



**MANFRED FÜLLSACK**

# **FROM DATA TO INTELLIGENCE**

**An Introduction to Machine  
Learning and Artificial Intelligence**

**Graz University  
Library Publishing**



**Manfred Füllsack**

**From Data to Intelligence: An Introduction to Machine Learning and Artificial Intelligence**

**MANFRED FÜLLSACK**

# **FROM DATA TO INTELLIGENCE**

**An Introduction to  
Machine Learning and  
Artificial Intelligence**

**Graz University Library Publishing**



**Zitiervorschlag:**

Manfred Füllsack, From Data to Intelligence. An Introduction to Machine Learning and Artificial Intelligence. Graz University Library Publishing 2024.

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <https://portal.dnb.de> abrufbar.

© 2024 bei Manfred Füllsack



CC BY 4.0 2024 by Manfred Füllsack

Manfred Füllsack ORCID 0000-0002-7772-4061

Dieses Werk ist lizenziert unter der Creative Commons Attribution 4.0 Lizenz (BY). Diese Lizenz erlaubt unter Voraussetzung der Namensnennung der Urheberin die Bearbeitung, Vervielfältigung und Verbreitung des Materials in jedem Format oder Medium für beliebige Zwecke, auch kommerziell. (Lizenztext: <https://creativecommons.org/licenses/by/4.0/deed.de>)

Die Bedingungen der Creative-Commons-Lizenz gelten nur für Originalmaterial. Die Wiederverwendung von Material aus anderen Quellen (gekennzeichnet mit Quellenangabe) wie z.B. Schaubilder, Abbildungen, Fotos und Textauszüge erfordert ggf. weitere Nutzungsgenehmigungen durch den jeweiligen Rechteinhaber.

**Graz University Library Publishing**

Universitätsplatz 3a

8010 Graz

<https://library-publishing.uni-graz.at>

Grafische Grundkonzeption: Roman Klug, Presse und Kommunikation, Universität Graz

Coverbild: Generated with Stable diffusion

Typografie: Source Serif Pro und Roboto

eISBN 978-3-903374-31-7

DOI 10.25364/9783903374317

# Contents

Introduction .....	7
1 The use of models in nature – Anticipatory systems .....	12
1.1 Reinforcement-learning .....	14
1.2 Evolutionary Computation .....	20
1.3 The Free energy principle .....	22
1.4 Empowerment .....	25
1.5 Causal Entropy .....	30
2 Machine-based modeling – aka Machine learning .....	34
2.1 Analytical modeling – Linear regression .....	35
2.2 Linear and non-linear data discrimination .....	38
2.3 Data preparation .....	41
2.4 Supervised learning and Information gain .....	43
2.5 Applying a Decision Tree Classifier .....	45
2.6 Obtaining and evaluating results .....	48
2.7 Improving results .....	51
2.8 The bias-variance tradeoff, generalization and overfitting .....	53
2.9 Data augmentation .....	55
2.10 Feature reduction .....	56
3 Common Machine Learning tools .....	59
3.1 Support Vector Machine .....	59
3.2 k-Nearest-Neighbor .....	62
3.3 Naïve Bayes .....	63
4 Artificial Neural Networks .....	66
4.1 The Perceptron .....	66
4.2 Back propagation – the fundament of Deep learning .....	70
4.3 Gradient Descent .....	73
4.4 The MNIST-digits-example .....	76
4.5 Neural Network types .....	79
4.5.1 Long Short Term Memory (LSTM) Neural Networks .....	80

- 4.5.2 Recurrent Neural Networks ..... 82
- 4.5.3 Convolutional Neural Networks (CNNs) ..... 83
- 4.6 Unsupervised Learners ..... 87
  - 4.6.1 Self-organizing Maps (SOMs, aka Kohonen-networks) ..... 87
  - 4.6.2 Autoencoders ..... 89
- 5 Natural Language Processing ..... 93
  - 5.1 Mining in unstructured and analog texts ..... 97
  - 5.2 Latent Semantic Analysis ..... 98
  - 5.3 Vector representations ..... 103
    - 5.3.1 Assessing similarities by vector representation ..... 104
    - 5.3.2 Vector Space Representation of Words – Word2Vec ..... 106
    - 5.3.3 Vectorization examples ..... 108
      - 5.3.3.1 Vector arithmetics ..... 111
      - 5.3.3.2 Sentence similarity ..... 111
    - 5.3.4 Document similarity – Doc2Vec ..... 112
    - 5.3.5 Vectorization beyond textual contexts ..... 117
  - 5.4 Sentiment analysis ..... 118
  - 5.5 Transformers, or: Attention is all you need ..... 122
- 6 ChatGPT et al. .... 124
- 7 Epilogue: Data ethics ..... 130
- References ..... 134

# Introduction

This book introduces and explains essential prerequisites for understanding, applying, researching, and further developing the tools currently debated under the terms Machine Learning (ML) and Artificial Intelligence (AI). It strives to be an introductory and comprehensive guide for readers with little prior knowledge, while also offering deeper insights for those interested in advanced aspects and methods beyond the core of the research thread. Overall, this book is intended for anyone seeking a comprehensive understanding of the methods and computer-based applications underlying AI-technology. While digital literacy is beneficial, it is not a prerequisite for understanding the content.

A few years ago, I would have been very reluctant to call a book on machine learning “From Data to Intelligence”. The term “intelligence” seemed to me both too vaguely defined and far too ambitious to be used in the context of learning computer systems; especially as the history of this research thread is riddled with highly-hyped expectations and promises of machines becoming intelligent, with equally prominent retractions of these promises. The term “artificial intelligence” was coined back in the 1950s and the first “AI winter”, a period of widespread skepticism and reduced funding, was experienced in the 1970ies. However, the introduction of the Transformer technology in 2017 (Vaswani et al. 2017), and in particular its application in Large Language Models (LLMs), has entailed such an amazing boost in performance that anyone querying ChatGPT or any of the other now numerous LLMs would probably not hesitate to associate a notion of intelligence with these systems. Also in science, it is now assumed that some of the performances, such as maintaining context, thinking by analogy, following instructions or breaking down complex relations, correspond to behavior that we would call intelligent in humans. As a consequence, it is speculated that Alan Turing’s famous test, in which artificial intelligence is given if a human questioner cannot distinguish the answers of a computer program from those of a human, has been passed by GPT-4 (Biever 2023; Jones and Bergen 2024).

Nevertheless, as we will see in much more detail, the core of the technology currently used is to train artificial neural networks to calculate *probabilities* that certain

tokens, such as the letters of a word, the words of a sentence or the sentences in a text, will be followed by other letters/words/sentences. These probabilities are calculated from an enormous amount of training data, e.g., almost all texts available on the internet, with such high precision that human users can receive largely satisfactory responses to their queries. Nevertheless, these responses, such as the answer of an LLM to a question, build on nothing more than aggregated probabilities. LLMs and similar tools are therefore often referred to as “statistical parrots”, making it more than questionable whether human-like intelligence is at stake. If we consider however, that humans, for their part, also depend on being confronted with large amounts of data in order to learn what makes them who they are, and that our knowledge and assumptions on which our daily decisions are based often look like mere probabilities themselves, then the difference between the way machine and human intelligence work might not be as huge as is often emphasized. I wouldn’t want to call humans “statistical parrots,” but given the fact that throughout our history we have liked to describe ourselves as unique and central, only to often learn that we are not so unique after all, I would be willing to forgo reserving characteristics such as intelligence for humans. That aside, books usually do well to have a somewhat bold title anyway. I therefore accept the term here at least as a *telos* of the research field being discussed.

Speaking of books: it should be mentioned that books are slow media. The rapid pace of digital development, particularly in AI research, presents a significant challenge for writing a book that commonly is expected to be comprehensive and up-to-date. The field’s growing interest and the increasing number of researchers working worldwide on new methods and insights have created enormous dynamics and particular conditions. Information is usually disseminated online first in this field and, due to time pressure, sometimes not peer-reviewed, let alone printed on paper. Additionally, many methodological details are not published due to fears of losing a competitive edge in the global market. Writing a ‘slow’ book about ML and AI thus faces several challenges, making it prudent to pursue a specific focus that faster media might lack.

One of these foci emphasizes the fundamentals and prerequisite technologies on which current methods are based, aiming to provide a comprehensive foundation for understanding the latest developments through self-study. The book covers the state of the art provided by platforms such as ChatGPT-4o, Gemini 1.5 or Claude 3.5. However, new versions and add-ons are introduced almost weekly. The book will not manage to cover all of the latest developments in a timely manner. It is an introduction to a highly dynamic and rapidly developing field. You will have to upgrade and update your knowledge from additional sources.



In other respects, this book may have a unique character due to the author's profession as a systems scientist, specialized in analyzing and coding computer-based models of complex systems. It bases on the assumption that most ML methods and models are complex systems, such as in particular the core technology of today's AI-tools, neural networks. This asks for an explanation about what systems sciences can offer to help understanding how AI-tools are working.

Let me briefly elaborate on this, since it may introductorily clarify some core features. Systems sciences is a unique scientific discipline that fundamentally believes that interacting components can aggregate into phenomena that cannot be understood by analyzing the components in isolation. For example, consider the particles of a gas. Their movement and interaction generate pressure and temperature, which are observable features of the gas. However, these features cannot be observed or measured on an individual particle. A gas particle on its own does not have pressure. Pressure arises through interaction. Systems scientists say that pressure *emerges*.

A systems scientist thus would emphasize that the ability of neural networks to identify patterns in large datasets, along with the intelligence-like abilities of Generative Pretrained Transformers (GPTs), emerges from the interaction of their components—the neurons in the network. In other words, intelligence, whether human or artificial, is believed to be a result of interaction. It cannot be understood by examining components, such as neurons, in isolation. This provides an initial clue as to why the functioning of neural networks is so difficult to analyze and to understand. Neural networks cannot be disassembled to understand their operations, as one would do with a mechanical machine. Analyzing components in isolation will not reveal the effects we are interested in.

Another concept of systems sciences that may help to understand the workings of AI-tools is the *attractor*. The interactions from which a system with specific qualities or abilities may emerge often start out as a jumble of components interacting with each other's dynamics. In this state, components may chaotically bump into each other and influence other components in various unordered ways, until, after some jumble, they find a kind of relationship to each other in which opposing forces are balanced. These relationships are called *equilibria*, often illustrated as a marble rolling in a bowl until it finds balance at the lowest point. Physicists refer to this as a potential well. When the marble is slightly displaced from this lowest point, such as by shaking the bowl, it will follow gravity and return to the lowest point. In this regard, the equilibrium, or lowest point, acts as an attractor. It attracts the marble, guiding its movements within the bowl.

Such equilibria are found in manyfold contexts. Systems scientists even tend to think that our world, or more accurately, everything that we perceive as somehow stable in our world, is affected by such equilibria. The movement of the planets in our solar system and their relationships are the result of an equilibrium reached after millions of years of chaotic motion. On another scale, our social order can be viewed as an equilibrium maintained and continuously regenerated by interacting individuals with their conflicting wishes and plans. The key point is that once such an equilibrium is established, it acts as an attractor. Deviations do not necessarily cause the system to reorganize completely; instead, they are smoothed out as the marble returns to the lowest point of the bowl. We describe equilibria as being *resilient* to a certain extent.

Similar is true for neural networks. Have you ever looked at the sky on a summer day and suddenly noticed a cloud that resembled an animal? And once you perceive this animal, your interpretation seems to retain form, even as the cloud changes with the wind. Or have you been impressed by your ability to briefly glimpse a tiny fraction of something, like a word on an advertising board while driving by, and still clearly understand what was written despite not being interested? Sometimes, you might suddenly think of an odd word and not know where it came from, only to realize later that you saw it somewhere without consciously reading it. These interpretations occur because our perception is driven by equilibria that neurons (or, more technically, the weights of their connections, as discussed in the chapter on neural networks) have established through training on numerous pictures of animals and words. Training in this regard means that the components of the neural network together with the data on which it is to be trained have been made to interact. They were exposed to each other, initially ‘chaotically’ (we will later see that there is indeed an arbitrary aspect to this), and made to find a viable relation to each other, an equilibrium that in the course of training got engraved into the network. Once the equilibrium is found, it acts as an attractor, attracting perception towards learned concepts, allowing us to see animals in clouds and recognize words even when only partially viewed for a brief moment. In other words, our brains, trained to differentiate things like animal shapes or words, attract these interpretations even when data is missing or erroneous. The marble is running to the lowest point in the bowl and lets these concepts be reproduced easily, which is advantageous for survival.

Keep in mind, these equilibria are established in interaction, with no chance to separate components without ruining the equilibrium. Neural networks cannot be disassembled to examine them in detail, as one might do with the parts of a car to understand why it is fast. This is why the capabilities and potentials of AI tools are

often surprising and difficult to understand (Castelvecchi 2016; Adadi and Berrada 2018). In particular their latest manifestations, the so-called Large Language Models, seem on the way to shake the fundamentals of the way we think about thinking, including the way we conduct science. Their use and further development pose a range of difficult questions for the near future (Ribeiro, Singh, and Guestrin 2016; Barredo Arrieta et al. 2020; Mitchell and Krakauer 2023; Butlin et al. 2023). It has been reported that even the software developers and engineers who themselves designed and trained these AI-tools were amazed by their unexpected performance. In some instances, scientists known for their sobriety and level-headedness have fallen to the impression that these tools possess properties such as sentience, self-awareness, or an irrepressible drive for independence.

Yet a third peculiarity in the specific approach of systems sciences may be helpful in understanding the workings of machine learning tools. This is the conviction that many of the special characteristics of complex systems have to do with *mapping* or *modeling* of facts. System Scientists assume that complex systems such as our brain are modeling the world in order to perceive it as such. What we experience as “the world” is a *model* generated and continuously adapted by our brain throughout our daily lives. And since our brain itself is part of this world, it is modeled as a model itself. We therefore refer to the output of such systems as models, and we call the systems themselves models as well. Computer-based ML-tools and in general the methods of AI are seen as a sort of algorithmic models generated in respect to regularities and patterns found in large data sets. We start this book therefore with some rather basic and general reflections about the preconditions and effects of using models.

# 1 The use of models in nature – Anticipatory systems

Picture a population of very simple organisms on a search hunt for nutrients. In a most basic scenario, these organisms move and distribute randomly in their environment and, if lucky, find food by coincident. Gradually, they will distribute evenly in their possibility space, that is, in the space they can reach *in principle*, with all states in this space being equally likely, implying that after some time, their distribution will not allow deducing their initial state. In other words, their distribution is not based on history. No kind of memory is involved. In physics, such systems are called *ergodic*.

In a slightly more complex scenario these organisms could have some kind of sensors for distinguishing lower and higher concentrations of a scent the nutrient diffuses and thus are able to follow a gradient of thinning scent uphill to find the food source. Note that this implies some kind of a *temporal* concept, that is, a notion of *now* – without food but a scent that can be followed – and *then* – when the food source is found. This anticipation hence seems to imply the emergence of time. In a still more complex scenario such organisms may even dispose of a sort of memory to remember the direction to a prolific food source. In this scenario, these organisms will focus their movement actively on the food source and thus will not distribute evenly over the possibility space. In physics, this is called a *non-ergodic* system. It is governed by history. What is more, to remember a food source is useless without having an idea about where you yourself are in relation to this location. To re-find a food source hence implies the maintenance of some sort of *map*, which as a minimum includes the position of the food source plus your own position relative to it. Such a map could be seen as a *self-model* – a simple model at first, but after all a model that changes a lot in regard to survival strategies. While the randomly moving organism needs luck to find food, the model-using organism can *pre-test* its ac-

tivities in the *virtual space* of its model, thus being able to *anticipate* auspicious options (for a more comprehensive explanation about the use of models in the context of intelligence see: Hawkins 2021).

Systems that use such a model conception have been called *anticipatory systems* (Rosen 2009). They are able to deploy an internal representation of their external world to maintain *homeostasis*, or in other words, to maintain dynamical equilibria (e.g., body temperature, blood pressure etc.) against the overall drive towards entropy. Note however that a model is not to be had for free. To evolve it<sup>1</sup> and to maintain and update it, is costly, which for a metabolism means supplying an extra portion of energy: the energy that is needed to survive without model plus the energy needed for maintaining the model. The model's complexity hence is subject to a trade-off: the more detailed the model the better the chance to survive, but the higher the costs of maintenance.

So how do models emerge? In an attempt to insinuate an answer to this question, one might maybe regard a population of bacteria or similar simple organisms as something like a proto-model incorporated by evolution. The argument for this would be that such a population consists of individua which – initially maybe just by chance – live and act in a way that successfully supports their existence and provides them with high reproduction probability, while other individua act less advantageous and thus do not obtain high evolutionary fitness. This population thus could be seen as representing a distribution of “ways to live and to act”, of which some are *rewarded* by evolution through high fitness, and of which others, who act less advantageous, are *penalized*. One may say that nature “rewards” and “penalizes” ways to live and “memorizes” these outcomes in form of fitness distributions incorporated in a population. Nature evolves a “model” by following the principle of rewarding actions, which reach a goal, and penalizing the ones, which do not. This is the principle of so-called Reinforcement Learning.

---

<sup>1</sup> To evolve a model can be seen as anticipation itself. Models can start covering their costs only once they exist. Evolving a model thus may appear like taking a loan on future survival that will be more successful once a model is at hand. See for this seeming inconsistency the chapter on irreducible complexity at: <http://systems-sciences.uni-graz.at/etextbook/sw1/evolution.html>

## 1.1 Reinforcement-learning

Reinforcement-learning (RL) is a widely applied machine learning method that in its basics is simple and easy to understand, but in its more advanced variants is a very powerful and flexible method for artificial intelligence research. Among others, it is a component of the Alpha-Zero algorithm, which in 2017 in just a few days of training defeated Alpha-Go, the AI algorithm that beat Lee Zedol, one of the world's best Go players (Silver et al. 2018; Schrittwieser et al. 2020).

The objective of RL is simply to automatically generate a model, which tells an agent what action to take under what circumstances. The model is generated by repeatedly trying to reach a goal and collecting (digital) rewards for those actions which help bringing the agent closer to its goal and getting (digital) penalties for those actions which distance it from its goal. Basically, this works without any supervision by simply applying brute computational force, meaning that the agent has to be made reaching its goal a large number of times. We will explain the basics of this method on a very simple example task, in which a software agent – the red dot in the image below – has to find the shortest path from each possible position on a 2-dimensional grid – the grey patches – to its goal – the small green house in the center of the grid.

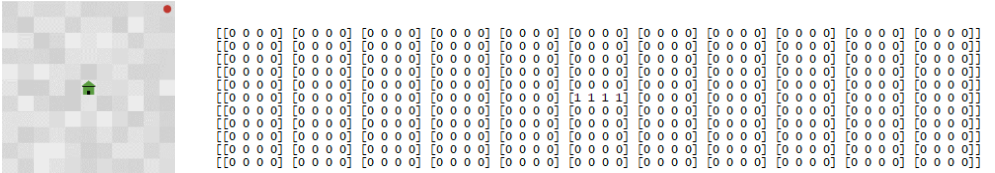


Fig. 1: Agent (red) navigates a 2D grid to reach a goal (green) (own illustration).

The agent starts out by randomly moving on the grid in regard to its Von-Neumann-neighborhood, which means that its movements are restricted to one move each timestep to the North, to the East, to the South or to the West within the limits of the grid. Consequently, the agent has a maximum of 4 options on each position it can be on. This means the so-called *state space* of this agent is of size 4 per movement. With a grid-size of 11x11 this sums up to an overall state space of 484 options and thus to rather limited complexity. Note that the game of Tic-Tac-Toe for instance has an overall state space of  $10^3$ , the game of Chess has one of  $10^{75}$ , the game of Go

has  $10^{171}$ , while the estimation for the number of atoms in the observable universe runs up from  $10^{78}$  to  $10^{82}$ . It should be conceivable that the very complex task to win in Chess or in Go necessitates a completely different kind of RL-model than the one to solve the simple task in our example. Nevertheless, algorithms for solving these tasks are based on similar principles.

The model in our example manifests in the form of a so-called *Q-table* containing *Q*-(for quality) values, depicted to the right of the display in the above image. Such *Q*-tables consist of vectors for each state  $s$ , with each entry in these vectors indicating the *quality* for each action  $a$  that is possible in this state. Since possible actions in our example are only [go North, go East, go South, go West], each vector holds the entries [N, E, S, W], always in that same order. An entry in the *Q*-table thus corresponds to a *state-action-tuple*  $(s, a)$  and tells the agent: “if you are in state  $s$  and take action  $a$ , your reward (penalty) will be  $x$ ”.

Initially, the *Q*-table is filled with zeros, as shown in the depiction above. Only the state that represents the attained goal (the green house) is filled with the reward value (1 in this case). When the learning process is started, the agent is made to perform a random walk, or more correctly, to check the *Q*-table and to decide in each step for the action with the highest *Q* (entry) in the given situation. Since all entries are zero initially, however, the agent practically performs a random walk at this stage. If it should (coincidentally) run into a forbidden option – as for example to move over the edge of the grid –, it records a penalty of -1 into the corresponding entry in the *Q*-table. All other values are kept the same (i.e., zero at this point) until the agent (coincidentally) finds the goal for the first time. At this point, it can report *Q*-values for the *last* action that preceded the goal, that is, the last state-action tuple before reaching the goal. If adhering to the Bellman Optimality Equation (Bellman 1957) – which provides one commonly applied reward function –, the agent calculates these values and replaces the corresponding *Q*-table entry according to the formula:

$$Q_{t+1}(s, a) = (1 - \alpha) * Q_t(s, a) + \alpha * (r + \gamma * \max(Q_t(s_{t+1}, a))$$

with  $Q$  indicating the quality (the entry),  $(s, a)$  the state-action-tuple,  $r$  the reward for attaining the goal,  $\alpha$  a learning rate,  $\gamma$  a discounting rate for  $r$  and  $\max(Q_t(s_{t+1}, a))$  is the maximum value of the *Q*-vector found for the next position, in this case the attained goal.

Note, that  $\alpha$  determines how much of the already available knowledge and how much of the currently newly gained knowledge will be considered in calculating the new *Q*-value. And  $\gamma$  determines a discount rate for considering the reward found in

a certain state. The further the agent is from its final learning target, the more the Q-values will be discounted, thus generating a gradient towards the target.

This discounting of Q-values is necessary to overcome the so-called *credit assignment problem*, which centers around the question of how to know what is advantageous in an action if the reward (like winning a chess game) is several game-moves away? In short, the solution for this problem consists of the three steps:

- Accumulate rewards backwards from repeatedly attaining the learning goal.
- Do this by assessing the quality (the Q-value) of each possible decision in each state in the state-space (for example with the Bellman Optimality Equation)
- Write state-action- (or Q-) values into a Q-table

Accordingly, the Q-table after first time attaining the goal will look like depicted below. Obviously, the agent has reached the goal coming from the West of the goal by deciding to go East. The corresponding entry changed into  $Q_{t+1}(s, a) = 0.48$  (with  $\alpha = 0.5, \gamma = 0.95$ ).

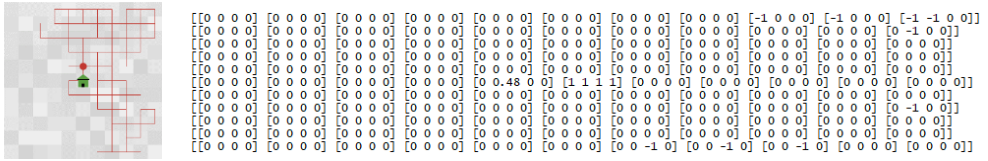


Fig. 2: Q-table after the agent first reaches the goal (own illustration).

All the other values remain the same and the agent is placed on a (randomly chosen) new starting position, again made to select a path according to highest Q-values, but practically still performing a random walk until it hits the goal or, this time, another state with an entry higher than zero. At this point it again updates the quality (the Q) of the action that preceded this move according to the above formula.

In this way, the agent by and by fills all entries of the Q-table with values, which can be read as rewards for respective actions. Actions that bring the agent closer to its goal will have higher Q-values (higher quality) than disadvantageous actions. See below an example of an accordingly filled Q-table.



```

[[-1 0.38 0.35 -1] [-1 0.01 0.59 0.42] [-1 0.36 0.45 0.45] [-1 0.45 0.46 0.4
[0.2 0.58 0.26 -1] [0.42 0.35 0.64 0.32] [0.3 0 0.67 0.21] [0.08 0.26 0.72
[0.37 0.63 0.43 -1] [0.58 0.68 0.58 0.53] [0.52 0.72 0.68 0.56] [0.63 0.56
[0.26 0.64 0.22 -1] [0.15 0.72 0.16 0.29] [0.66 0.72 0.76 0.55] [0.71 0.7
[0.49 0.4 0.2 -1] [0 0.76 0.54 0.19] [0.72 0.8 0.78 0.72] [0.76 0.85 0.85
[0.4 0.41 0.23 -1] [0 0.78 0.32 0.19] [0.67 0.85 0.68 0.33] [0.79 0.9 0.8
[0.19 0.49 0.16 -1] [0 0.76 0.28 0.13] [0.75 0.8 0.26 0.67] [0.84 0.85 0.57
[0.13 0.19 0.27 -1] [0.13 0.53 0 0] [0.2 0.72 0 0.05] [0.8 0.75 0.72 0.47]
[0.06 0.48 0.1 -1] [0 0.67 0.15 0] [0.29 0.72 0.5 0.26] [0.65 0.76 0.67 0.1
[0.12 0.16 0.14 -1] [0.47 0 0.24 0.07] [0.66 0.51 0.23 0] [0.36 0.72 0.37
[0.11 0.18 -1 -1] [0.13 0.31 -1 0.18] [0.54 0.41 -1 0.25] [0.62 0.46 -1 0.2

```

Fig. 3: Clipping of an updated Q-table (own illustration).

A completely filled Q-table can be used by the agent to deterministically follow the gradient of increasing Q-values uphill to its goal, which, in a static environment, means to possess an efficient model for reliably reaching the goal. The learning process would be over at this point. In an evolving world however, goals, or the path to these goals, may change over time. What has been learned so far, may become suboptimal and necessitate re-learning. The model needs to be updated.

In such situation, it makes sense to consider a possibility to not just *exploit* what has been learned so far, but also to *explore* new options to a certain extent. For considering such an *exploitation/exploration-ratio*, a parameter  $\epsilon$  can be introduced into the Bellman equation that determines the probability of considering the Q-value with regard to the probability  $1 - \epsilon$  of choosing a random action. In this simple way, the model can be held flexible and adaptive to new circumstances, and can be applied to a wide range of interesting and complex tasks. As one example in this regard, have a look at this project: <https://blog.einstein.ai/the-ai-economist/>, where RL is used to optimize tax policies on the background of the contradicting goals of improving equality and increasing productivity.

Other currently quite prominent examples of applying RL are the manifold amalgamations with other machine learning methods, before all with artificial neural networks (see chapter 4) in order to solve complex game winning tasks. These experiments are often addressed as *Deep-Q-Learning* and have produced such impressive results as for example AlphaGo's 2016 victory over Go-grandmaster Lee Sedol, AlphaZero's 2017 victory over AlphaGo (Silver et al. 2018) and 2019 MuZero's victory over AlphaZero (Schrittwieser et al. 2020). All these algorithms are advanced AI-systems trained to win board games like Go, Chess or Shogi based on RL, with the difference that Q-values are not calculated like in the Bellman equation but "found" by deep neural networks, which in the later versions of these algorithms are initiated even without any pre-learning from human games. Instead, these algorithms are made to play large numbers of games against copies of themselves, in the course of which they accumulate the necessary knowledge. One version, MuZero (10-2021),

even learns without being taught the rules of the games. It is, as one of its developers put it “really discovering for itself how to build a model and understand it just from first principles.”

Another achievement of DeepMind, the Google-firm that developed these algorithms, is *AlphaFold*, a deep learning artificial intelligence neural network system that is able to uncover the complex 3D-structure of proteins. The model made headlines when it was placed first in the overall rankings of the 13th Critical Assessment of Techniques for Protein Structure Prediction (CASP) in December 2018 and could repeat the success two years later at CASP 14 achieving a level of accuracy much higher than any other competitor<sup>2</sup>.

We will talk about neural networks, which are the core of these algorithms, later in this book. For now, please note that reinforcement learning is a simple, though potentially powerful method for making machines learn to accomplish complex tasks. However, as natural learning by reward and penalty may seem, there is one fundamental drawback. Reinforcement learning, in principle, acts like a teacher grading you after an exam and not providing any information about where you performed good and where you could still improve. In other words, the basic variant of RL (as explained in the example above) does not provide any information “about the road” to accomplishment (which is less a problem in so-called *symmetric zero-sum games*, such as Chess and Go, but is of more concern with complex tasks). Rewards and penalties are only assigned once a goal is reached, which with the learning of complex tasks may either take a very long time or is not feasible at all.

A simple example in this regard provides the task to reach an uphill goal by making the learning agent swing forth and back to gather speed for the uphill move. Each time in this swinging when the agent does not gain enough speed it will be penalized, thus making failure much more consequential than success. A common method to overcome this problem is *reward engineering* (aka reward shaping), that is in this example, to not only reward the actual goal reaching swing, but also the swinging itself. However, this requires to know ahead that swinging is beneficial in this task. In other words, it requires domain knowledge, which is not always at hand and often tricky to apply.

Another way of overcoming the problem of sparse rewards is to make RL learn from the details of failed attempts, for example from how far and in what direction a failed attempt missed the goal. For this, failed attempts are stored in a kind of buffer

---

<sup>2</sup> See among others: <https://www.technologyreview.com/2020/11/30/1012712/deepmind-protein-folding-ai-solved-biology-science-drugs-disease/>

of past experiences, which can serve as a database for so-called *off-policy learning*. In the framework of our picture of model-based actions, this means that RL generates a much more *general model* than the above-mentioned Q-table, from which it then can infer the relevance of its actions in order to finally generate the actual model needed for reliably attaining the goal. This method is called *Hindsight Experience Replay (HER)*, and has a bit of the flavor of adding some fictitious data to a world model by imagining what would happen if the circumstances were different. Andrychowicz et al. (2018) show that HER can improve the performance of RL significantly. The principle is an extended, more complex way of model generation by simply trial-and-error exploring an environment (see also <https://www.youtube.com/watch?v=Dvd1jQe3pq0> ).

This trial-and-error exploring of the *state space*, that is, the space of all possible actions in a given situation, is one of the core problems of RL though. Even relatively simple tasks, which humans master with ease, can have vast state spaces. Just consider driving a car. How many actions could be performed in each moment of driving? And how many of them would be completely ill-suited, such as e.g., reading a book, drinking or sending phone-messages while driving. To explore these vast state spaces and filter all at least somehow suitable actions would be beyond the capabilities of even the most powerful computers.

Therefore, great efforts are being made to pre-reduce this space to manageable sizes. One method that is investigated in the framework of RL has been suggested under the term *Inverse Reinforcement Learning* (Ng and Russell 2000). It builds on the idea of extracting a preliminary reward function from the observed behavior of an agent, possibly a human. Consider the task of autonomous driving. The conventional way of defining a reward function would be to picture a correct way of driving a car, such as stopping at red lights, avoiding pedestrians, etc., and engineer a reward function from this conception. This however, requires an exhaustive list of all the “correct” actions of driving and a list of weights describing how important each of them is (pedestrians more than red lights, etc.). Inverse RL in contrast, derives an *approximated* reward-function from a set of human-generated driving data, which, depending on the size of this data, initially will provide a rather simplified model of driving. Ng and Russell (2000) showed, that such an approximate quantification of how good or bad certain actions are can reduce the option space dramatically and thus can serve as a base from which standard RL methods can go on fine-tuning the model.

## 1.2 Evolutionary Computation

In the previous section, we mentioned the possibility of considering populations of organisms as manifestations of evolutionary emerged models. The very fact that such populations can exist over periods of time suggests that they have found a way to adapt to their particular environment and thus mirror it in some way, or “model” it in their own manifestation.

In natural systems this is usually done through the evolutionary principles of variation, inheritance and selection, which have proven to be very efficient means of adaptation. No wonder, these principles early on have been used in machine learning implementations for finding optimal solutions for complex problems (Mitchell 2011). In order to briefly explain these principles in this context, think about the problem of finding an optimal strategy for competing in a Prisoner’s Dilemma (PD) situation<sup>3</sup>. In this situation, betrayal (aka defection or non-cooperation) is the dominant strategy, since betrayal is your best choice if your opponent cooperates (remains silent), and it still yields a higher payoff than any other option (i.e. less years in prison) if your opponent should defect (i.e. betray too). The dilemma however, is that if both of you defect, each of you is worse off than if you both would cooperate. However, you do not know what the other will do. It is thus highly unlikely to change overall behavior from individually beneficial defection to communally beneficial cooperation (for more details see among others Axelrod 1984).

In the beginning of the 1980ies, the problem of this unlikely change from defection to cooperation was famously tackled with the help of a *genetic algorithm* and thus became one of the first better-known examples of evolutionary computation. The political scientist Robert Axelrod (1984, 1987) endowed a population of software agents with a “chromosome”, basically just a vector consisting of zeros and ones defining the behavior, that is, the PD-strategy of an agent. This vector codified the outcome of the last three confrontations of an agent in the PD (“memory three-strategy”) and therewith provided information for deciding whether to cooperate or to defect in the next confrontation.

Axelrod’s experiment started out with an initial population of 20 agents with randomly generated 70-bit chromosomes (i.e., chromosomes with 70 genes) playing a virtual PD against each other, thereby developing a “fitness”, which was deter-

---

<sup>3</sup> See <http://systems-sciences.uni-graz.at/etextbook/gametheory/prisonersdilemma.html>

mined by the average success in about 150 PD-confrontations. The agents could reproduce in respect to their “fitness”, with their chromosomes being inherited via “crossover” and “mutation”, two technologies commonly deployed in computational evolution.

In *crossover*, two parent chromosomes are cut at the same position and combined with each other to yield two new, genetically different chromosomes, thus bequeathing part of the mother’s and part of the father’s chromosome to the offspring agent, which is nothing but a simplified version of biological recombination of genes. Thereafter, a small part of the chromosome (one or a few genes) is altered via *mutation*, i.e. random variation. The resulting chromosomes then are repeatedly exposed to “natural selection”, in which the “survival of the fittest”, that is the higher reproductive chances of the ones that perform best, drive adaptation.

When running this evolutionary algorithm, the PD-agents initially tend to drift away from cooperative behavior. Less cooperative strategies fare better, because early on there are only few other cooperative strategies they can score with. After 10 to 20 generations however, some strategies develop chromosome-patterns, with which they react reciprocally to other cooperative agents. The more these patterns stabilize, the better cooperation performs. And with better performance, cooperators gain more reproductive success and thus more chance in spreading their strategy.

Today, evolutionary computation is a wide and dynamically developing sub-domain of machine learning, with applications ranging from biologically inspired experiments in replicating the evolution of mobility in soft robots (see among others <https://www.youtube.com/watch?v=z9ptOeByLA4>) up to classification and prediction tasks such as the prediction of weather or protein structure (Pedersen and Moulton 1996). Genetic algorithms are even used to evolve aspects of particular machine-learning systems, such as weights for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots. The picture below (found in Wikipedia) shows an antenna used in the 2006 NASA ST5 spacecraft. The complicated shape was generated in evolutionary computation designed to create the best radiation pattern. It is known as an evolved antenna.



Fig. 4: Evolved antenna, taken from Lohn et al. (2004).

### 1.3 The Free energy principle

Regarding populations that maintain homeostasis as manifesting a model of their environment conveys a rather abstract notion of a model. Probably closer to the idea of a model as we use it, would be a sort of individual internalization of our environment to create a virtual image of the world in which we have to find our way. This kind of model is subject of a famous conception with interesting implications for machine learning discussed under the term *free energy principle*. Karl Friston (2006, 2009, 2010) suggested this principle for the functioning of brains, with brains being (individual) physical manifestations of models that holders of brains use to maintain their homeostasis. The brain is the physical expression of an internally maintained model of the external world as perceived through senses. The actions of the holder of the brain can be seen as permanently trying to maximize model evidence, that is, to maximize the probability that what is modeled corresponds to what is perceived through senses (expressed as  $p(s|m)$ , with  $p$  being the probability,  $s$  expressing a sensory input and  $m$  expressing a corresponding model prediction. The sign  $|$  stands for “given that”). Maximizing model evidence can also be expressed as the attempt to minimize the *surprise* that an incorrect prediction from the model would mean to the holder of the brain.

The concept of surprise can be formalized in the framework of *information theory* as suggested by Claude E. Shannon (1948). To understand this, consider the distribution of letters in written text. The letter *e* for instance appears far more frequent

than the letter *h*. To encounter an *h* in texts thus is more surprising than to encounter an *e*. Of course, this surprise depends also on context. In English to encounter an *h* after a *t* or after an *s* is less surprising than to encounter it on its own.

In his research, Shannon was interested in the lossless transfer of information between a sender and a receiver. Considering symbols with which information can be transferred – the letters of the alphabet for example –, Shannon suggested to measure the expectation of a symbol being chosen from a given set of possible symbols. The certainty or uncertainty about a symbol being chosen, that is the *Shannon entropy*, then depends on the size of the set of different symbols and on the information that determines a choice. Formulated in terms of observations, this information corresponds to the question about how many single observations are necessary to unambiguously determine a symbol in a given set of symbols. For example, to select the letter *h* unambiguously from an alphabet with 26 letters, it needs 5 consecutive observations of the form “the letter is in the first half of the alphabet”, “the letter is in the second half of the first half of the alphabet”, “the letter is in the first half of the second half of the first half of the alphabet”, and so on.

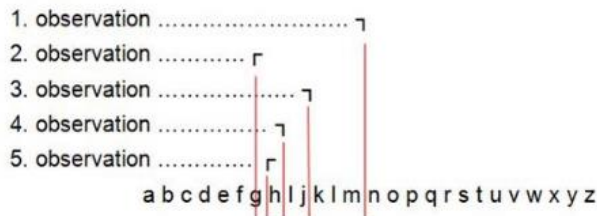


Fig. 5: Selecting the letter ‘h’ from an alphabet of 26 letters (own illustration).

Assuming that all letters in this alphabet have equal probabilities of occurring in a message (which of course is *not* the case in natural languages), one of them would have the occurrence probability  $p = \frac{1}{26}$ . To unambiguously distinguish 26 letters hence, one needs a minimum set of 5 observations corresponding to a set of 5 binary choices (i.e. “yes / no”, respectively “1 / 0”) expressed in the relation  $2^5 = 32$ . (Note that  $32 > 26$ . This set provides more selection possibilities than needed, it is *redundant*, but is the minimum set for unambiguously identifying 26 letters. The ASCII-code for example uses binary numbers with seven digits and therefore allows capturing  $2^7 = 128$  different symbols).

The corresponding formula for the set size is  $2^I = N$ , with  $I$  indicating the number of observations and  $N$  indicating the number of letters in the alphabet. With  $I$  being the binary logarithm, this allows expressing the number of needed observations as  $I = \log_2 N$ . As said, in alphabets with equally likely letters the occurrence probability  $p$  of a letter would be  $\frac{1}{N}$ , implying  $I = \log_2 \frac{1}{p}$ , with  $I$  now expressing the information content of the single letter, or alternatively formulated, its *surprise value*. It expresses the fact that the more often a letter or a sign occurs in a message, the lower is its surprise value, and vice versa. Rare signs hence have high surprise values. Based on this insight, Shannon was able to express the information content of a message with  $n$  letters in terms of entropy, that is, in terms of the average uncertainty with which a letter would be chosen:  $I = -\sum_{i=0}^n p_i \log_2 p_i$

In respect to the free energy principle, Karl Friston used this formalization to express the surprise of a sensory input on the base of a model prediction as  $-\log_2 p(s|m)$ . He suggested that, in general, the actions of brain users, that is, of anticipatory systems are always directed towards minimizing this surprise, or at least to keep it beneath a maximal variational bound. This bound is called free energy. The suggestion rests upon the fact that self-organizing biological agents resist a tendency to disorder by minimizing the entropy of their sensory states in relation to their model predictions. In analogy to Shannon's formula, Friston expresses entropy as  $[p(s|m)] = \lim_{T \rightarrow \infty} \frac{1}{T} \int_0^T -\ln p(s|m) dt$ , implying that minimizing entropy corresponds to suppressing surprise over time. To give an example, consider a fish in water (Friston 2009). In comparison to what states a fish theoretically could be in (out of the water, in mid-air, at the moon ...) the number of states a fish usually is in is relatively small. The distribution of fish states thus has low entropy, implying that fish avoid surprising states (e.g., being out of the water). However, the fish cannot evaluate surprise directly, because this would necessitate knowing all possible states it could be in. For knowing its world, the fish uses a model, and this model cannot hold all these states. The model is an evolutionary evolved abstraction, and as said, the extent to which it abstracts from the world is related to the extra resources that can be procured through its use.

So, what the fish actually assesses is not the surprise of a divergence between its sensory inputs and *any* possible state in its world, but just the divergence between sensory inputs and the *representation* of its world as generated by its model. It hence is the model's *prediction error* that Friston describes as free energy. The fish tries to minimize it either by changing the representation, i.e., adjusting the model (aka learning), or by action, that is, by bringing itself or its environment into a state that corresponds to the model prediction (e.g., getting back into water). In summary,



the fish resists a natural tendency to disorder by minimizing a free-energy bound on surprise, which entails either adjusting its model or acting on the environment to avoid surprises.

Mathematically, the states as foreseen in the model as well as the states as mediated through sensory inputs can be expressed as probabilistic distributions, which corresponds to the assumption of the so called Bayesian Brain Hypothesis, the notion that the brain is an inference or Helmholtz machine (Dayan et al. 1995). The divergence between these distributions can then be grasped with statistical means – the Kullback-Leibler-divergence for instance – and its minimization can be formulated as a gradient descent on free-energy to furnish differential equations (Buckley et al. 2017). We will discuss gradient descent in section 4.3.

## 1.4 Empowerment

The free energy principle explains the alignment of model predictions and sensory inputs. It does not, however, say anything about the purpose of this alignment or the purpose for which surprise needs to be minimized. On a most general level of course, such purpose would be the survival of the holder of the model. Sensory inputs however, are rarely directly related to survival. Touching a hot stove for instance, may be painful, but it is not always lethal. The pain you feel is just an *indirect* signal that hot items can be detrimental for survival. In regard to evolutionary selection it does not provide a clear signal for picking the fittest. For some perverted reason you may like to touch stoves, but still be fit to survive.

In general, it could be argued that life provides rather sparse direct signals for evolutionary selection. It therefore has been suggested that evolution is complemented with additional basic principles that provide more local but also more universal signals for directing actions or maintaining homeostasis of complex systems. One such basic principle has been proposed in the realm of robotics and artificial agents under the term *empowerment* (Klyubin/Polani/Nehaniv 2005, Klyubin/Polani 2005, Salge/ Glackin/Polani 2013, 2014). In a nutshell, the principle is a suggestion for quantifying the degrees of freedom (or options) an agent (e.g. an organism) has in controlling its environment based on its current world model. More precisely, the concept distinguishes between a specific empowerment value for an agent's *current* state according to its world model (that is, the agent's *current* control over its world as seen with its model), and respective values of all *possible* states as perceivable with this model, weighted by their probability (i.e. the agent's possible control), and

it uses this difference for directing the agent's actions towards states that, compared to the current state, would *enlarge its empowerment*. To put it very simple, the organism or the agent simply screens all current options that are perceivable with its on-board means, compares their empowerment values, and then goes for the option that enlarges its empowerment, or in other words, that promises more control, with control here meaning possibilities for further actions. By striving to enlarge its empowerment, the agent simply selects actions that increase its space of possibilities, thereby following a very simple, but universal principle that bases solely on local and therefore dense signals which are directly perceivable by the agent.

Empowerment hence stands for the possibilities that an organism has to act on its world, or in other words, for the control it has over its world, as seen from its own subjective (i.e. its model's) perspective. If you are aware to have options – to stand up, to walk, to talk, to sing, to learn, even to leave the room, etc. – you are *empowered*, if you have no options or you are not aware of them, you are not. The crucial point is that this amount of empowerment again can be expressed in terms of information theory and thus be used for simulations. It is grasped as the *maximal potential causal flow* (Ay/Polani 2008) from an agent's actuators (the means it has for performing some action) to its sensors (the means it has for sensing the state of its world). In terms of information theory, this has been formalized as *channel capacity* (Shannon 1948), that is, as the maximum mutual information, measured in *bits*, that a probability distribution of *received* signals on average contains about the probability distribution of the signals that originally were *sent*. In this case, it would be the conditional probability distribution of observed actuations and sensor dates, that is, the channel capacity between an agent's actions at a certain time and its sensoric stimuli at a later time. It can also be interpreted as the amount of information an agent could potentially “inject” into its world via its actuators and later capture via its sensors.

Consider this in terms of a so called *perception-action-loop* (Powers 2005). In this loop an agent chooses an action based on its sensor inputs from the preceding time step, with the effect that this action will influence the state of the world and thus also the next sensor input of the agent, with this cycle repeating and thus defining an information flow from the agent's actuators to the agent's sensors at a later time step. The maximum of this mutual information, measured as the capacity of the channel between actuators and sensors, here is called *empowerment*, that is, the number of options for actions given the world as perceived through the sensors.

Note that empowerment defines a possibility space for action, from which then, if following the empowerment principle, the action that promises the largest next-

step-empowerment is chosen. The next step can be just one, or it can be a whole range of next steps, thus defining a horizon of possibilities. How large this horizon is, depends on how many next-steps an agent can look into the future, or in other words, as we said, how complex the model is that the agent is able to maintain.

All this may become clearer, if we consider the following example of an agent being positioned on a 2-dimensional finite grid world, in which five kinds of actions are possible: to go North, to go East, to go South, to go West or to stay where you are. An agent with a horizon of just one time step will be able to check these five possibilities in respect to the neighboring patches around him. Positioned in the center of this world, it will perceive that it has these five possibilities. Positioned at one of the edges of this world however, it will perceive that it has one possibility less. Positioned in a corner, it even has two options less. If this agent is made to follow the empowerment principle, it thus will move away from the corners and edges, because avoiding edges means higher empowerment. The picture below shows such an agent as it moved two steps according to the empowerment principle, with the one-step-horizon empowerment values indicated. The first step drove the agent to the South, away from the corner where its empowerment-value was only 1.58. On the patch to the South however, with a value of 2, it found that it can still sense a better empowerment value on the western patch. There, with a value of 2.32, no higher empowerment can be found.

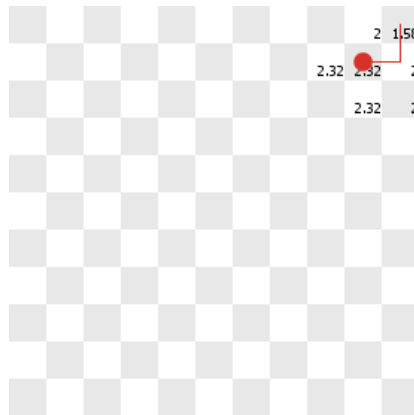


Fig. 6: Agent with one-step empowerment (own illustration).

Now consider the same agent with a larger horizon, that is, with the ability to look further into the future, that is, further into its space of options. This agent may

sense the empowerment constraints of the world's edges even further away. It thus will move towards the center of the world, because there its perceived empowerment is largest. The picture below shows the path of an agent with a 4-step empowerment, again with the empowerment values indicated as perceived by this agent. Note that the values are different to the ones of the 1-step agent above, since they depend on the number of steps an agent can look into the future. In this case, the empowerment is calculated as the base-2 logarithm of visible actions, that is  $E = \log_2(2n^2 + 2n + 1)$  for patches without constraints, with  $n$  indicating the number of steps the agent can look into the future.

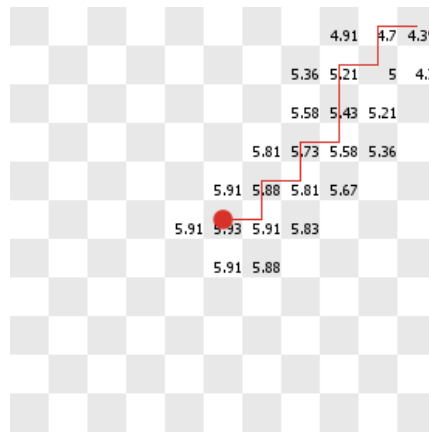


Fig. 7: Agent with 4-step empowerment (own illustration).

The nice feature of the empowerment principle is that it can be applied to a wide range of problems, where not more than just local information about the current situation is needed to drive an agent towards a higher-level goal. In other words, no historical experience is needed. The example below for instance shows an agent trying to find a way out of a compartmented part of the grid (yellow) through a relatively long bottleneck by simply following the empowerment gradient uphill. Note that the agent at each step of its path has no other information than the empowerment value of the adjacent free cells as calculated by looking  $n$  steps into the future.

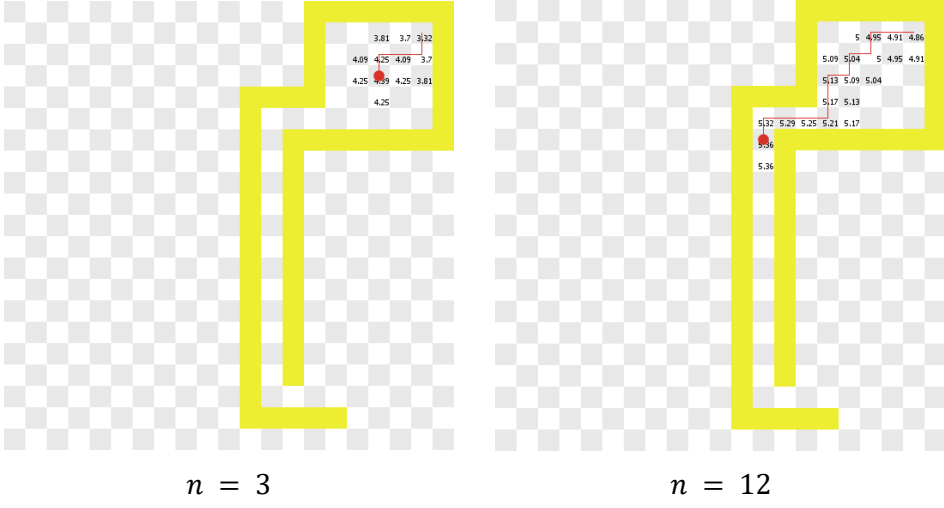


Fig. 8: Agents with 3-step (left) and 12-step (right) empowerment (own illustration).

In the shown examples, the horizon of the agent is still too small to escape from the compartment. In the left case, with a horizon of  $n = 3$ , the agent considers the center of the compartment as providing the largest empowerment and consequently, as in the preceding example, moves there. In the right case, the horizon of  $n = 12$  suffices to let the agent find the bottleneck as possibility to enlarge its empowerment but then, being there, does not perceive the still higher empowerment at the end of the tunnel.

In the following two examples, with  $n = 13$  and  $n = 20$ , the agent succeeds to flee the compartment and to traverse the bottleneck. In the left case, with  $n = 13$ , it is able to perceive the center of the open space to the left of the compartment as enlarging empowerment and consequently moves there. In the right case however, with  $n = 20$ , its horizon appears to be too large for this open space. It obviously can compare the empowerment of nearly all cells on the grid, including the ones in the compartment, and finds that the largest empowerment is to be found to the south of the center. To assess the perception of the agent, the cells in this case are colored in respect to the empowerment that is perceived (the darker, the higher the empowerment relative to the horizon).

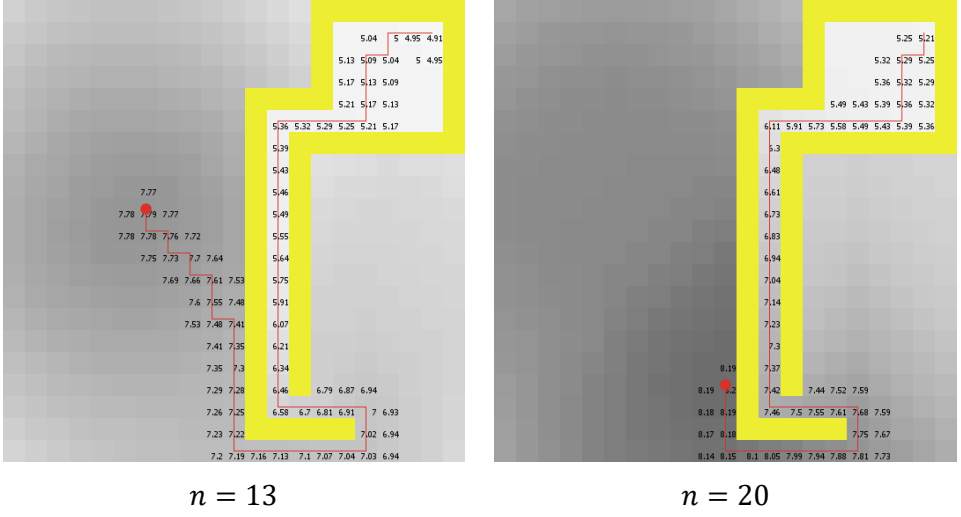


Fig. 9: Agents with 13-step (left) and 20-step (right) empowerment (own illustration).

## 1.5 Causal Entropy

A concept that is very similar to the empowerment principle is the proposal to consider *Causal Entropic Forces (CEF)* as the driving principle for orientating model-based actions (Wissner-Gross/Freer 2013, Cerezo/Ballester 2018).

The concept of CEF connects the idea of the free energy and the empowerment principle. Similar to empowerment, a central aspect of CEF is the horizon that an agent can screen with the help of its model. In most cases this will be a temporal horizon, conceived as a so called *causal cone* (below image), which comprises the set of all paths the actions of an agent can possibly take, starting from an initial state  $x(0)$  and allowed to act continuously over a time interval of length  $\tau$ . Within this cone, *causal slices* (blue) can be conceived, which consist of the consequences of the agent's actions at time  $t$ . Time  $t$  is chosen so that it is possible to score these consequences, or more precisely the actions that led to these consequences. In the empowerment concept a similar scoring was done by the empowerment value. Here the authors, quite generally at first, suggest a *reward function* – for instance a score of 1 for a state when the agent is still alive due to its actions, and 0 when it is dead – and relate this function to a policy that scans all consequences at time  $t$  for deciding on the next action in time  $t + 1$ .

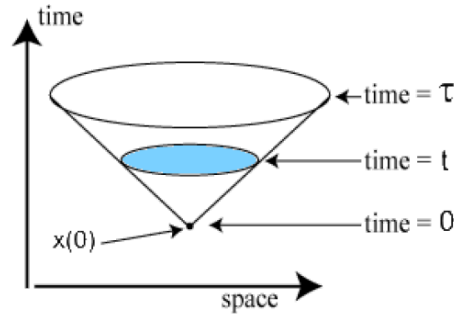


Fig. 10: Temporal horizon as causal cone (own illustration).

Remember that, like in the empowerment concept, scoring is done virtually in the model of the agent, not in regard to real actions. The idea again is to have a set of virtual test pilots – here called *walkers* – prescreening the possibility space. We could say, these walkers simulate actions in the model and the scanning policy then determines the distribution of the consequences of these actions.

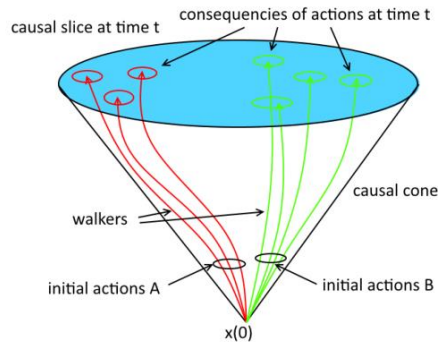


Fig. 11: Density distribution of walkers (own illustration).

One important parameter in this is the distribution of walkers and hence their density  $D$  in different parts of the slice. In the image to the right and the left-most image below the density of walkers is higher in the right than in the left half of the slice (4 green walkers to 3 red walkers). As said, each slice corresponds to the outcome of (virtually tested) actions at a time  $t$  in the future, at which the potential reward of each simulated action can be assessed with a reward function. Imagine a scenario, in which the density of these rewards is higher in the left half of the slice, whereas

the density of walkers is higher in the right half (second-left image below). In order to orientate the screening process of possible futures, it would be good to make the density of walkers proportional to the density of rewards in the slice. To achieve this, walkers are assigned a probability in respect to the level of their reward and the density of walkers at their position in the slice, which then determines whether they are cloned from a low-reward to the high-reward area. In this way, it is guaranteed that walkers prevalently screen promising futures while not completely ignoring the less promising ones which could proof advantageous in another moment of time.

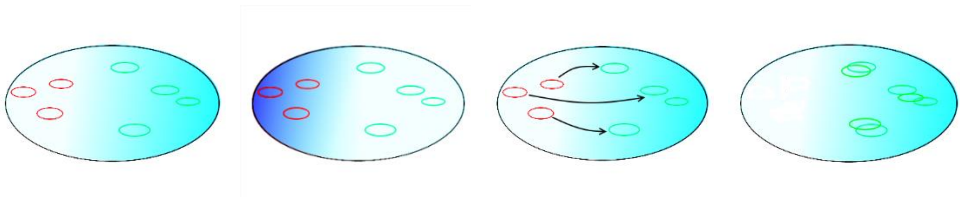


Fig. 12: Probability distribution based on walker density (own illustration).

This relatively simple concept can be applied to a wide range of tasks, which all focus on the one principle of maximizing the number of possible futures. Wissner-Gross/Freer (2013) for instance show that it easily solves the classical problem of balancing a rod upright on a moving cart. Cerezo/Ballester (2018) show impressive examples of steering a simulated spaceship with a hook on a rubber band forming a chaotic oscillator that is highly sensitive to small changes in initial conditions. The hook is meant to catch and deliver a rock into a special zone in a closed environment (see left image and <https://youtu.be/HLbThk624jI>), thus providing a tough goal for the experiment. In another setting Cerezo et al. (2018) apply the principle to 55 Atari 2600 games from OpenAI Gym (<https://gym.openai.com/>), a feat that lately has become a widely used benchmark for AI methods.



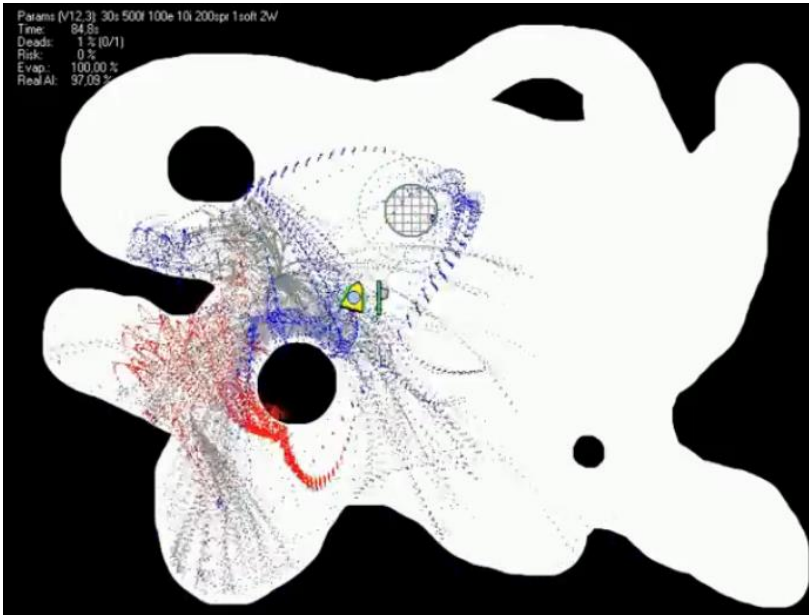


Fig. 13: Chaotic oscillator spaceship, taken from Cerezo/Ballester (2018).

They show that the Causal Entropic Forces principle, or more precisely their fractal version of it, can play these games faster and more efficiently than human players and some of the most advanced deep learning methods can. In their explanations, the central principle of maximizing the entropy of a system (or trying to capture as many future histories as possible) is ambitiously linked to a general theory of intelligence unifying so diverse fields as cosmology, computer science, animal and human behavior and thermodynamics.

In the context at hand, the essential point is the effect of deploying a *model*. Generating and using a model internalizes the Darwinian evolutionary principle of deploying a large number of instances for selecting the few options among them, which promise to maintain the homeostasis of a dynamic system. Other than in natural evolution, where a great number of instances have to be sacrificed to find the few fittest, these anticipatory systems do not waste *real* but just *virtual* instances for *pre-testing* their options.

# 2 Machine-based modeling – aka Machine learning

In general, one could say that modeling is a process of reducing the complexity of a phenomenon in a *virtual* setting to a degree that then, in a given *real* context, can be handled. Despite this complexity reduction however, a model should always remain complex enough to reproduce all important and characteristic traits of the phenomenon. A model should not over-simplify. Modeling thus consists of reducing complexity on the one hand, while keeping those aspects in focus that are thought to be most relevant or most descriptive for the phenomenon. Since usually, many traits that are thought to be characteristic for one phenomenon can be found in others as well, this implies a good portion of *generalization*.

To give an example: say you talk to ten people about the environmental situation of our planet and observe that the ones who think that we might run into a problem seem to have obtained higher education than the ones who rather are indifferent to the situation. You assume environmental awareness and education to be correlated. Consequently, you assign your observations a nominal value between 1 and 10, arrange everything in a table and then draw the values in their relation to each other onto a scatterplot, as shown below.

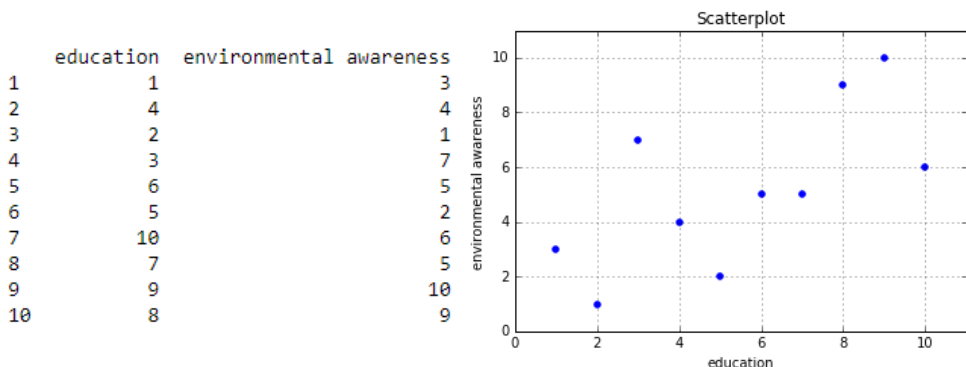


Fig. 14: Higher education and environmental awareness (own illustration).

The distribution of the observed data looks as if you are right. Environmental awareness seems to rise with education. So, you decide to derive a *model* that can give you a general idea about how much environmental awareness rises with increasing education. A simple way to do this is to draw a *linear regression line* through your observations.

## 2.1 Analytical modeling – Linear regression

A linear regression line can be defined as the line that is obtained when minimizing the distances of the data points to the line. In figure 15 these distances are represented as the length of the short black lines that connect the red line to the blue data points. The red line is the linear regression line.

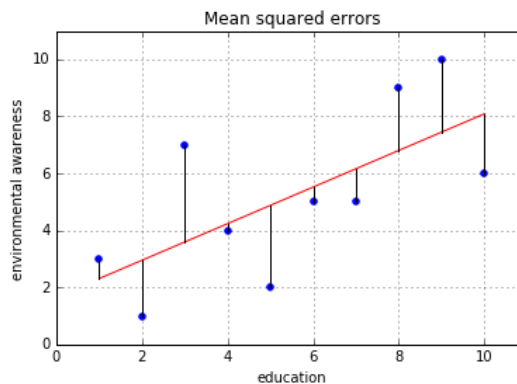


Fig. 15: Linear regression line and distances to data points (own illustration).

As you can see, the red line has a slope that is increasing from the left to the right. This slope represents the relationship that environmental awareness has to education in our example. According to this model, when education increases by one unit, environmental awareness seems to increase by an amount that is represented by the slope.

Mathematically, lines can be expressed by their slope and an interception point, which usually is the intercept with the y-axes. A formula for a line could be  $y = m * x + b$ .

$x + k$ , with  $m$  indicating the slope and  $k$  the intercept on the y-axes. In order to obtain a linear regression model hence, you have to find the two parameters slope and intercept. Analytically, these parameters can be calculated with the following formulas:

$$m = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

and

$$k = \bar{y} - m\bar{x}$$

where  $x$  would be the observations of the educational state,  $y$  the instances of the environmental awareness state and  $\bar{x}$  and  $\bar{y}$  the means of both respectively.

Fortunately, with a computer at your disposal you don't have to calculate linear regression yourself. In Python, there are several specialized packages for doing this, with one of the most comfortable maybe being the linear regression module in the statistical package *scipy* ([www.scipy.org](http://www.scipy.org)), which uses the so called *least square method*. The code below draws the data, calculates a linear regression and draws the line through it, as shown in figure 15.

```
from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# data
edu = np.array([1, 4, 2, 3, 6, 5, 10, 7, 9, 8])
envo = np.array([3, 4, 1, 7, 5, 2, 6, 5, 10, 9])

# apply linear regression to data
slope, intercept, r_value, p_value, std_err = stats.linregress(edu, envo)

print(slope, intercept)

# calculate regression line
line = []
X = range(1, 11)
for x in X:
    line.append(slope * x + intercept)

# plot
plt.plot(X, line, 'r')
plt.scatter(edu, envo, color='b', marker='o')
plt.xlabel('education')
plt.ylabel('environmental awareness')
plt.title('linear regression line')
plt.xlim(0, 11)
plt.ylim(0, 11)
plt.grid()
```

Fig. 16: Python code for computing a linear regression.

The model that is generated in this way is both, an *abstraction* of the observed data since it does not account for the specifics of the individual data points. And at the same time, if the regression line is determined correctly, it also may allow for a *generalization* of the correlation of education and environmental awareness. The model, if correct, could be applied to other observations of which, for example, only the state of education is known. Having a set of educational states plus the model would then allow deriving a notion of the corresponding states of environmental awareness from it.

In many cases, the parameters of a model can be found *analytically*. In the example above for instance, slope and interception of a linear regression line were found with the help of well-known mathematical formulas. Not for all phenomena however, mathematical methods will be at hand. Although it is no problem to derive even highly complex multivariate correlations mathematically<sup>4</sup>, it is usually quite difficult to find appropriate parameter values mathematically. However, – *and this is where the realm of machine learning begins* –, model parameters can also be found and tweaked iteratively with the help of a computer algorithm. In our example above for instance, the slope and intercept parameters of the linear regression model can be found by *iteratively adjusting* the squared distances of the data points to the line so that their overall distance is minimized. The code-example below does this by using the *fit*-function of the linear regression module of Python’s machine learning library *scikit-learn* (<http://scikit-learn.org>). The model parameters in this case are considered as *weights* that are changed with regard to a *cost function*<sup>5</sup>, that is, to a function that indicates in each iterative step how bad the intermediate line as suggested by the current state of the weights fits the average least square distance. This “bad” fit is called the least square error and is tried to be minimized by successively (i.e., iteratively) adjusting the weights. In simple cases this adjustment could be done just by hand, but with more complex models of course it makes sense to have this efficiently done by a computer. In such cases we speak of machine learning. The model used for expressing the relation between education and environmental awareness is generated by the machine.

---

<sup>4</sup> This fact led John von Neumann to the famous formulation: “With four parameters I can fit an elephant, and with five I can make him wiggle his trunk”, see: <https://www.youtube.com/watch?v=KfNPAX-plLbc>

<sup>5</sup> Note: the alternative of a *cost function* is an *utility function* that indicates how good a models’ prediction fits the data.

```

from sklearn import datasets, linear_model

# data
edu = np.array([1, 4, 2, 3, 6, 5, 10, 7, 9, 8])
envo = np.array([3, 4, 1, 7, 5, 2, 6, 5, 10, 9])

# transform to format as demanded by sklearn-models
edu = [[x] for x in edu]
envo = [[y] for y in envo]

# define the ML-model to be used
regr = linear_model.LinearRegression()

# train the model (tweak its parameters)
regr.fit(edu, envo)

# plot
plt.plot(edu, regr.predict(edu), color='red')
plt.scatter(edu, envo, color='b', marker = 'o')
plt.xlabel('education')
plt.ylabel('environmental awareness')
plt.title('linear regression line')
plt.xlim(0, 11)
plt.ylim(0, 11)
plt.grid()

```

Fig. 17: Code-example, using the *scikit-learn* machine learning library.

Note that, as with natural evolution, the iterative adjustment of parameter values usually does not yield *exact* solutions as in mathematical analysis. Most often solutions found by machine learning methods are approximations – approximations however, which in many cases are good enough to be useful.

## 2.2 Linear and non-linear data discrimination

Note also that it is a crude simplification to assume that education is correlated with environmental awareness in a simple linear way. Whereas education, in terms of acquisition of knowledge, theoretically could be increased infinitely, environmental awareness rather seems to run up to a level at which either you have it or you don't, but you are not going to increase it any further – at least not *linearly*. Its development rather appears to saturate at a certain level at which it does not make sense to conceive further increases.

Linear Regression is, as the name indicates, a *linear data discrimination method*. It models data with a straight line. A little bit more complex is the *non-linear data discrimination method* *Polynomial regression*. Using a simple quadratic equation and some added Gaussian noise to model the education/awareness-relation as just described, we can fit data distributions in such cases with the *PolynomialFeatures* module of *scikit-learn*. The result may look similar to the plot in figure 18.

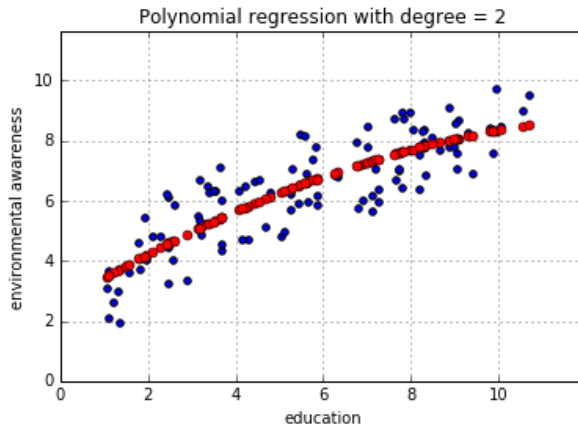


Fig. 18: Data distribution fitted using the *Polynomial Features* module (own illustration).

This module allows the adjustment of a so-called hyper-parameter which is called *degree* (see `sklearn.preprocessing.PolynomialFeatures`), letting you indicate the degree of the polynomial that is considered for fitting the data. If you would assume a more complex relation of environmental awareness and education, with maybe a couple of more influencing factors, you could try to use higher degree polynomials by setting this hyper-parameter accordingly. A polynomial regression of the same data with degree 10 is shown in figure 19. Note however, that in principle you can fit nearly every data distribution by using high enough polynomials. So, if the distribution of your data points is caused by simple relations plus pure noise (as in the example at hand) and not by any hidden extra influences, then using higher order polynomials will cause your model to overfit, meaning that it approaches an accurate representation of this one particular data set, but does not generalize to similar cases anymore. We will come back to this important problem of overfitting a little bit further down the line. For now, keep in mind that learning implies the ability to

generalize. You will not want to learn to drive your car just on the handful of roads that you use for training.

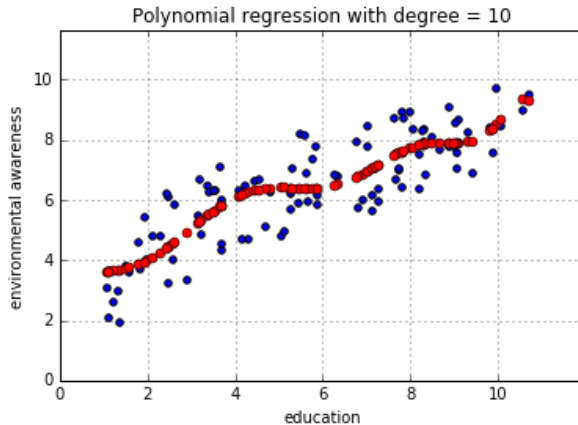


Fig. 19: Polynomial regression, fitted with degree 10 (own illustration).

So, environmental awareness and education may relate *non-linearly* to each other. But of course, there is more to it. Environmental awareness does not depend on education alone. There may be a lot more aspects responsible for whether a person considers the state of the environment at risk or whether she is not able to see any problems at all. In terms of modeling, the number of aspects (aka *features*) considered in such a setting increases the complexity (the *dimensionality*) of the model and may lessen its accuracy and its predictive power. Basically however, the principle remains the same, just the number of parameters needed for defining a valid model may change when considering non-linear relationships and/or more than two features in a machine learning quest. Fortunately, due to today's powerful digital means, the dimensionality of a problem does not inhibit machine-based modeling attempts. Contemporary machine learning makes it possible to consider large amounts of features that are interrelated in complex non-linear ways for determining aspects like a person's environmental attitude. Modeling in this respect is no longer an activity done by humans but by powerful digital devices.



## 2.3 Data preparation

A higher dimensional problem from a similar background that can be tackled with machine learning is the question whether particular characteristics and attitudes (i.e., features) of people can be interrelated in a model so that a general inclination for adopting a photovoltaic system can be derived. The data set for this investigation was obtained in a sociological survey undertaken in Austria in the year 2015 in a range of communities that installed communal photovoltaic systems and were wondering what kind of people participate in this cooperative activity. (Reinsberger et al. 2015). People in these communities were asked to indicate the strength of their attitudes towards aspects like environmental protection, financial assets, believe in technical progress, energy autarky etc. on a graphical scale which then was translated into a Likert-scale with nominal values ranging from one to five. We take this data as an example for how data needs to be pre-processed in a machine learning investigation. An excerpt of the data set is shown below in the form of a so-called *DataFrame*-object generated with the Python data analysis module *pandas* (<https://pandas.pydata.org/>). The rows in this *DataFrame* indicate individual *instances of the data set*, in this case the people questioned. The columns hold the *features* used for characterizing the instances, and the entries in each cell are the *attributes* of each person's features.

	target	envir. protection	financial asset	believe in techn. progress	energy autarky	energy aut. of community	environm. awareness	appearance of home	pers. social capital	social capital of community	appearance of community	comm envir prote
18	2	1.666667	1.666667	2.25	4.000000	2.000000	2.333333	3.0	NaN	2.333333	2.5	1.000000
19	2	1.666667	2.000000	1.00	4.333333	3.333333	3.888889	2.0	NaN	3.666667	4.0	2.333333
20	2	-999.000000	-999.000000	2.00	-999.000000	3.000000	3.333333	-999.0	NaN	3.000000	3.0	3.000000
21	2	1.000000	1.000000	1.50	1.000000	1.000000	2.555556	1.0	NaN	3.000000	2.0	1.666667
22	2	1.666667	1.333333	2.00	2.666667	-999.000000	3.000000	1.5	NaN	-999.000000	-999.0	-999.0

Fig. 20: Glimpse of survey data on PV-attitudes (own illustration).

The data set is small and could be easily scrutinized with common statistical means. But its size and its characteristics make it a good introductory example on which fundamental methods of machine learning can be explained.

As you can see, the first column is called 'target'. It holds information about the final decision of the respondent whether to adopt or not adopt the photovoltaic system.

Sociologists used the numbers 1 for indicating adopters and 2 for non-adopters, which is un-common in machine learning context. For Boolean values usually 1 and 0 would be used respectively.

Also, you probably noticed numbers like -999.000000 in the data excerpt. Obviously, these numbers are not in the range between 1 and 5. Sociologists used these numbers for indicating cases where they did not get clear answers in the range of the Likert-scale. Usually, knowing that -999.0 is not a valid answer, does not impede to proceed with interpreting the data. In a machine learning context however, exposing such numbers to an algorithm would be highly misleading. To get rid of such numbers or to exchange them for others is typically done in the course of data pre-processing.

```
# change target values
data['target'] = data['target'].map({1: 1, 2: 0})

# exchange -999s for NaNs
data = data.replace(-9, np.nan)
data = data.replace(-88, np.nan)
data = data.replace(-99, np.nan)
data = data.replace(-999.0, np.nan)
data = data.replace('#NULL!', np.nan)

# replace NaNs with mean of columns
data = data.fillna(data.mean())

print('This data holds %s instances' %len(data))
num_adopters = sum([1 for x in data['target'] if x == 1])
print('The number of adopters is: %s' %num_adopters)
data[18:23]
```

Fig. 21: Example code, treating misleading numbers.

The code in figure 21 gives an example of how such misleading numbers can be treated. It changes the target-values 2 to 0 and it replaces the sociologists' codification of unusable answers at first with so-called 'NaNs', which stand for "Not a Number", and then replaces the 'NaNs' with the mean of the column fractions with equal targets in which they are found. Finally, it provides us with information about the size of the data set and the number of adopters in it and reprints the below data excerpt to show the applied changes.

This data holds 913 instances  
The number of adopters is: 259

	target	envir. protection	financial asset	believe in techn. progress	energy autarky	energy aut. of community	environm. awareness	appearance of home	pers. social capital	social capital of community	appearance of community	comm envir. prote
18	0	1.666667	1.666667	2.25	4.000000	2.000000	2.333333	3.000000	NaN	2.333333	2.500000	1.000
19	0	1.666667	2.000000	1.00	4.333333	3.333333	3.888889	2.000000	NaN	3.666667	4.000000	2.333
20	0	1.398168	1.784238	2.00	1.563053	3.000000	3.333333	1.844542	NaN	3.000000	3.000000	3.000
21	0	1.000000	1.000000	1.50	1.000000	1.000000	2.555556	1.000000	NaN	3.000000	2.000000	1.666
22	0	1.666667	1.333333	2.00	2.666667	1.793641	3.000000	1.500000	NaN	3.012648	3.210851	1.712

Fig. 22: Modified data (own illustration).

## 2.4 Supervised learning and Information gain

In the case of this survey, people were asked about their attitudes towards photo-voltaics *after* they had decided whether to participate in the communal system or not. Their decision is known and thus can be related to the other surveyed features in order to build a model for generalizing the information it contains. A typical machine learning task with regard to this example would be to analyze the relation of feature values to the target values and to use it for adjusting (*fitting*) the parameters of a model so that this model then can *predict* the decisions of people of which just the features but not the target values are known. The task hence consists in making an algorithm find and learn the regularities in the relation of features to targets so that it can be applied to cases where features but no targets are known. To make this still clearer, consider a bank collecting information on clients in order to find out who of them is not going to pay back credit loans. The bank will try to derive the regularities in the features of those who paid back in the past, and it will also investigate the feature patterns of those who did not pay back. It will use the paid/not paid target values in relation with the features for learning. This kind of ML-learning is called *supervised learning*.

Obviously, some of the analyzed features will be more indicative than others. Therefore, a first step in machine learning often is to find out which parts of the data hold most information in respect to the target. This is done by a methodology called *information gain*, using concepts like the Gini-coefficient or the Shannon-entropy for determining the correlation of features to targets.

Consider a slightly different and dramatically downsized dataset for the PV-adoption example. It has just five instances, three features and a target value that is named “Adopt”.

Name	Account balance	Age	Employed	Adopt
Alfred	450000	45	Yes	Yes
Bob	35000	34	No	No
Chris	115000	37	Yes	No
Dora	29000	25	No	No
Edgar	72000	51	Yes	Yes

Fig. 23: Downsized dataset for the PV-adoption example (own illustration).

Looking at the set may give you a quick idea about which features are more indicative than others. Obviously, the feature “Account balance” holds three people whose balance is above 70.000, Alfred, Chris and Edgar, of which only Alfred and Edgar adopted a PV-system. Similarly, the employed feature indicates the same three people as having a job (and thus a regular income), but again only two of them adopted the PV-system. The age-feature however, unambiguously indicates older people, Alfred and Edgar who are both above 40, as having an inclination towards PV-adoption. Obviously, age, when split for example at 40, is the most informative feature in this set.

In huge data sets of course, with hundreds of features and several thousand instances, this kind of correlation is usually not so easily detected. In the larger example of the communal PV-adoption above other methods to derive feature/target correlations are needed.

Two measures that can be used for this are *Gini-impurity* and *Shannon-entropy*. The Gini-coefficient is used in inequality research. It indicates the distribution of resources, with a Gini of 0 indicating that all resources are evenly spread (all have the same) and a Gini of 1 indicating an utmost uneven distribution (one has all). Analogously, a feature with a Gini-impurity of 0 (when split appropriately, as in the age-feature with a split of 40) corresponds well with the target values and a Gini-impurity of 1 does not give any information for classification. Similarly, entropy, which originates in thermodynamics as a measure of disorder, ranges from 0 to 1, with 0 indicating a pure feature set corresponding in all cases to the target value and 1 indicating maximal impurity and thus providing no information for classification.

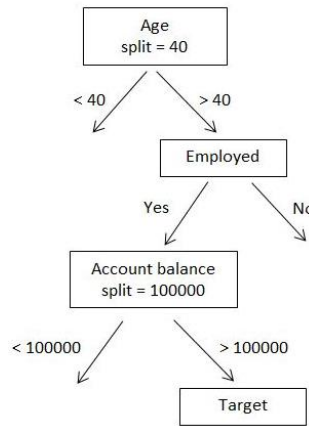


Fig. 24: A decision tree constructed using feature ranking indices (own illustration).

These indices can be used to rank the features of a data set and to construct a so-called *decision tree*, as shown in figure 24. By following its branches downward, the tree, once generated, can be used like a filter for classifying instances of which no target value is known.

Of course, this example provides a very simple tree comprising just three features. Our data set of the PV-adoption decisions will generate a much richer tree. To obtain it, we can again use the *scikit-learn* module of Python, which fortunately offers tools for all the steps needed in an investigation of this kind. In the following we will go through these steps one by one in order to provide a general overview of the possibilities that this methodology offers.

## 2.5 Applying a Decision Tree Classifier

Once we have cleaned and arranged our PV-adoption data as described above, we have to separate the features from the target by putting them into different files. In *pandas* this can be done with the small code snippet below, where *df* indicates the *DataFrame*-object in which the data is stored. The variables *X* and *y* will hold the feature and target values respectively.

```
X = df.drop('target', axis=1)
y = df['target'].astype(int)
```

Fig. 25: Code snippet for separating feature (X) and target (y) values.

Once this is done, the data in *X* (the feature values) should be transformed to a common scale. In the case of the dataset on PV-adoption this may be of less importance since all features range between 1 and 5 anyway. In the small example data set above however, features like “Age” with a range from 25 to 51 are considered together with features like “Account balance” ranging from 29.000 to 450.000, that is, a couple of orders of magnitude larger than “Age”. Such differences in scale can “irritate” machine learning estimators and thus should be avoided. The module *scikit-learn* offers various scalers that can be used to transform data accordingly. In this case, we use the module’s *StandardScaler* to arrange the values around a mean of zero with unit variance. Do not forget to save the scaler once it is fitted to the data. You’ll need it again when you want to apply your trained estimator to unseen data (that is, data that was not used for training). Unseen data has to be brought into the same scale before it can be classified with the trained estimator. Scaling and saving can be done with the small code snippet below.

```
from sklearn.preprocessing import StandardScaler
from sklearn.externals import joblib

# scaling
scaler = StandardScaler().fit(X)
X_scaled = scaler.transform(X)

# save scaler to disk
speicherpfad = 'C:\\your_directory_path\\'
filename = 'fitted_StandardScaler.sav'
joblib.dump(scaler, speicherpfad + filename)
```

Fig. 26: Code snippet for scaling and saving.

The next step is an important one. As said, *scikit-learn* offers a specialized module for generating a decision tree model, which after being trained can be used to classify data of which no target value is known. Training works analogously to our initial example about education and environmental awareness. The branches and the split values in the tree can be seen as the model’s parameters, which in this case again are not mathematically defined, but iteratively tweaked with regard to a best-

fit function. In most cases, the result of this iterative adjustment process will be a model that is not absolutely perfect in reproducing your data. However, this is no problem, but to some extent even necessary, since, as we said, you want your model to be able to *generalize* what it learned to other instances. If your model reproduces your data 100%, chances are high that it has learned to replicate just this one particular data constellation and is not able to recognize any comparable regularities in any other data set. This is called *overfitting* and can be seen as a kind of one-track specialism. The model is useless in all but this one particular case. We will come back to this troublesome problem a little bit later.

So your model will (hopefully) not be perfect. But of course it should do better than guessing. For being able to find out how well your model performs, a first common procedure is to split your data set into a *training set* and a *test set*. The former will be used for actually adjusting the parameters of your model so that it reproduces the essential regularities in the data. And the latter will then be used to test the quality of this adjustment. Since training and test sets are both taken from the original data set for which all target values are known, this provides a simple method for checking the results of the learning process. The so-called train-test-split can be done in the following way, where we separate a fraction of 75% of the data for training and keep another 25% (`test_size=0.25`) for testing.

```
from sklearn.model_selection import train_test_split  
  
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.25, random_state=42)
```

Fig. 27: Train-test-split with 75% training data and 25% testing data.

Now, that everything is prepared, we can define the machine learning-tool we want to use, in this case a decision tree classifier, and apply it to the training set of the data. For this, we need just three lines of code, as shown below. All machine learning-tools of the *scikit-learn* module (and of comparable other modules) can be accessed in this way, so that once the preparation of your data is done, you can try a broad range of methods simply by changing the name of the tool.

```
from sklearn import tree

# define the model
tree_clf = tree.DecisionTreeClassifier()

# apply it in default state to data
tree_clf.fit(X_train, y_train)
```

Fig. 28: Defining and applying a decision tree classifier.

As mentioned in the annotation of the code, we applied the decision tree classifier to the data in its default state. This means, we use the tool as it comes out of the box of the *scikit-learn* toolkit. Most of these tools however, have so-called *hyper-parameters* for fine-tuning their mode of internal operations. The use of the Gini- or the Entropy-measure for the purity of features for example are determined by such hyper-parameters. Gini-impurity is the default criterion for the *scikit-learn* decision tree. Changing it to entropy needs the insertion of `criterion='entropy'` into the parenthesis after the classifier name, like so:

```
tree_clf = tree.DecisionTreeClassifier(criterion='entropy')
```

Fig. 29: Changing the hyperparameter to 'entropy'.

## 2.6 Obtaining and evaluating results

Having the model trained, we want to know how well it performs. For this, we expose it to the test features that we separated with the test-train-split in the file *X-test* and save the result in a file called *clf\_prediction*. This can then be evaluated with a couple of tools, again in this case provided by *scikit-learn*. The code snippet below yields the following output. We can see four different standard performance measures often used in evaluations of machine learning results: *accuracy*, *precision*, *recall* and *f1-score*.



```

from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

clf_prediction = tree_clf.predict(X_test)
print('\nAccuracy: ', accuracy_score(y_test, clf_prediction))
print('\nClassification report:\n', classification_report(y_test, clf_prediction))

```

Fig. 30: Code snippet outputting standard performance measures.

```

Accuracy:  0.606986899563

Classification report:
              precision    recall  f1-score   support

     0         0.71         0.72         0.72        156
     1         0.38         0.36         0.37         73

avg / total         0.60         0.61         0.60        229

```

Fig. 31: Output of above code.

*Accuracy* indicates the degree of closeness of a classification to the actual (true) values. 60% is not an overwhelming result, but after all it is better than guessing, and it is just our first result. *Precision* is the accuracy of positive predictions. It gives an idea of the degree to which repeated classifications under unchanged conditions would show the same results. We can see that when our decision tree classifier claims that an instance is a non-adopter (= 0) it is correct only in 71% of the cases. And when it claims an adopter (=1) it is even less accurate. It is correct only in 38% of the cases. Truly, not a fantastic result.

Precision is rarely used on its own. In the above shown classification report it comes together with a measure that is called *recall* (aka sensitivity), which indicates the fraction of instances that are detected. We can see that only 72% of the non-adopters and 36% of the adopters are classified correctly.

Precision and recall stand in a *tradeoff*-relation to each other. Increasing precision decreases recall, and vice versa. When you try to improve the accuracy of adopter detections (the percentage of correctly classified adopters), you lose out on the percentage of adopters that are detected at all. It depends on what you are up to. You may want to be correct in all adopter detections – then improve precision. Or you may want not to miss any adopters – then improve recall. In the case at hand, we have precision and recall values at similar levels, but for cases where there would

be different levels, *scikit-learn* provides yet another performance measure: the *f1-score*, which is the harmonic mean of precision and recall. Other than the regular mean, the harmonic mean emphasizes the weight of low values. Therefore, the *f1-score* increases only when both, precision and recall, increase together. In order to optimize your results, you should thus try to improve the *f1-score*.

Accuracy, precision, recall and hence also *f1-score* can be calculated from values that often are presented in the form of a so-called *confusion matrix*. These values are called *True Positives (TP)*, *True Negatives (TN)*, *False Positives (FP)* and *False Negatives (FN)* and are used to represent the ratio of correctly to incorrectly classified instances.

```
from sklearn.metrics import confusion_matrix
pd.DataFrame(
    confusion_matrix(y_test, clf_prediction),
    columns=['Predicted non-adopters', 'Predicted adopters'],
    index=['True non-adopters', 'True adopters'])
```

Fig. 32: Code for generating a confusion matrix.

Applying the trained decision tree classifier with the code that is shown above to the separated test data yields the confusion matrix below. It shows that of 156 non-adopters in the test data set 113 were correctly classified, but 43 were falsely classified as adopters. And of 73 adopters only 26 were correctly recognized as adopters and 47 were misclassified. In this case (coincidentally, since the algorithm has no sense for semantics), the 113 correctly classified non-adopters would stand for the True Positives and the 26 adopters for the True Negatives. The 47 predicted non-adopters are the False Positives and the 43 predicted adopters are the False Negatives. Note that the main diagonal holds the correctly classified instances.

	Predicted non-adopters	Predicted adopters
True non-adopters	113	43
True adopters	47	26

Fig. 33: Confusion matrix as generated from of above code.

The following shows how the mentioned performance measures are calculated from these values.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

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

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

$$f1 - score = \frac{2TP}{2TP + FP + FN}$$

## 2.7 Improving results

Applying the train-test-split as shown above, we separated our data set into 75% training data and 25% test data. Since having more examples usually provides more information for the classifier, training on only 75% of the data means forgoing some possibilities.

A common remedy for this shortcoming is *cross validation*, which separates the data set repeatedly into different chunks for evaluation. The following code for example evaluates our decision tree classifier five times on different randomly generated subsets of the data by repeatedly applying the train-test-split, evaluating the accuracy of the separate runs and finally printing the mean and the standard deviation of the results. As you can see, *scikit-learn* again holds a specialized modul to implement this possibility in a simple way.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_clf, X_train, y_train, cv = 5, scoring = 'accuracy')
print(scores)
print(scores.mean())
print(scores.std())

[ 0.60144928  0.72992701  0.72992701  0.72058824  0.72794118]
0.701966540345
0.0503760761252
```

Fig. 34: Code for applying cross-validation.

A somehow similar method for getting better results from machine learning investigations is to use *ensemble methods* for training. So called *voting classifiers* for

example aggregate the predictions from several estimators and then use the result that was predicted most frequently. Surprisingly, due to the *law of large numbers*, this may produce a more accurate result than even the best of these classifiers could produce on its own. You can use different classifier methods for such an ensemble technique, or you can use the same method on different subsets of the data as above in cross validation. If you allow for a replacement of the sampled subsets for training, this is called *bagging* (short for *bootstrap aggregating*), and if you use training instances only one time per classifier, this is called *pasting*. *scikit-learn* once again holds a modul for these options, which, when applied in the bagging mode, as shown below, yields a slightly better result than the decision tree trained on a single 75%-training set (see the table below).

```
from sklearn.ensemble import BaggingClassifier

# bagging
bag_clf = BaggingClassifier(tree_clf, n_estimators=10, bootstrap=True, n_jobs=-1)
# pasting
#bag_clf = BaggingClassifier(tree_clf, n_estimators=10, bootstrap=False, n_jobs=-1)

bag_clf.fit(X_train, y_train)

clf_prediction = bag_clf.predict(X_test)
print('\nAccuracy: ', accuracy_score(y_test, clf_prediction))
print('\nClassification report:\n', classification_report(y_test, clf_prediction))

from sklearn.metrics import confusion_matrix
pd.DataFrame(
    confusion_matrix(y_test, clf_prediction),
    columns=['Predicted non-adopters', 'Predicted adopters'],
    index=['True non-adopters', 'True adopters'])
```

Fig. 35: Code for using an ensemble method.

Accuracy: 0.689956331878

Classification report:

	precision	recall	f1-score	support
0	0.72	0.90	0.80	156
1	0.53	0.23	0.32	73
avg / total	0.66	0.69	0.65	229

	Predicted non-adopters	Predicted adopters
True non-adopters	141	15
True adopters	56	17

Fig. 36: Output of above code.

A particular bagging-method that is very popular with decision trees is called *Random Forest Classifier*. It differs slightly from the above shown bagging classifier in that it searches for the best feature (the one with least Gini-impurity) not among all features, but repeatedly only in a random subset of them. This usually yields a better model with less *variance*, but maybe slightly higher *bias*.

## 2.8 The bias-variance tradeoff, generalization and overfitting

Bias and variance are two important terms in the evaluation of results from supervised machine learning. Both are sources of error that, similar to the precision-recall-tradeoff, demand opposed improvements. Bias stems from an erroneous assumption in the learning algorithm, for example the assumption that the data is linearly separable although it is not, that is, that it can be classified for example with linear regression as we have seen in our first example in the beginning of this chapter. Bias errors result in so-called underfitting (the yellow line in figure 37), indicating that the generated model does not mirror all characteristics of the data, for example the fact that the true separation line is curved (black line in figure 37). Variance on the other hand, indicates high sensitivity to small fluctuations in the training data. The model overfits the data by accounting for small details which are characteristic only for the training set at hand and not for any other data (the green curve in figure 37). As mentioned earlier, this reduces or thwarts the generalizability of the model. The model reproduces just the data at hand.

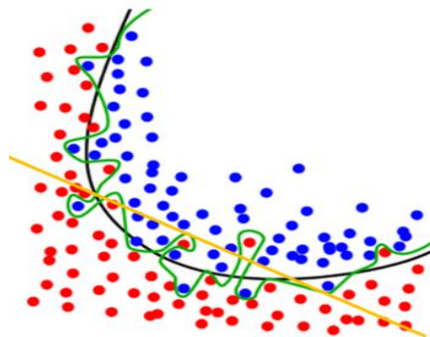


Fig. 37: Underfitting (yellow), overfitting (green), and optimal separation (black), taken from Haake (2022), p. 24.

Apart from bias and variance a third source of error in data is the irreducible error, resulting from noise in the problem itself. This error should be accepted, since it does not reduce the generalizability of the model.

As in our example of PV-adoption, usually, when training an estimator, a first performance indicator to consider is accuracy, the ratio of true classifications. Of course, it is fine to get high accuracy values since this means that the data holds regularities that can be used for classification in principle. However, high accuracy can also indicate an overfitting model. As indicated by the green line in the image above, all data points can be classified correctly, accuracy is high thus, but the model is useless since it will hardly be applicable to any other data. Therefore, in all regularity, overfitting is the worse of the two problems. It is less easily detected, but can render your model worthless.

So, what to do to avoid overfitting? There are several possibilities. A first one is simply to start investigating your data with rather simple estimators (as for example the linear regression estimator), use the results as benchmark and gradually step up with the complexity of your tool to see if the additional complexity is worth it. Increasing a model's complexity will increase its variance and reduce its bias. If you should happen to run into two models with comparable performance, then, according to the rule of Occam's razor, pick the simpler one. This will also benefit your algorithm's runtime.

Another possibility is to constrain your models' weights. Remember that the free parameters of a model – slope and intercept in the case of linear regression for example – are seen as weights that are iteratively adjusted in the course of training. These weights determine a model's complexity. If there are many of them (and Artificial Neural Networks, which we will discuss later on, can have hundreds of thousands), they are capable, when moving freely, to account for nearly every small detail of a data set. Constraining them thus, means to impair their adjustability so that rather the broad characteristics are considered and not every irrelevant detail in data. *scikit-learn* holds several easily implementable methods for constraining the weights of a classifier (see for example: Ridge-, Tikhonov-, or Lasso-regularization or Elastic Nets, [https://scikit-learn.org/stable/modules/linear\\_model.html](https://scikit-learn.org/stable/modules/linear_model.html)). However, they do not apply to a decision tree or a random forest classifier.

## 2.9 Data augmentation

An obvious way to keep a model from overfitting is to use more data to train it. Surely, a larger data set will provide more chance to find its relevant characteristics – if there are any. But what if – like in the case of the PV-adoption – there is simply no more data at hand. Your data set is small but you still think that it holds the essential characteristics for training a model. In this case you could just blow up your data by varying it slightly according to statistical considerations. The code below for example, doubles the original PV-adoption data by taking the mean and the standard deviation of all features in regard to whether the target value is 0 or 1 and uses these for generating new normally (around the same mean with same standard deviation) distributed instances and concatenates them with the original data.

```
# generate separate dataframes
# for instances with target = 0
df_target_0 = df[df['target'] == 0]
l_zeros = len(df_target_0)
print(l_zeros)

# and for instances with target = 1
df_target_1 = df[df['target'] == 1]
l_ones = len(df_target_1)
print(l_ones)

# put both in a list
NIs = [df_target_0, df_target_1]
lengths = [l_zeros, l_ones]
targets = [0, 1]

# prepare a new DataFrame that contains the original data
df_augmented = df

for x, item in enumerate(NIs):
    # generate a new dataframe without target values
    df_feat = item.drop('target', axis = 1)

    # write the means of features into a pandas Series
    df_feat_m = df_feat.mean()
    # write the standard deviation of features into a pandas Series
    df_feat_s = df_feat.std()

    # prepare a List of the Lists of new instances to be created
    NIL = []
    # define number of new instances to be created (double original size)
    f = lengths[x]
    for m in range(f):
        # prepare a list of new instances to be created
        # with target value already appended
        NI = [targets[x]]
        for n, feat in enumerate(df_feat_m.values):
            # create new ones normally distributed around mean of originals
            NI.append(np.random.normal(feat, df_feat_s.values[n]))
        NIL.append(NI)

    # create a pandas DataFrame with the new instances
    ni = pd.DataFrame(NIL, columns = df.columns)
    # concatenate data frames
    df_augmented = pd.concat([df_augmented, ni], ignore_index = True)

df_augmented.tail()
```

Fig. 38: Code for doubling the PV-adoption data.

To assume that all feature values are likewise normally distributed is a strong assumption of course. However, with a little bit of patience one could work out the statistical properties of the features in more details and then simply use them to create additional *synthetic data* for training.

## 2.10 Feature reduction

Yet another way to avoid overfitting is feature reduction. As we have seen, features contribute in varying degrees to the predictability of data. Depending on their purity in respect to the target, they provide more or less information for classification – and hence are ranked accordingly in the decision tree. While the small example decision tree in figure 24 was easy to screen for feature importance, the PV-adoption example is more difficult to comprehend. However, it too can be visualized with the help of an online-tool called *graphviz* (<https://www.graphviz.org/>). The code below generates and saves a *.dot*-file to your computer that has to be inserted at <http://www.webgraphviz.com/> in order to generate a tree like the one in figure 40 below.

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file='<path to your directory>\\PV-tree.dot',
    feature_names = df.columns[:-1],
    class_names = df.columns[-1],
    rounded = True,
    filled = True
)
```

Fig. 39: Code snippet for visualization at <http://www.webgraphviz.com/>.

In principle, if you zoom in deep enough into this tree, you could screen the leaves for important features and split-values in the same way as in the small example tree above. However, there are better ways to do this than by optical selection.



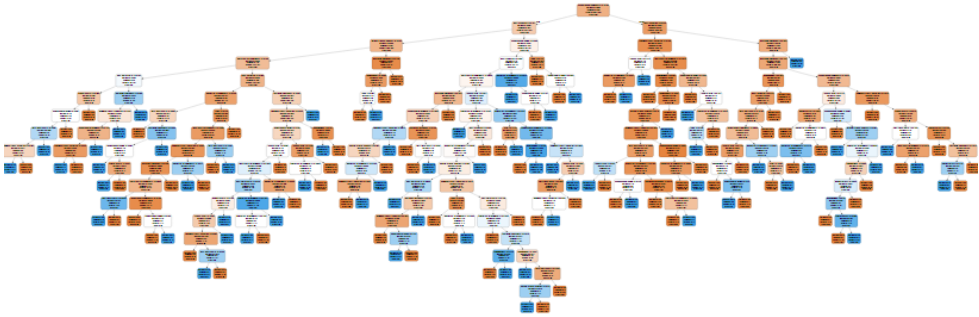


Fig. 40: Decision tree generated with above code (own illustration).

An alternative way to find out about the importance of features is *Gini-importance*. It is computed as the (normalized) total reduction of impurity brought by a feature. *scikit-learn* offers a simple function to rank features according to Gini-importance. It can be plotted with the code snippet below.

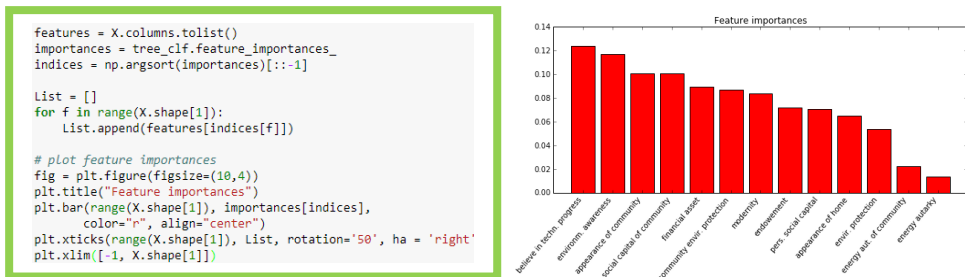


Fig. 41: Calculating and visualizing Gini-Importance (own illustration).

Other options offered by *scikit-learn* are *Recursive Feature Elimination (RFE)* for instance, or the *SelectFromModel* meta-transformer (see [http://scikit-learn.org/stable/modules/feature\\_selection.html](http://scikit-learn.org/stable/modules/feature_selection.html)). While the first one recursively considers smaller and smaller sets of features and checks how much a performance indicator loses out by dropping a feature, the second allows removing features from trained models if indicators like feature importance are beneath a certain threshold. Both allow to determine an optimal number of features to be considered.

Yet another often deployed method for determining the optimal number of features to consider is *Principal Component Analysis (PCA)*. This is an *unsupervised* method,

meaning there are no target values with which an optimal result can be compared. It uses the mathematical technique of Singular Value Decomposition (SVD) to decompose a multivariate dataset into the set of successive orthogonal components, which is responsible for the most variance in the dataset. Its idea is to identify dimensions along which the variance in the dataset enfolds and to keep just those dimensions which explain a large enough fraction of the variance. Often it can be shown for example that 95% of the variance in a dataset can be obtained with a number of components that is much smaller than the number of features. Considering fewer features, as said, prevents overfitting and saves computer runtime.

# 3 Common Machine Learning tools

Apart from Decision Trees and their ensemble mode, the Random Forest classifier, there is a great variety of other machine learning tools in use, of which the lately most dynamically developing ones are Artificial Neural Networks. In this book we will not be able to cover all these tools in detail, but will introduce Neural Networks in the next chapter. Before that, we should at least briefly mention three very powerful and versatile methods that due to their effectiveness are in wide use. They are the so-called Support Vector Machine, the k-Nearest Neighbor classifier and the Naïve Bayes method.

## 3.1 Support Vector Machine

The *Support Vector Machine* (SVM) can be used for linear and non-linear classification as well as for regression tasks. It is well-suited for complex but not too large data sets, since its computational costs in terms of runtime are rather high. In a nutshell, a SVM works by trying to separate classes of data with the widest possible street that can be built between them. The following two plots may explain this principle.

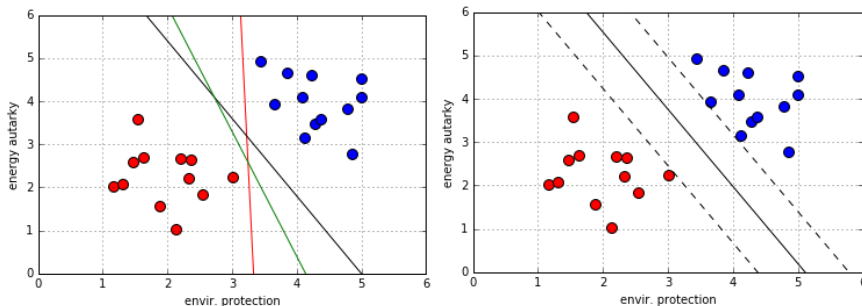


Fig. 42: Left: Three different linear classifiers (red, green, and black).  
Right: SVM classification (own illustration).

Both plots show classifications in two dimensions of the data. In the left plot we see three different linear classifiers (red, green and black), all separating the red and the blue classes. Although being 100 percent correct in each case, the three classifiers draw completely different classification lines. If you apply these models to unseen data (i.e., data *not* used in training), it would depend very much on the orientation of this data whether one of the classifiers would predict class membership correctly. As a contrast, the right picture above shows the working principle of an SVM. The solid black line in the middle is the actual separation line drawn by the classifier so that the two dotted lines to the left and the right define the widest possible street between the red and the blue class. In this case, no alternative separation seems possible. The classification will be the same in each case.

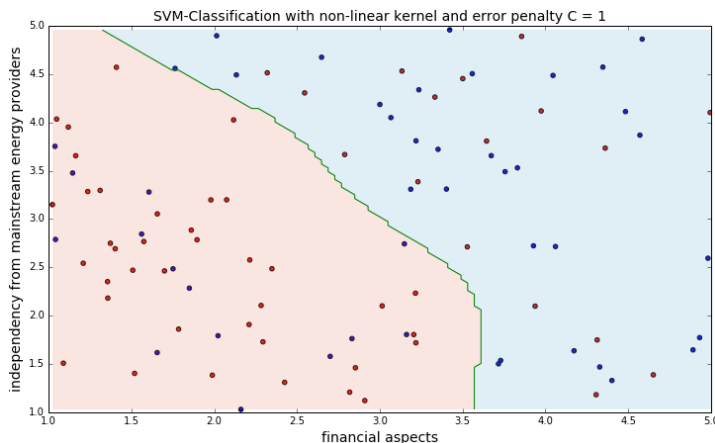


Fig. 43: SVM classification with hinge loss (own illustration).

An SVM thus tries to find the line, plane or hyperplane between classes of data that represents the largest separation, or margin, between them. This separation is chosen so that the distance from it to the nearest data point on each side is maximized, thus defining a *channel*. When applied to real data however, such a channel cannot always be found. The margin thus is considered a *soft margin* indicating the *best* solution that can be obtained. The SVM-algorithm looks for this soft margin by *penalizing* its solutions for misclassified data points and then chooses the separation which yields least penalty, or *loss*, as the error penalty in the context of SVMs is called. A common loss-function used in SVMs is *hinge loss* indicating that the penalty for an incorrectly classified data-point increases linearly with the distance from the

separation. A simpler form of loss is *zero-one-loss*, assigning a penalty of zero to correctly classified points and a loss of one to incorrectly classified points. The plot in figure 43 shows an SVM-classification with hinge-loss.

As many other machine learning-methods introduced in this chapter, SVMs can be used for *regression tasks*. So far, we discussed only classification tasks, that is, problems where instances with particular features have to be sorted into different classes, for example into the classes of PV-adopters and non-PV-adopters. Machine learning can also be used for problems in which you want to predict a value instead of a class, for example the price of a car given its mileage, its engine, its brand, its age, etc.

The code in figure 44 shows a regression example. It generates random data around a cosine-function and uses an SVM with a non-linear polynomial kernel to fit the data values. The result is shown in the plot to the right.

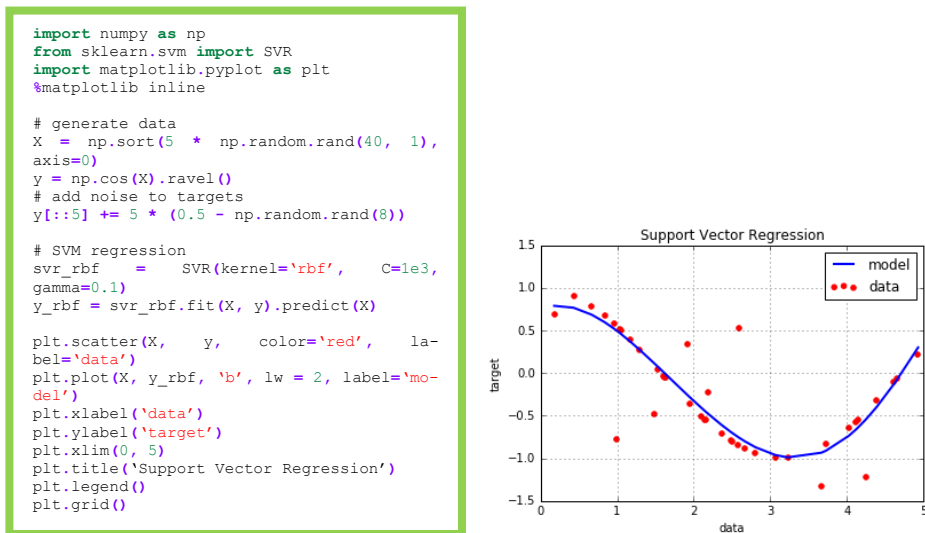


Fig. 44: Left: Code snippet for SVM regression example. Right: Plot displaying SVM-regression results (own illustration).

### 3.2 k-Nearest-Neighbor

Another way to consider linearly not-separable data is to build on the principle of similarity of data points, or more precisely, the similarity of feature vectors. The reasoning behind this is the simple assumption that data instances inhabiting the same region in data space have more in common than data instances from different regions. Since feature vectors have the mathematical form of coordinates, albeit in a higher dimensional space, one common way to compute similarity is *Euclidian distance*.

According to the Pythagorean theorem, in two dimensions, the Euclidian distance of two points A and B with vectors  $(x_A, y_A)$  and  $(x_B, y_B)$  can be calculated with the formula  $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$ , with calculations in higher dimensions analogously. The distance of data points hence can be compared and used to differentiate data points in respect to their distance from each other. So, in the case of *predictive modeling*, that is, in the case of an attempt to predict the behavior of a so far unknown data instance from data on which an algorithm was trained, one could look at (already assessed) close-by data points, so called *neighbors*, and orientate prediction on the behavior of these “nearest neighbors”. If a person’s feature vector is close (in terms of Euclidian distance for instance) to other people who adopted PV, chances are high that this someone will also adopt.

A question concerning this method is how many neighbors shall best be considered. In order to prevent ties, it is often suggested that the number of neighbors – usually denoted with  $k$  (that’s why the method is called *k*-Nearest Neighbors) – should be an odd number and the prediction is then orientated on the majority rule. Another often used possibility however is to weigh neighbors in respect to their distance. Intuitively, one would agree that the closer ones of the  $k$  neighbors should have more predictive power than the ones further away.

The *k*-Nearest Neighbors algorithm can be used for non-linear classification as well as for regression tasks. It can be used in a supervised and in an unsupervised way. The latter is providing a foundation for many other learning methods, notably spectral clustering ([sklearn.cluster.SpectralClustering.html](http://sklearn.cluster.SpectralClustering.html)).

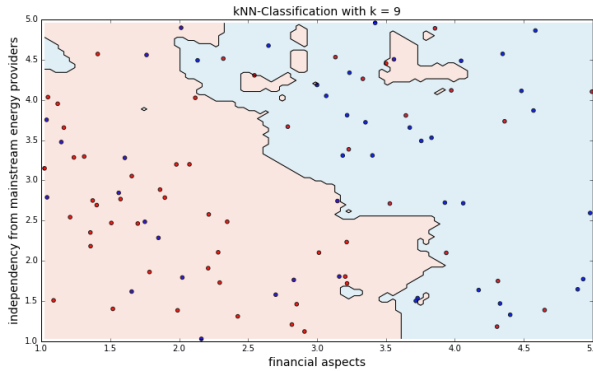


Fig. 45: Classification with k-Nearest Neighbor (own illustration).

### 3.3 Naïve Bayes

Another effective method for non-linear data discrimination is Naïve Bayes, which builds on the famous rule of the English statistician Thomas Bayes (1701-1761) for deriving joint probabilities. The algorithm is often used in email-spam-filters to separate mails from spam. In order to understand its working principle, consider the probability of someone investing in a photo voltaic system given some kind of evidence  $E$ , for example the evidence of having high environmental awareness. This could be indicated as  $p(\text{investment} \mid \text{evidence})$ , reading “the probability of investment given that evidence”.

In general, if there are two events,  $A$  and  $B$ , with the probabilities  $p(A)$  and  $p(B)$ , the so-called *joint probability* of  $p(AB)$ , that is, the probability that both events occur, can be calculated as  $p(AB) = p(A) * p(B \mid A)$  or alternatively  $p(AB) = p(B) * p(A \mid B)$ . This implies that  $p(A) * p(B \mid A) = p(B) * p(A \mid B)$  and so, if both sides are divided by  $p(A)$ , one gets

$$p(B \mid A) = \frac{p(A \mid B) * p(B)}{p(A)}$$

the famous *Bayes' Rule*.

To comprehend this more clearly, let's assume that  $B$  is some kind of Hypothesis  $H$ , for example the hypothesis that someone will invest in a PV-installation, and  $E$  is evidence, for example the high environmental awareness of this person. Renaming Bayes' rule gives

$$p(H | E) = \frac{p(E | H) * p(H)}{p(E)}$$

The advantage of this transformation is that all three terms in it are more easily assessable than the probability of someone investing in PV given high environmental awareness. In many cases, the probability of high environmental awareness given that someone owns a PV-installation, as well as the probabilities of PV-installations and of high environmental awareness in general, that is, its occurrence in the overall population, can be empirically observed.

Now let's call the event that a target variable will take on a particular value  $C = c$ , for example that a person indeed adopts a PV-installation (or that an incoming email is spam). Rewriting Bayes' rule gives

$$(C = c | E) = \frac{p(E|C=c)*p(C=c)}{p(E)}$$

In the data set, the evidence  $E$  is the feature vector of this person (that is, the vector containing all known characteristics of this person).  $p(C = c | E)$  is called the *posterior probability*.  $p(C = c)$ , the so-called *prior probability*, can be taken as the "base rate" of  $c$ , that is, the prevalence of  $c$  in the whole population. The term  $p(E | C = c)$ , which is the so-called *likelihood* of seeing the evidence  $E$  when  $C = c$ , can be computed from the data as the percentage of examples of class  $c$  that have feature vector  $E$ . Finally,  $p(E)$  is the likelihood of  $E$  in general, calculated as the percentage occurrence of  $E$  among all examples.

One problem with this calculation however, is the fact that the feature vectors in the data can be very specific. Usually, they get more specific the larger the vector is. Large feature vectors hence may not allow estimating their probability of occurrence with any confidence, since hardly any vectors will be exactly the same. To overcome this, it is often assumed that the features are conditionally independent given that  $C = c$ , meaning that a probability  $p(e_i)$  does not say anything about the probability  $p(e_j)$  and therefore (if we write  $c$  instead of  $C = c$  for the sake of simplicity) Bayes' rule can be regarded as

$$p(c | E) = \frac{p(e_1|c)*p(e_2|c)*...*p(e_n|c)*p(c)}{p(E)}$$

Each of the  $p(e_i | c)$  terms can be computed directly from the data. Instead of looking for the match of entire feature vectors, it can be derived from the proportion of an individual feature  $e$  in the fraction of  $C = c$ , that is for example in the fraction of PV-adopters.



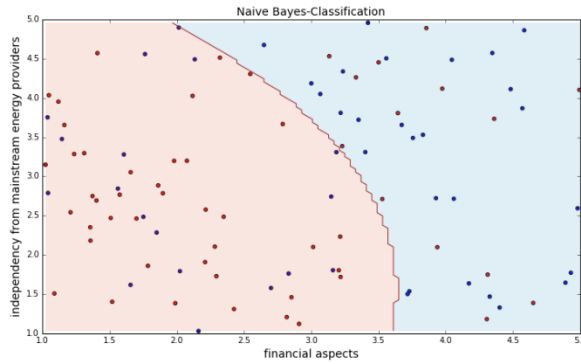


Fig. 46: Classification using Naive Bayes algorithm (own illustration).

An advantage of the Naive Bayes algorithm is that it can be used as an *incremental learner*, meaning that it updates its “knowledge” in real time with every single new instance added to the data set. It does not have to be started anew. Each new adopter or non-adopter, or each new mail or spam-mail adds more information to the system. The plot in figure 46 shows a classification of the PV-adoption data, done with the *GaussianNB* Naive Bayes-algorithm as provided by *scikit.learn*.

# 4 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computer-generated adaptive systems believed to work similar to biological neural networks, in particular to the human brain. Like other machine learning tools as well, they are used to find structures and regularities in complex data sets. Nowadays, due to their efficiency, they are deployed in a wide range of every-day gadgets, from smart phones to car carburetors or washing machines. Most spectacular of course is their application in the context of Artificial Intelligence (AI).

ANNs consist of a large number of identical or very similar components, neurons, which interact with each other on the principle of a rather simple mechanism, a so-called threshold function. In other words, they operate on the basis of a distributed representation of their knowledge. Crucial for their performance is not so much a specialization of individual components but a complex reference structure of many similar and surprisingly simple components with collective, *aggregated* performance. In this regard, they are prime examples of complex systems.

## 4.1 The Perceptron

The essential aspect of the operation of a neural network is the state of excitement of its neurons, or more exactly, the transmission of these states of excitement triggering excitements in other neurons, which amplifies or weakens the connections between these neurons. This transmission can be explained on the example of the operation mode of the so-called *perceptron*, a predecessor-ANN that was suggested by Frank Rosenblatt in 1958 in order to simulate the receptors of the retina.

This perceptron is a computer generated “virtual” network consisting of two so-called input neurons, a hidden neuron, and one output neuron (see figure 47). The two input neurons have connections to the hidden neuron which in its turn is connected to the output neuron. The connections of all these neurons are weighted

with an initially randomly assigned numeric value, often simply between -1 and +1. Note, that the connections in any network can be represented with the entries in a symmetric matrix, with row- and column-titles holding the designations of neurons and the corresponding matrix-entries holding information on whether a connection exists or not (1 or 0), or a numerical value for the strength, i.e., the *weight*, of a connection. In this way, the transmission of excitements in ANNs can be calculated by way of matrix-calculations. ANNs basically exist in the form of large matrices, stored on computers or servers, and being processed with specialized hardware like graphical- (GPUs) or tensor processing units (TPUs). More on this later on.

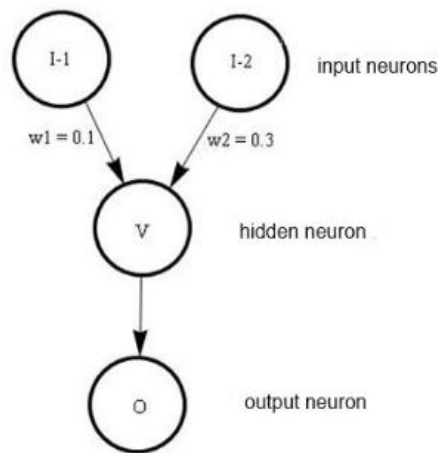


Fig. 47: Perceptron (own illustration).

In order to comprehend the basic working principle of ANNs in more details, consider the task of learning the correct outputs of a logical operation, let's say the inclusive OR-function. This function is meant to answer two Boolean inputs of which either one of them or both of them are *true* (represented by a 1) with a *true* (1) output, and of which both are *false* (represented as 0) with a *false* (0) output (see the input-output-table below). Our machine – in this case a perceptron like the one in figure 47 will do – has to learn to provide these outputs when confronted with the inputs  $I_1$  and  $I_2$ .

$I_1$	$I_2$	output
0	0	0
0	1	1
1	0	1
1	1	1

Fig. 48: Input-output table illustrating the logical OR-function.

In a first step, the information [0, 0] is put into the network via the input neurons and is then processed from there to the hidden neuron by multiplying it with the weight of the connections. The hidden neuron then sums the weighted input values as  $net = w_1 * I_1 + w_2 * I_2$  and rounds this sum on a threshold value, which here for example could be 0.5 in order to determine the actual output value for this first step of learning. In this case, the results from  $net$  larger than or equal to the threshold can be rounded to 1 and the results smaller are rounded to 0, which is possible in this case since there are only binary outputs in the example.

While learning, this generated output value is compared to the expected *target*, which is 0 in the case of two 0-inputs. The actual learning process now consists in iteratively applying the four possible inputs and increasing or decreasing the – initially randomly assigned – connection weights with regard to a learning rate so that the *generated output* in each of the four possible cases coincides with the *expected output*, i.e., the target.

So, if, for example, the initial connection weights were assigned as  $w_1 = 0.1$  and  $w_2 = 0.3$  to the network and the learning rate would be 0.2, the learning process would proceed in the following way:

- In the first step, according to the above table, the two values 0 and 0 are introduced to the input neurons and forwarded to the hidden neuron. Weighted with 0.1 and 0.3 (i.e., multiplied with these values) this produces the  $net$  value of  $0 + 0 = 0$  and thus, even without being rounded on the threshold 0.5, the expected result. Hence, the weights are not changed in this step.
- In the next step, the next two values in the table, 0 and 1, are introduced and passed into the input neurons. With the set weights, this now yields  $net = 0 + 0.3$  and thus, rounded on the threshold 0.5, a  $net$  value of 0, which in this case is not the desired result. In consequence, a weight will have to be changed. Usually, this is the weight with the value that has

changed in comparison with the previous case. It will be increased by a *learning rate* of, say, 0.2 to  $w_2 = 0.5$ . Rounding the *net* value now on the threshold yields 1, which corresponds to the target.

- With the next row's entry of 1 and 0 the weighting results in 0.1 and 0, and thus, rounded on the threshold value 0.5, yields a *net* value of 0, which in this case too is not the desired result. Once again, the weight for the new value, now for  $I_1$ , is increased by the learning rate 0.2 to  $w_1 = 0.3$ . The *net* value now is  $\text{round}(0.3 * 1 + 0.5 * 0) = 0$ , which does not yet meet the expected result, but is accepted for the moment.
- With the last entry of 1 and 1 in this first training round the weighting yields  $w_1 = 0.3$  and  $w_2 = 0.5$  and thus rounded on the threshold 0.5 a *net* value of 1, which in this case corresponds to the expected result. The weight must not be changed.

The learning process has now completed one pass through all possible cases of the data and thus starts over again at the first row of the above table.

- Again, the input values 0 and 0 are passed to the input neurons and now, with a weighting of 0.3 and 0.5 yield a *net* value of 0, the expected result. The weights are not changed.
- With the next entry of 0 and 1 the weighting yields 0 and 0.5, and thus, rounded on the threshold, a *net* value of 1, which corresponds to the desired result. The weights are not changed.
- With the entry 1 and 0 the weighting yields 0.3 and 0, and thus, rounded on the threshold, a *net* value of 0, which does not correspond to the expected result. The weight corresponding to the changed value, i.e., the weight to  $I_1$ , is increased by the learning rate of 0.2 to  $w_1 = 0.5$ . The *net* value now, with  $\text{round}(0.5 * 0.5 * 1 + 0)$ , is 1, the desired result.
- The last inputs of this round are 1 and 1. This yields a weighting of  $0.5 + 0.5$ , which corresponds (even without rounding) to the expected result.

Again, the learning process has completed one pass through all possible cases and starts over again. As it turns out however, with the current weighting all *generated outputs* correspond to the *expected outputs* of the inclusive OR-function. In other words, the difference between *generated* and *expected output*, also known as the *Error*, is zero. The learning process thus is complete in this case. The weighting of the network connections is "fine-tuned" so that each time one of the possible binary combinations is introduced to the input neurons the output corresponds to the expected result.

This iterative approach to coherent weightings describes the basic working principle of neural networks. It works analogously with much more complex problems, even though more hidden neurons and, depending on the size of the input and output, more input and output neurons are used. In cases where the deviation from the desired result is not always 1 as in our example, the error can be multiplied with the learning rate (In our example, a multiplication of 1 would not affect the learning rate) with connection weights being changed accordingly.

## 4.2 Back propagation – the fundament of Deep learning

Compared to the power of today's ANNs, the example of the logical OR operation of course is very simple. Back in the 1960ies however, a similar simple logical operation, the so-called XOR (or exclusive OR) operation<sup>6</sup>, caused temporary irritations in the development of artificial neural networks. In a much-discussed book on Perceptrons, Marvin Minsky and Seymour Papert (1972) pointed out that the solutions to the XOR function cannot be separated by means of a simple straight line in a two-dimensional space and thus cannot be learned in the form described above. Their objection cooled the euphoria about the possibilities of neural networks in the field of artificial intelligence research.

Only a short time later, however, proposals were made to connect several layers of hidden neurons in order to gain higher resolution in so called *Multi-layer Perceptrons* (MLPs). These layered network types, where information propagates through several layers of neurons into considerable processing depths, are responsible for the fact that the corresponding research and application today is discussed under the name *Deep learning*. In such “deep” neural networks each internal layer of hidden neurons serves as a new input for the next layer, which necessitates a different method for adjusting the connection weights when training the network.

---

<sup>6</sup> In contrast to the inclusive OR, the XOR-operator yields the output 0 for the inputs 1 and 1.

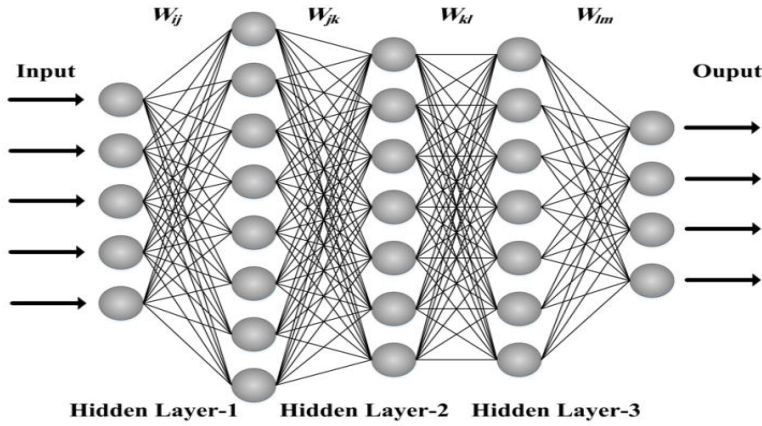


Fig. 49: Multi-layer Perceptron (MLP), taken from Bill et al. (2021)

In simple single-layer Perceptrons, the adjustment of the weights is done immediately in response to the difference of expected and generated output. The adjustment, so to speak, is an immediate reaction of the network to its input data. This process is called *forward-propagation*.

In complex Multi-layer (or “deep”) neural networks, however, weight adjustments can only be done when the information has passed through all the layers of hidden neurons, since only then the calculated output can be compared with the expected output. The process of forward-propagation therefore is complemented with a second process called *back-propagation*, in the course of which the weights are adjusted “from back to front”. This process starts from the connections of the output neurons to the neurons of the last hidden layer and “distributes” the error backwards through the network to all connection weights of all layers. In some details explained: if the generated output as rounded on the threshold is called the activation  $A$  of a neuron, then the error of an output neuron  $E_o$  is determined by multiplying the activation with its complement and with the difference of the activation and the expected output  $O$ , that is:  $E_o = (O - A) * A * (1 - A)$ . Analogously, the error  $E_v$  of a hidden neuron is determined by multiplying the sum of the products of the errors of the preceding neurons  $E_o$  (or  $E_v$ ) with the weights of the connections leading to them and with the activation and its complement:  $E_v = \sum_{j=1}^n (E_{oj} * W_j) * A_j * (1 - A_j)$ . The corresponding weights  $w_j$  then are adjusted in respect to the learning rate  $d$  multiplied with the error of the preceding neuron  $E_o$  (or  $E_v$ ) and the activation of the neuron to which the connection leads back:  $w_{i(t+1)} = w_{i(t)} + d * E * A$ .

A. If all goes well, the network, by iterating these adjustments, approaches a state that captures the regularities of the data to be learned.

In the single-layer Perceptron example, we used a simple threshold function for rounding the output values. In more complex data landscapes however, more complex functions are used to determine the activation of a neuron. Quite common for example is the deployment of a so-called sigmoid function<sup>7</sup>, which “forces” the input values into the interval between 0 and 1 according to their proximity to the threshold value. Usually, also so-called bias-neurons are added to the layers of an ANN. Since, as in the example above, relevant inputs to the network may consist of zeros at times and, multiplied by the connection weights, would then result in zero information for the network, these bias neurons guarantee to generate constant “neutral inputs” in dependence of the weights of their connections.

It should be mentioned at this point that multi-layer ANNs exhibit remarkable characteristics apart from their ability to learn complex matters. Interesting for instance is the fact that two trained ANNs can perform equally well on a problem, but still show completely different internal structures. The actual weighing of their connections may differ at large but still solve the same task. This may suggest the assumption that the internal structure of human brains also differs even though they accomplish same tasks with comparable quality. Another interesting feature is the ability of artificial neural networks to learn incrementally *in actu*, that is, while being used. Unlike hard-wired circuits, they are never fixed for good in their function. They adapt permanently to new inputs and thus can be deployed under very dynamic conditions, as for example in the case of autonomous driving on public streets. This circumstance is responsible for the fact that neural networks function relatively fail-safe. While in hard-wired circuits the failure of a single, often even irrelevant component can lead to total failure of the entire system, neural networks are surprisingly resistant to perturbations or partial losses. Probably the most obvious examples are stroke patients with severe brain damages, who sometimes, after appropriate therapy, can find their way back to behaviors that can be seen as relatively “normal” in view of their injuries. In these cases, the non-destroyed parts of their brains are able to relearn the lost functions and thus compensate for the damage. This characteristic of networks is called *graceful degradation* and often is implemented intentionally in technical systems where individual partial failures should not result in total failure, as for example in the supply of electricity in power grids or of information in the internet. An accepted redundancy of connections makes

---

<sup>7</sup> For example, of the form  $s = \frac{1}{1 + e^{-x}}$



the network resilient. If a part of it fails, other connections take over, often without the consumer noticing.

### 4.3 Gradient Descent

As mentioned, the difference between generated and expected output is called the error of the classifier model. This error can be expressed in terms of a cost function. Minimizing a cost function (or an error) is often done with an algorithm called gradient descent, which is a very generic optimization method that can be usefully applied to every-day problems as well. Imagine for instance you are on top of a grass mountain and suddenly fog is coming up. You can't see anything, but you want to get back down to the valley quickly and safely. What are your options? At first, this seems easy. You simply follow the path that goes down the steepest from where you are until the ground becomes flat. If the grass mountain has a simple topology, the flattening subsoil will announce that you are approaching the valley.

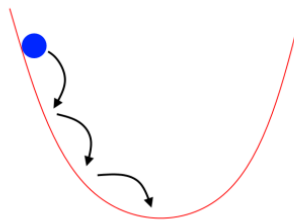


Fig. 50: Descending the steepest path until reaching flat ground (own illustration).

However, your stride length could be an important parameter in this. If you take very small steps, it will take you quite a long time to come down to the valley. On the other hand, if you take too large steps (– imagine much larger steps than you could do in reality –) you may overshoot the deepest point and climb up the other side of the valley again.

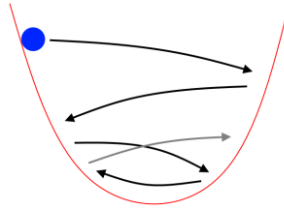


Fig. 51: Overshooting the deepest point of a valley (own illustration).

If the mountain additionally has a complex structure, with pre-summits, local valleys and narrow descents, too large steps could make you miss a narrow descent that leads to the global valley and lets you end up on a suboptimal plateau, a *local minimum*.

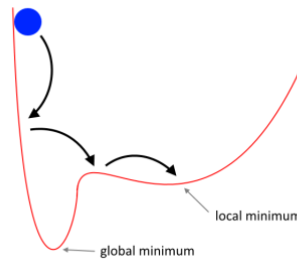


Fig. 52: Risk of settling on a suboptimal plateau, aka *local minimum* (own illustration).

An obvious method to prevent this could therefore be to adjust your stride length to the steepness of the terrain, for example, to reduce the stride length when the terrain becomes flatter, or to consider an additional *momentum* in order to keep you going for some time even so the path, after reaching a local minimum, leads uphill again. The momentum then could help you overcome local valleys and lead you to the global minimum.

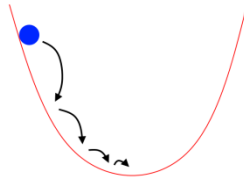


Fig. 53: Adapting stride length to terrain steepness (own illustration).

Mathematically going downhill in this way means following the negative gradient of the cost (or error) function for each dimension. The gradient, mathematically denoted with the *Nabla*-operator  $\nabla$ , is a vector pointing at the steepest ascent of a position in the cost-landscape and indicating the slope at this point with its length. The slope is found as the partial derivative of the cost-function for each dimension. This is like asking, what is the slope of the hill under my feet if I face East? And then asking the same question for North, South and all other dimensions. The gradient vector points uphill. Following the negative gradient thus defines an iterative procedure  $a_{n+1} = a_n - \gamma \nabla f(a_n)$ , with  $a$  being a model parameter that is slightly changed in each step, and  $\gamma$  expressing the stride length (or error-decreasing-rate, respectively learning rate). If the stride length is adjustable,  $\gamma$  is changed in each iteration, with a multitude of specialized methods being available to govern this change. In practice, gradient descent is often started from random values (called random initialization) and iterated until the cost (or error) is reduced to a level, which may be not the absolute (mathematical) minimum, but which is small enough to be acceptable.

If the cost function is convex and continuous, as is mostly the case with MLPs, gradient descent should converge to the global minimum. A factor that could impede this is when the features in your data set have different scales, like in the case of “Account balance” and “Age” in the toy example above. In such cases, convergence could be misdirected at first and thus take a long time to find the minimum. To prevent this, data should be scaled before training an ANN on it.

While learning, gradient descent can be calculated in each iteration for each feature in each instance of the training data. This is called *batch gradient descent* and can be computationally very expensive. A cheaper variant therefore is suggested under the name *stochastic gradient descent*, which picks a random instance in the training set at every step and computes the gradients based only on this instance. This makes learning much faster but a bit erratic in respect to finding the global minimum. Its

final parameter values may be good, but not optimal. On the other hand, the erratic bouncing of the stochastic gradient descent can help the procedure to leave local minima and eventually find the global ones, if, for example by decreasing the learning rate (the stride length), the to-and-fro-bouncing of the descent can be controlled. Starting out with long steps and gradually decreasing the step size for allowing the algorithm to settle to the local minimum is called *simulated annealing*. It resembles the process of annealing in metallurgy where molten metal is slowly cooled down.<sup>8</sup>

#### 4.4 The MNIST-digits-example

To further illustrate the working principles of artificial neural networks let's take a brief look at the task of teaching an ANN to read and distinguish handwritten digits from "0" to "9". This is a common introductory task in machine learning, which usually is done with the so called MNIST database of handwritten digits (see <http://yann.lecun.com/exdb/mnist/>). The database contains a set of 50.000 digital images for training and a separate 10.000 image-set for validation purposes. Six examples from this data set are shown in figure 54. The pixelization is intended to illustrate that each pixel is an input for a neuron.

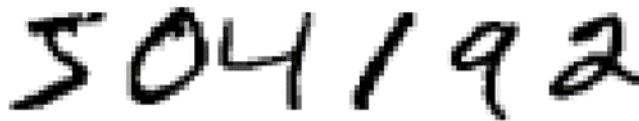


Fig. 54: Six examples of handwritten digits (pixelization intended), taken from <http://yann.lecun.com/exdb/mnist/>.

As can be seen, the scanned hand-writings have a rather low resolution of  $28 \times 28 = 784$  pixels, which in most cases suffices to clearly distinguish the digits. In some cases, however, – see examples below – even humans will have difficulties to distinguish the digits.

---

<sup>8</sup> Besides batch and stochastic gradient descent, where either all or just one instance are chosen for gradient calculation per iteration, there is also a method called *mini-batch gradient descent*, which allows defining the size of random sets of instances used for calculations.



Fig. 55: Examples of unclear digits (pixelization intended), taken from <http://yann.lecun.com/exdb/mnist/>.

The MNIST-digit images are black and white only. The information in them can be transmitted according to the principle shown in the stylized matrix below. Pixels that are not affected by the hand-writing are considered zero-entries in the matrix, and pixels that are touched by the *A* are considered one-entries. Respectively, if there is color in an image, entries in the matrix would consist of corresponding values. The information in this matrix then is provided to the input-layer of the ANN, which in this case would need to have 784 input neurons.

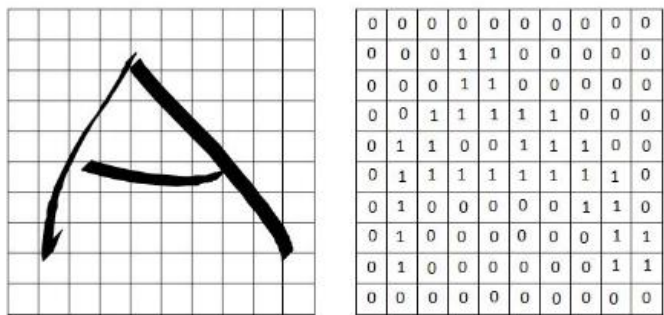


Fig 56: Handwriting and corresponding pixel matrix (own illustration).

The ten digits from “0” to “9” are uniformly distributed among the 60.000 instances in the dataset so that each “5” for example could be applied approximately 6000 times in different variations to the neural network. To resolve this variety, an ANN with more than just one layer of hidden neurons is needed. The Multi-layer Perceptron (MLP) as provided by *scikit-learn* can be trained on the data using the kind of coding, which we have seen in the applications before. The code snippet below imports the data (from a zipped file), prepares training and test data sets, defines parameters for the MLP (for details see [sklearn.neural\\_network.MLPClassifier.html](#)),

trains it (via the *fit*-method), plots a curve that shows the iterative loss of the error, tests the trained MLP on the test data and prints a classification report and a confusion matrix.

```
import gzip
import cPickle
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
%matplotlib inline

# import data
f = gzip.open('<path_to_data>\mnist.pkl.gz', 'rb')
trainings_data, validation_data, test_data = cPickle.load(f)
f.close()

# separate data from target value
X, y = trainings_data[0], trainings_data[1]

# define parameters for Multi-layer Perceptron (mlp)
param = {'solver': 'sgd', 'learning_rate': 'constant', 'momentum':
0, 'learning_rate_init': 0.2}
label = "constant learning-rate"

mlp = MLPClassifier(verbose=0, random_state=0, max_iter = 150,
```

Fig. 57: Python code for MLP training and evaluation.

Error loss curve, classification report and confusion matrix are shown in figure 58. As can be seen, the results are not bad. Precision, recall and f1-score are at 98%, which is quite impressive given that some of the digits are hard to read even for human interpreters. The main diagonal of the confusion matrix holds the correctly classified instances and we can see from the off-diagonal entries that only very few cases were miss-classified.

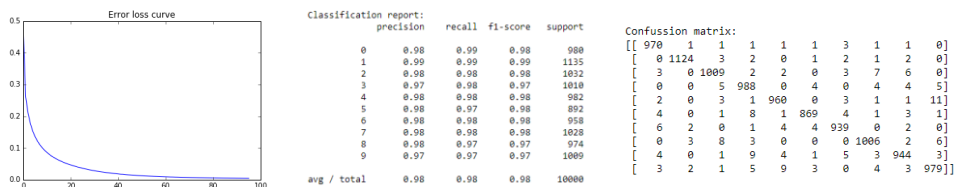


Fig. 58: MLP-error loss curve, classification report, and confusion matrix (own illustration).

For comparison, classification report and confusion matrix of the results of a Support Vector Classifier trained on the same data set are shown below. In this case, the Support Vector Machine took a factor of 20 longer to learn and to be tested, and the results are by far not as impressive as the ones of the MLP.

Classification report:					Confussion matrix:										
	precision	recall	f1-score	support											
0	0.96	0.99	0.97	980	[[ 967  0  1  0  0  5  4  1  2  0]										
1	0.97	0.99	0.98	1135	[  0 1120  2  3  0  1  3  1  5  0]										
2	0.94	0.93	0.94	1032	[  9  1 962  7 10  1 13 11 16  2]										
3	0.93	0.94	0.93	1010	[  1  1 14 950  1 17  1 10 11  4]										
4	0.93	0.95	0.94	982	[  1  1  7  0 937  0  7  2  2 25]										
5	0.93	0.91	0.92	892	[  7  4  5 33  7 808 11  2 10  5]										
6	0.95	0.96	0.96	958	[ 10  3  4  1  5 10 924  0  1  0]										
7	0.95	0.93	0.94	1028	[  2 13 22  5  7  1  0 954  4 20]										
8	0.94	0.91	0.93	974	[  4  6  6 14  8 24 10  8 891  3]										
9	0.94	0.91	0.93	1009	[ 10  6  0 12 33  5  1 14  6 922]]										
avg / total	0.94	0.94	0.94	10000											

Fig. 59: SVM-classification report and confusion matrix (own illustration).

A good place to delve deeper into how artificial neural networks work, and especially to try out the effect of different neuron constellations, is the “Neural Network Playground” on Google’s platform *tensorflow* at <https://playground.tensorflow.org/>.

## 4.5 Neural Network types

The efficiency of multi-layer neural networks has led to a hype about the possibilities of *deep learning* in recent years. Since about 2012 in particular, new types of ANNs have been proposed on an almost monthly basis, which have skyrocketed in their efficiency in many regards. It seems that deep learning is bringing the long-standing dream of artificial intelligence within reach. The following provides a brief, condensed overview of some of these ANN-types specialized on various tasks. More detailed information and good introductions can easily be found online, for example on the tutorial site of *tensorflow*: <https://www.tensorflow.org/tutorials>

### 4.5.1 Long Short Term Memory (LSTM) Neural Networks

One issue of using gradient descent for back propagation in multi-layer Neural Networks is the so-called *vanishing or exploding gradient problem*. This problem arises from the process of minimizing the difference of generated and expected output by determining the error signal through the derivative of the activation function (see section 4.2. and 4.3.). In this derivation, the slope and direction of the descent is defined by multiplying the scaling factor with the error term. If however, scaling factor and error are both less than 1 already, their multiplication makes the signal vanish when being propagated through several layers. As a consequence, the error does not propagate back far enough to the layers close to the input layer. These layers are not appropriately trained. On the other hand, if scaling factor and error are both larger than 1, their multiplication can make the error term explode while being propagated backwards. Both possibilities prove suboptimal for the training of multi-layer neural networks.

To account for this problem, two pioneers of neural network research, Sepp Hochreiter and Jürgen Schmidhuber, proposed a network architecture that guaranteed a constant and thus applicable error flow into the network layers (Hochreiter/Schmidhuber 1997). This architecture foresees additional components in the neural network, a so-called “cell” and three “gates”, which regulate the flow of information from layer to layer. The “cell” is made to remember values over arbitrary time intervals and the three “gates” – called “input”, “output” and “forget gate” – regulate the flow of information into and out of the cell (see figure 61). An effect of these additional components is that such Long-Short-Term-Memory Neural Networks (LSTMs) are able to consider a sort of *context knowledge* when being trained on new data. They do not classify by accounting for each instance anew, but can consider experiences from earlier phases in their training, just as humans understand words based on their understanding of previous words, or as they understand scenes in a movie from earlier scenes in the same movie.

With this ability, LSTMs turned out to be great learners in the context of long-term patterns in time series, audio recordings or texts, and an important precursor of the transformer technology. With regard to text processing, an experiment conducted at Stanford University in 2015 gives an impressive example of the learning power of LSTMs. Andrej Karpathy and his colleagues (2015) trained an LSTM on various texts, among them the famous novel “War and Peace” by Leo Tolstoj consisting of 3.258.246 characters of English text (see: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>). As a direct predecessor of the Generative Pre-trained Transformers



(GPTs, see chapter 6), the LSTM simply was made to model the probability distribution of the next character in a sequence (e.g., a word or a sentence) given a sequence of previous characters. In this way, it learned to generate new text from existing one. Exemplified on a very simple example, the task hence was to derive the probability of the four possible letters “h”, “e”, “l” and “o” by being trained on the sequence “hello”, or in more details, the probability of “e” given the context of “h”, the probability of “l” given the context of “he”, the probability of another “l” given the context of “hel”, and finally the probability “o” given the context of “hell”. In order to demonstrate its learning progress, the LSTM, while being trained on the entire corpus of the “War and Peace”-text, was made to generate samples every 100 iterations of training. Some exemplary output is shown below:

After 100 iterations of training, the LSTM produced texts like the following:

tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne  
'nhthnee e plia tkllrgd t o idoe ns,smtt h ne etie h,hregtrs niglike,aoaenns  
lng

After 300 iterations:

“Tmont thithey” fomesscerliund Keushey. Thom here sheulke, anmerenith  
ol sivh I lalterthend Bleipile shuwly fil on aseterlome coaniogennc Phe lism  
thond hon at. MeiDimorotion in ther thize.”

500 iterations

we counter. He stutn co des. His stanted out one ofler that concossions and  
was to gearang reay Jotrets and with fre colt off paitt thin wall. Which das  
stimn

700 iterations

Aftair fall unsuch that the hall for Prince Velzonski’s that me of her hearly,  
and behs to so arwage fiving were to it beloge, pavu say falling misfort how,  
and Gogition is so overelical and offer.

1200 iterations

“Kite vouch!” he repeated by her door. “But I would be done and quarts,  
feeling, then, son is people....”

2000 iterations

“Why do what that day,” replied Natasha, and wishing to himself the fact  
the princess, Princess Mary was easier, fed in had oftended him. Pierre aking  
his soul came to the packs and drove up his father-in-law women.

## 4.5.2 Recurrent Neural Networks

LSTMs are instances of so called Recurrent Neural Networks (RNNs), which have the nice feature of being open to input and output sequences of arbitrary length, opposite to the Perceptron-like networks we encountered so far, which need to be fixed in regard to input and output. With this, RNNs are widely used for transcribing-speech-to-text tasks, for machine translation, for generating handwritten texts, for predicting stock market prices, next notes in melodies (<https://magenta.tensorflow.org/>)<sup>9</sup> or the trajectories of oncoming cars in autonomous driving. They are also applied in *computer vision*, as for example in video classification or *image captioning*, which consists of the task to describe a classified image with words (see the examples below, taken from Vinyals et al. 2014).

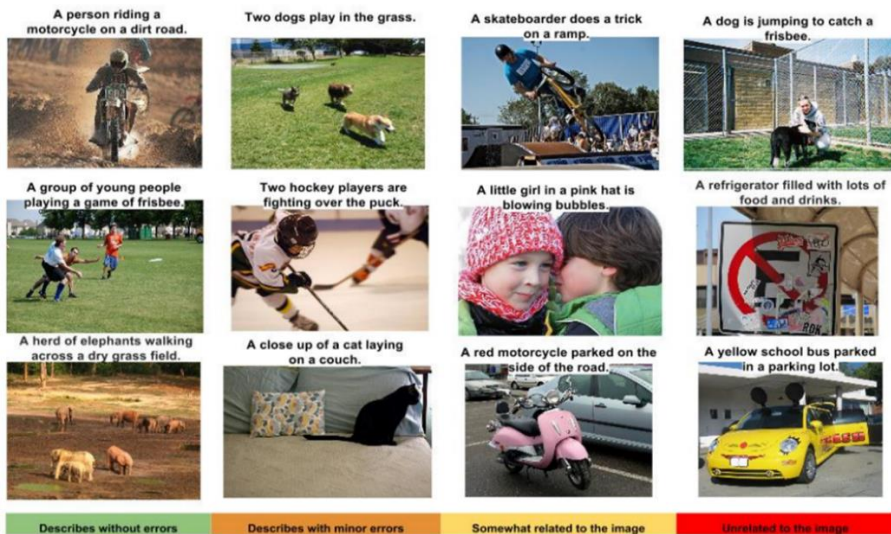


Fig. 60: Examples of image captioning, taken from Vinyals et al. (2014).

The most striking feature of Recurrent Neural Networks are loops in their structure that allow information to persist. They are designed to recognize patterns in sequences of data, such as text, genomes, handwriting, spoken words or numerical time series data emanating from sensors or other kinds of monitoring. In general,

<sup>9</sup> See [https://cdn2.vox-cdn.com/uploads/chorus\\_asset/file/6577761/Google\\_-\\_Magenta\\_music\\_sample.0.mp3](https://cdn2.vox-cdn.com/uploads/chorus_asset/file/6577761/Google_-_Magenta_music_sample.0.mp3)

they can be applied whenever it is necessary to take time and sequence into account. In other words, RNNs have a temporal dimension. They use a kind of memory to present information from previous learning steps to current ones *as an addition*, thereby being able to find correlations between events separated by time. These correlations are called “long-term dependencies” causing RNNs to share once found weights over time.

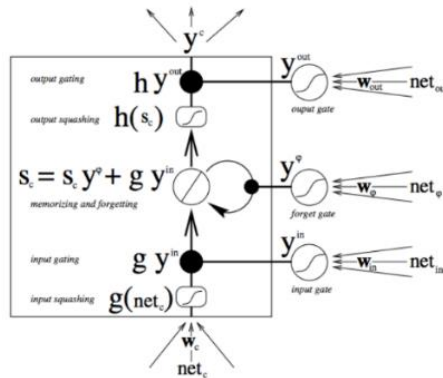


Fig. 61: Backpropagation through time, taken from Mai (2019).

Remember that backpropagation in feedforward networks moves backward from the final error through the outputs, weights and inputs of each hidden layer, assigning those weights responsibility for a portion of the error by calculating their partial derivatives. These derivatives are then used by the learning rule (e.g. stochastic gradient descent) to adjust the weights up- or downwards to reduce the error. In contrast to this, RNNs rely on an extension of backpropagation called backpropagation through time (BPTT). Time, in this case, is expressed by an ordered series of calculations linking one time step to the next. This simply extends the series of nested composite functions (e.g.  $(g(h(x)))$ ) that all neural networks consist of by an additional time component.

### 4.5.3 Convolutional Neural Networks (CNNs)

A far-reaching area for the application of ANNs is pattern recognition as it is used in image classification or image capturing. A special class of so called Deep Neural Networks (DNNs), called Convolutional Neural Networks (CNN), recently is being

applied to these tasks with great success. The connectivity patterns of these networks too are inspired by biological processes. They resemble the organization of the visual cortex in animals, with certain highly interconnected layers of neurons specialized on dissolving particular structures like for instance horizontal or vertical lines or rounded objects of particular colors etc. Their working principle, which is a sort of division of perception labor, is derived from infamous experiments on young animals, which during infancy were exposed to reduced environments of for example only horizontal structures and then, in their adult life, never regained the possibility to perceive vertical structures. From this it was concluded that the visual cortex consists of various areas able to specialize differently and thus is in need of appropriate training. Obviously, once trained, these parts respond primarily to certain structures, while other parts may specialize on different ones, with these receptive fields overlapping partially so that an entire visual field can be covered.

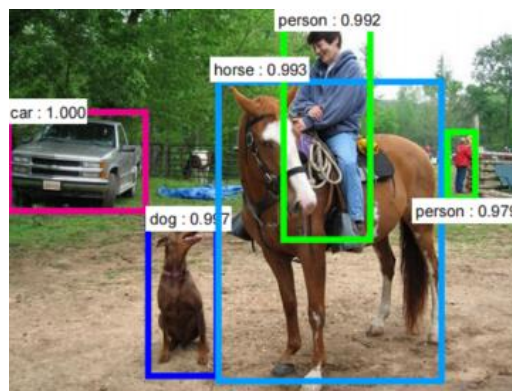


Fig. 62: Pattern recognition as used in image capturing, taken from Ibrahim, Saab, and Sbeity (2019).

Within the so-called convolutional layers of these networks, the information of, for example, pixels in an image gets compressed in a process that samples the data with rolling windows of certain sizes. To do this, neurons in a convolutional layer are not connected to all neurons in the next layer (as in the MLP), but are only connected to those neurons located within a small rectangle in this layer (figure 63). The rectangles then are rolled with a certain *stride* length  $s$  over the information of the lower layer.

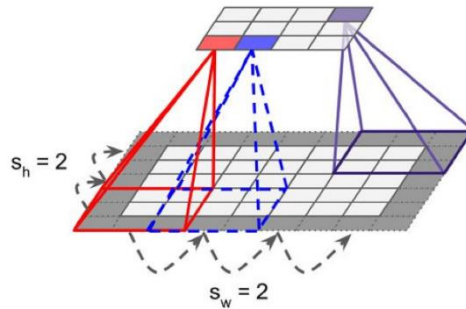


Fig. 63: Neurons in a convolutional layer, taken from Géron (2017), p. 364.

In this way, the network can focus on particular features, horizontal structures for instance, and assemble their information into higher level features in the next layer, and so on. The adjusted network connections, which contain the information of these features, thus work as *filters*, which ignore everything in their receptive field except for the structures on which they were trained. Figure 64 shows the result of the interaction of two such filters (taken from Géron 2017).

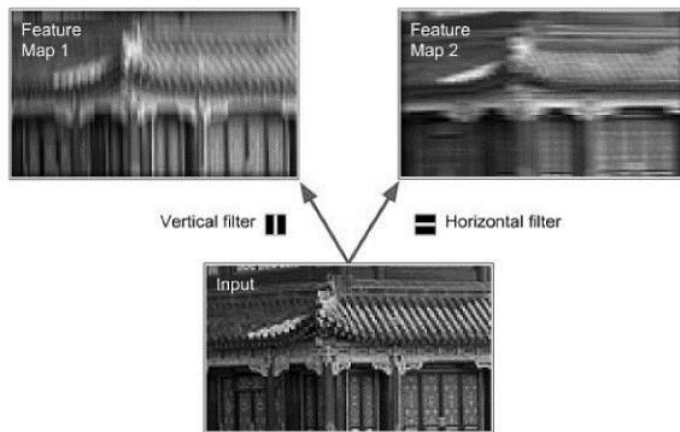


Fig. 64: Applying two CNN-filters, taken from Géron (2017), p. 365.

Convolutional layers can be composed of several such feature maps, which themselves are stacked together to rich layers of interconnected neurons. Usually, network parameters are kept the same within a stack but can vary over different stacks. Stacked convolutional layers than are interconnected to other layers with so-called

*pooling layers*, which serve the task to subsample (or shrink) the input information in order to reduce the computational load, thereby usually also reducing the risk of overfitting. The pooling layers are connected in the same way as the convolutional layers, but their connections do not have weights in the usual sense, but aggregate inputs using an aggregation function such as the *max* or the *mean*. In this way, only the *max* (respectively *mean*) input values of these layers make it to the next layer, which obviously can reduce information dramatically. Processed images thus get smaller and smaller by passing several pooling layers, but they also get deeper and deeper by passing different convolutional layers, implying greater depth of focus for different features. In this way, convolutional and pooling layers thus are actually just *extracting* the features, for instance of an image. The final classification is done by a conventional fully connected multi-layer perceptron that is added to the end of the stack for assembling the actual predictions of the CNN (see the scheme below).

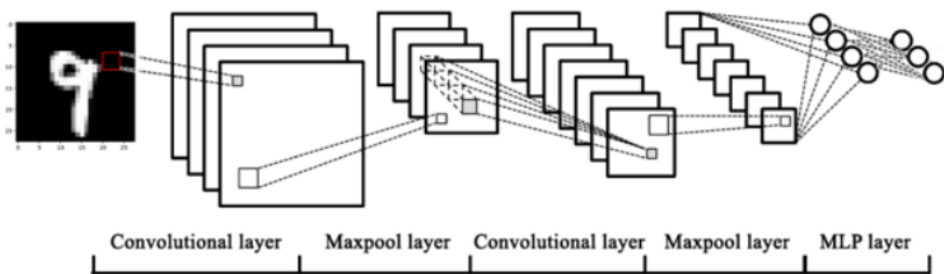


Fig. 65: CNN with fully connected MLP added to the end, taken from Géron (2017), p. 371.

The simultaneous application of multiple filters together with pooling in CNNs can produce impressive results of image capturing, implying rapid progress in machine-based pattern recognition and learning tasks. This progress in designing more and more powerful architectures of CNNs can be followed on the example of the drop of the error rate in competitions such as the ILSVRC-ImageNet Challenge (<http://image-net.org/challenges/LSVRC/>), where year to year new milestones in image recognition are reported.

## 4.6 Unsupervised Learners

What we discussed so far have mainly been *supervised* learning algorithms, which base on the idea of learning on a labeled data set containing a target-value to which the output-values generated by an ML-tool can be compared while training. The difference between generated and expected output (i.e., the target), can then be used to adjust the internal structure (i.e., the weights) of the tool. Labeled data is not always at hand though, or it needs to be produced at great expense, which often is done by cheap labor in third world countries. Besides *supervised* learning however, there is also a range of *unsupervised* methods, which are able to find structures in data by themselves, without being fitted to target values. Often these methods are used for clustering and compression of data or for dimensionality reduction tasks.

### 4.6.1 Self-organizing Maps (SOMs, aka Kohonen-networks)

An instance of an unsupervised learning artificial neural network is the so-called Self-Organizing Map (SOM) or, as named after its inventor Teuvo Kohonen, the Kohonen network. This ANN belongs to the class of *unsupervised* learners, implying that they are not trained in respect to any expected output, but adapt themselves to regularities in the data landscape. Unlike backpropagation networks, SOMs are not built on the distinction of input, hidden and output neurons. Instead, they can be seen as an adaptive network structure that clings to input data by stepwise rapprochement of network areas around certain neurons. The details of this process can be illustrated on the example of the yellow net as shown in the images below.

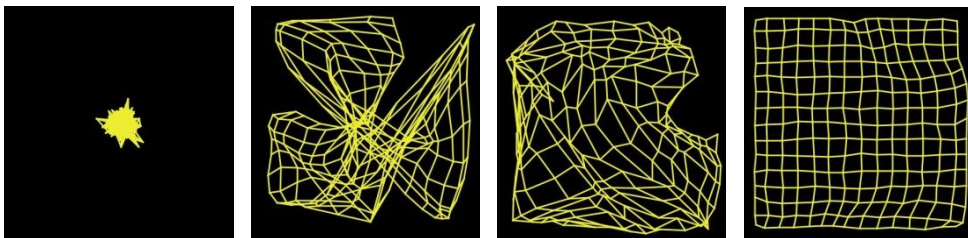


Fig. 66: SOM training in four stages (own illustration).

The input for this SOM consists of the two-dimensional coordinates of grid points on the black background square. At the beginning of training, the yellow net shown in the figures is crumpled together (left image). All connection weights are randomly set to a small value so that the net is folded around the zero-zero-coordinates in the center of the background square. The  $xy$ -coordinates of a randomly selected grid point of the black background square then are used as a first input signal and compared with the coordinates of the nodes of the yellow network. These nodes represent the neurons of the network. The node with the smallest Euclidean distance to the input date is considered the *Best matching unit* (BMU) for the first calculation step, implying that the BMU is seen as the current center of an approximation process, in the course of which the connection weights of the neurons within a certain radius (i.e., in this case the coordinates of the nodes of the yellow net) are adjusted with a learning rate to bring them closer to the input. In this process, the larger the distance of the neurons from the BMU, the smaller will be the intensity of the rapprochement. As a consequence, the weights of the neurons directly connected to the BMU are changed more and the weights of the neurons further away are changed less intensely. The reduction (here called *redu*) of the change impact follows the formula

$$redu = \exp\left(\frac{-d^2}{2l}\right),$$

where  $d$  is the distance of a neuron to the BMU within the radius and  $l$  is the learning rate.

After each learning step, a consecutive input date, here the coordinates of another grid point, is randomly chosen and subjected to the same calculation. This is repeated for all input data and then iterated with a slightly lower learning rate and a slightly reduced radius. The reduction of radius and learning rate follow the formula

$$redu2 = i * \exp\left(\frac{-t}{\left(\frac{n}{\log r^2}\right)}\right),$$

where  $i$  denotes the initial radius or initial learning rate,  $t$  the number of learning steps already performed,  $n$  a value of 1000, which is reduced by a certain amount after each calculation round, and  $r$  is the radius as it currently applies.

Note that the parameters in this type of network are not absolute values, but are themselves subject to subsequent adjustment processes that depend on the type of data to be explored. In some cases, for example, higher learning rates can pay off because the network learns faster. In other cases, similar to the mentioned problems in the gradient descent, they may counteract the convergence of the network,



because the weights' oscillations are too large to allow for a suitable adaptation. Unfortunately, there are hardly any general rules for fine-tuning such networks. However, they often manage to self-adjust to very complex data landscapes effectively and surprisingly quick. As in the example of the folded network, SOMs are often used for dimensionality reduction and similar compression tasks. They are also able to solve combinatorial problems that can hardly be calculated in a conventional way, such as the *Travelling Salesman Problem*, in which an agent has to find a Hamiltonian cycle, that is, a shortest path through a number of cities (white dots in the following images) without visiting one of the cities twice. This task is known to be NP-hard, implying, briefly said, that computational brute-force searches can take astronomical times to come up with results. The images below show the SOM in four different iterations of its training. As with similar learners, the solutions found are never absolute optimal, but generally represent sufficiently good approximations.

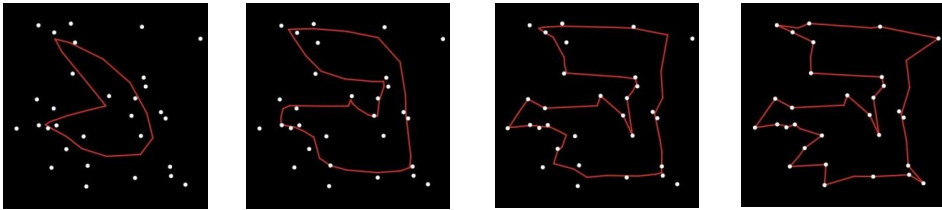


Fig. 67: Travelling Salesman: Hamiltonian cycle through multiple cities (own illustration).

#### 4.6.2 Autoencoders

Another interesting kind of *unsupervised* learning algorithms are so called Autoencoders. These ANNs are capable of learning efficient representations of input data, called *codings*. Typically, codings have much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction and for the detection of relevant features. With this property, autoencoders can be used for unsupervised pre-training of deep neural networks. Additionally, they can be used for generating *synthetic data*, that is, data that is statistically very similar to a set of training data for example if this data should be subject to privacy issues.

Surprisingly, the working principle of Autoencoders is quite simple. Basically, they learn by just copying input data to the output neurons. In this, they are severely constrained by the network structure, so that the network is prevented from trivially copying input to output. Constraints can consist of limitations to the size of the internal representations of the network (see figure 68), or of an addition of noise to the input data and the task to recover the original input. These restrictions force the network to learn efficient ways to represent the data after all. The codings then are byproducts of the Autoencoder's attempt to learn to reproduce data *under constraints*.

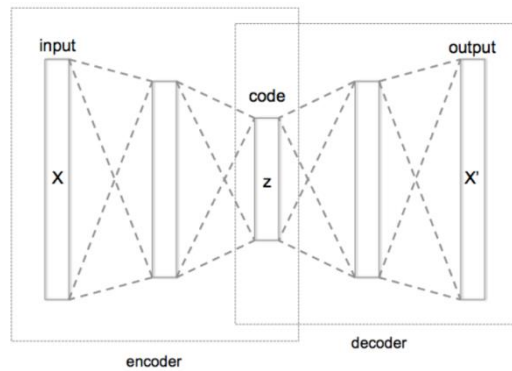


Fig. 68: Autoencoder, reducing input data dimensionality (own illustration).

A simple explanation why Autoencoders are good learners draws on a similarity to human needs for learning. If we would be good in memorizing long sequences, for example the first 1000 digits of the Fibonacci-Sequence, it would maybe not be necessary to think about a rule that generates this sequence. Since our memory capacities are limited however, it makes sense to look for general rules. The same applies to Autoencoders. They are efficient in pattern recognition because they are constrained in representing the Whole directly as such. Their restriction pushes them to try to discover and exploit patterns – similar to trained chess players who are able to remember all positions on a game board by looking at it just for 5 seconds when the positions are part of an actual game. When these positions are randomly assigned however (i.e., not part of a real game), implying that there are no patterns resulting from rules, their memories fail just as the ones of average people.

Autoencoders are always composed of two parts, an *encoder part* converting the information to an internal representation, and a *decoder part* that converts the internal representation to the output (see figure 68). The decoder is sometimes also called *reconstruction*, with the loss function having a *reconstruction loss* that penalizes the model when its' reconstructions are different from the inputs. The output layer always has the same number of neurons as the input layer. The hidden layers though are of lower dimension, which is sometimes called *undercomplete*, thus forcing the model to learn the most important features in the input data and ignore the unimportant ones. Figure 69 shows an example result of applying an undercomplete autoencoder to the digits of the MNIST-dataset, with leaving just as much information as needed for classifying the digits.



Fig. 69: Applying undercomplete autoencoder to MNIST-digits (pixelation intended, own illustration).

Autoencoders can have multiple hidden layers, making the ANN a *stacked* (or *deep*) autoencoder, usually with a symmetric structure. With additional layers the learning capacity of the ANN increases. But too many hidden layers can make the autoencoder perform similar to regular ANNs, meaning that in consequence – since it has the same number of outputs and inputs and no target values to compare its learning progress to – it just learns to reconstruct the training data perfectly, thus *overfitting* and not being able to generalize to new data. The fine tuning of an autoencoder therefore often consists of finding the right number of layers.

When training the autoencoder, it is often useful to train different parts of it (or actually, different simpler autoencoders) separately and then stack the pre-trained parts together. This can be done by simply copying the adjusted weights and biases to a new instance of the stacked ANN. Also, it is quite common to use autoencoders for the pre-training of ANNs on unlabeled data, which often is easy to obtain. Just download a couple of thousand pictures containing for example animals from the internet. To label them, i.e., to determine a target value in identifying the animals and putting the name into a separate column, can be time-consuming and costly. In such cases, the dataset can be pre-classified with an Autoencoder, which groups

the pictures into separate (unnamed) categories in respect to the similarities it recognizes in them. The weights of the layers that are responsible for these classifications can then be transferred to another ANN where this (often frozen) pre-trained information is used as a sort of target value for learning to further classify the pictures.

One interesting variant of the autoencoder is the *Variational Autoencoder*. As mentioned earlier, this type of neural network can be deployed to generate *synthetic data*, for example in cases where the real data cannot be used freely for privacy reasons, such as sensible health data for instance. Instead of directly producing a coding for a given input, the Variational Autoencoder generates several of them and takes their mean and their standard deviation. This *mean coding* then is applied to the decoder as usual to generate an output which is statistically very similar to the input, but still not identical. The images below show synthetic “hand-written” digits that were artificially generated in this way. Some of them could perhaps pass as indeed handwritten by humans.

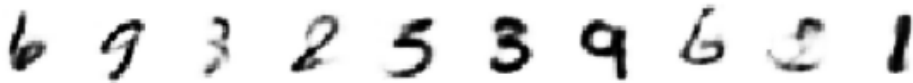


Fig. 70: Synthetic “hand-written” digits, taken from Géron (2017), p. 431.

In 2017, a very powerful type of ANNs called Transformer Neural Networks has been introduced and consequently has taken ML-methods to a new level of efficiency. Transformer applications range from Natural Language Processing (NLP) as in the case of GPTs via artificial music generation up to possibilities for increasing the safety of autonomous driving. Since these ANNs emerged in the course of NLP-research and base on some of its core concepts, we will first discuss some of the prerequisites of this technique. To some extent, some of the methods explained below have become obsolete in the course of these developments. Or differently put, the possibilities offered by transformers provide more efficient solutions for some of the problems. However, to fully understand how these types of neural networks work and what they can do, it is still extremely helpful to know and understand the underlying problems and their previous attempts to solve them. Only with this, the implications of the current methods can be grasped with sufficient critical awareness.

# 5 Natural Language Processing

Natural Language Processing (NLP) includes a multitude of methods and techniques which circle around the tasks of understanding and generating human-understandable texts with digital machines. One of its core methods that offers a starting point for gaining insights into its methods is text mining.

Text mining – sometimes also known as text analytics – refers to the (often automated) process of deriving structured information and meaningful numeric indices from (mostly) unstructured textual information (often from the internet) in order to access and analyze this information with statistical and machine learning methods. Text mining involves several steps of natural language processing, including tokenizing, stop words exclusion, stemming, parsing, categorization, text clustering, word frequency analysis, part-of-speech identification, and many more.

The following examples from a text mining survey in a scientific journal (Schober et al. 2018) may give a brief overview on some of the possibilities associated with text mining.

In a first data retrieval step, the complete collection of publications of an open access journal for ecological conservation was downloaded from the internet. The gathered text corpus comprised 475 pdf-formatted scientific papers – the first one from Oct 2002 and the last one from July 2014, with a data volume of 345 MB. In this case, the papers were readily available in pdf-format. Very often however, web-scraping techniques have to be applied to retrieve such text corpuses, in particular if texts need to be read-out from html-pages and similar sources. We will not go into the subtleties of web-scraping here, but refer to the broad range of specialized software that is available online for this purpose. Useful Python-libraries include *scrapy* (<https://scrapy.org>), *urllib* (<https://docs.python.org/3/library/urllib.html>), *requests* (<https://docs.python-requests.org/en/latest/>), *BeautifulSoup* (<https://pypi.org/project/beautifulsoup4/>) and others.

In our case, for preprocessing the papers, a Python-script, essentially drawing on the *Natural Language Toolkit* – Python’s NLTK module ([www.nltk.org](http://www.nltk.org)) – was used to transform the publications into txt-files of about 22 MB. By extracting and dropping *stop words*, that is, short words like articles or propositions which contain little to no relevant information, and some special words that, like the journal’s name, repeat in each paper, the text corpus was further reduced. Finally, after converting the texts into NLTK’s specific text format, it contains 1.798.948 words.

With that, a first analytical step can be applied, which creates a frequency distribution of the 30 most frequent words in the journal (figure 71).

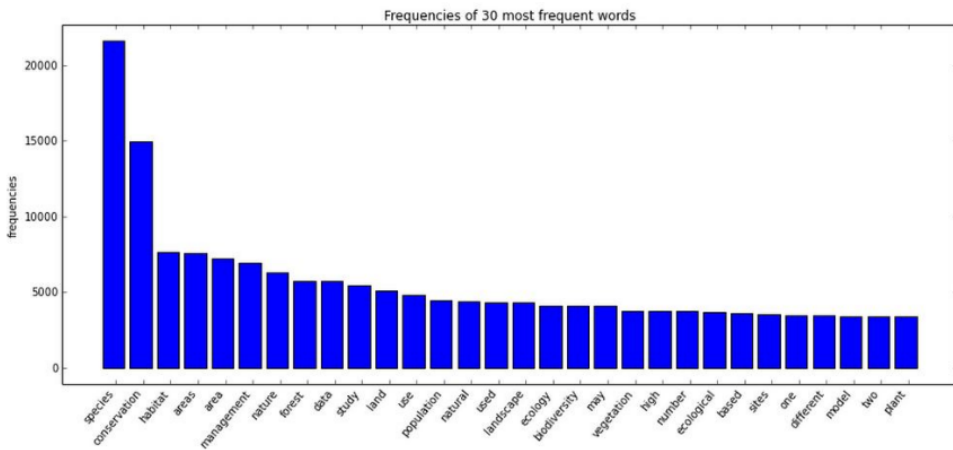


Fig. 71: Frequency distribution of 30 most frequent words (own illustration).

Frequency distributions are not confined to one-word frequencies. The plot in figure 72 shows a distribution of the most frequent two-word combinations.

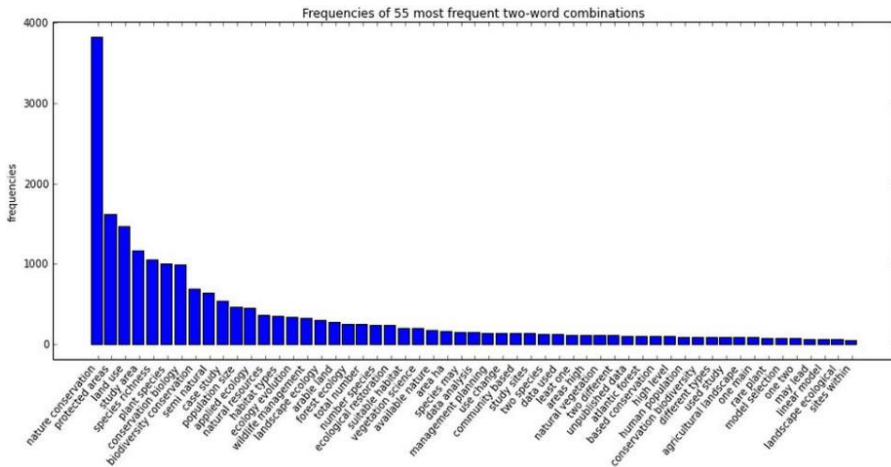


Fig. 72: Distribution of the most frequent two-word combinations (own illustration).

The analytical tools of the NLTK allow for a wide variety of interesting text explorations. An example provides the NLTK-function *concordance()* which allows to investigate the context a word appears in. The function generates the output below – in this case for the word “species”. (Have in mind that stop words were removed in the example).

```
Building index...
Displaying 25 of 21601 matches:
nal level natura focuses endangered species habitats therefore specific approach
as specific importance conservation species habitats listed fauna flora habitat
ke care positive development target species habitats member states responsible
dian value obtained study mammalian species ralls lethal equivalents could meas
t pandas respectively hu one bamboo species ye le guo year blossom cycle qing in
dependency sin gle food source one species bamboo utilized giant pandas ye le a
giant pandas ye le additional bamboo species transplanted area habitat loss may
may even greater importance bamboo species dependence threatening sur vival gi
iability assessment tool threatened species management conservation australian
standardise nature landscape amount species certain envi methodological standar
odological approaches many areas eg species habitat surveys tasks landscape pla
ce orientated sectoral eg different species animals plants planning orientated
region targets nature conserva tion species ecosystems processes types land use
ts animals even indicators ex ample species numbers number endangered species r
e species numbers number endangered species reference regional national red dat
scape visions objectives concerning species biotopes existing expected conflict
uthing obligatory fauna guild group species using resources respect food compar
tandard fauna use criteria rareness species like criterion endangering rareness
criterion evaluation used specific species characteristic land unit landscape
nservation evaluation gener el rare species despite exceptions need protection
e exceptions need protection common species effects endanger ing serious surviv
particular responsibility survival species exists population planning area end
olated relic population traced back species well genetic bio diversity biodiver
ted evaluation state ment area high species rich ness normative dimension evalu
development populations endan gered species recent years european centre nature
```

Fig. 73: Results of applying the NLTK-function *concordance()*.

The results of this step can be used to generate frequency distributions of the *contexts* a word appears in, in this case again the word “species”, considering only the words before and after the concordanced word.

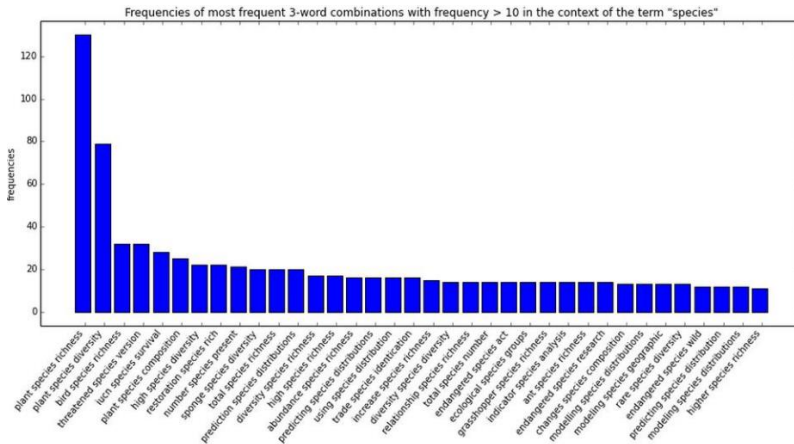


Fig. 74: Frequency distributions of the *contexts* a word appears in (own illustration).

Iterating through contexts and finding *contexts of context words*, allows generating rich networks of word-context usages, which can be visualized either directly with the help of the Python-module *networkx* (<https://networkx.github.io/>) (left image below), or with specialized graph-visualization tools like *Gephi* (<https://gephi.org/>) (right image below).

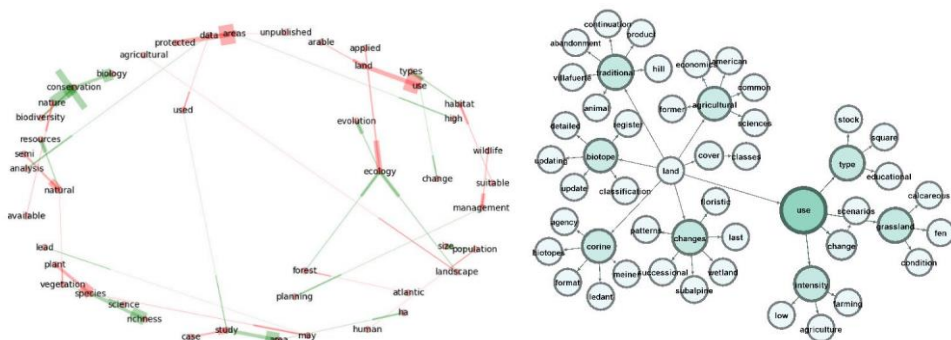


Fig. 75: Network-visualization of context words (own illustration).



## 5.1 Mining in unstructured and analog texts

A particular tricky feat is the extraction of computer-readable texts from unstructured or analog text sources. The image below shows the example of job advertisements being extracted from an old printed newspaper page stored in pdf-format (for more details see: Schober/Kittel/Füllsack 2016). With the help of the Python modules *pdfminer* (<https://github.com/euske/pdfminer>) and *opencv* (<http://opencv.org/>), which among others are able to separate text columns and apply *Optical Character Recognition* (OCR), parts of the texts can be readily extracted (see table on the right of in figure 76).

The image displays a scanned page from a 1950 newspaper, specifically the 'Kleine Anzeigen' (Small Ads) section. The page contains several job advertisements and notices. On the right side, a table lists extracted text from these ads, with red annotations highlighting OCR errors and corrections.

1 Offene Stellen
Melker ab 1. März 1950 (lesen) Viehstand Pferdeknecht Dauerposten, gute Verpflegung und Lohn. Franz Leitner, Volders bei Innsbruck. 2 Pferde Innsbruck.
Juengerer
Jazztrompeter mit Nebeneinstrument (Geige oder Akkordeon) sofort gesucht. Guter Nebenverdienst. Kabarett Kaerntnerhof, Graz. 7430
Fuer
Laudniaedchen oder Volksdeutsche, wird Haushalt, mit kleiner Landwirtschaft bei gutem Lohn aufgenommen. Graz-Sankt Peter, Peterstallstrasse 18. Fuer Gutsgearteter selbststaendiger Gaertner und Jungaertner gesucht. Schriftliche unter Oststeiermark 1760 Kleine Zeitung.
Bewerbungen
Fuer mittlere Landwirtschaft
Ein aufgenommen:
In der Nghe Vuglaets werden Landarbeiter und eine Landarbeiterin, Haushaltmitarbeiter, oder ein Landarbeiter-Ehepaar, Wohnung, Verpflegung und beste Bezahlung zugesichert. Zusdr an Kleine Zeitung, Villach.
die im (Gastwirtschaft)
auch
479a
50 bis 50 Schilling
Tagesverdienst
durch Besuch von Haushalten und Landwirten Vorstellen nur von 1013 und 1518 Uhr, am 3. Maerz in Spittal a.d. Dr., Bahnhofsgasthof
Eril Stuetze der Hausfrau mit fuer gepflegten Haushalt, per sofort gesucht. Vorstellen zwischen 12 u. 15 Uhr bei Kaspar, Graz, Glacisstrasse Nr. 67. 1758
Kodikenatissen
Zu vermieten

Fig. 76: Example of text extraction and OCR capabilities (own illustration).

Note that after extraction several words show incorrect spellings, due to the quality of the original newspaper printing. Many words (e.g. “Laudniaedchen”, which should read “Landmädchen”, i.e., “country girl”) cannot be identified by the OCR. While such inaccuracies hardly pose difficulties to human readers, computer programs need preparation for these cases. One particularly helpful method in this

context is the so-called *Levenshtein similarity* (or *L-ratio*, Levenshtein 1966), which also provides a good example of how computational power is used for processing such kind of information. The Levenshtein similarity quantifies the similarity of two words by the number of letter exchange operations (insertions, deletions, or replacements of characters) on one of the two words that are necessary to obtain the other word. These operations are defined as having different costs, for example insertions and deletions may have costs of 1, replacements costs of 2, etc. The sum of the costs per operation defines a distance between words, called the Levenshtein distance. The Levenshtein similarity itself then is calculated as the sum of the length of the strings minus the Levenshtein distance  $L_{dist}$ , divided by the sum of the length of the strings.

$$L_{ratio}(a, b) = \frac{\sum(\text{len}(a), \text{len}(b)) - L_{dist}}{\sum(\text{len}(a), \text{len}(b))}$$

To give an example, the string lengths of the term “productive” equals 10 and the string lengths of the German term “produktiv”, which is just a translation, equals 9, so the sum of the string lengths is 19 and their Levenshtein distance is 2, operatively corresponding to one replacement (costs of 2) and one deletion (costs of 1). Consequently, the Levenshtein similarity is calculated as  $\frac{(19-3)}{19}$ , i.e. approximately 0.84. In this way, words in a text corpus can be compared with the words for example in an online-dictionary. After defining a threshold similarity (e.g. 0.8), a part of the misspellings can be corrected by exchanging words in the job-ads for words in the dictionary if their Levenshtein similarity is above threshold.

## 5.2 Latent Semantic Analysis

Apart from statistical methods for processing and analyzing large text corpuses, more involved methods exist, which to some extents are capable of extracting *semantic* aspects (i.e., “meaning”) from texts. One of these methods is *Latent Semantic Analysis* (LSA), sometimes also known as Latent Semantic Indexing (LSI) (Deerwester et al. 1990). LSA is a mathematical method for revealing latent relationships within a collection of documents. It assumes that words close in meaning will occur in similar pieces of text. Rather than looking at each document by itself, LSA looks at a corpus of documents as a whole and analyses the correlation and context of terms within this corpus in order to identify relationships. A typical result of an LSA would be a search engine search for the term “sand” which among others also

returns documents that do not contain the term “sand” but contain terms like “beach” or “shore”. LSA would have shown that the term “sand” is semantically close to the term “beach” in this case.

LSA starts out from the fact that the words of our language do not just unambiguously refer to one concept of meaning (left in figure 77), but can have multiple meanings (right in figure 77).

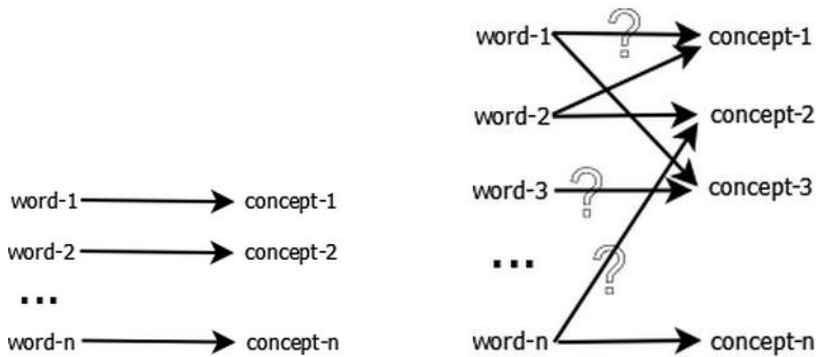


Fig. 77: Ambiguity of language (own illustration).

In natural language, these ambiguities are solved by the context in which words are used. While the word “bank” for instance, when used in the context of terms like “mortgage”, “loans” or “rates”, probably refers to a financial institution, it could refer to a river bank when used together with “lures”, “casting”, and “fish”. In order to find the meanings or concepts behind the words hence, LSA attempts to map both words and their contexts into a “concept space” in which different meanings can be compared. For this, it filters out some of the noise that arises when different authors use different words to express the same meaning. Basically thus, LSA is a statistical method, which builds on the following simplifications:

- Word contexts (which usually are called documents, with the size of a document ranging from a sentence to a paragraph to whole articles) are represented as „bags of words”, where the order of the words is not important. What counts however, is how many times each word occurs.
- Concepts are represented as patterns of words that usually appear together in a document, such as “leash”, “treat”, and “obey” may appear together in a document about dog training.

- Words are treated as having just one meaning, although this is clearly not the case (as mentioned above).
- Usually, a set of words called “stop words” is excluded from the analysis. These words, like “and”, “or”, “for”, “in”, “of”, “the”, “to” etc. do not contribute much (if any) meaning to a context.
- Words are stemmed, meaning that they are reduced to their word stem, like “measure” in “measurement”, “measuring” or “measure” etc.

The first step of an LSA consists of creating a *term-document matrix*, in which each row represents a word and each column a document. The cells of the matrix contain the frequencies with which the term occurs in the document. Terms are reduced to their stem and stop words are excluded.

As an example, consider the following nine sentences (documents D1 - D9) containing definitions of productivity:

- D1: “A measure of the efficiency of a person, machine, factory, system, etc., in converting inputs into useful outputs. “
- D2: “Productivity is computed by dividing average output per period by the total costs incurred or resources consumed in that period. “
- D3: “Productivity is a critical determinant of cost efficiency. “
- D4: “An economic measure of output per unit of input. Inputs include labor and capital, while output is typically measured in revenues and other GDP components. “
- D5: “Productivity is measured and tracked by many economists as a clue for predicting future levels of GDP growth.”
- D6: “Productivity gains are vital to the economy because they allow us to accomplish more with less. “
- D7: “Productivity is the ratio of output to inputs in production; it is an average measure of the efficiency of production. “
- D8: “The rate at which radiant energy is used by producers to form organic substances as food for consumers. “
- D9: “Productivity is commonly defined as a ratio between the output volume and the volume of inputs. “

We do not consider all words in these documents, but focus on the following list of already stemmed terms:

['measur', 'effici', 'machin', 'factori', 'system', 'input', 'output', 'averag', 'cost', 'resourc', 'consum', 'econom', 'labor', 'revenu', 'gdp', 'predict', 'futur', 'growth', 'gain', 'accomplish', 'energi', 'produc', 'food']

	D1	D2	D3	D4	D5	D6	D7	D8	D9
measur	1	0	0	2	1	0	1	0	0
effici	1	0	1	0	0	0	1	0	0
machin	1	0	0	0	0	0	0	0	0
factori	1	0	0	0	0	0	0	0	0
system	1	0	0	0	0	0	0	0	0
input	1	0	0	2	0	0	1	0	1
output	1	1	0	2	0	0	1	0	1
averag	0	1	0	0	0	0	1	0	0
cost	0	1	1	0	0	0	0	0	0
resourc	0	1	0	0	0	0	0	0	0
consum	0	1	0	0	0	0	0	1	0
econom	0	0	0	1	0	0	0	0	0
labor	0	0	0	1	0	0	0	0	0
revenu	0	0	0	1	0	0	0	0	0
gdp	0	0	0	1	1	0	0	0	0
predict	0	0	0	0	1	0	0	0	0
futur	0	0	0	0	1	0	0	0	0
growth	0	0	0	0	1	0	0	0	0
gain	0	0	0	0	0	1	0	0	0
accomplish	0	0	0	0	0	1	0	0	0
energi	0	0	0	0	0	0	0	1	0
produc	0	0	0	0	0	0	0	1	0
food	0	0	0	0	0	0	0	1	0

Fig. 78: *Term-document matrix* as generated from documents D1 – D9.

In the next step, the raw matrix counts are modified so that rare words are weighted more heavily than often used words. In this way, a word that occurs only in a small number of documents is weighted more heavily than a word that occurs in most of the documents. A common weighting method is called *TFIDF* (Term Frequency – Inverse Document Frequency), which replaces the count in each cell according to the following formula:

$$FIDF = \frac{N_{i,j}}{N_{*,j}} * \log \frac{D}{D_i},$$

where

- $N_{i,j}$  = the number of times word  $i$  appears in document  $j$  (the original cell count).
- $N_{*,j}$  = the number of total words in document  $j$  (sum of the entries in column  $j$ ).
- $D$  = the number of documents (the number of columns).
- $D_i$  = the number of documents in which word  $i$  appears (number of non-zero columns in row  $i$ ).

The  $S_k$  matrix of singular values can provide information about how many dimensions or “concepts” should be considered. One way to do this, is to plot the squared singular values, as shown to the right, indicating that the first value seems to add significantly more information to the analysis than the others. However, this first dimension provides an absolute value. For documents, it corresponds to the length of the document, for words, it corresponds to the number of times a word is used in all documents. To get more meaningful information, it can make sense to ignore the first dimension and consider some of the following ones.

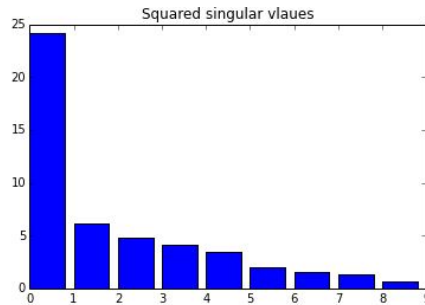


Fig. 79: Squared singular values (own illustration).

In the case of this example, the focus is on the second and third dimension. With the SVD done for  $k = 3$  hence and ignoring the first values in each matrix, the second and third values of the  $T_k$  matrix provide the coordinates of each word in a concept space and the second and third values in the  $P_k$  matrix provide the coordinates

of each document in the concept space. Plotting this concept space, as shown below, provides information as to which words are clustered in the vicinity of which documents (sentences), and vice versa, which documents are indicative for which words.

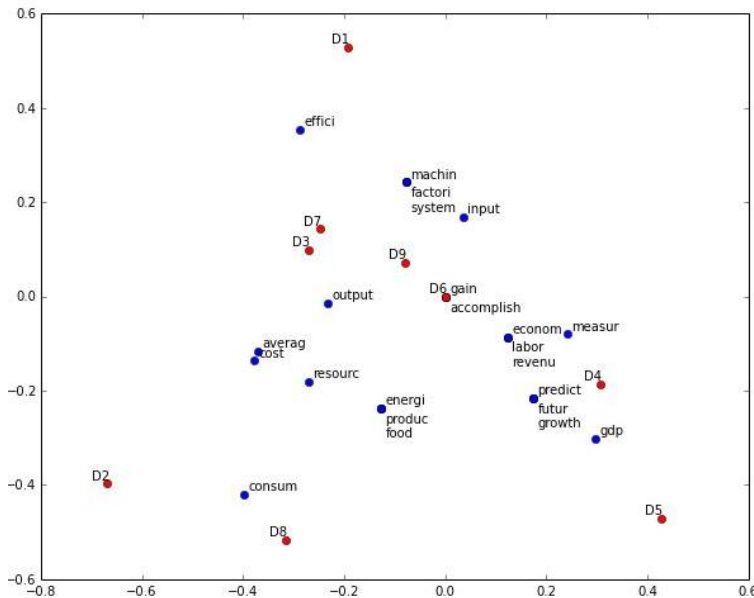


Fig. 80: Concept space illustrating word and document clustering (own illustration).

### 5.3 Vector representations

Although LSA and its derivatives like Latent Dirichlet Allocation (LDA) or stochastic LDA are interesting and powerful tools for text analyses where the meaning of words need to be considered through analyzing the context of these words, the real breakthrough in assessing semantics was achieved by representing the use of words in language in the form of vectors. With vector representation it became possible to consider complex linguistic concepts like e.g., synonyms or antonyms or references to information from preceding sentences etc. Most current translation software (<https://translate.google.at>, <https://www.deepl.com/translator>) builds on this possibility.

### 5.3.1 Assessing similarities by vector representation

For understanding the principles of this methodology, it is necessary to know how similarities can be mathematically assessed. To see how this is done, consider the following example.

The table below shows the 2019 ratio of population to Gross Domestic Product (GDP) for certain countries, according to the IMF World Economic Outlook database<sup>10</sup>. If we plot this data with population on the  $x$ -axes and GDP on the  $y$ -axes, we can clearly see that some countries – Japan, Germany, Italy, UK – appear to be more *similar* in these two aspects than, say, the United States and India.

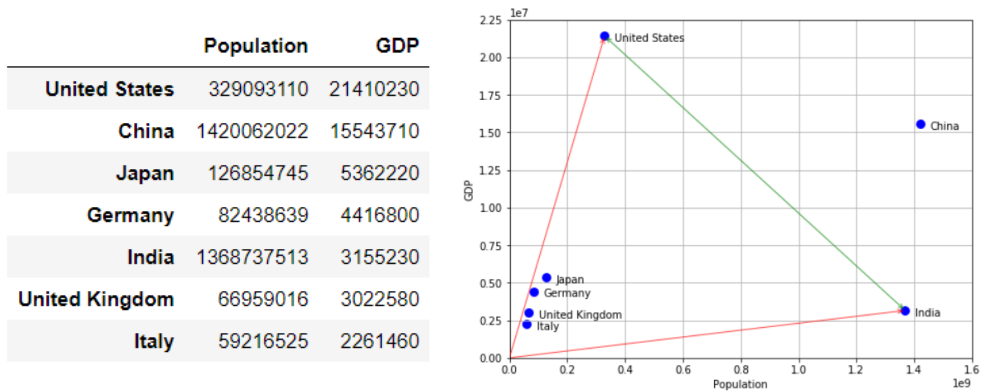


Fig. 81: Ratio of population to GDP for selected countries (own illustration).

One way to mathematically assess this similarity between data points would be to calculate the *Euclidian distance*, which gives the length of the path connecting the data points (the green line). This can be done with the formula  $ED = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ , if we consider the population values as the  $x$ -part and the GDP-values as the  $y$ -part of two-dimensional coordinate-vectors. This formula yields an  $ED = 1\,039\,804\,659,40$  for the distance between the USA and India and an  $ED = 44\,426\,166,73$  for the distance between Germany and Japan.

<sup>10</sup> <https://www.imf.org/external/pubs/ft/weo/2018/01/weodata/download.aspx>



Another, and more common way of comparing data point similarities is to calculate the *cosine similarity* of the vectors  $A$  and  $B$  pointing to the data points (red lines). The formula for this is

$$S = \frac{\sum_{i=1}^n AB_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}}$$

The vectors are given by the coordinates of the above plot, that is, by the values for population and GDP for each country. The cosine is defined by the angle between the vectors. If the vectors are identical – with an angle of  $0^\circ$  between them –, then the cosine is 1. If they are orthogonal to each other ( $90^\circ$ ), the cosine equals 0, and if they are exactly opposed – with an angle of  $180^\circ$  – then the cosine is -1, as can be easily derived from the unit circle as shown below.

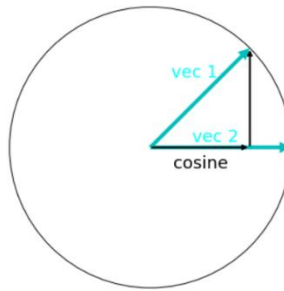


Fig. 82: Two vectors on the unit circle (own illustration).

For getting results in correspondence to the unit circle example from calculating the cosine similarity of the countries in our database, we have to normalize the values prior to calculation. We can do this, as shown in the code box, with the `MinMaxScaler` of the *scikit-learn* Python module, which also offers a function for the cosine similarity. Executing the code yields a  $CS = 0.24182053$  for USA and India and a  $CS = 0.98916326$  for Japan and Germany.

```
# normalize before applying cosine similarity
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics.pairwise import cosine_similarity

sc_prod = MinMaxScaler().fit(prod).transform(prod)
print(cosine_similarity([sc_prod[0], sc_prod[4]])) # US/India
print(cosine_similarity([sc_prod[2], sc_prod[3]])) # Japan/Germany
```

Fig. 83: Code for calculating cosine similarity.

There are several more methods to consider the similarity of vectors<sup>11</sup>. The important aspect for text mining however is that *words as well as sentences as well as whole documents can be vectorized too*. In other words, they can be represented as vectors in a multi-dimensional *vector space* and consequently can be compared analogously to how here a couple of countries is compared.

### 5.3.2 Vector Space Representation of Words – Word2Vec

How can words be vectorized? And may be even more important: why should we want to do this in the first place?

As is obvious, words have a “physical” appearance in the letters of which they consist. Additionally, however, words also have a semantic aspect, which makes them carriers of *meaning*. The physical appearance of a word does not convey this meaning by itself. Looking at the word “bank”, as said, does not tell us whether it indicates a financial institution or something to sit on. The letters on their own do not reveal the meaning of a word. Rather, it is the *context*, in which words are used. Context mediates semantics. If we want computers to process natural language thus, it is necessary to make the computer attentive to the context in which words are used. This can be done through vectorization.

To understand how, consider for an example the first sentence of Dickens’ novel “A Tale of Two Cities”. The sentence reads:

“It was the best of times, it was the worst of times.”

Context in this case can be seen as the words surrounding a particular word in a certain radius (or window). The word “best” for example in this sentence is surrounded by the words “the” and “of” in radius = 1, or by “was the” and “of times” in radius = 2, and so on. If for simplicity we consider just a radius of 1 for each of the words in this sentence, we can derive the following table:

---

<sup>11</sup> See e.g.: <http://dataaspirant.com/2015/04/11/five-most-popular-similarity-measures-implementation-in-python/>

	START _	was	it _	the	was _	best	the _	of	best _	times	of _	it	times _	was	was _	worst	worst _	times	of _	END
it		1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
was		0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
the		0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
best		0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
of		0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
times		0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
worst		0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 84: Possible contexts within a radius of 1 word.

The header of this table holds the contexts that are possible in radius = 1. The table entries hold the numbers of word-occurrences in these contexts. Note that the contexts “it \_ the” and “the \_ of” appear two times in the example sentence. The corresponding entries in the table therefore represent the word “was” as two times occurring, and the words “best” and “worst” as two 1-entries for the context “the \_ of”. The rows of the table thus give the radius-1-vectors of the words in the left-most column, representing the semantic aspect of these words as they have it in this particular sentence (and only there). These vectors can be used for vector calculations, as we have applied them above, with the – mathematically irrelevant – difference that these vectors now have ten dimensions and not just two.

As you can see from the table, the vectors for the words “best” and “worst” - [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] - are identical, implying that these words are “the most similar words” in this case, which of course does not correspond to what we would expect. “Best” and “worst” are *antonyms*, not *synonyms* in English language. Obviously, our small toy example misleads the attempt to derive meaning from it. Therefore, in realistic settings *word vectors* never are generated from a single sentence. Usually, huge collections of texts are used, of which it is expected that in their comprehensiveness they represent a complete picture of how words are used in natural language. These text collections then are analyzed iteratively analogously to the example above, albeit with larger and different radii considered and computation thus being much more intensive. However, other than in the count-based methods of Latent Semantic Analysis and corresponding matrix transformations (see section 5.2), such word embeddings can be produced with specialized neural networks, for example the so-called Word2Vec models, as developed by Tomas Mikolov at Google in 2013 (Mikolov et al. 2013)<sup>12</sup>. These models are shallow, two-layer neural networks

<sup>12</sup> See also <https://code.google.com/archive/p/word2vec/>

taking a large corpus of text as input and producing a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space.

Fortunately, we do not have to create such high dimensional word vector spaces by ourselves. At the time being, several pre-trained collections of *word embedding databases* exist on the internet. Often, they are downloadable for free, or they come included into specialized modules for natural language processing, such as for instance<sup>13</sup> the Global Vectors for Word Representation (GloVe)<sup>14</sup> by the Stanford University included in the Python module spaCy (<https://spacy.io/>), trained on the text corpus of the complete English-language Wikipedia.

However, in more specialized cases – such as particular scientific contexts for instance – you may get better results when training Word2Vec models on your own. For this, different methods exist, distinguished by the way the distributed representation of words is considered (Mikolov et al. 2013). One such method is the so-called *continuous bag-of-words* (CBOW) architecture, meant to predict a word from a window of surrounding context words with the order of context words not influencing prediction. Another method is the *skip-gram* architecture where words are meant to predict the surrounding window of context words and nearby context words are weighted more heavily than more distant context words. According to the developer of the methods, CBOW is faster while skip-gram is slower but does a better job for infrequent words (Mikolov et al. 2013).

### 5.3.3 Vectorization examples<sup>15</sup>

In order to illustrate the use of such word embeddings, we can compare word vectors on the background of the above-mentioned GloVe database from Stanford, which, as said, is included in the Python module spaCy. The database can be accessed with the following code:

---

<sup>13</sup> For others see: <https://research.fb.com/fasttext/> or <https://code.google.com/archive/p/word2vec/> trained on roughly 100 billion words from the Google News dataset

<sup>14</sup> <https://nlp.stanford.edu/projects/glove/>

<sup>15</sup> Note that the given examples are toy examples, made up to illustrate the use of the methods and not meant to procure scientifically relevant insights. Nevertheless, results at times are impressive.

```
from __future__ import unicode_literals
import spacy
spacy.prefer_gpu()
nlp = spacy.load('en')
```

Fig. 85: Code for accessing the database with the Python module spaCy.

Once we have the pre-trained database on our computer, we can perform queries against this collection of word vectors. For example, we can ask for the cosine similarity of certain terms to the word “sustainable”. The following code executes a query for the words ‘renewable’, ‘viable’, ‘environment’, ‘climate’, ‘economy’, ‘Norway’, ‘Germany’ and ‘USA’.

```
import numpy as np
from numpy import dot
from numpy.linalg import norm

# generating vector
def vec(s):
    return nlp.vocab[s].vector

# cosine similarity
def cosine(vec1, vec2):
    if norm(vec1) > 0 and norm(vec2) > 0:
        return dot(vec1, vec2) / (norm(vec1) * norm(vec2))
    else:
        return 0

T = ['renewable', 'viable', 'environment', 'climate', 'economy',
     'Norway', 'Germany', 'USA']

for t in T:
    print(t + ': ', cosine(vec('sustainability'), vec(t)))
```

Fig. 86: Code for calculating Cosine Similarity.

The query yields the following cosine similarities to the term “sustainable”:

- renewable: 0.6980572
- viable: 0.56681484
- environment: 0.5673896
- climate: 0.5098402
- economy: 0.513057
- Norway: 0.1233881
- Germany: 0.12099999
- USA: 0.1176934

Next, we can load a text of interest and query it for the terms in it that are most similar (in terms of cosine similarity) to a given word. The piece of code in figure 87 loads the 1987 Brundtland-report of Our Common Future<sup>16</sup>.

```
datei = 'Brundtlandreport'
doc = nlp(open('C:\\temp\\Text-mining\\' + datei + '.txt').read())

# take only words with letters (alphabet = is_alpha)
tokens = list(set([w.text.lower() for w in doc if w.is_alpha]))

def spacy_closest(token_list, vec_to_check, n=20):
    return sorted(token_list,
                   key=lambda x: cosine(vec_to_check, vec(x)),
                   reverse=True)[:n]

closest = spacy_closest(tokens, vec("sustainability"))
closest_cleaned = [x for x in closest if 'sustain' not in x.lower()]
print(closest_cleaned)
```

Fig. 87: Code for loading the 1987 Brundtland-report.

It tokenizes it and investigates it for the most similar words to „sustainability“. As result, we get the following list:

['environmentally', 'livelihoods', 'biomass', 'renewable', 'biogas', 'cogeneration', 'renewables', 'geothermal', 'bioenergy', 'hydropower', 'environmental', 'ecological', 'conservancy', 'biodiversity', 'fisheries', 'reforestation', 'conservation']

If we do the same for the 2018 Sustainable Development Goals Report<sup>17</sup> we get a very similar list, however with terms like “ecology” or “cogeneration” replaced by “innovation”, “irrigation”, “cropland” etc.:

['environmentally', 'livelihoods', 'geothermal', 'bioenergy', 'renewables', 'biomass', 'hydropower', 'renewable', 'environmental', 'biodiversity', 'conservation', 'agriculture', 'innovations', 'innovation', 'irrigation', 'agricultural', 'cropland']

<sup>16</sup> <http://www.un-documents.net/our-common-future.pdf>

<sup>17</sup> <https://unstats.un.org/sdgs/files/report/2018/TheSustainableDevelopmentGoalsReport2018-EN.pdf>

### 5.3.3.1 Vector arithmetics

Interestingly, we can perform vector arithmetic with these word vectors, and in some cases, they seem to yield quite reasonable results. Mathematically adding for example the vectors of “livable” and “planet” together yields other vectors for the words in the following list:

['sustainability', 'livelihoods', 'environmentally', 'earth', 'ecological']

And subtracting the vector for “sustainable” from the one for “economy” yields:

['inflation', 'slowdown', 'recession', 'unemployment', 'collapse', 'shortfall']

### 5.3.3.2 Sentence similarity

From word vectors, *sentence vectors* can be generated, for example by taking the average of the word vectors for the vector of a sentence. These sentence vectors then can be compared as well. What is more, by making up new sentences, we can check what similar sentences a query in a document would produce. For example, if we query the Brundlandt-report with the sentence:

Will the human race survive?

the algorithm “answers” with the following suggestions, identified as the five sentences in the Brundtland-report that are most (cosine) similar to our question:

It will depend most of all upon humanity’s ability to prevent an arms race in space.

---

Will it not be fatal to our civilization, to the ecosystem of our planet?

---

Perhaps the greatest threat to the Earth’s environment, to sustainable human progress, and indeed to survival is the possibility of nuclear war, increased daily by the continuing arms race and its spread to outer space.

---

Nations must not become prisoners of their own arms race.

---

It would be grim irony indeed if just as new genetic engineering techniques begin to let us peer into life's diversity and use genes more effectively to better the human condition, we looked and found this treasure sadly depleted.

Note that there is not yet a full transformer technology involved in these example queries, like we now know its effects from the use of ChatGPT. The resulting answers however seem similar to certain extent, indicating that vectorization is one of the methods at the core of Generative Pre-trained Transformers.

### 5.3.4 Document similarity – Doc2Vec

Analogously to generating sentence vectors from word vectors, we can generate *document vectors* from sentence vectors. However, there are better methods for this than just taking the average of vectors. And fortunately, these methods too are provided for free, for example in the form of the Python module *gensim* (<https://radimrehurek.com/gensim/>), developed by Radim Řehůřek (Rehurek/Sojka 2010).

In section 5.2. of this book, we analyzed a set of definitions of the economic term “productivity” with the help of Latent Semantic Analysis (LSA). We plotted the thereby found term similarities on a two-dimensional graph.

As in this example, we consider each of these productivity definitions (consisting most of the time of just one sentence) as a (small) document in the following<sup>18</sup>. Subsequently, we apply these documents to vectorization with a *gensim* Doc2Vec-model. As a reminder, the definitions, taken from various economics textbooks, were:

- “A measure of the efficiency of a person, machine, factory, system, etc., in converting inputs into useful outputs.”
- “Productivity is computed by dividing average output per period by the total costs incurred or resources consumed in that period.”
- “Productivity is a critical determinant of cost efficiency.”

---

<sup>18</sup> Note that each of these definitions could be a paragraph, a scientific paper or even a complete book instead.



- “An economic measure of output per unit of input. Inputs include labor and capital, while output is typically measured in revenues and other GDP components.”
- “Productivity is measured and tracked by many economists as a clue for predicting future levels of GDP growth.”
- “Productivity gains are vital to the economy because they allow us to accomplish more with less.”
- “Productivity is the ratio of output to inputs in production; it is an average measure of the efficiency of production.”
- “The rate at which radiant energy is used by producers to form organic substances as food for consumers.”
- “Productivity isnt everything, but in the long run it is almost everything.”
- “Productivity is commonly defined as a ratio between the output volume and the volume of inputs.”

Let’s see which of these productivity definitions are most similar. For this, we first tokenize them and bring them into a particular form, called `TaggedDocument` in the context of *gensim*, which for the first definition looks like the following<sup>19</sup>:

```
TaggedDocument(words=['a', 'measure', 'of', 'the', 'efficiency', 'of', 'a', 'person', 'machine', 'factory', 'system', 'etc', 'in', 'converting', 'inputs', 'into', 'useful', 'outputs'], tags=['0'])
```

With this preparation, we can train a `Doc2Vec`-model on the given definitions. The difference to LSA is that the contexts of the words in these documents are “learned” by a neural network, which iterates many times over the training data (i.e., the set of definitions) and by and by adjusts its weights to optimally represent the found word contexts. This means that the resulting document vectors are subject to parameter variations such as training intensity (epochs trained), learning rate, considered window and vector size, and several more, and will need some experimentation to yield optimal results. Nevertheless, many text miners believe that the method produces better results than LSA and related methods. The code shown below defines and trains such a neural network.

---

<sup>19</sup> Note that the first definition is tagged with the number zero ['0'], the second with one ['1'], and so on.

```

# define and train Doc2Vec model
from gensim.models.doc2vec import Doc2Vec

max_epochs = 100 # training epochs
vec_size = 60 # dimensions considered
alpha = 0.025 # initial learning rate

model = Doc2Vec(vector_size = vec_size,
                alpha = alpha,
                min_alpha = 0.00025,
                min_count = 1, # Ignores all words with total frequency lower than this
                window = 3, # size of context considered
                workers = 4, # how many CPUs?
                dm = 1) # dm defines the training algorithm. dm=1 means 'distributed memory'
(FV-DM) # dm=0 means 'distributed bag of words' (FV-DBOW).
# Distributed Memory model preserves the word order in a document
# whereas Distributed Bag of words just uses the bag of words approach, which doesn't preserve any
word order.

model.build_vocab(tagged_data)

for epoch in range(max_epochs):
    #print('iteration {0}'.format(epoch))
    model.train(tagged_data,
                total_examples = model.corpus_count,
                epochs = model.epochs)
    # decrease the learning rate
    model.alpha -= 0.0002
    # fix learning rate, no decay
    model.min_alpha = model.alpha

#model.save('C:\\temp\\Text-mining\\d2v_2.model')
print('Model trained')

```

Fig. 88: Code for defining and training a Doc2Vec model.

After training, we can query the model for instance for the five definitions that are most similar to the first definition in our training set. This yields the following ranked list, showing the cosine similarity at the end of each definition.

- productivity is a critical determinant of cost efficiency 0.758
- productivity is the ratio of output to inputs in production; it is an average measure of the efficiency of production 0.58
- productivity is commonly defined as a ratio between the output volume and the volume of inputs 0.526
- productivity isnt everything but in the long run it is almost everything 0.447
- an economic measure of output per unit of input inputs include labor and capital while output is typically measured in revenues and other gdp components 0.332

More interestingly again may be a query with a new sentence that is not in the training set of productivity definitions. The code below compares the sentence “Does it make sense to constantly strive for more?” with the definitions in the training set and again lists the five most similar sentences as a sort of answer to our question.

```

new_text = 'Does it make sense to constantly strive for more?'

new_vector = model.infer_vector(new_text.lower().split())
sims = model.docvecs.most_similar([new_vector])
for s in sims[:5]:
    print(data[int(s[0])], round(s[1], 2))

```

Fig. 89: Comparing a sentence with the definitions in the training set.

- Productivity isn't everything, but in the long run it is almost everything. 0.63
- Productivity gains are vital to the economy because they allow us to accomplish more with less. 0.61
- Productivity is a critical determinant of cost efficiency 0.6
- The rate at which radiant energy is used by producers to form organic substances as food for consumers 0.55
- Productivity is the ratio of output to inputs in production; it is an average measure of the efficiency of production. 0.49

Finally, we can use a clustering method called t-Distributed Stochastic Neighbor Embedding (Maaten/Hinton 2008) (tSNE, included in `sklearn.manifold`<sup>20</sup>) to bring our 60-features vectors down to just two dimension so that we can plot it as a graph. As you can see, the term productivity is pretty much centered in this depiction, and the ten documents (each colored differently) are somehow star-like clustered around this center. The algorithm however, does not well account for the readability of the data labels. One would need to apply additional means to draw data points further apart.

---

<sup>20</sup> <https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>

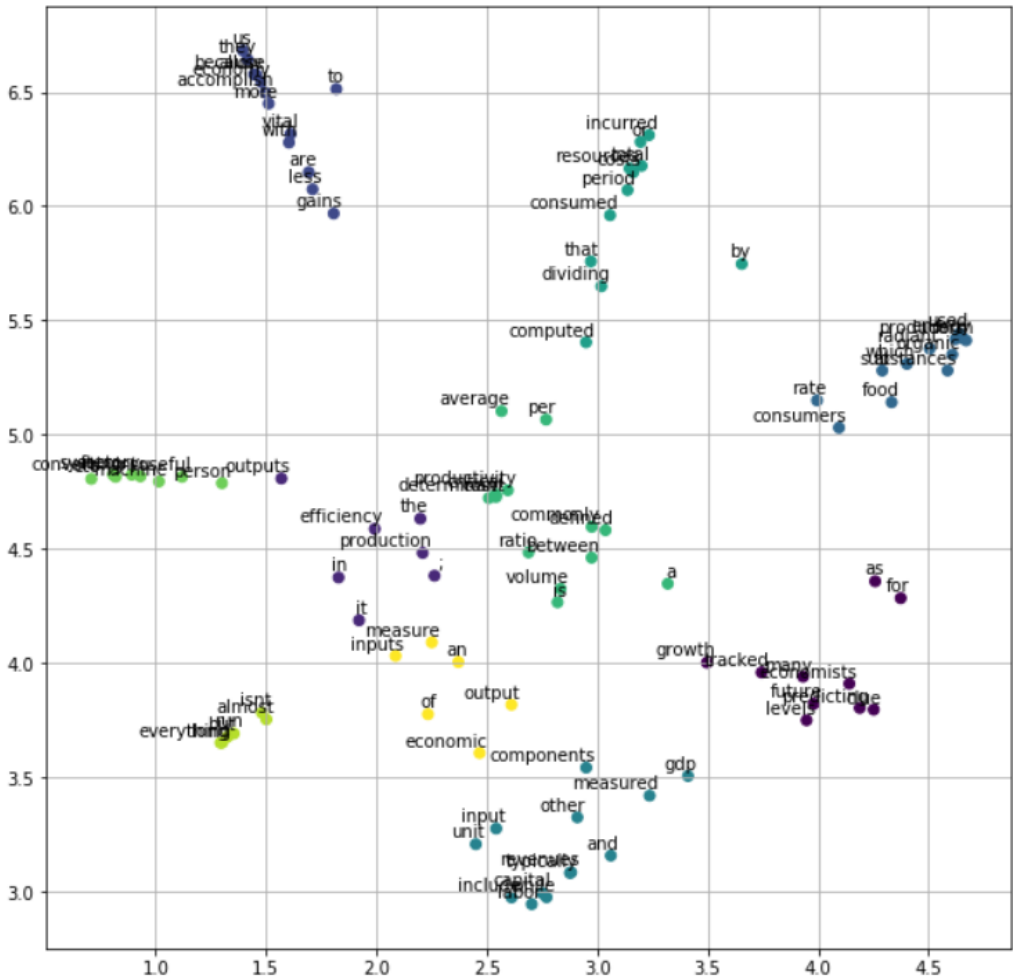


Fig. 90: Result of t-Distributed Stochastic Neighbor Embedding (tSNE) (own illustration).

In this example, the analytical unit to be considered as a vector is a word. Such undivided units are also called *unigrams*. A bigram in contrast, is a concatenation of two consecutive tokens or words. Often in this context however, the term *n-gram* is used to refer to the concatenation of any *n* consecutive tokens. Even single syllables can be *n*-grams and considered in the same way as words in vector representations. The Python library *fastText* (<https://fasttext.cc/>) for instance, allows to consider *n*-grams and is said to yield much better results on the level of considering syllables.

fastText was created by Facebook’s AI Research (FAIR) lab. It uses a neural network for word embedding, i.e., vectorization.<sup>21</sup>

### 5.3.5 Vectorization beyond textual contexts

There is yet another interesting aspect of the methodology that lays at the core of these techniques. A couple of internet companies like Yahoo, Spotify or AirBNB have applied the vector representation approach to recommender systems and advertising<sup>22</sup>. The idea behind this is simple but intriguing. If the meaning of a word can be inferred from its context – that is, from the words around a word –, then the habitual action of people might be derivable from actions these people undertake in the context of an action. Thus, large internet companies with access to recorded consumer behavior, e.g., in the form of clicks on online-products, have reasoned that the time series of online user activity offer them the same opportunity for inferring meaning from context as context words do it in texts. Users who are browsing around and interacting with different content on the internet, thus, allow it to infer the abstract qualities of this content from what content they are interacting with before and after. The companies, which adopted this method, report that they could lift their click-through-rate (CTR) at times quite impressively with this method.

If you reconsider the approach under this perspective, vectorization (i.e., word-embedding or Word2Vec) appears to have a much broader scope than NLP alone. It offers a potentially comprehensive way to make things (such as words) that resist quantification fit for being processed on a computer. It does this by directing attention away from the things themselves to their context. Conditions for application are simply that things are separable clearly enough, that they are finite in a given context and that you have the possibility to observe (and record) a sufficiently large amount of them.

---

<sup>21</sup> A gensim-implementation is available at [https://radimrehurek.com/gensim/auto\\_examples/tutorials/run\\_fasttext.html](https://radimrehurek.com/gensim/auto_examples/tutorials/run_fasttext.html). Additionally, the facebook-lab provides for a Python library itself: <https://fasttext.cc/docs/en/supervised-tutorial.html>

<sup>22</sup> See for example: <https://towardsdatascience.com/using-word2vec-for-music-recommendations-bb9649ac2484> or <https://erikbern.com/2013/11/02/model-benchmarks.html> or <https://medium.com/airbnb-engineering/listing-embeddings-for-similar-listing-recommendations-and-real-time-personalization-in-search-601172f7603e>

## 5.4 Sentiment analysis

Another commonly applied automated text analyses technique is Sentiment Analysis (aka Sentiment Detection or Opinion Mining), which tries to infer people's sentiments as expressed in text documents. To illustrate this technique, the following refers to an example, provided by (Liu 2010), of an "opinionated" document (i.e. a document that expresses the opinion of its author):

(1) I bought an iPhone a few days ago. (2) It was such a nice phone. (3) The touch screen was really cool. (4) The voice quality was clear too. (5) Although the battery life was not long, that is ok for me. (6) However, my mother was mad with me as I did not tell her before I bought it. (7) She also thought the phone was too expensive, and wanted me to return it to the shop.

As can easily be seen, there are several opinions expressed in this review. Sentences (2), (3) and (4) express positive opinions, while sentences (5), (6) and (7) express negative ones. All opinions have some targets or objects on which sentiments are expressed. The opinion in sentence (2) is on the iPhone as a whole, and the opinions in sentences (3), (4) and (5) are on the "touch screen", "voice quality" and "battery life" respectively. The opinion in sentence (7) is on the price of the iPhone, but the opinion/emotion in sentence (6) is on "me", not the iPhone. This can be important, since users often may be interested in opinions on certain targets or objects, but not on all. Finally, the source or holder of the opinions in sentences (2), (3), (4) and (5) is the author of the review ("I"), but in sentences (6) and (7) it is "my mother". Good sentiment analysis would have to be able to distinguish all these cases. However, what for a human reader seems intuitive and easily done, can be an arduous task for a machine. For being able to comprehend the difficulty, consider the following terminology:

- object: iPhone
- component (can be an object in its turn): battery
- feature (or topic): battery life
- general opinion: I like iPhone"
- specific opinion: "The touch screen of iPhone is really cool"
- explicit feature: "The battery life of this phone is too short"
- implicit feature: "This phone is too large"
- feature indicator: "large" is not a synonym of size. It is just an indicator.
- opinion holder or source: the holder of an opinion
- orientation of an opinion on a feature: positive, negative or neutral.

- explicit opinion: “The phone is great”
- implicit opinion: “The phone broke in two days”
- strength of opinion, can be scaled: e.g., strong (“This phone is a piece of junk”), weak (“I think this phone is fine”).
- direct opinion: a quintuple  $(o_j, f_{jk}, oo_{ijkl}, h_i, t_l)$ , where  $o_j$  is an object,  $f_{jk}$  is a feature of the object  $o_j$ ,  $oo_{ijkl}$  is the orientation or polarity of the opinion on feature  $f_{jk}$  of object  $o_j$ ,  $h_i$  is the opinion holder and  $t_l$  is the time when the opinion is expressed by  $h_i$ . The opinion orientation  $oo_{ijkl}$  can be positive, negative or neutral
- comparative opinion: a relation of similarities or differences between two or more objects, and/or object preferences of the opinion holder based on some of the shared features of the objects. usually expressed using the comparative or superlative form of an adjective or adverb

The extraction of direct opinions thus would consist of the following steps, given an opinionated document  $d$ ,

1. discover all opinion quintuples  $(o_j, f_{jk}, oo_{ijkl}, h_i, t_l)$  in  $d$ , and
2. identify all the synonyms  $(W_{jk})$  and feature indicators  $I_{jk}$  of each feature  $f_{jk}$  in  $d$ .
3. then generate a feature-based summary of the opinions, which could look like follows:

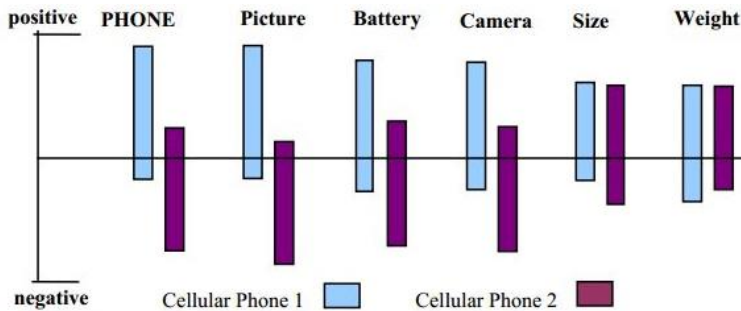


Fig. 91: *Buzz summary* showing frequencies of mentions of different competing objects, taken from Liu (2010), p. 636.

Alternatively, a buzz summary would show the frequency of mentions of different competing objects and thus would inform about the popularity of objects (products or brands) in a market place.

A related method is Trend tracking, which monitors the time a product needs to be opinionated and how opinions change over time. The related concept of Named Entity Recognition (NER) comprises the tasks to identify opinion holders, object names and time of postings, and to consider aspects like the frequency of terms, adjectives as important indicators of subjectivities and opinions, or so-called opinion words (or opinion phrases and idioms) like “beautiful”, “wonderful”, “good”, “amazing” as indicating positive sentiments, and “bad”, “poor”, or “terrible” as indicating negatives ones. A special focus thereby is often put on negations since they may change the opinion orientation in a sentence, like for example in the sentence “I don’t like this camera”, where the phrase “I like” indicates positive sentiment, while “don’t” changes its orientation.

Before the introduction of the transformer technology and tools like ChatGPT (see next chapter), these methods needed to be laboriously prepared individually for specific computer-based text examinations with lots of efforts and good coding skills. Now they have been incorporated as prerequisites in the further development of the so-called Large Language Models (LLMs). These LLMs are able to analyze any text within seconds with regard to the sentiments expressed in it. As an example, figure 92 shows a sentiment analysis of a part of a song text, analyzed with ChatGPT-4o.

Even quantifying the sentiments and visualizing them, for example as a trajectory in correlation to the text, is no problem for the technology. The same applies to other scientific analysis methods, which previously required time-consuming research work and collaboration of often large interdisciplinary research teams to execute them. It seems predictable that the advent of this technology is on the way to fundamentally change several aspects of sciences as we know it – and with them every other aspect of our existence that is affected by the processing of knowledge.





The provided text is rich in imagery and emotion, conveying a range of sentiments throughout. Here is an analysis of the overall sentiment and key segments:

#### 1. Opening Reflections:

- "I heard it in the wind last night / It sounded like applause / Did you get a round resounding for you / Way up here / Seems like many dim years ago / Since I heard that face to face / So seen you face to face / Though tonight I can feel you here"
- **Sentiment:** Nostalgic and melancholic. The speaker reflects on past moments and feelings of connection, creating a sense of longing and bittersweet remembrance.

#### 2. Observations and Messages:

- "I get these notes / On butterflies and lilac sprays / From girls who just have to tell me / They saw you somewhere"
- **Sentiment:** Slightly wistful. The messages about the person being seen elsewhere evoke a sense of loss or absence.

#### 3. Poet and Singer:

- "In some office sits a poet / And he trembles as he sings / And he asks some guy / To circulate his soul around"
- **Sentiment:** Reflective and contemplative. This section portrays the vulnerability and yearning of a poet seeking to share his inner self.

#### 4. Rise to Fame:

- "On your mark red ribbon runner / The caressing rev of motors / Finely tuned like fancy women / In Thirties evening gowns / Up the charts / Off to the airport / Your name's in the news / Everything's first class / The lights go down / And it's just you up there / Getting them to feel like that"
- **Sentiment:** Admiring yet critical. There is an acknowledgment of success and the allure of fame, but also an underlying critique of its superficial aspects.

Fig. 92: ChatGPT-4o sentiment analysis of a part of a song text.

## 5.5 Transformers, or: Attention is all you need

The comprehension of qualities in texts like opinions and sentiments, is, as we have seen, very dependent on a clear understanding of the meaning of words in their context, that is, on the semantics of a text. A tricky problem in this regard, closely related to the difficulty of understanding context, is the comprehension of references. Consider the following two sentences:

The trophy does not fit into the bag because it is too large.

The trophy does not fit into the bag because it is too small.

While it is no problem for a human to understand that the word “it” in the first sentence refers to the trophy and in the second sentence to the bag, this poses a difficulty for digital machines. This kind of non-comprehensions is a well-known problem in AI-research commonly attributed to missing “world knowledge”, i.e., the knowledge we humans have about many things but are not aware that we have it, such as e.g., knowing that an apple, when loosing connection to the tree, due to gravity will fall to the ground and not rise to the sky. In the context of research in natural-language understanding (NLU) comprehension as needed for the trophy-example above can be achieved with the help of a type of neural network that, when embedding (or vectorizing) words, uses a kind of specialized internal memory to consider contexts from earlier phases of its learning process. In section 4.5.1. we spoke about LSTM neural networks, which have an architecture that allows them to consider information distributions over long sequences if they are trained on them. Theoretically, such LSTMs could pass information even over infinitely long sequences. In practice, however, there capacity is limited due to the vanishing gradient problem. Eventually, LSTMs too tend to forget earlier tokens.

To tackle this problem, a different network architecture under the term Transformer has been suggested (Vaswani et al. 2017) that is able to maintain direct connections to all previous timestamps it encounters, allowing information to be passed on over much longer sequences, with the cost however, that the ANN is now directly connected to an exploding amount of input. Transformers thus need a possibility to separate important from unimportant input. For this, they use an algorithm called attention.

Attention is used to focus the processing activity of a neural network on those contexts of inputs – words or sequences of words – which, according to the so far processed information, seem more important than others. Encoder-Decoder networks for example (see section 4.6.2) are, in their undercompleteness, able to generate

compressed vector representations of the context of a word, which, when applied in an attention-based variant, are in each step assembled together and compared to each other. This comparison is exposed to a softmax function, which normalizes the vector so that each of its components lies in the interval  $[0,1]$  and the components add up to 1 making it possible to interpret them as probabilities. Larger input components thus correspond to larger probabilities implying higher chances to attract the network's attention. The relevant contexts then are used to e.g., translate words from one language to another. With this, the German translation of the phrase "to book a flight" can correctly be "einen Flug buchen", instead of (as in earlier translation programs) "zu Buch ein Flug".

Such transformer NNs were with impressive success applied to a wide variety of tasks, many of which until recently are considered beyond the meaningful use of ANNs. Language translation is just one example. One of these language representation models has made headlines under the name of BERT, which stands for Bidirectional Encoder Representations from Transformers (Devlin et al. 2019) and has been used to create state-of-the-art models for tasks like question answering or language inference. It has pushed a range of benchmarks, which have been proposed to test the performance of such language models, such as GLUE (Wang et al., 2018) and decaNLP (McCann et al., 2018). Other attention-grabbing results have been achieved with contextualized word representations like ELMo (Peters et al. 2018) and in particular with the Generative Pre-trained Transformer, short GPT, which we will discuss in the next chapter. Early impressive examples of human-like text generation with autoregressive language models can be seen at <https://deepai.org/machine-learning-model/text-generator>. Similar impressive has been the Google-project LaMDA, short for "Language Model for Dialogue Applications", which as the name says, can engage in a sort of free-flowing conversation about a seemingly endless number of topics (see <https://blog.google/technology/ai/lamda/>). More details on techniques like these are provided at sites like <https://ai.stanford.edu/blog/contextual/>

# 6 ChatGPT et al.

In December 2022, the formerly open-source oriented startup OpenAI (<https://openai.com/>) enabled free Internet access to ChatGPT version 3.5, sparking tremendous public interest in the prospects for Artificial General Intelligence (AGI), that is, in the prospects of a kind of AI that does not solely focus on specific functions or solving specific problems like the gaming DNNs AlphaGo or AlphaZero, but instead strives towards a kind of human or even super-human general problem-solving intelligence. The explicitly AGI-oriented activities of OpenAI sparked a wide range of debates about the prospects, risks and consequences of machines reaching or exceeding human cognitive capabilities (Mitchell and Kraukauer 2023) or even sentience (Butlin et al. 2023).

Indeed, meanwhile many of the utterances elicited by ChatGPT and Co led even problem-savvy experts familiar with ML methods to interpret more than just machine logic into the probability-generated AI-chats. To give a sense for the capabilities of this evolution of the transformer technology, I asked ChatGPT-4 (<https://chat.openai.com/>) to briefly introduce itself and its development history. See the answer on the next page in figure 91.

One of the cornerstones of this technology's capabilities is the upscaling of data resources used to train the Transformer Neural Network. A highly diverse and very large data set with several terabytes of text data from the internet, from books, articles, websites and other sources is used. Some even say it encompasses already most of the texts currently available in digital form, which is the reason why these neural networks are called Large Language Models (LLMs). Their training involves running computations on large-scale clusters of GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units), and it took a significant amount of time (reportedly several weeks), the exact duration of which as well as the exact size of the data set is kept secret by OpenAI. The need for computation required enormous financial resources, which forced OpenAI among others into an agreement with Microsoft using their Azure supercomputing infrastructure, powered by Nvidia GPUs, that allegedly was built specifically for OpenAI and that reportedly costed "hundreds of millions of dollars". As a result of this support by a proprietary company, OpenAI's initially explicitly highlighted open-source ideology was put aside

and key information about the exact size and architecture of data and code is withheld.

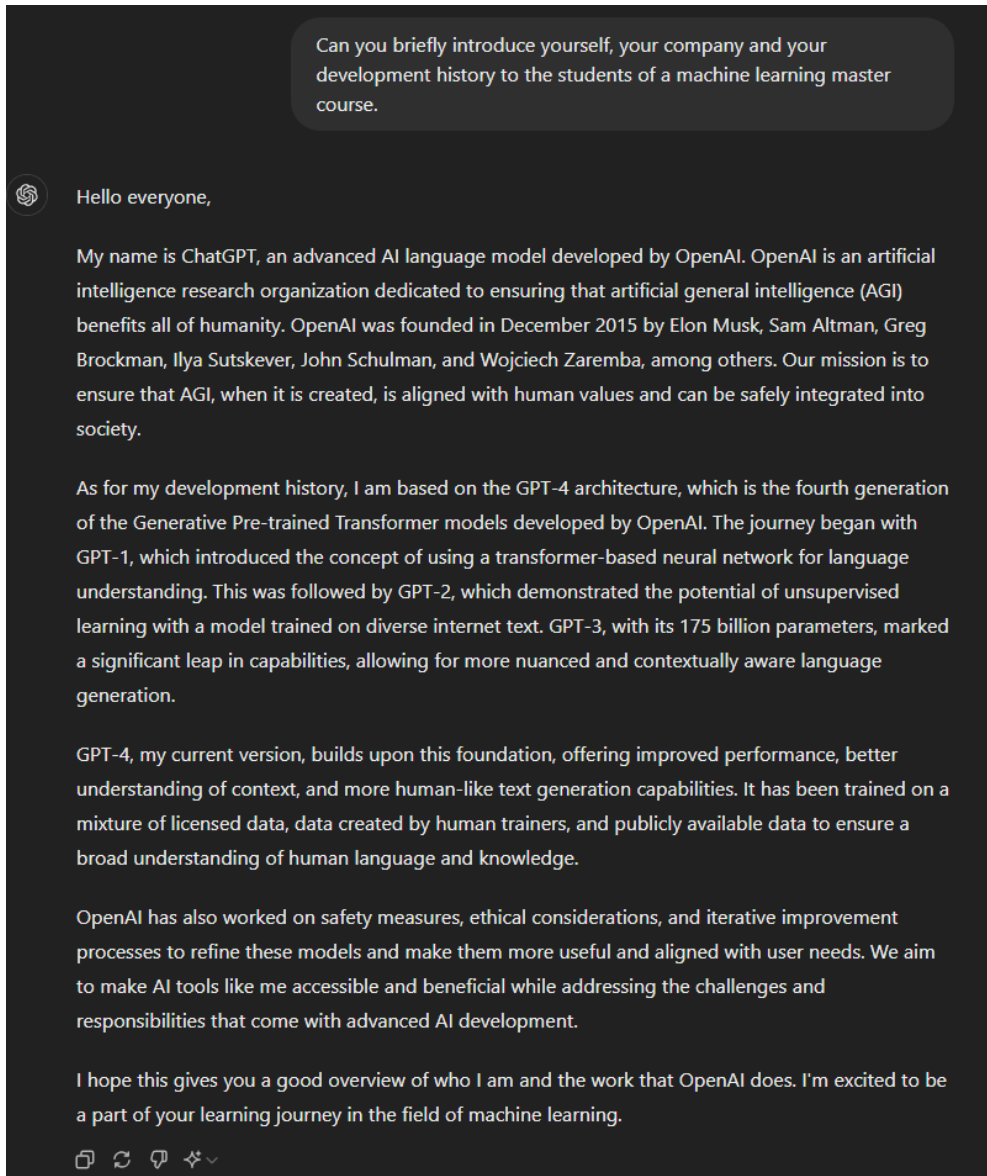


Fig. 93: Answer of ChatGPT-4o to request of briefly introducing itself.

In response to this loss of openness, meanwhile numerous other ventures formed as an alternative on an allegedly open-source basis to train and specialize LLMs. Unfortunately, many of these too are not as open as the original open-source ideology would suggest it. The hype around AI that ChatGPT has triggered, but above all the technical and financial effort involved in training, obviously puts business first for many, so that most projects are still characterized by a great lack of transparency and verifiability. Nevertheless, there are now a large number of highly specialized projects based on the transformer architecture and on the fact that already trained neural networks can be further trained for specific knowledge domains with additional data (see for a list: <https://huggingface.co/docs/transformers/index>).

Additionally, to the initial unsupervised training step on a massive corpus of text data – a process referred to as *pre-training* and meant to obtain the principal ability to understand the structure of language and generate human-like text-output – GPTs are finetuned in several regards. One of these adjustment steps involves *instruction training*, which enables the model to understand and follow user instructions accurately. With this, GPTs become able to answer questions and to engage in dialogues. Fine-tuning is done with supervised methods on curated datasets, comprising examples with instructions and corresponding desired outputs, covering a wide range of tasks and topics. Additionally, Reinforcement Learning from Human Feedback (RLHF) is employed, using human feedback on responses and reward models trained to predict the quality of the responses. Finally, extensive evaluations try to ensure the models perform well across different tasks and scenarios, involving benchmarking (i.e., comparing the model's performance against established benchmarks and other models) and real-world testing (deploying the model in controlled real-world settings).

With these fine-tunings, GPTs have become quite powerful tools for supporting scientific investigations. A wide range of tools is offered, particularly specialized on doing and aiding scientific research work. Currently it seems that this will put scientific research on a completely new footing. Some examples of these GPT-based tools are listed in the following:

- Consensus (<https://consensus.app/>) gives research-backed answers to yes-or-no questions
- Semantic Scholar (<https://www.semanticscholar.org/>) and Elicit (<https://elicit.org/>) act as digital assistants — tidying up bibliographies, suggesting new papers and generating research summaries.

- Research Rabbit (<https://www.researchrabbit.ai/>), LitMaps (<https://www.litmaps.com/>) identify sources and Scite (<https://scite.ai/home>) — which tells a user not only how often papers are cited, but in what context — tracks academic discourse.
- Iris (<https://iris.ai/>) creates map-like visualizations that connect papers around themes. Feeding a ‘seed paper’ into Iris generates a nested map of related publications, which resembles a map of the world.

In addition, web browsers such as the Google Chrome browser offer the possibility to have GPTs quickly summarize documents in pdf format, as they are usually available in scientific repositories<sup>23</sup>, which may dramatically reduce the reading efforts of scientists.

Another game changer for many scientific tasks is the fact that LLMs were not trained on plain texts alone, but also on large amounts of computer code. The models are therefore able to generate code, seemingly effortlessly, in almost all common programming languages. More specialized versions of such coder-tools as for example the Github Copilot (<https://github.com/features/copilot>, not to be confused with Microsoft’s Copilot platform) are comprehensive AI-based developer tools, including debugging and security remediation assistance. Scientists, who often are not trained in writing computer code themselves, are no longer dependent on seeking aid from software engineers and developers when in need of doing some more complex computation or of simply generating a correct and meaningful plot. The tools thus can save an enormous amount of time, which previously used to be spent on fine-tuning and making code snippets effective instead of on creative research work. Another example of supporting former tedious research work may offer the tool Whisper (<https://github.com/openai/whisper>), a machine learning model for speech recognition and transcription, also created by OpenAI, which makes it possible to convert spoken audio directly into text transcriptions. Other, maybe not that scientifically relevant tools are text-to-image or text-to-video GPTs, like the massively popular models Midjourney, Stable Diffusion, DALL-E or Sora (for more details on this see <https://huggingface.co/blog/text-to-video>).

One aspect that can affect scientific research, but has become particularly apparent with the introduction of text-to-image or text-to-video tools, is the tendency of LLMs to present misinformation in a quite convincing way. This is due to the fact that, on the one hand, the tools are efficiently trained to provide expected answers and

---

<sup>23</sup> Less relevant for scientific purposes Youtube movies can be summarized too.

hardly ever point out that certain knowledge is not available to them<sup>24</sup>. On the other hand, as described, the technology is based on the rather simple principle of calculating probabilities for a token following certain other tokens. Even though these probabilities are calculated in great detail, they of course impart no certainties. Especially when topics are concerned for which the training data holds little or no data at all, the answers of the tools can be incorrect. The models then tend to make up answers and hence should always be treated with caution. OpenAI for instance warns that ChatGPT “sometimes writes plausible-sounding but incorrect or nonsensical responses.” This behavior is commonly referred to as “hallucination”. (Alkaissi et al. 2023).

A well-known example of such hallucinations could initially be achieved by asking the tools to multiply large numbers. The result, when compared with that of an electronic calculator, often turned out to be wrong because the probability that particular this very multiplication occurred somewhere in the training data was very low. The tool however, trained to provide an answer in any case, hallucinated one. Such fairly simple errors have now been eliminated by connecting GPTs to a variety of external applications, such as calculators, but also databases or other repositories such as Wikipedia. If the GPT recognizes that it may provide an incorrect answer, it is designed to obtain the information from external sources. The question, however, is whether the tool recognizes that its answer may be flawed.

Beyond NLP, the transformer technology is applied to a wide range of tasks nowadays. Examples include time series forecasting (see among others <https://medium.com/mllearning-ai/transformer-implementation-for-time-series-forecasting-a9db2db5c820>), the artificial generation of music (<https://magenta.tensorflow.org/music-transformer>), or protein structure prediction (Jumper et al. 2021, Rives et al. 2021), a field said to revolutionize medical treatment. Each of these research threads is said to be lifted to completely new levels through the application of these powerful ANNs.

The GPT technology appears to represent a quantum leap on the way to General Artificial Intelligence (AGI), which even optimists would not have imagined in this form just a few years ago. The intensity with which further development is now being worked on worldwide<sup>25</sup> suggests that even the most stringent benchmark tests

---

<sup>24</sup> One of the exceptions concerned the initial indication by GPT-3.5 that it had only been trained with data up to the year 2021.

<sup>25</sup> For a probably non-exhaustive list, see, among others: [https://en.wikipedia.org/wiki/List\\_of\\_artificial\\_intelligence\\_projects](https://en.wikipedia.org/wiki/List_of_artificial_intelligence_projects)



will soon no longer be sufficient to reliably distinguish human from artificial intelligence. As already mentioned, some consider the Turing test to have been passed. The fact that the existing tools are increasingly being used to make successor tools even more efficient, and that even this is largely automated already, evokes associations with the prophesied Singularity, the point where AI exceeds human intelligence (Kurzweil 2006), and certainly casts a disconcerting light on statements like Sam Altman's according to which his company OpenAI is working on the "last great invention of humanity".

# 7 Epilogue: Data ethics

We have seen that the scope of data sciences is vast and that Machine learning comprises a wide range of interesting tools for automatically detecting regularities and structure in data. The research activities in this area are likely to be of great relevance for our immediate future. However, the methods developed in this process, in addition to their certainly positive aspects, harbor a number of dangers and risks for misuse. In concluding therefore, we should at least briefly address some ethical aspects that are inevitably associated with the topic.

Data collection and automated decision making about individuals is ubiquitous and pervasive today, making this a highly contentious topic. In particular with the introduction of ChatGPT by OpenAI in late 2022, the density of debates has skyrocketed. Countless discussions are led, from discipline-specific academic debates to international policy dialogs on AI governance and regulation. An armada of books and papers, both academic and activist, is being published, and the concerns and warnings are numerous. Debates on misuse of machine learning methods cover manipulated political campaigning, the spread of fake news, the emergence of filter bubbles and discriminating decisions in credit access or parole granting and manifold kinds of invasion in privacy. To exemplify the sorts of problems that can be expected from these methods and at the same time to point out possibilities for improvement and their limitations, we end this book with briefly discussing two sensible issues in this regard: the reintroduction of socially unwanted biases and discriminatory decision-making into governance procedures, and the efforts for aligning AI with human values.

Around 2017, a case made headlines when a commercial tool used by a governance agency for assessing a criminal defendant's likelihood of becoming a recidivist – a term used to describe criminals who re-offend – was tested for its underlying accuracy and whether the algorithm was biased against certain social groups<sup>26</sup>. The test found out that black defendants were far more likely than white defendants to be incorrectly judged to be at a higher risk of recidivism, while white defendants were

---

<sup>26</sup> <https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing>

more likely than black defendants to be incorrectly flagged as low risk<sup>27</sup>. The testers thus argued that the deployed algorithmic system produces a racial bias. The company which developed the system answered with a counter-report that showed that the system was treating black and white populations in a similar way, implying that no discrimination was taking place.

As it turned out, both tests were statistically sound, but were attacking the problem from different angles. The one was looking at the system's overall performance and found no difference between black and white subgroups (see left plot below). The other looked for whether comparable (low-risk and high-risk) classes across black and white subgroups were treated in the same way and found a difference (right plot below).<sup>28</sup>

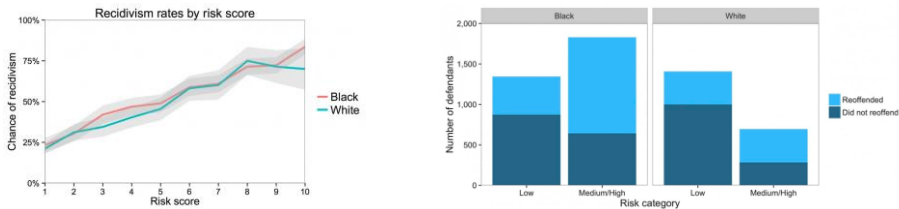


Fig. 94: Left: overall performance analysis. Right: Analysis of comparable classes, taken from Kittel (2019), p. 4.

Both approaches thus seemed correct, but their results incompatible. The question about which treatment is the fair one will have to be met on a social and political level. The *European General Data Protection Regulation* (GDPR) holds several regulations for such issues, aimed at bringing more transparency into the use of data and automated decision-making, including for instance a “right to explanation” of how one’s data is used.

Unfortunately, as we have seen, in many cases the “right to explanation” may be counteracted by the opacity of the “inner thinking” of machine learning algorithms, above all of contemporary ANNs. Although several attempts are currently being made to increase the transparency and interpretability of models through statistics

<sup>27</sup> <https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm>

<sup>28</sup> Note that mathematically it seems impossible to achieve fairness from both angles at the same time (Kleinberg/Mullainathan/Raghavan 2016). A condition for this would be that both classes have the same distributions across subgroups, which will be rarely met by reality.

and visualization (Ribeiro/Singh/Guestrin 2016, Alicioglu and Sun 2022) and to propose analytical frameworks for reducing the complexity of algorithmic decision making (Zarsky 2016, Doshi-Velez/Kim 2017, Krafft, Zweig, and König 2022), in all likelihood the problem will persist. The digital machine, as a powerful problem solver, is a complex system. It will not stop creating new problems on its own.

As in the example above, a crucial effort in mitigating possibilities of misuse, generating incorrect information or privacy invasions lies on aligning the workings of machine learning tools with behaviors and decisions that we, humans, would want to experience from other humans. The scope of these expectations is huge, since people and cultures are divers. But we still seem to have a general idea of an acceptable scope, which we like to address as *human values*. Since the introduction of GPTs, the alignment of these tools with human values has become a critical focus in the field of machine learning. The process involves several key steps to ensure that these models operate within ethical boundaries and produce outputs that are safe and reliable. Here is an overview in a nutshell: an emphasis, as in the example above, is of course on data curation and preprocessing, where diverse and representative datasets are compiled, and harmful content is filtered out to minimize biases inherent in the training data. If this is done, supervised fine-tuning and reinforcement learning from human feedback (RLHF) can be employed to guide a model's training process towards human acceptable behavior. In this, human annotators can provide labels and corrections, helping the model learn context and appropriateness. The iterative nature of RLHF, where models are rewarded for producing aligned responses, further refines their outputs.

Usually, to enhance these alignments Human-in-the-Loop (HITL) processes and regular ethical and bias audits are deployed. HITL involves continuous human oversight, allowing for real-time corrections and feedback that improve model performance. Ethical and bias audits, employing both automated tools and expert reviews, help identify and mitigate potential issues. Furthermore, advancements in explainability and transparency, such as attention mechanisms and transparency reports, contribute to the ethical deployment of GPT models. The goal are dynamic and context-aware tools, where continuous learning and contextual understanding enable them to adapt to new situations and user intents without extensive retraining, so that AI systems do not only excel in performance but also align with societal values and ethical standards. The road is still rocky though, and the goal is far from being reached in every regard. But the progress made so far is undeniably huge and is definitely set to make our world a different place. In order to cope with these new conditions, it seems essential to understand the basic principles and functionality

of the tools that will shape our world's future. This is what this book strives to support.

## References

- Adadi, Amina, and Mohammed Berrada. 2018. "Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)." *IEEE Access* 6:52138–60. <https://doi.org/10.1109/ACCESS.2018.2870052>.
- Alicioglu, Gulsum, and Bo Sun. 2022. "A Survey of Visual Analytics for Explainable Artificial Intelligence Methods." *Computers & Graphics* 102 (February):502–20. <https://doi.org/10.1016/j.cag.2021.09.002>.
- Alkaiissi, Hussam, Samy I. McFarlane, Hussam Alkaiissi, and Samy I. McFarlane. 2023. "Artificial Hallucinations in ChatGPT: Implications in Scientific Writing." *Cureus* 15 (2). <https://doi.org/10.7759/cureus.35179>.
- Andrychowicz, Marcin, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. 2018. "Hindsight Experience Replay." *arXiv:1707.01495 [Cs]*, February. <http://arxiv.org/abs/1707.01495>.
- Axelrod, Robert. 1984. *The Evolution of Cooperation*. New York: Basic Books.
- . 1987. "The Evolution of Strategies in the Iterated Prisoner's Dilemma." In *Genetic Algorithms and Simulated Annealing*. Research Notes in Artificial Intelligence. Morgan Kaufmann, Los Altos, CA.
- Ay, Nihat, and Daniel Polani. 2008. "Information Flows in Causal Networks." *Advances in Complex Systems* 11 (01): 17–41. <https://doi.org/10.1142/S0219525908001465>.
- Barredo Arrieta, Alejandro, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bannetot, Siham Tabik, Alberto Barbado, Salvador Garcia, et al. 2020. "Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI." *Information Fusion* 58 (June):82–115. <https://doi.org/10.1016/j.inffus.2019.12.012>.
- Bellman, Richard. 1957. *Dynamic Programming*. Princeton University Press.
- Biever, Celeste. 2023. "ChatGPT Broke the Turing Test — the Race Is on for New Ways to Assess AI." *Nature* 619 (7971): 686–89. <https://doi.org/10.1038/d41586-023-02361-7>.
- Bill, Jeremiah, Lance Champagne, Bruce Cox, and Trevor Bihl. 2021. "Meta-Heuristic Optimization Methods for Quaternion-Valued Neural Networks." *Mathematics* 9 (9): 938. <https://doi.org/10.3390/math9090938>.
- Buckley, Christopher L., Chang Sub Kim, Simon McGregor, and Anil K. Seth. 2017. "The Free Energy Principle for Action and Perception: A Mathematical Review." *arXiv:1705.09156 [q-Bio]*, May. <http://arxiv.org/abs/1705.09156>.
- Butlin, Patrick, Robert Long, Eric Elmoznino, Yoshua Bengio, Jonathan Birch, Axel Constant, George Deane, et al. 2023. "Consciousness in Artificial Intelligence: Insights from the Science of Consciousness." *arXiv*. <http://arxiv.org/abs/2308.08708>.
- Castelvecchi, Davide. 2016. "Can We Open the Black Box of AI?" *Nature* 538 (7623): 20–23. <https://doi.org/10.1038/538020a>.
- Cerezo, Sergio Hernandez, and Guillem Duran Ballester. 2018. "Fractal AI: A Fragile Theory of Intelligence." *arXiv:1803.05049 [Cs]*, March. <http://arxiv.org/abs/1803.05049>.

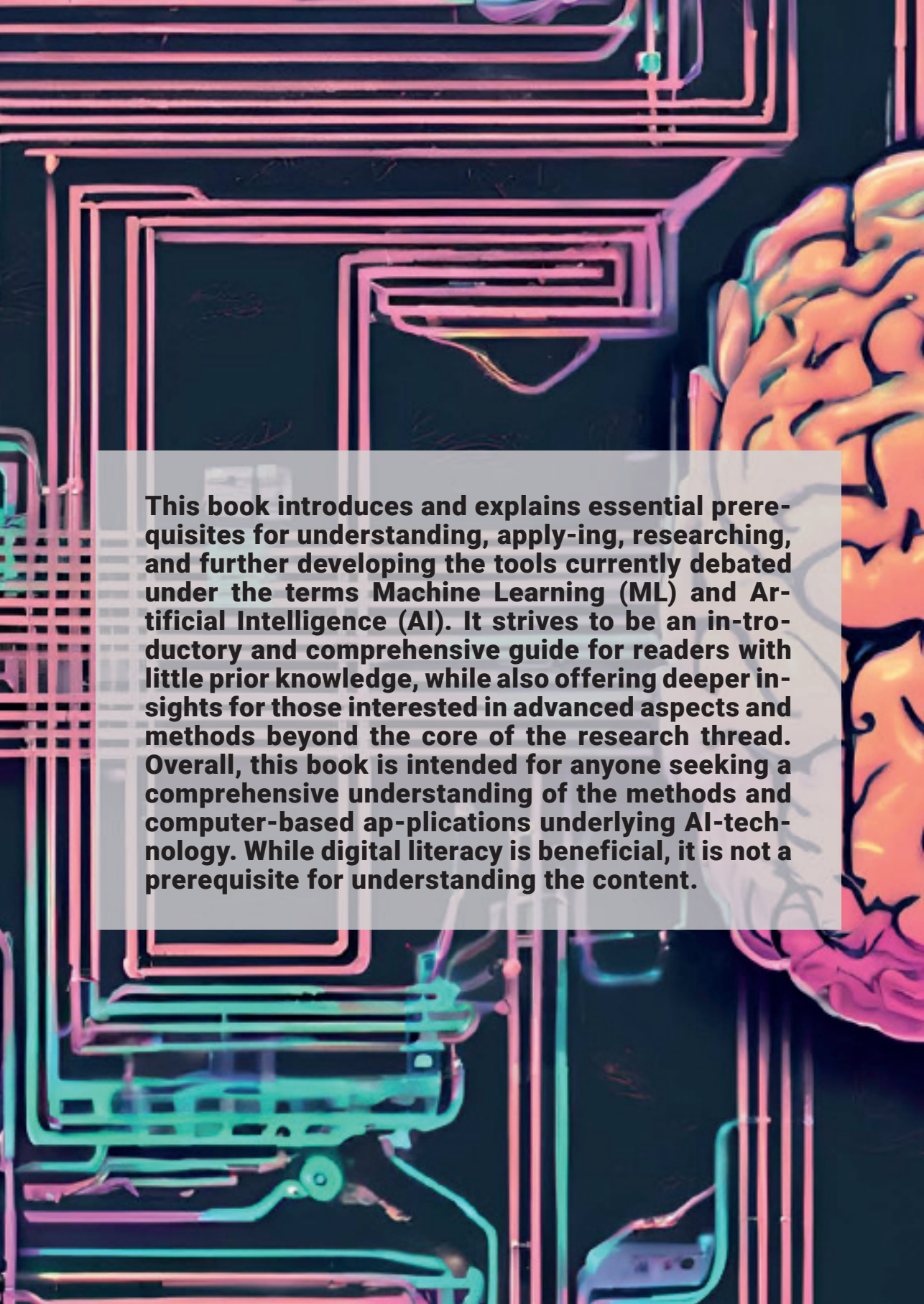
- Cerezo, Sergio Hernandez, Guillem Duran Ballester, and Spiros Baxevanakis. 2018. "Solving Atari Games Using Fractals And Entropy." *arXiv:1807.01081 [Cs]*, July. <http://arxiv.org/abs/1807.01081>.
- Dayan, Peter, Geoffrey E. Hinton, Radford M. Neal, and Richard S. Zemel. 1995. "The Helmholtz Machine." *Neural Computation* 7 (5): 889–904.
- Deerwester, Scott, and And Others. 1990. "Indexing by Latent Semantic Analysis." *Journal of the American Society for Information Science* 41 (6): 391–407.
- Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *arXiv:1810.04805 [Cs]*, May. <http://arxiv.org/abs/1810.04805>.
- Doshi-Velez, Finale, and Been Kim. 2017. "Towards A Rigorous Science of Interpretable Machine Learning." *arXiv:1702.08608 [Cs, Stat]*, February. <http://arxiv.org/abs/1702.08608>.
- Friston, Karl. 2009. "The Free-Energy Principle: A Rough Guide to the Brain?" *Trends in Cognitive Sciences* 13 (7): 293–301. <https://doi.org/10.1016/j.tics.2009.04.005>.
- . 2010. "The Free-Energy Principle: A Unified Brain Theory?" *Nature Reviews Neuroscience* 11 (2): 127–38. <https://doi.org/10.1038/nrn2787>.
- Friston, Karl, James Kilner, and Lee Harrison. 2006. "A Free Energy Principle for the Brain." *Journal of Physiology-Paris* 100 (1–3): 70–87. <https://doi.org/10.1016/j.jphysparis.2006.10.001>.
- Géron, Aurélien. 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques for Building Intelligent Systems*. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly UK Ltd.
- Haake, Daniel. 2022. "Theoretische Grundlagen des Maschinellen Lernens." In *Prognose von Wohnungseinbrüchen mit Hilfe von Machine-Learning-Algorithmen*, edited by Daniel Haake, 13–40. Wiesbaden: Springer Fachmedien. [https://doi.org/10.1007/978-3-658-37660-4\\_2](https://doi.org/10.1007/978-3-658-37660-4_2).
- Hawkins, Jeff. 2021. *A Thousand Brains: A New Theory of Intelligence*. New York: Basic Books.
- Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9 (8): 1735–80.
- Ibrahim, Zein Al Abidin, Marwa Saab, and Ihab Sbeity. 2019. "VideoToVecs: A New Video Representation Based on Deep Learning Techniques for Video Classification and Clustering." *SN Applied Sciences* 1 (6): 560. <https://doi.org/10.1007/s42452-019-0573-6>.
- Jones, Cameron R., and Benjamin K. Bergen. 2024. "Does GPT-4 Pass the Turing Test?" *arXiv*. <http://arxiv.org/abs/2310.20216>.
- Jumper, John, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, et al. 2021. "Highly Accurate Protein Structure Prediction with AlphaFold." *Nature* 596 (7873): 583–89. <https://doi.org/10.1038/s41586-021-03819-2>.
- Karpathy, Andrej, Justin Johnson, and Li Fei-Fei. 2015. "Visualizing and Understanding Recurrent Networks." *arXiv:1506.02078 [Cs]*, June. <http://arxiv.org/abs/1506.02078>.

- Kittel, Christopher. 2019. *Ensuring Fairness and Accountability in Opaque, Algorithmic Decision-Making Systems. An Evaluation of Consistency, Comprehensiveness and Comparability across Individual-Level and Group-Level Metrics*. MA. University of Graz.
- Kleinberg, Jon, Sendhil Mullainathan, and Manish Raghavan. 2016. "Inherent Trade-Offs in the Fair Determination of Risk Scores." *arXiv:1609.05807 [Cs, Stat]*, September. <http://arxiv.org/abs/1609.05807>.
- Klyubin, Alexander S., Daniel Polani, and Chrystopher L. Nehaniv. 2005. "All Else Being Equal Be Empowered." [https://doi.org/10.1007/11553090\\_75](https://doi.org/10.1007/11553090_75).
- Klyubin, Alexander S, and Daniel Polani. 2005. "Empowerment: A Universal Agent-Centric Measure of Control" *Evolutionary Computation*, 2005. The 2005 IEEE Congress (1): 128–35.
- Krafft, Tobias D., Katharina A. Zweig, and Pascal D. König. 2022. "How to Regulate Algorithmic Decision-Making: A Framework of Regulatory Requirements for Different Applications." *Regulation & Governance* 16 (1): 119–36. <https://doi.org/10.1111/rego.12369>.
- Kurzweil, Ray. 2006. *The Singularity Is Near: When Humans Transcend Biology*. New York: Penguin Books.
- Levenshtein, Vladimir I. 1966. "Binary Codes Capable of Correcting Deletions, Insertions and Reversals." *Soviet Physics Doklady* 10 (February):707.
- Liu, Bing. 2010. "Sentiment Analysis and Subjectivity." In *Handbook of Natural Language Processing, Second Edition*. Taylor and Francis Group, Boca.
- Lohn, Jason D., Derek S. Linden, Gregory S. Hornby, and William F. Kraus. 2004. "Evolutionary Design of an X-Band Antenna for NASA's Space Technology 5 Mission." In *IEEE Antennas and Propagation Society Symposium, 2004.*, 3:2313-2316 Vol.3. <https://doi.org/10.1109/APS.2004.1331834>.
- Maaten, Laurens van der, and Geoffrey Hinton. 2008. "Visualizing Data Using T-SNE." *Journal of Machine Learning Research* 9 (Nov): 2579–2605.
- Mai, Xiaodong. 2019. "Efficient Multimedia Information Mining Framework Based on Deep Learning and Self-Organizing Model." *Multimedia Tools and Applications* 78 (4): 4605–22. <https://doi.org/10.1007/s11042-018-6406-6>.
- Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." *arXiv:1301.3781 [Cs]*, January. <http://arxiv.org/abs/1301.3781>.
- Minsky, Marvin Lee, and Seymour Papert. 1972. *Perceptrons: An Introduction to Computational Geometry*. Mit Press.
- Mitchell, Melanie. 2011. *Complexity: A Guided Tour*. 1 edition. Oxford England; New York: Oxford University Press.
- Mitchell, Melanie, and David C. Krakauer. 2023. "The Debate Over Understanding in AI's Large Language Models." *arXiv*. <http://arxiv.org/abs/2210.13966>.
- Ng, Andrew Y., and Stuart Russell. 2000. "Algorithms for Inverse Reinforcement Learning." In *In Proc. 17th International Conf. on Machine Learning*, 663–70. Morgan Kaufmann.



- Pedersen, Jan T, and John Moul. 1996. "Genetic Algorithms for Protein Structure Prediction." *Current Opinion in Structural Biology* 6 (2): 227–31. [https://doi.org/10.1016/S0959-440X\(96\)80079-0](https://doi.org/10.1016/S0959-440X(96)80079-0).
- Peters, Matthew E., Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. "Deep Contextualized Word Representations." *arXiv:1802.05365 [Cs]*, March. <http://arxiv.org/abs/1802.05365>.
- Powers, William T. 2005. *Behavior: The Control Of Perception*. Revised, Expanded. New Canaan, Conn: Benchmark Pubns Inc.
- Rehurek, Radim, and Petr Sojka. 2010. "Software Framework for Topic Modelling with Large Corpora." In *Proceedings of LREC 2010 Workshop New Challenges for NLP*.
- Reinsberger, Kathrin, Thomas Brudermann, Stefanie Hatzl, Eva Fleiß, and Alfred Posch. 2015. "Photovoltaic Diffusion from the Bottom-up: Analytical Investigation of Critical Factors." *Applied Energy* 159 (C): 178–87.
- Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. 2016. "'Why Should I Trust You?': Explaining the Predictions of Any Classifier." *arXiv:1602.04938 [Cs, Stat]*, February. <http://arxiv.org/abs/1602.04938>.
- Rives, Alexander, Joshua Meier, Tom Sercu, Siddharth Goyal, Zeming Lin, Jason Liu, Demi Guo, et al. 2021. "Biological Structure and Function Emerge from Scaling Unsupervised Learning to 250 Million Protein Sequences." *Proceedings of the National Academy of Sciences* 118 (15). <https://doi.org/10.1073/pnas.2016239118>.
- Rosen, Judith. 2009. "Robert Rosen's Anticipatory Systems Theory: The Art and Science of Thinking Ahead." In *Proceedings of the 53rd Annual Meeting of the ISSS-2009, Brisbane, Australia*. Vol. 1. <http://journals.issis.org/index.php/proceedings53rd/article/view/1249>.
- Salge, Christoph, Cornelius Glackin, and Daniel Polani. 2013. "Empowerment -- an Introduction." *arXiv:1310.1863 [Nlin]*, October. <http://arxiv.org/abs/1310.1863>.
- . 2014. "Changing the Environment Based on Empowerment as Intrinsic Motivation." *Entropy* 16 (5): 2789–2819. <https://doi.org/10.3390/e16052789>.
- Schober, Andreas, Christopher Kittel, Rupert J. Baumgartner, and Manfred Füllsack. 2018. "Identifying Dominant Topics Appearing in the Journal of Cleaner Production." *Journal of Cleaner Production* 190 (July):160–68. <https://doi.org/10.1016/j.jclepro.2018.04.124>.
- Schober, Andreas, Christopher Kittel, and Manfred Füllsack. 2016. "Die Digitale Rationalisierung im Spiegel von Stellenanzeigen. Automatisierte Textanalyse zu Annahmen des „Task-Based Approach“." *SOZIOLOGIE IN ÖSTERREICH - INTERNATIONALE VERFLECHTUNGEN* (Hg. Helmut Staubmann) 0 (0). <https://webapp.uibk.ac.at/ojs2/index.php/oegs-publikation/article/view/19>.
- Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, et al. 2020. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model." *Nature* 588 (7839): 604–9. <https://doi.org/10.1038/s41586-020-03051-4>.
- Shannon, Claude E. 1948. "A Mathematical Theory of Communication." *The Bell System Technical Journal* 27 (3): 379–423. <https://doi.org/10.1002/j.1538-7305.1948.tb01338.x>.

- Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, et al. 2018. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play." *Science* 362 (6419): 1140–44. <https://doi.org/10.1126/science.aar6404>.
- Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." *arXiv:1706.03762 [Cs]*, December. <http://arxiv.org/abs/1706.03762>.
- Vinyals, Oriol, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2014. "Show and Tell: A Neural Image Caption Generator." *arXiv:1411.4555 [Cs]*, November. <http://arxiv.org/abs/1411.4555>.
- Wissner-Gross, Alexander D., and Cameron E. Freer. 2013. "Causal Entropic Forces." *Physical Review Letters* 110 (16). <https://doi.org/10.1103/PhysRevLett.110.168702>.
- Zarsky, Tal. 2016. "The Trouble with Algorithmic Decisions: An Analytic Road Map to Examine Efficiency and Fairness in Automated and Opaque Decision Making." *Science, Technology, & Human Values* 41 (1): 118–32. <https://doi.org/10.1177/0162243915605575>.



**This book introduces and explains essential prerequisites for understanding, applying, researching, and further developing the tools currently debated under the terms Machine Learning (ML) and Artificial Intelligence (AI). It strives to be an introductory and comprehensive guide for readers with little prior knowledge, while also offering deeper insights for those interested in advanced aspects and methods beyond the core of the research thread. Overall, this book is intended for anyone seeking a comprehensive understanding of the methods and computer-based applications underlying AI-technology. While digital literacy is beneficial, it is not a prerequisite for understanding the content.**