Roberto Battiti · Mauro Brunato

The LION Way

Machine Learning *plus* Intelligent Optimization

Version 3.0 — December 2017

intelligent-optimization.org/LIONbook

Version 3.0 — December 2017

ROBERTO BATTITI AND MAURO BRUNATO.
The LION way. Machine Learning plus Intelligent Optimization. Version 3.0
LIONlab, University of Trento, Italy, December 2017 (original: Feb 2014).

```
@book {
    author = "Roberto Battiti and Mauro Brunato",
    title = "The LION way. Machine Learning {\em plus} Intelligent Optimization",
    publisher = "LIONlab, University of Trento, Italy",
    year = "2017",
    month = "December",
    url = "http://intelligent-optimization.org/LIONbook/"
}
```

For inquiries about this book and related materials, refer to the book's web page:

http://intelligent-optimization.org/LIONbook/

To interact with the authors (we welcome comments and suggestions):

battiti@alumni.caltech.edu, mauro.brunato@unitn.it

Contents

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 tireless power** for continually improving processes, decisions, products and services. This is related to decision making but goes far beyond that. Decision making picks the best among a set of possible solutions which is given, optimization actively **creates new solutions**.

Optimization fuels **automated creativity and innovation**. It looks like a manifest contradiction, because creativity

is usually *not* related to automation. This is why the message you will find in this book is disruptive, far from trivial, even irritating and provoking for people believing that machines are only for shallow mechanical and repetitive tasks.

Starting from Galileo Galilei (1564-1642), to change the world with science, not only to interpret it with philosophy, one needs **measurements and experiments**. "Measure what is measurable, and make measurable what is not so." Measurements start shy and humble but permit a gradual and pragmatic conquering of the world as far as production means and quality of life are concerned.

Almost all business problems can be formulated as **finding an optimal decision $\boldsymbol{x}$ by maximizing a measure** *goodness*$(\boldsymbol{x})$. For a concrete mental image, think of $\boldsymbol{x}$ 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, newspaper, web, social), and *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, and *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 traveled.

Formulating a problem as "optimize a goodness function" also **encourages 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, to 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 create and search for one or more optimal choices. And when conditions or priorities change, just revise the goals quantified by the goodness measure, restart the optimization process, *et voilà*. To be sure, 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 awesome 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*$(\boldsymbol{x})$ associating a result to each input configuration $\boldsymbol{x}$. 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 conversation for 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 goodness model can be built by machine learning from abundant data**.

Learning and Intelligent Optimization (LION) is the combination of learning from data and optimization to solve complex and dynamic problems. The LION way is about increasing the automation level and connecting data directly to decisions and actions. **Prescriptive analytics** is the third and final phase, beyond descriptive and predictive analytics. With support of the right software, 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 a car, but the user (driver) does not need to know the inner workings of the engine in order to realize its 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 measurements of 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 [242]. Around 1951 he began his pioneering work on applying insights in statistics to the valuation of new gold mines by 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 $\boldsymbol{x}$ is 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 $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3 \ldots$ to 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 $GoldModel(\boldsymbol{x})$, which could generalize the experimental results by predicting output values for each position $\boldsymbol{x}$. The model could be used by an optimizer to identify the next point to dig, by finding the position $\boldsymbol{x}_{best}$ maximizing the estimated gold output of $GoldModel(\boldsymbol{x})$.

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 identifying the highest point on this model surface, and the corresponding position where to dig the next mine.

This technique is now called *Kriging* and it is based on the idea that the output 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 modeling 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 $IdealMate(\boldsymbol{x})$ 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 homework you are invited to define in precise mathematical terms the *IdealMate* function for your ideal partner. Even assuming that you can identify some building

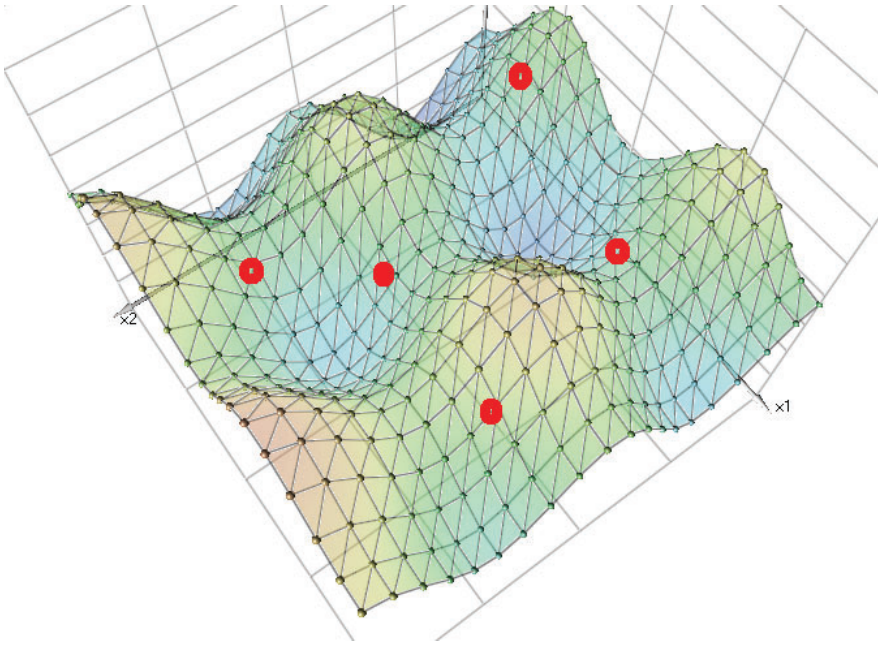


Figure 1.2: *Kriging* method constructing a model from samples. Some samples are shown as dots. Height and color of the surface depend on gold content.

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 *IdealMate*($\boldsymbol{x}$) that the system will optimize. Only after considering different candidates and giving feedback to the matching service you may hope to identify your best significant other.

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 1.7.

Enterprises are flooded with data in digital form. Big data is a popular term to refer to abundant and partially structured data. By the way, data used to be much bigger *with respect to available storage* in the seventies and eighties so that the term "big data" now is more related to marketing hype than to reality: a single PC will easily deal with all data produced by all apart from the biggest companies.

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 billions in spending on information tech-

nology. 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 deal with software capable of self-improvement and rapid adaptation to new data and revised 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 three fundamental needs from their data:

1. to **understand** the current business processes and to review past performance;
2. to **predict** the effect of business decisions;
3. to **improve** profitability by identifying and implementing informed and rational decisions about critical business factors.

Traditional **descriptive** business intelligence excels at recording and 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 rear view 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 rigidity and lack of quick adaptation.

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 creation of improving solutions. **Prescriptive analytics** leads from the data directly to the best improving plan, **from data, to actionable insight, to actions**!

1.5 Implementing LION

The steps for fully adopting LION as a business practice vary depending on the current business state, and most importantly, on 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 defining the structure of the underlying data, the most important factor is to establish a 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. 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 job market will also be disrupted. Software learning from human examples will infer the rules we tacitly apply but do not explicitly understand. This will eliminate barriers to further automation and substitute machines for workers in many tasks requiring adaptability, common sense and creativity, possibly putting the middle class at risk[367].

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 will help enterprises to lead the competition and avoid being burned by wild prairie fires, and will help individuals to remain competitive in the high-skill job market.

1.6 Teaching and learning in Internet times

Figure 1.3: A professor teaching "ex cathedra" in a University.

Books were made by hand in the middle ages, carefully written by *amanuensis* on parchment made from animal skins. In addition to hundreds of killed animals, a total of some man years were required for a single illustrated copy. The University institution, *universitas magistrorum et scholarium* – "community of teachers and scholars", the first one founded in 1088 in Bologna, was created to make knowledge cheaper and accessible to more people, and, by the way, more free . A professor used to stay in front of an audience to read a book "*ex cathedra*" to the students for subsequent discussion and deepening. For most of them, this was the only way to get knowledge.

We think that books still have a role is this era of information-at-your-fingertips, but a different one with respect to the ages before internet and the web. Nobody needs books to get detailed knowledge, demonstration of theorems,

pieces of software, original research papers. But human people need a **global vision and connected ideas** to solve problems in a professional manner. A book is like a mental map to orient our steps in a foreign territory. In a tourism analogy one needs to know the whole "meaning" of Trentino-South Tirol, its peaks (cathedrals in the Alps), its abundance of public mountain huts, the pleasure of dining together with people hiking and climbing from all over the world, in order to be motivated to spend a vacation here. Then the dots can be easily connected with web reservations, train schedules, getting the proper boots, etc.

Our goal is to give a clear map of machine learning and intelligent optimization, **some concrete and connected ideas that will stick to your mind** and that will guide you in the search for details and software when they are needed. On the contrary, if too much effort is spent on details, they will be easily forgotten some days after your university exam is passed. As an example, one can easily find software for realizing a specific ML model from examples, but the lack of a global vision about how to proceed can lead to catastrophic mistakes like confusing training error with generalization error, or putting the random ID of a customer in the inputs to derive some marketing measure, or picking a sub-symbolic neural-network model when a human explanation of a diagnosis is required.

Needless to say, the definition of what is basic and what is detail is personal and subject to discussion. Our choice is based on tens of courses on the subject but we take the full responsibility for the lack of completeness in this huge territory covering machine learning and optimization.

If one does not know to which port one is sailing, no wind is favorable (*Ignoranti quem portum petat, nullus suus ventus est* - Seneca).

1.7 A "hands on" community 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 by us and by our community of readers will be posted on the book's website:

`http://intelligent-optimization.org/LIONbook/.`

A sign of gratitude goes to the many contributors to our effort. Let's start from photos and drawings. Venice photo by Carlo Nicolini for LION4@VENICE 2010. Dante's painting in the Introduction by Domenico di Michelino, Florence, 1465. Mushroom basket image by George Chernilevsky. Brain drawing in the Neural Networks chapter by Leonardo da Vinci (1452 – 1519). Deep network for clustering figure by Geoffrey Hinton. Photo of prof. Vapnik in

the SVM chapter from Yann LeCun. Extreme Learning Machine image by Guangbin Huang. Reservoir architecture image by Herbert Jaeger. Venice painting in the Democracy chapter by Canaletto, 1730. Painting in the Clustering chapter by Michelangelo Buonarroti (Sistine Chapel), 1541. Wikipedia has been mined for some explanatory images, while the authors and their sons contributed to the wikipedia project with additional images and definitions. Hopfield network figure by Gorayni, energy landscape by Mrazvan22. Lagrange multipliers figure from Nexcis (wikipedia). Reschensee photo in reservoir chapter by Markus Bernet. Frog image by André Karwath. Hopfield network images by Alejandro Cartas Ayala. Viterbis' algoritm figure by Luz Abril Torres-Méndez. Fig. 24.10 from Randall Munroe *xkcd* webcomic. Dart photo in Chapter 25 by Harris Morgan. Initial figure of Satisfiability chapter from Squidsoup art group installation.

Last but not least, we are happy to acknowledge the growing contribution of our readers to the quality of this book including Patrizia Nardon, Fabian Pedregosa, Fred Glover, Alberto Todeschini, Yaser Abu-Mostafa, Marco Dallariva, Enrico Sartori, Danilo Tomasoni, Nick Maravich, Marco Zugliani, Dinara Mukhlisullina, Rohit Jain, Jon Lehto, George Hart, Markus Dreyer, Yuyi Wang, Gianluca Bortoli, Davide Pedranz, Stefano Fioravanzo. Cartoons are courtesy of Marco Dianti. We are always pleased to communicate with our readers. Please email us with comments, suggestions or *errata* and we will be glad to add your name in the next edition. You find a contact form and email addresses in our LIONlab website:

`http://intelligent-optimization.org/`.

Addendum to Version 3.0

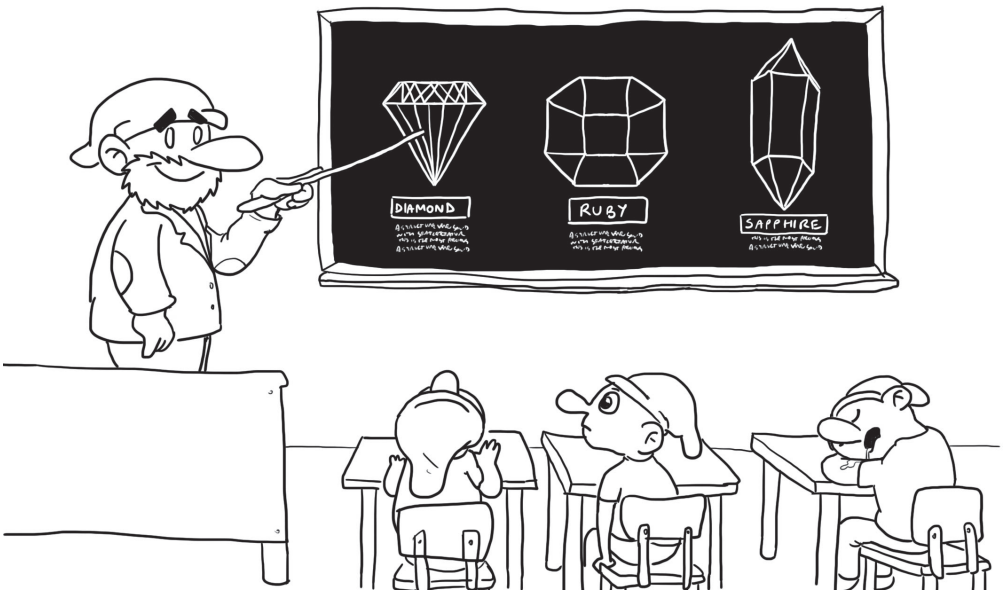
You are now reading version 3.0 of this book: the main changes with respect to the previous version are:

- A new section on **hashing** (Sec. 2.2) and locality-sensitive Hashing (LSH) and approximated nearest neighbors (Sec. 2.3). Hashing is one of the most useful tricks in computer science, but not widely known to ML users.
- The addition of more details related to entropy and mutual information (Sec. 7.6), including generalization to continous variables.
- The addition of a new section about **projection pursuit** (Sec. 20.6) in Chapter 20 dedicated to linear projections.
- The addition of a new Chapter about Feature extraction and Independent Component Analysis (ICA) (Chapter 21)

Chapter 2

Lazy learning: nearest neighbors

Natura non facit saltus
Nature does not make jumps



If you still remember how you learned 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 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.

Figure 2.1: Mushroom hunting requires classifying edible and poisonous species.

In **supervised learning** a system is trained by a supervisor (teacher) giving **labeled examples**. Each example is an array, a *vector* of input parameters $\boldsymbol{x}$ 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 awesome Trentino for vacation, not for work. When 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 positive* classifications of mushrooms (classifying it as non-poisonous when in fact it is) can be very harmful to your health.

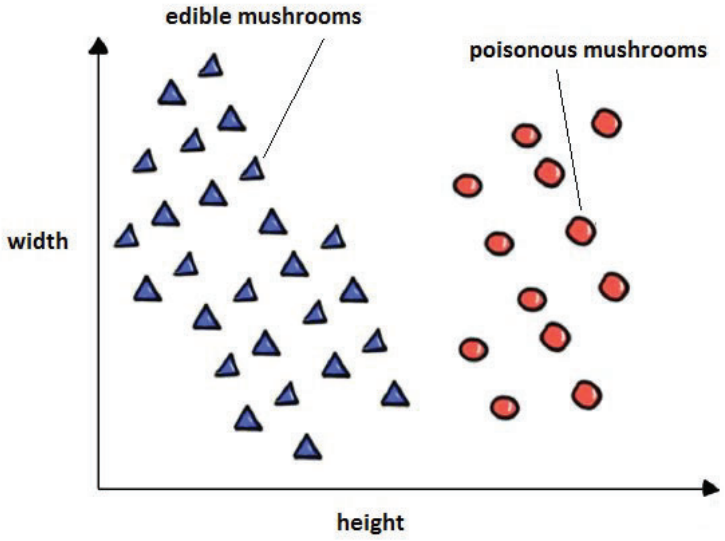


Figure 2.2: A toy example: two features (width and height) to classify edible and poisonous mushrooms.

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 labels) are stored and no action is taken until a new input pattern demands an output value. The system is a **lazy learner**: it does nothing but store labelled 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 simple, surprisingly this technique is effective in many cases. 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 only 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

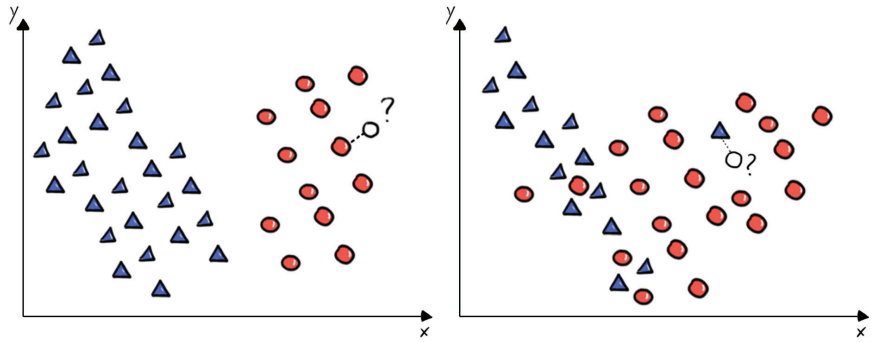


Figure 2.3: Nearest neighbor classification: a clear situation (left), a more confusing situation (right). In the second case the nearest neighbor to the query point with the question mark belongs to the wrong class although most other close neighbors belong to the right class.

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 ℓ be the number of labeled examples and $k \leq \ell$ be a fixed positive integer, 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 $\boldsymbol{x}$ and the stored 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 often achieving low estimation errors. Unfortunately it requires massive amount of memory, and lots of computation in the prediction phase. To reduce memory consumption one can *cluster* the examples, by grouping similar cases together. Only the prototypes (*centroids*) of the identified clusters are then stored. 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** generalize the nearest-neighbors idea in a flexible and smooth manner; the distant points are not brutally excluded, all points contribute to the output but with a significance ("weight") related to their distance from the query point.

2.2 From brute-force to smarter lookups: Hashing

Up to now, the topic of nearest neighbors looked simple. But we did not consider CPU time, in particular when the number of stored items ℓ and/or the input dimensionality d become large. A **brute-force** method to find a nearest neighbor calculates all possible distances to identify the smallest one, in time $O(\ell\ d)$, i.e., growing linearly in $\ell\ d$.

If you are not familiar with the asymptotic notation, the letter O (also referred to as *order* of the function) denotes the "growth rate" and determines how a function value grows for very large inputs and apart from a constant multiplicative term. In detail, $f(x) \in O(g(x))$ if there exists $c > 0$ and x_0 such that $f(x) \leq cg(x)$ whenever $x \geq x_0$.

Worst-case situations, corresponding to malicious counter-examples, are loved by theoreticians but can be very rare in most practical situations. Worst-cases situations can be like small flies on an elephant of highly probable well-behaved cases. Can we develop faster methods, at least **on average** if not in the worst case?

The answer is yes, provided that we invest in supporting data structures, or if we tolerate **approximated results**. The topic is rapidly getting complex, so that you may want to skip this Section at a first reading. On the other hand, finding closest neighbors is so central in machine learning that we cannot resist mentioning some basic tricks and results.

Examples of applications in **very large databases** include finding duplicate pages on the Web (for more efficient search engines), image retrieval (from multi-dimensional image descriptors), music retrieval, computational biology and drug design, computational linguistics, etc.

Let's start from a radical simplification: imagine that the potential query points are picked only from the set of ℓ stored examples. Why should one search for items which are already known? Well, one may want to retrieve information *associated* with the items, like output labels, or values. For example, imagine to set up a telephone directory which associates names with telephone numbers. A generalization is the so-called **dictionary data structure** in computer science, storing *key-value* pairs (k, v) so that, when a key k is given, the corresponding value v is returned. "Brute-force" means comparing all possible stored keys with the query k and returning the corresponding v as soon as the correct key is found.

To understand how hard this search can be, imagine a telephone directory with randomly placed names, where the only effective strategy to search a name is to scan the whole directory from the beginning until a match is found.

Now, if names are alphabetically ordered, one can open the directory at an approximated good starting point, to then proceed gradually to search before or after the opening page depending on what one finds there. This is much faster than searching through all pages, on average (the worst case can be in a town with most citizens sharing the same family name).

A better supporting data structure (like an alphabetically ordered index) means more work at the beginning, but a faster lookup. **Binary search** can be used for the lookup: the content at the middle position in the index is retrieved: if the key comes before (in alphabetical order), the search is recursively executed on the first half of the list, if the key comes later, the search is recursively executed on the second half of the list.

For a uniform distribution of keys, binary search in an ℓ-entry directory passes from $O(\ell)$ to $O(\log \ell)$ lookup time[1], on average. This is already a huge improvement for large numbers of stored keys. An index in a database (like in most SQL databases) is built precisely with this purpose. Can we do better? Can we search for a key, out of a set of ℓ, in a time which is approximately constant, on average?

Yes, we can: by using **hashing**, one can retrieve stored items in almost constant time, a very surprising result at first. To "hash" means to chop into small pieces, mince, mix up, from Latin *ascia* (ax). The idea is surprisingly simple: take your key k and "chop it" in a brutal (but deterministic!) manner to obtain an integer number. Consider the integer as an *index*, or an *address*, and store the corresponding value v at that address in memory. If one uses a function hash(k) which is fast to compute, lookup is fast: **hash the key to get the address, retrieve the stored key** at the calculated address.

Are we done? Almost. If you got your coffee this morning you may notice that *hashing* could produce, by chance, the *same* integer address for two different keys (Fig. 2.4). These *collisions* need to be dealt with, otherwise some values get lost. The solution is to have, at the obtained address, not only the space for a single value, but for a **bucket**

[1] If you know enough about logarithms, you may ask: "what base?" But remember, the asymptotic notation discards multiplicative constants

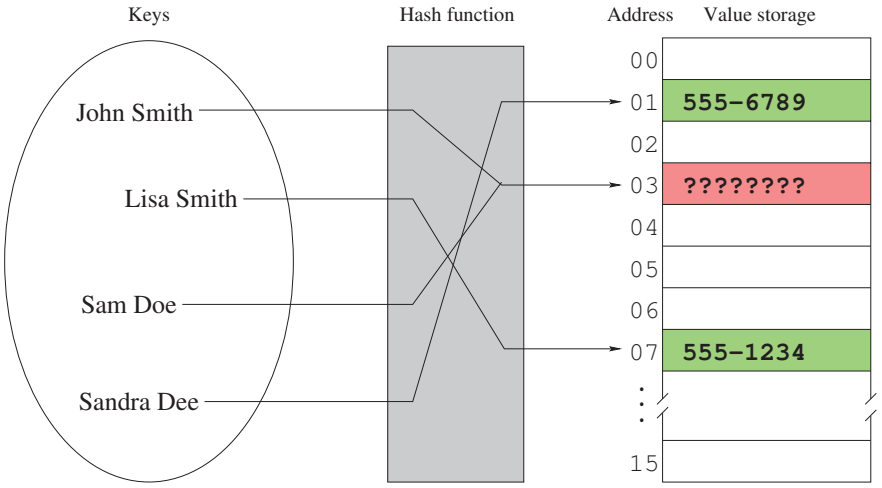


Figure 2.4: A collision when hashing names.

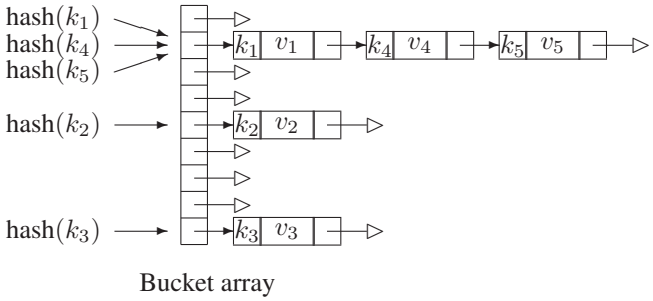


Figure 2.5: Open hashing scheme: items (configuration, or compressed hashed value, etc.) are stored in "buckets." The index of the bucket array is calculated from the configuration.

containing all *key-value* pairs for the keys hashed there, so that one can still identify the correct value in spite of the collision.

In the telephone directory example, hash(name) could give the number of a directory page (our physical *bucket*), which is big enough to contain all name-telno pairs to be stored there.

Hashing should be designed to make collisions rare and to spread the stored keys approximately in a uniform manner onto a given range of integers. This is why it is called hashing after all, the "ax" will scatter keys in a pseudo-random way onto a range of integers. To avoid wasting memory, the range of integers should be a small multiple of the total number ℓ of stored items.

The storage of labeled examples through a **hash table** requires CPU time for lookup that is approximately constant with respect to the number of items to store.

An example of a memory configuration for the hashing scheme is shown in Fig. 27.5. From the key k one obtains an index into a "bucket array." The items (k_i with associated information v_i etc.) are then stored in linked lists starting from the indexed array entry. Each cell contains the address of the next one (or *null* if it is the last cell), this is called **chaining**. Both storage and retrieval require an approximately constant amount of time if: i) the number of stored

items is not much larger than the size of the bucket array, and ii) the hashing function scatters the items with a uniform probability over the different array indices. More precisely, given a hash table with m slots that stores ℓ elements, a load factor $\alpha = \ell/m$ is defined. If collisions are resolved by chaining, searches take $O(1 + \alpha)$ time, on average.

2.3 Locality-sensitive Hashing (LSH) and approximated nearest neighbors

When hashing is used to map from a set of keys to the corresponding values, one is dealing with memorization and **databases**, not with machine learning and generalizing.

Returning to the nearest-neighbors for ML context just considered in Sec. 2.1, imagine now that the query $\boldsymbol{q}$ is a point placed in the d-dimensional space $\mathbb{R}^d$, and not necessarily equal to one of the stored points. Standard hash tables fail: remember that a hash function is intentionally meant to randomize things, and the new point $\boldsymbol{q}$ will be hashed to an index without any clear relationship with the index of nearby points in $\mathbb{R}^d$.

Locality-Sensitive Hashing (LSH) considers specialized versions of hashing, designed to **preferentially map close items to the same bucket**, with a probability which is higher for nearby points than for distant ones [336].

LSH is the first example of a **randomized algorithm**. A randomized algorithm contains calls to a random number generator, and its output is studied with probability and statistics. LSH does not guarantee an exact answer to a nearest-neighbor query but it provides **a high-probability guarantee that it returns the correct answer** or one close to it. By investing more computational effort, the probability can be pushed as high as desired. Because a small probability of failure in identifying the exact nearest neighbors is acceptable for most ML applications, LSH is becoming a widely used tool for very-large-scale applications.

Let's assume that one wants to guarantee with a probability equal to $1-\delta$ that the nearest neighbor will be retrieved for any query point, in a large database.

A possibility is to consider **randomized linear projections** of points in $\mathbb{R}^d$ as hashing functions. If you are not familiar with projections, consider points in 3 dimensions, projected onto a plane by a torch. Rendering a multidimensional sphere onto a two-dimensional page is a good example (Fig.2.6). Or read Chapter 20 for more details.

The motivation for LSH is based on the idea that, if two points are close together, they will remain close together after a "projection" operation. On the contrary, if two points are far apart, for most random projections they will remain distant: they will be mapped to close positions only for rare choices of the orientation of the projection. LSH therefore differs from conventional and cryptographic hash functions because it aims at **maximizing the probability of a "collision" for similar items**.

One creates **projections from a number of different random directions**, keeps track of the nearby projected points, those falling in the same buckets and notes the points that appear close to each other in more than one projection. Finally, one scans the remaining candidate list to identify the closest point. To limit the probability of failure (of returning a wrong nearest neighbor), one can adjust the number of randomized projections.

The starting point and intuition is simple, the technical details and analysis are complex and can be found in [109]. We just sketch a random projection algorithm to understand the main building blocks.

If $\boldsymbol{q}$ is a query point, a randomized scalar projection is obtained by a scalar product:

$$\text{hash}(\boldsymbol{q}) = \boldsymbol{p} \cdot \boldsymbol{q}$$

in which $\boldsymbol{p}$ is a vector with components that are selected at random from a Gaussian distribution, for example $\mathcal{N}(0, 1)$.

This scalar projection is then *quantized* into a set of hash bins, with the intention that nearby items in the original space will tend fall into the same bin. The resulting full hash function is given by:

$$\text{hash}_{\boldsymbol{p},b}(\boldsymbol{q}) = \left\lfloor \frac{\boldsymbol{p} \cdot \boldsymbol{q} + b}{w} \right\rfloor,$$

where w is the width of each quantization bin, b is a random value uniformly distributed between 0 and w, and the "floor" operator $\lfloor \cdot \rfloor$ maps a positive real value to an integer by eliminating the digits after the dot (e.g., $\lfloor 2.34 \rfloor = 2$). Note that quantizing a set of real values still means that nearby points *can* end up in different bins if by chance they are

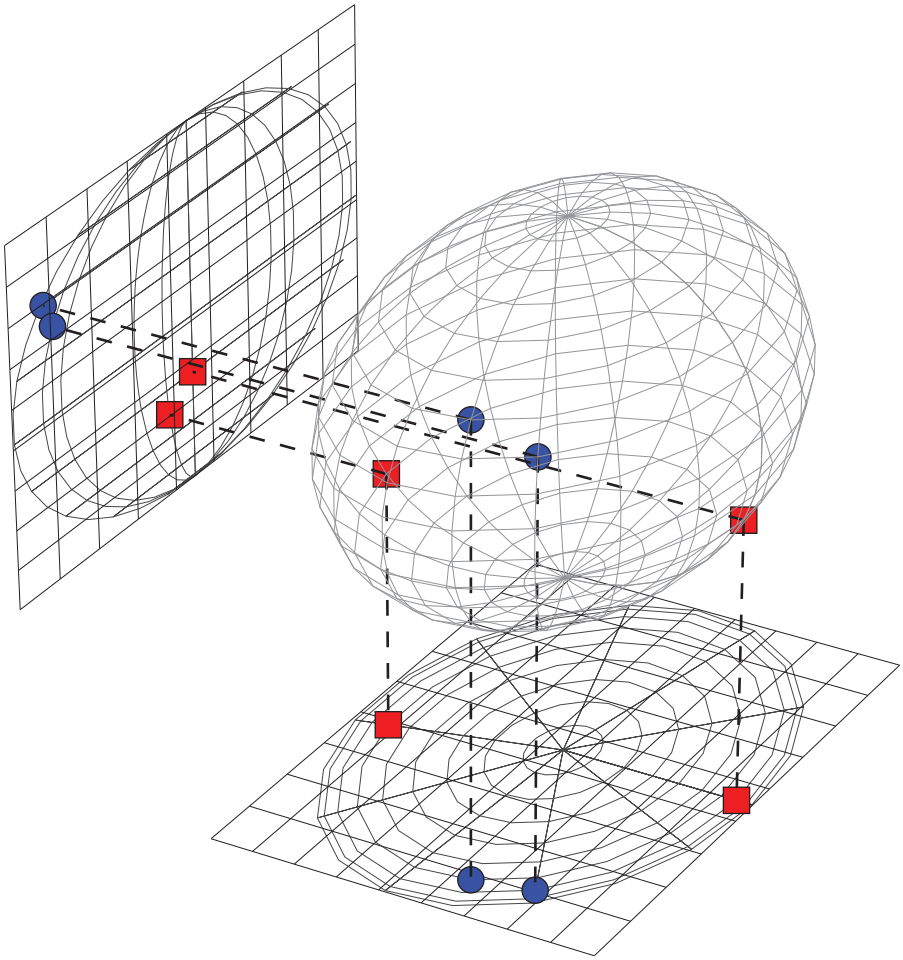


Figure 2.6: Locality-sensitive hashing by projections: points that are close to each other in the high-dimensional space are guaranteed to be close in the projection; the opposite is not always true.

on different sides of a boundary between bins (e.g., one is mapped to $2.99999w$, the other one to $3.00001w$). Adding a random value b ensures that if two such points are unlucky and sit on the opposite side of a boundary for one hash function, they can be more lucky and end up in the *same* bin for a different choice of b.

The two starting facts are that:

- close points will have a large probability of falling into the same bucket:

$$\Pr(\text{hash}(\boldsymbol{p}) = \text{hash}(\boldsymbol{q})) \geq P_1 \quad \text{for} \quad \|\boldsymbol{p} - \boldsymbol{q}\| \leq R_1;$$

- points $\boldsymbol{p}$ and $\boldsymbol{q}$ that are far apart have a low probability $P_2 < P_1$ to fall into the same bucket:

$$\Pr(\text{hash}(\boldsymbol{p}) = \text{hash}(\boldsymbol{q})) \leq P_2 \quad \text{for} \quad \|\boldsymbol{p} - \boldsymbol{q}\| \geq R_2,$$

  where $R_2 > R_1$.

Due to the linearity of the dot product, the difference between two image points $\|\text{hash}(\boldsymbol{p}) - \text{hash}(\boldsymbol{q})\|$ has a magnitude whose distribution is proportional to $\|\boldsymbol{p} - \boldsymbol{q}\|$ and therefore, $P_1 > P_2$.

Since the gap between the probabilities P_1 and P_2 could be quite small, an **amplification** process is needed in order to achieve the desired probabilities of collision, by performing k independent dot products in parallel. A far point could end up in the same bucket for an unlucky projection, but being unlucky k times has a much smaller probability.

Within each set of k dot products, one achieves success if the query and the nearest neighbor are in the same bin in *all* k dot products. This occurs with probability P_1^k, which decreases as we include more dot products. To reduce the impact of an "unlucky" quantization in any one projection, we form L independent projections and pool the neighbors from all of these. A true near neighbor will be unlikely to be unlucky in all the L projections.

A large k (AND-ing more projections) will filter out bad (distant) points, while a large L (OR-ing the contents of all L retrieved buckets) will ensure that we do not throw away the baby (the closest neighbors) with the dirty water.

Let's assume that one wants to retrieve R-neighbors of a query point $\boldsymbol{q}$ (neighbors at distance less than R). The query algorithm is:

- For each $j = 1, 2, \ldots, L$:

  1. Retrieve the points from the bucket obtained by concatenating the k integers in the j-th hash table.
  2. For each retrieved point, compute the distance from $\boldsymbol{q}$ to it, and report the point if it is a correct answer (an R-near neighbor).
  3. (optional) Stop as soon as the number of reported points is more than L'.

As an additional optimization, because many bins can be empty in a k-dimensional space, one can use standard hashing to efficiently store and retrieve only the non-empty bins.

After running the math so that the desired probability of success is guaranteed by a proper choice of k and L, the time needed to calculate and hash the projections is $O(dkL)$, a huge improvement with respect to the linear scan of all points, provided that the time to deal with collisions is limited. A practical approach to choosing k is introduced in the E2LSH package [10], by starting from estimates of CPU time obtained over a small set of sample queries.

The research in this area is still active, some of the best results have been encouraged by the need to analyze huge amounts of data, and obtained in the last five-ten years [245, 109].

2.4 Space-partitioning data structure: k-d trees

LSH is not the only possibility to speed up search in **computational geometry**. Other **space-partitioning data structures** have been proposed for organizing points in a d-dimensional space. Trees of boxes in $\mathbb{R}^n$ are a standard way to go from a linear CPU time to one logarithmic in the number of stored points. ***k*-d trees** are a special case of binary space partitioning trees. The root of the tree corresponds to the entire $\mathbb{R}^d$ space, at each node a left and right

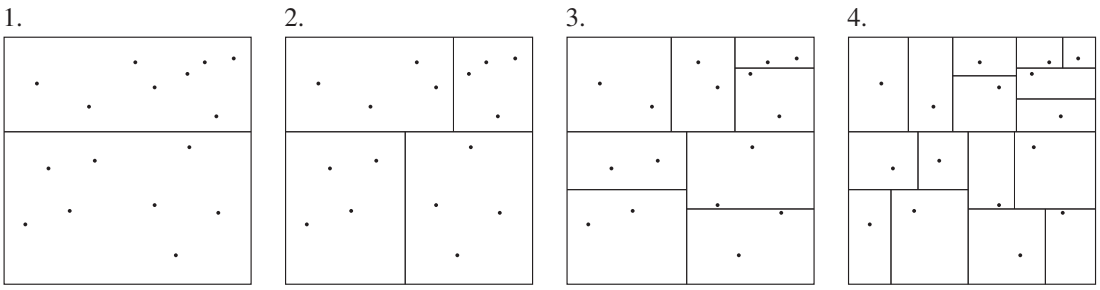


Figure 2.7: A k-d tree data structure.

subtrees are defined by a test on a single coordinate. If the coordinate c and threshold x_c are chosen for a particular split, all points whose c-th coordinate is less than x_c will appear in the left subtree, the other points in the right subtree. Imagine a sequence of hyperplanes perpendicular to coordinate axes leading to boxes containing two boxes, containing two boxes... untile a single point remains. If each split divides the points approximately into two equal parts, a **logarithmic time for searching** is obtained. The initial N points are divided by two, then again, ..., until $N/2^s \approx 1$ which means $s \approx \log_2 N$.

A logarithmic time for searching exact matches is not particularly relevant since $O(1)$ time can be obtained by hashing. But k-d trees become interesting for nearest-neighbor searches.

Searching for a nearest neighbour in a k-d tree proceeds as follows: Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e., it goes left or right depending on whether the point is lesser than or greater than the current node in the split dimension). Once the algorithm reaches a leaf node, it saves that node point as the "current best." After having this bound on the minimum distance the algorithm unwinds the recursion of the tree to check if more promising candidate subtrees exist. A decision whether subtrees need to be examined is made by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Details vary for different implementations and assumptions, building a static k-d tree from N points can be done in $O(N \log N)$ time, finding one nearest neighbour in a balanced k-d tree with randomly distributed points takes $O(\log N)$ time on average, [51, 137]

Appropriate data structures can bring huge improvements with respect to brute-force techniques, but have to be chosen and used with great care. In particular k-d trees maintain logarithmic search times only in small dimensions. The **curse of dimensionality** strikes: as soon as the number of dimension d grows so that the number of points is not sufficiently large (approximately $N \gg 2^k$), most of the points in the tree will be evaluated during the search and the efficiency becomes worse than exhaustive search (given the added overhead in building the structure). Approximate nearest-neighbour methods like LSH should be used instead.

Never use a data structure without considering if the assumptions for its range of validity hold in your case, or you may be greatly disappointed.

Gist

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

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

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.

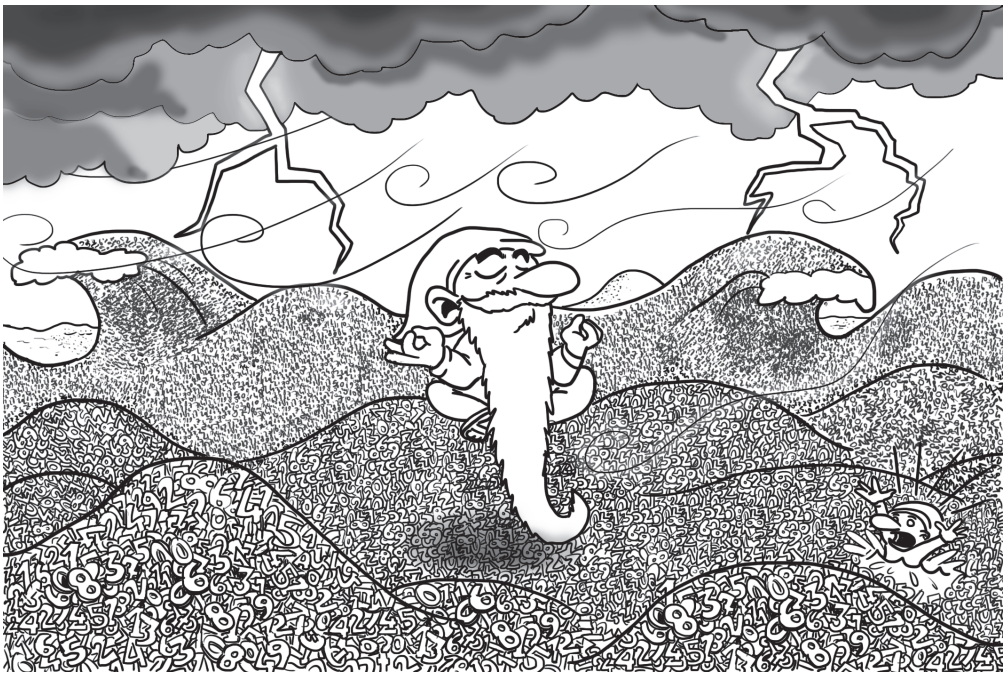
Investing in smart data-structures is critical to obtain fast exact or approximated nearest-neighbors searches. **Hashing** is a key trick, and now you can see a deeper meaning of the "hashtag" term popular in social networks, related to searching for keys and retrieving . . . tweets. **LSH (locality-sensitive hashing)** is not to be confused with psychedelic LSD, its advantages are very real in a context of big data.

Now, do not sleep motivated by this lazy-learning topic but go ahead and digest additional chapters of this book.

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 a wide range of events through compact models, with **unifying different cases by discovering the underlying explanatory laws**.

Above all, learning from examples is only a means to reach the real end: **generalization, the capability of explaining new cases**, of predicting new outputs, in the same application area but not already encountered during the learning phase. On the contrary, learning by heart or memorizing are considered very poor forms of learning, useful for beginners but not for real experts. If the goal is generalization, **estimating the performance has to be done with extreme care**. Observing the behavior of the model on the learning examples does not guarantee a proper generalization and may lead to unjustified optimism. After all, a student who is good at learning by heart and repeating the exact words of the teacher will not always end up being the most successful individual in life!

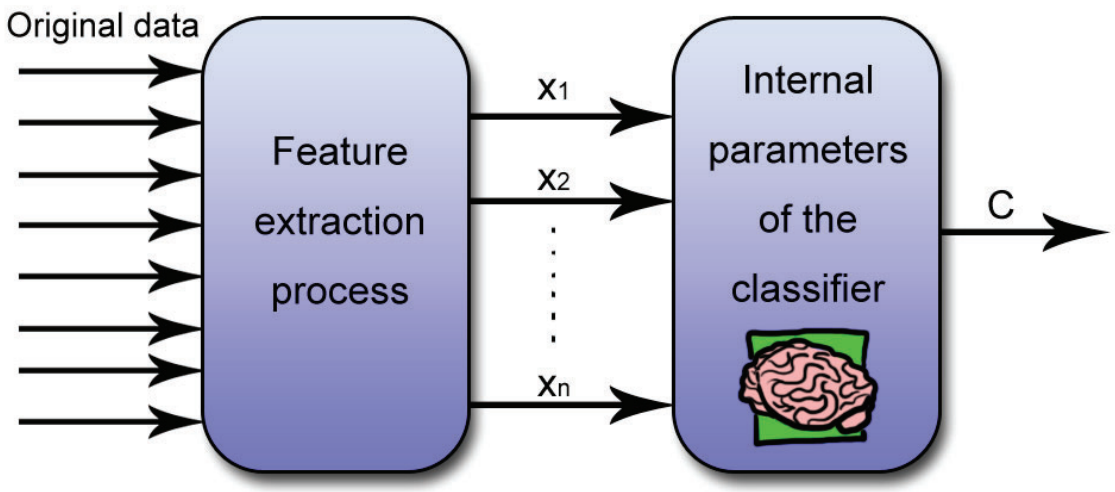


Figure 3.1: Supervised learning architecture: feature extraction and classification.

Let's now define the machine learning context (ML for short) so that its high voltage can be switched on without electric shock injuries caused by improper usage and over-optimism. Furthermore, switching on ML does not imply switching off our brain.

In fact, before starting the machine learning process, it is useful to clean and extract an informative subset or a combination of the original data by using intuition and intelligence. Features (or attributes) are individual measurable properties of the phenomena being observed with useful information to derive the output value. **Feature selection** (for a subset) and **feature extraction** (for a combination) are the names of this preparatory phase (Fig. 3.1).

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. The analogy is with a competent professor filtering and preparing teaching material for an effective lesson.

To fix the notation, a training set of ℓ *tuples* (ordered lists of elements) is considered, where each tuple is of the form $(\boldsymbol{x}_i, y_i)$, $i = 1, \ldots, \ell$; $\boldsymbol{x}_i$ 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*, where y_i is a real numeric value; and *classification*, 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 $y_i \in \{1, \ldots, N\}$. For example, think about classifying a mushroom as edible or poisonous.

In **regression**, the output is a real number, and the objective is to model the relationship between a dependent vari-

able (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 as anxious people and sellers of hair loss cures know very well.

In these cases there is a natural way to transform a "crisp" 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** 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* of 0.2 in the class of hairy persons can be more appropriate. Probability is used properly when there are experiments with uncertain outcome which can be repeated.

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 person or a more sophisticated system 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. In optical character recognition, 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. A subsequent step may consider adjacent characters or semantic information to disambiguate the two possibilities.

3.1 Learning from labeled examples: minimization and generalization

Supervised learning uses the examples to build an association (a function) $y = \hat{f}(\boldsymbol{x})$ 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**) $\boldsymbol{w}$.

For a concrete image, think about a mincer transforming inputs into outputs, with tunable gears and levers to regulate it. Or 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 is extracted from the training examples. The magic of ML is that the gears are not fixed by hand but automatically, through optimization, by showing examples of correct input-output pairs.

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 some human insight, while the **determination of the best parameters is fully automated**, this is why the method is called *machine* learning after all. The free parameters are fixed by **demanding that the learned model works correctly on the examples** in the *training set*.

As a true believer in the power of optimization one defines an *error measure* to be minimized[1], and runs (automated) optimization to determine the 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 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,

[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.

by sampling neighboring points with his feet, and by moving always in the direction of steepest descent. No "human vision" is available to the computer to "see" the lakes, only the possibility to sample one point at a time. By repeating the two steps of sampling in the neighborhood of the current point – in $\boldsymbol{w}$ space – and moving to a neighbor that decreases the error, one builds a trajectory leading to smaller and smaller values. Amazingly, this simple process is sufficient to reach an appropriate $\boldsymbol{w}^*$ value for many applications.

Let's speak mathematics. 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 with respect to 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 [383, 384, 311].

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 in our neural network tends to depend smoothly on the neuron inputs, in turn caused by chemical and electrical interactions in their dendritic trees.

Up to now, you may think that machine learning equals optimization of a performance measure on the training examples, but one ingredient is still missing. 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 controlling the "model complexity", i.e., the proper architecture and number of parameters, to reach an appropriate compromise between bias and variance.

The issue goes back to problems related to measuring physical systems. When measuring a system one encounters two kinds of possible errors: *systematic errors* (bias) and *random errors* (variance), as shown in Fig.3.2. Systematic errors, leading to *inaccuracy*, refer to deviations that are not due to chance alone. An example occurs with a measuring device that is improperly calibrated so that it consistently overestimates (or underestimates) the measurements. Random error has no preferred direction, so we expect that averaging over a large number of observations will yield a net effect of zero. The impact of random error, *imprecision*, can be minimized with large sample sizes. The precision related to the number of significant figures in digits: it refers to the *stability* of that measurement when repeated many times. If you measure height of a person with a mater stick (say 182 centimeters), it is useless to use more than 3-4 digits, nobody would answer 182.368638 cm). Bias, on the other hand, has a net direction and magnitude, so that averaging over a large number of observations does not eliminate its effect.

Let's now focus on machine learning and introduce the **bias–variance decomposition of squared error**[212]. The context is that the output is obtained by a true functional, but noisy relation of the input: $y_i = f(x_i) + \epsilon$, where the noise ϵ has zero mean and variance σ^2.

The objective of ML is to find a function $\hat{f}(x)$, that approximates the true function $y = f(x)$ as well as possible, in particular by minimizing the squared error $(y - \hat{f}(x))^2$. Given the noise ϵ in the true function one must be prepared to accept an *irreducible error* in any function $\hat{f}(x)$. If one samples from the same distribution and averages over the

[2]In some cases the closeness between points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ can be measured by the standard Euclidean distance, in other cases more problem-specific distance measures are needed.

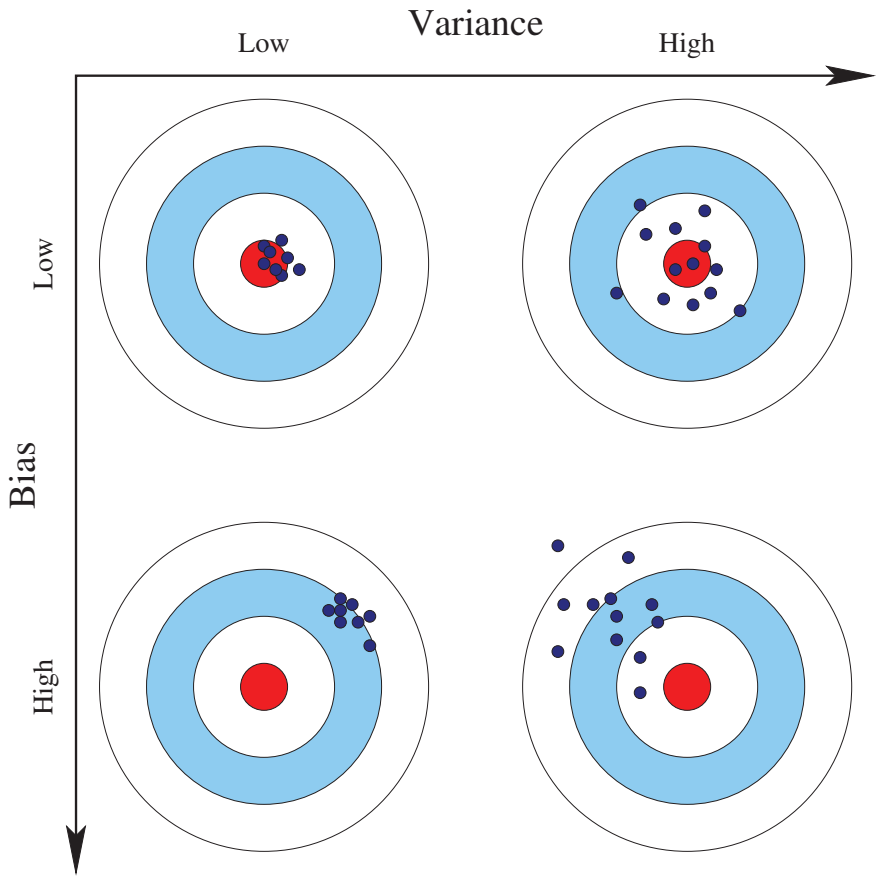


Figure 3.2: Difference between bias (systematic errors) and variance (random errors) when playing darts.

different choices of $x_1, \ldots, x_n, y_1, \ldots, y_n$, one can decompose the expected error on an unseen sample x as follows:

$$\mathrm{E}\Big[\big(y - \hat{f}(x)\big)^2\Big] = \mathrm{Bias}\big[\hat{f}(x)\big]^2 + \mathrm{Var}\big[\hat{f}(x)\big] + \sigma^2$$

Where Bias measures how the expected model output differs from the "true" $f(x)$ value:

$$\mathrm{Bias}\big[\hat{f}(x)\big] = \mathrm{E}\big[\hat{f}(x)\big] - f(x)$$

and Var measures the scatter in the distribution of $\hat{f}(x)$ values around its mean:

$$\mathrm{Var}\big[\hat{f}(x)\big] = \mathrm{E}\Big[\big(\hat{f}(x) - \mathrm{E}[\hat{f}(x)]\big)^2\Big]$$

The irreducible error σ^2 gives a lower bound on the expected error on unseen examples. The more complex and flexible the model $\hat{f}(x)$ is, the lower the bias will be. However, flexibility will make the model details "move" more when the training examples change, and hence its variance will be larger.

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

[3] Occam's razor is attributed to the 14th-century theologian and Franciscan friar Father William of Ockham who wrote "entities must not be

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 for 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 $\boldsymbol{x}$, one assigns a class y by maximizing the posterior probability of a class given the 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 shortcut implied by the second option is profound, **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.

Understanding that one does not need to be a domain expert to bring a measurable contribution is a small step for a man, but one giant leap along the LION way. Needless to say, successful businesses complement expertise with humble but powerful data- and optimization-driven tools.

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 shameful and unforgivable mistake to evaluate the performance of the learning systems on the same examples used 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 is why questions at university exams change from time to time...

Let's assume we have a supervisor (a software program or an experimental process) who can generate labeled examples with a given probability distribution. 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 seven mushrooms.

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. 3.3. A typical performance measure is the **root mean square (abbreviated RMS)** error between the output of

multiplied beyond necessity" (*entia non sunt multiplicanda praeter necessitatem*). To quote Isaac Newton, "We are to admit no more causes of natural things than 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."

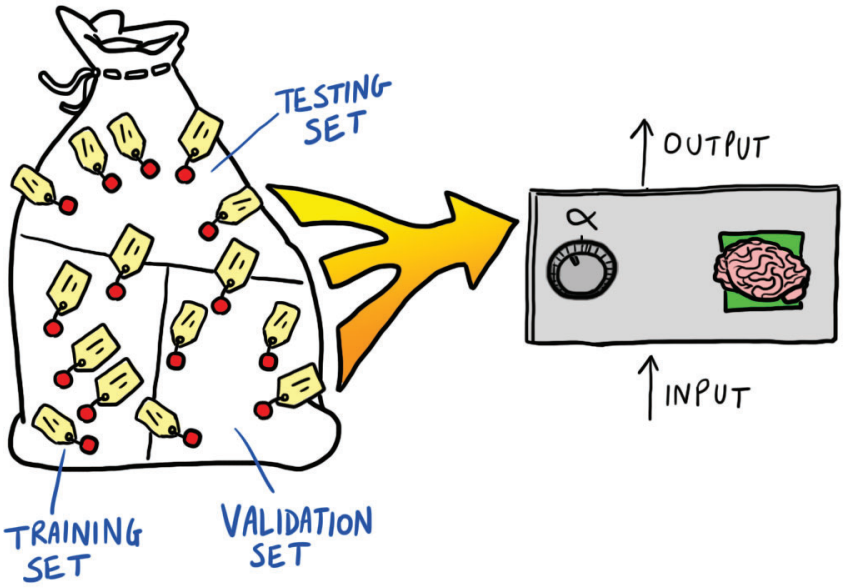


Figure 3.3: Labeled examples need to be split into training, validation and test sets.

the system and the correct output given by the supervisor. The RMS value of a set of values is the *square root* of the arithmetic *mean* (average) of the *squares* of the original values. If e_i is the error on the i-th example, the RMS value is given by:

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

.

In general, the learning process optimizes the model parameters to make the model *reproduce* the output of 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 set will be larger than the error on the training set. This discrepancy is likely to become severe if training is excessive, leading to **over-fitting** (**overtraining**), 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 use more of them for training and risk a poor and noisy measurement of the performance, or to have a more robust measure at the price of reducing the training examples? 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. $K-1$ subsamples are used for training, a single subsample is used for validation. The process is then repeated K times (the folds), with each

of the K subsamples used exactly once for validation. The results from the folds are then averaged to produce a single estimation. The advantage 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 (in this case K equals the number of examples).

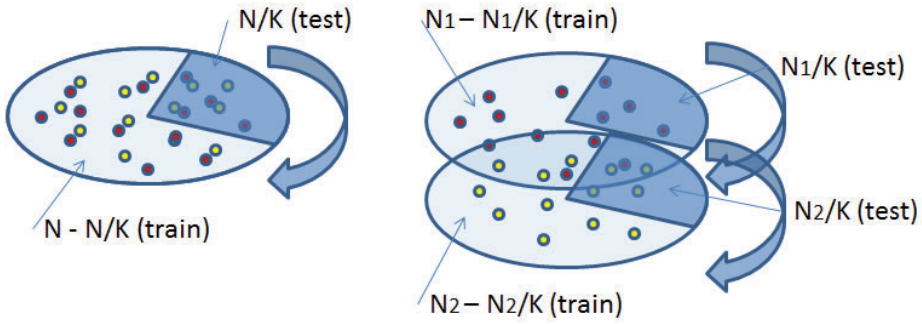


Figure 3.4: Stratified cross-validation, examples of two classes. In normal cross-validation $1/K$ of the examples are kept for testing, the slice is then "rotated" K times. In stratification, a separate slice is kept for each class to maintain the relative proportion of cases of the two classes.

Stratified cross-validation is an improvement to avoid different class balances in the training and validation set. It avoids that, by chance, a class is more present in the training examples and therefore less present in validation (with respect to its average presence among all examples). With stratification, the ℓ/K testing patterns are extracted separately for examples of each class, to ensure a fair balance among the different classes (Fig. 3.4).

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, the weights of the "multi-purpose box" to be customized. Think for example about deciding the termination criterion for an iterative minimization technique (when to stop training), or the number of hidden neurons in a multilayer perceptron, or appropriate values for the crucial parameters of Support Vector Machine (SVM). Finding optimal values for the meta-parameters implies **reusing** the validation set 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 one reuses 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*.

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.

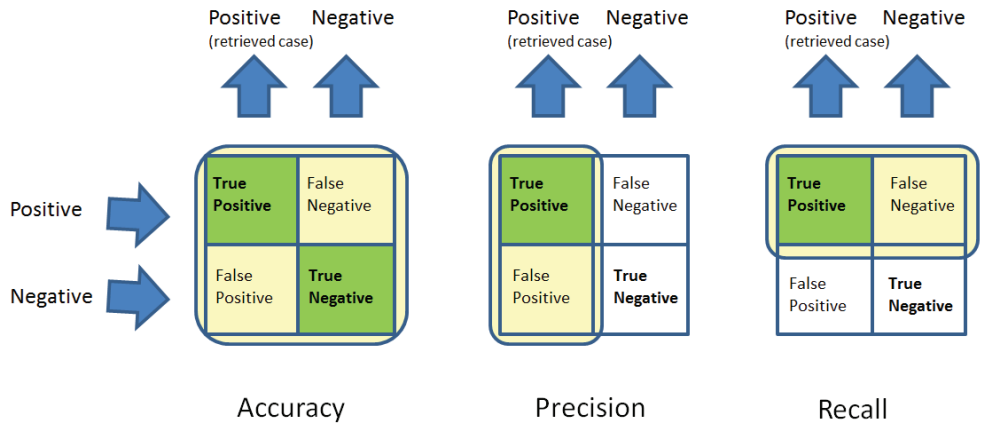


Figure 3.5: Each line in the matrix reports the different classifications for cases of a class. You can visualize cases entering at the left and being sorted out by the different columns to exit at the top. Accuracy is defined as the fraction of correct answers over the total, precision as the fraction 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 dark gray part by the cases in the light gray 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 confusions (Fig. 3.5).

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** is the number of true positives (the items *correctly* labeled as belonging to the positive class) divided by the total number of elements labeled as positive (the sum of true positives and false positives, 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. 3.6).

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).

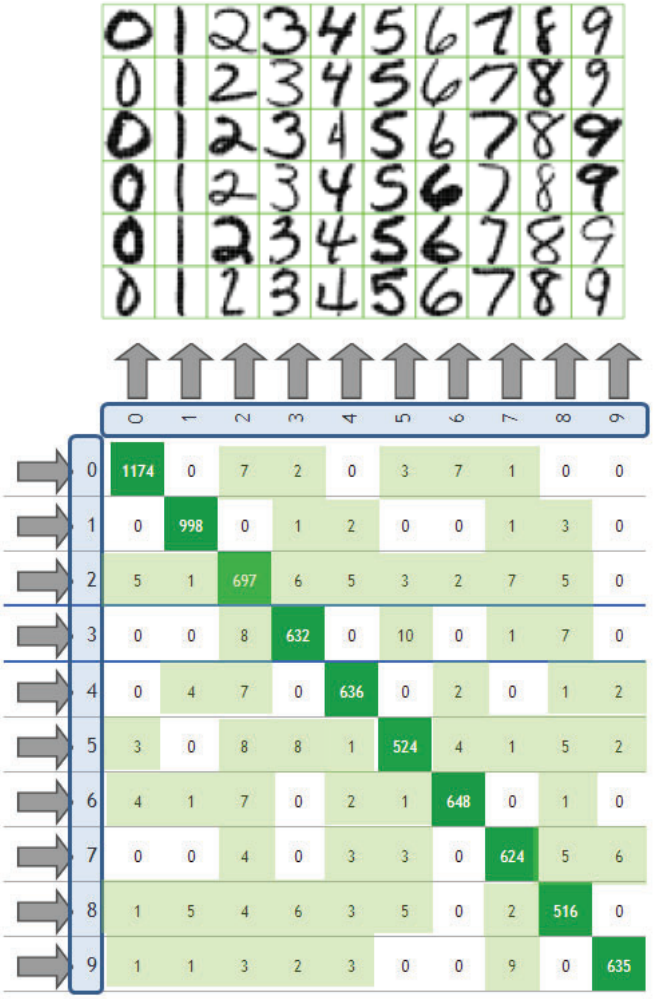


Figure 3.6: Confusion matrix for optical character recognition (handwritten ZIP code digits). The various confusions make sense. For example the digit “3” is recognized correctly in 632 cases, confused with “2” in 8 cases, with “5” in 10 cases, with “8” in 7 cases. “3” is never confused as “1” or “4”, digits with a very different shape.

Gist

The goal of machine learning is to exploit 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 overly complex models, and therefore to 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 with respect to systems designed by hand from domain knowledge.

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, and 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.

There 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.

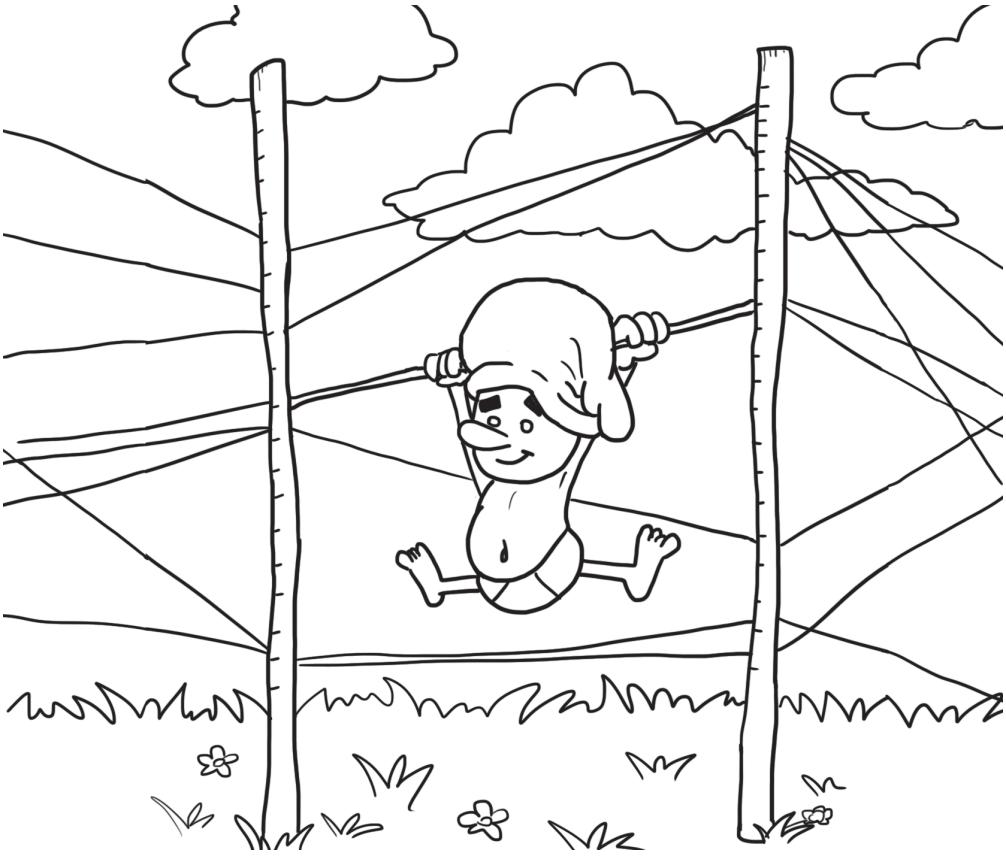
Part I

Supervised learning

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 equations 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.

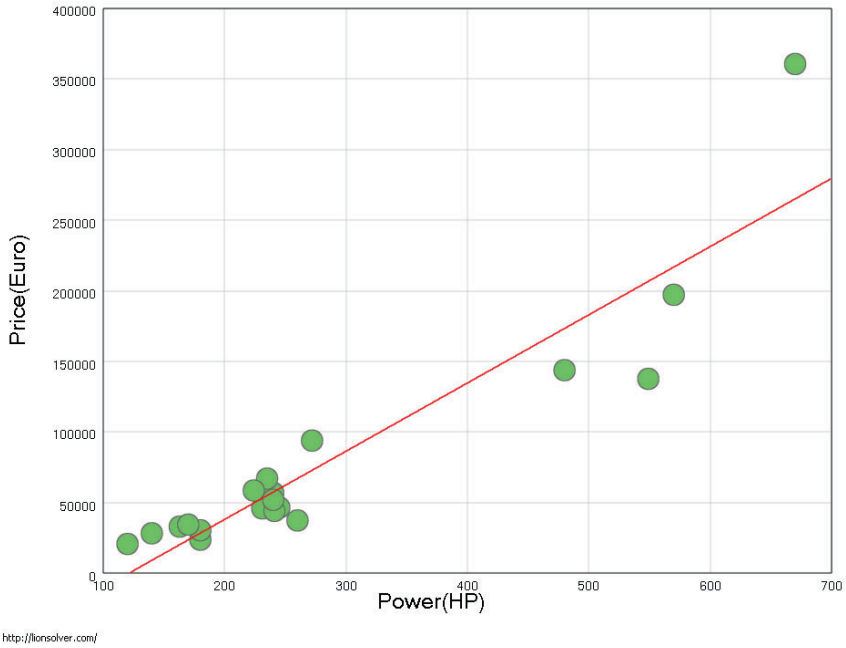


Figure 4.1: Data about price and power of different car models. A linear model is shown as a line.

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 (the line) 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

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

[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.

where $\boldsymbol{w} = (w_1, \ldots, w_d)$ is the vector of *weights* to be determined and ϵ_i is the error. In some cases the error ϵ_i is assumed to have a Gaussian distribution. 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 needed 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 the experimental data as closely as possible. 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, measurements have errors, 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), low but typically larger than zero.

To use a linear model, you do not need to know *how* the equation is minimized. True believers in 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.

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 $f(0) = 0$. One can 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 through the simple creation of an additional dummy input fixed to 1. One defines $\boldsymbol{x} = (1, x_1, \ldots, x_d)$, so that equation (4.1) remains valid also for affine models.

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 but it is applied to *nonlinear features* calculated from the raw input data instead of the original input $\boldsymbol{x}$. 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 $\boldsymbol{\varphi}(\boldsymbol{x}) = (\phi_1(\boldsymbol{x}), \ldots, \phi_n(\boldsymbol{x}))$ 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 $\boldsymbol{\varphi}(\boldsymbol{x})$, as follows:

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

The output is a weighted sum of the derived features.

4.3 Linear models for classification

Section 4.1 considers a linear function that fits the observed data, 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 perpendicular to the 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 classification. The method for determining the **best separating linear function** (geometrically a hyperplane) depends 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. Furthermore, least squares not only achieves

separation of the two classes (if separation is possible), but **robust separation, with a boundary which is far from the examples**, a *leitmotif* we will encounter in the following chapters. By forcing the outputs to be far (+1 and -1) and penalizing squared errors, the fragility caused by accepting any separating hyperplane disappears. A generic separating hyperplane may cause some examples of the two classes to have outputs very close to zero (like 0.000001 for the "+1" class, -0.000001 for the "−1" class), a small noise or measurement error on new cases will suffice to ruin the correct classification, and have them cross the boundary to the wrong side (see also Fig 12.1).

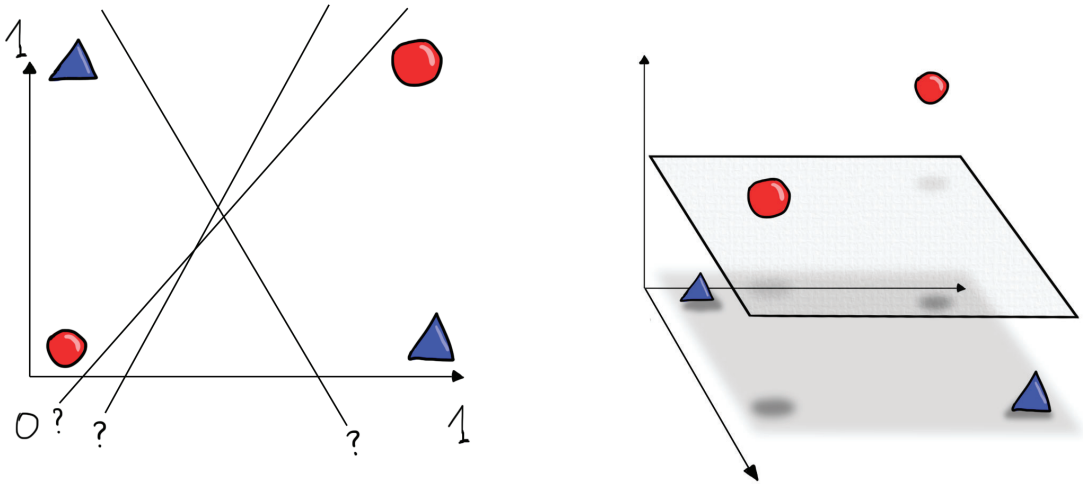


Figure 4.2: A case where a linear separation is impossible (XOR function, left). A linear separability with a hyperplane can be obtained by mapping the point in a nonlinear way to a higher-dimensional space.

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) 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 that 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.

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 communicate via chemical synapses (Fig. 4.3). Synapses [2]

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

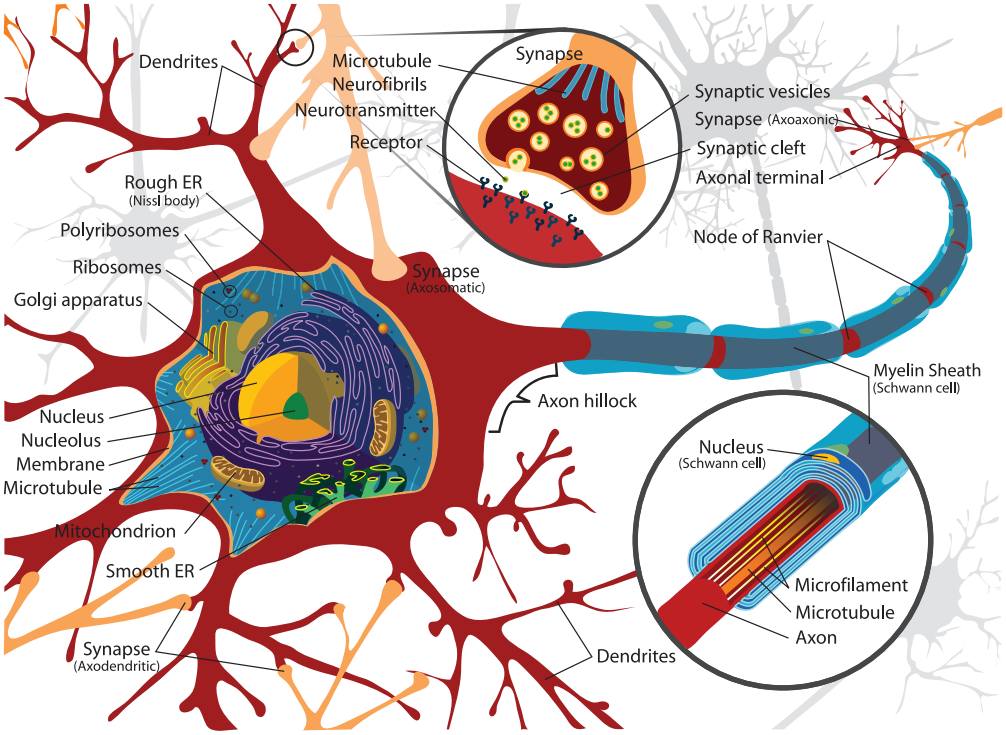


Figure 4.3: Neurons and synapses in the human brain.

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

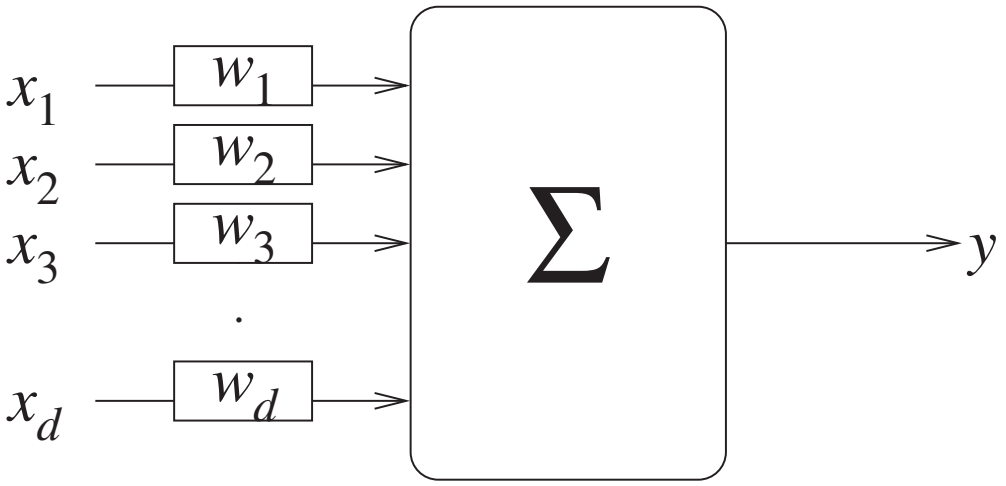


Figure 4.4: The Perceptron: the output is obtained by a weighted sum of the inputs passed through a final threshold function.

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 can be seen as the abstract and functional representation of a single nerve cell.

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.

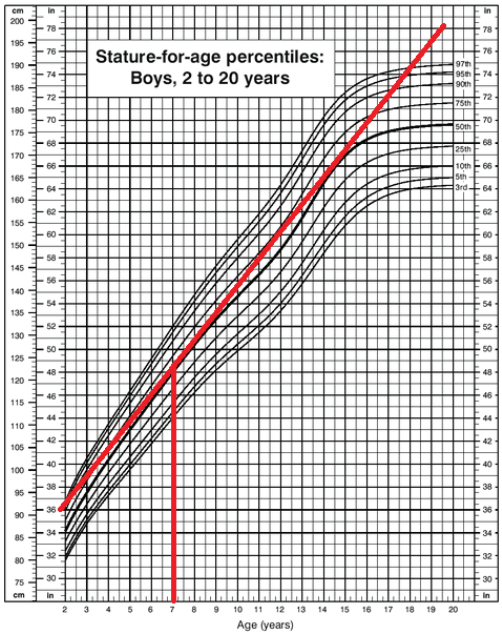


Figure 4.5: Functions describing physical phenomena tend to be *smooth*. The stature-for-age curve can be approximated well by a tangent line (dark) from 2 to about 15 years.

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:

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

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 neighborhood 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 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 the system too much to reduce the error on the training examples. Reaching very small or zero training error can 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. Actually, one 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.

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 equation determines 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 least-squares "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 the *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 synaptic weights. 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 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 connecting the best-fit line to the experimental data.

If features are transformed by some $\boldsymbol{\varphi}$ function (as a trick to deal with nonlinear relationships), the solution is very similar. Let $\boldsymbol{x}'_i = \boldsymbol{\varphi}(\boldsymbol{x}_i)$, $i = 1, \ldots, \ell$, 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:

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

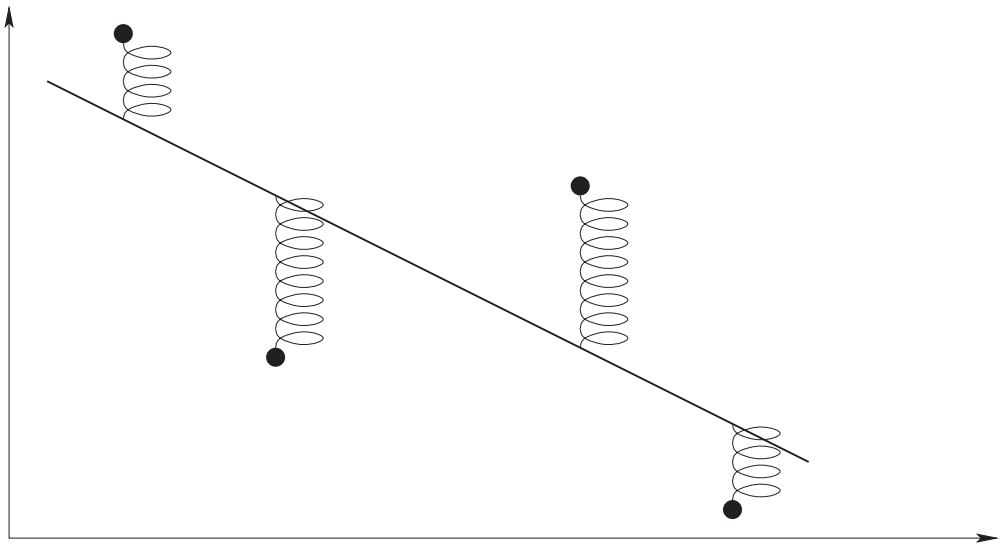


Figure 4.6: Physics gives an intuitive spring analogy for least squares fits. The best fit is the line that minimizes the overall potential energy of the system (proportional to the sum of the squares of the spring length).

4.7 Numerical instabilities and ridge regression

Real numbers (like π and "most" numbers) cannot be represented in a digital computer, they are "faked". 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 $x_1, \ldots, x_\ell$ 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.

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:

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

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

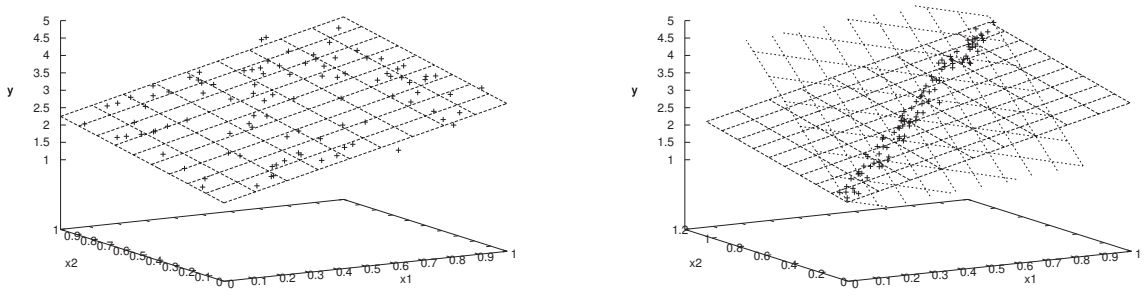


Figure 4.7: A well-spread training set (left) provides a stable numerical model, whereas a bad choice of sample points (right) may result in wildly changing planes, including very steep ones (adapted from [27]).

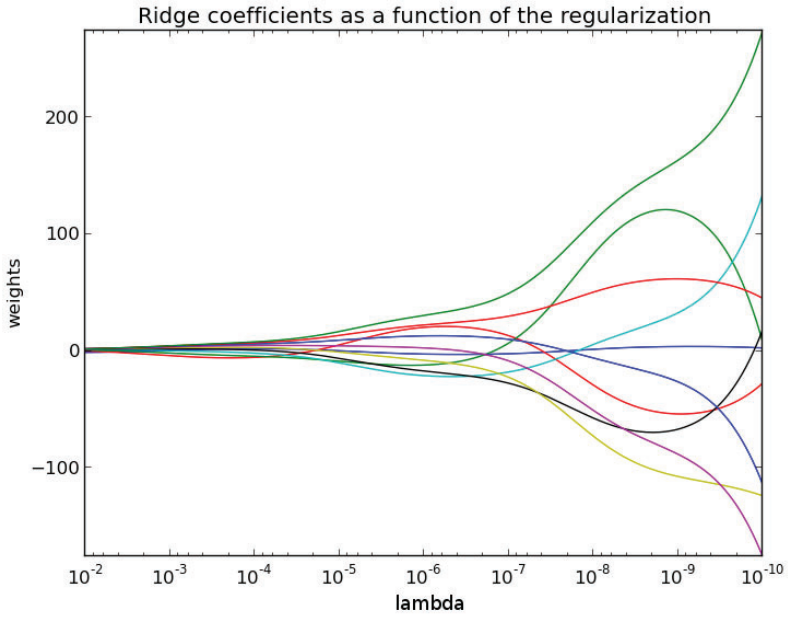


Figure 4.8: A "ridge" plot showing how weights tend to be contracted for large values of λ.

the right plot of Fig. 4.7. The term "ridge" refers to the graphical ridge pattern created when the optimal weights are plotted as a function of λ. As you can imagine, larger λ values causes an overall shrinkage of the weights (Fig. 4.8).

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 but 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 output values of the training examples. Minimization can be "one shot" by generalizing matrix inversion in linear algebra (**pseudo-inverse**), 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.

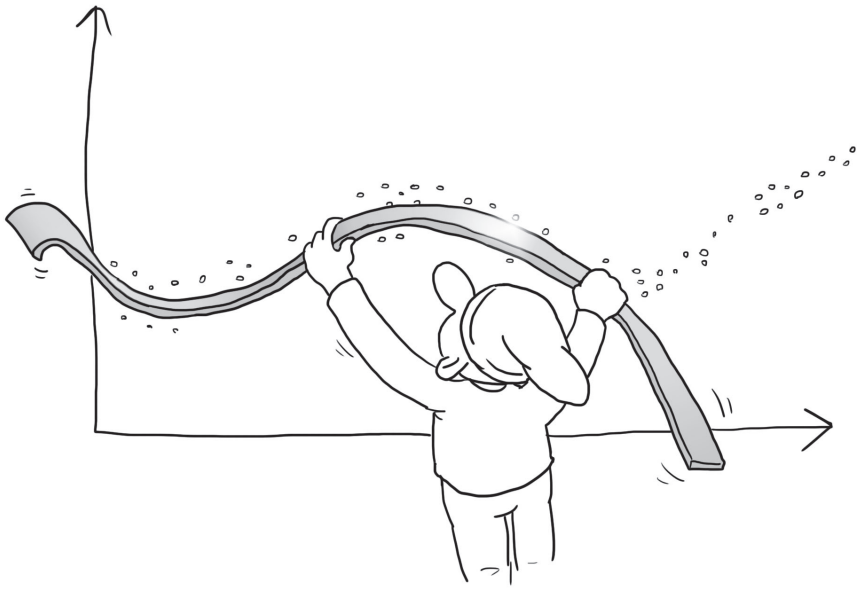
Real numbers do not live in a computer and their approximation by limited-size binary numbers is a possible cause of mistakes and instability (small perturbations of the sample points leading to large changes in the results).

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, models with the coefficients appearing in a linear manner (a.k.a. linear-in-the-coefficients). The output of a serious effort is not only a single "take it or leave it" model. Usually one deals 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, 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.

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 one can evaluate the function on new and unknown x values.

Identifying significant patterns and relationships implies eliminating insignificant details like measurement *noise*, stochastic errors cause by the finite-precision physical measurements. Think about modeling how the height of a person changes with age. If you repeat measuring with a high precision instrument, you will 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.

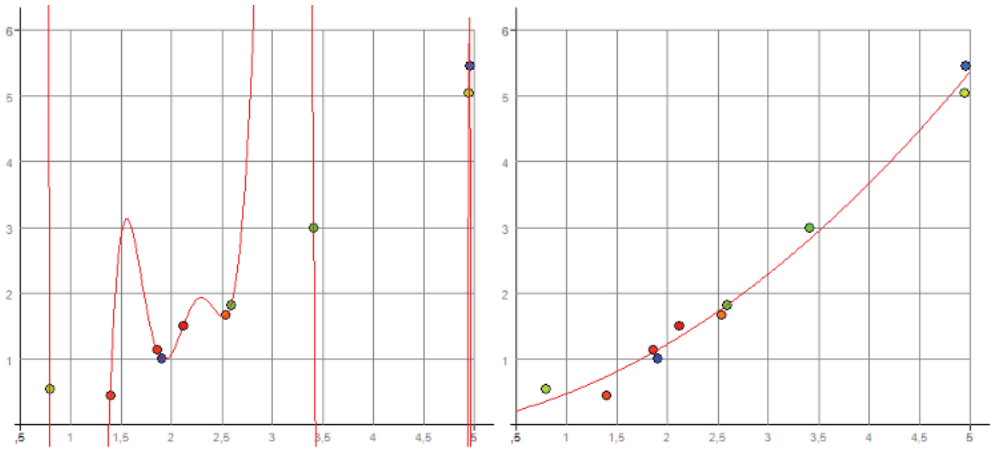


Figure 5.1: Comparison between interpolation and fitting. The number of free parameters of the polynomial (equal to its degree minus one) changes from the number of data points (left), to three (right).

Coming back to the model, we do *not* demand that the function models also the noise and that the plot passes exactly through the sample values. I.e., we don't require that $y_i = f(x_i)$ for all points. We do not deal with **interpolation** but with **fitting** (being compatible, similar or consistent). Losing absolute fidelity is not a weakness but a strength, providing an opportunity to simplifying the analysis, create more powerful models and permit reasoning that isn't bogged down by trivia. A comparison between a function interpolating all the examples, and a much simpler one, like in Fig. 5.1, shows the obvious difference. *Occam's razor* illustrates this basic principle that simpler models should be preferred over unnecessarily complicated ones.

The freedom to choose among 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.

5.1 Goodness of fit and and chi-square

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 meter stick, 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.

Now that you have a precise way of measuring the quality of a polynomial model by the normalized *chi-square*, 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:

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

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.

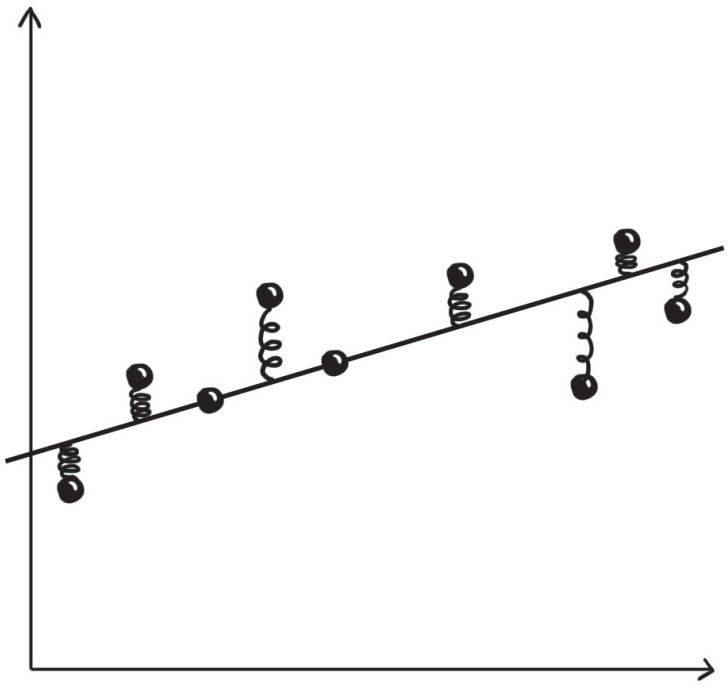


Figure 5.2: A fit by a line with a physical analogy: each data point is connected by a spring to the fitting line. The strength of the spring is proportional to $1/{\sigma_i}^2$. The minimum energy configuration corresponds to the minimum *chi-square*.

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.

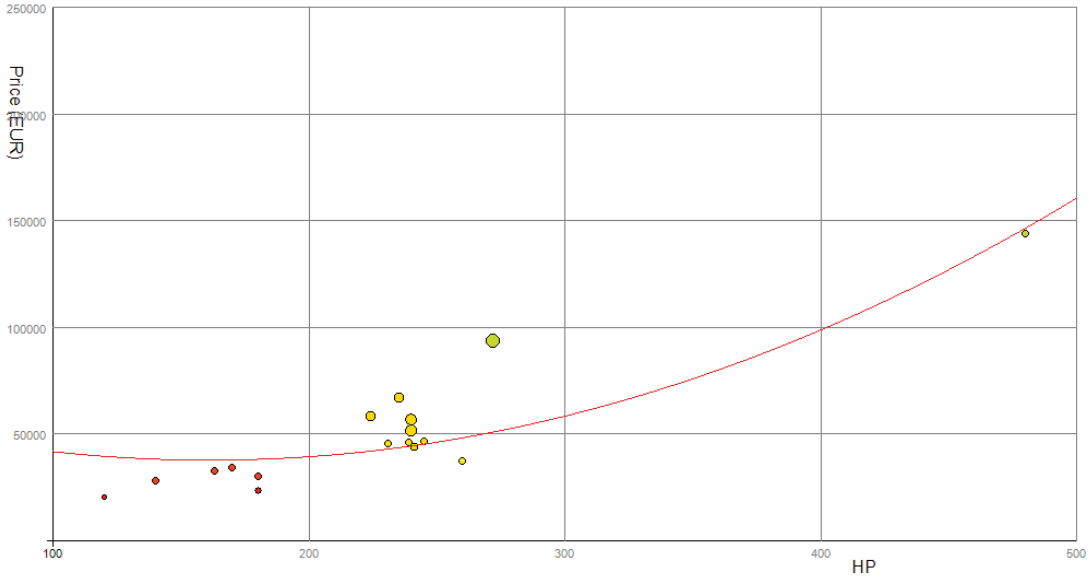


Figure 5.3: A polynomial fit: price of cars as a function of engine power.

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 line against 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 with respect to 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.

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 Least squares and maximum likelihood estimation

Now that the basic technology for generalized least-squares fitting is known, let's consider additional motivations from statistics. 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.

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 is now clear.

5.2.1 Hypothesis testing

Statistical hypothesis testing can be used to **judge the quality** of a model. The fundamental question to ask is: considering the N experimental points and the estimated M parameters, what is the probability that, by chance, values equal to or larger than the 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.

Fisher introduced the concept of **null hypothesis** by an example[127], the now famous "lady tasting tea" experiment. A lady claimed the ability to determine the means of tea preparation by taste. Fisher proposed an experiment and an analysis to test her claim. She was to be offered 8 cups of tea, 4 prepared by each method, for determination. He proposed the null hypothesis that she possessed no such ability, so she was just guessing. With this assumption,

the number of correct guesses (the test statistic) formed a binomial distribution. Fisher calculated that her chance of guessing all cups correctly was 1/70. Fisher commented: "... the null hypothesis is never proved or established, but is possibly disproved, in the course of experimentation. Every experiment may be said to exist only in order to give the facts a chance of disproving the null hypothesis."

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 $\nu = N - M$. 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). 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[1].

The **reduced chi-square** statistic χ^2_{red} is simply the chi-squared divided by the number of degrees of freedom, in our example $\nu = N - M$. The advantage of the reduced chi-squared is that it already normalizes for the number of data points and model complexity. Some rules of thumb follow.

- A value $\chi^2_{\text{red}} \approx 1$ 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_{red} 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_{red} 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. The model is 'over-fitting' the data: either the model is improperly fitting noise, or the error variance has been overestimated. We should try reducing the polynomial degree[2].

The importance of the **number of degrees of freedom** ν, which decreases when 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) with respect to one with smaller errors but a larger number of parameters.

By using the goodness-of-fit Q measure one 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.

[1] 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.

[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.

5.2.2 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 with respect to 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[3]. 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**. 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.

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 once (Fig. 5.5). By simple math and for large numbers N of examples, about 37% of the examples (actually approximately

[3] 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.

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

$1/e$) are not present in a sample, because they are replaced by multiple copies of the original cases[4].

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

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**

[4]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 [300].

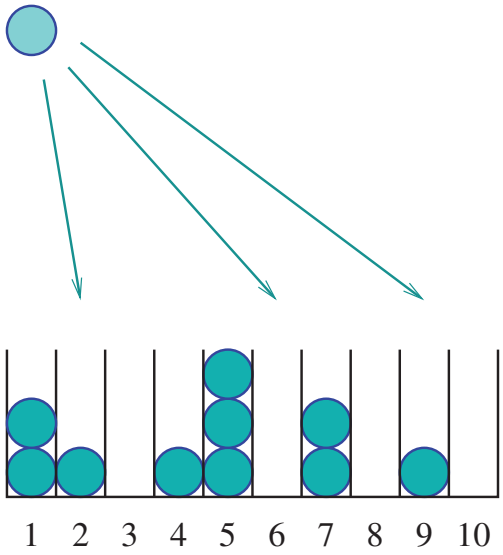


Figure 5.5: Bootstrapping: 10 balls are thrown with uniform probability to end up in the 10 boxes. They decide which cases and how many copies are present in the bootstrap sample (two copies of case 1, one copy of case 2, zero copies of case 3,...

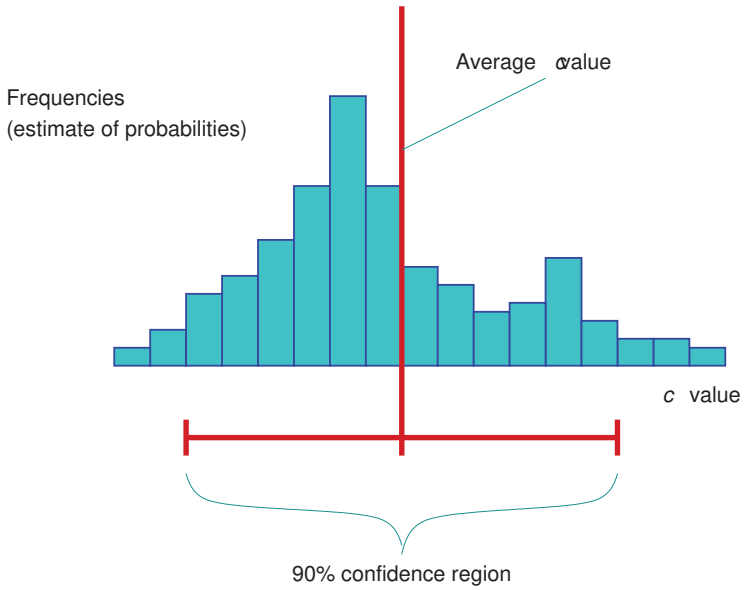


Figure 5.6: Confidence interval: from the histogram characterizing the distribution of the estimated c values one derives the region around the average value collecting 90% of the cases. Other confidence levels can be used, like 68.3%, 95.4%. etc. (the historical probability values corresponding to σ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

value μ. Given a set $\mathcal{X}$ of N values x_i, the average is

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

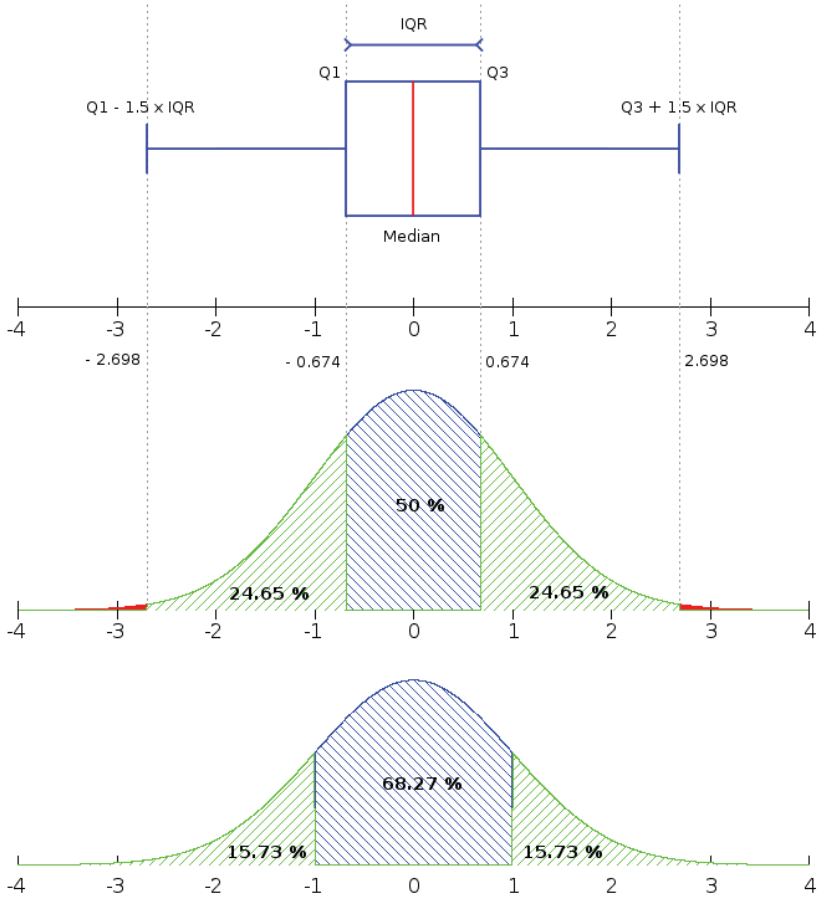


Figure 5.7: Comparison between box plot (above) and a normal distribution. The X axis shows positions in terms of the standard deviation σ. For example, in the bottom plot, 68.27% of the points fall within plus or minus one σ from the mean value.

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 (25th percentile), the median, and the upper quartile (75th 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-in-the-coefficients 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 coefficients is large with respect to the number of input-output examples, **over-fitting** appears and it is dangerous to use the model to derive outputs for novel input values.

The **goodness of a polynomial fit** 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 the sand of invalid assumptions**. Luckily, approaches based on easily affordable massive computation like cross-validation are easy to understand and robust.

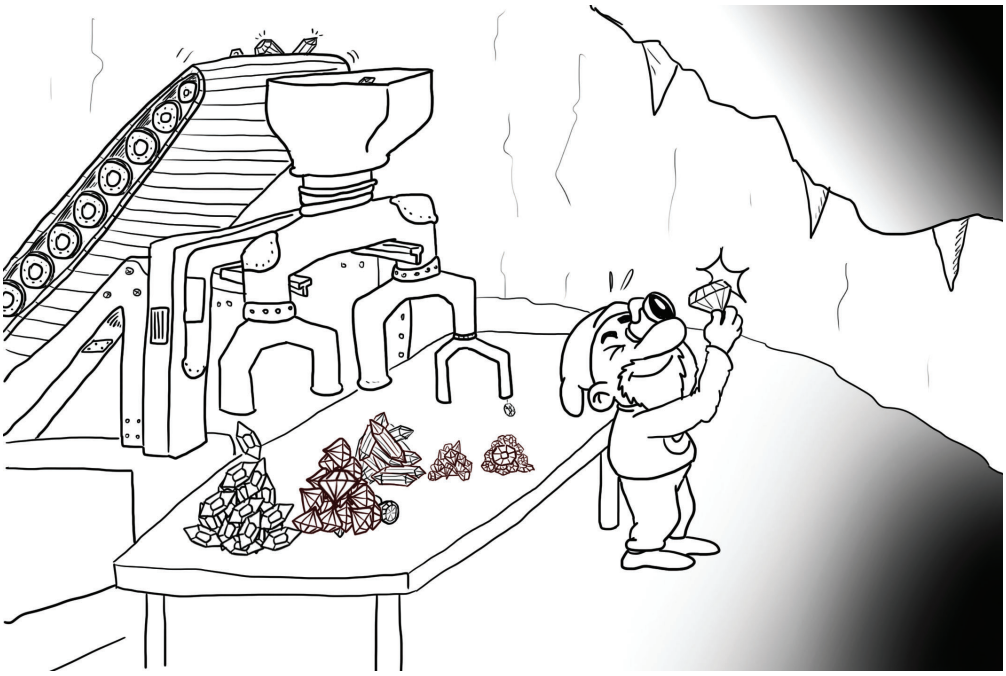
"Absurdities" like **bootstrapping** (re-sampling the same data with replacement, and repeating the estimation process in a Monte Carlo fashion) can be used to obtain confidence intervals around estimated parameter values.

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".

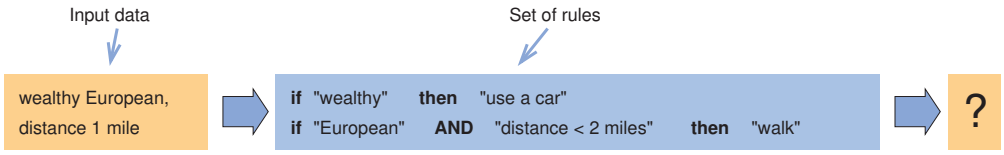


Figure 6.1: A set of unstructured rules can lead to contradictory classifications.

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 which can be 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 to ask next depends on the answer to the previous questions 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, until the problem is so simple (the leaf) that the answer is ready.

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", consisting of examples of the same class. This class is returned as classification output for all cases trickling down to reach that leaf.

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

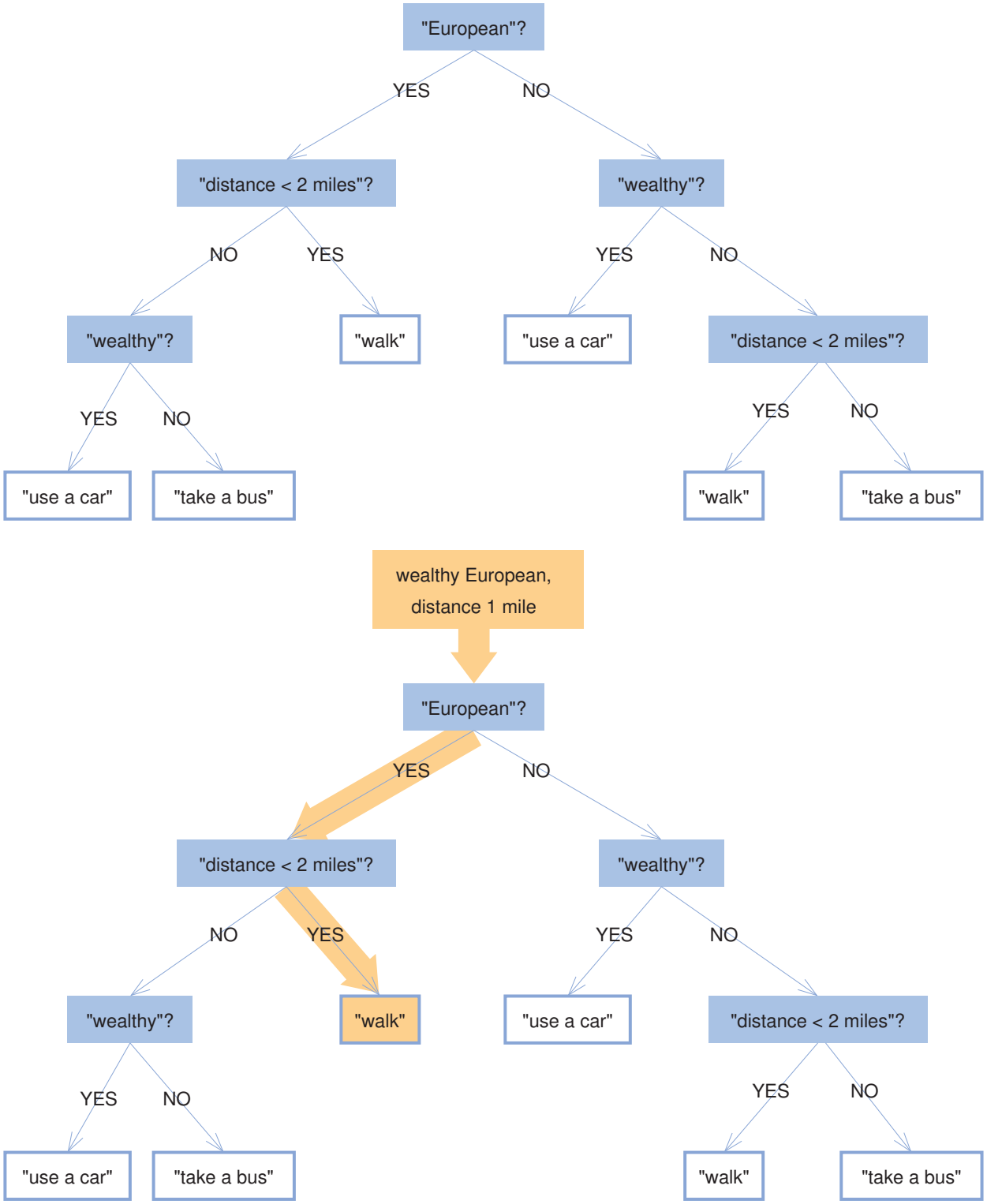


Figure 6.2: A decision tree (top), and the same tree working to reach a classification (bottom). The data point arriving at the split node is sent to its left or right child node according to the result of the test function.

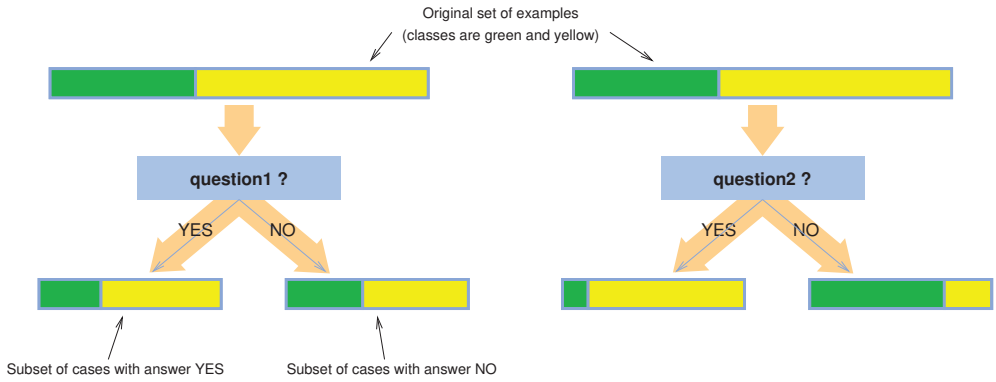


Figure 6.3: Purification of sets (examples of two classes): question2 produces purer subsets of examples at the children nodes.

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.

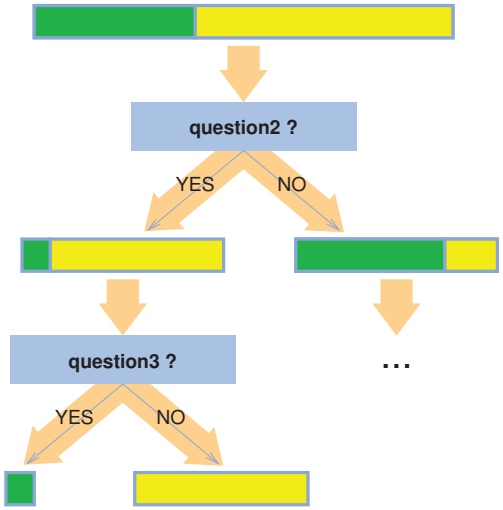


Figure 6.4: Recursive step in tree building: after the initial purification by *question2*, the same method is applied on the left and right example subsets. In this case *question3* is sufficient to completely purify the subsets. No additional recursive call is executed on the pure subsets.

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 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.

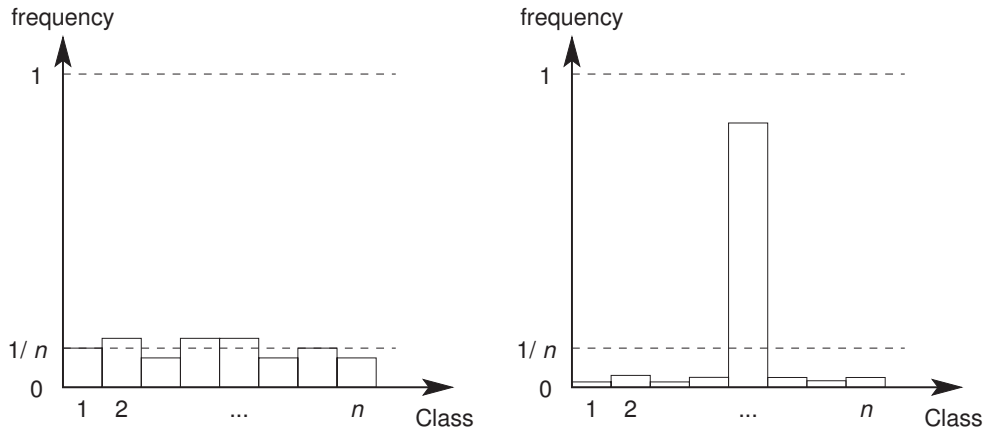


Figure 6.5: Two distributions with widely different entropy. High entropy (left): events have similar probabilities, the uncertainty is close to the maximum one ($\log n$). Low entropy (right): events have very different probabilities, the uncertainty is small and close to zero because one event collects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

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 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 [301]. 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. For each class, one adds 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 ($p_i \approx f_i$):

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

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 [66].

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** [66]. 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 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. 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.

As a final remark, do not confuse the construction of the decision trees (using the labeled examples, the purity measures, the choice of appropriate questions) with the usage of a built tree. When used, a input case is rapidly tested with a single chain of questions leading from the root to the final assigned leaf.

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[315, 29]. The analogy is with human society: in many cases setting up a **committee of experts** is a way to reach a decision with

[1] For curiosity, 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 socioeconomical variables — without any explanation for the layperson, of course.

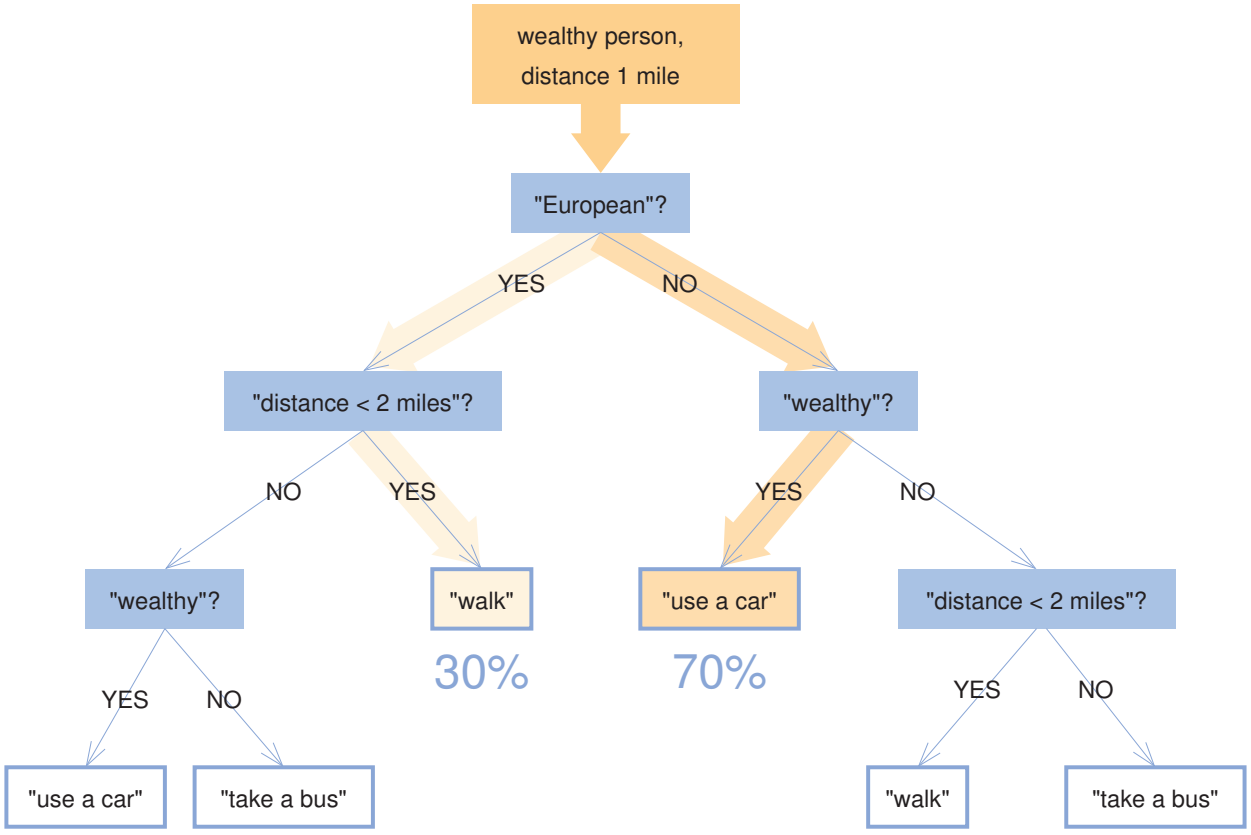


Figure 6.6: Missing nationality information. The data point arriving at the top node is sent both to its left and right child nodes with different weights depending on the frequency of "YES" and "NO" answers in the training set.

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" [353] 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).

Democratic ensembles seem particularly effective 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 [29] to using ensembles of trees in input devices for game consoles[2] [105].

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 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** [183, 65]:

- 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.

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

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.

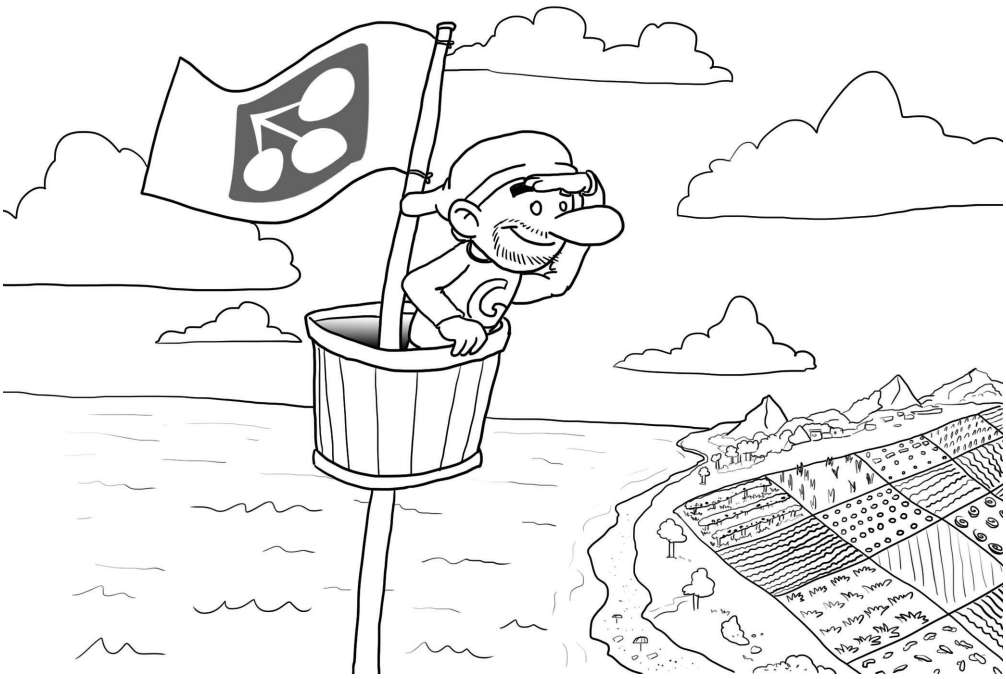
The abundance of memory and computing power permits training very large numbers of different trees. They can be fruitfully used as **decision forests** by collecting all outputs and averaging (regression) or voting (classification). Decision forests have various positive features: like all trees they 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 22) 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 becomes easier if the model is small.

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 problem with many possible solutions and without formal guarantees of optimality. No simple recipe exists.

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, more or less illuminated…), 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.

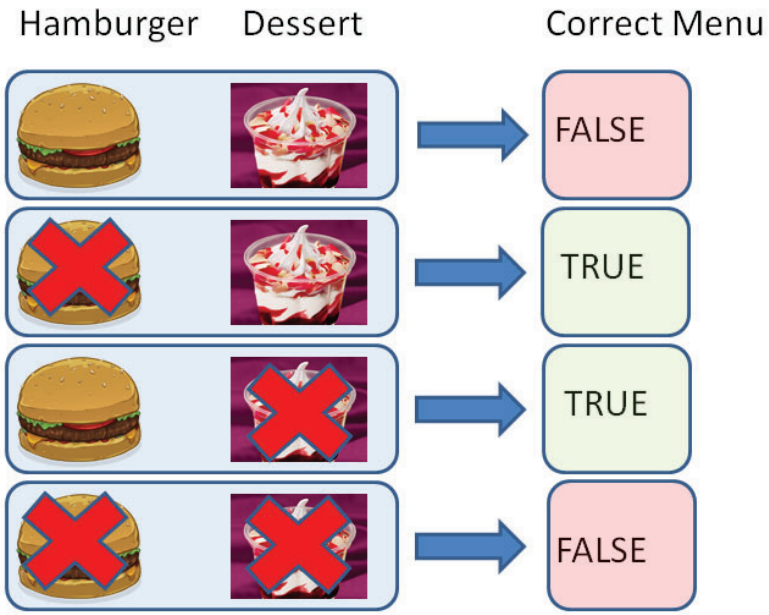


Figure 7.1: A classifier with two binary inputs and one output. Single features in isolation are not informative, both input features are needed and sufficient for a correct classification.

- **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 the ranked features in the order of their individual discrimination power and checks the effect on output error reduction through 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.

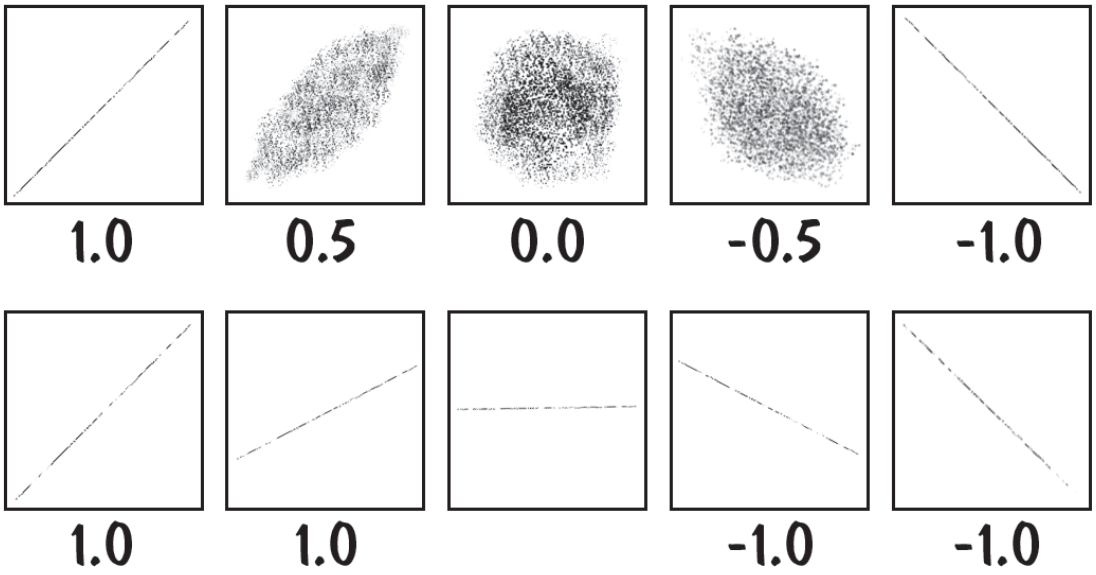


Figure 7.2: Examples of data distributions and corresponding correlation values. Remember that values are divided by the standard deviation, this is why the linear distributions of the bottom plots all have the same maximum correlation coefficient (positive 1 or negative -1). Also, note the sudden jump from +1 to -1 in the bottom plots: the correlation coefficient can be fragile in some cases.

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:

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

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

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

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. 37.1.

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 measures a **relationship between a numeric input and a categorical output**.

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.

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:

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

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:

$$\eta^2_{X_i,Y} = \rho^2_{X_i,C}.$$

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

7.4 Chi-square test to deny statistical independence

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 in 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 one can 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 Heuristic relevance based on nearest neighbors: Relief

There are multivariate relevance criteria to rank individual features according to their relevance *in the context of others*. To illustrate this concept, the Relief algorithm [233] derives a ranking index for classification problems based on the K-nearest neighbors algorithm. To evaluate the index, one first identifies for each example, in the original feature space, the K closest examples of the same class (nearest hits) and the K closest examples of a different class (nearest

misses.). Then, in projection on feature j, the sum of the distances between the examples and their **nearest misses** is compared to the sum of distances to their **nearest hits**.

The idea is that a feature is good if neighbors (in the original space) of the same class tend to have close values of that feature, while neighbors of different classes tend to have different values. The radical approximation is caused by considering only a certain number of nearest neighbors instead of the entire set of examples, and a simple discrimination based on the similarity of features in the projected space (in the subset of selected features).

In a variation fo Relief in [167], one first identifies in the original feature space, for each example $\mathbf{x}_i$, the K closest examples of the same class $\mathbf{x}_{H_k}(i), k = 1, ..., K$ (nearest hits) and the K closest examples of a different class $\mathbf{x}_{M_k}(i)$ (nearest misses). Then, considering the individual feature j, the sum of the distances between the examples and their nearest misses is compared to the sum of distances to their nearest hits. In equation (7.4), the ratio of these two quantities is used to create an index of relevance $Rel(j)$ independent of feature scale variations.

$$Rel(j) = \frac{\sum_{i=1}^{m}\sum_{k=1}^{K}|x_{i,j} - x_{M_k(i),j}|}{\sum_{i=1}^{m}\sum_{k=1}^{K}|x_{i,j} - x_{H_k(i),j}|} \tag{7.4}$$

The larger $Rel(j)$ the better the j-th feature is separating near misses from near hits. A randomized set of examples can be considered to speedup the evaluation.

Relief does not discriminate between redundant features, and it can be fooled by low numbers of training instances. Updates to the algorithm in order to improve the reliability, make it robust to incomplete data, and generalising it to multi-class problems are presented in [237].

7.6 Entropy and mutual information (MIFS)

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 is. 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 [21].

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

$$H(Y) = -\sum_{y \in Y} \Pr(y) \log \Pr(y). \tag{7.5}$$

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:

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

[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.

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 defined as the expected value of $H(Y|x_i)$ over all values $x_i \in X_i$ that the i-th feature can have:

$$H(Y|X_i) = E_{x_i \in X_i}\big[H(Y|x_i)\big] = \sum_{x_i \in X_i} \Pr(x_i) H(Y|x_i). \tag{7.6}$$

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:

$$I(X_i; Y) = I(Y; X_i) = H(Y) - H(Y|X_i). \tag{7.7}$$

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

$$I(X_i; Y) = \sum_{y, x_i} \Pr(y, x_i) \log \frac{\Pr(y, x_i)}{\Pr(y)\Pr(x_i)}. \tag{7.8}$$

In classification, Mutual Information is related to upper and lower bounds on the optimal **Bayes error rate**.

The Bayes error rate is the lowest possible error rate that a classifier can achieve. If we knew the true probability $\Pr(y|\boldsymbol{x})$ of every outcome $y = C_1, \ldots, C_\ell$ for any input pattern $\boldsymbol{x}$, then our best choice for a given pattern $\bar{\boldsymbol{x}}$ would be class C_i maximizing $\Pr(C_i|\bar{\boldsymbol{x}})$ or, equivalently, $p(\bar{\boldsymbol{x}}|C_i)\Pr(C_i)$, which is proportional to the former (by Bayes' rule "posterior equals prior times likelihood ratio"), $p(\boldsymbol{x}|y)$ being the probability density of $\boldsymbol{x}$ given the class y.

The probability of error for a given pattern $\bar{\boldsymbol{x}}$ is the sum of the other, non-winning probabilities, and the Bayes error rate is obtained by integrating the error probability over the whole input space:

$$\begin{aligned} E_{\text{Bayes}} = \Pr(\text{error}) &= \sum_{i=1}^{\ell} \int_{H_i} \left(\sum_{j \neq i} \Pr(C_j|\boldsymbol{x}) \right) p(\boldsymbol{x})\, d\boldsymbol{x} \\ &= \sum_{i=1}^{\ell} \sum_{j \neq i} \int_{H_i} p(\boldsymbol{x}|C_j)\Pr(C_j)\, d\boldsymbol{x}, \end{aligned} \tag{7.9}$$

where the input space has been partitioned into areas $H_1, \ldots, H_\ell$: H_i is the area of the feature space associated by the optimal classifier to output classes C_i, with ties broken in any way. Note the fact that no classifier can escape this error, which is caused by the intrinsic randomness of the outcome.

In the case of $\ell = 2$ classes, Eq. (7.9) reduces to

$$E_{\text{Bayes}} = \overbrace{\int_{H_1} p(\boldsymbol{x}|C_2)\Pr(C_2)\, d\boldsymbol{x}}^{E_1} + \overbrace{\int_{H_2} p(\boldsymbol{x}|C_1)\Pr(C_1)\, d\boldsymbol{x}}^{E_2},$$

and the two integrals correspond to the two grey areas in Fig. 7.3, where the feature space is optimally divided into regions H_1 and H_2 depending on the error distributions. This error is clearly irreducible: any classifier that operates differently is bound to have a larger error. For instance, refer to Fig. 7.3: if patterns in E' are attributed to C_1 (i.e., features $\boldsymbol{x} \in H'$ are classified as C_2 instead of the more likely C_1), then the additional term E' (dashed area) is added to the error.

In particular, an upper bound containing the Mutual Information is [63]

$$E_{\text{Bayes}} \leq \frac{1}{2} H(Y|X) = \frac{1}{2}\big(H(Y) - I(Y, X)\big),$$

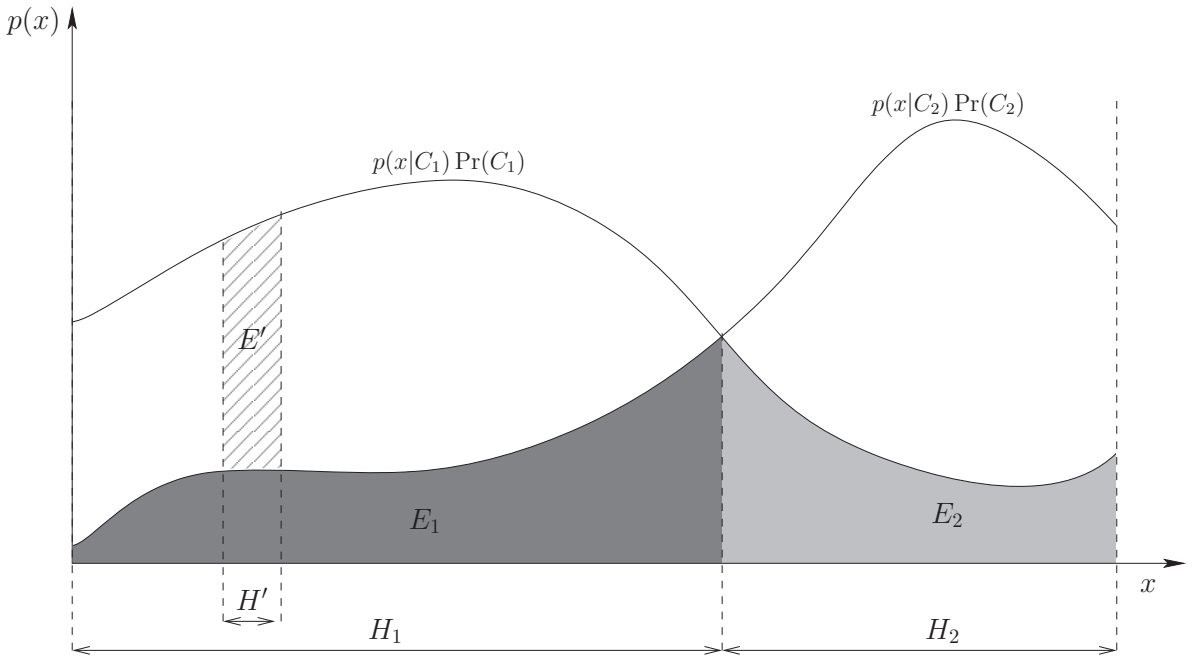


Figure 7.3: Optimal classification and Bayes error rate. Dashed area E' corresponds to additional error term for a suboptimal algorithm that misclassifies patterns in H'.

where X is a feature vector. Because the output class entropy cannot be changed, the upper bound is minimized when the Mutual Information $I(Y, X)$ is maximized.

Let's stress that the **mutual information is qualitatively different from the linear correlation**. A feature can be informative even if not linearly correlated with the output; a feature can provide useful information even if it is correlated with a second already-selected feature (linear correlation between two features does not imply redundancy!). 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.

But be advised: **having information is necessary to predict the output, but not sufficient**. The information has to be extracted by "learnable" techniques. A pseudo-random function linking a seed x to the generated value y gives the entire information to predict y from x, but identifying the function from examples can be demanding, if not impossible. Is this example too academic? Well, consider the case of an ID (like a randomized social security number). All details about a citizen can be predicted (actually, derived without ambiguity) after knowing the ID, but that does not mean that an ID is a useful feature to be used for generalization. When using MI for feature selection do not be fooled by IDs! **IDs (if randomized) should never be chosen as input features**.

Although very powerful in theory, estimating mutual information for vectors with large dimension after starting from labeled samples is not a trivial task. Discretization (quantization) of continuos variables into a discrete set of values, and substitution of probability density functions with counts of occurrences in **histogram bins** is s standard techniques, but large errors are present if most bins are empty or with very few events. In particular, estimation in very large-dimensional spaces is daunting because the number of bins can explode.

In some cases, estimates of the probability densities are derived before calculating MI, by using Parzen windows,

kernel methods, or the assumption of a specific form of the densities (like Gaussian). In contrast to conventional estimators based on hitograms and binnings, entropy estimates taken directly from k-nearest neighbor distances are presented in [240]: they are data efficient, adaptive (the resolution is higher where data are more numerous), and have minimal bias. A seminal definition of the MIFS strategy (Mutual Information for Feature Selection) and heuristic methods which use only mutual information between individual features and the output and between couples of features is presented in [21], aiming at selecting a set of relevant but non-redundant features.

7.6.1 Entropy and Mutual Information for continuous variables

Let X be a random variable with a probability density function $p(\boldsymbol{x})$ whose support is a set $\mathbb{X}$. The **differential entropy** $H(X)$, or $H(p)$, is defined as

$$H(X) = -\int_{\mathbb{X}} p(\boldsymbol{x}) \log p(\boldsymbol{x})\, d\boldsymbol{x}. \tag{7.10}$$

One must take care in applying properties of discrete entropy to differential entropy, since probability density functions can be greater than 1, so that the differential entropy can be negative.

The chain rule for differential entropy holds as in the discrete case:

$$H(X_1, \ldots, X_n) = \sum_{i=1}^{n} H(X_i|X_1, \ldots, X_{i-1}) \leq \sum H(X_i). \tag{7.11}$$

Differential entropy is translation invariant, i.e., $H(X + c) = H(X)$ for a constant c, but in general not invariant under arbitrary invertible maps. In particular, for a constant a, $H(aX) = H(X) + \log|a|$. This is why maximization only makes sense with additional contraints: for example, the maximum differential entropy distribution under constraint of a given variance is the Gaussian. Gaussian is the "least interesting" (least structured) distribution according to the entropic criterion, a fact which will be used in Projection Pursuit (Sec 20.6) and Independent Component Analysis (Sec. 21.2).

For a vector-valued random variable $\mathbf{X}$ and a matrix A, $H(A\mathbf{X}) = H(\mathbf{X}) + \log|\det A|$, $\det A$ being the determinant of the matrix.

In general, for a bijective map from a random vector to another random vector with same dimension $\mathbf{Y} = m(\mathbf{X})$, the corresponding entropies are related via [289]:

$$H(\mathbf{Y}) = H(\mathbf{X}) + \int f(\boldsymbol{x}) \log \left| \frac{\partial m}{\partial \boldsymbol{x}} \right| d\boldsymbol{x} \tag{7.12}$$

where $|\partial m/\partial \boldsymbol{x}|$ is the *Jacobian determinant of the transformation* m. The Jacobian matrix $\partial m/\partial \boldsymbol{x}$, containing partial derivatives $\partial m_i/\partial x_j$, defines a linear map which is the best linear approximation of the function m near the point $\boldsymbol{x}$. This linear map is thus the generalization of the usual notion of derivative. The absolute value of the **Jacobian determinant** of a square matrix at $\boldsymbol{x}$ gives us the factor by which the mapping m **expands or shrinks volumes** near $\boldsymbol{x}$. As a special case, when m is a rigid rotation, translation, or combination thereof, the Jacobian determinant is always 1, and therefore $H(Y) = H(X)$.

For the Mutual Information in the case of continuous random variables, summation is replaced by a definite double integral:

$$I(X;Y) = \int_Y \int_X p(\boldsymbol{x}, \boldsymbol{y}) \log \left(\frac{p(\boldsymbol{x}, \boldsymbol{y})}{p(\boldsymbol{x})\, p(\boldsymbol{y})} \right) d\boldsymbol{x}\, d\boldsymbol{y}, \tag{7.13}$$

where $p(\boldsymbol{x}, \boldsymbol{y})$ is now the joint probability density function of X and Y, and $p(\boldsymbol{x})$ and $p(\boldsymbol{y})$ are the marginal probability density functions.

The continuous mutual information $I(X;Y)$ has the distinction of retaining its fundamental significance as a measure of discrete information since it is actually the limit of the discrete mutual information of partitions of X and

Y as these partitions become finer and finer. Thus it is invariant under non-linear homeomorphisms (continuous and uniquely invertible maps), including linear transformations of X and Y.

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, and never pick randomized IDs as input features (convince yourself that the useful information in the feature can actually be *extracted* by ML techniques).

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

Models based on matrix factorization

La calunnia è un venticello / Un'auretta assai gentile
Che insensibile sottile / Leggermente dolcemente
Incomincia a sussurrar.
(Rossini — Il Barbiere di Siviglia)
Calumny is a little breeze, /a gentle zephyr,
which insensibly, subtly, / lightly and sweetly,
commences to whisper.

In this section we consider applications of linear algebra transformation to determine "factors of variations", in particular matrix factorization.

To make the presentation concrete, the guiding application is **collaborative filtering and recommendation**. Word of mouth has always been a powerful and effective technique to spread information and opinions from person to person, in a viral manner. It is a distributed and human way to **mine the data implicitly contained in many human minds**. It is effective because we naturally tend to speak with people similar to us, who share our habits, opinions,

	Movie 1	Movie 2	Movie 3	Movie 4
User 1	1	4	2	1
User 2	1	5	1	0
User 3	1	0	0	0

Table 8.1: A rating matrix.

way of life. By choosing to interact with a selected and small number of similar people we effectively **filter** the data. It is then up to us to integrate and weigh the information that we receive, to reach our final decisions. A similar process can be simulated through data mining and modeling methods. Starting from the raw data (potentially huge quantities, ranging from thousands to billions of items) one extracts information bits which are relevant for the specific final user, based on models of his explicit or implicit preferences, and on similarities with other people.

An interesting application is in the marketing sector: data collected about users and products, either bought or at least evaluated, can be used to estimate how a customer would evaluate a product he did not see before. The final purpose of predicting evaluations is to encourage the user to buy, for example by recommending a list of items corresponding to the highest predicted evaluations. Advertising is more effective if the presented products are filtered based on the user preferences. Other applications are in web mining, where the objective is to identify pages which the user may be interested in, while searching for information. The purpose is therefore to **imitate word of mouth** diffusion of information, for positive (fame) as well as for negative opinions (defamation).

Collaborative filtering and recommendation is a method of predicting the interests of a person by collecting taste information from many other collaborating people. The underlying assumption is that **those who agreed in the past tend to agree again in the future**. For example, a collaborative recommendation system for movies could make **personal predictions** about which movie a user should like, given some knowledge of the user's tastes and the information gathered from many other users.

Consider a user-item matrix R where the value of each entry r_{ui} is the rating of user u for item i as in Table 8.1. Every user u can vote for item i with a rating in the interval $[\text{min\_rating}, \text{max\_rating}]$. Let's get more concrete and assume that $\text{min\_rating} = 1$ and $\text{max\_rating} = 5$ and that value 0 is used to denote the unknown ratings. One wants to predict the unknown ratings in the matrix, either by some direct manner or by finding a more compact way to represent the data and by using the compact representation for prediction.

8.1 Combining ratings by similar users

A simple method to predict an unknown rating r_{ui} considers the ratings of other users on the same item i, and the *similarity* between user u and other users. A generic unknown rating r_{ui} is calculated by the following equation:

$$r_{ui} = \frac{\sum_{\text{known } r_{ki}} \text{similarity}(u,k) \cdot r_{ki}}{\sum_{\text{known } r_{ki}} \text{similarity}(u,k)}. \tag{8.1}$$

The vote is predicted with a weighted average of the other users' votes, with weights given by similarities, as shown in Fig. 8.1. The motivation is that similar users tend to give similar votes. If the denominator of the above equation is 0, then r_{ui} is calculated by default as the average value of all known ratings r_{ki}. If nobody rates item i, then r_{ui} is 0.

In a similar way, one could average the votes given by the same user on *different items*, with a weight proportional to the similarity between items, as explained in the bottom part of Fig. 8.1 (similar items tend to be judged in a similar manner).

The crucial issue is how to measure similarities. In this simplified context, the only knowledge about a user must be derived from his past evaluations of the different products. Therefore, in the above equation (8.1), the similarity

between two users (u, k) is obtained by measuring the similarity between two vectors $(\mathbf{v}_u, \mathbf{v}_k)$, the u^{th}, k^{th} rows of the rating matrix R.

The usual cosine similarity between two vectors can be used in a standard implementation, but different and problem-specific metrics can be tested, as explained in Sec.17.2.

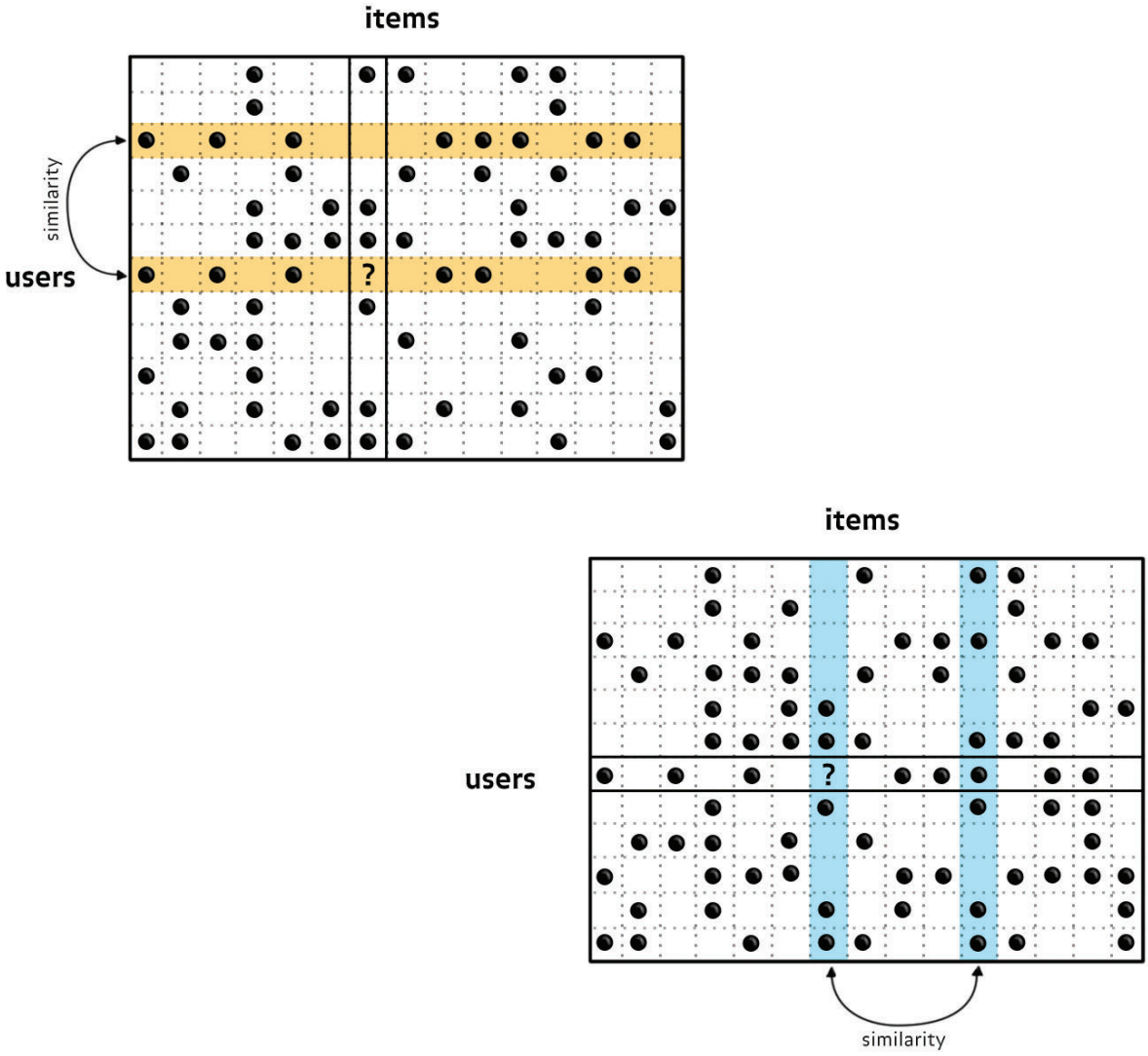


Figure 8.1: Collaborative recommendation. In order to predict unknown values, one computes a weighted average of known votes, weighted by the similarity either between users (rows) or between items (columns).

To be honest, people have very different ways of expressing opinions. A movie recommended as acceptable by an understating English reviewer may end up being a fantastic movie for a hyperbolizing Italian one. If you go by what a hypercritical reviewer says, you are going to end up seeing very few movies, and it can be useful to **discount individual evaluations** before using them to measure similarities and obtain predictions.

Let r_{uj} be the rating of user u for item j. Let I_u be the set of items that user u has rated. The mean rating for user

u is $\bar{r}_u = \frac{1}{|I_u|}\sum_{j\in I_u} r_{uj}$. Let the active user be denoted by subscript a. The goal is to predict the preference for an item i, or p_{ai}.

Because the votes may not be centered around zero, the system may have difficulties in reproducing the votes by scalar products. To help the system it can be useful to *center* the data by subtracting the average values. In detail, the prediction can be done by using the formula:

$$p_{ai} = \bar{r}_a + \frac{\sum_u w_{au}(r_{ui} - \bar{r}_u)}{\sum_u |w_{au}|}. \tag{8.2}$$

where the summation over u is over the set of users who have rated item i, and w_{au} is the weight between the active user a and user u. This weight can be defined as the Pearson correlation coefficient,

$$w_{au} = \frac{\sum_i (r_{ai} - \bar{r}_a)(r_{ui} - \bar{r}_u)}{\sqrt{\sum_i (r_{ai} - \bar{r}_a)^2}\sqrt{\sum_i (r_{ui} - \bar{r}_u)^2}}. \tag{8.3}$$

The summations over i are over the set $I_u \cap I_a$.

8.2 Models based on matrix factorization

The **sparsity** of the raw user-item evaluation matrix can be a problem. Each user evaluates only a very small subset of items and most entries are unknown. By **compressing and summarizing the user characteristics** into a much smaller vector, one hopes to reach better generalization results, and possibly a better understanding of the model, as explained by the Occam's razor principle.

A possible way to determine the interest or the vote of a user for an item is to associate a small **vector of characteristics with each user and with each item** and then deriving votes by observing the similarity between the user and item characteristic vectors. This operation can be done by hand, but it may be very time-consuming and it may not identify some characteristics which are crucial for the prediction. Let's see how the process can be automated.

Let us use vector $\boldsymbol{q}_i \in \Re^f$ to contain the characteristics (factors) of item i, and vector $\boldsymbol{p}_u \in \Re^f$ to denote the extent of interest that user u has in each item factor. One would like to obtain the rating of a user u for an item i by a simple scalar product of the corresponding vectors:

$$\hat{r} = \boldsymbol{q}_i^T \boldsymbol{p}_u. \tag{8.4}$$

For example, if the factors are built by hand, the aspect weights of movie *Terminator* can be exemplified as $(\text{action} = 5, \text{romance} = 1)$, and the interest of user *Patricia* in the movie aspects is $(\text{action} = 2, \text{romance} = 5)$, therefore the rating of user *Patricia* for movie *Terminator* is $5 \cdot 2 + 1 \cdot 5 = 15$.

Among automated ways to build effective factors to predict ratings, the traditional Singular Value Decomposition (SVD) can be used to find informative $\boldsymbol{q}_i$'s and $\boldsymbol{p}_u$'s. By using SVD, one decomposes a matrix R containing *all* ratings as $R = U\Sigma M^T$, where the rows of matrix U and M are the set of $\boldsymbol{p}_u$ and $\boldsymbol{q}_i$, scaled by the diagonal matrix Σ. By considering the size of the diagonal values in Σ one can then reduce the dimension of the vectors, and keep only the most relevant components. Unfortunately, in most cases one does *not* have the value of all cells in the rating matrix.

More flexible and robust learning algorithms based on optimization can be used to find effective approximations of the factor vectors $\boldsymbol{q}_i$ and $\boldsymbol{p}_u$. As usual, examples of expressed votes guide the learning process: to learn the factor vectors $\boldsymbol{q}_i$ and $\boldsymbol{p}_u$, one aims at minimizing the *regularized squared error (RSE)* on the set of *known* ratings:

$$\text{RSE} = \frac{1}{|K|}\sum_{(u,i)\in K} (r_{ui} - \boldsymbol{q}_i^T \boldsymbol{p}_u)^2 + \lambda(\|\boldsymbol{q}_i\|^2 + \|\boldsymbol{p}_u\|^2). \tag{8.5}$$

Here, K is the set of the (u, i) pairs for which r_{ui} is known (the training set). Let us know that the first term in the summation is the squared error between the model $\boldsymbol{q}_i^T \boldsymbol{p}_u$ and the known result r_{ui}. To facilitate *generalization* (prediction of novel ratings) it is useful to penalize the magnitudes of the factor vectors in a manner proportional to a

constant λ. This term is called a **regularizing term**. When example ratings abound, most of the contribution to RSE derives from errors in reconstructing the expressed votes. On the other hand, when ratings are scarce, the regularizing term becomes crucial and it acts to discourage very large vectors with potentially dramatic (and wrong) effects on the predictions.

The problem is now one of minimizing a continuous function of the free parameters $\boldsymbol{p}_u$ and $\boldsymbol{q}_i$, for which methods illustrated in Chapter 26 can be used, for example the traditional gradient descent. The gradient of RSE is calculated as follows:

$$\frac{\partial \text{RSE}}{\partial \boldsymbol{q}_i} = \frac{2}{|K|} \sum_{(u,i)\in K} ((r_{ui} - \boldsymbol{q}_i^T \boldsymbol{p}_u)(-\boldsymbol{p}_u) + \lambda \boldsymbol{q}_i);$$
$$\frac{\partial \text{RSE}}{\partial \boldsymbol{p}_u} = \frac{2}{|K|} \sum_{(u,i)\in K} ((r_{ui} - \boldsymbol{q}_i^T \boldsymbol{p}_u)(-\boldsymbol{q}_i) + \lambda \boldsymbol{p}_u).$$

The term $\frac{1}{|K|}$ is a constant and therefore not influencing the minimization. One can start with random initial values for $\boldsymbol{q}_i$ and $\boldsymbol{p}_u$ and then iterate: at each step a small change in the direction of the negative gradient is executed to reduce the RSE error.

8.2.1 A more refined model: adding biases

As illustrated for the case of the simpler methods in Sec. 8.1, the rating of user u for item i does not only depend on the interaction $\boldsymbol{q}_i^T \boldsymbol{p}_u$ between two vectors $\boldsymbol{p}_u$ and $\boldsymbol{q}_i$, but also on the bias of a user or item. In other words, some people usually give higher ratings, and some items often receive higher ratings than others. The bias involved in rating r_{ui} can be described as: $b_{ui} = \mu + b_i + b_u$, where μ is the overall average, b_i and b_u are the observed deviation of user u and item i from the average, respectively. For example, assume that one wants to estimate the rating of user *Joe* for movie *Titanic*. Assume that the average rating over all movies, μ, is 3.7 stars. Furthermore, *Titanic* is better than an average movie, so it tends to be rated 0.5 stars above the average. On the other hand, *Joe* is a critical user, who tends to rate 0.3 stars lower than the average. Thus, the baseline estimate for *Titanic* rating by *Joe* would be 3.9 stars $(3.7 + 0.5 - 0.3)$.

According to this refined model, the estimated rating $\hat{r}_{ui}$ of user u and item i is calculated as:

$$\hat{r}_{ui} = \mu + b_i + b_u + \boldsymbol{q}_i^T \boldsymbol{p}_u. \tag{8.6}$$

The observed rating is broken down into its four components: *global average*, *item bias*, *user bias*, and *user-item interaction*. This allows each component to explain only the part of a rating relevant to it. The learning algorithm learns by minimizing the following regularized squared error function considering also bias factors (RSEB):

$$\text{RSEB} = \frac{1}{|K|} \sum_{u,i\in K} (r_{ui} - \mu - b_i - b_u - \boldsymbol{q}_i^T \boldsymbol{p}_u)^2 + \lambda(\|\boldsymbol{p}_u\|^2 + \|\boldsymbol{q}_i\|^2 + b_i^2 + b_u^2). \tag{8.7}$$

The usual default procedure starts by deriving the gradient and using steepest descent. A factorization process using steepest descent in action can be seen in Fig. 8.2: as the number of gradient descent iterations increases, the error (root mean squared error) over the training set decreases as expected. The error over the test set (votes not used during training) first decreases, but then reaches a plateau and eventually gradually increases. This is an instance of **over-training**: the system is trying to accurately reproduce the training examples but generalization worsens. Think about a student learning "by heart," without digesting the learning material to extract the relevant relationships.

Let us note **the power and flexibility of optimization**: if additional terms are added to the model, the best parameters can then be immediately identified by calculating the new partial derivatives and plugging them into the minimization algorithm. If one knows how to optimize, one can focus on the problem definition, then *rapidly* try many alternative models and test the obtained generalization results on validation data.

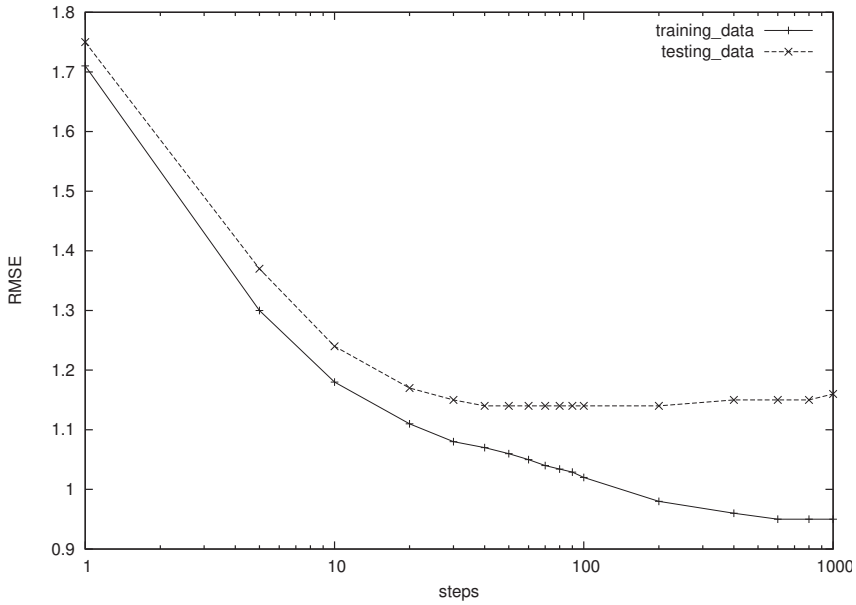


Figure 8.2: A factorization in action: training and testing performance as a function of gradient descent iterations.

Gist

When a potential customer visits your e-commerce web portal, he'll look at some items, put others in his cart, purchase them, write a comment and a score, leaving a trail, a scent that a well trained "nose" can follow.

All this information is there for you to improve your service: just like a good shopkeeper, who greets his patrons by name and proactively shows them what they'll like most, your website will lure customers with a personalized choice of items.

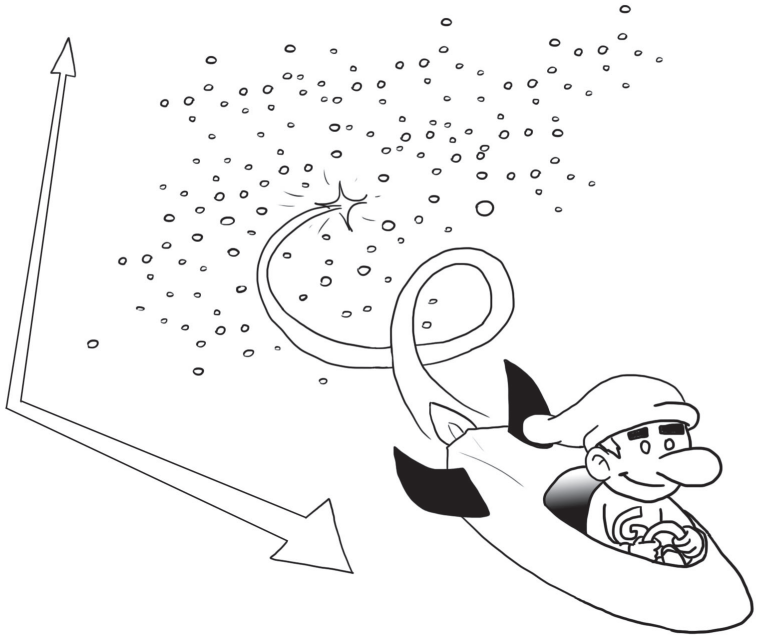
Collaborative filtering precisely does this: by memorizing and analyzing the behavior of customers, it simultaneously profiles visitors and items, grouping people by similar purchase habits, and **predicting what item a customer might like most**. This personalization is done without specific domain knowledge, by just mining the collective behaviour of customers. This is why a nerdy professor can end up consulting for a sophisticated fashion business.

Now think twice before clicking on a gossip title in your favorite online newspaper. If you do, more and more gossip-related news will appear on your personalized frontpage (and maybe also in different websites that you will visit, thanks to sharing of marketing data and *behavioral retargeting* strategies).

Chapter 9

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**).

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).

9.1 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. 9.1) 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).

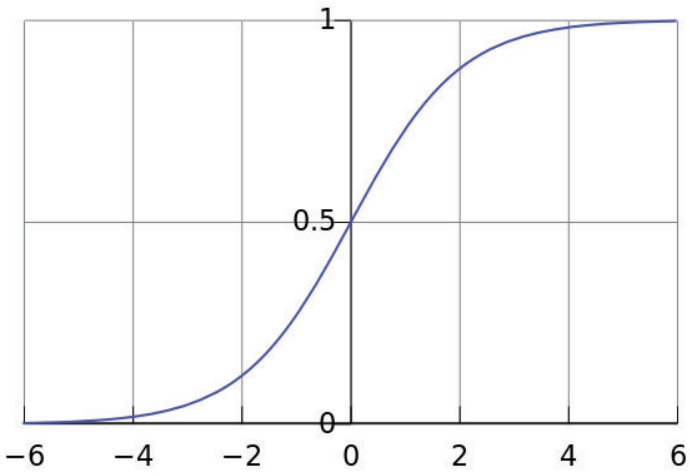


Figure 9.1: A logistic function transforms input values into an output value in the range 0-1, in a smooth manner. The output of the function can be interpreted as a probability.

A logistic curve is a common *sigmoid function*. 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.

A standard logistic function is defined as:

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

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 37:

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

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:

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

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.

9.2 Locally-Weighted Regression

In Section 4.1 we have seen how to determine the coefficients of a linear dependence, fixed coefficients for the entire input range. 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.

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

Through weighting, the overall global dependence can be quite complex. When the model is evaluated at different points, one still uses linear regression, but the 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.

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), 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 example. To avoid confusion let's 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 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 equa-

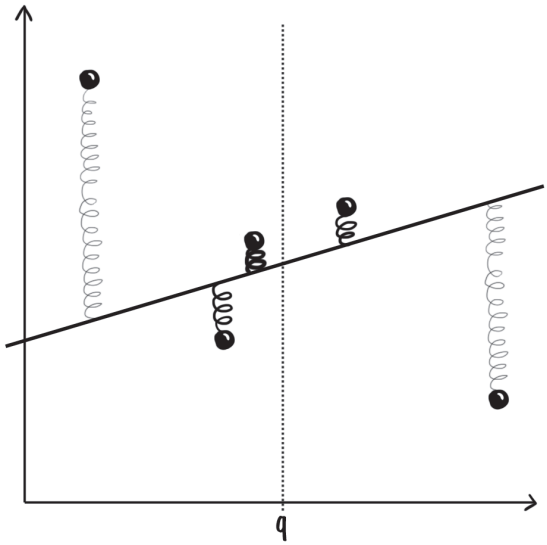


Figure 9.2: The spring analogy for the weighted least squares fit (compare with Fig. 4.6). Now springs have different elastic constants, thicker meaning harder, so that their contribution to the overall potential energy is weighted. In the above case, harder springs are for points closer to the query point $\boldsymbol{q}$.

tion (4.2), where weights are implicitly uniform):

$$\text{error}(\boldsymbol{w}; s_1, \ldots, s_n) = \sum_{i=1}^{\ell} s_i (\boldsymbol{w}^T \cdot \boldsymbol{x}_i - y_i)^2. \tag{9.1}$$

From the viewpoint of the spring analogy discussed in Section 4.1, the distribution of different weights to sample points corresponds to using springs with a different elastic constant (strength), as shown in Fig. 9.2. Minimization of equation (9.1) is obtained by requiring its gradient with respect to $\boldsymbol{w}$ to be equal to zero, and we obtain the following solution:

$$\boldsymbol{w}^* = (X^T S^2 X)^{-1} X^T S^2 \boldsymbol{y}; \tag{9.2}$$

where $S = \text{diag}(s_1, \ldots, s_d)$, while X and $\boldsymbol{y}$ are defined as in equation 4.5, page 42. Note that equation (9.2) reduces to equation (4.5) when all weights are equal.

A possible function used to assign significance values to the stored examples according to their distance from the query point 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. 9.3 (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 significance of each sample point and the subsequent linear fit are recomputed for each query point, providing the curve shown in Fig. 9.3 (bottom).

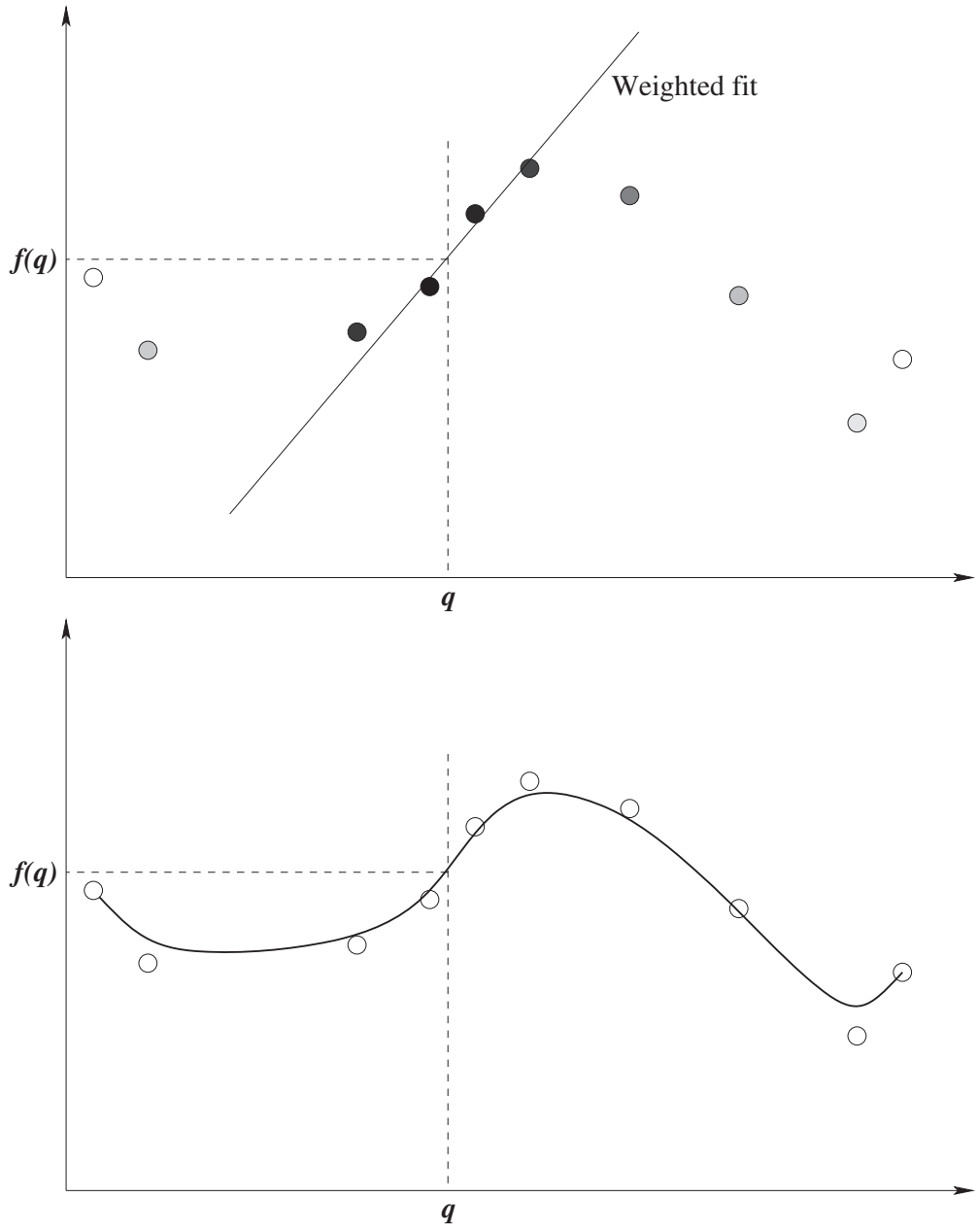


Figure 9.3: Top: evaluation of LWR model at query point $\boldsymbol{q}$, sample point significance is represented by the interior shade. Bottom: Evaluation over all points, each point requires a different linear fit.

9.2.1 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.

In **Bayesian Locally Weighted Regression**, denoted as B-LWR, one specifies *prior information* about what values

the coefficients should have. 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

$$\bar{\boldsymbol{w}} = (\Sigma^{-1} + X^T S^2 X)^{-1}(X^T S^2 \boldsymbol{y}). \tag{9.3}$$

Note that the removal of prior knowledge corresponds to having infinite variance on the prior assumptions, therefore Σ^{-1} becomes null and equation (9.3) reduces to equation (9.2). 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:

$$Var(\hat{y}(\boldsymbol{q})) = \boldsymbol{q}^T V_{\boldsymbol{w}} \boldsymbol{q} \sigma^2. \tag{9.4}$$

9.3 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 [358] 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.

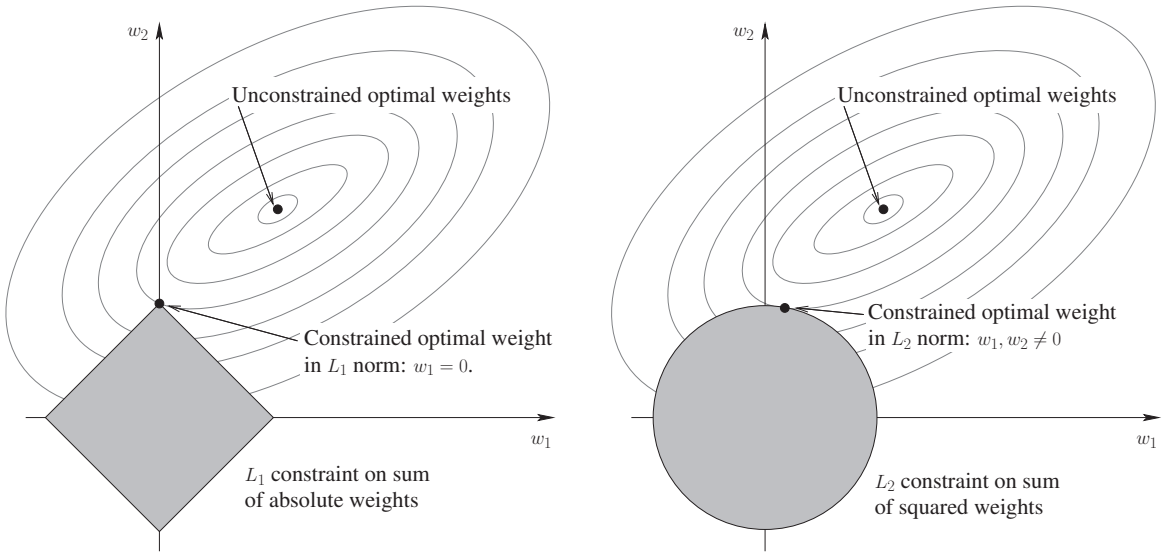


Figure 9.4: In LASSO, the best solution is where the contours of the quadratic error function touch the square, and this will sometimes occur at a corner, corresponding to some zero coefficients. On the contrary the quadratic constraint of ridge regression does not have corners for the contours to hit and hence zero values for the weights will rarely result.

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, explained in Section 26.5, this is equivalent to an unconstrained minimization of the *least squares* penalty with $\lambda\|\boldsymbol{w}\|_1$ added:

$$\text{LASSOerror}(\boldsymbol{w}; \lambda) = \sum_{i=1}^{\ell} (\boldsymbol{w}^T \cdot \boldsymbol{x}_i - y_i)^2 + \lambda \sum_{j=0}^{d} |w_j|. \tag{9.5}$$

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 process**.

Let's note that the term that penalizes large weights in equation (9.5) 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.

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. This is the case of the **LASSO** technique to shrink and select inputs. LASSO reduces the number of weights different from zero, and therefore the number of inputs which influence the output.

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

Chapter 10

Neural networks: multi-layer perceptrons

Quegli che pigliavano per altore altro che la natura, maestra de' maestri, s'affaticavano invano.
Those who took other inspiration than from nature, master of masters,
were laboring 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 through 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.

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.

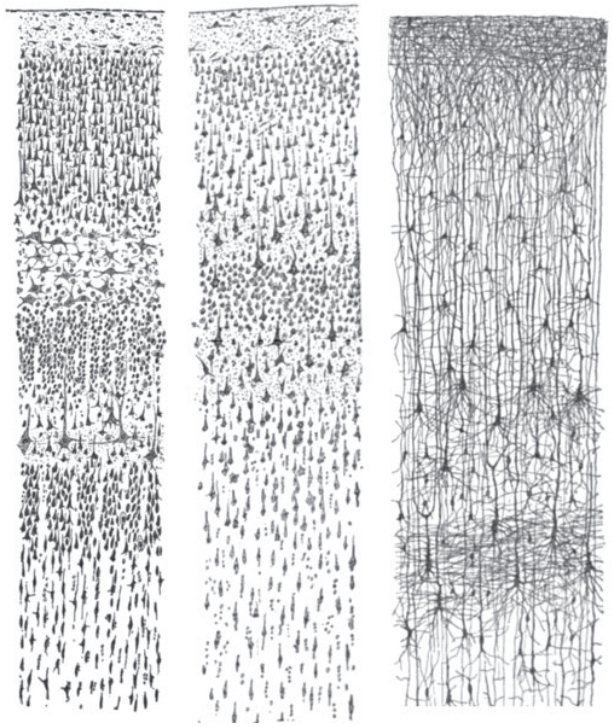


Figure 10.1: Three drawings of cortical lamination by Santiago Ramon y Cajal, each showing a vertical cross-section, with the surface of the cortex at the top. The different stains show the cell bodies of neurons and the dendrites and axons of a random subset of neurons.

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 (it is easy to check that the composition of linear functions remain linear). Organized layers are 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. 10.1).

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 difficult to simulate on a computer. The two styles of intelligent behavior are now widely recognized, leading also to popular books about "fast and slow thinking" [225].

In any case, "airplanes don't flap their wings." Even if real brains are a 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 **feed-forward multilayer perceptron neural networks**, without feedback loops.

10.1 Multilayer Perceptrons (MLP)

The logistic regression model of Section 9.1 in Chapter 9 was a simple way to add the "minimal amount of non-linearity" 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[1] (Fig. 10.2).

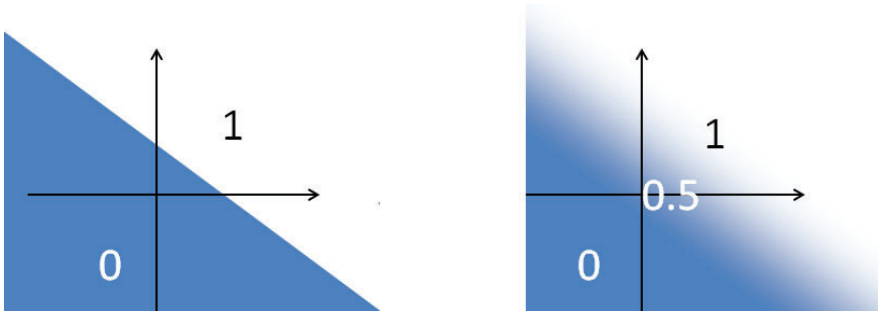


Figure 10.2: Effect of the logistic function. Linear model with threshold (left), smooth sigmoidal transition (right).

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 a plane or 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[2]. 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. An example of a parametric model is an oscillation $\sin(\omega x)$, in which the parameter ω has to be determined from the experimental data. The first linear transformation will provide a first "hidden layer" of outputs (hidden because internal and not directly visible as final outputs), additional transformations will produce the visible outputs from the hidden layers.

[1] 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 $LogLikelihood$ for logistic regression.

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

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 feed-forward information flow (no loops). The architecture is illustrated in Fig. 10.3.

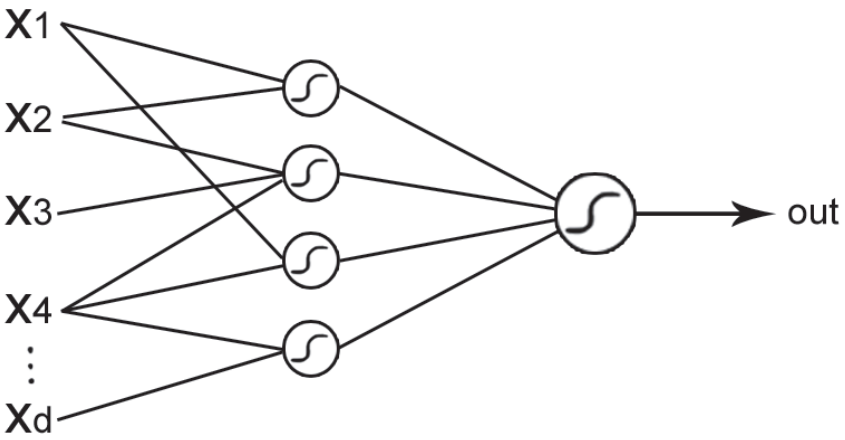


Figure 10.3: Multilayer perceptron: the nonlinearities introduced by the sigmoidal transfer functions at the intermediate (hidden) layers permit arbitrary continuous mappings to be created. A single hidden layer is present in this image.

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. 9.1):

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

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 separated by a hyperplane in input space, MLPs are **universal approximators** [190]: 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 result: 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 the existence of an MLP approximator, how can one find it rapidly 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.

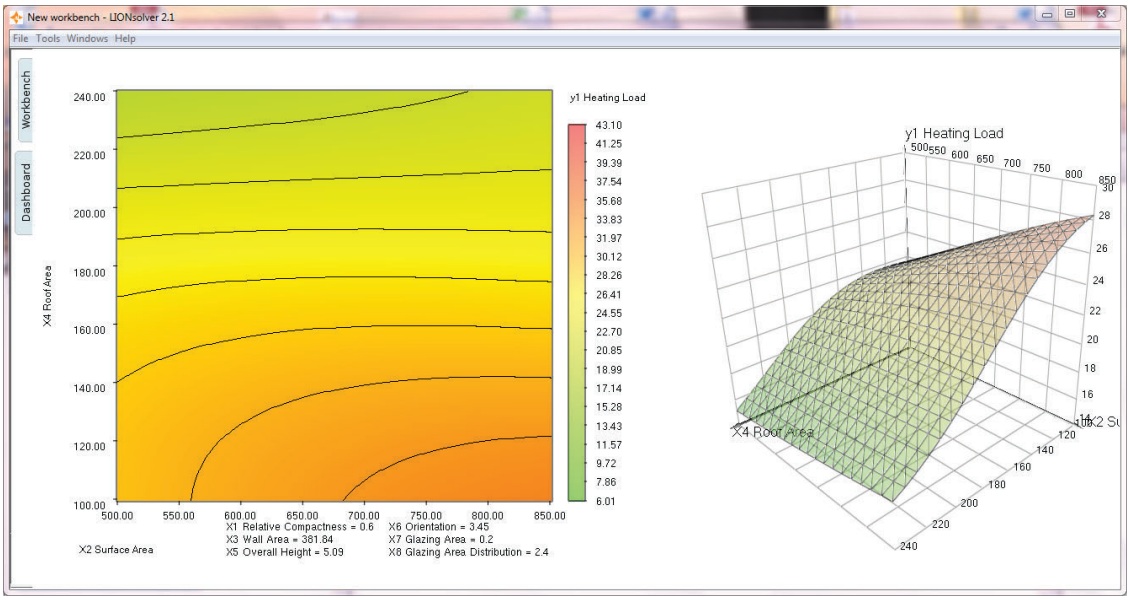


Figure 10.4: Analyzing a neural network output with LIONsolver Sweeper. The output value, the energy consumed to heat a house in winter, is shown as a function of input parameters. Color coded output (left), surface plot (right). Nonlinearities are visible.

As an example of an MLP input-output transformation, Fig. 10.4 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.

10.2 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 with respect to the weights and by taking a small step in the direction of the negative gradient. If the gradient is different from zero, there is a sufficiently small step in the direction of the negative gradient which will decrease the function value.

The technical issue is now one of *calculating* partial derivatives by using the **chain rule** in calculus for computing the derivative of the composition of two or more functions. 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:

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

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 networks 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 **backpropagation** of the error consists precisely in the above exercise: gradient calculation and small step in the direction of the negative gradient [383, 384, 311].

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

$$E(w) \;=\; \frac{1}{2}\sum_{p=1}^{P} E_p \;=\; \frac{1}{2}\sum_{p=1}^{P}(t_p \;-\; o_p(w))^2, \tag{10.1}$$

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. 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.

In the following sections we present two gradient-based techniques: standard *batch* backpropagation and a version with adaptive learning rate (*bold driver BP*, see [19]), and the on-line stochastic backpropagation of [311].

10.2.1 Batch and "Bold Driver" Backpropagation

The batch backpropagation update is a textbook form of gradient descent. After summing all derivatives related to each example and obtaining the gradient $g_k = \nabla E(w_k)$, the weights at the next iteration $k+1$ are updated as follows:

$$w_{k+1} = w_k \;-\; \epsilon\, g_k. \tag{10.2}$$

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:

$$w_{k+1} = w_k - \epsilon_k g_k, \tag{10.3}$$

$$\text{where} \quad \epsilon_k \;\text{ minimizes } \quad E(w_k - \epsilon g_k). \tag{10.4}$$

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 tends to zero).

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

$$\epsilon(t) = \begin{cases} \rho\,\epsilon(t-1), & \text{if } E(w(t)) < E(w(t-1)); \\ \sigma^l\,\epsilon(t-1), & \text{if } E(w(t)) > E(w(t-1)) \text{ using } \epsilon(t-1), \end{cases} \tag{10.5}$$

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 self-adaptive *bold driver* backprop is close and usually better than the one obtained by appropriately choosing a *fixed* learning rate. Nonetheless, being a simple form of *gradient descent*, the technique suffers from the common limitation of techniques that use the gradient as a search direction.

10.2.2 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 with respect to 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:

$$w_{k+1} = w_k - \epsilon \nabla E_p(w_k), \tag{10.6}$$

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[3]. 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.

Small batches BP is a third compromise option between the batch and online version. In this case, only a stochastic subset (a batch) of B patterns is run forward and back-propagated to accumulate the partial gradient. The weights are therefore modified every B forward passes. Of course, the extreme cases are online BP when B equals one, batch BP when B equals the total number of patterns.

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.

10.2.3 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 derivatives** 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

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

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 leading to a zigzagging path, see Fig. 26.11. 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 [20]. 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 [20], [19]. More details will be described in Chapter 26 dedicated to optimization of continuous functions.

Gist

Creating artificial intelligence based on the "real thing" is the topic of artificial neural networks research. **Multilayer perceptron neural networks** (MLPs) are a flexible (non-parametric) modeling architecture composed of layers of sigmoidal units interconnected in a feed-forward 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.

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

Chapter 11

Deep and convolutional networks

As a single footstep will not make a path on the earth,
so a single thought will not make a pathway in the mind.
To make a deep physical path, we walk again and again.
To make a deep mental path, we must think over and over
the kind of thoughts we wish to dominate our lives.
(Henry David Thoreau)

In this period machine learning is experiencing a soft revolution, in which ideas born a long time ago enjoy a second youth. Deep learning and convolutional networks are a promising direction, but we consider also alternative and crucially different directions like reservoir and extreme computing in the next chapters.

An anecdote tells that a team of graduate students led by Professor Geoffrey E. Hinton decided to enter a contest at the last minute with a deep learning system developed with no specific domain knowledge and won the top prize in 2012. The system had to predict which molecule was most likely to be an effective medicine. Today many advanced applications of computer vision and speech recognition are based on deep networks.

This chapter presents **deep neural networks** and **convolutional networks**. The long-term dream of deep networks is that of developing intelligent systems in a completely automated manner directly from abundant data (both labeled and unlabeled), without human experts to extract useful features by hand before the system is trained. The plan is to

have a hierarchy of levels in a feedforward network, self-organized so that the first layers extract basic building blocks (features) which are then combined to obtain more and more complex features in the subsequent layers (for example, features invariant under translation or rotation in image processing). **Convolutional networks** deal with **pre-wiring** an architecture which is appropriate for a domain (typically computer vision and speech processing), by inserting constraints, like locality of receptive fields, and by sharing weights. In our visual system and in image processing, basic local filtering operations like contrast enhancement or edge detection are applied over the entire image. It would be a waste of resources to ask ML to identify a different filter for every pixel and it would be masochistic to forget that the system is dealing with images, with a two-dimensional structure and local relationships.

11.1 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 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 in MLPs 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 neurological evidence in our brain, theoretical arguments demonstrate that some classes of input-output mappings are much easier to build when more hidden layers are considered [47].

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, closer to the sensory input, so that 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[1]: 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. In some cases internal representations in the first layers will not differ too much from what can be obtained by setting the corresponding 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 with respect to 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 Chapter 12.

More recently, a revival of **deep neural networks** (MLPs with many hidden layers) 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 [47].

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 (**unsupervised pre-training**);

[1] 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). A partial derivative is the limit of the ratio $\Delta f / \Delta w$ when the magnitude of the change Δw goes to zero.

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 when the number of unlabeled (unclassified) examples is much larger than the number of labeled ones, 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 humans to describe image content costs much more. 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.

11.1.1 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. 11.1). 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.

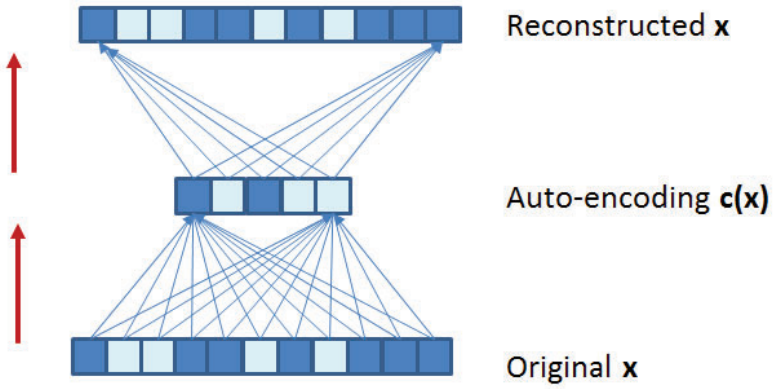


Figure 11.1: Auto-encoder.

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. In addition, tons of unlabeled examples can be added for a more robust training (with better generalization) of the auto-encoder.

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. 11.2), add an additional layer (initialized with small random weights),

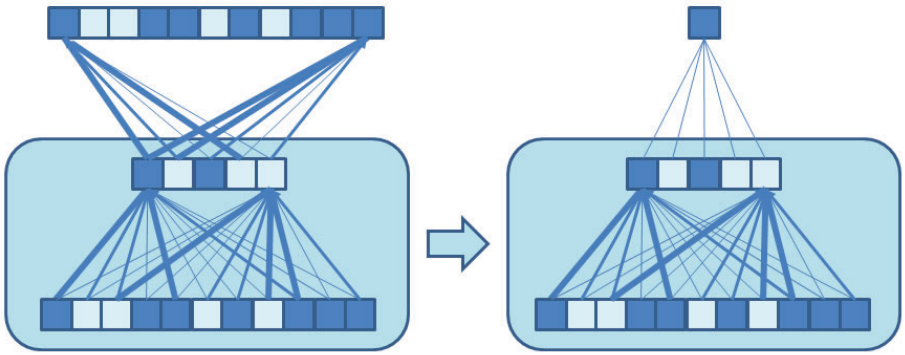


Figure 11.2: Using an auto-encoder trained from unlabeled data to initialize an MLP network.

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.

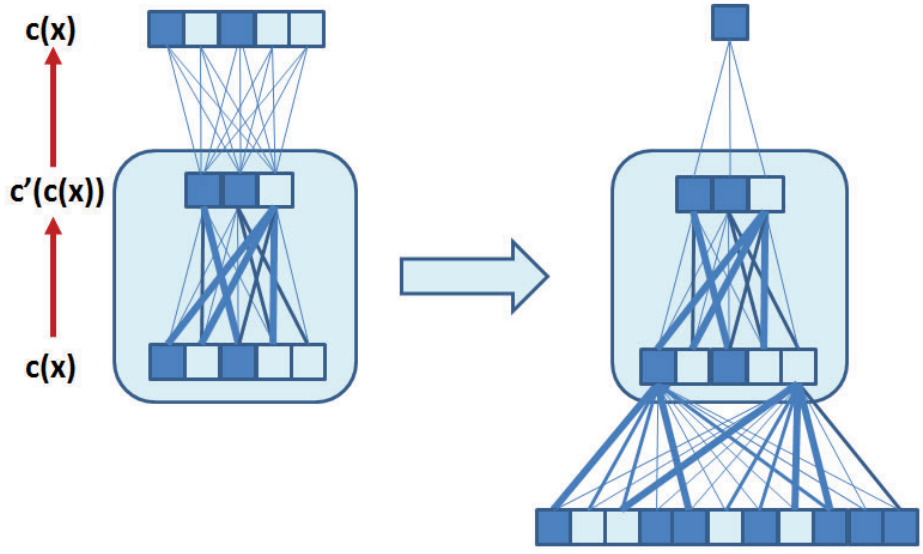


Figure 11.3: Recursive training of auto-encoders to build deeper networks.

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. 11.3).

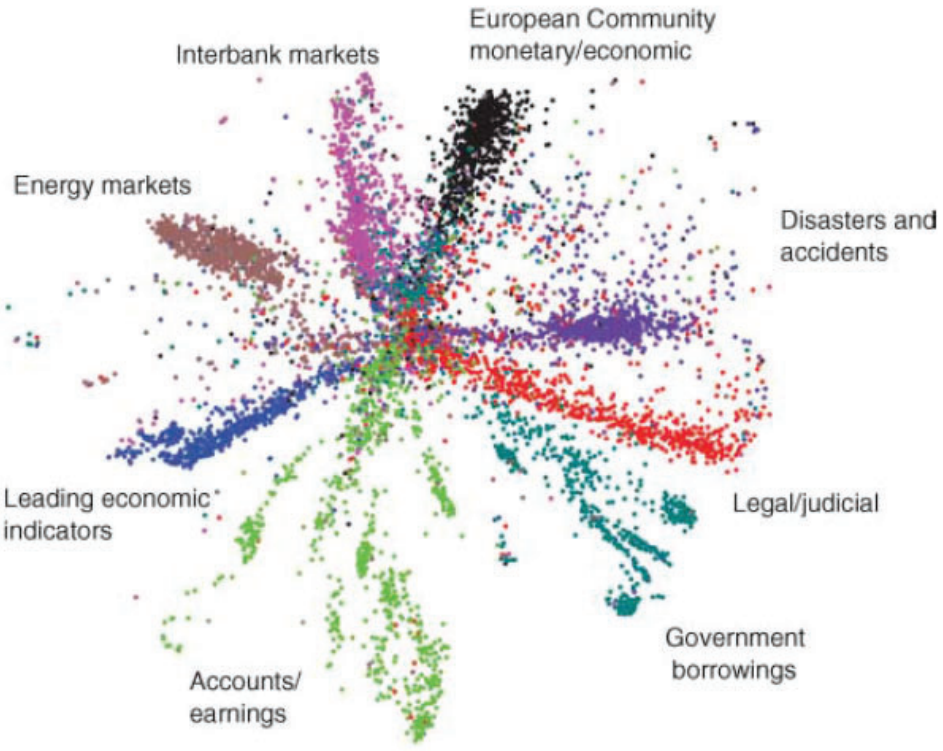


Figure 11.4: The codes produced by a 2000- 500-250-125-2 autoencoder on news stories by Reuters. Clusters corresponding to different topics, with different colors, are clearly visible (details in [181]).

In addition to being useful for pre-training neural networks, very **deep auto-encoders can be useful for visualization and clustering**. For example, news stories by Reuters[2] represented as a vector of document-specific probabilities of the 2000 commonest word stems, can be auto-encoded so that the bottleneck compressed layer contains only two units. The two-dimensional coordinates corresponding to story are visualized on a two-dimensional plane in Fig. 11.4. Different clusters approximately corresponding to the different topics are clearly visible in the two-dimensional space, the two (or more) coordinates in the bottleneck layer can therefore be used 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 [47].

[2]The Reuter Corpus Volume 2 is available at http:// trec.nist.gov/data/reuters/reuters.html.

11.1.2 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 [369] (**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 [180, 181], 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** [182]: 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.

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 coordinated 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 [182].

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 through **curriculum learning** [48]. 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 then proceeding to the more complex ones. For example, when learning music, first the single notes are learned, 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.

After all, magic systems for learning languages while you sleep and listen to recorded voices may not be a complete fraud :)

11.2 Local receptive fields and convolutional networks

Advanced animal brains are quick in learning how to process images and to recognize their content. An infant recognizes his mother already in the first days of life. This speed would be impossible without the help of a **pre-wired architecture** already available to process two-dimensional images. In particular, locality plays a big role: the first neurons which process nearby points in an image projected to the retina have local receptive fields and are mapped to nearby points in the visual cortex. Specialized low-level detectors like edge or movement detectors are available in biological brains.

Figure 11.5: *Bufo Bufo*, the common toad, was used in studies of toad form vision. Feature detectors in a frog retina are hard-wired and specialized to detect a fly at the distance that the frog could strike.

For example, when analyzing "on-off" ganglion cells in frogs – responding to both the transition from light to dark and from dark to light – with very restricted receptive fields (about the size of a fly at the distance that the frog could strike), it is difficult to avoid the conclusion that the 'on-off' units are matched to the stimulus and act as fly detectors [16] (Fig. 11.5).

When considering artificial neural networks, there is little doubt that recognizing images can be greatly simplified if some knowledge about image processing is pre-wired in the neural network. Only a masochist would forget the two-dimensional structure of the image and provide as input a one-dimensional array of pixel values at randomly scattered positions (if not convinced, apply a random permutation to the pixels of this page and try reading it).

In traditional models of pattern recognition, hand-designed features extract relevant information from the input and eliminates irrelevant variabilities. A trainable classifier like a MLP can then categorize the resulting feature vectors into classes. A potentially more interesting scheme is to eliminate the feature extractor, feeding the network with raw inputs, and to rely on back-propagation to turn the first few layers into an appropriate feature extractor. This brute-force approach faces difficulties related to the very large input dimension (causing many weights and possible over-training) and to the absence of any **built-in invariance** with respect to translations, rotations or local distortions of the inputs. For a frog, a fly remains a fly even if rotated and translated.

In principle, a sufficiently large fully-connected network could learn to produce outputs that are invariant with respect to such variations. However, learning such a task would probably result in multiple units with similar weight patterns positioned at various locations in the input. In **convolutional neural networks** (CNN) [249], some shift invariance is automatically obtained by forcing the replication of weight configurations across space. A kernel with local connectivity in the image plane is repeated at different positions in the image (the **weights are shared**). Local correlations are the reasons for the well-known advantages of extracting and combining local features before recognizing spatial or temporal objects. Convolutional networks force the extraction of local features by restricting the receptive fields of hidden units to be local.

The mathematical operation of applying the same local filter at different spatial positions is called **convolution**.

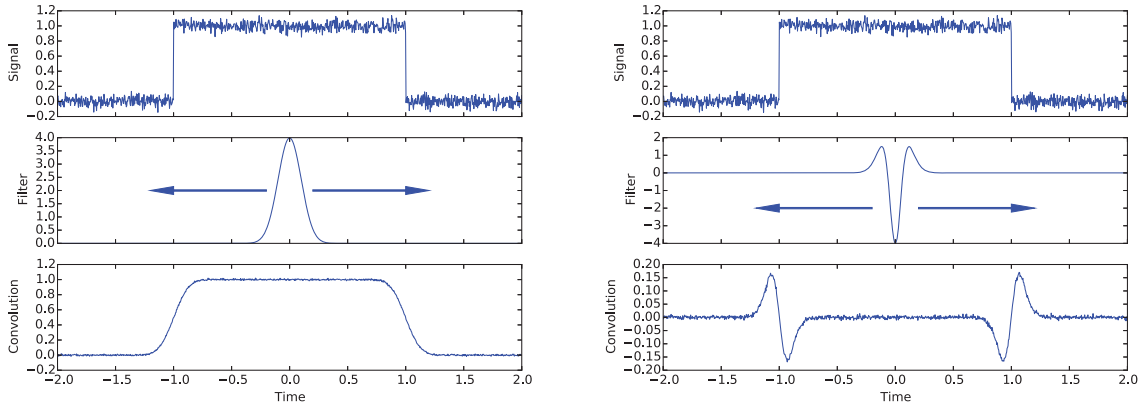


Figure 11.6: Two examples of convolution. Left: Gaussian smoothing; right: DoG border enhancement.

The use of localized kernels to extract local features is well exemplified by signal processing, where convolution is one of the most important mathematical tools. Fig. 11.6 shows two typical examples. On the left, a noisy signal is filtered by convolution with a Gaussian kernel; the outcome is a smoother version of the original signal. The procedure is called *blurring* (in computer vision), *low-pass filtering or de-noising* (signal processing), *smoothing out*. Mathematically, given a signal $s(t)$ and a filter $f(t)$, the convolution operation is given by

$$s * f(t) = \int_{-\infty}^{+\infty} s(x) f(t-x)\ \mathrm{d}x. \tag{11.1}$$

In other words, the filtering kernel f "sweeps" the signal by a weighted integral. If the Gaussian kernel has unit area, the result is a weighted average of the original signal. On the right of Fig. 11.6, a more interesting example uses the difference between two Gaussian kernels with different amplitudes. Two blurred versions of the signal with different smoothing windows are subtracted. The resulting kernel is called **Difference of Gaussians** (DoG), and its application highlights the points in which the signal has sudden, significant changes.

The convolution formula (11.1) can be easily extended in two dimensions and discretized for use in neural networks. To this aim, let x_{ij} be the pixels of an $m \times n$ image. The filtering kernel will be represented by an $(2r+1) \times (2r+1)$ matrix of weights w_{ij}, where the *radius* r is usually very small. Convolution produces a new $m \times n$ image whose pixels y_{ij} are

$$y_{ij} = \sum_{h=0}^{2r} \sum_{k=0}^{2r} w_{hk} x_{i+r+1-h, j+r+1-k}. \tag{11.2}$$

In the formula we assume that indices are zero-based. In order to obtain a resulting image of the same size as the original, we must also assume that the original image has an r-sized border in all directions; as an alternative, the resulting image will be smaller by r pixels in all directions.

Observe the "$t - x$" in equation (11.1): to retain many useful mathematical properties the convolution operator requires the two functions to be swept in opposite directions, as reproduced in (11.2). Most software packages can be configured to work either this way, or by having the kernel and the input swept in the same direction. In the latter case, the network is said to operate in *cross-correlation* mode, and the result is a proper inner product. The only actual difference between the two modes is the order in which the weights are stored, moving between the two representations only requires a $180°$ rotation of the kernels. In other terms, a convolution layer takes the inner product of the linear filter and the underlying receptive field.

Convolutional networks combine three ingredients to ensure some degree of shift and distortion invariance: **local receptive fields, shared weights** (or weight replication), and, sometimes, spatial or temporal subsampling (**pooling**).

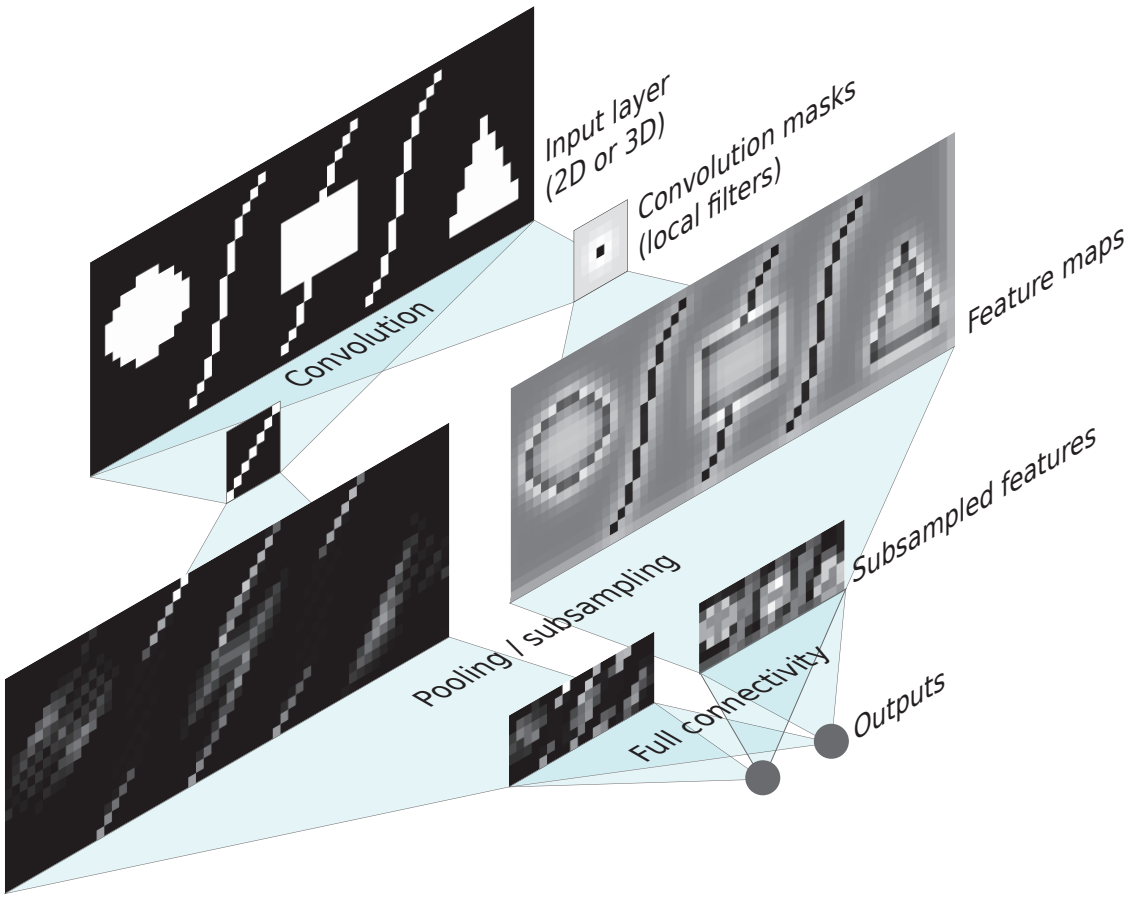


Figure 11.7: Basic convolutional network: the inputs (image pixels) are swept by a convolution operation against a small set of input weights acting as local feature extractors. The resulting feature maps are subsampled by a pooling operation, and the smaller set of neurons is passed through a traditional, completely connected output layer.

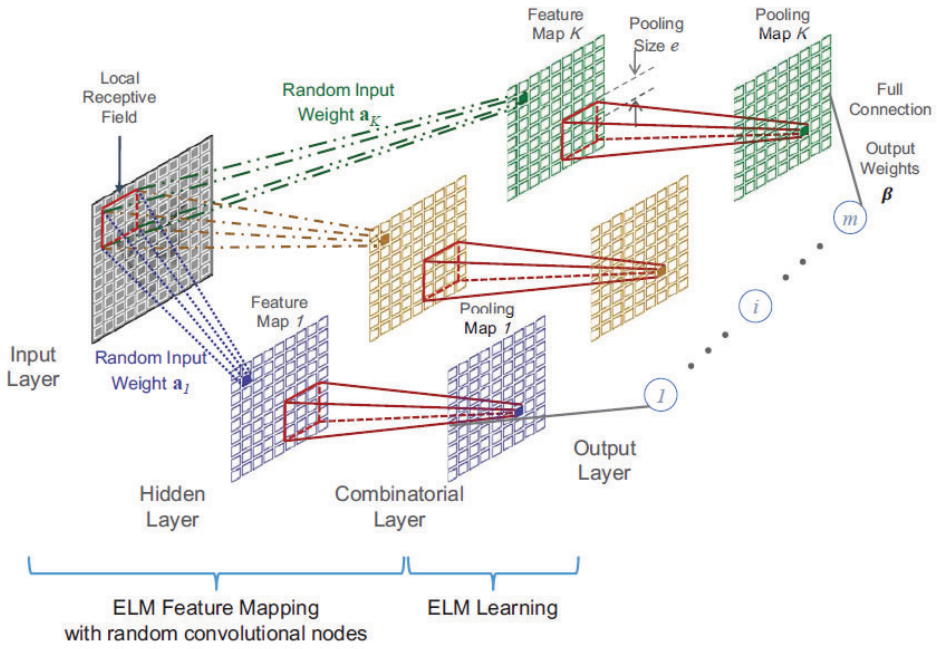


Figure 11.8: A structured architecture with local receptive fields (convolutional) and pooling layers (adapted from [165]).

As shown in Fig. 11.7, by applying the same local receptive fields throughout the image, neurons can extract elementary visual features such as oriented edges, end-points, corners, or similar patterns in speech spectrograms. These features are then combined by the higher layers.

The outputs of a set of neurons with shared weights, replicated at different points in the image, is called a **feature map**, obtained by convolution followed by a nonlinear activation function at every local portion of the input. In the upper portion of Fig. 11.7, an input image is "swept" by two filters, generating two feature maps whose neurons are more or less activated by the presence of the corresponding feature in the local receptive field. In the example, one filter has specialized to recognize slanted lines, the other has learned to enhance borders *à la* DoG.

Usually, each **convolutional layer** is followed by an additional **pooling layer** (see the bottom part of Fig. 11.7) which performs a local averaging and a sub-sampling, reducing the resolution of the feature map, and therefore reducing the sensitivity of the output to shifts and distortions. In its basic form, a pooling layer divides each feature layer into non-overlapping rectangles and applies a simple "summarizing" operation to each rectangle's pixels. Common operations are:

- the maximum value among all pixels in the rectangle (*max-pooling*);
- the average of pixel values in the rectangle (*average-pooling*);
- the square root of the sum of all squared pixel values (i.e., the *Frobenius norm* of the rectangle).

Deeper architectures can implement a cascade of convolutional and pooling layers, possibly enhancing the robustness of the output by means of other schemes such as the random dropout technique discussed in Section 11.1.2. Once the number of neurons is small enough, fully connected feed-forward layers complete the network.

Convolutional neural networks are still a hot research topic and a state-of-the-art tool for complex image and speech processing tasks, far too wide to be reviewed in this book. An example of the layered and structured architecture considered is shown in Fig. 11.8, for the recent proposal of randomly created weights for the units in the first

layers [165]. The authors of [253] propose a kind of "fractal" network-in-network architecture, building micro neural networks with more complex structures to abstract the data within the receptive field (going beyond the linear scalar product between filter coefficients and image pixels in traditional CNN). The micro neural networks (MLPs) are then replicated over the image.

Gist

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.

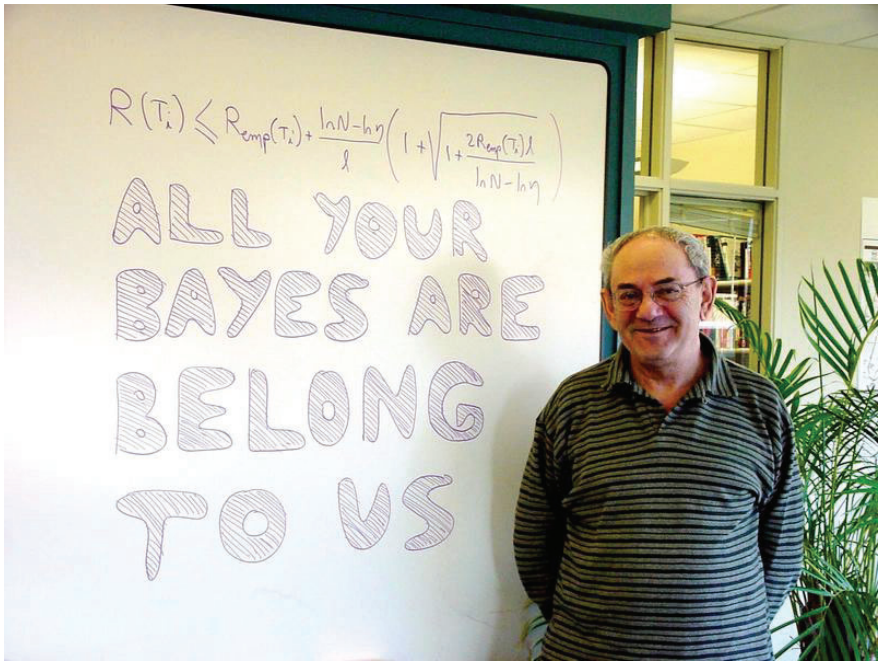
Convolutional neural networks are a good example of an idea inspired by biology that results in competitive engineering solutions and suggest **pre-wired architectures** with domain knowledge embedded in it.

Neural networks are like a glacial lake. You dive into the water, but you can't see how deep they are going to be.

Chapter 12

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)



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

[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`

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)**.

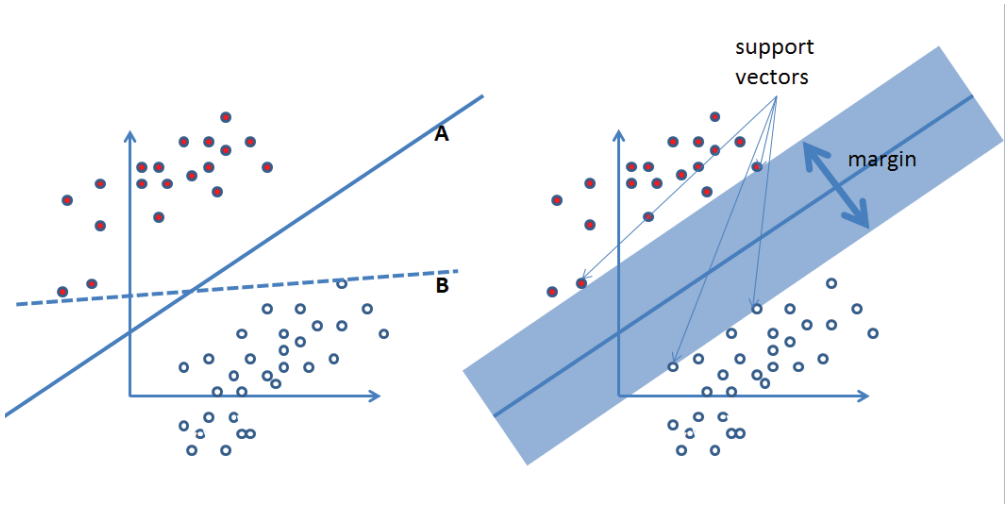


Figure 12.1: Explaining the basis of Support Vectors Machines. The *margin* of line A is larger than that of line B. A large margin increases the probability that new examples will fall in the right side of the separator line. *Support vectors* are the points touching the widest possible margin.

The term SVM sounds technical but the rationale is simple to grasp. Let's consider the two classes (dark and bright dots) in Fig. 12.1 (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. 12.1, right). We encountered a similar issue with linear models for classification and least-squares (Section 4.3). Lest-squares minimizes the average squared error, SVMs minimize the maximum distance but the objective of a robust and safe boundary between classes is shared.

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

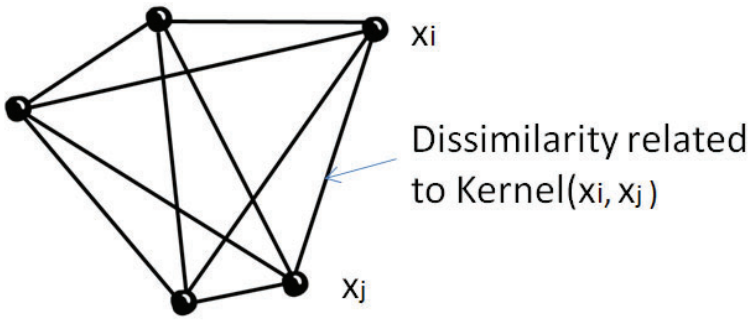


Figure 12.2: Initial information for SVM learning are similarity values between couples of input points $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, where K is known as *kernel function*. These values, under some conditions, can be interpreted as scalar products obtained after mapping the initial inputs 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").

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. 12.2):

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

A symmetric and positive semi-definite *Gram Matrix* containing the kernel values for couples of points fuses information about data and kernel[3]. 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.

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

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

12.1 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 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 $\boldsymbol{x}_i \rightarrow y_i$ 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:

$$R(\boldsymbol{w}) = \int \|y - f(\boldsymbol{x}, \boldsymbol{w})\| \, \mathrm{d}P(\boldsymbol{x}, y), \tag{12.1}$$

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

$$R_{\text{emp}}(\boldsymbol{w}) = \frac{1}{\ell} \sum_{i=1}^{\ell} \|y_i - f(\boldsymbol{x}_i, \boldsymbol{w})\|. \tag{12.2}$$

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 (12.1) with the function $f(\boldsymbol{x}, \hat{\boldsymbol{w}})$ which minimizes the empirical risk in (12.2).

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

$$\mathcal{H}_1 \subset \mathcal{H}_2 \subset \cdots \subset \mathcal{H}_n \subset \cdots \tag{12.3}$$

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:

$$\min_{\mathcal{H}_n} \left(R_{\text{emp}}(\boldsymbol{w}) + \sqrt{\frac{h(n)}{\ell}} \right). \tag{12.4}$$

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.

12.1.1 Linearly separable problems

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

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

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. 12.3): 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).

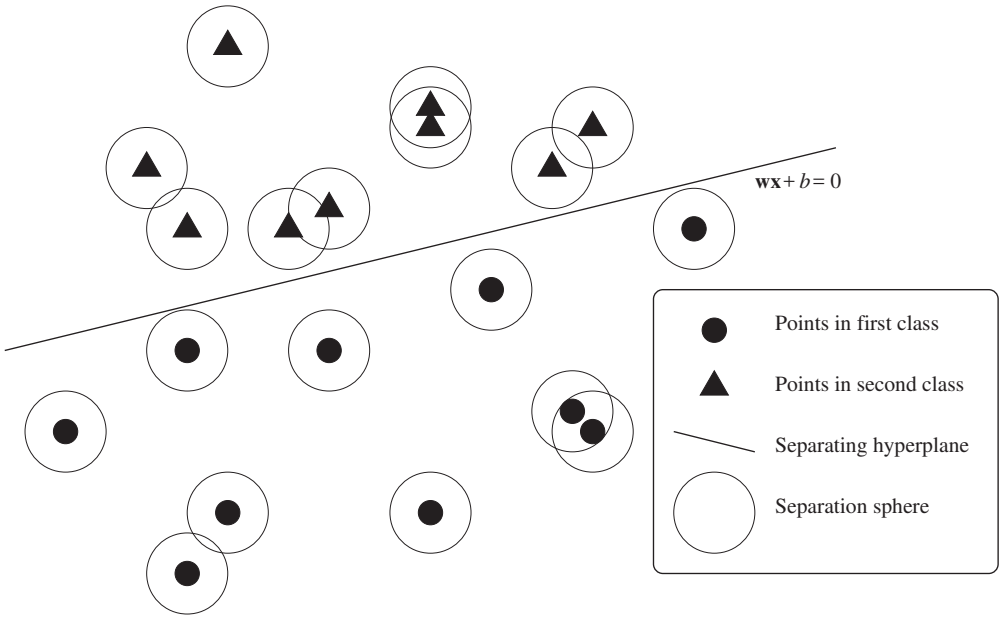


Figure 12.3: Hypothesis space constraint. The separating hyperplane must maximize the margin. Intuitively, no point has to be too close to the boundary so that some noise in the input data and future data generated by the same probability distribution will not ruin the classification.

The problem can be formulated as:

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

and 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, as explained in Section 26.5, is as follows:

$$\begin{array}{ll} \text{Maximize}_{\boldsymbol{\Lambda}} & \boldsymbol{\Lambda}\cdot\mathbf{1} - \frac{1}{2}\boldsymbol{\Lambda}\cdot D\cdot\boldsymbol{\Lambda} \\ \text{subject to} & \begin{cases} \boldsymbol{\Lambda}\cdot\boldsymbol{y} = 0 \\ \boldsymbol{\Lambda} \geq 0 \end{cases} ; \end{array} \tag{12.5}$$

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 (12.5) 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^*.

12.1.2 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 $\boldsymbol{\Xi}$) can be introduced, solving the following problem:

$$\begin{array}{ll} \text{Minimize}_{\boldsymbol{w},b,\boldsymbol{\Xi}} & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_{i=1}^{\ell}\xi_i\right)^k \\ \text{subject to} & \begin{cases} y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 - \xi_i & i = 1,\ldots,\ell \\ \xi_i \geq 0 & i = 1,\ldots,\ell \\ \|\boldsymbol{w}\|^2 \leq c_r; \end{cases} \end{array} \tag{12.6}$$

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.

12.1.3 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:

$$\begin{array}{ll} \text{Maximize}_{\boldsymbol{\Lambda}} & \boldsymbol{\Lambda}\cdot\mathbf{1} - \frac{1}{2}\boldsymbol{\Lambda}\cdot D\cdot\boldsymbol{\Lambda} \\ \text{subject to} & \begin{cases} \boldsymbol{\Lambda}\cdot\boldsymbol{y} = 0 \\ 0 \leq \boldsymbol{\Lambda} \leq C\mathbf{1}, \end{cases} \end{array} \tag{12.7}$$

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:

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

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 [284]. The work in [216] 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.

12.1.4 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:

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

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:

$$\begin{aligned} &\text{Minimize}_{\boldsymbol{w}, b, \boldsymbol{\Xi}} && \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_{i=1}^{\ell} \xi_i^* + \sum_{i=1}^{\ell} \xi_i\right) \\ &\text{subject to} && \begin{cases} y_i - \boldsymbol{w} \cdot \boldsymbol{x}_i - b \leq \varepsilon - \xi_i^* & i = 1, \ldots, \ell \\ \boldsymbol{w} \cdot \boldsymbol{x}_i + b - y_i \leq \varepsilon - \xi_i & i = 1, \ldots, \ell \\ \xi_i^* \geq 0 & i = 1, \ldots, \ell \\ \xi_i \geq 0 & i = 1, \ldots, \ell \\ \|\boldsymbol{w}\|^2 \leq c_r. \end{cases} \end{aligned} \tag{12.8}$$

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 [339].

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" used 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 through 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 13

Least-squares and robust kernel machines

Science may be described as the art of systematic over-simplification.
(Karl Popper)

The initial proposal of Support Vector Machines was based on the idea of mapping the data into a higher dimensional input space and of constructing an optimal separating hyperplane in this space, one maximizing the "safety" margin. As shown in Section 12.1.1, the requirement that points fall safely on the correct side of the separating hyperplane leads to **inequalities** like:

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 \qquad i = 1, \ldots, \ell,$$

then corrected with the addition of a constraint violation ξ_i:

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 - \xi_i \qquad i = 1, \ldots, \ell$$

The maximization of the margin leads to minimizing $\|\boldsymbol{w}\|$. The dual convex **quadratic program (QP)** obtained is solvable to optimality without the problem of local minima encountered in MLP and similar techniques.

Enthusiastic practitioners followed the SVM / convex QP wave, but two issues remained in the background. The first issue has to do with **identifying the proper kernel**: linear separability with good generalization requires a proper way of measuring similarities

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

between a point to be classified and the training examples. A crude analogy is that of a gentle teacher solving a big portion of the problem and leaving to the learner only a final and trivial part (QP optimization to identify the optimal hyperplane). Deep learning (Section 11.1) is a way of building intermediate features directly from the data in an automated manner.

A second issue has to do with computing times. QP is solvable, but CPU times increase very rapidly for nontrivial problems with many examples. QP is needed because of inequalities, so the temptation to abandon them in favor of simpler equalities is worth exploring. Encouraging equalities and penalizing mistakes in a quadratic manner leads to good old linear equations, faster to solve and easier to interpret. This chapter is dedicated to recent development in the area of **least-squares Support Vector Machines**. As we will see, quadratic penalties do not encourage **sparsity**, which can be recovered by other means. In addition, quadratic penalties can be fragile in the presence of **outliers** because big deviations are squared. Outliers can be caused by measurement errors, and one would like to avoid a few of them spoiling a modeling effort. A possible cure is by robust versions which limit the penalty for deviations which are suspiciously large.

The springs in the figure are related to the familiar physical interpretation of springs connecting data points and fitted models, with a quadratic potential energy.

13.1 Least-Squares Support Vector Machine Classifiers

After the support vector interpretation of ridge regression for function estimation in[313], kernel-based **Least-squares SVM classifiers** are proposed by Suykens and Vandewalle [355].

The least squares version of the SVM classifier is obtained by reformulating the minimization problem as:

$$\begin{aligned} &\text{Minimize}_{\boldsymbol{w},b,e} \quad J_2(\boldsymbol{w}, e) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{\gamma}{2}\sum_{i=1}^{\ell} e_{c,i}^2 \\ &\text{subject to} \qquad\quad y_i\left[w^T\boldsymbol{\varphi}(\boldsymbol{x}_i) + b\right] = 1 - e_{c,i}, \quad i = 1, \ldots, \ell. \end{aligned}$$

The hyperparameter γ can be tuned to determine the appropriate amount of regularization versus the sum of squared errors.

The least squares SVM (LS-SVM) classifier formulation above implicitly corresponds to a regression interpretation with binary targets $y_i = \pm 1$.

Using $y_i^2 = 1$, we have

$$\sum_{i=1}^{\ell} e_{c,i}^2 = \sum_{i=1}^{\ell} (y_i e_{c,i})^2 = \sum_{i=1}^{\ell} \left[y_i - (\boldsymbol{w}^T\boldsymbol{\varphi}(\boldsymbol{x}_i) + b)\right]^2 = \sum_{i=1}^{\ell} e_i^2,$$

with $e_i = y_i - (\boldsymbol{w}^T\boldsymbol{\varphi}(\boldsymbol{x}_i) + b)$. This error also makes sense for least squares data fitting, so that the same end results holds for the regression case. Note that the cost function J_2 consists of a sum-squared-errors (SSE) fitting error and a regularization term penalizing large weights, which is also a standard procedure for training MLPs and is related to ridge regression, encountered in Section 4.7.

The solution of the LS-SVM regressor will be obtained by constructing the Lagrangian function:

$$\begin{aligned} L_2(\boldsymbol{w}, b, e, \boldsymbol{\alpha}) &= J_2(\boldsymbol{w}, e) - \sum_{i=1}^{\ell} \alpha_i \Big\{ \left[\boldsymbol{w}^T\boldsymbol{\varphi}(\boldsymbol{x}_i) + b\right] + e_i - y_i \Big\} \\ &= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + \frac{\gamma}{2}\sum_{i=1}^{\ell} e_i^2 - \sum_{i=1}^{\ell} \alpha_i \Big\{ \left[\boldsymbol{w}^T\boldsymbol{\varphi}(\boldsymbol{x}_i) + b\right] + e_i - y_i \Big\}, \end{aligned}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_\ell)^T \in \mathbb{R}^\ell$ are the Lagrange multipliers, also called support values.

As usual, the requirement that the gradient is zero at the minimum leads to a linear system of equations instead of a quadratic programming problem (the derivatives of a quadratic form lead to a linear expression):

$$\begin{pmatrix} 0 & \mathbf{1}_\ell^T \\ \mathbf{1}_\ell & \Omega + \gamma^{-1} I_\ell \end{pmatrix} \begin{pmatrix} b \\ \boldsymbol{\alpha} \end{pmatrix} = \begin{pmatrix} 0 \\ \boldsymbol{y} \end{pmatrix}, \tag{13.1}$$

in which $\boldsymbol{y} = (y_1, \ldots, y_\ell)^T$, $\mathbf{1}_\ell = (1, \ldots, 1)^T$, I_ℓ is the $\ell \times \ell$ identity matrix, and $\Omega \in \mathbb{R}^{\ell \times \ell}$ is the kernel matrix defined by $\Omega_{ij} = \boldsymbol{\varphi}(\boldsymbol{x}_i)^T \boldsymbol{\varphi}(\boldsymbol{x}_j) = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

The "kernel trick" applies, one does not need to explicitly calculate the map $\boldsymbol{\varphi}$, only scalar products are needed. This trick is useful because the weight vector $\boldsymbol{w}$ can be infinite dimensional, and impossible to calculate in certain cases.

The classifier is found by solving the linear set of equations (13.1) instead of quadratic programming, and the resulting LS-SVM model for function estimation becomes

$$y(\boldsymbol{x}) = \sum_{k=1}^{\ell} \alpha_k K(\boldsymbol{x}, \boldsymbol{x}_k) + b.$$

A possibility is an RBF kernel characterized by the width parameter σ:

$$K(\boldsymbol{x}_1, \boldsymbol{x}_2) = e^{-\frac{\|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2}{\sigma^2}}.$$

In this case, the support values $\alpha_k = \gamma e_k$ are proportional to the errors at the data points, while in the case of standard SVM most values are equal to zero.

An example of SVM in action to learn the two-spiral benchmark problem is shown in Fig.13.1.

13.2 Robust weighted least square SVM

The issues of robustness and sparse approximation for LS-SVM are studied in [354]. The linear system (13.1) can be efficiently solved either directly or by iterative methods such as conjugate gradient (Section 26.3.2). However, LS-SVM solutions have some potential drawbacks. The first drawback is that **sparseness is lost**. Every data point is contributing to the model and the relative importance of a data point is given by its support value. The second well-known drawback is that the use of a SSE cost function without regularization can lead to estimates which are **less robust with respect to outliers** in the data or when the assumption of a Gaussian distribution for the error variables is not correct.

The problem with outliers is that big errors become huge after squaring them. The cure is to discount very large errors (maybe caused by mistakes or very rare events) through **weighted least square** to produce a more robust estimate.

This is done by first applying an unweighted LS-SVM and then associating weights to the error variables based upon the resulting error variables from the first stage. One ends up solving a sequence of weighted LS-SVMs starting from the unweighted version. The motivation is to **adapt the underlying cost function to the training data**, instead of imposing the cost function beforehand (in a way, this is a form of meta-learning).

In order to obtain a robust estimate based upon the previous LS-SVM solution, in a subsequent step, one can weight the error variables $e_k = \alpha_k / \gamma$ by weighting factors v_k This leads to the optimization problem:

$$\min_{\boldsymbol{w}^*, b^*, e^*} \frac{1}{2} \boldsymbol{w}^{*T} \boldsymbol{w}^* + \frac{1}{2} \gamma \sum_{k=1}^{\ell} v_k e_k^{*2}.$$

The unknown variables for this weighted LS-SVM problem are denoted by the "$*$" symbol.

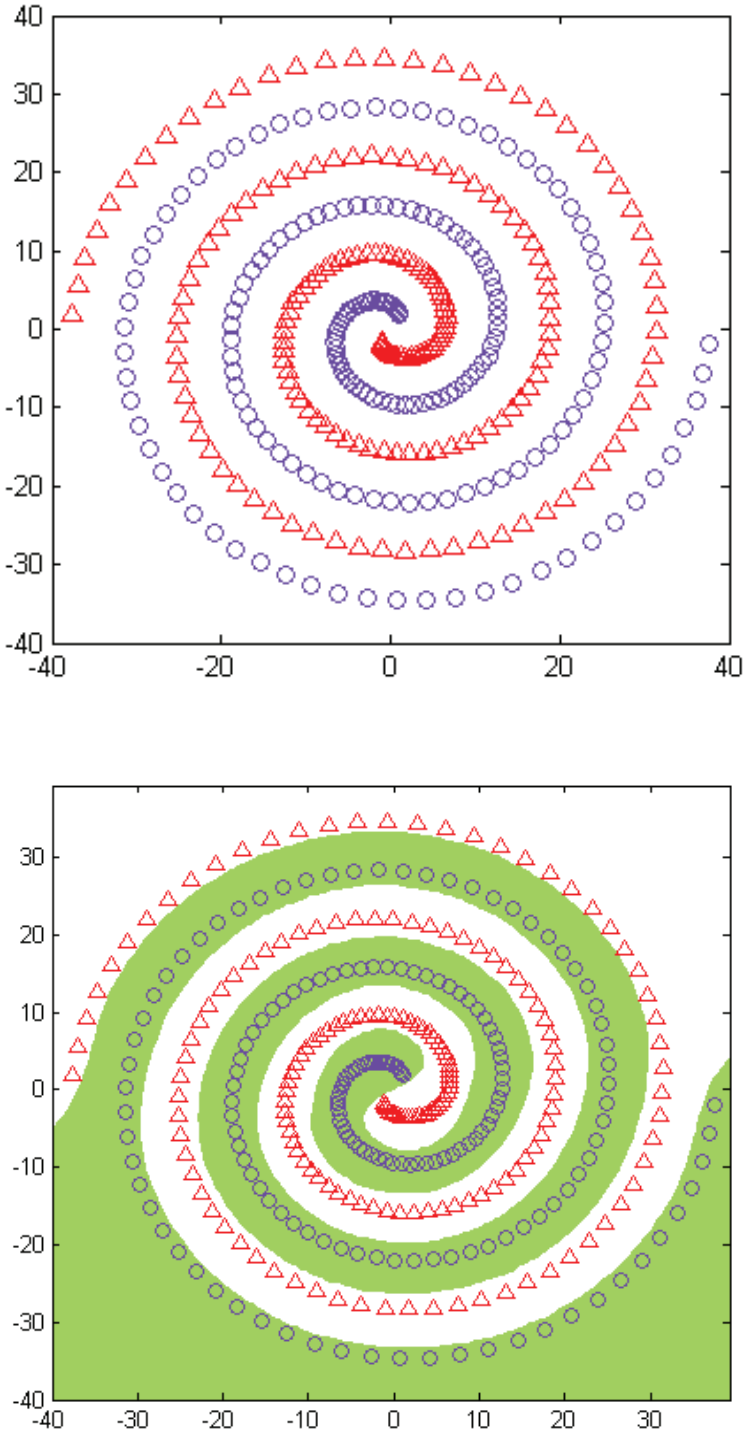


Figure 13.1: SVM tests on a two-spiral classification problem with the two classes indicated by circles and triangles. The figure shows the excellent generalization performance for an SVM.

1. **Algorithm** LS_SVM (ℓ training data items)
2. Find optimal (γ, σ) by k-fold cross-validation with linear system (13.1).
3. $e_k \leftarrow \alpha_k/\gamma$
4. Compute $\hat{s}$ from the e_k distribution using (13.2).
5. Determine the weights v_k based upon e_k, $\hat{s}$, as in (13.3).
6. Solve (13.1) for $\boldsymbol{\alpha}^*$ and b^*, giving the model
7. $$y(\boldsymbol{x}) = \sum_{k=1}^{\ell} \alpha_k^* K(\boldsymbol{x}, \boldsymbol{x}_k) + b^*.$$

Figure 13.2: The weighted LS-SVM algorithm

The choice of the weights v_k is based upon the error variables $e_k = \alpha_k/\gamma$ from the (unweighted) LS-SVM case. First one obtains $\hat{s}$, a robust estimate of the standard deviation of the LS-SVM error variables e_k:

$$\hat{s} = \frac{\text{IQR}}{2 \cdot 0.6745} \tag{13.2}$$

The interquartile range IQR is the difference between the 75th percentile and the 25th percentile. Of course, extreme cases like outliers do not count in this estimate, and this is why it is robust. In detail, robust estimates [309] can be obtained by taking:

$$v_k = \begin{cases} 1 & \text{if } |e_k/\hat{s}| < c_1 \\ \dfrac{c_2 - |e_k/\hat{s}|}{c_2 - c_1} & \text{if } c_1 \leq |e_k/\hat{s}| \leq c_2 \\ 10^{-4} & \text{otherwise.} \end{cases} \tag{13.3}$$

The constants c_1 and c_2 are typically chosen as $c_1 = 2.5$ and $c_2 = 3$. This is a reasonable choice taking into account the fact that, for a Gaussian distribution, very few residuals will be larger than $2.5\hat{s}$. Then errors which are suspiciously high with respect to a Gaussian distribution are discounted by giving them smaller and smaller weights.

If needed, the above procedure can be repeated iteratively, but in practice one single additional weighted LS-SVM step will often be sufficient. The final algorithm is shown in Fig. 13.2.

An important notion in robust estimation is the **breakdown point of an estimator**. It is the smallest fraction of contamination (with outliers) of a given data set that can cause an estimate which is arbitrarily far away from the estimated parameters obtained with the uncontaminated data set. Standard least-squares estimate in linear regression without regularization has a low breakdown point. Weighted LS-SVM greatly improves the breakdown point.

13.3 Recovering sparsity by pruning

While standard SVMs possess a sparseness property in the sense that many α_k values are equal to zero, this is not the case for LS-SVM's due to the fact that $\alpha_k = \gamma e_k$ from the conditions for optimality. Support values reveal the relative importance of the data points for contributing to the model.

While pruning methods for MLPs (such as *optimal brain damage* [250] and *optimal brain surgeon* [174]) involve Hessian matrices, the pruning in LS-SVMs proposed in [354] can be done based upon the solution vector itself. Sparseness can be imposed to the weighted LS-SVM solution by gradually pruning the sorted support value spectrum, i.e., by zeroing out the smaller α_i's: this way, less meaningful data points (as indicated by their support values) are removed and the LS-SVM is re-computed on the basis of the remaining points, while validating on the complete training data set.

By omitting a relative and small amount of the least meaningful data points (with α_k set to zero) and by re-estimating the LS-SVM, one obtains a sparse approximation. In order to guarantee a good generalization performance,

1. **Algorithm** `LS_SVM_pruning` (ℓ training data items)
2. $\ell' \leftarrow \ell$
3. **repeat until** performance degrades
4. Apply `LS_SVM` to the ℓ' training data
5. Sort training data according to decreasing magnitudes $|\alpha_k^*|$
6. Remove the last M data items in the sorted $|\alpha_k^*|$ spectrum
7. $\ell' \leftarrow \ell' - M$

Figure 13.3: The weighted LS-SVM pruning algorithm

in each of these pruning steps one can optimize (γ, σ), e.g., by defining an independent validation set, or 10-fold cross-validation. Figure 13.3 outlines the resulting algorithm.

Typically, doing a number of pruning steps without modification of (γ, σ) will be possible. When the generalization performance starts degrading (checked e.g. on a validation set or by means of cross-validation [365]) an update of (γ, σ) will be needed. The fact that (γ, σ) determination can be kept *localized* is a possible advantage of this method in comparison with other approaches which need to solve a QP problem for the several possible choices of the hyperparameters.

13.4 Algorithmic improvements: tuned QP, primal versions, no offset

Improvements to SVM are related both to detailed implementations of Quadratic Programming adapted to this scenario and to slight modifications of the problem definition, but with potentially big effects on CPU times and final performance [217].

The quadratic form in (12.7) involves a matrix that has a number of elements equal to the square of the number of training examples (matrix elements containing all possible kernel "similarities" between couples of examples). The first approach to splitting large SVM learning problems into a series of smaller optimization tasks is proposed in [62] as the "**chunking**" algorithm. It starts with a random subset of the data, solves this problem, and iteratively adds examples which violate the optimality conditions.

The work in [299] shows how much can be gained by passing from off-the-shelf QP software to a special-purpose implementation (and, incidentally, how much can be gained by studying the mathematical details of a problem). **Sequential Minimal Optimization** (SMO) proposes to break the large QP problem derived for SVM into a series of smallest possible QP problems. These small QP problems are solved analytically, avoiding a time-consuming numerical QP optimization as an inner loop. The amount of memory required for SMO is linear in the training set size, therefore SMO can handle very large training sets. Because large matrix computation is avoided, SMO scales somewhere between linear and quadratic in the training set size for various test problems, while a standard projected conjugate gradient (PCG) chunking algorithm scales somewhere between linear and cubic in the training set size. SMO's computation time is dominated by SVM evaluation, hence it is fastest for linear SVMs and sparse data sets.

An improved algorithm for training SVMs on large-scale problems (**SVMlight**) is proposed in [217]. The algorithm is based on a decomposition strategy and addresses the problem of selecting the variables for the working set in an effective and efficient way. Furthermore, a technique for "shrinking" the problem during the optimization process is introduced: during the optimization process it often becomes clear fairly early that certain examples are unlikely to end up as support vectors (SV), and by eliminating them the problem gets smaller. This is found particularly effective for large learning tasks where the fraction of SVs is small compared to the sample size. SVMlight's memory requirement is linear in the number of training examples and in the number of SVs.

Solving SVM in the primal space is proposed in [80]. Most literature on SVMs concentrates on the dual optimization problem. The authors of [80] argue that the primal problem can also be solved efficiently and that there is no reason for ignoring this possibility. On the contrary, from the primal point of view new families of algorithms for

large scale SVM training can be investigated. The usual main reasons mentioned for solving this problem in the dual are:

1. The duality theory provides a convenient way to deal with the constraints.
2. The dual optimization problem can be written in terms of dot products, thereby making it possible to use kernel functions.

Newton optimization in the primal yields exactly the same computational complexity as optimizing the dual. When it comes to approximate solution, primal optimization can be superior because it is more focused on minimizing what we are interested in: the primal objective function. Primal optimization might have advantages for large scale optimization. Indeed, when the number of training points is large, the number of support vectors is also typically large and it becomes intractable to compute the exact solution. One has to resort to approximations, but introducing approximations in the dual may not be wise. There is indeed **no guarantee that an approximate dual solution yields a good approximate primal solution**.

In a different direction, [345] develops and analyzes a training algorithm for **support vector machine classifiers without offset**. Historically, SVMs were motivated by a geometrical illustration of their linear decision surface in the feature space, like in Fig. 12.3. This illustration justified the use of an offset b that moves the decision surface from the origin. However, this geometrical interpretation has serious flaws. Even if visualizations can be powerful, one should **never base algorithmic choices on simple illustrations in low-dimensional spaces**.

It turns our that SVM optimization with offset imposes more restrictions on solvers than the optimization problem without offset does. The offset leads to an additional equality constraint in the dual optimization problem, which in turn makes it necessary to update at least two dual variables at each iteration of commonly used solvers such as sequential minimal optimization (SMO) [299].

The authors of [345] develop algorithms for SVMs without offset. These algorithms not only achieve a classification accuracy that is comparable to the one for SVMs with offset, but also run significantly faster.

Gist

Least-squares Support Vector Machines require equalities instead of inequalities for classifying patterns (like the usual trick of mapping positive cases to $+1$, negative cases to -1). In this manner the quadratic penalty on the errors leads to **linear equations** after taking partial derivatives and demanding a zero gradient.

Quadratic penalties increase big deviations so that a few **outliers** can distort the model. **Robust statistics** produces methods that are not unduly affected by outliers, by limiting their effect on the goodness function to be minimized. Suspiciously large errors are discounted by giving them small weights and obtaining **robust weighted least square SVM**.

The sparseness which is lost in the quadratic formulation can be recovered through **pruning** techniques, so that less meaningful data points are removed and the LS-SVM is re-computed on the basis of the remaining points.

The antediluvian least-squares method minimizing the sum of squared residuals remains an effective pillar to support also recent methods like SVM. Never underestimate good old techniques and linear algebra when comparing with the latest proposals.

Chapter 14

Structured Machine Learning, Text and Web Mining

Wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified.
(Vannevar Bush, 1945)

Up to now we have considered data with a flat structure, numbers conveniently collected in vectors. But there are interesting situations in which data (observations) come with a very structured form, imagine the graph of a **network** describing relationships between people or between genes (genetic regulatory networks), a causal network linking a hidden syndrome to symptoms, or imagine analyzing text or understanding language.

Structured machine learning refers to learning structured hypotheses from data with rich internal structure usually in the form of one or more relations[118]. The data might include structured inputs as well as outputs, parts of

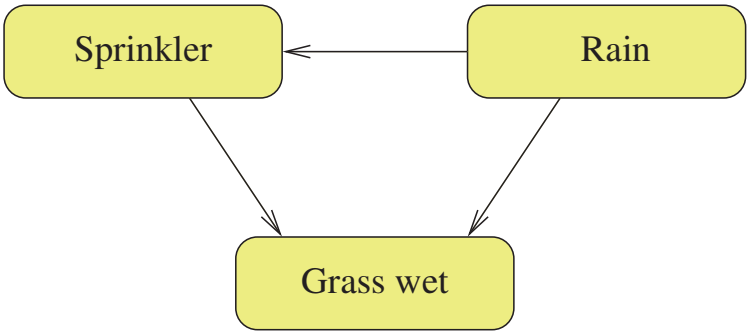


Figure 14.1: A simple Bayesian network: nodes and relationships.

which may be noisy, or missing. **Probabilistic graphical models** is an umbrella term to cover Bayesian networks, Markov networks and variations thereof.

Applications include a variety of tasks such as learning to parse and translate sentences, studying social networks, predicting the pharmacological properties of molecules, and interpreting visual scenes. Predictions can be about node properties (e.g., "Is a user in a social network going to vote democrat or republican?"), about link properties or link existence ("Will customer A buy product B?", "Are two proteins interacting?"), or about entire networks ("Is this a network of terrorists?"). An empirical fact which can be used for collective classifications is **homophily** (i.e., "love of the same"), the tendency of individuals to associate and bond with similar others. Linked entities are likely to share attribute values ("If A is a subscriber of a cellular company and B is connected to A, B is more likely to be a subscriber of the same company"), and entities sharing common neighbors are more likely to be linked ("friend of friends are friends").

The topic is rapidly getting very technical and we are forced to mention only some relevant models. In this introductory chapter we consider briefly Bayesian belief networks (Sec. 14.1), Markov networks (Sec. 14.2) and inductive logic programming (Sec. 14.3), and dedicate more space to text and web mining (from Sec. 14.4), also because of its relevance and concrete aspect which facilitates understanding.

14.1 Bayesian networks

A **Bayesian network, or belief network** is a **graphical representation of uncertain knowledge**, easy to build and to interpret, yet with a formal probabilistic semantics making it suitable for statistical interpretation [176, 211].

Up to now, the objective has been that of maximizing some performance index (correct recognition, squared error, etc.). But there is another positive objective for intelligent systems, one that becomes a necessity for certain applications: **human understanding and high-level explanation**. In healthcare, purely automated systems identifying a diagnosis from symptoms and exams are not yet acceptable. A physician has to check and understand (and be responsible in case of lawsuits). An example of systems based on rules are decision trees (Chapter 6), but their structure based on deterministic hierarchical chain of "if ... then" rules limit their applicability. A long time ago, "**expert systems**" with high-level (symbolic) rules defined solely by experts were popular but soon encountered problems related to fragility, difficulty in maintenance for dynamic systems, exploding times for running the inference engine.

A Bayesian network is a useful compromise: **domain experts can define an initial network structure**, but it is amenable to refinement through **learning, by fine-tuning probabilities** or modifying some parts of the network structure.

In details, a Bayesian network is a probabilistic graphical model that represents a set of random variables and their conditional dependencies via **a directed acyclic graph (DAG)**. For example, it could represent the probabilistic relationships between diseases and symptoms. Given symptoms, the network can be used to compute the probabilities

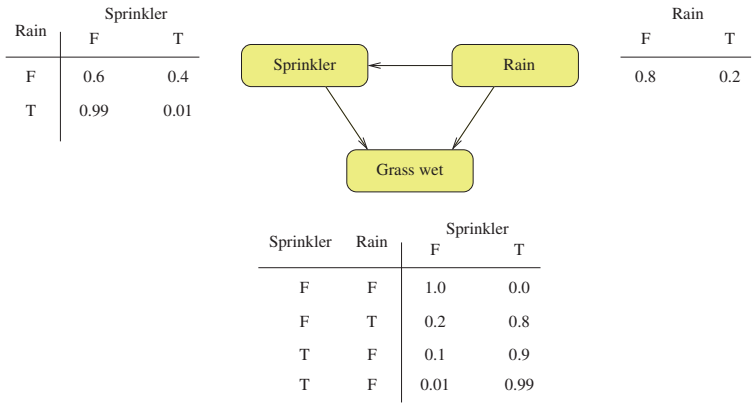


Figure 14.2: A simple Bayesian network: nodes and probability tables.

of various diseases. Nodes represent random variables: they can be observable quantities, latent variables, unknown parameters or hypotheses. Edges represent **conditional dependencies**. Each node is associated with a probability function that takes, as input, a particular set of values for the node's parent variables, and gives (as output) the probability of the variable represented by the node.

A toy example is in Fig. 14.1 Two events can cause grass to be wet: either the sprinkler is on or it's raining. Also, the rain has a direct effect on the use of the sprinkler (when it rains, the sprinkler is usually not turned on). All three variables have two possible values, T (for true) and F (for false). Possible probability tables for this examples are in Fig. 14.2.

The joint probability function is:

$$\Pr(G, S, R) = \Pr(G|S, R)\Pr(S|R)\Pr(R)$$

where the names of the variables have been abbreviated to G = "Grass wet" (yes/no), S = "Sprinkler turned on" (yes/no), and R = "Raining" (yes/no). The model can answer questions like "What is the probability that it is raining, given the grass is wet?"

Not surprisingly, Bayesian networks are related to the **Bayesian philosophy**. The probability of an event represents the **degree of belief** that the event will occur in an experiment. This is also called the subjective interpretation, different from the *frequentist* interpretation of probability (frequency in a series of repeated experiments). According to Bayes, the probability of event e depends on the state of knowledge ξ of the person providing the probability $p(e|\xi)$. One of the main motors for probabilistic reasoning is **Bayes' theorem**:

$$\Pr(X|Y, \xi) = \frac{\Pr(Y|X, \xi)}{\Pr(Y|\xi)}\Pr(X|\xi), \text{ for } \Pr(Y|\xi) > 0$$

The standard names for the probability of X before we know Y ($P(X|\xi)$) is the *prior*, the probability $P(X|Y, \xi)$ after we know Y is called the *posterior*. Because probabilities sum up to one, one can forget about the denominator and just remember:

$$P(X|Y, \xi) \propto P(Y|X, \xi)P(X|\xi)$$

To remember: "Posterior equals prior times likelihood". Important ingredients in the machinery of Bayes networks are the *chain rule*:

$$p(x_1, \ldots, x_n|\xi) = \prod_{i=1}^{n} p(x_i|x_1, \ldots, x_{i-1}, \xi) \tag{14.1}$$

the *generalized sum rule* (variable Y is *marginalized out*) :

$$\sum_{Y} P(X, Y|\xi) = P(X|\xi) \tag{14.2}$$

and the *expansion rule*:

$$P(X|\xi) = \sum_{Y} P(X|Y, \xi)P(Y|\xi) \tag{14.3}$$

Given a set of events $x_1, \ldots, x_n$, the joint probability distribution function $p(x_1, \ldots, x_n)$ contains the entire probabilistic knowledge. Unfortunately, dealing directly with a generic p.d.f. is out of question for computational reasons (the number of values to store is at least 2^n, in the lucky case of events with just two possibilities) and for understanding (we are very bad at reasoning with very large-dimensional tables).

Luckily, useful real-world Bayesian networks have a **very sparse structure**: the relevant tables are low-dimensional and the relationships are only among a limited set of nodes. In detail, in the factors of the chain rule of equation (14.1), for every x_i there will be a limited subset Π_i such that x_i and $\{x_1, \ldots, x_n\}$ are conditionally independent given Π_i, i.e.,

$$p(x_i|x_1, \ldots, x_{i-1}, \xi) = p(x_i|\Pi_i, \xi)$$

In the graph representing the Bayesian network, the parents of node x_i correspond to the set Π_i. In addition to the connectivity structure defining the parent - child relationship, the tables $p(x_i|\Pi_i, \xi)$ contain the relevant probability distributions.

Let's note that Bayesian networks are not uniquely defined: the **structure depends on variable order**, and a careless ordering may fail to reveal many conditional dependencies.

On the other hand, with a carefully selected ordering (guided by the knowledge and intuition of the domain expert), the joint probability distribution can be decomposed into manageable pieces. In this manner **probabilistic inference**, i.e., computing probabilities of interest from a joint probability distribution, becomes doable in acceptable times (no sums over 2^n or more possibilities!). In spite of the "*divide et impera*" approach, efficient probabilistic inference is not trivial and different algorithms have been proposed [176]. Exact inference in arbitrary Bayesian networks, and even approximate inference is NP-hard [106], but these negative results should not discourage, there is help for reasonably small networks, particular topologies, particular queries.

Unfortunately, rarely is the domain so clear and the probabilistic knowledge so refined to design a fully functional Bayesian network from scratch. In other words, this is another area in which huge **opportunities for learning from data** exist. In particular, one can **learn probabilities**. One can start with a prior distribution depending on available knowledge ξ and update probabilities with Bayes rule to obtain posterior distributions after running experiments. In other cases, the **structure of the network** is also uncertain and one may want to refine the structure when more and more data becomes available. Prior probabilities can be defined over network structures, then posterior probability distributions over structures can be derived.

Unfortunately, the number of possible structures to sum over explodes when the network is more than a small toy problem and radical approximations are needed. Luckily, even crude approximations that consider only **a single "good" network structure** can be sufficient. Identifying a good network compatible with the measured data can be done through **local search (LS)** mechanisms (LS is explained in Chapter 24). LS can be particularly effective when the score can be *updated* rapidly for small changes, like the addition or deletion of an edge, and not recomputed from scratch. In LS one needs heuristic scoring metrics to measure the "goodness of fit" of a particular network to prior knowledge and data. One can search for the network structure with the highest posterior probability (**maximum a posteriori – MAP – structure**). As it was the case for neural networks, overly-complex networks may easily lead to very large posterior probability and should be discouraged, for example through Akaike information criterion [7]. More sophisticated local search mechanism going beyond local optimality can be employed (Chapter 27).

As you imagine, the topic is rapidly getting very technical and we have to stop. But continue your exploration in case you encounter applications characterized by a rich structure of relationships (in some cases relationships of cause and effects), not excessively large, with abundance of existing domain knowledge and requiring high-level

explanation. Notable cases are medical diagnosis, computational biology and bioinformatics (e.g., gene regulatory networks), semantic search, image processing, law, decision support systems, financial informatics (risk analysis). Generalizations of Bayesian networks that can represent and solve decision problems under uncertainty are called **influence diagrams**.

14.2 Markov networks

Markov random fields (or **Markov networks**) are based on *undirected* graphical models, possibly with cyclic dependencies. Different Markov properties can be assumed, like the fact that the variable associated to a node is *conditionally independent* of all other variables *given its neighbors*. For a concrete image, imagine particles moving among nodes (so that the probability of a node is related to the fraction of particles sitting in that specific node) and imagine that you sit at one node. The stochastic arrival or departure of particles depend only on the local situation and not on the distant nodes (and each particle forgets about the previous part of its trajectory). Markov random fields are studied in Statistical Physics to explain "spin glasses", and enjoy a large popularity in computer vision. A rich theory is available. To rapidly hint at some results, in a commonly used class of Markov random fields the joint probability density can be *factorized* according to the cliques of the graph (cliques are subset of nodes which are completely connected).

$$P(X = x) = \prod_{C \in \mathrm{cl}(G)} \phi_C(x_C) \tag{14.4}$$

This is a huge simplification if cliques are small (in a typical computer vision applications cliques can be associated to neighboring pixels in an image).

Consider a field where values at the nodes can be among a finite set. Any Markov random field (with a strictly positive density) can be written as log-linear model with feature functions f_k such that the full-joint distribution can be written as

$$P(X = x) = \frac{1}{Z} \exp\left(\sum_k w_k^\top f_k(x_{\{k\}}) \right)$$

where the notation $w_k^\top f_k(x_{\{k\}}) = \sum_{i=1}^{N_k} w_{k,i} \cdot f_{k,i}(x_{\{k\}})$ is simply a dot product over field configurations, and Z is the partition function (needed so that probabilities sum up to one):

$$Z = \sum_{x \in \mathcal{X}} \exp\left(\sum_k w_k^\top f_k(x_{\{k\}}) \right).$$

Here, $\mathcal{X}$ denotes the set of all possible assignments of values to all the network's random variables. Usually, the feature functions $f_{k,i}$ are defined such that they are indicators of the clique's configuration, i.e. $f_{k,i}(x_{\{k\}}) = 1$ if $x_{\{k\}}$ corresponds to the $i-th$ possible configuration of the $k-th$ clique and 0 otherwise.

Gibbs sampling can be used in Markov networks to sample from the joint probability distribution. The intuition behind Gibbs sampling is to start from a configuration of $\mathbf{x}$ (a vector of random variables), then consider randomly one variable x_j at a time and update the current value with one derived from the conditional distribution specified by $p(x_j|x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)$. The method is therefore very fast if conditional probabilities are a function of a small subset of local variables. Gibbs sampling is a **Markov chain Monte Carlo (MCMC)**algorithm for obtaining a sequence of observations which are approximated from a specified multivariate probability distribution. The samples in the generated **Markov chain** approximate the joint distribution of all variables, the expected value of any variable can be approximated by **averaging over all the samples**. But let's note: samples are not independent: subsequent samples are in fact highly correlated. It is common to ignore some number of samples at the beginning (the so-called *burn-in* period), and then consider only every n-th sample when averaging values to compute an expectation. Some additional black magic is involved in many applications in statistical inference. In machine learning, missing values for some input variables can be handled by simply fixing the values of all variables whose values are known, and sampling from the remainder.

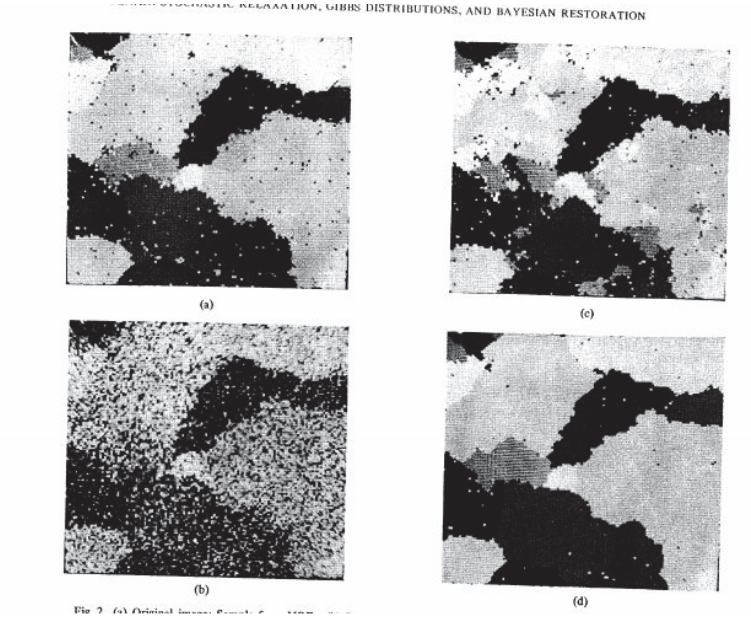


Figure 14.3: Gibbs sampling in computer vision (image restoration) from [143].

A very influential paper with applications in computer vision is [143]. The authors propose an analogy between images and statistical mechanics. Pixel gray levels and presence and orientation of edges are viewed as state of atoms in a lattice-like system (a Markov random field). The probability of a pixel value depends (mostly) on the probability of a set of neighboring pixels, which means that the probability density functions factorize as a product of "local" terms, easy and fast to calculate, see equation (14.4).

Maximum *a posteriori* (MAP) estimates of images given a degraded observation (with some noise at pixel levels, like in old-style television without robust error-correcting coding), are derived by Gibbs sampling and Simulated Annealing versions. A mental image is that the initial image is modified at randomized pixel locations, and flickers more at the beginning, less when an approximation of a stationary state maximizing the probability is reached. In spite of the enormous influence and number of theoretical studies originating from Statistics and Physics, like all MCMC methods, Gibbs sampling tends to be extremely slow for all apart simple and small-scale applications, and therefore should not generate excessive expectations.

14.3 Inductive logic programming (ILP)

The early phase of Artificial Intelligence concentrated on **symbolic approaches based on if-then rules (knowledge base)** and **inference engines**. Formal logic, knowledge representation and automated reasoning were under the spotlight.

The dream was to separate **knowledge acquisition**, in which a domain expert would make the critical information required for the system to work *explicit*, from **automated reasoning**. By removing the need to write conventional code, and therefore removing the need for trained programmers, experts could develop systems themselves trough **expert systems**. The use of rules to explicitly represent knowledge also enabled **explanation capabilities**, even in English (human) language.

In expert systems an inference engine is being driven by the antecedent (left hand side) or the consequent (right

hand side) of the rule. In forward chaining an antecedent *fires* and asserts the consequent. For example, consider the following toy example with the rule:

$$R1 : Man(x) => Mortal(x)$$

A simple example of forward chaining would be to assert $Man(Socrates)$ to the system and then trigger the inference engine. It would match $R1$ and assert $Mortal(Socrates)$ into the knowledge base. Backward chaining is less straightforward: the system looks at possible conclusions and works backward to see if they might be true.

Logic programming is a programming paradigm based on formal logic at the base of this high-level approach to developing intelligent systems. A program written in a logic programming language is a set of sentences in logical form, expressing facts and rules about some problem domain. Prolog is a notable programming language in this area. In all of these languages, rules are written in the form of clauses:

$$H \ :- \ B_1, \ldots, B_n.$$

and are read declaratively as logical implications:

$$H \ \text{if} \ B_1 \ \text{and} \ \ldots \ \text{and} \ B_n.$$

H is called the head of the rule and $B_1, \ldots, B_n$ is called the body. Let's note that the ": $-$" symbol is used to fake a left arrow by standard keyboard symbols (keyboard design was for secretaries not for programmers!).

The knowledge-base and inference-engine dream generated hype and then following disillusion, when it turned out that passing from rapid prototypes to real-world systems in many cases created an explosion of CPU times for reasoning, difficulty in maintaining systems, **fragility** when application was at the boundary of the domain. While the rules for an expert system were more comprehensible than typical computer code they still had a formal syntax where a misplaced comma or other character could cause havoc as with any other computer language.

Nonetheless, research is still in specific areas like natural language processing (language parsing), bio-informatics, hardware-software verification. The new developments are mentioned in this book for completeness but also because ILP is deeply involved with **learning and optimization**, to simplify system developments, take account of examples in addition to rules, make the systems more robust and dynamic.

In particular, **Inductive logic programming (ILP)**[277] is a subfield of machine learning which uses logic programming as a uniform representation for *examples*, background knowledge and hypotheses. Given an encoding of the known background knowledge and a set of **examples** represented as a logical database of facts, an ILP system will **derive a hypothesised logic program** which entails all the positive and none of the negative examples. The term "inductive" here refers to a philosophical induction process of suggesting a theory to explain observed *facts*. Current research[118] deals with *reasoning about the identity* of objects (like recognizing that vehicles found in different images represent the same physical object), *inventing new objects* to achieve compact hypotheses which explain empirical observations (like inventing an unknown enzyme to explain effects in a metabolic network), *incremental theory revision* (background knowledge can be incomplete and incorrect, it can be revised when abundant experimental data are present), *logical experimental design* (optimal design of experiments to test hypothesis in laboratory settings).

Structured machine learning[118] is an umbrella term for ML techniques that involve predicting structured objects, like a parse tree (part-of-speech tagging natural language) or a symbolic interpretation of a visual scene, rather than scalar discrete or real values. Structured ML includes learning logical representations, like in ILP or Bayesian networks. The current research deals with handling uncertainty in a principled manner by incorporating probabilities into logical and relational representations. The long-term goal is to deal with the dynamic nature of systems, to deal with *shifting sands* of non-stationary distribution in order to achieve graceful degradation. From the computational point of view while searching for the best structure, dynamic programming (Viterbi and variations), local search, greedy search and **beam search** (which extends a greedy approach by considering a certain number B of interesting candidates to modify instead of a single one), are in the bag of tools to achieve acceptable CPU times to deal with real-world systems.

```
<html>
 <head>
  <title>Learning and Intelligent Optimization</title>
  <meta name="author" content="Roberto Battiti">
  <meta name="keywords" content="LION, ML, optimization, big data">
 </head>
 <body>
  <h1>The LION way is the future</h1>
  The reasons are explained in the
  <a href="intelligent-optimization.org"> LIONlab homepage </a>.
 </body>
</html>
```

Figure 14.4: An example of a web page written in HTML, the Hypertext Markup Language which is standard to describe the overall page structure.

Combining logical and statistical approaches is far from trivial but can produce advantages related to speed of development and human explanation in areas for which logical rules are present (e.g, natural language, robotics, vision). As an example, automated car driving requires both following many symbolic rules (including speed limits, road signs, safety requirements, etc.) and reacting very rapidly to unusual driving conditions or adapting the driving style in order to reduce fuel consumption, in some cases in sub-symbolic manners through learning-by-example.

14.4 Text and web mining: the context

When the data consist of a collection of documents, we can still use many of the techniques used to analyze numerical data but we need to adapt them, by suitably **preprocessing the documents and fine-tuning the ML methods**. Preprocessing transforms texts into vectors containing numeric values. Fine-tuning the ML methods has to deal with the fact that these vectors may possess a huge number of coordinates, that words have synonyms, that texts have a structure going beyond a bag of words, facts which require specific ad hoc metrics, feature selection and extraction.

Information retrieval deals mostly with searching for documents and for information within documents, and **web mining** is related to adapting methods to the context of the world wide web. The Web is an *unstructured* (or, at most, *semi-structured*) collection of data mostly in form of human-readable texts and images, connected by hyper-links. The Web is not a database: a complete description of data items structure (a *schema*) is missing, it is just a messy collection of human-readable data and human-exploitable hyper-links. There are efforts to help machines (computers) to automatically extract meaning from web pages through semantic support, but the task is daunting given the anarchic and continuously evolving structure of the web. "Semantic" means related to the "meaning" of the data items, and the so-called semantic web is an effort to add meta-data —data about the meaning of data— to enable automated agents and other software to access the Web more intelligently, for example understanding that a field is the name of a person, another field the age, another one the address, etc.

"**Big data**" is a popular marketing term for a collection of data sets so large, complex and unstructured that it becomes difficult to handle by using traditional data processing applications.

In addition to text, it is important to remember that web pages contain tags that modify the *appearance* and the *meaning* of the text, they contain links to other documents (**hyper-links**) and, in some cases, **meta-data** describing the meaning of the different parts. A short example of a page written in HTML is shown in Fig. 14.4. *No web page is an island entire of itself*: in fact hyperlinks help in searching for, ranking, and classifying pages in the web. Of course, similar linked structures are present in other areas, like social networks, bibliographic references in research papers, etc.

14.5 Retrieving and organizing information from the web

Before taking the plunge into the more interesting web mining tasks like ranking, clustering and classification, let's start with a short introduction about how to collect the raw content of the web pages (**crawling**), and structure it so that it is ready for further analysis (**indexing**). If you do not care about how the raw data is obtained and you are just interested in a high-level view, you may skip these sections and jump to Section 14.6.

The collected documents are processed into an **index** suitable for answering queries and retrieving information. Unlike the context of RDBMS (relational databases), *the order of answers is fundamental*: the user wants to see relevant data first. In other words: one aims at maximizing the probability that the first few answers will satisfy the user's needs. The union of a web crawler and a web index is a **search engine**.

In some cases **topic directories** are built to simplify searching. They are treelike structures (taxonomies), initially designed by hand. The process of organizing the documents can be automated by **clustering and unsupervised learning** methods. The purpose is the automated discovery of groups in the set of documents so that documents *inside* the same group are *more similar* than documents in different groups. As one may expect, similarity measures are a crucial issue when designing automated document clustering techniques.

14.5.1 Crawling

The processing of the web information starts with **crawling**, systematic methods to visit web pages and harvest the information contained therein. The basic crawling principle consists of visiting the web graph by starting from a given set of URLs, fetching and collecting the corresponding pages, scanning collected pages for hyperlinks to pages that have not been collected yet.

If you are familiar with *graphs*, nodes represent web pages, edges represent links and the task is to *visit the graph*, i.e., visit all nodes in a systematic manner while avoiding duplicate visits. While a basic crawler implementing a visit of the web graph can be put together with a little knowledge of the underlying communication protocol (HTTP), avoiding the many pitfalls requires careful design considerations:

- many web servers assume that a human mind is driving the web requests, therefore they assume that any attempt at fetching many pages per second is an attack, and respond by denying access;
- more and more pages on the Internet are dynamic in many subtle ways: their content depends on data previously input by the user, by pre-existing client cookies, even by the position the request is being originated from; therefore, any attempt to automatically collect all available information fails, and some user intelligence must be put into the system;
- resolving host names might take longer than fetching the data itself; in general, identifying the real bottleneck is not an easy task;
- the web is now dominated by virtual servers with their many-to-many relationships (Domain names to IP addresses, URLs to pages, not to mention mirrored or plagiarized info), so that identifying what has already been visited is becoming more and more difficult.

A minimal crawler architecture is shown in Fig. 14.5: a queue of still unvisited URLs is maintained; every time a page is fetched, it is scanned for new URLs. In order to overcome the aforementioned DNS bottleneck, a preliminary DNS request can be issued long before the URL is finally requested. Avoiding page and URL duplicates is also important, so various "is it new?" checkpoints can be placed throughout the workflow.

14.5.2 Indexing

Indexing is a required **preprocessing so that queries can be answered rapidly**. The simplest kind of queries, and by far the most used, involves one or more terms, in some cases combined by Boolean operators. For example one may

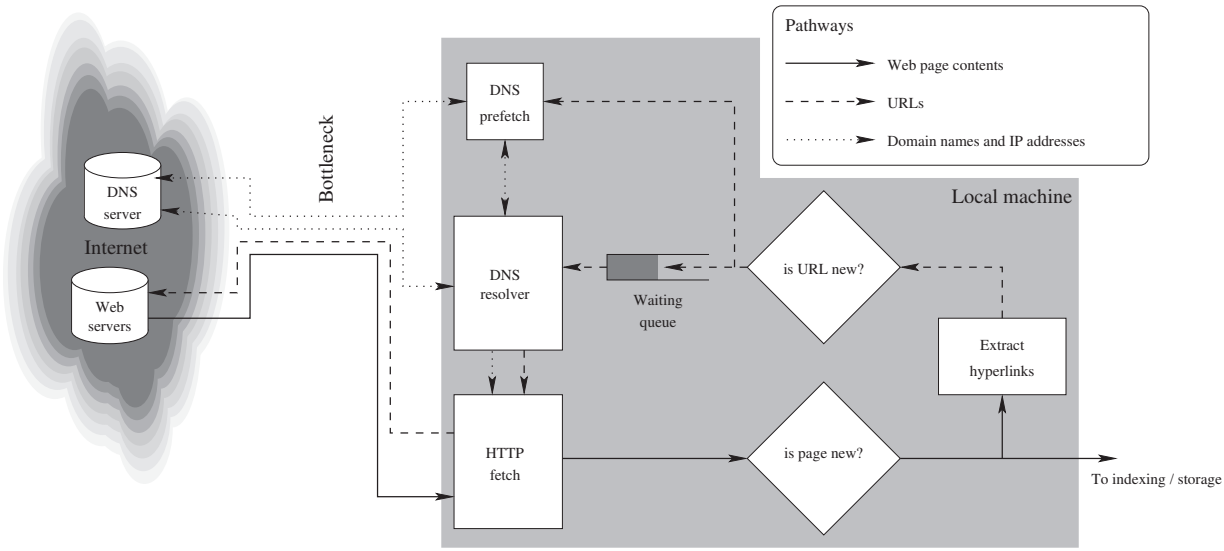


Figure 14.5: A basic crawler architecture: URLs are extracted from fetched pages and enqueued; domain names are pre-fetched to overcome the potential bottleneck.

search for: documents containing the word "Reactive" but not the word "Search"; documents containing the phrase "Reactive Search Optimization"; documents where "Reactive" and "Search" occur in the same sentence, etc.

Before building indices, documents undergo a sequence of cleaning steps, often including the following: HTML tags and other non-relevant markup items are filtered out (there are some exceptions: some meta-information should be retained, heading tags might provide information about the relevance and visibility of words); punctuation can be removed and replaced, if needed, by end-of-sentence markers; character casing is made uniform (e.g., all lowercase); the remaining text is *tokenized*, i.e., divided into words; very common words ("and", "I", "the"...), also known as *stopwords*, are removed; variant forms of the same word are collapsed to their *stem* (so that "play", "playing" and "played" all correspond to the same token).

While not all of the original information is preserved in the process, from the information retrieval point of view, the lost part is mainly noise, and a user who sends the query "Shakespeare play" to a search engine will expect results containing the words "Shakespeare plays" too, with proper casing and plural forms.

Fig. 14.6 shows two sample documents[1], d_1 and d_2, where the subscript denotes the position of the token in the document:

A direct index is a table mapping term ID `tid` to document's ID and position (`did`,`pos`). Such table, shown in the left-hand side of Fig. 14.6, makes searching for all documents containing a token very inefficient (one has to scan the entire table). An inverted index is a table obtained by "transposing" the previous one (right-hand side of Fig. 14.6), and giving for each token the list of documents that contain it.

14.6 Information retrieval and ranking

After the raw content of the web is properly saved and preprocessed (indexed), let's now consider the more interesting task of searching for documents, and searching for information within documents, also called Information retrieval (IR). In general, one wants to retrieve documents which are *relevant* to a query and which are of *good quality*. If one

[1] WILLIAM SHAKESPEARE — *The Life and Death of Richard the Second*, Act IV, Scene 1.

d_1 = My$_1$ care$_2$ is$_3$ loss$_4$ of$_5$ care$_6$, by$_7$ old$_8$ care$_9$ done$_{10}$.
d_2 = Your$_1$ care$_2$ is$_3$ gain$_4$ of$_5$ care$_6$, by$_7$ new$_8$ care$_9$ won$_{10}$.

tid	did	pos
my	1	1
care	1	2
is	1	3
⋮	⋮	⋮
new	2	8
care	2	9
won	2	10

tid	pos list
my	d_1/1
care	d_1/2,6,9 // d_2/2,6,9
is	d_1/3 // d_2/3
loss	d_1/4
of	d_1/5 // d_2/5
by	d_1/7 // d_2/7
old	d_1/8
done	d_1/10
your	d_2/1
gain	d_2/4
new	d_2/8
won	d_2/10

Figure 14.6: Two documents (top) and their direct (left) and inverted (right) index, from [75].

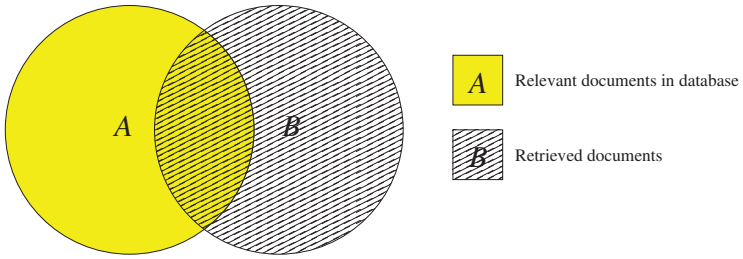


Figure 14.7: Information retrieval: relevant and retrieved documents.

searches for "loss" and "care" one may retrieve a piece by Shakespeare, as well as documents about, let's say, "hair loss care," which are probably of inferior quality, if you are interested in literature and not in hair loss.

Standard definitions of performance measures when retrieving documents are as follows. If A is the set of relevant documents, and B the set of retrieved documents, see also Fig. 14.7 and Fig. 3.5 in Section 4.3, one identifies:

- retrieved relevant items (true positives): $A \cap B$;
- retrieved irrelevant items (false positives): $B \setminus A$;
- unretrieved relevant items (false negatives): $A \setminus B$.

The **precision** of a retrieval system is defined as the fraction of retrieved documents that is relevant:

$$\text{precision} = \frac{|A \cap B|}{|B|}.$$

The **recall** of the system is defined as the fraction of relevant documents that is retrieved:

$$\text{recall} = \frac{|A \cap B|}{|A|}.$$

The recall measure usually is not so relevant for web searches, where the number of relevant documents typically is too large for a human to examine. For search engines, the order in which results are presented to the user is fundamental. In general, such order implies *ranking* the documents, so an adequate performance measure should favor those methods that place relevant documents in the highest ranks, and show them first in the user browser as response to a search.

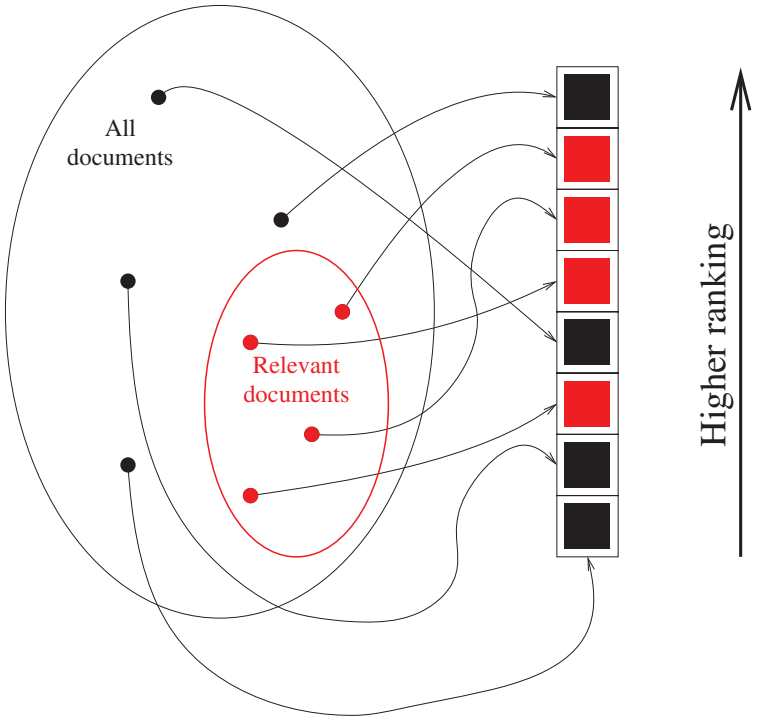


Figure 14.8: A ranking example.

Consider Fig. 14.8, where the darker dots represent relevant documents, while the ranking order is provided on the right. It is clear that the best ranking procedure should place the bright (red) elements in the top positions. Let's introduce more specialized definition of performance to take this into account.

Let D be a corpus of $n = |D|$ documents, and let q be a query. Define $D_q \subset D$ as the set of all relevant documents for query q. We assume that D_q represents the "desired" answer of the system. Let $(d_1^q, d_2^q, \ldots, d_n^q)$ be an ordering ("ranking") of D returned by the system in response to query q. Let $(r_1^q, r_2^q, \ldots, r_n^q)$ be defined as

$$r_i^q = \begin{cases} 1 & \text{if } d_i^q \in D_q \\ 0 & \text{otherwise.} \end{cases}$$

We can now define rank-dependent versions of the recall and precision figures, which will help us answer the question "how would we rate the performance of our system if we only took the top-k ranked answers?"

The recall at rank k is defined as the fraction of relevant documents found in the top k positions:

$$\text{recall}_q(k) = \frac{1}{|D_q|} \sum_{i=1}^{k} r_i^q,$$

and similarly for the precision:

$$\text{precision}_q(k) = \frac{1}{k}\sum_{i=1}^{k} r_i^q.$$

As usual, there are no free meals: when analyzing the ranked list, the recall can be increased by increasing k; but then, more and more irrelevant documents occur, driving down the precision (**precision-recall tradeoff**).

14.6.1 From Documents to Vectors: the Vector-Space Model

To use standard techniques designed for vector spaces to search, cluster, and classify documents, we first need to map each document to a vector (the **vector-space model**).

After preprocessing, our document is now a *bag of words*, actually a bag of tokens, and the most straightforward way to obtain a vector is to: i) fix a set of terms (tokens), ii) have a separate axis to represents each term (token), iii) set the value of the vector along the axis t to zero if the document does not contain the token t, to a number greater than zero if the document contains the token one or more times.

If $n(d,t)$ is the number of times that document d contains the term t, the **term frequency** $\text{TF}(d,t)$ of term t in document d is defined as a figure that increases monotonically with the relative frequency of t in d. Some possible definitions are the following:

$$\begin{aligned}
\text{TF}(d,t) &= \frac{n(d,t)}{\sum_\tau n(d,\tau)} \\
\text{TF}_{\text{SMART}}(d,t) &= \begin{cases} 0 & \text{if } n(d,t)=0 \\ 1+\log\bigl(1+\log n(d,t)\bigr) & \text{otherwise.} \end{cases}
\end{aligned}$$

The TF_{SMART} formula is meant to avoid an exaggerated value along a dimension if a term is present too many times. This was a frequent case in the initial years of the web, when simple search engines were just counting term occurrences. It used to be that many pages contained for example the term "sex" repeated hundreds of times, aiming at reaching a high rank for many users searches. Actually, the more recent search engines combat this kind of spamming by using the hyperlink information, as we will see later.

Actually, often the most interesting terms are the ones which do not appear in many documents (**rare terms** like "C++", "Reactive Search Optimization", "stochastic" are probably more informative than "is", "nice", "free", "excellent"), and an **inverse document frequency** can be defined as a figure that monotonically *decreases* as the overall frequency of a term in the whole document corpus increases:

$$\text{IDF}(t) = \log\frac{1+|D|}{|D_t|},$$

where D_t is the set of documents containing term t, and the logarithm is used to avoid an exaggerated multiplier for very rare terms. Let's note that the above methods to derive vectors are heuristic and not based on fundamental principles like information theory. If you think that the logarithm is not appropriate, feel free to experiment with other functions. After discounting the importance of weak terms appearing in too many documents, a specific document d in TF-IDF space (term-frequency inverse-document-frequency) is represented by vector

$$\boldsymbol{d} = (d_t)_{t\in\text{terms}} \in \mathbb{R}^{\text{terms}},$$

where component d_t is

$$d_t = \text{TF}(d,t)\,\text{IDF}(t).$$

A query q is a sequence of terms, therefore it admits a representation $\boldsymbol{q} = (q_t)$ in the same space as documents. Given the query $\boldsymbol{q}$ and the document $\boldsymbol{d}$, one can now measure their proximity, by considering vector-space similarity measures, as shown in Fig. 14.9. Two frequently used proximity measures in TF-IDF space are listed below.

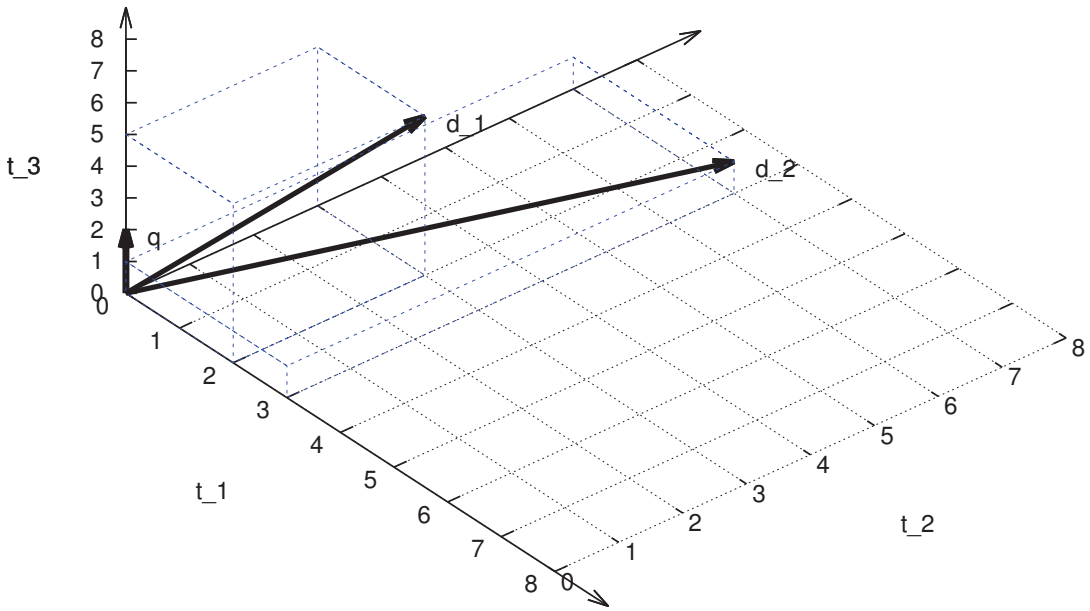


Figure 14.9: Geometric Interpretation.

- Euclidean distance $\|\boldsymbol{d} - \boldsymbol{q}\|$. To avoid artifacts, vectors should be normalized,i.e., an n-fold replica of document $\boldsymbol{d}$ should have the same similarity to $\boldsymbol{q}$ as $\boldsymbol{d}$ itself:

$$\left\| \frac{\boldsymbol{d}}{\|\boldsymbol{d}\|} - \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|} \right\|.$$

- Cosine similarity, i.e., the cosine of the angle between vectors $\boldsymbol{d}$ and $\boldsymbol{q}$:

$$\frac{\boldsymbol{d} \cdot \boldsymbol{q}}{\|\boldsymbol{d}\| \, \|\boldsymbol{q}\|},$$

  see also equation (17.3).

An Information Retrieval system based on TF-IDF coordinates therefore works as follows. First build an inverse index with TF(t, d) and IDF(t) information. When given a query, map it onto TF-IDF space, sort documents according to the similarity metric, return the most similar documents. Searching methods can be extended in different ways, for example to search for phrases. The book [75] presents more details on the topic which cannot be presented in this short introductory chapter.

Please note that there is nothing magic in the traditional TF-IDF representation: it is just a heuristic recipe to give more weight to more informative words, so that the standard metrics given above can produce reasonable results. More

sophisticated metric-learning or feature-selection methods based on information content (like mutual information) can lead to superior results but require a deeper knowledge.

14.6.2 Relevance feedback

As mentioned above, after transforming the query into a vector, a vector-similarity measure can be used to identify a set of most similar documents to return to the user.

Unfortunately, the average web query is as few as one or two terms long, and it is not surprising that a lot of irrelevant documents can be retrieved. This is why either a serious measure to rank qualitatively superior documents is needed (like PageRank in Section 14.7) or at least a way to rapidly get **feedback from the user** and use it to form a better query.

Rocchio's Method is based on updating the vector used for the first query to make it more similar to vectors describing documents that the user identifies as relevant (likes), and less similar to the ones classified as irrelevant (dislikes), as illustrated in Fig. 14.10. For a mental image, think about documents that the user liked *attracting* the query vector, and documents that he disliked *repelling it*. In details, the query vector is updated as:

$$\boldsymbol{q}' = \alpha \boldsymbol{q} + \beta \sum_{\boldsymbol{d} \in D_+} \boldsymbol{d} - \gamma \sum_{\boldsymbol{d} \in D_-} \boldsymbol{d},$$

where D_+ is a set of retrieved documents liked by the user, and D_- is a set of retrieved documents that the user dislikes. Parameters α, β and γ control the amount of modification. The careful reader may notice a resemblance with the way in which prototype vectors are updated in the self-organizing maps in Chapter 19.

14.6.3 More complex similarity measures

In a TF-IDF vector space, we can define the "similarity" between two items as a decreasing function of distance — its inverse, for example, although it goes to infinity when comparing an object to itself, requiring some correction. If the elements admit a set representation, another similarity criterion is available, the *Jaccard coefficient*. Let A and B be two (finite) sets, the Jaccard coefficient of A and B is defined as

$$r'(A, B) = \frac{A \cap B}{A \cup B}.$$

Its aim should be clear: compare what is common to the two sets (their intersection) to their total size. It varies between 0 and 1, where $r'(A, B) = 0$ implies that the two sets have no common element and $r'(A, B) = 1$ means that A and B are equal. An additional important property is that $1 - r'(A, B)$ is a *distance*, it obeys all the properties of a metric.

Let us adopt a more document-centric definition. If d is a document, let's define $T(d)$ as the set of tokens (terms) it contains. Note that, as always when referring to sets, elements have no multiplicity and we are just interested in a binary model where a term either occurs or does not. Then, the *Jaccard coefficient* of the two documents is

$$r'(d_1, d_2) = \frac{|T(d_1) \cap T(d_2)|}{|T(d_1) \cup T(d_2)|}.$$

The use of the Jaccard coefficient in search can be motivated as follows: a query is usually seen by the user as a set of terms without repetitions, and no user would ever write a query such as "reactive reactive search" into Google expecting it to return documents containing the word "reactive" twice as frequently as the word "search".

The skeleton of an algorithm for computing the *Jaccard coefficient* $r'(\cdot, \cdot)$ is the following one.

- For each $d \in D$:
  - for each term $t \in T(d)$: put record (t, d) on file f_1.

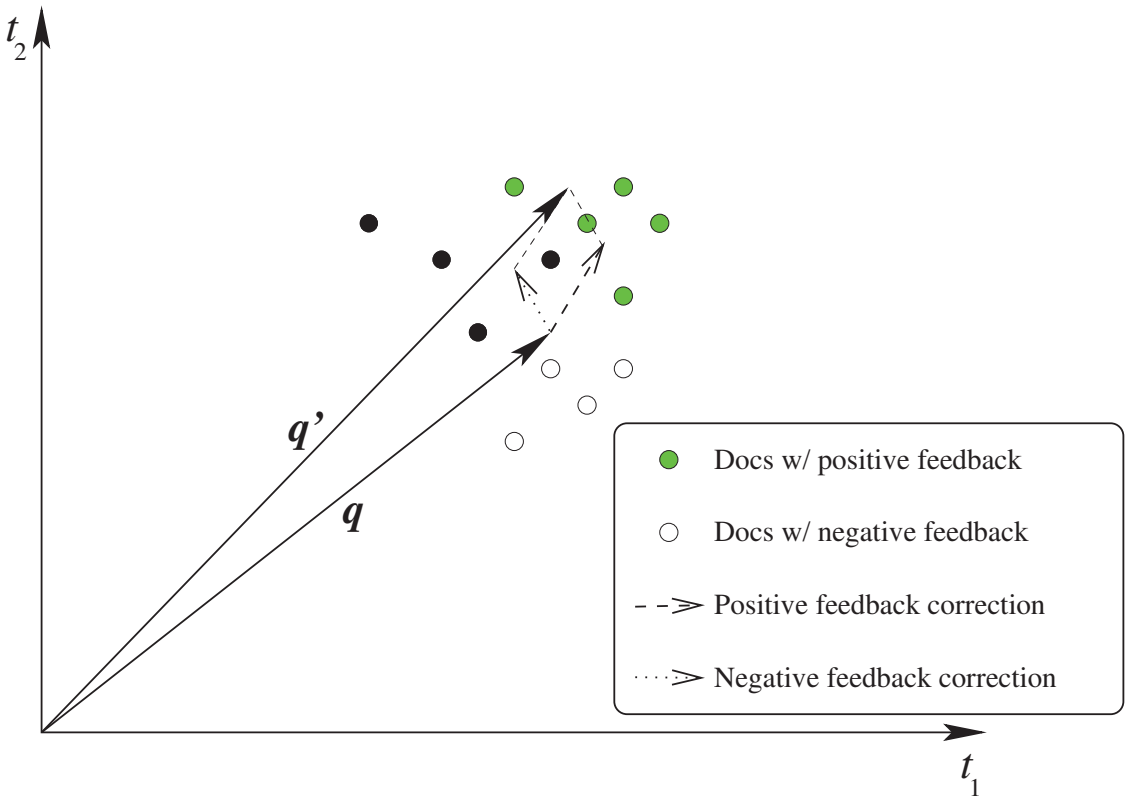


Figure 14.10: Rocchio's method.

- Sort f_1 in (t, d) order and aggregate into form (t, D_t).
- For each term t scanned from f_1:
  - for each pair $d_1, d_2 \in D_t$: put record (d_1, d_2, t) on file f_2.
- Sort f_2 on (d_1, d_2) and aggregate by adding on the third field.

Some possible tricks to reduce the search cost are related to pre-computing the Jaccard coefficient for all pairs of documents and queries, which can require huge amounts of storage and CPU time, or reducing the set of pairs, by pre-associating every document or query to a list of a small and fixed number of the most similar documents. In addition, very frequent terms (having low IDF) can be omitted entirely from consideration.

In practice, in many cases one is interested in *approximating* the coefficient. An interesting randomized algorithm using random permutations is available.

If one uses probabilities, given sets A and B, one starts from this interesting equality:

$$\frac{|A \cap B|}{|A \cup B|} = \Pr(x \in A \cap B | x \in A \cup B).$$

If we can estimate the above probability, we can estimate the Jaccard coefficient. What we can do is to generate random elements in the set $S \subset \{1, \ldots, n\}$ and to estimate the probability by the ratio of events.

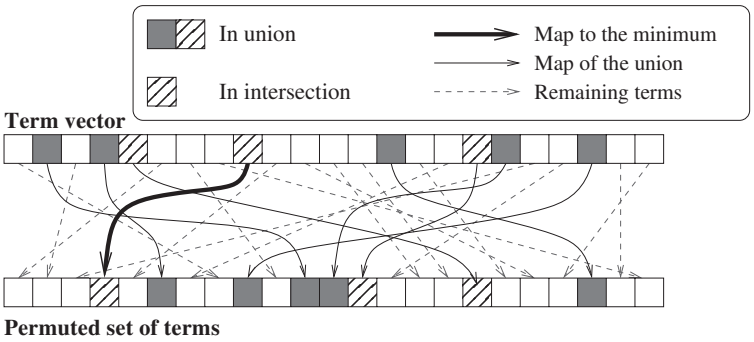


Figure 14.11: Building a permutation.

To select a random element from a set $S \subset \{1, \ldots, n\}$ one can select a random permutation π on n elements and pick the element in S such that its image in π is minimum:

$$x = \arg\min_{x \in S} \pi(x) = \arg\min \pi(S)$$

When applied to $A \cup B$, this method locates an element in the intersection if and only if

$$\min \pi(A) = \min \pi(B).$$

We can therefore estimate the above ratio by applying random permutations and checking if the two minima are coincident.

Let's demonstrate why permutations work. Given sets $A, B \subset \{1, \ldots, n\}$, to derive the probability that the two minima are coincident, let us count how many permutations $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ (of the $n!$ possible) have the property

$$\min \pi(A) = \min \pi(B)$$

by building one such permutation. From Fig. 14.11 it should be clear that:

- The image of $A \cup B$ (lighter and darker squares) can be chosen in $\binom{n}{|A \cup B|}$ different ways.
- Within such image, the minimum element can be chosen within the $|A \cap B|$ elements that form the intersection (thick arrow).
- The remaining elements in the image of $A \cup B$ can be permuted in $(|A \cup B| - 1)!$ ways (thin arrow).
- The elements not in $A \cup B$ can be permuted in any of $(n - |A \cup B|)!$ ways (light arrows).

After multiplying all these, one obtains:

$$\binom{n}{|A \cup B|} \cdot |A \cap B| \cdot (|A \cup B| - 1)! \cdot (n - |A \cup B|)! = n! \frac{|A \cap B|}{|A \cup B|},$$

and after dividing by the total number of permutations one derives the desired equality.

A randomized but inefficient algorithm is as follows:

- generate a set Π of m permutations on the set of terms;
- $k \leftarrow 0$

- for each $\pi \in \Pi$:
  - if $\min \pi(T(d_1)) = \min \pi(T(d_2))$ then $k \leftarrow k + 1$;
- estimate $r'(d_1, d_2) \approx \dfrac{k}{m}$.

By combining a randomized algorithm with suitable data structures working on external storage and simultaneous computation of the coefficients for many documents, one manages to deal with the enormous amount of documents contained in the world wide web!

14.7 Using the hyperlinks to rank web pages

There are so many web pages that the issue is not only that of retrieving a few set of pages relevant to the query, but that of retrieving a set of *high quality* and relevant pages. The issue predates the web and is encountered also when reading books, or papers. One would like to concentrate on good quality papers written on a subject, without wasting time on poor quality ones. In scientific communities, a paper is considered of good quality if it is *cited* by other good quality papers, meaning that some colleagues found the paper useful and acknowledged it by putting it the list of citations. For a more mundane analogy, a candidate for employment is valued if many other valued people recommend him. In general, see also Fig. 14.12, in a social network of relationships between people, a high reputation is obtained by having other highly-reputed people recommending you. It is not sufficient to convince many low-rank individuals to support you, there are no shortcuts!

After a seminal paper by Marchiori [260] highlighting the importance of *hyper-information* (information in the hyperlinks), Larry Page and Sergey Brin developed the PageRank algorithm, which follows the same basic social networks principles, by substituting "recommendations" and "citations" with hyperlinks [286] (the authors then became Google founders). They define a "measure of prestige" such that the prestige of a page is related to how many pages of prestige link to it. Let's note that this is a *recursive definition*. To measure the prestige of a page one needs to have the prestige of pages pointing to it, and so on. In short, their solution is: start with an initial distribution of prestige values, iterate the prestige calculation for the different nodes, and stop when the values do not change too much after recalculation, as simple as that! At first reading, this process seems prone to pitfalls. What guarantee do we have that the process converges, hopefully to the same limiting distribution, not depending on the initial distribution of values?

Now: it is fascinating how the solution to this problem is related to basic linear algebra concepts of **eigenvalues and eigenvectors**, as well as concepts related to **Markov chains**. Let's summarize the main relationships.

First, let's see how iterating the prestige calculation after starting from an initial distribution is related to the classic **power iteration method for finding the dominant eigenvector** of a matrix. We proceed very rapidly, skipping mathematical details, only to give the flavor of the method.

The rank of a page is calculated by examining the incoming links (the hyper-links of other pages pointing to the given page). Each incoming link from page i contributes a partial rank equal to the rank of i divided by the number of i's outgoing links, as shown in Fig. 14.13. The human motivation for the division is clear: a page of high rank but pointing to a very large number of pages is like a person of good quality but recommending a too large number of people. Without the division, the owner of a top-ranked page could influence all pages in the world just by putting an enormous number of outgoing links.

Given the above recalculation rule, once the network of hyper-links is given, the computation of new rank values $\mathbf{p}^k$ at iteration k is obtained by a *linear* transformation of the previous values through a matrix, which we denote as M, as follows:

$$\mathbf{p}^k = M\mathbf{p}^{k-1}.$$

The matrix M will depend only on the connectivity structure, on the links between pages. Now, after starting from the initial rank distribution $\mathbf{p}^0$ and executing k recalculations:

$$\mathbf{p}^k = M^k\mathbf{p}^0.$$

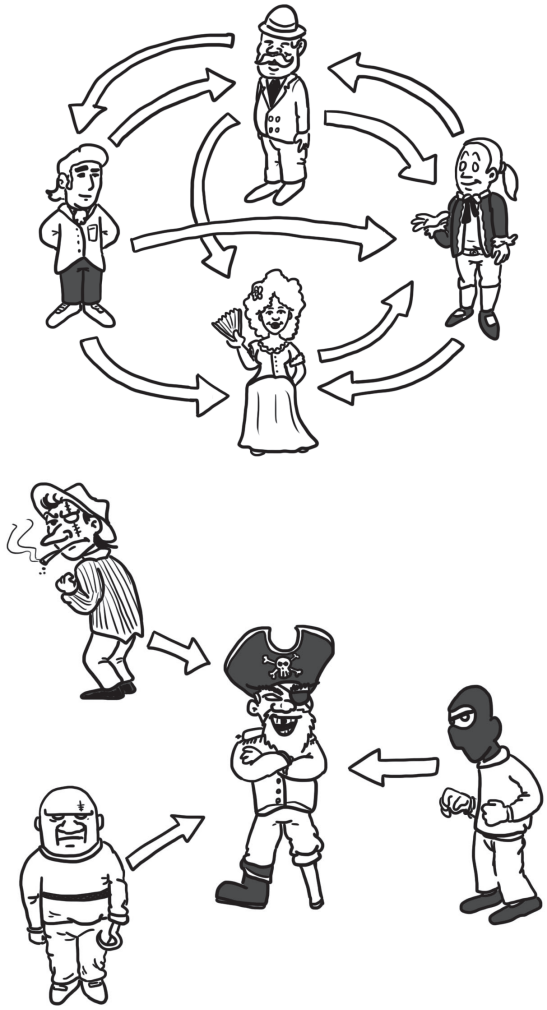


Figure 14.12: Prestige in social networks: recommendations from (or relationship with) high-rank individuals (above) are more effective to reach a high rank than recommendations by low-rank ones (below).

Assume that a basis of eigenvectors of M is available, let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the n eigenvalues, and let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ be the corresponding eigenvectors. Suppose that λ_1 is the dominant eigenvalue, so that $|\lambda_1| > |\lambda_j|$ for $j > 1$.

The initial vector $\mathbf{p}^0$ can be written as a linear combination of the basis vectors:

$$\mathbf{p}^0 = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_n\mathbf{v}_n.$$

If $\mathbf{p}^0$ is chosen randomly (with uniform probability), then $c_1 \neq 0$ with probability 1. Now, using linearity and the defining property of eigenvectors, one immediately obtains:

$$\begin{aligned} M^k\mathbf{p}^0 &= c_1M^k\mathbf{v}_1 + c_2M^k\mathbf{v}_2 + \cdots + c_nM^k\mathbf{v}_n \\ &= c_1\lambda_1^k\mathbf{v}_1 + c_2\lambda_2^k\mathbf{v}_2 + \cdots + c_n\lambda_n^k\mathbf{v}_n \\ &= c_1\lambda_1^k\left(\mathbf{v}_1 + \tfrac{c_2}{c_1}\left(\tfrac{\lambda_2}{\lambda_1}\right)^k\mathbf{v}_2 + \cdots + \tfrac{c_n}{c_1}\left(\tfrac{\lambda_n}{\lambda_1}\right)^k\mathbf{v}_n\right). \end{aligned}$$

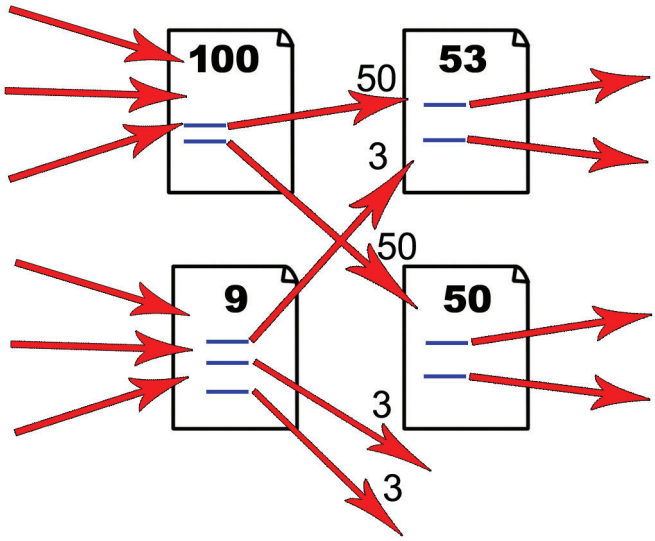


Figure 14.13: Recalculating the rank of a page in PageRank. The initial rank is distributed along the outgoing links (adapted from the original paper).

When the number of recalculations k, equal to the power of the matrix M^k, goes to infinity, all terms tend to zero apart from the one proportional to the dominant eigenvector. A simple iteration of matrix multiplication after starting from almost arbitrary initial conditions is indeed sufficient to extract the dominant eigenvector!

Let's now consider a different interpretation related to Markov chains, and imagine that you want to analyze a system representing the **movement of a web surfer** on the various web pages. Assume that a surfer is navigating through outgoing links forever, picking them uniformly at random. Let the starting page u be taken with probability p_u^0. Let E be the adjacency matrix of the web: $(u, v) \in E$ (or $E_{uv} = 1$) if and only if there is a link from page u to page v. What is the probability p_v^i of the surfer being at page v after i clicks?

Let's start with a single step. What is the probability p_v^1 that surfer is at page v after one step? Let

$$N_u = \sum_v E_{uv}$$

be the *out-degree* of page u (sum of u-th row of E). Suppose no parallel edges exist,

$$p_v^1 = \sum_{(u,v)\in E} \frac{p_u^0}{N_u}.$$

By normalizing E to have row sums equal to 1

$$L_{uv} = \frac{E_{uv}}{N_u},$$

we get

$$p_v^1 = \sum_u L_{uv} p_u^0 \qquad \text{or} \qquad \mathbf{p}^1 = L^T \mathbf{p}^0.$$

Let's now consider the situation after i steps:

$$\mathbf{p}^i = L^T \mathbf{p}^{i-1}.$$

If E is *irreducible* and *aperiodic* (none is actually true but the problem can be cured), then

$$\lim_{i \to \infty} \mathbf{p}^i = \mathbf{p},$$

where $\mathbf{p}$ is the principal eigenvector of L^T, a.k.a. its *stationary distribution*:

$$\mathbf{p} = L^T \mathbf{p} \qquad \text{(eigenvalue is 1)}.$$

But p_u it the *prestige* of page u determined also by the previous interpretation. It is now clear how the prestige can be interpreted also as probability that a random surfer following links will be found at a given page.

Let's now deal with bad properties of real-world transition matrices. Surveys show that the Web is not strongly connected, and that random walks can be trapped into cycles. A possible fix is to introduce a "damping factor" corresponding to a user that occasionally stops following links: with an arbitrary probability d of going to a random page (even unconnected) at every step. The transition becomes:

$$\mathbf{p}^i = \left((1-d) L^T + \frac{d}{N} \mathbf{1}_N \right) \mathbf{p}^{i-1}.$$

The eigenvector of the matrix corresponding to the largest eigenvalue can be obtained as follows.

- Start with random vector $\mathbf{p} \leftarrow \mathbf{p}^0$;
- repeat:
  - update vector:
$$\mathbf{p} \leftarrow \left((1-d) L^T + \frac{d}{N} \mathbf{1}_N \right) \mathbf{p};$$
  - from time to time, normalize it:
$$\mathbf{p} \leftarrow \frac{\mathbf{p}}{\|\mathbf{p}\|_1}.$$

Normalization avoids very large components and therefore numerical problems with finite-precision computation. Of course, for the application we are not interested in *absolute* prestige values but in *relative* ones. The absolute values depend on the chosen range (one may measure prestige on a range from 0 to 10, or on a range from 0 to 100, etc.) but what is relevant is that a page is, say, three times more prestigious than another one. Normalization is a simple way to discount multiples of the given normalized eigenvector.

In practical applications, the notion of prestige is so fuzzy that nobody will ever care about obtaining the actual eigenvector with high precision! To have a flavor of how long is required for convergence, in his original paper Page says that 52 iterations are enough for about 3×10^8 pages, quite an exciting result paving the way to significant business applications.

14.8 Identifying hubs and authorities: HITS

Let's now consider a different analysis of the web. In a scientific community all good articles are either seminal (i.e., many others reference to them) or surveys (i.e., they reference to many others). In the web, pages may be *authorities* or *hubs* [146]. For example portals are very good hubs, even if they do not contain significant information, and they are used only as starting points to reach good quality pages.

To reflect this distinction, let's introduce *two* score measures, called **hubness** and **authority**:

$$\boldsymbol{h} = (h_u), \qquad \boldsymbol{a} = (a_u).$$

Let's now summarize the HITS algorithm (Hyperlink-Induced Topic Search). In the HITS algorithm, the first step is to retrieve the set of results to the search query. Given query q, let R_q be the *root set* returned by an IR system. The computation is performed only on this result set, not across all Web pages. Authority and hub values are defined in terms of one another in a mutual recursion.

The *expanded set* is formed by adding all nodes linked to the root set:

$$V_q = R_q \cup \{u : ((u \to v) \vee (v \to u)) \wedge v \in R_q\}.$$

Let E_q be the induced link subset, $G_q = (V_q, E_q)$. The recurrent relationship is defined as follows. Let the hub score h_u be proportional to sum of referred authorities, let the authority score a_u be proportional to the sum of referring hubs.

$$\begin{aligned} \boldsymbol{a} &= E^T \boldsymbol{h} \\ \boldsymbol{h} &= E\boldsymbol{a}. \end{aligned}$$

The iterated method is therefore given by:

- initialize $\boldsymbol{a}$ and $\boldsymbol{h}$ (e.g., uniformly) ;
- repeat:
  - $\boldsymbol{h} \leftarrow E\boldsymbol{a}$;
  - $\boldsymbol{a} \leftarrow E^T\boldsymbol{h}$;
  - normalize $\boldsymbol{h}$ and $\boldsymbol{a}$.

The top-ranking authorities and hubs are reported to the user.

The principal eigenvector identifies the largest dense bipartite subgraph. To find smaller sets, the other eigenvectors must be explored. There are iterative methods that remove known eigenvectors from a system: they reduce the search subspace once an eigenvector is identified.

Although of theoretical interest, HITS is not commonly used by search engines, also because pre-computing hubness and authority values for different queries is not doable in practice, the algorithm has to run after the query is executed and this makes the algorithm very heavy for general-purpose usage. Coming back to the PageRank algorithms, let's note that it is independent of page content and therefore a suitable combination with the content, depending on the query, must be executed. Google's way of combining query and ranking is unknown. Probably, empirical parameters and manual inspection are necessary.

14.9 Clustering

The motivations for clustering are related to the huge number of documents retrieved by web searches. To avoid overloading the user, identifying groups of closely related documents is useful, for example to show only a small number of representative prototypes.

Automatically identified clusters can also be a help for later manual classification à la Yahoo. In addition, if a user is interested in document d, he is likely to be interested in documents in the same cluster and therefore pre-computed clusters permit to obtain more documents similar to the one under examination on demand.

Let's note that queries can be ambiguous, especially *web* queries. For example, if one searches for `star`, one may look for movie stars, or for celestial objects, clearly two very different topics.

Mutual similarities in term vector space can help grouping similar documents together, i.e., to find "clusters" of documents. Let D be the corpus of documents (or other entities) to be grouped together by similarity. Items $d \in D$ are characterized either *internally* by some intrinsic property (e.g., terms contained, coordinates in TF-IDF space) or *externally* by a measure of distance $\delta(d_1, d_2)$ or similarity $\rho(d_1, d_2)$ between pairs. Examples are: Euclidean distance, dot product, Jaccard coefficient. After defining the metric, the usual bottom-up or top-down clustering techniques can be used. The methods are explained in Chapter 17 and 18.

Gist

Some interesting applications involve more **structure** than simple "flat" vectors of measures, for example **relationships between entities modeled by graphs and networks**. In this case **probabilistic graphical models** like Bayesian networks or Markov networks and Inductive Logic Programming can be used to describe the initial knowledge, refine it based on examples, build symbolic (human) explanations usable for debugging the knowledge and for explaining how a certain conclusion has been reached.

Web and text-mining are highly-relevant application areas with a vast expanse of data, some of it structured, some partially structured or not at all. **Crawling and indexing** are systematic methods to visit web pages, harvest the information contained therein and prepare data structures for searching, information retrieval and ranking.

By transforming text into vectors of data (e.g., frequencies of selected words as in the **vector-space model**) some traditional ML techniques can be reused, but the richer amount of structure in web documents permits a more focused analysis.

Web-mining schemes find explicit relationships between documents (web links), infer implicit ones (by clustering), rank the most relevant pages in a network of connected sites or identify the most relevant and well-connected person in a network of people. Abstraction helps to use similar tools for networks of pages and networks of people. As a notable example, the use of hyperlinks and linear algebra tools (eigenvectors and eigenvalues), previously used to rank researchers in bibliometrics, leads to a very powerful technique to **rank web pages**, now at the basis of Google search-engine technology.

From now on, you will look at your hyperlinks, Facebook "Likes" and Twitter "Followers" (or at the social network software which will be the most popular when you read this book) with new analytic and aware eyes.

Chapter 15

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 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.

15.1 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. The demonstration is simple by measuring the area under the binomial distribution where more than $M/2$ models are wrong. Unfortunately errors tend to be *correlated* in practical cases. If a pattern is difficult to recognize, it will be difficult for *many* models, and the probability that many of them will be wrong will be higher than the product of individual mistake probabilities so that the advantage will be less dramatic. Think about stained digits in a zip code on a letter: the stain will create hard difficulties to *many* models and therefore correlate their mistakes.

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**. The "law of large numbers" in statistics explains why, under certain conditions, the average of the results obtained from a large number of trials tends to be close to the expected value, and why it tends to become closer as more trials are performed.

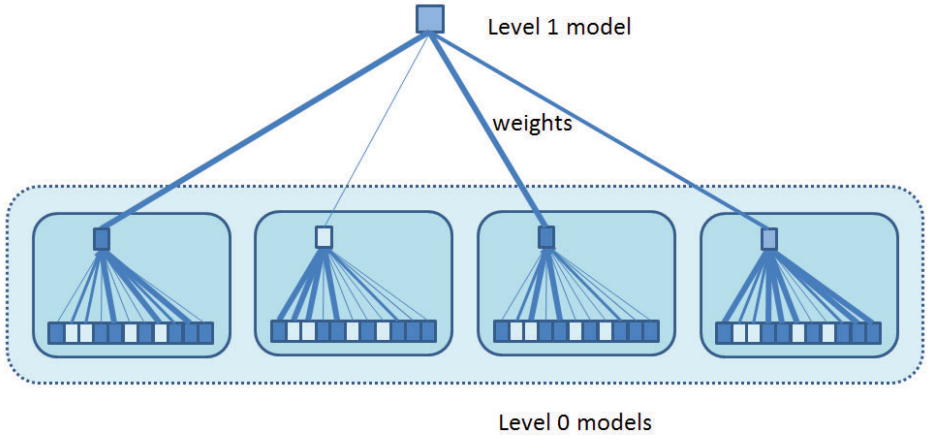


Figure 15.1: Blending different models by adding an additional model on top of them (stacking).

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. 15.1). This is the basic idea of **stacked generalization** [385]. 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 [359]:

- 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).

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. 15.2). 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.

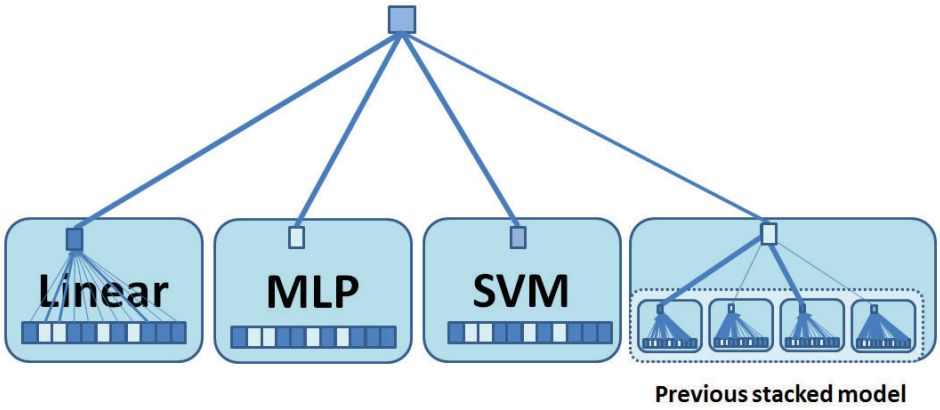


Figure 15.2: Stacking can be applied to different models, including previously stacked ones.

An interesting option is the **feature-weighted linear stacking** [332]. 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.

15.2 Diversity by manipulating examples: bagging and boosting

For 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 [116].

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 [132]. 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 iteration, weights are updated so that **more weight is given to the examples which are misclassified** by the current model (Fig. 15.3). 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.

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:

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

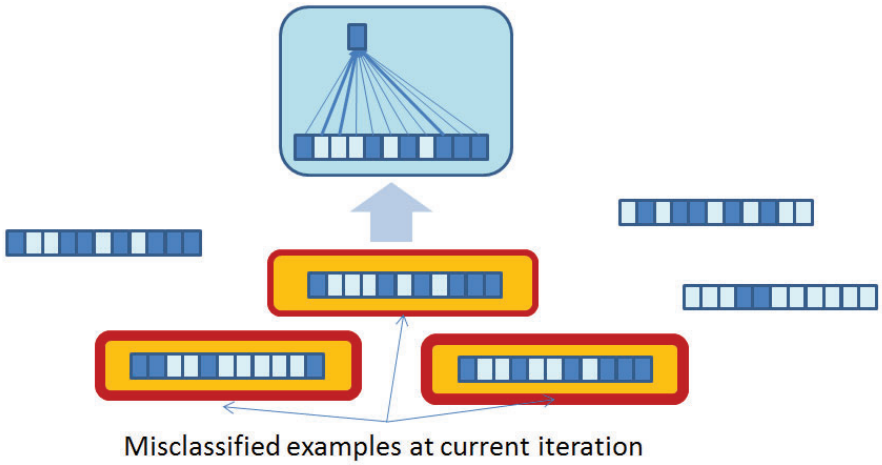


Figure 15.3: In boosting, misclassified examples at the current iteration are given more weight when training an additional model to be added to the committee.

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 15.6, `AdaBoost` can be viewed as a **stage-wise algorithm** for minimizing the following error function:

$$\sum_i \exp\left(-y_i \sum_l w_l h_l(\boldsymbol{x}_i)\right), \tag{15.1}$$

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

15.3 Diversity by manipulating features

Different **subsets of features** can be used to train different models (Fig. 15.4). In some cases, it can be useful to group features according to different characteristics. In [86] this 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.

15.4 Diversity by manipulating outputs: error-correcting codes

Error-correcting codes (ECC) are designed so that they are robust with respect to a certain number of mistakes during transmission by noisy lines (Fig. 15.5). For example, if the codeword for "one" is "`111`" and that for "zero"

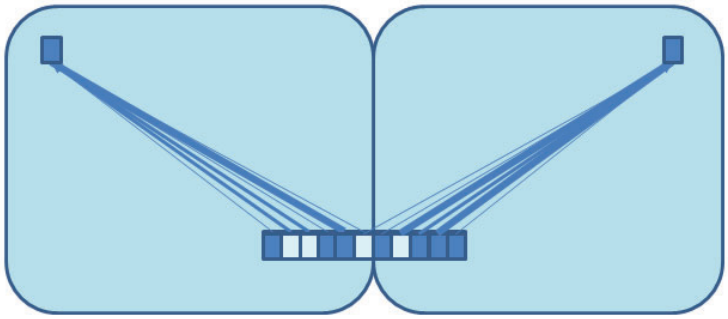


Figure 15.4: Using different subsets of features to create different models. The method is not limited to linear models.

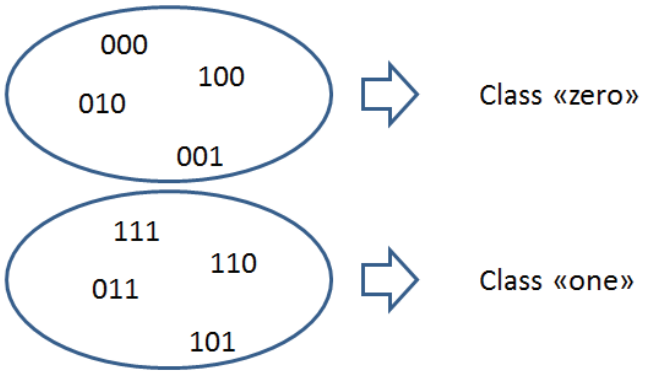


Figure 15.5: In error-correcting codes a redundant encoding is designed to resist a certain number of mistaken bits.

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 [117] 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.

15.5 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 randomized 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.

15.6 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** [133].

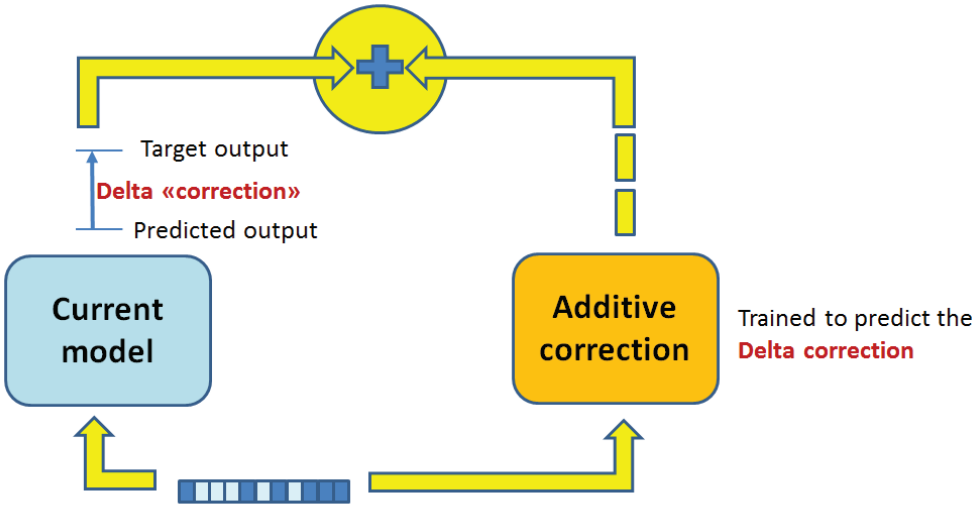


Figure 15.6: Additive model step: the error of the current model on the training examples is measured. A second model is added aiming at canceling the error.

Let's start from simple functions

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

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. 15.6). If least-squares

is used as a fitting criterion:

$$(\beta_m, \boldsymbol{\gamma}_m) = \arg\min_{(\beta,\boldsymbol{\gamma})} E\left[\left(y - F_{m-1}(\boldsymbol{x}) - \beta b(\boldsymbol{x};\boldsymbol{\gamma})\right)^2\right], \tag{15.2}$$

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. 15.7):

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

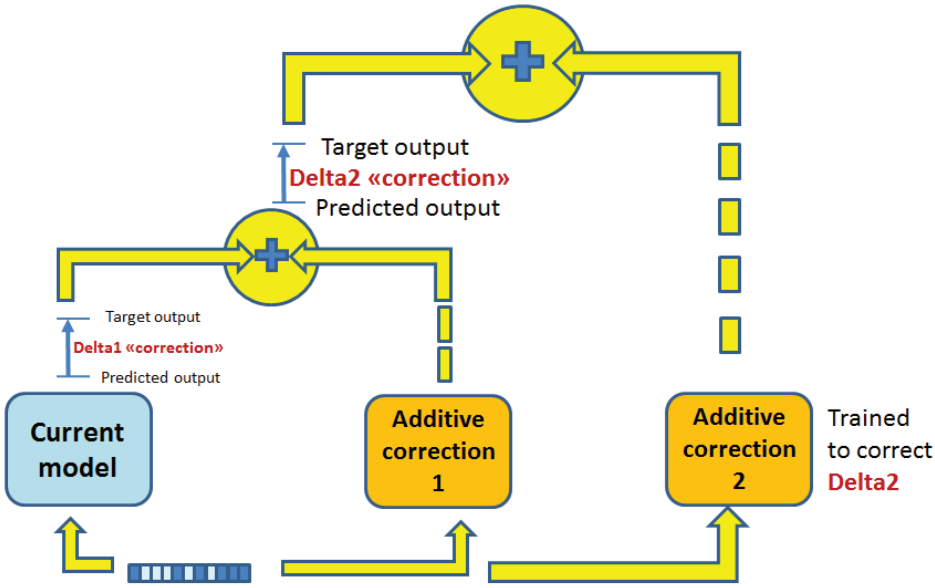


Figure 15.7: The greedy forward step-wise approach in additive models, one iterates by adding new components to cancel the remaining error.

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. 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 9.1): one uses the additive model $H(\boldsymbol{x})$ for 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

$$\Pr(y = 1|\boldsymbol{x}) = \frac{e^{H(\boldsymbol{x})}}{1 + e^{H(\boldsymbol{x})}}, \tag{15.3}$$

guarantees probability estimates in the correct $[0, 1]$ range. In fact, $H(\boldsymbol{x})$ is modeling the *input* of the logistic function in equation (15.3).

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[1] for minimizing $E\left[e^{-yH(\boldsymbol{x})}\right]$. The technical details and additional explorations are in the original paper [133].

15.7 Gradient boosting machines

Boosting is a very active research area, with a lot of flexibility, speed of realization and successful applications. [135] develops a **general gradient-descent boosting paradigm for additive expansion based on any fitting criterion**. Special enhancements are derived in particular for the case in which additive components are regression trees.

While we refer to the original publication for all theoretical details, we follow the explanation of [84] for the case of convex and differentiable loss functions. The objective is to obtain an **ensemble of k trees**, such that the predicted output $\hat{y}_i$ is given by the sum of the individual trees:

$$\hat{y}_i \ = \ \sum_{k=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F} \tag{15.4}$$

where $\mathcal{F}$ is the space of regression trees. Each tree partitions the input space into T zones ("leaves"), each zone is associated with a constant output value w, the function $q(\mathbf{x})$ maps an input values to a specific leaf.

$$\mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}, \quad q : R^m \rightarrow T, \quad \mathbf{w} \in R^T$$

To learn the model one minimizes a regularized objective function:

$$\mathcal{L} = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k) \tag{15.5}$$

where:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \|\mathbf{w}\|^2 \tag{15.6}$$

The loss function l measures the difference between the predicted $\hat{y}_i$ and teh target y_i, **differentiable and convex**. The term Ω penalizes overly-complex models, with too many leaves or very large weights, aiming at a better generalization.

Let's now consider how the addition of a tree can reduce the regularized objective. The **greedy** aspect corresponds to adding at iteration t the tree which brings the largest possible reduction, by minimizing:

$$\mathcal{L}^{(t)} = \sum_i \left[l(y_i, \hat{y}_i)^{(t-1)} + f_t(\mathbf{x}_i) \right] + \Omega(f_t) \tag{15.7}$$

The previous trees are left untouched. Instead of minimizing $\mathcal{L}^{(t)}$ one minimizes its second-order Taylor expansion, remembering that the delta in its input parameter is in this case the additional contribution $f_t(\mathbf{x}_i)$.

$$\mathcal{L}^{(t)} \approx \sum_i \left[l(y_i, \hat{y}_i)^{(t-1)} + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t(\mathbf{x}_i)^2 \right] + \Omega(f_t) \tag{15.8}$$

[1] 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.

where g_i and h_i are the first and second derivative of the loss function with respect to the predicted output: $g_i = \frac{\partial l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)}}$, $h_i = \frac{\partial^2 l(y_i, \hat{y}_i^{(t-1)})}{\partial \hat{y}_i^{(t-1)^2}}$. The derivatives have to be calculated at the current output value $\hat{y}_i^{(t-1)}$. Because we are minimizing, we can remove the constant term and we are left with:

$$\mathcal{L}^{(t)} = \sum_i \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t(\mathbf{x}_i)^2 \right] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2 \tag{15.9}$$

The tree model is simple, just *constant* values over separate regions. It makes sense to sum separately over the examples belonging to the different leaves (regions). Let's define $I_j = \{i | q(\mathbf{x}_i) = j\}$ as the set of examples with input values in leaf j. We can rewrite equation (15.9) as follows:

$$\mathcal{L}^{(t)} = \sum_{j=1}^{T} \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) {w_j}^2 \right] + \gamma T \tag{15.10}$$

If the tree structure $q(\mathbf{x})$ is fixed, the optimal weight ${w_j}^*$ is the one minimizing the parabola (let's remember that h_i are positive by assumption):

$${w_j}^* = \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{15.11}$$

and the objective function value decreases by:

$$\Delta\mathcal{L}^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{15.12}$$

Conclusion: for a fixed tree, with simple and fast algebra, we can calculate the optimal values for the weight in the leaves (in the Taylor approximation), and the corresponding decrease in the objective function $\Delta\mathcal{L}$. if the tree is not fixed, we can now see if a local modification leads to a better $\Delta\mathcal{L}$ value! A simple local modification the addition of an additional split, leading to an additional leaf. The best possible split can be chosen greedily as the one leading to the best possible $\Delta\mathcal{L}$ in equation (15.11).

A greedy algorithm starts from a single leaf and iteratively adds branches to the tree. Assume that I_L and I_R are the sets of examples ending up in the left and right nodes after the split. Letting $I = I_L \cup I_R$, the loss reduction after the split is given by the difference of the above reductions in $\mathcal{L}$

$$\mathcal{L}_{split} = \frac{1}{2} \left[\sum_{j=1}^{T} \frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma T \tag{15.13}$$

The meaning of γ is evident. The split must lead to a reduction in $\mathcal{L}$ larger than γ to be accepted. If γ gets larger, more and more "insignificant" splits are avoided.

The basic algorithm for constructing the tree is evident. Start from an initial root (all examples in the same leaf), evaluate a set of candidate splits (each split is defined by a variable and a threshold) with the "loss reduction" formula (15.12), pick the best split. Repeat the entire process for all leaves and possible splits until no reduction is possible.

Additional flexibility can be obtained by: (i) adding a shrinkage parameter [136]. Shrinkage scales the newly added weights by a factor η after each step of tree boosting, reduces the influence of each individual tree and leaves space for future trees to improve the model, (ii) using sub-sampling (of features and of examples) to increase the diversity of the individual trees, going in the direction of random forests (see Chapter 6.2), (iii) using heuristics to consider only a subset of all possible splits (to reduce CPU time) (iv) dealing in an expicit manner with sparsity and missing values [84]. The increase in the number of parameters and possible choices demands more automated ways for choosing the meta-parameters appropriate for a specific task, for example by cross-validation, a hot area for future research.

15.8 Democracy for better accuracy-rejection compromises

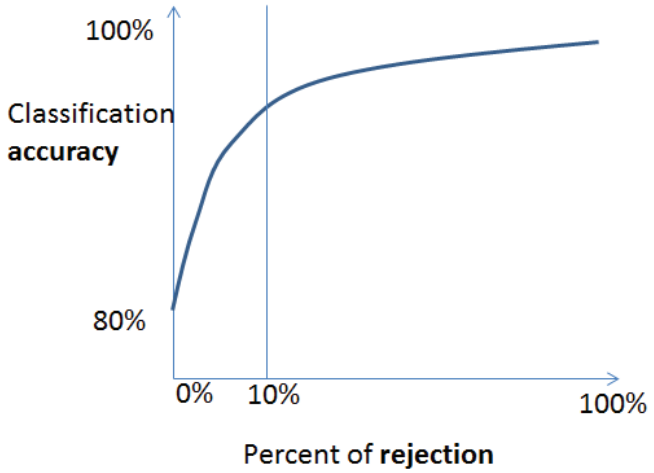


Figure 15.8: The accuracy-rejection compromise curve. Better accuracy can be obtained by rejecting some difficult cases.

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 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. 15.8, describing the attainable accuracy performance as a function of the rejection rate.

For historical reasons, a used term is **receiver operating characteristic (ROC)**. A **ROC curve** is a graphical plot that illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The **area under the curve (AUC) or AUROC** criterion can be used to evaluate different classifiers.

$$AUC = \int_{\infty}^{-\infty} \text{TPR}(T)\text{FPR}'(T)\, dT \tag{15.14}$$

where T is the varying threshold parameter. AUC is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one.

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. 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.

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).

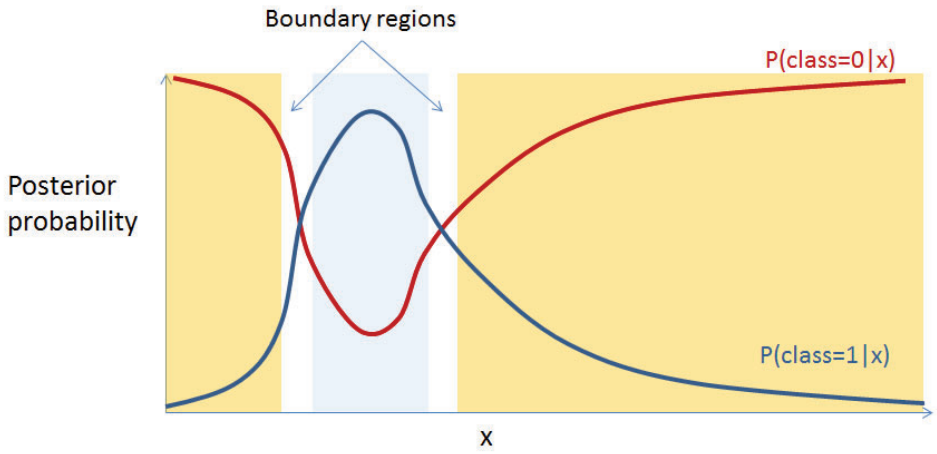


Figure 15.9: Transition regions in a Bayesian classifier. If the input patterns falling in the transition areas close to the boundaries are rejected, the average accuracy for the accepted cases increases.

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. 15.9).

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 [43]. 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, one 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.

Chapter 16

Recurrent networks and reservoir computing

Music is a reservoir... of sounds.
(Dexter Gordon)

A "pet hypothesis" which dominated neural networks and machine learning research for a long period is the idea that humans or computers spend huge efforts to extract building blocks (features) of growing complexity in order to solve complex tasks. Deep learning, supervised pre-training, stage-wise feature extraction are examples.

By negating the above hypothesis one obtains **reservoir learning**. The idea is to prepare a huge **reservoir of random features**, which are then tapped to build the final system, usually by a simple least-squares fit between the hidden outputs of the reservoir and the problem outputs. This sounds too quick and dirty to produce useful systems, but a growing amount of evidence shows that reservoir techniques are incredibly effective in many contexts. In some cases, they produce competitive results, or at least quick initial results which can be rapidly improved by an additional

tuning phase.

The **biological plausibility** of these techniques is probably higher than that of complex training mechanisms if one thinks about the rapidity of learning to ride a bicycle, to sing a song, to pronounce a new word. The fact that a handful of examples is sufficient to learn can be explained by the availability of **"random" building blocks**, with a proper architecture, ready to be tapped and rapidly fine-tuned.

16.1 Recurrent neural networks

Up to now we considered machine learning systems without a notion of time, history and memory. Better said, time and iterations played a role only *during* training, but not when the system is operational. During operation, the outputs depends only on the inputs, in a "feed-forward" one-shot manner, no cycle involved. Biological systems do not always work in this simple way. On the contrary, when singing a song, the output depends not only on the current input but also on the previous inputs, on the previous history. The same is true for playing music, speaking, heart beating, breathing, walking, etc. which involve cycles, oscillations and a rich dynamical behavior going beyond single-step input-output machines (which realize mathematical *functions*).

Artificial **recurrent neural networks** (**RNNs**) are distinguished from the more widely used feed-forward neural networks because of cycles in their connection topology. Some outputs are fed back to some nodes of the network, so that they influence also future outputs, providing the network with some *memory* of the past. Depending on the model, the computation can proceed via synchronized steps (think about a global clock ticking to make each unit collect and process the current inputs to produce a new output), or in asynchronous mode (each unit wakes up at random times and updates its output), or via continuous dynamics regulated mathematically by differential equations. Figure 16.1 shows the basic structure of a recurrent network: hidden unit outputs can be fed back as inputs; outputs are fed back to hidden units and to output units. This latter requirement is important if subsequent outputs are strongly correlated and the network can better operate incrementally; depending on the case, some of these feedback channels may not be implemented.

Let's present a concrete example to develop some intuition. A recurrent network with no inputs, four hidden sigmoid units and two linear output units was trained to follow a circle (centered on the origin, radius 1, three turns of 16 steps each, initial transient of 4 steps). After a short training phase, the system follows an approximated circular trajectory shown in Fig. 16.2 The corresponding values for the four hidden units are shown in Fig. 16.3.

The existence of cycles has a profound impact (a recent review is [256]):

- A RNN can develop a self-sustained temporal activation dynamics along its recurrent connection pathways, even in the absence of input. A RNN is a **dynamical system**, while feedforward networks are functions.

- If driven by an input signal, a RNN preserves in its internal state a nonlinear transformation of the input history. It has a dynamical memory, and is able to process temporal context information.

From a dynamical systems perspective, there are two main classes of RNNs. The first class is characterized by an energy-minimizing stochastic dynamics and **symmetric connections** (the trajectory of the network outputs finds local minima of a suitable "energy" function, think about a kind of modified gradient descent). Known example are **Hopfield networks** derived from statistical physics [189], Boltzmann machines [6], and Deep Belief Networks [181]. These systems are mostly trained with **unsupervised learning**. Typical targeted functionalities in this field are *associative memories* (the retrieved memory correspond to local minima of the energy function), data compression, the unsupervised modeling of data distributions, and static pattern classification. The model is run for multiple time steps per single input instance to reach some type of convergence or equilibrium.

The second big class of RNN models typically features a deterministic update dynamics and **directed connections**. Systems from this class implement nonlinear filters, which transform an input time series into an output time series. The mathematical background consists of nonlinear dynamical systems. The standard training mode is **supervised**.

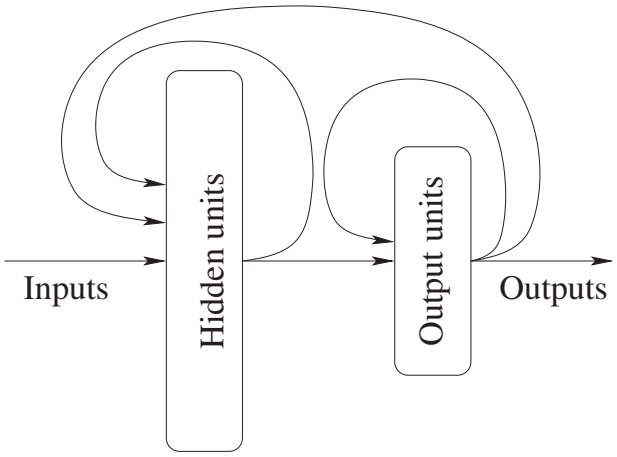


Figure 16.1: The basic scheme of a recurrent network.

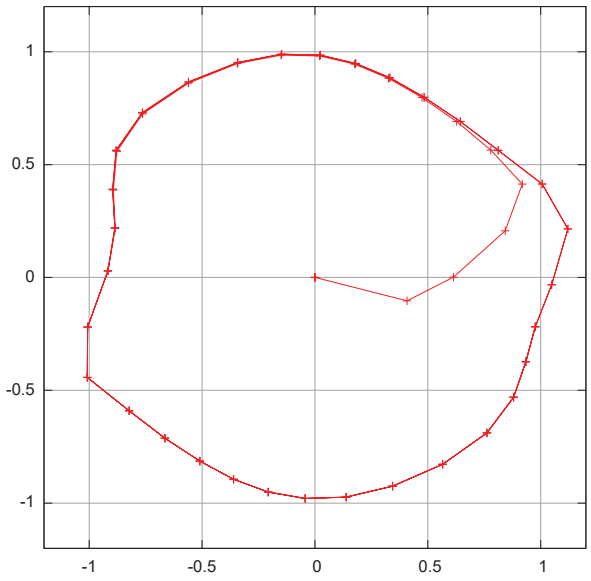


Figure 16.2: Recurrent network trained on a circular trajectory: output sequence starting from null inputs.

16.2 Energy-minimizing Hopfield networks

A **Hopfield network** (Fig.16.4), defined in [189], consists of binary threshold units, i.e., they only take on two different output values (normally 1 or -1), depending on whether the input exceeds a threshold or not. Symmetric weights w_{ij} connect pairs of units (with no self-connection: $w_{ii} = 0$). A unit is updated as follows:

$$s_i \leftarrow \begin{cases} +1 & \text{if } \sum_j w_{ij} s_j \geq \theta_i \\ -1 & \text{otherwise,} \end{cases} \tag{16.1}$$

where s_i is the output state of unit i and θ_i is the threshold. Updates can be performed **asynchronously** (a unit is picked at random and updated) or **synchronously** (all units are updated at the same time at the tick of a central clock). Asynchronous updates have a more biological or physical flavor (*spin glasses* are a related model in physics). The initial output values are progressively changed by the update rule so that the state depends on the initial condition but also on the sequence of updates. If an output is signalled by a flashing (+1) or inactive (-1) LED —actually this was a homework hardware realization by one of the authors— one observes a flickering pattern in time. The main question is: what is the possible *meaning* of this flickering pattern, which kind of computation can be executed? The answer by Hopfield is related to **optimizing a suitable energy function** (in math and physics called a *Lyapunov function*) :

$$E = -\frac{1}{2} \sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

One can easily demonstrate that, when units are chosen to be updated, with symmetric connections, the **energy E will either decrease or remain equal**. Under repeated updating the network will eventually converge to a state which is a local minimum in the energy function. Local minima in the energy function are **stable states** for the network. The flickering pattern will stabilize and show a stable pattern of light. The "meaning" of Hopfield networks is explained by a dynamical system which, when started from an initial input, **seeks out local minima in the attraction basin of the initial point**, like a drop of water flowing to a lake in a drainage basin. The facts that outputs are constrained in a box is crucial. If not, one is minimizing a quadratic form which could be unlimited (going to minus infinity).

Programming a Hopfield net amounts to **carving out appropriate local minima in the energy landscape**. The Hebbian learning rule for modifying weights during training was introduced by Donald Hebb in 1949 to explain "associative learning." The simultaneous activation of neuron cells leads to pronounced increases in synaptic strength between those cells: "Neurons that fire together, wire together. Neurons that fire out of sync, fail to link." The Hebbian rule is local and incremental. For the Hopfield Networks, it is implemented in the following manner, when learning N binary patterns:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{N} x_i^\mu x_j^\mu, \qquad i, j = 1, \ldots, n$$

where each pattern $\boldsymbol{x}^\mu = (x_1^\mu, \ldots, x_n^\mu)$ is an n-bit sequence, and n is also the number of neurons in the network. If the bits corresponding to neurons i and j are equal in pattern $\boldsymbol{x}^\mu$, then the product $x_i^\mu x_j^\mu$ will have a positive effect on the weight w_{ij} and the values of neurons i and j will tend to become equal. The opposite happens if the bits corresponding to neurons i and j are different.

Depending on the number of nodes in the network, the Hebbian rule will be able to "carve" in the energy landscape a set of local minima that are close to the N patterns used to train it, provided that N is not too large (a rule of thumb is that the number N of patterns should not exceed about 13.8% of the number n of neurons [177]). When the update rule (16.1) is repeatedly applied by first assigning a specific pattern $\boldsymbol{x}^\nu$ to the neurons, the network will settle to a local minimum, thereby retrieving the stored pattern that is closest to $\boldsymbol{x}^\nu$, as shown in Fig. 16.5.

With Hopfield nets one can build **content-addressable memory** systems: the network will converge to a "remembered" state if it is given only part of the content (with other bits set randomly). The net can be used to recover from a distorted input the trained state that is most similar to that input. This is called **associative memory**, related to

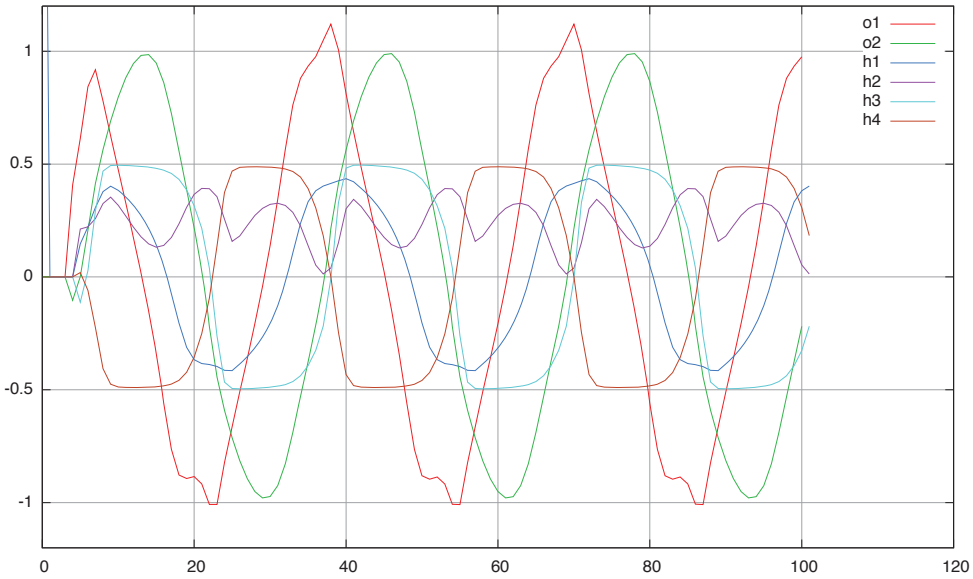


Figure 16.3: Recurrent network trained on a circular trajectory: outputs and hidden units.

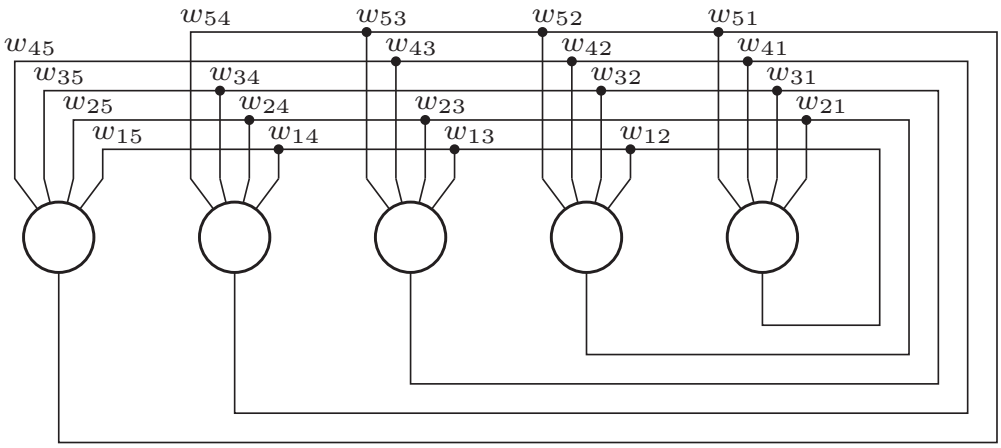


Figure 16.4: A Hopfield network with 5 neurons and feedback loops.

error-correcting codes. Generalizations with continuous weights have dynamics similar to gradient descent, with the descent direction modified with respect to the gradient (but still descending).

Although of formidable theoretical and scientific interest, real-world applications of Hopfield networks face some difficulties related to *spurious patterns*, i.e., local minima that do not correspond to programmed memories, and to *capacity limits*. When too many patterns are stored, one stored item can be confused with another upon retrieval. Note that human memory has similar characteristics, and semantically related items tend to confuse the individual and lead to the recollection of wrong patterns.

16.3 RNN and backpropagation through time

Let's now consider more general RNNs, without the requirement of symmetric weights and binary outputs (Fig. 16.1).

This kind of network can be simulated with a feed-forward network by **unrolling** it as shown in Fig. 16.6. The only change needed to a standard feed-forward MLP is to allow some of the inputs to be fed directly to the output neurons, in addition to the hidden layer(s). The standard training method is called "**backpropagation through time**" or BPTT, in which standard back-propagation for feed-forward networks is applied to the above unrolled model [379]. Recurrent neural networks, with output of some units fed back to other units, are difficult to train with derivative-based techniques because of the **vanishing and exploding gradient problems** [49]. Exploding gradients refers to the large increase in the norm of the gradient during training, caused by the explosion of the long-term components, which can grow exponentially more than short-term ones. The vanishing gradients problem refers to the opposite behavior, when long-term components go exponentially fast to norm 0, making it impossible for the model to learn correlation between temporally distant events. The problems are caused by the large number of iterations which can be present in RNN, which cause a small weight change to be exponentially increased or decreased. Modifications to cure the above problems (gradient norm clipping to deal with exploding gradients and soft constraints for the vanishing gradients problem) are presented in [293].

In a way, RNNs stress the limits of derivative-based techniques for optimization and motivate the adoption of derivative-free techniques or of radical changes in the training architecture like reservoir learning and extreme learning machines considered in the following sections.

16.4 Reservoir learning for recurrent neural networks

RNNs (of the second type that we introduced, i.e., without symmetry constraints) are highly promising tools for non-linear time series applications [256]. They are biologically plausible (recurrent connection pathways are present in brains) universal approximators of dynamical systems, under fairly mild and general assumptions.

A number of training algorithms proposed in the past suffer from the following shortcomings:

- The gradual change of network parameters during learning drives the network dynamics through **bifurcations**: the gradient information degenerates and may become ill-defined. Therefore, convergence cannot be guaranteed.
- Many costly update cycles may be necessary, resulting in excessive training times for large networks (with more than tens of units).
- Dependencies requiring long-range memory are hard to learn, because the necessary gradient information exponentially dissolves over time.
- Advanced training algorithms are complex with many global control parameters. They need skill and experience to be used.

In the current century a radically new approach is being proposed independently under the name of *Liquid State Machines* [258] and *Echo State Networks* [210], here collectively referred to as **Reservoir Computing** (RC). RC avoids the shortcomings of gradient-descent RNN by the following prescription (Fig.16.7):

- A recurrent network is **randomly created** and remains unchanged during training. This RNN is called the **reservoir**. It is passively excited by the input signal and maintains in its state a nonlinear transformation of the input history.
- The desired output signal is generated as a **linear function** of the neuron's signals from the input- excited reservoir. This linear combination is obtained by linear regression, for example by using least-squares.

Reservoir Computing is rapidly becoming one of the basic tools for RNN modeling, showing better modeling accuracy, universal modeling capacity for continuous-time, continuous-value real-time systems. RC can explains why biological brains can carry out accurate computations in spite of noisy physical components. Last but not least, models can be extended by adding more output units to the same reservoir, without interfering with the previous functionality.

In some cases, a completely random reservoir is not sufficient and the current research deals with developing suitable reservoir design and adaptation methods. RC is departing from a brute-force random approach and becoming a paradigm for using different methods for (i) producing/adapting the reservoir, and (ii) training different types of readouts. An updated review is presented in [256].

16.5 Extreme learning machines

A separate but related stream of research considers feed-forward systems, like multi-layer perceptrons, in which the initial layers are created randomly and only the final layer is trained by linear regression. **Extreme Learning Machines** are proposed in [194] by adopting the standard pseudo-inverse technique for least-squares fitting described in Section 4.1. Both ELM and RC overcome the problems associated with traditional neural network training algorithms, such as local minima and vanishing or exploding gradients, by **modifying only the output weights** using a simple and efficient linear regression algorithm. The main difference between the two approaches is that the reservoir of RC architectures contains recurrent connections, giving it a short-term memory, whereas ELMs are a pure feedforward architecture without short-term memory.

The "surprising" fact that useful learning occurs in MLP even if the initial layers are random has been observed starting from the initial developments in neural networks. For example, Rosenblatt [308] and other early investigators favored randomly chosen input feature detectors. E. Baum [44] claims that in some simulations one may fix the weights of the connections on one level and simply adjust the connections on the other level, with no significant gain by adjusting the weights on both levels simultaneously. More recently, an extensive theoretical and practical investigation is presented in [195], which coined the term "**Extreme Learning Machine**."

It is well known from linear algebra that, through matrix inversion, single-layer feed-forward networks (SLFNs) with N hidden nodes and randomly chosen input weights and hidden layer biases can exactly learn N distinct observations, under conditions of full matrix rank (linear independence). Of course, generalization is not guaranteed, but learning becomes a trivial one-shot operation by matrix inversion.

The work in [195] rigorously proves that the input weights and hidden layer biases of SLFNs can be randomly assigned if the activation functions in the hidden layer are infinitely differentiable. After they are chosen randomly, SLFNs can be considered as a linear system and the output weights can be analytically determined through simple generalized inverse (pseudo-inverse) operation of the hidden layer output matrices. Various generalizations considering different kinds of hidden nodes and architectures have been proposed (Fig. 16.8). In some cases, ELMs can learn much faster than traditional algorithms like back-propagation (BP), while obtaining better generalization performance, in particular if the norm of weights is controlled through the usual quadratic penalty. The number of hidden neurons leading to optimal performance can be much bigger that that for backpropagation learning, an indication that a large pool of random units needs to be created to tap a sufficient number of useful units to determine the outputs. A recent review is [193].

Related investigations are [214], which evaluates multi-stage architectures for object recognition and considers also random filters, and [382], which proposes the *no-prop* method (same as ELM, but with iterative least-squares techniques). Architecture with random weights are studied in [314]. They show that certain convolutional pooling architectures (sharing connection weights at different spatial positions in image processing) can be inherently frequency

selective and translation invariant, even with random weights. Based on this they propose **using random weights to evaluate candidate architectures**, thereby sidestepping the time-consuming learning process. A surprising fraction of the performance of certain state-of-the-art methods can be attributed to the architecture alone, although subsequent fine-tuning usually does improve the final performance.

It is clear that Reservoir Computing and ELM follow similar research directions, although originally RC concentrated mostly on RNN, ELM on feed-forward systems. Reservoir computing and extreme learning are considered jointly in [73].

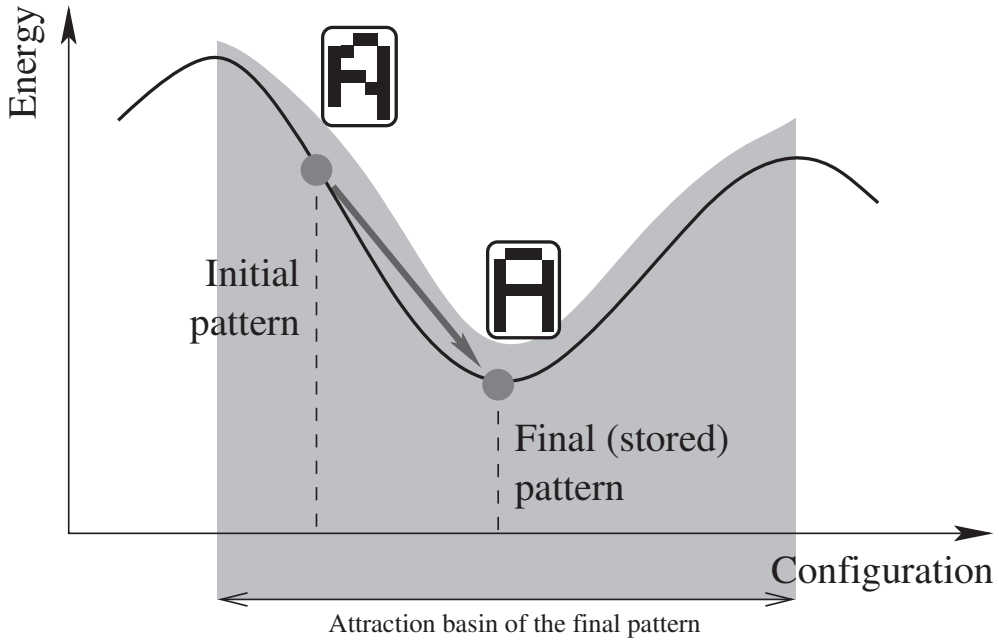


Figure 16.5: Energy Landscape of a Hopfield Network, highlighting the initial state of the network (up the hill), an attractor state to which it will eventually converge, and a basin of attraction (shaded).

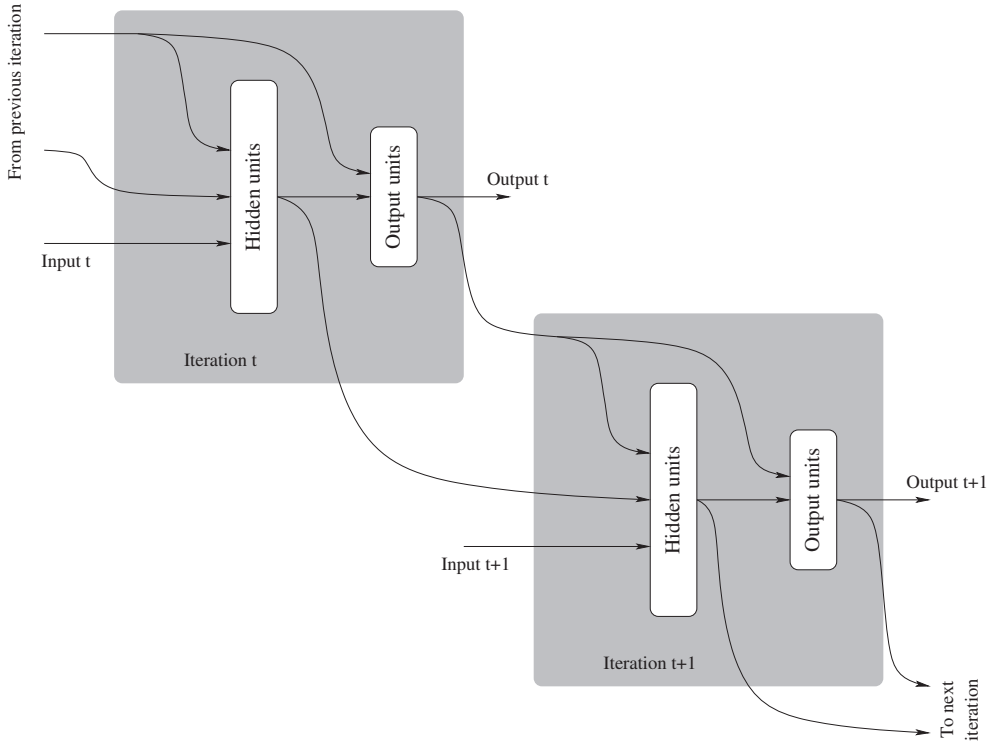


Figure 16.6: The basic recurrent network of Fig. 16.1 can be unrolled as a cascade of feed-forward networks, each fed by an iteration's inputs and by the previous iteration's hidden and output values.

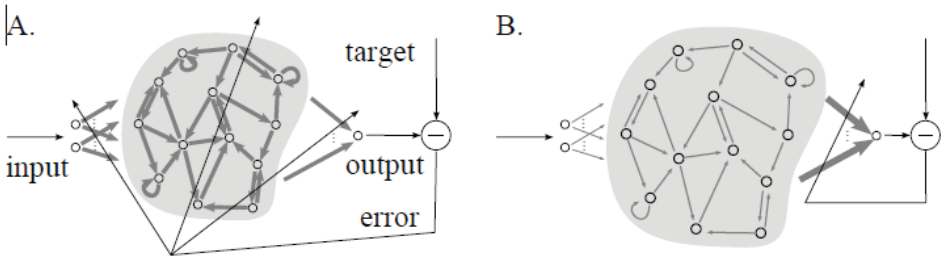


Figure 16.7: (Left) Traditional gradient-descent-based RNN training methods adapt all connection weights (bold arrows), (Right) In Reservoir Computing, only the RNN-to-output weights are modified (adapted from [256]) .

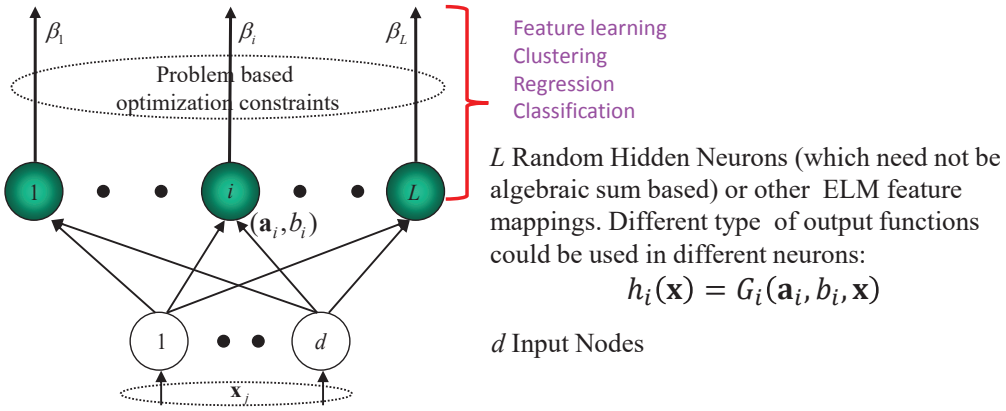


Figure 16.8: The hidden nodes in ELM can consist of different type of computational nodes (adapted from [193]).

Gist

Recurrent neural networks with feedback loops allow the transition from "mathematical functions" (feed-forward networks) to full-fledged **dynamical systems** with evolution in time and internal memory.

Machine learning for recurrent neural networks is very hard, in particular for derivative-based methods. The many cycles involved can cause derivatives to explode or to vanish.

The recently proposed **reservoir computing** (RC) and **extreme learning machines** (ELM) methods take a radical approach, in a way contrary to that of deep learning, by creating vast amounts of **random building blocks** (random features), and limiting learning to a final linear combination layer. A specific problem "taps" the useful building blocks in the reservoir and weighs them appropriately to get the final solution.

Given the difficulties of realizing deep derivative-based techniques with noisy biological neural hardware, the success of this brute-force "randomized construction plus final tuning" approach gives new hope to explain parts of our brain and to engineer fast and flexible learning machines.

We are happy to live in a sparkling research period, when crazy and wildly different ideas advance the frontier of ML and neural networks through spectacular plot twists and paradigm changes.

Part II

Unsupervised learning and clustering

Chapter 17

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 learned 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[1].

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. 17.1). 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 cannot 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 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.

As you imagine, the practical applications of clustering are endless. To mention some examples, in **market segmentation** one divides a broad marketplace into parts, or segments, and then implements strategies to target the common needs and desires of the segmented customers. In **finance**, clustering groups stocks with a similar behavior, for diversifying the portfolio and reducing risk. In **health care**, diseases are clusters of abnormal conditions that affect our body. In **text mining**, different words are grouped together based on the structure and meaning of the analyzed texts. A **semantic network** represents relations between concepts. It is a directed or undirected graph consisting of vertices, which represent concepts, and labeled edges, which represent relations. The different relations (e.g., "is an", "has", "lives in", ...) underline that there is no single way to group entities.

[1]To be honest, the radical simplification implied by giving names takes mysticism away from the world, a price to pay for conquering it with our technical means. Rainer Maria Rilke expresses this concept in his "In Celebration of Me" (1909):

> I am so afraid of people's words.
> They describe so distinctly everything:
> And this they call dog and that they call house,
> here the start and there the end.
> ...
> I want to warn and object: Let the things be!
> I enjoy listening to the sound they are making.
> But you always touch: and they hush and stand still.
> That's how you kill.

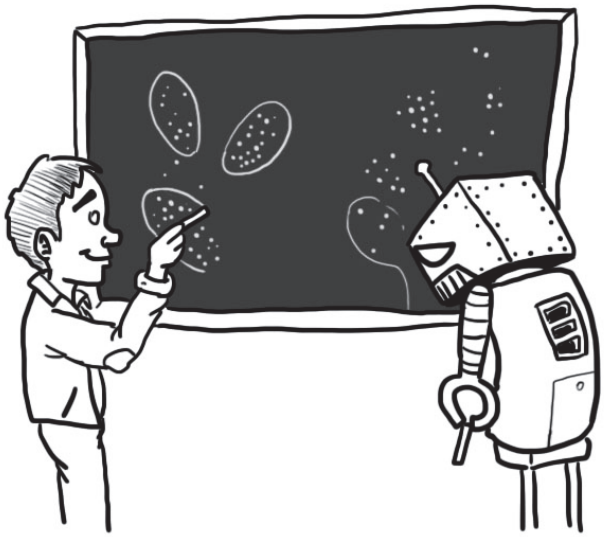


Figure 17.1: Clustering is deeply rooted into the human activity of grouping and naming entities.

17.1 Approaches for unsupervised learning

Given the creativity and different objectives of clustering, there are 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 are familiar from natural sciences, think about the organization of zoological or botanical species.

More advanced and flexible unsupervised strategies are known under the umbrella term 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 an underlying **probabilistic process**. By modeling it one understands the structure and the different clusters. **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

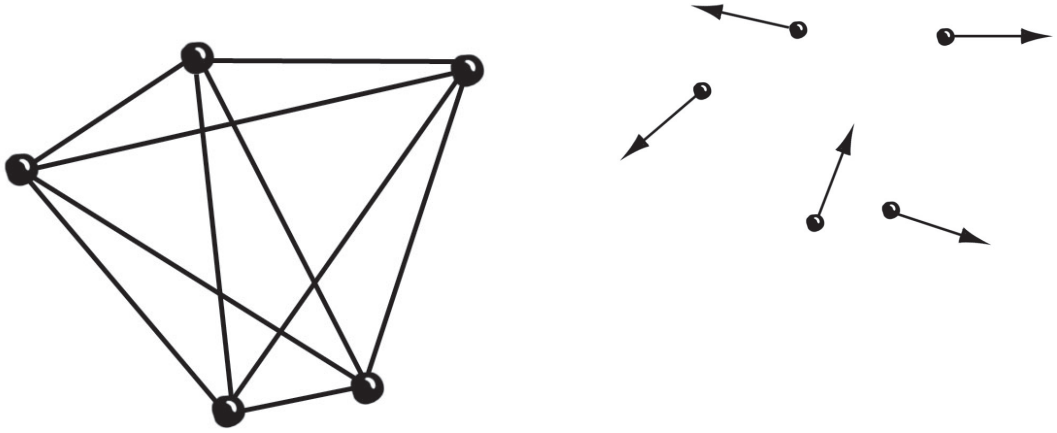


Figure 17.2: External representation by relationships (left) and internal representation with coordinates (right). In the first case mutual similarities between pairs are given, in the second case individual vectors.

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**.

17.2 Clustering: Representation and metric

There are two different contexts in clustering, depending on how the entities to be clustered are organized (Fig. 17.2). 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…), caloric 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**:

$$\delta_E(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| = \sqrt{\sum_{i=1}^{M} (x_i - y_i)^2}. \tag{17.1}$$

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:

$$\delta_{\text{Manhattan}}(\mathbf{x}, \mathbf{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_1 = \sum_{i=1}^{M} |x_i - y_i|. \tag{17.2}$$

As usual, there is no absolutely right or wrong norm. 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 based on 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**:

$$\text{cosine-similarity}(\mathbf{x}, \mathbf{y}) = \cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{\sum_{i=1}^{M} x_i \, y_i}{\sqrt{\sum_{i=1}^{M} (x_i)^2} \sqrt{\sum_{i=1}^{M} (y_i)^2}}, \tag{17.3}$$

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:

$$\delta_{norm}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{M} \left(\frac{x_i - y_i}{\text{maxval}_i - \text{minval}_i} \right)^2}, \tag{17.4}$$

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.

17.3 K-means for hard and soft clustering

Hard clustering partitions 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:

$$\min \sum_{d_1, d_2 \in C_i} \delta(\mathbf{x}_{d_1}, \mathbf{x}_{d_2}). \tag{17.5}$$

  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:

$$\min \sum_{d \in C_i} \delta(\mathbf{x}_d, \mathbf{p}_i). \tag{17.6}$$

- Maximization of inter-cluster distance. One wants the different clusters to be clearly separated.

As anticipated, the two 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 represent groups of entities by a single vector, one can select the prototypes which minimize the average **quantization error**, the error incurred when the entities are substituted with their prototypes:

$$\text{Quantization Error} = \sum_d \|\mathbf{x}_d - \mathbf{p}_{c(d)}\|^2, \tag{17.7}$$

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. 17.3, consists of the following steps.

1. Choose the number of clusters k.

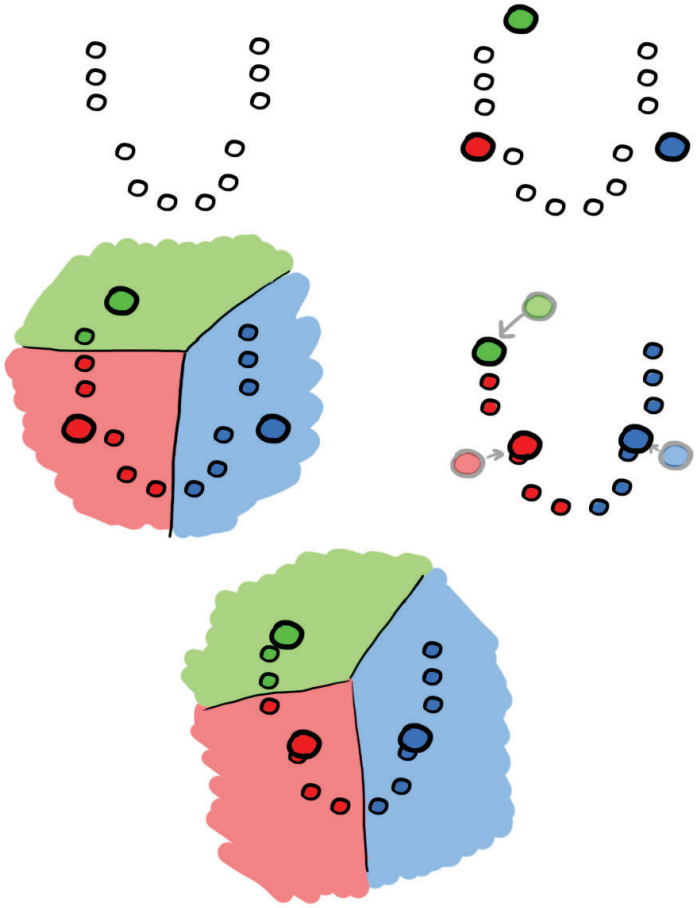


Figure 17.3: K-means algorithm in action (from top to bottom, left to right). Initial centroids are placed. Space is subdivided into portions close to the centroids (Voronoi diagram: each portion contains the points which have the given centroid as closest prototype). New centroids are calculated. A new space subdivision is obtained.

2. Randomly generate k clusters and determine the cluster centroids $\mathbf{p}_c$, or pick k randomly chosen training examples as cluster centroids. In other words, the initial centroid positions can be chosen randomly among 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:

$$\mathbf{p}_c \leftarrow \frac{\sum_{\text{entities in cluster } c} \mathbf{x}}{\text{number of entities in cluster } c}.$$

The main advantages of this algorithm are simplicity and speed, it can be used also on large datasets. **K-means clustering** can be seen as a skeletal version of the **Expectation-Maximization** algorithm[2]: if the assignment of the

[2] In statistics, an expectation-maximization (EM) algorithm is a method for finding maximum likelihood or maximum a posteriori (MAP)

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 cluster 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. 17.3.

Up to now we considered hard clustering, where the assignment of entities to clusters is crisp. In some cases a softer approach is more appropriate, in which **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:

$$\text{membership}(\mathbf{x}, c) = \frac{e^{-\delta(\mathbf{x},\mathbf{p}_c)}}{\sum_c e^{-\delta(\mathbf{x},\mathbf{p}_c)}}. \tag{17.8}$$

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 $\mathbf{x}$:

$$\Delta\mathbf{p}_c = \eta \cdot \text{membership}(\mathbf{x}, c) \cdot (\mathbf{x} - \mathbf{p}_c); \tag{17.9}$$

$$\mathbf{p}_c \leftarrow \mathbf{p}_c + \Delta\mathbf{p}_c. \tag{17.10}$$

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:

$$\mathbf{p}_c \leftarrow \mathbf{p}_c + \Delta_{total}\mathbf{p}_c. \tag{17.11}$$

When the parameter η is small, the two updates tend to produce very similar results, when η grows 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.

The k-means results can be visualized by a scatterplot display, as in Fig. 17.4. 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.

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.

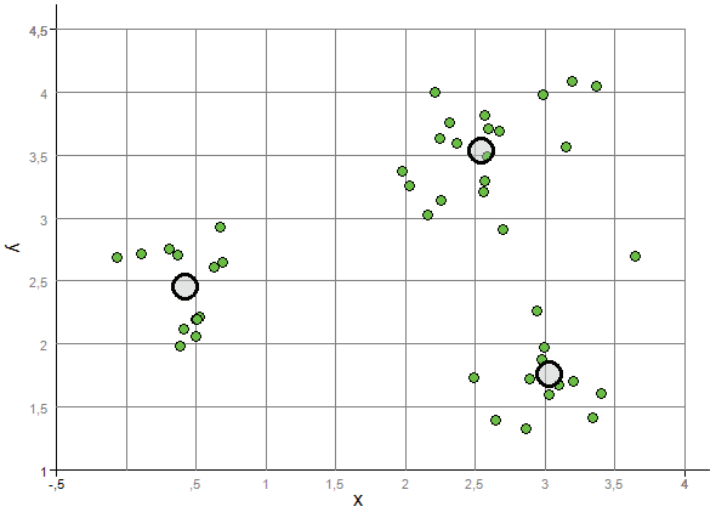


Figure 17.4: K-means clustering. Individual points and cluster prototypes are shown.

Gist

Unsupervised learning deals with building models using only input data, without resorting to classification labels. In particular, clustering aims at grouping similar cases together, 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 favorite little pet.

Chapter 18

Bottom-up (agglomerative) clustering

Birds of a feather flock together.
(Proverb: Those of similar taste congregate in groups)

Clustering methods require setting many parameters, such as the appropriate number of clusters in K-means, as explained in Chapter 17. 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. 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 groups are merged.

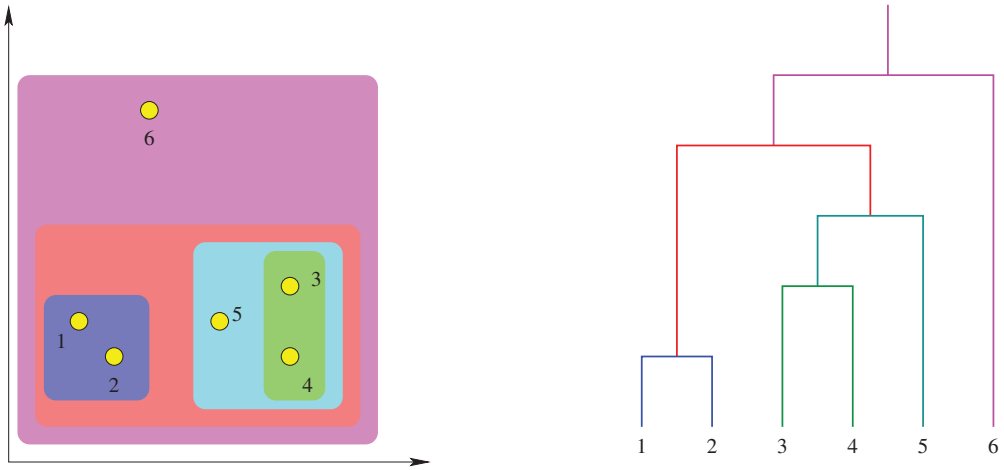


Figure 18.1: A dendrogram obtained by bottom-up clustering of points in two dimensions (with the standard Euclidean distance). Each point is an entity described by two numeric values.

18.1 Merging criteria and dendrograms

Let $\mathcal{C}$ represent the current clustering as a collection of subsets of entities, the individual clusters C. Thus $\mathcal{C}$ defines a partition: each entity belongs to one and only one cluster. Initially $\mathcal{C}$ consists of singleton groups, each with one entity.

As in top-down clustering, bottom-up merging also needs a measure of distance to guide the 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. 18.1-18.2.

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

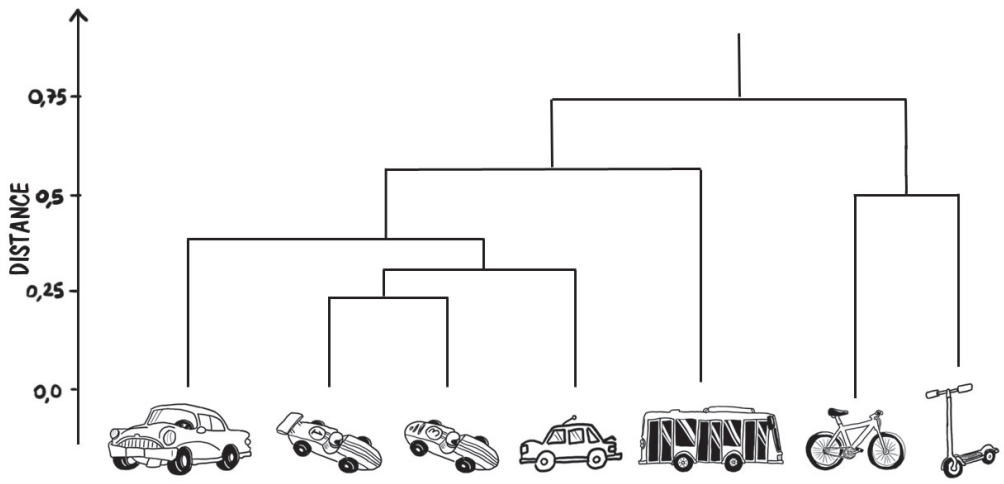


Figure 18.2: A dendrogram obtained by bottom-up clustering of vehicles.

ruler over the dendrogram, from the bottom to the top of the plot. 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.

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 the dendrogram, that can be a possible candidate level to cut horizontally and identify "natural" clusters.

18.2 A distance adapted to the distribution of points: Mahalanobis

The **Mahalanobis distance** was prompted by the problem of identifying the similarities of human 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 summarized 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.

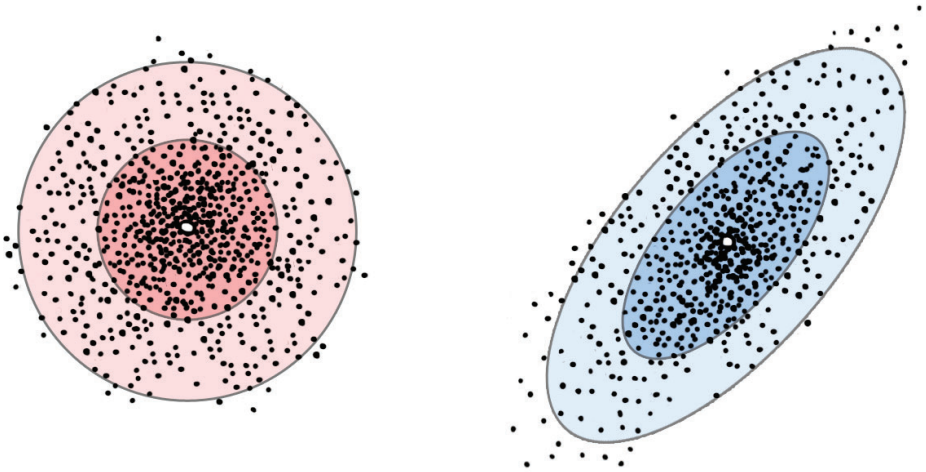


Figure 18.3: In the case on the left we can use the Euclidean distance as a dissimilarity measure, while in the other case we need to refer to the Mahalanobis distance, because the data are distributed in an ellipsoidal shape.

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 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:

$$\bar{\boldsymbol{p}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i. \tag{18.1}$$

Let the covariance matrix components are defined as:

$$S_{ij} = \frac{1}{n}\sum_{k=1}^{n}(p_{ki} - \bar{p}_i)(p_{kj} - \bar{p}_j), \qquad i, j = 1, .., D. \tag{18.2}$$

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. 18.3 illustrates the concept. In detail, the **Mahalanobis distance** of a vector $\boldsymbol{x}$ from a set of values with mean $\boldsymbol{\mu}$ and covariance matrix $\mathbf{S}$ is defined as:

$$D_M(\boldsymbol{x}) = \sqrt{(\boldsymbol{x} - \boldsymbol{\mu})^T \mathbf{S}^{-1} (\boldsymbol{x} - \boldsymbol{\mu})}. \tag{18.3}$$

The Mahalanobis distance can also be defined as a dissimilarity measure between two random vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of the same distribution with the covariance matrix $\mathbf{S}$:

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^T \mathbf{S}^{-1} (\boldsymbol{x} - \boldsymbol{y})}. \tag{18.4}$$

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**:

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{D} \frac{(x_i - y_i)^2}{\sigma_i^2}}, \tag{18.5}$$

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 through ellipsoids characterized by a fixed Mahalanobis distance from the barycenter, we are ready to understand the way in which clusters can be visualized.

18.3 Visualization of clustering and parallel coordinates

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:

$$T_C = \begin{pmatrix} S_{11} & S_{12} & S_{13} & 0 \\ S_{21} & S_{22} & S_{23} & 0 \\ S_{31} & S_{32} & S_{33} & 0 \\ \bar{p}_1 & \bar{p}_2 & \bar{p}_3 & 1 \end{pmatrix}. \tag{18.6}$$

When moving between hierarchical clustering levels, cluster C will split into several clusters $C_1, \ldots, C_l$. To preserve the proper mental image, a parametric transition from ellipsoid T_C to its l offspring $T_{C_1}, \ldots, T_{C_l}$ will be animated and the ellipsoids

$$T_{C_i}^{\lambda} = (1 - \lambda) T_C + \lambda T_{C_i}, \qquad i = 1, \ldots, l$$

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.

Fig. 18.4 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.

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 and direct tool to use for visualizing clusters of large-dimensional data is the **parallel coordinates** display. An example for the three clusters of the original Fisher Iris data is shown in Fig. 18.5. In a parallel coordinates plot, each vertical axe corresponds to a coordinate. A point in n-dimensional space is represented as a *polyline* (a line composed of attached segments) with vertices on the parallel axes. The position of the vertex on the i-th axis corresponds to the i-th coordinate of the point. By acting on filters, the number of points displayed can be reduced to concentrate on the most interesting ones. Furthermore, different ordering of the axes, colored lines, highlighting, background colors, etc. can add useful information to the plot. At a glance, both the individual items and the overall structure of clusters (or subset of items) is visible.

The parallel coordinates plot is probably the least known but simplest and most effective way to visualize n-dimensional points, as soon as n is larger than the two or three dimensions we can observe directly with our eyes. You do not need to be an engineer (the enlightened caste which is already using them) to use parallel coordinates plots.

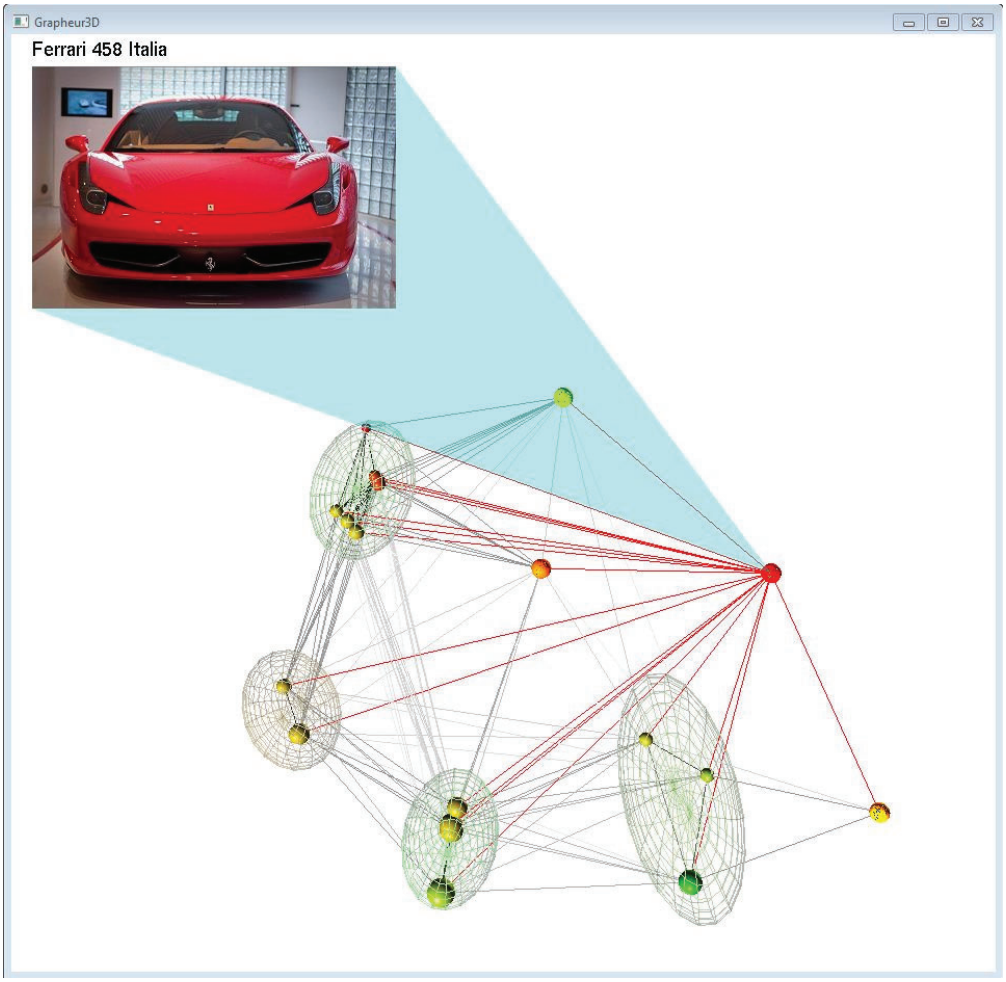


Figure 18.4: Clustering cars from mechanical characteristics.

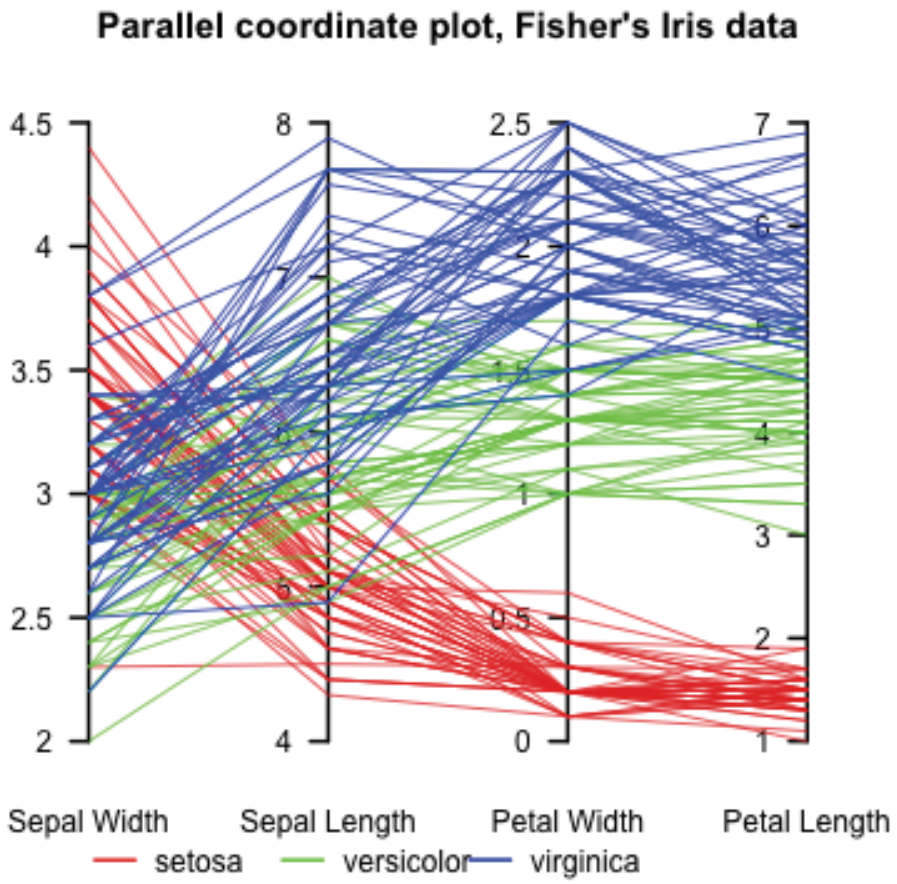


Figure 18.5: Parallel coordinates plot for Fisher Iris data (each flower is characterized by four geometrical measures). A vertical axe for each dimension. Each item is represented by a polyline cutting the i-th axe at the item's value of the i-th coordinate.

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.

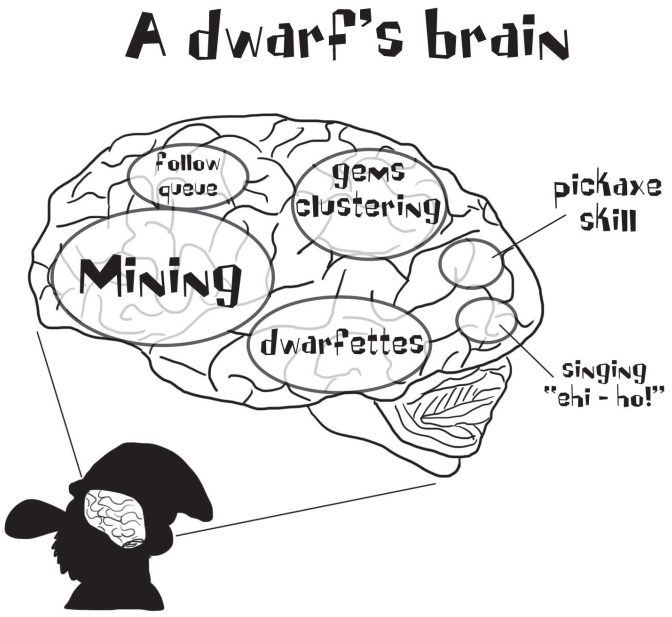
This method is called "**bottom-up**" because it starts from individual data points, merges the most similar ones and then proceeds by merging the most similar sets, until a single set is obtained. The number of clusters is not specified at the beginning: a proper number can be obtained by cutting the 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.

Chapter 19

Self-organizing maps

The grandmother cell *is a hypothetical neuron*
that represents any complex and specific concept or object.
It activates when a person's brain "sees, hears, or otherwise sensibly discriminates"
such a specific entity as his or her grandmother.
(Jerry Lettvin, 1969)



From the previous chapters, you are now familiar with the basic clustering techniques. Clustering identifies group of similar data, in some cases with a hierarchical structure (groups, then groups containing groups, ...). If an internal representation is available, a group can be represented with a prototype. This chapter deals with **prototypes arranged according to a regular grid-like structure and influencing each other if they are neighbors in this grid**.

The idea is to **cluster data (entities) while at the same time visualizing this clustered structure on a two-dimensional map**. One wants a visualization that is at least approximately coherent with the clustering — this should be puzzling enough to continue reading.

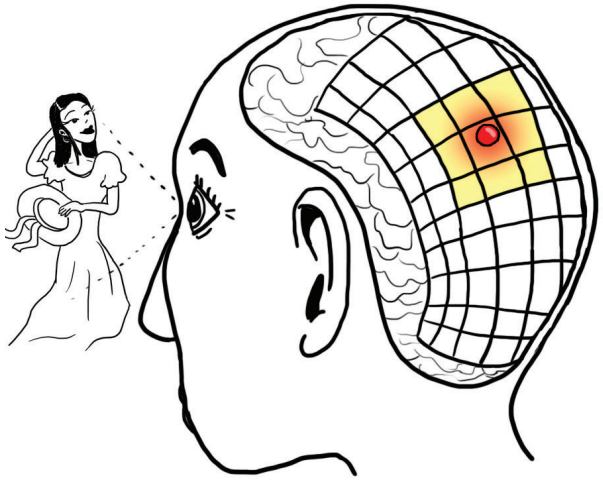


Figure 19.1: The presence of certain external stimuli activates a region in the brain ("*grandmother cell*"). In some areas, neurons are approximately organized according to a two-dimensional structure, like in the cortex, the surface layer of the brain where most high-level functionalities are located.

Let each cluster i be associated with a representative vector, a prototype $\mathbf{p}_i$. In the field of marketing, it is usual to identify different customer types, and describe them through prototypes (wealthy single individual, middle-class worker with family, etc.). A prototype will have the same number of coordinates as our entities and each component of the vector will describe a representative value for the given cluster, for example the average value of the entities contained in the cluster.

A coherent visualization demands that similar prototypes are placed in nearby positions in the 2D visualization space. Of course, for high-dimensional problems (problems with more than two coordinates), no exact solution is available and one aims at approximations which are sufficient for us to reason about the data. A **self-organizing map (SOM)** is a type of artificial neural network that is trained by using unsupervised learning to produce a two-dimensional representation of the training samples, called a map. This model was introduced as an artificial neural network by Teuvo Kohonen, and is also called a **Kohonen map**.

19.1 An artificial cortex to map entities to prototypes

A self-organizing map consists of component nodes or neurons. The arrangement of nodes is a regular placement in a two-dimensional grid. In some cases the grid is hexagonal, so that each node has six closest neighbors instead of four neighbors like in a traditional squared grid (Fig. 19.3). Associated with each node i is a prototype vector $\mathbf{p}_i$ of the same dimension as the input data vectors, *and* a position in the map space.

The analogy is again with our neural system: the **neurons are organized according to a physical network of connections in the brain, in practice two or three-dimensional**. Some neurons are tuned by evolution and training to fire electrical signals for particular events, as shown in Fig. 19.1. For example, a neuron may fire when your mother enters your visual field. The prototype is in this case given by visual features corresponding to your mother, the position is the physical position of the neuron in the brain.

Another principle of our neural systems is that, in many cases, **neurons that are neighbors tend to fire for similar input data** (a neighbor of the neuron firing for your mother presence may be close to one firing for an old photo of your mother). After training, the self-organizing map describes a mapping from a higher-dimensional input space to

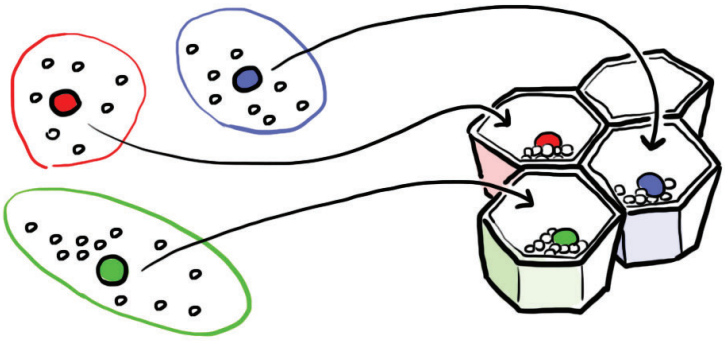


Figure 19.2: A SOM maps entities in a multi-dimensional space into cells in two-dimensional space. Each cell contains a prototype and the entities for which the prototypes is the most similar one.

a two-dimensional map space. Each cell in two dimensions corresponds to a neuron and contains a prototype vector. A generic entity is then mapped (or assigned) to the neuron with the prototype vector which is *nearest* to the vector describing the entity, as shown in Fig. 19.2. The training can start from a random initial configuration of the prototype vectors (for example picked to be equal to a random subset of entities) and then iterate by presenting and mapping randomly selected cases. The winning neuron $c(\mathbf{x})$, or c for brevity, is identified as the one with the prototype closest to the vector describing the current case $\mathbf{x}$:

$$c(\mathbf{x}) = \arg\min_i \|\mathbf{x} - \mathbf{p}_i\|. \tag{19.1}$$

Then the winning prototype $\mathbf{p}_{c(\mathbf{x})}$ is changed to make it more similar to the one of the current case presented to the network. In addition, the prototypes of *nearby* vectors are also changed in a similar manner, although by smaller and smaller amounts as the distance in the grid increases.

Think about a democratic system in which voters (entities) are asked to educate a set of regularly-arranged representatives (like in a chamber of the parliament) so that at least one of them represents a cluster of related ideas, and in which representatives sitting in nearby positions influence each other, and tend to become similar. Two "force fields" are active, attractive forces between entities and prototypes, and attractive forces between neighboring prototypes on the grid. Entities (voters) compete for prototypes: each entity is pulling its *winning* prototype and, to a less extent, the neighbors of the winning prototype, to move a bit towards itself, to make it progressively more similar. Of course, different entities are pulling in different directions and the resulting dynamical system is very complex.

After explaining the basic mechanism and motivations, let's now fix the details. In an online learning scheme, at each iteration t a random entity $\mathbf{x}$ is extracted, its winning neuron c is determined, and all prototype vectors $\mathbf{p}_i(t)$ at iteration (time) t are then modified as follows:

$$\mathbf{p}_i(t+1) \leftarrow \mathbf{p}_i(t) \; + \; \eta(t) \cdot \mathrm{Act}\big(c(\mathbf{x}), i, \sigma(t)\big) \cdot (\mathbf{x} - \mathbf{p}_i(t)), \tag{19.2}$$

where $\eta(t)$ is a time-dependent small learning rate, $\mathrm{Act}\big(c, i, \sigma(t)\big)$ is an **activation function** which depends on the distance between the two neurons in the 2D grid, and on a time-dependent radius $\sigma(t)$. The two neurons involved in the formula are: the winning neuron c for pattern $\mathbf{x}$, and the neuron i whose pattern $\mathbf{p}_i(t)$ is being updated. The modification mechanism resembles the update described in equation (17.9) for soft clustering in k-means, but with the important difference given by the regular two-dimensional organization of the neurons, which now determines the activation level.

To help convergence, usually the learning rate decreases in time, and the same happens for the radius parameter. The idea is that at the beginning the neuron prototypes are moving faster (*neural plasticity* is higher for young children!), and they tend to activate a larger set of neighbors, while movements are smaller and limited to a smaller set

of neighbors in the last phase, when hopefully the arrangement already identified the main characteristics of the data distribution and only a fine-tuning is needed. The learning rate in some cases decreases like $\eta(t) = A/(B+t)$. A reasonable default can be $\eta(t) = 1/(20+t)$.

In batch training, all N entities $\mathbf{x}_j$ are presented to the SOM and their winning neurons $c(\mathbf{x}_j)$ are identified before proceeding to an update as follows:

$$\mathbf{p}_i(t+1) \leftarrow \frac{\sum_{j=1}^{N} \mathrm{Act}\big(c(\mathbf{x}_j), i, \sigma(t)\big) \cdot \mathbf{x}_j}{\sum_{j=1}^{N} \mathrm{Act}\big(c(\mathbf{x}_j), i, \sigma(t)\big)}. \tag{19.3}$$

Each prototype is updated with a weighted average over all entities, the weight being proportional to the vicinity in neuron grid space (usually two-dimensional) between the winning neuron prototype and the current prototype.

Because of the system complexity, one is encouraged to try different parameters and different time schedules until acceptable results are obtained. For example, a suitable neighborhood activation function can be:

$$\mathrm{Act}\big(c, i, \sigma(t)\big) = \exp\left(-\frac{{d_{ci}}^2}{2\,\sigma^2(t)}\right), \tag{19.4}$$

where d_{ci} is the distance between the two neurons in the two-dimensional grid, and $\sigma(t)$ is a neighborhood radius, at the beginning including more neighbors than the closest ones, at the end including only a set of close neighbors. Be careful not to confuse the distance between neurons in the grid, as shown in Fig. 19.3, with the distance between prototype vectors in the original multi-dimensional space of the data!

Let $\mathrm{TOT_{SOM}}$ be the total number of SOM neurons, and $\mathrm{TOT_{iter}}$ the number of iterations executed. The default value starts from $\sqrt{\mathrm{TOT_{SOM}}}$, a value close to the radius of the grid if the grid is a square one, and ends with the value 2, as follows:

$$\sigma(t) = \frac{(\mathrm{TOT_{iter}} - t)\sqrt{\mathrm{TOT_{SOM}}} + 2t}{\mathrm{TOT_{iter}}}. \tag{19.5}$$

One should not be discouraged by the complexity intrinsic in this and in similar mapping tasks: in many cases acceptable results are obtained by considering simple default values for the basic parameters. On the other hand, it is not surprising that a basic mapping mechanism of our brain is indeed characterized by a high complexity level, we are intelligent and in part unpredictable human beings, aren't we?

19.2 Using an adult SOM for classification

Even if you do not want to indulge in the *debauchery of indices* of the above mathematical details, you can still use SOM effectively as a guide in reasoning about your problem. After training, the SOM can be used to classify new objects by finding the closest (winning) prototype and assigning the new object to the corresponding cell, as illustrated in Fig. 19.4. In many cases, after looking at the prototypes, it will be easy to give *names* to the different cells, to help reasoning and remembering. But let's note that cells may discover *unusual* combinations, leading to interesting insight and *detection of novel groups*, and not only to re-discovering trivial classifications.

Let's imagine that you trained your SOM on marketing data: each cell may represent a characteristic group of customers. Possible names could be: "wealthy-single-individual," "poor-family-with-kids," "senior-retired-person," "spoiled-adolescent," etc. When a new customer arrives, you may easily identify the appropriate prototype, for example to pick the best strategy to sell him your products. If you are a movie fan, and you train your SOM to classify different kinds of movies, you may use a SOM to classify a new movie, for example to predict if you will like it or not (with a high probability).

In a SOM, the quality of training can be measured by the **quantization error** (the average error incurred by substituting an entity $\mathbf{x}$ with its winning prototype vector $\mathbf{p}_{c(\mathbf{x})}$, i.e., the average distance from each data vector to its nearest prototype) or by the **topological error**, which is related to the failure to assign close vectors in the original high-dimensional space to close neurons in neuron-grid space (usually two-dimensional). The topological error can

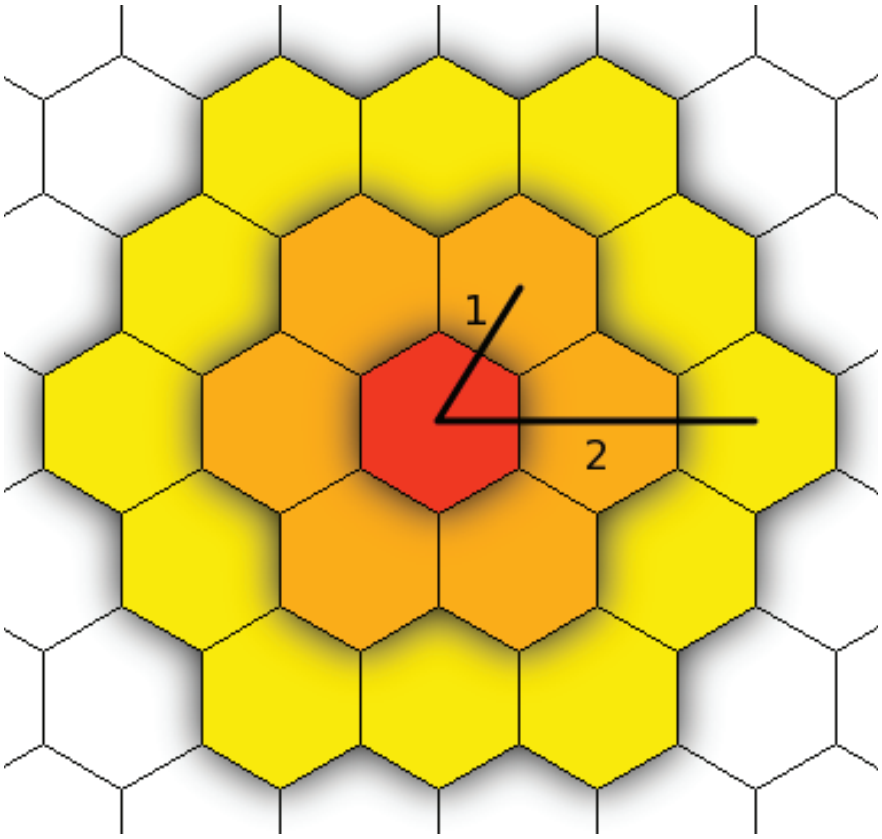


Figure 19.3: An example of neighborhood in a self organizing map: neighboring cell at distance 1 and 2 from the central are shown.

be evaluated as the fraction of all data vectors for which the first and the second nearest prototypes (in the original multi-dimensional space) are not represented by adjacent neurons in the grid mapping.

Color coding can be used to represent the value of the data point along a dimension, while the size of each hexagon can represent the value along another dimension (Fig. 19.5). The colored maps are called components or **component planes** and can be compared to identify local relationships. One can devise interesting new analysis techniques by combining the SOM map with a scatterplot display, or with a parallel coordinates plot (Fig. 18.5). For example, when the mouse pointer is moved over the SOM cells, the position of the prototype vector associated with each cell can be shown in a scatterplot or in a parallel coordinates plot. In this manner, details of relevant entities can be further analyzed.

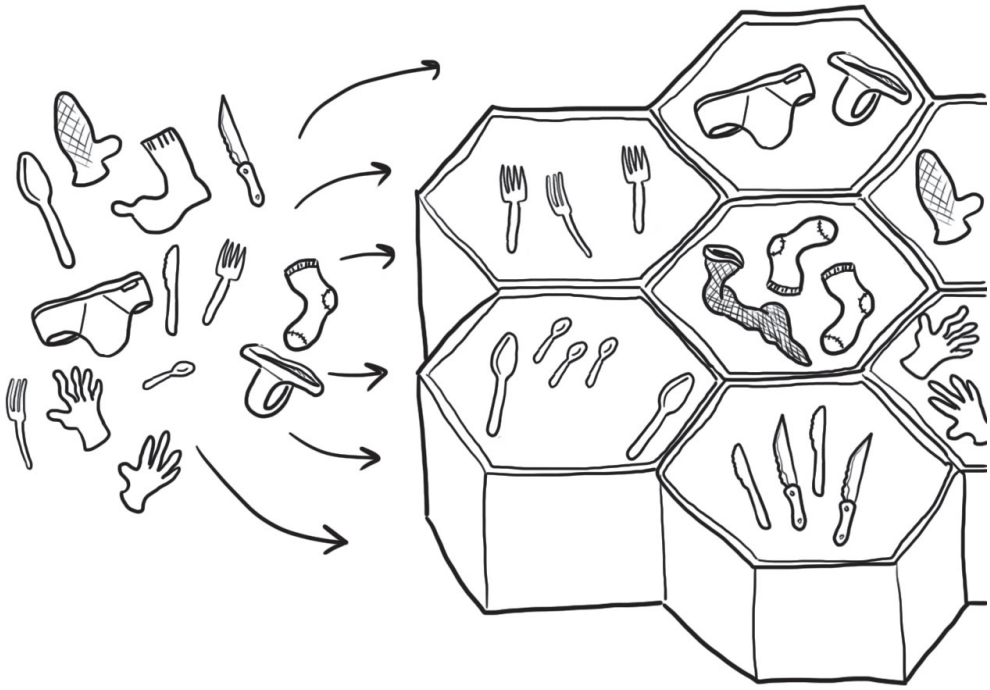


Figure 19.4: An analogy for the SOM: each cell is like a drawer in a well-organized piece of furniture. Neighboring drawers are used to contain similar objects.

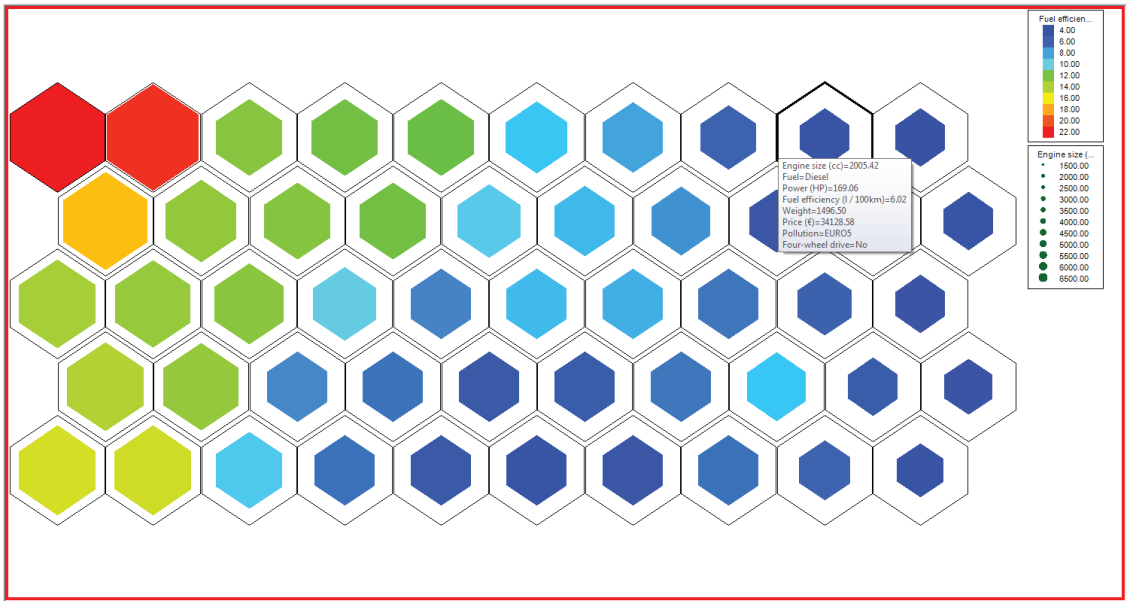


Figure 19.5: A SOM, color and size depends on two coordinates of the prototype vectors. Prototypes can be examined by passing with the mouse over the cell (LIONoso.org software).

Gist

Self-organizing maps reach two objectives: placing a set of prototypes close to clusters of data points, and having the prototypes organized in a two-dimensional grid so that neighboring prototypes in the grid tend to be mapped to similar data points.

The motivations are in part biological (our neural cortex is approximately organized according to two- and three-dimensional arrangement of neurons) and in part related to visualization. A two-dimensional grid can be visualized on the screen, and the characteristics of the prototypes are not randomly scattered but slowly varying, because of the neighboring relationships, leading to more intelligible visualizations.

If data points are imagined as schooling fishes in the sea, a SOM is an elastic fisherman's network aiming at capturing the largest number of them without breaking up.

Chapter 20

Dimensionality reduction by projection

You, who are blessed with shade as well as light, you, who are gifted with two eyes, endowed with a knowledge of perspective, and charmed with the enjoyment of various colors, you, who can actually see an angle, and contemplate the complete circumference of a Circle in the happy region of the Three Dimensions – how shall I make it clear to you the extreme difficulty which we in Flatland experience in recognizing one another's configuration?
(Flatland - 1884 -Edwin Abbott Abbott)

In **exploratory data analysis** one is actually using the **unsupervised learning capabilities of our brain** to identify interesting patterns and relationships in the data. It is often useful to map entities to two dimensions, so that they can be analyzed by our eyes. The mapping has to preserve as much as possible the relevant information present in the original data, describing **similarities and diversities** between entities. For example, think about a marketing manager analyzing similarities and differences between customers, so that different campaigns can be tuned to the different groups, or think about the head of a human resources department who aims at classifying the competencies possessed by different employees. We would like to organize entities in two dimensions so that **similar objects are near each other and dissimilar objects are far from each other**. Let's note that there is a clear difference between this and SOM

maps. In SOM, prototype vectors associated with a two-dimensional grid are moved (their coordinates are changed) to cover the original data space, whereas here the original points are mapped in different ways onto a two-dimensional surface.

Following the discussion in Chapter 17 (see Fig. 17.2), it's useful to recall the two ways in which initial information can be given to the system. A first possibility is that entities are characterized by an **internal structure (a vector of coordinates)**. In this case the raw coordinates can be used to *derive* a similarity measure, as by considering the Euclidean distance between the two corresponding vectors. A second possibility is that entities are endowed with an **external structure of pairwise relationships**, expressed by similarities or dissimilarities. We deal only with dissimilarities to avoid confusion and we leave to the reader the simple exercise of transforming the formulas to handle the other case. To establish a suitable notation, let the n entities be characterized by some mutual dissimilarities δ_{ij}. Some dissimilarities may be unknown.

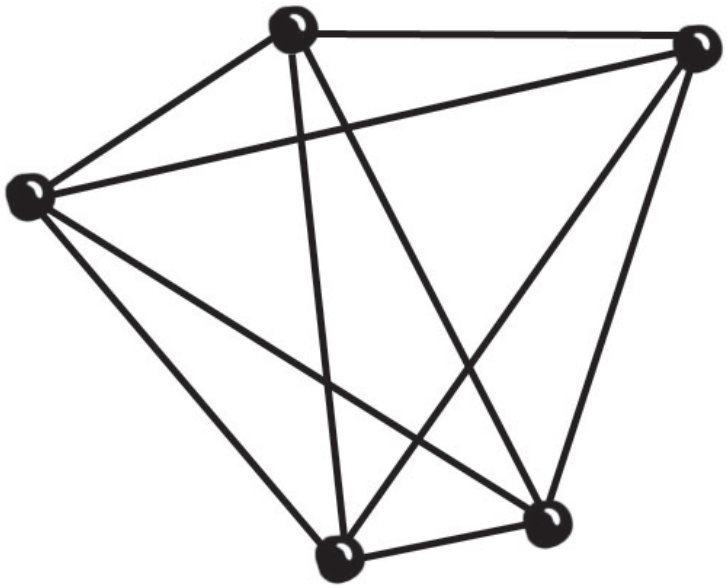


Figure 20.1: A graph with external dissimilarities (edges) between entities (nodes).

An appropriate model is an undirected graph, like the one illustrated in Fig. 20.1, in which each entity is represented by a node, and a connection with weight δ_{ij} is present between two nodes, if and only if a distance δ_{ij} is defined for the corresponding entities. The set of edges in the graph is denoted by E.

The two situations (coordinates or relationships) can be combined. In some cases the information given to the system consists of **both coordinates and relationships**. As a very concrete example, imagine that some automated clustering method has been applied to the data vectors. We can then declare two entities to be dissimilar ($\delta_{ij} = 1$) if and only if they do not belong to the same cluster. This additional information can be used to encourage a visualization where items coming *from the same cluster* tend to be close in the two-dimensional space. In other cases, indications about dissimilarities among items given by people can help for tuning the visualization and adapting it to the user's wishes.

A way to distinguish the different contexts has to do with the *level of supervision* in the visualization, i.e., the type of hints given to the process. The type of supervision ranges from a purely **unsupervised** approach (only coordinates are given) to a **supervised** approach (relationships or dissimilarities are fully specified), to mixed approaches combining unsupervised exploration in the vector space of coordinated and labeling methods.

After clarifying the context, depending on the available data, it is time to consider ways to use the data to produce useful visualizations. Methods derived from linear algebra are described in the following sections, while more general

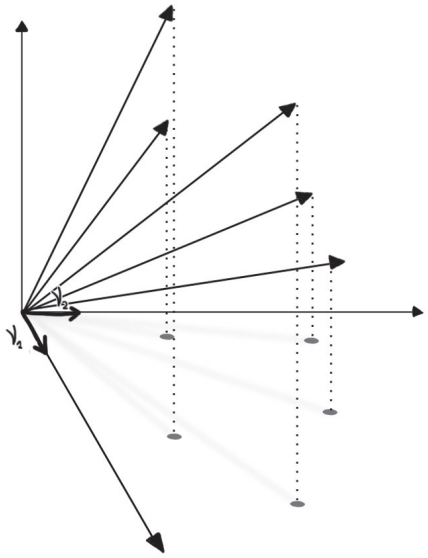


Figure 20.2: A projection: each dotted line connecting a vector to its projection is perpendicular to the plane defined by ν^1 and ν^2 (in this case the direction vectors are the X and Y axes, in general a projection can be on any plane identified by two linearly independent vectors).

nonlinear methods are described in future chapters. As usual, linear methods are simpler to understand, while nonlinear methods are in principle more powerful, although more complex.

20.1 Linear projections

This chapter starts from linear algebra, Let n be the number of vectors (entities), and let m be the dimension of each vector (the number of coordinates). For convenience, the n vectors can be stored as rows of an $n \times m$ matrix X. To help the reader, Latin indices i, j ranges over the data items, while Greek indices α, β range over the coordinates. Thus $X_{i\alpha}$ is the α-th coordinates of item i. For the rest of this chapter we assume that the **data is centered**, i.e., that the mean of each coordinate over the entire dataset is zero: $\sum_{i=1}^{n} X_{i\alpha} = 0$. If the original data are not centered, they can be preprocessed by a trivial translation. In other words, we are not interested in the absolute positions of the data points, but in their relative positions with respect to the other data. We denote by S the $m \times m$ biased **covariance matrix**: $S = \frac{1}{n} X^T X$, with the components: $S_{\alpha,\beta} = \frac{1}{n} \sum_{i=1}^{n} X_{i\alpha} X_{i\beta}$.

It is called covariance because each term measures how two coordinates tend to *vary together* for the different data points. The sum in the covariance will have a large positive value if positive values of the first coordinate tend to be accompanied by positive values of the second coordinate, and the same tends to be true for negative values. Actually, the covariance is changed if the values of a coordinate are multiplied by a constant (this happens every time one changes physical units, for example from millimeters to kilometers). A measure which does not depend on changes of physical units is the *correlation coefficient*, derived by dividing the covariance by the product of the standard deviations of the involved coordinates. (See Section 7.2.)

We consider a linear transformation L of the items to a space of dimension p, the usual value for p is two, but we keep this presentation more general. L is represented by a $p \times m$ matrix, acting on vector x by the usual matrix multiplication $y = Lx$. Each coordinate α of y is obtained by a scalar product of a row ν^α of L and the original

coordinate vector x. The p rows $\nu^1, \ldots, \nu^p \in \mathbb{R}^m$ of L are called **direction vectors** and in the following we assume that they have unit norm $\|\nu^\alpha\| = 1$. Therefore each coordinate α in the transformed p-dimensional space is obtained by **projecting** the original vector x onto ν^α. If we project all items and we repeat for all coordinates we obtain the **coordinate vectors** $x^1 = X\nu^1, \ldots, x^p = X\nu^p$.

Among the possible linear transformations, interesting visualizations are obtained by **orthogonal projections**: the direction vectors $\nu^1, \ldots, \nu^p$ are mutually orthogonal and with unit norm: $\nu^\alpha \cdot \nu^\beta = \delta_{\alpha,\beta}, \alpha, \beta = 1, \ldots, p$, as illustrated in Fig. 20.2. Please note that here $\delta_{\alpha,\beta}$ is the usual Kronecker delta, equal to 1 if and only if the two indices are equal, not to be confused with dissimilarities! An example of orthogonal projection is a selection of a subset of the original coordinates (in this case $\nu^\alpha = (0, 0, \ldots, 1, \ldots, 0, 0)$, a vector with 1 for the selected coordinate and 0 in all other places). Other examples are obtained by first rotating the original vectors and then selecting a subset of coordinates.

The visualization is simple because it shows **genuine properties** of the data, corresponding to the intuitive notion of navigating in the original space, far from the data points, and looking at the data from different points of view[1]. Think about placing a two-dimensional screen at an arbitrary orientation, switching a lamp on (very far from the data), and observing the projected shadows. On the contrary, nonlinear transformations may deform the original data distribution in arbitrary and potentially very complex and counter-intuitive ways, as by viewing the world through a deforming lens.

As an additional feature of linear projections, it is easy to explain the p-dimensional coordinates because each one is a linear combination of the original coordinates (for example, the magnitude of the combination coefficients "explains" a lot about the relevance of the initial coordinates in the projection).

Last but not least, the storage requirements are limited to storing the direction vectors, and the computational complexity for projecting each point is the usual one for matrix-vector multiplication.

Now that the motivation is present, let's consider some of the most successful linear visualization methods.

20.2 Principal Components Analysis (PCA)

To *understand* the meaning of this historic transformation (PCA was invented in 1901 by Karl Pearson) it is useful to concentrate on what PCA is trying to accomplish. As usual, *optimization is the source of power* and helps us to understand the deep meaning of operations. Now, PCA finds the orthogonal projection that **maximizes the sum of all squared pairwise distances** between the projected data elements.

If dist^p_{ij} is the distance between the projections of two data points i and j:

$$\text{dist}^p_{ij} = \sqrt{\sum_{\alpha=1}^{p} ((X\nu^\alpha)_i - (X\nu^\alpha)_j)^2},$$

PCA maximizes:

$$\sum_{i<j} (\text{dist}^p_{ij})^2. \tag{20.1}$$

The objective is to spread the points as much as possible, but the fact of considering only projections implies that the mutual distances cannot increase beyond the original ones: $\text{dist}^p_{ij} \leq \text{dist}_{ij}$ (just consider Pythagoras theorem applied to the triangle defined by the original vector, the projection and the vector connecting the projection to the original vector). The best that we can obtain is to approximate as much as possible the original sum of squared distances:

$$\max_{\nu^1,\ldots,\nu^p} \sum_{i<j} (\text{dist}^p_{ij})^2 \leq \sum_{i<j} (\text{dist}_{ij})^2. \tag{20.2}$$

[1] Actually the mapping executed by our eye or by cameras is a perspective viewing projection so that the analogy is not to be taken literally.

After introducing the $n \times n$ *unit Laplacian* matrix L^u, as $L^u_{ij} = (n \cdot \delta_{ij} - 1)$, the optimization problem can be posed as the solution of:

$$\begin{aligned} \max_{\nu^1,\ldots,\nu^p} \quad & \sum_{\alpha=1}^{p} (\nu^\alpha)^T X^T L^u X \nu^\alpha \\ \text{subject to} \quad & \nu^\alpha \cdot \nu^\beta = \delta_{\alpha,\beta}, \qquad \alpha, \beta = 1, \ldots, p. \end{aligned} \tag{20.3}$$

The **Laplacian matrix** is in general a key tool for describing **pairwise relationships** between entities. In fact, it is used a lot in the study of graphs, where the pairwise relationship is given by weighted edges connecting two nodes. In general, it is an $n \times n$ symmetric positive semidefinite matrix, with zero row and column sums. Its usefulness is related to the possibility to express in a compact manner the **weighted sum of all pairwise squared distances**:

$$x^T L x = \sum_{i<j} -L_{ij}(x_i - x_j)^2. \tag{20.4}$$

Considering the p coordinate vectors introduced above, it is easy to check that:

$$\sum_{\alpha=1}^{p} (x^\alpha)^T L x^\alpha = \sum_{i<j} -L_{ij} \, (\text{dist}^p_{ij})^2. \tag{20.5}$$

The optimal solution of equation (20.3) is given by the p **eigenvectors with largest eigenvalues** of the $m \times m$ matrix $X^T L^u X$. For centered coordinates, this matrix is identical to the covariance matrix apart from a multiplicative factor (which does not influence the eigenvectors) $X^T L^u X = n^2 S$. The solutions for PCA is obtained by finding the **eigenvectors of the covariance matrix**. We prefer the above form with the Laplacian matrix which can be easily generalized to cases where we have additional information about relationships between data points. (See Section 20.3.) While we are not giving the details in this section, the fact that solutions are eigenvectors is again related to formulating the problem as a maximization task: the original quantity to be minimized is quadratic, and requiring zero gradient and satisfaction of constraints leads to linear (eigenvalue) equations.

As a side-effect of the above solution, PCA transforms a number of possibly correlated variables into a smaller number of uncorrelated variables called principal components. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. Another interesting explanations of PCA is that it minimizes the mean squared error incurred when approximating the data with their projection.

A geometric interpretation in three dimensions is shown in Fig. 20.3. If the data points form an m-dimensional ellipsoidal cloud, the eigenvectors of the covariance matrix are the principal axes of the ellipsoid. Principal component analysis reduces the dimensionality by restricting attention to the directions along which the scatter of the cloud is the greatest.

PCA is a simple and very popular transformation but with obvious limits (maybe too popular?). It simply performs a coordinate rotation that aligns the transformed axes with the directions of maximum variance. Having a larger variance is not always related to having a larger *information content*, for example it can be related to having a larger measurement noise. In addition, the variance along a coordinate can be easily enlarged by multiplying the coordinate by a constant, but the information content of course does not change. In other words, the PCA results depend on choosing appropriate physical units, a spherical cloud of points can become elongated for the superficial reason that millimeters instead of meters are used to measure a physical distance. Furthermore, because a sum of squared distances is involved in the optimization of equation (20.1), PCA is **sensitive to outliers**. Points which are very far from most of the other points contribute with large (squared) distances and encourage a choice of direction vectors which may be very different from those chosen if the outliers are eliminated. When PCA is used to identify promising features for classification in supervised learning, its main limitation is that it makes no use of the class label of the feature vector.

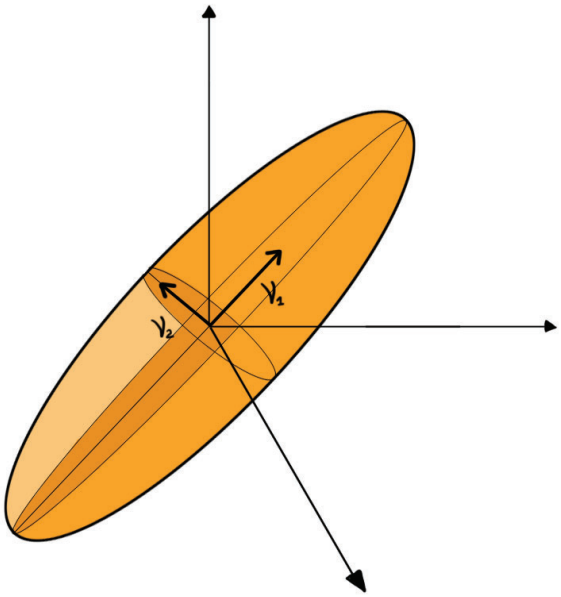


Figure 20.3: Principal component analysis, data are described by an ellipsoid. The first eigenvector is in the direction of the longest principal axis, the second eigenvector lies in a plane perpendicular to the first one, along the longest axis of the two-dimensional ellipse.

There is no guarantee that the directions of maximum variance will contain good features for discrimination, see also Chapter 7 about selecting and ranking features.

The computational cost is related to solving the eigenvalues-eigenvectors problem for a matrix of dimension $m \times m$. Let's note that the number of points n is not relevant, therefore the method is particularly fast when the number of initial coordinates is limited, even if the number of data points is very large. More details about PCA can be found in [238].

20.3 Weighted PCA: combining coordinates and relationships

As we mentioned before, in some cases additional information is available about the data in the form of *relationships* between (some) entities. For example, we may have a *class label*, so that some pairs are in the same class and we would like them to be close in the projected distance. Or we may have additional information about dissimilarities beyond what is obtained from the raw data coordinates.

Fortunately, we can extend PCA to incorporate additional information. For example, we can minimize a **weighted** sum of the squared projected distances:

$$\sum_{i<j} d_{ij} \cdot (\text{dist}_{ij}^{p})^2. \tag{20.6}$$

If a weight d_{ij} is large, a large contribution to the function to be maximized is obtained if the corresponding dist_{ij}^{p} is

also large. We can then interpret d_{ij} as weights measuring the importance that points i and j are placed far apart in the low dimensional projection space, let's call them dissimilarities. As in the unweighted case, we can now assign to the problem an $n \times n$ Laplacian matrix L^d:

$$L^d_{ij} = \begin{cases} \sum_{j=1}^{n} d_{ij} & \text{if } i = j \\ -d_{ij} & \text{otherwise} \end{cases}, \tag{20.7}$$

and obtain the optimal projection with the direction vectors given by the p highest eigenvectors of the matrix $X^T L^d X$.

We can now use the dissimilarity values to create different variations of PCA. In normalized PCA $d_{ij} = 1/\text{dist}_{i,j}$ to discount large original distances in the optimization. This can be useful to increase the original PCA robustness with respect to outliers.

In supervised PCA, with data labeled as belonging to different classes, we can set the dissimilarities d_{ij} to a small value ϵ if i and j belong to the same class, to a value 1 if they belong to different classes. This weighing instructs the projection that it is more important to put at large distances points of different clusters. If ϵ is zero, the internal structure of each cluster is set only indirectly according to the inter-cluster relationships of its members.

20.4 Linear discrimination by ratio optimization

Other possibilities to project points while considering also class labels arise by optimizing **ratios of quantities**. Obviously, maximizing a ratio reflects a **compromise** between maximizing the numerator and minimizing the denominator.

Let's consider a c-way classification problem, the standard case being with two output classes. Fisher analysis deals with finding a vector ν_F such that, when the original vectors are projected onto it, values of the different classes are separated in the best possible way.

A nice separation is obtained when the sample *means* of the projected points are as different as possible, when normalized with respect to the average *scatter* of the projected points. The division by the scatter corresponds to the intuition that what matters is not the separation of means *per se*, but the fact that values are sufficiently clustered around their means so that the classes can be clearly *separated*. This is not possible if they are scattered so that most values are mixed in the same area, even if the means are separated.

Let n_i be the number of points in the i-th cluster, let μ_i and S_i be the mean vector and the biased covariance matrix for the i-th cluster. The matrix $S_{\text{within}} = \frac{1}{n}\sum_{i=1}^{c} n_i S_i$ is the **average within-cluster covariance matrix** and $S_{\text{between}} = \frac{1}{n}\sum_{i=1}^{c} n_i \mu_i \mu_i^T$ the **average between cluster covariance matrix**.

In detail, the Fisher linear discriminant is defined as the linear function $y = \nu^T \mathbf{x}$ for which the following ratio is maximized:

$$\frac{\nu^T S_{\text{between}}\, \nu}{\nu^T S_{\text{within}}\, \nu}. \tag{20.8}$$

Think about **maximizing the ratio of between-class to within-class scatter**: we want to maximally separate the clusters (the role of the numerator where the projections of the means count) and keep the clusters as compact as possible (the role of the denominator).

It can be proved that the maximizer of Fisher's criterion is the same as the maximizer of:

$$\frac{\nu^T S_{\text{between}}\, \nu}{\nu^T\, S\, \nu}. \tag{20.9}$$

For the special but interesting case of two classes, see also Fig. 20.4, after specializing the above equations, the Fisher linear discriminant is defined as the linear function $y = \mathbf{w}^T \mathbf{x}$ for which the following criterion function is maximized:

$$\text{Separation}(\mathbf{w}) = \frac{\|\widetilde{m}_1 - \widetilde{m}_2\|^2}{\widetilde{s}_1^2 + \widetilde{s}_2^2}, \tag{20.10}$$

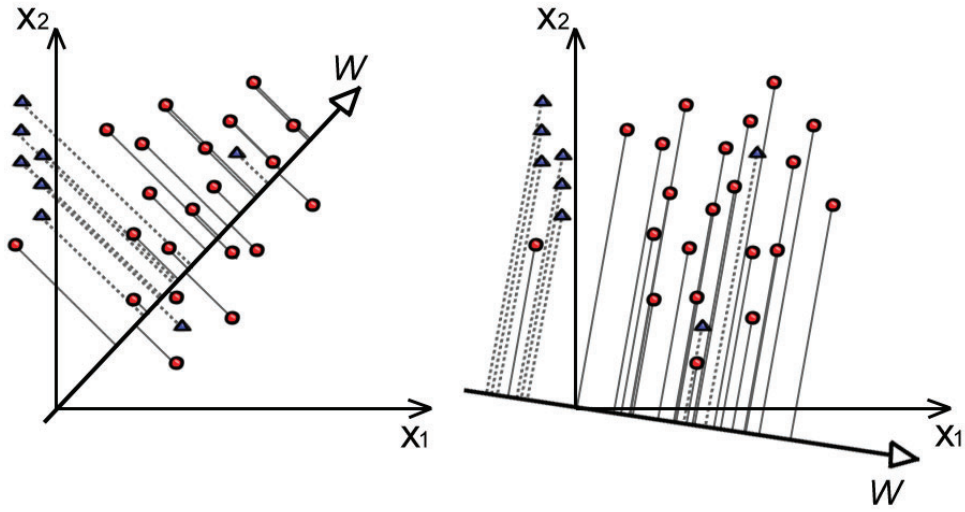


Figure 20.4: Fisher linear discrimination (triangles belong to one class, circles to another one): the one-dimensional projection on the left mixes projected points from the two classes, while the projection on the right best separates the projected sample means with respect to the projected scatter.

where $\widetilde{m}_i$ is the sample mean for the projected points $\widetilde{m}_i = (1/n_i) \sum_{y \in Class_i} y$, and $\widetilde{s}_i^2$ is the *scatter* of the projected samples of each class: $\widetilde{s}_i^2 = \sum_{y \in Class_i} (y - \widetilde{m}_i)^2$. Think about maximizing the ratio of between-class to total within-class scatter. The solution is:

$$\mathbf{w}_F = (S_w)^{-1}(\mathbf{m}_1 - \mathbf{m}_2), \tag{20.11}$$

where $\mathbf{m}_i$ is the d-dimensional sample mean for class i and S_w is the sum of the two scatter matrices S_i defined as follows:

$$S_i = \sum_{\mathbf{x} \in Class_i} (\mathbf{x}_i - \mathbf{m}_i)(\mathbf{x}_i - \mathbf{m}_i)^T. \tag{20.12}$$

An interesting application of Fisher linear projection is for selecting features in neural networks and general model-building techniques based on supervised learning. (See Section 20.4.1.)

20.4.1 Fisher discrimination index for selecting features

Let's consider a two-way classification problem (with two output classes) with input vectors of d dimensions. Fisher analysis deals with finding the vector $\mathbf{w}_F$ so that, when the original vectors are projected onto it, values of two classes are separated in the best possible way. The method has already been encountered in Section 20.4.

Let's remind that a nice separation of the projected points is obtained when the sample *means* of the projected points are as different as possible, when normalized with respect to the average *scatter* of the projected points. The division by the scatter corresponds to the intuition that what matters is not the separation of means *per se*, but the fact that values are sufficiently clustered around their means so that the two classes can be clearly separated. This is not possible if they are scattered so that most values are mixed in the same area, even if the means are separated.

We can now rate the importance of feature i according to the magnitude of the i-th component of the Fisher vector $\mathbf{w}_F$ defined in equations (20.11)–(20.12). Identifying the largest components in Fisher vector will heuristically identify the most relevant directions (coordinates) for separating the classes. In other words, if the direction of a coordinate vector is similar to the direction of the Fisher vector, projecting onto the given coordinate axis can be approximately used to separate the two classes, instead of projecting onto the original Fisher vector. Let's note that the criterion is

empirical because it is based on *linear* projections: in some cases a *nonlinear* combination of features which rank poorly by the above Fisher criterion may do an excellent job in separating classes.

If the number of dimension d is very large, inverting the S_w matrix in equation (20.11) can be numerically difficult, and this first measure can be insufficient to correctly order many features.

A simpler and possibly more effective criterion to rank feature k, called **Fisher's discrimination index**, is to measure the Separation$(\mathbf{w})$ value defined in equation (20.10) when considering a vector $\mathbf{e}_k$ along the specific direction k (zero everywhere, 1 only in the k-th coordinate). In other words, we want to measure the discrimination which can be achieved if only the k-th coordinate of the points is considered.

20.5 Fisher's linear discriminant analysis (LDA)

The original Fisher method described above aims at finding a single direction vector (a single projection). For finding p direction vectors, the idea can be generalized to the popular technique known as Fisher's **linear discriminant analysis (LDA)**, which is based on maximizing the following ratio:

$$\begin{array}{ll} \max\limits_{\nu^1,\ldots,\nu^p} & \dfrac{\sum_{\alpha=1}^{p} (\nu^\alpha)^T \; S_{\text{between}} \; \nu^\alpha}{\sum_{\alpha=1}^{p} (\nu^\alpha)^T \; S \; \nu^\alpha} \\ \text{subject to} & (\nu^\alpha)^T S \nu^\beta = \delta_{\alpha\beta}, \qquad \alpha, \beta = 1, \ldots, p. \end{array} \tag{20.13}$$

Although widely used, LDA, like the basic PCA, is sensitive to outliers and it does not take the shape and size of clusters into consideration. A more flexible generalization is based on maximizing a ratio of the following form:

$$\begin{array}{ll} \max\limits_{\nu^1,\ldots,\nu^p} & \dfrac{\sum_{i<j} d_{ij} (\text{dist}_{ij}^p)^2}{\sum_{i<j} \text{sim}_{ij} (\text{dist}_{ij}^p)^2} \\ \text{subject to} & (\nu^\alpha)^T X^t L^s X \nu^\beta = \delta_{\alpha\beta}, \qquad \alpha, \beta = 1, \ldots, p, \end{array}$$

where d_{ij} are dissimilarity weights, sim_{ij} are similarity weights (they express the preference for placing two entities together in the projection), and L^s is the Laplacian matrix corresponding to the similarities

$$L_{ij}^s = \begin{cases} \sum_{j=1}^{n} \text{sim}_{ij} & \text{if } i = j \\ -\text{sim}_{ij} & \text{if } i \neq j \end{cases}. \tag{20.14}$$

After defining a generalized eigenvector problem of (A, B) as the solution of $Ax = \lambda Bx$, the optimal solution of equation (20.13) is given by the p highest generalized eigenvectors of $(X^T L^d X, X^T L^s X)$.

Apart from the mathematical details, remember that finding an optimal projection requires defining in measurable ways what is meant by optimality. Above we have seen ways of **combining unsupervised (based only on coordinates) and supervised information (based on relationships)**, and ways of giving different weights to different preferences for placing items distant or close.

After the user intelligence is spent on defining the optimization problem, what is left is to derive $m \times m$ matrices by the appropriate multiplications and to solve an $m \times m$ generalized eigenvector problem in an efficient and numerically stable way. Of course, the technique is very fast when the number of original coordinates m is limited, even if the number of points to project is very large.

Interestingly enough, eigenvectors will be encountered again in chapter 14 about web mining, for ranking web pages.

20.6 Projection Pursuit: searching for interesting structure guided by an explicit index

Projection pursuit (PP) is a catchy name proposed in [138] and based on [243] for exploratory data analysis based on the explicit definition of an objective function (called *projection index*). "Pursuit" refers to the systematic use of optimization techniques to identify the best projection, or at least a locally-optimal one, according to the index. Let's stress that structure can be obscured by projections but never enhanced (projection is a shadow of an actual structure in many dimensions) and therefore linear projections are always solid low-danger techniques.

Many of the methods of classical multivariate analysis (like PCA in Sec. 20.2 or LDA in Sec. 20.5) are special cases of PP, but this dedicated section underlines some interesting observations of the most advanced PP methods, following [134] and [197].

The purpose of *exploratory* projection pursuit is to discover non-obvious (non-linear) effects in the data by low-dimensional projections. The human gift for pattern recognition works best by observing data projected in two or three dimensions. Linear effects are "obvious" because they can be easily identified by PCA, i.e., by deriving and using the covariance structure of variable pairs.

20.6.1 Normal Gaussian distributions are non-interesting: sphering or whitening

If the cloud of data points follows with high fidelity an elliptically symmetric distribution like that of Fig. 20.6, the covariance matrix tells the whole story. A normal distribution (following a multi-dimensional Gaussian probability density function) is a paradigm of a non-interesting structure. If we project a normal distribution, we obtain another normal distribution, usually quite boring. Let's consider that a normal distribution can be the real-world equivalent of a constant value (if we measure the height of a person with more digits than required by our instrument's precision, we get an approximately normal distribution of values).

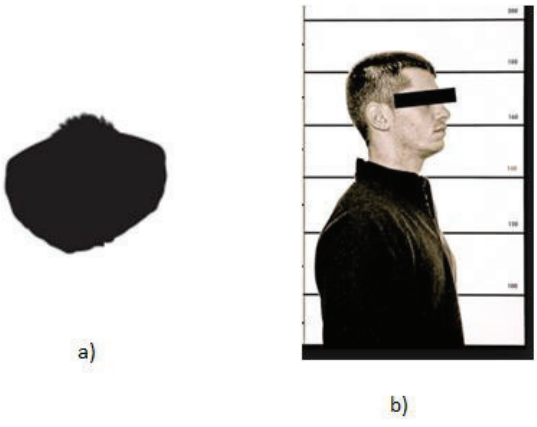


Figure 20.5: Projection Pursuit: if we project a human body, the interest of a projection is related to the amount of nonlinearity. A projection from above leads to a quasi-normal distribution (a), a lateral projection is of bigger use to identify people (b).

If the task confirms that normal distributions are of no interest, a "first aid" treatment of the data consists of **sphering** or **whitening**. *Sphering* means squeeze, stretch and rotate data until the ellipsoid describing the covariance

structure (Fig. 20.6) becomes a sphere. The alternative term of "whitening" originates from signal processing and reminds that the input vector is transformed into a *white noise* vector, a signal whose samples are regarded as a sequence of serially uncorrelated random variables with zero mean and finite variance.

In detail, a **whitening transformation** is a decorrelation transformation that transforms an arbitrary set of variables having a known covariance matrix M into a set of new variables whose covariance is the identity matrix (meaning that they are uncorrelated and all have variance 1).

A first step consists of **centering** the data, by subtracting the average vector. Then the correlation matrix M is derived as the expected value of the outer product of X with itself, namely:

$$M = \mathrm{E}[XX^T]$$

When M is symmetric and positive definite (and therefore not singular), it has a positive definite symmetric square root $M^{1/2}$, such that $M^{1/2}M^{1/2} = M$. Since M is positive definite, $M^{1/2}$ is invertible, and the vector $Y = M^{-1/2}X$ has covariance matrix:

$$\mathrm{Cov}(Y) = \mathrm{E}[YY^T] = M^{-1/2}\,\mathrm{E}[XX^T](M^{-1/2})^T = M^{-1/2}MM^{-1/2} = I$$

and is therefore a white random vector.

If M is singular (and hence not positive definite), the vector X can still be mapped to a smaller white vector Y with m elements, where m is the number of non-zero eigenvalues of M.

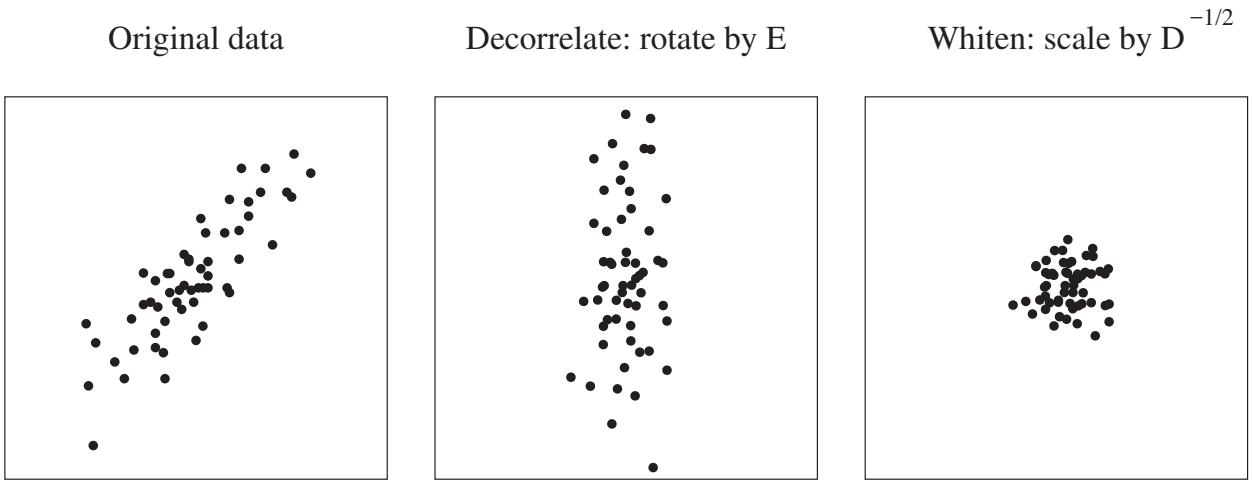


Figure 20.6: Sphering a data distribution.

A first underlying principle of PP is: the level of interest is proportional to the degree of nonlinear structure in the projection. **Normal distributions have no interest** and can be discounted by **whitening** or **sphering** the data points before proceeding with PP.

In many cases, one demands that PP results are invariant with respect to affine transformations of the data. In the real-world, an affine transformation can be related to changing offset and scale of measurement units (like in a Celsius to Fahrenheit conversion of temperatures) and it would be unpleasant if such a trivial change could create or destroy interesting structure (but remember that PCA results are going to be changed by an affine transformation: therefore PCA has to be used with great care and a high level of danger).

20.6.2 Index to measure non-normality

If it is confirmed that normal distributions and not of interest, one must pay attention to the fact that *most* projections have a natural tendency to obtain normal distributions.

This natural tendency is caused by the **central limit theorem (CLT)**[289]: given certain conditions, the arithmetic mean of a sufficiently large number of iterates of independent random variables will be *approximately normally distributed*, regardless of the underlying distribution.

The fact that most projections (arithmetic means of scaled single variables) tend to produce normal distribution, and therefore identifying one by manual search is close to impossible for high-dimensional data, is a strong motivation in favor of a systematic search of interesting ones by PP. One is searching for needles (non-normal projections) is a haystack of approximately normal ones.

Most PP applications seek distribution that exhibit clustering or other kinds of nonlinear structures in the main body of the distribution, and not so much in the tails. The most computationally attractive indices of non-normality are based on polynomial moments.

The procedure to obtain this measure of non-normality can be as follows. One seeks a linear combination:

$$X = \alpha^T Z$$

so that the projected probability density $p_\alpha(X)$ is different from a Gaussian (normal) distribution in its main body. After using the normal cumulative distribution function (cdf):

$$\Phi(X) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{X} \exp(-t^2/2)dt$$

and introducing the variable R

$$R = 2\Phi(X) - 1$$

R will be uniformly distributed in the interval $-1 \leq R \leq 1$ if X is normally distributed. In general, the density for the new variable R can be obtained as:

$$p_R(R) = \frac{1}{2} p_\alpha \left(\Phi^{-1} \left(\frac{R+1}{2} \right) \right) \Big/ g \left(\Phi^{-1} \left(\frac{R+1}{2} \right) \right)$$

where $g(X)$ is the standard normal density.

One can then measure the non-uniformity of R as the integral of the squared distance between the real probability density in the new variable R and the uniform probability density equal to $1/2$ over the entire interval.

$$\int_{-1}^{1} \left(p_R(R) - \frac{1}{2} \right)^2 dR.$$

The projection index to be maximized can be taken as a suitable approximation of the above integral. Traditionally, an expansion in terms of Legendre polynomials can be used [134]. It is of comfort that the final results of PP tend to be quite robust with respect to the details of the approximation as well as estimation of densities from data points.

Indices with derivatives which can be computed analytically can be used as ingredient in **gradient ascent** techniques to find local maxima of the index, but global derivative-free optimization schemes can be adopted if derivatives are not present (see Chapters 26 and 25).

The above derivation can be generalized for projections into more than one dimension. As a sub-optimal alternative, the different vector for the multi-dimensional projection can be determined in successive steps, with *greedy* versions of PP. The general idea is to identify a first projection and then to *remove the structure that makes the specific direction interesting*, otherwise one would re-discover the same direction again. The structure can be removed by applying a transformation that renders the projected density a normal distribution in the projected subspace [134], therefore lowering the local optimum responsible for identifying the first direction (see also Chapter 31 for ways of modifying the objective function to escape from previously fund local optima).

If one considers optimization, one has to be careful that a multiplicity of suboptimal local optima can be created by sampling fluctuations (high frequency ripples are superimposed on the main structure of the objective function). Optimization methods combining global search schemes with local convergence are recommended (Chapter 26).

Being an explanatory method, traditional PP can often discover strong nonlinear effects which cause the researcher to look harder at the data aiming at insight and understanding.

In a way, PP is valid when PCA does not offer any significant insight. PCA is very sensitive to the definition of units of measures and is easily misled by the overall distribution of data.

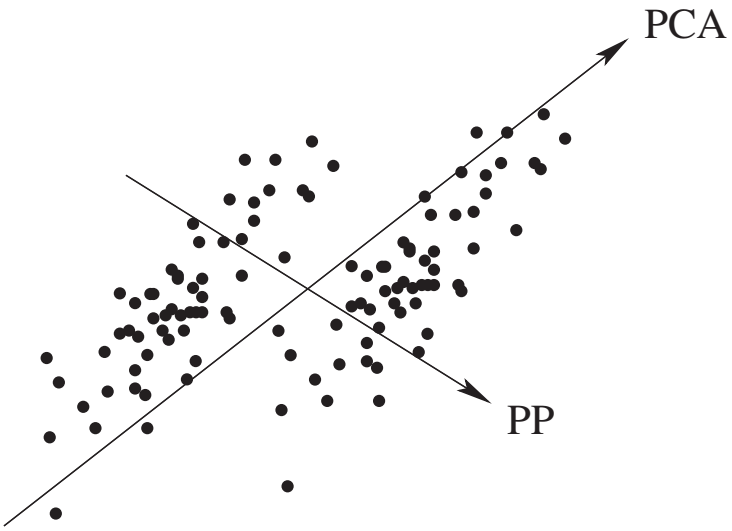


Figure 20.7: PCA and PP can identify very different directions for projecting data. In this case PCA identifies the overall direction of distribution but completely misses the clustering structure of the two clouds of points.

The usefulness of PP in finding projections can be seen in Fig. 20.7. The projection on the projection pursuit direction, which is horizontal, clearly shows the clustered structure of the data. The projection on the first principal component (vertical), on the other hand, fails to show this structure.

Gist

Visualizations (visual representations of abstract data) help the human unsupervised learning capabilities to extract knowledge from the data. Because visualizations are for our visual system, they are limited to the two dimensions of our retina (three in case of stereo vision).

A simple way to transform data into two-dimensional views is through projections (actually, projections can be to more than two or three dimensions if a computer is using the projected points). **Orthogonal projections** can be intuitively explained as looking at the data from different and distant points of view.

Because there are infinite ways to project data, the power of optimization comes to the rescue to select some of them through clear objectives. In particular, **Principal Component Analysis (PCA)** identifies the orthogonal projection that spreads the points as much as possible in the projection plane. In spite of its popularity, PCA can fail to produce relevant insight: having a larger variance is not always related to having the largest information content, or the best possible discrimination.

If **relationships** are available in addition to the raw coordinates (e.g., knowledge that some points are in the same or in different class), they can be used to modify PCA and obtain more meaningful projections.

When class labels are present, **Fisher discrimination** projects data so that the *ratio* of difference in the projected means of points belonging to different classes *divided* by the intra-class scatter is maximized.

Projection pursuit (PP) can be used for exploring data in order to find a few (typically one-two) directions so that the projection identifies interesting nonlinear effects in an automated fashion, guided by an objective function which measures deviations from Gaussian distributions.

If linear relationships identified by the covariance matrix are not considered of interest, a **whitening or sphering** transformation can render the data uncorrelated and with unit variance. What are left are the *nonlinear* effects.

Alchemists used projection to mix powdered philosopher's stone with molten base metals in order to transmute them into gold. You can use it for a more successful enterprise, to transform raw data points into precious and robust insight.

Chapter 21

Feature extraction and Independent Component Analysis

Like other parties of the kind, it was first silent, then talky, then argumentative, then disputatious, then unintelligible, then altogether, then inarticulate, and then drunk.
George Gordon, Lord Byron

The previous chapter considered feature selection: the identification of a subset of informative attributes as a preprocessing phase before building ML models.

Now, in many real-world situations, the relevant measurements do not correspond to individual signals but to *combinations* of them. Human expertise can be complemented by automatic **feature construction** techniques. In some methods, feature construction is embedded in the machine learning process. For examples the "hidden units" of artificial neural networks in MLPs (Sec. 10.1) can be considered as automatically extracted attributes, internal

representations of higher complexity and nonlinearity when one passes from the first to higher hidden layers. Deep learning (Sec. 11.1) aims at building features of progressively higher complexity in an automated fashion. But in this chapter we focus mostly on **explicit preprocessing efforts to construct features** guided by principles and dedicated objective functions to be optimized, often in an unsupervised manner.

For a concrete situation, imagine **two people talking at a cocktail party**, the signal recorded from a microphone will be a mixture of two voice signals, approximately a linear combination with coefficients related to the distance of the speakers. If one could extract from the raw measurements the individual voices, it is clear that subsequent intelligent processing (like natural language comprehension) will be greatly facilitated. Listening to an unborn baby's heartbeat is not direct. An ECG generates a pattern based on electrical activity of the heart, which closely follows heart function, but two signals originating from the mother's and the baby's heartbeats will be superimposed and need to be separated. In a similar way, many physical phenomena are characterized by a superposition of signals, an electroencephalogram (EEG) measurement by electrode on the human scalp contains contributions from many different brain regions, the height of a person is influenced by many causes (nutrition, genetics, work, age, etc.), the cash flow of a business depends on a variety of factors, in an image the intestity values are caused by the combinations of different objects and illumitation sources, in biology a gene expression level may be considered the sum of many different biological processes. In many application areas the **measurements provided by a device contain interesting phenomena mixed up, often in an approximately linear manner** [348, 202].

Identification of the basic factors (latent variables) controlling the output variability has value both for the subsequent intelligent analysis and for understanding more about the basic causes of a phenomenon.

One is left with the following challenge: can one identify a certain number of basic factors from a number of measurements in which the various factors are (often linearly) confused? Can one separate the source signals without a precise knowledge of the different signals (without requiring supervised learning) but only with generic high-level assumptions? **Blind Source Separation (BSS)** is a term used in signal processing to estimate individual source components from their mixtures at multiple sensors. It is called *blind* because it does not require any other information besides the mixtures (and some high-level structural assumptions).

In the language of Machine Learning, one is dealing with a special kind of **feature extraction**, in which the extraction is some combination of the basic raw figures, for example a suitable linear combination. This is different from feature selection in which no combination of different features is considered.

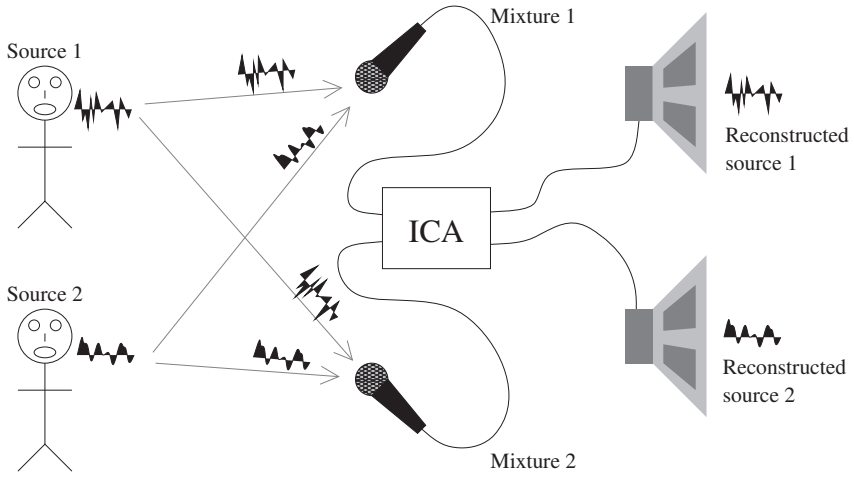


Figure 21.1: A practical context related to Independent Component Analysis. Two people are talking at a cocktail party.

In the following sections, first a review of simple "first-aid" feature transformation is given (Sec. 21.1). Then Independent Component Analysis (ICA) is presented as a way to identify basic hidden variables (sources) producing

measured signals, in the assupmtion of independence and non-Gaussianity (Sec. 21.2). ICA searches for a linear transformation. More complex, possibly non-linear, transforms of raw features are considered when one aims at maximizing the Mutual Information between constructed features and output (Sec. 21.3). The transform is parametric, and the best parameter values can be identified by optimization.

This crucial feature-construction step can be the initial phase of a more complex process. After an initial set of features are constructed, they can be validated and selected with the feature selection techniques already explained in Chapter 7.

21.1 Simple preprocessing for feature extraction

In some cases, raw features make the machine learning problem difficult in a masochistic way. For example, a bad choice for the units of measurements (millimeters, kilometers, light-years) can cause large numerical errors in many algorithms (like gradient descent) or render some inputs negligible even if they contain precious information. Simple preprocessing methods should always be applied before considering more advanced feature extraction techniques. A list derived from [167] is the following one.

Standardization Consider inputs with widely different ranges, for example one measuring distances in meters, one measuring in light-years, or with a different offset like Celsius and Fahrenheit measurements of temperature. A classical centering and scaling of the data is often used:

$$x_i \leftarrow (x_i - \mu_i)/\sigma_i,$$

where μ_i and σ_i are the mean and the standard deviation of feature x_i over training examples. Of course, the same transformation will then be applied to new cases in generalization (with means and standard deviations measured on the *training* set).

Normalization If $\mathbf{x}$ is the vector of inputs and one thinks that the *direction* of the vector is more meaningful than its *magnitude* for subsequent processing, the vector should be divided by its norm:

$$\mathbf{x} \leftarrow \mathbf{x}/\|\mathbf{x}\|$$

where a suitable norm is considered. An example is the case where $\mathbf{x}$ describes a histogram of colors contained in an image (x_i is number of pixels with color i). It makes sense to normalize $\mathbf{x}$ by dividing it by the total number of counts in order to remove the dependence on the size of the image.

Whitening If one suspects that linear relationships are not significant a whitening or sphering transformation can be applied (Sec. 20.6.1). As in all cases, care must be used: in many cases linear relationships tell most of the story!

Signal enhancement by application-specific transforms or local filters The signal-to-noise ratio may be improved by applying signal or image-processing filters like background removal, de-noising, smoothing, or sharpening. A simple example of local filters are edge detection or edge enhancement, e.g., by convolutional methods using hand-crafted kernels. More complex but widely used global transformations techniques are Fourier transform and wavelet transforms.

Monomials with raw features If one suspects nonlinear dependencies, one can try to *increase* the dimensionality of the data by adding products of the original features $x_{k1}x_{k2}, ..., x_{kM}$, in the hope that a simpler model will be sufficient after this step (for example traditional least-squares or SVM).

Quantization of values In some cases, passing from a real value to a small set of integers can reduce noise and simplify processing (for example, to measure Mutual Information, as described in Sec. 7.6).

Transformation of variable type In some cases, categorical data should be transformed into numerical data. For example, if the age of a person is initially described with keywords ("child", "adolescent", "adult", "senior"), assigning numbers (1,2,3,4) will allow for using metric comparisons. In other cases, the contrary makes sense, if the initial numerical data is not related to metric comparisons but is to be intended as an ID for different categories, like people used to do in the old days, when memory was costly and using ID instead of keywords lead to some economy.

Other ways to construct features are described in different chapters, for example Principal Component Analysis (Sec. 20.2), Projection Pursuit (Sec. 20.6), Deep Learning (Chapter 11).

21.2 Independent Component Analysis (ICA)

The success of Independent Component Analysis (ICA) depends on a plausible assumption regarding the nature of the physical world: **the basic variables or signals are independent and non-Gaussian (non-normal)**. Independence means that the value of one signal cannot be used to predict anything about the other signals. Linear combinations of Gaussian variables are also Gaussian, this is why non-Gaussianity is important to identiify particular (optimal) linear transforms.

At a party, one assumes that the value of a voice signal from a speaker cannot be used to predict the voice signal of a second speaker. Like all approximations, this is not completely true, a speaker may start talking only when a second speaker is silent, but ICA methods tend to be quite robust. In practice, most measured signals are derived from many independent physical processes, and are therefore mixtures of independent signals. The emphasis on identifying structured (non-Guassian) distributions is shared with the Projection Pursuit method (Sec. 20.6). This is also what distinguishes ICA from Principal Component Analysis (PCA) described in Sec. 20.2 .

ICA is an unsupervised, **exploratory, or data-driven method**: one can simply measure a phenomenon without specific detailed knowledge, investigate the structure of the data when suitable hypotheses are not yet available.

Following [202], let the observed variables be $x_i(t), i = 1, ..., n, t = 1, ..., T$. Index t can be interpreted as time, or index of different observations.

One assumes that the observed variables can be modelled as linear combinations of hidden (latent) variables $s_j(t), j = 1, ..., m$, with some unknown coefficients a_{ij},

$$x_i(t) = \sum_{j=1}^{m} a_{ij} s_j(t), \text{forall} i = 1, ..., n.$$

At a first look, the puzzling fact is that we observe only the variables $x_i(t)$, whereas both the mixing coefficients a_{ij} and the independent components $s_i(t)$ are to be estimated (no supervised knowledge of $s_i(t)$ is available). For sure, one can hope to estimate each component only up to a multiplying scalar factor: the same results is obtained by multiplying sources by α and dividing mixing coefficients by the same value (in measurements, this corresponds to using different fundamental units of length, mass, time, etc. - trivial changes indeed). In addition, the result is the same by permuting components and rows of the mixing matrix, one cannot expect a standard ordering of the components.

Let's forget about time and consider the x_i and the s_i as realizations of random variables. The different n measures x_i and s_i can be collected in vectors $\mathbf{x}$ and $\mathbf{s}$ so that the model becomes:

$$\mathbf{x} = \mathbf{A}\mathbf{s}$$

The main breakthrough in the theory of ICA was the realization that the puzzle can be solved by making **the unconventional assumption that the independent components are not Gaussian**, in addition to the assumtion that the components s_i are statistically independent (i.e., the joint probability density $p(s_1, ..., s_m)$ is the product of the marginal densities $\prod_j p_j(s_j)$). The second assuption appears also in standard Factor Analysis (a statistical method like PCA used to describe variability among observed, correlated variables in terms of a potentially lower number

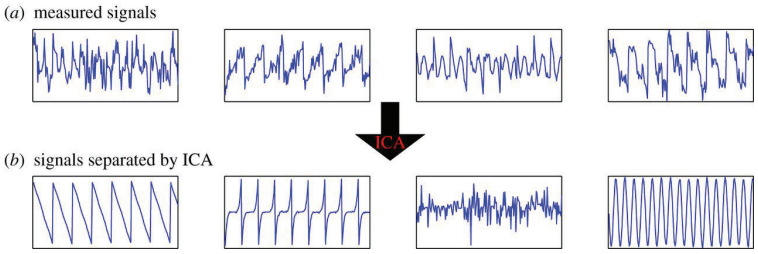


Figure 21.2: ICA objective is to recover from the different measurements in (a), the original source signals that were mixed together, as shown in (b). Source: [202].

of unobserved variables called factors) in which the factors are assumed uncorrelated and Gaussian, which implies statistical independence. Only in the non-Gaussian case independence means more than uncorrelatedness (lack of linear correlation).

As in the discussion about Mutual Information (Sec. 7.6), one can have variables with no linear correlation but high mutual dependency. As an example, consider a distribution which places equal probability to four points located in a cross-like manner in two dimensions: $(-1,0),(1,0),(0,-1),(0,1)$, the correlation is zero but knowledge of one variable (e.g., of x_1) helps when predicting the second one (e.g., $x_1 = -1$ implies $x_2 = 0$), a situation of positive mutual information.

The steps in ICA are the following ones. First, a **whitening** transformation (Sec. 20.6.1) discounts the "trivial" linear relationships to create white variables with zero correlation and unit variance.

The whitening matrix $\mathbf{V}$ can be easily found by PCA, the transformed matrix $\tilde{\mathbf{A}} = \mathbf{VA}$ is now orthogonal, and the transformed variables are obtained as: $\mathbf{z} = \tilde{\mathbf{A}}\mathbf{x}$

After whitening, one can constrain the estimation of the mixing matrix to the space of orthogonal matrices, which reduces the number of free parameters and makes numerical optimization faster and more stable.

Let's note that whitening is not uniquely defined. If $\mathbf{z}$ is white, then any orthogonal transform $\mathbf{Uz}$, with $\mathbf{U}$ being an orthogonal matrix, is white as well. Mere information of uncorrelatedness does not lead to a unique decomposition and the assumpton of non-Gaussianity is crucial. For Gaussian variables, uncorrelatedness implies independence, whitening exhausts all the dependence information in the data, and we can estimate the mixing matrix only up to an arbitrary orthogonal matrix. For non-Gaussian variables, on the other hand, whitening does not at all imply independence, and there is much more nonlinear information in the data than what is used in whitening.

In fact, one can estimate $\tilde{\mathbf{A}}$ by maximizing some objective function that is related to a measure of non-Gaussianity of the components. As usual, a clear definition of the objectives is sufficient to tap the power of optimization. The main approaches to define an objective function for ICA are maximum-likelihood estimation [296], and minimization of the mutual information between estimated component signals [96], leading to similar objective functions.

The derivation based on the Mutual Information is as follows. After remembering equation (7.12) about the role of the Jacobian determinant in changing entropy after a transformation (because probability densities are transformed by volume changes), and the fact that the ICA transformation is linear (and therefore the Jacobian does not depending on $\mathbf{z}$), one gets ($\mathbf{s} = \tilde{\mathbf{A}}\mathbf{z}$) :

$$I(s_1,...,s_m) = \sum_j H(s_j) - H(\mathbf{s}) = \sum_j H(s_j) - H(\mathbf{z}) - \log|\det\tilde{\mathbf{A}}|. \tag{21.1}$$

If one constrains the s_i to be uncorrelated and of unit variance then $\det\tilde{\mathbf{A}}$ has to be constant (volumes cannot change if the whitening-sphering transformation has to be preserved). But $H(\mathbf{z})$ is also constant w.r.t. the transformation weights and therefore one is left with a sum of entropies of the transformed variables (the hidden sources to be determined) [203].

If one remembers the definition of entropy as minus the average of the *logarithm* of probabilities (Sec. 7.6), assumes that probability is distributed equally on the measurements and neglects multiplicative constants, the objective

function is usually formulated in terms of the inverse of the orthogonal matrix $\tilde{\mathbf{A}}$ (if a matrix is orthogonal its transpose is equal to its inverse), whose rows are denoted by $\mathbf{w}_i^T$, as:

$$\text{ICAobjective}(\mathbf{w}_i) = \sum_{i=1}^{n}\sum_{t=1}^{T} \log \text{pdf}_i(\mathbf{w}_i^T \mathbf{z}(t)),$$

where pdf_i is the probability density function of s_i, s_i being estimated by $\mathbf{w}_i^T \mathbf{z}(t)$.

This objective function depends only on the marginal densities of the estimated independent components $\mathbf{w}_i^T \mathbf{z}(t)$. Each term $\sum_{t=1}^{T} \log \text{pdf}_i(\mathbf{w}_i^T \mathbf{z}(t))$ can be interpreted as a measure of non-Gaussianity (entropy) of the estimated component. It is an estimate of the negative differential entropy of the components, which is *maximized* for a Gaussian variable (for fixed variance). Here one is going in the contrary direction, therefore away from Gaussianity!

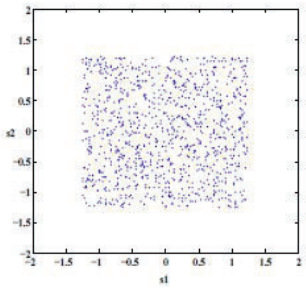


Figure 5: Original sources

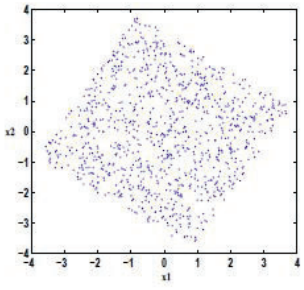


Figure 7: Joint density of whitened signals obtained from whitening the mixed sources

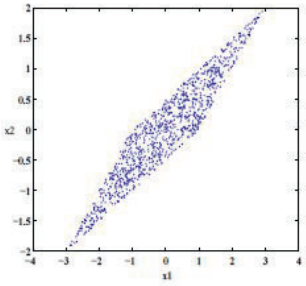


Figure 6: Mixed sources

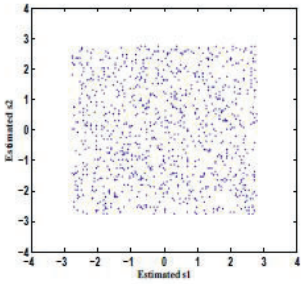


Figure 8: ICA solution (Estimated sources)

Figure 21.3: ICA at work. Original sources (a), measurements (b), whitened measurements (c), rotation to identify sources (c). For Gaussian signals, whitening would produce a sphere, no way to identify an optimal rotation leading to non-Gaussian signals!

Maximizing the nongaussianity of $\mathbf{w}_i^T \mathbf{z}(t)$ thus gives us one of the independent components. In fact, the optimization landscape in the m-dimensional space of vector $\mathbf{w}$ has $2m$ local maxima, two for each independent component, corresponding to s_i and $-s_i$ (recall that the independent components can be estimated only up to a multiplicative sign). To find several independent components, one needs to find all local maxima. Because the different independent components are uncorrelated, one can always constrain the search to the space that gives estimates uncorrelated (orthogonal) with the previous ones.

Rough approximations of the *log-pdf* are used in practice for computational reasons, and robust and fast optimization schemes are proposed for example in [201] (FastICA).

Fortunately for ICA, non-Gaussianity is quite widespread in many applications dealing with physical measurements, and therefore ICA has become a standard tool in machine learning and signal processing in the last decade. Some recent developments in ICA are: analysis of causal relations (structural equation modelling and identification of Bayesian networks), testing independent components (robustness of identification w.r.t. chance), analysing multiple datasets (like EEG signals related to different patients), modelling dependencies between the components (generalizing beyond the independence assumption), special versions for non-negative variables, time-varying signals, etc. [202].

Precise estimators of mutual information (MI) to find the least dependent components in a linearly mixed signal is considered in [347], using a recently proposed k-nearest-neighbor-based algorithm for the MI estimator. The obtained MILCA method (mutual-information-based **least dependent component** analysis) relaxes the assumption of strict independence, henceforth the name "least dependent". After preprocessing the data (by centering, whitening, etc.), the "grouping property" of MI:

$$I(X, Y, Z) = I((X, Y), Z) + I(X, Y).$$

together with invariance under homeomorphisms of a single variable are used to obtain an incremental rule:

$$I(X', Y', Z, ...) = I(X, Y, Z, ...) + [I(X', Y') - I(X, Y)].$$

Since any such transformation in m dimensions can be factorized into pairwise transformations, this means that one only has to compute pairwise MIs for the minimization. Thus one needs to compute the full high-dimensional MI only once. In particular, any rotation can be represented as a product of rotations which act only in some 2x2 subspace. MILCA works with actual dependencies between reconstructed sources (as measured by mutual information). The estimated dependencies can be used to cluster sources, gaining additional insight. It is applied in the cited paper to a fetal ECG recording from the abdomen of a pregnant woman to extract a clean fetal ECG. Clustering identifies the two groups of mother- and child-related sources.

STÖGBAUER *et al.*

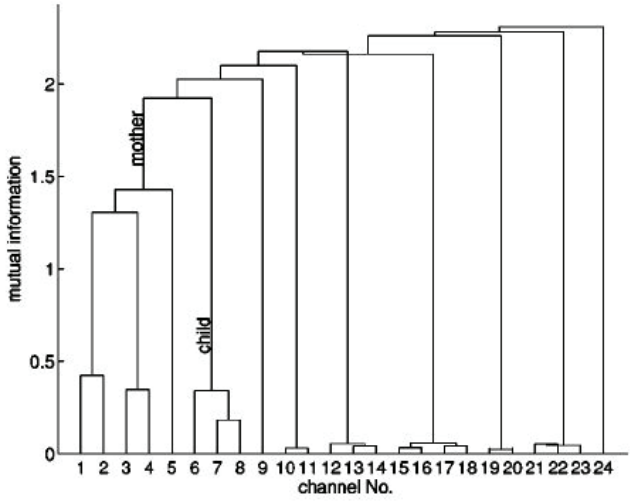


FIG. 18. Dendrogram for Fig. 17. Heights of each cluster correspond to $I(X, X)$ of the cluster $ii(k=3)$

Figure 21.4: MILCA: Dendrogram built from the Mutual Information between sources. Heights of each cluster correspond to $I(X_i, X_j)$.

21.2.1 ICA and Projection Pursuit

As the careful reader probably noticed, ICA is deeply related to Projection Persuit (Sec. 20.6) Projection pursuit aims at finding "interesting" projections of multidimensional data for optimal visualization, density estimation and regression. In one-dimensional projection pursuit, one searches for directions such that the projections of the data have interesting distributions with non-trivial structure. In many cases the Gaussian distribution (in some cases corresponding to a constant factor modified by ramdom errors in the measurement apparatus) is the least interesting one, and the most interesting directions are those that show the **least Gaussian distribution**. This objective is shared by the ICA model.

As a difference to be noted, no data model or assumption about independent components is made in standard projection pursuit. If the ICA model holds, optimizing the ICA non-Gaussianity measures produces independent components; if the model does not hold, then what one gets are the projection pursuit directions.

21.3 Feature Extraction by Mutual Information Maximization

As mentioned in Sec. 7.6, the Mutual Information measure can be used to select features, in a manner which is independent of the particular ML model [21]. In the same way, the Mutual Information can be used to construct informative and non-redundant features. In supervised classification or regression, one can design parametric functions of the basic features. **Optimization of the Mutual Information between the constructed parametric features and the output** can be used to determine an optimal configuration of the parameters.

Estimating MI from a finite set of examples is computationally demanding. Therefore suitable approximations are usually derived to obtain affordable and effective algorithms. For example, a quadratic divergence measure, which does not require prior assumptions about class densities is considered in [362].

Finding a transform to lower dimensions might be easier than selecting features, which is by definition a discrete process, given an appropriate criterion that measures the joint "importance" of a set of features. If the criterion is differentiable with respect to the parameters of the transform, and if the transform is smooth, then one can learn the transform by some form of gradient-descent-optimization of the criterion. ICA (Sec. 21.2) can be used as a tool to find "interesting" projections of the data by finding linear projections that looks rich of clusters and non-Gaussian structure, but it is completely unsupervised with regard to the class labels, and may not be optimal to enhance class separability.

If $\mathbf{y}_i = g(\mathbf{w}, \mathbf{x}_i)$ is a feature transform parametrized by $\mathbf{w}$, the goal is now to find a differentiable estimate of the Mutual information from the examples so that gradient descent can produce better and better values of $\mathbf{w}$, as in Fig 21.5.

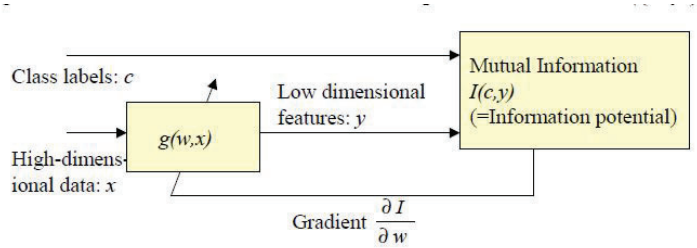


Figure 21.5: Learning feature transforms by maximizing the mutual information between class labels and transformed features (from [362]).

If the aim is not to compute an accurate value of the entropy of a particular distribution, but rather to find a distribution that maximizes or minimizes the entropy given some constraints, then a large number of entropy measures alternative to the original Shannon's definition can be used, producing the same distribution as the result of optimiza-

tion [229]. In particular, **Renyi entropy** of order α is defined as:

$$H_{R_\alpha(X)} = \frac{1}{1-\alpha} \log \left(\sum_{i=1}^{n} p_i^\alpha \right)$$

where $\alpha \geq 0$ and $\alpha \neq 1$. As α approaches zero, the Renyi entropy increasingly weighs all possible events equally, regardless of their probabilities. The limit for $\alpha \to 1$ is the Shannon entropy.

It turns out that Renyi's quadratic measure (a function of the square of the density function), when combined with Parzen density estimation method using Gaussian kernels, provides significant computational savings: it can be estimated as a sum of local interactions, as defined by the kernel, over all pairs of samples. After completing the exercise to calculate derivatives in the given scheme [362] (Parzen windows and Renyi entropy), the method can be applied both to linear and non-linear tranforms.

The two main advantages of the method are that: i) it provides a non-parametric estimate of the mutual information without simplifying assumptions, like Gaussianity about the class densities, ii) it is usable with training datasets of the order of tens of thousands of samples. In static pattern recognition tasks, such feature transforms can extract more discriminatory information from the source features than alternative techniques like Fisher linear discriminant analysis (LDA) (Sec 20.5). Judging from the experiments, the method works extremely well only in transforms to low dimensions, approximately less than 10, probably because of limits in the the Parzen density estimation.

Gist

Feature extraction (or construction) goes beyond feature selection, aiming at **building more interesting features** from the raw data, in order to facilitate further processing and, hopefully, gain more insight about the modeled process.

Some **"first-aid" techniques** like normalization, quantization, application of application-dependent local or global filters should not be neglected, otherwise one risks masochism.

Raw measurements are often a combination (approximately linear) of basic signals or **hidden variables**. Identifying these factors of variations which explain the measurements variability, is invaluable. In the assumption of statistically independent signals and non-Gaussian (structured, high-entropy) probability distributions, the signals causing the phenomenon can be identified even before supervision is applied (**Independent Component Analysis**). Mixtures of source signals are almost always Gaussian (central limit theorem), and it is fairly safe to assume that non-Gaussian signals must, therefore, be source signals. Gaussian variables are forbidden for ICA!

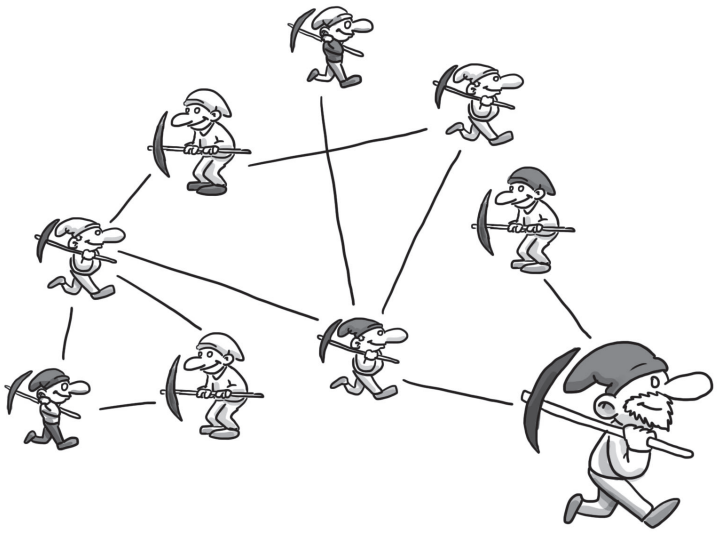
More complex, also non-linear, features can be constructed by considering parametric feature extractors, and by determining suitable parameters via **maximization of the Mutual Information between extracted features and output**. This is more easily said than done as Mutual Information is not easy to estimate, suitable approximations are often considered in practice.

If you listened to your unborn baby's heartbeat, now you know which methods you have to thank.

Chapter 22

Visualizing graphs and networks by nonlinear maps

No man is an Iland, intire of it selfe; [. . .]
any mans death diminishes me, because I am involved in Mankinde;
And therefore never send to know for whom the bell tolls;
It tolls for thee.
(John Donne, 1623)



After considering visualizations based on linear projections in Chapter 20, let's now consider more general ways to feed the human unsupervised learning capabilities and derive *insight* from data. We assume that the n entities to be displayed are not necessarily characterized by internal coordinates, but only by item-to-item (i.e., *external*) **relationships** such as dissimilarities between two items i and j denoted by d_{ij}. If items do have coordinates, such external relationships can be obtained by simple ways, as explained in Sec. 17.2.

In the general case, however, the external dissimilarity measure is not computed as a distance, and may not be available for every pair of items. An appropriate model for this situation is an **undirected weighted graph** $G(V, E)$, given by a set of vertices (or nodes) V, and edges $E \subset V \times V$. Each entity is represented by a node, and a connection (i, j) labeled d_{ij} is present between two nodes if and only if a dissimilarity is defined for the corresponding entities, as shown in Fig. 20.1. We assume that similarities are positive but we do not consider any other assumption (like the validity of triangular inequalities). For example, in marketing the similarity between two products could be derived by sampling customers and asking them to evaluate the product similarity along a given scale.

22.1 Multidimensional Scaling (MDS) Visualization by stress minimization

Given our eyes, visualization is only in two or three dimensions. The aim is therefore to place items on the 2D plane (or in 3D space) so that their **mutual distances are as close as possible to their dissimilarities**. In general, a perfect placement obeying all dissimilarities is impossible; therefore, a precise criterion is needed in order to define what placements can be considered as acceptable.

The problem is the following: given a set of items with (positive) dissimilarities d_{ij}, find the two-dimensional or three-dimensional coordinates $\boldsymbol{p}_i$ for all items that provide a convenient placement, one preserving the original dissimilarities as much as possible. The simplest objective is **stress minimization**, stress being the amount by which a *visualized* dissimilarity is compressed or expanded with respect to the original dissimilarity. It is intuitive, physical, and useful also as a starting point to understand more complex approaches.

A straightforward error measure quantifies by how much the distances in the plane are different with respect to the original dissimilarities. For simplicity we consider a two-dimensional visualization.

Let $\delta_{ij} = \sqrt{(\boldsymbol{p}_i - \boldsymbol{p}_j)^T(\boldsymbol{p}_i - \boldsymbol{p}_j)}$ be the distance between the coordinates of items i and j on the plane. A natural *global mapping error* can be defined as the sum of the squared individual errors:

$$\sum_{(i,j)\in E} (d_{ij} - \delta_{ij})^2.$$

No contribution to the error is present for missing edges (for couples of points without dissimilarity values). Additional flexibility can be obtained by adding an arbitrary weight w_{ij} representing the impact that an individual error has on the overall stress:

$$\text{global mapping error} = \text{Stress} = \sum_{(i,j)\in E} w_{ij}(d_{ij} - \delta_{ij})^2. \tag{22.1}$$

For example, if $w_{ij} = 1/{d_{ij}}^2$, one considers the *relative* errors $(\delta_{ij} - d_{ij})/d_{ij}$ instead of the absolute errors. The value $w_{ij} = 1$ is the default.

An exact solution reproduces all original dissimilarities: $\delta_{ij} = d_{ij}$, with zero error. Low error means that many distances tend to be rather close to the original ones. The problem is now to *minimize* the *global mapping error* by changing the point positions $\boldsymbol{p}_i$. The freedom in placing point in two dimensions is complete, and therefore the optimization problem has a very large number of dimensions, equal to twice the number of entities. The situation is different in Chapter 20 with mappings executed by a linear projection.

There is a nice physical model related to minimizing the above *global mapping error*, explaining the term *Stress* to denote the function to be minimized. A spring is attached to each pair of points with a length at rest equal to d_{ij}, the desired distance, and with an elastic constant (resistance to deformation) equal to the weight w_{ij}. The term $w_{ij}(\delta_{ij} - d_{ij})^2$ can be considered as the potential energy of a spring which is elongated or compressed with respect to the rest length. The initial position of the points can be chosen randomly and the movement is constrained to two dimensions. The system will start oscillating and, provided that some friction is present, oscillations will gradually be damped, leading to a stable situation, a locally optimal configuration of the overall stress function.

The physical system can of course be simulated on a computer, leading to what is called the **force-directed approach for drawing graphs**. Methods based on this approach consist of two main components. The first is the

model that quantifies the quality of a drawing (or of the two-dimensional map if you prefer a more technical term). The second is an **optimization method** for computing a drawing that is locally optimal with respect to this model. The resulting final layout brings the system to equilibrium, so that the total force on each vertex is zero, or equivalently, the potential energy is locally minimal with respect to the vertex positions.

If you do not like Physics but prefer Math, you can forget about simulating physical details like friction and concentrate on minimizing the stress function through gradient descent: calculate partial derivatives and use an optimization method to reach a global optimum, again *optimization is the source of power*!

Figure 22.1: 2D Visualization by stress minimization.

Examples of visualizations are in Fig. 22.1, which shows a **social network** of friends interested in different mountaineering activities, and in Fig. 22.2, which shows a social network of politicians with similarities related to their activity in the parliament. Note how the main political groups are automatically *clustered* as an interesting side-effect of encouraging the placement of similar people in similar positions. Through a **focus and context visualization**, one can either concentrate on the local network of connections around a single politician (*focus*), or see also the *context* given by the complete set of connections (Fig. 22.3). It is therefore simple to navigate from an entity to a neighbor, to a neighbor of a neighbor, etc., to likewise track complex relationships in a fast and effective manner. Criminal investigations are another possible application of this method.

In addition to visualization, reducing data dimensionality by finding a mapping from the original space to one with a smaller number of dimensions (also called **Multidimensional Scaling (MDS)** [244] has value also to extract relevant features for subsequent processing by ML. In this case, the mapping is defined via parametric definitions so that is can be applied also to new points, when the ML system has is used for new points in generalization mode [61].

22.2 A one-dimensional case: spectral graph drawing

A paradigmatic case consists of mapping the set of n points to one dimension while keeping similar points as close as possible. As usual, one needs to define a quantity to minimize, related to the goodness of the one-dimensional drawing. Let x_i be the one-dimensional coordinate assigned to point i (and let $\boldsymbol{x}$ denote the vector of all such coordinates). A

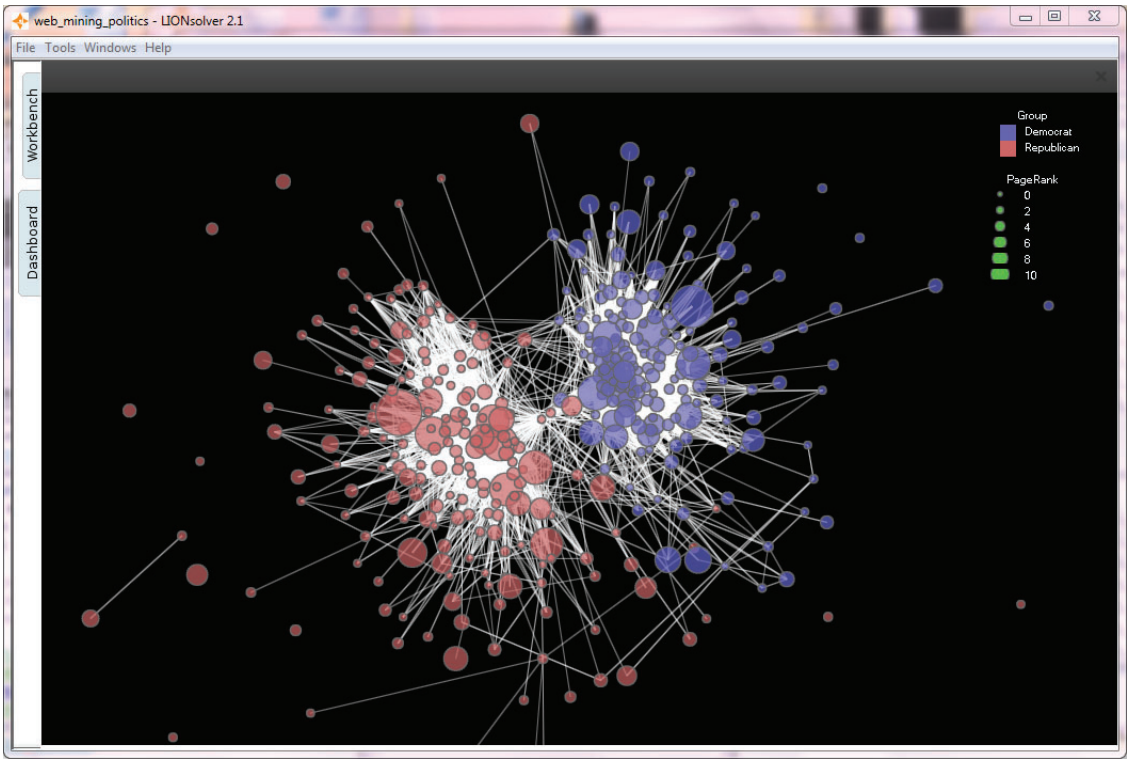


Figure 22.2: Social network analysis: visualizing the network of USA representatives. The two parties (which are not available to the clustering software) emerge as two very different clusters.

useful quantity is **Hall's energy**, first proposed in the 70's:

$$E_{\text{Hall}} = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij}(x_i - x_j)^2. \tag{22.2}$$

The interpretation of this formula is to square the individual distances (so that the function will be differentiable and the differentiation will lead to linear equations) and sum them weighted by the similarities between pairs. When w_{ij} is large the function E_{Hall} gets a large contribution from the $(x_i - x_j)^2$ term and therefore this definition should encourage placing similar points at close positions, to avoid paying a large penalty and obtaining a large E_{Hall} value. Through Hall's energy, large similarities encourage close positions in the placement.

Stop for a minute to identify a serious weakness in the above definition and the proceed. Now one is completely free to pick a coordinate x_i for each point and by picking very small coordinates (or coordinates which are very similar) the energy goes to zero but we are left with a trivial solution: mapping all points to the same position. The definition can be repaired by considering that one is not interested in the *absolute* values of the coordinates but in their *relative* values. As usual, the optimized drawing should not depend on choosing meters or millimeters as units. One

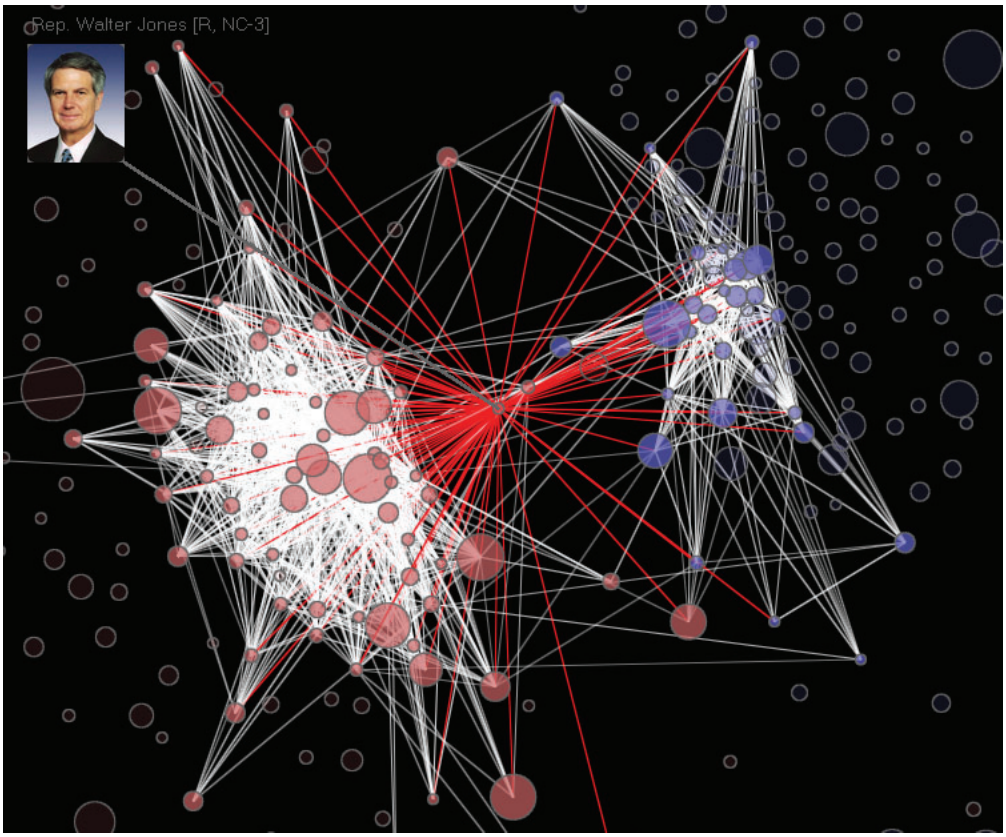


Figure 22.3: Navigating in a social network of politicians: the network of a single representative.

can therefore fix the length of the x vector to one and the problem becomes:

$$\text{minimize} \qquad \left(\sum_{(i,j)\in E} w_{ij}(x_i - x_j)^2 \right) \tag{22.3}$$

$$\text{subject to} \qquad \|\boldsymbol{x}\|^2 = \boldsymbol{x}^T\boldsymbol{x} = \sum_{i=1}^{n} x_i^2 = 1. \tag{22.4}$$

For convenience let $N(i) = \{j|(i,j) \in E\}$ be the neighborhood of node i and $\deg(i) = \sum_{j\in N(i)} w_{ij}$ its weighted degree. After defining the Laplacian matrix L^G associated with the graph:

$$L^G_{ij} = \begin{cases} \deg(i) & \text{if } i = j \\ -w_{ij} & \text{if } i \neq j \end{cases}, \tag{22.5}$$

one can obtain Hall's energy as $E_{\text{Hall}} = \boldsymbol{x}^T L^G \boldsymbol{x}$.

The energy and the constraint are invariant under translation. We can eliminate this degree of freedom by requiring that the mean of $\boldsymbol{x}$ be zero: $\sum_{i=1}^{n} x_i = \boldsymbol{x}^T \mathbf{1}_n = 0$ (where the vector $\mathbf{1}_n$ contains all 1's). Finally, the 1-dimensional optimal layout can be described as the solution of the following constrained minimization problem:

$$\begin{array}{ll} \text{minimize} & \boldsymbol{x}^T L^G \boldsymbol{x} \\ \text{subject to} & \begin{cases} \boldsymbol{x}^T \boldsymbol{x} = 1 \\ \boldsymbol{x}^T \mathbf{1}_n = 0 \end{cases} \end{array} .$$

By standard optimization and linear algebra results, provided that the graph is connected, the resulting minimum value of the energy is the so-called *algebraic connectivity* of the graph, i.e., the second smallest eigenvalue λ_1 of L^G (L^G is singular, therefore the smallest eigenvalue is $\lambda_0 = 0$), while the solution is the corresponding eigenvector $\boldsymbol{v}_1$, also known as the *Fiedler vector*. The result is elegant and it deserves an inspiring name: **spectral graph drawing**, or **spectral layout**. The term "spectral" has nothing to do with ghosts and scary movies but with the usage of eigenvectors and eigenvalues in Physics to study the distribution of energy emitted by a radiant source (spectrum), of vibration modes, etc.

Unfortunately elegance must be neglected when going to more than one dimension. Let's call the second dimension y. A trivial generalization will make the second vector coordinate $\boldsymbol{y}$ the same as $\boldsymbol{x}$, not a big gain: all points will be aligned on the diagonal line, not really a two-dimensional plot. To get something more usable we must force the solution value for the y coordinate to be different from the one for the x coordinate.

A reasonable requirement is to ask that the two coordinate vectors are not correlated ($\boldsymbol{y}^T \boldsymbol{x} = 0$), so that the additional dimension will give us some new information, "new" in the sense that it is not linearly related to the previous values, not in a deep information-theoretic meaning. The problem for $\boldsymbol{y}$ now becomes:

$$\begin{array}{ll} \text{minimize} & \boldsymbol{y}^T L^G \boldsymbol{y} \\ \text{subject to} & \begin{cases} \boldsymbol{y}^T \boldsymbol{y} = 1 \\ \boldsymbol{y}^T \mathbf{1}_n = 0 \\ \boldsymbol{y}^T \boldsymbol{x} = 0 \end{cases} \end{array} .$$

Potential difficulties are inherited from the difficulties in solving very large eigenvector problems. Multi-scale techniques and iterative techniques to calculate principal eigenvectors can help.

In spite of the elegance related to minimizing a simple function with a well known linear algebra result, there are no guarantees that the aesthetic properties of the layout correspond to the user preferences. In particular, there are no requirements forbidding the methods to place too many nodes too close together so that they become hardly visible. Furthermore, there is no guarantee that requiring an uncorrelated y coordinate corresponds to the best aesthetic results.

Real-world layouts typically require energies (functions to be minimized) designed in close agreement with the specific preferences. By defining a clear energy, one separates the concern about **the goal** (the desired layout characteristics) from the concern about **how the goal can be reached**, at least approximately, by optimization techniques.

22.3 Complex graph layout criteria

Let us consider the following simple graph connectivity matrix, where two nodes i and j are connected if and only if the matrix entry (i, j) is 1:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

corresponding to a three-node graph whose only requirement is that node 2 has distance 1 from the other two nodes:

$$d_{12} = d_{23} = 1.$$

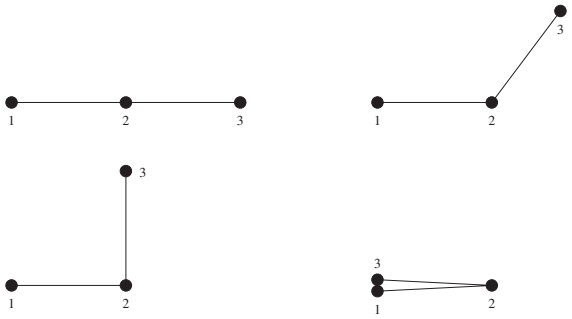


Figure 22.4: Equivalent stress-minimizing layouts when too few constraints are defined.

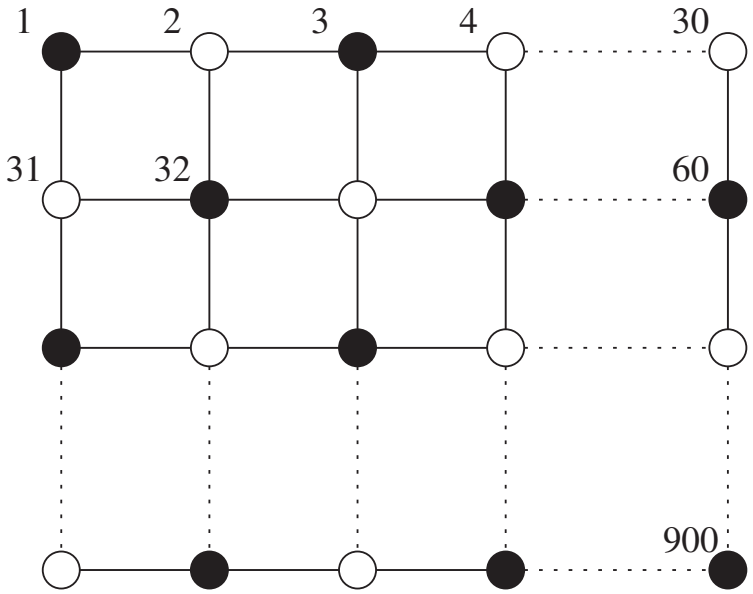


Figure 22.5: A 30×30 lattice with first-neighbor edges.

The layouts shown in Fig. 22.4 are perfectly equivalent from a stress-minimization approach: the absence of an edge (e.g., between nodes 1 and 3) implies that the mutual distance of the corresponding nodes is irrelevant if the energy function to be optimized contains only terms related to *connected* pairs.

The problem is even worse in large graphs where many indifferent pairs of nodes exist. Fig. 22.5 shows a 30×30 rectangular lattice where only first-neighborhood edges are defined (requiring unit distance between the endpoints). An "optimal" layout obtained by minimizing equation (22.1) is shown in Fig. 22.6. Many degenerate locally-optimal layouts exist. They can be obtained by alternately coloring nodes in black and white in a checkerboard fashion, so that black nodes are only connected to white ones and *vice versa*. A one-dimensional solution packing all black nodes at $x = 0$ and all white nodes at $x = 1$ trivially satisfies all distance constraints, so that the *global mapping error* defined in equation (22.1) is zero.

The introduction of a default large distance for unconnected nodes can easily solve the problem. Fig. 22.7 shows the optimal layout obtained by equation (22.1) on the same 30×30 lattice where disconnected nodes i and j have a large required distance $d_{ij} = 20$ with a very small weight $w_{ij} = 10^{-5}$. Observe that, the layout being unknown a priori, it is often difficult to define a convenient default distance. In the case of Fig. 22.7, for instance, the value of 20

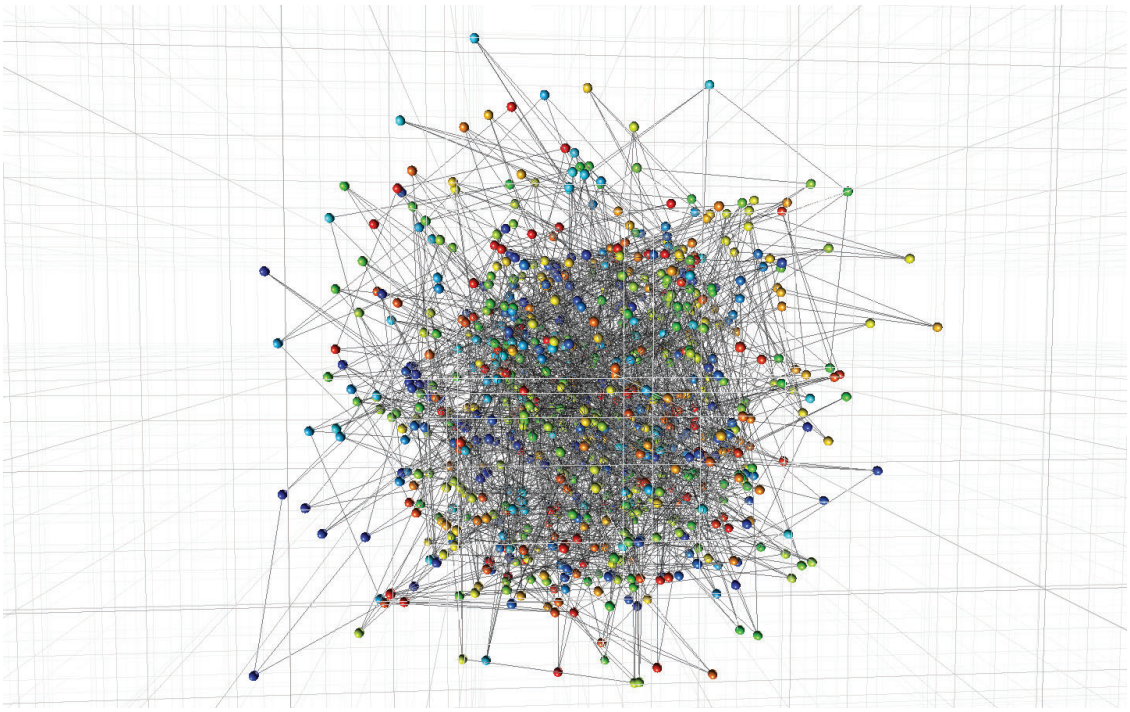


Figure 22.6: An "optimal" layout from the stress-minimization point of view, which is not optimal for understanding the network structure.

is too little to ensure a correct layout, and the overall graph bends into a spherical shape.

A second approach, shown in Fig. 22.8, is the completion of the distance matrix by the *shortest path* calculation. All distances between nodes which are not directly connected are set equal to the *shortest path* between such nodes. For instance, nodes 1 and 32 of the lattice shown in Fig. 22.5 have a shortest path length equal to 2 (one horizontal and one vertical edge). Without the requirement that shortest paths distances are reproduced, nothing prohibits nodes 1 and 32 to be placed at a very small distance in the visualization. As soon as the requirement is active, this bad behavior is discouraged by a large penalty and nodes tend to untangle, passing from configurations like that of Fig. 22.6 to configurations like that of Fig. 22.8.

Notice that the minimum path distance is always larger than the Euclidean distance in the grid layout. As an example, the shortest-path distance between the two diagonally extreme nodes 1 and 900 is $(30-1)\cdot 2 = 58$, while the expected Euclidean distance in the grid layout would be $(30-1)\cdot\sqrt{2} \approx 41.01$, hence the pillow-shaped layout of Fig. 22.8.

More complex functions to be minimized for graph layout consider **additional aesthetic criteria**, like minimizing the number of edge crossings, or guaranteeing a certain minimum angle between edges connected to a node (small angles make readability difficult), or allowing curved edges, etc. A complete enumeration is out of the scope of this introductory chapter. In all cases, after defining in quantitative terms a suitable compromise between the desirable aesthetic criteria, one has to search for an effective minimization algorithm (the source of power!), in most cases looking for an approximate but fast solution.

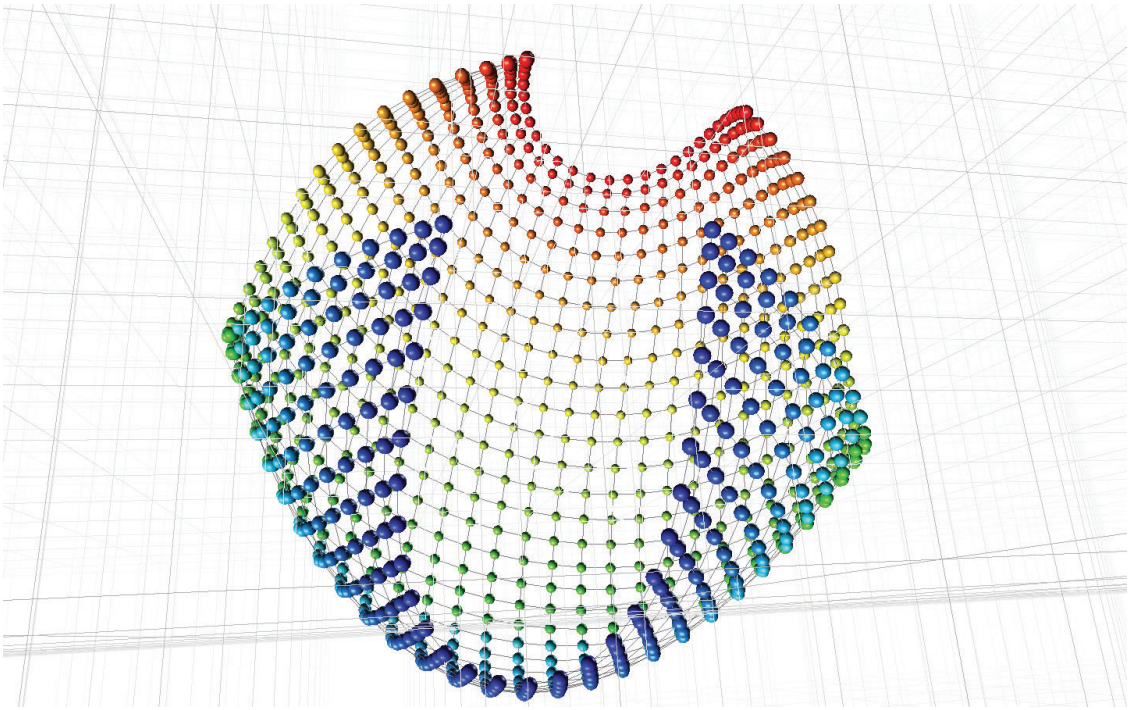


Figure 22.7: First solution to missing constraints: add a default repulsion.

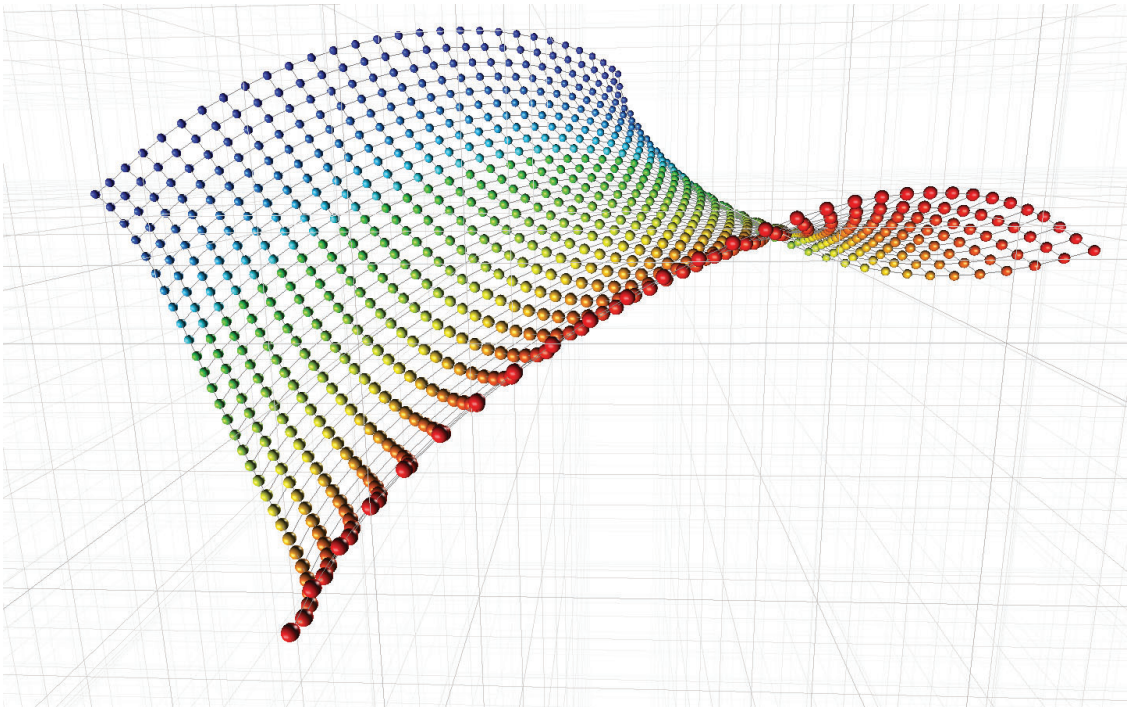


Figure 22.8: Second solution to missing constraints: complete distances by computing the minimum path between nodes.

Gist

Graph layout techniques can be used to visualize relationships between entities.

If some dissimilarities are available, drawing entities in two dimensions so that similar items are close to each other is precious to identify groups (clusters) and relationships between groups.

Stress minimization resorts to a physical model. Each dissimilarity value generates a spring between entities in n dimensions. The task is to "sandwich" the network by squeezing it to a plane, while minimizing the elongation or shortening of the various springs. By the way, if springs are substituted with rigid bars, squeezing becomes impossible: in general an exact solution to map points to a plane and maintain *all* dissimilarity values unchanged does not exist. If you imagine each point as a person at a party, everybody will move on the floor to be away from disagreeable people and close to likeable ones. The fact that everybody moves at the same time can make parties (and visualizations) very stressing (suboptimal).

As for clustering, there is not an absolute best graph (or network) layout. Through optimization one defines the **objectives** (the quantitative meaning of "optimal layout") and then identifies the best possible mapping which maximizes them. Often one tries many possibilities before identifying a proper visualization.

Social network analysis is used to study networks of interacting persons. In business, a similarity between employees can be defined by the number of messages they exchange. If you design a layout of the network of employees with this metric you will easily identify clusters of colleagues working together, maybe loser connections between different groups, and maybe some isolated individuals who either concentrate a lot, or prefer the telephone, or ... are not too committed to the business.

Chapter 23

Semi-supervised learning

A mind is a fire to be kindled, not a vessel to be filled.
(Plutarch)

Let us consider the international airport example which motivated unsupervised learning methods in Chapter 19: you walk through a gate and clearly identify clusters of people speaking different languages, even if the language names are unknown. Now, if *some* people languages are identified, for example if some people are waving flags or wearing costumes of their countries, for sure one could select only the labeled speakers and run a supervised learning algorithm to map phonetic characteristics to languages.

The question now is: can one also use some information from the *unlabeled* people to improve language classification? Let us note that clusters of people usually speak the same language ("birds of a feather flock together") and we may be tempted to label some of the unknown speakers with the same language as the one spoken by at least one member of the same cluster. If the assumption is true, one greatly increases the number of examples and can improve the overall generalization capability of the trained classifier. For example, young children clustered with their older

and identified parents can be added to the database so that even young people voices (usually with higher frequencies) can be correctly classified.

In a similar manner, one can use some supervised data to aid unsupervised learning and clustering. This is the underlying idea of semi-supervised learning: **use both the labeled examples and also (some) unlabeled ones to improve the overall classification accuracy**.

If the assumptions work, one gets a very valuable performance boost in all cases when **labeled examples are scarce and unlabeled ones abundant**. Think for example at web pages: human labeling is very costly and only a minuscule subset of web pages are labeled. By contrast, an enormous and growing number of unlabeled pages is present.

23.1 Learning with partially unsupervised data

Semi-supervised learning (SSL) uses both supervised and unsupervised data to improve performance. The standard form of supervision are **labels** associated with some examples. In this case the training set X is divided into a labeled portion $X_L = \{x_1, \ldots, x_l\}$, for which labels $Y_L = \{y_1, \ldots, y_l\}$ are given, and an unlabeled portion $X_U = \{x_{l+1}, \ldots, x_{l+u}\}$.

Other forms of supervision can be related to constraints or **hints** given to the system [4]. For example, the hints can take the form "the output function must be growing as a function of one input coordinate," while constraints can be formulated as "these two points must be in the same class" (**must-link**) or "these two points cannot be in the same class" (**cannot-link**).

A first idea, originated in the sixties [320] is the so-called self-learning or **self-labeling** method where a wrapper algorithm repeatedly uses a supervised learning method. Initially, learning is executed on the labeled examples. Then, some additional unlabeled examples are labeled by using the current trained system, and learning is repeated by adding the newly labeled examples. Heuristically, one could try adding labels to the examples which are labeled with the largest confidence. Although appealing, the effect of the wrapper depends on the supervised method enclosed and it is unclear when self-labeling is effective.

A context related to SSL was introduced by Vapnik as **transductive learning**. Inductive learning wants to derive a prediction function valid for arbitrary inputs, while transductive learning just aims at predicting only a fixed set of test points, by using all available information. Usually, transductive learning is based on a **labeled graph representation of the data**, labeled nodes are classified training examples, edges represent similarity/dissimilarity relationships or constraints. A combinatorial optimization on the labels to maximize an overall consistency measure is then performed.

In general, SSL looks promising if the unsupervised information about the density $p(x)$ is useful in deriving $p(y|x)$. By analogy with the supervised learning smoothness assumption, the **semi-supervised smoothness assumption** states that if two input points x_1 and x_2 *in a high-density region* are close, the corresponding outputs y_1 and y_2 should also be close. By transitivity, if two points are linked by a path in a high density region (they belong to the same cluster) their outputs should be close. If points are close but in a low-density area, the requirement that outputs are similar is less stringent.

If we equate unsupervised learning with clustering, the **cluster assumption** states that if two points are in the same cluster, they are likely to belong to the same class. In this case, the use of unlabeled points is useful to find boundaries between clusters with better accuracy, and then improve the overall classification by the above assumption.

An equivalent formulation is the **low-density separation assumption**: decision boundaries between different classes should lie in low-density regions, and should not split single clusters, as shown in Fig. 23.1.

The above assumptions correspond very closely to the international airport analogy: if one wants to separate different languages he had better not split clusters of people but draw boundaries in empty areas.

A different paradigm is the assumption that data lie approximately on a **low-dimensional manifold**, as shown in Fig. 23.2. A manifold is a mathematical space that on a small enough scale resembles Euclidean space of a specific dimension. For example, a line and a circle are one-dimensional manifolds, a plane and a sphere (the surface of a ball) are two-dimensional manifolds. More formally, every point of an n-dimensional manifold has a neighborhood homeomorphic to an open subset of the n-dimensional space R^n. The curse of dimensionality for input data of very

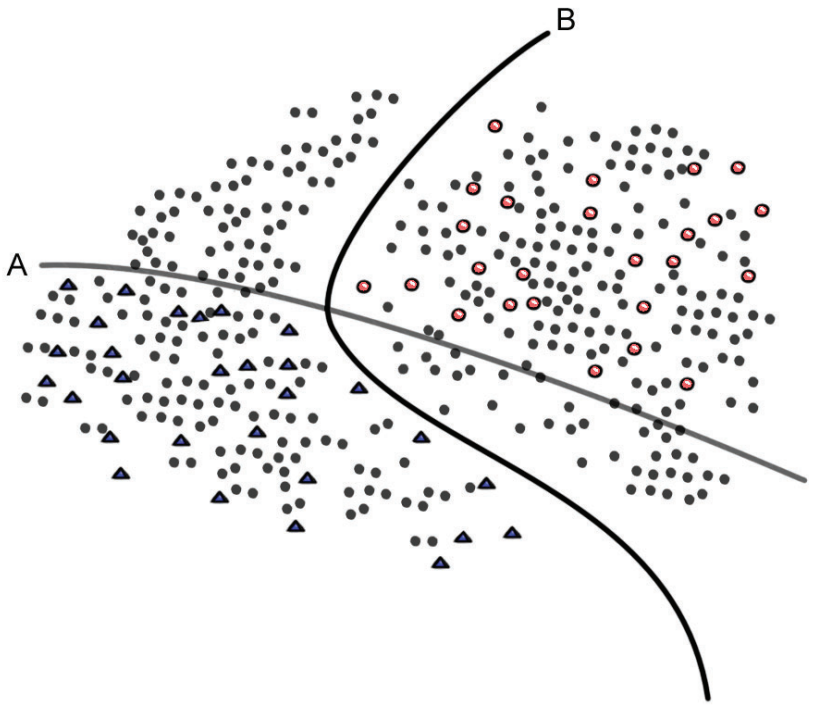


Figure 23.1: Low-density separation assumption. Even if boundary A correctly separates the labeled examples, boundary B is better because it crosses a low-density region. Information about unlabeled data produces a better classification.

large dimensions is avoided by first identifying a manifold where most data lie. Then an appropriate metric is given by **geodesic distances** on the manifold (the analogy is with distances covered by airplanes on the Earth surface), and the standard smoothness assumption is considered on the low-dimensional manifold. The more data is available, the better one identifies the relevant manifold and the corresponding metric to use in supervised learning (consider for example a nearest-neighbor classifier, where the vicinity is given by geodesic distances on the manifold).

23.1.1 Separation in low-density areas

Some SSL techniques are based on encouraging the separation between classes (the decision boundaries) to pass through low-density areas, away from most data examples.

An immediate algorithm is obtained by adopting a margin-maximization algorithm like SVM and maximizing the margin for both labeled and unlabeled examples, this is called **transductive SVM** (TSVM). To the function to be minimized one adds a term like:

$$\lambda_2 \sum_{unlabeled\ data\ i} (1 - |f(x_i)|), \tag{23.1}$$

$f(x_i)$ being the classification function which has to be greater than 1 for one class, less than -1 for the other class. The

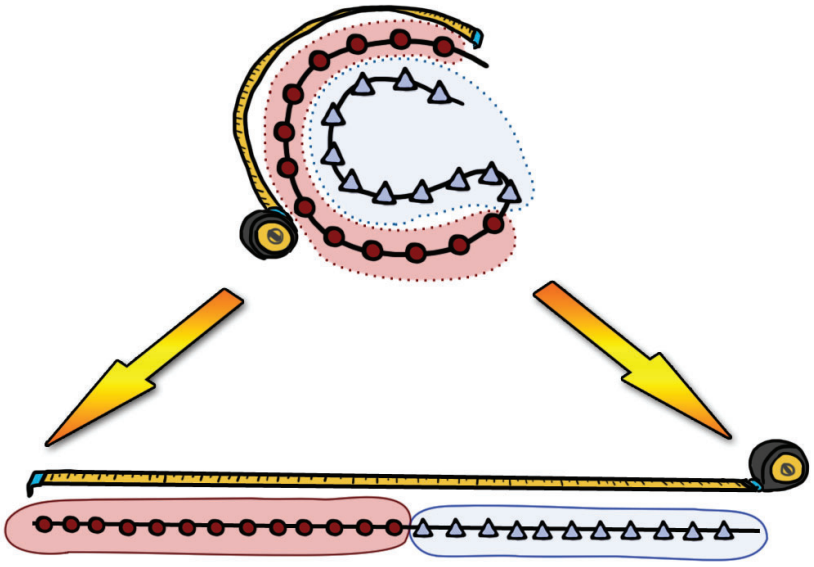


Figure 23.2: The geodesic distance can help to separate classes lying on a manifold.

penalty introduced in the function is of λ_2 when $f(x_i) = 0$, and it linearly becomes equal to zero when $f(x_i)$ becomes 1 or, in the other direction, when $f(x_i)$ becomes -1 (the penalty has a triangular form centered around zero). In other words, a penalty is incurred if an unlabeled data point falls in the "gray" boundary region where $|f(x_i)| \leq 1$: therefore unlabeled data tend to **guide the linear boundary away from the dense regions**. The corresponding problem is not convex and therefore robust heuristic optimization schemes have to be adopted, for example *deterministic annealing* strategies which start from an easy problem and gradually transform it into the TSVM optimization function [333]. The continuation approach of [78] follows a similar paradigm of first optimizing an "ironed" version of the function and then gradually introducing finer and finer details.

23.1.2 Graph-based algorithms

The graph-based methods are based on representing the problem as a graph, where nodes correspond to examples and edges are labeled with the pairwise similarity w_{ij} of two nodes i and j. As usual, one can think in terms of similarities, or in terms of dissimilarities/distances.

An approximation of the **geodesic distance of two points along the manifold** can be approximated by deriving the **minimum-path distances** between couples of points from the initial pairwise distances.

Let's introduce the matrix $\mathbf{W}$ to represent similarities $\mathbf{W}_{ij} = w_{ij}$ if the edge is present, zero otherwise, and the diagonal degree matrix $\mathbf{D}$, so that $\mathbf{D}_{ii} = \sum_j w_{ij}$.

The basic method to encourage **smoothness along light edges** (smoothness when connected nodes are similar) is related to defining and using the **graph Laplacian** operator. The normalized $\mathcal{L}$ and non-normalized combinatorial graph Laplacian operator $\mathbf{L}$ are defined as:

$$\mathcal{L} = \mathbf{I} - \mathbf{D}^{-1/2}\,\mathbf{W}\,\mathbf{D}^{-1/2}, \tag{23.2}$$

$$\mathbf{L} = \mathbf{D} - \mathbf{W}. \tag{23.3}$$

The graph Laplacian is related to the more traditional Laplace operator (denoted with ∇^2) used for continuous functions $f(x_1, \ldots, x_n)$:

$$\nabla^2 \phi = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial {x_i}^2}. \tag{23.4}$$

In fact, the Laplacian matrix of a lattice, when applied to the values of f at the vertices, corresponds to the finite-differences approximation of the continuous operator on a regular grid of points. The Laplacian matrix of a graph can be seen as a generalization of the lattice definition.

The Laplacian $\nabla^2 f(\boldsymbol{x})$ of a function f at a point $\boldsymbol{x}$, up to a constant depending on the dimension, is the rate at which the average value of f over spheres centered at $\boldsymbol{x}$, deviates from $f(\boldsymbol{x})$ as the radius of the sphere grows. Zero means that the average value on a sphere is equal to the value at the center.

A motivation for the Laplacian appearing in physics is that solutions to $\nabla^2 f = 0$ in a region U are functions that make the Dirichlet energy functional stationary:

$$E(f) = \frac{1}{2} \int_U \|\nabla f\|^2 \, \mathrm{d}x. \tag{23.5}$$

The **smoothing action** is clear: one aims at identifying locally optimal configurations that minimize the average square of the gradient modulus. Once again, the optimization point of view clarifies the meaning.

Iterative ways to solve the above $\nabla f = 0$ equation on a grid involves repeatedly substituting the value at a grid point with a weighted average of the values on its neighbors.

A similar smoothing action is obtained on graphs, the goal is to obtain a distribution of values on the graph so that **the value at a node is equal to the weighted average of the neighboring values**.

A semi-supervised learning using Gaussian fields and harmonic function is proposed in [392]. Classification algorithms for Gaussian fields can be seen as a form of nearest-neighbor approach, where the nearest labeled examples are computed by a random walk on the graph. The method's equations are related to electrical networks and to spectral graph theory. The problem is represented as a graph, with some nodes labeled with $y \in 0, 1$ (for simplicity we consider a binary labeling). Weighted edges represent similarities: w_{ij} is large for similar cases. For example $w_{ij} = \exp\{-\|\mathbf{x}_i - \mathbf{x}_j\|^2_{\mathbf{A}}\}$ for a suitable metric. The strategy is to first compute a "smooth" real-valued function f for all nodes and then assign labels based on f. The "smoothness" desire of having similar values between similar points is expressed by formulating the problem as one of minimizing the quadratic energy function

$$E(f) = \frac{1}{2} \sum_{i,j} w_{ij} (f(i) - f(j))^2. \tag{23.6}$$

The minimum energy function is *harmonic*: it satisfies $Lf = 0$ on unlabeled points, and is equal to the label value on the labeled ones. L is the graph Laplacian defined above, and the harmonic property means that the value of f at an unlabeled point is equal to the weighted average of f at neighboring points:

$$f(j) = \frac{\sum_i w_{ij} f(i)}{\sum_i w_{ij}}. \tag{23.7}$$

In matrix notation: $f = \mathbf{P} f$, where $\mathbf{P} = \mathbf{D}^{-1} \mathbf{W}$. This is consistent with the intuitive notion of smoothness with respect to the similarity relationships. The graph-based smoothing operation is illustrated in Fig. 23.3.

A simple rule is to label a node i with 1 if $f(i) > 1/2$, 0 otherwise.

The connection with **random walk** is as follows: imagine a walker starting from an unlabeled node i and moving to a neighbor j with probability $\mathbf{P}_{ij}$. The walk stops when the first labeled node is encountered. Then $f(i)$ is the probability that the walker stops at a node labeled 1.

The **electrical network** interpretation is as follows: nodes labeled 1 are connected to a positive voltage source, nodes labeled 0 to ground. Edges are resistors with conductance w_{ij}. Then f is the resulting voltage on the unlabeled nodes, which minimizes the energy dissipation. Ways to incorporate class prior knowledge (desirable proportions of the two classes) by modifying the threshold to label the nodes, as well as possible ways to learn a weight matrix $\mathbf{W}$ from labeled and unlabeled data are described in [392].

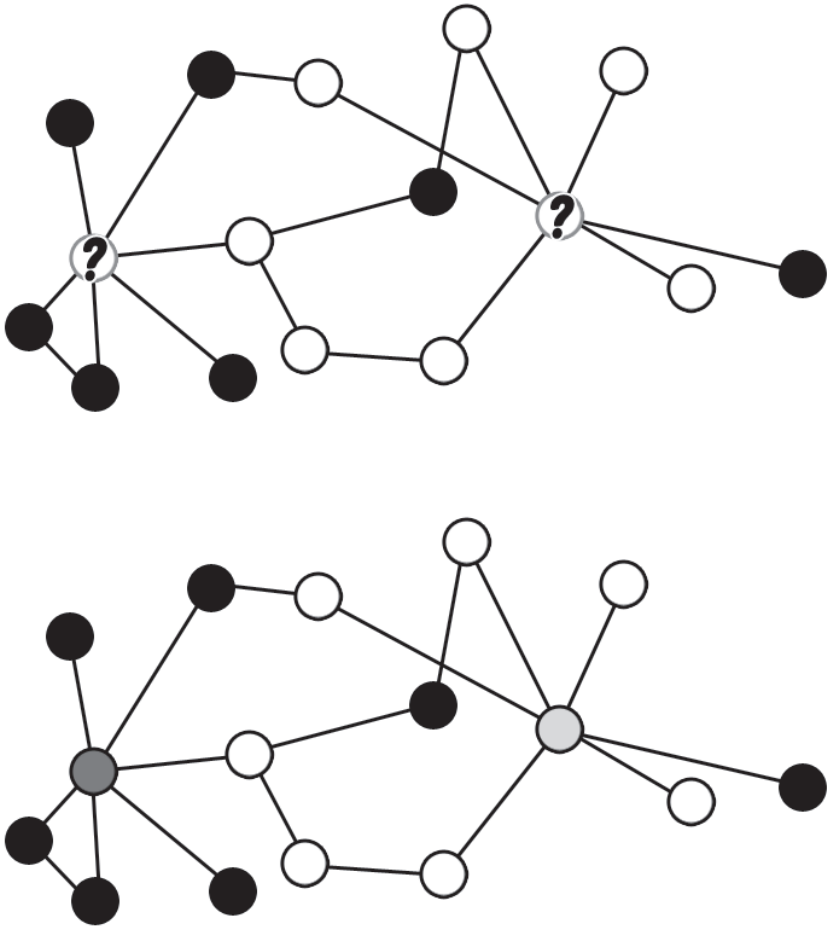


Figure 23.3: The values of the unknown nodes in the graph are computed as the average of the neighbors.

23.1.3 Learning the metric

Some semi-supervised algorithms proceed in two steps: first **a new metric or representation is identified** by performing an unsupervised step on all data (ignoring the existence of labels), then a pure supervised learning phase is executed by using the newly identified metric or representation.

The two steps are in fact implementing the semi-supervised smoothness assumption, by ensuring that the new metric or representation satisfies that distances are small in the high density regions.

Let's note that some graph-based methods are closely related to this way of proceeding: the construction of the graph from the data can be seen as an unsupervised change of representation.

23.1.4 Integrating constraints and metric learning

In many cases, when dealing with an optimization problem defined over more than one variable, a sequential method which first minimizes over the first variable, then over the second (leaving the first variable untouched) etc. gives in general a solution which can be improved if all variables are considered at the same time. This is clear because the freedom of movement in the input space is increased: in the first case one moves only along coordinate axes, in the