Data in Systems Sciences

Author: Manfred Füllsack


Please do not quote, as this is work in progress

Contents

Introduction........................................................................................................................................... 3
1. The use of models in nature – Anticipatory systems ........................................................................ 3
1.1. The Free energy principle................................................................................................................ 4
1.2. Empowerment.................................................................................................................................. 6
1.3. Causal Entropy................................................................................................................................. 9
2.1. Analytical modeling – Linear regression....................................................................................... 12
2.2. Linear and non-linear data discrimination..................................................................................... 13
2.3. Data preparation ............................................................................................................................ 15
2.4. Supervised learning and Information gain ................................................................................... 16
2.5. Applying a Decision Tree Classifier ............................................................................................. 17
2.6. Obtaining and evaluating results.................................................................................................... 19
2.7. Improving results............................................................................................................................ 20
2.8. The bias-variance tradeoff, generalization and overfitting............................................................. 21
2.9. Data augmentation......................................................................................................................... 22
2.10. Feature reduction.......................................................................................................................... 23
3. Some common tools........................................................................................................................... 25
3.1. Support Vector Machine ............................................................................................................... 25
3.2. k-Nearest-Neighbor ..................................................................................................................... 26
3.3. Naïve Bayes ................................................................................................................................ 27
4. Artificial Neural Networks............................................................................................................... 28
4.1. The Perceptron .............................................................................................................................. 29
4.2. Back propagation Neural Networks ............................................................................................. 31
4.3. Gradient Descent............................................................................................................................ 32
4.4. The MNIST-digits-example .......................................................................................................... 34
4.5. Neural Network types .................................................................................................................. 36
4.5.1. Long Short Term Memory (LSTM) Neural Networks............................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.2.</td>
<td>Recurrent Neural Networks</td>
<td>37</td>
</tr>
<tr>
<td>4.5.3.</td>
<td>Convolutional Neural Networks (CNNs)</td>
<td>39</td>
</tr>
<tr>
<td>4.6.</td>
<td>Unsupervised Learners</td>
<td>40</td>
</tr>
<tr>
<td>4.6.1.</td>
<td>Self-organizing Maps (SOMs, aka Kohonen-networks)</td>
<td>40</td>
</tr>
<tr>
<td>4.6.2.</td>
<td>Autoencoders</td>
<td>42</td>
</tr>
<tr>
<td>5.</td>
<td>Text mining</td>
<td>43</td>
</tr>
<tr>
<td>5.1.</td>
<td>Mining in unstructured and analog texts</td>
<td>46</td>
</tr>
<tr>
<td>5.2.</td>
<td>Latent Semantic Analysis</td>
<td>47</td>
</tr>
<tr>
<td>5.3.</td>
<td>Assessing similarities by vector representation</td>
<td>50</td>
</tr>
<tr>
<td>5.4.</td>
<td>Sentiment analysis</td>
<td>59</td>
</tr>
<tr>
<td>6.</td>
<td>Epilogue: Data ethics - the Fairness issue</td>
<td>60</td>
</tr>
<tr>
<td>7.</td>
<td>References</td>
<td>62</td>
</tr>
</tbody>
</table>
Introduction

This script contains material taught in the course “Data in Systems Sciences” in the study program USW at the University of Graz. Eventually, it is meant to become a chapter of a book (written together with Dr. Georg Jäger), which will cover the entire content of the systems sciences lectures in this study program.

This last part of the book provides an overview of algorithmic models used for data analysis and machine learning. Conceived for exploring patterns in large data sets, an interesting aspect of these tools is that in consequence of their efficiency and applicability, research and development has turned them into complex systems themselves. They are thus not just methods for data exploration but highly interesting objects for scientific investigations themselves. Contemporary Artificial Neural Networks for example, are said to have reached a degree of complexity that impedes complete comprehension (Castelvecchi 2016), implying that the use and further development of these tools may bring up some difficult questions in the near future (Ribeiro/Singh/Guestrin 2016). It is therefore of prime importance to have an at least basic understanding of their working principles and their capabilities. This is what this chapter strives to convey. Note however, that the following account is not much more than an introduction to a highly dynamic and rapidly developing field of complex systems research. You will have to upgrade and update your knowledge from additional sources.

As insinuated above, computer-based Machine Learning tools and in general the methods of Artificial Intelligence can be seen as a sort of algorithmic models. We start this introductory chapter 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 activities in the *virtual space* of its model, thus being able to *anticipate* auspicious options.

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\(^1\) 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.

### 1.1. The Free energy principle

The use of models to maintain homeostasis is currently widely discussed in the domain of artificial systems. A famous conception of model use with interesting implications for machine learning is the **free energy principle** as Karl Friston (2006, 2009, 2010) suggested it for the functioning of brains, with brains being the 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

---

\(^1\) 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: [http://systems-sciences.uni-graz.at/etextbook/sw1/evolution.html](http://systems-sciences.uni-graz.at/etextbook/sw1/evolution.html)
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.

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", resp. "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 $2^l = N$, with $l$ indicating the number of observations and $N$ indicating the number of letters in the alphabet. With $l$ being the binary logarithm, this allows expressing the number of needed observations as $l = \log_2 N$. As said, in alphabets with equally likely letters the occurrence probability $p$ of a letter would be $\frac{1}{N}$, implying $l = \log_2 \frac{1}{p}$, with $l$ 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 $[p(s|m)] = \lim_{T \to \infty} \frac{1}{T} \int_0^T -\ln p(s|m) dt$, 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.

1.2. 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.

6
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 agents 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 to the left 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.
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 to the right 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.

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.

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).

\[ n = 13 \]

\[ n = 20 \]

**1.3. 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** (left 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 \( t + 1 \).
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.

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.

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. 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.


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.

<table>
<thead>
<tr>
<th>education</th>
<th>environmental awareness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

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 the image to the right 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.

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 1 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 \( y = m \times x + k \), 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 stats-package of scipy (www.scipy.org), which uses the so called least square method. The code to the right draws the data, calculates a linear regression and draws the line through it, as shown in the image above.

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 – a fact that 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=KfNPAXplLbc), 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 to the left 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, 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.

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 to the right.

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 the plot below. Note however, that in principal 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 the 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.

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 amount 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, it is just the number of parameters needed for defining a valid model that changes 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 in principal. 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 than was translated into a Likert-scale with nominal values ranging from one to five. An excerpt of this 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.

<table>
<thead>
<tr>
<th>target</th>
<th>env. protection</th>
<th>financial asset</th>
<th>believe in techn. progress</th>
<th>energy aut. of community</th>
<th>environm. awareness</th>
<th>appearance of home</th>
<th>pers. social capital</th>
<th>social capital of community</th>
<th>appearance of community</th>
<th>comm envir protect</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2</td>
<td>1.656667</td>
<td>1.656667</td>
<td>2.25</td>
<td>4.000000</td>
<td>2.000000</td>
<td>2.333333</td>
<td>3.0</td>
<td>NaN</td>
<td>2.333333</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>1.656667</td>
<td>2.000000</td>
<td>1.00</td>
<td>4.333333</td>
<td>3.333333</td>
<td>3.888889</td>
<td>2.0</td>
<td>NaN</td>
<td>3.566667</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>-999.000000</td>
<td>-999.000000</td>
<td>2.00</td>
<td>-999.000000</td>
<td>3.000000</td>
<td>3.333333</td>
<td>-999.0</td>
<td>NaN</td>
<td>3.000000</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.50</td>
<td>1.000000</td>
<td>1.000000</td>
<td>2.555556</td>
<td>1.0</td>
<td>NaN</td>
<td>3.000000</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>1.656667</td>
<td>1.333333</td>
<td>2.00</td>
<td>2.666667</td>
<td>-999.000000</td>
<td>3.000000</td>
<td>1.5</td>
<td>NaN</td>
<td>-999.000000</td>
</tr>
</tbody>
</table>

The data set is small and could be scrutinized easily 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 studied.

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. 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 preprocessing. The code below 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.

2.4. Supervised learning and Information gain

In the case of this survey, people were asked about their attitudes towards photovoltaics 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, which in this case is called an instance of 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”. 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.

These indices can be used to rank the features of a data set and to construct a so called decision tree, as shown on the left. 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 to the right, 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.

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 if 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 to the right.

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 specialization. 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. This 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.

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-tool of the scikit-learn module 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.
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:

```python
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)
```

### 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 safe the result in a file called clf_prediction. This can then be evaluated with a couple of tools provided by scikit-learn. The code snippet to the right yields the following output.

```python
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

clf_prediction = tree_clf.predict(X_test)
predictions = accuracy_score(y_test, clf_prediction)
print('Accuracy: {:.2f}'.format(predictions))
predictions = classification_report(y_test, clf_prediction)
print('Classification report:
')
```

**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 its 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, 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.

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

<table>
<thead>
<tr>
<th></th>
<th>Predicted non-adopters</th>
<th>Predicted adopters</th>
</tr>
</thead>
<tbody>
<tr>
<td>True non-adopters</td>
<td>113</td>
<td>43</td>
</tr>
<tr>
<td>True adopters</td>
<td>47</td>
<td>26</td>
</tr>
</tbody>
</table>

\[
\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \quad \quad \quad \quad \quad \text{Precision} = \frac{TP}{TP + FP}
\]

\[
\text{Recall} = \frac{TP}{TP + FN} \quad \quad \quad \quad \quad f1\text{-}score = \frac{2TP}{2TP + FP + FN}
\]

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.

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 above, yields a slightly better result than the decision tree trained on a single 75%-training set (see the table to the right).

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 (see the yellow line in the image to the right). Bias errors result in so called underfitting, indicating that the generated model does not mirror all characteristics of the data, for example that the true separation line is curved (black line in the right image).
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 and not for any other data (the green curve in the image above). As mentioned earlier, this reduces or thwarts the generalizability of the model. The model reproduces just the data at hand.

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, and 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 their result as benchmark and gradually step up with the complexity of your tool to see if the additional complexity is worth it. Increasing a models complexity will increase its variance and reduce its bias. If you should happen to run into two models with comparable performance, than, 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 further below, 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 they rather will consider the broad characteristics 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). 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.
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 on page 17 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 the next page.

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

```python
features = X.columns.tolist()
importances = tree.feature_importances_
indices = np.argsort(importances)[::-1]
List = []
for f in range(X.shape[1]):
    List.append([features[indices[f]]])

# plot feature importances
fig, ax = plt.subplots(figsize=(10,4))
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices], color="r", align="center")
plt.xticks(range(X.shape[1]), List, rotation='50', ha='right')
plt.xlim([-1, X.shape[1]])
```

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 features to consider is **Principal Component Analysis** (PCA). This is an **unsupervised** method using 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. Some common 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 currently 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 Naive 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 pictures may explain this principle.

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.

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 to the left shows an SVM-classification with hinge-loss.

As many others of the 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 to the right 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 below.

```
import numpy as np
from sklearn.svm import SVR
import matplotlib.pyplot as plt
%matplotlib inline

# generate data
X = np.sort(5 * np.random.rand(40, 1), axis=0)
y = np.cos(X).ravel()
# add noise to targets
y[::5] += 5 * (0.5 - np.random.rand(8))

# SVM regression
svr_rbf = SVR(kernel='rbf', C=1e3, gamma=0.1)
y_rbf = svr_rbf.fit(X, y).predict(X)

plt.scatter(X, y, color='red', label='data')
plt.plot(X, y_rbf, 'b', label='model')
plt.xlabel('data')
plt.ylabel('target')
plt.xlim(0, 5)
plt.title('Support Vector Regression')
plt.legend()
plt.grid()
```

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).

### 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 $p(AB) = p(A) \ast p(B \mid A)$ or alternatively $p(AB) = p(B) \ast p(A \mid B)$. This implies that $p(A) \ast p(B \mid A) = p(B) \ast p(A \mid B)$ and so, if both sides are divided by $p(A)$, one gets $p(B \mid A) = \frac{p(A \mid B) \ast p(B)}{p(A)}$, 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 an evidence, for example the high environmental awareness of this person. Renaming Bayes’ rule gives $(H \mid E) = \frac{p(E \mid H) \ast p(H)}{p(E)}$. 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

\[
(C = c \mid E) = \frac{p(E \mid C = c) \cdot p(C = c)}{p(E)}.
\]

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). \( p(C = c \mid E) \) 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 \( p(E \mid C = c) \), 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

\[
p(c \mid E) = \frac{p(e_1 \mid c) \cdot p(e_2 \mid c) \cdots p(e_n \mid c) \cdot p(c)}{p(E)}.
\]

Each of the \( p(e_i \mid c) \) 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.

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 to the right shows a classification of the PV-adoption data, done with the Naive Bayes-algorithm as provided by scikit.learn (GaussianNB).

### 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) and Artificial Life (AL).

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 excitations 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 the image to the right). 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 between −1 and +1. Remember the network chapter (→), where we showed that the connections in a network can be represented with the entries in a matrix. The transmission of excitement in ANNs thus basically is done in matrix-calculations.

In order to comprehend the basic working principle of ANNs in more details, consider the task of learning the correct outputs of the logical operation of 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 to the left). Our machine now has to learn to provide these outputs, when confronted with the inputs $I_1$ and $I_2$.

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 $net = w_1 \cdot I_1 + w_2 \cdot I_2$ 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 case.
- 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. This is the weight with the value that has changed in comparison with the previous case. It will be increased by the learning rate of 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 $I_2$, is increased by the learning rate 0.2 to $w_1 = 0.3$. The net value now is $\text{round}(0.3 \times 1 + 0.5 \times 0) = 0$, 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 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 $I_2$, is increased by the learning rate of 0.2 to $w_1 = 0.5$. The net value now, with $\text{round}(0.5 \times 0.5 \times 1 + 0)$, 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 Neural Networks

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\(^3\), caused temporary irritations in the development of artificial neural networks. In a much-discussed book on Perceptrons, Marvin Minsky and Seymore 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.

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). In such MLPs 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.

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 Perceptrons, 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: \(E_O = (O - A) \cdot A \cdot (1 - A)\). 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

\(^3\) In contrast to the inclusive OR, the XOR-operator yields the output 0 for the inputs 1 and 1
complement: \( E_v = \sum_{j=1}^{n} (E_{0,j} * W_j) * A_j * (1 - A_j) \). The corresponding weights \( w_j \) then are adjusted in respect to the learning rate \( d \) multiplied with the error of the preceding neuron \( E_0 \) (or \( E_v \)) and the activation of the neuron to which the connection leads back: \( w_i(t+1) = w_i(t) + d * E * A \).

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 (for example of the form \( s = \frac{1}{1 + e^{-x}} \)), 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 weighting 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, 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 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 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.

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. 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.

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.

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

\[
\mathbf{a}_{n+1} = \mathbf{a}_n - \gamma \nabla f(\mathbf{a}_n),
\]

with \( \mathbf{a} \) being a model parameter that is slightly changed in each step, and \( \gamma \) expressing the stride length (or error-decreasing-rate, resp. 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 on page 17. 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.\(^4\)

### 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/](http://yann.lecun.com/exdb/mnist/)). The database contains a set of 50.000 digit images for training and a separate 10.000 image-set for validation purposes. Six examples from this data set are shown to the left. As can be seen, the scanned hand-writings have a rather low resolution of 28 x 28 = 784 pixels, which in most cases suffices to clearly distinguish the digits. In some cases however – see the examples to the right –, even humans will have difficulties to distinguish the digits.

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 handwriting 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.

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

\(^4\) 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.
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.

```python
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, **param)
mlp.fit(X, y)

# show the loss of error
plt.plot(mlp.loss_curve_)
plt.title('Error loss curve')

# test on test_data and show performance
y_test = test_data[1]
mlp_prediction = mlp.predict(test_data[0])

print "Classification report:
", classification_report(y_test, mlp_prediction)
print "Confusion matrix:
", confusion_matrix(y_test, mlp_prediction)
```

Error loss curve, classification report and confusion matrix are shown below. 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.

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.
4.5. Neural Network types

In the last years, research in machine learning and artificial intelligence emitted a steady stream of ANN-types specialized on various tasks. The following provides a brief, condensed overview.

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. 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 scene 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. With regard to this last option, 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 Tolstoi consisting of 3.258.246 characters of English text (see: http://karpathy.github.io/2015/05/21/rnn-effectiveness/). 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-iafhatawiaihrdemot lytdws e ,ftfi, astai f ogoh eoa e rrranbyne ‘nhthnee e plia tklrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns ing

After 300 iterations:

"Tmont thithey" fomesscerliund Keushey. Thom here sheulke, anmerenith ol sivh l lalterthend Bleipile shuwy 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 otf 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 ofter.

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 oftened 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/) 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).

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, 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.

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. Those 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.

---

5 https://cdn2.vox-cdn.com/uploads/chorus_asset/file/6577761/Google_-_Magenta_music_sample.0.mp3
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, 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.

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 (see image to the right). The rectangles then are rolled with a certain stride length $s$ over the information of the lower layer. 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. The image to the left shows the result of the interaction of two such filters (taken from Géron 2017).

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).

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

Besides the group of supervised learning algorithms that we discussed so far, there is also a range of unsupervised methods, which are able to find structures in data by themselves, without being fitted to any 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.
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. 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 (left picture above). 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 \( redu = \exp(-\frac{d^2}{2l}) \), 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 \( redu2 = i \times \exp(-\frac{t}{n \log r}) \), 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.
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 however 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 the illustration below), 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.

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 image above). 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. The image to the right 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.
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 differences 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.

![Synthetic Handwritten Digits](image)

Generated with code provided in (Géron 2017)

### 5. 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 usually 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.
A first data retrieval step loads the complete collection of papers from an open access journal for ecological conservation from the internet. The gathered text corpus comprises 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. A Python-script, essentially drawing on the **Natural Language Toolkit** (Python’s NLTK modul, [www.nltk.org](http://www.nltk.org)), is used to transform these papers into a *txt*-file 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 is further reduced. Finally, after converting the texts into NLTK’s specific text format, it contains 1,798,948 words.

Now a first analytical step can be applied, which creates a frequency distribution of the 30 most frequent words in the journal (shown above).

![Frequency distribution of 30 most frequent words](chart1.png)

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

![Frequency distribution of 30 most frequent two-word combinations](chart2.png)

The analytical tools in 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 on the right – in this case for the word “species”. (Have in mind that stop words were removed in the example).
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.

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).
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 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 right half of the image).

Note that after extraction several words show incorrect spellings, due to the quality of the original newspaper printing. Many words (e.g. “Laundiaedchen”, which should read “Landmädchen” (“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.

$$L_{ratio}(a, b) = \frac{\Sigma(len(a), len(b)) - L_{dist}}{\Sigma(len(a), len(b))}$$
To give an example, the string lengths of the term “productive” equals 10 and the string lengths of the German term “produkativ”, 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 extent 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 is based on the assumption 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 found 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 (like in the left image below), but can have multiple meanings (right image).

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 just 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', 'resource', 'consum', 'econom', 'labor', 'revenu', 'gdp', 'predict', 'futur', 'growth', 'gain', 'accomplish', 'energi', 'produc', 'food']

The table above shows the 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:

\[ FIDF = \frac{N_{i,j}}{N_{*,j}} \cdot \log \frac{D}{D_i} \]

- \( N_{i,j} \) = 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 \( i \)).

In this way, both, words that are concentrated in certain documents as well as words that only appear in a few documents, are emphasized – the first ones by the \( \frac{N_{i,j}}{N_{*,j}} \) ratio and the second ones by the \( \log \frac{D}{D_i} \) term.

In the third step, an algorithm called Singular Value Decomposition (SVD) is used to generate a reduced dimensional representation of the TFIDF-matrix, which emphasizes the strongest relationships of words and documents and discards the unneeded noise. In other words, the SVD makes the best possible reconstruction of the matrix with the least possible information. To do this, it throws out values which do not add information, and emphasizes strong patterns and trends, which do. However, SVD is context-dependent. It needs a decision about how many dimensions or “concepts” to use when approximating the matrix. Unfortunately, there is no exact method and it can be tricky to figure out how many dimensions or “concepts” are needed to approximate the matrix. If the corpus of documents is large, typical dimensions numbers range from 100 to 500. In small cases, like in this example case, just a few dimensions will do.

By keeping only the \( k \)-largest singular values and their associated vectors, SVD decomposes the matrix into the product of three other matrices \(-T_k * S_k * P_k\) – where \( S_k \) contains the singular- or eigenvalues of the matrix. The rows in the \( T_k \) matrix represent the term vectors and the rows in the \( P_k \) matrix represent the document vectors.

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.

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.

5.3. Assessing similarities by vector representation

The starting point for an alternative kind of automated text comparison is the possibility of assessing similarities mathematically. 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. 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.

<table>
<thead>
<tr>
<th>Country</th>
<th>Population</th>
<th>GDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>329093110</td>
<td>21410230</td>
</tr>
<tr>
<td>China</td>
<td>1420062022</td>
<td>15543710</td>
</tr>
<tr>
<td>Japan</td>
<td>128854745</td>
<td>5392220</td>
</tr>
<tr>
<td>Germany</td>
<td>82438639</td>
<td>4419800</td>
</tr>
<tr>
<td>India</td>
<td>1368737513</td>
<td>3155250</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>60959616</td>
<td>3022580</td>
</tr>
<tr>
<td>Italy</td>
<td>59216625</td>
<td>2291460</td>
</tr>
</tbody>
</table>

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 (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.

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° between them –, then the cosine is 1. If they are orthogonal to each other (90°), the cosine equals 0, and if they are exactly opposed – with an angle of 180° - then the cosine is -1, as can be easily derived from the unit circle as shown to the right.

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 modul, which also offers a function for the cosine similarity. Executing the code will yield a $CS = 0.24182053$ for USA and India and a $CS = 0.98916326$ for Japan and Germany.

There are several more methods to consider the similarity of vectors. 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 what we just did with a couple of countries.

5.3.1. 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” for instance 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:

```
# 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
```

---

7 see e.g. [http://dataaspirant.com/2015/04/11/five-most-popular-similarity-measures-implementation-in-python/](http://dataaspirant.com/2015/04/11/five-most-popular-similarity-measures-implementation-in-python/)
“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:

<table>
<thead>
<tr>
<th>START</th>
<th>was</th>
<th>it</th>
<th>the</th>
<th>was_best</th>
<th>best</th>
<th>times</th>
<th>of</th>
<th>best_times</th>
<th>of</th>
<th>it</th>
<th>times</th>
<th>was</th>
<th>worst</th>
<th>worst_times</th>
<th>of</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>was</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>the</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>best</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>times</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>worst</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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, this small toy example misleads our attempt to derive meaning from it. Therefore, in realistic settings word vectors never are generated from a single sentence. Rather, 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 xxx), such word embeddings nowadays often are produced with so-called Word2Vec models, as developed by Tomas Mikolov at Google in 2013 (Mikolov et al. 2013)8. These models are shallow, two-layer neural networks 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 instance9 the Global Vectors for Word Representation (GloVe)10 by the

---

8 See also https://code.google.com/archive/p/word2vec/
9 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
10 https://nlp.stanford.edu/projects/glove/
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).

5.3.2. Examples

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:

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

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'.

```python
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)))
```

The query yields the following cosine similarities with 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

11 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.
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 code at the right loads the 1987 Brundtland-report of Our Common Future\(^\text{12}\). It tokenizes it and investigates it for the most similar words to “sustainability”. As a 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\(^\text{13}\) 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"

### 5.3.2.1. Vector arithmetics

Interestingly, we can perform vector arithmetics with these word vectors, and in some cases, they seem to yield quite reasonable results. Adding for example the vectors of “livable” and “planet” together yields 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"

### 5.3.2.2. Sentence similarity

From word vectors sentence vectors can be generate, 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 Brundtland-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.

5.3.3. 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. 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). The found term similarities were plotted on a two-dimensional graph (see p.50).

As in this example, the following will consider each of these definitions (consisting most of the time of just one sentence) as a (small) document. We will apply these documents to vectorization with a gensim Doc2Vec-model. The definitions, taken from various economics textbooks, are:

1. "A measure of the efficiency of a person, machine, factory, system, etc., in converting inputs into useful outputs."
2. "Productivity is computed by dividing average output per period by the total costs incurred or resources consumed in that period."
3. "Productivity is a critical determinant of cost efficiency."
4. "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."
5. "Productivity is measured and tracked by many economists as a clue for predicting future levels of GDP growth."
6. "Productivity gains are vital to the economy because they allow us to accomplish more with less."
7. "Productivity is the ratio of output to inputs in production; it is an average measure of the efficiency of production."
8. "The rate at which radiant energy is used by producers to form organic substances as food for consumers."
9. "Productivity isn't everything, but in the long run it is almost everything."
10. "Productivity is commonly defined as a ratio between the output volume and the volume of inputs."

---

Note that each of these definitions could be a paragraph, a scientific paper or even a complete book instead.
Let’s see which of these productivity definitions are most similar. For this, we have to first tokenize them and bring them into a particular form, called TaggedDocument in genism, which looks like the following for the first definition:

```python
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 (for more details see section 4 on artificial neural networks). 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 today believe that the method produces better results than LSA and related methods. The code shown below defines and trains such a neural network.

```
# 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’ (PV-DM)
# dm =0 means ‘distributed bag of words’ (PV-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')```

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.

1.) productivity is a critical determinant of cost efficiency 0.758
2.) productivity is the ratio of output to inputs in production; it i
   s an average measure of the efficiency of production 0.58

---

15 Note that the first definition is tagged with the number zero [‘0’], the second with one [‘1’], and so on.
productivity is commonly defined as a ratio between the output volume and the volume of inputs 0.526
productivity isn't 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 to the left 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.

1.) Productivity isn’t everything, but in the long run it is almost everything. 0.63
2.) Productivity gains are vital to the economy because they allow us to accomplish more with less. 0.61
3.) Productivity is a critical determinant of cost efficiency 0.6
4.) The rate at which radiant energy is used by producers to form organic substances as food for consumers 0.55
5.) 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 t-distributed stochastic neighbor embedding (Maaten/Hinton 2008) (tSNE, included in sklearn.manifold) 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.

5.3.4. Beyond textual contexts

There is yet another interesting aspect of this type of methodology that transcends the field of text mining. Recently, a couple of internet companies like Yahoo, Spotify or AirBNB have started to apply the Word2Vec approach to recommender systems and advertising. 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 –, than 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 (see links in Fn.12).

If you reconsider the approach under this perspective, vectorization (i.e. the Word2Vec method) appears to have a much broader scope than text analysis 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.

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. The opinions all 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 for it.
- 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 \((o_j, f_{jk}, o_{oi}, h_i, t_l)\) 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:

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.

6. Epilogue: Data ethics - the Fairness issue

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 (for a more detailed overview see a.o.: Floridi/Taddeo 2016).

Data collection and automated decision making about individuals is ubiquitous and pervasive today, making this a highly contentious topic. Recent high-profile public 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 working principles of these methods once more and at the same time to point out possibilities for improvement and their limits, we end this chapter with briefly
discussing a particularly sensible issue in this regard: the reintroduction of socially unwanted biases and discriminatory decision-making into governance procedures.

Recently, 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. 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 more likely than black defendants to be incorrectly flagged as low risk. 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).

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 new 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 and visualization (Ribeiro/Singh/Guestrin 2016) and to propose analytical frameworks for reducing the complexity of algorithmic decision making (Zarsky 2016, Doshi-Velez/Kim 2017), in all likelihood the problem will persist. The digital machine, as a powerful problem solver, will

19 https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm
20 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.
not stop creating new problems on its own. The only way to cope with this is to try to keep up with its development and understand its way of operation. That’s what this script aims to support.

7. References


