification forests** from a set of classification trees.

Cross-validated committees prepare different training sets by leaving out disjoint subsets of training data. In this case the various models are the side-effect of using cross-validation as ingredient in estimating a model performance (and no additional CPU is required).

A more dynamic way of manipulating the training set is via **boosting**. The term has to do with the fact that weak classifiers (although with a performance which must be slightly better than random) can be “boosted” to obtain an accurate committee [16]. Like bagging, boosting creates multiple models, but the model generated at each iteration is built in an **adaptive** manner, to directly improve the combination of previously created models. The algorithm `AdaBoost` maintains a set of weights over the training examples. After each iterations weights are updated so that **more weight is given to the examples which are misclassified** by the current model (Fig. 11.3). Think about a professional teacher, who is organizing the future lessons to insist more on the cases which were not already understood by the students.

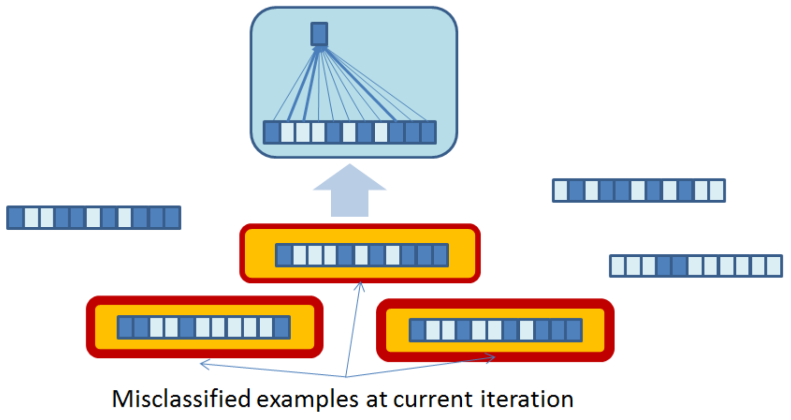


Figure 11.3: In boosting, misclassified examples at the current iteration are given more weight when training an additional model to be added to the committee.

The final classifier $h_f(\boldsymbol{x})$ is given by a weighted vote of the individual classifiers, and the weight of each classifier reflects its accuracy on the weighted training set it was trained on:

$$h_f(\boldsymbol{x}) = \sum_l w_l h_l(\boldsymbol{x}).$$

Because we are true believers in the power of optimization, the best way to understand boosting is by the function it optimizes. Different variations can then be obtained (and understood) by changing the function to be optimized or by changing the detailed optimization scheme. To define the error function, let's assume that the outputs y_i of each training example are $+1$ or -1. The quantity $m_i = y_i h(\boldsymbol{x}_i)$, called the *margin* of classifier h on the training data, is positive if the classification is correct, negative otherwise. As explained later in Section 11.6, `AdaBoost` can be viewed as a **stage-wise algorithm** for minimizing the following error function:

$$\sum_i \exp\left(-y_i \sum_l w_l h_l(\boldsymbol{x}_i)\right), \tag{11.1}$$

the negative exponential of the margin of the weighted voted classifier. This is equivalent to **maximizing the margin on the training data**.

11.3 Diversity by manipulating features

Different **subsets of features** can be used to train different models (Fig. 11.4). In some cases, it can be useful to group features according to different characteristics. In [11] this

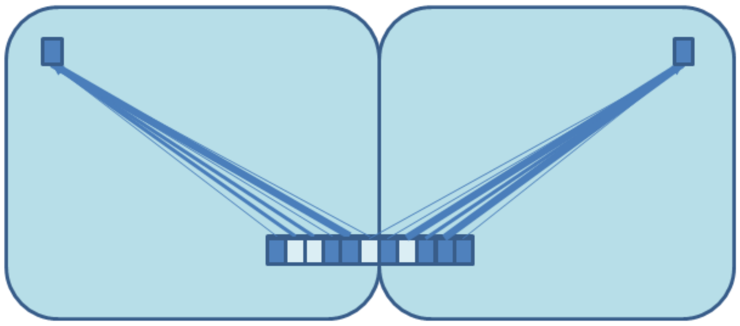


Figure 11.4: Using different subsets of features to create different models. The method is not limited to linear models.

method was used to identify volcanoes on Venus with human expert-level performance. Because the different models need to be accurate, using subsets of features works only when input features are highly redundant.

11.4 Diversity by manipulating outputs (error-correcting codes)

Error-correcting codes (ECC) are designed so that they are robust w.r.t. a certain number of mistakes during transmission by noisy lines (Fig. 11.5). For example, if the codeword for "one" is "`111`" and that for "zero" is "`000`", a mistake in a bit like in "`101`" can be accepted (the codeword will still be mapped to the correct "`111`"). **Error-correcting output coding** is proposed in [14] for designing committees of classifiers.

The idea of applying ECC to design committees is that each output class j is encoded as an L-bit codeword C_j. The l-th trained classifier in the committee has the task to predict the l-th bit of the codeword. After generating all bits by the L classifiers in the committee, the output class is the one with the closest codeword (in Hamming distance, i.e., measuring the number of different bits). Because codewords are redundant, a certain number of mistakes made by some individual classifiers can be corrected by the committee.

As you can expect, the different ensemble methods can be combined. For example, error-correcting output coding can be combined with boosting, or with feature selection, in some cases with superior results.

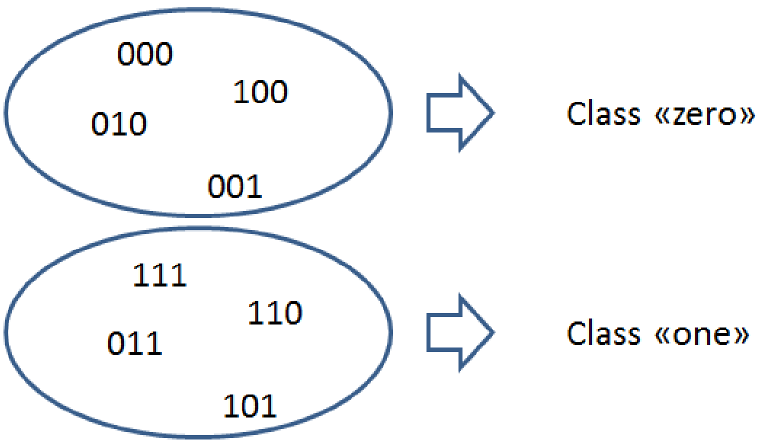


Figure 11.5: In error-correcting codes a redundant encoding is designed to resist a certain number of mistaken bits.

11.5 Diversity by injecting randomness during training

Many training techniques have randomized steps in their bellies. This randomness is a very natural way to obtain diverse models (by changing the seed of the random number generator). For example, MLP starts from randomize initial weights. Tree algorithms can decide in a randomized manner the next feature to test in an internal node, as it was already described for obtaining decision forests.

Last but not least, most optimization methods used for training have space for randomized ingredients. For example, stochastic gradient descent presents patterns in a randomized order.

11.6 Additive logistic regression

We just encountered boosting as a way of sequentially applying a classification algorithm to re-weighted versions of the training examples and then taking a weighted majority vote of the sequence of models thus produced.

As an example of the power of optimization, boosting can be interpreted as a way to apply **additive logistic regression, a method for fitting an additive model $\sum_m h_m(\boldsymbol{x})$ in a forward stage-wise manner** [17].

Let's start from simple functions

$$h_m(\boldsymbol{x}) = \beta_m b(\boldsymbol{x}; \boldsymbol{\gamma}_m),$$

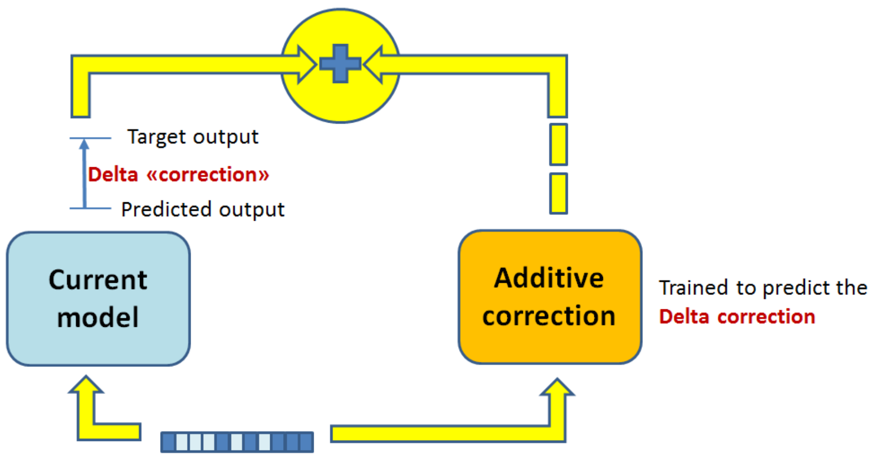


Figure 11.6: Additive model step: the error of the current models on the training examples is measured. A second model is added aiming at canceling the error.

each characterized by a set of parameters $\boldsymbol{\gamma}_m$ and a multiplier β_m acting as a weight. One can build an additive model composed of M such functions as:

$$H_M(\boldsymbol{x}) = \sum_{m=1}^{M} h_m(\boldsymbol{x}) = \sum_{m=1}^{M} \beta_m b(\boldsymbol{x}; \boldsymbol{\gamma}_m).$$

With a **greedy forward stepwise approach** one can identify at each iteration the best parameters $(\beta_m, \boldsymbol{\gamma}_m)$ so that **the newly added simple function tends to correct the error of the previous model** $F_{m-1}(\boldsymbol{x})$ (Fig. 11.6). If least-squares is used as a fitting criterion:

$$(\beta_m, \boldsymbol{\gamma}_m) = \underset{(\beta,\boldsymbol{\gamma})}{\arg\min}\, E\left[\left(y - F_{m-1}(\boldsymbol{x}) - \beta b(\boldsymbol{x}; \boldsymbol{\gamma})\right)^2\right], \tag{11.2}$$

where $E[\cdot]$ is the expected value, estimated by summing over the examples. This greedy procedure can be generalized as **backfitting**, where one iterates by fitting one of the parameters couple $(\beta_m, \boldsymbol{\gamma}_m)$ at each step, not necessarily the last couple. Let's note that this method only requires an algorithm for fitting a *single* weak learner $\beta b(\boldsymbol{x}; \boldsymbol{\gamma})$ to data, which is applied repeatedly to modified versions of the original data (Fig. 11.7):

$$y_m \leftarrow y - \sum_{k \neq m} h_k(\boldsymbol{x}).$$

For classification problems, using the squared-error loss (with respect to ideal 0 or 1 output values) leads to trouble. If one would like to estimate the posterior probability $\Pr(y = j|\boldsymbol{x})$, there is no guarantee that the output will be limited in the $[0, 1]$ range.

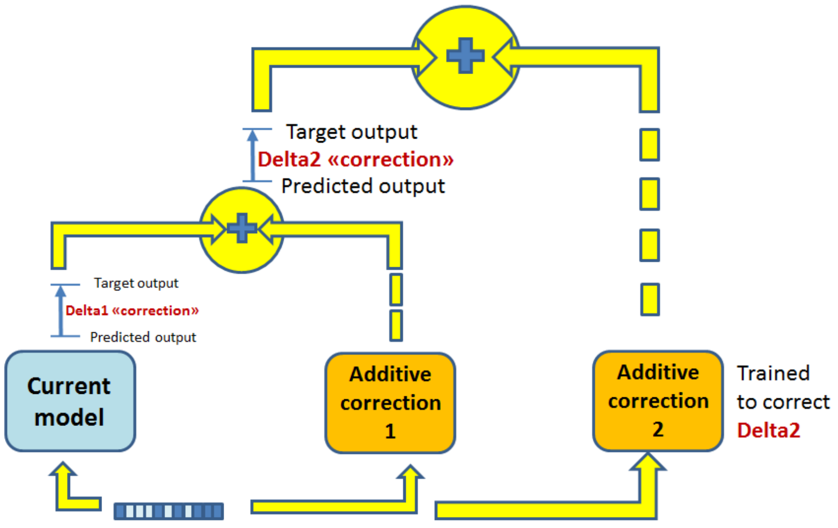


Figure 11.7: The greedy forward step-wise approach in additive models, one iterates by adding new components to cancel the remaining error.

Furthermore, the squared error penalizes not only real errors (like predicting 0 when 1 is requested), but also it **penalizes classifications which are "too correct"** (like predicting 2 when 1 is required).

Logistic regression comes to the rescue (Section 8.1): one uses the additive model $H(\boldsymbol{x})$ to predicting an "intermediate" value, which is then *squashed* onto the correct $[0,1]$ range by the logistic function to obtain the final output in form of a probability.

An additive logistic model has the form

$$\ln \frac{\Pr(y=1|\boldsymbol{x})}{\Pr(y=-1|\boldsymbol{x})} = H(\boldsymbol{x}),$$

where the *logit* transformation on the left monotonically maps probability $\Pr(y = 1|\boldsymbol{x}) \in [0,1]$ onto the whole real axis. Therefore, the logit transform, together with its inverse

$$\Pr(y=1|\boldsymbol{x}) = \frac{e^{H(\boldsymbol{x})}}{1+e^{H(\boldsymbol{x})}}, \tag{11.3}$$

guarantees probability estimates in the correct $[0,1]$ range. In fact, $H(\boldsymbol{x})$ is modeling the *input* of the logistic function in Eq. (11.3).

Now, if one considers the expected value $E\left[e^{-yH(\boldsymbol{x})}\right]$, one can demonstrate that this quantity is minimized when

$$H(x) = \frac{1}{2} \ln \frac{\Pr(y=1|\boldsymbol{x})}{\Pr(y=-1|\boldsymbol{x})},$$

i.e., the symmetric logistic transform of $\Pr(y = 1|\boldsymbol{x})$ (note the factor $1/2$ in front). The interesting result is that `AdaBoost` builds an additive logistic regression model via Newton-like updates[3] for minimizing $E\left[e^{-yH(\boldsymbol{x})}\right]$. The technical details and additional explorations are in the original paper [17].

11.7 Democracy for better accuracy-rejection compromises

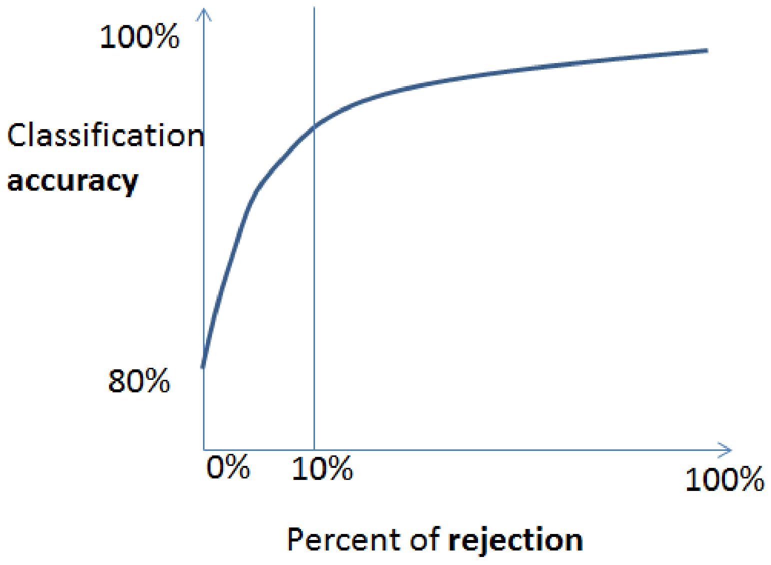


Figure 11.8: The accuracy-rejection compromise curve. Better accuracy can be obtain by rejecting some difficult cases.

In many practical applications of pattern recognition systems there is another "knob" to be turned: the fraction of cases to be rejected. **Rejecting some difficult cases and having a human person dealing with them** (or a more complex and costly second-level system) can be better than accepting and classifying everything. As an example, in optical character recognition (e.g., zip code recognition), difficult cases can arise because of bad writing or because of segmentation and preprocessing mistakes. In these

[3]Optimization with Newton-like steps, as it will become clear in the future chapters, means that a quadratic approximation in the parameters is derived, and the best parameters are obtained as the minimum of the quadratic model.

cases, a human expert may come up with a better classification, or may want to look at the original postcard in the case of a preprocessing mistake. Let's assume that the ML system has this additional knob to be turned and some cases *can* be rejected. One comes up with an **accuracy-rejection curve** like the one in Fig. 11.8, describing the attainable accuracy performance as a function of the rejection rate. If the system is working in an intelligent manner, **the most difficult and undecided cases will be rejected first**, so that the accuracy will rapidly increase even for modest rejection rates [4].

For simplicity, let's consider a two-class problem, and a trained model with an output approximating the **posterior probability** for class 1. If the output is close to one, the decision is clear-cut, and the correct class is 1 with high probability. A problem arises if the output is close to 0.5. In this case the system is "undecided". If the estimated probability is close to 0.5, the two classes have a similar probability and mistakes will be frequent (if probabilities are correct, the probability of mistake is equal to 0.5 in this case). If the correct probabilities are known, the theoretically best **Bayesian classifier** decides for class 1 if $P(class = 1|x)$ is greater than $1/2$, for class 0 otherwise. The mistakes will be equal to the remaining probability. For example, if $P(class = 1|x)$ is 0.8, mistakes will be done with probability 0.2 (the probability that cases of class two having a certain x value are classified as class 1).

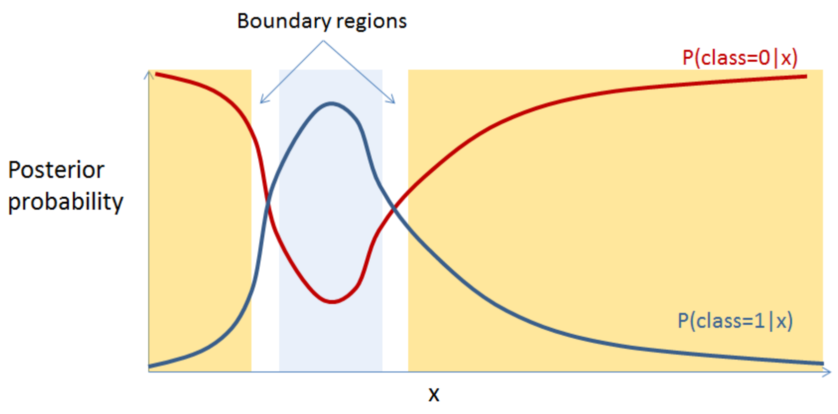


Figure 11.9: Transition regions in a Bayesian classifier. If the input patterns falling in the transition areas close to the boundaries are rejected, the average accuracy for the accepted cases increases.

[4]A related "tradeoff" curve in signal detection is the *receiver operating characteristic (ROC)*, a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the total actual positives (TPR = true positive rate) vs. the fraction of false positives out of the total actual negatives (FPR = false positive rate), at various threshold settings.

Setting a positive threshold T on the posterior probability, demanding it to be greater than $(1/2 + T)$ is the best possible "knob" to increase the accuracy by rejecting the cases which do not satisfy this criterion. One is rejecting the patterns which are close to the boundary between the two classes, where cases of both classes are mixed with probability close to $1/2$ (Fig. 11.9).

Now, if probabilities are not known but are *estimated* through machine learning, having a committee of classifiers gives many opportunities to obtain more flexibility and realize superior accuracy-rejection curves [6]. For example, one can get **probabilistic combinations of teams**, or portfolios with more classifiers, by activating each classifier with a different probability. One can consider the agreement between all of them, or a **qualified majority**, as a signal of cases which can be assigned with high confidence, and therefore to be accepted by the system. Finally, even more flexibility can be obtained by considering the output probabilities (not only the classifications), averaging and thresholding. If there are more than two classes, on can require that the average probability is above a first threshold, while the distance with the second best class is above a second threshold.

Experiments show that superior results and higher levels of flexibility in the accuracy-rejection compromise can be easily obtained, by reusing in an intelligent manner the many classifiers which are produced in any case while solving a task.

Gist

Having a number of **different** but similarly **accurate** machine learning models allows for many ways of increasing performance beyond that of the individual systems (**ensemble methods, committees, democracy in ML**).

In **stacking or blending** the systems are combined by adding another layer working on top of the outputs of the individual models.

Different ways are available to create diversity in a strategic manner. In **bagging (bootstrap aggregation)**, the same set of examples is sampled with replacement. In **boosting**, related to additive models, a series of models is trained so that the most difficult examples for the current system get a **larger weight** for the latest added component. Using different subsets of features or different random number generators in randomized schemes are additional possibilities to create diversity. **Error-correcting output codes** use a redundant set of models coding for the various output bits to increase robustness with respect to individual mistakes.

Additive logistic regression is an elegant way to explain boosting via additive models and Newton-like optimization schemes. Optimization is boosting our knowledge of boosting.

Ensemble methods in machine learning resemble jazz music: the whole is greater than the sum of its parts. Musicians and models working together, feeding off one another, create more than they would by themselves.

Part II

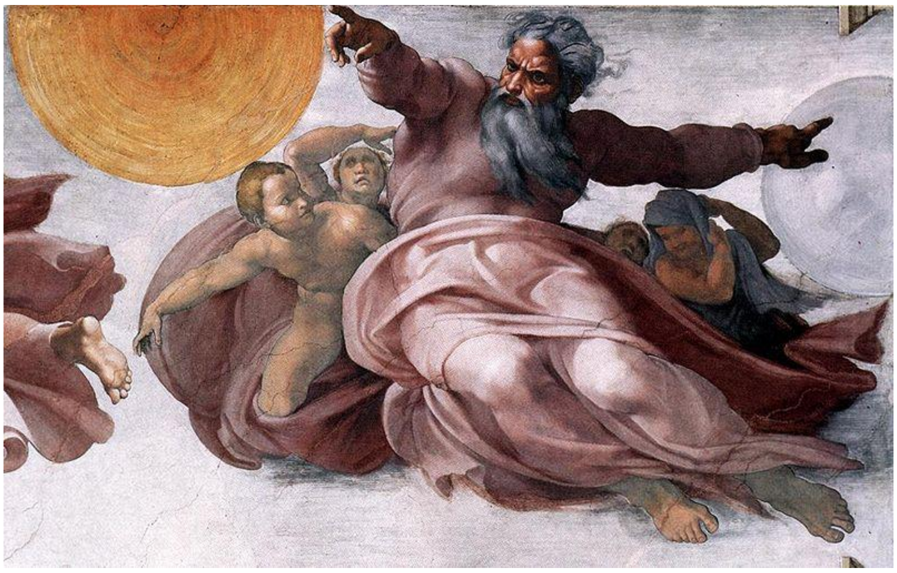
Unsupervised learning and clustering

Latest chapter revision: November 14, 2013

Chapter 12

Top-down clustering: K-means

First God made heaven and earth. The earth was without form and void, and darkness was upon the face of the deep; and the Spirit of God was moving over the face of the waters. And God said, "Let there be light"; and there was light. And God saw that the light was good; and God ***separated the light from the darkness. God called the light Day, and the darkness he called Night****. [...]*
So out of the ground the Lord God formed every beast of the field and every bird of the air, and brought them to the man to see what he would call them; and whatever the man called every living creature, that was its name. ***The man gave names*** *to all cattle, and to the birds of the air, and to every beast of the field.*
(Book of Genesis)



This chapter starts a new part of the book and enters a new territory. Up to now we considered *supervised* learning methods, while the issue of this part is: **What can be learnt without teachers and labels**?

Like the energy emanating in the above painting by Michelangelo suggests, we are entering a more creative region, which contains concepts related to exploration, discovery, different and unexpected outcomes. The task is not to slavishly follow a teacher but to gain freedom in generating models. In most cases the freedom is not desired but it is the only way to proceed.

Let's imagine you place a child in front of a television screen. Even without a teacher he will immediately differentiate between a broken screen, showing a "snowy" random noise pattern, and different television programs like cartoons and world news. Most probably, he will show more excitement for cartoons than for world news and for random noise. The appearance of a working TV screen (and the appearance of the world) is not random, but highly structured, arranged according to explicit or implicit plans. For another example of unsupervised learning, let's assume that entities represent speakers of different languages, and coordinates are related to audio measurements of their spoken language (such as frequencies, amplitudes, etc.). While walking in an international airport, most people can readily identify clusters of different language speakers based on the audible characteristics of the language. We may for example easily distinguish English speakers from Italian speakers, even if we cannot name the language being spoken.

Modeling and understanding structure (forms, patterns, clumps of interesting events) is at the basis of our cognitive abilities. The use of *names* and language is deeply rooted in the organizing capabilities of our brain. In essence, a name is a way to group different experiences so that we can start speaking and reasoning. Socrates is a *man*, all men are *mortal*, therefore Socrates is mortal.

For example, animal species (and the corresponding names) are introduced to reason about **common characteristics** instead of individual ones ("The man gave names to all cattle"). In geography, continents, countries, regions, cities, neighborhoods represent clusters of geographical entities at different scales. Clustering is related to the very human activity of **grouping similar things together, abstracting** them and giving names to the classes of objects (Fig. 12.1). Think about categorizing men and women, a task we perform with a high degree of confidence in spite of significant individual variation.

Clustering has to do with **compression of information**. When the amount of data is too much for a human to digest, **cognitive overload** results and the finite amount of "working memory" in our brain is insufficient to handle the task. Actually, the number of data points chosen for analysis can be reduced by using filters to restrict the range of data values. But this is not always the best choice, as in this case we are filtering data based on individual coordinates, while **a more global picture** may be preferable.

Clustering methods work by collecting similar points together in an intelligent and data-driven manner, so that one's attention can be concentrated on a small but relevant set of **prototypes**. The prototype summarizes the information contained in the subset of cases which it represents. When similar cases are grouped together, one can reason about groups instead of individual entities, therefore reducing the number of different possibilities.

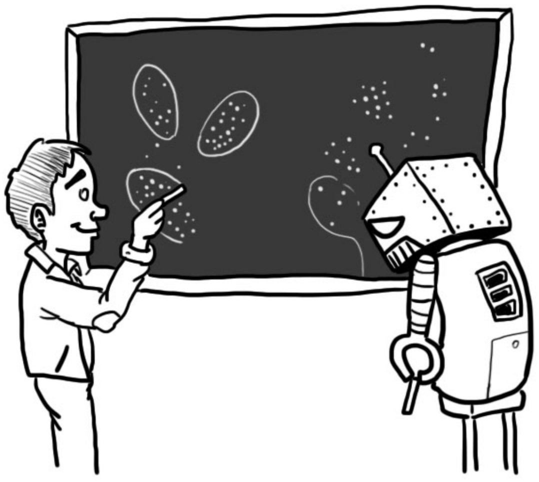


Figure 12.1: Clustering is deeply rooted into the human activity of grouping and naming entities.

As you imagine, the practical applications of clustering are endless. To mention some examples, in **market segmentation** one divides a broad target market into subsets of consumers who have common needs, and then implements strategies to target their needs and desires. In **finance**, clustering is used to group stocks with a similar behavior, for diversifying the portfolio and reducing risk. In **health care**, diseases are clusters of abnormal conditions that affects our body. In **text mining**, different words are grouped together based on the structure and meaning of the analysed texts. A **semantic network** can represents semantic relations between concepts. It is a directed or undirected graph consisting of vertices, which represent concepts, and edges. The different relationships (e.g., ”is an”, ”has”, ”lives in”, ...) underline that there is no single way to group entities.

12.1 Approaches for unsupervised learning

Given the creativity inherent in clustering, you expect wildly different ways to proceed. It is traditional to subdivide methods into top-down and bottom-up techniques.

In **top-down or divisive clustering** one decides about a number of classes and then proceeds to separate the different cases into the classes, aiming at putting similar cases together. Let’s note that classes do not have labels, only the subdivision matters. Think about organizing your laundry in a cabinet with a fixed number of drawers. If you are an adult person (if you are a happy teenager, please ask your mother or father) you will probably end up putting socks with similar socks, shirts with shirts, etc.

In **bottom-up or agglomerative clustering** one leaves the data speak for themselves and starts by merging (associating) the most similar items. As soon as larger groupings of items are created, one proceeds by merging the most similar groups, and so on. The process is stopped when the grouping makes sense, which of course depends on the specific metric, application area and user judgment. The final result will be a hierarchical organization of larger and larger sets (known as *dendrogram*), reflecting the progressively larger mergers. Dendrograms should be familiar from natural sciences, think about the organization of zoological or botanical species.

More advanced and flexible unsupervised strategies are known under the umbrellas of **dimensionality reduction**: in order to reduce the number of coordinates to describe a set of experimental data one needs to understand the structure and the ”directions of variation” of the different cases. If one is clustering people faces, the directions of variation can be related to eyes color, distance between nose and mouth, distance between nose and eyes, etc. All faces can be obtained by changing some tens of parameters, for sure much less than the total number of pixels in an image.

Another way to model a set of cases is to assume that they are produced by un underlying **probabilistic process**, so that modeling the process becomes a way to understand the structure (and the different clusters, if any). **Generative models** aim at identifying and modeling the probability distributions of the process producing the observed examples. Think about grouping books by deriving a model for the topics and words used by different authors (without knowing the author names beforehand). An author will pick topics with a certain probability. After the topic is fixed, words related to the topic will be generated with specific probabilities. For sure, the process will not generate masterpieces but similar final word probabilities, in many cases sufficient to recognize an unknown author.

Our visual system is extremely powerful at clustering salient parts of an image and **visualizations** like linear or nonlinear **projections** onto low-dimensional spaces (usually with two dimensions) can be very effective to identify structure and clusters ”by hand”, well, actually ”by eyes”.

Last but not least, very interesting and challenging applications require a mixture of supervised and unsupervised strategies (**semi-supervised learning**). Think about ”big data” applications related to clustering zillions of web pages. Labels can be very costly (because they can require a human person to classify the page) and therefore rare. By adding a potentially huge set of unlabeled pages to the scarce labeled set one can greatly improve the final result.

After clarifying the overall landscape, this chapter will focus on the popular and effective top-down technique known as **k-means clustering**.

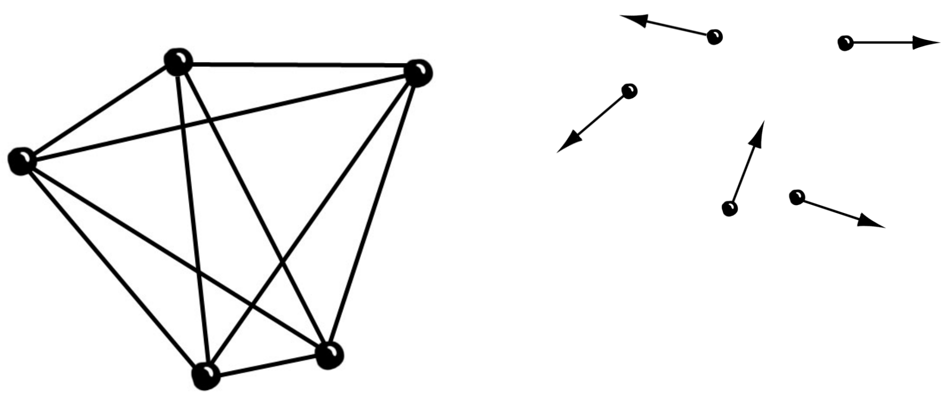


Figure 12.2: External representation by relationships (left) and internal representation with coordinates (right). In the first case mutual similarities between pairs are given, in the second case individual vectors.

12.2 Clustering: Representation and metric

There are two different contexts in clustering, depending on how the entities to be clustered are organized (Fig. 12.2). In some cases one starts from an **internal representation** of each entity (typically an M-dimensional vector $\mathbf{x}_d$ assigned to entity d) and derives mutual dissimilarities or mutual similarities from the internal representation. In this case one can derive **prototypes (or centroids)** for each cluster, for example by averaging the characteristics of the contained entities (the vectors). In other cases only an **external representation** of dissimilarities is available and the resulting model is an undirected and weighted graph of entities connected by edges.

For example, imagine that market research for a supermarket indicates that stocking similar foods on nearby shelves results in more revenue. An internal representation of a specific food can be a vector of numbers describing: type of food (1=meat, 2=fish…), calorie content, color, box dimension, suggested age of consumption, etc. Similarities can then be derived by comparing the vectors, by Euclidean metric or scalar products.

An external representation can instead be formed by polling the customers, asking them to rate the similarity of pairs of product X and Y (on a fixed scale, for example from 0 to 10), and then deriving external similarities by averaging the customer votes.

The effectiveness of a clustering method depends on the **similarity metric** (how to measure similarities) which needs to be strongly problem-dependent. The traditional Euclidean metric is in some cases appropriate when the different coordinates have a similar range and a comparable level of significance, it is not if different units of measure are used. For example, if a policeman compares faces by measuring eyes distance in millimeters and mouth-nose distance in kilometers, he will make the Euclidean metric almost meaningless. Similarly, if some data in the housing market represent the house color, it will not be significant when clustering houses for business purposes. Instead, the color can be extremely significant when clustering paintings of houses by different artists. The metric is indeed problem-specific, and this is why we denote the dissimilarity between entities $\mathbf{x}$ and $\mathbf{y}$ by $\delta(\mathbf{x}, \mathbf{y})$, leaving to the implementation to specify how it is calculated.

If an internal representation is present, a metric can be derived by the usual **Euclidean distance**:

$$\delta_E(\mathbf{x}, \mathbf{y}) = \|\mathbf{y} - \mathbf{x}\| = \sqrt{\sum_{i=1}^{M} (x_i - y_i)^2}. \tag{12.1}$$

In three dimensions this is the traditional distance, measured by squaring the edges and taking the square root. The notation $\|\mathbf{x}\|_2$ for a vector $\mathbf{x}$ means the Euclidean norm, and the subscript 2 is usually omitted.

Another notable norm is the Manhattan or taxicab norm, so called because it measures the distance a taxi has to drive in a rectangular street grid to get from the origin to the point x:

$$d_{ij}^{\text{Manhattan}} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_1 = \sum_{k=1}^{n} |\boldsymbol{x}_{ik} - \boldsymbol{x}_{j_k}|. \tag{12.2}$$

As usual, there is no absolutely right or wrong norm: for each problem a norm must reflect appropriate ways to measure distances. Taxicabs in New York prefer the Manhattan norm, while airplane pilots prefer the Euclidean norm (at least for short distances, then the curvature of earth requires still different distance measures used in geodetics). In some cases, the appropriate way to measure distances is recognized only *after* judging the clustering results, which makes the effort creative and open-ended.

Another possibility is that of starting from similarities given by scalar products of the two normalized vectors and then taking the inverse to obtain a dissimilarity. In detail, the normalized scalar product between vectors $\mathbf{x}$ and $\mathbf{y}$, by analogy with geometry in two and three dimensions, can be interpreted as the *cosine* of the angle between them, and is therefore known as **cosine similarity**:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{\sum_{i=1}^{M} x_i \times y_i}{\sqrt{\sum_{i=1}^{M} (x_i)^2} \times \sqrt{\sum_{i=1}^{M} (y_i)^2}}, \tag{12.3}$$

and after taking the inverse one obtains the dissimilarity: $\delta(\mathbf{x}, \mathbf{y}) = \|\mathbf{x}\|\|\mathbf{y}\|/(\epsilon + \mathbf{x} \cdot \mathbf{y})$, ϵ being a small quantity to avoid dividing by zero.

Let's note that the cosine similarity depends only on the *direction* of the two vectors, and it does not change if components are multiplied by a fixed number, while the Euclidean distance changes if one vector is multiplied by a scalar value. A weakness of the standard Euclidean distance is that values for different coordinates can have very different ranges, so that the distance may be dominated by a subset of coordinates. This can happen if units of measures are picked in different ways, for example if some coordinates are measured in millimeters, other in kilograms, other in kilometers: it is always very unpleasant if the analysis crucially depends on picking a suitable set of physical units. To avoid this trouble we need to make values *dimensionless*, without physical units. Furthermore we may as well normalize them so that all values range between zero and one before measuring distances.

The above can be accomplished by defining:

$$\delta_{norm}(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{i=1}^{M}\left(\frac{x_i - y_i}{\mathrm{maxval}_i - \mathrm{minval}_i}\right)^2}, \tag{12.4}$$

where M is the number of coordinates, minval_i and maxval_i are the minimum and maximum values achieved by coordinate i for all entities.

In the general case, a positive-definite matrix $\mathbf{M}$ can be determined to transform the original metric:

$$d_{ij} = \sqrt{(\boldsymbol{x}_i - \boldsymbol{x}_j)^T \mathbf{M} (\boldsymbol{x}_i - \boldsymbol{x}_j)}.$$

An example is the Mahalanobis distance which takes into account the correlations of the data set and is scale-invariant (it does not change if we measure quantities with different units, like millimeters or kilometers). More details about the Mahalanobis distance will be discussed in future chapters.

12.3 K-means for hard and soft clustering

The **hard clustering problem** consists of partitioning the entities D into k disjoint subsets $C = \{C_1, \ldots, C_k\}$ to reach the following objectives.

- Minimization of the average intra-cluster dissimilarities:

$$\min \sum_{d_1,d_2 \in C_i} \delta(\mathbf{x}_{d_1}, \mathbf{x}_{d_2}). \tag{12.5}$$

If an internal representation is available, the cluster centroids $\mathbf{p}_i$ can be derived as the average of the internal representation vectors over the members of the i-th cluster $\mathbf{p}_i = (1/|C_i|)\sum_{d\in C_i}\mathbf{x}_d$.

In these cases the intra-cluster distances can be measured with respect to the cluster centroid $\mathbf{p}_i$, obtaining the related but different minimization problems:

$$\min \sum_{d\in C_i} \delta(\mathbf{x}_d, \mathbf{p}_i). \tag{12.6}$$

- Maximization of inter-cluster distance. One wants the different clusters to be clearly separated.

As anticipated, the various objectives are not always compatible, clustering is indeed a **multi-objective optimization** task. It is left to the final user to weigh the importance of achieving clusters of very similar entities *versus* achieving clusters which are well separated, also depending on the chosen number of clusters.

Divisive algorithms are among the simplest clustering algorithms. They begin with the whole set and proceed to divide it into successively smaller clusters. One simple method is to decide the number of clusters k at the start and subdivide the data set into *k subsets*. If the results are not satisfactory, the algorithm can be reapplied using a different k value.

If one wants to have groups of entities represented by a single vector, obtaining a more *compact way* to express them, a suitable approach is to select the prototypes which minimize the average **quantization error**, the error incurred when the entities are substituted with their prototypes:

$$\text{Quantization Error} = \sum_d \|\mathbf{x}_d - \mathbf{p}_{c(d)}\|^2, \tag{12.7}$$

where $c(d)$ is the cluster associated with data d.

In statistics and machine learning, **k-means clustering** partitions the observations into k clusters represented by **centroids** (prototypes for cluster c, denoted as $\mathbf{p}_c$), so that each observation belongs to the cluster with the *nearest* centroid. The iterative method to determine the prototypes in k-means, illustrated in Fig. 12.3, consists of the following steps.

1. Choose the number of clusters k.

2. Randomly generate k clusters and determine the cluster centroids $\mathbf{p}_c$, or directly generate k random points as cluster centroids (in other words, the initial centroid positions are chosen randomly from the original data points).

3. Repeat the following steps until some convergence criterion is met, usually when the last assignment hasn't changed, or a maximum number of iterations has been executed.

   (a) Assign each point $\mathbf{x}$ to the nearest cluster centroid, the one minimizing $\delta(\mathbf{x}, \mathbf{p}_c)$.

   (b) Recompute the new cluster centroid by averaging the points assigned in the previous step:

$$\mathbf{p}_c \leftarrow \frac{\sum_{\text{entities in cluster } c} \mathbf{x}}{\text{number of entities in cluster } c}.$$

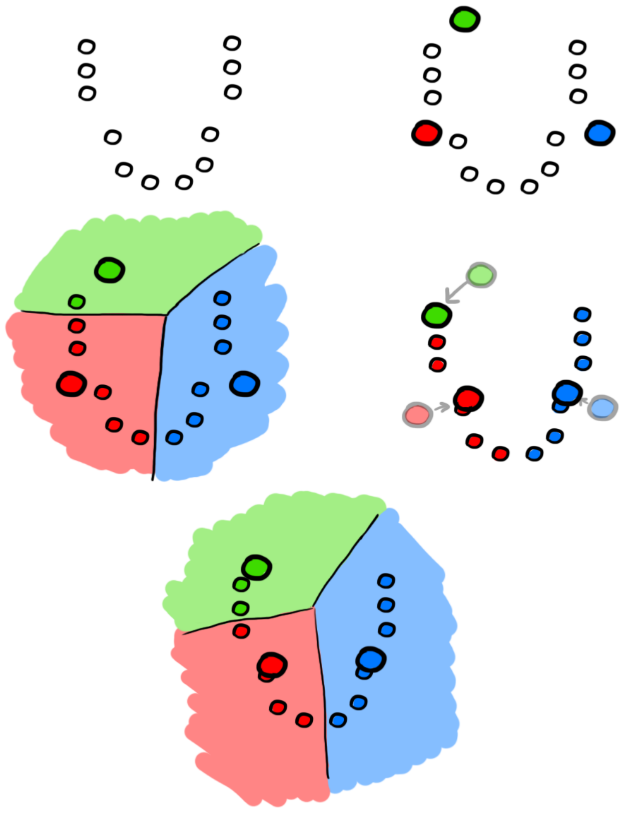


Figure 12.3: K-means algorithm in action (from top to bottom, left to right). Initial centroids are placed. Space is subdivided into portions close to the centroids (Voronoi diagram: each portion contains the points which have the given centroid as closest prototype). New centroids are calculated. A new space subdivision is obtained.

The main advantages of this algorithm are its simplicity and speed which allow us to use it on large datasets. **K-means clustering** can be seen as a skeletal version of the **Expectation Maximization** algorithm[1]: if the assignment of the examples to the cluster is known, then it is possible to compute the centroids and, on the other hand, once the centroids are known it is easy to calculate the clusters assignments. Because at the beginning both the cluster centroids (the parameters of the various clusters) and the memberships are unknown, one cycles through steps of assignment and recalculation of the centroid parameters, aiming at a consistent situation.

Given a set of prototypes, an interesting concept is the **Voronoi diagram**. Each prototype $\mathbf{p}_c$ is assigned to a Voronoi cell, consisting of all points closer to $\mathbf{p}_c$ than to any other prototype. The segments of the Voronoi diagram are all the points in the space that are equidistant to the two nearest sites. The Voronoi nodes are the points equidistant to three (or more) sites. An example is shown in Fig. 12.3.

Up to now we considered **hard-clustering**, where the assignment of entities to clusters is crisp. In some cases this is not appropriate and it is necessary to employ a softer approach where **assignments are not crisp, but probabilistic or fuzzy**. The assignment of each entity is defined in terms of a probability (or fuzzy value) associated with its membership in different clusters, so that values sum up to one. Consider for example a clustering of bald versus non-bald people. A middle-aged man with some hair left on his head may feel he is mistreated if associated with the cluster of bald people. By the way, in this case it is also inappropriate to talk about a probability of being bald, and a fuzzy membership is more suitable: one may decide that the person belongs to the cluster of bald people with a fuzzy value of 0.4 and to the cluster of hairy people with a fuzzy value of 0.6.

In **soft-clustering**, the cluster membership can be defined as a decreasing function of the dissimilarities, for example as:

$$\text{membership}(\mathbf{x}, c) = \frac{e^{-\delta(\mathbf{x},\mathbf{p}_c)}}{\sum_c e^{-\delta(\mathbf{x},\mathbf{p}_c)}}. \tag{12.8}$$

For updating the cluster centroids one can proceed either with a **batch** or with an **online update**. In the online update one repeatedly considers an entity $\mathbf{x}$, for example by randomly extracting it from the entire set, derives its current fuzzy cluster memberships and updates all prototypes so that the closer prototypes tend to become even closer to the given entity $\mathbf{x}$:

[1] In statistics, an expectation-maximization (EM) algorithm is a method for finding maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models, where the model depends on unobserved latent variables. EM is an iterative method which alternates between performing an expectation (E) step, which computes the expectation of the log-likelihood evaluated using the current estimate for the latent variables, and a maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the E step.

$$\Delta\mathbf{p}_c = \eta \cdot \text{membership}(\mathbf{x}, c) \cdot (\mathbf{x} - \mathbf{p}_c); \tag{12.9}$$
$$\mathbf{p}_c \leftarrow \mathbf{p}_c + \Delta\mathbf{p}_c. \tag{12.10}$$

With a physical analogy, in the above equations the prototype is pulled by each entity to move along the vector $(\mathbf{x} - \mathbf{p}_c)$, and therefore to become closer to $\mathbf{x}$, with a force proportional to the membership.

In the batch update, one first sums update contributions over all entities to obtain $\Delta_{total}\mathbf{p}_c$, and then proceed to update, as follows:

$$\mathbf{p}_c \leftarrow \mathbf{p}_c + \Delta_{total}\mathbf{p}_c. \tag{12.11}$$

When the parameter η is small, the two updates tend to produce very similar results, when η increases increasingly different results can be obtained. The online update avoids summing all contributions before moving the prototype, and it is therefore suggested when the number of data items becomes very large.

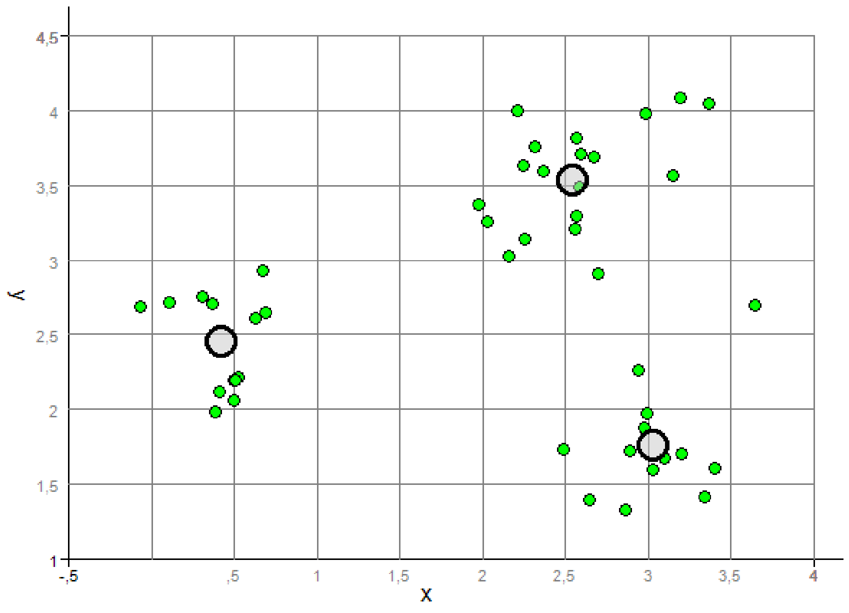


Figure 12.4: K-means clustering. Individual points and cluster prototypes are shown.

The k-means results can be visualized by a scatterplot display, as in Fig. 12.4. The k cluster prototypes are marked with large gray circles. The data points associated with a cluster are those for which the given prototype is the closest one among the k prototypes.

Gist

Unsupervised learning deals with building models using only input data, without resorting to classification labels. In particular, clustering aims at grouping similar cases in the same group, dissimilar cases in different groups. The information to start the clustering process can be given as relationships between couples of points (**external representation**) or as vectors describing individual points (**internal representation**). In the second case, an average vector can be taken as a **prototype** for the members of a cluster.

The objectives of clustering are: compressing the information by abstraction (considering the groups instead of the individual members), identifying the global structure of the experimental points (which are typically not distributed randomly in the input space but ”clustered” in selected regions), reducing the cognitive overload by using prototypes.

There is not a single ”best” clustering criterion. Interesting results depend on the way to measure **similarities** and on the relevance of the grouping for the subsequent steps. In particular, one trades off two objectives: a high similarity among members of the same cluster and a high dissimilarity among members of different clusters.

In **top-down clustering** one proceeds by selecting the desired number of classes and subdividing the cases. K-means starts by positioning K prototypes, assigning cases to their closets prototypes, recomputing the prototypes as averages of the cases assigned to them, ...

Clustering gives you a new perspective to look at your *dog Toby*. *Dog* is a cluster of living organisms with four paws, barking and wagging their tails when happy, *Toby* is a cluster of all experiences and emotions related to your favourite little pet.

Chapter 13

Bottom-up (agglomerative) clustering

Birds of a feather flock together.
(Proverb: Those of similar taste congregate in groups)

In general, clustering methods require setting many parameters, such as choosing the appropriate number of clusters in k-means, as explained in Chapter 12. A way to avoid choosing the number of clusters at the beginning consists of building progressively larger clusters, in a hierarchical manner, and leaving the choice of the most appropriate number and size of clusters to a subsequent analysis phase. This is called **bottom-up, agglomerative clustering**. Hierarchical algorithms find successive clusters by using previously established clusters, beginning with each element as a separate cluster and merging them into successively larger clusters. At each step the most similar clusters are merged.

Latest chapter revision: November 14, 2013

13.1 Merging criteria and dendrograms

Let $\mathcal{C}$ represent the current clustering as a collection of subsets of our set of entities, the individual clusters C. Thus $\mathcal{C}$ defines a partition: each entity belongs to one and only one cluster. Initially $\mathcal{C}$ is a collection of singleton groups, each with one entity.

Just as in top-down clustering, bottom-up merging also needs a measure of distance to guide the clustering process. In this case, the relevant measure is the distance between two clusters $C, D \in \mathcal{C}$, let's call it $\overline{\delta}(C, D)$, which is derived from the original distance between entities $\delta(x, y)$. There are at least three different ways to define it, leading to very different results. In fact, it is possible to consider the average distance between pairs, the maximum, or the minimum distance, as follows:

$$\begin{aligned}
\overline{\delta}_{ave}(C, D) &= \frac{\sum_{x\in C,\, y\in D} \delta(x, y)}{|C| \cdot |D|}; \\
\overline{\delta}_{min}(C, D) &= \min_{x\in C,\, y\in D} \delta(x, y); \\
\overline{\delta}_{max}(C, D) &= \max_{x\in C,\, y\in D} \delta(x, y).
\end{aligned}$$

The algorithm now proceeds with the following steps:

1. find C and D in the current $\mathcal{C}$ with minimum distance $\overline{\delta}^* = \min_{C\neq D} \overline{\delta}(C, D)$;
2. substitute C and D with their union $C \cup D$, and register $\overline{\delta}^*$ as the distance for which the specific merging occurred;

until a single cluster containing all entities is obtained.

The history of the hierarchical merging process and the distance values at which the various merging operations occurred can be used to plot a **dendrogram** (from Greek *dendron* "tree", *-gramma* "drawing") illustrating the process in a visual manner, as shown in Fig. 13.1-13.2.

The dendrogram is a tree where the original entities are at the bottom and each merging is represented with a horizontal line connecting the two fused clusters. The position of the horizontal line on the Y axis shows the value of the distance $\overline{\delta}^*$ at which the fusion occurred. To reconstruct the clustering process, imagine that you move a horizontal ruler over the dendrogram, from the bottom to the top of the plot.

By selecting a value of the desired distance level and cutting horizontally across the dendrogram, the number of clusters at that level and their members are immediately obtained, by following the subtrees down to the leaves. This provides a simple visual mechanism for analyzing the hierarchical structure and determining the appropriate number of clusters, depending on the application and also on the detailed dendrogram structure. For example, if a large gap in distance levels is present along the Y axis of

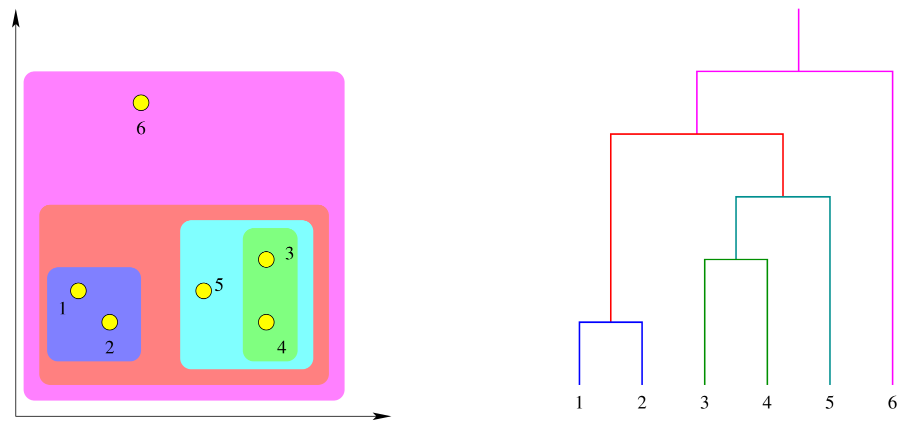


Figure 13.1: A dendrogram obtained by bottom-up clustering of points in two dimensions (with the standard Euclidean distance). Each point is an entity described by two numeric values.

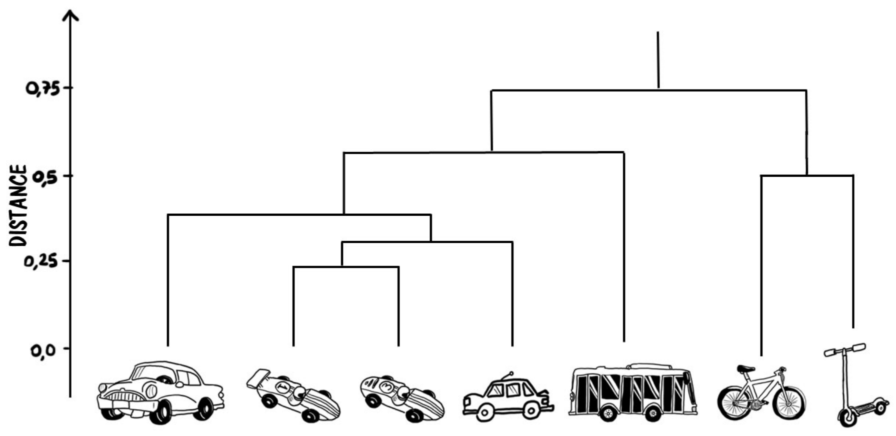


Figure 13.2: A dendrogram obtained by bottom-up clustering of vehicles.

the dendrogram, that can be a possible candidate level to cut horizontally and identify "natural" clusters.

Dendrograms are close cousins of the trees used in the natural sciences to visually represent related species, in which the root represents the oldest common ancestor species, and the branches indicate successively more recent divisions leading to different species. In addition to standard trees one has a quantitative measure of the *distance* at which a fusion between two subsets occurs, so that it is useful to plot the tree elongated along the Y axis. The individual items are at the bottom, the entire set (the root tree) at the top, and each merging is indicated by a horizontal segment placed at a Y position given by the distance at which fusion occurs, as illustrated in Fig. 13.2

13.2 A distance adapted to the distribution of points: Mahalanobis

The **Mahalanobis distance** was prompted by the problem of identifying the similarities of skulls based on measurements (in 1927) and is now widely used to **take the data distribution into account when measuring dissimilarities**. The data distribution is modeled by the correlation matrix.

After a set of points are grouped together in a cluster one would like to describe the whole cluster in quantitative (*holistic*) terms, instead of considering just the cloud of points grouped together. In the following we assume that the clouds of points forming the clusters have simple ball-shaped or "elliptic" forms, excluding for the moment more complex arrangements like for example spirals, zigzagging or similar convoluted forms.

In addition, given a new test point in N-dimensional Euclidean space, one would like to estimate the probability that the new point belongs to the cluster. A first step can be to find the average or center of mass of the sample points. Intuitively, the closer the point in question is to this center of mass, the more likely it is to belong to the set.

However, we also need to know if the set is spread out over a large range or a small range, so that we can decide whether a given distance from the center can be considered large or not. The simplistic approach is to estimate the standard deviation σ of the distances of the sample points from the center of mass. If the distance between the test point and the center of mass is less than one standard deviation, then we might conclude that the new test point belongs to the set with a high probability. This intuitive approach can be made quantitative by defining the **normalized distance** between the test point and the set to be $(x - \mu)/\sigma$. By plugging this into the normal distribution we can derive the probability of the test point belonging to the set.

The drawback of the above approach is that it assumes that the sample points are distributed in a spherical manner. If the distribution is highly non-spherical, for instance ellipsoidal, then one would expect the probability of the test point belonging to the set

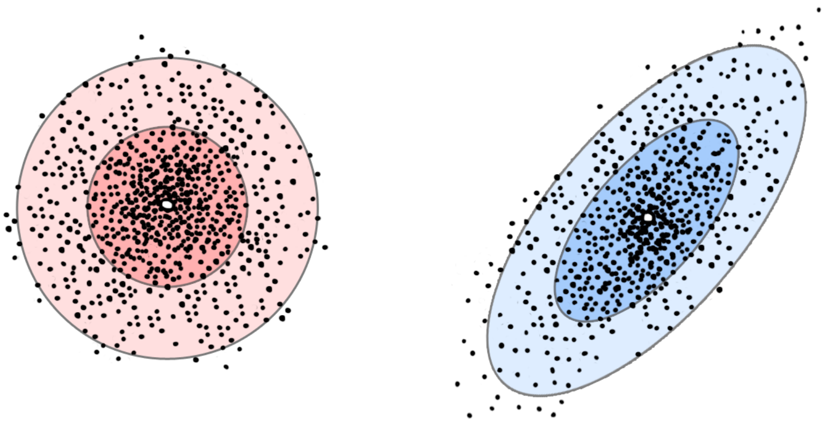


Figure 13.3: In the case on the left we can use the Euclidean distance as a dissimilarity measure, while in the other case we need to refer to the Mahalanobis distance, because the data are distributed in an ellipsoidal shape.

to depend not only on the distance from the center of mass, but also on the direction. In those directions where the ellipsoid has a short axis the test point must be closer, while in those where the axis is long the test point can be further away from the center.

The ellipsoid that best represents the set's probability distribution can be estimated by building the **covariance matrix** of the samples.

Let $C = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ be a cluster in D dimensions. The center $\bar{\boldsymbol{p}}$ of the cluster is the mean value:

$$\bar{\boldsymbol{p}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i. \tag{13.1}$$

Let the covariance matrix components are defined as:

$$S_{ij} = \frac{1}{n}\sum_{k=1}^{n}(p_{ki} - \bar{p}_i)(p_{kj} - \bar{p}_j), \qquad i, j = 1, .., D. \tag{13.2}$$

The Mahalanobis distance is simply the distance of the test point from the center of mass divided by the width of the ellipsoid in the direction of the test point. Fig. 13.3 illustrates the concept.

In detail, the **Mahalanobis distance** of a vector $\mathbf{x}$ from a set of values with mean μ and covariance matrix $\mathbf{S}$ is defined as:

$$D_M(x) = \sqrt{(x-\mu)^T S^{-1}(x-\mu)}. \tag{13.3}$$

The Mahalanobis distance can also be defined as a dissimilarity measure between two random vectors $\vec{x}$ and $\vec{y}$ of the same distribution with the covariance matrix $\mathbf{S}$:

$$d(\vec{x}, \vec{y}) = \sqrt{(\vec{x}-\vec{y})^T S^{-1}(\vec{x}-\vec{y})}. \tag{13.4}$$

If the covariance matrix is the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. If the covariance matrix is diagonal, then the resulting distance measure is called the **normalized Euclidean distance**:

$$d(\vec{x}, \vec{y}) = \sqrt{\sum_{i=1}^{N} \frac{(x_i - y_i)^2}{\sigma_i^2}}, \tag{13.5}$$

where σ_i is the standard deviation of the x_i over the sample set.

After clarifying the concepts of Mahalanobis distance and of the possible description of a cloud of points in a cluster though ellipsoids characterized by a fixed Mahalanobis distance from the barycenter, we are ready to understand the way in which clusters can be visualized.

13.3 Visualization of clustering

This section explains how clusters can be visualized in 3D (feel free to skip it without any effect on understanding the subsequent chapters). In order to graphically represent a cluster, one can visualize its **inertial ellipsoid**, whose surface is the locus of points having unit distance from the cluster's average position according to the Mahalanobis metric given by the cluster's dispersion. One starts by projecting the data points to three dimensions and calculating the corresponding 3x3 covariance matrix.

In graphical packages for three-dimensional rendering, points can be represented as row vectors of homogeneous coordinates in $\mathbb{R}^4$ with the infinite plane represented as $(x, y, z, 0)$, the projective coordinate transformation mapping the unit sphere into the desired ellipsoid is represented by the following matrix:

$$T_C = \begin{pmatrix} S_{11} & S_{12} & S_{13} & 0 \\ S_{21} & S_{22} & S_{23} & 0 \\ S_{31} & S_{32} & S_{33} & 0 \\ \bar{p}_1 & \bar{p}_2 & \bar{p}_3 & 1 \end{pmatrix}. \tag{13.6}$$

When moving between hierarchical clustering levels, cluster C will split into several clusters $C_1, \dots, C_l$. To preserve the proper mental image, a parametric transition from ellipsoid T_C to its l offspring $T_{C_1}, \dots, T_{C_l}$ will be animated and the ellipsoids

$$T_{C_i}^{\lambda} = (1 - \lambda)T_C + \lambda T_{C_i}, \qquad i = 1, \dots, l$$

will be drawn with parameter λ uniformly varying from 0 to 1 in a given time interval (currently 1 second). This will effectively show the original ellipsoid *morphing* into its offspring.

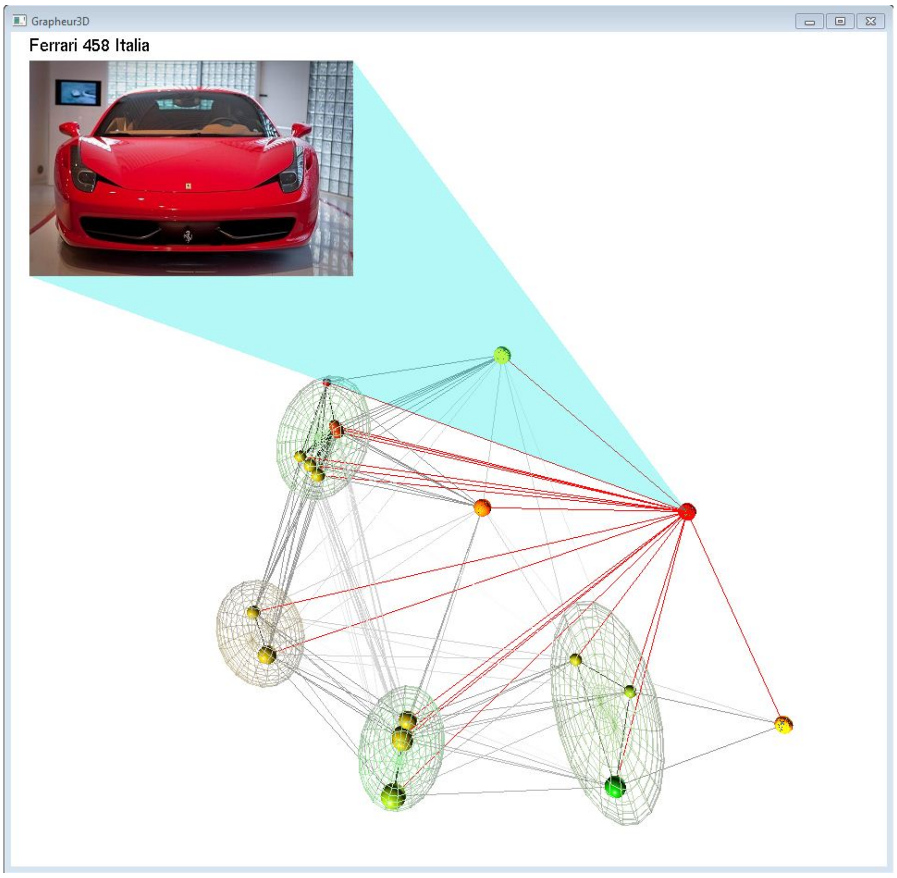


Figure 13.4: Clustering cars from mechanical characteristics.

One can navigate up and down the clustering hierarchy until identifying a suitable number of clusters for conducting the analysis. Then prototypes can be examined to provide a summarized version of the data. A particularly useful tool to use for navigating in this manner is the parallel coordinates display (see the Appendix). By moving the active selectors, the number of points displayed in navigation can be reduced to concentrate on the most interesting ones.

Fig. 13.4 shows some clusters resulting from the analysis of a set of cars, characterized by a vector containing mechanical characteristics and price. The edge intensity is related to the object distances: the closer the objects the darker the color.

Gist

Agglomerative clustering builds **a tree (a hierarchical organization)** containing data points. If trees are unfamiliar to you, think about the folders that you may use to organize your documents, either physically or in a computer (docs related to a project together, then folders related to the different projects merged into a "work in progress" folder, etc.).

Imagine that you have no secretary and no time to do it by hand: a bottom-up clustering method can do the work for you, provided that you set up an appropriate way to measure similarities between individual data points and between sets of already merged points.

This method is called "**bottom-up**" because it starts from individual data points, merges the most similar ones and then proceeds by merging the most similar sets, until a single set is obtained. The number of clusters is not specified at the beginning: a proper number can by obtained by cutting the tree (a.k.a. **dendrogram**) at an appropriate level of similarity, after experimenting with different cuts.

Through agglomerative clustering, Santa can now organize all Christmas presents as a single huge red box. After one opens it one finds a set of boxes, after opening them, still other boxes, until the "leaf" boxes contain the actual presents.

Bibliography

[1] R. Battiti. Accelerated back-propagation learning: Two optimization methods. *Complex Systems*, 3(4):331–342, 1989.

[2] R. Battiti. First-and second-order methods for learning: Between steepest descent and newton's method. *Neural Computation*, 4:141–166, 1992.

[3] R. Battiti. Using the mutual information for selecting features in supervised neural net learning. *IEEE Transactions on Neural Networks*, 5(4):537–550, 1994.

[4] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer Science Interfaces*. Springer Verlag, 2008.

[5] R. Battiti and A. M. Colla. Democracy in neural nets: Voting schemes for accuracy. *Neural Networks*, 7(4):691–707, 1994.

[6] Roberto Battiti and Anna Maria Colla. Democracy in neural nets: Voting schemes for classification. *Neural Networks*, 7(4):691–707, 1994.

[7] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[8] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[9] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[10] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and regression trees*. Chapman&Hall / CRC press, 1993.

[11] Kevin J Cherkauer. Human expert-level performance on a scientific image analysis task by a system using combined artificial neural networks. In *Working notes of the AAAI workshop on integrating multiple learned models*, pages 15–21. Citeseer, 1996.

[12] Antonio Criminisi. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends in Computer Graphics and Vision*, 7(2-3):81–227, 2011.

[13] Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pages 1–15. Springer, 2000.

[14] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *arXiv preprint cs/9501101*, 1995.

[15] Ralph B. DAgostino, Albert Belanger, and Jr Ralph B. DAgostino. *A Suggestion for Using Powerful and Informative Tests of Normality*, volume 44. 1990.

[16] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[17] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[18] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[19] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[20] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[21] Tin Kam Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844, 1998.

[22] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[23] T. Joachims. Making large-scale SVM learning practical. In Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11. MIT-Press, Cambridge, Mass., 1999.

[24] Daniel Kahneman. Thinking, fast and slow, farrar, straus and giroux, 2011.

[25] D.G. Krige. A statistical approach to some mine valuations and allied problems at the witwatersrand. Master's thesis, University of Witwatersrand, 1951.

[26] E. Osuna, R. Freund, and F. Girosi. Support vector machines: Training and applications. Technical Report AIM-1602, MIT Artificial Intelligence Laboratory and Center for Biological and Computational Learning, 1997.

[27] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[28] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[29] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In D.E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations*. MIT Press, 1986.

[30] Robert E Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[31] Joseph Sill, Gábor Takács, Lester Mackey, and David Lin. Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*, 2009.

[32] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. Technical Report NeuroCOLT NC-TR-98-030, Royal Holloway College, University of London, UK, 1998.

[33] James Surowiecki. *The wisdom of crowds*. Random House Digital, Inc., 2005.

[34] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[35] Kai Ming Ting and Ian H Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.

[36] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 9999:3371–3408, 2010.

[37] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. 1960.

[38] Bernard Widrow and Samuel D Stearns. Adaptive signal processing. *Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985, 491 p.*, 1, 1985.

[39] David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.