

An Introduction to Artificial Intelligence

— Lecture Notes for 2023 —

Prof. Dr. Karl Stroetmann

February 23, 2023

These lecture notes, their LaTeX sources, and the programs discussed in these lecture notes are all available at

https://github.com/karlstroetmann/Artificial-Intelligence.

In particular, the lecture notes are found in the directory Lecture-Notes in the file artificial-intelligence.pdf. The lecture notes are subject to continuous change. Provided the program git is installed on your computer, the command

git clone https://github.com/karlstroetmann/Artificial-Intelligence.git

clones the repository containing the lecture notes and stores them on your local disc. Once you have cloned the repository, the command

git pull

can be used to load the current version of these lecture notes from Github. As artificial intelligence is a very active area of research, these lecture notes will always be incomplete and hence will change from time to time. If you find any typos, errors, or inconsistencies, please contact me via discord or, if that is not possible, email me at:

karl.stroetmann@dhbw-mannheim.de

You are also welcome to send a pull request on GitHub.

Contents

Contents													
1	Intr	Introduction											
	1.1	What is Artificial Intelligence?	4										
	1.2	Overview	4										
	1.3	Literature	6										
2	Search												
	2.1	The Sliding Puzzle	12										
	2.2	Breadth First Search	16										
		2.2.1 A Queue Based Implementation of Breadth First Search	19										
	2.3	Depth First Search	20										
		2.3.1 A Recursive Implementation of Depth First Search	21										
	2.4	Iterative Deepening	22										
	2.5	Bidirectional Breadth First Search	25										
	2.6	Best First Search	28										
	2.7	A* Search	32										
		2.7.1 Completeness and Optimality of A* Search	35										
	2.8	Bidirectional A* Search	39										
	2.9	Iterative Deepening A* Search	41										
	2.10	A*-IDA* Search	44										
3	Solving Constraint Satisfaction Problems												
	3.1	Formal Definition of CSPs	51										
		3.1.1 Example: Map Colouring	51										
		3.1.2 Example: The Eight Queens Puzzle	53										
		3.1.3 Applications	56										
	3.2	Brute Force Search	57										
	3.3	Backtracking Search	59										
	3.4	Constraint Propagation	63										
	3.5	Consistency Checking	71										
	3.6	Local Search	78										
	3.7	Z3	82										
		3.7.1 A Simple Text Problem	82										
		3.7.2 The Knight's Tour	85										

CONTENTS

4	Play	ying Games 90											
	4.1	Basic Definitions											
	4.2	Tic-Tac-Toe											
		4.2.1 A Naive Implementation of Tic-Tac-Toe											
		4.2.2 A Bitboard-Based Implementation of Tic-Tac-Toe											
	4.3	The Minimax Algorithm											
		4.3.1 Memoization											
	4.4	Alpha-Beta Pruning											
		4.4.1 Alpha-Beta Pruning with Memoization											
	4.5	Progressive Deepening											
5	Equational Theorem Proving 111												
J	5.1	Equational Proofs											
	0.1	5.1.1 A Calculus for Equality											
		5.1.2 Equational Proofs											
		5.1.3 Proofs via Rewriting											
	5.2	Confluence											
	$5.2 \\ 5.3$	The Knuth-Bendix Order											
	5.4	Unification											
	5.5												
	5.6	Literature											
6	Line	ear Regression 133											
	6.1	Simple Linear Regression											
		6.1.1 Assessing the Quality of Linear Regression											
		6.1.2 Putting the Theory to the Test											
	6.2	General Linear Regression											
		6.2.1 Some Useful Gradients											
		6.2.2 Deriving the Normal Equation											
		6.2.3 Implementation											
	6.3	Polynomial Regression											
	6.4	Ridge Regression											
7	Clas	ssification 154											
•	7.1	Introduction											
		7.1.1 Notation											
		7.1.2 Applications of Classification											
	7.2	Digression: The Method of Gradient Ascent											
	7.3	Logistic Regression											
	1.0	7.3.1 The Sigmoid Function											
		7.3.2 The Model of Logistic Regression											
		7.3.3 Implementing Logistic Regression											
		7.3.4 Logistic Regression with SciKit-Learn											
	7.4												
	7.4												
	7.5	Naive Bayes Classifiers											
		7.5.1 Example: Spam Detection											
		7.5.2 Naive Bayes Classifier with Numerical Features											
	7 C	7.5.3 Example: Gender Estimation											
	7.6	Support Vector Machines											

CONTENTS

		7.6.1 7.6.2	Non-Optimality of Logistic Regression							
8	Neu		tworks	201						
	8.1		rward Neural Networks							
	8.2	Backpi	copagation	206						
		8.2.1	Definition of some Auxiliary Variables	206						
		8.2.2	The Hadamard Product	207						
		8.2.3	Backpropagation: The Equations	207						
		8.2.4	A Proof of the Backpropagation Equations	209						
	8.3	3 Stochastic Gradient Descent								
	8.4	Implen	nentation	213						
	8.5	Autom	atic Differentiation	220						
Bibliography										
List of Figures										
Index										

Chapter 1

Introduction

1.1 What is Artificial Intelligence?

Before we start to dive into the subject of Artificial Intelligence (usually abbreviated as AI) we should ask ourselves what we mean by the phrase "artificial intelligence". According to the Oxford Dictionary of English, artificial intelligence is:

The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages.

There have been two competitive approaches to machine learning. The first approach is based on symbolic logic and is known as symbolic AI. Typical tasks addressed with this approach were the development of automatic theorem provers, programs to perform symbolic integration, or programs to play chess like Deep Blue. In the beginning, this approach was the dominant paradigm in AI. The second approach is known as machine learning. Arthur Samuel defined machine learning as "the field of study that gives computers the ability to learn without being explicitly programmed" [Sam59]. Machine learning is mostly responsible for the recent hype in AI.

1.2 Overview

This lecture consists of two parts.

- 1. The first part deals with symbolic AI. The common theme of symbolic AI is declarative programming. The main idea of declarative programming is that we start with a formal problem specification. This problem specification is a concise description of the problem that is to be solved, which is then fed into an automatic problem solver that returns a solution of the problem. Originally, declarative programming was a very general approach to problem-solving. The idea was that in order to solve a problem, the problem would first be formulated as a logic formula and an automated theorem prover would then be able to solve the problem. The programming language Prolog was based on that paradigm. Unfortunately, the idea of declarative programming has turned out to be unsuitable as a general problem-solving framework for two reasons:
 - (a) First, it is very difficult to specify practical problems completely in a logical framework.
 - (b) Second, even in those cases where it is feasible to completely specify a problem via logic formulae, automatic theorem proving is, in general, not powerful enough to automatically find a solution.

However, there are a number of domains where the approach of declarative programming has turned out to be useful. In particular, we show how declarative programming can be used to solve problems in the following domains.

- (a) Search problems are problems where the task is to find a path in a graph. A typical example of a search problem is the fifteen puzzle. We discuss various state-of-the-art algorithms that can solve search problems.
- (b) Constraint satisfaction problems have great practical importance. Today, very efficient constraint solvers have been developed to solve various constraint satisfaction problems that occur in practise.
 - We will first discuss various techniques for solving constraint satisfaction problems. Furthermore, we discuss Z3, which is a state-of-the-art automatic theorem prover and constraint solver developed by Microsoft.
- (c) Games like chess or checkers can be specified declaratively and there are various techniques for computers to find optimal strategies for playing adversarial games.

Additionally we will cover automatic theorem proving. As we have already discussed resolution theorem proving in our lecture on logic, we discuss equational theorem proving in the last chapter of the first part of this lecture.

2. In the second part of this lecture, we discuss machine learning. In the last ten years, a number of advances in machine learning have made it into the headlines of the news. The latest hype is caused by ChatGPT. When asked to describe itself in 100 words, ChatGPT gave the following answer:

ChatGPT is a large language model created by OpenAI. It uses cutting-edge machine learning techniques such as deep learning and neural networks to generate human-like text. It can answer questions, write essays, and carry out a wide variety of natural language tasks. ChatGPT is fine-tuned on a massive dataset and can generate responses to a wide range of topics. It's designed to be a powerful tool for automated content generation, customer service, and other applications where understanding and generating human language is critical. With a powerful language model like ChatGPT, it's now possible to automate many tasks that once required human intelligence.

In order to temper the hype about ChatGPT a bit, I would like to share the following conversation that I recently had with ChatGPT.

- **Q:** I have as many brothers as I have sisters. My sister has twice as many brothers as she has sisters. How many children does my father have?
- **A:** It is not possible to determine the number of children your father has based on the information provided. Additional information is needed.
- **Q:** What information do you need to proceed?
- **A:** I would need to know the number of brothers and sisters you have in order to determine the number of children your father has.
- **Q:** You can find these numbers if you convert the information I have given into two equations, where b is the number of boys and g is the number of girls. You can then solve this system of equations.
- A: Sure, using the information you've provided, we can set up the following system of equations:

- (a) b+g=bs (the number of brothers is equal to the number of sisters)
- (b) b = 2gs (your sister has twice as many brothers as she has sisters) To find the number of children your father has, we can substitute the second equation into the first equation:
- (c) b + q = bs
- (d) 2gs + g = gs
- (e) 2gs + g = g(b+g)
- (f) $2gs + g = g^2$

then we can solve for g and find the total number of children.

As you can see for yourself, ChatGPT doesn't get any closer to solving this problem.

It is fair to say that currently, machine learning is the hottest topic in computer science. Among others, we discuss the following algorithms:

- (a) Linear regression is one of the most fundamental machine learning algorithms. In machine learning, we are given a number of data pairs of the form $\langle \mathbf{x}_i, y_i \rangle$ where $i \in \{1, \dots, N\}$ and for all $i \in \{1, \dots, N\}$ we have $\mathbf{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$. We assume that there is an unknown function $f: \mathbb{R}^m \to \mathbb{R}$ such that $y_i \approx f(\mathbf{x}_i)$. Our task is to find an approximation for the function f. When using linear regression, we assume that the function f is linear in its arguments. Although this sounds like a strong assumption, we will see that linear regression is surprisingly powerful in practice.
- (b) In a classification problem we again have N pairs of the form $\langle \mathbf{x}_i, y_i \rangle$. As before, we have $\mathbf{x}_i \in \mathbb{R}^m$, but now $y_i \in \mathbb{B}$, where \mathbb{B} is the set of Boolean values, i.e. we have $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$. The task is then to find a function $f : \mathbb{R}^m \to \mathbb{B}$ such that the equation $y_i = f(\mathbf{x}_i)$ is true for most $i \in \{1, \dots, N\}$. A typical classification problem is spam detection. The first algorithm we introduce to solve classification problems is logistic regression. After that, we study support vector machines and naive Bayes classifiers.
- (c) Finally, we discuss neural networks. As an example, we will build a neural network that is able to recognize digits.

1.3 Literature

The main sources of these lecture notes are the following:

1. A course on artificial intelligence that was offered on the EDX platform. The course materials are available at

http://ai.berkeley.edu/home.html.

2. The book

Introduction to Artificial Intelligence

written by Stuart Russell and Peter Norvig [RN20].

3. A course on artificial intelligence that is offered on Udacity. The title of the course is

Intro to Artificial Intelligence

and the course is given by Peter Norvig, who is director of research at Google and Sebastian Thrun, who is the chairman of Udacity.

When it comes to the exam, you should be fine if you have mastered the material that is covered in my lecture notes. Hence, there is no need to buy any book or take any other course.

Remark: The programs presented in these lecture notes are expected to run with the *Python* version 3.10. The file

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/ai.sh

contains a list of commands that can be used to create the Python environment that I have used to test the notebooks presented in my lectures.

Chapter 2

Search

This chapter is the first of three chapters where we will solve problems by making use of declarative programming. The idea of declarative programming is that rather than developing a program to solve a specific problem, we implement an algorithm that can solve a whole class of problems. Then, in order to solve a problem that falls within this class, we just have to specify the problem, which is usually much easier than writing a program that solves the given problem. In this chapter, this idea is illustrated via search problems. First, we define the notion of a search problem formally. This notion is then illustrated with two examples. We start with the missionaries and cannibals problem. Next, we use the sliding puzzle as our running example. After that, we introduce various algorithms for solving search problems. In particular, we present

- 1. breadth first search,
- 2. depth first search,
- 3. iterative deepening,
- 4. bidirectional breadth first search,
- 5. A* search and bidirectional A* search,
- 6. iterative deepening A* search, and
- 7. A^* -IDA* search.

Definition 1 (Search Problem) A search problem is a tuple of the form

$$\mathcal{P} = \langle Q, \text{ next_states}, \text{ start}, \text{ goal} \rangle$$

where

- 1. Q is the set of states, also known as the state space.
- 2. next_states is a function taking a state as input and returning the set of those states that can be reached from the given state in one step, i.e. we have

$$next_states: Q \rightarrow 2^Q$$
.

The function next_states gives rise to the transition relation R, which is a binary relation on Q, i.e. we have $R \subseteq Q \times Q$. This relation is defined as follows:

$$R := \{ \langle s_1, s_2 \rangle \in Q \times Q \mid s_2 \in \mathtt{next_states}(s_1) \}.$$

If either $\langle s_1, s_2 \rangle \in R$ or $\langle s_2, s_1 \rangle \in R$, then s_1 and s_2 are called neighboring states.

- 3. start is the start state, hence start $\in Q$.
- 4. goal is the goal state, hence goal $\in Q$.

Sometimes, instead of a single state goal there is a set of states Goals.

A path is a list $[s_1, \cdots, s_n]$ such that $s_{i+1} \in \texttt{next_states}(s_i)$ for all $i \in \{1, \cdots, n-1\}$. The length of this path is defined as the length of this list minus 1, i.e. the path $[s_1, \cdots, s_n]$ has length n-1. The reason for defining the length of this path as n-1 and not n is that the path consists of n-1 edges of the form $\langle s_i, s_{i+1} \rangle$ where $i \in \{1, \cdots, n-1\}$. A path $[s_1, \cdots, s_n]$ is a solution to the search problem $\mathcal P$ iff the following conditions are satisfied:

- 1. $s_1 = \mathtt{start}$, i.e. the first element of the path is the start state.
- 2. $s_n = \text{goal}$, i.e. the last element of the path is the goal state.

If instead of a single goal we have a set of Goals, then the last condition is changed into

 $s_n \in Goals.$

A path $p = [s_1, \dots, s_n]$ is a minimal solution to the search problem \mathcal{P} iff it is a solution and, furthermore, the length of p is minimal among all other solutions.

Remark: In the literature, a state is often called a node. In these lecture notes, I will sometimes refer to states as nodes.

Example: We illustrate the notion of a search problem with the following example, which is also known as the missionaries and cannibals problem: Three missionaries and three infidels have to cross a river that runs from the north to the south. Initially, both the missionaries and the infidels are on the western shore. There is just one small boat that can carry at most two passengers. Both the missionaries and the infidels can steer the boat. However, if at any time the missionaries are confronted with a majority of infidels on either shore of the river, then the missionaries have a problem. Figure 2.1 shows an artist's rendition of the problem.

Figure 2.2 shows a formalization of the missionaries and cannibals problem as a search problem. We discuss this formalization line by line.

1. Line 1 defines the auxiliary function problem.

If m is the number of missionaries on a given shore, while i is the number of infidels on that same shore, then problem(m, i) is True iff the missionaries have a problem on that shore. There is a problem if the number of missionaries is greater than 0 but less than the number of infidels.

2. Line 4 defines the auxiliary function no_problem.

If m is the number of missionaries on the western shore and i is the number of infidels on that shore, then the expression $no_problem(m,i)$ is True, if there is no problem for the missionaries on either shore.

The implementation of this function uses the fact that if m is the number of missionaries on the western shore, then 3-m is the number of missionaries on the eastern shore. Similarly, if i is the number of infidels on the western shore, then the number of infidels on the eastern shore is 3-i.

¹Thanks to Marcel Vilas for providing this beautiful painting as well as an animation of this problem.



Figure 2.1: Start state of the missionaries-and-infidels problem.

3. Lines 7 to 20 define the function next_states. A state s is represented as a triple of the form s = (m, i, b) where $m \in \{0, 1, 2, 3\}, i \in \{0, 1, 2, 3\}, and b \in \{0, 1\}.$

Here m, i, and b are, respectively, the number of missionaries, the number of infidels, and the number of boats on the western shore.

- (a) Line 8 extracts the components m, i, and b from the state s.
- (b) Line 9 checks whether the boat is on the western shore.
- (c) If this is the case, then the states reachable from the given state s are those states where mb missionaries and ib infidels cross the river. After mb missionaries and ib infidels have crossed the river and reached the eastern shore, m-mb missionaries and i-ib infidels remain on the western shore. Of course, after the crossing, the boat is no longer on the western shore. Therefore, the new state has the form

$$(m - mb, i - ib, 0).$$

This explains line 10.

(d) Since the number mb of missionaries leaving the western shore can not be greater than the number m of all missionaries on the western shore, we have the condition

```
def problem(m, i):
            return 0 < m < i
2
3
       def no_problem(m, i):
4
            return not problem(m, i) and not problem(3 - m, 3 - i)
5
6
       def next_states(state):
            m, i, b = state
            if b == 1:
                return { (m - mb, i - ib, 0) for mb in range(m+1)
10
                                                for ib in range(i+1)
11
                                                if 1 \le mb + ib \le 2 and
12
                                                   no_problem(m - mb, i - ib)
13
                        }
14
            else:
                return { (m + mb, i + ib, 1) for mb in range(3-m+1)
                                                for ib in range(3-i+1)
17
                                                if 1 \le mb + ib \le 2 and
18
                                                   no_problem(m + mb, i + ib)
19
                        }
20
21
       start = (3, 3, 1)
22
       goal = (0, 0, 0)
```

Figure 2.2: The missionary and cannibals problem coded as a search problem.

```
mb \in \{0, \cdots, m\},\
```

which is implemented by the line

```
for mb in range(m+1).
```

There is a similar condition for the number of infidels crossing:

$$ib \in \{0, \cdots, i\}$$

which is implemented by

```
for ib in range(i+1).
```

(e) Furthermore, we have to check that the number of persons crossing the river is at least 1 and at most 2. This explains the condition

```
1 \le mb + ib \le 2.
```

(f) Finally, there should be no problem in the new state on either shore. This is checked using the expression

```
noProblem(m - mb, i - ib).
```

4. If the boat is on the eastern shore instead, then the missionaries and the infidels will be crossing the river from the eastern shore to the western shore. Therefore, the number of missionaries and infidels on the western shore is now increased. Hence, in this case the new state has the form

```
(m + mb, i + ib, 1).
```

Here, mb is the number of missionaries arriving on the western shore and ib is the number of arriving infidels. As the number of missionaries on the eastern shore is 3-m and the number of infidels on the eastern shore is 3-i, mb has to be a member of the set $\{0, \dots, 3-m\}$, while ib has to be a member of the set $\{0, \dots, 3-i\}$.

5. Finally, the start state and the goal state are defined in line 22 and line 23.

The code in Figure 2.2 does not define the set of states Q of the search problem. The reason is that, in order to solve the problem, we do not need to define this set. If we wanted to, we could define the set of states as follows:

However, in general the set of states is not needed by the algorithms solving search problems and in many cases this set is so big that it would be impossible to compute and store it. Hence, in practice the set of states is only an abstract notion that is needed in order to specify the function next_states, but it is not implemented.

Figure 2.3 shows a graphical representation of the transition relation of the missionaries and cannibals puzzle. In that figure, for every state, both the western and the eastern shore are shown. The start state is covered with a blue ellipse, while the goal state is covered with a green ellipse. The figure clearly shows that the problem is solvable and that there is a solution involving just 11 crossings of the river.

2.1 The Sliding Puzzle

The missionaries and cannibals problem is rather small and therefore it is not useful when we want to compare the efficiency of various algorithms for solving search problems. Therefore, we will now present a problem that has a greater complexity: The 3×3 sliding puzzle uses a square board, where each side has a length of 3. This board is subdivided into $3 \times 3 = 9$ squares of length 1. Of these 9 squares, 8 are occupied with square tiles that are numbered from 1 to 8. One square remains empty. Figure 2.4 on page 13 shows two possible states of this sliding puzzle. The 4×4 sliding puzzle is similar to the 3×3 sliding puzzle, but uses a square board of size 4 instead. The 4×4 sliding puzzle is also known as the 15 puzzle, while the 3×3 puzzle is called the 8 puzzle.

In order to solve the 3×3 sliding puzzle shown in Figure 2.4 we have to transform the state shown on the left of Figure 2.4 into the state shown on the right of this figure. The following operations are permitted when transforming a state of the sliding puzzle:

- 1. If a tile is to the left of the free square, this tile can be moved to the right.
- 2. If a tile is to the right of the free square, this tile can be moved to the left.
- 3. If a tile is above the free square, this tile can be moved down.
- 4. If a tile is below the free square, this tile can be moved up.



Figure 2.3: A graphical representation of the missionaries and cannibals problem.



Figure 2.4: The 3×3 sliding puzzle.

In order to get a feeling for the complexity of the sliding puzzle, you can check the page

https://www.helpfulgames.com/subjects/brain-training/sliding-puzzle.html.

The sliding puzzle is much more complex than the missionaries and cannibals problem because the

state space is much larger. For the case of the 3×3 sliding puzzle, there are 9 squares that can be positioned in 9! different ways. It turns out that only half of these positions are reachable from a given start state. Therefore, the effective number of states for the 3×3 sliding puzzle is

$$9!/2 = 181,440.$$

This is already a big number, but 181,440 states can still be stored on a modern computer. However, the 4×4 sliding puzzle has

$$16!/2 = 10,461,394,944,000$$

different states reachable from a given start state. If a state is represented as a matrix containing 16 numbers and we store every number using just 4 bits, we still need $16 \cdot 4 = 64$ bits or 8 bytes for every state. Hence, we would need a total of

$$(16!/2) \cdot 8 = 83,691,159,552,000$$

bytes to store every state. We would thus need about 84 terabytes to store the set of all states. As few computers are equipped with this kind of memory, it is obvious that we won't be able to store the entire state space in memory.

Figure 2.5 shows how the 3×3 sliding puzzle can be formulated as a search problem. In order to discuss the program, we first have to understand that states are represented as tuples of tuples. For example, the state shown above on the left side in Figure 2.4 is represented as the tuple:

```
( (8, 0, 6),
(5, 4, 7),
(2, 3, 1)
```

Here, we have represented the empty tile as 0. If states are represented as tuples of tuples, given a state s, the expression s[r][c] returns the tile in the row r and column c, where the counting of rows and columns starts from 0. We have to represent states as tuples of tuples rather than lists of lists since tuples are immutable while lists are mutable and we need to store states in sets later. In Python, sets can only store immutable objects. However, we also have to manipulate the states. To this end, we have to first transform the states to lists of lists, which can be manipulated. After the manipulation, these lists of lists have to be transformed back to tuples of tuples. We proceed to discuss the program shown in Figure 2.5 line by line.

- 1. The function to_list transforms a tuple of tuples into a list of lists.
- 2. The function to_tuple transforms a list of lists into a tuple of tuples.
- 3. find_tile is an auxiliary function that is needed to implement the function next_states. It is called with a number and a State and returns the row and column where the tile labelled with number can be found.
- 4. move_dir takes a State, the row and the column where to find the empty square and a direction in which the empty square should be moved. This direction is specified via the two variables dx and dy. The tile at the position $\langle row + dx, col + dy \rangle$ is moved into the position $\langle row, col \rangle$, while the tile at position $\langle row + dx, col + dy \rangle$ becomes empty.
- 5. Given a State, the function next_states computes the set of all states that can be reached in one step from State. The basic idea is to find the position of the empty tile and then try to move the empty tile in all possible directions. If the empty tile is found at position $\langle row, col \rangle$ and the direction of the movement is given as $\langle dx, dy \rangle$, then in order to ensure that the empty

 \Diamond

```
def to_list(State):
            return [list(row) for row in State]
2
       def to_tuple(State):
            tuple(tuple(row) for row in State)
       def find_tile(tile, State):
5
            n = len(State)
            for row in range(n):
                for col in range(n):
                    if State[row][col] == tile:
                         return row, col
10
11
       def move_dir(State, row, col, dx, dy):
12
            State = to_list(State)
13
                                     ] = State[row + dx][col + dy]
            State[row
                           ][col
14
            State[row + dx][col + dy] = 0
            return to_tuple(State)
17
       def next_states(State):
18
                        = len(State)
19
            row, col
                       = find_tile(0, State)
20
            New_States = set()
21
            Directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
22
            for dx, dy in Directions:
                if row + dx in range(n) and col + dy in range(n):
                    New_States.add(move_dir(State, row, col, dx, dy))
25
            return New_States
26
27
       start = ((8, 0, 6),
28
                  (5, 4, 7),
29
                  (2, 3, 1)
                )
31
32
       goal = ((0, 1, 2),
33
                 (3, 4, 5),
34
                 (6, 7, 8)
35
               )
36
```

Figure 2.5: The 3×3 sliding puzzle.

```
tile can be moved to the position \langle row + dx, col + dy \rangle, we have to ensure that both row + dx \in \{0, \dots, n-1\} and col + dy \in \{0, \dots, n-1\} hold, where n is the size of the board.
```

Next, we want to develop an algorithm that can solve puzzles of the kind described so far. The most basic algorithm to solve search problems is breadth first search. We discuss this algorithm next.

2.2 Breadth First Search

Informally, breadth first search, abbreviated as BFS, works as follows:

1. Given a search problem $\langle Q, \text{ next_states}, \text{ start}, \text{ goal} \rangle$, we initialize a set Frontier to contain the state start.

In general, Frontier contains those states that have just been discovered and whose successors have not yet been seen.

2. As long as the set Frontier does not contain the state goal, we recompute this set by adding all states to it that can be reached in one step from a state in Frontier. Then, the states that had been previously present in Frontier are removed. These old states are then added to the set Visited.

In order to avoid going around in circles, an implementation of breadth first search keeps track of those states that have been visited in the set Visited. Once a state has been added to the set Visited, it will never be revisited again. Furthermore, in order to keep track of the path leading to the goal, we utilize a dictionary called Parent. For every state s that is in Frontier, Parent[s] is the state that caused s to be added to the set Frontier, i.e. for all states $s \in F$ rontier we have

 $s \in \texttt{next_states}(\texttt{Parent}[s]).$

```
def search(start, goal, next_states):
           Frontier = { start }
           Visited = set()
           Parent
                     = { start: start }
4
           while Frontier:
                NewFrontier = set()
6
                for s in Frontier:
                    for ns in next_states(s):
                        if ns not in Visited and ns not in Frontier:
                            NewFrontier.add(ns)
10
                            Parent[ns] = s
11
                            if ns == goal:
12
                                 return path_to(goal, Parent)
13
                Visited |= Frontier
14
                Frontier = NewFrontier
```

Figure 2.6: Breadth first search.

Figure 2.6 on page 16 shows an implementation of breadth first search in *Python*. The function search takes three arguments to solve a search problem:

- (a) start is the start state of the search problem,
- (b) goal is the goal state of the search problem, and
- (c) next_states is a function with signature

```
next_states: Q \to 2^Q,
```

where Q is the set of states. For every state $s \in Q$, $next_states(s)$ is the set of states that can be reached from s in one step.

If successful, search returns a path from start to goal that is a solution of the search problem

```
\langle Q, \mathtt{next\_states}, \mathtt{start}, \mathtt{goal} \rangle.
```

Next, we discuss the implementation of the function search:

- 1. Frontier is the set of all those states that have been encountered but whose neighbours have not yet been explored. Initially, it contains the state start.
- 2. Visited is the set of all those states, all of whose neighbours have already been added to the set Frontier in the last iteration of the while loop. In order to avoid infinite loops, these states must not be visited again.
- 3. Parent is a dictionary keeping track of the predecessors of the state that have been reached. The only state with no real predecessor is the state start. By convention, start is its own predecessor.
- 4. As long as the set Frontier is not empty, we add all neighbours of states in Frontier that have not yet been visited to the set NewFrontier. When doing this, we keep track of the path leading to a new state ns by storing its parent in the dictionary Parent.
- 5. If the new state happens to be the state goal, we return a path leading from start to goal. The function path_to() is shown in Figure 2.7 on page 17.
- 6. After we have collected all successors of states in Frontier, the states in the set Frontier have been visited and are therefore added to the set Visited, while the set Frontier is updated to NewFrontier.

```
def path_to(state, Parent):
    p = Parent[state]
    if p == state:
        return [state]
    return path_to(p, Parent) + [state]
```

Figure 2.7: The function path_to.

The function call path_to(state, Parent) constructs a path reaching from start to state in reverse by looking up the parent states. It uses the fact that only the start state is its own parent.

If we try breadth first search to solve the missionaries and cannibals problem, we obtain the solution shown in Figure 2.8. 15 nodes had to be expanded to find this solution. To keep this in perspective, we note that Figure 2.3 shows that the entire state space contains 16 states. Therefore, with the exception of one state, we have inspected all the states. This is a typical behaviour for breadth first search.

Next, let us try to solve the 3×3 sliding puzzle. It takes about 2 seconds to solve this problem on my computer², while 181,439 states are touched. Again, we see that breadth first search touches nearly

²I happen to own an iMac from 2017. This iMac is equipped with 32 Gigabytes of main memory and a quad core 3.4 GHz "Intel Core i5" processor. I suspect this to be the I5-7500 (Kaby Lake) processor.

1	MMM	KKK	В	~~~~			
2				> KK >			
3	MMM	K		~~~~		KK	В
4				< K <			
5	MMM	KK	В	~~~~		K	
6				> KK >			
7	MMM			~~~~		KKK	В
8				< K <			
9	MMM	K	В	~~~~		KK	
10				> MM >			
11	M	K		~~~~	MM	KK	В
12				< M K <			
13	MM	KK	В	~~~~	M	K	
14				> MM >			
15		KK		~~~~	MMM	K	В
16				< K <			
17		KKK	В	~~~~	MMM		
18				> KK >			
19		K		~~~~	MMM	KK	В
20				< K <			
21		KK	В	~~~~	MMM	K	
22				> KK >			
23				~~~~	MMM	KKK	В

Figure 2.8: A solution of the missionaries and cannibals problem.

all the states reachable from the start state. If we measure the memory consumption, we discover that the program uses about 90 megabytes of memory.

Breadth first search has two important properties:

- (a) Breadth first search is complete: If there is a solution to the given search problem, then breadth first search is going to find it.
- (b) The solution found by breadth first search is optimal, i.e. it is one of the shortest possible solutions.

Proof: Both of these claims can be shown simultaneously. Consider the implementation of breadth first search shown in Figure 2.6 on page 16. We prove by induction on the number of iterations of the while loop that after n iterations of the while loop, the set Frontier contains exactly those states that have a distance of n to the state start.

Base Case: n = 0.

After 0 iterations of the while loop, i.e. before the first iteration of this loop, the set Frontier only contains the state start. As this is the only state that has a distance of 0 to the state start, the claim is true in this case.

Induction Step: $n \mapsto n+1$.

In the induction step we assume the claim is true after n iterations. Then, in the next iteration all states that can be reached in one step from a state in Frontier are added to the new Frontier,

provided there is no shorter path to them. By induction hypothesis, there is a shorter path to a state if this state is already a member of the set Visited. In this case, the state would not be added to NewFrontier. Otherwise, the shortest path to a state that is reached in iteration n+1 has the length n+1 and the state is added to NewFrontier. Hence, the claim is true after n+1 iterations also.

Now, if there is a path from start to goal, there must also be a shortest path. Assume this path has a length of k. Then, goal is reached in the kth iteration and the shortest path is returned.

The fact that breadth first search is both complete and the path returned is optimal is rather satisfying. However, breadth first search still has a big downside that makes it unusable for many problems: If the goal is far from the start, breadth first search will use a lot of memory because it will store a large part of the state space in the set Visited. In many cases, the state space is so big that this is not possible. For example, it is impossible to solve the more interesting cases of the 4×4 sliding puzzle using breadth first search.

2.2.1 A Queue Based Implementation of Breadth First Search

In the literature, for example in Figure 3.9 of Russell & Norvig [RN20], breadth first search is often implemented using a queue data structure.

```
from collections import deque
2
       def search(start, goal, next_states):
            Frontier = deque([start])
                     = { start: start }
            Parent
            while Frontier:
6
                state = Frontier.popleft()
                if state == goal:
                    return path_to(state, Parent)
9
                for ns in next_states(state):
10
                    if ns not in Parent:
                        Parent[ns] = state
12
                        Frontier.append(ns)
13
```

Figure 2.9: A queue based implementation of breadth first search.

Figure 2.9 on page 19 shows an implementation of breadth first search that uses a queue to store the set Frontier. Here we use the module deque from the package collections. This module implements a double-ended queue, which is implemented as a doubly linked list. Besides the constructor, our implementation uses two methods from the class deque:

- 1. Line 4 initializes the Frontier as a double-ended queue that contains the state start.
- 2. In line 7 we remove the oldest element in the queue Frontier, which is supposed to be at the left end of the queue. This is achieved via the method popleft.
- 3. In line 14 we add the states that have not been encountered previously at the right end of the queue Frontier using the method append.

Additionally, we have used the fact that the information contained in the set Visited is already available in the dictionary Parent, because when we visit a state s, we add an entry for Parent[s]. As a result, this implementation of breadth first search is slightly faster than our previous implementation. Furthermore, only 76 megabytes of memory are used for the computation.

2.3 Depth First Search

To overcome the memory limitations of breadth first search, the depth first search algorithm has been developed. Depth first search is abbreviated as DFS. There are two ideas involved when going from breadth first search to depth first search:

- 1. In order to save memory, DFS removes the set Visited of BFS.
- 2. While BFS ensures that every state is visited by implementing the Frontier as a queue, DFS replaces this queue by a stack. This way, DFS tries to get as far away from the state start as early as possible. In order to prevent loops, we still have the parent dictionary.

Since a stack can be implemented as an ordinary *Python* list, we don't need the module deque anymore.

```
def search(start, goal, next_states):
            Stack = [start]
           Parent = { start: start }
            while Stack:
                state = Stack.pop()
                for ns in next_states(state):
                    if ns not in Parent:
                        Parent[ns] = state
                        Stack.append(ns)
                        if ns == goal:
10
                            return path_to(goal, Parent)
11
12
       def path_to(state, Parent):
13
          Path = [state]
14
           while state != Parent[state]:
               state = Parent[state]
16
               Path = [ state ] + Path
17
           return Path
18
```

Figure 2.10: The depth first search algorithm.

The resulting algorithm is shown in Figure 2.10 on page 20. Basically, in this implementation, a path is searched to its end before trying an alternative. This way, we might be able to find a goal that is far away from start without exploring the whole state space.

The implementation of search works as follows:

1. Any states that are encountered during the search are placed on top of the stack Stack.

- 2. In order to record the information how a state has been added to the Stack, we have the dictionary Parent. For every state s that is on Stack, Parent[s] returns a state p such that $s \in \texttt{next_states}(p)$, i.e. p is the state that immediately precedes s on the path that leads from start to s.
- 3. Initially, Stack only contains the state start.
- 4. As long as Stack is not empty, the state on top of Stack is replaced by all states that can be reached in one step from state. However, in order to prevent depth first search to run in circles, only those states ns from the set next_states(state) are appended to Stack that have not been encountered previously. This is checked by testing whether ns is in the domain of Parent.
- 5. When the goal is reached, a path leading from start to goal is returned.
- 6. We have reimplemented the function path_to using a while loop. The reason is that the recursive implementation that we had used before is not viable when the path gets too long because the recursion limit in *Python* is set to 3000 and hence the previous implementation of path_to does not work if the path exceeds a length of 3000.

When we test the implementation shown above with the 3×3 sliding puzzle, it takes 264 milliseconds on my computer to find a solution. This is an improvement compared to breadth first search. The memory consumption is reduced to 3 megabytes. This is still a lot and is due to the fact that we still have to maintain the dictionary Parent. Fortunately, we will be able to get rid of the dictionary Parent when we develop a recursive implementation of depth first search in the following subsection.

However, there is also bad news: the solution that is found has a length of 17,510 steps. As the shortest path from start to goal has only 31 steps, the solution found by depth first search is very far from being optimal.

2.3.1 A Recursive Implementation of Depth First Search

Sometimes, the depth first search algorithm is presented as a recursive algorithm, since this leads to an implementation that is slightly shorter and also easier to understand. What is more, we no longer need the dictionary Parent to record the parent of each node. The resulting implementation is shown in Figure 2.11 on page 22.

The only purpose of the function search is to call the function dfs, which needs two additional arguments. These arguments are called Path and PathSet. The idea is that Path is a path leading from the state start to the current state that is the first argument of the function dfs, while PathSet is a set containing all the elements of the path Path. The argument PathSet is only used for efficiency reasons: In order to avoid infinite loops, when we discover a node we have to check that this node does not occur already in Path. However, checking whether an element occurs in the list Path is much slower than checking whether the element occurs in the corresponding set PathSet. On the first invocation of dfs, the parameter state is equal to start and therefore Path is initialized as the list containing only start.

The implementation of dfs works as follows:

- 1. If state is equal to goal, our search is successful. Since by assumption the list Path is a path connecting start and state and we have checked that state is equal to goal, we can return Path as our solution.
- 2. Otherwise, next_states(state) is the set of states that are reachable from state in one step. Any of the states ns in this set could be the next state on a path that leads to goal. Therefore, we try recursively to reach goal from every state ns. Note that we have to change Path to the

```
def search(start, goal, next_states):
            return dfs(start, goal, next_states, [start], { start })
3
       def dfs(state, goal, next_states, Path, PathSet):
4
            if state == goal:
5
                return Path
6
           for ns in next_states(state):
                if ns not in PathSet:
                    PathSet.add(ns)
                    Result = dfs(ns, goal, next_states, Path + [ns], PathSet)
10
                    if Result:
11
                        return Result
12
                    PathSet.remove(ns)
13
           return None
14
```

Figure 2.11: A recursive implementation of depth first search.

list

when we call the function dfs recursively. This way, we retain the invariant of dfs that the list Path is a path connecting start with state.

- 3. We still have to avoid running in circles. In the recursive version of depth first search, this is achieved by checking that the state ns is not already a member of the set PathSet. In the non-recursive version of depth first search, we had used the set Parent instead. The current implementation no longer has a need for the dictionary Parent. This is very fortunate since it reduces the memory requirements of depth first search considerably.
- 4. If one of the recursive calls of dfs returns a list, this list is a solution to our search problem and hence it is returned. However, if instead None is returned, the for loop needs to carry on and test the other successors of state.
- 5. If the search was unsuccessful, we have to remove the state ns from the PathSet in order to maintain the invariant that PathSet is the set of all elements in the list Path.
- 6. Note that the recursive invocation of dfs returns None if the end of the for loop is reached and no solution has been returned so far.

Unfortunately, due to a bug in Python~3.10, the $Python~kernel~just~dies~when~trying~to~solve~the~<math>3\times3$ sliding puzzle. With Python~3.8 it took 297 milliseconds on my computer to find a solution. The memory consumption was back to 50 megabytes. Furthermore, the solution had a length of 20,923 steps, which explains the high memory consumption.

2.4 Iterative Deepening

The fact that the stack-based version of depth first search took less than one second to find a solution is very impressive, but the fact that this solution has a length of more than ten thousand steps

is disappointing. The question is, whether it might be possible to force depth first search to find the shortest solution. The answer to this question leads to an algorithm that is known as iterative deepening. The main idea behind iterative deepening is to run depth first with a depth limit d. This limit enforces that a solution has at most a length of d. If no solution is found at a depth of d, the new depth d+1 can be tried next and the process can be continued until a solution is found. The program shown in Figure 2.12 on page 23 implements this strategy. We proceed to discuss the details of this program.

```
def search(start, goal, next_states):
            limit = 1
            while True:
3
                Path = dls(start, goal, next_states, [start], { start }, limit)
                if Path is not None:
                     return Path
                limit += 1
       def dls(state, goal, next_states, Path, PathSet, limit):
9
            if state == goal:
10
                return Path
11
            if len(Path) == limit:
12
                return None
            for ns in next_states(state):
14
                if ns not in PathSet:
15
                     Path
                             .append(ns)
16
                     PathSet.add(ns)
17
                     Result = dls(ns, goal, next_states, Path, PathSet, limit)
18
                     if Result:
19
                         return Result
20
                     else:
21
                         Path
                                 .pop()
22
                         PathSet.remove(ns)
23
            return None
24
```

Figure 2.12: Iterative deepening implemented in Python.

- 1. The function search initializes the variable limit to 1 and tries to find a solution to the search problem that has a length that is less than or equal to limit. If a solution is found, it is returned. Otherwise, the variable limit is incremented by one and a new instance of depth first search is started. This process continues until either a solution is found or the sun rises in the west.
- 2. The function dls implements a recursive version of depth first search but takes care to compute only those paths that have a length of at most limit. The name dls is short for depth limited search. If the Path has reached a length of limit but does not end in goal, the function returns None instead of trying to extend this Path. Otherwise, the implementation is the same as the recursive implementation of depth first search that was shown in Figure 2.11 on page 22 and that has been discussed in the previous section.

The nice thing about the program presented in this section is the fact that it uses only 136 kilobytes of memory. The reason is that the Path can never have a size that is longer than limit. However, when we run this program to solve the 3×3 sliding puzzle, the algorithm takes about 7 minutes. There are two reasons for the long computation time:

1. First, it is quite wasteful to run the search for a depth limit of 1, 2, 3, \cdots all the way up to 31. Essentially, all the computations done with a limit less than 32 are wasted. However, this process is not as wasteful as we might first expect. To see this, assume that the number of states reached is doubled³ after every iteration. Then the number of states to explore when searching with a depth limit of d is roughly 2^d . Hence, when we run depth limited search up to depth d, the number of states visited is

$$1 + 2^{1} + 2^{2} + \dots + 2^{d} = \sum_{i=0}^{d} 2^{i} = 2^{d+1} - 1.$$

Therefore, if the solution is found at a depth of d+1, we will explore at most 2^{d+1} states to find the solution if we would do depth first search with a depth limit of d+1. If, instead, we use iterative deepening, we have wastefully explored an additional number of $2^{d+1}-1$ states. Hence, we will visit only twice the number of states with iterative deepening than we would have visited with depth limited search with the correct depth limit.

2. Given a state s that is reachable from the start, there often are a huge number of different paths that lead from start to s. The version of iterative deepening presented in this section tries all of these paths and hence needs a large amount of time.

To check what is really going on, we can change the initial value of limit that is set to 1 in line 2 of Figure 2.12 on page 23. If we set this value to 31, which is one less that the value that is needed, the program needs about 5 minutes to compute the solution. However, if this value is set to 32, then the program is able to find the solution in less than two minutes. The reason is that in the case that limit has the value 31, the program has to check all possible lists Path that have a length of at most 31. Unfortunately, there is no such list, so all possible states that have a distance of at most 30 from start have to be explored. However, if limit has the value 32, it is sufficient to find any Path of length 32 that leads to the goal and if that Path has been found, the program can return immediately. The following exercise digs deeper into this observation.

Exercise 1: Assume the set of states Q is defined as

$$Q := \{ \langle a, b \rangle \mid a \in \mathbb{N} \land b \in \mathbb{N} \}.$$

Furthermore, the states start and goal are defined as

$$start := \langle 0, 0 \rangle$$
 and $goal := \langle n, n \rangle$ where $n \in \mathbb{N}$.

Next, the function next_states is defined as

$$\mathtt{next_states}\big(\langle a,b\rangle\big) := \big\{\langle a+1,b\rangle, \langle a,b+1\rangle\big\}.$$

Finally, the search problem \mathcal{P} is defined as

$$\mathcal{P} := \langle Q, \mathtt{next_states}, \mathtt{start}, \mathtt{goal} \rangle.$$

Given a natural number n, compute the number of different solutions of this search problem and prove your claim. The Figure 2.13 on page 25 shows possible solutions in a graph.

³When we run breadth first search for the sliding puzzle, we can observe that at least at the beginning of the search, the number of states is roughly doubled after each step. This observation holds true for the first 16 steps.



Figure 2.13: Example for possible paths in a graph

Hint: The expression giving the number of different solutions contains factorials. In order to get a better feeling for the asymptotic growth of this expression we can use Stirling's approximation of the factorial. Stirling's approximation of n! is given as follows:

$$n! \sim \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n$$
.

Exercise 2: If there is no solution, the implementation of iterative deepening that is shown in Figure 2.12 does not terminate. The reason is that the function dls does not distinguish between the following two reasons for failure:

- 1. It can fail to find a solution because the depth limit is reached.
- 2. It can also fail because it has exhausted all possible paths without hitting the depth limit.

Improve the implementation of iterative deepening so that it will always terminate eventually, provided the state space is finite.

2.5 Bidirectional Breadth First Search

Breadth first search first visits all states that have a distance of 1 from start, then all states that have a distance of 2, then of 3 and so on until finally the goal is found. If the length of the shortest path from start to goal is d, then all states that have a distance of at most d will be visited. In many search problems, the number of states grows exponentially with the distance, i.e. there is a branching factor b such that the set of all states that have a distance of at most d from start is roughly

$$1 + b + b^2 + b^3 + \dots + b^d = \frac{b^{d+1} - 1}{b-1} = \mathcal{O}(b^d).$$

At least this is true in the beginning of the search. As the size of the memory that is needed is the most constraining factor when searching, it is important to cut down this size. One simple idea is to start searching both from the node start and the node goal simultaneously. This approach is known as bidirectional search. The justification is that the path starting from start and the path starting from goal will meet in the middle and hence they will both have a size of approximately d/2. If this is the case, only

$$2 \cdot (1 + b + \dots + b^{\frac{d}{2}}) = 2 \cdot \frac{b^{\frac{d}{2} + 1} - 1}{b - 1}$$

nodes need to be explored and even for modest values of b this number is much smaller than

$$\frac{b^{d+1}-1}{b-1}$$

which is the number of nodes expanded in breadth first search. For example, assume that the branching factor b = 2 and that the length of the shortest path leading from start to goal is d = 40. Then we need to explore

$$2^{41} - 1 = 2,199,023,255,551$$

states in breadth first search, while we only have to explore

$$2 \cdot \left(2^{\frac{40}{2}+1} - 1\right) = 4,194,302$$

states with bidirectional breadth first search. While it is certainly feasible to keep four million states in memory, keeping two trillion states in memory is impossible on most devices. The Figure 2.14 on page 26 should demonstrate, how the conventional search algorithm has to use a lot more space than the bidirectional approach.



Figure 2.14: Example of space usage of conventional and bidrectional-BFS

Figure 2.15 on page 27 shows the implementation of bidirectional breadth first search. Essentially, we have two copies of the breadth first search program shown in Figure 2.6. However, since the information that was stored in the set Visited in the implementation of BFS shown in Figure 2.6 is also available in the dictionary Parent, we have removed the variable Visited in our implementation of bidirectional breadth first search.

Let us discuss the details of the implementation.

- 1. The variable FrontierA is the frontier that starts from the state start, while FrontierB is the frontier that starts from the state goal.
- 2. For every state s that is in FrontierA, ParentA[s] is the state that caused s to be added to the set FrontierA. Similarly, for every state s that is in FrontierB, ParentB[s] is the state that caused s to be added to the set FrontierB.
- 3. The bidirectional search keeps running for as long as both sets FrontierA and FrontierB are non-empty and a path has not yet been found.
- 4. The while loop first computes those states that can be reached from FrontierA and have not been visited from start. If a state ns is a neighbour of a state s from the set FrontierA and the state ns has already been encountered during the search that started from goal, then a path leading from start to goal has been found and this path is returned. The function combinePaths that computes this path by combining the path that leads from start to ns and then from ns to goal is shown in Figure 2.16 on page 27.

```
def search(start, goal, next_states):
           FrontierA = { start }
           ParentA
                      = { start: start}
           FrontierB = { goal }
                      = { goal: goal}
           ParentB
           while FrontierA and FrontierB:
                NewFrontier = set()
                for s in FrontierA:
                    for ns in next_states(s):
                        if ns not in ParentA:
10
                            NewFrontier |= { ns }
11
                            ParentA[ns] = s
12
                            if ns in ParentB:
13
                                 return combinePaths(ns, ParentA, ParentB)
14
                FrontierA
                            = NewFrontier
                NewFrontier = set()
                for s in FrontierB:
17
                    for ns in next_states(s):
18
                        if ns not in ParentB:
19
                            NewFrontier |= { ns }
20
                            ParentB[ns] = s
21
                            if ns in ParentA:
22
23
                                 return combinePaths(ns, ParentA, ParentB)
                FrontierB = NewFrontier
```

Figure 2.15: Bidirectional breadth first search.

5. Next, the while loop performs the same computation with the role of the states start and goal exchanged.

On my computer, bidirectional breadth first search solves the 3×3 sliding puzzle in 128 milliseconds and uses 4 megabytes. However, bidirectional breadth first search is still not able to solve the more interesting cases of the 4×4 sliding puzzle since the portion of the search space that needs to be computed is just too big to fit into memory.

```
def combinePaths(state, ParentA, ParentB):

Path1 = path_to(state, ParentA)

Path2 = path_to(state, ParentB)

return Path1[:-1] + Path2[::-1] # Path2 is reversed
```

Figure 2.16: Combining two paths.

2.6 Best First Search

Up to now, all the search algorithms we have discussed have been essentially blind. Given a state s and all of its neighbours, they had no idea which of the neighbours they should pick because they had no conception which of these neighbours might be more promising than the other neighbours. Search algorithms that know nothing about the distance of a state to the goal are called blind. Russell and Norvig [RN20] use the name uninformed search instead of blind search.

If a human tries to solve a search problem, she will usually develop an intuition that certain states are more favourable than other states because they seem to be closer to the solution. In order to formalise this procedure, we next define the notion of a search heuristic.

Definition 2 (Search Heuristic) Given a search problem

$$\mathcal{P} = \langle Q, \mathtt{next_states}, \mathtt{start}, \mathtt{goal} \rangle$$
,

a search heuristic or simply heuristic is a function

$$h: Q \to \mathbb{R}$$

that computes an approximation of the distance of a given state s to the state goal. The heuristic is admissible if it never overestimates the true distance, i.e. if the function

$$d:Q\to\mathbb{N}$$

computes the true distance from a state s to the goal, then we must have

$$h(s) \le d(s)$$
 for all $s \in Q$.

Hence, the heuristic is admissible iff it is optimistic: Although it never overestimates the distance to the goal, it is free to underestimate this distance.

Finally, the heuristic h is called consistent iff we have

$$h(\texttt{goal}) = 0$$
 and $h(s_1) \le 1 + h(s_2)$ for all $s_2 \in \texttt{next_states}(s_1)$.

Let us explain the idea behind the notion of consistency. First, if we are already at the goal, the heuristic should notice this and therefore we need to have h(goal) = 0. Secondly, assume we are at the state s_1 and s_2 is a neighbour of s_1 , i.e. we have that

$$s_2 \in \texttt{next_states}(s_1)$$
.

Now if our heuristic h assumes that the distance of s_2 from the goal is $h(s_2)$, then the distance of s_1 from the goal can be at most $1 + h(s_2)$ because starting from s_1 we can first go to s_2 in one step and then from s_2 to goal in $h(s_2)$ steps for a total of $1 + h(s_2)$ steps. Of course, it is possible that there exists a shorter path from s_1 leading to the goal than the one that visits s_2 first. Hence, we have the inequality

$$h(s_1) < 1 + h(s_2)$$
.

The Figure 2.17 on page 28 demonstrates this equation.



Figure 2.17: Visual demonstration of a heuristic function

Theorem 3 Every consistent heuristic is an admissible heuristic.

Proof: Assume that the heuristic h is consistent. Assume further that $s \in Q$ is some state such that there is a path p from s to the goal. Assume this path has the form

$$p = [s_n, s_{n-1}, \dots, s_1, s_0],$$
 where $s_n = s$ and $s_0 = \text{goal}.$

Then the length of the path p is n and we have to show that $h(s) \leq n$. In order to prove this claim, we show that we have

$$h(s_k) \le k$$
 for all $k \in \{0, 1, \dots, n\}$.

This claim is shown by induction on k.

B.C.: k = 0.

We have $h(s_0) = h(goal) = 0 \le 0$, because the fact that h is consistent implies h(goal) = 0.

I.S.: $k \mapsto k+1$.

We have to show that $h(s_{k+1}) \leq k+1$ holds. This is shown as follows:

$$h(s_{k+1}) \le 1 + h(s_k)$$
 because $s_k \in \texttt{next_states}(s_{k+1})$ and h is consistent,
 $\le 1 + k$ because $h(s_k) \le k$ by induction hypotheses.

It is natural to ask whether the last theorem can be reversed, i.e. whether every admissible heuristic is also consistent. The answer to this question is negative since there are some *contorted* heuristics that are admissible but that fail to be consistent. However, in practice it turns out that most admissible heuristics are also consistent. Therefore, when we construct consistent heuristics later, we will start with admissible heuristics, since these are easy to find. We will then have to check that these heuristics are also consistent.

Examples: In the following, we will discuss several heuristics for the sliding puzzle.

1. The simplest heuristic that is admissible is the function h(s) := 0. Since we have

$$0 < 1 + 0$$
,

this heuristic is obviously consistent, but when we use this heuristic, we are back to blind search.

2. The next heuristic is the number of misplaced tiles heuristic. For a state s, this heuristic counts the number of tiles in s that are not in their final position, i.e. that are not in the same position as the corresponding tile in goal. For example, in Figure 2.4 on page 13 in the state depicted to the left, only the tile with the label 4 is in the same position as in the state depicted to the right. Hence, there are 7 misplaced tiles.

As every misplaced tile must be moved at least once and every step in the sliding puzzle moves at most one tile, it is obvious that this heuristic is admissible. It is also consistent. First, the goal has no misplaced tiles, hence its heuristic is 0. Second, in every step of the sliding puzzle only one tile is moved. Therefore the number of misplaced tiles in two neighbouring states can differ by at most one and hence the inequality

$$h(s_1) \le 1 + h(s_2)$$

is satisfied for any neighbouring states s_1 and s_2 . Unfortunately, the number of misplaced tiles heuristic is very crude and therefore not particularly useful.

3. The Manhattan heuristic improves on the previous heuristic. For two points $\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle \in \mathbb{R}^2$ the Manhattan distance of these points is defined as

```
d_1(\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle) := |x_2 - x_1| + |y_2 - y_1|.
```

The Manhattan distance is also called the L_1 norm of the difference vector $\langle x_2 - x_1, y_2 - y_1 \rangle$. If we associate Cartesian coordinates with the tiles of the sliding puzzle such that the tile in the upper left corner has coordinates $\langle 1, 1 \rangle$ and the coordinates of the tile in the lower right corner are $\langle 3, 3 \rangle$, then the Manhattan distance of two positions measures how many steps it takes to move a tile from the first position to the second position if we are allowed to move the tile horizontally or vertically regardless of the fact that the intermediate positions might be blocked by other tiles. To compute the Manhattan heuristic for a state s with respect to the goal, we first define the function pos(t,s) for all tiles $t \in \{1, \dots, 8\}$ in a given state s as follows:

$$pos(t, s) = \langle row, col \rangle \stackrel{\text{def}}{\iff} s[row][col] = t,$$

i.e. given a state s, the expression pos(t, s) computes the Cartesian coordinates of the tile t with respect to the state s. Then we can define the Manhattan heuristic h for the 3×3 puzzle as follows:

$$h(s) := \sum_{t=1}^{8} d_1(\mathsf{pos}(t,s),\,\mathsf{pos}(t,\mathsf{goal})).$$

The Manhattan heuristic measures the number of moves that would be needed if we wanted to put every tile of s into its final position and if we were allowed to slide tiles over each other. Figure 2.18 on page 30 shows how the Manhattan distance can be computed. The code given in that figure works for a general $n \times n$ sliding puzzle. It takes two states stateA and stateB and computes the Manhattan distance between these states.

```
def manhattan(stateA, stateB):
         n = len(stateA)
2
         result = 0
3
         for rowA in range(n):
             for colA in range(n):
                 tile = stateA[rowA][colA]
                 if tile != 0:
7
                     rowB, colB = find_tile(tile, stateB)
8
                      result += abs(rowA - rowB) + abs(colA - colB)
9
         return result
10
```

Figure 2.18: The Manhattan distance between two states.

- (a) First, the size n of the puzzle is computed by checking the number of rows of stateA.
- (b) Next, the for loops iterates over all rows and columns of stateA that do not contain a blank tile. Remember that the blank tile is coded using the number 0. The tile at position (rowA, colA) in stateA is computed using the expression stateA[rowA][colA] and the corresponding position (rowB, colB) of this tile in state stateB is computed using the function find_tile.
- (c) Finally, the Manhattan distance between the two positions (rowA, colA) and (rowB, colB)

is added to the result.

The Manhattan heuristic is admissible. The reason is that if $s_2 \in \texttt{next_states}(s_1)$, then there can be only one tile t such that the position of t in s_1 is different from the position of t in s_2 . Furthermore, this position differs by either one row or one column. Therefore,

$$|h(s_1) - h(s_2)| = 1$$

and hence $h(s_1) \le 1 + h(s_2)$.

Now we are ready to present best first search. This algorithm is derived from the stack based version of depth first search. However, instead of using a stack, the algorithm uses a priority queue. In this priority queue, the paths are ordered with respect to the estimated distance of the state at the end of the path from the goal. We always expand the path next that seems to be closest to the goal.

In Python the module heapq provides priority queues that are implemented as heaps. Technically, these heaps are just lists. In order to use them as priority queues, the entries of these lists will be pairs of the form (p, o), where p is the priority of the object o. Usually, the priorities are numbers and, contra-intuitively, high priorities correspond to **small** numbers, that is (p_1, o_1) has a higher priority than (p_2, o_2) if and only if $p_1 < p_2$. We need only two functions from the module heapq:

- 1. Given a heap H, the function heapq.heappop(H) returns and removes the pair from H that has the highest priority.
- 2. Given a heap H, the function heapq.heappush(H,(p,o)) pushes the pair (p,o) onto the heap H. This method does not return a value. Instead, the heap H is changed in place.

```
def search(start, goal, next_states, heuristic):
         PrioQueue = [ (heuristic(start, goal), [start]) ]
2
         while PrioQueue:
             _, Path = heapq.heappop(PrioQueue)
                     = Path[-1]
             state
5
             if state == goal:
6
                 return Path
             for ns in next_states(state):
8
                 if ns not in Path:
9
                      d = heuristic(ns, goal)
10
                      heapq.heappush(PrioQueue, (d, Path + [ns]))
11
```

Figure 2.19: The best first search algorithm.

The function search shown in Figure 2.19 on page 31 takes four parameters. The first three of these parameters are the same as in the previous search algorithms. The last parameter heuristic is a function that takes two states and then estimates the distance between these states. Later, we will use the Manhattan distance to serve as the parameter heuristic. The details of the implementation are as follows:

1. The variable PrioQueue serves as a priority queue. This priority queue is initialized as a list containing the pair $\langle d, [\mathtt{start}] \rangle$, where d is the estimated distance of a path leading from \mathtt{start}

to goal. In PrioQueue we store the paths in pairs of the form

```
⟨estimate, Path⟩,
```

where Path is a list of states starting from the node start. If the last node on this list is called state, then we have

```
estimate = heuristic(state, goal),
```

i.e. estimate is the estimated distance between this state and goal and hence an estimate of the number of steps needed to complete Path into a solution. This ensures, that the best path, i.e. the path whose last state is nearest to the goal is at the beginning of the set PrioQueue.

2. As long as PrioQueue is not empty, we take the Path from the beginning of this priority queue and remove it from the queue. The state at the end of Path is named state.

The function heappop(PrioQueue) returns the smallest pair from PrioQueue and, furthermore, this pair is removed from PrioQueue.

- 3. Next, we inspect all neighbouring states of state that have not already occurred in the Path leading to state.
- 4. If state is the goal, a solution has been found and is returned.
- 5. Otherwise, the states reachable from state are inserted into the priority queue. When these states are inserted, we have to compute their estimated distance to goal since this distance is used as the priority in PrioQueue.

Best first search solves the instance of the 3×3 puzzle shown in Figure 2.4 on page 13 in 5 milliseconds and uses 44 kilobytes to do so. However, the solution that is found takes 75 steps. While this is not as ridiculous as the solution found by depth first search, it is still far from an optimal solution. Best first search is able to solve the 4×4 puzzle shown in Figure 2.23 on page 41 in 12 seconds. However, it needs 426 megabytes to do so.

It should be noted that the fact that the Manhattan distance is a consistent heuristic is of no consequence for best first search. Only the A* algorithm, which is presented next, makes use of this fact.

2.7 A* Search

We have seen that best first search can be very fast. However, the solution returned by best first search is not optimal. Next, we describe the A^* search algorithm, which also uses a heuristic. If the heuristic used with the A^* search algorithm is consistent, then the path that is computed by A^* search is guaranteed to be a shortest path.

The A* search algorithm works similar to best first search, but instead of using the heuristic as the priority, the priority f(s) of every state s is given as

$$f(s) := q(s) + h(s),$$

where g(s) computes the length of the path leading from start to s and h(s) is the heuristical estimate of the distance from s to goal. Hence, f(s) is the estimate of the total distance that a path connecting start and goal via the state s would use. The details of the A* algorithm are given in Figure 2.20 on page 33. We discuss the implementation below.

The function search takes 4 parameters:

1. start is a state. This state represents the start state of the search problem.

```
from Set import Set
2
     def search(start, goal, next_states, heuristic):
3
                  = { start:start }
4
         Distance = { start: 0 }
5
         estGoal = heuristic(start, goal)
6
         Estimate = { start: estGoal }
         Frontier = Set()
         Frontier.insert( (estGoal, start) )
         while Frontier:
10
             estimate, state = Frontier.pop()
11
             if state == goal:
12
                 return path_to(state, Parent)
13
             stateDist = Distance[state]
14
             for ns in next_states(state):
                 oldEstimate = Estimate.get(ns, None)
16
                 newEstimate = stateDist + 1 + heuristic(ns, goal)
17
                 if oldEstimate is None or newEstimate < oldEstimate:
18
                      Distance[ns] = stateDist + 1
19
                      Estimate[ns] = newEstimate
20
                      Parent[ns]
                                   = state
21
                      Frontier.insert( (newEstimate, ns) )
22
                      if oldEstimate is not None:
                          Frontier.delete( (oldEstimate, ns) )
24
```

Figure 2.20: The A* search algorithm.

- 2. goal is the goal state.
- 3. next_states is a function that takes a state as a parameter. For a state s,

```
next_states(s)
```

computes the set of all those states that can be reached from s in a single step.

4. heuristic is a function that takes two parameters. For two states s_1 and s_2 , the expression

```
heuristic(s_1, s_2)
```

computes an estimate of the distance between s_1 and s_2 .

The function search maintains 4 variables that are crucial for the understanding of the algorithm.

1. Parent is a dictionary associating a parent state with those states that have already been encountered during the search, i.e. we have

```
Parent[s_2] = s_1 \Rightarrow s_2 \in next\_states(s_1).
```

The only exception is the state start: We have Parent[start] = start instead.

Once the goal has been found, the dictionary Parent is used to compute the path from start to goal.

2. Distance is a dictionary. For every state s that has been encountered so far, this dictionary records the length of the shortest path from start to s. Initially, Distance is only defined for the state start.

3. Estimate is a dictionary. For every state s encountered in the search, Estimate[s] is an estimate of the length that a path from start to goal would have if it would pass through the state s. This estimate is calculated using the equation

```
\texttt{Estimate}[s] = \texttt{Distance}[s] + \texttt{heuristic}(s, \texttt{goal}).
```

Instead of recalculating this sum every time we need it, we store it in the dictionary Estimate. This increases the efficiency of the algorithm.

4. Frontier is a priority queue. The elements of Frontier are pairs of the form

```
(d,s) such that d = \text{Estimate}[s],
```

i.e. if $(d, s) \in \text{Frontier}$, then the state s has been encountered in the search and it is estimated that a path leading from start to goal and passing through s would have a length of d.

Unfortunately, we can not use the module heapq to implement this priority queue, because heapq does not provide a function that removes an element from the queue. Therefore, I was forced to create my own version of a priority queue. I have implemented this version of priority queues in the module Set, which is imported in line 1. The API of this module provides the following functions:

- (a) Set() creates an empty priority queue.
- (b) S.isEmpty() checks whether the priority queue S is empty.
- (c) S.member(x) checks whether x is an element of the priority queue S.
- (d) S.insert(x) inserts x into the priority queue S. This does not return a new priority queue but rather modifies the priority queue S.
- (e) S.delete(x) deletes x from the priority queue S. This does not return a new priority queue but rather modifies the priority queue S.
- (f) S.pop() returns the smallest element of the priority queue S. Furthermore, this element is removed from S.

Since priority queues are implemented as ordered binary trees, the elements of a priority queue need to be comparable, i.e. if x and y are inserted into a priority queue, then the expression x < y must return a Boolean value and the relation < has to define linear order. This implies that the elements inserted into the priority queue must be homogeneous, i.e. they must all be of the same type.

After the variables described above have been initialized, the A* algorithm runs in a while loop that does only terminate if either a solution is found or the priority queue Frontier is exhausted and hence the state goal is not reachable from the state start.

1. First, the state with the smallest estimated distance for a path connecting start with goal and passing through state is chosen from the priority queue Frontier. Note that the call of the function pop does not only return the pair

```
(estimate, state)
```

from Frontier that has the lowest value of estimate, but also removes this pair from the priority queue Frontier.

2. Now if this state is the goal, then a solution has been found. The path corresponding to this solution is computed by the function path_to, which utilizes the dictionary Parent. The resulting path is then returned and the function terminates.

- 3. Otherwise, we retrieve the length of the path leading from start to state from the dictionary Distance and store this length in the variable stateDist. We will later prove that stateDist is not just the length of some path connecting start and state but rather the length of the shortest path connecting these two states.
- 4. Next, we have a for loop that iterates over all states that can be reached in one step from state.
 - (a) For every neighbour ns of state we check the estimated length of a solution passing through ns and store this length in oldEstimate. Note that oldEstimate is undefined, i.e. it has the value None, if we haven't yet encountered the node ns in our search.
 - (b) If a solution connects start with goal while passing first through state and then through ns, the estimated length of this solution would be

```
stateDist + 1 + heuristic(ns, goal).
```

Therefore this value is stored in newEstimate.

(c) Next, we need to check whether this new solution that first passes through state and then proceeds to ns is better than the previous solution that passes through ns. This check is done by comparing newEstimate and oldEstimate. Note that we have to take care of the fact that there might be no valid oldEstimate, i.e. the variable oldEstimate might have the value None.

In case the new solution seems to be better than the old solution, we have to update the Parent dictionary, the Distance dictionary, and the Estimate dictionary. Furthermore, we have to update the priority queue Frontier. Here, we have to take care to remove the previous entry for the state ns if it exists, which is the case if oldEstimate is different from None.

The A* search algorithm has been discovered by Hart, Nilsson, and Raphael and was first published in 1968 [HNR68]. However, there was a subtle bug in the first publication which was corrected in 1972 [HNR72].

When we run A^* search on the 3×3 sliding puzzle, it takes about half a second to solve the instance shown in Figure 2.4 on page 13. If we just look at the time, this looks worse than best first search. However, the comparison is a bit unfair, because the module heapq that is used in best first search has been implemented in C, while the module Set is implemented in Python. Furthermore, the good news about A^* search is that:

- 1. The path which is found is optimal.
- 2. Only 10,061 states are touched in the search for a solution. This is more than a tenfold reduction when compared with breadth first search. As a result, the A^* search algorithm uses only 6 megabytes of memory.

2.7.1 Completeness and Optimality of A* Search

In order to prove the completeness and the optimality of the A^* search algorithm it is convenient to reformulate the algorithm: In particular, we replace the dictionary Parent by a dictionary PathDict. For every state s that is reached during A^* search, PathDict[s] returns a path that connects the node start with the node s. Figure 2.21 on page 36 shows this version of the algorithm.

2.7. A* Search Chapter 2. Search

```
def search(start, goal, next_states, heuristic):
         PathDict = { start: [start] }
2
         Distance = { start: 0 }
3
         estGoal = heuristic(start, goal)
4
         Estimate = { start: estGoal }
         Frontier = Set()
6
         Frontier.insert( (estGoal, start) )
         while Frontier:
             _, state = Frontier.pop()
             if state == goal:
10
                 return PathDict[state]
11
             stateDist = Distance[state]
12
             for ns in next_states(state):
13
                 oldEstimate = Estimate.get(ns, None)
14
                 newEstimate = stateDist + 1 + heuristic(ns, goal)
15
                 if oldEstimate is None or newEstimate < oldEstimate:
16
                      Distance[ns] = stateDist + 1
17
                      Estimate[ns] = newEstimate
18
                      PathDict[ns] = PathDict[state] + [ns]
19
                      Frontier.insert( (newEstimate, ns) )
20
                      if oldEstimate is not None:
^{21}
                          Frontier.delete( (oldEstimate, ns) )
22
```

Figure 2.21: A path based implementation of A* search.

Theorem 4 (Completeness and Optimality of A* Search) Assume

```
\mathcal{P} = \langle Q, \mathtt{next\_states}, \mathtt{start}, \mathtt{goal} \rangle
```

is a search problem and

```
heuristic: Q \to \mathbb{N}
```

is a consistent heuristic for the search problem \mathcal{P} . Then the A^* search algorithm is both complete and optimal, i.e. if there is a path from start to goal, then the search is successful and, furthermore, the solution that is computed is a shortest path leading from start to goal.

Proof: In order to prove this theorem, we first need to define two notions for states.

- A state has been visited iff it has been entered into the dictionary Distance, i.e. a state s is visited iff Distance[s] is defined. Note that in this case also PathDict[s] is defined.
- A state is defined to be explored when it is removed from the priority queue Frontier.

Before we prove the theorem, let us establish the following auxiliary claim:

Claim One: If a state s is visited and P := PathDict[s], then P is a path from start to s and the length of P is equal to Distance[s].

We establish this claim by a straightforward induction.

2.7. A* Search Chapter 2. Search

BC: The first path that is entered into the dictionary PathDict is the path [start]. This path connects the node start with the node start. Obviously, this path has the length 0 and that is exactly the value that is entered in the dictionary Distance in line 3.

IS: If, in line 19, we add the path

$$PathDict[state] + [ns]$$

as a path leading to the state ns, then by our induction hypotheses we know that

$$P := PathDict[state]$$

is a path leading from start to state and that, furthermore, the length of the path P is equal to Distance[state]. When we append the state ns to this path P, the length of the resulting path is the length of P plus one and, because $ns \in next_states(state)$, this new path

$$P + [ns]$$

connects start with ns. The induction hypotheses tell us that Distance[state] is equal to the length of P. Therefore, the length of the new path is Distance[state] + 1 and since stateDist is defined as Distance[state], the correct length is stored in line 17.

This concludes the proof of Claim One.

Another notion we need to introduce is a function g that is defined for all states s and returns the distance of s from the state start, i.e. we have

$$g(s) := \mathtt{dist}(\mathtt{start}, s) \quad \text{ for all } s \in Q.$$

Here, the expression dist(start, s) returns the length of the <u>shortest</u> path from start to s. From the definition of the dictionary Distance and Claim One it is obvious that

$$g(s) \leq \mathtt{Distance}[s].$$

The reason is that every time an entry for a state s is added to the dictionary Distance, we have found a path from start to s that has the length Distance[s]. This might not be the shortest path, hence g(s) might be less than the length of this path.

Next, we introduce the notion of the estimated total distance of a state s that has been visited during the search. The estimated total distance of a state s is written as f(s) and denotes the estimated length of a path starting in start and ending in goal that visits the state s in between. Formally, f(s) is defined as follows:

$$f(s) := Distance[s] + heuristic(s, goal).$$

Here, Distance[s] is the number of steps that it takes to reach the state s from start when following the path stored in PathDict[s], while heuristic(s, goal) is the estimated distance of a path from s to goal. Note that the priority queue Frontier is ordered according to the estimated total distance. The shorter the estimated total distance of a node s is, the higher the priority of s. In order to prove the theorem we need to establish the following claim.

Claim Two: If a state $s \in Q$ is explored, i.e. if it has been removed from the priority queue Frontier, then we have

$$Distance[s] = g(s),$$

i.e. the path leading from start to s is guaranteed to be a shortest path.

Proof of Claim Two: The proof of Claim Two is a proof by contradiction. We assume that s is a state that has been explored such that

2.7. A* Search Chapter 2. Search

i.e. we assume that $P_1 := \mathtt{PathDict}[s]$ is not a shortest path from start to s. Then, there must be another path P_2 from start to s that is shorter than P_1 . Both P_1 and P_2 start at the same state start. Hence, these two paths must have the form

$$P_1 = [\mathtt{start}, x_1, \cdots, x_k, x_{k+1}, \cdots, x_{n-1}, s], \quad P_2 = [\mathtt{start}, x_1, \cdots, x_k, y_{k+1}, \cdots, y_{m-1}, s]$$

where $x_{k+1} \neq y_{k+1}$. Here, x_k is the state at which the paths P_1 and P_2 diverge for the first time. Note that the index k could be 0. In that case the two paths would already diverge at the state start, but in general k will be some non-negative integer. Since we assume that P_1 is longer than P_2 we will have m < n. Furthermore, according to Claim One we have

$$Distance[s] = n.$$

Now it is time to make use of the fact that the heuristic h is consistent. First, since $s \in \texttt{next_states}(y_{m-1})$ we have that

$$heuristic(y_{m-1}) \le 1 + heuristic(s).$$

Next, as we have $y_{m-1} \in \texttt{next_states}(y_{m-2})$ the consistency of heuristic implies

$$heuristic(y_{m-2}) \le 1 + heuristic(y_{m-1}) \le 2 + heuristic(s).$$

Continuing this way we conclude that

$$\text{heuristic}(y_{m-i}) \le i + \text{heuristic}(s) \quad \text{for all } i \in \{1, \dots, m-k-1\}.$$

Define i := m - k - 1. Since m - i = m - (m - k - 1) = k + 1 we have shown that

$$heuristic(y_{k+1}) \le m - k - 1 + heuristic(s).$$

This implies

$$\begin{split} f(y_{k+1}) &= & \operatorname{Distance}[y_{k+1}] + \operatorname{heuristic}(y_{k+1}) \\ &\leq & \operatorname{Distance}[y_{k+1}] + m - k - 1 + \operatorname{heuristic}(s) \\ &= & (k+1) + m - k - 1 + \operatorname{heuristic}(s) & \operatorname{since} \operatorname{Distance}[y_{k+1}] = k + 1 \\ &= & m + \operatorname{heuristic}(s). \end{split}$$

Next, we compute f(s). We have

$$f(s) = Distance[s] + heuristic(s) = n + heuristic(s).$$

However, since we have m < n this implies

$$f(y_{k+1}) \le m + \text{heuristic}(s) < n + \text{heuristic}(s) = f(s).$$

Now f(s) is the priority of the state s in the priority queue, while $f(y_{k+1})$ is the priority attached to the state y_{k+1} . At the latest, the state y_{k+1} is put onto the priority queue Frontier after the state x_k is explored. But then the priority of y_{k+1} is higher than the priority of the state s and hence it is explored prior to s. This implies that the state y_{k+2} is visited before the state s is visited. Similarly to the proof that

$$f(y_{k+1}) \le m + \text{heuristic}(s) < n + \text{heuristic}(s) = f(s),$$

we can see that

$$f(y_{k+2}) \le k+2+(m-k-2)+\text{heuristic}(s)=m+\text{heuristic}(s) < f(s).$$

Hence, the state y_{k+2} is explored before the state s is explored. In general, we have

$$f(y_i) \le m + \text{heuristic}(s) < f(s)$$
 for all $j \in \{k+1, \dots, m-1\}$.

Since all states of the form

$$y_i$$
 for $j \in \{k+1, \dots, m-1\}$

have a priority that is higher than the state s, these states are all explored prior to the state s. In particular, after the state y_{m-1} has been explored, we will have

$$Distance[s] = m$$
 instead of $Distance[s] = n$.

This contradiction proves that our assumption

must be wrong and therefore we know that the equation Distance[s] = g(s) must hold for all states s that have been explored. Hence the validity of Claim Two has been established.

Claim Two implies that if the A* algorithm finds a path from start to goal, then this path must be optimal. This follows from the fact that before we check whether the search has reached the goal, the state that is compared to goal has been explored and, according to Claim Two, must therefore have Distance[goal] = g(goal). As g(goal) is the true distance of the state goal from the state start, this implies that the path that has been found is optimal.

Finally, we have to show that the algorithm is complete, i.e. we have to show that if there is a path connecting start and goal, then we will find a path. Therefore, let us assume that there is a path P connecting start and goal and that, furthermore, this path is shorter than any other such path and has length n. By reasoning analogously to the proof of Claim Two it can be shown that the states making up this path P all have a priority less or equal than n. Therefore, these states will all be explored before a state with a priority greater than n is explored. However, the set of states that have a priority of at most n is finite, because it is a subset of the set of states that have a distance from start that is at most n and this latter set is obviously finite. Therefore, unless the goal is reached before, all the states on the path P will eventually be explored and this implies that the goal is eventually found.

2.8 Bidirectional A* Search

So far, the best search algorithm we have encountered is bidirectional breadth first search. However, in terms of memory consumption, the A* algorithm also looks very promising. Hence, it might be a good idea to combine these two algorithms. Figure 2.22 on page 40 shows the resulting program. This program relates to the A* algorithm shown in Figure 2.20 on page 33 as the algorithm for bidirectional search shown in Figure 2.15 on page 27 relates to breadth first search shown in Figure 2.6 on page 16. The only new idea is that we alternate between the A* search starting from start and the A* search starting from goal depending on the estimated total distance:

- (a) As long as the search starting from start is more promising, we remove states from FrontierA.
- (b) Once the total estimated distance of a path starting from goal is less than the best total estimated distance of a path starting from start, we switch and remove states from FrontierB.

When we run bidirectional A^* search for the 3×3 sliding puzzle shown in Figure 2.4 on page 13, the program takes 150 milliseconds and uses only 2,963 states. Therefore, I have tried to solve the 4×4 sliding puzzle shown in Figure 2.23 on page 41 using bidirectional A^* search. A solution of 40 steps was found in 1.3 seconds. Only 17,626 states had to be processed to compute this solution! As the shortest path connecting start and goal has a length of 36 steps, this shows that bidirectional A^* search does not find the shortest path.

```
def search(start, goal, next_states, heuristic):
         estimate = heuristic(start, goal)
                  = { start: start };
                                             ParentB
                                                        = { goal : goal }
         DistanceA = { start: 0 };
                                             DistanceB = { goal : 0 }
         EstimateA = { start: estimate };
                                             EstimateB = { goal : estimate }
         FrontierA = Set();
                                             FrontierB = Set()
         FrontierA.insert( (estimate, start) )
         FrontierB.insert( (estimate, goal ) )
         while FrontierA and FrontierB:
             guessA, stateA = FrontierA.pop()
10
             guessB, stateB = FrontierB.pop()
11
             stateADist = DistanceA[stateA]
12
             stateBDist = DistanceB[stateB]
13
             if guessA <= guessB:
14
                 FrontierB.insert( (guessB, stateB) )
                 for ns in next_states(stateA):
16
                     oldEstimate = EstimateA.get(ns, None)
17
                     newEstimate = stateADist + 1 + heuristic(ns, goal)
18
                     if oldEstimate is None or newEstimate < oldEstimate:
19
                         ParentA [ns] = stateA
20
                         DistanceA[ns] = stateADist + 1
21
                         EstimateA[ns] = newEstimate
22
                         FrontierA.insert( (newEstimate, ns) )
                          if oldEstimate is not None:
                              FrontierA.delete( (oldEstimate, ns) )
25
                     if DistanceB.get(ns, None) is not None:
26
                         return combinePaths(ns, ParentA, ParentB)
27
             else:
28
                 FrontierA.insert( (guessA, stateA) )
29
                 for ns in next_states(stateB):
30
                     oldEstimate = EstimateB.get(ns, None)
31
                     newEstimate = stateBDist + 1 + heuristic(start, ns)
32
                     if oldEstimate is None or newEstimate < oldEstimate:
33
                         ParentB [ns] = stateB
34
                         DistanceB[ns] = stateBDist + 1
35
                         EstimateB[ns] = newEstimate
36
                         FrontierB.insert( (newEstimate, ns) )
37
                          if oldEstimate is not None:
                              FrontierB.delete( (oldEstimate, ns) )
39
                     if DistanceA.get(ns, None) is not None:
40
                          return combinePaths(ns, ParentA, ParentB)
41
```

Figure 2.22: Bidirectional A^* search.

```
3),
    start = ( (
                0, 1,
                        2,
1
             (4, 5, 6, 8),
2
             (14, 7, 11, 10),
3
                9, 15, 12, 13)
4
5
         = ( (
                0, 1, 2, 3),
6
                4, 5, 6, 7),
7
             (8, 9, 10, 11),
8
             (12, 13, 14, 15)
9
           )
10
```

Figure 2.23: A start state and a goal state for the 4×4 sliding puzzle.

2.9 Iterative Deepening A* Search

So far, we have combined A* search with bidirectional search and achieved good results. When memory space is too limited for bidirectional A* search to be possible, we can instead combine A* search with iterative deepening. The resulting search technique is known as iterative deepening A* search and is commonly abbreviated as IDA search. It has been invented by Richard Korf [Kor85]. Figure 2.24 on page 42 shows an implementation of IDA search. We proceed to discuss this program.

- 1. As in the A* search algorithm, the function search takes four parameters.
 - (a) start is a state. This state represents the start state of the search problem.
 - (b) goal is the goal state.
 - (c) next_states is a function that takes a state s as a parameter and computes the set of all those states that can be reached from s in a single step.
 - (d) heuristic is a function that takes two parameters s_1 and s_2 , where s_1 and s_2 are states. The expression

```
heuristic(s_1, s_2)
```

computes an estimate of the distance between s_1 and s_2 . In IDA search it is required that this estimate is optimistic, i.e. the heuristic has to be admissible.

Note that this is different from A^* search. A^* search requires that the heuristic is consistent.

- 2. The function search initializes limit to be an estimate of the distance between start and goal. As we assume that the function heuristic is optimistic, we know that there is no path from start to goal that is shorter than limit. Hence, we start our search by assuming that we might find a path that has a length of limit.
- 3. Next, we start a while loop. In this loop, we call the function dl_search (depth limited search) to compute a path from start to goal that has a length of at most limit. The function dl_search is described in detail below. When the function dl_search returns, there are two cases:
 - (a) dl_search does find a path. In this case, this path is returned in the variable Path and the value of this variable is a list. This list is returned as the solution to the search problem.

```
def search(start, goal, next_states, heuristic):
         limit = heuristic(start, goal)
2
         while True:
3
             Path = dl_search([start], goal, next_states, limit, heuristic)
             if isinstance(Path, list):
                 return Path
             limit = Path
     def dl_search(Path, goal, next_states, limit, heuristic):
                  = Path[-1]
10
         distance = len(Path) - 1
11
                  = distance + heuristic(state, goal)
12
         if total > limit:
13
             return total
14
         if state == goal:
             return Path
16
         smallest = float("Inf") # infinity
17
         for ns in next_states(state):
18
             if ns not in Path:
19
                 Solution = dl_search(Path+[ns], goal, next_states, limit, heuristic)
20
                 if isinstance(Solution, list):
21
                     return Solution
22
                 smallest = min(smallest, Solution)
         return smallest
```

Figure 2.24: Iterative deepening A^* search.

(b) dl_search is not able to find a path within the given limit. In this case, dl_search will not return a list representing a path, but instead it will return a number. This number will specify the minimal length that any path leading from start to goal needs to have. This number is then used to update the limit which is used for the next invocation of dl_search.

Note that the fact that dl_search is able to compute this new limit is a significant enhancement over iterative deepening. While we had to test every single possible length in iterative deepening, now the fact that we can intelligently update the limit results in a considerable saving of computation time.

We proceed to discuss the function dl_search. This function takes 5 parameters, which we describe next.

- 1. Path is a list of states. This list starts with the state start. If state is the last state on this list, then Path represents a path leading from start to state.
- 2. goal is another state. The purpose of the recursive invocations of dl_search is to find a path from state to goal, where state is the last element of the list Path.
- 3. next_states is a function that takes a state s as input and computes the set of states that are reachable from s in one step.

- 4. limit is the upper bound for the length of the path from start to goal. If the function dl_search is not able to find a path from start to goal that has a length of at most limit, then the search is unsuccessful. In that case, instead of a path the function dl_search returns a new estimate for the distance between start and goal. Of course, this new estimate will be bigger than limit.
- 5. heuristic is a function taking two states as arguments. The invocation heuristic(s_1, s_2) computes an estimate of the distance between the states s_1 and s_2 . It is assumed that this estimate is optimistic, i.e. the value returned by heuristic(s_1, s_2) is less than or equal to the true distance between s_1 and s_2 .

We proceed to describe the implementation of the function dl_search

- 1. The variable state is assigned the last element of Path. Hence, Path connects start and state.
- 2. The length of the path connecting start and state is stored in distance.
- 3. Since heuristic is assumed to be optimistic, if we want to extend Path, then the best we can hope for is to find a path from start to goal that has a length of

This length is computed and saved in the variable total.

- 4. If total is bigger than limit, it is not possible to find a path from start to goal passing through state that has a length of at most limit. Hence, in this case we return total to communicate that the limit needs to be increased to have at least a value of total.
- 5. If we are lucky and state is equal to goal, the search is successful and Path is returned.
- 6. Otherwise, we iterate over all nodes ns reachable from state that have not already been visited by Path. If ns is a node of this kind, we extend the Path so that this node is visited next. The resulting path has the form

Next, we recursively start a new search starting from the node ns. If this search is successful, the resulting path is returned. Otherwise, the search returns the minimum distance that is needed to reach the state goal from the state start on a path via the state ns. If this distance is smaller than the distance smallest that is returned from visiting previous neighbouring nodes, the variable smallest is updated accordingly. This way, if the for loop is not able to return a path leading to goal, the variable smallest contains a lower bound for the distance that is needed to reach goal by a path that extends the given Path.

Note: At this point, a natural question is to ask whether the for loop should collect all paths leading to goal and then only return the path that is shortest. However, this is not necessary: Every time the function dl_search is invoked it is already guaranteed that there is no path that is shorter than the parameter limit. Therefore, if dl_search is able to find a path that has a length of at most limit, this path is known to be optimal.

Iterative deepening A* is a complete search algorithm that does find an optimal path, provided that the employed heuristic is optimistic. On the instance of the 3×3 sliding puzzle shown on Figure 2.4 on page 13, this algorithm takes about 0.8 seconds to solve the puzzle. Only about 170 kilobytes of memory are necessary for this search. For the 4×4 sliding puzzle shown in Figure 2.23, the algorithm takes about 2.5 seconds and uses 184 kilobytes. Although this is more than the time needed by

bidirectional A* search, the good news is that the IDA* algorithm does not need much memory since basically only the path discovered so far is stored in memory. Hence, IDA* is a viable alternative if the available memory is not sufficient to support the bidirectional A* algorithm.

2.10 A^* -IDA* Search

So far, of all the search algorithms we have tried, the bidirectional A* search has performed best. However, bidirectional A* search is only feasible if sufficient memory is available. While IDA* requires more time, its memory consumption is much lower than the memory consumption of bidirectional A*. Hence, it is natural to try to combine the A* algorithm and the IDA* algorithm. Concretely, the idea is to run an A* search from the start node until memory is more or less exhausted. Then, we start IDA* from the goal node and search until we find any of the nodes discovered by the A* search that had been started from the start node.

```
def search(start, goal, next_states, heuristic, size):
                  = { start:start }
         Parent
         Distance = { start: 0 }
         estGoal = heuristic(start, goal)
        Estimate = { start: estGoal }
        Frontier = Set()
        Frontier.insert( (estGoal, start) )
         while len(Distance) < size and Frontier:
             estimate, state = Frontier.pop()
             if state == goal:
10
                 return path_to(state, Parent)
11
             stateDist = Distance[state]
12
             for ns in next_states(state):
13
                 oldEstimate = Estimate.get(ns, None)
14
                 newEstimate = stateDist + 1 + heuristic(ns, goal)
                 if oldEstimate is None or newEstimate < oldEstimate:
16
                     Distance[ns] = stateDist + 1
17
                     Estimate[ns] = newEstimate
18
                     Parent[ns]
                                   = state
19
                     Frontier.insert( (newEstimate, ns) )
20
                     if oldEstimate is not None:
21
                         Frontier.delete( (oldEstimate, ns) )
22
        Path = id_search(goal, start, next_states, heuristic, Distance)
         return path_to(Path[-1], Parent) + Path[::-1][1:]
```

Figure 2.25: The A*-IDA* search algorithm, part I.

An implementation of the A*-IDA* algorithm is shown in Figure 2.25 on page 44 and Figure 2.26 on page 46. We begin with a discussion of the function search.

- 1. The function search takes 5 arguments.
 - (a) start and goal are states. The function tries to find a path connecting start and goal.

(b) next_states is a function that takes a state s as input and computes the set of states that are reachable from s in one step.

- (c) heuristic computes an estimate of the distance between s_1 and s_2 . It is assumed that this estimate is both admissible and consistent.
- (d) size is the maximal number of states that the A* search is allowed to explore before the algorithm switches over to IDA* search.
- 2. The basic idea behind the A*-IDA* algorithm is to first use A* search to find a path from start to goal. If this is successfully done without visiting more than size nodes, the algorithm terminates and returns the path that has been found. Otherwise, the algorithm switches over to an IDA* search that starts from goal and tries to connect goal to any of the nodes that have been encountered during the A* search. To this end, the function search maintains the following variables.
 - (a) Parent is a dictionary associating a parent state with those states that have already been encountered during the search, i.e. we have

$$Parent[s_2] = s_1 \Rightarrow s_2 \in next_states(s_1).$$

Once the goal has been found, this dictionary is used to compute the path from start to goal.

- (b) Distance is a dictionary that remembers for every state s that is encountered during the A^* search the length of the shortest path from start to s.
- (c) Estimate is a dictionary. For every state s encountered in the A^* search, Estimate [s] is an estimate of the length that a path from start to goal would have if it would pass through the state s. This estimate is calculated using the equation

$$Estimate[s] = Distance[s] + heuristic(s, goal).$$

Instead of recalculating this sum every time we need it, we store the sum in the dictionary Estimate.

(d) Frontier is a priority queue. The elements of Frontier are pairs of the form

```
(d,s) such that d = \text{Estimate}[s],
```

i.e. if $(d, s) \in Frontier$, then the state s has been encountered in the A* search and it is estimated that a path leading from start to goal and passing through s would have a length of d. This priority queue is implemented as an ordered set.

- 3. The A* search runs exactly as discussed in section 2.7. The only difference is that the while loop is terminated once the dictionary Distance has more than size entries. If we are lucky, the A* search is already able to find the goal and the algorithm terminates.
- 4. Otherwise, the function id_search is called. This function starts an iterative deepening A* search from the node goal. This works as described in section 2.9. This search terminates as soon as a state is found that has already been encountered during the A* search. The set of these nodes is given to the function id_search via the dictionary Distance. The function id_search returns a path leading from goal to a state s that is a key of the dictionary Distance and hence has been reached by the A* search. In order to compute a path from start to goal, we still have to compute a path from start to s. This path is then combined with the path P and the resulting path is returned.

The expression Path[::-1] in line 24 reverses the list Path and hence Path[::-1] is a path leading from the state s to the state goal. Path[::-1][-1] is the same path but without the

state s, as this state is already included as the last state of the path returned by the expression path_to(Path[-1], Parent).

```
def id_search(goal, start, next_states, heuristic, Distance):
1
         limit = 0
2
         while True:
             Path = dl_search([goal], start, next_states, limit, heuristic, Distance)
             if isinstance(Path, list):
                 return Path
             limit = Path
    def dl_search(Path, start, next_states, limit, heuristic, Distance):
9
         state = Path[-1]
10
         total = len(Path) - 1 + heuristic(state, start)
11
         if total > limit:
12
             return total
13
         if state in Distance:
             return Path
15
         smallest = float('Inf')
16
         for ns in next_states(state):
17
             if ns not in Path:
18
                 result = dl_search( Path + [ns], start, next_states, limit,
19
                                      heuristic, Distance
20
                 if isinstance(result, list):
                      return result
23
                 smallest = min(smallest, result)
24
         return smallest
25
```

Figure 2.26: The A*-IDA* search algorithm, part II.

Iterative deepening A*-IDA* is a complete search algorithm. On the instance of the 3×3 sliding puzzle shown on Figure 2.4 on page 13, this algorithm takes about 0.14 seconds to solve the puzzle. For the 4×4 sliding puzzle, if the algorithm is allowed to visit at most 3000 states, the algorithm takes less than 1.33 seconds. A*-IDA* is not optimal. For example the solution computed for the 4×4 sliding puzzle has a length of 40, while the shortest solution for the given problem only requires 36 steps.

Exercise 3: The eight queens puzzle is the problem of placing eight chess queens on a chessboard so that no two queens can attack each other. In chess, a queen can attack by moving horizontally, vertically, or diagonally.

- (a) Reformulate the eight queens puzzle as a search problem.
- (b) Compute an upper bound for the number of states.
- (c) Which of the algorithms we have discusses are suitable to solve this problem?

(d) Compute all 92 solutions of the eight queens puzzle.

Hint: It is easiest to encode states as lists. For example, the solution of the eight queens puzzle that is shown in Figure 2.27 would be represented as the list

because the queen in the first row is positioned in column 6, the queen in the second row is positioned in column 4, and so on. The start state would then be the empty list and given a state L, all states from the set nextState(L) would be lists of the form L + [c]. If #L = k, then the state l + [c] has k+1 queens, where the queen in row k+1 has been placed in column c. A frame for solving this problem is available at $\texttt{https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/1 Search/N-Queens-Frame.ipynb.$



Figure 2.27: A solution of the eight queens puzzle.

Exercise 4: The founder of Taoism, the Chinese philosopher Laozi once said:

"A journey of a thousand miles begins but with a single step".

This proverb is the foundation of taoistic search. The idea is, instead of trying to reach the goal directly, we rather define some intermediate states which are easier to reach than the goal state and that are nearer to the goal than the start state. To make this idea more precise, consider the following instance of the 15-puzzle, where the states Start and Goal are given as follows:

Start :=	++	Goal :=	++
	15 14 8		1 2 3
	++		++
	12 10 11 13		4 5 6 7
	++		++
	9 6 2 5		8 9 10 11
	++		++
	1 3 4 7		12 13 14 15
	++		++

In order to solve this instance of the 15-puzzle, we could try to first move the tiles numbered with 14 and 15 into the lower right corner. The resulting state would have the following form:

+-		+-		+-		-+-		+
	*		*		*		*	
+-		+-		+-		-+-		+
	*		*		*		*	1
+-		+-		+-		-+-		-+
1	*		*	I	*		*	1
+-		+-		+-		-+-		-+
	*		*	1:	14	:	15	
+-		-+-		-+-		-+-		-+

Here, the character "*" is used as a wildcard character, i.e. we do not care about the actual character in the state, for we only want to ensure that the first two tiles are positioned correctly. Once we have reached a state specified by the pattern given above, we could then proceed to reach a state that is described by the following pattern:

```
+---+--+--+

| * | * | * | * | * |

+---+---+---+

| * | * | * | * |

+---+---+---+

| 12 | 13 | 14 | 15 |

+---+---+---+
```

We have now solved the bottom line of the puzzle. In a similar way, we can try to solve the line above the bottom line. After that, the next step would then be to reach a goal of the form

+	+	+	+	+
*	*	1 2	3	
+	+	+	+	+
*	*	6	7	١
+	+	+	+	+
8	9	10	11	١
+	+	+	+	+
12	13	14	15	
+	+	+	+	+

The final step would then solve the puzzle. I have prepared a framework for taoistic search. The file

Python/1 Search/Taoistic-Search-Frame.ipynb

from my github repository at https://github.com/karlstroetmann/Artificial-Intelligence contains a framework to solve the sliding puzzle using taoistic search where some functions are left unimplemented. Your task is to implement the missing functions in this file and thereby solve the puzzle.

Chapter 3

Solving Constraint Satisfaction Problems

In this chapter we discuss various algorithms for solving constraint satisfaction problems. In a constraint satisfaction problem we are given a set of formulas and search for values that can be assigned to the variables occurring in these formulas so that all of the formulas evaluate as true. Constraint satisfaction problems can be seen as a refinement of the search problems discussed in the previous chapter: In a search problem, the states are abstract and therefore have no structure that can be exploited to guide the search, while in a constraint satisfaction problem, the states have a structure, as these states are variable assignments. This structure can be exploited to guide the search. This chapter is structured as follows:

- (a) The first section defines the notion of a constraint satisfaction problem. In order to illustrate this notion, we present two examples: map colouring and the eight queens puzzle. After that, we discuss applications of constraint satisfaction problems.
- (b) The simplest algorithm to solve a constraint satisfaction problem is via brute force search. The idea behind *brute force search* is to test all possible variable assignments.
- (c) In most cases, the search space is so large that it is not feasible to enumerate all variable assignments. Backtracking search improves on brute force search by mixing the generation of variable assignments with the testing of the constraints. This approach improves the performance of the brute search algorithm drastically.
- (d) Backtracking search can be refined by using both constraint propagation and the most restricted variable heuristic.
- (e) Furthermore, checking the consistency of the values assigned to different variables can reduce the size of the search space considerably.
- (f) Local search is a completely different approach to solve constraint satisfaction problems. This approach is especially useful if the constraint satisfaction problem is huge, but not complicated.
- (g) Finally, we discuss the theorem prover Z3, which can be used as an industrial strength constraint solver. Here, a *constraint solver* is a piece of software that takes a *constraint satisfaction problem* as input and returns a *solution* of this problem.

When we have finished our discussion of constraint satisfaction problems, we will have implemented a constraint solver that is able to solve instances of the most difficult Sudoku puzzles in seconds.

3.1 Formal Definition of Constraint Satisfaction Problems

Formally, we define a constraint satisfaction problem as a triple

$$\mathcal{P} := \langle \mathtt{Vars}, \mathtt{Values}, \mathtt{Constraints} \rangle$$

where

- (a) Vars is a set of strings which serve as variables,
- (b) Values is a set of values that can be assigned to the variables in Vars.
- (c) Constraints is a set of formulas from first order logic. Each of these formulas is called a constraint of \mathcal{P} .

In order to be able to interpret these formulas, we need a first order structure $S = \langle \mathcal{U}, \mathcal{J} \rangle$. Here, \mathcal{U} is the universe of S and we will assume that this universe is identical to the set Values, that is we have

$$\mathcal{U} = \mathtt{Values}.$$

The second component \mathcal{J} defines the interpretations of the function symbols and predicate symbols that are used in the formulas defining the constraints. In the following we assume that these interpretations are understood from the context of the constraint satisfaction problem \mathcal{P} and hence need not be specified explicitly. Later, when we discuss examples of constraint satisfaction problems, both the function symbols and the predicate symbols will be interpreted by functions written in Python.

In the following, the abbreviation CSP is short for constraint satisfaction problem. Given a CSP

$$\mathcal{P} = \langle Vars, Values, Constraints \rangle$$
,

a variable assignment for \mathcal{P} is a function

$$A: { t Vars} o { t Values}$$

that maps variables to values. A variable assignment A is a solution of the CSP \mathcal{P} if, given the assignment A, all constraints of \mathcal{P} are satisfied, i.e. we have

$$eval(f, A) = true$$
 for all $f \in Constraints$.

Finally, a partial variable assignment B for \mathcal{P} is a function

```
B: \mathtt{Vars} \to \mathtt{Values} \cup \{\Omega\} where \Omega denotes the undefined value.
```

Hence, a partial variable assignment does not assign values to all variables. Instead, it assigns values only to a subset of the set Vars. The domain dom(B) of a partial variable assignment B is the set of those variables that are assigned a value different from Ω , i.e. we define

$$\mathrm{dom}(B) := \big\{ x \in \mathrm{Vars} \mid B(x) \neq \Omega \big\}.$$

We illustrate the definitions given so far by presenting two examples.

3.1.1 Example: Map Colouring

In map colouring a map showing different states and their borders is given and the task is to colour the different states such that no two states that have a common border share the same colour. Figure 3.1 on page 52 shows a map of Australia. There are seven different states in Australia:

1. Western Australia, abbreviated as WA,



Figure 3.1: A map of Australia.

- 2. Northern Territory, abbreviated as NT,
- 3. South Australia, abbreviated as SA,
- 4. Queensland, abbreviated as Q,
- 5. New South Wales, abbreviated as NSW,
- 6. Victoria, abbreviated as V, and
- 7. Tasmania, abbreviated as T.

Figure 3.1 would certainly look better if different states that share a common border had been coloured with different colours. For the purpose of this example let us assume that we only have the three colours red, green, and blue available. The task is then to colour the different states in a way that no two neighbouring states share the same colour. This task can be formalized as a constraint satisfaction problem. To this end we define:

- 1. $Vars := \{WA, NT, SA, Q, NSW, V, T\},$
- 2. Values := {red, green, blue},
- 3. Constraints :=

 $\big\{ \mathtt{WA} \neq \mathtt{NT}, \, \mathtt{WA} \neq \mathtt{SA}, \, \mathtt{SA} \neq \mathtt{Q}, \, \mathtt{NT} \neq \mathtt{Q}, \, \mathtt{SA} \neq \mathtt{Q}, \, \mathtt{SA} \neq \mathtt{NSW}, \, \mathtt{SA} \neq \mathtt{V}, \, \mathtt{Q} \neq \mathtt{NSW}, \, \mathtt{NSW} \neq \mathtt{V} \big\}.$

Then $\mathcal{P} := \langle \mathtt{Vars}, \mathtt{Values}, \mathtt{Constraints} \rangle$ is a constraint satisfaction problem. If we define the assignment A such that

- (a) A(WA) = blue,
- (b) A(NT) = red,
- (c) A(SA) = green,
- (d) A(Q) = blue,
- (e) A(NSW) = red,
- (f) A(V) = blue,
- (g) A(T) = red,

then it is straightforward to check that this assignment is indeed a solution to the constraint satisfaction problem \mathcal{P} .

3.1.2 Example: The Eight Queens Puzzle

The eight queens puzzle asks to put 8 queens on a chessboard such that no queen can attack another queen. In chess, a queen can attack all pieces that are either in the same row, the same column, or the same diagonal. If we want to put 8 queens on a chessboard such that no two queens can attack each other, we have to put exactly one queen in every row: If we would put more than one queen in a row, the queens in that row could attack each other. If we would leave a row empty, then, given that the other rows contain at most one queen, there would be less than 8 queens on the board. Therefore, in order to model the eight queens problem as a constraint satisfaction problem, we will use the following set of variables:

$$Vars := \{V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8\},\$$

where for $i \in \{1, \dots, 8\}$ the variable V_i specifies the column of the queen that is placed in row i. As the column numbers run from 1 up to 8, we define the set Values as

$$Values := \{1, 2, 3, 4, 5, 6, 7, 8\}.$$

Next, let us define the constraints. There are two different types of constraints.

1. We need constraints that express that no two queens that are positioned in different rows share the same column. To capture these constraints, we define

$$\texttt{DifferentCols} := \{ \mathbf{V}_i \neq \mathbf{V}_j \mid i \in \{1, \cdots, 8\} \land j \in \{1, \cdots, 8\} \land j < i \}.$$

Here the condition j < i ensures that, for example, while we have the constraint $V_2 \neq V_1$ we do not also have the constraint $V_1 \neq V_2$, as the latter constraint would be redundant if the former constraint had already been established.

2. We need constraints that express that no two queens positioned in different rows share the same diagonal. The queens in row i and row j share the same diagonal iff the equation

$$|i - j| = |V_i - V_j|$$

holds. The expression |i - j| is the absolute value of the difference of the rows of the queens in row i and row j, while the expression $|V_i - V_j|$ is the absolute value of the difference of the columns of these queens. To capture these constraints, we define

$$\mathtt{DifferentDiags} := \big\{ |i-j| \neq |\mathtt{V}_i - \mathtt{V}_j| \ \big| \ i \in \{1, \cdots, 8\} \land j \in \{1, \cdots, 8\} \land j < i \big\}.$$

For a fixed pair of values $\langle j, V_i \rangle$ the equations

$$V_i = V_j - j + i$$
 and $V_i = V_j + j - i$

are the linear equations for the straight lines with slope 1 and -1 that pass through $\langle j, V_i \rangle$.

Then, the set of constraints is defined as

${\tt Constraints} := {\tt DifferentCols} \cup {\tt DifferentDiags}$

and the eight queens problem can be stated as the constraint satisfaction problem

$$\mathcal{P} := \langle \mathtt{Vars}, \mathtt{Values}, \mathtt{Constraints} \rangle.$$

If we define the assignment A such that

$$A(V_1) := 4$$
, $A(V_2) := 7$, $A(V_3) := 5$, $A(V_4) := 2$, $A(V_5) := 6$, $A(V_6) := 1$,

$$A(V_7) := 3, \ A(V_8) := 8,$$

then it is easy to see that this assignment is a solution of the eight queens problem. This solution is shown in Figure 3.2 on page 54. In this figure, we have numbered the rows from bottom to top, i.e. the topmost row is row number 8 and therefore the column of the queen in the first row is determined by the variable V_8 .



Figure 3.2: A solution of the eight queens puzzle.

Later, when we develop algorithms to solve CSPs, we will represent variable assignments and partial variable assignments as Python dictionaries. For example, A would then be represented as the dictionary

$$A := \{V_1 : 4, V_2 : 7, V_3 : 5, V_4 : 2, V_5 : 6, V_6 : 1, V_7 : 3, V_8 : 8\}.$$

If we define

$$B := \{V_1 : 4, V_2 : 7, V_3 : 3\},\$$

then B is a partial variable assignment and $dom(B) = \{V_1, V_2, V_3\}$. This partial variable assignment is shown in Figure 3.3 on page 55. Note that the bottom-most row is the row number 1.



Figure 3.3: A partial solution of the eight queens puzzle.

```
def create_csp(n):
          S
                           = range(1, n+1)
2
                           = \{f'V\{i\}' \text{ for } i \text{ in } S\}
          Variables
3
          Values
                           = set(S)
4
         DifferentCols
                          = { f'V\{i\} != V\{j\}' for i in S
5
                                                  for j in S
6
                                                  if i < j
         DifferentDiags = { f'abs(V{j} - V{i}) != {j - i}' for i in S
10
                                                                   if i < j
11
12
         return Variables, Values, DifferentCols | DifferentDiags
13
```

Figure 3.4: The n queens problem formulated as a Csp.

Figure 3.4 on page 56 shows a *Python* program that can be used to create a CSP that encodes the eight queens puzzle. The code shown in this figure is more general than necessary. Given a natural number n, the function call $\mathtt{create_csp}(n)$ creates a constraint satisfaction problem \mathcal{P} that generalizes the eight queens problem to the problem of placing n queens on a board of size n times n such that no queen can capture another queen. The fact that the n-queen problem is parameterized by the number of queens n gives us the ability to check how the running time of the algorithms for solving CSPs scales with the size of the problem.

The beauty of constraint programming is the fact that we will be able to develop a so called constraint solver that takes as input a CSP like the one produced by the program shown in Figure 3.4 and that is capable of computing a solution automatically. In effect, this enables us to use declarative programming: Instead of developing an algorithm that solves a given problem we confine ourselves to specifying the problem precisely and then let a general purpose problem solver do the job of computing the solution. This approach of declarative programming was one of the main ideas incorporated in the programming language Prolog. While Prolog could not live up to its promises as a viable general purpose programming language, constraint programming has proved to be very useful in a number of domains.

3.1.3 Applications

Besides the toy problems discussed so far, there are a number of industrial applications of constraint satisfaction problems. The most important application seem to be variants of scheduling problems. A simple example of a scheduling problem is the problem of generating a timetable for a school. A school has various teachers, each of which can teach some subjects but not others. Furthermore, there are a number of classes that must be taught in different subjects. The problem is then to assign teachers to classes and to create a timetable. A special case of scheduling problems is crew scheduling. For example, airlines have to solve a crew scheduling problem in order to efficiently assign crews of pilots and crews of stewards to their aircraft. Stewards and pilots work in different crews as they have different required resting times.

3.2 Brute Force Search

The most straightforward algorithm to solve a CSP is to test all possible combinations of assigning values to variables. This approach is known as brute-force search. If there are n different values that can be assigned to k variables, this approach amounts to checking at most n^k different variable assignments. For example, for the eight queens problem there are 8 variables and 8 possible values and hence there are at most

```
8^8 = 2^{24} = 16,777,216
```

different assignments that need to be tested. Given the clock rate of modern computers, checking a million assignments per second is plausible. Hence, this approach is able to solve the eight queens problem in about 30 seconds. An implementation of brute force search is shown in Figure 3.5 on page 57.

```
def solve(P):
         return brute_force_search({}, P)
2
    def brute_force_search(Assignment, csp):
4
         Variables, Values, Constraints = csp
         if len(Assignment) == len(Variables): # all variables have been assigned
             if check_all_constraints(Assignment, Constraints):
                 return Assignment
             else:
                 return None
10
         var = arb(Variables - set(Assignment.keys()))
11
         for value in Values:
12
             NewAss = Assignment.copy()
13
             NewAss[var] = value
14
             result = brute_force_search(NewAss, csp)
15
             if result != None:
16
                 return result
17
         return None
18
```

Figure 3.5: Solving a CSP via brute force search.

The function solve takes a constraint satisfaction problem P as its input. This CSP is given as a triple of the form

```
P = (Variables, Values, Constraints).
```

The sole purpose of the function solve is to call the function brute_force_search, which needs an additional argument. This argument is a partial variable assignment that is initially empty. Every recursive iteration of the function brute_force_search assigns one additional variable.

- 1. Assignment is a partial variable assignment. Initially, this assignment will be the empty dictionary. Every recursive call of brute_force_search adds the assignment of one variable to the given assignment.
- 2. csp is a triple of the form

```
csp = (Variables, Values, Constraints).
```

Here, Constraints is a set of Boolean expressions that are given as strings. These strings have to follow the syntax of *Python* so that they can be evaluated using the *Python* function eval.

The implementation of brute_force_search works as follows:

1. If all variables have been assigned a value, the dictionary Assignment will have the same number of entries as the set Variables has elements. Hence, in that case Assignment is a complete assignment of all variables and we now have to test whether all constraints are satisfied. This is done using the auxiliary function check_all_constraints that is shown in Figure 3.6 on page 58. If the current Assignment does indeed satisfy all constraints, it is a solution to the given CSP and is therefore returned.

If, instead, some constraint is violated, then brute_force_search returns the value None.

2. If the assignment is not yet complete, we arbitrarily pick a variable var from the set of those Variables that still have no value assigned. Then, for every possible value in the set Values, we extend the current partial Assignment to a new assignment NewAss that satisfies

```
NewAss[var] = value.
```

Next, the algorithm recursively tries to find a solution for this new partial assignment. If this recursive call succeeds, the solution it has computed is returned. Otherwise, the next value for the given variable var is tried.

3. If none of the values work for var, the function returns None.

```
def check_all_constraints(Assignment, Constraints):

A = Assignment.copy()

return all(eval(f, A) for f in Constraints)
```

Figure 3.6: Auxiliary functions for brute force search.

The function check_all_constraints takes a complete variable Assignment as its first input. The second input is the set Constraints which is a set of *Python* expressions. For all expressions f from the set Constraints, the function check_all_constraints checks whether f yields True under the given variable assignment. This check is done using the function eval, which is a predefined function. This function takes two arguments:

- (a) The first argument is a Python expression f.
- (b) The second argument is a variable assignment A, that is represented as a dictionary.

The function eval evaluates the expression f. In order to do this, any variables occurring in f are assigned values according to the variable assignment A. As a side effect, the function eval changes the dictionary A that is used as its second argument. This is the reason we have to make a copy of the Assignment that is given as the first argument of the function check_all_constraints.

When I tested the program discussed above with the eight queens problem, it took about 30 seconds to compute a solution. In contrast, the seven queens problem took about 1.7 second. As we have

$$\frac{8^8}{7^7} \approx 20$$
 and $30/1.7 \approx 18$

this shows that the computation time does indeed roughly grow with the number of possible assignments that have to be checked. However, the correspondence is not exact. The reason is that we stop our search as soon as a solution is found. If we are lucky and the given CSP is easy to solve, this might happen when we have checked only a small portion of the set of all possible assignments.

3.3 Backtracking Search

For the n queens problem the number of possible variable assignments growth as fast as n^n . This growth is super-exponential and this is what usually happens when we scale a CSP up. The reason is that the number of all variable assignments is given as

```
\operatorname{card}(\operatorname{Values})^{\operatorname{card}(\operatorname{Vars})}
```

where for a set M, the expression $\operatorname{card}(M)$ returns the number of elements of M. For this reason, brute force search is only viable for small problems. One approach to solve a CSP that is both conceptually simple and at least more efficient than brute force search is backtracking. The idea is to try to evaluate constraints as soon as possible: If C is a constraint and B is a partial assignment such that all the variables occurring in C have already been assigned a value in B and the evaluation of C fails, then there is no point in trying to complete the variable assignment B. Hence, in backtracking we evaluate a constraint C as soon as all of its variables have been assigned a value. If C is not valid, we discard the current partial variable assignment. This approach can result in huge time savings when compared to the baseline of brute force search.

Figure 3.7 on page 59 shows a simple CSP solver that employs the backtracking strategy. We discuss this program next. The function solve takes a constraint satisfaction problem P as input and tries to find a solution.

```
def solve(P):
    Variables, Values, Constraints = P
    csp = (Variables, Values, [(f, collect_variables(f)) for f in Constraints])
    try:
        return backtrack_search({}, csp)
    except Backtrack:
        return None
```

Figure 3.7: A backtracking CSP solver.

- 1. First, the CSP P is split into its components.
- 2. Next, for every constraint f of the given Csp, we compute the set of variables that are used in f. This is done using the function collect_variables that is shown in Figure 3.10 on page 62. These variables are then stored together with the constraint f and the correspondingly modified data structure is stored in the variable csp and is called an augmented Csp.

The reason to compute and store these variables is efficiency: When we later check whether a constraint f is satisfied for a partial variable assignment Assignment where Assignment is stored as a dictionary, we only need to check the constraint f iff all of the variables occurring in f are elements of the domain of Assignment. It would be wasteful to compute these variables every time that a partial variable assignment is extended.

3. Next, we call the function backtrack_search to compute a solution of CSP. This function is enclosed in a try-except-block that catches exceptions of class Backtrack. This class is defined as follows:

```
class Backtrack(Exception):
    pass
```

Its only purpose is to create a name for the special kind of exceptions used to administer back-tracking. The reason for enclosing the call to backtrack_search in a try-except-block is that the function backtrack_search either returns a solution or, if it is not able to find a solution, it raises an exception of class Backtrack. The try-except-block ensures that this exception is silently discarded.

```
def backtrack_search(Assignment, P):
         Variables, Values, Constraints = P
2
         if len(Assignment) == len(Variables):
             return Assignment
         var = arb(Variables - set(Assignment.keys()))
         for value in Values:
             try:
                 if is_consistent(var, value, Assignment, Constraints):
                     NewAss = Assignment.copy()
9
                     NewAss[var] = value
10
                     return backtrack_search(NewAss, P)
11
             except Backtrack:
12
                 continue
13
         raise Backtrack()
14
```

Figure 3.8: A backtracking CSP solver: The function backtrack_search.

Next, we discuss the implementation of the function backtrack_search that is shown in Figure 3.8 on page 60. This function receives a partial assignment Assignment as input together with an augmented CSP P. This partial assignment is consistent with P: If f is a constraint of CSP such that all the variables occurring in f are members of dom(Assignment), then evaluating f using Assignment yields true. Initially, this partial assignment is empty and hence trivially consistent. The idea is to extend this partial assignment until it is a complete variable assignment. We take care to ensure that this partial variable assignment remains consistent when it is extended. This way, once this assignment is complete it has to satisfy all the constraints of the given CSP.

- 1. First, the augmented CSP P is split into its components.
- 2. Next, if Assignment is already a complete variable assignment, i.e. if the dictionary Assignment has as many elements as there are variables, then it must be a solution of the CSP and, therefore, it is returned. The reason is that the function backtrack_search is only called with a consistent partial assignment.

- 3. Otherwise, we have to extend the partial Assignment. In order to do so, we first have to select a variable var that has not yet been assigned a value in Assignment so far. This is done in line 5 using the function arb that selects an arbitrary variable from its input set.
- 4. Next, we try to assign a value to the selected variable var. After assigning a value to var, we immediately check whether this assignment would be consistent with the constraints using the function is_consistent. If the partial Assignment turns out to be consistent, the partial variable Assignment is extended to the new partial assignment NewAss that satisfies

```
NewAss[var] = value.
```

Then, the function backtrack_search is called recursively to complete this new partial assignment. If this is successful, the resulting assignment is a solution that is returned. Otherwise, the recursive call of backtrack_search will raise an exception. This exception is muted by the try-except-block that surrounds the call to backtrack_search. In that case, the for-loop generates a new value that can be assigned to the variable var. If all possible values have been tried and none was successful, the for-loop ends and we have to backtrack, i.e. we have to reassign one of the variables that have been assigned earlier. This is done by raising a Backtrack exception. This exception is then caught by one of the prior invocations of backtrack_search. If all variable assignments have been tried and none is successful, then the Backtrack exception propagates back to the function solve, which will return None in that case.

Figure 3.9: The definition of the function is_consistent.

We still need to discuss the implementation of the auxiliary function is_consistent shown in Figure 3.9. This function takes a variable var, a value, a partial Assignment and a set of Constraints as arguments. It is assumed that Assignment is partially consistent with respect to the set Constraints, i.e. for every formula f occurring in Constraints such that

```
vars(f) \subseteq dom(Assignment)
```

holds, the formula f evaluates to True given the Assignment. The purpose of is_consistent is to check, whether the extended assignment

```
NewAssign := Assignment \cup {var \mapsto value}
```

that assigns value to the variable var is still partially consistent with Constraints. To this end, the for-loop iterates over all formulas f in Constraints. However, we only have to check those formulas f that contain the variable var and, furthermore, have the property that

```
vars(f) \subseteq dom(NewAssign),
```

i.e. all variables occurring in the formula **f** need to have a value assigned in NewAssign. The reasoning is as follows:

- 1. If var does not occur in the formula f, then adding var to Assignment cannot change the result of evaluating f and as Assignment is assumed to be partially consistent with respect to f, NewAssign is also partially consistent with respect to f.
- 2. If dom(NewAssign) \(\mathref{Z} \) vars(f), then f can not be evaluated anyway.

Note that the domain of a variable assignment A can be computed with the expression A.keys() since A is represented as a dictionary in *Python*.

Figure 3.10: The function collectVars.

Finally, let us discuss the function collect_variables that is shown in Figure 3.10 on page 62. This function uses the module extractVariables that provides the function extractVars(e). This function takes a string e that can be interpreted as a Python expression as its argument and returns the set of all variables and function symbols occurring in the expression e. As we only want to keep the variable names, the function collect_variables takes care to eliminate the function symbols. This is done by making use of the fact that all function symbols that have been defined are members of the list dir(__builtins__). It turns out that the keyword "and", "or", and "not" also need to be removed since they might also be members of the set returned by extractVars(expr).

If we use the program discussed in this section, we can solve the 8 queens problem in about 22 milliseconds. Hence, for the eight queens problem backtracking is more than a thousand times faster than brute force search.

Exercise 5: There are many different versions of the *zebra puzzle*. The version below is taken from *Wikipedia*. The puzzle reads as follows:

- (a) There are five houses.
- (b) The Englishman lives in the red house.
- (c) The Spaniard owns the dog.
- (d) Coffee is drunk in the green house.
- (e) The Ukrainian drinks tea.
- (f) The green house is immediately to the right of the ivory house.
- (g) The Old Gold smoker owns snails.
- (h) Kools are smoked in the yellow house.
- (i) Milk is drunk in the middle house.

- (j) The Norwegian lives in the first house.
- (k) The man who smokes Chesterfields lives in the house next to the man with the fox.
- (1) Kools are smoked in the house next to the house where the horse is kept.
- (m) The Lucky Strike smoker drinks orange juice.
- (n) The Japanese smokes Parliaments.
- (o) The Norwegian lives next to the blue house.
- (p) Who drinks water?
- (q) Who owns the zebra?

In order to solve the puzzle, we also have to know the following facts:

- Each of the five houses is painted in a different colour.
- The inhabitants of the five houses are of different nationalities, and
- they own different pets, drink different beverages, and smoke different brands of cigarettes.

Formulate the zebra puzzle as a constraint satisfaction problem and solve the puzzle using the program discussed in this section.

3.4 Constraint Propagation

Once we have chosen a value for a variable, this choice influences the values that are still available for other variables. For example, suppose that in order to solve the n queens problem we place the queen in row one in the second column, then no other queen can be placed in that column. Furthermore, due to the constraints on diagonals, the queen in row two can not be placed in any of the first three columns. Abstractly, constraint propagation works as follows.

1. Before the search is started, we create a dictionary ValuesPerVar. Initially, for every variable x, the set

ValuesPerVar[x]

contains all values v from the set Values. As soon as we discover that assigning a value v to the variable v is inconsistent with the variable assignments that have already taken place for other variables, the value v will be removed from the set ValuesPerVar[v].

2. As long as the given CSP is not solved, we choose a variable x that has not been assigned a value yet. This variable is chosen using the most constrained variable heuristic: We choose a variable x such that the number of values in the set

ValuesPerVar[x]

is minimal. This is done because we have to find values for all variables. If the current partial variable assignment can not be completed into a solution, then we want to find out this fact as soon as possible. Therefore, we try to find the values for the most difficult variables first. A variable is more difficult to get right if it has only a few values left that can be used to instantiate it.

- 3. Once we have picked a variable x, we next iterate over all values v in ValuesPerVar[x]. If the problem is hard, then it pays off to sort the values using the least constraining value heuristic: The idea is that we compute for all values in ValuesPerVar[x] how much they constrain the values of other variables. The details of how this is done will be given later. Once we have picked a value v for the variable x that puts the least constraints on the other variables, we assign this value v to the variable x and then we propagate the consequences of this assignment:
 - (a) For every constraint f that mentions only the variable x and one other variable y that has not yet been instantiated, we compute the set Legal of those values from ValuesPerVar[y] that can be assigned to y without violating the constraint f.
 - (b) Then, the set ValuesPerVar[y] is updated to the set Legal and we go back to step 2.

Finally, we can explain how the least constraining value heuristic orders the values: For every value v from the set ValuesPerVar[x] the heuristic computes the shrinkage number, which is the number of values that have to be removed from the sets ValuesPerVar[y] where y is any variable that is different from x and that has not yet been assigned. Then the different values are ordered ascendingly with respect to their shrinkage number. Hence values with a low shrinkage number are tried before values with a higher shrinkage number.

It turns out that elaborating the idea outlined above can enhance the performance of backtracking search considerably. Figure 3.11 on page 64 shows an implementation of constraint propagation. In addition to the ideas described above, this implementation takes care of unary constraints, i.e. constraints that contain only a single variable, as these constraints can be solved prior to the other constraints without backtracking.

```
def solve(P, lcv=True):
         Variables, Values, Constraints = P
                      = { (f, collect_variables(f)) for f in Constraints }
3
         ValuesPerVar = { v: Values for v in Variables }
         UnaryConstrs = { (f, V) for f, V in Annotated if len(V) == 1 }
5
         OtherConstrs = { (f, V) for f, V in Annotated if len(V) >= 2 }
6
         try:
             for f, V in UnaryConstrs:
                 var = arb(V)
                 ValuesPerVar[var] = solve_unary(f, var, ValuesPerVar[var])
10
             return backtrack_search({}, ValuesPerVar, OtherConstrs, lcv)
11
         except Backtrack:
12
             return None
13
```

Figure 3.11: Constraint Propagation.

In order to implement constraint propagation, it is necessary to administer the values that can be used to instantiate the different variables separately, i.e. for every variable \mathbf{x} we need to know which values are admissible for \mathbf{x} . To this end, we need a dictionary ValuesPerVar that contains the set of possible values for every variable \mathbf{x} . Initially, this dictionary assigns the set Values to every variable. Next, we take care of the unary constraints and shrink these sets so that the unary constraints are satisfied. Then, whenever we assign a value to a variable \mathbf{x} , we inspect those constraints that mention the variable \mathbf{x} and exactly one other yet unassigned variable \mathbf{y} and shrink the set of values

ValuesPerVar[y] that can be assigned to this variables y. This process is called constraint propagation and is described in more detail below when we discuss the function propagate.

1. The function solve receives a CSP P and a Boolean flag lcv. The CSP P is first split into its three components and the constraints are annotated with the sets of variables occurring in them. These annotated constraints are stored in the set Annotated.

The flag lcv controls whether we order the values using the least constraining value heuristic. If this flag is True, the least constraining value heuristic is used. Otherwise the values are ordered arbitrarily.

2. The most important data structure maintained by the function solve is the dictionary

ValuesPerVar.

Given a variable v, this dictionary assigns the set of values that can be used to instantiate this variable. Initially, this set is the same for all variables and is equal to Values.

- 3. In order to solve the unary constraints we first have to find them. The set UnaryConstrs contains all those pairs (f, V) from the set of annotated constraints such that the set of variables V contains just a single variable.
- 4. Similarly, the set OtherConstrs contains those constraints that involve two or more variables.
- 5. In order to solve the unary constraints, we iterate over all unary constraints and shrink the set of values associated with the variable occurring in the constraint as dictated by the constraint. This is done using the function solve_unary.
- 6. Then, we start backtracking search using the function backtrack_search. Besides backtracking, the implementation of backtrack_search that we present below implements the most constraint variable heuristic and constraint propagation.

```
def solve_unary(f, x, Values):
    Legal = { value for value in Values if eval(f, { x: value }) }
    if not Legal:
        raise Backtrack()
    return Legal
```

Figure 3.12: Implementation of solve_unary.

The function solve_unary shown in Figure 3.12 on page 65 takes a unary constraint f, the variable x occurring in f and the set of values Values that can be assigned to this variable. It returns the subset of values that can be substituted for the variable x without violating the given constraint f. If this set is empty, a Backtrack exception is raised since in that case the given CSP is unsolvable.

The function backtrack_search shown in Figure 3.13 on page 66 is called with a partial variable Assignment that is guaranteed to be consistent, a dictionary ValuesPerVar associating every variable with the set of values that might be substituted for this variable, a set of annotated Constraints, and a Boolean flag lcv that controls whether the least constraining value heuristic is used. It tries to complete Assignment and thereby computes a solution of the given Csp.

```
def backtrack_search(Assignment, ValuesPerVar, Constraints, lcv):
         if len(Assignment) == len(ValuesPerVar):
2
             return Assignment
         x = most_constrained_variable(Assignment, ValuesPerVar)
         if lcv and len(ValuesPerVar[x]) > 1:
             ValueList = least_constraining(x, ValuesPerVar, Assignment, Constraints)
         else:
             ValueList = ValuesPerVar[x]
         for v in ValueList:
             try:
10
                 NewValues = propagate(x, v, Assignment, Constraints, ValuesPerVar)
11
                 NewAssign = Assignment.copy()
12
                 NewAssign[x] = v
13
                 return backtrack_search(NewAssign, NewValues, Constraints, lcv)
14
             except Backtrack:
15
                 continue
16
         raise Backtrack()
17
```

Figure 3.13: Implementation of backtrack_search.

- 1. If the partial Assignment is already complete, i.e. if it assigns a value to every variable, then a solution to the given CSP has been found and this solution is returned. As the dictionary ValuesPerVar has an entry for every variable, its size is the same as the number of variables. Therefore, Assignment is complete iff it has the same size as ValuesPerVar.
- 2. Otherwise, we choose a variable x such that the number of values that can still be used to instantiate x is minimal. This strategy is known as the most constrained variable heuristic. The variable x is computed using the function most_constrained_variable that is shown in Figure 3.14 on page 67.

The logic behind choosing a maximally constrained variables is that these variables are the most difficult to get right. If we have a partial assignment that is inconsistent, then we will discover this fact earlier if we try the most difficult variables first. This might save us a lot of unnecessary backtracking later.

3. Next, we try to find a value that can be assigned to the variable x. If the least constraining value heuristic is used, we try those values first that have the smallest shrinkage number. Given a variable x, the shrinkage number for assigning the value v to x is the total number of values that have to be removed from the sets ValuesPerVar[y] when x is set to v. Here y ranges over all variables that are yet unassigned and are different from x.

The function <code>least_constraining</code> returns a list of all values <code>v</code> that can be substituted for <code>x</code>. This list is sorted ascendingly w.r.t. the shrinkage number of <code>v</code>. However, the function is only used if the flag <code>lcv</code> is set. The reason is that computing the shrinkage number is computationally quite expensive. Therefore, this only pays off for hard problems. If the flag is not set, the values are tried in an arbitrary order.

Note that since ValuesPerVar[x] is, in general, smaller than the set of all values of the CSP, the for-loop in this version of backtracking search is more efficient than the corresponding for-loop

in backtracking search discussed in the previous section.

- 4. If assigning the value v to the variable x is consistent, we propagate the consequences of this assignment using the function propagate shown in Figure 3.17 on page 70. This function updates the dictionary ValuesPerVar for all variables that are still unassigned.
- 5. Finally, the partial variable Assignment is updated to include the assignment of v to x and the recursive call to backtrack_search tries to complete this new assignment and thereby compute a solution to the given CSP.

Figure 3.14: Finding a most constrained variable.

Figure 3.14 on page 67 shows the implementation of the function most_constrained_variable. The function most_constrained_variable takes a partial Assignment and a dictionary ValuesPerVar returning for all variables x the set of values ValuesPerVar[x] that can be assigned to x.

- 1. First, this function computes the set of Unassigned variables. For every variable x that has not yet been assigned a value in Assignment this set contains the pair (x, len(U)), where U is the set of values that still might be tried for the variable x.
- 2. Next, minSize is the minimum size of the sets ValuesPerVar[x] for all unassigned variables.
- 3. Finally, an arbitrary variable x that has only minSize values available is returned.

```
def least_constraining(x, ValuesPerVar, Assignment, Constraints):

NumbersValues = []

for value in ValuesPerVar[x]:

ReducedValues = ValuesPerVar.copy()

num_removed = shrinkage(x, value, Assignment, ReducedValues, Constraints)

if num_removed != math.inf:

NumbersValues.append((num_removed, value))

NumbersValues.sort(key=lambda p: p[0])
```

Figure 3.15: Finding the least constraining value.

Figure 3.15 on page 67 shows the implementation of the function least_constraining. This function takes four arguments.

1. x is the variable that needs to be assigned a value next.

- 2. ValuesPerVar is a dictionary. For every variable y that has not yet been assigned a value, ValuesPerVar[y] is the set of values that can be assigned to y without violating a constraint.
- 3. Assignment is a partial variable assignment.
- 4. Constraints is a set of annotated constraints.

The purpose of the function least_constraining is to compute the shrinkage numbers of every value v in ValuesPerVar[x] and to sort these values according to their shrinkage numbers. To this end the list NumbersValues is a list of pairs of the form (n,v) where v is a value and n is the shrinkage number of v. To compute this list, the function shrinkage computes the number of values that had to be removed from all sets ValuesPerVar[y] when x is set to value. The function shrinkage returns the number math.inf to signal the fact that it has found the assignment $\{x \mapsto value\}$ to be inconsistent. After all shrinkage numbers have been computed, the list NumbersValues is sorted ascendingly w.r.t. the shrinkage numbers.

```
def shrinkage(x, value, Assignment, ValuesPerVar, Constraints):
                          # number of values removed from ValuesPerVar
         BoundVars = set(Assignment.keys())
3
         for f, Vars in Constraints:
4
             if x in Vars:
5
                  UnboundVars = Vars - BoundVars - { x }
                  if len(UnboundVars) == 1:
                      y = arb(UnboundVars)
8
                      Legal = set()
9
                      for w in ValuesPerVar[y]:
10
                          NewAssign
                                        = Assignment.copy()
11
                          NewAssign[x] = value
12
                          NewAssign[y] = w
13
                          if eval(f, NewAssign):
14
                               Legal.add(w)
15
                          else:
16
17
                               count += 1
                      if len(Legal) == 0:
18
                          return math.inf
19
                      ValuesPerVar[x] = Legal
20
21
         return count
```

Figure 3.16: Computing the shrinkage number.

The function shrinkage shown in Figure 3.16 on page 68 computes the shrinkage numbers of the given value that is assigned to the variable x. It takes the following inputs:

- (a) x is a variable and value is the value that is assigned to the variable x.
- (b) Assignment is a partial assignment that contains assignments for variables that are different from x.
- (c) ValuesPerVar is a dictionary assigning sets of values to all variables.

(d) Constraints is a set of annotated constraints, i.e. this set contains pairs of the form (f, Vars), where f is a constraint and Vars is the set of variables occurring in f.

The function shrinkage has to compute the number of values that have to be removed from the set ValuesPerVar[y] for all variables y that are different from x and that are still unbound if the value value is assigned to the variable x. The function is implemented as follows:

- 1. count is the number of values that have to be removed form ValuesPerVar[y] for some variable y if we set the variable x to the value v. Initially, count is set to 0.
- 2. BoundVars is the set of those variable that already have a value assigned in Assignment.
- 3. Next, shrinkage iterates over all constraints f such that the variable x occurs in f.
- 4. UnboundVars is the set of those variables occurring in f that are different from x and that do not yet have a value assigned. These variables are called unbound variables since we still need to assign values for these variables.
- 5. If the set UnboundVars contains just a single variable, the function arb returns this variable y.
- 6. Now we have to find those values w that can be assigned to y without violating the constraint f. Each time we find a value w from ValuesPerVar[y] such that the assignment

$$\mathtt{Assignment} \cup \{\mathtt{x} \mapsto \mathtt{v}, \mathtt{y} \mapsto \mathtt{w}\}$$

violates the constraint f, we increase the number count since the value w has to be removed from ValuesPerVar[y]. If the constraint f is not violated, the value w is added to the set Legal of values that may be assigned to y.

7. If, after all values form NewValues[y] have been tested, it turns out that Legal is still empty, then this means that the constraint f is inconsistent with assigning value to the variable x because once we assign value to x we won't be able to find a value w that can be assigned to y. In this case the function shrinkage returns math.inf to signal this inconsistency.

The function propagate shown in Figure 3.17 on page 70 implements constraint propagation. It takes the following inputs:

- (a) x is a variable and v is a value that is assigned to the variable x.
- (b) Assignment is a partial assignment that contains assignments for variables that are different from x.
- (c) Constraints is a set of annotated constraints, i.e. this set contains pairs of the form (f, Vars), where f is a constraint and Vars is the set of variables occurring in f.
- (d) ValuesPerVar is a dictionary assigning sets of values to all variables.

The purpose of the function propagate is to restrict the values of variables different from x by propagating the consequences of setting x to v. To this end the function propagate updates the dictionary ValuesPerVar by taking into account the consequences of assigning the value v to the variable v. The implementation of propagate proceeds as follows.

- 1. Initially, we copy the Dictionary ValuesPerVar to the dictionary ValuesDict
- 2. As x is assigned the value v, the corresponding entry in the dictionary ValuesDict is changed accordingly.

```
def propagate(x, v, Assignment, Constraints, ValuesPerVar):
         ValuesDict = ValuesPerVar.copy()
2
         ValuesDict[x] = { v }
         BoundVars = set(Assignment.keys())
         for f, Vars in Constraints:
             if x in Vars:
6
                 UnboundVars = Vars - BoundVars - { x }
                 if len(UnboundVars) == 1:
                      y = arb(UnboundVars)
                      Legal = set()
10
                      for w in ValuesDict[y]:
11
                          NewAssign = Assignment.copy()
12
                          NewAssign[x] = v
13
                          NewAssign[y] = w
14
                          if eval(f, NewAssign):
                              Legal.add(w)
16
                      if len(Legal) == 0:
17
                          raise Backtrack()
18
                      ValuesDict[y] = Legal
19
         return ValuesDict
20
```

Figure 3.17: Constraint Propagation.

- 3. BoundVars is the set of those variable that already have a value assigned.
- 4. Next, propagate iterates over all constraints f such that the variable x occurs in f.
- 5. UnboundVars is the set of those variables occurring in f that are different from x and that do not yet have a value assigned.
- 6. If there is exactly one unbound variable y in the constraint f, then we can test those values that satisfy f and recompute the set ValuesDict[x].
- 7. As the set UnboundVars contains just a single variable in line 9, the function arb returns this variable.
- 8. In order to recompute the set ValuesDict[y], all values w in ValuesDict[y] are tested. The set Legal contains all values w that can be assigned to the variable y without violating the constraint f.
- 9. If it turns out that Legal is the empty set, then this means that the constraint f is inconsistent with assigning the value v to the variable x. Hence, in this case the search has to backtrack.
- 10. Otherwise, the set of admissible values for y is updated to be the set Legal.
- 11. Finally, the dictionary ValuesDict is returned.

I have tested the program described in this section using the eight queens puzzle. It takes about 18 milliseconds to find a solution if the least constraining value heuristic is not used. I have also tested

it with the Zebra Puzzle described in a previous exercise. It solves this puzzle in 21 milliseconds. To compare, the backtracking algorithm shown in the previous section takes roughly 10 seconds to solve this puzzle. Hence, for problems like the eight queens puzzle, the least constraining value heuristic does not pay off. However, for more complicated puzzles like Sudoku puzzles, the least constraining value heuristic speeds up the computation.

3.5 Consistency Checking

So far, the constraints in the constraints satisfaction problems discussed are either unary constraints or binary constraints: A unary constraint is a constraint f such that the formula f contains only one variable, while a binary constraint contains two variables. If we have a constraint satisfaction problem that involves also constraints that mention more than two variables, then the constraint propagation shown in the previous section is not as effective as it is only used for a constraint f if all but one variable of f have been assigned. For example, consider the cryptarithmetic puzzle shown in Figure 3.18 on page 71. The idea is that the letters "S", "E", "N", "D", "M", "O", "R", "Y" are interpreted as variables ranging over the set of decimal digits, i.e. these variables can take values in the set $\{0,1,2,3,4,5,6,7,8,9\}$. Then, the string "SEND" is interpreted as a decimal number, i.e. it is interpreted as the number

$$S \cdot 10^3 + E \cdot 10^2 + N \cdot 10^1 + D \cdot 10^0$$

The strings "MORE and "MONEY" are interpreted similarly. To make the problem interesting, the assumption is that different variables have different values. Furthermore, the digits at the beginning of a number should be different from 0.



Figure 3.18: A cryptarithmetic puzzle

A naïve approach to solve this problem would be to code it as a constraint satisfaction problem that has, among others, the following constraint:

$$(\mathtt{S} \cdot 10^3 + \mathtt{E} \cdot 10^2 + \mathtt{N} \cdot 10 + \mathtt{D}) + (\mathtt{M} \cdot 10^3 + \mathtt{O} \cdot 10^2 + \mathtt{R} \cdot 10 + \mathtt{E}) = \mathtt{M} \cdot 10^4 + \mathtt{O} \cdot 10^3 + \mathtt{N} \cdot 10^2 + \mathtt{E} \cdot 10 + \mathtt{Y}.$$

The problem with this constraint is that it involves far too many variables. As this constraint can only be checked when all the variables have values assigned to them, the backtracking search would essentially boil down to a mere brute force search. We would have 8 variables that each could take 10 different values and hence we would have to test 10^8 possible assignments. In order to do better, we have to perform the addition shown in Figure 3.18 column by column, just as it is taught in elementary school. Figure 3.19 on page 72 shows how this can be implemented in Python.

Notice that we have introduced three additional variables "C1", "C2", "C3". These variables serve as the carry digits. For example, "C1" is the carry digit that we get when we add the final digits of the two numbers, i.e. we have

$$D + E = C1 \cdot 10 + Y.$$

This equation still contains four variables. We can split this equation into two smaller equations that

```
def crypto_csp():
                      = { 'S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y' }
         Digits
2
                      = Digits | { 'C1', 'C2', 'C3' }
         Variables
3
         Values
                      = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}
         Constraints = allDifferent(Digits)
                                         % 10 == Y', '(D + E)
         Constraints |= { '(D + E)
                                                                // 10 == C1',
                           '(N + R + C1) \% 10 == E', '(N + R + C1) // 10 == C2',
                           '(E + 0 + C2) % 10 == N', '(E + 0 + C2) // 10 == C3',
                           '(S + M + C3) \% 10 == 0', '(S + M + C3) // 10 == M'
10
         Constraints |= { 'S != 0', 'M != 0' }
11
         Constraints = \{ 'C1 < 2', 'C2 < 2', 'C3 < 2' \}
12
         return Variables, Values, Constraints
13
14
    def allDifferent(Variables):
15
         return { f'{x} != {y}' for x in Variables
16
                                 for y in Variables
17
                                 if x < y
18
            }
19
```

Figure 3.19: Formulating "SEND + MORE = MONEY" as a CSP.

each involve only three variables with the help of the modulo operator "%" and the operator for integer division "//" as follows:

```
(D + E) \% 10 = Y and (D + E) // 10 = C1.
```

If we solve the cryptarithmetic puzzle as coded in Figure 3.19 on page 72 using the constraint solver developed, then solving the puzzle takes about a second on my computer. The reason is that most constraints involve either three or four variables and therefore the effects of constraint propagation kick only in when many variables have already been initialized. However, we can solve the problem in less than 50 milliseconds if we add the following constraints for the variables "C1", "C2", "C3":

```
"C1 < 2", "C2 < 2", "C3 < 2".
```

Although these constraints are certainly true, the problem with this approach is that we would prefer our constraint solver to figure out these constraints by itself. After all, since D and E are both less than 10, their sum is obviously less than 20 and hence the carry C1 has to be less than 2. This line of reasoning is known as consistency maintenance: Assume that the formula f is a constraint and the set of variables occurring in f has the form

$$Var(f) = \{x\} \cup R \quad \text{where } x \notin R,$$

i.e. the variable x occurs in the constraint f and, furthermore, $R = \{y_1, \dots, y_n\}$ is the set of all variables occurring in f that are different from x. In addition, assume that we have a dictionary ValuesPerVar such that for every variable y, the dictionary entry ValuesPerVar[y] is the set of values that can be substituted for the variable y. The formal definition follows.

Definition 5 (Consistent Value for a Variable) A value v is consistent for the variable x with respect to the constraint f iff the partial assignment $\{x \mapsto v\}$ can be extended to an assignment A

satisfying the constraint f, i.e. for every variable y_i that is different from x we have to find a value $w_i \in \mathtt{ValuesPerVar}[y_i]$ such that the resulting assignment $A = \{x \mapsto v, y_1 \mapsto w_1, \cdots, y_n \mapsto w_n\}$ satisfies the equations

$$eval(f, A) = True.$$

Here, the function eval takes a formula f and a variable assignment A and evaluates f using this assignment. \diamond

Given a CSP $\mathcal{P} = \langle Vars, Values, Constraints \rangle$, the algorithm for consistency maintenance is shown below.

1. The dictionary ValuesPerVar is initialized as follows:

$$ValuesPerVar[x] := Values$$
 for all $x \in Variables$,

i.e. initially every variable x can take any value from the set of Values.

2. Next, the set UncheckedVariables is initialized to the set of all Variables:

UncheckedVariables := Variables.

3. As long as the set Unchecked Variables is not empty, we remove one variable x from this set:

```
x := UncheckedVariables.pop()
```

- 4. We iterate over all constraints f such that x occurs in f.
 - (a) For every value $v \in ValuesPerVar[x]$ we check whether v is consistent with f.
 - (b) If the value v is not consistent with f, then v is removed from ValuesPerVar[x]. Furthermore, all variables connected to x are added to the set of UncheckedVariables. Here we define y variable $y \neq x$ to be connected to x if there is some constraint f such that both x and y occur in f. The reason is that some of their values might have become inconsistent by removing the value v from ValuesPerVar[x].
- 5. Once the set UncheckedVariables is empty, the algorithm terminates. Otherwise, we jump back to step 3 and remove the next variable from the set UncheckedVariables.

The algorithm terminates as every iteration removes either a variable from the set UncheckedVariables or it removes a value from one of the sets ValuesPerVar[y] for some variable y. Although the set UncheckedVariables can grow during the algorithm, the union

$$\bigcup_{x \in \mathtt{Vars}} \mathtt{ValuesPerVar}[x]$$

can never grow: Every time the set UncheckedVariables grows, for some variable x the set

shrinks. As the sets ValuesPerVar[x] are finite for all variables x, the set UncheckedVariables can only grow a finite number of times. Once the set UncheckedVariables does not grow any more, every iteration of the algorithm removes one variable from this set and hence the algorithm terminates eventually.

Figure 3.20 on page 74 shows how consistency maintenance can be implemented in *Python*. The function enforce_consistency takes four arguments.

(a) ValuesPerVar is a dictionary associating the set of possible values with each variable.

```
def enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected):
        UncheckedVars = set(Var2Formulas.keys())
         while UncheckedVars:
             variable
                         = UncheckedVars.pop()
             RemovedVals = set()
             for f in Var2Formulas[variable]:
                 OtherVars = Annotated[f] - { variable }
                 for value in ValuesPerVar[variable]:
                     if not exists_values(variable,value,f,OtherVars,ValuesPerVar):
                         RemovedVals
                                        |= { value }
10
                         UncheckedVars |= Connected[variable]
11
             ValuesPerVar[variable] -= RemovedVals
12
             if len(ValuesPerVar[variable]) == 0: # the problem is unsolvable
13
                 raise Backtrack()
14
```

Figure 3.20: Consistency maintenance in Python.

- (b) Var2Formulas is a dictionary. For every variable x, Var2Formulas[x] is the set of those constraints f such that x occurs in f.
- (c) Annotated is a dictionary mapping constraints to the set of variables occurring in them, i.e. if f is a constraint, then Annotated [f] is the set of variables occurring in f.
- (d) Connected is a dictionary that takes a variable x and returns the set of all variables that are connected to x via a common constraint f, i.e. we have $y \in \text{Connected}[x]$ iff there exists a constraint f such that both x and y occur in f and, furthermore, $x \neq y$.

The function enforce_consistency modifies the dictionary ValuesPerVar so that once the function has terminated, for every variable x the values in the set ValuesPerVar[x] are consistent with the constraints for x. The implementation works as follows:

- 1. Initially, all variables need to be checked for consistency. Therefore, UncheckedVars is defined to be the set of all variables that occur in any of the constraints.
- 2. The while-loop iterates as long as there are still variables x left in UncheckedVars such that the consistency of ValuesPerVar[x] has not been established.
- 3. Next, a variable variable is selected and removed from UncheckedVars.
- 4. RemovedVals is the subset of those values that are found to be inconsistent with some constraint for variable.
- 5. We iterate over all constraints $f \in Var2Formulas[variable]$.
- 6. OtherVars is the set of variables occurring in f that are different from the chosen variable variable.
- 7. We iterate over all value ∈ ValuesPerVar[variable] that can be substituted for the variable variable and check whether value is consistent with f. To this end, we need to find values that can be assigned to the variables in the set OtherVars such that f evaluates as True. This is checked using the function exists_values.

- 8. If we do not find such values, then value is inconsistent for the variable variable w.r.t. f and needs to be removed from the set ValuesPerVar[variable]. Furthermore, all variables that are connected to variable have to be added to the set UncheckedVars. The reason is that once a value is removed for the variable var, the value assigned to another variable y occurring in a constraint that mentions both var and y might now become inconsistent.
- 9. The set of values that are known to be consistent for variable is stored as ValuesPerVar[variable].
- 10. If there are no consistent values for variable left, the problem is unsolvable and an exception is raised.

```
def exists_values(var, val, f, Vars, ValuesPerVar):
         Assignments = all_assignments(Vars, ValuesPerVar)
2
         return any(eval(f, extend(A, var, val)) for A in Assignments)
3
4
     def extend(A, x, v):
5
         B = A.copy()
6
         B[x] = v
7
         return B
8
9
     def all_assignments(Variables, ValuesPerVar):
10
         Variables = set(Variables) # turn frozenset into a set
11
         if not Variables:
12
             return [ {} ]
                             # list containing empty assignment
13
         var
                      = Variables.pop()
14
         Assignments = all_assignments(Variables, ValuesPerVar)
15
         return [ extend(A, var, val) for A in Assignments
16
                                        for val in ValuesPerVar[var]
17
                ]
```

Figure 3.21: The implementation of exists_value.

Figure 3.21 on page 75 shows the implementation of the function exists_values that is used in the implementation of enforce_consistency. This function is called with five arguments.

- (a) var is variable.
- (b) val is a value that is to be assigned to var.
- (c) f is a constraint such that the variable var occurs in f
- (d) Vars is the set of all those other variables occurring in f, i.e. the set of those variables that occur in f but that are different from var.
- (e) ValuesPerVar is a dictionary associating the set of possible values with each variable.

The function checks whether the partial assignment $\{var \mapsto val\}$ can be extended so that the constraint f is satisfied. To this end it needs to create the set of all possible assignments. This set is generated using the function all_assignments. This function gets a set of variables Vars and a

dictionary that assigns to every variable var in Vars the set of values that might be assigned to var. It returns a list containing all possible variable assignments. The implementation proceeds as follows:

- 1. As the argument Variables is a frozenset but we need to modify this set for the recursive call of all_assignments, we transform the frozenset into a set.
- 2. If the set of variables Vars is empty, the empty dictionary can serve as a mapping that assigns a value to every variable in Vars.
- 3. Otherwise, we remove a variable var from Vars and get the set of Values that can be assigned to var.
- 4. Recursively, we create the set of all Assignments that associate values with the remaining variables.
- 5. Finally, the set of all possible assignments is the set of all combinations of assigning a value val ∈ Values to var and assigning the remaining variables according to an assignment A ∈ Assignments. Here we have to make use of the function extend that takes a dictionary A, a key x not occurring in A and a value v and returns a new dictionary that maps x to v and otherwise coincides with A.

On one hand, consistency checking is a pre-processing step that creates a lot of overhead.¹ Therefore, it might actually slow down the solution of some constraint satisfaction problems that are easy to solve using just backtracking and constraint propagation. On the other hand, many difficult constraint satisfaction problems can not be solved without consistency checking.

Figure 3.22 on page 77 shows how consistency checking is integrated into a constraint solver as a pre-processing step. The procedure solve(P) takes a constraint satisfaction problem P as input. The function solve converts the CSP P into an augmented CSP where every constraint f is annotated with the variables occurring in f. Furthermore, the function solve maintains the following data structures:

- 1. VarsInConstrs is the set of all variables occurring in any constraint.
- 2. ValuesPerVar is a dictionary mapping variables to sets of values. For every variable x occurring in a constraint of P, the expression ValuesPerVar(x) is the set of values that can be used to instantiate the variable x. Initially, ValuesPerVar(x) is set to Values, but as the search for a solution proceeds, the sets ValuesPerVar(x) are reduced by removing any values that cannot be part of a solution.
- 3. Annotated is a dictionary. For every constraint f we have that Annotated[f] is the set of all variables occurring in f.
- 4. UnaryConstrs is a set of pairs of the form (f, V) where f is a constraint containing only a single variable and V is the set containing just this variable.
- 5. OtherConstrs is a set of pairs of the form (f, V) where f is a constraint containing more than one variable and V is the set of all variables occurring in f.
- 6. Connected is a dictionary mapping variables to sets of variables. If x is a variable, then Connected[x] is the set of those variables y such that there is a constraint f that mentions both the variable x and the variable y.

¹To be fair, the implementation shown in this section is far from optimal. In particular, by remembering which combinations of variables and values work for a given formula, the overhead can be reduced significantly. I have refrained from implementing this optimization because I did not want the code to get too complex.

```
def solve(P):
         Variables, Values, Constraints = P
         VarsInConstrs = union([ collect_variables(f) for f in Constraints ])
         MisspelledVars = (VarsInConstrs - Variables) | (Variables - VarsInConstrs)
         if MisspelledVars:
             print("Did you misspell any of the following Variables?")
6
             for v in MisspelledVars:
                 print(v)
         ValuesPerVar = { x: Values.copy() for x in Variables }
                      = { f: collect_variables(f) for f in Constraints }
10
         UnaryConstrs = { (f, V) for f, V in Annotated.items()
11
                                  if len(V) == 1
12
                        }
13
         OtherConstrs = { (f, V) for f, V in Annotated.items()
14
                                  if len(V) >= 2
                        }
16
                      = {}
         Connected
17
         Var2Formulas = variables_2_formulas(OtherConstrs)
18
         for x in Variables:
19
             Connected[x] = union([ V for f, V in Annotated.items()
20
                                       if x in V
                                   ]) - { x }
22
         try:
             for f, V in UnaryConstrs:
                 var
                                    = arb(V)
25
                 ValuesPerVar[var] = solve_unary(f, var, ValuesPerVar[var])
26
             enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected)
27
             for x, Values in ValuesPerVar.items():
28
                 print(f'{x}: {Values}')
29
             return backtrack_search({}, ValuesPerVar, OtherConstrs)
31
         except Backtrack:
             return None
```

Figure 3.22: A constraint solver with consistency checking as a preprocessing step.

7. Var2Formulas is a dictionary mapping variables to sets of formulas. For every variable x, Var2Formulas [x] is the set of all those non-unary constraints f such that x occurs in f.

After initializing these data structures, the unary constraints are immediately solved. Then the function enforce_consistency performs consistency maintenance: Formally, we define: A value v is consistent for x with respect to the constraint f iff the partial assignment $\{x \mapsto v\}$ can be extended to an assignment A satisfying the constraint f, i.e. for every variable y_i occurring in f there is a value $w_i \in ValuesPerVar[y]$ such that

$$evaluate(f, \{x \mapsto v, y_1 \mapsto w_1, \cdots, y_n \mapsto w_n\}) = True.$$

The call to enforce_consistency shrinks the sets ValuesPerVars[x] until all values in ValuesPerVars[x] are consistent with respect to all constraints.

Finally, backtrack_search is called to solve the remaining constraint satisfaction problem by the means of both backtracking and constraint propagation.

3.6 Local Search

There is another approach to solve constraint satisfaction problems. This approach is known as local search. The basic idea is simple: Given as constraint satisfaction problem \mathcal{C} of the form

 $\mathcal{P} := \langle \mathtt{Variables}, \mathtt{Values}, \mathtt{Constraints} \rangle$

local search works as follows:

- 1. Use consistency checking as an optional pre-processing step.
- 2. Initialize the values of the variables in Variables randomly.
- 3. If all Constraints are satisfied, return the solution.
- 4. For every $x \in Variables$, count the number of unsatisfied constraints that involve the variable x.
- 5. Set maxNum to be the maximum of these numbers, i.e. maxNum is the maximal number of unsatisfied constraints for any variable.
- 6. Compute the set maxVars of those variables that have maxNum unsatisfied constraints.
- 7. Randomly choose a variable x from the set maxVars.
- 8. Find a value $d \in Values$ such that by assigning d to the variable x, the number of unsatisfied constraints for the variable x is minimized.

If there is more than one value d with this property, choose the value d randomly from those values that minimize the number of unsatisfied constraints.

9. Rinse and repeat until a solution is found.

Figure 3.23 on page 79 shows the preprocessing step. The function solve takes a constraint satisfaction problem P as its argument and performs consistency checking similar to the algorithm discussed in the previous section. Following the preprocessing it calls the function local_search that solves the given Csp.

Figure 3.24 on page 80 shows an implementation of local search in *Python*. We proceed to discuss this program line by line.

- 1. The function local_search takes three parameters.
 - (a) Variables is the set of all variables occurring in the given CSP.
 - (b) ValuesPerVar is a dictionary. For every variable x, ValuesPerVar[x] is the set of values that can be used to instantiate x.
 - (c) Annotated is a dictionary. For every constraint f, Annotated [f] is the set of variables occurring in f.

If the computation is successful, local_search returns a dictionary that encodes a solution of the given CSP by mapping variables to values.

```
def solve(P):
         Variables, Values, Constraints = P
2
         VarsInConstrs = union([ collect_variables(f) for f in Constraints ])
3
         MisspelledVars = (VarsInConstrs - Variables) | (Variables - VarsInConstrs)
         if MisspelledVars:
             print("Did you misspell any of the following Variables?")
6
             for v in MisspelledVars:
                 print(v)
         ValuesPerVar = { x: Values for x in Variables }
                      = { f: collect_variables(f) for f in Constraints }
10
         Connected
                      = {}
11
         Var2Formulas = variables_2_formulas(Annotated)
12
         for x in Variables:
13
             Connected[x] = union([V for f, V in Annotated.items() if x in V]) - {x}
14
         try:
             enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected)
16
         except Failure:
17
             return None
18
         return local_search(Variables, ValuesPerVar, Annotated)
19
```

Figure 3.23: A constraint solver using local search.

2. The set Variables is turned into a list. This is necessary because the function

```
random.choice(L)
```

that is used to select a random element from L expects its argument L to be indexable, i.e. for a number $k \in \{0, \dots, \text{len}(L) - 1\}$ the expression L[k] needs to be defined.

- 3. Assign is a dictionary mapping all variables from the set Variables to values from the set Values. Initially the values are assigned randomly.
- 4. The variable iteration counts the number of times that we have changed the assignment Assign by reassigning a variable.
- 5. If we have reassigned a variable x in the last iteration of the loop, then we do not want to reassign it again in the next step since otherwise the program could get stuck in an infinite loop. Therefore, the variable lastVar stores the variable that has been reassigned in the previous iteration. We will ensure that in the next iteration step, another variable is chosen for reassignment.
- 6. At the beginning of the while loop, we count the number of conflicts for all variables, i.e. if x is a variable that is different from the variable that has been reassigned in the last iteration, then we count the number of conflicts that x causes. This number is defined as the number of constraints f such that
 - (a) x occurs in f and
 - (b) f is not satisfied.

```
def local_search(Variables, ValuesPerVar, Annotated):
         Variables = list(Variables)
                   = { x: random.choice(list(ValuesPerVar[x])) for x in Variables }
         iteration = 0
         lastVar
                   = arb(Variables)
         while True:
             Conflicts = [(numConflicts(x, Assign, Annotated), x) for x in Variables
                                                                     if x != lastVar
             maxNum, _ = Set.last(cast_to_Set(Conflicts))
10
             if maxNum == 0 and numConflicts(lastVar, Assign, Annotated) == 0:
11
                 print(f'Number of iterations: {iteration}')
12
                 return Assign
13
             if iteration % 11 == 0:
                                         # avoid infinite loop
14
                 x = random.choice(Variables)
                       # choose var with max number of conflicts
             else:
16
                 FaultyVars = [ var for (num, var) in Conflicts if num == maxNum ]
17
                 x = random.choice(FaultyVars)
18
             if iteration % 13 == 0:
                                            # avoid infinite loop
19
                 newVal = random.choice(list(ValuesPerVar[x]))
20
             else:
21
                 Conflicts = [ (numConflicts(x, extend(Assign, x, v), Annotated), v)
22
                                for v in ValuesPerVar[x]
                              ]
                 minNum, _ = Set.first(cast_to_Set(Conflicts))
25
                 ValuesForX = [ val for (n, val) in Conflicts if n == minNum ]
26
                             = random.choice(ValuesForX)
                 newVal
27
             Assign[x] = newVal
28
             lastVar
29
             iteration += 1
30
```

Figure 3.24: Implementation of local search.

This is done using the function numConflicts shown in Figure 3.25 on page 81. The list Conflicts defined in line 7 contains pairs of the form (n, x) where x is a variable and n is the number of conflicts that this variable is involved in.

- 7. In line 10 the list Conflicts is turned into a set that is represented as an ordered binary set. This set is effectively a priority queue that is ordered by the number of conflicts. We pick the variable with the most conflicts from this set and store the number of conflicts in maxNum, i.e. maxNum is the maximum number of conflicts that any variable is involved in.
- 8. Now if maxNum is 0 and additionally the variable lastVar that is excluded from the computation of the set Conflicts has no conflicts, then the given CSP has been solved and the solution is returned.
- 9. Otherwise, the list FaultyVars defined in line 17 collects those variables that have a maximal

number of conflicts.

- 10. In line 18 we choose a random variable x from this list as the variable to be reassigned. However, this is only done ten out of eleven times. In order to avoid running into an infinite loop where we keep changing the same variables, every 11^{th} iteration chooses x randomly. This is controlled by the test iteration % 11 == 0 in line 16.
- 11. Line 22 computes a list Conflicts that this time contains pairs of the form (n, v) where n is the number of conflicts that the variable x would cause if we would assign the value v to x.
- 12. Line 25 casts the list Conflicts into a set that is represented as an ordered binary tree. This ordered binary tree is used as a priority queue that is ordered by the number of conflicts. We pick the smallest number of conflicts that any value v causes when x is assigned to v.
- 13. ValuesForX is the list of those values that cause only minNum conflicts when assigned to x.
- 14. newVal is a random element from this list that is then assigned to x. Again, this is only done twelve out of thirteen times. The 13th time a random value is assigned to x instead.
- 15. In line 29 we remember that we have reassigned x in this iteration so that we don't reassign x in the next iteration again.

Figure 3.25: The function numConflicts.

The function numConficts is shown in Figure 3.25 on page 81. If x is a variable, Assign is a variable assignment and Annotated is a list of pairs of the form (f, V) where f is a constraint and V is the set of variables occurring in f, then numConficts(x, Assign, Annotated) is the number of conflicts caused by the variable x.

Using the program discussed in this section, the n queens problem can be solved for a n=1000 in 30 minutes. As the memory requirements for local search are small, even much higher problem sizes can be tackled if sufficient time is available. It is a fact that often large problems, which are not inherently difficult, can be solved much faster with local search than with any other algorithm. However, we have to note that local search is incomplete: If a constraint satisfaction problem \mathcal{P} has no solution, then local search loops forever. Therefore, in practise a dual approach is used to solve a constraint satisfaction problem. The constraint solver starts two threads: The first search does local search, the second thread tries to solve the problem via some refinement of backtracking. The first thread that terminates wins. The resulting algorithm is complete and, for a solvable problem, will have a performance that is similar to the performance of local search. If the problem is unsolvable, this will eventually be discovered by backtracking. Note, however, that the constraint satisfaction problem is NP-complete. Hence, it is unlikely that there is an efficient algorithm that works always. However, today many practically relevant constraint satisfaction problems can be solved in a reasonably short time.

3.7 Z3

We conclude this chapter with a discussion of the solver Z3. Z3 implements most of the state-of-the-art constraint solving algorithms and is exceptionally powerful. We introduce Z3 via a series of examples.

3.7.1 A Simple Text Problem

The following is a simple text problem from my old 8th grade math book.

- I have as many brothers as I have sisters.
- My sister has twice as many brothers as she has sisters.
- How many children does my father have?

However, in order to solve this puzzle we need two additional assumptions.

- 1. My father has no illegitimate children.
- 2. All of my fathers children identify themselves as either male or female.

Strangely, in my old math book these assumptions are not mentioned.

We can now infer the number of children. If we denote the number of boys with the variable b and the number of girls with g, the problem statements are equivalent to the following two equations:

- (a) b-1=g.
- (b) $2 \cdot (q-1) = b$.

Before we can start to solve this problem, we have to install Z3 via pip using the following command in the shell:

Once we have done this and we have added the directory

to the environment variable PATH, we can use the file shown in Figure 3.26 to solve the problem. The command to invoke Z3 has the form

where file.z3 is the name of the file that stores the Z3 specification of the problem.

(a) Line 1 and 2 declare the variables b and g as integer variables.

With Z3 we are not confined to use a finite set of values. Instead we can use integer variables and floating point variables.

The syntax of Z3 files is similar to the syntax of the programming language lisp. Later, we will only use the *Python* API of Z3. Therefore, you do not need to worry about this syntax.

(b) Line 4 specifies the equation b-1=g as a constraint.

Note that we have to use prefix notation for all operators.

- (c) Similarly, line 5 specifies the equation 2*(g-1)=b as a constraint.
- (d) Line 7 asks Z3 to check whether the problem is solvable.

```
(declare-const b Int)
(declare-const g Int)

(assert (= (- b 1) g))
(assert (= (* 2 (- g 1)) b))

(check-sat)
(get-model)
```

Figure 3.26: Solving a simple text problem with Z3.

(e) Line 8 prints the solution of the problem.

If we run this command with the specification shown in Figure 3.26, then we get the output shown below:

```
sat
(
(define-fun b () Int 4)
(define-fun g () Int 3)
)
```

The string "sat" tells us that the problem is solvable and the following lines show that b = 4 and g = 3 is the solution, i.e. there are 4 boys and 3 girls.

Instead of using the command line to solve CSPs we will utilize the *Python* interface of Z3. There are two reasons why this is more convenient:

- 1. In an interesting CSP there can easily be hundreds of variables and thousands of constraints. It would be very inconvenient if we had to write these variables and constraints manually into a file.
- 2. The *Python* interface allows us to extract the solution that has been computed so that we can then proceed to use the values of the solution in our own programs.

The Python program shown in Figure 3.27 solves the text problem given above via the *Python API* of Z3.

1. In line 1 we import the module **z3** so that we can use the Python API of **Z3**. The documentation of this API is available at the following address:

```
https://ericpony.github.io/z3py-tutorial/guide-examples.htm
```

- 2. Lines 3 and 4 creates the Z3 variables boys and girls as integer valued variables. The function Int takes one argument, which has to be a string. This string is the name of the variable. We store these variables in Python variables of the same name. It would be possible to use different names for the Python variables, but that would be very confusing.
- 3. Line 6 creates an object of the class Solver. This is the constraint solver provided by Z3.

```
import z3
2
     boys = z3.Int('boys')
3
     girls = z3.Int('girls')
     S = z3.Solver()
     S.add(boys - 1 == girls)
     S.add(2 * (girls - 1) == boys)
     S.check()
10
     solution = S.model()
11
12
     b = solution[boys].as_long()
13
     g = solution[girls].as_long()
14
15
     print(f'My father has {b + g} children.')
```

Figure 3.27: Solving a simple text problem.

- 4. Lines 8 and 9 add the constraints expressing that the number of girls is one less than the number of boys and that my sister has twice as many brothers as she has sisters as constraints to the solver S.
- 5. In line 10 the method check examines whether the given set of constraints is satisfiable. In general, this method returns one of the following results:
 - (a) sat is returned if the problem is solvable, (sat is short for satisfiable)
 - (b) unsat is returned if the problem is unsolvable,
 - (c) unknown is returned if Z3 is not powerful enough to solve the given problem.
- 6. Since in our case the method check returns sat, we can extract the solution that is computed via the method model in line 11.
- 7. In order to extract the values that have been computed by Z3 for the variables boys and girls, we can use dictionary syntax and write solution[boys] and solution[girls] to extract these values. However, these values are not stored as integers but rather as objects of the class IntNumRef, which is some internal class of Z3 to store integers. This class provides the method as_long that converts its object into an integer number.

Exercise 6: Solve the following text problem using Z3.

- (a) A Japanese deli offers both penguins and parrots.
- (b) A parrot and a penguin together cost 666 bucks.
- (c) The penguin costs 600 bucks more than the parrot.
- (d) What is the price of the parrot?

 \Diamond

You may assume that the prizes of these delicacies are integer valued.

Exercise 7: Solve the following text problem using Z3.

- (a) A train travels at a uniform speed for 360 miles.
- (b) The train would have taken 48 minutes less to travel the same distance if it had been faster by 5 miles per hour.
- (c) Find the speed of the train!

Hints:

- 1. As the speed is a real number you should declare this variable via the Z3 function Real instead of using the function Int.
- 2. 48 minutes are four fifth of an hour. The fraction $\frac{4}{5}$ can be represented in Z3 by the expression Q(4, 5).
- 3. When you formulate the information given above, you will get a system of **non-linear** equations, which is equivalent to a quadratic equation. This quadratic equation has two different solutions. One of these solutions is negative. In order to exclude the negative solution you need to add a constraint stating that the speed of the train has to be greater than zero.
- 4. The solution will be some real number which is represented internally as an object of type RatNumRef. If o is an object of this type, then this object can be converted to a string as follows:

o.as_decimal(17)

Here, 17 is the number of digits following the decimal point. This string can be then converted to a float by using the function float.

3.7.2 The Knight's Tour

In this subsection we will solve the puzzle The Knight's Tour using Z3. This puzzle asks whether it is possible for a knight to visit all 64 squares of the board and return to its starting square in 64 moves. The tour starts in one of the corners of the board.

In order to model this puzzle as a constraint satisfaction problem we first have to decide on the variables that we want to use. The idea is to have 65 variables that describe the position of the knight after its i^{th} move where $i=0,1,\cdots,64$. However, it turns out that it is best to split the values of these positions up into a row and a column. If we do this, we end up with 130 variables of the form

$$R_i \text{ and } C_i \quad \text{ for } i \in \{0, 1, \dots, 64\}.$$

Here R_i denotes the row of the knight after its i^{th} move, while C_i denotes the corresponding column. Next, we have to formulate the constraints. In this case, there are two kinds of constraints:

- 1. We have to specify that the move from the position $\langle R_i, C_i \rangle$ to the position $\langle R_{i+1}, C_{i+1} \rangle$ is legal move for a knight. In chess, there are two ways for a knight to move:
 - (a) The knight can move two squares horizontally left or right followed by moving vertically one square up or down, or
 - (b) the knight can move two squares vertically up or down followed by moving one square left or right.

Figure 3.28 shows all legal moves of a knight that is positioned in the square e4. Therefore, a formula that expresses that the i^{th} move is a legal move of the knight is a disjunction of the following eight formulas that each describe one possible way for the knight to move:

- (a) $R_{i+1} = R_i + 2 \wedge C_{i+1} = C_i + 1$,
- (b) $R_{i+1} = R_i + 2 \land C_{i+1} = C_i 1$,
- (c) $R_{i+1} = R_i 2 \wedge C_{i+1} = C_i + 1$,
- (d) $R_{i+1} = R_i 2 \land C_{i+1} = C_i 1$,
- (e) $R_{i+1} = R_i + 1 \wedge C_{i+1} = C_i + 2$,
- (f) $R_{i+1} = R_i + 1 \land C_{i+1} = C_i 2$,
- (g) $R_{i+1} = R_i 1 \wedge C_{i+1} = C_i + 2$,
- (h) $R_{i+1} = R_i 1 \land C_{i+1} = C_i 2$.
- 2. Furthermore, we have to specify that the position $\langle R_i, C_i \rangle$ is different from the position $\langle R_j, C_j \rangle$ if $i \neq j$.



Figure 3.28: The moves of a knight, courtesy of chess.com.

Figure 3.29 shows how we can formulate the puzzle using Z3.

1. In line 1 we import everything from the library z3 so that we can write, e.g. And(x, y) instead of having to write z3. And(x, y).

The expression z3.And(x,y) computes the conjunction of x and y.

2. It is not convenient to declare all of the 130 variables R_i and C_i for $i = 0, 1, \dots, 64$ explicitly. Instead, we will write a function that creates and declares these variables. To implement this

```
import * from z3
     def row(i): return f'R{i}'
     def col(i): return f'C{i}'
     def all_variables():
         Variables = set()
         for i in range(64+1):
             Variables.add(row(i))
             Variables.add(col(i))
         return Variables
10
11
12
     def is_knight_move(i):
         r = row(i)
13
         c = col(i)
         rX = row(i+1)
         cX = col(i+1)
         Formulas = set()
         for delta_r, delta_c in [(1, 2), (2, 1)]:
             Formulas.add(f'And(\{rX\} == \{r\} + \{delta_r\}, \{cX\} == \{c\} + \{delta_c\})')
             Formulas.add(f'And(\{rX\} == \{r\} + \{delta_r\}, \{cX\} + \{delta_c\} == \{c\})')
20
             Formulas.add(f'And(\{rX\} + \{delta_r\} == \{r\}, \{cX\} == \{c\} + \{delta_c\})')
^{21}
             Formulas.add(f'And({rX} + {delta_r} == {r}, {cX} + {delta_c} == {c})')
         return 'Or(' + ', '.join(Formulas) + ')'
24
     def all_different():
25
         Result = set()
26
         for i in range(62+1):
27
             for j in range (i+1, 63+1):
                  Result.add(f'Or(\{row(i)\}) = \{row(j)\}, \{col(i)\} = \{col(j)\})')
29
         return Result
31
     def all_constraints():
32
         Constraints = all_different()
33
         Constraints.add(f'{row(0)} == 0')
34
         Constraints.add(f'{col(0)} == 0')
35
         Constraints.add(f'\{row(64)\} == 0')
         Constraints.add(f'\{col(64)\} == 0')
         for i in range(63+1):
             Constraints.add(is_knight_move(i))
40
         for i in range (64+1):
             Constraints.add(f'{row(i)} >= 0')
             Constraints.add(f'\{col(i)\} >= 0')
42
43
         return Constraints
```

Figure 3.29: The Knight's Tour: Computing the constraints.

function, we define the auxiliary functions row and col in line 2 and 3. Given a natural number i, the expression row(i) returns the string 'Ri' and col(i) returns the string 'Ci'. These strings in turn represent the variables R_i and C_i .

- 3. The function all_variables returns a set of all variable names.
- 4. The function is knight move checks whether the move from position i specified as $\langle R_i, C_i \rangle$ to the position $\langle R_{i+1}, C_{i+1} \rangle$ is a legal move for a knight.
- 5. The function all_different computes a set of formulas that state that the positions $\langle \mathbf{R}_i, \mathbf{C}_i \rangle$ for $i = 0, 1, \dots, 63$ are all different from each other.
- 6. The function all_constraints computes the set of all constraints. In addition to the constraints already discussed this function specifies that the knight starts its tour at the leftmost topmost corner of the board and that the tour also ends in this corner.

Additionally there are constraints that the variables R_i and C_i are all non-negative. These constraints are needed as we will model the variables with bit vectors of length 4. These bit vectors store integers in two's complement representation. In two's complement representation of a bit vector of length 4 we can model integers from the set $\{-8, \dots, 7\}$. If we add the number 1 to a 4-bit bit vector v that represents the number 7, then an overflow will occur and the result will be -8 instead of 8. This could happen in the additions that are performed in the formulas computed by the function is knight move. We can exclude these cases by adding the constraints that all variables are non-negative.

```
def solve(Constraints, Variables):
         Environment = {}
2
         exec('import z3', Environment)
         for v in Variables:
             exec(f'\{v\} = z3.BitVec(f''\{v\}'', 4)', Environment)
         s = z3.Solver()
         for c in Constraints:
             s.add(eval(c, Environment))
         result = str(s.check())
         if result == 'sat':
10
             m = s.model()
11
             S = { v: m[eval(v, Environment)] for v in Variables }
12
             return S
13
         elif result == 'unsat':
14
             print('The problem is not solvable.')
15
         else:
16
             print('Z3 cannot determine whether the problem is solvable.')
17
```

Figure 3.30: The function solve.

Finally, the function solve that is shown in Figure 3.30 on page 88 can be used to solve the puzzle. This function takes two arguments:

(a) Constraints is a set of strings that are interpreted as Z3 constraints.

 \Diamond

(b) Variables is a set of strings that are interpreted as variables.

The purpose of the function **solve** is to find a solution of the given Csp. If successful, it returns a dictionary that maps every variable name to the corresponding value of the solution that has been found.

- 1. In line 1 we define the dictionary Environment which will serve as the local environment for the functions exec and eval below.
- 2. We import everything form the package z3 into this environment in line 45.
- 3. Then we declare that the strings from the set Variable represent Z3 bit-vector variables of length 4.
- 4. We create a solver object in line 6 and add the constraints to this solver in the following two lines
- 5. The function check tries to build a model satisfying the constraints, while the function model extracts this model if it exists.
- 6. Finally, in line 12 we create a dictionary that maps all of our variables to the corresponding values that are found in the model. Note that we have to turn the variable names, that are stored as strings in the set Variables, into objects that represent the corresponding Z3 variables using the function eval.

This dictionary is then returned.

	3	9						7
			7			4	9	2
				6	5		8	3
			6		3	2	7	
				4		8		
5	6							
		5	2		9			1
	2	1					4	
7						5		

Table 3.1: A super hard sudoku from the magazine "Zeit Online".

Exercise 8: Table 3.1 on page 89 shows a sudoku that I have taken from the Zeit Online magazine. Solve this Sudoku using Z3. I have written a frame for you to use that can be found at

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/ Python/2 Constraint Solver/Sudoku-Z3-Frame.ipynb

In this chapter we could only give a glimpse of the theory of constraint satisfaction problems. For further details on the theory of CSPs, consult the book Constraint Processing by Rina Dechter [Dec03].

Chapter 4

Playing Games

One major breakthrough for the field of artificial intelligence happened in 1997 when the chess-playing computer Deep Blue was able to beat the World Chess Champion Garry Kasparov by $3^1/2-2^1/2$. While Deep Blue was based on special hardware, according to the computer chess rating list of the 11th of February 2023, the chess program Stockfish runs on ordinary desktop computers and has an Elo rating of 3530. To compare, according to the Fide list of February 2023, the current World Chess Champion Magnus Carlsen has an Elo rating of 2852. If two players differ by more than 400 in their ELO ranking, the lower ranked player does not stand a chance to win or even draw against the higher ranked player. Hence, Magnus Carlsen wouldn't stand a chance to win or draw a game against Stockfish. In 2017, at the Future of Go Summit, the computer program AlphaGo was able to beat Ke Jie, who was at that time considered to be the best human Go player in the world. Besides Go and chess, there are many other games where today the performance of a computer exceeds the performance of human players. To name just one more example, in 2019 the program Pluribus was able to beat fifteen professional poker players in six-player no-limit Texas Hold'em poker resoundingly. ¹

This chapter is structured as follows:

- (a) We define the notion of deterministic two player zero sum games in the next section.
- (b) To illustrate this definition we describe the game tic-tac-toe in this framework.
- (c) The minimax algorithm is a simple algorithm to play games and is described next.
- (d) Alpha-beta pruning is an improvement of the minimax algorithm.
- (e) Finally, we consider the case of those games that, due to memory limitations, can not be solved with the pure version of alpha-beta pruning. For these games we discuss depth-limited adversarial search.

4.1 Basic Definitions

In order to investigate how a computer can play a game we define a game \mathcal{G} as a six-tuple

 $\mathcal{G} = \langle \mathtt{States}, s_0, \mathtt{Players}, \mathtt{nextStates}, \mathtt{finished}, \mathtt{utility} \rangle$

where the components are interpreted as follows:

1. States is the set of all possible states of the game.

We will only consider games where the set States is finite.

¹Well-informed circles report that all 15 professional players had to go home stark naked.

- 2. $s_0 \in \text{States}$ is the start state.
- 3. Players is the list of the players of the game. The first element in Players is the player to start the game and after that the players take turns. As we only consider two person games, we assume that Players is a list of length two.
- 4. nextStates is a function that takes a state $s \in$ States and a player $p \in$ Players and returns the set of states that can be reached if the player p has to make a move in the state s. Hence, the signature of nextStates is given as follows:

```
\mathtt{nextStates}: \mathtt{States} \times \mathtt{Players} \to 2^{\mathtt{States}}.
```

5. finished is a function that takes a state s and decides whether the games is finished. Therefore, the signature of finished is

```
finished: States \to \mathbb{B}.
```

Here, \mathbb{B} is the set of Boolean values, i.e. we have $\mathbb{B} := \{ \text{true}, \text{false} \}$.

Using the function finished, we define the set TerminalStates as the set of those states such that the game has finished, i.e. we define

```
TerminalStates := \{s \in \text{States} \mid \text{finished}(s)\}.
```

6. utility is a function that takes a state s such that $s \in \text{TerminalStates}$ as its argument. It returns the value that the game has for the first player, i.e. for Players[1]. In general, a value is a real number, but in all of our examples, this value will be an element from the set $\{-1,0,+1\}$. If utility(s) = -1, then the first player has lost the game, if utility(s) = 1, then the first player has won the game, and if utility(s) = 0, then the game drawn. Hence the signature of utility is

```
utility: TerminalStates \rightarrow \{-1, 0, +1\}.
```

We will only consider so called two person zero sum games. This means that the list Players has exactly two elements. If we call these players A and B, i.e. if we have

$$\mathtt{Players} = [\mathtt{A},\mathtt{B}],$$

then the game is called a zero sum game if A has won the game if and only if B has lost the game and vice versa. Games like Go, chess, and draughts are two person zero sum games. We proceed to discuss a simple example.

4.2 Tic-Tac-Toe

The game tic-tac-toe is played on a square board of size 3×3 . On every turn, the first player puts an "X" on one of the free squares of the board when it is her turn, while the second player puts an "O" onto a free square when it is his turn. If the first player manages to place three Xs in a row, column, or diagonal, she has won the game. Similarly, if the second player manages to put three Os in a row, column, or diagonal, he is the winner. Otherwise, the game is drawn. In this section we present two different implementations of tic-tac-toe:

- 1. We begin with a naive implementation of tic-tac-toe that is easy to understand but has a high memory footprint.
- 2. After that, we present an implementation that is based on bitboards and has only a fraction of the memory requirements of the naive implementation.

4.2.1 A Naive Implementation of Tic-Tac-Toe

```
gPlayers = [ "X", "0" ]
              = tuple( tuple(" " for col in range(3)) for row in range(3))
2
     def to_list (State): return [list(row) for row in State]
3
     def to_tuple(State): return tuple(tuple(row) for row in State)
4
     def empty(State):
         return [ (row, col) for row in range(3)
                              for col in range(3)
8
                              if State[row][col] == ' '
9
                ]
10
11
     def next_states(State, player):
12
         Empty = empty(State)
13
         Result = []
         for row, col in Empty:
15
             NextState
                                  = to_list(State)
16
             NextState[row][col] = player
17
             Result.append( to_tuple(NextState) )
18
         return Result
19
20
     gAllLines = [ [ (row, col) for col in range(3) ] for row in range(3) ] \
21
               + [ [ (row, col) for row in range(3) ] for col in range(3) ] \
               + [ [ (idx,
                              idx) for idx in range(3) ] ]
23
               + [ [ (idx, 2-idx) for idx in range(3) ] ]
24
25
     def utility(State):
26
         for Pairs in gAllLines:
27
             Marks = { State[row][col] for row, col in Pairs }
             if len(Marks) == 1 and Marks != { ' ' }:
                 if Marks == { 'X' }:
30
                      return 1
31
                 else:
32
                     return -1
33
         for row in range(3):
34
             for col in range(3):
35
                 if State[row][col] == ' ': # the board is not filled
                     return None
37
38
         return 0
39
     def finished(State): return utility(State) != None
40
```

Figure 4.1: A *Python* implementation of tic-tac-toe.

Figure 4.1 on page 92 shows a *Python* implementation of tic-tac-toe.

- 1. The variable gPlayers stores the list of players. Traditionally, we use the characters "X" and "O" to name the players.
- 2. The variable gStart stores the start state, which is an empty board. States are represented as tuples of tuples. If S is a state and $r, c \in \{0, 1, 2\}$, then S[r][c] is the mark in row r and column c. To represent states we have to use immutable data types, i.e. tuples instead of lists, as we need to store states in sets later. The entries in the inner tuples are the characters "X", "0", and the blank character "". As the state gStart is the empty board, it is represented as a tuple of three tuples containing three blanks each:

- 3. As we need to manipulate States, we need a function that converts them into lists of lists. This function is called to_list.
- 4. We also need to convert the lists of lists back into tuples of tuples. This is achieved by the function to_tuple.
- 5. Given a state S the function empty(S) returns the list of pairs (row, col) such that S[row][col] is a blank character. These pairs are the coordinates of the fields on the board S that are not yet occupied by either an "X" or an "0".
- 6. The function next_states takes a State and a player and computes the list of states that can be reached from State if player is to move next. To this end, it first computes the set of empty positions, i.e. those positions that have not yet been marked by either player. Every position is represented as a pair of the form (row, col) where row specifies the row and col specifies the column of the position. The position (row, col) is empty in State iff

```
State[row][col] = " ".
```

The computation of the empty position has been sourced out to the function empty. The function nextStates then iterates over these empty positions. For every empty position (row, col) it creates a new state NextState that results from the current State by putting the mark of player in this position. The resulting states are collected in the list Result and returned.

Note that we had to turn the **State** into a list of list in order to manipulate it. The manipulated State is then cast into a tuple of tuples.

7. The function utility takes a State as its argument. If the game is finished in the given State, it returns the value that this State has for the player "X". If the outcome of the game is not yet decided, the value None is returned instead.

In order to achieve its goal, the procedure first computes the set of all sets of coordinate pairs that either specify a horizontal, vertical, or diagonal line on a 3×3 tic-tac-toe board. Concretely, the variable gallLines has the following value:

```
 \begin{bmatrix} [(0,0),(0,1),(0,2)], & [(1,0),(1,1),(1,2)], & [(2,0),(2,1),(2,2)], \\ [(0,0),(1,0),(2,0)], & [(0,1),(1,1),(2,1)], & [(0,2),(1,2),(2,2)], \\ [(0,0),(1,1),(2,2)], & [(2,0),(1,1),(0,2)] \end{bmatrix}
```

The first line in this expression gives the sets of pairs defining the rows, the second line defines the columns, and the last line yields the two diagonals. Given a state State and a set Pairs, the set

```
Marks = { State[row][col] : (row, col) in Pairs }
```

is the set of all marks in the line specified by Pairs. For example, if

Pairs =
$$\{ (1, 1), (2, 2), (3, 3) \},$$

then Marks is the set of marks on the falling diagonal. The game is decided if all entries in a set of the form

```
Marks := { State[row][col] : (row, col) in Pairs }
```

where Pairs is a list from gAllLines either have the value "X" or the value "O". In this case, the set Marks has exactly one element which is different from the blank. If this element is "X", then the game is won by "X", otherwise the element must be "O" and hence the game has a value of -1 for "X".

If there are any empty squares on the board, but the game has not yet been decided, then the function returns None. Finally, if there are no more empty squares left, the game is a draw.

8. The function finished takes a State and checks whether the game is finished. To this end it computes the utility of the state for the player "X". If this utility is different from None, then game is finished. Note that it does make no difference whether we take the utility of the state for the player "X" or for the player "O": If the game is finished for "X", then it is also finished for "O" and vice versa.

The implementation shown so far has one important drawback: Every state needs 256 bytes in memory. This can be checked using the *Python* function sys.getsizeof. Therefore, we show a leaner implementation next.

4.2.2 A Bitboard-Based Implementation of Tic-Tac-Toe

If we have to reduce the memory requirements of the states, then we can store the states as integers. The first nine bits of these integers store the position of the Xes, while the next nine bits store the positions of the Os. This kind of representation where a state is coded as a series of bit in an integer is known as a bitboard. This is much more efficient than storing states as tuples of tuples of characters. Figure 4.2 on page 95 shows an implementation of tic-tac-toe that is based on a bitboard. We proceed to discuss the details of this implementation.

```
gPlayers = [0, 1]
     gStart = 0
2
3
     def set_bits(Bits):
4
         result = 0
5
         for b in Bits:
6
             result |= 1 << b
         return result
     set_bit = lambda n: 1 << n
10
11
     def empty(state):
12
         Free = { n for n in range(9) }
13
         Free -= { n for n in range(9) if state & (1 << n) != 0 }
14
         Free -= \{ n \text{ for } n \text{ in range(9) if state } \& (1 << (9 + n)) != 0 \}
15
         return Free
17
     def next_states(state, player):
18
         Empty = empty(state)
19
         Result = []
20
         for n in Empty:
21
              next_state = state | set_bit(player * 9 + n)
22
              Result.append(next_state)
         return Result
24
25
     gAllLines = [set_bits([0,1,2]), set_bits([3,4,5]), set_bits([6,7,8]),
26
                    set_bits([0,3,6]), set_bits([1,4,7]), set_bits([2,5,8]),
27
                    set_bits([0,4,8]), set_bits([2,4,6])
28
29
     def utility(state):
30
         for mask in gAllLines:
31
              if state & mask == mask:
32
                  return 1
                                          # player 'X' has won
33
              if (state >> 9) & mask == mask:
34
                  return -1
                                          # player 'O' has won
35
         # 511 == 2**9 - 1 = 0b1_1111_1111
36
         if (state & 511) | (state >> 9) != 511: # the board is not yet filled
37
             return None
         return 0 # it's a draw
39
40
     finished = lambda state: utility(state) != None
41
```

Figure 4.2: Tic-Tac-Toe implemented by a bitboard.

1. When we use bitboards to implement tic-tac-toe it is more convenient to store the players as numbers. The first player X is encoded as the number 0, while the second player O is encoded as

the number 1.

- 2. In the state gStart, no mark has been placed on the board. Hence all bits are unset and therefore this state is represented by the number 0.
- 3. The function set_bits takes a list of natural numbers as its argument Bits. These numbers specify the bits that should be set. It returns an integer where all bits specified in the argument Bits are set to 1 and all other bits are set to 0.
- 4. The function set_bit takes a natural number n as its argument. It returns a number where the nth bit is set to 1 and all other bits are set to 0.
- 5. Given a state that is represented as a number, the function empty(state) returns the set of indexes of those cells such that neither player X nor player O has placed a mark in the cell.
 - Note that there are 9 cells on the board. Each of these cells can hold either an 'X' or an '0'. If the i^{th} cell is marked with a 'X', then the i^{th} bit of state is set. If instead the i^{th} cell is marked with an '0', then the $(9+i)^{\text{th}}$ bit of state is set. If the i^{th} cell is not yet marked, then both the i^{th} bit and the $(9+i)^{\text{th}}$ bit are 0.
- 6. Given a state and the player who is next to move, the function next_states computes the set of states that can be reached from state. Note that player X is encoded as the number 0, while player O is encoded as the number 1.
- 7. The global variable gallLines is a list of eight bit masks. These masks can be used to test whether there are three identical marks in a row, column, or diagonal.
- 8. The function utility takes two arguments:
 - (a) state is an integer representing the board.
 - (b) player specifies a player. Here player X is encoded as the number 0, while player O is encoded as the number 1.

The function returns 1 if player has won the game, -1 if the game is lost for player, 0 if it's a draw, and None if the game has not yet been decided.

9. The function finished returns True if the game is over.

4.3 The Minimax Algorithm

Having defined the notion of a game, our next task is to come up with an algorithm that can play a game. The algorithm that is easiest to implement is the minimax algorithm. This algorithm is based on the notion of the value of a state. Conceptually, the notion of the value of a state is an extension of the notion of the utility of a state. While the utility is only defined for terminal states, the value is defined for all states. Formally, we define a function

$$maxValue: States \rightarrow \{-1, 0, +1\}$$

that takes a state $s \in S$ tates and returns the value that the state s has for the first player, who tries to maximize the value of the state, provided that both the player p and his opponent play optimally. The easiest way to define this function is via recursion. As the maxValue function is an extension of the utility function, the base case is as follows:

$$finished(s) \rightarrow maxValue(s) = utility(s).$$
 (1)

If the game is not yet finished, we define

$$\neg \mathtt{finished}(s) \to \mathtt{maxValue}(s) = \max \big(\big\{ \mathtt{minValue}(n) \mid n \in \mathtt{nextStates}(s, \mathtt{gPlayers}[0]) \big\} \big). \tag{2}$$

The reason is that, if the game is not finished yet, the maximizing player gPlayers[0] has to evaluate all possible moves. From these, the player will choose the move that maximizes the value of the game for herself. In order to do so, the player computes the set nextStates(s, gPlayers[0]) of all states that can be reached from the state s in any one move of the player gPlayers[0]. Now if n is a state that results from player gPlayers[0] making some move, then in state n it is the turn of the other player gPlayers[1] to make a move. However, this player is the minimizing player who tries to achieve the state with the minimal value. Hence, in order to evaluate the state n, we have to call the function minValue recursively as minValue(n). The function minValue has the same signature as maxValue and is defined by the following recursive equations

- 1. $finished(s) \rightarrow minValue(s) = utility(s)$.
- 2. $\neg finished(s) \rightarrow minValue(s) = min(\{maxValue(n) \mid n \in nextStates(s, gPlayers[1])\})$.

In the future we will sometimes speak of the value function. This name is used as a synonym for the function maxValue.

Figure 4.3 on page 98 shows an implementation of the functions maxValue and minValue. It also shows the function best_move. This function takes a State such that X is to move in this state. It returns a pair (v, s) where s is a state that is optimal for the player X and such that s can be reached in one step from State. Furthermore, v is the value of this state.

- (a) To this end, it first computes the set NS of all states that can be reached from the given State in one step if X is to move next.
- (b) bestValue is the best value that X can achieve in the given State.
- (c) BestMoves is the set of states that X can move to and that are optimal for her.
- (d) The function returns randomly one of those states ns ∈ NS such that the value of ns is optimal, i.e. is equal to bestValue. We use randomization here since we want to have more interesting games. If we would always choose the first state that achieves the best value, then our program would always make the same move in a given state. Hence, playing the program would get boring much sooner.

```
def maxValue(State):
         if finished(State):
             return utility(State)
         return max([ minValue(ns) for ns in next_states(State, gPlayers[0]) ])
    def minValue(State):
         if finished(State):
             return utility(State)
         return min([ maxValue(ns) for ns in next_states(State, gPlayers[1]) ])
10
    def best_move(State):
11
                   = next_states(State, gPlayers[0])
12
                   = maxValue(State)
         bestVal
13
        BestMoves = [s for s in NS if minValue(s) == bestVal]
14
        BestState = random.choice(BestMoves)
15
         return bestVal, BestState
```

Figure 4.3: The Minimax algorithm.

4.3.1 Memoization

Let us consider how many states have to be explored in the case of tic-tac-toe by the minimax algorithm described previously. We have 9 possible moves for player X in the start state, then the player O can respond with 8 moves, then there are 7 moves for player O and so on until in the end player X has only 1 move left. If we disregard the fact that some games are decided after fewer than 9 moves, the functions maxValue and minValue need to consider a total of

$$9 \cdot 8 \cdot 7 \cdot \ldots \cdot 2 \cdot 1 = 9! = 362\,880$$

different moves. However, if we count the number of possibilities of putting 5 $\,$ 0s and 4 $\,$ Xs on a 3×3 board, we see that there are only

$$\binom{9}{5} = \frac{9!}{5! \cdot 4!} = 126$$

possibilities, because we only have to count the number of ways that there are to put 5 0s on 9 different positions and that number is the same as the number of subsets of five elements from a set of 9 elements. Therefore, if we disregard the fact that some games are decided after fewer than nine moves, there are a factor of $5! \cdot 4! = 2880$ less terminal states than there are possible sequences of moves!

As we have to evaluate not just terminal states but all states, the saving is actually a bit smaller than 2880. The next exercise explores this in more detail.

We can use memoization to exploit the fact that the number of states is much smaller than the number of possible game sequences. Figure 4.4 on page 99 shows how this can be implemented.

```
gCache = {}
2
     def memoize(f):
3
         global gCache
         def f_memoized(*args):
              if args in gCache:
                  return gCache[args]
             result = f(*args)
             gCache[args] = result
10
             return result
11
12
         return f_memoized
13
14
     maxValue = memoize(maxValue)
15
     minValue = memoize(minValue)
16
```

Figure 4.4: Memoization.

- 1. gCache is a dictionary that is initially empty. This dictionary is used as a memory cache by the function memoize.
- 2. The function memoize is a second order function that takes a function f as its argument. It creates a memoized version of the function f: This memoized version of f, which is called f_memoized, first tries to retrieve the value of f from the dictionary gCache. If this is successful, the cached value is returned. Otherwise, the function f is called to compute the result. This result is then stored in the dictionary gCache before it is returned. The function memoize returns the memoized version of f.
- 3. All that needs to be done is then to memoize both the function maxValue and the function minValue. In this case it is not a problem that these functions share the same dictionary gCache because maxValue is only called for states where X has to make the next move, while minValue is only called for states where O has to make the next move. If this wouldn't be the case, the name of the function would have to be stored in gCache also.

```
def play_game(canvas):
         State = gStart
2
         while True:
             val, State = best_move(State);
             draw(State, canvas, f'For me, the game has the value {val}.')
             if finished(State):
                 final_msg(State)
                 return
             IPython.display.clear_output(wait=True)
             State = get_move(State)
10
             draw(State, canvas, '')
11
             if finished(State):
12
                 IPython.display.clear_output(wait=True)
13
                 final_msg(State)
14
                 return
```

Figure 4.5: The function play_game.

Figure 4.5 on page 100 presents the implementation of the function play_game that is used to play a game.

- 1. Initially, State is the startState.
- 2. As long as the game is not finished, the procedure keeps running.
- 3. We assume that the computer goes first.
- 4. The function best_move is used compute the move of the computer. This resulting state is then displayed.
- 5. After that, it is checked whether the game is finished.
- 6. If the game is not yet finished, the user is asked to make his move via the function get_move. The state resulting from this move is then returned and displayed.
- 7. Next, we have to check whether the game is finished after the move of the user has been executed.

In order to better understand the reason for using memoization in the implementation of the functions maxValue and minValue we introduce the following notion.

```
Definition 6 (Game Tree) Assume that
```

```
\mathcal{G}=\langle \mathtt{States}, s_0, \mathtt{Players}, \mathtt{nextStates}, \mathtt{finished}, \mathtt{utility} \rangle is a game. Then a play of length n is a list of states of the form [s_0, s_1, \cdots, s_n] such that s_o=\mathtt{Start} and \forall i\in\{0,\cdots,n-1\}: s_{i+1}\in\mathtt{nextStates}(s_i,p_i), where the players p_i are defined as follows:
```

 \Diamond

Therefore, p_i is the first element of the list Players if i is even and p_i is the second element of this list if i is odd. The game tree of the game \mathcal{G} is the set of all possible plays.

The following exercise shows why memoization is so important.

Exercise 9: In simplified tic-tac-toe the game only ends when there are no more empty squares left. The player X wins if she has more rows, columns, or diagonals of three Xs than the player O has rows, columns, or diagonals of three Os. Similarly, the player O wins if he has more rows, columns, or diagonals of three Os than the player X has rows, columns, or diagonals of three Xs. Otherwise, the game is a draw.

- (a) Derive a formula to compute the size of the game tree of simplified tic-tac-toe.
- (b) Write a short program to evaluate the formula derived in part (a) of this exercise.
- (c) Derive a formula that gives the number of all states of simplified tic-tac-toe.

Notice that this question does not ask for the number of all terminal states but rather asks for all states.

(d) Evaluate the formula derived in part (c) of this exercise.

Hint: You don't have to do the calculation in your head.

4.4 Alpha-Beta Pruning

In this section we discuss α - β -Pruning. This is a search technique that can prune large numbers of the search space and thereby increase the efficiency of a game playing program. The basic idea is to provide two additional arguments to the functions maxValue and minValue. Traditionally, these arguments are called α and β . In order to be able to distinguish between the old functions maxValue and minValue and its improved version, we call the improved versions alphaBetaMax and alphaBetaMin. The idea is that these functions are related by the following requirements:

1. As long as maxValue(s) is between α and β , the function alphaBetaMax computes the same result as the function maxValue, i.e. we have

```
\alpha \leq \max Value(s) \leq \beta \rightarrow \operatorname{alphaBetaMax}(s, \alpha, \beta) = \max Value(s).
```

2. If $maxValue(s) < \alpha$, we require that the value returned by alphaBetaMax is less than or equal to α , i.e. we have

```
\max Value(s) < \alpha \rightarrow alphaBetaMax(s, \alpha, \beta) \leq \alpha.
```

3. Similarly, if $maxValue(s) > \beta$, we require that the value returned by alphaBetaMax is bigger than or equal to β , i.e. we have

```
\beta < \mathtt{maxValue}(s) \rightarrow \beta \leq \mathtt{alphaBetaMax}(s, \alpha, \beta).
```

Similar to the way that the function maxValue is approximated by the function alphaBetaMax, the function minValue is approximated by the function alphaBetaMin. We have:

- 1. $\alpha \leq \min Value(s) \leq \beta \rightarrow alphaBetaMin(s, \alpha, \beta) = \min Value(s)$.
- 2. $minValue(s) < \alpha \rightarrow alphaBetaMin(s, \alpha, \beta) \le \alpha$.
- $3. \ \beta < \mathtt{minValue}(s) \ \rightarrow \ \beta \leq \mathtt{alphaBetaMin}(s,\alpha,\beta).$

Although alphaBetaMax(s) and alphaBetaMin(s) are only approximations of maxValue(s) and minValue(s), it turns out that these approximations are all that is needed. Once the function alphaBetaMax is implemented, the function maxValue can then be computed as

```
maxValue(s) := alphaBetaMax(s, -1, +1).
```

The reason is that we already know that $-1 \le \max Value(s) \le +1$ and hence the first case of the specification of alphaBetaMax guarantees that the equation

```
maxValue(s) = alphaBetaMax(s, -1, +1)
```

holds. Similarly, the function minValue can be computed as

```
minValue(s) := alphaBetaMin(s, -1, +1).
```

Figure 4.6 on page 102 shows an implementation of the functions alphaBetaMax and alphaBetaMin that satisfies the specification given above. Since alphaBetaMax and alphaBetaMin are implemented as mutually recursive functions, the fact that the implementations of alphaBetaMax and alphaBetaMin satisfy the specifications given above can be established by computational induction. A proof can be found in the article by Donald E. Knuth and Ronald W. Moore [KM75].

```
def alphaBetaMax(State, alpha, beta):
       if finished(State):
            return utility(State)
       for ns in next_states(State, gPlayers[0]):
            value = alphaBetaMin(ns, alpha, beta)
            if value >= beta:
                return value
            alpha = max(alpha, value)
       return alpha
9
10
   def alphaBetaMin(State, alpha, beta):
11
       if finished(State):
12
            return utility(State)
       for ns in next_states(State, gPlayers[1]):
14
            value = alphaBetaMax(ns, alpha, beta)
15
            if value <= alpha:
16
                return value
17
           beta = min(beta, value)
18
       return beta
19
```

Figure 4.6: α - β -Pruning.

We proceed to discuss the implementation of the function alphaBetaMax, which is shown in Figure 4.6 on page 102.

- 1. If State is a terminal state, the function returns the utility of the given State.
- 2. We iterate over all successor states $ns \in next_states(State, gPlayers[0])$.
- 3. We have to recursively evaluate the states ns with respect to the minimizing player gPlayers[1]. Hence we call the function alphaBetaMin when evaluating the state ns.

4. As the specification of alphaBetaMax asks us to compute the value of State only in those cases where it is less than or equal to beta, once we find a successor state s that has a value that is at least as big as beta we can stop any further evaluation of the successor states and return value.

This shortcut results in significant savings of computation time!

5. Once we have found a successor state that has a value value greater than alpha, we can increase alpha to value. The reason is, that once we know we can achieve value we are no longer interested in any smaller values. This is the reason for assigning the maximum of value and alpha to alpha.

After this assignment, alpha will be at least as big as value and according to the specification of alphaBetaMax we can therefore return alpha.

Remark: There is a nice simulator for alpha-beta-pruning available at the following web address: https://pascscha.ch/info2/abTreePractice/.

Exercise 10: The game Nim works as follows:

- (a) There are four rows of matches:
 - 1. the first row contains 1 match,
 - 2. the second row contains 3 matches,
 - 3. the third row contains 5 matches, and
 - 4. the fourth row contains 7 matches.
- (b) The player whose turn it is first selects a line.

Then she removes any number of matches from this line.

(c) The player that removes the last match has won the game.

Implement this game by adapting the notebook

Artificial-Intelligence/blob/master/Python/3 Games/Nim-Frame.ipynb. Then, test the game using the notebook

Artificial-Intelligence/blob/master/Python/3 Games/Alpha-Beta-Pruning-Pure.ipynb.

4.4.1 Alpha-Beta Pruning with Memoization

Adding memoization to the functions maxValue and minValue is non-trivial. If memoization is added in a naive way, then the cache might have many entries for the same state that differ only in their values for the parameters α and β . Although this is not a problem for trivial games like Tic-Tac-Toe, it becomes a problem once we try to implement more complex games like Connect Four. The reason is that for those games we are no longer able to compute the complete game tree. Instead, we need to approximate the value of a state with the help of a heuristic. Then, α and β will no longer be confined to the values from the set $\{-1,0,1\}$ and hence there will be many function calls of the form

$$maxValue(s, \alpha, \beta)$$

where the state s is the same but α and β are different. If we would try to store every combination of s, α , and β we would waste a lot of memory and, furthermore, we would have only a small number of cache hits. Therefore, we will now present a more effective way to cache the functions maxValue and

minValue. To this end we will define a function evaluate that is called as follows:

$$evaluate(s, f, \alpha, \beta)$$

where the parameters are interpreted as follows:

- (a) s is a state that is to be evaluated.
- (b) f is either the function alphaBetaMax or the function alphaBetaMin. If in state s the first player has to move, then f = alphaBetaMax, otherwise we have f = alphaBetaMin.
- (c) We interpret the parameters α and β in the same way as we did when we used them with the functions alphaBetaMax and alphaBetaMin.

The function evaluate encapsulates calls to the functions alphaBetaMax and alphaBetaMin. Given a state s, a function f, and the values of α and β , it first checks whether the value of $f(s, \alpha, \beta)$ has already been computed and is stored in the cache. If this is the case, the value is returned. Otherwise, the function f is called.

The function evaluate makes use of a global variable gCache. This variable is used as a cache to store the results of the function evaluate. This cache is implemented as a dictionary. The keys of this dictionary are the states, not triples of the form $\langle s, \alpha, \beta \rangle$. The values are stored in pairs of the form $\langle \text{flag}, v \rangle$, where v is a value computed by the function evaluate (s, f, α, β) , while $\text{flag} \in \{'\leq', '=', '\geq'\}$. The cache satisfies the following specification:

- 1. $gCache[s] = ('=', v) \rightarrow f(s, \alpha, \beta) = v$,
 - i.e. if the flag is equal to '=', then the value stored in gCache[s] is the value computed for the given state s by the function f.
- 2. $gCache[s] = ('\leq', v) \rightarrow f(s, \alpha, \beta) \leq v$,

i.e. if the flag is equal to ' \leq ', then the value stored in gCache[s] is an upper bound for the value returned from $f(s, \alpha, \beta)$.

- 3. $gCache[s] = ('>', v) \rightarrow f(s, \alpha, \beta) > v$,
 - i.e. if the flag is equal to ' \geq ', then the value stored in gCache[State] is a lower bound for $f(s, \alpha, \beta)$.

If gCache[s] is defined, then the computation of $evaluate(s, f, \alpha, \beta)$ proceeds according to the following case distinction:

1. If the stored value v is exact, we can return this value:

$$\mathtt{gCache}[s] = ('=', v) \rightarrow \mathtt{evaluate}(s, f, \alpha, \beta) = v.$$

2. If the stored value v is an upper bound and this upper bound is less or equal than α , then we know that the true value of s is less or equal than α and hence we can also return the value v:

$$gCache[s] = ('\leq', v) \land v \leq \alpha \rightarrow evaluate(s, f, \alpha, \beta) = v.$$

3. If the stored value v is an upper bound and this upper bound is bigger than α but less than β , then we know that the true value is less or equal than v and hence we shrink the interval $[\alpha, \beta]$ into the interval $[\alpha, v]$.

$$\mathtt{gCache}[s] = ('\leq', v) \land \alpha < v < \beta \rightarrow \mathtt{evaluate}(s, f, \alpha, \beta) = f(s, \alpha, v).$$

In this case, gCache is updated.

4. If the stored value v is an upper bound and this upper bound is bigger or equal than β , then the stored value isn't of any help.

$$gCache[s] = ('\leq', v) \land \beta \leq v \rightarrow evaluate(s, f, \alpha, \beta) = f(s, \alpha, \beta).$$

In this case, gCache needs to be updated.

5. If the stored value v is a lower bound and this lower bound is greater or equal than β , then we know that the true value is bigger or equal than β and hence we can return the value v:

$$\mathtt{gCache}[s] = ('\geq', v) \land \beta \leq v \rightarrow \mathtt{evaluate}(s, f, \alpha, \beta) = v.$$

6. If the stored value v is a lower bound and this lower bound is less than β but bigger than α , then we know that the true value is bigger or equal than v and hence we shrink the interval $[\alpha, \beta]$ into the interval $[v, \beta]$:

$$\mathtt{gCache}[s] = ('\geq', v) \land \alpha < v < \beta \rightarrow \mathtt{evaluate}(s, f, \alpha, \beta) = f(s, v, \beta).$$

In this case, gCache is updated.

7. If the stored value v is a lower bound and this lower bound is less or equal than α , then the stored value isn't of any help.

$$\mathtt{gCache}[s] = ('\geq', v) \land v \leq \alpha \rightarrow \mathtt{evaluate}(s, f, \alpha, \beta) = f(s, \alpha, \beta).$$

In this case, the gCache needs to be updated.

```
def evaluate(State, f, alpha=-1, beta=1):
         global gCache
2
         if State in gCache:
3
             flag, v = gCache[State]
4
              if flag == '=':
5
                  return v
6
              if flag == '<=':
                  if v <= alpha:
                      return v
                  elif alpha < v < beta:
10
11
                      w = f(State, alpha, v)
                      store_cache(State, alpha, v, w)
12
                      return w
13
                  else:
14
                      w = f(State, alpha, beta)
15
                      store_cache(State, alpha, beta, w)
16
                      return w
17
              if flag == '>=':
18
                  if beta <= v:
19
                      return v
20
                  elif alpha < v < beta:
21
                      w = f(State, v, beta)
22
                      store_cache(State, v, beta, w)
23
                      return w
                  else:
25
                      w = f(State, alpha, beta)
26
                      store_cache(State, alpha, beta, w)
27
                      return w
28
         else: # no value stored in gCache for State
29
              v = f(State, alpha, beta)
30
              store_cache(State, alpha, beta, v)
             return v
32
33
     def store_cache(State, alpha, beta, v):
34
         global gCache
35
              v <= alpha:
36
              gCache[State] = ('<=', v)
37
         elif v < beta: # alpha < v
              gCache[State] = ('=', v)
39
         else: # beta <= v
40
              gCache[State] = ('>=', v)
41
```

Figure 4.7: Implementation of the function evaluate.

```
def maxValue(State, alpha, beta):
       if finished(State):
            return utility(State)
       for ns in next_states(State, gPlayers[0]):
           value = evaluate(ns, minValue, alpha, beta)
           if value >= beta:
                return value
            alpha = max(alpha, value)
       return alpha
   def minValue(State, alpha, beta):
11
       if finished(State):
12
            return utility(State)
13
       for ns in next_states(State, gPlayers[1]):
14
            value = evaluate(ns, maxValue, alpha, beta)
            if value <= alpha:</pre>
                return value
           beta = min(beta, value)
19
       return beta
```

Figure 4.8: Cached implementation of the functions alphaBetaMax and alphaBetaMin.

4.5 Progressive Deepening

In practice, most games are far too complex to be evaluated completely, i.e. the size of the set States is so big that even the fastest computer does not stand a chance to explore this set completely. For example, it is believed² that in chess there are about $4.48 \cdot 10^{44}$ different states that could occur in a game. Hence, it is impossible to explore all possible states in chess. Instead, we have to limit the exploration in a way that is similar to the way professional players evaluate their games: Usually, a player considers all variations of a game for, say, the next three moves. After a given number of moves, the value of a position is estimated using an evaluation heuristic. This function approximates the true value of a given state via a heuristic.

```
def pd_evaluate(State, limit, f=maxValue):
    for l in range(limit+1):
        value = evaluate(State, l, f)
        if value in [-1, 1]:
            return value
    return value
```

Figure 4.9: Progressive Deepening

In order to implement this idea, we add a parameter limit to the procedures alphaBetaMax and alphaBetaMin that were shown in the previous section. On every recursive invocation of the functions alphaBetaMax and alphaBetaMin, the parameter limit is decreased. Once the limit reaches 0, instead of invoking the function alphaBetaMax or alphaBetaMin again, we try to estimate the value of the given State using an evaluation heuristic. This leads to the code shown in Figure 4.10 on page 109.

When we compare this Figure with Figure 4.6 on page 102, the only difference is in line 4 where we test whether the limit is 0. In this case, instead of trying to recursively evaluate the states reachable from State, we evaluate the State with a heuristic function that tries to guess the approximate value of a given state. Notice that in the calls of the function evaluate we have to take care to decrease the parameter limit. The function evaluate is responsible for administering the cache as previously.

There is one further difference between the functions maxValue and minValue shown in Figure 4.10 and those versions of these functions that were shown previously: In Figure 4.10 the NextStates are stored in a priority queue such that the move that is considered to be the best has the highest priority. This way, the best moves are tried first and as a result alpha-beta-pruning is able to prune larger parts of the search space. In order to guess which move is best we use the cached values of the corresponding states. This is the real reason for using progressive deepening: When we evaluate the states with a depth limit of l, we can use the values of the states that has been stored previously when those states were evaluated with a depth limit of l-1.

²For reference, compare the wikipedia article on the so-called Shannon number. The Shannon number estimates that there are at least 10^{120} different plays in chess. However, the number of states is estimated to be about $(4.48\pm0.37)\cdot10^{44}$.

```
def maxValue(State, limit, alpha=-1, beta=1):
         if finished(State):
2
             return utility(State)
         if limit == 0:
             return heuristic(State)
         value
                    = alpha
6
         NextStates = next_states(State, gPlayers[0])
                    = [] # empty priority queue
         for ns in NextStates:
             val = value_cache(ns, limit-2)
             if val == None:
                 val = -1 # unknown values are assumed to be worse than known values
             # heaps are sorted ascendingly, hence the minus
13
             heapq.heappush(Moves, (-val, ns))
14
         while Moves != []:
15
             _, ns = heapq.heappop(Moves)
16
             value = max(value, evaluate(ns, limit-1, minValue, value, beta))
             if value >= beta:
                 return value
19
         return value
20
21
     def minValue(State, limit, alpha=-1, beta=1):
22
         if finished(State):
23
             return utility(State)
         if limit == 0:
25
             return heuristic(State)
                    = beta
         value
         NextStates = next_states(State, gPlayers[1])
28
                    = []
                          # empty priority queue
29
         for ns in NextStates:
30
             val = value_cache(ns, limit-2)
             if val == None:
32
                 val = 1
             heapq.heappush(Moves, (val, ns))
         while Moves != []:
35
             _, ns = heapq.heappop(Moves)
36
             value = min(value, evaluate(ns, limit-1, maxValue, alpha, value))
37
             if value <= alpha:
38
                 return value
39
         return value
40
41
         def value_cache(State, limit):
         flag, value = gCache.get((State, limit), ('?', None))
43
             return value
44
```

Figure 4.10: Depth-limited α - β -pruning.

For a game like tic-tac-toe it is difficult to come up with a decent heuristic. A very crude approach would be to define:

This heuristic would simply estimate the value of all states to be 0. As this heuristic is only called after it has been tested that the game has not yet been decided, this approach is not utterly unreasonable. For a more complex game like chess, the heuristic could instead be a weighted count of all pieces. Concretely, the algorithm for estimating the value of a state would work as follows:

1. Initially, the variable sum is set to 0:

$$sum := 0;$$

2. We would count the number of white rooks Rook_{white} and black rooks Rook_{black}, subtract these numbers from each other and multiply the difference by 5. The resulting number would be added to sum:

$$sum += (Rook_{white} - Rook_{black}) \cdot 5;$$

3. We would count the number of white bishops Bishop_{white} and black bishops Bishop_{black}, subtract these numbers from each other and multiply the difference by 3. The resulting number would be added to sum:

$$sum += (Bishop_{white} - Bishop_{black}) \cdot 3;$$

4. In a similar way we would count knights, queens, and pawns. Approximately, the weights of knights are 3, a queen is worth 9 and a pawn is worth 1.

The resulting sum can then be used as an approximation of the value of a state. More details about the weights of the pieces can be found in the Wikipedia article "chess piece relative value".

Exercise 11: Read up on the game Connect Four. You can play it online at

```
https://connect4.gamesolver.org/en/
```

Your task is to implement this game. On my github page (https://github.com/karlstroetmann) at

```
Artificial-Intelligence/blob/master/Python/3 Games/Connect-Four-Frame.ipynb
```

is a frame that can be used to solve this exercise. Once you have a running implementation of Connect Four, try to improve the strength of your program by adding a non-trivial heuristic to evaluate non-terminal states. As an example of a non-trivial heuristic you can define a triple as a set of three marks of either Xs or Os in a row that is followed by a blank space. The blank space could also be between the marks. Now if there is a state s that has s triples of Ss and s triples of Ss and the game is not finished, then define

$$\mathtt{value}(s, \mathtt{X}, \mathtt{limit}, \alpha, \beta) = \frac{a - b}{10} \quad \text{if } \mathtt{limit} = 0.$$

Chapter 5

Equational Theorem Proving

Mathematics, and in particular mathematical theorem proving is clearly related to the concept of intelligence. Automatic theorem proving is the branch of artificial intelligence that investigates the use of artificial intelligence in mathematics. The topic of automatic theorem proving can easily fill more than one book. For reasons of time we will restrict this chapter to deal with equational theorem proving. In equational theorem proving we are given a set of axioms that are equations and we ask, which other equations can be derived from these axioms. For example, a group $\mathcal G$ is defined as a triple of the form

$$\mathcal{G} = \langle G, e, \cdot, i \rangle$$

such that

- G is a set. The elements of this set are called group elements.
- \bullet $e \in G$,

i.e. e is an element of G. For reasons that will soon become obvious, e is called the left-neutral element.

• $\cdot: G \times G \to G$,

i.e. \cdot is a function mapping pairs of group elements to group elements. The function \cdot is called the multiplication of the group \mathcal{G} .

• $i: G \to G$,

i.e. i is a function mapping group elements to group elements. For any group element $x \in G$ the element i(x) is called the <u>left-inverse</u> of x.

- Furthermore, the following group axioms have to be satisfied:
 - 1. $e \cdot x = x$,
 - $2. \ i(x) \cdot x = e,$
 - 3. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.

In abstract algebra, it is shown that these axioms imply the equation

 $x \cdot i(x) = e$, i.e. the left inverse of any group element x is also a right inverse of x.

A possible proof runs as follows:

$$x \cdot i(x) = e \cdot (x \cdot i(x))$$
 because e is left-neutral
$$= (i(x \cdot i(x)) \cdot (x \cdot i(x))) \cdot (x \cdot i(x))$$
 because $i(x \cdot i(x)) \cdot (x \cdot i(x)) = e$
$$= i(x \cdot i(x)) \cdot ((x \cdot i(x)) \cdot (x \cdot i(x)))$$
 associativity
$$= i(x \cdot i(x)) \cdot (x \cdot (i(x) \cdot (x \cdot i(x))))$$
 associativity
$$= i(x \cdot i(x)) \cdot (x \cdot ((i(x) \cdot x) \cdot i(x)))$$
 associativity
$$= i(x \cdot i(x)) \cdot (x \cdot (e \cdot i(x)))$$
 because $i(x) \cdot x = e$
$$= i(x \cdot i(x)) \cdot (x \cdot i(x))$$
 because $e \cdot i(x) = i(x)$ because $e \cdot i(x) = i(x)$

Obviously, it is not trivial to come up with proofs of this kind. Fortunately, there is a systematic approach to solve this and similar equational problems: In this chapter, we present an algorithm that can automatically find equational proofs of the kind shown above. This algorithm is known as the Knuth-Bendix completion algorithm. It was discovered by Donald E. Knuth and Peter B. Bendix [KB70].

The rest of this chapter is structured as follows.

- 1. In the next section, we will formally introduce equational proofs and term rewriting.
- 2. After that, we discuss abstract properties of relations. For example, we introduce the notion of confluence and prove the Church-Rosser theorem and Newman's lemma.
- 3. The third section discusses term orderings and introduces the Knuth-Bendix ordering.
- 4. The last section presents the Knuth-Bendix completion algorithm

5.1 Equational Proofs

This section defines the notion of an equational proof precisely and discusses how equational proofs can be carried out via term rewriting. In order to do this, we have to define a number of more elementary notions like functions symbols, variables, terms, substitutions, et cetera. We begin with the notion of a signature.

Definition 7 (Signature) A signature is a triple of the form

$$\Sigma = \langle \mathcal{V}, \mathcal{F}, arity \rangle$$
,

where we have the following:

- 1. \mathcal{V} is the set of variables.
- 2. \mathcal{F} is the set of function symbols.
- 3. arity is a function such that

arity :
$$\mathcal{F} \to \mathbb{N}$$
.

If we have arity(f) = n, then f is said to be an n-ary function symbol.

4. We have $\mathcal{V} \cap \mathcal{F} = \{\}$, i.e. variables are different from function symbols.

 \Diamond

Example: The signature of group theory Σ_G can be defined as follows:

- (a) $V := \{w, x, y, z\},\$
- (b) $\mathcal{F} := \{e, i, \cdot\},\$
- (c) arity := $\{e \mapsto 0, i \mapsto 1, \cdot \mapsto 2\},\$

i.e. e is a constant symbol, i is a unary function symbol, and \cdot is a binary function symbol.

(d)
$$\Sigma = \langle \mathcal{V}, \mathcal{F}, arity \rangle$$
.

Having defined the notion of a signature we proceed to define terms.

Definition 8 (Term, \mathcal{T}_{Σ}) If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \textit{arity} \rangle$ is a signature, the set of Σ -terms \mathcal{T}_{Σ} is defined inductively:

- 1. For every variable $x \in \mathcal{V}$ we have $x \in \mathcal{T}_{\Sigma}$.
- 2. If $f \in \mathcal{F}$ and arity(f) = 0, then $f \in \mathcal{T}_{\Sigma}$.
- 3. If $f \in \mathcal{F}$ and $n := \mathit{arity}(f) > 0$ and, furthermore, $t_1, \cdots, t_n \in \mathcal{T}_{\Sigma}$, then we have $f(t_1, \cdots, t_n) \in \mathcal{T}_{\Sigma}$.

Example: Given the signature Σ_G defined above, we have the following:

- 1. $x \in \mathcal{T}_{\Sigma_G}$, because every variable is a Σ_G -term.
- 2. $e \in \mathcal{T}_{\Sigma_G}$.
- 3. $\cdot (e, x) \in \mathcal{T}_{\Sigma_C}$.
- 4. $\cdot (\cdot(x,y),z) \in \mathcal{T}_{\Sigma_G}$.

Remark: Later on we will often use an infix notation for binary function symbols. In general, if f is a binary function symbol, then the term $f(t_1, t_2)$ is written as $t_1 f t_2$. If this notation would result in an ambiguity because either t_1 or t_2 is also written in infix notation, then we use parenthesis to resolve the ambiguity. For example, we will write

$$(x \cdot y) \cdot z$$
 instead of $\cdot (\cdot (x, y), z) \in \mathcal{T}_{\Sigma_G}$.

Note that we cannot write the term $\cdot (\cdot (x, y), z)$ as $x \cdot y \cdot z$ because that notation is ambiguous, since it can be interpreted as either $(x \cdot y) \cdot z$ or $x \cdot (y \cdot z)$.

Definition 9 (Σ-**Equation**) Assume a signature Σ is given. A Σ-equation is a pair $\langle s, t \rangle$ such that both s and t are Σ-terms. The Σ-equation $\langle s, t \rangle$ is written as

$$s \approx t$$
.

Remark: We use the notation $s \approx t$ instead of the notation s = t in order to distinguish between the notion of a Σ -equation and the notion of equality of terms. So when s and t are Σ -terms and we write s = t we do mean that s and t are literally the same terms, while writing $s \approx t$ means that we are interested in the logical consequences that would follow from the assumption that the interpretation of s and t are the same in certain Σ -algebras. The notion of a Σ -algebra is defined next.

 \Diamond

 \Diamond

Definition 10 (Σ -Algebra) Assume a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \textit{arity} \rangle$ is given. A Σ -algebra¹ is a pair of the form $\mathfrak{A} = \langle A, \mathcal{J} \rangle$ where:

- 1. A is a nonempty set that is called the universe of the Σ -algebra \mathfrak{A} .
- 2. $\mathcal J$ is the interpretation of the function symbols. Technically, $\mathcal J$ is a function that is defined on the set $\mathcal F$ of all function symbols. For every function symbol $f \in \mathcal F$ we have that

$$\mathcal{J}(f): A^{\mathsf{arity}(f)} \to A$$
,

i.e. $\mathcal{J}(f)$ is a function from A^n to A where n is the arity of the function symbol f.

If $\mathfrak{A}=\langle A,\mathcal{J}\rangle$ is a Σ -algebra, then the function $\mathcal{J}(f)$ is usually written more concisely as $f^{\mathfrak{A}}$. \diamond

Example: In this example we construct a Σ_G -algebra where Σ_G is the signature of group theory defined earlier. We define $G := \{0, 1\}$ and define the interpretations $\mathcal{J}(f)$ for $f \in \{e, i, \cdot\}$ as follows:

- 1. $\mathcal{J}(e) := 0$.
- 2. $\mathcal{J}(i) := \{0 \mapsto 0, 1 \mapsto 1\}.$
- 3. $\mathcal{J}(\cdot) = \{ \langle 0, 0 \rangle \mapsto 0, \langle 0, 1 \rangle \mapsto 1, \langle 1, 0 \rangle \mapsto 1, \langle 1, 1 \rangle \mapsto 0 \}.$

Then $\mathfrak{G} = \langle G, \mathcal{J} \rangle$ is a Σ_G -algebra.

Definition 11 (Valid Equation) If $\Sigma = \langle \mathcal{V}, \mathcal{F}, \textit{arity} \rangle$ is a signature, $\mathfrak{A} = \langle A, \mathcal{J} \rangle$ is a Σ -algebra, and I is a variable assignment, i.e. I is a mapping of the form $I : \mathcal{V} \to A$, then we can evaluate Σ -terms in \mathfrak{A} as follows:

- 1. eval(x, I) := I(x) for all $x \in \mathcal{V}$.
- 2. $\operatorname{eval}(c, I) := c^{\mathfrak{A}}$ for every constant symbol $c \in \mathcal{F}$.
- 3. $\operatorname{eval}(f(t_1,\dots,t_n),I) := f^{\mathfrak{A}}(\operatorname{eval}(t_1,I),\dots,\operatorname{eval}(t_n,I)).$

A Σ -equation $s \approx t$ is valid in the Σ -algebra $\mathfrak A$ iff we have

$$eval(s, I) = eval(t, I)$$
 for all variable assignments $I: \mathcal{V} \to A$.

This is written as

$$\mathfrak{A} \models s \approx t$$

and we say that $\mathfrak A$ satisfies the equation $s \approx t$.

Example: Continuing the previous example we have the following:

- 1. $\mathfrak{G} \models e \cdot x \approx x$,
- 2. $\mathfrak{G} \models i(x) \cdot x \approx e$,

3.
$$\mathfrak{G} \models (x \cdot y) \cdot z \approx x \cdot (y \cdot z)$$
.

¹The notion of a Σ -algebra is a notion that is used both in logic and in universal algebra. In universal algebra, a Σ -algebra is also known as an algebraic structure. This notion is not related to and should not be confused with the notion of a σ -algebra, which is a notion used in the field of measure theory. Note that the notion used in measure theory is always written with a lower case σ , while the notion used in logic is written with a capital Σ .

Definition 12 (E-Variety) Assume that Σ is a signature and E is a set of Σ -equations. The collection of all Σ -structures that satisfy every equation from E is called the E-variety. To put it differently, the Σ -structure $\mathfrak A$ is a member of the E-variety iff

$$\mathfrak{A} \models s \approx t$$
 for every equation $s \approx t$ in E .

Example: Define $E := \{e \cdot x = x, i(x) \cdot x = e, (x \cdot y) \cdot z = x \cdot (y \cdot z)\}$. This set of equations defines the variety of groups. You can check that the Σ_G -algebra \mathfrak{G} is a member of this variety and hence it is a group.

Given a set of Σ -equations E it is natural to ask which other equations are logical consequences of the equations in E. This notion is defined below.

Definition 13 (Logical Consequence) Assume a signature Σ and a set E of Σ -equations to be given. Then the equation $s \approx t$ is a logical consequence of E iff we have

 $\mathfrak{A} \models s \approx t$ for every Σ -algebra \mathfrak{A} that is a member of the E-variety.

If $s \approx t$ is a logical consequence of the set of equations E, then this is written as

$$E \models s \approx t$$
.

Therefore we have $E \models s \approx t$ if and only if every Σ -algebra that satisfies all equations from E also satisfies the equation $s \approx t$.

Example: If we define
$$E := \{e \cdot x = x, i(x) \cdot x = e, (x \cdot y) \cdot z = x \cdot (y \cdot z)\}$$
, then we have $E \models x \cdot i(x) \approx e$.

The notion $E \models s \approx t$ is a semantic notion. We cannot hope to implement this notion directly because if a set of equations E and a possible logical consequence $s \approx t$ is given, there are, in general, infinitely many Σ -algebras that have to be checked. Fortunately, the notion $E \models s \approx t$ has a syntactical analog $E \vdash s \approx t$ (read: E proves $s \approx t$) that can be implemented and that is at least semi-decidable, i.e. we can create a program that given a set of equations E and an equation $s \approx t$ will return True if $E \vdash s \approx t$ holds, and will either return False or run forever if $E \vdash s \approx t$ does not hold. Even more fortunately, Gödels completeness theorem implies that the syntactical notion coincides with the semantic notion, i.e. we have

$$E \models s \approx t$$
 if and only if $E \vdash s \approx t$.

5.1.1 A Calculus for Equality

In this subsection we assume a signature Σ and a set of Σ -equations E to be given. Our goal is to define the provability notion $E \vdash s \approx t$, which is read as E proves $s \approx t$. However, in order to do this we first need to define the notion of a substitution.

Definition 14 (Σ-Substitution) Assume that a signature $\Sigma = \langle \mathcal{V}, \mathcal{F}, \textit{arity} \rangle$ is given. A Σ-substitution σ is a map of the form

$$\sigma: \mathcal{V} \to \mathcal{T}_{\Sigma}$$

such that the set $dom(\sigma) := \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is finite. If we have $dom(\sigma) = \{x_1, \dots, x_n\}$ and $t_i = \sigma(x_i)$ for all $i = 1, \dots, n$, then we use the following notation:

$$\sigma = \{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}.$$

A substitution $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ can be applied to a term t by replacing the variables x_i with the terms t_i . We will use the postfix notation $t\sigma$ to denote the application of the substitution σ to the term t. Formally, the notation $t\sigma$ is defined by induction on t:

- 1. $x\sigma := \sigma(x)$ for all $x \in \mathcal{V}$.
- 2. $c\sigma = c$ for every constant $c \in \mathcal{F}$.
- 3. $f(t_1, \dots, t_n)\sigma := f(t_1\sigma, \dots, t_n\sigma)$.

Now we are ready to formally define the notion $E \vdash s \approx t$. This notion is defined inductively.

- 1. $E \vdash s \approx t$ for every Σ -equations $(s \approx t) \in E$. (Axioms)
- 2. $E \vdash s \approx s$ for every Σ -term s. (Reflexivity)
- 3. If $E \vdash s \approx t$, then $E \vdash t \approx s$. (Symmetry)
- 4. If $E \vdash r \approx s$ and $E \vdash s \approx t$, then $E \vdash r \approx t$. (Transitivity)
- 5. If $\operatorname{arity}(f) = n$ and $E \vdash s_i \approx t_i$ for all $i \in \{1, \dots, n\}$, then $E \vdash f(s_1, \dots, s_n) \approx f(t_1, \dots, t_n)$. (Congruence)
- 6. If $E \vdash s \approx t$ and σ is a Σ -substitution, then $E \vdash s\sigma \approx t\sigma$. (Stability)

The definition of $E \vdash s \approx t$ that has been given above is due to Gottfried Wilhelm Leibniz.

5.1.2 Equational Proofs

It turns out that, although it is possible to implement the notion $E \models s \approx t$ directly, it is more efficient to refine this notion a little bit. To this end we need to introduce the notion of a position u in a term t, the notion of the subterm of a given term t at a given position, and the notion of replacing a subterm at a given position by another term. We define these notions next.

Definition 15 (Positions of a Term t, Pos(t))

Given a term t the set of all positions of t is written as $\mathcal{P}os(t)$ and is defined by induction on t:

- 1. $\mathcal{P}os(x) := \{[\]\}$ for every variable x.
- 2. $\mathcal{P}os(c) := \{[]\}$ for every constant c.
- 3. $\operatorname{\mathcal{P}os}(f(t_1,\cdots,t_n)):=\{[\,]\}\cup\bigcup_{i=1}^n\{[i]+u\mid u\in\operatorname{\mathcal{P}os}(t_i)\}$ for every term $f(t_1,\cdots,t_n)$.

Definition 16 (Subterm at a given Position, t/u)

Given a term t and a position $u \in \mathcal{P}os(t)$, the subterm of t at position u is written as t/u. This expression is defined by induction on u.

1.
$$t/[] := t$$
,

2.
$$f(t_1, \dots, t_n)/([i] + u) := t_i/u$$
.

Definition 17 (Subterm Replacement, $t[u \mapsto s]$)

Given a term t, a position $u \in \mathcal{P}os(t)$, and a term s, the expression $t[u \mapsto s]$ denotes the term that results from t when the subterm t/u is replaced by the term s. This expression is defined by induction on u.

1.
$$t[[] \mapsto s] := s$$
,

2.
$$f(t_1, \dots, t_n)[[i] + u \mapsto s] := f(t_1, \dots, t_{i-1}, t_i[u \mapsto s], t_{i+1}, \dots, t_n).$$

Example: Define $t := (x \cdot y) \cdot z$. Then we have

$$\mathcal{P}os((x \cdot y) \cdot z) = \{[], [1], [1, 1], [1, 2], [2]\}.$$

Furthermore, we have the following:

1.
$$((x \cdot y) \cdot z)/[] = (x \cdot y) \cdot z$$
,

2.
$$((x \cdot y) \cdot z)/[1] = x \cdot y$$
, and

3.
$$((x \cdot y) \cdot z)/[1,2] = y$$
.

We also have

$$((x \cdot y) \cdot z)[1] \mapsto y \cdot x = (y \cdot x) \cdot z.$$

Definition 18 (\leftrightarrow_E) Given a set of Σ -equations E and two Σ -terms s and t we define that

$$s \leftrightarrow_E t$$

holds if and only if the following conditions are satisfied:

- (a) There exists an equation $l \approx r$ such that either $(l \approx r) \in E$ or $(r \approx l) \in E$.
- (b) There is a position $u \in \mathcal{P}os(s)$ and a substitution σ such that $s/u = l\sigma$.

(c)
$$t = s[u \mapsto r\sigma]$$
.

To put this is words: We have $s \leftrightarrow_E t$ iff there is an equation $l \approx r$ such that either the equation $l \approx r$ or the equation $r \approx l$ is an element of the set of equations E and, furthermore, s contains the subterm $l\sigma$ and t results from s by replacing the subterm $l\sigma$ with the subterm $r\sigma$.

Example: If we have $E = \{i(x) \cdot x \approx e\}$ then

$$(i(a) \cdot a) \cdot b \leftrightarrow_E e \cdot b$$

because the right hand side $e \cdot b$ results from the left hand side $(i(a) \cdot a) \cdot b$ by replacing the subterm $i(a) \cdot a$ that occurs at position [1] by the term e. This is possible because the equation $i(x) \cdot x \approx e$ tells us that $i(a) \cdot a$ is equal to e.

Next, we define the relation \leftrightarrow_E^* as the reflexive and transitive closure of the relation \leftrightarrow_E .

Definition 19 (\leftrightarrow_E^*) For Σ -terms s and t the notion $s \leftrightarrow_E^* t$ is defined inductively as follows:

- 1. We have $s \leftrightarrow_E^* s$ for every Σ -term s.
- 2. If $s \leftrightarrow_E t$, then $s \leftrightarrow_E^* t$.
- 3. If u is a Σ -term such that both $s \leftrightarrow_E u$ and $u \leftrightarrow_E^* t$ holds, then we have $s \leftrightarrow_E^* t$.

Given this definition it is now possible to show the following:

$$E \vdash s \approx t$$
 if and only if $s \leftrightarrow_E^* t$.

For each of the two directions that has to be proven, the proof can be done by a straightforward but lengthy induction.

5.1.3 Proofs via Rewriting

Implementing the relation \leftrightarrow_E is rather inefficient because every equation from E can be used in two different ways: If E contains the equation $l \approx r$ we can use this equation from left to right to replace a subterm of the form $l\sigma$ by the term $r\sigma$, or we can use this equation from right to left by replacing a subterm of the form $r\sigma$ by $l\sigma$. Additionally, unless we have derived the equation we want to prove, we don't know when to stop our proof efforts. It was the brilliant idea of Donald E. Knuth and Peter B. Bendix [KB70] who realized that, provided the equations can be ordered in a way such that the left hand side is always more complex than the right hand side, then it is possible to use these equations only in one direction, if certain additional equations are added to E in this process. To proceed, we need the following definition.

Definition 20 (Rewrite Order)

A binary relation $\prec \subseteq \mathcal{T}_{\Sigma} \times \mathcal{T}_{\Sigma}$ is a rewrite order if and only if we have the following:

- 1. \prec is a strict partial order on \mathcal{T}_{Σ} , i.e. we have
 - (a) $\neg (s \prec s)$ for all $s \in \mathcal{T}_{\Sigma}$, i.e. \prec is irreflexive.
 - (b) $r \prec s \land s \prec t \Rightarrow r \prec t$ for all $r, s, t \in \mathcal{T}_{\Sigma}$, i.e. \prec is transitive.
- 2. \prec is stable under substitutions, i.e we have

$$r \prec l \Rightarrow r\sigma \prec l\sigma$$
 for every substitution σ .

3. \prec is a congruence, i.e. we have

$$r \prec l \implies s[u \mapsto r] \prec s[u \mapsto l]$$
 for every Σ -term s and every $u \in \mathcal{P}os(s)$.

4. The relation \prec is well-founded, i.e. there is no infinite sequence of the form $(s_n)_{n\in\mathbb{N}}$ such that we have

$$s_{n+1} \prec s_n$$
 for all $n \in \mathbb{N}$.

If E is a set of equations, then a binary relation $\prec \subseteq \mathcal{T}_{\Sigma} \times \mathcal{T}_{\Sigma}$ is a rewrite order w.r.t. E if, in addition to being a rewrite order, it satisfies

```
r \prec l for every equation (l \approx r) \in E.
```

This means that all equations in E are ordered such that the right hand side is smaller than the left hand side w.r.t. \prec .

Later we will see an example of a rewrite order. For now, we just assume that the relation \prec is a rewrite order w.r.t. a given set of equations R. The equations in R will now be called rewrite rules. We proceed to define the relation \rightarrow_R on \mathcal{T}_{Σ} .

Definition 21 (Rewrite Relation \rightarrow_R)

Given a set of rewrite rules R and two Σ -terms s and t we define that

$$s \rightarrow_R t$$
 (read: s rewrites to t)

if and only if there exists a rewrite rule $(l \approx r) \in R$ such that the following conditions are satisfied:

(a) There is a position $u \in \mathcal{P}os(s)$ and a substitution σ such that $s/u = l\sigma$.

(b)
$$t = s[u \mapsto r\sigma].$$

Similar to the definition of \leftrightarrow_E^* we now define \to_R^* as the reflexive and transitive closure of \to_R .

Definition 22 (\rightarrow_R^*) For Σ -terms s and t the notion $s \rightarrow_R^* t$ is defined inductively as follows:

- 1. We have $s \to_R^* s$ for every Σ -term s.
- 2. If $s \to_R t$, then $s \to_R^* t$.
- 3. If u is a Σ -term such that both $s \to_R u$ and $u \to_R^* t$ holds, then we have $s \to_R^* t$.

Definition 23 (Normal Form) A Σ -term s is in normal form if there is no Σ -term t such that $s \to_R t$, i.e. the term s cannot be simplified anymore by rewriting.

The basic idea of a rewrite proof of an equation $s \approx t$ is now the following:

1. We rewrite s using the rewrite rules from R into a term \hat{s} that is in normal form:

$$s \to_R s_1 \to_R s_2 \to_R \cdots \to_R s_m = \widehat{s}$$

2. Similarly, we rewrite t using the rewrite rules from R into a term \hat{t} that is in normal form:

$$t \to_R t_1 \to_R t_2 \to_R \cdots \to_R t_n = \hat{t}.$$

3. If the relation $s \to t$ is confluent (this notion is defined in the next section), then we have

$$s \leftrightarrow_E t \quad \Leftrightarrow \quad \widehat{s} = \widehat{t}.$$

This way, we can reduce the question whether $s \leftrightarrow_E^* t$ holds to the computation of normal forms. The latter can often be done quite efficiently. The rest of this chapter proceeds as follows:

- (a) In the next section we discuss the notion of confluence and prove a theorem that can be used to show that a relation is confluent.
- (b) After that we discuss rewrite orderings in more detail. In particular, we present the Knuth-Bendix order, which is the rewrite order that we use later to construct a number of confluent term rewriting systems.
- (c) Then, we present the Knuth-Bendix completion algorithm that can enrich a set of equation so that the rewrite relation \rightarrow_R becomes confluent.
- (d) Finally, we present an implementation of the Knuth-Bendix completion algorithm.

5.2 Confluence

In this section we assume that a binary relation \to is given on a set M, i.e. we have $\to \subseteq M \times M$. Instead of writing $(a,b) \in \to$ we use infix notation and write $a \to b$. Furthermore, we assume that \to is well-founded, i.e. there is no infinite sequence $(x_n)_{n \in \mathbb{N}}$ such that

$$s_n \to s_{n+1}$$
 holds for all $n \in \mathbb{N}$.

We denote the equivalence relation generated by \rightarrow as $\stackrel{*}{\leftrightarrow}$ and the reflexive and transitive closure of \rightarrow is written as $\stackrel{*}{\rightarrow}$.

Definition 24 (Confluence) The relation $\rightarrow \subseteq M \times M$ is confluent iff the following holds:

$$\forall a,b,c \in M: \left(a \stackrel{*}{\rightarrow} b \ \land \ a \stackrel{*}{\rightarrow} c \quad \Rightarrow \quad \exists d \in M: \left(b \stackrel{*}{\rightarrow} d \ \land \ c \stackrel{*}{\rightarrow} d\right)\right) \qquad \diamond$$

The next theorem shows that confluence is all we need to reduce the relation $\stackrel{*}{\leftrightarrow}$ to the relation $\stackrel{*}{\rightarrow}$.

Theorem 25 (Church-Rosser) If the relation $\rightarrow \subseteq M \times M$ is confluent, then we have

$$\forall a, b \in M : (a \stackrel{*}{\leftrightarrow} b \iff \exists c \in M : (a \stackrel{*}{\rightarrow} c \land b \stackrel{*}{\rightarrow} c)).$$

Proof: If $a \stackrel{*}{\leftrightarrow} b$ then there is a finite sequence $(s_k)_{k \in \{0,\dots,n\}}$ such that

$$a = s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_{n-1} \leftrightarrow s_n = b.$$

We prove by induction on n that there is an element $c \in M$ such that both $a \stackrel{*}{\to} c$ and $b \stackrel{*}{\to} c$ holds.

Base Case: n = 0.

Then we have a = b and we can define c := a.

Induction Step: $n \mapsto n+1$

We have $a = s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_n \leftrightarrow s_{n+1} = b$. By induction hypotheses we know that there exists a $d \in M$ such that

$$a \stackrel{*}{\to} d$$
 and $s_n \stackrel{*}{\to} d$

hold. Furthermore, we either have

$$s_n \to b$$
 or $b \to s_n$.

We discuss these cases one by one.

1. Case: $s_n \to b$.

Since we also have $s_n \stackrel{*}{\to} d$, the confluence of the relation \to shows that there is an element $c \in M$ such

$$b \stackrel{*}{\to} c$$
 and $d \stackrel{*}{\to} c$

holds. From $a \stackrel{*}{\to} d$ and $d \stackrel{*}{\to} c$ we have that $a \stackrel{*}{\to} c$. Since we already know that $b \stackrel{*}{\to} c$, the proof is complete in this case.

2. Case: $b \to s_n$.

Since we have $b \to s_n$ and $s_n \stackrel{*}{\to} d$, we can conclude $b \stackrel{*}{\to} d$. Since we also have $a \stackrel{*}{\to} d$, the proof is complete if we define c := d.

In general, it is hard to prove that a relation \rightarrow is confluent. Things get easier if the relation \rightarrow is well-founded, since then there is a weaker notion than confluence that is already sufficient to guarantee confluence.

Definition 26 (Local Confluence)

The relation $\rightarrow \subseteq M \times M$ is locally confluent iff the following holds:

$$\forall a, b, c \in M : (a \to b \land a \to c \quad \Rightarrow \quad \exists d \in M : (b \stackrel{*}{\to} d \land c \stackrel{*}{\to} d)) \qquad \diamond$$

Theorem 27 (Transfinite Induction)

Assume the relation $\rightarrow \subseteq M \times M$ is well-founded and F(x) is some formula. If we have that

$$\forall b \in M : \left(a \xrightarrow{+} b \Rightarrow F(b)\right) \Rightarrow F(a) \text{ holds for all } a \in M, \tag{TI}$$

then we can conclude that $\forall a \in M : F(a)$ holds.

Proof: Above, $\stackrel{+}{\rightarrow}$ denotes the transitive closure of \rightarrow . We call b a successor of a if $a \stackrel{+}{\rightarrow} b$ holds. The proof principle of transfinite induction is correct for a well-founded relation because, first, if a has no successors b, then the premise of (TI) which is

$$\forall b \in M : \left(a \stackrel{+}{\to} b \Rightarrow F(b)\right)$$

is vacuously true and hence by (TI) we know that F(a) has to be true for all $a \in M$ that have no successors. Now assume F(a) were false for some $a \in M$. Then there must be a successor a_1 of a such that $F(a_1)$ is false because otherwise F(a) would be true. But then there must be a successor a_2 of a_1 such that $F(a_2)$ is false. Proceeding in this way we can construct an infinite sequence

$$a \xrightarrow{+} a_1 \xrightarrow{+} a_2 \xrightarrow{+} \cdots \xrightarrow{+} a_n \xrightarrow{+} a_{n+1} \rightarrow \cdots$$

such that $F(a_n)$ is false. But this would contradict the well-foundedness of the relation \to . Hence there can be no $a \in M$ such that F(a) is false.

Theorem 28 (Newman's Lemma)

If the relation $\rightarrow \subseteq M \times M$ is well-founded and locally confluent, then it is already confluent.

Proof: Given any $a \in M$, we define the following formula:

$$F(a) := \forall b, c \in M : \left(a \stackrel{*}{\to} b \land a \stackrel{*}{\to} c \quad \Rightarrow \quad \exists d \in M : \left(b \stackrel{*}{\to} d \land c \stackrel{*}{\to} d\right)\right)$$

We prove that F(a) holds for all $a \in M$ by transfinite induction. Therefore, in order to prove F(a) we may assume that F(b) already holds for all successors b of a. So let us assume that we have

$$a \stackrel{*}{\to} b$$
 and $a \stackrel{*}{\to} c$.

We have to find an element $d \in M$ such that both $b \stackrel{*}{\to} d$ and $c \stackrel{*}{\to} d$ holds. Now since $a \stackrel{*}{\to} b$, either a = b or there is an element b_1 such that

$$a \to b_1 \stackrel{*}{\to} b$$

holds. If a = b we can define d := c and because of $a \stackrel{*}{\to} c$ we would then have both

$$b \stackrel{*}{\to} d$$
 and $c \stackrel{*}{\to} d$

and therefore, in the case a = b, we are done. Similarly, since $a \stackrel{*}{\to} c$ we either have a = c or there is an element c_1 such that

$$a \rightarrow c_1 \stackrel{*}{\rightarrow} c$$

holds. If a = c we can define d := b and because of $a \stackrel{*}{\to} b$ we would then have both

$$b \stackrel{*}{\to} d$$
 and $c \stackrel{*}{\to} d$

and are done again. Now the case that is left is the following:

$$a \to b_1 \stackrel{*}{\to} b$$
 and $a \to c_1 \stackrel{*}{\to} c$.

Since \rightarrow is locally confluent and we have both $a \rightarrow b_1$ and $a \rightarrow c_1$ there exists an element d_1 such that we have

$$b_1 \stackrel{*}{\to} d_1$$
 and $c_1 \stackrel{*}{\to} d_1$.

Now as b_1 is a successor of a and we have both

$$b_1 \stackrel{*}{\to} b$$
 and $b_1 \stackrel{*}{\to} d_1$,

our induction hypotheses tells us that there is an element d_2 such that we have both

$$b \stackrel{*}{\to} d_2$$
 and $d_1 \stackrel{*}{\to} d_2$.

Now we have $c_1 \stackrel{*}{\to} d_1$ and $d_1 \stackrel{*}{\to} d_2$, which implies

$$c_1 \stackrel{*}{\rightarrow} d_2$$

As we also have $c_1 \stackrel{*}{\to} c$ we have both

$$c_1 \stackrel{*}{\to} d_2$$
 and $c_1 \stackrel{*}{\to} c$.

Since c_1 is a successor of a, the induction hypotheses tells us that there is an element d such that we have both

$$d_2 \stackrel{*}{\to} d$$
 and $c \stackrel{*}{\to} d$.

As we have $b \stackrel{*}{\to} d_2$ and $d_2 \stackrel{*}{\to} d$ we can conclude $b \stackrel{*}{\to} d$. Hence we have

$$b \stackrel{*}{\to} d$$
 and $c \stackrel{*}{\to} d$

and the proof is complete. Figure 5.1 on page 122 shows how the different elements are related and conveys the idea of the proof in a concise way. \Box



Figure 5.1: The Proof of Newman's Lemma.

5.3 The Knuth-Bendix Order

In this section we define the Knuth-Bendix order \prec on the set \mathcal{T}_{Σ} of Σ -terms. In order to do so, three prerequisites need to be satisfied:

- 1. We need to assign a weight w(f) to every function symbol f. These weights are natural numbers. In addition, there must be at most one function symbol g such that w(g) = 0. Furthermore, if w(g) = 0, then g has to be unary.
- 2. We need to have a strict total order < on the set of function symbols, i.e. the following conditions need to be satisfied:
 - (a) The relation < is irreflexive, that is we have $\neg(f < f)$ for all function symbols f.
 - (b) The relation < is transitive, that is we have

$$f < g \wedge g < h \Rightarrow f < h \quad \text{ for all function symbols } f, \, g, \, \text{and } h.$$

(c) The relation < is total, that is we have

$$f < g \lor f = g \lor g < f$$
 for all function symbols f and g .

3. The order < on the function symbols has to be admissible with respect to the weight function w, i.e. the following condition needs to be satisfied:

$$w(f) = 0 \rightarrow \forall g : (g \neq f \rightarrow g < f).$$

To put this in words: If the function symbol f has a weight of 0, then all other function symbols g have to be smaller than f w.r.t. the strict order <. Note that this implies that there can be at most one function symbol with f such that w(f) = 0. This function symbol f is then the maximum w.r.t. the order <.

Given the function w that assigns a weight to all function symbols, we can define the weight w(t) of a Σ -term t by induction on t.

1. w(x) := 1 for all variables x.

2.
$$w(f(t_1, \dots, t_n)) := w(f) + \sum_{i=1}^{n} w(t_i)$$
.

Furthermore, we define the function

$$\mathtt{count}: \mathcal{T}_\Sigma imes \mathcal{V} o \mathbb{N}$$

that takes a term t and a variable x and returns the number of times that x occurs in t. We define count(t,x) by induction on t.

- 1. count(x, x) := 1 for every variable $x \in \mathcal{V}$.
- 2. count(y, x) := 0 if $x \neq y$ for all variables $x, y \in \mathcal{V}$.
- 3. $\operatorname{count}(f(t_1,\dots,t_n),x) := \sum_{i=1}^n \operatorname{count}(t_i,x).$

Now we are ready to define the Knuth-Bendix order. Given two terms s and t we have $s \prec t$ iff one of the following two conditions hold:

1. w(s) < w(t) and $count(s, x) \le count(t, x)$ for all variables x occurring in s.

- 2. w(s) = w(t), $count(s, x) \le count(t, x)$ for all variables x occurring in s, and one of the following subconditions holds:
 - (a) $t = f^n(s)$ where $n \ge 1$ and f is the maximum w.r.t. the order < on function symbols, i.e. we have g < f for all function symbols $g \ne f$.
 - (b) $s = f(s_1, \dots, s_m), t = g(t_1, \dots, t_n), \text{ and } f < g.$
 - (c) $s = f(s_1, \dots, s_m), t = f(t_1, \dots, t_m), \text{ and } [s_1, \dots, s_m] \prec_{\text{lex}} [t_1, \dots, t_m].$

Here, \prec_{lex} denotes the lexicographic extension of the ordering \prec to lists of terms. It is defined as follows:

$$[x] + R_1 \prec_{\text{lex}} [y] + R_2 \iff x \prec y \lor (x = y \land R_1 \prec_{\text{lex}} R_2)$$

Theorem 29 The Knuth-Bendix order is a rewrite order.

Proving that the Knuth-Bendix order is a strict partial order on the set \mathcal{T}_{Σ} that is stable and a congruence can be done via induction on the structure of the terms. The hard part of the proof is to show that the Knuth-Bendix order is well-founded. A proof is given in the book by Franz Baader and Tobias Nipkow [BN98].

5.4 Unification

This section introduces the notion of a most general unifier of two terms. To begin, we define the composition of two Σ -substitutions.

Definition 30 (Composition of Substitutions) Assume that

$$\sigma = \{x_1 \mapsto s_1, \dots, x_m \mapsto s_m\}$$
 and $\tau = \{y_1 \mapsto t_1, \dots, y_n \mapsto t_n\}$

are two substitutions such that $dom(\sigma) \cap dom(\tau) = \{\}$. We define the composition $\sigma\tau$ of σ and τ as

$$\sigma\tau := \{x_1 \mapsto s_1\tau, \cdots, x_m \mapsto s_m\tau, \ y_1 \mapsto t_1, \cdots, y_n \mapsto t_n\}$$

Example: If we define

$$\sigma := \{x_1 \mapsto c, x_2 \mapsto f(x_3)\}$$
 and $\tau := \{x_3 \mapsto h(c,c), x_4 \mapsto d\},$

then we have

$$\sigma\tau = \{x_1 \mapsto c, \ x_2 \mapsto f(h(c,c)), \ x_3 \mapsto h(c,c), \ x_4 \mapsto d\}.$$

Proposition 31 If t is a term and σ and τ are substitutions such that $dom(\sigma) \cap dom(\tau) = \{\}$ holds, then we have

$$(t\sigma)\tau = t(\sigma\tau).$$

This proposition may be proven by induction on t.

Definition 32 (Syntactical Equation) A syntactical equation is a pair $\langle s, t \rangle$ of terms. It is written as s = t. A system of syntactical equations is a set of syntactical equations.

Definition 33 (Unifier) A substitution σ solves a syntactical equation $s \doteq t$ iff we have $s\sigma = t\sigma$. If E is a system of syntactical equations and σ is a substitution that solves every syntactical equations in E, then σ is a unifier of E.

If $E = \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}$ is a system of syntactical equations and σ is a substitution, then we define

$$E\sigma := \{s_1\sigma \doteq t_1\sigma, \cdots, s_n\sigma \doteq t_n\sigma\}.$$

Example: Let us consider the syntactical equation

$$p(x_1, f(x_4)) \doteq p(x_2, x_3)$$

and define the substitution

$$\sigma := \{x_1 \mapsto x_2, \ x_3 \mapsto f(x_4)\}.$$

Then σ solves the given syntactical equation because we have

$$p(x_1, f(x_4))\sigma = p(x_2, f(x_4))$$
 und
 $p(x_2, x_3)\sigma = p(x_2, f(x_4)).$

Next we develop an algorithm for solving a system of syntactical equations. The algorithm we present was published by Martelli and Montanari [MM82]. To begin, we first consider the cases where a syntactical equation s = t is unsolvable. There are two cases: A syntactical equation of the form

$$f(s_1,\cdots,s_m) \doteq g(t_1,\cdots,t_n)$$

is certainly unsolvable if f and g are different function symbols. The reason is that for any substitution σ we have that

$$f(s_1, \dots, s_m)\sigma = f(s_1\sigma, \dots, s_m\sigma)$$
 und $g(t_1, \dots, t_n)\sigma = g(t_1\sigma, \dots, t_n\sigma)$.

If $f \neq g$, then the terms $f(s_1, \dots, s_m)\sigma$ and $g(t_1, \dots, t_n)\sigma$ start with different function symbols and hence they can't be identical.

The other case where a syntactical equation is unsolvable, is a syntactical equation of the following form:

$$x \doteq f(t_1, \dots, t_n)$$
 where $x \in \text{var}(f(t_1, \dots, t_n))$.

This syntactical equation is unsolvable because the term $f(t_1, \dots, t_n)\sigma$ will always contain at least one more occurrence of the function symbol f than the term $x\sigma$.

Now we are able to present an algorithm for solving a system of syntactical equations, provided the system is solvable. The algorithm will also discover if a system of syntactical equations is unsolvable. The algorithm works on pairs of the form $\langle F, \tau \rangle$ where F is a system of syntactical equations and τ is a substitution. The algorithm starts with the pair $\langle E, \{\} \rangle$. Here E is the system of syntactical equations that is to be solved and $\{\}$ represents the empty substitution. The system works by simplifying the pairs $\langle F, \tau \rangle$ using certain reduction rules that are presented below. These reduction rules are applied until we either discover that the system of syntactical equations is unsolvable or else we reduce the pairs until we finally arrive at a pair of the form $\langle \{\}, \mu \rangle$. In this case μ is a unifier of the system of syntactical equations E. The reduction rules are as follows:

1. If $y \in \mathcal{V}$ is a variable that does <u>not</u> occur in the term t, then we can perform the following reduction:

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \quad \leadsto \quad \langle E\{y \mapsto t\}, \sigma\{y \mapsto t\} \rangle \quad \text{if } y \in \mathcal{V} \text{ and } y \not \in \text{var}(t)$$

This reduction rule can be understood as follows: If the system of syntactical equations that is to be solved contains a syntactical equation of the form $y \doteq t$, where the variable y does not occur in the term t, then the syntactical equation $y \doteq t$ can be removed if we apply the substitution $\{y \mapsto t\}$ to both components of the pair

$$\langle E \cup \{y \doteq t\}, \sigma \rangle$$
.

2. If the variable y occurs in the term t, i.e. if $y \in Var(t)$ and, furthermore, $t \neq y$, then the system of syntactical equations $E \cup \{y = t\}$ has no solution. We write this as

$$\langle E \cup \{y \doteq t\}, \sigma \rangle \rightsquigarrow \Omega \quad \text{if } y \in \text{var}(t) \text{ and } y \neq t.$$

3. If $y \in \mathcal{V}$ and $t \notin \mathcal{V}$, then we have:

$$\langle E \cup \{t \doteq y\}, \sigma \rangle \quad \leadsto \quad \langle E \cup \{y \doteq t\}, \sigma \rangle \quad \text{if } y \in \mathcal{V} \text{ and } t \not\in \mathcal{V}.$$

After we apply this rule, we can apply either the first or the second reduction rule thereafter.

4. Trivial syntactical equations can be deleted:

$$\langle E \cup \{x \doteq x\}, \sigma \rangle \quad \leadsto \quad \langle E, \sigma \rangle \quad \text{if } x \in \mathcal{V}.$$

5. If f is an n-ary function symbol we have

$$\langle E \cup \{f(s_1, \dots, s_n) \doteq f(t_1, \dots, t_n)\}, \sigma \rangle \iff \langle E \cup \{s_1 \doteq t_1, \dots, s_n \doteq t_n\}, \sigma \rangle.$$

This rule is the reason that we have to work with a system of syntactical equations, because even if we start with a single syntactical equation the rule given above can increase the number of syntactical equations.

A special case of this rule is the following:

$$\langle E \cup \{c \doteq c\}, \sigma \rangle \rightsquigarrow \langle E, \sigma \rangle.$$

Here c is a nullary function symbol.

6. The system of syntactical equations $E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}$ has no solution if the function symbols f and g are different. Hence we have

$$\langle E \cup \{f(s_1, \dots, s_m) \doteq g(t_1, \dots, t_n)\}, \sigma \rangle \rightsquigarrow \Omega \quad \text{provided } f \neq g.$$

If a system of syntactical equations E is given and we start with the pair $\langle E, \{\} \rangle$, then we can apply the rules given above until one of the following two cases happens:

- 1. We use the second or the sixth of the reduction rules given above. In this case the system of syntactical equations E is unsolvable.
- 2. The pair $\langle E, \{\} \rangle$ is reduced into a pair of the form $\langle \{\}, \mu \rangle$. Then μ is a unifier of E. In this case we write $\mu = \text{mgu}(E)$. If $E = \{s \doteq t\}$, we write $\mu = \text{mgu}(s,t)$. The abbreviation mgu is short for "most general unifier".

Example: We show how to solve the syntactical equation

$$p(x_1, f(x_4)) \doteq p(x_2, x_3).$$

 \Diamond

We have the following reductions:

$$\langle \{p(x_1, f(x_4)) \doteq p(x_2, x_3)\}, \{\} \rangle$$

$$\rightsquigarrow \langle \{x_1 \doteq x_2, f(x_4) \doteq x_3\}, \{\} \rangle$$

$$\rightsquigarrow \langle \{f(x_4) \doteq x_3\}, \{x_1 \mapsto x_2\} \rangle$$

$$\rightsquigarrow \langle \{x_3 \doteq f(x_4)\}, \{x_1 \mapsto x_2\} \rangle$$

$$\rightsquigarrow \langle \{\}, \{x_1 \mapsto x_2, x_3 \mapsto f(x_4)\} \rangle$$

Hence the method is successful and we have that the substitution

$$\{x_1 \mapsto x_2, x_3 \mapsto f(x_4)\}$$

is a solution of the syntactical equation given above.

Example: Next we try to solve the following system of syntactical equations:

$$E = \{ p(h(x_1, c)) \doteq p(x_2), \ q(x_2, d) \doteq q(h(d, c), x_4) \}$$

We have the following reductions:

Hence the substitution $\{x_4 \mapsto d, x_2 \mapsto h(d, c), x_1 \mapsto d\}$ is a solution of the system of syntactical equations given above.

5.5 The Knuth-Bendix Algorithm

Assume we have been given a set R of rewrite rules such that

$$r \prec l$$
 holds for all $l \approx r$ in R

such that the relation \prec is a rewrite order. Given two terms s and t, the Church-Rosser Theorem tells us, that we can decide the question whether $s \leftrightarrow_R^* t$ holds by rewriting s and t into normal forms, provided the relation \to_R is confluent. By Newman's Lemma we know that local confluence is sufficient. Donald E. Knuth and Peter B. Bendix [KB70] have discovered a way to decide whether the term rewriting relation \to_R is locally confluent. To understand their idea, we introduce the notion of a critical pair.

 \Diamond

 \Diamond

Definition 34 (Critical Pair)

Assume we have been given the equations $l_1 \approx r_1$ and $l_2 \approx r_2$. These equations generate a critical pair if and only if all of the following conditions hold:

- (a) There exists a position $u \in \mathcal{P}os(l_1)$ such that l_1/u is not a variable.
- (b) The subterm l_1/u of l_1 is unifiable with l_2 . For the following, assume that μ is a most general unifier of l_1/u and l_2 , i.e. we have

$$\mu = \text{mgu}(l_1/u, l_2).$$

(c) The term s results from rewriting the term $l_1\mu$ by rewriting the subterm $l_1\mu/u$ to the new subterm $r_2\mu$ using the rewrite rule $l_2\approx r_2$:

$$s = l_1 \mu [u \mapsto r_2 \mu].$$

(d) The term t results from rewriting the term $l_1\mu$ into the term $r_1\mu$ using the rule $l_1pprox r_1$, i.e. we have

$$t = r_1 \mu$$
.

Then the pair $\langle s, t \rangle$, which is

$$\langle l_1 \mu [u \mapsto r_2 \mu], r_1 \mu \rangle$$

is a critical pair of $l_1 \approx r_1$ and $l_2 \approx r_2$.

Example: The following example assumes the signature Σ_G from group theory as given. We start with the two equations $(x \cdot y) \cdot z \approx x \cdot (y \cdot z)$ and $i(w) \cdot w \approx e$. Then u = [1] is a position in the term $(x \cdot y) \cdot z$ and we have

$$((x \cdot y) \cdot z)/[1] = x \cdot y$$
, which is not a variable.

The term $x \cdot y$ can be unified with the term $i(w) \cdot w$ and we have

$$\mu := \operatorname{mgu}(x \cdot y, i(w) \cdot w) = \{x \mapsto i(w), \ y \mapsto w\}.$$

Therefore we have

$$((x \cdot y) \cdot z)\mu = (i(w) \cdot w) \cdot z$$

and the latter term can be rewritten by the equation $i(w) \cdot w \approx e$ into the term $e \cdot t$, i.e. we have

$$(i(w) \cdot w) \cdot z \to_{\{i(w) \cdot w \approx e\}} e \cdot z.$$

Furthermore, the same term $(i(w) \cdot w) \cdot z$ can be rewritten by the equation $(x \cdot y) \cdot z \approx x \cdot (y \cdot z)$ into the term $i(w) \cdot (w \cdot z)$:

$$(i(w) \cdot w) \cdot z \to_{\{(x \cdot y) \cdot z \approx x \cdot (y \cdot z)\}} i(w) \cdot (w \cdot z).$$

Therefore, the pair

$$\langle e \cdot z, i(w) \cdot (w \cdot z) \rangle$$

is a critical pair of the two equations $(x \cdot y) \cdot z \approx x \cdot (y \cdot z)$ and $i(w) \cdot w \approx e$.

Remark: If $\langle s, t \rangle$ is a critical pair from two equations in a set R, then the equation $s \approx t$ follows from R, i.e. we have

$$R \models s \approx t$$
.

Definition 35 (Confluent Critical Pair)

A critical pair $\langle s_1, s_2 \rangle$ is confluent w.r.t. a rewrite relation R iff there is a term t such that t is in R-normal form and we have both

$$s_1 \stackrel{*}{\rightarrow}_R t$$
 and $s_2 \stackrel{*}{\rightarrow}_R t$.

Theorem 36 (Knuth-Bendix) If R is a set of rewrite equations such that all critical pairs between equations from R are confluent, then the rewrite relation $\stackrel{*}{\to}_R$ is confluent and hence the question, whether $R \models s \approx t$ can be decided by rewriting both s and t into normal forms \widehat{s} and \widehat{t} :

$$s \stackrel{*}{\to}_R \widehat{s} \quad \text{ and } \quad t \stackrel{*}{\to}_R \widehat{t}$$

Then we have

$$R \models s \approx t$$
 if and only if $\widehat{s} = \widehat{t}$.

To make the above theorem work, if we start with a set E of equations, we first have to order them into a set of rewrite rules R. In general, this will not be sufficient because there will be critical pairs that are not confluent. However, if we can orient these newly derived critical pairs into new rewrite rules, we might be able to extend the set R to a new set of rewrite \widehat{R} such that all critical pairs from equations from \widehat{R} are confluent.

Knuth-Bendix Algorithm: Given a set of equations E the Knuth-Bendix algorithm proceeds as follows:

- 1. We define suitable weight for the function symbols occurring in E and order the function symbols such that every equation $(s \approx t) \in E$ can be ordered as either $s \prec t$ or $t \prec s$. If this is not possible, the algorithm fails.
- 2. Otherwise, call R the set of oriented rewrite rules that result from orienting the equations in E into rewrite rules.
- 3. Compute all critical pairs that can be build from equations in R.
 - (a) If all critical pairs are confluent, then the rewrite relation $\stackrel{*}{\to}_R$ is confluent and the algorithm is successful.
 - (b) If we have found a critical pair $\langle s, t \rangle$ that is not confluent, we orient the equation $s \approx t$ into a rewrite rule. If this is impossible, the algorithm fails. Otherwise, we add the oriented equation to R. Now the set R could generate additional critical pairs. Hence we must go back to the beginning of step 3.

The algorithm shown above can have three different outcomes:

- 1. It can fail because it has generated an equation that can not be oriented into a rewrite rule.
- 2. It can stop with a set of rewrite rules R such that \rightarrow_R is confluent.
- 3. It can run forever because an infinite set of critical pairs is generated.

My GitHub repository contains the Jupyter notebook

which contains an implementation of the Knuth-Bendix algorithm. It also contains a number of equational theories E where the Knuth-Bendix algorithm is successful.

Example: We test the Knuth-Bendix algorithm with the axioms of group theory. These axioms are the following:

- (a) $1 \cdot x \approx x$,
- (b) $i(x) \cdot x \approx 1$, and
- (c) $(x \cdot y) \cdot z \approx x \cdot (y \cdot z)$.

It is obvious how these equations can be oriented into rewrite rules. The resulting rules are:

- $1 \cdot x \to x$,
- $i(x) \cdot x \to 1$, and
- $(x \cdot y) \cdot z \to x \cdot (y \cdot z)$.

From $(x \cdot y) \cdot z \to x \cdot (y \cdot z)$ and $i(a) \cdot a \to 1$ we conclude

$$(i(a) \cdot a) \cdot z \to i(a) \cdot (a \cdot z)$$
 and $(i(a) \cdot a) \cdot z \to 1 \cdot z$

Hence we have found the new rewrite rule

$$i(a) \cdot (a \cdot z) \to z$$
.

From $i(a) \cdot (a \cdot z) \to z$ and $i(x) \cdot x$ we conclude

$$i(i(x)) \cdot (i(x) \cdot x) \to x$$
 and $i(i(x)) \cdot (i(x) \cdot x) \to i(i(x)) \cdot 1$.

Hence we have found the new rewrite rule

$$i(i(x))\cdot 1\to x.$$

From $i(x) \cdot (x \cdot y) \to y$ and $i(a) \cdot (a \cdot b) \to b$ we conclude

$$i(i(a)) \cdot (i(a) \cdot (a \cdot b)) \rightarrow a \cdot b$$
 and $i(i(a)) \cdot (i(a) \cdot (a \cdot b)) \rightarrow i(i(a)) \cdot b$.

Hence we have found the new rewrite rule

$$i(i(a)) \cdot b \to a \cdot b$$

From $i(i(a)) \cdot b \to a \cdot b$ and $i(i(x)) \cdot 1 \to x$ we conclude

$$i(i(a)) \cdot 1 \to a \cdot 1$$
 and $i(i(a)) \cdot 1 \to a$

Hence we have found the new rewrite rule

$$a \cdot 1 \rightarrow a$$
.

At this point, the rewrite rule $i(i(x)) \cdot 1 \to x$ is simplified into the rule

$$i(i(x)) \to x$$
.

From $i(x) \cdot (x \cdot y) \to y$ and $i(i(a)) \to a$ we conclude

$$i(i(a)) \cdot (i(a) \cdot y) \to y$$
 and $i(i(a)) \cdot (i(a) \cdot y) \to a \cdot (i(a) \cdot y)$.

Hence we have found the new rule

$$a \cdot (i(a) \cdot y) \rightarrow y$$
.

From $i(x) \cdot x \to 1$ and $i(i(a)) \to a$ we conclude

$$i(i(a)) \cdot i(a) \to 1$$
 and $i(i(a)) \cdot i(a) \to a \cdot i(a)$.

Hence we have found the new rule

$$a \cdot i(a) \to 1$$
.

From $a \cdot i(a) \to 1$ and $1 \cdot x \to x$ we conclude

$$1 \cdot i(1) \to 1$$
 and $1 \cdot i(1) \to i(1)$.

Hence we have shown

$$i(1) \rightarrow 1$$
.

From $a \cdot i(a) \to 1$ and $(x \cdot y) \cdot z \to x \cdot (y \cdot z)$ and we conclude

$$(x \cdot y) \cdot i(x \cdot y) \to 1$$
 and $(x \cdot y) \cdot i(x \cdot y) \to x \cdot (y \cdot i(x \cdot y)).$

Hence we have found the new rule

$$x \cdot (y \cdot i(x \cdot y)) \to 1.$$

From $a \cdot (i(a) \cdot b) \to b$ and $x \cdot (y \cdot i(x \cdot y)) \to 1$ we conclude

$$a \cdot \Big(i(a) \cdot \Big(y \cdot i\big(i(a) \cdot y\big)\Big)\Big) \to y \cdot i\big(i(a) \cdot y\big) \quad \text{ und } \quad a \cdot \Big(i(a) \cdot \Big(y \cdot i\big(i(a) \cdot y\big)\Big)\Big) \to a \cdot 1 \quad \text{ and } \quad a \cdot \Big(i(a) \cdot \Big(y \cdot i\big(i(a) \cdot y\big)\Big)\Big) \to a \cdot 1$$

As we already know that $a \cdot 1 \to a$ we have found the new rule

$$y \cdot i(i(a) \cdot y) \to a$$
.

From $y \cdot i(i(a) \cdot y) \to a$ and $i(i(x)) \to x$ we conclude

$$y \cdot i(i(i(x)) \cdot y) \to i(x)$$
 and $y \cdot i(i(i(x)) \cdot y) \to y \cdot i(x \cdot y)$.

Hence we have found the new rule

$$y \cdot i(x \cdot y) \rightarrow i(x)$$
.

From $i(a)\cdot (a\cdot b)\to b$ and $y\cdot i(x\cdot y)\to i(x)$ we conclude

$$i(a) \cdot (a \cdot i(x \cdot a)) \to i(x \cdot a)$$
 and $i(a) \cdot (a \cdot i(x \cdot a)) \to i(a) \cdot i(x)$.

Hence we have found the new rule

$$i(x \cdot a) \rightarrow i(a) \cdot i(x)$$
.

This last rule makes the rules

$$y \cdot i(x \cdot y) \to i(x), \quad y \cdot i(i(a) \cdot y) \to a, \quad \text{and} \quad x \cdot (y \cdot i(x \cdot y)) \to 1$$

redundant as all of these rules can be simplified to an identity using the rule $i(x \cdot a) \to i(a) \cdot i(x)$. Therefore, we have found the following set of rewrite rules.

- 1. $1 \cdot x \to x$,
- $2. i(x) \cdot x \to 1,$
- 3. $(x \cdot y) \cdot z \to x \cdot (y \cdot z)$,
- 4. $i(a) \cdot (a \cdot z) \rightarrow z$.
- 5. $a \cdot 1 \rightarrow a$,
- 6. $i(1) \to 1$,
- 7. $i(i(x)) \to x$,

- 8. $a \cdot i(a) \rightarrow 1$,
- 9. $a \cdot (i(a) \cdot y) \rightarrow y$,
- 10. $i(x \cdot a) \rightarrow i(a) \cdot i(x)$.

It can be shown that all critical pairs resulting from these rules can be simplified to identities. Hence this set is a confluent set of rewrite rules for group theory. Therefore, the validity of any equation $s \approx t$ in group theory can be checked by rewriting s and t into normal forms using the rewrite rules given above. Then the equation $s \approx t$ is valid in group theory if and only if the normal forms of s and t are identical.

Exercise 12: A quasi-group is a structure

$$\mathcal{G} = \langle G, \cdot, /, \setminus \rangle$$

such that

- 1. G is a non-empty set,
- $2. : G \times G \rightarrow G$
- 3. $/: G \times G \rightarrow G$,
- $4. \setminus : G \times G \to G.$
- 5. Furthermore, the following axioms have to be satisfied:
 - (a) $x \cdot (x \setminus y) = y$,
 - (b) $(x/y) \cdot y = x$,
 - (c) $x \setminus (x \cdot y) = y$,
 - (d) $(x \cdot y)/y = x$.

Compute the set of all non-trivial critical pairs from these equations.

Hint: The two non-trivial critical pairs arise from trying to simplify the left hand side of equation (d) with equation (a) and from simplifying the left hand side of equation (c) with (b).

5.6 Literature

The book Term Rewriting and All That by Franz Baader and Tobias Nipkow [BN98] gives a much more detailed account of equational theorem proving via term rewriting.

Chapter 6

Linear Regression

A great deal of the current success of artificial intelligence is due to recent advances in machine learning. In order to get a first taste of what machine learning is about, we introduce linear regression in this chapter, since linear regression is one of the most basic algorithms in machine learning. It is also the foundation for more advanced forms of machine learning like logistic regression and neural networks. Furthermore, linear regression is surprisingly powerful. Finally, many of the fundamental problems of machine learning can already be illustrated with linear regression. Therefore it is only natural that we begin our study of machine learning with the study of linear regression.

6.1 Simple Linear Regression

Assume we want to know how the engine displacement of a car engine relates to its fuel consumption. One approach to understand this relation would be to derive a theoretical model that is able to predict the fuel consumption from the engine displacement by using the appropriate laws of physics and chemistry. However, due to our lack of understanding of the underlying theory, this is not an option for us. Instead, we follow a statistical approach and collect data from a large number of cars. For these cars, we compare their engine displacement with the corresponding fuel consumption. This way, we will collect a set of m observations of the form $\langle x_1, y_1 \rangle, \dots, \langle x_m, y_m \rangle$ where x_i is the engine displacement of the engine in the i-th car, while y_i is the fuel consumption of the i-th car. We call x the independent variable, while y is the dependent variable. We define the vectors \mathbf{x} and \mathbf{y} as follows:

$$\mathbf{x} := \langle x_1, \cdots, x_m \rangle^{\top}$$
 and $\mathbf{y} := \langle y_1, \cdots, y_m \rangle^{\top}$.

Here, the postfix operator $^{\top}$ is interpreted as the transposition operator, i.e. \mathbf{x} and \mathbf{y} are considered to be column vectors. By using the transposition operator we are able to write these vectors in a single line.

In linear regression, we use a linear model and assume that the dependent variable y_i is related to the independent variable x_i via a linear equation of the form

$$y_i = \vartheta_1 \cdot x_i + \vartheta_0.$$

We do not expect this equation to hold exactly. The reason is that there are many other factors besides the engine displacement that influence the fuel consumption. For example, both the weight of a car and its aerodynamics certainly influence the fuel consumption. We want to calculate those values ϑ_0 and ϑ_1 such that the <u>mean squared error</u>, which is defined as

$$MSE(\vartheta_0, \vartheta_1) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} (\vartheta_1 \cdot x_i + \vartheta_0 - y_i)^2, \tag{6.1}$$

is minimized. It can be shown that the solution to this minimization problem is given as follows:

$$\vartheta_1 = r_{x,y} \cdot \frac{s_y}{s_x} \quad \text{and} \quad \vartheta_0 = \bar{\mathbf{y}} - \vartheta_1 \cdot \bar{\mathbf{x}}.$$
(6.2)

This solution makes use of the values $r_{x,y}$, s_x , and s_y . In order to define these values, we first define the sample mean values $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ of \mathbf{x} and \mathbf{y} respectively, i.e. we have

$$\bar{\mathbf{x}} = \frac{1}{m} \cdot \sum_{i=1}^{m} x_i$$
 and $\bar{\mathbf{y}} = \frac{1}{m} \cdot \sum_{i=1}^{m} y_i$.

Furthermore, s_x and s_y are the corrected sample standard deviations of **x** and **y**, i.e. we have

$$s_x = \sqrt{\frac{1}{m-1} \cdot \sum_{i=1}^m (x_i - \bar{\mathbf{x}})^2}$$
 and $s_y = \sqrt{\frac{1}{m-1} \cdot \sum_{i=1}^m (y_i - \bar{\mathbf{y}})^2}$.

In the rest of these lecture notes, s_x and s_y will be referred to as sample standard deviation, i.e. we drop the attribute *corrected*. Next, $Cov[\mathbf{x}, \mathbf{y}]$ is the sample covariance and is defined as

$$Cov[\mathbf{x}, \mathbf{y}] = \frac{1}{(m-1)} \cdot \sum_{i=1}^{m} (x_i - \bar{\mathbf{x}}) \cdot (y_i - \bar{\mathbf{y}}).$$

Finally, $r_{x,y}$ is the sample correlation coefficient that is defined as

$$r_{x,y} = \frac{1}{(m-1) \cdot s_x \cdot s_y} \cdot \sum_{i=1}^{m} (x_i - \bar{\mathbf{x}}) \cdot (y_i - \bar{\mathbf{y}}) = \frac{\operatorname{Cov}[\mathbf{x}, \mathbf{y}]}{s_x \cdot s_y}.$$

The number $r_{x,y}$ is also known as the Pearson correlation coefficient or Pearson's r. It is named after Karl Pearson (1857 – 1936). Note that the formula for the parameter ϑ_1 can be simplified to

$$\vartheta_1 = \frac{\sum_{i=1}^{m} (x_i - \bar{\mathbf{x}}) \cdot (y_i - \bar{\mathbf{y}})}{\sum_{i=1}^{m} (x_i - \bar{\mathbf{x}})^2}$$

$$(6.3)$$

This latter formula should be used to calculate ϑ_1 . However, the previous formula is also useful because it shows that the correlation coefficient is identical to the coefficient ϑ_1 , provided the variables \mathbf{x} and \mathbf{y} have been normalized so that their standard deviation is 1.

Exercise 13: Prove Equation 6.2 and Equation 6.3.

Hint: The expression $MSE(\vartheta_0, \vartheta_1)$ is a quadratic function with respect to the parameters ϑ_0 and ϑ_1 . Therefore, it has exactly one global minimum. Take the partial derivatives of $MSE(\vartheta_0, \vartheta_1)$ with respect to ϑ_0 and ϑ_1 . If the expression $MSE(\vartheta_0, \vartheta_1)$ is minimal, then these partial derivatives have to be equal to 0.

6.1.1 Assessing the Quality of Linear Regression

Assume that we have been given a set of m observations of the form $\langle x_1, y_1 \rangle, \dots, \langle x_m, y_m \rangle$ and that we have calculated the parameters ϑ_0 and ϑ_1 according to Equation 6.2 and Equation 6.3. Provided that not all x_i have the same value, these formulae will return two numbers for ϑ_0 and ϑ_1 that define a linear model for \mathbf{y} in terms of \mathbf{x} . However, at the moment we still lack a number that tells us how good this linear model really is. In order to judge the quality of the linear model given by

$$y = \vartheta_0 + \vartheta_1 \cdot x$$

we can compute the mean squared error according to Equation 6.1. However, the mean squared error is an absolute number that, by itself, is difficult to interpret. The reason is that the variable \mathbf{y} might be inherently noisy and we have to relate this noise to the mean squared error. Now the noise contained in \mathbf{y} can be measured by the sample variance of \mathbf{y} and is given by the formula

$$Var(y) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} (y_i - \bar{y})^2. \tag{6.4}$$

If we compare this formula to the formula for the mean squared error

$$\mathrm{MSE}(\vartheta_0,\vartheta_1) := \frac{1}{m-1} \cdot \sum_{i=1}^m \bigl(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\bigr)^2 = \frac{1}{m-1} \cdot \sum_{i=1}^m \bigl(y_i - \vartheta_1 \cdot x_i - \vartheta_0\bigr)^2,$$

we see that the sample variance of \mathbf{y} is an upper bound for the mean squared error since we have

$$Var(\mathbf{y}) = MSE(\bar{\mathbf{y}}, 0),$$

i.e. the sample variance is the value that we would get for the mean squared error if we set ϑ_0 to the average value of \mathbf{y} and ϑ_1 to zero. Since ϑ_0 and ϑ_1 are chosen to minimize the mean squared error, we have

$$MSE(\vartheta_0, \vartheta_1) \leq MSE(\bar{\mathbf{y}}, 0) = Var(\mathbf{y}).$$

The mean squared error is an absolute value and, therefore, difficult to interpret. The fraction

$$\frac{\texttt{MSE}(\vartheta_0,\vartheta_1}{\texttt{Var}(y)}$$

is called the proportion of the unexplained variance because it is the variance that is still left if we use our linear model to predict the values of \mathbf{y} given the values of \mathbf{x} . The proportion of the explained variance which is also known as the \mathbb{R}^2 statistic is defined as

$$\mathbf{R}^2 := \frac{\mathbf{Var}(\mathbf{y}) - \mathbf{MSE}(\vartheta_0, \vartheta_1)}{\mathbf{Var}(\mathbf{y})} = 1 - \frac{\mathbf{MSE}(\vartheta_0, \vartheta_1)}{\mathbf{Var}(\mathbf{y})}. \tag{6.5}$$

The statistic \mathbb{R}^2 measures the quality of our model: If it is small, then our model does not explain the variation of the value of \mathbf{y} when the value of \mathbf{x} changes. On the other hand, if it is near to 100%, then our model does a good job in explaining the variation of \mathbf{y} when \mathbf{x} changes.

Since the formulae for Var(y) and $MSE(\vartheta_0, \vartheta_1)$ have the same denominator m-1, this denominator can be cancelled when \mathbb{R}^2 is computed. To this end we define the total sum of squares TSS as

$$\mathtt{TSS} := \sum_{i=1}^m \left(y_i - \bar{\mathbf{y}}\right)^2 = (m-1) \cdot \mathtt{Var}(\mathbf{y})$$

and the residual sum of squares RSS as

$$\mathrm{RSS} := \sum_{i=1}^m \bigl(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\bigr)^2 = (m-1) \cdot \mathrm{MSE}(\vartheta_0, \vartheta_1).$$

Then the formula for the R² statistic can be written as

$$R^2 = 1 - \frac{RSS}{TSS}.$$

This is the formula that we will use when we implement simple linear regression.

It should be noted that R^2 is the square of Pearson's r. The notation is a bit inconsistent since Pearson's r is written in lower case, while R^2 is written in upper case. However, since this is the notation used in most books on statistics, we will use it too. The number R^2 is also known as the

coefficient of determination. It tells us to what extent the value of the variable y is determined by the value of x.

6.1.2 Putting the Theory to the Test

In order to get a better feeling for linear regression, we want to test it to investigate the factors that determine the fuel consumption of cars. Figure 6.1 on page 136 shows the head of the data file "cars.csv" which I have adapted from the file

```
https://www.statlearning.com/s/Auto.csv.
```

Figure 6.1 on page 136 shows the column headers and the first ten data entries contained in this file. Altogether, this file contains data of 392 different car models.

```
mpg, cyl, displacement,
1
                                 hp, weight,
                                               acc, year, name
                       307.0, 130.0, 3504.0, 12.0,
                                                       70, chevrolet chevelle malibu
    18.0,
3
    15.0,
             8,
                       350.0, 165.0, 3693.0, 11.5,
                                                       70, buick skylark 320
    18.0,
            8,
                       318.0, 150.0, 3436.0, 11.0,
                                                       70, plymouth satellite
    16.0,
            8,
                       304.0, 150.0, 3433.0, 12.0,
                                                       70, amc rebel sst
    17.0,
            8,
                       302.0, 140.0, 3449.0, 10.5,
                                                       70, ford torino
    15.0,
                       429.0, 198.0, 4341.0, 10.0,
                                                       70, ford galaxie 500
            8,
    14.0,
            8,
                       454.0, 220.0, 4354.0, 9.0,
                                                       70, chevrolet impala
                       440.0, 215.0, 4312.0, 8.5,
    14.0,
            8,
                                                       70, plymouth fury iii
                       455.0, 225.0, 4425.0, 10.0,
                                                       70, pontiac catalina
    14.0,
             8,
10
                       390.0, 190.0, 3850.0, 8.5,
    15.0,
            8,
                                                       70, amc ambassador dpl
11
```

Figure 6.1: The head of the file cars.csv.

The file "cars.csv" is part of the data set accompanying the excellent book Introduction to Statistical Learning by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani [JWHT14]. The file cars.csv contains the fuel consumption of a number of different cars that were in widespread use during the seventies and early eighties of the last century. The first column of this data set gives the miles per gallon of a car, i.e. the number of miles a car can drive with one gallon of gas. Note that this number is in reciprocal relation to the fuel consumption: If a car A can drive twice as many miles per gallon than another car B, then the fuel consumption of A is half of the fuel consumption of B. Furthermore, besides the miles per gallon, for every car the following other parameters are listed:

- 1. cyl is the number of cylinders,
- 2. displacement is the engine displacement in cubic inches, (100 cubic inch is 1.638 706 4 litres)
- 3. hp is the engine power given in units of horsepower,
- 4. weight is the weight in pounds (1 pound is the same as 0.453 592 37 kg),
- 5. acc is the acceleration given as the time in seconds needed to accelerate from 0 miles per hour to 60 miles per hour,
- 6. year is the year in which the model was introduced, and
- 7. name is the name of the model.

Our aim is to determine what part of the fuel consumption of a car is explained by its engine displacement. To this end, I have written the function simple_linear_regression shown in Figure 6.2 on page 137.

```
def simple_linear_regression(X, Y):
2
         This function implements linear regression.
3
                  explaining variable, numpy array
         * X:
         * Y:
                  dependent variable, numpy array
         Output: The R2 value of the linear regression.
                = len(X)
10
         xMean = np.mean(X);
11
         yMean = np.mean(Y);
12
         theta1 = np.sum((X - xMean) * (Y - yMean)) / np.sum((X - xMean) ** 2)
         theta0 = yMean - theta1 * xMean;
                = np.sum((Y - yMean) ** 2)
         TSS
15
         RSS
                = np.sum((theta1 * X + theta0 - Y) ** 2)
16
                = 1 - RSS / TSS;
         R2
17
         return R2
18
```

Figure 6.2: Simple Linear Regression

The procedure simple_linear_regression takes two arguments:

- (a) X is a NumPy array containing the independent variable.
- (b) Y is a NumPy array containing the dependent variable.

The implementation of the procedure simple_linear_regression works as follows:

- 1. m is the number of data that are present in the array X.
- 2. xMean is the mean value \bar{x} of the independent variable x.
- 3. yMean is the mean value $\bar{\mathbf{y}}$ of the dependent variable \mathbf{y} .
- 4. The coefficient theta1 is computed according to Equation 6.3, which is repeated here for convenience:

$$\vartheta_1 = \frac{\sum_{i=1}^{m} (x_i - \bar{\mathbf{x}}) \cdot (y_i - \bar{\mathbf{y}})}{\sum_{i=1}^{m} (x_i - \bar{\mathbf{x}})^2}.$$

Note that the expression (X - xMean) computes an array of the same shape as X by subtracting xMean from every entries of X. Next, the expression (X - xMean) * (Y - yMean) computes the elementwise product of the arrays X - xMean and Y - yMean. The expression (X-xMean)**2 computes the elementwise squares of the array X - xMean. Finally, the function sum computes the sum of all the elements of an array.

5. The coefficient theta0 is computed according to Equation 6.2, which reads

$$\vartheta_0 = \bar{\mathbf{y}} - \vartheta_1 \cdot \bar{\mathbf{x}}.$$

6. TSS is the total sum of squares and is computed using the formula

$$TSS = \sum_{i=1}^{m} (y_i - \bar{\mathbf{y}})^2.$$

7. RSS is the residual sum of squares and is computed as

$$RSS := \sum_{i=1}^{m} (\vartheta_1 \cdot x_i + \vartheta_0 - y_i)^2.$$

8. R2 is the R² statistic and measures the proportion of the explained variance. It is computed using the formula

$$\mathtt{R}^2 = \frac{\mathtt{TSS} - \mathtt{RSS}}{\mathtt{TSS}}.$$

```
import csv
    import numpy as np
2
    with open('cars.csv') as input_file:
                      = csv.reader(input_file, delimiter=',')
         reader
5
         line_count
6
         kpl
         displacement = []
         for row in reader:
9
             if line_count != 0:
10
                              .append(float(row[0]) * 0.00425144)
                 displacement.append(float(row[2]) * 0.0163871)
12
                 line_count += 1
13
    m = len(displacement)
14
    X = np.array(displacement)
15
    Y = np.array([1 / kpl[i] for i in range(m)])
16
    R2 = simple_linear_regression(X, Y)
17
    print(f'The explained variance is {R2}%')
```

Figure 6.3: Calling the procedure simple_linear_regression.

In order to use the function we can use the code that is shown in Figure 6.3 on page 138.

- 1. We import the module csv in order to be able to read the Csv file "cars.csv" conveniently.
- 2. We import the module numpy in order to use NumPy arrays.
- 3. We open the file "cars.csv".
- 4. This file is processed as a Csv file where different columns are separated by the character ",".

dependent variable	explained variance
displacement	0.75
number of cylinders	0.70
horsepower	0.73
weight	0.78
acceleration	0.21
year of build	0.31

Table 6.1: Explained variance for various dependent variables.

- 5. kpl is a list of the numbers that appear in the first column of the Csv file. The numbers in the Csv file are interpreted as the miles per gallon of a car. These numbers are converted into metric units, i.e. how many kilometers a car can run on a litre.
- 6. displacement is a list the numbers appearing in the third column of the CSV file. These numbers are interpreted as the *engine displacement* in cubic inches. These numbers are converted to litres.
- 7. The first line of the Csv file contains a header. This header is skipped. In order to do so we use the variable line_count.
- 8. m is the number of data pairs that have been read.
- 9. The independent variable X is given by the engine displacement. In order to be able to use NumPy features later we convert this list into a NumPy array.
- 10. The dependent variable Y is given by the inverse of the variable kph.
- 11. Finally, the coefficient of determination R^2 is computed. Here we use the function simple_linear_regression that is shown in Figure 6.2 on page 137.

In the same way as we have computed the coefficient of determination that measures how the fuel consumption is influenced by the engine displacement we can also compute the coefficient of determination for other variables like the number of cylinders or the weight of the car. The resulting values are shown in Table 6.1. It seems that, given the data in the file "cars.csv", the best indicator for the fuel consumption is the weight of a car. The displacement, the power hp of an engine, and the number of cylinders cyl are also good predictors. But notice that the weight is the real cause of fuel consumption: If a car is heavy, it will also need a more powerful engine. Hence the variable hp is correlated with the variable weight and will therefore also provide a reasonable explanation of the fuel consumption, although the high engine power is not the most important cause of the fuel consumption.

6.2 General Linear Regression

In practise, it is rarely the case that a given observed variable y only depends on a single variable x. To take the example of the fuel consumption of a car further, in general we would expect that the fuel consumption of a car depends not only on the engine displacement of the car but is also related to the other parameters. For example, it seems reasonable to assume that the mass of the car should influence its fuel consumption. To be able to model this kind of behaviour, we present the theory of general linear regression. In a general regression problem we are given a list of m pairs of the form

 $\langle \mathbf{x}^{(i)}, y^{(i)} \rangle$ where $\mathbf{x}^{(i)} \in \mathbb{R}^p$ and $y^{(i)} \in \mathbb{R}$ for all $i \in \{1, \dots, m\}$. The number p is called the number of features, while the pairs are called the training examples. Our goal is to compute a function

$$F: \mathbb{R}^p \to \mathbb{R}$$

such that $F(\mathbf{x}^{(i)})$ approximates $y^{(i)}$ as precisely as posssible for all $i \in \{1, \dots, m\}$, i.e. we want to have

$$F(\mathbf{x}^{(i)}) \approx y^{(i)}$$
 for all $i \in \{1, \dots, m\}$.

In order to make the notation $F(\mathbf{x}^{(i)}) \approx y^{(i)}$ more precise, we define the mean squared error

$$MSE := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \left(F(\mathbf{x}^{(i)}) - y^{(i)} \right)^{2}. \tag{6.6}$$

Then, given the list of training examples $[\langle \mathbf{x}^{(1)}, y^{(1)} \rangle, \cdots, \langle \mathbf{x}^{(n)}, y^{(n)} \rangle]$, our goal is to minimize the MSE. In order to proceed, we need to have a model for the function F. The simplest model is a linear model, i.e. we assume that F is given as

$$F(\mathbf{x}) = \sum_{j=1}^{p} w_j \cdot x_j + b = \mathbf{x}^{\top} \cdot \mathbf{w} + b$$
 where $\mathbf{w} \in \mathbb{R}^p$ and $b \in \mathbb{R}$.

Here, the expression $\mathbf{x}^{\top} \cdot \mathbf{w}$ denotes the matrix product of the vector \mathbf{x}^{\top} , which is viewed as a 1-by-p matrix, and the vector \mathbf{w} , where \mathbf{w} is viewed as a p-by-1 matrix. Alternatively, this expression could be interpreted as the dot product of the vector \mathbf{x} and the vector \mathbf{w} . At this point you might wonder why it is useful to introduce matrix notation here. The reason is that this notation shortens the formula and, furthermore, is more efficient to implement since most programming languages used in machine learning have special library support for matrix operations. Provided the computer is equipped with a graphics card, some programming languages are even able to delegate matrix operations to the graphics card. This results in a considerable speed-up.

The definition of F given above is the model used in linear regression. Here, **w** is called the weight vector and b is called the bias. It turns out that the notation can be simplified if we extend the p-dimensional feature vector **x** to a (p+1)-dimensional vector \mathbf{x}' such that

$$x'_j := x_j$$
 for all $j \in \{1, \dots, p\}$ and $x'_{p+1} := 1$.

To put it in words, the vector \mathbf{x}' results from the vector \mathbf{x} by appending the number 1:

$$\mathbf{x}' = \langle x_1, \cdots, x_p, 1 \rangle^{\top}$$
 where $\langle x_1, \cdots, x_p \rangle = \mathbf{x}^{\top}$.

Furthermore, we define

$$\mathbf{w}' := \langle w_1, \cdots, w_p, b \rangle^{\top}$$
 where $\langle w_1, \cdots, w_p \rangle = \mathbf{w}^{\top}$.

Then we have

$$F(\mathbf{x}) = \mathbf{x}^{\top} \cdot \mathbf{w} + b = \mathbf{x}'^{\top} \cdot \mathbf{w}'.$$

Hence, the bias has been incorporated into the weight vector at the cost of appending the number 1 at the end of input vector \mathbf{x} . As we want to use this simplification, from now on we assume that the input vectors $\mathbf{x}^{(i)}$ have all been extended so that their last component is always 1. Then we can just write \mathbf{x} and \mathbf{w} instead of \mathbf{x}' and \mathbf{w}' . Using this simplification, we define the function F as

$$F(\mathbf{x}) := \mathbf{x}^{\top} \cdot \mathbf{w}.$$

Now equation (6.6) can be rewritten as follows:

$$MSE(\mathbf{w}) = \frac{1}{m-1} \cdot \sum_{i=1}^{m} \left((\mathbf{x}^{(i)})^{\top} \cdot \mathbf{w} - y^{(i)} \right)^{2}. \tag{6.7}$$

Our aim is to rewrite the sum appearing in this equation as a scalar product of a vector with itself. To this end, we first define the vector \mathbf{y} as follows:

$$\mathbf{y} := \langle y^{(1)}, \cdots, y^{(m)} \rangle^{\top}.$$

Note that $\mathbf{y} \in \mathbb{R}^m$ since it has a component for all of the m training examples. Next, we define the design matrix X as follows:

$$X := \left(egin{array}{c} \left(\mathbf{x}^{(1)}
ight)^{ op} \ dots \left(\mathbf{x}^{(m)}
ight)^{ op} \end{array}
ight)$$

In the literature, X is also called the feature matrix. If X is defined in this way, the row vectors of the matrix X are the transpositions of the vectors $\mathbf{x}^{(i)}$. Then we have the following:

$$X \cdot \mathbf{w} - \mathbf{y} = \begin{pmatrix} (\mathbf{x}^{(1)})^{\top} \\ \vdots \\ (\mathbf{x}^{(m)})^{\top} \end{pmatrix} \cdot \mathbf{w} - \mathbf{y} = \begin{pmatrix} (\mathbf{x}^{(1)})^{\top} \cdot \mathbf{w} - y_1 \\ \vdots \\ (\mathbf{x}^{(m)})^{\top} \cdot \mathbf{w} - y_m \end{pmatrix}$$

Taking the square of the vector $X \cdot \mathbf{w} - \mathbf{y}$ we discover that we can rewrite equation (6.7) as follows:

$$MSE(\mathbf{w}) = \frac{1}{m-1} \cdot (X \cdot \mathbf{w} - \mathbf{y})^{\top} \cdot (X \cdot \mathbf{w} - \mathbf{y}). \tag{6.8}$$

6.2.1 Some Useful Gradients

In the last section, we have computed the mean squared error $MSE(\mathbf{w})$ using equation (6.8). Our goal is to minimize the $MSE(\mathbf{w})$ by choosing the weight vector \mathbf{w} appropriately. A necessary condition for $MSE(\mathbf{w})$ to be minimal is

$$\nabla MSE(\mathbf{w}) = \mathbf{0}$$

i.e. the gradient of $MSE(\mathbf{w})$ with respect to \mathbf{w} needs to be zero. In order to prepare for the computation of $\nabla MSE(\mathbf{w})$, we first compute the gradient of two simpler functions.

Computing the Gradient of $f(\mathbf{x}) = \mathbf{x}^{\top} \cdot C \cdot \mathbf{x}$

Suppose the function $f: \mathbb{R}^n \to \mathbb{R}$ is defined as

$$f(\mathbf{x}) := \mathbf{x}^{\top} \cdot C \cdot \mathbf{x}$$
 where $C \in \mathbb{R}^{n \times n}$.

If we write the matrix C as $C = (c_{i,j})_{\substack{i=1,\dots,n\\j=1,\dots,n}}$ and the vector \mathbf{x} as $\mathbf{x} = \langle x_1,\dots,x_n \rangle^{\top}$, then $f(\mathbf{x})$ can be computed as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i \cdot \sum_{j=1}^{n} c_{i,j} \cdot x_j = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i \cdot c_{i,j} \cdot x_j.$$

We compute the partial derivative of f with respect to x_k and use the product rule together with the definition of the Kronecker delta $\delta_{i,j}$, which is defined as 1 if i = j and as 0 otherwise:

$$\delta_{i,j} := \begin{cases} 1 & \text{if } i = j; \\ 0 & \text{otherwise.} \end{cases}$$

Then the partial derivative of f with respect to x_k , which is written as $\frac{\partial f}{\partial x_k}$, is computed as follows:

$$\frac{\partial f}{\partial x_k} = \sum_{i=1}^n \sum_{j=1}^n \left(\frac{\partial x_i}{\partial x_k} \cdot c_{i,j} \cdot x_j + x_i \cdot c_{i,j} \cdot \frac{\partial x_j}{\partial x_k} \right)
= \sum_{i=1}^n \sum_{j=1}^n \left(\delta_{i,k} \cdot c_{i,j} \cdot x_j + x_i \cdot c_{i,j} \cdot \delta_{j,k} \right)
= \sum_{j=1}^n c_{k,j} \cdot x_j + \sum_{i=1}^n x_i \cdot c_{i,k}
= \left(C \cdot \mathbf{x} \right)_k + \left(C^\top \cdot \mathbf{x} \right)_k$$

Hence we have shown that

$$\nabla f(\mathbf{x}) = (C + C^{\top}) \cdot \mathbf{x}.$$

If the matrix C is symmetric, i.e. if $C = C^{\top}$, this simplifies to

$$\nabla f(\mathbf{x}) = 2 \cdot C \cdot \mathbf{x}.$$

Next, if the function $g: \mathbb{R}^n \to \mathbb{R}$ is defined as

$$g(\mathbf{x}) := \mathbf{b}^{\top} \cdot A \cdot \mathbf{x}$$
, where $\mathbf{b} \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$,

then a similar, but slightly easier, calculation shows that

$$\nabla g(\mathbf{x}) = A^{\top} \cdot \mathbf{b}.$$

Exercise 14: Prove this equation.

6.2.2 Deriving the Normal Equation

Next, we will derive the so called normal equation for linear regression. To this end, we first expand the product in equation (6.8):

$$\mathsf{MSE}(\mathbf{w}) = \frac{1}{m-1} \cdot (X \cdot \mathbf{w} - \mathbf{y})^{\top} \cdot (X \cdot \mathbf{w} - \mathbf{y}) \\
= \frac{1}{m-1} \cdot (\mathbf{w}^{\top} \cdot X^{\top} - \mathbf{y}^{\top}) \cdot (X \cdot \mathbf{w} - \mathbf{y}) & \text{since } (A \cdot B)^{\top} = B^{\top} \cdot A^{\top} \\
= \frac{1}{m-1} \cdot (\mathbf{w}^{\top} \cdot X^{\top} \cdot X \cdot \mathbf{w} - \mathbf{y}^{\top} \cdot X \cdot \mathbf{w} - \mathbf{w}^{\top} \cdot X^{\top} \cdot \mathbf{y} + \mathbf{y}^{\top} \cdot \mathbf{y}) \\
= \frac{1}{m-1} \cdot (\mathbf{w}^{\top} \cdot X^{\top} \cdot X \cdot \mathbf{w} - 2 \cdot \mathbf{y}^{\top} \cdot X \cdot \mathbf{w} + \mathbf{y}^{\top} \cdot \mathbf{y}) & \text{since } \mathbf{w}^{\top} \cdot X^{\top} \cdot \mathbf{y} = \mathbf{y}^{\top} \cdot X \cdot \mathbf{w}$$

The fact that

$$\mathbf{w}^{\top} \cdot X^{\top} \cdot \mathbf{y} = \mathbf{y}^{\top} \cdot X \cdot \mathbf{w}$$

might not be immediately obvious. It follows from two facts:

1. For two matrices A and B such that the matrix product $A \cdot B$ is defined we have

$$(A \cdot B)^{\top} = B^{\top} \cdot A^{\top}.$$

2. The matrix product $\mathbf{w}^{\top} \cdot X^{\top} \cdot \mathbf{y}$ is a real number. The transpose r^{\top} of a real number r is the number itself, i.e. $r^{\top} = r$ for all $r \in \mathbb{R}$. Therefore, we have

$$\mathbf{w}^\top \cdot X^\top \cdot \mathbf{y} = \left(\mathbf{w}^\top \cdot X^\top \cdot \mathbf{y}\right)^\top = \mathbf{y}^\top \cdot X \cdot \mathbf{w}.$$

Hence we have shown that

$$MSE(\mathbf{w}) = \frac{1}{m-1} \cdot \left(\mathbf{w}^{\top} \cdot \left(X^{\top} \cdot X \right) \cdot \mathbf{w} - 2 \cdot \mathbf{y}^{\top} \cdot X \cdot \mathbf{w} + \mathbf{y}^{\top} \cdot \mathbf{y} \right)$$
(6.9)

holds. The matrix $X^{\top} \cdot X$ used in the first term is symmetric because

$$(X^{\top} \cdot X)^{\top} = X^{\top} \cdot (X^{\top})^{\top} = X^{\top} \cdot X.$$

Using the results from the previous section, i.e.

$$\nabla(\mathbf{x} \mapsto \mathbf{x}^{\top} \cdot C \cdot \mathbf{x}) = 2 \cdot C \cdot \mathbf{x}$$
 provided that $C^{\top} = C$ and $\nabla(\mathbf{x} \mapsto \mathbf{b}^{\top} \cdot A \cdot \mathbf{x}) = A^{\top} \cdot \mathbf{b}$,

we can now compute the gradient of $MSE(\mathbf{w})$ with respect to \mathbf{w} . The gradient of the term

$$\mathbf{w}^{\top} \cdot (X^{\top} \cdot X) \cdot \mathbf{w}$$

with respect to \mathbf{w} is $2 \cdot (X^{\top} \cdot X) \cdot \mathbf{w}$, while the gradient of the term

$$\mathbf{y}^{\top} \cdot X \cdot \mathbf{w}$$

is $X^{\top} \cdot \mathbf{y}$. Since the gradient of $\mathbf{y}^{\top} \cdot \mathbf{y}$ with respect to \mathbf{w} vanishes, the gradient of $MSE(\mathbf{w})$ is given as

$$\nabla \mathtt{MSE}(\mathbf{w}) = \frac{2}{m-1} \cdot \left(X^{\top} \cdot X \cdot \mathbf{w} - X^{\top} \cdot \mathbf{y} \right).$$

If the squared error $MSE(\mathbf{w})$ has a minimum for the weights \mathbf{w} , then we must have

$$\nabla \mathtt{MSE}(\mathbf{w}) = \mathbf{0}.$$

This leads to the equation

$$\frac{2}{m-1} \cdot \left(\boldsymbol{X}^{\top} \cdot \boldsymbol{X} \cdot \mathbf{w} - \boldsymbol{X}^{\top} \cdot \mathbf{y} \right) = \mathbf{0}.$$

This equation can be rewritten as

$$(6.10)$$

and is known as the normal equation.

Remark: Although the matrix $X^{\top} \cdot X$ will often be invertible, for numerical reasons it is not advisable to rewrite the normal equation as

$$\mathbf{w} = (X^{\top} \cdot X)^{-1} \cdot X^{\top} \cdot \mathbf{y}.$$

Instead, when solving the normal equation we will use the *Python* function numpy.linalg.solve(A, b), which takes a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ and solves the equation

$$A \cdot \mathbf{x} = \mathbf{b}$$
.

6.2.3 Implementation

Figure 6.4 on page 144 shows an implementation of general linear regression. The procedure

linear_regression(fileName, target, explaining, f)

takes four arguments:

- 1. fileName is a string that is interpreted as the name of a Csv file containing the data.
- 2. target is an integer that specifies the column that contains the dependent variable.

- 3. **explaining** is a list of integers. These integers specify the columns of the Csv file that contain the independent variables. These are also called the **explaining variables**.
- 4. f is a function that takes one floating point argument and outputs one floating point function. This function is used to modify the dependent variable.

Later, when we call the function linear_regression to investigate the fuel consumption, we will use the function

$$x \mapsto \frac{1}{x}$$

to transform the variable *miles per gallon* into a variable expressing the fuel consumption. The reason is that there is reciprocal relation between the number of miles that a car drives on one gallon of gasoline and the fuel consumption: If you drive only a few miles with one gallon of gas, then your fuel consumption is high.

```
import csv
1
     import numpy as np
2
3
     def linear_regression(fileName, target, explaining, f):
4
         with open(fileName) as input_file:
5
                         = csv.reader(input_file, delimiter=',')
             reader
6
             line_count = 0
                         = []
             goal
8
                         = []
             Causes
9
             for row in reader:
10
                  if line_count != 0:
11
                      goal .append(f(float(row[target])))
12
                      Causes.append([float(row[i]) for i in explaining] + [1.0])
                  line_count += 1
         m = len(goal)
15
         X = np.array(Causes)
16
         y = np.array(goal)
17
         w = np.linalg.solve(X.T @ X, X.T @ y)
18
               = np.sum((X @ w - y) ** 2)
19
         yMean = np.sum(y) / m
20
                = sum((y - yMean) ** 2)
         TSS
21
                = 1 - RSS / TSS
         R2
22
         return R2
23
24
     def main():
25
         explaining = [1, 2, 3, 4, 5, 6]
26
         R2 = linear_regression("cars.csv", 0, explaining, lambda x: 1/x)
27
         print(f'portion of explained variance : {R2}')
```

Figure 6.4: General linear regression.

The function linear_regression works as follows:

- 1. It reads the specified Csv file line by line and stores the data in the variables goal and Causes. This is done by creating a csv reader in line 6. This reader returns the entries in the specified input file line by line in the for loop in line 10.
 - (a) goal is a list containing the data of the dependent variable that was specified by target. This list is initialized in line 8. It is filled with data in line 12. Note that the values stored in goal are transformed by the function f.
 - (b) Causes is a list of lists containing the data of the explaining variables. Every row in the Csv file corresponds to one list in the list Causes. Note also that we append the number 1.0 to each of these lists. This corresponds to adding a constant feature to our data and it enables us to use the normal equations as we have derived them.
- 2. m is the number of data pairs and is computed in line 15.
- 3. Causes is transformed into the NumPy matrix X in line 16.
- 4. goal is transformed into the NumPy array y in line 17.
- 5. The normal equation $(X^{\top} \cdot X) \cdot \mathbf{w} = X^{\top} \cdot \mathbf{y}$ is formulated and solved using the function np.linalg.solve in line 18.
 - Note that X.T is the transpose of the matrix X. The operator @ computes the matrix product. Hence the expression X.T @ X is interpreted as $X^{\top} \cdot X$. Similarly, the expression X.T @ y is interpreted as $X^{\top} \cdot y$.
- 6. The expression (X @ w y) is the difference between the predictions of the linear model and the observed values y. By squaring it and the summing over all entries of the resulting vector we compute is the residual sum of squares RSS in line 19.
- 7. yMean is the mean value of the variable y.
- 8. TSS is the total sum of squares.
- 9. R2 is the proportion of the explained variance.

When we run the program shown in Figure 6.4 on page 144 with the data stored in cars.csv, which had been discussed previously, then the proportion of explained variance is 88%. Considering that our data does not take the aerodynamics of the cars into take account, this seems like a reasonable result. A Jupyter notebook containing a similar program is available at

```
https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/Linear-Regression.ipynb.
```

6.3 Polynomial Regression

Sometimes the model of linear regression is not flexible enough. One way to extend it is to add higher order features. For example, assume that we have two features x_1 and x_2 and that there is a dependent variable y that, can be computed from x_1 and x_2 but that depends on x_1 and x_2 in a non-linear way. Then, instead of having a feature matrix that just contains x_1 and x_2 we can extend the feature matrix by adding non-linear features like

$$x_1^2$$
, x_2^2 , and $x_1 \cdot x_2$.

Obviously, a model that is a linear combination of x_1 , x_2 , x_1^2 , x_2^2 , and $x_1 \cdot x_2$ is much more powerful

than a model that only depends on x_1 and x_2 . An example will make things clearer. Assume we have a dependent variable y that can be computed from an independent variable x_1 via the equation

$$y=\sqrt{x_1}$$
.

Furthermore, let us assume that in our data set there is a second independent variable x_2 which is strongly linearly correlated to x_1 .¹ In our toy example we will assume that x_2 is x_1 plus some small random noise. Figure 6.5 on page 146 shows a *Python* script that can be used to investigate this situation.

```
import numpy
                                  as np
     import sklearn.linear_model as lm
                                  # number of data points
    N = 20
    X1 = np.array([k for k in range(N)])
    X2 = np.array([k + 0.2 * (np.random.rand() - 0.5) for k in range(N)])
    Y = np.sqrt(X1)
                                  # Y is the square root of X1
    X1 = np.reshape(X1, (N, 1)) # turn X1 into an N x 1 matrix
    X2 = np.reshape(X2, (N, 1)) # turn X2 into an N x 1 matrix
    X = np.hstack([X1, X2])
                                  # combine X1 and X2 into an N x 2 matrix
10
11
    from sklearn.model_selection import train_test_split
12
13
    X_train, X_test, Y_train, Y_test = \
14
         train_test_split(X, Y, test_size=0.2, random_state=1)
15
16
    def linear_regression(X_train, Y_train, X_test, Y_test):
17
         M = lm.LinearRegression()
18
        M.fit(X_train, Y_train)
19
         train_score = M.score(X_train, Y_train)
20
         test_score = M.score(X_test , Y_test)
21
         return M, train_score, test_score
22
23
    M, train_score, test_score = linear_regression(X_train, Y_train, X_test, Y_test)
24
25
    from sklearn.preprocessing import PolynomialFeatures
26
27
    quadratic = PolynomialFeatures(2, include_bias=False)
28
    X_train_quadratic = quadratic.fit_transform(X_train)
29
    X_test_quadratic = quadratic.fit_transform(X_test)
30
    quadratic.get_feature_names(['x1', 'x2'])
31
32
    M, s1, s2 = \setminus
33
         linear_regression(X_train_quadratic, Y_train, X_test_quadratic, Y_test)
34
```

Figure 6.5: Polynomial Regression

¹We add this second variable in order be able to show the effect of overfitting.

- 1. In line 4 we set N to 20. This is the number of data points that we will generate.
- 2. The vector X1 contains the numbers form 0 up to n-1.
- 3. The vector X2 contains the same numbers but perturbed by some random noise. The function numpy.random.rand() returns a [0, 1.0]. Therefore, the expression

is a random number in the interval [-0.1, +0.1].

- 4. The dependent variable Y is the computed as the square root of the numbers in X1.
- 5. Next, we reshape these vectors and stack them into the design matrix X.
- 6. In line 14, we split the data into a training set and a test set. We will not use the test data set for training, but will only use it to evaluate our results in the end. This way we make sure that our model is able to generalize from the data it has seen.
 - In a real world situation, this dataset would be too small to be split into a training set and a test set. However, in a real world situation we would have more data and I wanted to keep this example simple.
- 7. Figure 6.6 on page 148 shows the data. The blue dots are plotted at the location (x_1, y) , while the yellow dots are plotted at the location (x_2, y) .
- 8. The function linear_regression takes both the training data and the test data. It uses the training data to build a linear model. Then it uses the test data to compute the proportion of explained variance.
- 9. Line 24 trains a linear model from the training data and evaluates this model with the help of the test data. The proportion of the explained variance is 93% for the training set, but only 88% for the test set.
 - Figure 6.7 on page 149 shows the linear model that is computed by this function.
- 10. In order to improve our model, we add second order features in line 28 and line 29. These lines augment the design matrix with columns that contain x_1^2 , $x_1 \cdot x_2$, and x_2^2 .
- 11. When we train a regression model with this new design matrix, we get an explained variance of 98% on the training set and 96% on the test set. Figure 6.8 on page 150 shows the resulting model.

Having tried second order terms, we are tempted to go further and try all terms up to the fourth order. If we do this, the proportion of explained variance on the training set improves to 99.99%. However, on the test set we get a proportion of explained variance of 96% which, although an improvement, is much worse than the explained variance on the training set. Figure 6.9 on page 151 shows the corresponding model.

If we get even bolder and try all terms up to the sixth order, the proportion of explained variance on the training set improves to 100%. However, on the test set we the proportion of explained variance drops of 86.5%. This is called overfitting: Our model is now so complex that it can essentially remember all the training examples it has seen. However, it is unable to generalize from these training examples. Figure 6.10 on page 152 shows the corresponding model. We see that regression curve gets wiggly, which is also an indication that we are experiencing overfitting.



Figure 6.6: $y = \sqrt{x_1}, x_2 = x_1 + \text{noise}.$

6.4 Ridge Regression

In linear regression, our goal is to minimize the mean squared error that has been defined as

$$\mathrm{MSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \sum_{i=1}^{m} \left((\mathbf{x}^{(i)})^{\top} \cdot \mathbf{w} - y^{(i)} \right)^{2}.$$

The previous example has shown that sometimes minimizing the mean squared error leads to overfitting. One way to avoid overfitting is called ridge-regression. Instead of minimizing the mean squared error, we instead minimize the expression

$$\mathtt{RegMSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \sum_{i=1}^m \Bigl(\bigl(\mathbf{x}^{(i)}\bigr)^\top \cdot \mathbf{w} - y^{(i)} \Bigr)^2 + \lambda \cdot \mathbf{w}^2.$$

Here, the parameter λ is called the regularization parameter. If λ is not too small, then minimizing RegMSE(w) forces us to find parameters w that are small. This prevents the model from getting to complex and this prevents over-fitting. Figure 6.11 on page 153 show the model we get when setting $\lambda = 0.05$. In this case the proportion of explained variance is 99.97% on the training set and 99.78% on the test set. This clearly shows that the model does generalize to example that it has not seen.

The regularization parameter λ is a so called hyper parameter that is usually determined by cross validation. This sound complicated, but actually is very simple: We split the training set into k parts. Then we remove one of these parts and train the model on the remaining k-1 parts. We use the



Figure 6.7: Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$.

removed part to try to find the best value for our hyper parameter by trial and error. We do this for all of the k parts. In each case we might find a sightly different optimal value λ_k . Finally, we take the mean $\bar{\lambda}$ of all values λ_k that we have found in our k experiments. Following that we use the complete training set one last time to find a model that uses the value $\bar{\lambda}$. It is important to keep the test set untouched while searching for the best value of λ .



Figure 6.8: Second Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$.

Exercise 15: The file "trees.csv", which is available at

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/trees.csv, contains data about 31 lovely cherry trees from the Allegheny National Forest in Pennsylvania that have fallen prey to a chainsaw massacre. I have taken this data from

http://www.statsci.org/data/general/cherry.txt.

- 1. The first column of this Csv file contains the diameter of these trees at a height of 54 inches above the ground.
- 2. The second column lists the heights of these trees in foot.
- 3. The third column list the volume of wood that has been harvested from these trees. This volume is given in cubic inches.

Try to derive a model that estimates the volume of the trees from the diameter and the height. \diamond

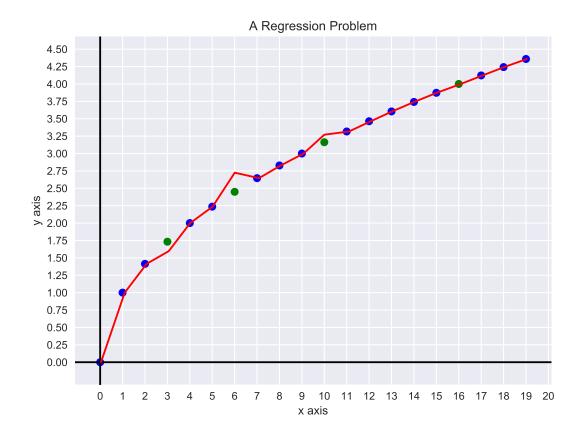


Figure 6.9: Fourth Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

Exercise 16: The file "nba.csv", which is available at

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/nba.csv, contains various data about professional basket ball players.

- 1. The first column gives the name of the player.
- 2. The second column specifies the position of the player.
- 3. The third column lists the height of the player.
- 4. The fourth column contains the weight of the player.
- 5. The fifth column shows the age of each player.

To what extend can you predict the weight of a player given his height and his age?

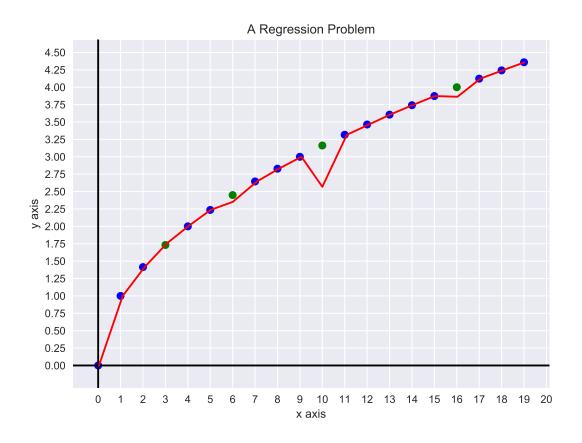


Figure 6.10: Sixth Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$.



Figure 6.11: Sixth Order Ridge Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$.

Chapter 7

Classification

One of the earliest application of artificial intelligence is classification. A good example of classification is spam detection. A system for spam detection classifies an email as either spam or not spam, where "not spam" is often abbreviated as "ham". To do so, it first computes various features of the email and then uses these features to determine whether the email is likely to be spam. For example, a possible feature would be the number of occurrences of the word "pharmacy" in the text of the email.

Another famous example of classification is character recognition. In character recognition, the input is an image depicting a character. This image is usually coded as a vector of grey values. The task is then to recognize the letter shown. While spam detection is a binary classification problem, character recognition is a multi-class classification problem, since there are 26 different characters.

7.1 Introduction

Formally, the classification problem in machine learning can be stated as follows: We are given a set of objects $S := \{o_1, \dots, o_n\}$ and a set of classes $C := \{c_1, \dots, c_k\}$. Furthermore, there exists a function

$$\mathtt{classify}: S \to C$$

that assigns a class classify(o) to every object $o \in S$. The set S is called the sample space. In the example of spam detection, the sample space S is the set of all emails that we might receive, i.e. S is the set of all strings, while the set of classes C is given as

$$C = \{ \mathtt{spam}, \mathtt{ham} \}.$$

Our goal is to compute the function classify. In order to do this, we use an approach known as supervised learning: We take a subset $S_{train} \subseteq S$ of emails where we already know whether the emails are spam or not. This set S_{train} is called the training set. Next, we define a set of D features for every $o \in S$. These features have to be computable, i.e. we must have a function

feature:
$$S \times \{1, \dots, D\} \to \mathbb{R}$$

such that feature(o, j) computes the j-th feature and we have to be able to implement this function with reasonable efficiency. In general, the values of the features are real values. However, there are cases where these values are just Booleans. If

$$feature(o, j) \in \mathbb{B}$$
 for all $o \in S$,

then the j-th feature is called a binary feature. If we encode false as -1 and true as +1, then the set of Boolean values \mathbb{B} can be considered a subset of \mathbb{R} and hence Boolean features can be considered as real numbers. For example, in the case of spam detection, the first feature could be the occurrence of the string "pharmacy". In this case, we would have

$$\mathtt{feature}(o,1) := \left\{ \begin{array}{ll} +1 & \mathrm{if} \ \mathtt{pharmacy} \in o, \\ -1 & \mathrm{if} \ \mathtt{pharmacy} \not \in o, \end{array} \right.$$

i.e. the first feature would be to check whether the email o contains the string "pharmacy". If we want to be more precise, we can instead define the first feature as

$$feature(o, 1) := count("pharmacy", o),$$

i.e. we would count the number of occurrences of the string "pharmacy" in our email o. As the value of

is always a natural number, in this case the first feature would be a discrete feature. However, we can be even more precise: Instead of just counting the number of occurrences of "pharmacy" we can compute its frequency. After all, there is a difference whether the string "pharmacy" occurs once in an email containing but a hundred characters or whether is occurs once in an email with a length of several thousand characters. To this end, we would then define the first feature as

$$\mathtt{feature}(o,1) := \frac{\mathtt{count}(\mathtt{"pharmacy"},o)}{\mathtt{len}(o)},$$

where len(o) defines the number of characters in the string o. In this case, the first feature would be a continuous feature and as this is the most general case, unless stated otherwise, we deal with the continuous case.

Having defined the features, we next need a model of the function classify that tries to approximate the function classify via the features. This model is given by a function

$${\tt model}: \mathbb{R}^D \to C$$

such that

$$model(feature(o, 1), \dots, feature(o, D)) \approx classify(o).$$

Using the function model, we can then approximate the function classify using a function guess that is defined as

$$guess(o) := model(feature(o, 1), \cdots, feature(o, D))$$

Most of the time, the function guess will only approximate the function classify, i.e. we will have

$$guess(o) = classify(o)$$

for most objects of $o \in S$ but not for all of them. The accuracy of our model is defined as the fraction of those objects that are classified correctly, i.e.

$$\mathtt{accuracy} := \frac{\mathtt{card} \big(\{o \in S \mid \mathtt{guess}(o) = \mathtt{classify}(o) \} \big)}{\mathtt{card}(S)}.$$

To keep matters simple, we will assume that the sample space is finite.

The function model is usually determined by a set of parameters or weights \mathbf{w} . In this case, we have

$$model(x) = model(x, w)$$

where \mathbf{x} is the vector of features, while \mathbf{w} is the vector of weights. Later, when we introduce logistic regression, we will assume that the number of weights is one more than the number of features. Then, the weights specify the relative importance of the different features. Furthermore, there will be a weight that is interpreted as a bias term.

When it comes to the choice of model, it is important to understand that, at least in practical

applications, <u>all</u> models are wrong. Nevertheless, <u>some</u> models are useful. There are two reasons for this:

- 1. We do not fully understand the function classify that we want to approximate by the function model.
- 2. In the most general setting, the function classify is so complex, that even if we could compute it exactly, the resulting model would be much too complicated.

The situation is similar in physics: Let us assume that we intend to model the fall of an object. A model that is a hundred percent accurate would have to include the following forces:

- (a) gravitational acceleration,
- (b) air friction,
- (c) tidal forces, i.e. the effects that the rotation of the earth has on moving objects,
- (d) celestial forces, i.e. the gravitational acceleration caused by celestial objects like the moon or the sun.
- (e) In the case of a metallic object we have to be aware of the magnetic forces caused by the geomagnetic field.
- (f) To be fully accurate, we might have to include corrections from relativistic physics and even quantum physics.
- (g) As physics is not a closed subject, there might be other forces at work which we still do not know of.

Hence, a correct model would be so complicated that it would be unmanageable and therefore useless. Let us summarize our introductory discussion of machine learning in general and classification in particular. A set S of objects and a set C of classes are given. Our goal is to approximate a function

$$classify: S \rightarrow C$$

using certain features of our objects. The function classify is then approximated using a function model as follows:

$$model(feature(o, 1), \dots, feature(o, D), \mathbf{w}) \approx classify(o).$$

The model depends on a vector of parameters \mathbf{w} . In order to learn these parameters, we are given a training set S_{train} that is a subset of S. As we are dealing with supervised learning, the function classify is known for all objects $o \in S_{train}$. Our goal is to determine the parameters \mathbf{w} such that the number of mistakes we make on the training set is minimized.

7.1.1 Notation

We conclude this introductory section by fixing some notation. Let us assume that the objects $o \in S_{train}$ are numbered from 1 to N, while the features are numbered from 1 to D. Then we define

1. $\mathbf{x}_i := \langle \mathtt{feature}(o_i, 1), \cdots, \mathtt{feature}(o_i, D) \rangle^\top$ for all $i \in \{1, \cdots, N\}$.

i.e. \mathbf{x}_i is a D-dimensional column vector that collects the features of the *i*-th training object.

2. $x_{i,j} := \mathtt{feature}(o_i, j)$ for all $i \in \{1, \dots, N\}$ and $j \in \{1, \dots, D\}$.

i.e. $x_{i,j}$ is the j-th feature of the i-th object. The numbers $x_i i, j$ are combined into the feature matrix X, i.e. we have

$$X = (x_{i,j})_{\substack{i \in \{1, \dots, N\} \\ j \in \{1, \dots, D\}.}}$$

3. $y_i := classify(o_i)$ for all $i \in \{1, \dots, N\}$

i.e. y_i is the class of the *i*-th object. These number are collected into the N-dimensional column vector \mathbf{y} .

7.1.2 Applications of Classification

Besides spam detection, there are many other classification problems that can be solved using machine learning. To give just one more example, imagine a general practitioner that receives a patient and examines her symptoms. In this case, the symptoms can be seen as the features of the patient. For example, these features could be

- (a) body temperature,
- (b) blood pressure,
- (c) heart rate,
- (d) body weight,
- (e) breathing difficulties,
- (f) age,

to name but a few of the possible features. Based on these symptoms, the general practitioner would then decide on an illness, i.e. the set of classes for the classification problem would be

$$\{commonCold, pneumonia, asthma, flu, Covid-19, \dots, unknown\}.$$

Hence, the task of disease diagnosis is a classification problem. This was one of the earliest problems that was tackled by artificial intelligence. As of today, computer-aided diagnosis and clinical decision support systems have been used for more than 40 years in many hospitals. Today, there are a number of diseases that can be diagnosed more accurately by a computer than by a specialist. One such example is the diagnosis of heart disease. Other applications of classification are the following:

- (a) image recognition,
- (b) speech recognition,
- (c) credit card fraud detection,
- (d) credit approval.

7.2 Digression: The Method of Gradient Ascent

In machine learning, it is often the case that we have to find either the maximum or the minimum of a function

$$f: \mathbb{R}^n \to \mathbb{R}$$
.

For example, when we discuss logistic regression in the next section, we will have to find the maximum of the likelihood function. To proceed, let us introduce the arg max function. The idea is that

$$\widehat{\mathbf{x}} = \arg\max_{\mathbf{x} \in \mathbb{R}^n} f$$

is that value of $\mathbf{x} \in \mathbb{R}^n$ that maximizes $f(\mathbf{x})$. Formally, we have

$$\forall \mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) \le f \Big(\arg \max_{\mathbf{x} \in \mathbb{R}^n} f \Big).$$

Of course, the expression $\arg\max_{\mathbf{x}\in\mathbb{R}^n}f$ is only defined when the maximum of f is unique. If the function f is differentiable, we know that a necessary condition for a vector $\hat{\mathbf{x}}\in\mathbb{R}^n$ to satisfy

$$\hat{\mathbf{x}} = \arg \max_{\mathbf{x} \in \mathbb{R}^n} f$$
 is that we must have $\nabla f(\hat{\mathbf{x}}) = \mathbf{0}$,

i.e. the gradient of f, which we will write as ∇f , vanishes at the maximum $\hat{\mathbf{x}}$.

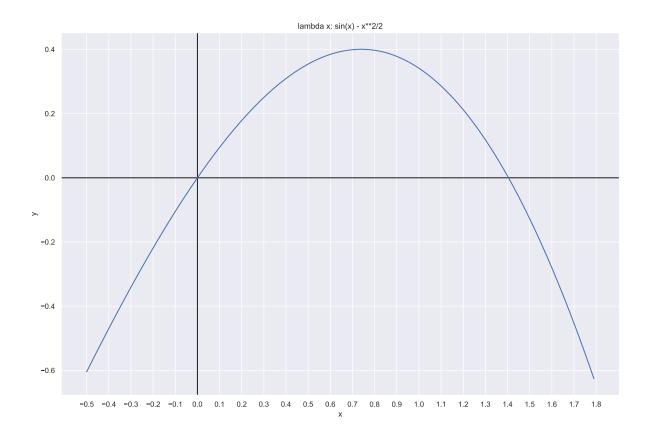


Figure 7.1: The function $x \mapsto \sin(x) - \frac{1}{2} \cdot x^2$.

Remember that the gradient of the function f is defined as the column vector

$$\nabla f := \left\langle \frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n} \right\rangle^{\top}.$$

Unfortunately, in many cases the equation

$$\nabla f(\widehat{\mathbf{x}}) = \mathbf{0}$$

cannot be solved in closed terms. This is already true in the one-dimensional case, i.e. if n = 1. For example, consider the function $f : \mathbb{R} \to \mathbb{R}$ that is defined as

$$f(x) := \sin(x) - \frac{1}{2} \cdot x^2.$$

This function is shown in Figure 7.1 on page 158. From the graph of the function it is obvious that this function has a maximum somewhere between 0.6 and 0.8. In order to compute this maximum, we can compute the derivative of f. This derivative is given as

$$f'(x) = \cos(x) - x$$

As it happens, the equation $\cos(x) - x = 0$ does not seem to have a solution in closed form. Hence, we can only approximate the solution numerically via a sequence of numbers $(x_n)_{n \in \mathbb{N}}$ such that the limit $\lim_{n \to \infty} x_n$ exists and is a solution of the equation $\cos(x) - x = 0$, i.e. we want to have

$$\cos\left(\lim_{n\to\infty}x_n\right) = \lim_{n\to\infty}x_n.$$

The method of gradient ascent is a numerical method that can be used to find the maximum of a function

$$f: \mathbb{R}^n \to \mathbb{R}$$

numerically. The basic idea is to take a vector $\mathbf{x}_0 \in \mathbb{R}^n$ as the start value and define a sequence of vectors $(\mathbf{x}_n)_{n \in \mathbb{N}}$ such that we have

$$f(\mathbf{x}_{n+1}) \ge f(\mathbf{x}_n)$$
 for all $n \in \mathbb{N}$.

Hopefully, this sequence will converge against $\hat{\mathbf{x}} = \arg \max_{\mathbf{x} \in \mathbb{R}^n} f$. If we do not really know where to start our search, we define $\mathbf{x}_0 := \mathbf{0}$. In order to compute \mathbf{x}_{n+1} given \mathbf{x}_n , the idea is to move from \mathbf{x}_n in that direction where we have the biggest change in the values of f. This direction happens to be the gradient of f at \mathbf{x}_n . Therefore, the definition of \mathbf{x}_{n+1} is given as follows:

$$\mathbf{x}_{n+1} := \mathbf{x}_n + \alpha \cdot \nabla f(\mathbf{x}_n)$$
 for all $n \in \mathbb{N}_0$.

Here, α is called the step size and is also known as the learning rate. It determines by how much we move in the direction of the gradient. In practise, it is best to adapt the step size dynamically during the iterations. Figure 7.2 on page 161 shows how this is done. The function findMaximum takes four arguments:

- 1. f is the function that is to be maximized. It is assumed that f takes a vector $\mathbf{x} \in \mathbb{R}^n$ as its input and that it returns a real number. Note that n might be 1. In that case the input to f is a real number.
- 2. gradF is the gradient of f. It takes a vector $\mathbf{x} \in \mathbb{R}^n$ as its input and returns the vector $\nabla \mathbf{f}(\mathbf{x})$.
- 3. **start** is a vector from \mathbb{R}^n that is used as the value of \mathbf{x}_0 . In practice, we will often use $\mathbf{0} \in \mathbb{R}^n$ as the start vector.
- 4. eps is the precision that we need for the maximum. We will have to say more on how eps is related to the precision later. As we are using double precision floating point arithmetic, it won't make sense to use a value for eps that is smaller than 10^{-15} .

Next, let us discuss the implementation of gradient ascent.

1. x is initialized with the parameter start. Hence, start is really the same as \mathbf{x}_0 .

- 2. fx is the value f(x).
- 3. alpha is the learning rate. We initialize alpha as 1.0. The learning rate will be adapted dynamically.
- 4. The body of the while loop starting in line 6 executes one iteration of gradient ascent.
- 5. In each iteration, we store the values of \mathbf{x}_n and $f(\mathbf{x}_n)$ in the variables **x0ld** and **f0ld**. This is needed since we need to ensure that the values of $f(\mathbf{x}_n)$ are increasing. If this value of $f(\mathbf{x}_{n+1})$ is not bigger that $f(\mathbf{x}_n)$ we revert to the old values.
- 6. Next, we compute \mathbf{x}_{n+1} in line 8 using the formula

$$\mathbf{x}_{n+1} := \mathbf{x}_n + \alpha \cdot \nabla f(\mathbf{x}_n).$$

- 7. The corresponding value $f(\mathbf{x}_{n+1})$ is computed in line 9.
- 8. If we are unlucky, $f(\mathbf{x}_{n+1})$ is smaller than $f(\mathbf{x}_n)$ instead of bigger. This might happen if the learning rate α is too large. Hence, in this case we decrease the value of α , discard both \mathbf{x}_{n+1} and $f(\mathbf{x}_{n+1})$ and start over again via the **continue** statement in line 13.
- 9. Otherwise, if $f(\mathbf{x}_{n+1})$ is indeed bigger than $f(\mathbf{x}_n)$, the vector \mathbf{x}_{n+1} is a better approximation of the maximum than the vector \mathbf{x}_n . In this case, in order to increase the speed of the convergence of our algorithm we will then increase the learning rate α by 20%.
- 10. The idea of our implementation is to stop the iteration when the relative difference of $f(\mathbf{x}_{n+1})$ and $f(\mathbf{x}_n)$ is less than ε or, to be more precise, if

$$f(\mathbf{x}_{n+1}) < f(\mathbf{x}_n) \cdot (1+\varepsilon).$$

As the sequence $(f(\mathbf{x}_n)_{n\in\mathbb{N}})$ will be monotonically increasing, i.e. we have

$$f(\mathbf{x}_{n+1}) \ge f(\mathbf{x}_n)$$
 for all $n \in \mathbb{N}$,

the condition given above is sufficient. Now, if the increment of $f(\mathbf{x}_{n+1})$ is less than $f(\mathbf{x}_n) \cdot (1+\varepsilon)$ we assume that we have reached the maximum with the required precision. In this case we return both the value of \mathbf{x} and the corresponding function value $f(\mathbf{x})$.

```
def findMaximum(f, gradF, start, eps):
                = start
2
         fx
                = f(x)
3
         alpha = 1.0
4
                = 1 # number of iterations
5
         while True:
6
              x0ld, f0ld = x, fx
              x += alpha * gradF(x)
              fx = f(x)
              if fx <= f0ld:
10
                  alpha *= 0.5
11
                  x, fx = x01d, f01d
12
                  continue
13
              else:
14
                  alpha *= 1.2
15
              if abs(fx - fOld) \le abs(fx) * eps:
16
                  return x, fx
17
              cnt += 1
18
```

Figure 7.2: The gradient ascent algorithm.

The implementation of gradient ascent given above is not the most sophisticated variant of this algorithm. Furthermore, there are algorithms that are more powerful than gradient ascent. The first of these methods is the conjugate gradient method. A refinement of this method is the BFGS-algorithm that has been invented by Broyden, Fletcher, Goldfarb, and Shanno. Unfortunately, we do not have the time to discuss these algorithms. However, our implementation of gradient ascent is sufficient for our applications and as this is not a course on numerical analysis but rather on artificial intelligence we will not delve deeper into this topic but, instead, we refer readers interested in more efficient algorithms to the literature [Sny05]. If you ever need to find the maximum of a function numerically, you should try to use a predefined library routine that implements a state of the art algorithm. For example, in Python the method minimize from the package scipy.optimize offers various algorithms for minimization.

7.3 Logistic Regression

If we have a model such that

```
model(x, w) \approx classify(x)
```

holds, then we want to choose the weight vector \mathbf{w} in a way such that the accuracy

$$\mathtt{accuracy}(\mathbf{w}) := \frac{\mathtt{card}\big(\big\{\mathbf{o} \in S \; \big| \; \mathtt{model}(\mathtt{feature}\big(\mathbf{o}), \mathbf{w}\big) = \mathtt{classify}(\mathbf{o})\big\}\big)}{\mathtt{card}(S)}$$

is maximized. However, there is a snag: The accuracy is not a smooth function of the weight vector \mathbf{w} . It can't be a smooth function because the number of errors of our model is a natural number and not a real number that could change smoothly when the weight vector \mathbf{w} is changed. Hence, the accuracy is not differentiable as a function of the weight vector. The way to proceed is to work with probabilities

instead. Instead of assigning a class to an object \mathbf{o} we rather assign a probability p to the object \mathbf{o} that measures how probable it is that object \mathbf{o} has a given class c. Then we try to maximize this probability. In logistic regression we use a linear model that is combined with the sigmoid function. Before we can discuss the details of logistic regression we need to define this function and state some of its properties.

7.3.1 The Sigmoid Function



Figure 7.3: The sigmoid function.

Definition 37 (Sigmoid Function) The sigmoid function $S: \mathbb{R} \to [0,1]$ is defined as

$$S(t) = \frac{1}{1 + \exp(-t)}.$$

Figure 7.3 on page 162 shows the sigmoid function. The sigmoid function is also known as the logistic function.

Let us note some immediate consequences of the definition of the sigmoid function. As we have

$$\lim_{x\to -\infty} \exp(-x) = \infty, \quad \lim_{x\to +\infty} \exp(-x) = 0, \quad \text{and} \quad \lim_{x\to \infty} \frac{1}{x} = 0,$$

the sigmoid function has the following properties:

$$\lim_{t\to -\infty} S(t) = 0 \quad \text{ and } \quad \lim_{t\to +\infty} S(t) = 1.$$

As the sigmoid function is monotonically increasing, this shows that indeed

$$0 \le S(t) \le 1$$
 for all $t \in \mathbb{R}$.

Therefore, the value of the sigmoid function can be interpreted as a probability. Another important

property of the sigmoid function is its symmetry. Figure 7.3 shows that if the sigmoid function is shifted down by $\frac{1}{2}$, the resulting function is centrally symmetric, i.e. we have

$$S(-t) - \frac{1}{2} = -\left(S(t) - \frac{1}{2}\right).$$

Adding $\frac{1}{2}$ on both sides of this equation shows that this is equivalent to the equation

$$S(-t) = 1 - S(t),$$

The proof of this fact runs as follows:

$$1 - S(t) = 1 - \frac{1}{1 + \exp(-t)}$$
 (by definition of $S(t)$)
$$= \frac{1 + \exp(-t) - 1}{1 + \exp(-t)}$$
 (common denominator)
$$= \frac{\exp(-t)}{1 + \exp(-t)}$$

$$= \frac{1}{\exp(t) + 1}$$
 (expand fraction by $\exp(t)$)
$$= \frac{1}{1 + \exp(+t)}$$

$$= S(-t)$$
 (by definition of $S(-t)$).

The exponential function can be expressed via the sigmoid function. Let us start with the definition of the sigmoid function.

$$S(t) = \frac{1}{1 + \exp(-t)}$$

Multiplying this equation with the denominator yields

$$S(t) \cdot (1 + \exp(-t)) = 1.$$

Dividing both sides by S(t) gives:

$$1 + \exp(-t) = \frac{1}{S(t)}$$

$$\Leftrightarrow \exp(-t) = \frac{1}{S(t)} - 1$$

$$\Leftrightarrow \exp(-t) = \frac{1 - S(t)}{S(t)}$$

We highlight this formula, as we need it later

$$\exp(-t) = \frac{1 - S(t)}{S(t)}.$$
 (7.1)

If we take the reciprocal of both sides of this equation, we have

$$\exp(t) = \frac{S(t)}{1 - S(t)}.$$

Applying the natural logarithm on both sides of this equation yields

$$t = \ln\left(\frac{S(t)}{1 - S(t)}\right).$$

This shows that the inverse of the sigmoid function is given as

$$S^{-1}(y) = \ln\left(\frac{y}{1-y}\right).$$

This function is known as the logit function. Next, let us compute the derivative of S(t), i.e. $S'(t) = \frac{dS}{dt}$. We have

$$S'(t) = -\frac{-\exp(-t)}{(1 + \exp(-t))^2}$$

$$= \exp(-t) \cdot S(t)^2$$

$$= \frac{1 - S(t)}{S(t)} \cdot S(t)^2 \quad \text{(by Equation 7.1)}$$

$$= (1 - S(t)) \cdot S(t)$$

We have shown

$$S'(t) = (1 - S(t)) \cdot S(t). \tag{7.2}$$

We will later need the derivative of the natural logarithm of the logistic function. We define

$$L(t) := \ln(S(t)).$$

Then we have

$$L'(t) = \frac{S'(t)}{S(t)}$$
 (by the chain rule)
$$= \frac{(1 - S(t)) \cdot S(t)}{S(t)}$$

$$= 1 - S(t)$$

$$= S(-t)$$
 (by symmetry)

As this is our most important result, we highlight it:

$$L'(t) = S(-t)$$
 where $L(t) := \ln(S(t))$.

7.3.2 The Model of Logistic Regression

In logistic regression we deal with binary classification, i.e. we assume that we just need to decide whether a given object is a member of a given class or not. We use the following model to compute the probability that an object o with features \mathbf{x} will be of the given class:

$$P(y = +1 \mid \mathbf{x}, \mathbf{w}) = S(\mathbf{x} \cdot \mathbf{w}).$$

Note that $P(y = +1 \mid \mathbf{x}, \mathbf{w})$ is the conditional probability that o has the given class, given its features \mathbf{x} and the weights \mathbf{w} . The expression $\mathbf{x} \cdot \mathbf{w}$ denotes the dot product of the vectors \mathbf{x} and \mathbf{w} , i.e. we

have

$$\mathbf{x} \cdot \mathbf{w} = \sum_{i=1}^{D} x_i \cdot w_i.$$

To simplify calculations, it is assumed that \mathbf{x} contains a constant feature that always takes the value of 1. Seeing this model for the first time you might think that this model is not very general and that it can only be applied in very special circumstances. However, the features x_i can be functions of arbitrary complexity and hence this model is much more general than it appears on first sight.

We assume that y can only take the values +1 or -1, e.g. in the example of spam detection y = 1 if the email is spam and y = -1 otherwise. Since complementary probabilities add up to 1, we have

$$P(y = -1 \mid \mathbf{x}, \mathbf{w}) = 1 - P(y = +1 \mid \mathbf{x}, \mathbf{w}) = 1 - S(\mathbf{x} \cdot \mathbf{w}) = S(-\mathbf{x} \cdot \mathbf{w}).$$

Hence, we can combine the equations for $P(y = -1 \mid \mathbf{x}, \mathbf{w})$ and $P(y = +1 \mid \mathbf{x}, \mathbf{w})$ into a single equation

$$P(y \mid \mathbf{x}, \mathbf{w}) = S(y \cdot (\mathbf{x} \cdot \mathbf{w})).$$

Given N objects o_1, \dots, o_n with feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ and classes y_1, \dots, y_n , we want to determine the weight vector \mathbf{w} such that the likelihood $\ell(\mathbf{X}, \mathbf{y})$ of all of our observations is maximized, where \mathbf{X} is the feature matrix that is defined as

$$\mathbf{X} := \left(egin{array}{c} \left(\mathbf{x}^{(1)}
ight)^{ op} \ dots \ \left(\mathbf{x}^{(m)}
ight)^{ op} \end{array}
ight).$$

This approach is called the maximum likelihood estimation of the weights. As we assume that the probabilities of different observations are independent, the individual probabilities have to be multiplied to compute the overall likelihood $\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ of a given training set:

$$\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \prod_{i=1}^{N} P(y_i \mid \mathbf{x}_i, \mathbf{w}).$$

Since it is easier to work with sums than with products, instead of maximizing the function $\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ we instead maximize the logarithm of $\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$. This logarithm is called the log-likelihood and is defined as

$$\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) := \ln(\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})).$$

As the natural logarithm is a monotonically increasing function, the functions $\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ and $\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ take their maximum at the same value of \mathbf{w} . As we have

$$\ln(a \cdot b) = \ln(a) + \ln(b),$$

the natural logarithm of the likelihood is

$$\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^{N} \ln \Big(S\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w}) \big) \Big) = \sum_{i=1}^{N} L\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w}) \big).$$

Our goal is to choose the weights \mathbf{w} such that the likelihood is maximized. Since this is the same as maximizing the log-likelihood, we need to determine the gradient of the log-likelihood with respect to the weights w_i , i.e. we need to compute the partial derivatives

$$\frac{\partial}{\partial w_j} \ell \ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$$
 for all $j \in \{1, \dots, D\}$.

In order to compute the partial derivative of $\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ with respect to the coefficients \mathbf{w} we need to compute the partial derivative of the dot product $\mathbf{x}_i \cdot \mathbf{w}$ with respect to the weights w_i . We define

$$h(\mathbf{w}) := \mathbf{x}_i \cdot \mathbf{w} = \sum_{k=1}^D x_{i,k} \cdot w_k.$$

Then we have

$$\frac{\partial}{\partial w_j} h(\mathbf{w}) = \frac{\partial}{\partial w_j} \sum_{k=1}^D x_{i,k} \cdot w_k = \sum_{k=1}^D x_{i,k} \cdot \frac{\partial}{\partial w_j} w_k = \sum_{k=1}^D x_{i,k} \cdot \delta_{j,k} = x_{i,j}.$$

Now we are ready to compute the partial derivative of $\ell\ell(\mathbf{X},\mathbf{y},\mathbf{w})$ with respect to \mathbf{w} :

$$\frac{\partial}{\partial w_j} \ell \ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$$

$$= \frac{\partial}{\partial w_j} \sum_{i=1}^{N} L(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w}))$$

$$= \sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S(-y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})), \text{ since } \frac{\mathrm{d}L(x)}{\mathrm{d}x} = S(-x).$$

Hence, the partial derivative of the log-likelihood function is given as follows:

$$\frac{\partial}{\partial w_j} \ell \ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^N y_i \cdot x_{i,j} \cdot S(-y_i \cdot \mathbf{x}_i \cdot \mathbf{w})$$

Next, we have to find the value of \mathbf{w} such that

$$\sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S(-y_i \cdot \mathbf{x}_i \cdot \mathbf{w}) = 0 \quad \text{ for all } j \in \{1, \dots, D\}.$$

These are D equations for the D variables w_1, \dots, w_D . Due to the occurrence of the sigmoid function, these equations are nonlinear. We can not solve these equations explicitly. Nevertheless, our computation of the gradient of the log-likelihood was not in vain: We will use the method of gradient ascent to find the value of \mathbf{w} that maximizes the log-likelihood. This method has been outlined in the previous section.

7.3.3 Implementing Logistic Regression

In this section we will give a simple implementation of logistic regression. We will use our implementation of logistic regression to predict whether a student will pass or fail a given exam. Figure 7.4 shows a Csv file I have borrowed from the Wikipedia page on logistic regression that contains the data we are going to explore. Concretely, this file stores the hours a student has learned for a particular exam and the fact whether the student has passed the exam or has failed. A passed exam is encoded as the number 1, while a failed exam is encoded as 0. The first column of the file stores these numbers. The second column stores the number of hours that the student has learned in order to pass the exam.

The program shown in Figure 7.5 on page 168 implements logistic regression. As there are a number of subtle points that might easily be overlooked otherwise, we proceed to discuss this program line by line.

```
Pass, Hours
   Ο,
           0.50
   Ο,
           0.75
   Ο,
           1.00
   Ο,
           1.25
   Ο,
           1.50
   Ο,
           1.75
   1,
           1.75
   Ο,
           2.00
           2.25
   1,
10
           2.50
   Ο,
11
           2.75
   1,
12
           3.00
   Ο,
13
   1,
           3.25
14
           3.50
   Ο,
15
   1,
           4.00
16
           4.25
   1,
17
           4.50
   1,
18
           4.75
   1,
19
           5.00
   1,
20
   1,
           5.50
21
```

Figure 7.4: Results of an exam.

```
import numpy as np
2
     # compute \frac{1}{1 + \exp(-t)}
3
     def sigmoid(t):
4
         return 1.0 / (1.0 + np.exp(-t))
     # compute \ln\left(\frac{1}{1+\exp(-t)}\right) and avoid overflow
     def logSigmoid(t):
8
         if t > -100:
9
              return -np.log(1.0 + np.exp(-t))
10
          else:
11
12
              return t
13
     def 11(X, y, w):
14
          n n n
15
          given the matrix X and the observations y,
16
          return the log likelihood for the weight vector w
17
         return np.sum([logSigmoid(y[i] * (X[i] @ w)) for i in range(len(X))])
19
20
     def gradLL(X, y, w):
21
22
          Compute the gradient of the log-likelihood with respect to w
23
          11 11 11
24
         Gradient = []
          for j in range(len(X[1])):
26
              L = [y[i]*X[i][j]*sigmoid(-y[i]*(X[i]@w)) for i in range(len(X))]
27
              Gradient.append(sum(L))
28
         return np.array(Gradient)
29
```

Figure 7.5: An implementation of logistic regression.

- 1. First, we import the module numpy. This module provides us with the functions log and exp for computing the logarithm and the exponential of a number or a vector. Furthermore, we need this module because the gradient of the log-likelihood is a vector and for efficiency reasons this vector should be stored as a NumPy array.
- 2. Line 3 implements the sigmoid function

$$S(x) = \frac{1}{1 + \exp(-x)}.$$

Since we are using NumPy to compute the exponential function, the parameter t that is used in our implementation can also be a vector.

3. Line 7 starts the implementation of the natural logarithm of the sigmoid function, i.e. we implement

$$L(x) = \ln(S(X)) = \ln\left(\frac{1}{1 + \exp(-x)}\right) = -\ln(1 + \exp(-x)).$$

The implementation is more complicated than you might expect. The reason has to do with numerical overflow. Consider values of x that are smaller than, say, -1000. The problem is that the expression $\exp(1000)$ evaluates to Infinity, which represents the mathematical value ∞ . But then $1 + \exp(1000)$ is also Infinity and finally $\log(1 + \exp(1000))$ is still Infinity. However, in reality we have

$$ln(1 + exp(1000)) \approx 1000$$

because $\exp(1000)$ is so big that adding 1 to it does not make much of a difference. The precise argument works as follows:

$$\ln(1 + \exp(x)) = \ln(\exp(x) \cdot (1 + \exp(-x)))$$

$$= \ln(\exp(x)) + \ln(1 + \exp(-x))$$

$$= x + \ln(1 + \exp(-x))$$

$$\approx x + \ln(1) + \exp(-x)$$
Taylor expansion of $\ln(1 + x)$

$$= x + 0 + \exp(-x)$$

$$\approx x$$
since $\exp(-x) \approx 0$ for large x

This is the reason that logSigmoid returns x if the value of x is less than -100.

4. The function $11(\mathbf{X}, \mathbf{y}, \mathbf{w})$ defined in line 14 computes the log-likelihood of the parameter \mathbf{w} given the available data \mathbf{X} and \mathbf{y} . We have

$$\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^{N} L(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})).$$

Here L denotes the natural logarithm of the sigmoid of the argument. It is assumed that \mathbf{X} is the feature matrix. Every observation corresponds to a row in this matrix, i.e. the vector \mathbf{x}_i is the feature vector containing the features of the i-th observation. \mathbf{y} is a vector describing the outcomes, i.e. the elements of this vector are either +1 or -1. Finally, \mathbf{w} is the vector of coefficients.

5. The function gradLL(x, y, w) in line 21 computes the gradient of the log-likelihood according to the formula

$$\frac{\partial}{\partial w_j} \ell \ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^N y_i \cdot x_{i,j} \cdot S(-y_i \cdot \mathbf{x}_i \cdot \mathbf{w}).$$

The different components of this gradient are combined into a vector. The arguments are the same as the arguments to the log-likelihood.

- 6. Finally, the function logisticRegressionFile that is shown in Figure 7.6 takes one argument. This argument is the name of the Csv file containing the data that are to be analysed. The task of this function is to read the Csv file, convert the data in the feature matrix **X** and the vector **y**, and then to use the method of gradient ascent to find the weight vector **w** that maximize the likelihood. In detail this function works as follows.
 - (a) The with statement that extends from line 34 to line 43 reads the data in the file that is specified by the parameter name.
 - (b) After the data has been read, the list Pass contains a list of floating point numbers that are either 0 or 1 specifying whether the student has passed the exam, while the list Hours contains the numbers of hours that the students have spent studying.
 - (c) These data are converted into the NumPy arrays x and y in line 44 and 45.
 - (d) **n** is the number of data points we have, i.e. it is the number of students.
 - (e) We reshape the vector \mathbf{x} into a matrix \mathbf{X} in line 47. As there is only a single feature, namely the hours a student has studied, all rows of this matrix have a length of 1.
 - (f) Next, we prepend a column of ones to this matrix. This is done in line 48. This frees us from dealing explicitly with a bias term in our model.
 - (g) In logistic regression we assume that the entries of the vector \mathbf{y} are either +1 or -1. As the data provided in our input file contains 1 and 0, we need to apply a function that maps 1 to +1 and 0 to -1. The function

$$y \mapsto 2 \cdot y - 1$$

fits this job description and is applied to transform the vector y appropriately in line 49.

(h) Now we are ready to run gradient ascent. As the start vector we use a vector containing only zeros. This vector is defined in line 50. The precision we use is 10^{-8} . We want to maximize the log-likelihood of a given weight vector w. Hence we define the function f(w) as l(x, y, w) in line 52, while the gradient of this function is defined in line 53. Line 54 call the function gradient_ascent that computes the value of w that maximizes the log-likelihood.

```
import csv
     import gradient_ascent
2
3
     def logisticRegression(name):
4
         with open(name) as file:
5
             reader = csv.reader(file, delimiter=',')
6
                    = 0 # line count
             Pass
                     = []
             Hours = []
             for row in reader:
10
                  if count != 0: # skip header
11
                      Pass .append(float(row[0]))
12
                      Hours.append(float(row[1]))
13
                  count += 1
14
         y = np.array(Pass)
15
         x = np.array(Hours)
         n = len(y)
17
         X = np.reshape(x, (n,1))
18
         X = np.append(np.ones((n, 1)), X, axis=-1)
19
         y = 2 * y - 1
20
                  = np.zeros((2,))
^{21}
         start
                  = 10 ** -8
         eps
22
                  = lambda w: ll(X, y, w)
                  = lambda w: gradLL(X, y, w)
24
         w, _, _ = gradient_ascent.findMaximum(f, gradF, start, eps)
25
         return w
26
```

Figure 7.6: The function logisticRegression.

If we run the function logisticRegressionFile using the data shown in Figure 7.4 the resulting values of the weight vector \mathbf{w} are

```
[-4.0746468959343405, 1.5033787070592017]
```

This shows that the probability P(h) that a student who has studied for h hours will pass the exam is given approximately as follows:

$$P(h) \approx \frac{1}{1 + \exp(4.1 - 1.5 \cdot h)}$$

Figure 7.7 shows a plot of this probability P(x).



Figure 7.7: Probability of passing an exam versus hours of studying.

7.3.4 Logistic Regression with SciKit-Learn

In this section we discuss how linear regression is done in the SciKit-Learn environment. We will improve on the previous example and study the data that is shown in Figure 7.8 on page 173. This Csv file contains data about a fictional exam. The first column indicates whether the student has passed or failed the exam. A passed exam is encoded as the integer 1, while a failed exam is encoded as the integer 0. The second column contains the number of hours that the student has studied for the exam. The third column contains the intelligence quotient, abbreviated as IQ. To better understand the data, we first plot it. This plot is shown in Figure 7.9 on page 174. The horizontal axis is used for the hours of study, while the vertical axis shows the IQ of the student. Students who have passed the exam are shown as blue dots, while those students who have failed their exam are shown as red dots.

When we inspect the diagram shown in Figure 7.9 we see that there are two outliers: There is one student who failed although he has an IQ of 125 and he did study for 3.5 hours. Maybe he was still drunk when he had to write the exam. There is also a student with an IQ of 104 who did pass while only studying for 2 hours. He just might have gotten lucky. We expect logistic regression to classify all other students correctly.

```
Pass, Hours, IQ
    0,0.50,110
2
    0,0.75,95
    0,1.00,118
    0,1.25,97
    0,1.50,100
6
    0,1.75,110
7
    0,1.75,115
    1,2.00,104
9
    1,2.25,120
10
    0,2.50,98
11
    1,2.75,118
12
    0,3.00,88
13
    1,3.25,108
14
    0,3.50,125
15
    1,4.00,109
16
    1,4.25,110
17
    1,4.50,112
18
    1,4.75,97
19
    1,5.00,102
20
    1,5.50,109
21
```

Figure 7.8: Results of an exam given hours of study and IQ.



Figure 7.9: Probability of passing an exam versus hours of studying.

```
import numpy
                                   as np
1
    import pandas
                                   as pd
2
    import sklearn.linear_model as lm
3
4
    ExamDF = pd.read_csv('exam-iq.csv')
5
    X
            = np.array(ExamDF[['Hours','IQ']])
6
    Y
            = np.array(ExamDF['Pass'], dtype=float)
7
    model = lm.LogisticRegression(C=10000, tol=1e-6, solver='newton-cg')
8
            = model.fit(X, Y)
9
            = M.intercept_[0]
10
    \vartheta_1, \vartheta_2 = M.coef_[0]
11
    errors = np.sum(np.abs(Y - model.predict(X)))
12
    print((len(Y) - errors) / len(Y))
13
```

Figure 7.10: Logistic Regression using SciKit-Learn

Figure 7.10 on page 174 shows a Python script that creates a logistic regression classifier with the help of the SciKit-Learn package.

- 1. In the first three lines we import the necessary modules. The support for logistic regression is located in the module sklearn.linear_model.
- 2. In line 5, the data from the file "exam-iq.csv" is read as a Pandas DataFrame.
- 3. Line 6 creates the feature matrix X by extracting the two independent variables "Hours" and "IQ".
- 4. Line 7 extracts the dependent variable "Pass". Since this variable is stored as an integer in the Csv file, we convert it into a floating point number. This is necessary because the method fit that we use later expects the dependent variable to be encoded as a floating point number.
- 5. Line 8 creates an object of class LogisticRegression. This object is initialized with a number of parameters:
 - (a) C specifies the amount of regularization. We will discuss the concept of regularization later when we discuss polynomial logistic regression in the next section. In this example we do not need any regularization. Setting C to a high value like 10 000 prevents regularization.
 - (b) tol is the tolerance that specifies when the iterative algorithm to maximize the log-likelihood should stop.
 - (c) solver specifies the numerical method that is used to find the maximum of the log-likelihood. By choosing "newton-cg" we specify that the conjugate gradient method should be used. This method is more sophisticated than gradient descent, but as this is not a course on numerical optimization we do not have the time to discuss it.
- 6. All the real work is happening in line 9, because there we use the method fit to create the logistic regression model.
- 7. The next two lines are needed to extract the coefficients ϑ_0 , ϑ_1 , and ϑ_2 that specify the logistic model. According to the model we have learned, the probability P(h) that a student, who has learned for h hours and has an IQ of q, will pass the exam, is given as

$$P(Y = 1|h, q) = S(\vartheta_0 + \vartheta_1 \cdot h + \vartheta_2 \cdot q)$$

In general, the model predicts that she will pass the exam if

$$\vartheta_0 + \vartheta_1 \cdot h + \vartheta_2 \cdot q \ge 0.$$

This can be rewritten as follows:

$$q \ge -\frac{\vartheta_0 + \vartheta_1 \cdot h}{\vartheta_2}.$$

The decision boundary are those values of (h,q) such that $P(h,q) = \frac{1}{2}$. This set of values satisfies the linear equation

$$q = -\frac{\vartheta_0 + \vartheta_1 \cdot h}{\vartheta_2}.$$

We have plotted this decision boundary as a green line in Figure 7.11 on page 176. Everybody who is located to the right of this line is predicted to pass the exam, while everybody who is located to the left is predicted to fail.



Figure 7.11: Probability of passing an exam versus hours of studying.

- 8. Line 12 computes the number of data for which the predictions of the model are wrong. The method predict(X) takes a design matrix X. Each row of X is assumed to be a feature vector. It creates a prediction vector. The *i*-th entry of this vector is 1 if the model predicts a pass for the *i*-th student. Otherwise this entry is 0.
- 9. Line 13 prints the accuracy. As Figure 7.11 shows, 3 examples have been miss-predicted. Two of these examples where bound to be miss-predicted, but the fact that the student with an IQ of 120 who has studied for 2.25 hours has also been miss-predicted is disappointing. The reason is that logistic regression does not maximize the accuracy but rather maximizes the log-likelihood. Later, we will discuss so called support vector machines. Often, a support vector machine is able to achieve a higher accuracy than logistic regression. However, support vector machines are also more complicated than logistic regression.

The jupyter notebook containing the computation discussed previously can be found at my github repository:

Artificial-Intelligence/blob/master/Python/Logistic-Regression-with-SciKit-Learn.ipynb

Exercise 17: The file iris.csv contains the sizes of both the sepals and the petals of three different specimen of the iris flower. The data is described in more detail here. Use logistic regression to predict whether a given plant is of the species iris setosa (Deutsch: Borsten-Schwertlilie), iris virginica¹, or iris versicolor (Deutsch: Verschiedenfarbige Schwertlilie). As logistic regression is only able to distinguish between two different classes, you have to build three different classifiers:

- The first classifier is able to distinguish iris setosa from other irises.
- The second classifier is able to distinguish iris virginica from other irises.

¹This plant is native to North America and hence has no German name.

• The third classifier is able to distinguish iris versicolor from other irises.

Your task is to implement these classifiers and to evaluate their accuracy. You should divide the data randomly into a training dataset, which is used for computing the coefficients of logistic regression and a test dataset, which you should only use to predict the accuracy of your model. To this end, the function

```
sklearn.model_selection.train_test_split
```

might be useful. Once you have created these classifiers, proceed to implement a classifier that inputs a feature vector and that outputs the class of the iris flower as a string. If you do this correctly, you can achieve an accuracy that exceeds 95%.

7.4 Polynomial Logistic Regression

Sometimes logistic regression does not work because the data is not linearly separable. For example, Figure 7.12 on page 177 shows a classification problem where we have two features x and y and, obviously, it is not possible to separate the blue dots from the red dots by a vertical line.



Figure 7.12: Some fake data.

If we try to separate the data in Figure 7.12 by logistic regression, we get the result shown in Figure 7.13 on page 178. The data points above the green line are classified as red, while the data

points below the green line are classified as blue. The accuracy achieved is about 61%, so more than 38% of the data have been miss-classified.



Figure 7.13: Fake data with linear decision boundary.

We can arrive at a better model if we extend our data with second order polynomial features, i.e. we will not only consider the features x and y but also use x^2 , y^2 , and $x \cdot y$ as features. Then the resulting decision boundary will be a conic section, i.e. it might be an ellipse, a parabola, or a hyperbola. Figure 7.14 on page 179 shows a *Python* script that reads the fake data shown in Figure 7.12, adds second order polynomial features to this data and then performs linear regression.

- 1. The first three lines import the modules needed for this task.
- 2. As we want to split our data into a training set and a test set, we import the function train_test_split from sklearn.model_selection.
- 3. Line 6 reads the data from the file "fake-data.csv".
- 4. Line 7 and 8 extract the independent variables "x" and "y" and the dependent variable "class" and stores them in the design matrix X and the vector Y, respectively.
- 5. Line 10 splits the data into a training set and a test set. We allocate 20% of our data to the test set, while the remaining 80% are used as training data. In order to be able to reproduce our results, we set random_state to a fixed value. This forces the random number generator to always produce the same split of the data.

```
import numpy as np
1
    import pandas as pd
2
    import sklearn.linear_model as lm
    from sklearn.model_selection import train_test_split
5
    DF = pd.read_csv('fake-data.csv')
6
    X = np.array(DF[['x','y']])
    Y = np.array(DF['class'])
8
9
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=1)
10
11
    def logistic_regression(X_train, Y_train, X_test, Y_test, reg=10000):
12
                 = lm.LogisticRegression(C=reg, tol=1e-6, solver='newton-cg')
13
                  = model.fit(X_train, Y_train)
14
                 = M.score(X_train, Y_train)
        score
15
        yPredict = M.predict(X_test)
16
        accuracy = np.sum(yPredict == Y_test) / len(Y_test)
17
        return M, score, accuracy
18
19
    def extend(X):
20
        n = len(X)
21
        fx = np.reshape(X[:,0], (n, 1))
22
        fy = np.reshape(X[:,1], (n, 1))
23
        return np.hstack([fx, fy, fx*fx, fy*fy, fx*fy])
24
25
    X_train_quadratic, X_test_quadratic = extend(X_train), extend(X_test)
    logistic_regression(X_train_quadratic, Y_train, X_test_quadratic, Y_test)
```

Figure 7.14: A script for second order logistic regression.

- 6. Line 12 defines the function logistic_regression. This function takes 5 arguments.
 - (a) X_train and Y_train are the training data.
 - (b) X_test and Y_ are the test data.
 - (c) reg is a regularization parameter. By default this parameter is set to a high value. Setting this parameter to a high value ensures that there is no regularization. The default is to use next to no regularization.

The function returns a triple of the form (M, score, accuracy) where

- (a) M is the model that has been found.
- (b) score is the fraction of data points from the training set that have been classified correctly.
- (c) accuracy is the fraction of data points from the test set that have been classified correctly.
- 7. The function extend(X) takes a design matrix X as its argument. In order to keep the implementation of this function simple, we assume that X has just two features, i.e. the matrix X has shape (n,2) where n is the number of rows of X.

- (a) We extract the two features of X in line 22 and 23.
- (b) In line 24 the new feature matrix is created by stacking the original features fx and fy together with the new features fx², fy², fx · fy,
- 8. Line 26 extends both the training set and the test set with second order features.
- 9. Line 27 performs logistic regression using the new features. The use of second order feature improves the accuracy on the training set to 84%, while the accuracy on the test set improves to 76%. Figure 7.15 on page 180 shows the resulting decision boundary.



Figure 7.15: Elliptical decision boundary for fake data.

As adding second order features has increased the accuracy considerably, we proceed to add higher order features. Figure 7.16 on page 181 shows how we can add arbitrary polynomial features of higher degree. This script is a continuation of the script shown in Figure 7.14.

- 1. Line 28 imports the function PolynomialFeatures from sklearn.preprocessing. This function can be used to add all polynomial features up to a given order. We do not need a bias term here as it is automatically added by the function LogisticRegression.
- 2. Line 30 creates an object that can be used to extend a design matrix with polynomial features up to degree 4

```
from sklearn.preprocessing import PolynomialFeatures

quartic = PolynomialFeatures(4, include_bias=False)

X_train_quartic = quartic.fit_transform(X_train)

X_test_quartic = quartic.fit_transform(X_test)

logistic_regression(X_train_quartic, Y_train, X_test_quartic, Y_test)
```

Figure 7.16: Polynomial Logistic Regression.

- 3. Line 31 and 32 extend the training set and the test set with all polynomial features up to degree 4.
- 4. When we perform logistic regression with these new features, we achieve an accuracy of 88.4% on the training set. However, the accuracy on the test set does not improve. Figure 7.17 on page 181 shows the data with the resulting decision boundary.



Figure 7.17: Fake data with a decision boundary of fourth order.

Nothing stops us from cranking the order of the polynomial features to be added higher. Figure 7.18 on page 182 shows what happens when we include all polynomial features up to a degree of 14.

In this case, we 100% accuracy on the training set. However, the accuracy on the test set is only 80%. Clearly, we are overfitting the data.



Figure 7.18: Fake data with a decision boundary of order 14.

In order to combat overfitting we need to regularize, i.e. we need to penalize high values of the parameters. If we lower the regularization parameter down to 100, then the accuracy on the training set drops down to 89.6%, but the accuracy on the test set increases to 88%. The resulting decision boundary is shown in Figure 7.19 on page 183. Clearly, this decision boundary looks less complicated than the boundary shown in Figure 7.18. Contrary to the previous figures, this figure shows all the data. The previous figures had only shown the training data.

Exercise 18:

- (a) Assume that a design matrix X has two features x_1 and x_2 . Given a natural number n, you want to extend this design matrix by adding all features of the form $x_1^{k_1} \cdot x_2^{k_2}$ such that $k_1 + k_2 \le n$, i.e. you want to add all features up to a degree of n. How many features will the extended design matrix have?
- (b) Next, assume that the design matrix X has three features x_1, x_2 , and x_3 . This time, you want to extend the design matrix by adding all features of the form $x_1^{k_1} \cdot x_2^{k_2} \cdot x_3^{k_3}$ where $k_1 + k_2 + k_3 \le n$. How many features will the extended design matrix have in this case?
- (c) In the general case, the design matrix has d features x_1, \dots, x_d . Assume you want to add all polynomial terms up to order n as new features, i.e. you want to add all features of the form



Figure 7.19: Fake data with a decision boundary of order 14, regularized.

$$x_1^{k_1} \cdot x_2^{k_2} \cdot \ldots \cdot x_d^{k_d}$$
 such that $k_1 + k_2 + \cdots + k_d \le n$.

How many features will the extended design matrix have in the general case?

Hint: You might find it helpful to revisit your old lecture notes on statistics.

7.5 Naive Bayes Classifiers

In this section we discuss naive Bayes classifiers. Naive Bayes classifiers are an alternative method for classification which is appropriate in cases where the features are not numerical but rather are categorical. It starts with Bayes' theorem: Assume we have some evidence E about an object o and want to know whether o belongs to some class C. Bayes' theorem tell us that the conditional probability P(C|E), i.e. the probability that o has class C given that we have observed the evidence E, is related to the conditional probability P(E|C), which is the probability that we observe the evidence E given that o has class C, in the following way:

$$P(C|E) = \frac{P(E|C) \cdot P(C)}{P(E)}.$$

This theorem is useful because often the conditional probability P(E|C) that we observe some evidence E in an object o of class C is readily available, but the conditional probability P(C|E) that an object has class C if we have observed the evidence E is unknown. In the context of machine learning

the evidence E is often given as a list of features f_1, \dots, f_m that we are able to observe or compute. In this case we have to rewrite Bayes' theorem as follows:

$$P(C \mid f_1 \wedge \cdots \wedge f_m) = \frac{P(f_1 \wedge \cdots \wedge f_m \mid C) \cdot P(C)}{P(f_1 \wedge \cdots \wedge f_m)}.$$

In order to apply this form of Bayes' theorem to the problem of classification, we have to rewrite the expression

$$P(f_1 \wedge \cdots \wedge f_m \mid C)$$
.

In order to be able to do this, we need some theory which will be developed next. The conditional probability P(A|B) that an event A happens when it is already known that B has happened is defined as

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

This equation can be rewritten as

$$P(A \wedge B) = P(A|B) \cdot P(B).$$

Because conditional probabilities are probabilities and hence obey all the laws for probabilities, this equation is also true for conditional probabilities:

$$P(A \wedge B \mid C) = P(A \mid B \wedge C) \cdot P(B \mid C).$$

This equation can be generalized to the so called chain rule of probability:

$$P(A_{1} \wedge \cdots \wedge A_{m} \mid C) = P(A_{1} \wedge \cdots \wedge A_{m-1} \mid A_{m} \wedge C) \cdot P(A_{m} \mid C)$$

$$= P(A_{1} \wedge \cdots \wedge A_{m-2} \mid A_{m-1} \wedge A_{m} \wedge C) \cdot P(A_{m-1} \mid A_{m} \wedge C) \cdot P(A_{m} \mid C)$$

$$= \cdots$$

$$= P(A_{1} \mid A_{2} \wedge \cdots \wedge A_{m} \wedge C) \cdot \cdots \cdot P(A_{m-1} \mid A_{m} \wedge C) \cdot P(A_{m} \mid C)$$

$$= \prod_{i=1}^{m} P(A_{i} \mid A_{i+1} \wedge \cdots \wedge A_{m} \wedge C)$$

Two events A and B are defined to be conditional independent given an event C if and only if we have

$$P(A \mid C) = P(A \mid B \land C).$$

To put this equation differently, once we know that C holds, when it comes to the estimating the probability of A, then it does not matter whether B holds or not. Now the important assumption that a naive Bayes classifier makes is that in order to estimate the class C of an object o that has features f_1, \dots, f_m it is assumed that the features f_1, \dots, f_m are conditionally independent once the class is known. In most applications of naive Bayes classifiers this assumption is not true because there might be some weak correlation between the features. That explains why naive Bayes classifiers are called naive. Still, in practise the conditional independence of the features given the class is often approximately true and therefore these classifiers are useful. If we make the assumption of conditional independence, then the probability $P(C \mid f_1 \wedge \dots \wedge f_m)$ of an object o with features f_1, \dots, f_m to be of class C is given as

$$P(C \mid f_1 \land \dots \land f_m) = \frac{P(f_1 \land \dots \land f_m \mid C) \cdot P(C)}{P(f_1 \land \dots \land f_m)} \cdot P(C)$$

$$= \frac{\prod_{i=1}^{m} P(f_i \mid f_{i+1} \land \dots \land f_m \land C)}{P(f_1 \land \dots \land f_m)} \cdot P(C)$$

$$= \frac{\prod_{i=1}^{m} P(f_i \mid C)}{P(f_1 \land \dots \land f_m)} \cdot P(C)$$

In the last line of the previous chain of equations we have used the fact that the features f_1, \dots, f_m are conditionally independent given C. Now a naive Bayes classifier works as follows: Assume we have a set of n classes $\mathcal{C} = \{C_1, \dots, C_n\}$ from which we have to choose the class of an object o given the features f_1, \dots, f_m . We assume that o has class C_k if and only if the probability $P(C_k \mid f_1 \land \dots \land f_m)$ is maximal with respect to all classes of \mathcal{C} . In order to be able to specify this in a more formal way, we define the $\arg\max$ function: Given a set S and a function $f: S \to \mathbb{R}$ that has exactly one maximum, we define

$$\arg\max_{x\in S} f(x) := \operatorname{arb}\Big(\big\{x\in S\mid \forall y\in S: f(y)\leq f(x)\big\}\Big),$$

where the function $\operatorname{arb}(M)$ returns an arbitrary element from the set M. Since we assume that f has exactly one element on the set S, the expression $\arg\max_{x\in S}f(x)$ is well defined. To put the definition of $\arg\max_{x\in S}f(x)$ differently, the idea is that $\arg\max_{x\in S}f(x)$ computes the value of x that maximizes f. Given the features f_1, \dots, f_m , the naive Bayes classifier computes the most probable class as follows:

$$exttt{NaiveBayes}(f_1,\cdots,f_m) := rg \max_{C \in \mathcal{C}} rac{\prod\limits_{i=1}^m P(f_i \mid C)}{P(f_1 \wedge \cdots \wedge f_m)} \cdot P(C)$$

It is important to observe that the denominator $P(f_1 \wedge \cdots \wedge f_m)$ does not depend on the class C. As we only need to determine the class with the maximal probability, not the exact probability of the class, we can simplify the definition by dropping this denominator. Therefore the definition of the naive Bayes classifier can be rewritten as follows:

$$\mathtt{NaiveBayes}(f_1,\cdots,f_m) := rg \max_{C \in \mathcal{C}} \left(\prod_{i=1}^m P(f_i \mid C) \right) \cdot P(C)$$

This equation can be implemented once we have a training set T of objects with known classes: The probability P(C) is the probability that an object o has class C if nothing else is known about this object. P(C) is estimated as the proportion of objects in T that are of class C:

$$P(C) \approx \frac{\mathrm{card}\big(\{t \in T \mid \mathrm{class}(t) = C\}\big)}{\mathrm{card}(T)}.$$

This expression is called the prior probability of C or sometimes just the prior of C. In this equation, given an object $t \in T$ the function $\mathtt{class}(t)$ determines the class of the object t, while $\mathtt{card}(M)$ returns the number of elements of the set M.

Next, given a feature f and a class C, we have to determine the conditional probability that an object of class C exhibits the feature f_i , i.e. we have to determine $P(f_i \mid C)$. This probability can be estimated as the proportion of those objects of class C in the training set T that possess the feature f:

$$P(f \mid C) \approx \frac{\mathrm{card} \big(\{t \in T \mid \mathrm{class}(t) = C \wedge \mathrm{has}(t,f) \} \big)}{\mathrm{card} \big(\{t \in T \mid \mathrm{class}(t) = C \} \big)}$$

Here, for an object t and a feature f the expression has(t, f) is true if and only if t has the feature f.

7.5.1 Example: Spam Detection

Spam detection is an important application of classification. We will see in this subsection that naive Bayes classifiers work well for spam detection. The directory

https://github.com/karlstroetmann/Artificial-Intelligence/tree/master/Python/EmailData contains 960 emails that are partitioned into four subdirectories:

- 1. spam-train contains 350 spam emails for training,
- 2. ham-train contains 350 non-spam emails for training,
- 3. spam-test contains 130 spam emails for testing,
- 4. ham-test contains 130 non-spam emails for testing.

This data has been collected by Ion Androutsopoulos. Figure 7.20 on page 194 and 7.21 on page 195 show parts of the notebook

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/Spam-Detection.ipynb.

This notebook implements a naive Bayes classifier for spam detection. We proceed to discuss the details of its implementation.

- 1. Initially we load a number of modules. In addition to numpy and math these are
 - (a) os to list the files in a directory.
 - (b) re for regular expressions.
- 2. We import the module Counter from the package collections. A Counter is a special type of dictionary that is well suited for counting objects.
- 3. We set some variables that would need to be adapted if this notebook would be used with a different set of Emails.
- 4. Using the function listdir from the module os we count the number of spam emails and the number of ham emails in the corresponding directories. These numbers are than used to compute the prior probabilities for spam and ham. In our example, the number of spam emails and the number of ham emails are both 350. Therefore, the prior probability of an email to be spam is $\frac{1}{2}$ and the prior probability for ham is also $\frac{1}{2}$.
- 5. The function get_words(fn) takes a filename fn as its argument.
 - (a) It reads the specified file as a string of text.
 - (b) This string of text is then converted to lower case.

 In our case the conversion to lower case would not be necessary as the emails that we use have been preprocessed and are already converted to lower case.
 - (c) The text string is split into a list of words using the function findall from the module re.

 The regular expression

specifies all strings that are made up of Unicode word characters and single quote characters. The list of words returned by findall is then converted to a set and returned.

6. The function read_all_files reads all files contained in the directories that are stored in the list Directories. It returns a Counter. For every word w this counter contains the number of those files that contain w.

Given a counter C and a set S, the function C.update(S) increments the count of every element x in C that occurs in the set S. For example, if

$$C = \mathtt{Counter}(\{\mathtt{a} \colon 1, \mathtt{b} \colon 2, \mathtt{c} \colon 3\}),$$

then the call $C.update(\{a,d\})$ changes the counter C such that afterwards

$$C = \mathtt{Counter}(\{\mathtt{a} \colon 2, \mathtt{b} \colon 2, \mathtt{c} \colon 3, \mathtt{d} \colon 1\}).$$

- 7. Word_Counter is a Counter object that contains the counts of all words that occur in any email.
- 8. Common_Words is list of the 2500 most common words occurring in all emails.
- 9. The function get_common_words(fn) takes a filename fn as its argument. It reads the specified file and returns set of all words in Common_Words that are found in the specified file.
- 10. The function count_common_words takes a directory as its argument. It returns a Counter that counts how often the words in Common_Words occur in any of the files in directory.
- 11. We use this function to count how often the most common words occur in spam and ham emails. These counts are stored in the dictionaries Spam_Counter and Ham__Counter.

The second part of our spam classifier is shown in Figure 7.21 on page 195 and is discussed below.

1. Given the dictionaries Spam_Counter and Ham__Counter, we proceed to compute the conditional probabilities that one of the most common word occurs in a spam or ham email. To this end we define the dictionaries Spam_Probability and Ham__Probability. For every word $w \in \text{Common_Words}$, we will have that Spam_Probability[w] is the conditional probability that the word w occurs in a spam email, i.e. we have

$$Spam_Probability[w] = P(w \mid C = spam).$$

A first attempt to estimate this probability is to approximate it as the fraction of all spam mails containing w. This would lead to the formula

$$P(w \mid C = \mathtt{spam}) = \frac{\mathtt{Spam_Counter}[w]}{N}, \quad \text{where N is the number of all spam training mails.}$$

However, this would imply that if $Spam_Counter[w] = 0$ because the training set has no spam mail that contains the word w, then the probability $P(w \mid C = spam)$ would be estimated as 0. Clearly, this cannot be right: Even if there is a word w that has so far never occurred in a spam mail, this does not mean that any mail containing this word is necessarily ham.

To get this right we use Laplace smoothing: We assume that there is one additional spam email that contains every word w from Common_Words. With this assumption, the formula for the conditional probability $P(w \mid C = \text{spam})$ is changed as follows:

$$P(w \mid C = \mathtt{spam}) = \frac{\mathtt{Spam_Counter}[w] + 1}{N+1}, \quad \text{where N is the number of all spam training mails.}$$

2. The function spam_probability takes a filename and computes the probability that the email contained in the given file is spam.

When implementing the formula

$$\arg\max_{C\in\mathcal{C}} \left(\prod_{i=1}^{m} P(f_i \mid C)\right) \cdot P(C)$$

we have to be careful, because a naive implementation will evaluate the product

$$\prod_{i=1}^{m} P(f_i \mid C)$$

as the number 0 due to numerical underflow. The trick to compute this product is to remember that

$$\ln(a \cdot b) = \ln(a) + \ln(b)$$

and therefore transform the product into a sum of logarithms:

$$\prod_{i=1}^{m} P(f_i \mid C) = \exp\left(\alpha + \sum_{i=1}^{m} \ln(P(f_i \mid C))\right) \cdot \exp(-\alpha)$$

Here, the constant α has to chosen such that the application of the exponential function to the value

$$\alpha + \sum_{i=1}^{m} \ln(P(f_i \mid C))$$

does not lead to an underflow error.

- 3. In order to evaluate the performance of this algorithm, we need to define two new concepts: precision and recall. Let us call the ham emails the positives, while the spam emails are called the negatives. Then we define
 - (a) true positives: ham emails that are classified as ham,
 - (b) false positives: spam emails that are classified as ham,
 - (c) true negatives: spam emails that are classified as spam,
 - (d) false negatives: ham emails that are classified as spam.

The precision of the spam classifier is then defined as

$$\label{eq:precision} \begin{aligned} \text{precision} &= \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false positives}} \end{aligned}$$

Therefore, the precision measures the percentage of the ham emails in the set of all emails that are classified as ham. The recall of the spam classifier is defined as

$${\tt recall} = \frac{number\ of\ true\ positives}{number\ of\ true\ positives + number\ of\ false\ negatives}$$

Therefore, the recall measures the percentage of those ham emails that are indeed classified as ham

Usually, it is very important that the recall is high as we don't want to miss a ham email because our classifier has incorrectly classified it as a spam email. On the other hand, having a high precision is not that important. After all, if 10% of the emails offered to us as ham are, in fact, spam, we might tolerate this. However, we would certainly not tolerate missing 10% of our ham emails because they are incorrectly specified as spam.

7.5.2 Naive Bayes Classifier with Numerical Features

We can build a naive Bayes classifier even if some of our features are numerical. Assume we have a feature f that is a numerical attribute. The tricky part is to come up with a way to compute the

conditional probability

$$P(f = x \mid C)$$

which is the conditional probability that the feature f has the value x if the object to classify has class C. The idea is to assume that for every class C the values of the feature f have a Gaussian distribution. Then we have

$$P(f = x \mid C) = \frac{1}{\sqrt{2 \cdot \pi} \cdot \sigma_{f,C}} \cdot \exp\left(-\frac{\left(x - \mu_{f,C}\right)^2}{2 \cdot \sigma_{f,C}^2}\right),$$

where $\mu_{f,C}$ is the mean value of the feature f for those objects that have class C, while $\sigma_{f,C}^2$ is the variance of the feature f for objects of class C.

Exercise 19: We have already investigated the file iris.csv in a previous exercise. This time, your task is to implement a naive Bayes classifier that is able to classify iris flowers. You can achieve an accuracy that exceeds 95%.

7.5.3 Example: Gender Estimation

In order to clarify the theory of naive Bayes classifiers, this section presents an example that shows how a naive Bayes classifier can be used. In this example, our goal is to estimate the gender of a first name. For example, the string "Bianca" is a female first name, while the string "Michael" is a male first name. A crude first attempt to distinguish female names from male ones is to look at the last character. Hence, our first classifier to solve this problem will use just a single feature. This feature can have one of 26 different values.

Figure 7.22 on page 196 shows a *Python* script that implements a naive Bayes classifier for gender prediction. In order to train our classifier, we need a training set of names that are marked as being either male. We happen to have two text files, "names-female.txt" and "names-male.txt" containing female and male first names.

- 1. We start by defining the function read_names. This function takes a file name as its argument and reads the specified file one line at a time. It returns a list of all the names given in the file. Care is taken that the newline character at the end of each line is discarded.
- 2. We use this function to read both the female names and the male names and store these names in the lists FemaleNames and MaleNames.
- 3. Next, we compute the prior probabilities P(Female) and P(Male) for the classes Female and Male. Previously, we have shown that the prior probability of a class C in a training set T is given as:

$$P(C) \approx \frac{\mathrm{card} \big(\{ t \in T \mid \mathrm{class}(t) = C \} \big)}{\mathrm{card}(T)}.$$

Therefore, these prior probability that a name is female is the fraction of the number of female names in the set of all names. Similarly, the prior probability that a name is male is the fraction of the number of male names in the set of all names. These probabilities are stored as pFemale and pMale.

4. The formula to compute the conditional probability of a feature f given a class C is as follows:

$$P(f \mid C) \approx \frac{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C \land \texttt{has}(t,f)\}\big)}{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C\}\big)}$$

The function $conditional_prop(c, g)$ takes a character c and a gender g and determines the conditional probability of seeing c as a last character of a name that has the gender g.

- 5. Next, we define the dictionary Conditional_Probability. For every character c and every gender $g \in \{ 'f', 'm' \}$, the entry Conditional_Probability[(c, g)] is the conditional probability of observing the last character c if the gender of the name is known to be g.
- 6. The dictionary Conditional_Probability can now be used to define the function classify(name) that takes a name as its input and outputs the estimated gender.
- 7. Finally, we check the accuracy of this classifier on the training set. When we run the program, we see that the accuracy attained is about 76%. Since we are using only a single feature here, this is a reasonable result.

The file NLTK-Introduction.ipynb contains a Jupyter notebook that uses the Natural Language Toolkit (NLTK) to implement a more sophisticated classifier for gender estimation.

7.6 Support Vector Machines

Support Vector Machines (abbreviated as SVMs) had been in invented in 1963 by Vladimir Naumovich Vapnik and Alexey Yakovlevich Chervonenkis. However, they only got widespread acceptance in 1995 when Cortes and Vapnik published a paper explaining the kernel trick [CV95]. This section will introduce support vector machines. In order to motivate SVMs, we first explain why logistic regression sometimes behaves suboptimally. After that, we explain the mathematical theory of support vector machines. Finally, we show how we can use the support vector machines provided by SciKit Learn.

7.6.1 Non-Optimality of Logistic Regression

Figure 7.23 on page 197 shows three points that belong to two different classes. The blue dot at position (3.5, 3.5) belongs to class 1, while the two red crosses at position (1, 2) and (2, 1) belong to the class -1. When we build a classifier to separate these two classes, we would ideally like the decision boundary to be the green line that passes through the points (0, 5) and (5, 0), since this line has the biggest distance from both classes. Figure 7.24 on page 198 shows how logistic regression deals with with this problem.

A close inspection of Figure 7.24 shows that the decision boundary calculated by logistic regression is nearer to the blue dot than it is to the red crosses. If we add a large number of blue points right next to the first blue points, the decision boundary found by logistic regression moves away from the blue points, as shown in Figure 7.25 on page 199, then the decision boundary moves away from the first blue point that marks the margin of the two classes. These new blue points do not add real information, since they are further away from the red crosses than the first blue point. Hence it is counter-intuitive that the addition of these points changes the decision boundary.

7.6.2 The Mathematical Theory of Support Vector Machines

The main idea of support vector machines is to have a decision boundary that is as simple as possible and that separates the different classes as much as possible. In two dimensions, the simplest decision boundary is a line. In n dimensions, the simplest decision boundary is an (n-1)-dimensional hyperplane. A hyperplane separates two different classes as much as possible if the distance to both classes is maximized.

A hyperplane can be defined by a vector \mathbf{w} that is perpendicular to the hyperplane together with a bias b: A vector \mathbf{x} is an element of the hyperplane if and only if

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$
.

In order for the decision boundary to separate the positive examples from the negative examples, we add the following two conditions. If $\mathbf{x}^{(i)}$ is a positive example, then we don't just want that

$$\mathbf{w} \cdot \mathbf{x}^{(i)} + b \ge 0.$$

Instead, we demand that

$$\mathbf{w} \cdot \mathbf{x}^{(i)} + b \ge 1 \tag{7.3}$$

holds. Similarly, if $\mathbf{x}^{(i)}$ is a negative example, we want to have that

$$\mathbf{w} \cdot \mathbf{x}^{(i)} + b \le -1 \tag{7.4}$$

holds. Let us define the class of a positive example to be +1 and the class of a negative example to be -1. Let y_i denotes the class of example $\mathbf{x}^{(i)}$. Let us multiply equation 7.3 by $y_i = 1$. Unsurprisingly, we get

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \ge 1. \tag{7.5}$$

Similarly, let us multiply equation 7.3 by $y_i = -1$. This time, things get more interesting as the direction of the inequality is flipped:

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \ge 1. \tag{7.6}$$

Notice that the equations 7.5 and 7.6 are the same! Hence inequation 7.5 holds for both positive and negative examples. We rewrite the last inequation as

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) - 1 \ge 0. \tag{7.7}$$

Those vectors $\mathbf{x}^{(i)}$ that satisfy the equality

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) - 1 = 0$$

are at the margins of their respective classes and are called support vectors. These vectors have the smallest distance to the hyperplane defined by \mathbf{w} and b. Let us compute the width of the separation of the positive class from the negative class if the decision boundary is given by the equation $\mathbf{w} \cdot \mathbf{x} + b = 0$. To this end, assume that \mathbf{x}_+ is a positive support vector, i.e. we have

$$\mathbf{w} \cdot \mathbf{x}_{+} + b = 1,\tag{7.8}$$

while \mathbf{x}_{-} is a negative support vector and therefore satisfies

$$\mathbf{w} \cdot \mathbf{x}_{-} + b = -1. \tag{7.9}$$

Since the vector \mathbf{w} is perpendicular to the hyperplane that defines the decision boundary, the width between the positive and the negative example is given by the projection $\mathbf{x}_+ - \mathbf{x}_-$ on the normalized vector \mathbf{w} :

$$width = (\mathbf{x}_{+} - \mathbf{x}_{-}) \cdot \frac{\mathbf{w}}{||\mathbf{w}||} = (\mathbf{x}_{+} \cdot \mathbf{w} - \mathbf{x}_{-} \cdot \mathbf{w}) \cdot \frac{1}{||\mathbf{w}||}$$

$$(7.10)$$

If we subtract equation 7.9 from equation 7.8, the constant b cancels and we are left with

$$\mathbf{x}_+ \cdot \mathbf{w} - \mathbf{x}_- \cdot \mathbf{w} = 2.$$

Substituting this equation into equation 7.10 yields the equation

$$\mathtt{width} = \frac{2}{||\mathbf{w}||}.$$

Hence in order to maximize the width of the separation of the two classes from the decision boundary we have to minimize the size of the vector \mathbf{w} subject to the constraints given in equation 7.7. Now minimizing \mathbf{w} is the same as minimizing

$$\frac{1}{2} \cdot ||\mathbf{w}||^2 = \frac{1}{2} \cdot \sum_{k=1}^d w_k^2,$$

where d is the number of features. Determining a minimum of a function that is subject to a set of constraints requires us to use Lagrange multipliers. Assuming our training set has the form $\{\mathbf{x}^{(1)}, \dots \mathbf{x}^{(n)}\}$, we define the Lagrangian $\mathcal{L}(\mathbf{w}, b, \alpha_1, \dots, \alpha_n)$ as follows:

$$\mathcal{L}(\mathbf{w}, b, \alpha_1, \cdots, \alpha_n) := \frac{1}{2} \cdot ||\mathbf{w}||^2 - \sum_{i=1}^n \alpha_i \cdot (y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1).$$

The sum in this Lagrangian sums over all training examples although not all training examples have to satisfy the constraint

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0.$$

This is not a problem because for those $i \in \{1, \dots, \}$ where we only have the inequality

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \ge 0$$

we can just assume that the corresponding Lagrange multiplier α_i is equal to 0. A necessary condition for the values of \mathbf{w} , b and α_i that minimize \mathcal{L} is that the partial derivatives of \mathcal{L} with respect to w_k , b and α_i are all 0. Let us first compute the partial derivative of \mathcal{L} with respect to w_k :

$$\frac{\partial \mathcal{L}}{\partial w_k} = w_k - \sum_{i=1}^n \alpha_i \cdot y_i \cdot x_k^{(i)} = 0$$

Therefore, we must have that

$$w_k = \sum_{i=1}^n \alpha_i \cdot y_i \cdot x_k^{(i)}$$

and this implies

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i \cdot y_i \cdot \mathbf{x}^{(i)} \tag{7.11}$$

Therefore the vector \mathbf{w} is a linear combination of the support vectors, where a vector $\mathbf{x}^{(i)}$ is a support vector iff $\alpha_i \neq 0$. Hence a support vector $\mathbf{x}^{(i)}$ must satisfy the equality

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 = 0.$$

Next, let us compute the partial derivative of \mathcal{L} with respect to b. We have

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \alpha_i \cdot y_i = 0$$

which implies that

$$\sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \tag{7.12}$$

Let us rewrite the Lagrangian by substituting **w** with the right hand side of equation 7.11:

$$\mathcal{L} = \frac{1}{2} \cdot \left(\sum_{i=1}^{n} \alpha_{i} \cdot y_{i} \cdot \mathbf{x}^{(i)} \right) \cdot \left(\sum_{j=1}^{n} \alpha_{j} \cdot y_{j} \cdot \mathbf{x}^{(j)} \right) - \sum_{i=1}^{n} \alpha_{i} \cdot \left(y_{i} \cdot \left(\mathbf{x}^{(i)} \cdot \left(\sum_{j=1}^{n} \alpha_{j} \cdot y_{j} \cdot \mathbf{x}^{(j)} \right) + b \right) - 1 \right)$$

$$= \frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{i} \cdot \alpha_{j} \cdot y_{i} \cdot y_{j} \cdot \left(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right) - \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{i} \cdot \alpha_{j} \cdot y_{i} \cdot y_{j} \cdot \left(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right) - b \cdot \sum_{i=1}^{n} \alpha_{i} \cdot y_{i} + \sum_{i=1}^{n} \alpha_{i}$$

$$= \sum_{i=1}^{n} \alpha_{i} - \frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_{i} \cdot \alpha_{j} \cdot y_{i} \cdot y_{j} \cdot \left(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right)$$

Now, the crucial observation is the following: The Lagrangian \mathcal{L} only depends on the dot products $\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)}$. Why is this a big deal? Often, a set of data point is not linearly separable in the given space. However, it might be possible to transform the feature vectors $\mathbf{x} \in \mathbb{R}^d$ into some higher dimensional space \mathbb{R}^h where h > d and the tow classes are separable. Concretely, it is sometimes possible to define a transformation function

$$\Phi: \mathbb{R}^d \to \mathbb{R}^h$$

such that the set of transformed data points $\{\Phi(\mathbf{x}^{(1)}), \dots, \Phi(\mathbf{x}^{(n)})\}$ is linearly separable. The question then is to find such a transformation Φ . Here is the punch line: As the Lagrangian does only depend on dot products, it is sufficient to define the transformed dot products

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}).$$

This is done with the help of so called kernel functions: We define the dot product $\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y})$ as

$$\Phi(\mathbf{x}) \cdot \Phi(\mathbf{y}) := k(\mathbf{x}, \mathbf{y})$$

where k is called a kernel function. There are two kernel functions that are quite popular:

1. Polynomial kernels have the form

$$k(\mathbf{x}, \mathbf{v}) = (\mathbf{x} \cdot \mathbf{v} + c)^n$$
,

where n is a natural number called the degree of the kernel. The number c is a hyperparameter that is often set to 1.

2. Gaussian kernels have the form

$$k(\mathbf{x}, \mathbf{y}) = \exp\left(\frac{||\mathbf{x} - \mathbf{y}||^2}{2 \cdot \sigma^2}\right).$$

Here, σ is a hyperparameter.

Using a kernel function to simulate a parameter transformation is known as the kernel trick. Experience has shown that the kernel functions given above often enable us to transform a data set into a space where the data set is linearly separable. Figure 7.26 on page 200 shows a set of point that is separable using a support vector machine with a Gaussian kernel.

If you want to know more about support vector machines, the free book Support Vector Machines Succinctly by Alexandre Kowalczyk [Kow17] is a good place to start.

```
1
    import os
    import re
2
    import numpy as np
    import math
5
    from collections import Counter
6
    spam_dir_train = 'EmailData/spam-train/'
8
    ham__dir_train = 'EmailData/ham-train/'
9
    spam_dir_test = 'EmailData/spam-test/'
10
    ham__dir_test = 'EmailData/ham-test/'
11
    Directories
                    = [spam_dir_train, ham_dir_train, spam_dir_test, ham_dir_test]
12
13
                = len(os.listdir(spam_dir_train))
14
    no_spam
               = len(os.listdir(ham__dir_train))
    no_ham
15
    spam_prior = no_spam / (no_spam + no_ham)
16
    ham__prior = no_ham / (no_spam + no_ham)
17
19
    def get_words(fn):
        file = open(fn)
20
        text = file.read()
21
        text = text.lower()
22
        return set(re.findall(r"[\w']+", text))
23
24
    def read_all_files(Directories):
25
        Words = Counter()
26
        for directory in Directories:
27
             for file_name in os.listdir(directory):
28
                 Words.update(get_words(directory + file_name))
29
        return Words
30
31
    Word_Counter = read_all_files(Directories)
32
    Common_Words = { w for w, _ in Word_Counter.most_common(2500) }
33
34
    def get_common_words(fn):
35
        return get_words(fn) & Common_Words
36
37
    def count_commmon_words(directory):
38
        Words = Counter()
39
        for file_name in os.listdir(directory):
40
             Words.update(get_common_words(directory + file_name))
41
        return Words
42
43
    spam_counter = count_commmon_words(spam_dir_train)
44
    ham__counter = count_commmon_words(ham__dir_train)
45
```

Figure 7.20: A Naive Bayes Classifier for Spam Detection: Part I

```
Spam_Probability = {}
46
    Ham__Probability = {}
47
    for w in Common_Words:
        Spam_Probability[w] = (Spam_Counter[w] + 1) / (no_spam + 1)
49
        Ham__Probability[w] = (Ham__Counter[w] + 1) / (no_ham + 1)
50
51
    def spam_probability(fn):
52
        log_p_spam = 0.0
53
        log_p_ham = 0.0
54
        words = get_common_words(fn)
        for w in Common_Words:
             if w in words:
57
                 log_p_spam += math.log(Spam_Probability[w])
58
                 log_p_ham += math.log(Ham__Probability[w])
59
             else:
60
                 log_p_spam += math.log(1.0 - Spam_Probability[w])
61
                 log_p_ham += math.log(1.0 - Ham__Probability[w])
        alpha = abs(min(log_p_spam, log_p_ham))
        if alpha > 400: # avoid overflow
64
             if log_p_spam < log_p_ham:</pre>
65
                 return 0
66
             else:
67
                 return 1
68
        p_spam = math.exp(log_p_spam + alpha) * spam_prior
69
        p_ham = math.exp(log_p_ham + alpha) * ham__prior
70
        return p_spam / (p_spam + p_ham)
72
    def precission_recall(spam_dir, ham_dir):
73
        TN = 0 # true negatives
74
        FP = 0 # false positives
75
        for email in os.listdir(spam_dir):
76
             if spam_probability(spam_dir + email) > 0.5:
77
                 TN += 1
             else:
79
                 FP += 1
80
        FN = 0 # false negatives
81
        TP = 0 # true positives
82
        for email in os.listdir(ham_dir):
83
             if spam_probability(ham_dir + email) > 0.5:
84
                 FN += 1
             else:
                 TP += 1
        precision = TP / (TP + FP)
88
                   = TP / (TP + FN)
        recall
89
        accuracy = (TN + TP) / (TN + TP + FN + FP)
90
        return precision, recall, accuracy
91
```

Figure 7.21: A Naive Bayes Classifier for Spam Detection: Part II

```
def read_names(file_name):
        Result = []
2
        with open(file_name, 'r') as file:
            for name in file:
                 Result.append(name[:-1]) # discard newline
        return Result
    FemaleNames = read_names('names-female.txt')
8
    MaleNames
                = read_names('names-male.txt'
9
    pFemale
                = len(FemaleNames) / (len(FemaleNames) + len(MaleNames))
10
                                    / (len(FemaleNames) + len(MaleNames))
    pMale
                 = len(MaleNames)
11
    def conditional_prop(c, g):
13
        if g == 'f':
14
            return len([n for n in FemaleNames if n[-1] == c]) / len(FemaleNames)
15
        else:
16
                                                 if n[-1] == c]) / len(MaleNames)
            return len([n for n in MaleNames
17
18
    Conditional_Probability = {}
19
    for c in 'abcdefghijklmnopqrstuvwxyz':
20
        for g in ['f', 'm']:
21
            Conditional_Probability[(c, g)] = conditional_prop(c, g)
22
23
    def classify(name):
24
        last
               = name[-1]
25
        female = Conditional_Probability[(last, 'f')] / pFemale
26
               = Conditional_Probability[(last, 'm')] / pMale
27
        if female >= male:
            return 'f'
29
        else:
30
            return 'm'
31
32
    total, correct = 0, 0
33
    for n in FemaleNames:
34
        if classify(n) == 'f':
            correct += 1
36
        total += 1
37
    for n in MaleNames:
38
        if classify(n) == 'm':
39
            correct += 1
40
        total += 1
41
    accuracy = correct / total
    print(f'The accuracy of our estimator is {accuracy}.')
43
```

Figure 7.22: A naive Bayes classifier for predicting the gender of a name.



Figure 7.23: Three points to separate.



Figure 7.24: Three points separated by logistic regression.



Figure 7.25: Points separated by logistic regression.



Figure 7.26: Points separated by a support vector machine.

Chapter 8

Neural Networks

In this chapter, we discuss artificial neural networks. Many of the most visible breakthroughs in artificial intelligence have been achieved through the use of neural networks:

- 1. The current system used by Google to automatically translate web pages is called "Google Neural Machine Translation" and, as the name suggests, is based on neural networks.
- 2. DeepL is another translator that is based on neural networks.
- 3. AlphaGo uses neural networks together with tree search [SHM⁺16]. It has beaten the world champion Ke Jie in the game of Go. AlphaGo has been succeeded by AlphaZero, which is even stronger than AlphaGo.
- 4. Image recognition is best done via neural networks.
- 5. Autonomous driving makes heavy use of neural networks.

The list given above is far from being complete. In this chapter, we will only discuss feedforward neural networks. Although recently both convolutional neural networks and recurrent neural networks have gotten a lot of attention, these type of neural networks are more difficult to understand and are therefore beyond the scope of this introduction. The rest of this chapter is strongly influenced by the online book

http://neuralnetworksanddeeplearning.com/index.html

that has been written by Michael Nielsen [Nie19]. This book is easy to read, carefully written, and free to access. I recommend this book to anybody who wants to dive deeper into the fascinating topic of neural networks.

We proceed to give an overview of the content of this chapter.

- 1. We start with the definition of feed forward neural networks and discuss their topology.
- 2. We introduce forward propagation, which is the way a neural network computes its predictions.
- 3. Similarly to our treatment of logistic regression, we define a cost function that measure the quality of the predictions of a neural network on a training set. In order to minimize this cost function using gradient descent, we have to compute the gradient of the cost function with respect to the weights of the neural network. The algorithm which is used to compute this gradient is called backpropagation.
- 4. In order to find the minimum of the cost function efficiently, we need an improved version of gradient descent. This improved version is known as stochastic gradient descent.

- 5. After having covered the theory, we implement a simple neural network that is able to recognize handwritten digits.
- 6. Finally, we discuss TensorFlow, which is an open source software library for machine learning in general and neural networks in particular.

8.1 Feedforward Neural Networks

A neural network is built from neurons. Neural networks are inspired by biological neurons. However, in order to understand artificial neural networks it is not necessary to know how biological neurons work and it is definitely not necessary to understand how networks of biological neurons, i.e. brains, work¹. Instead, we will use a mathematical abstraction of neurons that will serve as the foundation of the theory developed in this chapter. At the abstraction level that we are using to look at neural networks, a single neuron with m inputs is specified by a pair $\langle \mathbf{w}, b \rangle$ where the vector $\mathbf{w} \in \mathbb{R}^m$ is called the weight vector and the number $b \in \mathbb{R}$ is called the bias. Conceptually, a neuron is a function that maps an input vector $\mathbf{x} \in \mathbb{R}^m$ into the set \mathbb{R} of the real numbers. This function is defined as follows:

$$\mathbf{x} \mapsto a(\mathbf{x} \cdot \mathbf{w} + b).$$

Here, a is called the activation function. In our applications, we will use the sigmoid function as our activation function. This function has been defined previously in Definition 37 on page 162 as follows:

$$a(t) := S(t) = \frac{1}{1 + \exp(-t)}.$$

Another useful activation function is the so called ReLU function, which is defined as

$$a(t) = \max(0, x).$$

The abbreviation ReLU is short for rectified linear unit. The function modelling the neuron can be written more explicitly using index notation. If

$$\mathbf{w} = \langle w_1, \cdots, w_m \rangle^{\top}$$

is the weight vector and

$$\mathbf{x} = \langle x_1, \cdots, x_m \rangle^{\top}$$

is the input vector, and $b \in \mathbb{R}$ is the bias, then we have

$$\mathbf{x} \mapsto S\left(\left(\sum_{i=1}^{m} x_i \cdot w_i\right) + b\right).$$

If we compare this function to a similar function appearing in the last chapter, you will notice that a single neuron works just like a classifier in logistic regression. The only difference is that the bias b is now explicit in our notation. In logistic regression, we had assumed that the first component x_1 of our feature vector \mathbf{x} was always equal to 1. This assumption enabled us to incorporate the bias b into the weight vector \mathbf{w} .

A feedforward neural network is a layered network of neurons. Formally, the topology of a neural network is given by a number $L \in \mathbb{N}$ and a list $[m(1), \dots, m(L)]$ of L natural numbers. The number L is called the number of layers and for $l \in \{2, \dots, L\}$ the number m(l) is the number of neurons in the l-th layer. The first layer is called the input layer. The input layer does not contain neurons but instead just contains input nodes. The last layer (i.e. the layer with index L) is called the output layer and the remaining layers are called hidden layers. If there is more than one hidden layer, the

¹Actually, when it comes to brains, although there are many speculations, surprisingly little is known for a fact.



Figure 8.1: A neural network with topology [3, 6, 4, 2].

neural network is called a deep neural network. Figure 8.1 on page 203 shows a small neural network with two hidden layers. Including the input layer it has four layers and its topology is given by the list [3, 6, 4, 2]. A larger neural network with three hidden layers is shown in Figure 8.2 on page 204. I have written a small Jupyter notebook that can be used to draw diagrams of this kind. This notebook is available at NN-Architecture.ipynb in the Python subdirectory of my GitHub repository for this lecture.

If the topology of a neural network is $[m(1), \dots, m(L)]$, the input dimension is defined as m(1). Similarly, the output dimension is defined as m(L). The feedforward neural networks discussed in this chapter are fully connected: Every node in the l-th layer is connected to every node in the (l+1)-th layer via a weight. The weight $w_{j,k}^{(l)}$ is the weight of the connection from the k-th neuron in layer l-1 to the j-th neuron in layer l. The weights in layer l are combined into the weight matrix $W^{(l)}$ of the layer l: This matrix is defined as

$$W^{(l)} := (w_{i,k}^{(l)}).$$

Note that $W^{(l)}$ is an $m(l) \times m(l-1)$ matrix, i.e. we have

$$W^{(l)} \in \mathbb{R}^{m(l) \times m(l-1)}.$$

The j-th neuron in layer l has the bias $b_i^{(l)}$. These biases of layer l are combined into the bias vector

$$\mathbf{b}^{(l)} := \left\langle b_1^{(l)}, \cdots, b_{m(l)}^{(l)} \right\rangle^\top.$$



Figure 8.2: A neural network with topology [8, 12, 8, 6, 3].

The activation of the j-th neuron in layer l is denoted as $a_j^{(l)}$ and is defined recursively as follows:

1. For the input layer we have

$$a_j^{(1)}(\mathbf{x}) := x_j. \tag{FF1}$$

To put it differently, the input vector \mathbf{x} is the activation of the input nodes.

2. For all other layers we have

$$a_j^{(l)}(\mathbf{x}) := S\left(\left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)}(\mathbf{x})\right) + b_j^{(l)}\right) \quad \text{for all } l \in \{2, \cdots, L\}.$$
 (FF2)

The activation vector of layer l is defined as

$$\mathbf{a}^{(l)}(\mathbf{x}) := \langle a_1^{(l)}(\mathbf{x}), \cdots, a_{m(l)}^{(l)}(\mathbf{x}) \rangle^{\top}.$$

Using vector notation, the feed forward equations (FF1) and (FF2) can be rewritten as follows:

$$\mathbf{a}^{(1)}(\mathbf{x}) := \mathbf{x},\tag{FF1v}$$

$$\mathbf{a}^{(l)}(\mathbf{x}) := S\left(W^{(l)} \cdot \mathbf{a}^{(l-1)}(\mathbf{x}) + \mathbf{b}^{(l)}\right) \quad \text{for all } l \in \{2, \cdots, L\}.$$
 (FF2v)

The output of our neural network for an input \mathbf{x} is given by the neurons in the output layer, i.e. the output vector $\mathbf{o}(\mathbf{x}) \in \mathbb{R}^{m(L)}$ is defined as

$$\mathbf{o}(\mathbf{x}) := \left\langle a_1^{(L)}(\mathbf{x}), \cdots, a_{m(L)}^{(L)}(\mathbf{x}) \right\rangle^\top = \mathbf{a}^{(L)}(\mathbf{x}).$$

Note that the equations (FF1) and (FF2) describe how information propagates through the neural network:

1. Initially, the input vector \mathbf{x} is given and stored in the input layer of the neural network:

$$\mathbf{a}^{(1)}(\mathbf{x}) := \mathbf{x}.$$

2. The first layer of neurons, which is the second layer of nodes, is activated and computes the activation vector $\mathbf{a}^{(2)}$ according to the formula

$$\mathbf{a}^{(2)}(\mathbf{x}) := S(W^{(2)} \cdot \mathbf{a}^{(1)}(\mathbf{x}) + \mathbf{b}^{(2)}) = S(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}).$$

3. The second layer of neurons, which is the third layer of nodes, is activated and computes the activation vector $\mathbf{a}^{(3)}(\mathbf{x})$ according to the formula

$$\mathbf{a}^{(3)}(\mathbf{x}) := S(W^{(3)} \cdot \mathbf{a}^{(2)}(\mathbf{x}) + \mathbf{b}^{(3)}) = S(W^{(3)} \cdot S(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}) + \mathbf{b}^{(3)})$$

4. This proceeds until the output layer is reached and the output

$$\mathbf{o}(\mathbf{x}) := \mathbf{a}^{(L)}(\mathbf{x})$$

has been computed. As long as we use the sigmoid function as our activation function, every neuron of the neural network performs logistic regression.

Next, we assume that we have n training examples

$$\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)} \rangle$$
 for $i = 1, \dots, n$

such that

$$\mathbf{x}^{(i)} \in \mathbb{R}^{m(1)}$$
 and $\mathbf{y}^{(i)} \in \mathbb{R}^{m(L)}$.

Our goal is to choose the weight matrices $W^{(l)}$ and the bias vectors $\mathbf{b}^{(l)}$ in a way such that

$$\mathbf{o}(\mathbf{x}^{(i)}) = \mathbf{y}^{(i)}$$
 for all $i \in \{1, \dots, n\}$.

Unfortunately, in general we will not be able to achieve equality for all $i \in \{1, \dots, n\}$. Therefore, our goal is to minimize the error instead. To be more precise, the quadratic error cost function is defined as

$$C(W^{(2)}, \dots, W^{(L)}, \mathbf{b}^{(2)}, \dots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \dots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}) := \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} (\mathbf{o}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})^{2}.$$

Note that this cost function is additive in the training examples $\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)} \rangle$. In order to simplify the notation we define

$$C_{\mathbf{x},\mathbf{y}}\left(W^{(2)},\cdots,W^{(L)},\mathbf{b}^{(2)},\cdots,\mathbf{b}^{(L)}\right) := \frac{1}{2} \cdot \left(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}\right)^2$$

i.e. $C_{\mathbf{x},\mathbf{y}}$ is the part of the cost function that is associated with a single training example $\langle \mathbf{x}, \mathbf{y} \rangle$. Then we have

$$C\left(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\right) \\ := \frac{1}{n} \cdot \sum_{i=1}^{n} C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}} \left(W^{(2)}, \cdots W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\right).$$

As the notation

$$C_{\mathbf{x},\mathbf{y}}\left(W^{(2)},\cdots,W^{(L)},\mathbf{b}^{(2)},\cdots,\mathbf{b}^{(L)}\right)$$

is far too heavy, we will abbreviate this term as $C_{\mathbf{x},\mathbf{y}}$ in the following discussion of the backpropagation algorithm. Similarly, we abbreviate the quadratic error cost function as C. Our goal is to choose the weight matrices $W^{(l)}$ and the bias vectors $\mathbf{b}^{(l)}$ such that the quadratic error cost function C is minimized. We will use a variation of gradient descent to find this minimum². Unfortunately, the cost function C when regarded as a function of the weights and biases has many local minima. Hence, in practical applications all we can hope for is to find a local minimum that is good enough for the goal that we want to achieve.

8.2 Backpropagation

There are three reasons for the recent success of neural networks.

- 1. The computing power that is available today has vastly increased in the last 20 years. For example, today the NVIDIA RTX 3090 graphic card offers 35 teraflops in single precision performance. It needs a power supply that outputs 750 watt. Contrast this with ASCI White, which was the most powerful supercomputer in 2000: According to the article "History of Supercomputing", it offered a performance of 7.2 teraflops and needed 6 megawatt to operate. The cost to build ASCI White was about 110,000,000 \$\$. To compare, the NVIDIA RTX 3090 costs 1,499 \$\$.
- 2. The breakthrough in the theory of neural networks was the rediscovering of the backpropagation algorithm by David Rumelhart, Geoffrey Hinton, and Ronald Williams [RHW86] in 1986. The backpropagation algorithm had first been discovered by Arthur E. Bryson, Jr. and Yu-Chi Ho [BH69]. In recent years, there have been a number of other theoretical advances that have helped in speeding up the learning algorithms for neural networks.
- 3. Lastly, as neural networks have large sets of parameters, they need large sets of training examples. The recent digitization of our society has made these large data sets available.

Essentially, the backpropagation algorithm is an efficient way to compute the partial derivatives of the cost function C with respect to the weights $w_{j,k}^{(l)}$ and the biases $b_j^{(l)}$. Before we can proceed to compute these partial derivatives, we need to define some auxiliary variables.

8.2.1 Definition of some Auxiliary Variables

We start by defining the auxiliary variables $z_j^{(l)}$. The expressions $z_j^{(l)}$ are defined as the inputs of the activation function S of the j-th neuron in layer l:

$$z_j^{(l)} := \left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)}\right) + b_j^{(l)} \quad \text{ for all } j \in \{1, \cdots, m(l)\} \text{ and } l \in \{2, \cdots, L\}.$$

²In logistic regression we have tried to *maximize* the log-likelihood. Here, instead we *minimize* the quadratic error cost function. Hence, instead of gradient *ascent* we use gradient *descent*.

Of course, the term $a_k^{(l-1)}$ really is a function of the input vector \mathbf{x} . However, it is better to suppress this dependence in the notation since otherwise the formulas get too cluttered. Essentially, $z_j^{(l)}$ is the input to the sigmoid function when the activation $a_i^{(l)}$ is computed, i.e. we have

$$a_j^{(l)} = S\Big(z_j^{(l)}\Big).$$

Later, we will see that the partial derivatives of the cost function $C_{\mathbf{x},\mathbf{y}}$ with respect to both the weights $w_{j,k}^{(l)}$ and the biases $b_j^{(l)}$ can be computed easily if we first compute the partial derivatives of $C_{\mathbf{x},\mathbf{y}}$ with respect to $z_i^{(l)}$. Therefore we define

$$\varepsilon_j^{(l)} := \frac{\partial C_{\mathbf{x}, \mathbf{y}}}{\partial z_j^{(l)}} \quad \text{for all } j \in \{1, \cdots, m(l)\} \text{ and } l \in \{2, \cdots, L\},$$

that is we regard $C_{\mathbf{x},\mathbf{y}}$ as a function of the $z_j^{(l)}$ and take the partial derivatives according to these variables. Note that $\varepsilon_j^{(l)}$ depends on both \mathbf{x} and \mathbf{y} . We call $\varepsilon_j^{(l)}$ the error in the j-th neuron in the l-th layer. Since the notation would get too cumbersome if we would write this as $\varepsilon(\mathbf{x},\mathbf{y})_j^{(l)}$, we regard the training example $\langle \mathbf{x},\mathbf{y} \rangle$ as fixed for now. Next, the quantities $\varepsilon_j^{(l)}$ are combined into a vector:

$$\boldsymbol{\varepsilon}^{(l)} := \left(\begin{array}{c} \varepsilon_1^{(l)} \\ \vdots \\ \varepsilon_{m(l)}^{(l)} \end{array} \right).$$

The vector $\boldsymbol{\varepsilon}^{(l)}$ is called the error in layer l.

8.2.2 The Hadamard Product

Later, we will have need of the Hadamard product of two vectors. Assume that $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. The Hadamard product of \mathbf{x} and \mathbf{y} is a vector that is defined by multiplying the vectors \mathbf{x} and \mathbf{y} elementwise:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \odot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} := \begin{pmatrix} x_1 \cdot y_1 \\ x_2 \cdot y_2 \\ \vdots \\ x_n \cdot y_n \end{pmatrix},$$

i.e. the *i*-th component of the Hadamard product $\mathbf{x} \odot \mathbf{y}$ is the product of the *i*-th component of \mathbf{x} with the *i*-th component of \mathbf{y} . Do not confuse the Hadamard product with the dot product! Although both multiply the vectors componentwise, the Hadamard product returns a vector, while the dot product returns a number. Later, we will use the NumPy package to represent vectors. In NumPy, the Hadamard product of two vectors \mathbf{x} and \mathbf{y} is conveniently computed by the expression $\mathbf{x} * \mathbf{y}$.

8.2.3 Backpropagation: The Equations

Now we are ready to state the backpropagation equations. The first of these four equations reads as follows:

$$\varepsilon_j^{(L)} = \left(a_j^{(L)} - y_j\right) \cdot S'\left(z_j^{(L)}\right) \quad \text{for all } j \in \{1, \cdots, m(L)\},\tag{BP1}$$

where S'(x) denotes the derivative of the sigmoid function. We have shown in Chapter 7 that this derivative satisfies the equation

$$S'(t) = (1 - S(t)) \cdot S(t).$$

The equation (BP1) can also be written in vectorized form using the Hadamard product:

$$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'(\mathbf{z}^{(L)})$$
(BP1v)

Here, we have vectorized the application of the function S' to the vector $\mathbf{z}^{(L)}$, i.e. the expression $S'(\mathbf{z}^{(L)})$ is defined as follows:

$$S'\left(\begin{array}{c}z_1^{(L)}\\\vdots\\z_{m(L)}^{(L)}\end{array}\right):=\left(\begin{array}{c}S'(z_1^{(L)})\\\vdots\\S'(z_{m(L)}^{(L)})\end{array}\right).$$

The next equation computes $\varepsilon_j^{(l)}$ for l < L.

$$\varepsilon_j^{(l)} = \sum_{i=1}^{m(l+1)} w_{i,j}^{(l+1)} \cdot \varepsilon_i^{(l+1)} \cdot S'(z_j^{(l)}) \quad \text{for all } j \in \{1, \dots, m(l)\} \text{ and } l \in \{2, \dots, L-1\}.$$
 (BP2)

This equation is more succinct in vectorized notation:

$$\boldsymbol{\varepsilon}^{(l)} = \left(\left(W^{(l+1)} \right)^{\top} \cdot \boldsymbol{\varepsilon}^{(l+1)} \right) \odot S'(\mathbf{z}^{(l)}) \quad \text{for all } l \in \{2, \cdots, L-1\}.$$
 (BP2v)

Note that this equation computes the error in layer l for l < L in terms of the error in layer l+1: The error $\varepsilon^{(l+1)}$ at layer l+1 is propagated backwards through the neural network to produce the error $\varepsilon^{(l)}$ at layer l. This is the reason for calling the associated algorithm backpropagation.

Next, we have to compute the partial derivative of $C_{\mathbf{x},\mathbf{y}}$ with respect to the bias of the j-th neuron in layer l, which is denoted as $b_i^{(l)}$. We have

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial b_i^{(l)}} = \varepsilon_j^{(l)} \quad \text{for all } j \in \{1,\cdots,m(l)\} \text{ and } l \in \{2,\cdots,L\}. \tag{BP3}$$

This equation shows the reason for defining the error terms $\varepsilon_j^{(l)}$: What we really need to compute is the partial derivative of $C_{\mathbf{x},\mathbf{y}}$ with respect to the biases and the weights. The equation (BP3) and the equation (BP4) below show how this can be done once we have computed the error terms $\varepsilon_j^{(l)}$. In vectorized notation, the equation (BP3) takes the following form:

$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x}, \mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \quad \text{for all } l \in \{2, \cdots, L\}.$$
 (BP3v)

Here, $\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}}$ denotes the gradient of $C_{\mathbf{x},\mathbf{y}}$ with respect to the bias vector $\mathbf{b}^{(l)}$. Finally, we can compute the partial derivative of $C_{\mathbf{x},\mathbf{y}}$ with respect to the weights $w_{i,k}^{(l)}$:

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial w_{j,k}^{(l)}} = \varepsilon_j^{(l)} \cdot a_k^{(l-1)} \quad \text{for all } j \in \{1, \cdots, m(l)\}, \ k \in \{1, \cdots, m(l-1)\}, \ \text{and } l \in \{2, \cdots, L\}.$$
(BP4)

In vectorized notation, this equation can be written as:

$$\nabla_{W^{(l)}} C_{\mathbf{x}, \mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot (\mathbf{a}^{(l-1)})^{\top} \quad \text{for all } l \in \{2, \cdots, L\}.$$
(BP4v)

Here, the expression $\boldsymbol{\varepsilon}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^{\top}$ denotes the matrix product of the column vector $\boldsymbol{\varepsilon}^{(l)}$ that is regarded as an $m(l) \times 1$ matrix and the row vector $\left(\mathbf{a}^{(l-1)}\right)^{\top}$ that is regarded as an $1 \times m(l-1)$ matrix.

As the backpropagation equations are at the very core of the theory of neural networks, we highlight the vectorized form of these equations:

$$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'(\mathbf{z}^{(L)})$$
(BP1v)
$$\boldsymbol{\varepsilon}^{(l)} = \left((W^{(l+1)})^{\top} \cdot \boldsymbol{\varepsilon}^{(l+1)} \right) \odot S'(\mathbf{z}^{(l)}) \text{ for all } l \in \{2, \dots, L-1\}$$
(BP2v)
$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x}, \mathbf{y}} = \boldsymbol{\varepsilon}^{(l)}$$
(BP3v)
$$\nabla_{W^{(l)}} C_{\mathbf{x}, \mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot (\mathbf{a}^{(l-1)})^{\top}$$
for all $l \in \{2, \dots, L\}$ (BP4v)

$$\nabla_{\mathbf{p}(l)} C_{\mathbf{x},\mathbf{v}} = \boldsymbol{\varepsilon}^{(l)}$$
 for all $l \in \{2, \dots, L\}$ (BP3v)

$$\nabla_{W^{(l)}} C_{\mathbf{x}, \mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^{\top} \qquad \text{for all } l \in \{2, \dots, L\}$$
 (BP4v)

The equations (BP3) and (BP4) show why it was useful to introduce the vectors $\boldsymbol{\varepsilon}^{(l)}$: These vectors enable us to compute the partial derivatives of the cost function with respect to both the biases and the weights. The equations (BP1) and (BP2) show how the vectors $\boldsymbol{\varepsilon}^{(l)}$ can be computed. An implementation of backpropagation should use the vectorized versions of these equations since this is more efficient for two reasons:

- 1. Interpreted languages like Python take much more time to execute a loop than to execute a simple matrix-vector multiplication. The reason is that in a loop, in addition to executing the statement a given number of times, the statement has to be interpreted every time it is executed.
- 2. Languages that are optimized for machine learning often take care to delegate the execution of matrix operations to highly optimized functions that have been written in more efficient low level languages like C or assembler. Often, these functions are able to utilize all cores of the processor simultaneously. Furthermore, sometimes these functions can even use the graphical coprocessor which, because of parallelization, can do a matrix multiplication much faster than the floating point unit of a conventional processor.

8.2.4 A Proof of the Backpropagation Equations

Next, we are going to prove the backpropagation equations. Although the proof is a bit tedious, it should be accessible: We only need the chain rule from calculus and the chain rule from multivariable calculus.

Let us start with the proof of equations BP1. Remember that we have defined the numbers $\varepsilon_i^{(l)}$ as

$$\varepsilon_j^{(l)} = \frac{\partial C_{\mathbf{x}, \mathbf{y}}}{\partial z_j^{(l)}},$$

while the numbers $z_i^{(l)}$ have been defined as

$$z_j^{(l)} := \left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)}(\mathbf{x})\right) + b_j^{(l)}.$$

Since the quadratic error cost function $C_{\mathbf{x},\mathbf{y}}$ for the training example $\langle \mathbf{x},\mathbf{y} \rangle$ has been defined in terms of the activation $\mathbf{a}^{(L)}(\mathbf{x})$ as

$$C_{\mathbf{x},\mathbf{y}} = \frac{1}{2} \cdot \left(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}\right)^2$$

and we have $\mathbf{a}^{(L)}(\mathbf{x}) = S(\mathbf{z}^{(L)})$, the chain rule of calculus tells us that $\varepsilon_i^{(L)}$ can be computed as follows:

$$\begin{split} \varepsilon_j^{(L)} &= \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(L)}} \\ &= \frac{\partial}{\partial z_j^{(L)}} \frac{1}{2} \cdot \left(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y} \right)^2 \\ &= \frac{1}{2} \cdot \frac{\partial}{\partial z_j^{(L)}} \sum_{i=1}^{m(L)} \left(a_i^{(L)}(\mathbf{x}) - y_i \right)^2 \\ &= \frac{1}{2} \cdot \frac{\partial}{\partial z_j^{(L)}} \sum_{i=1}^{m(L)} \left(S(z_i^{(L)}) - y_i \right)^2 \\ &= \frac{1}{2} \cdot \sum_{i=1}^{m(L)} 2 \cdot \left(S(z_i^{(L)}) - y_i \right) \cdot \frac{\partial}{\partial z_j^{(L)}} S(z_i^{(L)}) \\ &= \sum_{i=1}^{m(L)} \left(S(z_i^{(L)}) - y_i \right) \cdot S'(z_i^{(L)}) \cdot \frac{\partial z_i^{(L)}}{\partial z_j^{(L)}} \\ &= \sum_{i=1}^{m(L)} \left(S(z_i^{(L)}) - y_i \right) \cdot S'(z_i^{(L)}) \cdot \delta_{i,j} \qquad \delta_{i,j} \text{ denotes the Kronecker delta} \\ &= \left(S(z_j^{(L)}) - y_j \right) \cdot S'(z_j^{(L)}) \\ &= \left(a_j^{(L)} - y_j \right) \cdot S'(z_j^{(L)}) \end{split}$$

Thus we have proven equation BP1.

We proceed to prove equation BP2. To this end we compute $\varepsilon_j^{(l)}$ for l < L. This time, we need the chain rule of multivariate calculus. As a reminder, the chain rule in multivariate calculus works as follows: Assume that the functions

$$f: \mathbb{R}^k \to \mathbb{R}$$
 and $q: \mathbb{R}^n \to \mathbb{R}^k$

are differentiable³. If the function $h: \mathbb{R}^n \to \mathbb{R}$ is defined as

$$h(\mathbf{x}) := f(g(\mathbf{x}))$$
 for all $\mathbf{x} \in \mathbb{R}^n$

then the partial derivative of h with respect to x_j satisfies

$$\frac{\partial h}{\partial x_j} = \sum_{i=1}^k \frac{\partial f}{\partial y_i} \cdot \frac{\partial g_i}{\partial x_j}.$$

We have

$$\begin{split} \varepsilon_j^{(l)} &= \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(l)}} \\ &= \sum_{i=1}^{m(l+1)} \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \quad \text{using the chain rule of multivariate calculus} \\ &= \sum_{i=1}^{m(l+1)} \varepsilon_i^{(l+1)} \cdot \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \quad \text{using the definition of } \varepsilon_i^{(l+1)} \end{split}$$

 $^{^{3}}$ If I had written this text in German, I would have said that f and g are "total differenzierbar".

In order to proceed, we have to remember the definition of $z_i^{(l+1)}$. We have

$$z_i^{(l+1)} = \left(\sum_{k=1}^{m(l)} w_{i,k}^{(l+1)} \cdot S(z_k^{(l)})\right) + b_i^{(l+1)}$$

Therefore, the partial derivatives $\frac{\partial z_i^{(l+1)}}{\partial z_i^{(l)}}$ can be computed as follows:

$$\frac{\partial z_{i}^{(l+1)}}{\partial z_{j}^{(l)}} = \sum_{k=1}^{m(l)} w_{i,k}^{(l+1)} \cdot S'(z_{k}^{(l)}) \cdot \frac{\partial z_{k}^{(l)}}{\partial z_{j}^{(l)}}
= \sum_{k=1}^{m(l)} w_{i,k}^{(l+1)} \cdot S'(z_{k}^{(l)}) \cdot \delta_{k,j}
= w_{i,j}^{(l+1)} \cdot S'(z_{j}^{(l)})$$

If we substitute this expression back into the result we got for $\varepsilon_j^{(l)}$ we have shown the following:

$$\varepsilon_{j}^{(l)} = \sum_{i=1}^{m(l+1)} \varepsilon_{i}^{(l+1)} \cdot \frac{\partial z_{i}^{(l+1)}}{\partial z_{j}^{(l)}}$$

$$= \sum_{i=1}^{m(l+1)} \varepsilon_{i}^{(l+1)} \cdot w_{i,j}^{(l+1)} \cdot S'(z_{j}^{(l)})$$

$$= \sum_{i=1}^{m(l+1)} w_{i,j}^{(l+1)} \cdot \varepsilon_{i}^{(l+1)} \cdot S'(z_{j}^{(l)})$$

Therefore, we have now proven equation (BP2).

We proceed to prove equation (BP4). According to the chain rule we have

$$\frac{\partial C_{\mathbf{x}, \mathbf{y}}}{\partial w_{i,k}^{(l)}} = \frac{\partial C_{\mathbf{x}, \mathbf{y}}}{\partial z_{i}^{(l)}} \cdot \frac{\partial z_{j}^{(l)}}{\partial w_{i,k}^{(l)}}$$

Now by definition of $\varepsilon_j^{(l)}$, the first factor on the right hand side of this equation is equal to $\varepsilon_j^{(l)}$:

$$\varepsilon_j^{(l)} = \frac{\partial C_{\mathbf{x}, \mathbf{y}}}{\partial z_j^{(l)}}.$$

In order to proceed, we need to evaluate the partial derivative $\frac{\partial z_j^{(L)}}{\partial w_{j,k}^{(l)}}$. The term $z_j^{(l)}$ has been defined as follows:

$$z_j^{(l)} = \left(\sum_{k=1}^{m(l)} w_{j,i}^{(l)} \cdot S(z_i^{(l-1)})\right) + b_j^{(l)}.$$

Hence we have

$$\frac{\partial z_{j}^{(l)}}{\partial w_{j,k}^{(l)}} = \sum_{i=1}^{m(l)} \frac{\partial w_{j,i}^{(l)}}{\partial w_{j,k}^{(l)}} \cdot S(z_{i}^{(l-1)})$$

$$= \sum_{i=1}^{m(l)} \delta_{i,k} \cdot S(z_{i}^{(l-1)})$$

$$= S(z_{k}^{(l-1)}) = a_{k}^{(l-1)}.$$

Combining these equations we arrive at

$$\frac{\partial C_{\mathbf{x}, \mathbf{y}}}{\partial w_{i,k}^{(l)}} = a_k^{(l-1)} \cdot \varepsilon_j^{(l)}.$$

Therefore, equation (BP4) has been verified.

Exercise 20: Prove equation (BP3).

8.3 Stochastic Gradient Descent

The equations describing backpropagation describe the gradient of the cost function for a single training example $\langle \mathbf{x}, \mathbf{y} \rangle$. However, when we train a neural network, we need to take all training examples into account. If we have n training examples

$$\langle \mathbf{x}^{(1)}, \mathbf{y}^{(1)} \rangle, \cdots, \langle \mathbf{x}^{(n)}, \mathbf{y}^{(n)} \rangle,$$

then the quadratic error cost function has been previously defined as the sum

$$C(W^{(2)}, \dots, W^{(L)}, \mathbf{b}^{(2)}, \dots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \dots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}) := \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} (\mathbf{o}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})^{2}.$$

In practical applications of neural networks, the number n of training examples is usually big. For example, when we later develop a neural network to classify handwritten digits, we will have 50,000 training examples. More ambitious projects that use neural networks to recognize objects in images use millions of training examples. When we compute the gradient of the quadratic error function with respect to a weight matrix $W^{(l)}$ or a bias vector $b^{(l)}$ we have to compute the sums

$$\frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}}{\partial w_{i,k}^{(l)}} \quad \text{ and } \quad \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}}{\partial b_{i}^{(l)}}$$

over all training examples in order to perform a single step of gradient descent. If n is large, this is computationally costly. Note that these sums can be regarded as computing average values. In stochastic gradient descent, we approximate these sums by randomly choosing a small subset of the training examples. In order to formulate this approximation in a convenient notation, let us assume that instead of using all n training examples, we just use the first m training examples. Then we approximate the sums shown above as follows:

$$\frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}}{\partial w_{j,k}^{(l)}} \approx \frac{1}{2 \cdot m} \cdot \sum_{i=1}^{m} \frac{\partial C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}}{\partial w_{j,k}^{(l)}} \quad \text{ and } \quad \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}}{\partial b_{j}^{(l)}} \approx \frac{1}{2 \cdot m} \cdot \sum_{i=1}^{m} \frac{\partial C_{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}}}{\partial b_{j}^{(l)}},$$

i.e. we approximate these sums by the average value of their first m training examples. Of course, in general we will not choose the first m training examples but rather we will choose m random training examples. The randomness of this choice is the reason this algorithm is called stochastic gradient descent. It turns out that if we take care that eventually all training examples are used during gradient descent, then the approximations given above can speed up the learning of neural networks substantially.

 \Diamond

8.4 Implementation

Next, we will take a look at a neural network that is able to recognize handwritten digits. The MNIST database of handwritten digits contains $70\,000$ images of handwritten digits. These images have a size of 28×28 pixels. Figure 8.3 shows the first 18 images.



Figure 8.3: The first 18 images of the MNIST dataset.

The 70,000 images are divided into three groups:

- 1. The first group contains 50,000 images and is designated as the training set.
- 2. The second group contains 10,000 images and is designated as the validation set.
- 3. The last group contains 10,000 images and is designated as the test set.

We will use the first 50 000 images to train a neural network, while the last group of 10 000 images will be used to check the accuracy of the trained network. As we do not use any regularization, we will not use the validation set. As a matter of convenience, the images have been converted into one large pickled zip files. You can download this file at the following address:

https://github.com/karlstroetmann/Artificial-Intelligence/raw/master/Python/mnist.pkl.gz

Next, we describe a *Python* program that loads these images, trains a neural network on it, and finally evaluates the accuracy of the network. The program is shown in the Figures 8.4, 8.5, 8.6, and 8.7.

The code on Figure 8.4 on page 214 shows the code to load the image files. We discuss it line by line.

- 1. Since the images have been compressed as a .gz file, we need the module gzip to uncompress the file.
- 2. The format that has been used to store the images is called pickle. This is a binary format that can be used to serialize *Python* objects into binary strings. These strings can then be stored as files and later be used to restore the corresponding *Python* objects. In order to read pickled objects, we import the module pickle.

```
import gzip
     import pickle
2
     import numpy
3
                               as np
     import matplotlib.pyplot as plt
4
     import random
5
6
    def vectorized_result(d):
              = np.zeros((10, 1), dtype=np.float32)
         e[d] = 1.0
         return e
10
11
    def load_data():
12
         with gzip.open('mnist.pkl.gz', 'rb') as f:
13
             train, validate, test = pickle.load(f, encoding="latin1")
14
                             = [np.reshape(x, (784, 1)) for x in train[0]]
         training_inputs
15
                             = [vectorized_result(y) for y in train[1]]
         training_results
                             = list(zip(training_inputs, training_results))
         training_data
17
         test_inputs
                             = [np.reshape(x, (784, 1)) for x in test[0]]
18
                             = list(zip(test_inputs, test[1]))
         test_data
19
         return training_data, test_data
20
^{21}
    training_data, test_data = load_data()
22
```

Figure 8.4: Code to load the image files.

- 3. The images of the handwritten digits that we are going to import have a size of 28×28 pixels. In this program, the images are stored as numpy arrays of size $28 \cdot 28 = 784$.
- 4. In order to be able to display these images, we import matplotlib.
- 5. Every image of a handwritten character is associated with a number $d \in \{0, \dots, 9\}$. We need to transform these numbers into the expected output of our neural network. This neural network will have 10 output nodes corresponding to these digits. The k-th output node will be 1 if the neural network has recognized the digit k, while all other output nodes will be 0.

The function vectorized_result(d) takes a digit $d \in \{0, \dots, 9\}$ and returns a number array e of shape (10, 1) such that e[d][0] = 1 and e[j][0] = 0 for $j \neq d$. For example, we have

```
[0.]], dtype=float32)
```

For reasons of efficiency we will only use single precision floating point numbers.

- 6. The function load_data reads the file mnist.pkl.gz, uncompresses it, and returns three lists:
 - (a) training_data stores the first 50 000 images. The images are stored as pairs (x, y): x is a number array of shape (784, 1) and y is a number array of shape (10, 1).
 - (b) test_data holds the remaining 10 000 images. test_data is a list containing 10,000 pairs of the form (\mathbf{x}, y) , where \mathbf{x} is a 784-dimensional number array containing the input image, and $y \in \{0, \dots, 9\}$ is the corresponding digit value.

Note that the formats for training data and test data are different. For the training data y is a vector, but for the test data y is a number.

```
def rndMatrix(rows, cols):
23
         return np.random.randn(rows, cols) / np.sqrt(cols)
24
25
     def sigmoid(x):
26
         return 1.0 / (1.0 + np.exp(-x))
27
28
     def sigmoid_prime(x):
29
         s = sigmoid(x)
30
         return s * (1 - s)
31
```

Figure 8.5: The constructor of the class network.

Figure 8.5 on page 215 shows the implementation of three utility functions.

- 1. The function $\mathtt{rndMatrix}(r,c)$ creates a matrix of shape (r,c) that is filled with random numbers. These numbers have a Gaussian distribution with mean 0 and variance 1/r. This function is used to initialize the weight matrices. It is important that not all weights are initialized to the same number, for otherwise they would stay the same and then the different neurons would effectively all calculate the same feature instead of different features. It is also important that the weights are not too big for otherwise the associated neurons would saturate and training would be very slow.
- 2. The function sigmoid(x) computes the sigmoid function. If x is a number, sigmoid(x) computes

$$S(x) := \frac{1}{1 + \exp(-x)}$$

If **x** is a vector of the form $\mathbf{x} = (x_1, \dots, x_n)^{\top}$, we have

$$S(\mathbf{x}) = (S(x_1), \cdots, S(x_n))^{\top}.$$

3. The function $sigmoid_prime(x)$ computes the derivative of the sigmoid function. The implementation is based on the equation:

$$S'(x) = S(x) \cdot (1 - S(x))$$

where x can either be a number or an array.

```
class Network(object):
         def __init__(self, hiddenSize):
33
             self.mInputSize = 28 * 28
34
             self.mHiddenSize = hiddenSize
35
             self.mOutputSize = 10
36
             self.mBiasesH
                               = np.zeros((self.mHiddenSize, 1))
37
             self.mBiasesO
                               = np.zeros((self.mOutputSize, 1))
             self.mWeightsH
                               = rndMatrix(self.mHiddenSize, self.mInputSize)
                               = rndMatrix(self.mOutputSize, self.mHiddenSize)
             self.mWeightsO
40
41
    def feedforward(self, x):
42
         AH = sigmoid(self.mWeightsH @ x + self.mBiasesH)
43
         AO = sigmoid(self.mWeightsO @ AH + self.mBiasesO)
44
         return AO
45
46
    def sgd(self, training_data, epochs, mbs, eta, test_data):
         n_test = len(test_data)
48
                = len(training_data)
49
         for j in range(epochs):
50
             random.shuffle(training_data)
51
             mini_batches = [training_data[k : k+mbs] for k in range(0, n, mbs)]
             for mini_batch in mini_batches:
53
                 self.update_mini_batch(mini_batch, eta)
             print('Epoch %2d: %d / %d' % (j, self.evaluate(test_data), n_test))
55
56
    def update_mini_batch(self, mini_batch, eta):
57
         nabla_BH = np.zeros((self.mHiddenSize, 1))
58
         nabla_BO = np.zeros((self.mOutputSize, 1))
59
         nabla_WH = np.zeros((self.mHiddenSize, self.mInputSize))
60
         nabla_WO = np.zeros((self.mOutputSize, self.mHiddenSize))
         for x, y in mini_batch:
             dltNbl_BH, dltNbl_BO, dltNbl_WH, dltNbl_WO = self.backprop(x, y)
             nabla_BH += dltNbl_BH
64
             nabla_BO += dltNbl_BO
65
             nabla_WH += dltNbl_WH
66
             nabla_WO += dltNbl_WO
67
         alpha = eta / len(mini_batch)
         self.mBiasesH -= alpha * nabla_BH
         self.mBiasesO -= alpha * nabla_BO
         self.mWeightsH -= alpha * nabla_WH
71
         self.mWeightsO -= alpha * nabla_WO
72
```

Figure 8.6: The class Network, part I.

Figure 8.6 shows the first part of the class Network. This class represents a neural network with one hidden layer.

- 1. The member variable mInputSize specifies the number of input nodes. The neural network for the recognition of handwritten digits has 784 inputs. These inputs are the grey values of the 28 × 28 pixels that constitute the image of the handwritten digit.
- 2. The member variable mHiddenSize, specifies the number of neurons in the hidden layer. We assume that there is only one hidden layer. I have experimented with 30 neurons, 40 neurons, 60 neurons, and 100 neurons.
 - (a) For 30 neurons, the trained neural network achieved an accuracy of 94.8%.
 - (b) For 60 neurons, the network achieved an accuracy of 96.1%.
 - (c) If there are 100 neurons in the hidden layer, the network achieved an accuracy of 97.8%. For 100 neurons, the number of weights in the hidden layer is $784 \cdot 100 = 78400$. Therefore, the number of weights is greater than the number of training examples. Hence, we should really use regularization in order to prevent over-fitting and increase the accuracy of the network.
- 3. The argument mOutputSize specifies the number of output neurons. For the neural network recognizing handwritten digits this number is 10 since there is an output neuron for every digit.
- 4. Besides storing the topology of the neural network, the class Network stores the biases and weights of all the neurons. The weights are initialized as random numbers.
 - (a) mBiasesH stores the bias vector of the hidden layer.
 - (b) mBiasesO stores the bias vector of the output layer.
 - (c) mWeightsH stores the weight matrix $W^{(2)}$, which specifies the weights connecting the input layer with the hidden layer.
 - (d) mWeightsO stores the weight matrix $W^{(3)}$, which specifies the weights connecting the hidden layer with the output layer.
- 5. The function feedforward receives an image **x** of a digit that is stored as a vector of shape (784,1) and computes the output of the neural network for this image. The code is a straightforward implementation of the equations (FF1v) and (FF2v). These equations are repeated here for convenience:

(a)
$$\mathbf{a}^{(1)}(\mathbf{x}) = \mathbf{x}$$
 (FF1v)

(b)
$$\mathbf{a}^{(l)}(\mathbf{x}) = S\left(W^{(l)} \cdot \mathbf{a}^{(l-1)}(\mathbf{x}) + \mathbf{b}^{(l)}\right)$$
 for all $l \in \{2, \dots, L\}$. (FF2v)

- 6. The method sgd implements stochastic gradient descent. It receives 5 arguments.
 - (a) training_data is a list of pairs of the form $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$. Here, $\mathbf{x}^{(i)}$ is a vector of dimension 784. This vector contains the pixels of an image showing one of the handwritten digits from the training set. $\mathbf{y}^{(i)}$ is the one-hot encoding of the digit that is shown in the image $\mathbf{x}^{(i)}$.
 - (b) epochs is the number of iterations of gradient descent.
 - (c) mbs is the size of the mini-batches that are used in stochastic gradient descent. I have achieved the fastest learning when I have used a mini-batch size of 10. Using a mini-batch size of 20 was slightly slower, but this parameter seems is not critical.
 - (d) eta is the learning rate.
 - (e) test_data is the list of test data. These data are only used to check the accuracy after every epoch, they are not used to determine the weights or biases.

The implementation of stochastic gradient descent executes a for-loop that runs epoch number of times. At the beginning of each iteration, the training data are shuffled randomly. Next, the data is chopped up into chunks of size mbs. These chunks are called mini-batches. The inner for-loop iterates over all mini-batches and executes one step of gradient descent that only uses the data from the given mini-batch. At the end of each iteration of the outer for-loop, the accuracy of the current version of the neural net is printed.

- 7. The method update_mini_batch performs one step of gradient descent for the data from one mini-batch. It receives two arguments.
 - (a) mini_batch is the list of training data that constitute one mini-batch.
 - (b) eta is the learning rate.

The implementation of update_mini_batch works as follows:

- (a) First, we initialize the vectors nabla_BH, nabla_BO and the matrices nabla_WH, nabla_WO to contain only zeros.
 - (a) nabla_BH will store the gradient of the bias vector of the hidden layer.
 - (b) nabla_BO will store the gradient of the bias vector of the output layer.
 - (c) nabla_WH will store the gradient of the weight matrix of the hidden layer.
 - (d) nabla_WO will store the gradient of the weight matrix of the output layer.
- (b) Next, we iterate of all training examples in the mini-batch and for every training example [x,y] we compute the contribution of this training example to the gradients of the cost function C, i.e. we compute

$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x}, \mathbf{y}}$$
 and $\nabla_{W^{(l)}} C_{\mathbf{x}, \mathbf{y}}$

for the hidden layer and the output layer. These gradients are computed by the function backprop.

(c) Finally, the bias vectors and the weight matrices are updated according to the learning rate and the computed gradients.

The method backprop that is shown in Figure 8.7 computes the gradients of the bias vectors and the weight matrices with respect to a single training example $\langle x, y \rangle$. The implementation of backprop proceeds as follows:

1. First, the vector ZH is computed according to the formula

$$\mathbf{z}^{(2)} = W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}.$$

Here, $W^{(2)}$ is the weight matrix of the hidden layer that is stored in mWeightsH, while $\mathbf{b}^{(2)}$ is the bias vector of the hidden layer. This vector is stored in mBiasesH.

- 2. The activation of the neurons in the hidden layer AH is computed by applying the sigmoid function to the vector $\mathbf{z}^{(2)}$.
- 3. Next, the vector ZH is computed according to the formula

$$\mathbf{z}^{(3)} = W^{(3)} \cdot \mathbf{x} + \mathbf{b}^{(3)}.$$

Here, $W^{(3)}$ is the weight matrix of the output layer that is stored in mWeightsO, while $\mathbf{b}^{(3)}$ is the bias vector of the output layer. This vector is stored in mBiasesO.

```
def backprop(self, x, y):
             # feedforward pass
74
             ZH = self.mWeightsH @ x + self.mBiasesH
75
             AH = sigmoid(ZH)
76
             Z0 = self.mWeights0 @ AH + self.mBiases0
             AO = sigmoid(ZO)
78
             # backwards pass, output layer
             epsilonO = (AO - y) # * sigmoid_prime(ZO)
             nabla_B0 = epsilon0
81
             nabla_WO = epsilonO @ AH.transpose()
82
             # backwards pass, hidden layer
83
             epsilonH = (self.mWeightsO.transpose() @ epsilonO) * sigmoid_prime(ZH)
             nabla_BH = epsilonH
85
             nabla_WH = epsilonH @ x.transpose()
86
             return (nabla_BH, nabla_BO, nabla_WH, nabla_WO)
         def evaluate(self, test_data):
89
             test_results = \
90
                  [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]
91
             return sum(int(y1 == y2) for (y1, y2) in test_results)
92
         # end of class Network
93
94
    net = Network(40)
    net.sgd(training_data, 60, 10, 0.1, test_data)
96
```

Figure 8.7: The class Network, part II.

4. The activation of the neurons in the output layer AO is computed by applying the sigmoid function to the vector $\mathbf{z}^{(3)}$.

These four step constitute the forward pass of backpropagation.

5. Next, the error in the output layer epsilon0 is computed using the backpropagation equation (BP1v)

$$\boldsymbol{\varepsilon}^{(3)} = (\mathbf{a}^{(3)} - \mathbf{y}) \odot S'(\mathbf{z}^{(3)}).$$

6. According to equation (BP3v), the gradient of the cost function with respect to the bias vector of the output layer is given as

$$\nabla_{\mathbf{b}^{(3)}} C_{\mathbf{x}, \mathbf{v}} = \boldsymbol{\varepsilon}^{(3)}.$$

This gradient is stored in the variable nabla_BO.

7. According to equation (BP4v), the gradient of the cost function with respect to the weight matrix of the output layer is given as

$$\nabla_{W^{(3)}} C_{\mathbf{x},\mathbf{v}} = \boldsymbol{\varepsilon}^{(3)} \cdot (\mathbf{a}^{(2)})^{\top}.$$

This gradient is stored in the variable nabla_WO.

8. Next, the error in the hidden layer epsilonH is computed using the backpropagation equation (BP2v)

$$\boldsymbol{\varepsilon}^{(2)} = \left(\left(W^{(3)} \right)^\top \cdot \boldsymbol{\varepsilon}^{(3)} \right) \odot S' \big(\mathbf{z}^{(2)} \big).$$

- 9. Finally, the gradients of the cost function with respect to the bias vector and the weight matrix of the hidden layer are computed. This is completely analogous to the computation of the corresponding gradients of the output layer.
- 10. The method evaluate is used to evaluate the accuracy of the neural network on the test data.
- 11. Finally, we see how the computation can be started by creating an object of class Network and then calling the method sgd.

The program discussed in this section is available as a jupyter notebook at the following link

Digit-Recognition-NN.ipynb.

The file is located in the directory Python/Neural Networks in my github repository for this lecture.

8.5 Automatic Differentiation

All modern libraries for Neural networks, i.e. TensorFlow and PyTorch make heavy use of automatic differentiation. Automatic differentiation is a technique for computing the gradient of a function that does neither rely on numeric approximation nor does it force us to compute symbolic derivatives manually. In fact, the technique is one of the major methodological breakthroughs in machine learning in particular and science and engineering in general in recent years. Although the idea was first published in 1964 by R. E. Wengert [Wen64], it has only been widely understood and accepted in recent years, cf. Baydin et. al. [BPRS18].

There are two modes of automatic differentiation: Forward mode and reverse mode. Forward mode is quite inefficient when the number n of input variables is big as it needs to traverse the computational graph n+1 times (once to compute the values and n times to compute the partial derivatives), while reverse mode needs to traverse the computational graph just twice. Hence for big values of n only reverse mode automatic differentiation is a viable option. We proceed to define the crucial notion of a computational graph.

Definition 38 (Computational Graph) A computational graph is a list of computational nodes. There are four types of computational nodes:

1. A variable node is tuple of length 1 of the form

$$\langle x, \rangle$$

where x is a variable from the set of variables $\{x_1, \dots, x_k\}$. This node represents the given input variable.

2. A constant node is a pair of the form

$$\langle n, r \rangle$$

where n is the name of the node and r is a floating point number. This node is interpreted as the assignment

$$n := r$$
.

The name n is a string that can be understood as the name of an auxiliary variable.

3. A unary node is a tuple of the form

$$\langle n, f, a \rangle$$

where n is the name of the node, f is an unary function symbol from the set

$$\{$$
sqrt, exp, ln , sin, cos, arctan $\}$

and \boldsymbol{a} is the name of another node occurring in the list. This node is interpreted as the assignment

$$n := f(a)$$
.

4. A binary node is a tuple of the form

$$\langle n, o, a_1, a_2 \rangle$$

where n is the name of the node, o is a binary operator from the set

$$\{+,-,*,/\}$$

and a_1 and a_2 are the names of other computational nodes.

This node is interpreted as the assignment

$$n := a_1 \ o \ a_2.$$

The name n is a string that can be understood as the name of an auxiliary variable.

Figure 8.8: A Computational Graph for $\sin(x_1 + x_2) \cdot \cos(x_1 - x_2) + (x_1 + x_2) \cdot (x_1 - x_2)$.

Example: Figure 8.8 shows a computational graph for the expression

$$\sin(x_1+x_2)\cdot\cos(x_1-x_2)+(x_1+x_2)\cdot(x_1-x_2).$$

This computational graph uses the variables x1 and x2. Figure 8.9 shows a rendering of this computational graph.

Definition 39 (admissible) A computational graph G is admissible if for every node of the form

$$\langle n, o, a_1, a_2 \rangle$$

that occurs in the list G there are nodes labelled with a_1 and a_2 that occur in the list G before this node and for every node of the form

$$\langle n, f, a \rangle$$



Figure 8.9: Rendering of the computation graph shown in Figure 8.8.

that occurs in the list G there is a node labelled with a that occurs in the list G before this node. If a computational graph is admissible, the nodes can be evaluated in the same order as they are listed in G.

```
def eval_graph(CG, Values):
         for node in CG:
2
             match node:
3
                  case (v, ):
                      pass
                  case (v, r):
                      Values[v] = r
                  case (v, '+', a1, a2):
8
                      Values[v] = Values[a1] + Values[a2]
9
                  case (v, '-', a1, a2):
10
                      Values[v] = Values[a1] - Values[a2]
11
                  case (v, '*', a1, a2):
12
                      Values[v] = Values[a1] * Values[a2]
                  case (v, '/', a1, a2):
                      Values[v] = Values[a1] / Values[a2]
15
                  case (v, 'sqrt', a):
16
                      Values[v] = math.sqrt(Values[a])
17
                  case (v, 'exp', a):
18
                      Values[v] = math.exp(Values[a])
19
                  case (v, 'log', a):
20
                      Values[v] = math.log(Values[a])
21
                  case (v, 'sin', a):
22
                      Values[v] = math.sin(Values[a])
23
                  case (v, 'cos', a):
24
                      Values[v] = math.cos(Values[a])
25
                  case (v, 'atan', a):
26
                      Values[v] = math.atan(Values[a])
27
         return Values['y']
28
```

Figure 8.10: A function that evaluates a computational graph.

 \Diamond

In order to evaluate an admissible computational graph that contains n variables, we will assume that the first n nodes are labelled with the variables x_1, \dots, x_n and that the last node in a computational node is labelled with the name y. Furthermore, we need a dictionary Values that assigns a value to each of the variables x_1, \dots, x_n . Then the function eval_graph that is shown in Figure 8.10 can be used to evaluate the nodes of the computational graph CG one by one. The idea is that initially the dictionary Values maps all variables to floating point values. Then the nodes of the computational graph are evaluated one by one. For example, if a node of the form

```
\langle v, +, a_1, a_2 \rangle
```

has to be evaluated, then we can assume that the nodes that are labelled with a_1 and a_2 have already been evaluated and that their values are stored in the dictionary Values. These values are added and the resulting value is stored under the key v in the dictionary Values.

In the following we will assume that all computational graphs are admissible. The crucial definition in the theory of reverse mode automatic differentiation is the notion of an adjoint, which will be given later after we have defined to notion of a parent of a node.

```
def parents(CG):
1
          Parents = {}
2
          for node in CG:
3
              match node:
4
                   case (p, _, a):
                       add_to_dictionary(Parents, a, p)
                   case (p, _, a1, a2):
                       add_to_dictionary(Parents, a1, p)
8
                       add_to_dictionary(Parents, a2, p)
9
          return Parents
10
11
     def add_to_dictionary(D, key, value):
12
          if key in D:
13
              D[key] \mid = \{ value \}
14
          else:
15
              D[key] = { value }
16
17
     def node_dictionary(CG):
18
         D = \{\}
19
          for node in CG:
20
                       = node[0]
              name
21
              D[name] = node
22
          return D
23
```

Figure 8.11: Auxiliary functions.

Definition 40 (Parent) If G is a computational graph and $\langle v, o, a_1, a_2 \rangle$ is a node in G, then v is a parent of the nodes that are labelled with a_1 and a_2 . Furthermore, if $\langle v, f, a \rangle$ is a node in G, then v is a parent of the node that is labelled with a.

Figure 8.11 shows the implementation of the function parents that can be used to compute the

parents of a node. It also contains the auxiliary function node_dictionary that takes a computational graph CG as its argument and returns a dictionary associating every node with its name.

Definition 41 (Adjoint) Assume G is a computational graph such that the last node is labelled with then name y. If v is any node in G, then the adjoint of v, which is written as \bar{v} , is defined as the partial derivative of the output variable y w.r.t. v, i.e.

$$\bar{v} := \frac{\partial \mathsf{y}}{\partial v}.$$

The next theorem is an immediate consequence of the multivariable chain rule.

Theorem 42 Assume v is a node of a computational graph G and that p_1, \dots, p_k are all the parents of this node in G. Then the adjoint \bar{v} of the node v is given as

$$\bar{v} = \frac{\partial y}{\partial v} = \sum_{i=1}^{k} \frac{\partial y}{\partial p_i} \cdot \frac{\partial p_i}{\partial v} = \sum_{i=1}^{k} \bar{p}_i \cdot \frac{\partial p_i}{\partial v}.$$

Example: To keep things simple, assume that the variables x_1 and x_2 that are shown in the computational graph in Figure 8.9 are both initialized with the value $\pi/4$. Before the adjoints can be computed, we have to compute the values associated with the nodes. These are as follows:

- 1. $v_1 = \pi/2$,
- 2. $v_2 = 0$,
- 3. $v_3 = 1$,
- 4. $v_4 = 1$,
- 5. $v_5 = 1$,
- 6. $v_6 = 0$.
- 7. y = 1.

Next, we compute the adjoints.

1.
$$\bar{y} = \frac{\partial y}{\partial y} = 1$$
,

$$2. \ \bar{\mathtt{v}}_6 = \frac{\partial \mathtt{y}}{\partial \mathtt{v}_6} = 1,$$

$$3. \ \bar{\mathtt{v}}_5 = \frac{\partial \mathtt{y}}{\partial \mathtt{v}_5} = 1.$$

4.
$$\bar{\mathbf{v}}_4 = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_5} \cdot \frac{\partial \mathbf{v}_5}{\partial \mathbf{v}_4} = \bar{\mathbf{v}}_5 \cdot \mathbf{v}_3 = 1 \cdot 1 = 1.$$

5.
$$\bar{\mathbf{v}}_3 = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_5} \cdot \frac{\partial \mathbf{v}_5}{\partial \mathbf{v}_3} = \bar{\mathbf{v}}_5 \cdot \mathbf{v}_4 = 1 \cdot 1 = 1.$$

6.
$$\bar{\mathbf{v}}_2 = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_6} \cdot \frac{\partial \mathbf{v}_6}{\partial \mathbf{v}_2} + \frac{\partial \mathbf{y}}{\partial \mathbf{v}_4} \cdot \frac{\partial \mathbf{v}_4}{\partial \mathbf{v}_2} = \bar{\mathbf{v}}_6 \cdot \mathbf{v}_1 - \bar{\mathbf{v}}_4 \cdot \sin(\mathbf{v}_2) = 1 \cdot \pi/2 - 1 \cdot \sin(0) = \pi/2 - 1 \cdot 0 = \pi/2.$$

7.
$$\bar{\mathbf{v}}_1 = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_6} \cdot \frac{\partial \mathbf{v}_6}{\partial \mathbf{v}_1} + \frac{\partial \mathbf{y}}{\partial \mathbf{v}_3} \cdot \frac{\partial \mathbf{v}_3}{\partial \mathbf{v}_1} = \bar{\mathbf{v}}_6 \cdot \mathbf{v}_2 + \bar{\mathbf{v}}_3 \cdot \cos(\mathbf{v}_1) = 1 \cdot 0 + 1 \cdot \cos(\pi/2) = 0 + 0 = 0.$$

$$8. \ \bar{\mathbf{x}}_1 = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_1} \cdot \frac{\partial \mathbf{v}_1}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{y}}{\partial \mathbf{v}_2} \cdot \frac{\partial \mathbf{v}_2}{\partial \mathbf{x}_1} = \bar{\mathbf{v}}_1 \cdot 1 + \bar{\mathbf{v}}_2 \cdot 1 = 0 \cdot 1 + \pi/2 \cdot 1 = \pi/2.$$

9.
$$\bar{\mathbf{x}}_2 = \frac{\partial \mathbf{y}}{\partial \mathbf{v}_1} \cdot \frac{\partial \mathbf{v}_1}{\partial \mathbf{x}_2} + \frac{\partial \mathbf{y}}{\partial \mathbf{v}_2} \cdot \frac{\partial \mathbf{v}_2}{\partial \mathbf{x}_2} = \bar{\mathbf{v}}_1 \cdot 1 + \bar{\mathbf{v}}_2 \cdot (-1) = 0 \cdot 1 + \pi/2 \cdot (-1) = -\pi/2.$$

Hence we have shown the following:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}_1} \left(\frac{\pi}{4}, \frac{\pi}{4} \right) = \frac{\pi}{2}$$
 and $\frac{\partial \mathbf{y}}{\partial \mathbf{x}_2} \left(\frac{\pi}{4}, \frac{\pi}{4} \right) = -\frac{\pi}{2}$.

Note that we have found the exact partial derivatives for a specific point, namely for the arguments $x_1 = \pi/4$ and $x_2 = \pi/4$. Automatic differentiation is not symbolic differentiation and hence is not able to derive general formulas but rather computes values for specific arguments. However, these values are not numerical approximations but are, instead, exact.

Of course, we do not want to perform computations like the following ourselves. The function partial_derivative shown in Figure 8.12 takes a computational Node and computes the partial derivative of this node with respect to the given argument arg. The last argument Values is a dictionary containing the values that are associated with the different nodes.

```
def partial_derivative(Node, arg, Values):
         match Node:
2
              case n, '+', a1, a2:
3
                  if arg == a1 == a2:
4
                       return 2
5
                  if arg == a1 or arg == a2:
6
                       return 1
              case n, '-', a1, a2:
                  if arg == a1 == a2:
                       return 0
10
                  if arg == a1:
11
                       return 1
12
                  if arg == a2:
13
                       return -1
14
              case n, '*', a1, a2:
15
                  if arg == a1 == a2:
16
                       return 2 * Values[a1]
17
                  if arg == a1:
18
                      return Values[a2]
19
                  if arg == a2:
20
                      return Values[a1]
^{21}
              case n, '/', a1, a2:
22
                  if arg == a1 == a2:
                       return 0
24
                  if arg == a1:
25
                      return 1 / Values[a2]
26
                  if arg == a2:
27
                       return -Values[a1] / Values[a2] ** 2
28
              case n, 'sqrt', a:
29
                  return 0.5 / math.sqrt(Values[a])
31
              case n, 'exp', a:
                  return math.exp(Values[a])
32
              case n, 'log', a:
33
                  return math.log(Values[a])
34
              case n, 'sin', a:
35
                  return math.cos(Values[a])
36
              case n, 'cos', a:
                  return -math.sin(Values[a])
39
              case n, 'atan', a:
                  return 1 / (1 + Values[a]**2)
40
```

Figure 8.12: Computing the partial derivative of a node.

The function adjoints shown in Figure 8.13 computes the adjoints of a given computational graph. It needs a dictionary Values that maps the variables x_1, \dots, x_n to their values. It returns a dictionary that associates all node names with their adjoints.

```
def adjoints(CG, Values):
         eval_graph(CG, Values)
2
         NodeDict = node_dictionary(CG)
         Parents = parents(CG)
                  = len(CG)
         Adjoints = {}
         Adjoints['y'] = 1
         for k in range(2, n+1):
             Node
                    = CG[-k]
9
                    = Node[0]
             name
10
             result = 0
11
             for parent_name in Parents[name]:
                 parent_node = NodeDict[parent_name]
13
                              = partial_derivative(parent_node, name, Values)
14
                 result += Adjoints[parent_name] * pd
15
             Adjoints[name] = result
16
         return Adjoints
17
```

Figure 8.13: Computing the adjoints of a computational graph.

Bibliography

- [BH69] Arthur Earl Bryson, Jr. and Yu-Chi Ho. Applied Optimal Control: Optimization, Estimation, and Control. Blaisdell Pub., Waltham, Massachusetts, 1969.
- [BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [BPRS18] Atılım Güneş Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1–43, 2018.
- [CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
- [Dec03] Rina Dechter. Constraint Processing. Morgan Kaufmann, 2003.
- [HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC4, 4(2):100–107, 1968.
- [HNR72] Peter Hart, Nils Nilsson, and Bertram Raphael. Correction to "A formal basis for the heuristic determination of minimum cost paths". SIGART Newsletter, 37:28–29, 1972.
- [JWHT14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. An Introduction to Statistical Learning: with Applications in R. Springer, 2014.
- [KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, Computational Problems in Abstract Algebra, pages 263–297. Pergamon Press, 1970.
- [KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
- [Kor85] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27:97–109, 1985.
- [Kow17] Alexandre Kowalczyk. Support Vector Machines Succinctly. Syncfusion, 2017.
- [MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. ACM Transactions on Programming Languages and Systems (TOPLAS), 4(2):258–282, 1982.
- [Nie19] Michael A. Nielsen. Neural Networks and Deep Learning. Determination Press, 2019.

BIBLIOGRAPHY BIBLIOGRAPHY

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1, pages 318–362. MIT Press, 1986.

- [RN20] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Pearson Education Limited, 4rd edition, 2020.
- [Sam59] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):535–554, 1959.
- [SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. Nature, 529:484, 2016.
- [Sny05] Jan A. Snyman. Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms. Springer Publishing, 2005.
- [Wen64] R.E. Wengert. A simple automatic derivative evaluation program. Communications of the ACM, 7(8):463–464, 1964.

List of Figures

2.1	Start state of the missionaries-and-infidels problem
2.2	The missionary and cannibals problem coded as a search problem
2.3	A graphical representation of the missionaries and cannibals problem
2.4	The 3×3 sliding puzzle
2.5	The 3×3 sliding puzzle
2.6	Breadth first search
2.7	The function path_to
2.8	A solution of the missionaries and cannibals problem
2.9	A queue based implementation of breadth first search
2.10	The depth first search algorithm
2.11	A recursive implementation of depth first search
2.12	Iterative deepening implemented in <i>Python</i>
2.13	Example for possible paths in a graph
2.14	Example of space usage of conventional and bidrectional-BFS
2.15	Bidirectional breadth first search
2.16	Combining two paths
2.17	Visual demonstration of a heuristic function
2.18	The Manhattan distance between two states
2.19	The best first search algorithm
	The A* search algorithm
2.21	A path based implementation of A* search
	Bidirectional A* search
2.23	A start state and a goal state for the 4×4 sliding puzzle
2.24	Iterative deepening A^* search
2.25	The A*-IDA* search algorithm, part I
	The A*-IDA* search algorithm, part II
	A solution of the eight queens puzzle
3.1	A map of Australia
3.2	A solution of the eight queens puzzle
3.3	A partial solution of the eight queens puzzle
3.4	The n queens problem formulated as a Csp
3.5	Solving a CSP via brute force search
3.6	Auxiliary functions for brute force search
3.7	A backtracking CSP solver
3.8	A backtracking CSP solver: The function backtrack_search 60
3.9	The definition of the function is_consistent
3.10	The function collectVars

LIST OF FIGURES

LIST OF FIGURES

	Constraint Propagation
3.12	Implementation of solve_unary
3.13	Implementation of backtrack_search
3.14	Finding a most constrained variable
3.15	Finding the least constraining value
3.16	Computing the shrinkage number
	Constraint Propagation
	A cryptarithmetic puzzle
	Formulating "SEND + MORE = MONEY" as a CSP
	Consistency maintenance in <i>Python</i>
	The implementation of exists_value
	A constraint solver with consistency checking as a preprocessing step
	A constraint solver using local search
	Implementation of local search
	The function numConflicts
	Solving a simple text problem with Z3
	Solving a simple text problem
	The moves of a knight, courtesy of chess.com
	The Knight's Tour: Computing the constraints
3.30	The function solve
4.1	A Python implementation of tic-tac-toe
4.1	Tic-Tac-Toe implemented by a bitboard
4.2	
	The Minimax algorithm
4.4	Memoization
4.5	The function play_game
4.6	α - β -Pruning
4.7	Implementation of the function evaluate
4.8	Cached implementation of the functions alphaBetaMax and alphaBetaMin 10
4.9	Progressive Deepening
4.10	Depth-limited α - β -pruning
5.1	The Proof of Newman's Lemma
6.1	The head of the file cars.csv
6.2	Simple Linear Regression
6.3	Calling the procedure simple_linear_regression
6.4	General linear regression
6.5	Polynomial Regression
6.6	$y = \sqrt{x_1}, x_2 = x_1 + \text{noise}.$ 140
6.7	Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$
6.8	Second Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$
6.9	Fourth Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$
6.10	· · · · · · · · · · · · · · · · · · ·
	Sixth Order Ridge Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \text{noise}$
0.11	
7.1	The function $x \mapsto \sin(x) - \frac{1}{2} \cdot x^2$
7.2	The gradient ascent algorithm
7.3	The sigmoid function
7.4	Results of an exam

LIST OF FIGURES

LIST OF FIGURES

7.5	An implementation of logistic regression
7.6	The function logisticRegression
7.7	Probability of passing an exam versus hours of studying
7.8	Results of an exam given hours of study and IQ
7.9	Probability of passing an exam versus hours of studying
7.10	Logistic Regression using SciKit-Learn
7.11	Probability of passing an exam versus hours of studying
7.12	Some fake data
7.13	Fake data with linear decision boundary
7.14	A script for second order logistic regression
	Elliptical decision boundary for fake data
7.16	Polynomial Logistic Regression
7.17	Fake data with a decision boundary of fourth order
7.18	Fake data with a decision boundary of order 14 182
7.19	Fake data with a decision boundary of order 14, regularized
	A Naive Bayes Classifier for Spam Detection: Part I
7.21	A Naive Bayes Classifier for Spam Detection: Part II
	A naive Bayes classifier for predicting the gender of a name
	Three points to separate
	Three points separated by logistic regression
	Points separated by logistic regression
7.26	Points separated by a support vector machine
8.1	A neural network with topology $[3,6,4,2]$
8.2	A neural network with topology $[8, 12, 8, 6, 3]$
8.3	The first 18 images of the MNIST dataset
8.4	Code to load the image files
8.5	The constructor of the class network
8.6	The class Network, part I
8.7	The class Network, part II
8.8	A Computational Graph for $\sin(x_1 + x_2) \cdot \cos(x_1 - x_2) + (x_1 + x_2) \cdot (x_1 - x_2)$
8.9	Rendering of the computation graph shown in Figure 8.8
8.10	A function that evaluates a computational graph
8.11	Auxiliary functions
8.12	Computing the partial derivative of a node
8 13	Computing the adjoints of a computational graph.

Index

E-variety, 115 R^2 , 135 Σ -equation, 113 Σ -term, 113 \mathcal{T}_{Σ} , 113 $s \doteq t$, 124 next_states, 8 15 puzzle, 12	chain rule, multivariate calculus, 210 classification problem, 154 composition $\sigma\tau$, 124 Composition von Substitutions, 124 computational graph, 220 computational node, 220 conflict, 81 connected variables, 74
8 puzzle, 12	consistency maintenance, 73
8 queens puzzle, 53 A* search, 32 A*-IDA* search, 44 accuracy, 155 activation function, 202	consistent heuristic, 28 constant feature, 165 constraint propagation, 63 constraint satisfaction problem, 51 corrected sample standard deviation, 134 critical pair, 128
activation vector, 204 adjoint, 224	declarative programming, 8
admissible heuristic, 28	deep blue, 90
admissible, computational graph, 221	deep neural network, 203
algorithm of Martelli and Montanari, 125	dependent variable, 133
AlphaGo, 90	depth first search, 20
arg max function, 158	design matrix, 141
artificial intelligence, definition, 4	double-ended queue, 19 doubly linked list, 19
backpropagation, 206, 208	· · · · · · · · · · · · · · · · · · ·
backpropagation equations, 207, 209	edge, 9
backtracking, 59	eight queens puzzle, 53
bias, 202	equational theorem proving, 111
bias term, 155	error in layer l , 207
bias vector, 203	feature, 154, 156
bidirectional search, 25	feature matrix, 141, 157
binary classification, 164	feed forward equations, 205
binary feature, 154	feed forward neural network, 202
bitboard, 94	frequency, 155
blind search, 28	fully connected, 203
branching factor, 25	function symbol, 112
breadth first search, 16	
centrally symmetric, 163 chain rule, 209	game, 90 general linear regression, 139 goal state, 9

INDEX INDEX

gradient 141	output layer, 202
gradient, 141	
gradient ascent, 157	overfitting, 147
gradient descent, 206	partial variable assignment, 51
group, 111	path, 9
group axioms, 111	Pearson correlation coefficient, 134
group element, 111	Pearson's r, 134
Hadamard product, 207	pickle, 213
heuristic, 28	Pluribus, 90
	polynomial regression, 145
hidden layer, 202	<u>, , , , , , , , , , , , , , , , , , , </u>
hyper parameter, 148	proportion of explained variance, 135
IDA* search, 41	proportion of unexplained variance, 135
independent variable, 133	quadratic error cost function, 205
input dimension, 203	quadratic circi cost ranction, 200
input layer, 202	rectified linear unit, 202
input node, 202	regularization parameter, 148
iterative deepening, 23	ReLU function, 202
recraeive deepening, 20	residual sum of squares, 135
Ke Jie, 90	ridge regression, 148
,	RSS, 135
least constraining value heuristic, 64	,
left-inverse, 111	sample correlation coefficient, 134
left-neutral element, 111	sample covariance, 134
likelihood, 165	sample mean value, 134
linear model, 133, 140	sample standard deviation, 134
linear regression, 133	sample variance, 135
local search, 78	search heuristic, 28
log-likelihood, 165	search problem, 8
logistic function, 162	search problem, solution, 9
logistic regression, 161	sigmoid function, 162
logit function, 164	signature, 112
	simple linear regression, 133
machine learning, definition, 4	sliding puzzle, 8, 12
map coloring, 51	solution of a Csp, 51
maximum likelihood estimation, 165	solution, minimal, 9
mean squared error, 133, 140	solution, of a search problem, 9
memoization, 98, 99	start state, 9
minimax algorithm, 96	state, 8
missionaries and cannibals, 9	Stirling's approximation of the factorial, 25
MNIST database of handwritten digits, 213	stochastic gradient descent, 212
most constrained variable heuristic, 63	supervised learning, 156
MSE, 140	symbolic AI, 4
1 0	syntactical equation, 124
node, 9	system of syntactical equations, 124
normal equation, 142, 143	.
number of layers, 202	term, 113
optimal solution, 18	terminal state, 91
optimistic heuristic, 28	tic-tac-toe, 92
	topology, 202
output dimension, 203	

INDEX

```
total sum of squares, 135
training example, 140
training set, 154
transfinite induction, 121
transposition operator, 133
TSS, 135
two person games, 91
unifier, 124
uninformed search, 28
utility of a state, 96
value of a state, 96
variable, 112
variable assignment, 51, 114
weight matrix, 203
weight of NN, 203
zebra puzzle, 62
zero sum game, 91
```