An Introduction to Artificial Intelligence

— Lecture Notes in Progress —

Prof. Dr. Karl Stroetmann

March 11, 2020

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:

`karl.stroetmann@dhbw-mannheim.de`

Contents

Abbreviations

Ai artificial intelligence

Bfs breadth first search

Csp constraint satisfaction problem

Dfs depth first search

Ida* iterative deepening A* search

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 have to answer the following question:

> What is 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 common theme of the first part is declarative programming. This approach is part of symbolic AI. The main idea of declarative programming is that we start with a problem specification. This is usually a short description of the problem that is to be solved. This description 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. Unfortunately, in this generality the idea of declarative programming has turned out to be unsuitable for real world problems 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 a complete logic based specification of a problem is feasible, automatic theorem proving is generally not powerful enough to find a solution automatically.

   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) Games like chess or checkers can be specified quite easily and there are various techniques for computers to find optimal strategies for playing adversarial games.

   (c) Constraint satisfaction problems have great practical importance. Today, very efficient constraint solvers have been developed to solve various constraint satisfaction problems that occur in practice.

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. 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 $i \in \{1, \cdots, N\}$ and for all $i \in \{1, \cdots, N\}$ 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 $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$. 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 $i \in \{1, \cdots, N\}$. 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 [RN09].

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.

The programs presented in these lecture notes are expected to run with version 3.7 of *Python*.

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 solving it from scratch. 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

$$\mathcal{P} = \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle$$

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

   $$\texttt{next\_states} : Q \to 2^Q.$$

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

$$R := \big\{ \langle s_1, s_2 \rangle \in Q \times Q \mid s_2 \in \texttt{next\_states}(s_1) \big\}.$$

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 $\texttt{start} \in Q$.

4. `goal` is the goal state, hence $\texttt{goal} \in Q$.

   Sometimes, instead of a single `goal` there is a set of goal 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. A path $[s_1, \cdots, s_n]$ is a solution to the search problem P iff the following conditions are satisfied:

1. $s_1 = \texttt{start}$, i.e. the first element of the path is the start state.

2. $s_n = \texttt{goal}$, 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

   $$s_n \in \texttt{Goals}.$$

A path $p = [s_1, \cdots, s_n]$ 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 also 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 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 $\texttt{problem}(m, i)$ 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 3 defines the auxiliary function `noProblem`.

   If m is the number of missionaries on the western shore and i is the number of infidels on that shore, then the expression $\texttt{noProblem}(m, i)$ 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 $3 - i$.

```
1  problem = lambda m, i: 0 < m < i
2
3  noProblem = lambda m, i: not problem(m, i) and not problem(3 - m, 3 - i)
4
5  def next\_states(state):
6      m, i, b = state
7      if b == 1:
8          return { (m - mb, i - ib, 0) for mb in range(m+1)
9                                       for ib in range(i+1)
10                                      if 1 <= mb + ib <= 2 and
11                                         noProblem(m - mb, i - ib)
12                 }
13     else:
14         return { (m + mb, i + ib, 1) for mb in range(3-m+1)
15                                      for ib in range(3-i+1)
16                                      if 1 <= mb + ib <= 2 and
17                                         noProblem(m + mb, i + ib)
18                 }
19
20 start = (3, 3, 1)
21 goal  = (0, 0, 0)
```

Figure 2.1: The missionary and cannibals problem coded as a search problem.

3. Lines 5 to 18 define the function `next_states`. A state s is represented as a triple of the form

   $$s = (m, i, b) \quad \text{where } m \in \{0,1,2,3\},\ i \in \{0,1,2,3\}, \text{ and } b \in \{0,1\}.$$

   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 6 extracts the components m, i, and b from the state s.

   (b) Line 7 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 8.

   (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

   $$\texttt{mb} \in \{0, \cdots, \texttt{m}\},$$

   which is implemented by the line

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

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

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

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 <= mb + ib <= 2.
```

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 $3 - \texttt{m}$ and the number of infidels on the eastern shore is $3 - \texttt{i}$, `mb` has to be a member of the set $\{0, \cdots, 3 - \texttt{m}\}$, while `ib` has to be a member of the set $\{0, \cdots, 3 - \texttt{i}\}$.

5. Finally the start state and the goal state are defined in line 20 and line 21.

The code in Figure 2.1 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:

```
States = { (m, i, b) for m in {0, 1, 2, 3}
                     for i in {0, 1, 2, 3}
                     for b in {0, 1}
                     if noProblem(m, i)
         }
```

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 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 2.2 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. ◇

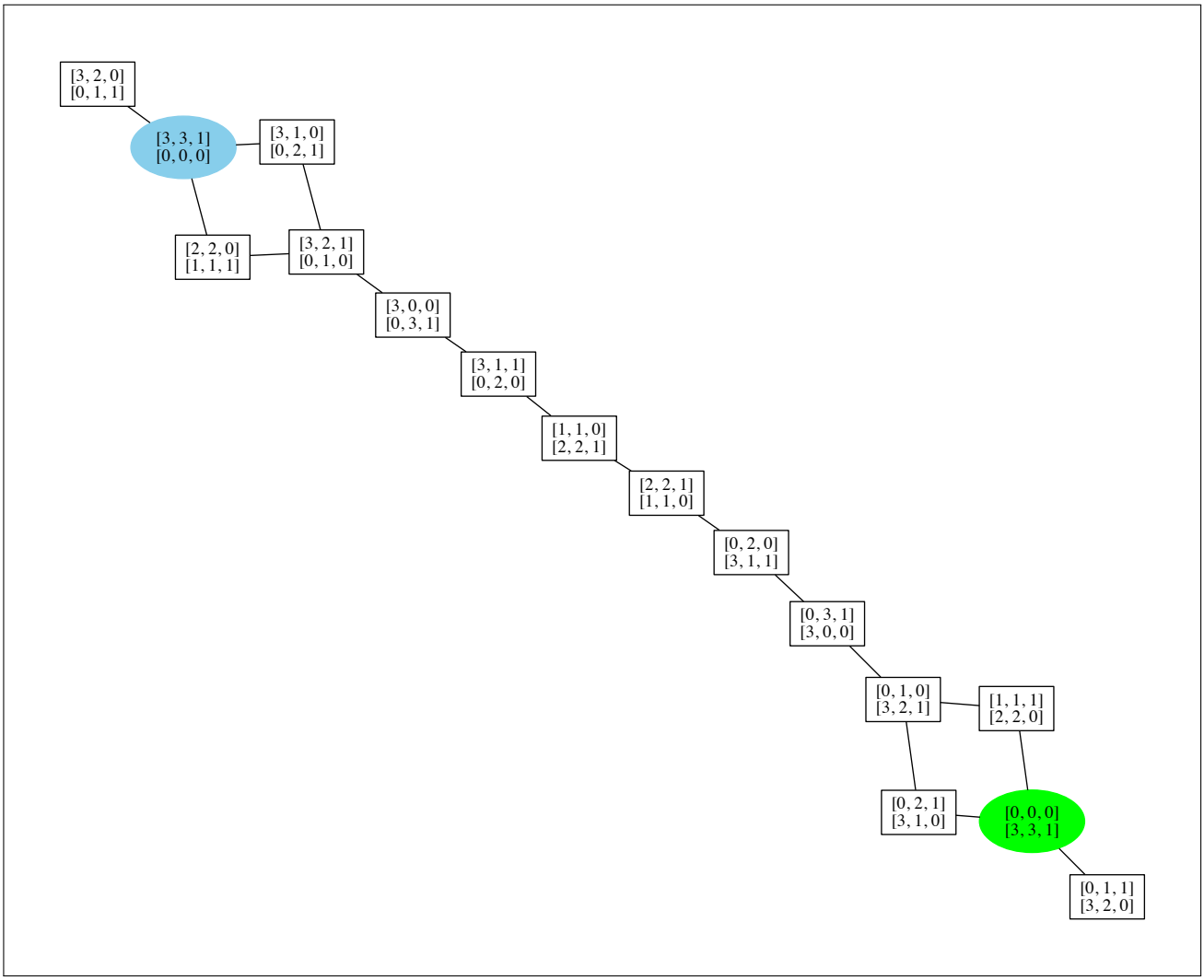

Figure 2.2: A graphical representation of the missionaries and cannibals problem.

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 bigger 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 2.3 on page 12 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 2.3 we have to transform the state shown on the left of Figure 2.3 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.

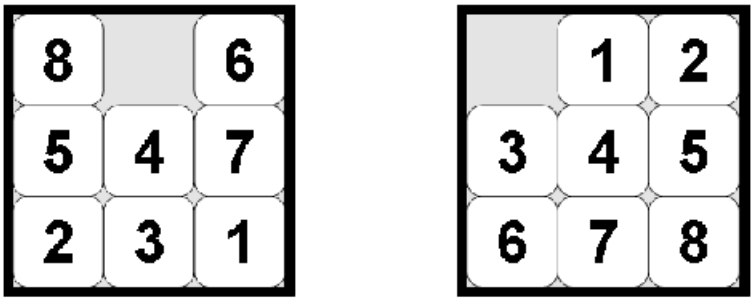

Figure 2.3: The 3×3 sliding puzzle.

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.

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

http://mypuzzle.org/sliding.

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 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 2.4 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 2.3 is represented as the tuple:

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

```
to_list = lambda State: [list(row) for row in State]

to_tuple = lambda State: tuple(tuple(row) for row in State)

def find_tile(tile, State):
    n = len(State)
    for row in range(n):
        for col in range(n):
            if State[row][col] == tile:
                return row, col

def move_dir(State, row, col, dx, dy):
    State = to_list(State)
    State[row     ][col     ] = State[row + dx][col + dy]
    State[row + dx][col + dy] = 0
    return to_tuple(State)

def next_states(State):
    n          = len(State)
    row, col   = find_tile(0, State)
    New_States = set()
    Directions = [ (1, 0), (-1, 0), (0, 1), (0, -1) ]
    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))
    return New_States

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

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

Figure 2.4: The 3×3 sliding puzzle.

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 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, later we have to manipulate the states. To this end, we have to first transform them 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 2.4 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. `findTile` 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. `moveDir` takes a `State`, the `row` and the `col`umn 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 $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$ is moved into the position $\langle \texttt{row}, \texttt{col} \rangle$, while the tile at position $\langle \texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy} \rangle$ 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 $(\texttt{row}, \texttt{col})$ and the direction of the movement is given as $(\texttt{dx}, \texttt{dy})$, then in order to ensure that the empty tile can be moved to the position $[\texttt{row} + \texttt{dx}, \texttt{col} + \texttt{dy}]$, we have to ensure that both

   $$\texttt{row} + \texttt{dx} \in \{0, \cdots, n-1\} \quad \text{and} \quad \texttt{col} + \texttt{dy} \in \{0, \cdots, n-1\}$$

   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 problems $\langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle$, 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 saved into a set `Visited`.

In order to avoid loops, 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`, $\texttt{Parent}[s]$ is the state that caused s to be added to the set `Frontier`, i.e. for all states $s \in \texttt{Visited} \cup \texttt{Frontier}$ we have

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

Figure 2.5 on page 15 shows an implementation of breadth first search in *Python*. We discuss this implementation line by line:

```
def search(start, goal, next_states):
    Frontier = { start }
    Visited  = set()
    Parent   = { start: start }
    while len(Frontier) > 0:
        NewFrontier = set()
        for s in Frontier:
            for ns in next_states(s):
                if ns not in Visited and ns not in Frontier:
                    NewFrontier.add(ns)
                    Parent[ns] = s
                    if ns == goal:
                        return path_to(goal, Parent)
        Visited |= Frontier
        Frontier = NewFrontier
```

Figure 2.5: Breadth first 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 whose neighbours have already been added to the set `Frontier`. 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 `pathTo()` is shown in Figure 2.6 on page 15.

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 2.6: The function `pathTo()`.

The function call `pathTo(state,Parent)` constructs a path reaching from `start` to `state` in reverse by looking up the parent states.

If we try breadth first search to solve the missionaries and cannibals problem, we obtain the solution shown in Figure 2.7. 15 nodes had to be expanded to find this solution. To keep this in perspective, we note that Figure 2.2 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.

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

Figure 2.7: A solution of the missionaries and cannibals problem.

Next, let us try to solve the 3×3 sliding puzzle. It takes about 2 seconds to solve this problem on my computer[1], while 181,439 states are touched. Again, we see that breadth first search touches nearly all the states reachable from the start state.

Breadth first search has two important properties:

1. Breadth first search is complete: If there is a solution to the given search problem, then breadth first search is going to find it.

2. The solution found by breadth first search is optimal, i.e. it is one of the shortest possible solutions.

[1] I 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.

Proof: Both of these claims can be shown simultaneously. Consider the implementation of breadth first search shown in Figure 2.5. An easy induction on the number of iterations of the `while` loop shows 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`. This claim is obviously true before the first iteration of the while loop as in this case, `Frontier` only contains the state `start`. 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. There is a shorter path to a state if this state is already a member of the set `Visited`. Otherwise, the shortest path to a state that is reached in iteration $n+1$ has the length $n+1$. 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 iteration number k 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 3.11 of Russell & Norvig [RN09], breadth first search is often implemented using a queue data structure. Figure 2.8 on page 18 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 8 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 16 we add the states that have not been encountered previously at the right end of the queue `Frontier` using the method `append`.

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.

```
from collections import deque

def search(start, goal, next_states):
    Frontier = deque([start])
    Visited  = set()
    Parent   = { start: start }
    while len(Frontier) > 0:
        state = Frontier.popleft()
        if state == goal:
            return path_to(state, Parent)
        Visited.add(state)
        newStates = next_states(state)
        for ns in newStates:
            if ns not in Visited and ns not in Parent:
                Parent[ns] = state
                Frontier.append(ns)
```

Figure 2.8: A queue based implementation of breadth first search.

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.

The resulting algorithm is shown in Figure 2.9 on page 18. 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.

```
def search(start, goal, next_states):
    Stack  = deque([start])
    Parent = { start: start }
    while len(Stack) > 0:
        state = Stack.pop()
        for ns in next_states(state):
            if ns == goal:
                return path_to(state, Parent) + [goal]
            if ns not in Parent:
                Parent[ns] = state
                Stack.append(ns)
```

Figure 2.9: The depth first search algorithm.

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 a dictionary `Parent`. For every state s that is on `Stack`, `Parent`$[s]$ returns a state p such that

$s \in$ `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 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.

When we test the implementation shown above with the 3×3 sliding puzzle, it takes about 0.25 seconds on my computer to find a solution. This is quite an improvement compared to breadth first search. Furthermore, there is no longer a need to keep the set `Visited` around, hence the memory requirements of depth first search are much smaller than the memory requirements of breadth first search. However, we still have to maintain the set `Parent`. Fortunately, we will be able to get rid of the set `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 more than $8,000$ 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 is 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 2.10 on page 19.

```
1  def search(start, goal, next_states):
2      return dfs(start, goal, next_states, [start])
3
4  def dfs(state, goal, next_states, Path):
5      if state == goal:
6          return Path
7      for ns in next_states(state):
8          if ns not in Path:
9              Result = dfs(ns, goal, next_states, Path + [ns])
10             if Result != None:
11                 return Result
```

Figure 2.10: A recursive implementation of depth first search.

The only purpose of the function `search` is to call the function `dfs`, which needs one additional argument. This argument is called `Path`. 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`. 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 list

   ```
   Path + [ns]
   ```

   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 list `Path`. 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. 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. The reason is that there is no `return` statement at the end of the function `dfs`. Hence, if the last line of the function `dfs` is reached, `None` is returned by default.

For the 3×3 puzzle, the recursive implementation takes about 26 second to compute the solution. In this case, the solution has a length of more than 20 000 steps. This, together with the fact that function calls are quite expensive in *Python*, is the reason that the recursive version is less efficient than the stack based implementation of depth first search. The good news is that this program does not need much memory. The only variable that uses considerable memory is the variable `Path`. If we can somehow keep the list `Path` short, then the recursive version of depth first search uses only a small fraction of the memory needed by breadth first search.

2.4 Iterative Deepening

The fact that the recursive version of depth first search took just one second to find a solution is very impressive. The questions 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 2.11 on page 21 implements this strategy. We proceed to discuss the details of this program.

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.

```
def search(start, goal, next_states):
    limit = 1
    while True:
        Path = depth_limited_search(start, goal, next_states, limit)
        if Path != None:
            return Path
        limit += 1

def depth_limited_search(state, goal, next_states, limit):
    Stack = deque([[start]])
    while len(Stack) > 0:
        Path  = Stack.pop()
        state = Path[-1]
        if state == goal:
            return Path
        if len(Path) >= limit:
            continue
        for ns in next_states(state):
            if ns not in Path:
                Stack.append(Path + [ns])
```

Figure 2.11: Iterative deepening implemented in *Python*.

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 `depthLimitedSearch` implements depth first search but takes care to compute only those paths that have a length of at most `limit`. The implementation shown in Figure 2.11 is stack based. In this implementation, the stack contains paths leading from `start` to the state at the end of a given path. Hence it is similar to the implementation of depth first search shown in Figure 2.9 on page 18.

3. The stack is initialized to contain the path [`start`].

4. In the `while`-loop, the first thing that happens is that the `Path` on top of the stack is removed from the stack. The state at the end of this `Path` is called `state`. If this `state` happens to be the `goal`, a solution to the search problem has been found and this solution is returned.

5. Otherwise, we check the length of `Path`. If this length is greater than or equal to the `limit`, the `Path` can be discarded as we have already checked that it does not end in the `goal`.

6. Next, the neighbours of `state` are computed. For every neighbour `ns` of `state` that has not yet been encountered in `Path`, we extend `Path` to a new list that ends in `ns` and push this new list onto the `Stack`.

7. This process is iterated until the `Stack` is exhausted.

The nice thing about the program presented in this section is the fact that it does not use much memory. The reason is that the stack can never have a size that is longer than `limit` and therefore the overall memory that is needed can be bounded by $\mathcal{O}(\texttt{limit}^2)$. However, when we run this program to solve the 3×3 sliding puzzle, the algorithm takes about 17 minutes. There are two reasons for this:

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 31 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 doubled[2] 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

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

   Therefore, if the solution is found at a depth of $d + 1$, we will explore roughly 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.

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

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

Furthermore, the states `start` and `goal` are defined as

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

Next, the function `next_states` is defined as

$$\texttt{next\_states}\big(\langle a, b\rangle\big) := \big\{\langle a + 1, b\rangle, \langle a, b + 1\rangle\big\}.$$

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

$$\mathcal{P} := \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal}\rangle.$$

Given a natural number n, compute the number of different solutions of this search problem and prove your claim.

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:

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

Exercise 2: If there is no solution, the implementation of iterative deepening that is shown in Figure

[2] When 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.

2.11 does not terminate. The reason is that the function `depthLimitedSearch` 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 tried all paths without hitting the depth limit but the `Stack` is exhausted.

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

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

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

$$2 \cdot \frac{b^{d/2+1} - 1}{b - 1}$$

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 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

$$2 \cdot \big(2^{40/2+1} - 1\big) = 4,194,302$$

states with bidirectional breadth first search. While it is certainly feasible to keep four million states in memory, keeping a two trillion states in memory is impossible on most devices.

Figure 2.12 on page 24 shows the implementation of bidirectional breadth first search. Essentially, we have to copy the breadth first program shown in Figure 2.5. 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`.

```
def search(start, goal, next_states):
    FrontierA = { start }
    VisitedA  = set()           # set of nodes expanded starting from start
    ParentA   = { start: start}
    FrontierB = { goal }
    VisitedB  = set()           # set of nodes expanded starting from goal
    ParentB   = { goal: goal}
    while len(FrontierA) > 0 and len(FrontierB) > 0:
        VisitedA  |= FrontierA
        VisitedB  |= FrontierB
        NewFrontier = set()
        for s in FrontierA:
            for ns in next_states(s):
                if ns not in VisitedA:
                    NewFrontier |= { ns }
                    ParentA[ns]  = s
                    if ns in VisitedB:
                        return combinePaths(ns, ParentA, ParentB)
        FrontierA   = NewFrontier
        NewFrontier = set()
        for s in FrontierB:
            for ns in next_states(s):
                if ns not in VisitedB:
                    NewFrontier |= { ns }
                    ParentB[ns]  = s
                    if ns in VisitedA:
                        return combinePaths(ns, ParentA, ParentB)
        FrontierB = NewFrontier
```

Figure 2.12: Bidirectional breadth first search.

2. `VisitedA` is the set of states that have been visited starting from `start`, while `VisitedB` is the set of states that have been visited starting from `goal`.

3. 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`.

4. 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.

5. Initially, the `while` loop adds the frontier sets to the visited sets as all the neighbours of the frontier sets will now be explored.

6. Then the `while` loop 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` to is shown in Figure 2.13 on page 25.

7. Next, the same computation is done with the role of the states `start` and `goal` exchanged.

On my computer, bidirectional breadth first search solves the 3×3 sliding puzzle in less than 120 milliseconds! However, bidirectional breadth first search is still not able to solve 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.

```
1  def combinePaths(state, ParentA, ParentB):
2          Path1 = path_to(state, ParentA)
3          Path2 = path_to(state, ParentB)
4          return Path1[:-1] + Path2[::-1] # Path2 is reversed
```

Figure 2.13: Combining two paths.

2.6 Best First Search

Up to now, all the search algorithms we have discussed were 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 [RN09] use the name uninformed search instead of blind search.

If a human tries to solve a problem, she usually will develop a feeling 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

$$\mathcal{P} = \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle,$$

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

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

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

$$h(s) \leq d(s) \quad \text{for all } s \in Q.$$

Hence, the heuristic is admissible iff it is optimistic: It must never overestimate the distance to the goal, but it is free to underestimate this distance.

Finally, the heuristic h is called consistent iff we have

$$h(\texttt{goal}) = 0 \quad \text{and} \quad h(s_1) \leq 1 + h(s_2) \quad \text{for all } s_2 \in \texttt{next\_states}(s_1). \qquad \diamond$$

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 $h(\texttt{goal}) = 0$. 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

$$h(s_1) \leq 1 + h(s_2).$$

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

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

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

$$h(s_k) \leq k \quad \text{for all } k \in \{0, 1, \cdots, n\}.$$

This claim is shown by induction on k.

B.C.: $k = 0$.

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

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

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

$$\begin{array}{rcll} h(s_{k+1}) & \leq & 1 + h(s_k) & \text{because } s_k \in \texttt{next\_states}(s_{k+1}) \text{ and } h \text{ is consistent,} \\ & \leq & 1 + k & \text{because } h(s_k) \leq k \text{ by induction hypotheses.} \end{array}$$

□

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

   $$0 \leq 1 + 0,$$

   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 2.3 on page 12 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 state can differ by at most one and hence the inequality

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

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

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

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 is $\langle 3, 3\rangle$, 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 $\texttt{pos}(t, s)$ for all tiles $t \in \{1, \cdots, 8\}$ in a given state s as follows:

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

i.e. given a state s, the expression $\texttt{pos}(t, s)$ 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:

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

The Manhattan heuristic measure the number of moves that would be needed if we wanted to put every tile of s into its final positions and if we were allowed to slide tiles over each other. Figure 2.14 on page 28 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.

   (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 $\langle$`rowA`, `colA`$\rangle$ in `stateA` is computed using the expression `stateA[rowA][colA]` and the corresponding position $\langle$`rowB`, `colB`$\rangle$ of this tile in state `stateB` is computed using the function `findTile`.

   (c) Finally, the Manhattan distance between the two positions $\langle$`rowA`, `colA`$\rangle$ and $\langle$`rowB`, `colB`$\rangle$ is added to the `result`.

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

Figure 2.14: The Manhattan distance between two states.

The Manhattan distance 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 $h(s_1) \leq 1 + h(s_2)$. □

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) removes the pair from H that has the highest priority. This pair is also returned.

2. Given a heap H, the function `heapq.heappush`$\big(H, (p, o)\big)$ pushes the pair (p, o) onto the heap H. This method does not return a value. Instead, the heap H is changed in place.

The function `search` shown in Figure 2.15 on page 29 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 $(d, [\texttt{start}])$, where d is the estimated distance of a path leading from `start` to `goal`. In `PrioQueue` we store the paths in pairs of the form

   $$\langle \texttt{estimate}, \texttt{Path} \rangle,$$

   where `Path` is a list of states starting from the node `start`. If the last node on this list is called

```
def search(start, goal, next_states, heuristic):
    PrioQueue = [ (heuristic(start, goal), [start]) ]
    while len(PrioQueue) > 0:
        _, Path = heapq.heappop(PrioQueue)
        state   = Path[-1]
        if state == goal:
            return Path
        for ns in next_states(state):
            if ns not in Path:
                d = heuristic(ns, goal)
                heapq.heappush(PrioQueue, (d, Path + [ns]))
```

Figure 2.15: The best first search algorithm.

`state`, then we have

$$\texttt{estimate} = \texttt{heuristic}(\texttt{state}, \texttt{goal}),$$

i.e. `estimate` is the estimated distance between these `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 2.3 on page 12 in less than 7 milliseconds. 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. Furthermore, best first search is still not strong enough to solve the 4×4 puzzle shown in Figure 2.19 on page 39.

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 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

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

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` while passing through the state s would use. The details of the A* algorithm are given in Figure 2.16 on page 30 and discussed below.

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

Figure 2.16: The A* search algorithm.

The function `search` takes 4 parameters:

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

   $$\texttt{heuristic}(s_1, s_2)$$

   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

   $$\texttt{Parent}[s_2] = s_1 \;\Rightarrow\; s_2 \in \texttt{next\_states}(s_1).$$

   The only exception is the state `start`: We have $\texttt{Parent}[\texttt{start}] = \texttt{start}$ 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.

3. `Estimate` is a dictionary. For every state s encountered in the search, $\texttt{Estimate}[s]$ 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

   $$(d, s) \quad \text{such that} \quad d = \texttt{Estimate}[s],$$

   i.e. if $(d, s) \in \texttt{Frontier}$, 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 my own 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 a 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

   $$\texttt{stateDist} + 1 + \texttt{heuristic}(\texttt{neighbour}, \texttt{goal}).$$

   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 2.3 on page 12. 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.

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. Furthermore, we have added a set `Explored` to the implementation. This set is only needed for the proof of the optimality of the path found by A* search. Figure 2.17 on page 34 shows this version of the algorithm.

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

$$\mathcal{P} = \langle Q, \texttt{next\_states}, \texttt{start}, \texttt{goal} \rangle$$

is a search problem and

$$\texttt{heuristic} : Q \to \mathbb{N}$$

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 discuss a number of notions that are needed in order to improve our understanding of the A* algorithm.

(a) 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.

(b) A state is said to have been explored iff all of its neighbouring states have been visited.

To emphasize the notion of explored states I have added the set `Explored` in the implementation of A* search that is shown in Figure 2.17 on page 34. Note that the variable `Explored` serves no purpose in the implementation of A* search: This variable is only written to but the algorithm never reads this variable.

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

Claim One: If a state s is visited and $P := \texttt{PathDict}[s]$, 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 computational induction.

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

Figure 2.17: A path based implementation of A* 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 we add in line 20 the path

$$\texttt{PathDict}[\texttt{state}] + [\texttt{ns}]$$

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

$$P := \texttt{PathDict}[\texttt{state}]$$

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 bigger by one than the length of P and it is obvious that this new path

$$P + [\texttt{ns}]$$

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 19.

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

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

Here, the expression $\texttt{dist}(\texttt{start}, s)$ returns the length of the shortest path from `start` to s. From the definition of the dictionary `Distance` and Claim One it is obvious that

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

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 $\texttt{Distance}[s]$. 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:

$$f(s) := \texttt{Distance}[s] + \texttt{heuristic}(s, \texttt{goal}).$$

Here, $\texttt{Distance}[s]$ is the number of steps that it takes to reach the state s from `start` when following the path stored in $\texttt{PathDict}[s]$, while $\texttt{heuristic}(s, \texttt{goal})$ 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 $s \in Q$ has been explored, then we have

$$\texttt{Distance}[s] = g(s),$$

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

$$g(s) < \texttt{Distance}[s],$$

i.e. we assume that $P_1 := \texttt{PathDict}[s]$ 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

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

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. 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

$$\texttt{Distance}[s] = n.$$

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

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

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

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

Continuing this way we conclude that

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

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

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

This implies

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

Next, we compute $f(s)$. We have

$$f(s) = \texttt{Distance}[s] + \texttt{heuristic}(s) = n + \texttt{heuristic}(s).$$

However, since we have $m < n$ this implies

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

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` immediately 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 means that the state y_{k+2} is explored before the state s is explored. Similarly to the proof that $f(y_{k+2}) \leq m + \texttt{heuristic}(s)$, we can see that

$$f(y_{k+2}) \leq k + 2 + (m - k - 2) + \texttt{heuristic}(s) = m + \texttt{heuristic}(s).$$

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

$$f(y_j) \leq m + \texttt{heuristic}(s) \quad \text{for all } j \in \{k+1, \cdots, m-1\}.$$

Since all states of the form

$$y_j \quad \text{for } j \in \{k+1, \cdots, m-1\}$$

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

$$\texttt{Distance}[s] = m \quad \text{instead of} \quad \texttt{Distance}[s] = n.$$

This contradiction proves that our assumption

$$\texttt{Distance}[s] > g(s)$$

must be wrong and therefore we know that the equation $\texttt{Distance}[s] = g(s)$ 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 $\texttt{Distance}[\texttt{goal}] = g(\texttt{goal})$. As $g(\texttt{goal})$ is the true distance of the state `goal` form 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 2.18 on page 38 shows the resulting program. This program relates to the A* algorithm shown in Figure 2.16 on page 30 as the algorithm for bidirectional search shown in Figure 2.12 on page 24 relates to breadth first search shown in Figure 2.5 on page 15. 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 2.3 on page 12, 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 2.19 on page 39 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 connection `start` and `goal` has a length of 36 steps, this shows that bidirectional A* search does not find the shortest path.

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 2.20 on page 39 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.

```
def search(start, goal, next_states, heuristic):
    estimate  = heuristic(start, goal)
    ParentA   = { start: start };      ParentB   = { goal : goal  }
    DistanceA = { start: 0 };          DistanceB = { goal : 0 }
    EstimateA = { start: estimate };   EstimateB = { goal : estimate }
    FrontierA = Set.Set();             FrontierB = Set.Set()
    FrontierA.insert( (estimate, start) )
    FrontierB.insert( (estimate, goal ) )
    while not FrontierA.isEmpty() and not FrontierB.isEmpty():
        guessA, stateA = FrontierA.pop()
        guessB, stateB = FrontierB.pop()
        stateADist = DistanceA[stateA]
        stateBDist = DistanceB[stateB]
        if guessA <= guessB:
            FrontierB.insert( (guessB, stateB) )
            for ns in next_states(stateA):
                oldEstimate = EstimateA.get(ns, None)
                newEstimate = stateADist + 1 + heuristic(ns, goal)
                if oldEstimate == None or newEstimate < oldEstimate:
                    ParentA  [ns] = stateA
                    DistanceA[ns] = stateADist + 1
                    EstimateA[ns] = newEstimate
                    FrontierA.insert( (newEstimate, ns) )
                    if oldEstimate != None:
                        FrontierA.delete( (oldEstimate, ns) )
                if DistanceB.get(ns, None) != None:
                    return combinePaths(ns, ParentA, ParentB)
        else:
            FrontierA.insert( (guessA, stateA) )
            for ns in next_states(stateB):
                oldEstimate = EstimateB.get(ns, None)
                newEstimate = stateBDist + 1 + heuristic(start, ns)
                if oldEstimate == None or newEstimate < oldEstimate:
                    ParentB  [ns] = stateB
                    DistanceB[ns] = stateBDist + 1
                    EstimateB[ns] = newEstimate
                    FrontierB.insert( (newEstimate, ns) )
                    if oldEstimate != None:
                        FrontierB.delete( (oldEstimate, ns) )
                if DistanceA.get(ns, None) != None:
                    return combinePaths(ns, ParentA, ParentB)
```

Figure 2.18: Bidirectional A* search.

(d) `heuristic` is a function that takes two parameters s_1 and s_2, where s_1 and s_2 are states. The expression

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

Figure 2.19: A start state and a goal state for the 4×4 sliding puzzle.

```python
def search(start, goal, next_states, heuristic):
    limit = heuristic(start, goal)
    while True:
        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):
    state    = Path[-1]
    distance = len(Path) - 1
    total    = distance + heuristic(state, goal)
    if total > limit:
        return total
    if state == goal:
        return Path
    smallest = float("Inf")  # infinity
    for ns in next_states(state):
        if ns not in Path:
            Solution = dl_search(Path + [ns], goal, next_states, limit, heuristic)
            if isinstance(Solution, list):
                return Solution
            smallest = min(smallest, Solution)
    return smallest
```

Figure 2.20: Iterative deepening A* search.

`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. `heuristic` has to be admissible.

Note that this is different from A* search. A* search requires that `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.

   (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 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 or equal than 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

   `distance` + `heuristic`(`state`, `goal`).

   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

   ```
   Path + [ns].
   ```

   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` variable 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 2.3 on page 12, this algorithm takes about 0.8 seconds to solve the puzzle. For the 4×4 sliding puzzle shown in Figure 2.19, the algorithm takes about 2.5 seconds. 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, from all of 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.

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

Figure 2.21: The A*-IDA* search algorithm, part `I`.

An implementation of the A*-IDA* algorithm is shown in Figure 2.21 on page 42 and Figure 2.22 on page 44. 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

$$\texttt{Parent}[s_2] = s_1 \;\Rightarrow\; s_2 \in \texttt{next\_states}(s_1).$$

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

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

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

$$(d, s) \quad \text{such that} \quad d = \texttt{Estimate}[s],$$

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).
   ```

Iterative deepening A*-IDA* is a complete search algorithm. On the instance of the 3×3 sliding puzzle shown on Figure 2.3 on page 12, 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 3 000 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 step.

```python
def id_search(goal, start, next_states, heuristic, Distance):
    limit = 0
    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):
    state = Path[-1]
    total = len(Path) - 1 + heuristic(state, start)
    if total > limit:
        return total
    if state in Distance:
        return Path
    smallest = float('Inf')
    for ns in next_states(state):
        if ns not in Path:
            result = dl_search( Path + [ns], start, next_states, limit,
                                heuristic, Distance
                              )
            if isinstance(result, list):
                return result
            smallest = min(smallest, result)
    return smallest
```

Figure 2.22: The A*-IDA* search algorithm, part II.

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 2.23 would be represented as the list

$$[6, 4, 7, 1, 8, 2, 5, 3]$$

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.

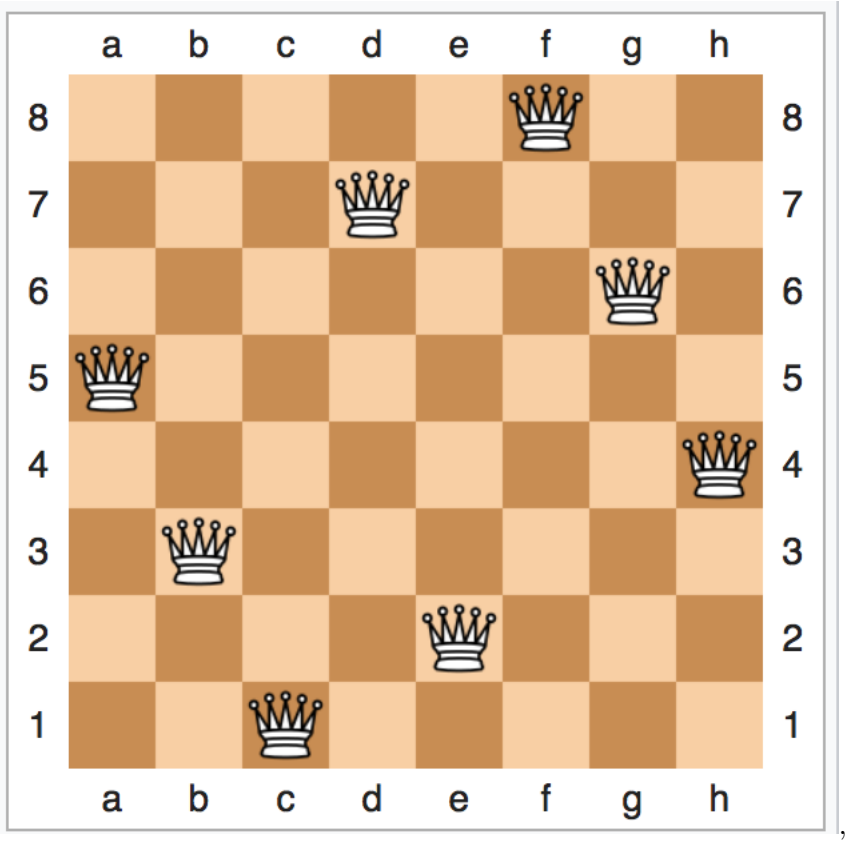

Figure 2.23: A solution of the eight queens puzzle.

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 the 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:

```
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * | * | * |
+---+---+---+---+
| * | * |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

```
+---+---+---+---+
| * | * | 2 | 3 |
+---+---+---+---+
| * | * | 6 | 7 |
+---+---+---+---+
| 8 | 9 |10 |11 |
+---+---+---+---+
|12 |13 |14 |15 |
+---+---+---+---+
```

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

Python/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 formulæ and search for values that can be assigned to the variables occurring in these formulæ so that all of the formulæ 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. 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. In many cases, 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) Finally, 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 too complicated.

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

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

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 formulæ from first order logic. Each of these formulæ is called a constraint of $\mathcal{P}$.

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

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

The second component $\mathcal{J}$ defines the interpretations of the function symbols and predicate symbols that are used in the formulæ defining the constraints. In the following we assume that these interpretations are understood from the context of the constraint satisfaction problem $\mathcal{P}$. 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

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

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

$$A : \texttt{Vars} \to \texttt{Values}.$$

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

$$\texttt{eval}(f, A) = \texttt{true} \quad \text{for all } f \in \texttt{Constraints}.$$

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

$$B : \texttt{Vars} \to \texttt{Values} \cup \{\Omega\} \quad \text{where } \Omega \text{ denotes the undefined value.}$$

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 $\texttt{dom}(B)$ of a partial variable assignment B is the set of those variables that are assigned a value different from Ω, i.e. we define

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

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

3.1.1 Example: Map Colouring

In map colouring a map showing different state 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 50 shows a map of Australia. There are seven different states in Australia:

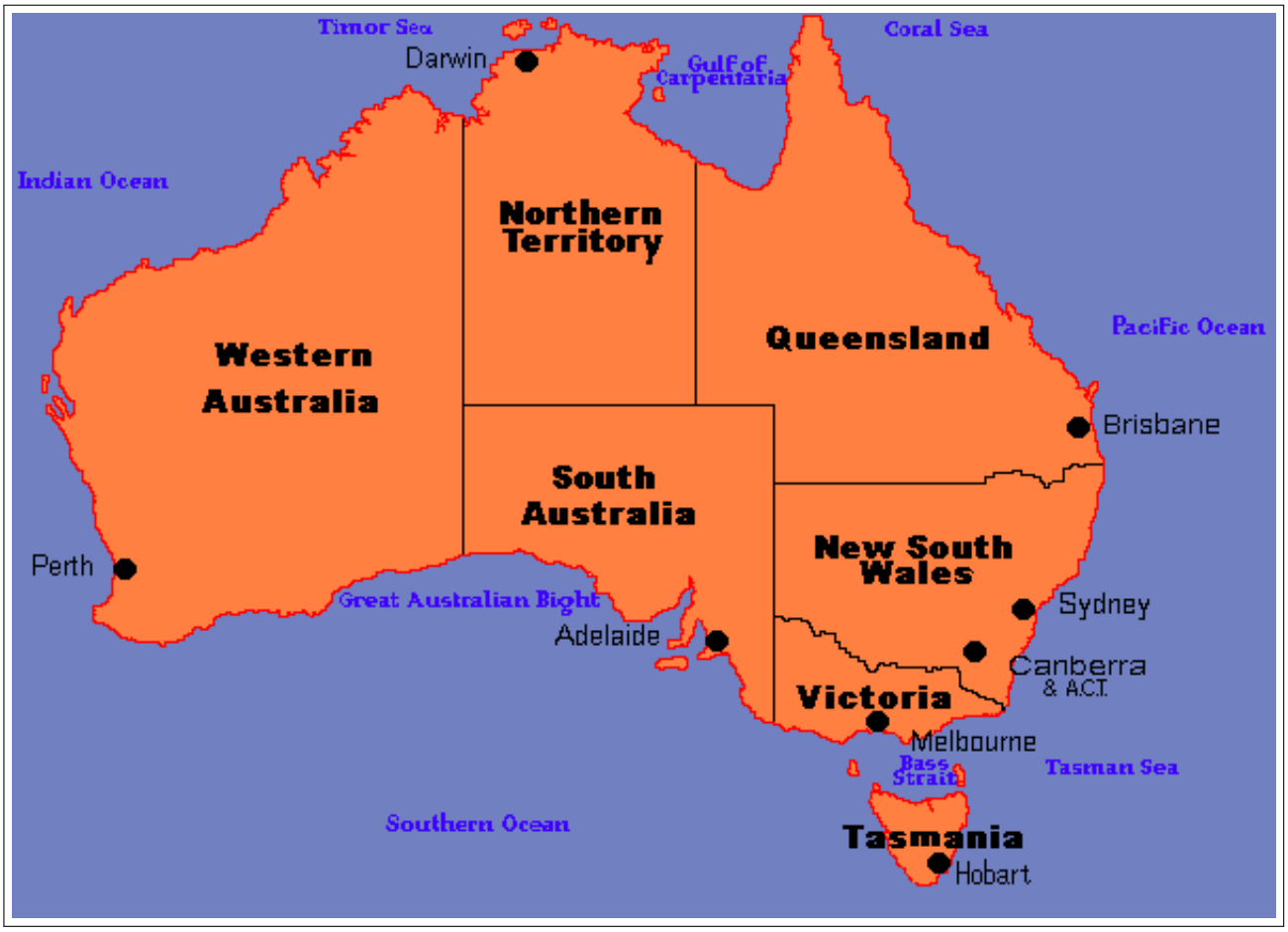

Figure 3.1: A map of Australia.

1. Western Australia, abbreviated as WA,
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 had been coloured with different colours. For the purpose of this example let us assume that we have only three colours 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` := $\{\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}, \text{T}\}$,

2. $\texttt{Values} := \{\texttt{red}, \texttt{green}, \texttt{blue}\}$,

3. $\texttt{Constraints} :=$

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

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

(a) $A(\text{WA}) = \texttt{blue}$,

(b) $A(\text{NT}) = \texttt{red}$,

(c) $A(\text{SA}) = \texttt{green}$,

(d) $A(\text{Q}) = \texttt{blue}$,

(e) $A(\text{NSW}) = \texttt{red}$,

(f) $A(\text{V}) = \texttt{blue}$,

(g) $A(\text{T}) = \texttt{red}$,

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 onto 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:

$$\texttt{Vars} := \{\texttt{V}_1, \texttt{V}_2, \texttt{V}_3, \texttt{V}_4, \texttt{V}_5, \texttt{V}_6, \texttt{V}_7, \texttt{V}_8\},$$

where for $i \in \{1, \cdots, 8\}$ the variable $\texttt{V}_i$ specifies the column of the queen that is placed in row i. As the columns run from one to eight, we define the set $\texttt{Values}$ as

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

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

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

$$\texttt{SameRow} := \big\{\texttt{V}_i \neq \texttt{V}_j \;\big|\; i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i\big\}.$$

   Here the condition $j < i$ ensures that, for example, while we have the constraint $\texttt{V}_2 \neq \texttt{V}_1$ we do not also have the constraint $\texttt{V}_1 \neq \texttt{V}_2$, as the latter constraint would be redundant if the former constraint had already been established.

2. We have 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

   $$\texttt{SameDiagonal} := \big\{ |i - j| \neq |\texttt{V}_i - \texttt{V}_j| \;\big|\; i \in \{1, \cdots, 8\} \wedge j \in \{1, \cdots, 8\} \wedge j < i \big\}.$$

   Essentially, these are just the linear equations for the straight lines with slope 1 and -1.

Then, the set of constraints is defined as

$$\texttt{Constraints} := \texttt{SameRow} \cup \texttt{SameDiagonal}$$

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

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

If we define the assignment A such that

$$A(V_1) := 4,\ A(V_2) := 8,\ A(V_3) := 1,\ A(V_4) := 2,\ A(V_5) := 6,\ A(V_6) := 2,$$
$$A(V_7) := 7,\ A(V_8) := 5,$$

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 52.

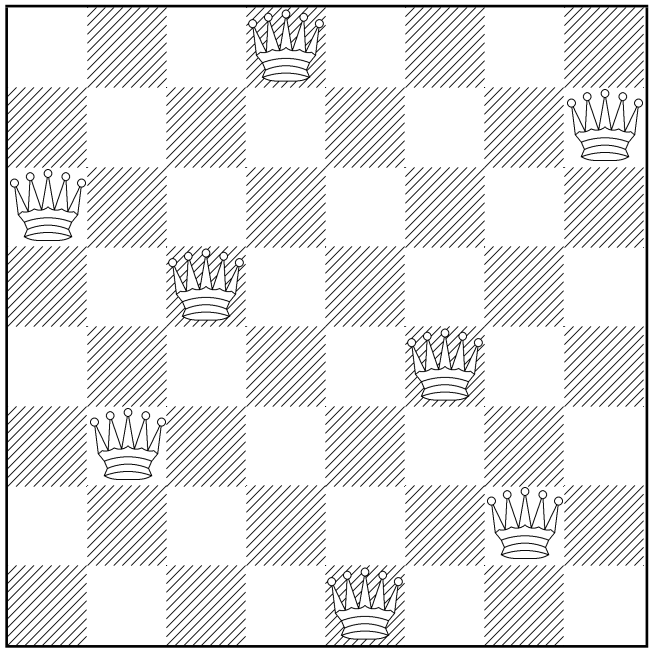

Figure 3.2: A solution of the eight queens problem.

Later, when we implement functions 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

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

If we define

$$B := \{V_1 : 4, V_2 : 8, V_3 : 1\},$$

then B is a partial variable assignment and $\texttt{dom}(B) = \{V_1, V_2, V_3\}$. This *partial variable assignment* is shown in Figure 3.3 on page 53.

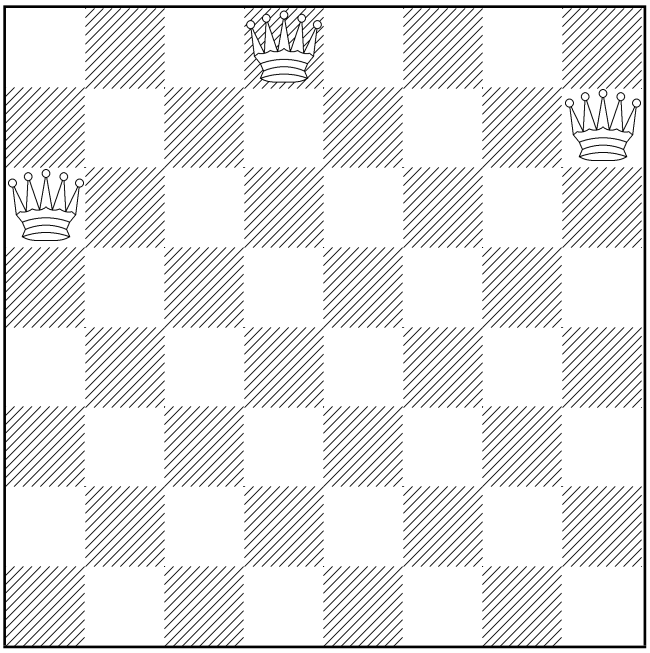

Figure 3.3: The partial assignment $\{V_1 : 4, V_2 : 8, V_3 : 1\}$.

Figure 3.4 on page 54 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 `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 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. In effect, this enables us to use declarative programming: Instead of specifying an algorithm that solves a problem we confine ourselves to specifying the problem precisely and then let a general purpose problem solver do the job of finding 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 the promisses made in declarative programming as a viable general purpose programming method, constraint programming has proven 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.

```
def create_csp(n):
    S            = range(1, n+1)
    Variables    = { f'V{i}' for i in S }
    Values       = set(S)
    SameRow      = { f'V{i} != V{j}' for i in S
                                     for j in S
                                     if  i < j
                   }
    SameDiagonal = { f'abs(V{j} - V{i}) != {j - i}' for i in S
                                                    for j in S
                                                    if  i < j
                   }
    return (Variables, Values, SameRow | SameDiagonal)
```

Figure 3.4: *Python* code to create the Csp representing the n-queens puzzle.

A simple example of a scheduling problem is the problem of generating a time table 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 time table. In practice, crew scheduling is an important problem. For example, airlines have to solve a crew scheduling problem in order to efficiently assign crews to aircrafts.

3.2 Brute Force Search

The most straightforward algorithm to solve a Csp is to test all possible combinations of assigning values to variables. If there are n different values that can be assigned to k variables, this amounts to checking n^k different assignments. For example, for the eight queens problem there are 8 variables and 8 possible values leading to

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

different assignments that need to be tested. Given the clock speed of modern computers, checking a million assignments per second is plausible. Hence, this approach is able to solve the eight queens problem in about 20 seconds. The approach of testing all possible combinations is known as brute force search. An implementation of brute force search is shown in Figure 3.5 on page 55.

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

$$P = (\texttt{Variables}, \texttt{Values}, \texttt{Constraints}).$$

The sole purpose of `search` is to call the function `brute_force_search`. The implementation of this function is recursive and the function takes two arguments.

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.

```
def solve(P):
    return brute_force_search({}, P)

def brute_force_search(Assignment, csp):
    Variables, Values, Constraints = csp
    if len(Assignment) == len(Variables): # all variables have been assigned
        if check_all_constraints(Assignment, Constraints):
            print('Assignment:', Assignment)
            return Assignment
        else:
            return None
    var = arb(Variables - set(Assignment.keys()))
    for value in Values:
        NewAss = Assignment.copy()
        NewAss[var] = value
        result = brute_force_search(NewAss, csp)
        if result != None:
            return result
```

Figure 3.5: Solving a CSP via brute force search.

2. `csp` is a triple of the form

   $$\texttt{csp} = (\texttt{Variables}, \texttt{Values}, \texttt{Constraints}).$$

   Here, `Constraints` is a set of first order logic formulæ that are given as strings. These strings have to follow the syntax of *Python* expressions 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 56. 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 the new assignment `NewAss` that satisfies

   $$\texttt{NewAss[var] = value}.$$

   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, another `value` is tried.

The function `check_all_constraints` takes a complete variable `Assignment` as its first input. The

```
19  def check_all_constraints(Assignment, Constraints):
20      A = Assignment.copy()
21      return all(eval(f, A) for f in Constraints)
```

Figure 3.6: Auxiliary functions for brute force search.

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 auxiliary function `eval_constraint`.

When I tested this brute force search with the eight queens problem, it took about 20 seconds to compute a solution. In contrast, the seven queens problem took roughly one second. As we have

$$\frac{8^8}{7^7} \approx 20$$

this shows that the computation time does indeed 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

Due to the combinatorial explosion of the search space, brute force search is only viable when we deal with small problems containing just a handful of variables. 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 backtrack and 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 57 shows a simple Csp solver that employs backtracking. We discuss this program next. The function `solve` takes a constraint satisfaction problem `P` as input and tries to find a solution.

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 3.9 on page 59. 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.

```
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

def backtrack_search(Assignment, P):
    Variables, Values, Constraints = P
    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()
                NewAss[var] = value
                return backtrack_search(NewAss, P)
        except Backtrack:
            continue
    raise Backtrack()
```

Figure 3.7: A backtracking CSP solver.

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 backtracking. 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.

Next, we discuss the implementation of the function `backtrack_search`. This function receives a partial assignment `Assignment` as input together with an augmented CSP. This partial assignment is consistent with `CSP`: 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 assignment that satisfies all constraints of the given `CSP`.

1. First, the augmented CSP 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 13 using the function `arb` that selects an arbitrary variable from its input set. This function is defined as follows:

```
def arb(S):
    for x in S:
        return x
```

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 `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 the `raise`-statement raises a `Backtrack` exception that is either called in the function `backtrack_search` or, if there is no solution, in the function `solve`.

```
1  def is_consistent(var, value, Assignment, Constraints):
2      NewAssign      = Assignment.copy()
3      NewAssign[var] = value
4      return all(eval(f, NewAssign) for (f, Vs) in Constraints
5                                    if  var in Vs and Vs <= NewAssign.keys()
6                 )
```

Figure 3.8: The definition of the function `is_consistent`.

We still need to discuss the implementation of the auxiliary function `is_consistent` shown in 3.8. 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

$$\texttt{vars(f)} \subseteq \texttt{dom(Assignment)}$$

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

$$\texttt{NewAssign} := \texttt{Assignment} \cup \{\texttt{var} \mapsto \texttt{value}\}$$

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

$$\texttt{Vars(Formula)} \subseteq \texttt{dom(NewAssign)},$$

i.e. all variables occurring in `Formula` need to have a value assigned in `NewAssign`. The reasoning is as follows:

1. If `var` does not occur in `Formula`, then adding `var` to `Assignment` cannot change the result of evaluating `Formula` and as `Assignment` is assumed to be partially consistent with respect to `Formula`, `NewAssign` is also partially consistent with respect to `Formula`.

2. If $\texttt{dom(NewAssign)} \not\subseteq \texttt{Vars(Formula)}$, then `Formula` can not be evaluated anyway.

```
1  import extractVariables as ev
2
3  def collect_variables(expr):
4      return { var for var in ev.extractVars(expr)
5                   if  var not in dir(__builtins__)
6             }
```

Figure 3.9: The function `collectVars`.

Finally, let us discuss the function `collect_variables` that is shown in Figure 3.9 on page 59. 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 a member of the list `dir(__builtin__)`.

If we use program discussed in this section, we can solve the 8 queens problem in less than 20 milliseconds. Hence, for the eight queens problem backtracking is about a thousand times faster than brute force search.

3.4 Constraint Propagation

Once we choose 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 1 in the second column, then no other queen can be placed in that column. Additionally, the queen in row 2 can then 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 `x` is inconsistent with the variable assignments that have already taken place for other variables, the value `v` will be removed from the set `ValuesPerVar[x]`.

2. As long as the problem 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 assigning the value `v` to the variable `x` does not violate a constraint, 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.

It turns out that elaborating the idea outline above can enhance the performance of backtracking search considerably. Figure 3.10 on page 60 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 without backtracking.

```
1  def solve(P):
2      Variables, Values, Constraints = P
3      Annotated    = { (f, collect_variables(f)) for f in Constraints }
4      ValuesPerVar = { v: Values for v in Variables }
5      UnaryConstrs = { (f, V) for f, V in Annotated
6                              if  len(V) == 1
7                      }
8      OtherConstrs = { (f, V) for f, V in Annotated
9                              if  len(V) >= 2
10                     }
11     try:
12         for f, V in UnaryConstrs:
13             var = arb(V)
14             ValuesPerVar[var] = solve_unary(f, var, ValuesPerVar[var])
15         return backtrack_search({}, ValuesPerVar, OtherConstrs)
16     except Backtrack:
17         return None
```

Figure 3.10: 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 `x` we need to know which values are admissible for `x`. To this end, we need a dictionary `ValuesPerVar` that contains the set of possible values for every variable `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 `x`, we inspect the constraints mentioning the variable `x` and shrink the set of values `ValuesPerVar`[`y`] that can be assigned to those variables `y` that are constrained by the variable `x`.

1. The function `solve` receives a `CSP`. This `CSP` 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`.

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`.

```
1  def solve_unary(f, x, Values):
2      Legal = { value for value in Values
3                      if  eval(f, { x: value })
4              }
5      if len(Legal) == 0:
6          raise Backtrack()
7      return Legal
```

Figure 3.11: Implementation of `solve_unary`.

The function `solve_unary` shown in Figure 3.11 on page 61 takes a unary constraint `f`, a variable `x` 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 `var` 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 3.12 on page 62 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 and a set of annotated `Constraints`. It tries to complete `Assignment` and thereby compute a solution of the given Csp.

```
def backtrack_search(Assignment, ValuesPerVar, Constraints):
    if len(Assignment) == len(ValuesPerVar):
        return Assignment
    x = most_constrained_variable(Assignment, ValuesPerVar)
    for v in ValuesPerVar[x]:
        try:
            if is_consistent(x, v, Assignment, Constraints):
                NewValues = propagate(x, v, Assignment,
                                      Constraints, ValuesPerVar)
                NewAssign = Assignment.copy()
                NewAssign[x] = v
                return backtrack_search(NewAssign, NewValues, Constraints)
        except Backtrack:
            continue
    raise Backtrack()
```

Figure 3.12: 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 variable is computed using the function `most_constrained_variable` that is shown in Figure 3.13 on page 63.

   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.

3. Next, all values that are still available for `x` are tried. 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 it is consistent to assign `v` to the variable `x`, we propagate the consequences of this assignment using the function `propagate` shown in Figure 3.14 on page 63. This function updates the values that are still allowed for the variables of theCsp once the value `v` has been assigned to the variable `x`.

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 3.13 on page 63 show 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 the set of values that still might be tried for this variable.

```
def most_constrained_variable(Assignment, ValuesPerVar):
    Unassigned = { (x, len(U)) for x, U in ValuesPerVar.items()
                               if  x not in Assignment
                 }
    minSize = min(lenU for x, lenU in Unassigned)
    for x, lenU in Unassigned:
        if lenU == minSize:
            return x
```

Figure 3.13: Finding a most constrained variable.

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 $(\texttt{x}, \texttt{len}(U))$, where U is the set of values that still might be tried for the variable `x`.
2. Next, `minSize` is the minimal number of values still available for a variable.
3. Finally, a variable `x` that has only `minSize` values available is returned.

```
def propagate(x, v, Assignment, Constraints, ValuesPerVar):
    ValuesDict = ValuesPerVar.copy()
    ValuesDict[x] = { v }
    BoundVars = set(Assignment.keys())
    for F, Vars in Constraints:
        if x in Vars:
            UnboundVars = Vars - BoundVars - { x }
            if len(UnboundVars) == 1:
                y = arb(UnboundVars)
                Legal = set()
                for w in ValuesDict[y]:
                    NewAssign = Assignment.copy()
                    NewAssign[x] = v
                    NewAssign[y] = w
                    if eval(F, NewAssign):
                        Legal.add(w)
                if len(Legal) == 0:
                    raise Backtrack()
                ValuesDict[y] = Legal
    return ValuesDict
```

Figure 3.14: Constraint Propagation.

The function `propagate` shown in Figure 3.14 on page 63 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 `x`. The implementation of `propagate` proceeds as follows.

1. As `x` is assigned the value `v`, the corresponding entry in the dictionary `ValuesPerVar` is changed accordingly.
2. `BoundVars` is the set of those variable that already have a value assigned.
3. Next, `propagate` 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 there is exactly one unbound variable `y` in the constraint `F`, then we can test those values that satisfy `F` and recompute the set `ValuesPerVar[x]`.
6. As the set `UnboundVars` contains just a single variable in line 7, the function `arb` returns this variable.
7. In order to recompute the set `ValuesPerVar[x]`, all values `w` in `ValuesPerVar[y]` are tested. The set `Legal` contains all values `w` that can be assigned to the variable `y` without violating the constraint `F`.
8. 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`.
9. Otherwise, the set of admissible values for `y` is updated to be the set `Legal`.
10. Finally, the updated dictionary `ValuesPerVar` is returned.

I have tested the program described in this section using the eight queens puzzle. It takes about 10 milliseconds to find a solution. I have also tested it with the Zebra Puzzle described in the next exercise. It solves this puzzle in 17 milliseconds. To compare, the backtracking algorithm shown in the previous section takes roughly 10 minutes to solve this puzzle.

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

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.
16. Who drinks water?
17. Who owns the zebra?

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

1. Each of the five houses is painted in a different colour.
2. The inhabitants of the five houses are of different nationalities,
3. they own different pets,
4. they drink different beverages, and
5. they 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. A framework for solving the zebra puzzle can be found at

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/Zebra-Frame.ipynb

When coding the zebra puzzle as a Csp you should be careful to name the variables correctly. A single spelling error can break your program. ◇

	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 6: Table 3.1 on page 66 shows a sudoku that I have taken from the Zeit Online magazine. Write a program that transforms this sudoku into a constraint satisfaction problem and solve the resulting Csp using the constraint satisfaction solver developed in this section. A framework for solving sudoku puzzles can be found at:

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

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 3.15 on page 66. 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

$$\texttt{S} \cdot 10^3 + \texttt{E} \cdot 10^2 + \texttt{N} \cdot 10^1 + \texttt{D} \cdot 10^0.$$

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 3.15: 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:

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

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 in Figure 3.15 column by column, just as it is taught in elementary school. Figure 3.16 on page 67 shows how this can be implemented in *Python*.

```
def crypto_csp():
    Digits       = { 'S', 'E', 'N', 'D', 'M', 'O', 'R', 'Y' }
    Variables    = Digits | { 'C1', 'C2', 'C3' }
    Values       = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
    Constraints  = allDifferent(Digits)
    Constraints |= { '(D + E)      % 10 == Y', '(D + E)      // 10 == C1',
                     '(N + R + C1) % 10 == E', '(N + R + C1) // 10 == C2',
                     '(E + O + C2) % 10 == N', '(E + O + C2) // 10 == C3',
                     '(S + M + C3) % 10 == O', '(S + M + C3) // 10 == M'
                   }
    Constraints |= { 'S != 0', 'M != 0' }
    Constraints |= { 'C1 < 2', 'C2 < 2', 'C3 < 2' }
    return Variables, Values, Constraints

def allDifferent(Variables):
    return { f'{x} != {y}' for x in Variables
                           for y in Variables
                           if x < y
       }
```

Figure 3.16: Formulating "SEND + MORE = MONEY" as a CSP.

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 do the addition of the last places of the two numbers, i.e. we have

$$\texttt{D} + \texttt{E} = \texttt{C1} \cdot 10 + \texttt{Y}.$$

This equation still contains four variables. We can reduce it to two equations that each involve only three variables as follows:

$$(\texttt{D} + \texttt{E}) \,\%\, 10 = \texttt{Y} \quad \text{and} \quad (\texttt{D} + \texttt{E}) \setminus 10 = \texttt{C1}.$$

Here, the symbol "//" denotes integer division, e.g. we have $7 \,//\, 3 = 2$ because $7 = 2 \cdot 3 + 1$. If we try to solve the cryptarithmetic puzzle as coded in Figure 3.16 on page 67 using the constraint solver developed in the previous section, we will be disappointed: Solving the puzzle takes about 7 seconds 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 if our constraint solver could figure out these constraints by itself. After all, since D and E are both less than 10, there 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

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

i.e. the variable x occurs in the constraint f and, furthermore, $R = \{y_1, \cdots, y_n\}$ is the set of all variables occurring in f that are different from x. Furthermore, assume that we have a dictionary ValuesPerVar such that for every variable y, the dictionary entry $\texttt{ValuesPerVar}[y]$ is the set of values that can be substituted for the variable y. 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 $y_i \in R$ we have to find a value $w_i \in \texttt{ValuesPerVar}[y_i]$ such that the resulting assignment $A = \{x \mapsto v, y_1 \mapsto w_1, \cdots, y_n \mapsto w_n\}$ satisfies the equations

$$\texttt{evaluate}(f, A) = \texttt{true}.$$

Here, the function evaluate takes a formula f and an assignment A and evaluates f using the assignment A. Now, consistency maintenance works as follows.

1. The dictionary ValuesPerVar is initialized as follows:

   $$\texttt{ValuesPerVar}[x] := \texttt{Values} \quad \text{for all } x \in \texttt{Variables},$$

   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:

   $$\texttt{UncheckedVariables} := \texttt{Variables}.$$

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

   $$x := \texttt{UncheckedVariables.pop}()$$

4. We iterate over all constraints f such that x occurs in f.

   (a) For every value $v \in \texttt{ValuesPerVar}[x]$ we check whether v is consistent with f.

   (b) If the value v is not consistent with f, then v is removed from $\texttt{ValuesPerVar}[x]$. 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 $\texttt{ValuesPerVar}[x]$.

5. Once 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 $\texttt{ValuesPerVar}[y]$ for some variable y. Although the set UncheckedVariables can grow during the algorithm, the union

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

can never grow: Every time the set `UncheckedVariables` grows, for some variable x the set `ValuesPerVar`$[x]$ 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.

```
def enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected):
    UncheckedVars = set(Var2Formulas.keys())
    while len(UncheckedVars) > 0:
        var         = UncheckedVars.pop()
        Constraints = Var2Formulas[var]
        Values      = ValuesPerVar[var]
        RemovedVals = set()
        for f in Constraints:
            OtherVars = Annotated[f] - { var }
            for value in Values:
                if not exists_values(var, value, f, OtherVars, ValuesPerVar):
                    RemovedVals   |= { value }
                    UncheckedVars |= Connected[var]
        Remaining = Values - RemovedVals
        if len(Remaining) == 0:
            raise Backtrack()
        ValuesPerVar[var] = Remaining
```

Figure 3.17: Consistency maintenance in Python.

Figure 3.17 on page 69 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.

(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 $y \in$ `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 `var` is selected and removed from `UncheckedVars`.
4. `Constraints` is the set of all constraints f such that `var` occurs in f.
5. `Values` is the set of those values that can be assigned to the variable `var`.
6. `RemovedVals` is the subset of those values that are found to be inconsistent with some constraint.
7. We iterate over all constraints `f` $\in$ `Constraints`.
8. `OtherVars` is the set of variables occurring in `f` that are different from the chosen variable `var`.
9. We iterate over all `value` $\in$ `Values` that can be substituted for the variable `var` 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`.
10. If we do not find such values, then `value` is inconsistent for the variables `var` w.r.t. `f` and needs to be removed from the set `ValuesPerVar`[`var`]. Furthermore, all variables that are connected to `var` 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.
11. If there are no consistent values for `var` left, we have to backtrack.
12. Otherwise, the set of values that are known to be consistent for `var` is stored as `ValuesPerVar`[`var`].

Figure 3.18 on page 71 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 `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`.

```
def exists_values(var, val, f, Vars, ValuesPerVar):
    Assignments = all_assignments(Vars, ValuesPerVar)
    return any(eval(f, extend(A, var, val)) for A in Assignments)

def extend(A, x, v):
    B = A.copy()
    B[x] = v
    return B

def all_assignments(Variables, ValuesPerVar):
    Variables = set(Variables) # turn frozenset into a set
    if len(Variables) == 0:
        return [ {} ]  # list containing empty assignment
    var         = Variables.pop()
    Values      = ValuesPerVar[var]
    Assignments = all_assignments(Variables, ValuesPerVar)
    return [ extend(A, var, val) for A in Assignments
                                 for val in ValuesPerVar[var]
           ]
```

Figure 3.18: The implementation of `exists_value`.

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` $\in$ `Values` to `var` and assigning the remaining variables according to an assignment `A` $\in$ `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 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 propagation. On the other hand, many difficult constraint satisfaction problems can not be solved without consistency checking.

Figure 3.19 on page 72 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] To 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 len(MisspelledVars) > 0:
        print("Did you misspell any of the following Variables?")
        for v in MisspelledVars:
            print(v)
    ValuesPerVar = { x: Values for x in Variables }
    Annotated    = { f: collect_variables(f) for f in Constraints }
    UnaryConstrs = { (f, V) for f, V in Annotated.items()
                            if  len(V) == 1
                   }
    OtherConstrs = { (f, V) for f, V in Annotated.items()
                            if  len(V) >= 2
                   }
    Connected    = {}
    Var2Formulas = variables_2_formulas(OtherConstrs)
    for x in Variables:
        Connected[x] = union([ V for f, V in Annotated.items()
                                 if  x in V
                             ]) - { x }
    try:
        for f, V in UnaryConstrs:
            var                = arb(V)
            ValuesPerVar[var] = solve_unary(f, var, ValuesPerVar[var])
        enforce_consistency(ValuesPerVar, Var2Formulas, Annotated, Connected)
        for x, Values in ValuesPerVar.items():
            print(f'{x}: {Values}')
        return backtrack_search({}, ValuesPerVar, OtherConstrs)
    except Backtrack:
        return None
```

Figure 3.19: A constraint solver with consistency checking as a preprocessing step.

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.

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 $\mathtt{y}_i$ occurring in f there is a value $w_i \in$ `ValuesPerVar`$[y]$ such that

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

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

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

local search works as follows:

1. Initialize the values of the variables in `Variables` randomly.

2. If all `Constraints` are satisfied, return the solution.

3. For every $x \in$ `Variables`, count the number of unsatisfied constraints that involve the variable x.

4. Set `maxNum` to be the maximum of these numbers, i.e. `maxNum` is the maximal number of unsatisfied constraints for any variable.

5. Compute the set `maxVars` of those variables that have `maxNum` unsatisfied constraints.

6. Randomly choose a variable x from the set `maxVars`.

7. 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.

```
def solve(P):
    Variables, Values, Constraints = P
    Variables = list(Variables)
    Values    = list(Values)
    Annotated = { (f, collect_variables(f)) for f in Constraints }
    Assign    = { x: random.choice(Values) 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))
        if maxNum == 0 and numConflicts(lastVar, Assign, Annotated) == 0:
            return Assign
        if iteration % 10  == 0:    # avoid infinite loop
            x = random.choice(Variables)
        else:     # choose var with max number of conflicts
            FaultyVars = [ var for (num, var) in Conflicts if  num == maxNum ]
            x = random.choice(FaultyVars)
        Conflicts = [ (numConflicts(x, extend(Assign, x, val), Annotated), val)
                      for val in Values
                    ]
        if iteration % 10 == 0:       # avoid infinite loop
            newVal = random.choice(Values)
        else:
            minNum, _  = Set.first(cast_to_Set(Conflicts))
            ValuesForX = [ val for (n, val) in Conflicts if n == minNum ]
            newVal     = random.choice(ValuesForX)
        Assign[x] = newVal
        lastVar   = x
        iteration += 1
```

Figure 3.20: A constraint solver using local search.

8. Rinse and repeat until a solution is found or the sun rises in the west.

Figure 3.20 on page 74 shows an implementation of these ideas in *Python*. We proceed to discuss this program line by line.

1. The function `solve` takes one parameter P, which is a Csp. If the computation is successful, `solve(P)` returns a dictionary that encodes a solution of the given Csp by mapping variables to values.

2. The sets `Variables` and `Values` are turned to lists. This is necessary because the function

   `random.choice`(L)

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

3. `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.

4. `Assign` is a dictionary mapping all variables form the set `Variables` to values from the set `Values`. Initially the values are assigned randomly.

5. The variable `iteration` counts the number of times that we have changed the assignment `Assign` by reassigning a variable.

6. If we have reassigned a variable x in the last iteration of the loop, then we do not want to reassign it again. Therefore, the variable `lastVar` stores the variable that has been reassigned in the previous iteration.

7. 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.

   This is done using the function `numConflicts` shown in Figure 3.21 on page 76. The list `Conflicts` defined in line 10 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.

8. In line 13 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.

9. 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.

10. Otherwise, the list `FaultyVars` defined in line 19 collects those variables that have a maximal number of conflicts.

11. In line 20 we choose a random variable `x` from this set as the variable to be reassigned. However, this is only done nine out of ten times. In order to avoid running into an infinite where we keep changing the same variables, every tenth iteration instead of choosing `x` as the variable violating the most constraints we choose `x` randomly. This is controlled by the test `iteration % 10 == 0` in line 16.

12. Line 21 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`.

13. Line 27 casts the list `Conflicts` 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 smallest number of conflicts that any value v causes when `x` is assigned to v.

14. `ValuesForX` is the list of those values that cause only `minNum` conflicts when assigned to `x`.

15. `newVal` is a random element from this list that is then assigned to `x`. Again, this is only done 9 out of ten times. The tenth time a random value is assigned to `x`.

16. In line 31 we remember that we have reassigned `x` in this iteration so that we don't reassign `x` in the next iteration again.

```
1  def numConflicts(x, Assign, Annotated):
2      NewAssign = Assign.copy()
3      return len([ (f, V) for (f, V) in Annotated
4                          if  x in V and not eval(f, NewAssign)
5                  ])
```

Figure 3.21: The function `numConflicts`.

The function `numConficts` is shown in Figure 3.21 on page 76. 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`$(x, \texttt{Assign}, \texttt{Annotated})$ 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. Hence, it seems that local search is better than the algorithms discussed previously. 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.

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 $3\frac{1}{2}-2\frac{1}{2}$. While Deep Blue was based on special hardware, according to the computer chess rating list of the 8th of February 2020, the chess program Stockfish runs on ordinary desktop computers and has an Elo rating of 3495. To compare, according to the Fide list of February 2020, the current World Chess Champion Magnus Carlsen has an Elo rating of just 2862. Hence, he wouldn't stand a chance to win 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.

In this chapter we want to investigate how a computer can play a game. To this end we define a game $\mathcal{G}$ as a six-tuple

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

where the components are interpreted as follows:

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

2. $s_0 \in \texttt{States}$ is the start state.

3. `Players` is the list of 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 $s \in \texttt{States}$ and a player $p \in \texttt{Players}$ 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:

   $$\texttt{nextStates} : \texttt{States} \times \texttt{Players} \to 2^{\texttt{States}}.$$

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

   $$\texttt{finished} : \texttt{States} \to \mathbb{B}.$$

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

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

$$\texttt{TerminalStates} := \{s \in \texttt{States} \mid \texttt{finished}(s)\}.$$

6. `utility` is a function that takes a state $s \in \texttt{TerminalStates}$ and a player $p \in \texttt{Players}$. It returns the value that the game has for player p. In general, a value is a real number, but in all of our examples, this value will be an element from the set $\{-1, 0, +1\}$. The value -1 indicates that player p has lost the game, if the value is $+1$ the player p has won the game, and if this value is 0, then the game is a draw. Hence the signature of `utility` is

$$\texttt{utility} : \texttt{TerminalStates} \times \texttt{Players} \to \{-1, 0, +1\}.$$

In this chapter 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

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

then the game is called a zero sum game iff we have

$$\forall s \in \texttt{TerminalStates} : \texttt{utility}(s, \texttt{A}) + \texttt{utility}(s, \texttt{B}) = 0,$$

i.e. the losses of player A are compensated by the wins of player `B` and vice versa. Games like go and chess are two person, zero sum games. We proceed to discuss an example.

4.1 Tic-Tac-Toe

The game tic-tac-toe is played on a square board of size 3×3. On every turn, one player puts an "`X`" on one of the free squares of the board, while the other player puts an "`O`" onto a free square when it is his turn. If the first player manages to place three `X`s in a row, column, or diagonal, she has won the game. Similarly, if the second player manages to put three `O`s in a row, column, or diagonal, this player is the winner. Otherwise, the game is drawn. Figure 4.1 on page 79 shows a *Python* implementation of tic-tac-toe.

1. The variable `Players` stores the list of players. Traditionally, the players in tic-tac-toe are called "`X`" and "`O`".

2. The variable `Start` 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`", "`O`", and the blank character " ". As the state `Start` 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`.

```
Players  = [ "X", "O" ]
Start    = tuple( tuple(" " for col in range(3)) for row in range(3))
to_list  = lambda State: [list(row) for row in State]
to_tuple = lambda State: tuple(tuple(row) for row in State)

def empty(S):
    return [ (row, col) for row in range(3)
                        for col in range(3)
                        if  S[row][col] == ' '
           ]

def next_states(State, player):
    Empty  = empty(State)
    Result = []
    for row, col in Empty:
        NextState           = to_list(State)
        NextState[row][col] = player
        Result.append( to_tuple(NextState) )
    return Result

All_Lines = [ [ (row, col) for col in range(3) ] for row in range(3) ] \
          + [ [ (row, col) for row in range(3) ] for col in range(3) ] \
          + [ [ (idx,   idx) for idx in range(3) ] ]                   \
          + [ [ (idx, 2-idx) for idx in range(3) ] ]

def utility(State, player):
    for Pairs in All_Lines:
        Marks = { State[row][col] for row, col in Pairs }
        if len(Marks) == 1 and  Marks != { ' ' }:
            if Marks == { player }:
                return  1
            else:
                return -1
    for row in range(3):
        for col in range(3):
            if State[row][col] == ' ':
                return None
    return 0

finished = lambda State: utility(State, "X") != None
```

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

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 $(\texttt{row}, \texttt{col})$ such that $S[\texttt{row}][\texttt{col}]$ 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 "O".

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 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 and a player as arguments. If the game is finished in the given State, it returns the value that this State has for the current player. 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 All_Lines has the following value:

   $$\Big[\big[(1,1),(1,2),(1,3)\big],\ \big[(2,1),(2,2),(2,3)\big],\ \big[(3,1),(3,2),(3,3)\big],$$
   $$\big[(1,1),(2,1),(3,1)\big],\ \big[(1,2),(2,2),(3,2)\big],\ \big[(1,3),(2,3),(3,3)\big],$$
   $$\big[(1,1),(2,2),(3,3)\big],\ \big[(3,1),(2,2),(1,3)\big] \Big]$$

   The first line in this expression gives the set of pairs defining the rows, the second line defines the columns, and the last line yields the tow 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 all_lines 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 the same as player, then the game is won by player, otherwise it must be the mark of his opponent and hence the game is lost for him.

   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.

4.2 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 explain 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*. While the utility is only defined for terminal states, the value is defined for all states. Formally, we define a function

$$\texttt{value} : \texttt{States} \times \texttt{Players} \rightarrow \{-1, 0, +1\}$$

that takes a state $s \in \texttt{States}$ and a player $p \in \texttt{Players}$ and returns the value of s provided both the player p and his opponent play optimally. The easiest way to define this function is via recursion. As the `value` function is an extension of the `utility` function, The base case is as follows:

$$\texttt{finished}(s) \rightarrow \texttt{value}(s, p) = \texttt{utility}(s, p). \tag{1}$$

If the game is not yet finished, assume that player o is the opponent of player p. Then we define

$$\neg\texttt{finished}(s) \rightarrow \texttt{value}(s, p) = \max\Big(\big\{-\texttt{value}(n, o) \mid n \in \texttt{nextStates}(s, p)\big\}\Big). \tag{2}$$

The reason is that, if the game is not finished yet, the player p has to evaluate all possible moves. From these, the player p will choose the move that maximizes the value of the game for herself. In order to do so, the player p computes the set $\texttt{nextStates}(s, p)$ of all states that can be reached from the state s in any one move of the player p. Now if n is a state that results from player p making some move, then in state n it is the turn of the other player o to make a move. Hence, in order to evaluate the state n, we have to call the function `value` recursively as $\texttt{value}(n, o)$. Since the gains of the other player o are the losses of the player p, we have to take the negative of $\texttt{value}(n, o)$. Figure 4.2 on page 82 shows an implementation of this strategy.

1. Given a player `p`, the function `other` computes the other player, which is the first element of the list `Players` that is different from p. This works because we assume that there are just two players and these players are the two elements of the list `Players` that has been defined in the notebook defining the game.

2. `Cache` is a dictionary that is initially empty. This dictionary is used as a memory cache by the function `memoize`.

3. 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 `Cache`. If this is successful, the function returns the cached value. Otherwise, the function f is called to compute the result. This result is then stored in the `Cache` before it is returned. The function `memoize` returns the memoized version of f.

4. The implementation of the function `value` implements the formulas (1) and (2) that were used to define the function `value` abstractly. However, note that we have preceded the definition

```
other = lambda p: [o for o in Players if o != p][0]

Cache = {}

def memoize(f):
    global Cache

    def f_memoized(*args):
        if args in Cache:
            return Cache[args]
        result = f(*args)
        Cache[args] = result
        return result

    return f_memoized

@memoize
def value(State, player):
    if finished(State): return utility(State, player)
    return max([-value(ns, other(player)) for ns in next_states(State, player)])

def best_move(State, player):
    NS        = next_states(State, player)
    bestVal   = value(State, player)
    BestMoves = [s for s in NS if -value(s, other(player)) == bestVal]
    BestState = random.choice(BestMoves)
    return bestVal, BestState

def play_game(canvas):
    State = Start
    while (True):
        firstPlayer = Players[0]
        val, State  = best_move(State, firstPlayer);
        draw(State, canvas)
        if finished(State):
            final_msg(State)
            break
        State = get_move(State)
        draw(State, canvas)
        if finished(State):
            final_msg(State)
            break
```

Figure 4.2: The Minimax algorithm.

of the function `value` with the decorator `@memoize`, which turns the function `value` into a memoized function. Hence, when the function `value` is called a second time with the same pair

of arguments, it does not recompute the value but rather the value is looked up from the variable `Cache` that stores all previous results computed by the function `value`. To understand why this is important, let us consider how many states would be explored in the case of tic-tac-toe if we would not use the idea of memorizing previous results. In this case, we have 9 moves for player `X` from the start state, then 8 moves for player `O`, then again 7 moves for player `O`. If we disregard the fact that some games are decided after fewer than 9 moves, the function `value` needs to consider

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

moves. However, if we count the number of possibilities of putting 5 "`O`"s and 4 "`X`"s 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 to put 5 "`O`"s on 9 positions and that number is the same as the number of subsets of five elements from a set of nine 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 to evaluate if we use memoization!

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

5. The function `best_move` takes a `State` and a `player` and returns a pair (v, s) where s is a state that is optimal for the `player` 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 `player` is to move next.

   (b) `bestValue` is the best value that `player` can achieve in the given `State`.

   (c) `BestMoves` is the set of states that `player` can move to and that are optimal for her.

   (d) The function returns randomly one of those states `ns` $\in$ `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.

6. The function `play_game` is used to play a game.

   (a) Initially, `State` is the `startState`.

   (b) As long as the game is not finished, the procedure keeps running.

   (c) We assume that the computer goes first and therefore define `firstPlayer` as the first element of the list `Players`. Next, the function `best_move` is used to compute the state that results from the best move of `firstPlayer`. This resulting state is then shown.

   (d) After that, it is checked whether the game is finished.

   (e) If the game is not yet finished, the user is asked to make its move via the function `get_move` that asks the user to enter a move. The state resulting from this move is then returned and displayed.

(f) Next, we have to check whether the game is finished after the move of the user has been executed.

(g) The `while`-loop keeps iterating until the game is finished. We do not have to put a test into the condition of this `while`-loop as we call the function `finished(State)` every time that a new `State` has been reached. If the game is finished, a message giving the result of the game is printed.

In order to better understand the reason for using memoization in the implementation of the function `value` we introduce the following notions.

Definition 5 (Game Tree) Assume that

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

is a game. Then a play of length n is a list of states of the form

$$[s_0, s_1, \cdots, s_n] \quad \text{such that} \quad \forall i \in \{0, \cdots, n-1\} : s_{i+1} \in \texttt{nextStates}(s_i, p_i),$$

where the players p_i are defined such that for all even $i \in \{0, \cdots, n-1\}$ we have that p_i is the first element in the list `Players`, while p_i is the second element otherwise. The game tree of the game $\mathcal{G}$ is the set of all possible plays. ◇

The following exercise shows why memoization is important.

Exercise 7: 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 `X`s than the player `O` has rows, columns, or diagonals of three `O`s. Similarly, the player `O` wins if he has more rows, columns, or diagonals of three `O`s than the player `X` has rows, columns, or diagonals of three `X`s. 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) Write a short program to evaluate the formula derived in part (c) of this exercise.

4.3 α-β-Pruning

The efficiency of the minimax algorithm can be improved if we provide two additional arguments to the function `value`. Traditionally, these arguments are called α and β. In order to be able to distinguish between the old function `value` and its improved version, we call the improved version `alphaBeta`. The idea is that the function `alphaBeta` and the function `value` are related by the following requirements:

1. As long as $\texttt{value}(s, p)$ is between α and β, the function `alphaBeta` computes the same result as the function `value`, i.e. we have

   $$\alpha \leq \texttt{value}(s, p) \leq \beta \;\rightarrow\; \texttt{alphaBeta}(s, p, \alpha, \beta) = \texttt{value}(s, p).$$

2. If $\texttt{value}(s, p) < \alpha$, we require that the value returned by `alphaBeta` is less than or equal to α, i.e. we have

   $$\texttt{value}(s, p) < \alpha \;\rightarrow\; \texttt{alphaBeta}(s, p, \alpha, \beta) \leq \alpha.$$

3. Similarly, if $\texttt{value}(s, p) > \beta$, we require that the value returned by `valueAlphaBeta` is bigger than or equal to β, i.e. we have

   $$\beta < \texttt{value}(s, p) \;\rightarrow\; \beta \leq \texttt{alphaBeta}(s, p, \alpha, \beta).$$

Therefore, $\texttt{alphaBeta}(\texttt{State}, \texttt{player})$ is only an approximation of $\texttt{value}(\texttt{State}, \texttt{player})$. However, it turns out that this approximation is all that is needed. Figure 4.3 on page 85 shows an implementation of the function `alphaBeta` that satisfies the specification given above. Once the function `alphaBeta` is implemented, the function `value` can then be computed as

$$\texttt{value}(s, p) := \texttt{alphaBeta}(s, p, -1, +1).$$

The reason is that we already know that $-1 \leq \texttt{value}(s, p) \leq +1$ and hence the first case of the specification of `alphaBeta` guarantees that the equation

$$\texttt{value}(s, p) = \texttt{alphaBeta}(s, p, -1, +1)$$

holds. Since `alphaBeta` is implemented as a recursive procedure, the fact that the implementation of `alphaBeta` shown in Figure 4.3 on page 85 satisfies the specification given above can be established by computational induction. A proof by computational induction can be found in an article by Donald E. Knuth and Ronald W. Moore [KM75].

```
1  def alphaBeta(State, player, alpha, beta):
2      if finished(State):
3          return utility(State, player)
4      val = alpha
5      for ns in next_states(State, player):
6          val = max(val, -value(ns, other(player), -beta, -alpha))
7          if val >= beta:
8              return val
9          alpha = max(val, alpha)
10     return val
```

Figure 4.3: α-β-Pruning.

We proceed to discuss the implementation of the function `alphaBeta`, which is shown in Figure 4.3 on page 85.

1. If `State` is a terminal state, the function returns the `utility` of the given `State` with respect to `player`.

2. The variable `val` is supposed to store the maximum of the values of all states that can be reached from the given `State` if `player` makes one move.

   According to the specification of `alphaBeta`, we are not interested in values that are less than `alpha`. Hence, it suffices to initialize `val` with `alpha`. This way, in the case that we have

   $$\texttt{value}(\texttt{State}, \texttt{player}) < \alpha,$$

   instead of returning the true value of the given `State`, the function $\texttt{alphaBeta}(\texttt{State}, \texttt{player}, \alpha, \beta)$ will instead return the value α, which is permitted by its specification.

3. Next, we iterate over all successor states `ns` $\in$ `next_states(State, player)`.

4. We have to recursively evaluate the states `ns` with respect to the opponent `other(player)`. Since the value of a state for the opponent is the negative of the value for `player`, we have to exchange the roles of α and β and prefix them with a negative sign. Note that in the recursive call we do not call the function `alphaBeta` directly, but rather we call the function `value`. This function is a wrapper for the function `alphaBeta`. The purpose of this wrapper is to memoize the function `alphaBeta`. This is more complicated now due to the presence of the parameters α and β. The details will be explained below when we discuss the function `value` that is shown in Figure 4.4.

5. As the specification of `alphaBeta` ask us to compute the value of `State` only in those cases where it is less than or equal to β, once we find a successor state s that has a value `val` that is at least as big as β we can stop any further evaluation of the successor states and return the value `val`.

   In practice, this shortcut results in significant savings of computation time!

6. Once we have found a successor state that has a value `val` greater than `alpha`, we can increase `alpha` to the value `val`. The reason is, that once we know we can achieve a value of `val` we are no longer interested in any values that are less than `val`. This is the reason for assigning to `alpha` the maximum of `val` and `alpha`.

```
1  def value(State, player, alpha=-1, beta=1):
2      global Cache
3      if State in Cache:
4          val, a, b = Cache[State]
5          if a <= alpha and beta <= b:
6              return val
7          else:
8              alpha = min(alpha, a)
9              beta  = max(beta , b)
10             val   = alphaBeta(State, player, alpha, beta)
11             Cache[State] = val, alpha, beta
12             return val
13     else:
14         val = alphaBeta(State, player, alpha, beta)
15         Cache[State] = val, alpha, beta
16         return val
```

Figure 4.4: The function `value` that memoizes the function `alphaBeta`.

The function `value` shown in Figure 4.4 is merely a wrapper of the function `alphaBeta` that memoizes the results of `alphaBeta`. This is more complicated now because of the parameters α and β. Of course, we could just use the decorator `memoize`. The problem with this approach is that the function `alphaBeta` might be called with the same value for its parameters `State` and `player`, but different values for the parameters `alpha` and `beta`. This would have two consequences:

1. The directory `Cache` would require much more memory as there are many more combinations of the parameters of the function `alphaBeta` then there had been combinations for the function `value` when we had implemented the minimax algorithm.

2. The directory `Cache` would become much less useful than in our implementation of the minimax algorithm because we would have many cache misses.

The solution is to ensure that any `State` is stored at most once in `Cache`. But instead of storing just the value, we also have to store the values of the parameters `alpha` and `beta` that have been used to compute the value of the given `State`, i.e. we now have

$$\texttt{Cache}[\texttt{State}] = (\texttt{value}, \texttt{alpha}, \texttt{beta})$$

if we have computed that

$$\texttt{alphaBeta}(\texttt{State}, \texttt{player}, \texttt{alpha}, \texttt{beta}) = \texttt{value}.$$

It turns out that there is no need to store the argument `player` since in the games that we consider this information can always be computed from the `State`.

Now the crucial idea of our implementation the function `value` is the following: If we ever want to compute

$$\texttt{alphaBeta}(\texttt{State}, \texttt{player}, \texttt{alpha}, \texttt{beta})$$

and we have that

$$\texttt{Cache}[\texttt{State}] = (\texttt{value}, a, b),$$

then we have to check whether the interval $[\texttt{alpha}, \texttt{beta}]$ is more general than the interval $[a, b]$, i.e. we have to check that

$$a \leq \texttt{alpha} \quad \text{and} \quad \beta \leq b$$

holds. In this case we know that indeed

$$\texttt{alphaBeta}(\texttt{State}, \texttt{player}, \texttt{alpha}, \texttt{beta}) = \texttt{value}$$

and hence we can use the value stored in the `Cache`. If the inequations $a \leq \texttt{alpha}$ and $\beta \leq b$ are not satisfied, then we compute

$$\texttt{alphaBeta}(\texttt{State}, \texttt{player}, \min(\texttt{alpha}, a), \max(\texttt{beta}, b))$$

and store the resulting value together with the new interval $[\min(\texttt{alpha}, a), \max(\texttt{beta}, b)]$ in the `Cache`. This way, the value stored in the `Cache` is more general than the previous value stored and we can expect more cache hits later.

4.4 Depth Limited Search

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 believed[1] that in chess there are about 10^{50} 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 game: Usually, a player considers all variations of the game for, say, the next three moves. After a given number of moves, the value of a position is estimated using an evaluation function. This function approximates the true value of a given state via a heuristic.

[1] For 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 10^{50}.

In order to implement this idea, we add a parameter `limit` to the procedure `alphaBeta` that was shown in the previous section. On every recursive invocation of the function `alphaBeta`, the parameter `limit` is decreased. Once the limit reaches 0, instead of invoking the function `alphaBeta` again recursively, we try to estimate the value of the given `State` using our evaluation function. This leads to the code shown in Figure 4.5 on page 88.

```
1  def alphaBeta(State, player, limit, heuristic, alpha=-1, beta=1):
2      if finished(State):
3          return utility(State, player)
4      if limit == 0:
5          return heuristic(State, player)
6      val = alpha
7      for ns in next_states(State, player):
8          val_ns = value(ns, other(player), limit-1, heuristic, -beta, -alpha)
9          val    = max(val, -val_ns)
10         if val >= beta:
11             return val
12         alpha = max(val, alpha)
13     return val
```

Figure 4.5: Depth-limited α-β-pruning.

When we compare this Figure with Figure 4.3 on page 85, 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 our `heuristic` function. Later in the recursive calls of the function `value` we have to take care to decrease the parameter `limit`.

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:

```
heuristic := [State, player] |-> 0;
```

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 $\texttt{Rook}_{\text{white}}$ and black rooks $\texttt{Rook}_{\text{black}}$, subtract these numbers from each other and multiply the difference by 5. The resulting number would be added to `sum`:

   $$\texttt{sum} \mathrel{+}= (\texttt{Rook}_{\text{white}} - \texttt{Rook}_{\text{black}}) \cdot 5;$$

3. We would count the number of white bishops $\texttt{Bishop}_{\text{white}}$ and black bishops $\texttt{Bishop}_{\text{black}}$, subtract these numbers from each other and multiply the difference by 3. The resulting number would be added to `sum`:

   $$\texttt{sum} \mathrel{+}= (\texttt{Bishop}_{\text{white}} - \texttt{Bishop}_{\text{black}}) \cdot 3;$$

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 8: Read up on the game Connect Four. You can play it online at

`http://www.connectfour.org/connect-4-online.php`

Your task is to implement this game. On my github page (`https://github.com/karlstroetmann`) at

Artificial-Intelligence/blob/master/Python/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 `X`s or `O`s 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 a triples of `X`s and b triples of `O`s and the game is not finished, then define

$$\texttt{value}(s, \texttt{X}, \texttt{limit}, \alpha, \beta) = \frac{a-b}{10} \quad \text{if } \texttt{limit} = 0. \qquad \diamond$$

Chapter 5

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.

5.1 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 $\langle x_1, y_1\rangle, \cdots, \langle x_m, y_m\rangle$ 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:

$$\mathbf{x} := \langle x_1, \cdots, x_m\rangle^\top \quad \text{and} \quad \mathbf{y} := \langle y_1, \cdots, y_m\rangle^\top.$$

Here, the 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 I am able to write these vectors in a single line.

In linear regression, we use a linear hypothesis 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 mean squared error, which is defined as

$$\texttt{MSE}(\vartheta_0, \vartheta_1) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\big)^2, \tag{5.1}$$

is minimized. It can be shown that the solution to this minimization problem is given as follows:

$$\vartheta_1 = r_{x,y} \cdot \frac{s_y}{s_x} \quad \text{and} \quad \vartheta_0 = \bar{\mathbf{y}} - \vartheta_1 \cdot \bar{\mathbf{x}}. \tag{5.2}$$

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

$$\bar{\mathbf{x}} = \frac{1}{m} \cdot \sum_{i=1}^{m} x_i \quad \text{and} \quad \bar{\mathbf{y}} = \frac{1}{m} \cdot \sum_{i=1}^{m} y_i.$$

Furthermore, s_x and s_y are the sample standard deviations of $\mathbf{x}$ and $\mathbf{y}$, i.e. we have

$$s_x = \sqrt{\frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(x_i - \bar{\mathbf{x}}\big)^2} \quad \text{and} \quad s_y = \sqrt{\frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(y_i - \bar{\mathbf{y}}\big)^2}\,.$$

Next, $\mathrm{Cov}[\mathbf{x}, \mathbf{y}]$ is the sample covariance and is defined as

$$\mathrm{Cov}[\mathbf{x}, \mathbf{y}] = \frac{1}{(m-1)} \cdot \sum_{i=1}^{m} \big(x_i - \bar{\mathbf{x}}\big) \cdot \big(y_i - \bar{\mathbf{y}}\big).$$

Finally, $r_{x,y}$ is the sample correlation coefficient that is defined as

$$r_{x,y} = \frac{1}{(m-1) \cdot s_x \cdot s_y} \cdot \sum_{i=1}^{m} \big(x_i - \bar{\mathbf{x}}\big) \cdot \big(y_i - \bar{\mathbf{y}}\big) = \frac{\mathrm{Cov}[\mathbf{x}, \mathbf{y}]}{s_x \cdot s_y}.$$

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

$$\vartheta_1 = \frac{\sum\limits_{i=1}^{m} \big(x_i - \bar{\mathbf{x}}\big) \cdot \big(y_i - \bar{\mathbf{y}}\big)}{\sum\limits_{i=1}^{m} \big(x_i - \bar{\mathbf{x}}\big)^2} \tag{5.3}$$

This latter formula should be used to calculate ϑ_1. However, the previous formula is also useful because it shows the 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 9: Prove Equation 5.2 and Equation 5.3.

Hint: Take the partial derivatives of $\mathtt{MSE}(\vartheta_0, \vartheta_1)$ with respect to ϑ_0 and ϑ_1. If the expression $\mathtt{MSE}(\vartheta_0, \vartheta_1)$ is minimal, then these partial derivatives have to be equal to 0. ◇

5.1.1 Assessing the Quality of Linear Regression

Assume that we have been given a set of m observations of the form $\langle x_1, y_1\rangle, \cdots, \langle x_m, y_m\rangle$ and that we have calculated the parameters ϑ_0 and ϑ_1 according to Equation 5.2 and Equation 5.3. 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 5.1. 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

$$\mathtt{Var}(y) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(y_i - \bar{y}\big)^2. \tag{5.4}$$

If we compare this formula to the formula for the mean squared error

$$\mathtt{MSE}(\vartheta_0, \vartheta_1) := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\big)^2 = \frac{1}{m-1} \cdot \sum_{i=1}^{m} \big(y_i - \vartheta_1 \cdot x_i - \vartheta_0\big)^2,$$

we see that the sample variance of $\mathbf{y}$ is an upper bound for the mean squared error since we have

$$\mathtt{Var}(\mathbf{y}) = \mathtt{MSE}(\bar{\mathbf{y}}, 0),$$

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

$$\mathtt{MSE}(\vartheta_0, \vartheta_1) \leq \mathtt{MSE}(\bar{\mathbf{y}}, 0) = \mathtt{Var}(\mathbf{y}).$$

The mean squared error is an absolute value and, therefore, difficult to interpret. The fraction

$$\frac{\mathtt{MSE}(\vartheta_0, \vartheta_1)}{\mathtt{Var}(y)}$$

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 $\mathtt{R}^2$ statistic is defined as

$$\mathtt{R}^2 := \frac{\mathtt{Var}(\mathbf{y}) - \mathtt{MSE}(\vartheta_0, \vartheta_1)}{\mathtt{Var}(\mathbf{y})} = 1 - \frac{\mathtt{MSE}(\vartheta_0, \vartheta_1)}{\mathtt{Var}(\mathbf{y})}. \tag{5.5}$$

The statistic $\mathtt{R}^2$ 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 $\mathtt{Var}(\mathbf{y})$ and $\mathtt{MSE}(\vartheta_0, \vartheta_1)$ have the same denominator $m-1$, this denominator can be cancelled when $\mathtt{R}^2$ is computed. To this end we define the total sum of squares $\mathtt{TSS}$ as

$$\mathtt{TSS} := \sum_{i=1}^{m} \big(y_i - \bar{\mathbf{y}}\big)^2 = (m-1) \cdot \mathtt{Var}(\mathbf{y})$$

and the residual sum of squares $\mathtt{RSS}$ as

$$\mathtt{RSS} := \sum_{i=1}^{m} \big(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\big)^2 = (m-1) \cdot \mathtt{MSE}(\vartheta_0, \vartheta_1).$$

Then the formula for the $\mathtt{R}^2$ statistic can be written as

$$\mathtt{R}^2 = 1 - \frac{\mathtt{RSS}}{\mathtt{TSS}}.$$

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 extend the value of the variable y is determined by the value of x.

5.1.2 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 5.1 on page 93 shows the head of the data file "`cars.csv`" which I have adapted from the file

`http://faculty.marshall.usc.edu/gareth-james/ISL/Auto.csv`.

Figure 5.1 on page 93 shows the column headers and the first ten data entries contained in this file. Altogether, this file contains data 392 different car models.

```
1    mpg, cyl, displacement,    hp, weight,  acc, year, name
2   18.0,   8,        307.0, 130.0, 3504.0, 12.0,   70, chevrolet chevelle malibu
3   15.0,   8,        350.0, 165.0, 3693.0, 11.5,   70, buick skylark 320
4   18.0,   8,        318.0, 150.0, 3436.0, 11.0,   70, plymouth satellite
5   16.0,   8,        304.0, 150.0, 3433.0, 12.0,   70, amc rebel sst
6   17.0,   8,        302.0, 140.0, 3449.0, 10.5,   70, ford torino
7   15.0,   8,        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
9   14.0,   8,        440.0, 215.0, 4312.0,  8.5,   70, plymouth fury iii
10  14.0,   8,        455.0, 225.0, 4425.0, 10.0,   70, pontiac catalina
11  15.0,   8,        390.0, 190.0, 3850.0,  8.5,   70, amc ambassador dpl
```

Figure 5.1: 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 1.63871 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 0.453592 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 5.2 on page 94.

```
1  def simple_linear_regression(X, Y):
2      """
3      This function implements linear regression.
4
5      * X:      explaining variable, numpy array
6      * Y:      dependent  variable, numpy array
7
8      Output: The R2 value of the linear regression.
9      """
10     m      = len(X)
11     xMean  = np.mean(X);
12     yMean  = np.mean(Y);
13     theta1 = np.sum( (X - xMean) * (Y - yMean) ) / np.sum((X - xMean) ** 2)
14     theta0 = yMean - theta1 * xMean;
15     TSS    = np.sum((Y - yMean) ** 2)
16     RSS    = np.sum((theta1 * X + theta0 - Y) ** 2)
17     R2     = 1 - RSS / TSS;
18     return R2
```

Figure 5.2: 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 `y`.
4. The coefficient `theta1` is computed according to Equation 5.3, which is repeated here for convenience:

$$\vartheta_1 = \frac{\sum_{i=1}^{m} \left(x_i - \bar{\mathbf{x}}\right) \cdot \left(y_i - \bar{\mathbf{y}}\right)}{\sum_{i=1}^{m} \left(x_i - \bar{\mathbf{x}}\right)^2}.$$

   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 5.2, 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

$$\texttt{TSS} = \sum_{i=1}^{m} \left(y_i - \bar{\mathbf{y}}\right)^2.$$

7. `RSS` is the residual sum of squares and is computed as

$$\texttt{RSS} := \sum_{i=1}^{m} \left(\vartheta_1 \cdot x_i + \vartheta_0 - y_i\right)^2.$$

8. `R2` is the $\texttt{R}^2$ statistic and measures the proportion of the explained variance. It is computed using the formula

$$\texttt{R}^2 = \frac{\texttt{TSS} - \texttt{RSS}}{\texttt{TSS}}.$$

```
1  import csv
2  import numpy as np
3
4  with open('cars.csv') as input_file:
5      reader       = csv.reader(input_file, delimiter=',')
6      line_count   = 0
7      kpl          = []
8      displacement = []
9      for row in reader:
10         if line_count != 0:  # skip header of file
11             kpl .append(float(row[0]) * 0.00425144)
12             displacement.append(float(row[2]) * 0.0163871)
13         line_count += 1
14 m  = len(displacement)
15 X  = np.array(displacement)
16 Y  = np.array([1 / kpl[i] for i in range(m)])
17 R2 = simple_linear_regression(X, Y)
18 print(f'The explained variance is {R2}%')
```

Figure 5.3: Calling the procedure `simple_linear_regression`.

In order to use the function we can use the code that is shown in Figure 5.3 on page 95.

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.

dependent variable	explained variance
displacement	0.75
number of cylinders	0.70
horse power	0.73
weight	0.78
acceleration	0.21
year of build	0.31

Table 5.1: Explained variance for various dependent variables.

3. We open the file "`cars.csv`".
4. This file is processed as a Csv file where different columns are separated by the character "`,`".
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 kilometer 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 using the function `simple_linear_regression`.

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 5.1. 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 has a big weight, 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.

5.2 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 $i \in \{1, \cdots, m\}$. 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 $F\big(\mathbf{x}^{(i)}\big)$ approximates $y^{(i)}$ as precisely as posssible for all $i \in \{1, \cdots, m\}$, i.e. we want to have

$$F\big(\mathbf{x}^{(i)}\big) \approx y^{(i)} \quad \text{for all } i \in \{1, \cdots, m\}.$$

In order to make the notation $F\big(\mathbf{x}^{(i)}\big) \approx y^{(i)}$ more precise, we define the mean squared error

$$\texttt{MSE} := \frac{1}{m-1} \cdot \sum_{i=1}^{m} \Big(F\big(\mathbf{x}^{(i)}\big) - y^{(i)}\Big)^2. \tag{5.6}$$

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 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

$$F(\mathbf{x}) = \sum_{j=1}^{p} w_j \cdot x_j + b = \mathbf{x}^\top \cdot \mathbf{w} + b \quad \text{where } \mathbf{w} \in \mathbb{R}^p \text{ and } b \in \mathbb{R}.$$

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-m matrix, and the vector $\mathbf{w}$, where $\mathbf{w}$ is viewed as a m-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, $\mathbf{w}$ 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 $\mathbf{x}$ to an $(p+1)$-dimensional vector $\mathbf{x}'$ such that

$$x'_j := x_j \quad \text{for all } j \in \{1, \cdots, p\} \quad \text{and} \quad x'_{p+1} := 1.$$

To put it in words, the vector $\mathbf{x}'$ results from the vector $\mathbf{x}$ by appending the number 1:

$$\mathbf{x}' = \langle x_1, \cdots, x_p, 1 \rangle^\top \quad \text{where } \langle x_1, \cdots, x_p \rangle = \mathbf{x}^\top.$$

Furthermore, we define

$$\mathbf{w}' := \langle w_1, \cdots, w_p, b \rangle^\top \quad \text{where } \langle w_1, \cdots, w_p \rangle = \mathbf{w}^\top.$$

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 1. Using this assumption, we define the function F as

$$F(\mathbf{x}) := \mathbf{x}^\top \cdot \mathbf{w}.$$

Now equation (5.6) can be rewritten as follows:

$$\texttt{MSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \sum_{i=1}^{m} \Big((\mathbf{x}^{(i)})^\top \cdot \mathbf{w} - y^{(i)} \Big)^2. \tag{5.7}$$

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:

$$X := \begin{pmatrix} (\mathbf{x}^{(1)})^\top \\ \vdots \\ (\mathbf{x}^{(m)})^\top \end{pmatrix}$$

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 (5.7) as follows:

$$\texttt{MSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big)^\top \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big). \tag{5.8}$$

5.2.1 Some Useful Gradients

In the last section, we have computed the mean squared error $\texttt{MSE}(\mathbf{w})$ using equation (5.8). Our goal is to minimize the $\texttt{MSE}(\mathbf{w})$ by choosing the weight vector $\mathbf{w}$ appropriately. A necessary condition for $\texttt{MSE}(\mathbf{w})$ to be minimal is

$$\nabla \texttt{MSE}(\mathbf{w}) = \mathbf{0},$$

i.e. the gradient of $\texttt{MSE}(\mathbf{w})$ with respect to $\mathbf{w}$ needs to be zero. In order to prepare for the computation of $\nabla \texttt{MSE}(\mathbf{w})$, 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

$$f(\mathbf{x}) := \mathbf{x}^\top \cdot C \cdot \mathbf{x} \quad \text{where } C \in \mathbb{R}^{n \times n}.$$

If we write the matrix C as $C = (c_{i,j})_{\substack{i=1,\cdots,n \\ j=1,\cdots,n}}$ and the vector $\mathbf{x}$ as $\mathbf{x} = \langle x_1, \cdots, x_n \rangle^\top$, 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 $\dfrac{\partial f}{\partial x_k}$, is computed as follows:

$$\begin{aligned} \frac{\partial f}{\partial x_k} &= \sum_{i=1}^{n} \sum_{j=1}^{n} \Big(\frac{\partial x_i}{\partial x_k} \cdot c_{i,j} \cdot x_j + x_i \cdot c_{i,j} \cdot \frac{\partial x_j}{\partial x_k} \Big) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \Big(\delta_{i,k} \cdot c_{i,j} \cdot x_j + x_i \cdot c_{i,j} \cdot \delta_{j,k} \Big) \\ &= \sum_{j=1}^{n} c_{k,j} \cdot x_j + \sum_{i=1}^{n} x_i \cdot c_{i,k} \\ &= \big(C \cdot \mathbf{x}\big)_k + \big(C^\top \cdot \mathbf{x}\big)_k \end{aligned}$$

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

$$g(\mathbf{x}) := \mathbf{b}^\top \cdot A \cdot \mathbf{x}, \quad \text{where } \mathbf{b} \in \mathbb{R}^n \text{ and } A \in \mathbb{R}^{n \times n},$$

then a similar calculation shows that

$$\nabla g(\mathbf{x}) = A^\top \cdot \mathbf{b}.$$

Exercise 10: Prove this equation.

5.2.2 Deriving the Normal Equation

Next, we derive the so called normal equation for linear regression. To this end, we first expand the product in equation (5.8):

$$\begin{aligned} \texttt{MSE}(\mathbf{w}) &= \frac{1}{m-1} \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big)^\top \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big) \\ &= \frac{1}{m-1} \cdot \big(\mathbf{w}^\top \cdot X^\top - \mathbf{y}^\top\big) \cdot \big(X \cdot \mathbf{w} - \mathbf{y}\big) && \text{since } (A \cdot B)^\top = B^\top \cdot A^\top \\ &= \frac{1}{m-1} \cdot \big(\mathbf{w}^\top \cdot X^\top \cdot X \cdot \mathbf{w} - \mathbf{y}^\top \cdot X \cdot \mathbf{w} - \mathbf{w}^\top \cdot X^\top \cdot \mathbf{y} + \mathbf{y}^\top \cdot \mathbf{y}\big) \\ &= \frac{1}{m-1} \cdot \big(\mathbf{w}^\top \cdot X^\top \cdot X \cdot \mathbf{w} - 2 \cdot \mathbf{y}^\top \cdot X \cdot \mathbf{w} + \mathbf{y}^\top \cdot \mathbf{y}\big) && \text{since } \mathbf{w}^\top \cdot X^\top \cdot \mathbf{y} = \mathbf{y}^\top \cdot X \cdot \mathbf{w} \end{aligned}$$

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

$$\texttt{MSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \left(\mathbf{w}^\top \cdot \left(X^\top \cdot X\right) \cdot \mathbf{w} - 2 \cdot \mathbf{y}^\top \cdot X \cdot \mathbf{w} + \mathbf{y}^\top \cdot \mathbf{y}\right) \tag{5.9}$$

holds. The matrix $X^\top \cdot X$ used in the first term is symmetric because

$$\left(X^\top \cdot X\right)^\top = X^\top \cdot \left(X^\top\right)^\top = X^\top \cdot X.$$

Using the results from the previous section we can now compute the gradient of $\texttt{MSE}(\mathbf{w})$ with respect to $\mathbf{w}$. The result is

$$\nabla\texttt{MSE}(\mathbf{w}) = \frac{2}{m-1} \cdot \left(X^\top \cdot X \cdot \mathbf{w} - X^\top \cdot \mathbf{y}\right).$$

If the squared error $\texttt{MSE}(\mathbf{w})$ has a minimum for the weights $\mathbf{w}$, then we must have

$$\nabla\texttt{MSE}(\mathbf{w}) = \mathbf{0}.$$

This leads to the equation

$$\frac{2}{m-1} \cdot \left(X^\top \cdot X \cdot \mathbf{w} - X^\top \cdot \mathbf{y}\right) = \mathbf{0}.$$

This equation can be rewritten as

$$\boxed{\left(X^\top \cdot X\right) \cdot \mathbf{w} = X^\top \cdot \mathbf{y}.} \tag{5.10}$$

This equation is called 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

$$\mathbf{w} = \left(X^\top \cdot X\right)^{-1} \cdot X^\top \cdot \mathbf{y}.$$

Instead, when solving the normal equation we will use the *Python* function $\texttt{numpy.linalg.solve}(A, b)$, which takes a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ and solves the equation

$$A \cdot \mathbf{x} = \mathbf{b}.$$ ◇

5.2.3 Implementation

Figure 5.4 on page 102 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.

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 $X^\top \cdot \mathbf{y}$.
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 $\mathbf{y}$.

```
import csv
import numpy as np

def linear_regression(fileName, target, explaining, f):
    with open(fileName) as input_file:
        reader     = csv.reader(input_file, delimiter=',')
        line_count = 0
        goal       = []
        Causes     = []
        for row in reader:
            if line_count != 0:
                goal  .append(f(float(row[target])))
                Causes.append([float(row[i]) for i in explaining] + [1.0])
            line_count += 1
    m = len(goal)
    X = np.array(Causes)
    y = np.array(goal)
    w = np.linalg.solve(X.T @ X, X.T @ y)
    RSS   = np.sum((X @ w - y) ** 2)
    yMean = np.sum(y) / m
    TSS   = sum((y - yMean) ** 2)
    R2    = 1 - RSS / TSS
    return R2

def main():
    explaining = [1, 2, 3, 4, 5, 6]
    R2 = linear_regression("cars.csv", 0, explaining, lambda x: 1/x)
    print(f'portion of explained variance : {R2}')
```

Figure 5.4: General linear regression.

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 5.4 on page 102 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.

5.3 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

$$x_1^2,\ x_2^2, \text{ and } x_1 \cdot x_2.$$

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 on 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 5.5 on page 104 shows a *Python* script that can be used to investigate this situation.

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

   ```
   0.2 * (np.random.rand() - 0.5)
   ```

   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 5.6 on page 105 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 5.7 on page 106 shows the linear model that is computed by this function.

[1] We add this second variable in order be able to show the effect of overfitting.

```
1  import numpy                as np
2  import sklearn.linear_model as lm
3
4  N  = 20                     # number of data points
5  X1 = np.array([k for k in range(N)])
6  X2 = np.array([k + 0.2 * (np.random.rand() - 0.5) for k in range(N)])
7  Y  = np.sqrt(X1)            # Y is the square root of X1
8  X1 = np.reshape(X1, (N, 1)) # turn X1 into an N x 1 matrix
9  X2 = np.reshape(X2, (N, 1)) # turn X2 into an N x 1 matrix
10 X  = np.hstack([X1, X2])    # combine X1 and X2 into an N x 2 matrix
11
12 from sklearn.model_selection import train_test_split
13
14 X_train, X_test, Y_train, Y_test = \
15     train_test_split(X, Y, test_size=0.2, random_state=1)
16
17 def linear_regression(X_train, Y_train, X_test, Y_test):
18     M = lm.LinearRegression()
19     M.fit(X_train, Y_train)
20     train_score = M.score(X_train, Y_train)
21     test_score  = M.score(X_test , Y_test)
22     return M, train_score, test_score
23
24 M, train_score, test_score = linear_regression(X_train, Y_train, X_test, Y_test)
25
26 from sklearn.preprocessing import PolynomialFeatures
27
28 quadratic = PolynomialFeatures(2, include_bias=False)
29 X_train_quadratic = quadratic.fit_transform(X_train)
30 X_test_quadratic  = quadratic.fit_transform(X_test)
31 quadratic.get_feature_names(['x1', 'x2'])
32
33 M, s1, s2 = \
34     linear_regression(X_train_quadratic, Y_train, X_test_quadratic, Y_test)
```

Figure 5.5: Polynomial Regression

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 5.8 on page 107 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%.

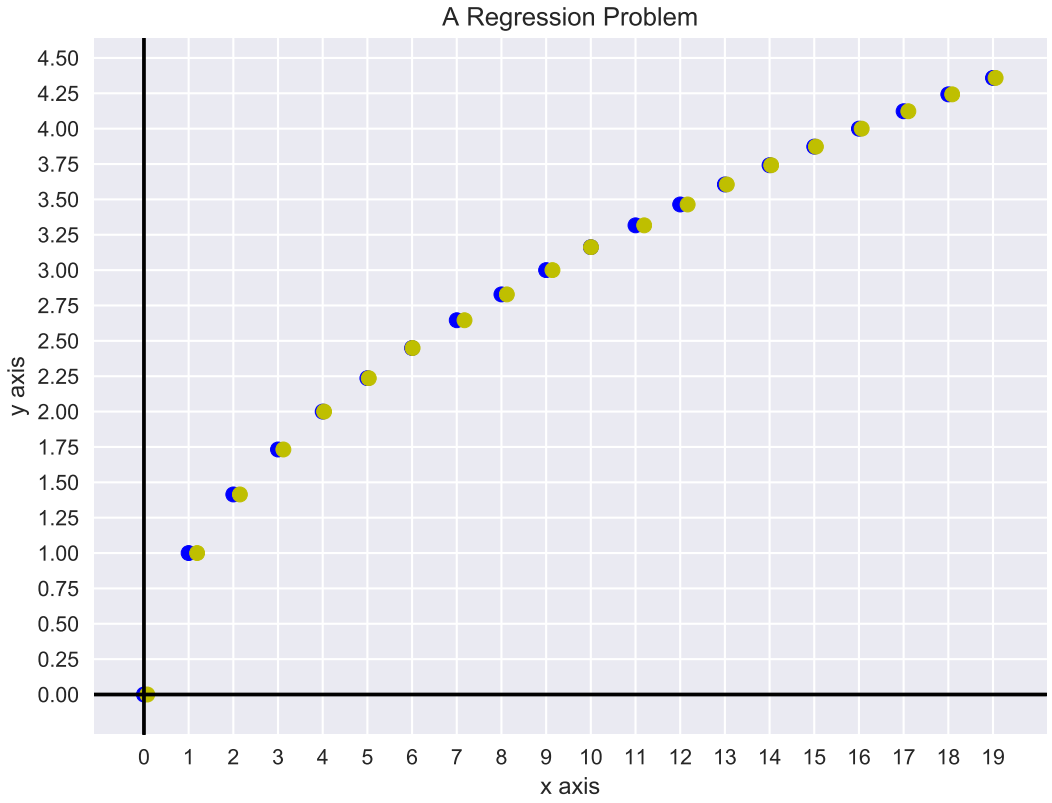

Figure 5.6: $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

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 5.9 on page 108 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 5.10 on page 109 shows the corresponding model. We see that regression curve gets wiggly, which is also an indication that we are experiencing overfitting.

5.4 Ridge Regression

In linear regression, our goal is to minimize the mean squared error that has been defined as

$$\texttt{MSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \sum_{i=1}^{m} \left(\left(\mathbf{x}^{(i)}\right)^\top \cdot \mathbf{w} - y^{(i)} \right)^2.$$

The previous example has shown that sometimes minimizing the mean squared error leads to overfit-

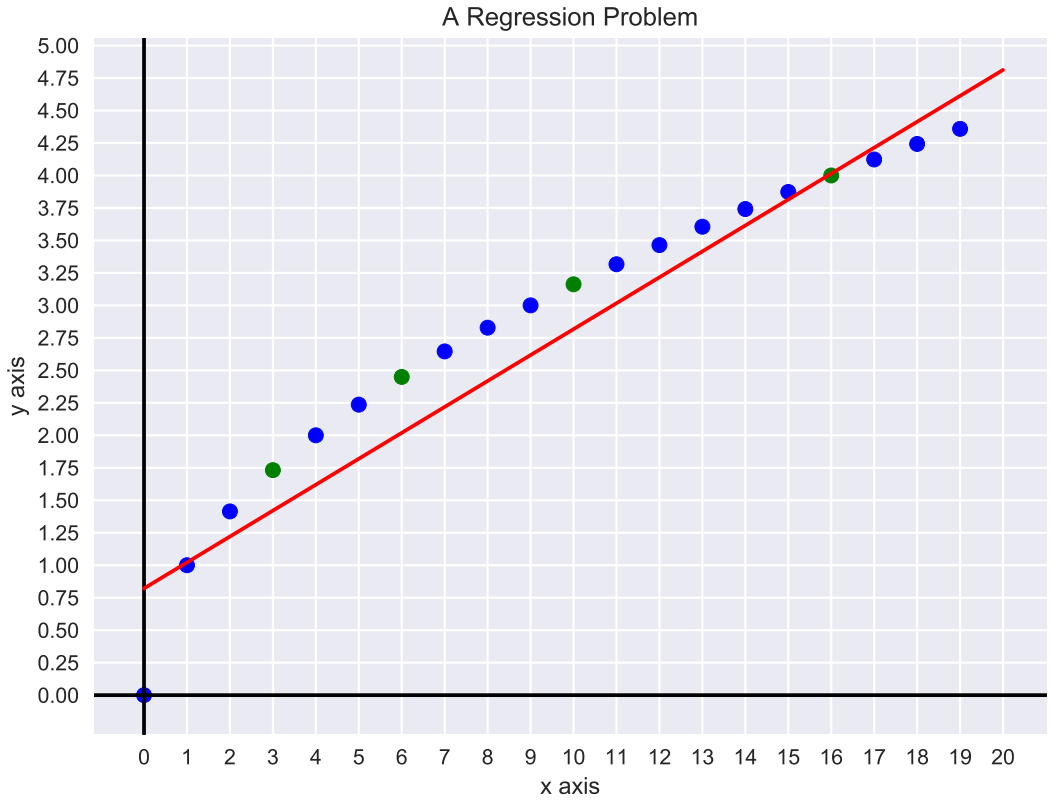

Figure 5.7: Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

ting. One way to avoid overfitting is called ridge-regression. Instead of minimizing the mean squared error, we instead minimize the expression

$$\texttt{RegMSE}(\mathbf{w}) = \frac{1}{m-1} \cdot \sum_{i=1}^{m} \left(\left(\mathbf{x}^{(i)}\right)^\top \cdot \mathbf{w} - y^{(i)} \right)^2 + \lambda \cdot \mathbf{w}^2.$$

Here, the parameter λ is called the regularization parameter. If λ is not too small, then minimizing $\texttt{RegMSE}(\mathbf{w})$ forces us to find parameters $\mathbf{w}$ that are small. This prevents the model from getting to complex and this prevents over-fitting. Figure 5.11 on page 110 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 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

Figure 5.8: Second Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

untouched while searching for the best value of λ.

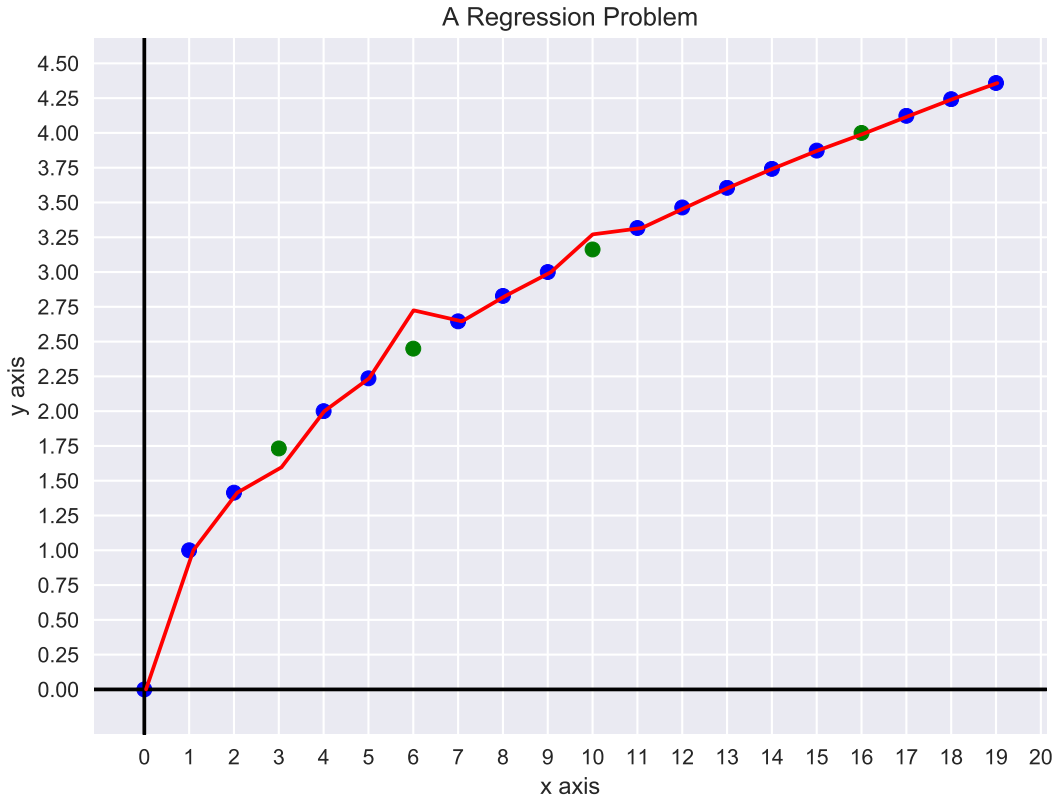

Figure 5.9: Fourth Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

Exercise 11: 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 inches.

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. ◇

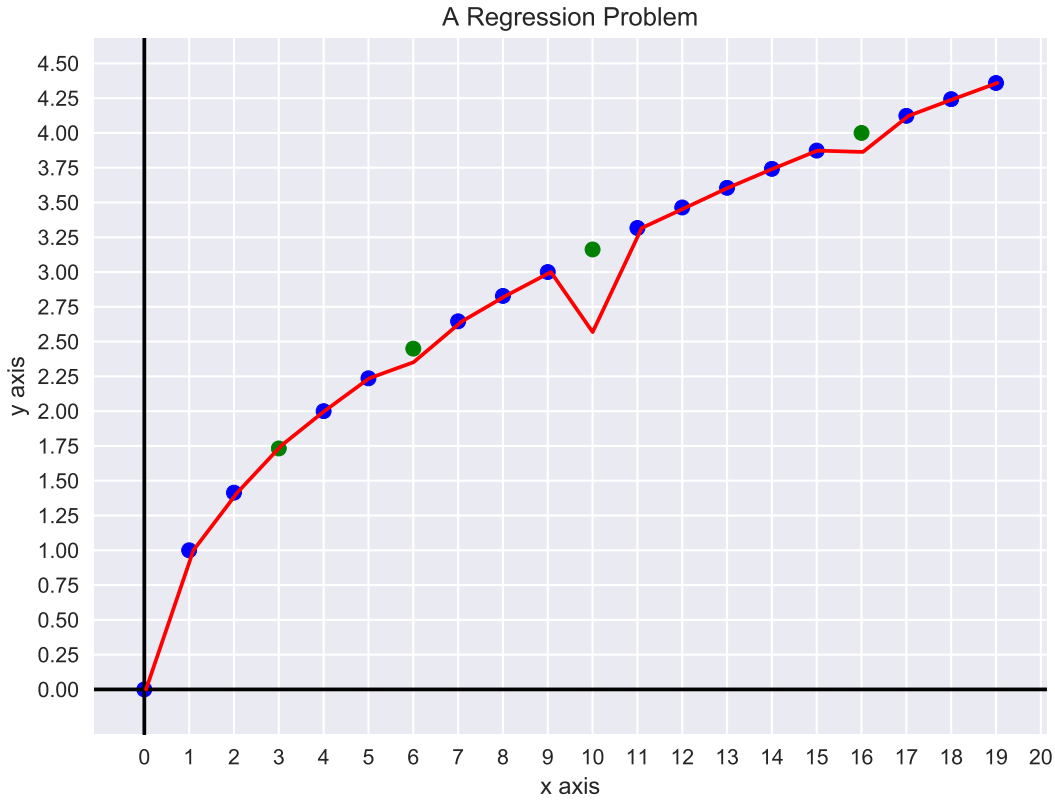

Figure 5.10: Sixth Order Linear Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

Exercise 12: 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? ◇

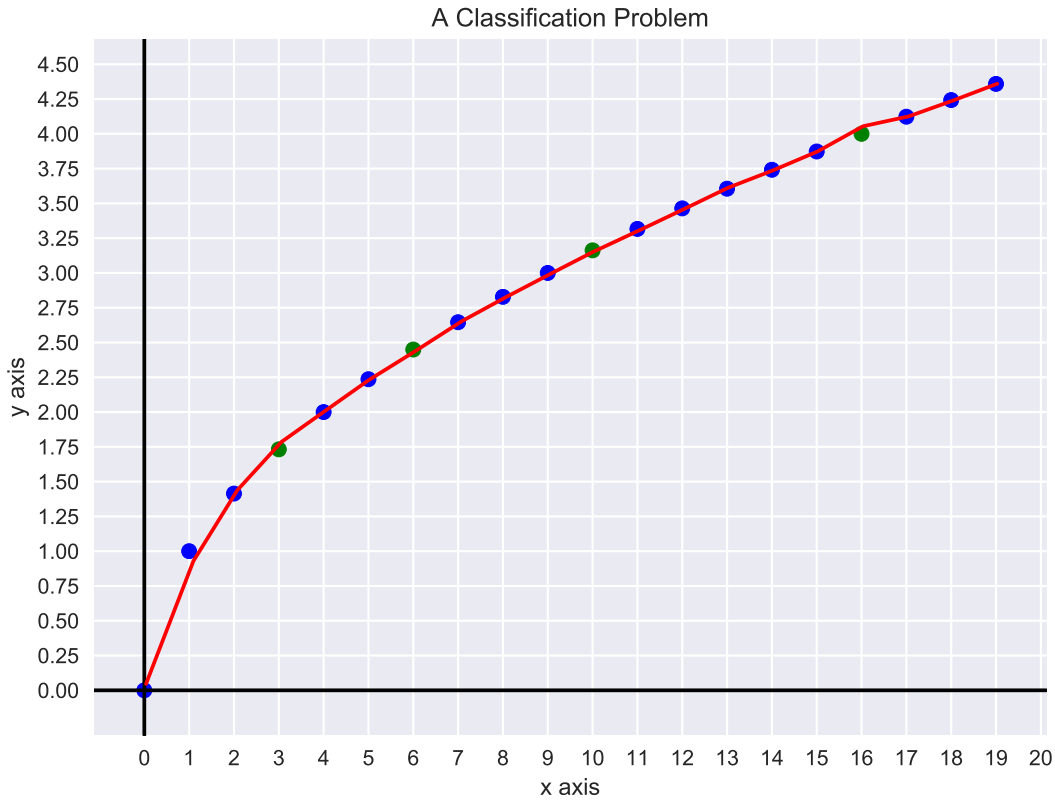

Figure 5.11: Sixth Order Ridge Regression for $y = \sqrt{x_1}$, $x_2 = x_1 + \texttt{noise}$.

Chapter 6

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.

6.1 Introduction

Formally, the classification problem in machine learning can be stated as follows: We are given a set of objects $S := \{o_1, \cdots, o_n\}$ and a set of classes $C := \{c_1, \cdots, c_k\}$. Furthermore, there exists a function

$$\texttt{classify} : S \to C$$

that assigns a class $\texttt{classify}(o)$ 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

$$C = \{\texttt{spam}, \texttt{ham}\}.$$

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

$$\texttt{feature} : S \times \{1, \cdots, D\} \to \mathbb{R}$$

such that $\texttt{feature}(o, j)$ 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

$$\texttt{feature}(o, j) \in \mathbb{B} \quad \text{for all } o \in S,$$

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

$$\texttt{feature}(o,1) := \begin{cases} +1 & \text{if } \texttt{pharmacy} \in o, \\ -1 & \text{if } \texttt{pharmacy} \notin o, \end{cases}$$

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

$$\texttt{feature}(o,1) := \texttt{count}(\texttt{"pharmacy"}, o),$$

i.e. we would count the number of occurrences of the string "pharmacy" in our email o. As the value of

$$\texttt{count}(\texttt{"pharmacy"}, o)$$

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

$$\texttt{feature}(o,1) := \frac{\texttt{count}(\texttt{"pharmacy"}, o)}{\#o},$$

where $\#o$ 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

$$\texttt{model} : \mathbb{R}^D \to C$$

such that

$$\texttt{model}\big(\texttt{feature}(o,1), \cdots, \texttt{feature}(o,D)\big) \approx \texttt{classify}(o).$$

Using the function `model`, we can then approximate the function classify using a function `guess` that is defined as

$$\texttt{guess}(o) := \texttt{model}\big(\texttt{feature}(o,1), \cdots, \texttt{feature}(o,D)\big)$$

Most of the time, the function `guess` will only approximate the function `classify`, i.e. we will have

$$\texttt{guess}(o) = \texttt{classify}(o)$$

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.

$$\texttt{accuracy} := \frac{\#\{o \in S \mid \texttt{guess}(o) = \texttt{classify}(o)\}}{\#S}.$$

If the set S is infinite, this equation has to be interpreted as a limit, i.e. we can define S_n as the set of strings that have a length of at most n. Then, the accuracy can be defined as a limit:

$$\texttt{accuracy} := \lim_{n\to\infty} \frac{\#\{o \in S_n \mid \texttt{guess}(o) = \texttt{classify}(o)\}}{\#S_n}.$$

The function `model` is usually determined by a set of parameters or weights $\mathbf{w}$. In this case, we have

$$\texttt{model}(\mathbf{x}) = \texttt{model}(\mathbf{x}, \mathbf{w})$$

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, all models are wrong. Nevertheless, some 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. 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

$$\texttt{classify} : S \to C$$

using certain features of our objects. The function `classify` is then approximated using a function `model` as follows:

$$\texttt{model}\big(\texttt{feature}(o,1), \cdots, \texttt{feature}(o,D), \mathbf{w}\big) \approx \texttt{classify}(o).$$

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.

6.1.1 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. $\mathbf{x}_i := \langle \texttt{feature}(o_i, 1), \cdots, \texttt{feature}(o_i, D) \rangle \quad$ for all $i \in \{1, \cdots, N\}$.
   i.e. $\mathbf{x}_i$ is a D-dimensional row vector that collects the features of the i-th training object.
2. $x_{i,j} := \texttt{feature}(o_i, j) \quad$ for all $i \in \{1, \cdots, N\}$ and $j \in \{1, \cdots, D\}$.
   i.e. $x_{i,j}$ is the j-th feature of the i-th object.
3. $y_i := \texttt{classify}(o_i) \quad$ for all $i \in \{1, \cdots, N\}$
   i.e. y_i is the class of the i-th object.

Mathematically, our goal is to maximize the accuracy of our model as a function of the parameters $\mathbf{w}$.

6.1.2 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

$$\{\texttt{commonCold}, \texttt{pneumonia}, \texttt{asthma}, \texttt{flu}, \cdots, \texttt{unknown}\}.$$

Hence, the task of disease diagnosis is a classification problem. This was one of the earliest problem 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:

1. image recognition,
2. speech recognition,
3. credit card fraud detection,
4. credit approval.

6.2 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

$$\forall \mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) \leq f\Big(\arg\max_{\mathbf{x} \in \mathbb{R}^n} f\Big).$$

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 $\widehat{\mathbf{x}} \in \mathbb{R}^n$ to satisfy

$$\widehat{\mathbf{x}} = \arg\max_{\mathbf{x} \in \mathbb{R}^n} f \quad \text{is that we must have} \quad \nabla f(\widehat{\mathbf{x}}) = \mathbf{0},$$

i.e. the gradient of f, which we will write as ∇f, vanishes at the maximum. Remember that the gradient of f is defined as the vector

$$\nabla f := \begin{pmatrix} \dfrac{\partial f}{\partial x_1} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{pmatrix}.$$

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 6.1 on page 116. 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 $\lim\limits_{n \to \infty} x_n$ exists and is a solution of the equation $\cos(x) - x = 0$, i.e. we want to have

$$\cos\Big(\lim_{n \to \infty} x_n\Big) = \lim_{n \to \infty} x_n.$$

The method of gradient ascent is a numerical method that can be used to find the maximum of a function

$$f : \mathbb{R}^n \to \mathbb{R}$$

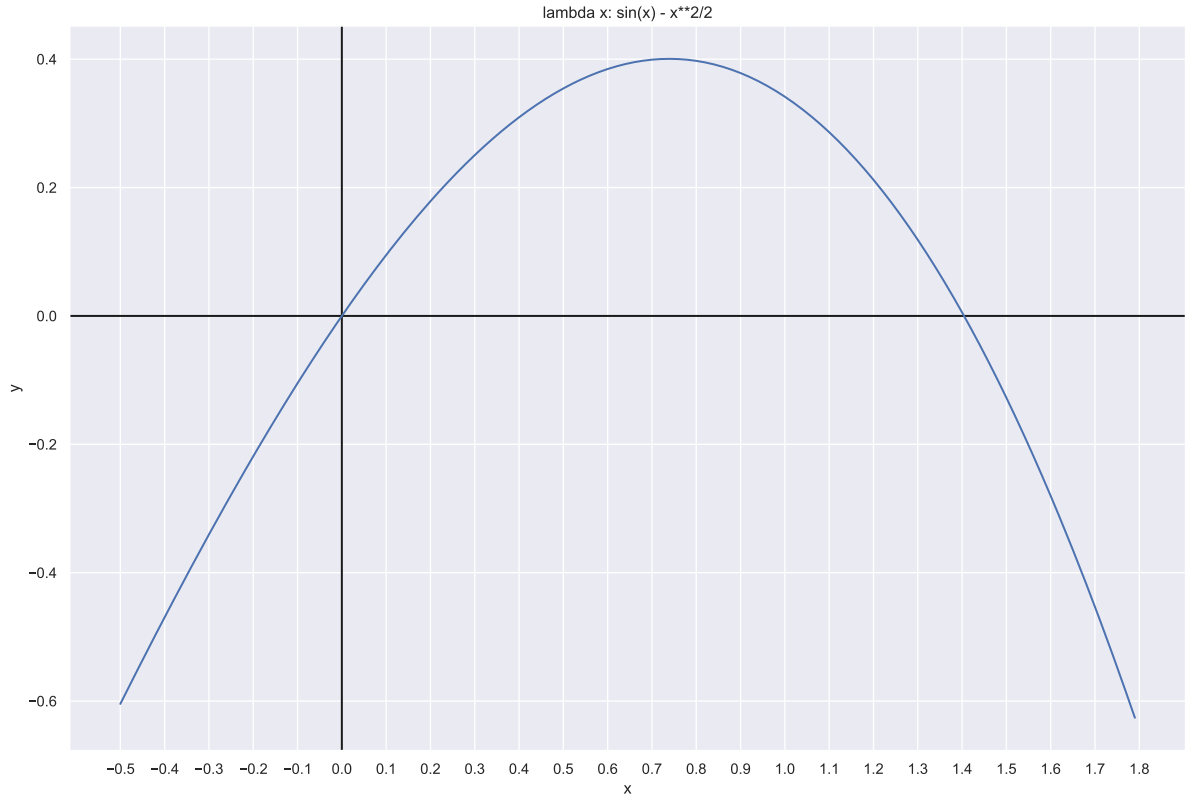

Figure 6.1: The function $x \mapsto \sin(x) - \frac{1}{2} \cdot x^2$.

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 $\big(\mathbf{x}_n\big)_{n \in \mathbb{N}}$ such that we have

$$f(\mathbf{x}_{n+1}) \geq f(\mathbf{x}_n) \quad \text{for all } n \in \mathbb{N}.$$

Hopefully, this sequence will converge against $\widehat{\mathbf{x}} = \arg\max_{\mathbf{x} \in \mathbb{R}^n} f$. 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:

$$\mathbf{x}_{n+1} := \mathbf{x}_n + \alpha \cdot \nabla f(\mathbf{x}_n) \quad \text{for all } n \in \mathbb{N}_0.$$

Here, α is called the step size 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 6.2 on page 118 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 `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 `x` $\in \mathbb{R}^n$ as its input and returns the vector ∇`f`(`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 that the function f takes for the argument `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 `xOld` and `fOld`. This is needed since we need to ensure that the values of $f(\mathbf{x}_n)$ are increasing.
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 $\big(f(\mathbf{x}_n)\big)_{n\in\mathbb{N}}$ will be monotonically increasing, i.e. we have
$$f(\mathbf{x}_{n+1}) \geq f(\mathbf{x}_n) \quad \text{for all } n \in \mathbb{N},$$
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 `x` and the corresponding function value f(`x`).

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

```
def findMaximum(f, gradF, start, eps):
    x     = start
    fx    = f(x)
    alpha = 1.0
    cnt   = 1  # number of iterations
    while True:
        xOld, fOld = x, fx
        x  += alpha * gradF(x)
        fx  = f(x)
        if fx <= fOld:
            alpha *= 0.5
            x, fx = xOld, fOld
            continue
        else:
            alpha *= 1.2
        if abs(fx - fOld) <= abs(fx) * eps:
            return x, fx
        cnt += 1
```

Figure 6.2: The gradient ascent 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.

6.3 Logistic Regression

If we have a model such that

$$\texttt{model}(\mathbf{x}, \mathbf{w}) \approx \texttt{classify}(\mathbf{x})$$

we want to choose the weight vector $\mathbf{w}$ in a way such that the accuracy

$$\texttt{accuracy}(\mathbf{w}) := \frac{\#\big\{\mathbf{o} \in S \;\big|\; \texttt{model}\big(\texttt{feature}(\mathbf{o}), \mathbf{w}\big) = \texttt{classify}(\mathbf{o})\big\}}{\#S}$$

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.

6.3.1 The Sigmoid Function

Definition 6 (Sigmoid Function) The sigmoid function $S : \mathbb{R} \to [0, 1]$ is defined as

$$S(t) = \frac{1}{1 + \exp(-t)}.$$

Figure 6.3 on page 119 shows the sigmoid function. The sigmoid function is also known as the logistic function. ◇

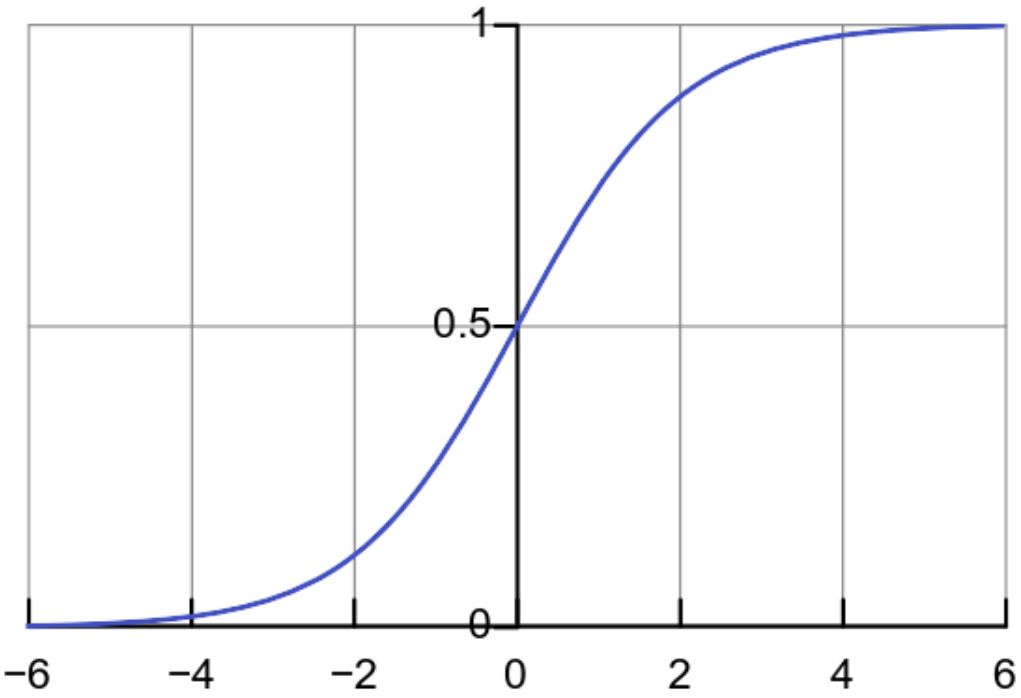

Figure 6.3: The sigmoid function.

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

$$0 \leq S(t) \leq 1 \quad \text{for all } t \in \mathbb{R}.$$

Therefore, the value of the sigmoid function can be interpreted as a probability. Another important property of the sigmoid function is its symmetry. Figure 6.3 shows that if the sigmoid function is

shifted down by $\frac{1}{2}$, the resulting function is centrally symmetric, i.e. we have

$$S(-t) - \frac{1}{2} = -\Big(S(t) - \frac{1}{2}\Big).$$

Adding $\frac{1}{2}$ on both sides of this equation shows that this is equivalent to the equation

$$\boxed{S(-t) = 1 - S(t),}$$

The proof of this fact runs as follows:

$$\begin{aligned}
1 - S(t) &= 1 - \frac{1}{1+\exp(-t)} && \text{(by definition of } S(t)\text{)} \\
&= \frac{1+\exp(-t)-1}{1+\exp(-t)} && \text{(common denominator)} \\
&= \frac{\exp(-t)}{1+\exp(-t)} \\
&= \frac{1}{\exp(+t)+1} && \text{(expand fraction by } \exp(t)\text{)} \\
&= \frac{1}{1+\exp(+t)} \\
&= S(-t) && \text{(by definition of } S(-t)\text{).} \quad \square
\end{aligned}$$

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

$$S(t) \cdot \big(1+\exp(-t)\big) = 1.$$

Dividing both sides by $S(t)$ gives:

$$\begin{aligned}
& 1+\exp(-t) = \frac{1}{S(t)} \\
\Leftrightarrow \quad & \exp(-t) = \frac{1}{S(t)} - 1 \\
\Leftrightarrow \quad & \exp(-t) = \frac{1-S(t)}{S(t)}
\end{aligned}$$

We highlight this formula, as we need it later

$$\boxed{\exp(-t) = \frac{1-S(t)}{S(t)}.} \tag{6.1}$$

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

$$\boxed{S^{-1}(y) = \ln\left(\frac{y}{1-y}\right).}$$

This function is known as the logit function. Next, let us compute the derivative of $S(t)$, i.e. $S'(t) = \frac{\mathrm{d}S}{\mathrm{d}t}$. We have

$$\begin{aligned} S'(t) &= -\frac{-\exp(-t)}{\big(1+\exp(-t)\big)^2} \\ &= \exp(-t) \cdot S(t)^2 \\ &= \frac{1 - S(t)}{S(t)} \cdot S(t)^2 \qquad \text{(by Equation 6.1)} \\ &= \big(1 - S(t)\big) \cdot S(t) \end{aligned}$$

We have shown

$$\boxed{S'(t) = \big(1 - S(t)\big) \cdot S(t).} \tag{6.2}$$

We will later need the derivative of the natural logarithm of the logistic function. We define

$$L(t) := \ln\big(S(t)\big).$$

Then we have

$$\begin{aligned} L'(t) &= \frac{S'(t)}{S(t)} \qquad \text{(by the chain rule)} \\ &= \frac{(1 - S(t)) \cdot S(t)}{S(t)} \\ &= 1 - S(t) \\ &= S(-t) \qquad \text{(by symmetry)} \end{aligned}$$

As this is our most important result, we highlight it:

$$\boxed{L'(t) = S(-t) \quad \text{where} \quad L(t) := \ln\big(S(t)\big).}$$

6.3.2 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 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 fact is that 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

$$\boxed{P(y \mid \mathbf{x}, \mathbf{w}) = S\big(y \cdot (\mathbf{x} \cdot \mathbf{w})\big).}$$

Given N objects $o_1, \cdots, o_n$ with feature vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n$ and classes $y_1, \cdots, y_n$, 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. 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:

$$\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \prod_{i=1}^{N} P(y_i \mid \mathbf{x}_i, \mathbf{w}).$$

Here, we have combined the different attribute vectors $\mathbf{x}_i$ into the matrix $\mathbf{X}$ such that $\mathbf{x}_i$ is the i-th row of the matrix $\mathbf{X}$. 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

$$\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) := \ln\big(\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})\big).$$

As the natural logarithm is a monotonically increasing function, the functions $\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ and $\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w})$ 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

$$\boxed{\ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = \sum_{i=1}^{N} \ln\Big(S\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big)\Big) = \sum_{i=1}^{N} L\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big).}$$

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 those values of the coefficients $\mathbf{w}$ that satisfy

$$\frac{\partial}{\partial w_j} \ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) = 0.$$

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_j. 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 $\mathbf{w}$:

$$\begin{aligned}
&\frac{\partial}{\partial\, w_j} \ell\ell(\mathbf{X}, \mathbf{y}, \mathbf{w}) \\
= \;& \frac{\partial}{\partial\, w_j} \sum_{i=1}^{N} L\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big) \\
= \;& \sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S\big(-y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big), \quad \text{since} \quad \frac{\mathrm{d}L(x)}{\mathrm{d}x} = S(-x).
\end{aligned}$$

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

$$\sum_{i=1}^{N} y_i \cdot x_{i,j} \cdot S(-y_i \cdot \mathbf{x}_i \cdot \mathbf{w}) = 0 \quad \text{for all } j \in \{1, \cdots, D\}.$$

These are D equations for the D variables $w_1, \cdots, w_D$. 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 for in vain: We will use the method of gradient ascent in order to find the value of $\mathbf{w}$ that maximizes the log-likelihood. This method has been outlined in the previous section.

6.3.3 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 6.4 shows a Csv file 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 6.5 on page 125 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.

```
1  Pass, Hours
2  0,    0.50
3  0,    0.75
4  0,    1.00
5  0,    1.25
6  0,    1.50
7  0,    1.75
8  1,    1.75
9  0,    2.00
10 1,    2.25
11 0,    2.50
12 1,    2.75
13 0,    3.00
14 1,    3.25
15 0,    3.50
16 1,    4.00
17 1,    4.25
18 1,    4.50
19 1,    4.75
20 1,    5.00
21 1,    5.50
```

Figure 6.4: Results of an exam.

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 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

$$L(x) = \ln\big(S(X)\big) = \ln\left(\frac{1}{1 + \exp(-x)}\right) = -\ln\big(1 + \exp(-x)\big).$$

   The implementation is more complicated than you might expect. The reason has to do with 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 `Infinity`. However, in reality we have

$$\ln\big(1 + \exp(1000)\big) \approx 1000$$

```python
import numpy as np

# compute 1/(1+exp(-t))
def sigmoid(t):
    return 1.0 / (1.0 + np.exp(-t))

# compute ln(1/(1+exp(-t))) and avoid overflow
def logSigmoid(t):
    if t > -100:
        return -np.log(1.0 + np.exp(-t))
    else:
        return t

def ll(X, y, w):
    """
    given the matrix X and the observations y,
    return the log likelihood for the weight vector w
    """
    return np.sum([logSigmoid(y[i] * (X[i] @ w)) for i in range(len(X))])

def gradLL(X, y, w):
    """
    Compute the gradient of the log-likelihood with respect to w
    """
    Gradient = []
    for j in range(len(X[1])):
        L = [y[i]*X[i][j]*sigmoid(-y[i] * (X[i] @ w)) for i in range(len(X))]
        Gradient.append(sum(L))
    return np.array(Gradient)
```

Figure 6.5: An implementation of logistic regression.

because exp(1000) is so big that adding 1 to it does not make much of a difference. The precise argument works as follows:

$$\begin{aligned}
\ln\big(1+\exp(x)\big) &= \ln\big(\exp(x)\cdot(1+\exp(-x))\big) \\
&= \ln\big(\exp(x)\big) + \ln\big(1+\exp(-x)\big) \\
&= x + \ln\big(1+\exp(-x)\big) \\
&\approx x + \ln(1) + \exp(-x) && \text{Taylor expansion of } \ln(1+x) \\
&= x + 0 + \exp(-x) \\
&\approx x && \text{since } \exp(-x) \approx 0 \text{ for large } x
\end{aligned}$$

This is the reason that `logSigmoid` returns `x` if the value of `x` is less than -100.

4. The function `ll`$(\mathbf{X}, \mathbf{y}, \mathbf{w})$ defined in line 14 computes the log-lokelihood of the parameter $\mathbf{w}$ given the available data $\mathbf{X}$ and $\mathbf{y}$. We have

$$\ell\ell(\mathbf{X}, \mathbf{y}; \mathbf{w}) = \sum_{i=1}^{N} L\big(y_i \cdot (\mathbf{x}_i \cdot \mathbf{w})\big).$$

   Here L denotes the natural logarithm of the sigmoid of the argument. It is assumed that $\mathbf{X}$ is a 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`$(\mathbf{x}, \mathbf{y}, \mathbf{w})$ in line 21 computes the gradient of the log-lokelihood 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-lokelihood.

6. Finally, the function `logisticRegressionFile` that is shown in Figure 6.6 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 $\mathbf{X}$ and the vector $\mathbf{y}$, and then to use the method of gradient ascent to find the weight vector $\mathbf{w}$ 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 spend 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 `x` into a matrix `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.

   (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 $\mathbf{y}$ 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-lokelihood of a given weight vector `w`. Hence we define the function `f`$(\mathtt{w})$ as `ll`$(\mathtt{X}, \mathtt{y}, \mathtt{w})$ 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-lokelihood.

```
import csv
import gradient_ascent

def logisticRegression(name):
    with open(name) as file:
        reader = csv.reader(file, delimiter=',')
        count  = 0  # line count
        Pass   = []
        Hours  = []
        for row in reader:
            if count != 0:  # skip header
                Pass .append(float(row[0]))
                Hours.append(float(row[1]))
            count += 1
    y = np.array(Pass)
    x = np.array(Hours)
    n = len(y)
    X = np.reshape(x, (n,1))
    X = np.append(np.ones((n, 1)), X, axis=-1)
    y = 2 * y - 1
    start   = np.zeros((2,))
    eps     = 10 ** -8
    f       = lambda w: ll(X, y, w)
    gradF   = lambda w: gradLL(X, y, w)
    w, _, _ = gradient_ascent.findMaximum(f, gradF, start, eps)
    return w
```

Figure 6.6: The function `logisticRegression`.

If we run the function `logisticRegressionFile` using the data shown in Figure 6.4 the resulting values of the weight vector `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 6.7 shows a plot of this probability $P(x)$.

6.3.4 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 6.8 on page 129. This Csv file contains data about a fictional exam. The first column indicates whether the student has

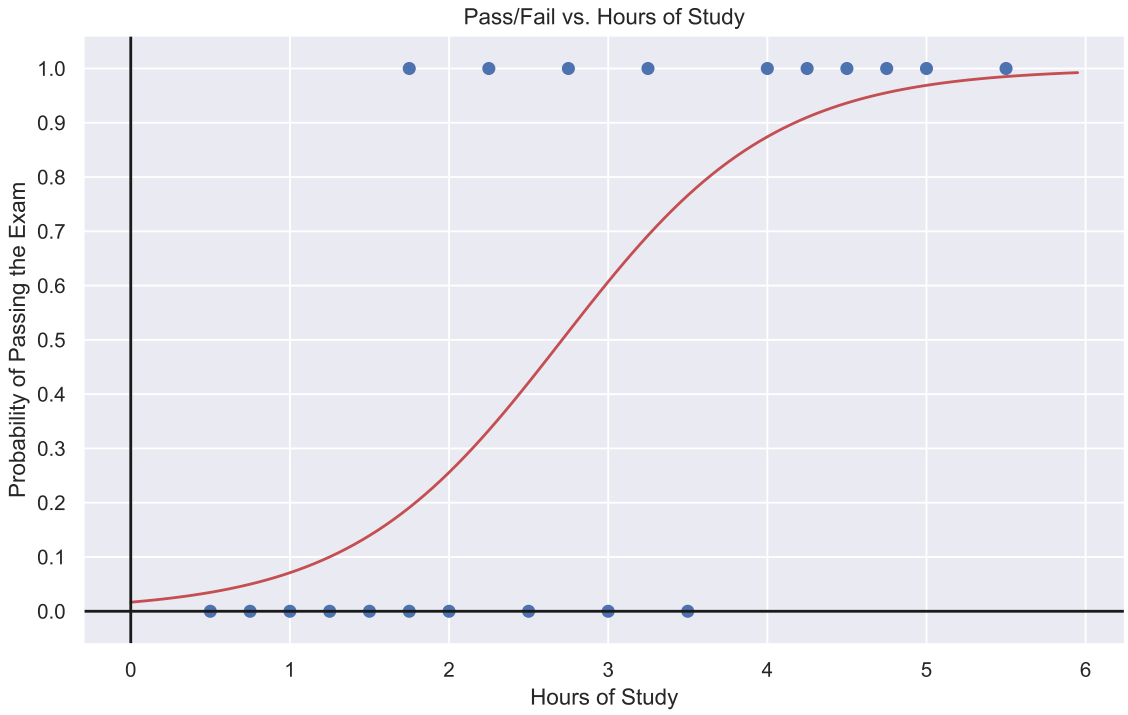

Figure 6.7: Probability of passing an exam versus hours of studying.

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 6.9 on page 130. 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 6.9 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.

Figure 6.10 on page 130 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

```
Pass,Hours,IQ
0,0.50,110
0,0.75,95
0,1.00,118
0,1.25,97
0,1.50,100
0,1.75,110
0,1.75,115
1,2.00,104
1,2.25,120
0,2.50,98
1,2.75,118
0,3.00,88
1,3.25,108
0,3.50,125
1,4.00,109
1,4.25,110
1,4.50,112
1,4.75,97
1,5.00,102
1,5.50,109
```

Figure 6.8: Results of an exam given hours of study and IQ.

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

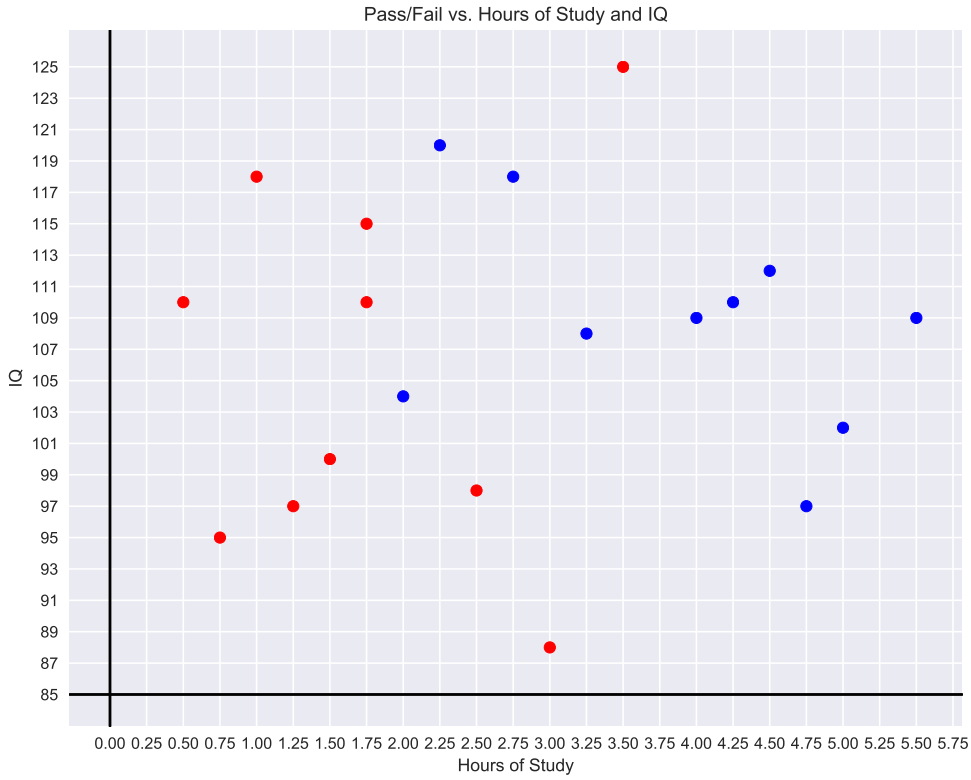

Figure 6.9: Probability of passing an exam versus hours of studying.

```
1  import numpy                as np
2  import pandas               as pd
3  import sklearn.linear_model as lm
4
5  ExamDF = pd.read_csv('exam-iq.csv')
6  X      = np.array(ExamDF[['Hours','IQ']])
7  Y      = np.array(ExamDF['Pass'], dtype=float)
8  model  = lm.LogisticRegression(C=10000, tol=1e-6, solver='newton-cg')
9  M      = model.fit(X, Y)
10 ϑ0     = M.intercept_[0]
11 ϑ1, ϑ2 = M.coef_[0]
12 errors = np.sum(np.abs(Y - model.predict(X)))
13 print((len(Y) - errors) / len(Y))
```

Figure 6.10: Logistic Regression using SciKit-Learn

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

$$\vartheta_0 + \vartheta_1 \cdot h + \vartheta_2 \cdot q \geq 0.$$

This can be rewritten as follows:

$$q \geq -\frac{\vartheta_0 + \vartheta_1 \cdot h}{\vartheta_2}.$$

The decision boundary are those values of (h, q) such that $P(h) = \frac{1}{2}$. 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 6.11 on page 131. Everybody who is located to the right this line is predicted to pass the exam, while everybody who is located to the left is predicted to fail.

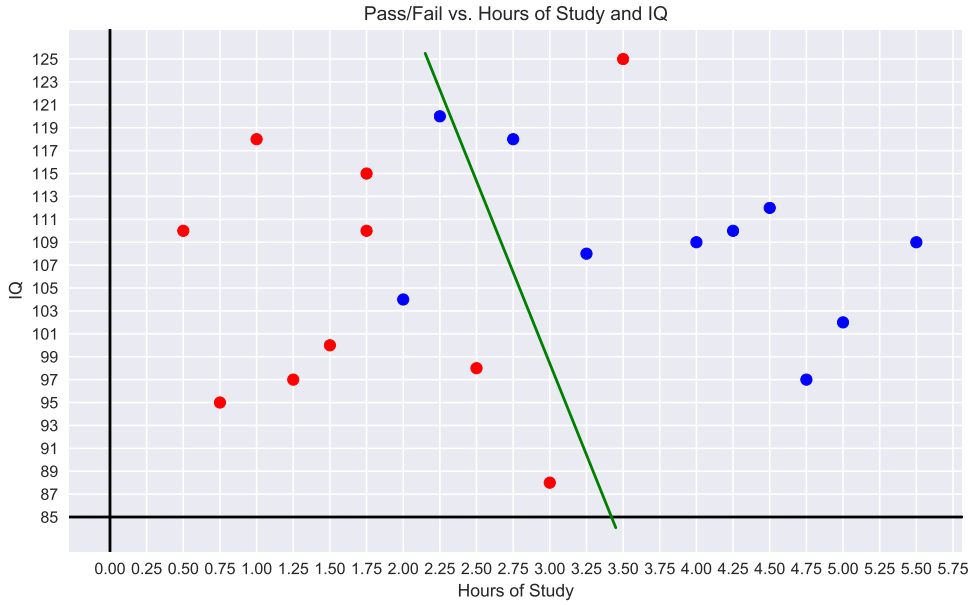

Figure 6.11: Probability of passing an exam versus hours of studying.

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 6.11 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 13: 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 virginica[1], 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.
- 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%. ◇

6.4 Polynomial Logistic Regression

Sometimes logistic regression does not work because the data is not linearly separable. For example, Figure 6.12 on page 133 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.

If we try to separate the data in Figure 6.12 by logistic regression, we get the result shown in Figure 6.13 on page 134. 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.

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 6.14 on page 135 shows a *Python* script that reads the fake data shown in Figure 6.12, adds second order polynomial features to this data and then performs linear regression.

[1] This plant is native to North America and hence has no German name.

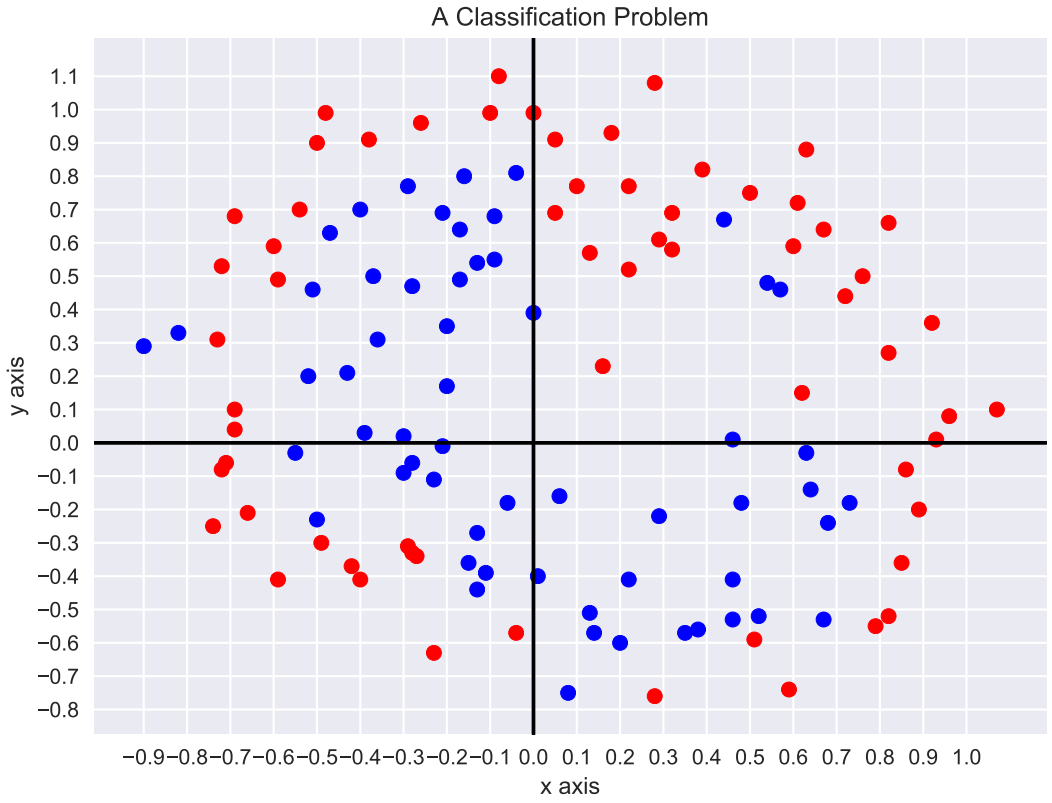

Figure 6.12: Some fake data.

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.
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.

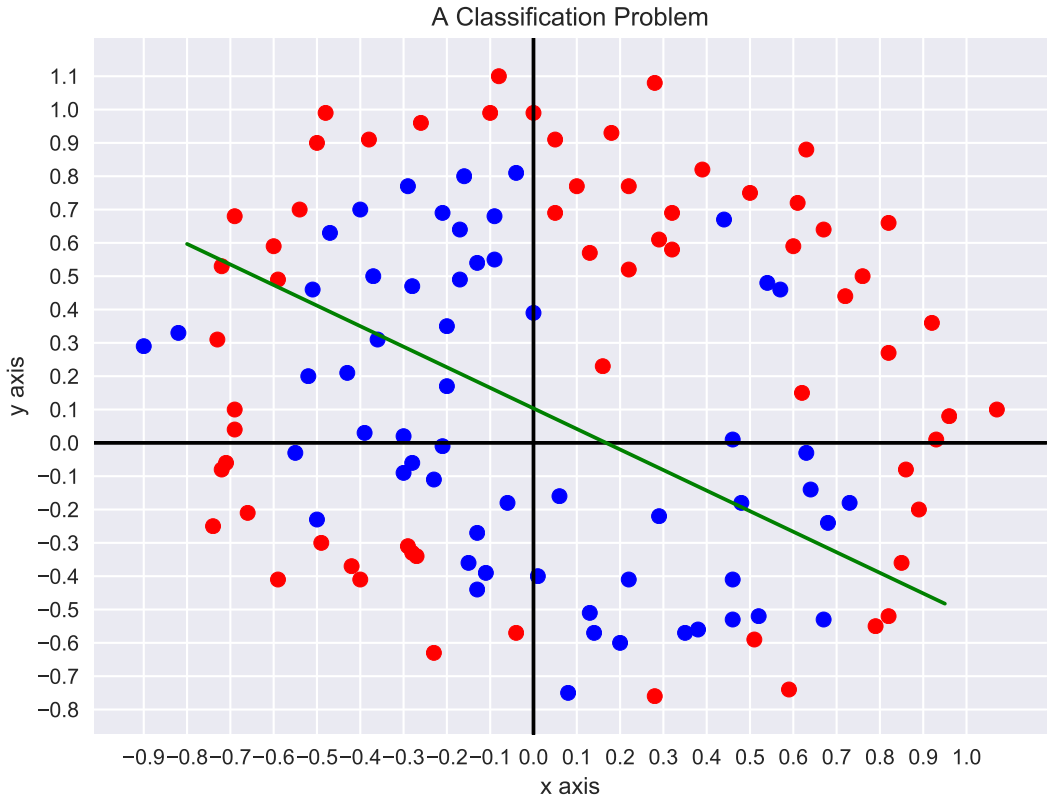

Figure 6.13: Fake data with linear decision boundary.

(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 $(M, \texttt{score}, \texttt{accuracy})$ 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 $\texttt{fx}^2$, $\texttt{fy}^2$, $\texttt{fx} \cdot \texttt{fy}$,

8. Line 26 extends both the training set and the test set with second order features.

```
1  import numpy  as np
2  import pandas as pd
3  import sklearn.linear_model as lm
4  from sklearn.model_selection import train_test_split
5
6  DF = pd.read_csv('fake-data.csv')
7  X  = np.array(DF[['x','y']])
8  Y  = np.array(DF['class'])
9
10 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=1)
11
12 def logistic_regression(X_train, Y_train, X_test, Y_test, reg=10000):
13     model    = lm.LogisticRegression(C=reg, tol=1e-6, solver='newton-cg')
14     M        = model.fit(X_train, Y_train)
15     score    = M.score(X_train, Y_train)
16     yPredict = M.predict(X_test)
17     accuracy = np.sum(yPredict == Y_test) / len(Y_test)
18     return M, score, accuracy
19
20 def extend(X):
21     n  = len(X)
22     fx = np.reshape(X[:,0], (n, 1))
23     fy = np.reshape(X[:,1], (n, 1))
24     return np.hstack([fx, fy, fx*fx, fy*fy, fx*fy])
25
26 X_train_quadratic, X_test_quadratic = extend(X_train), extend(X_test)
27 logistic_regression(X_train_quadratic, Y_train, X_test_quadratic, Y_test)
```

Figure 6.14: A script for second order logistic regression.

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 6.15 on page 136 shows the resulting decision boundary.

As adding second order features has increased the accuracy considerably, we proceed to add higher order features. Figure 6.16 on page 136 shows how we can add arbitrary polynomial features of higher degree. This script is a continuation of the script shown in Figure 6.14.

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

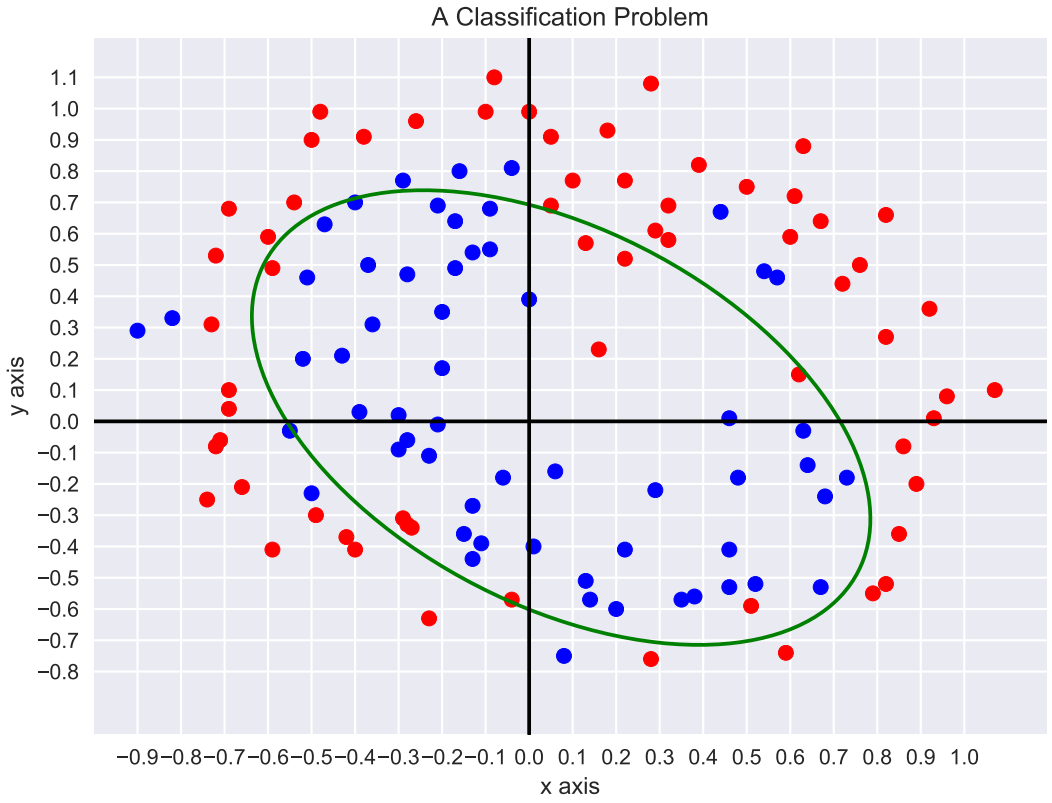

Figure 6.15: Elliptical decision boundary for fake data.

```
1  from sklearn.preprocessing import PolynomialFeatures
2
3  quartic = PolynomialFeatures(4, include_bias=False)
4  X_train_quartic = quartic.fit_transform(X_train)
5  X_test_quartic  = quartic.fit_transform(X_test)
6
7  logistic_regression(X_train_quartic, Y_train, X_test_quartic, Y_test)
```

Figure 6.16: 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 6.17 on page 137 shows the data with the resulting decision boundary.

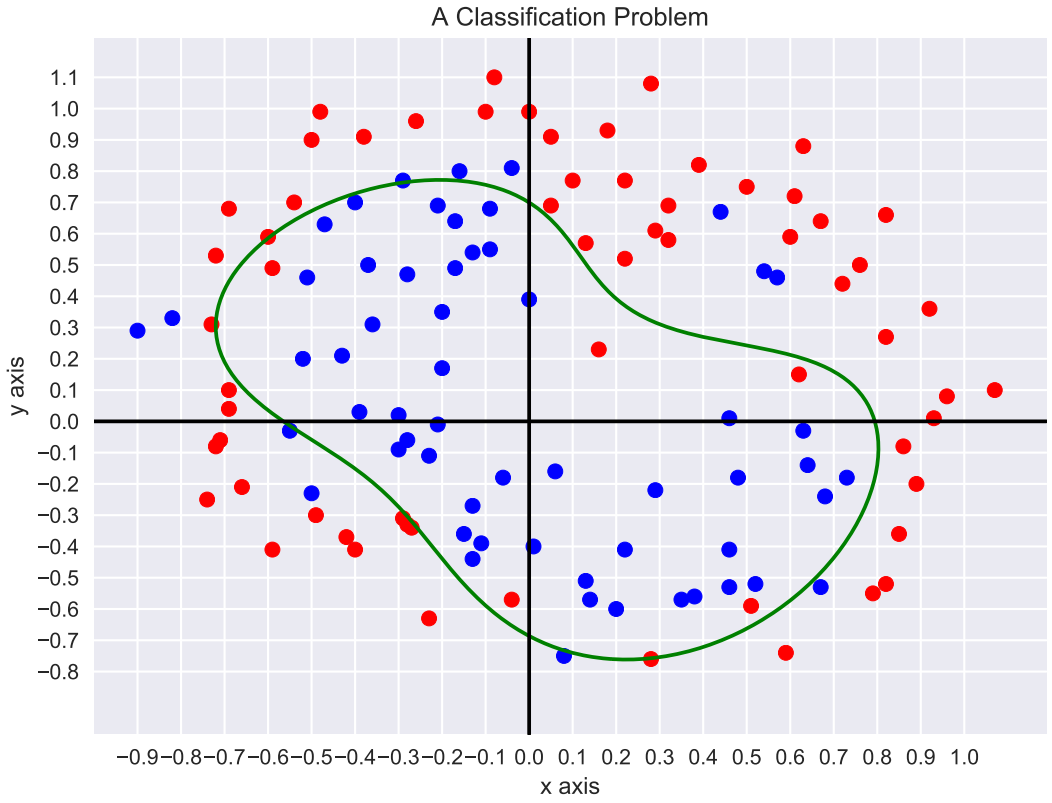

Figure 6.17: Fake data with a decision boundary of fourth order.

Nothing stops us from cranking the order of the polynomial features to be added higher. Figure 6.18 on page 138 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.

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 6.19 on page 139. Clearly, this decision boundary looks less complicated than the boundary shown in Figure 6.18. Contrary to the the previous figures, this figure shows all the data. The previous figures had only shown the training data.

Exercise 14:

(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 $k_1 + k_2 \leq n$, 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

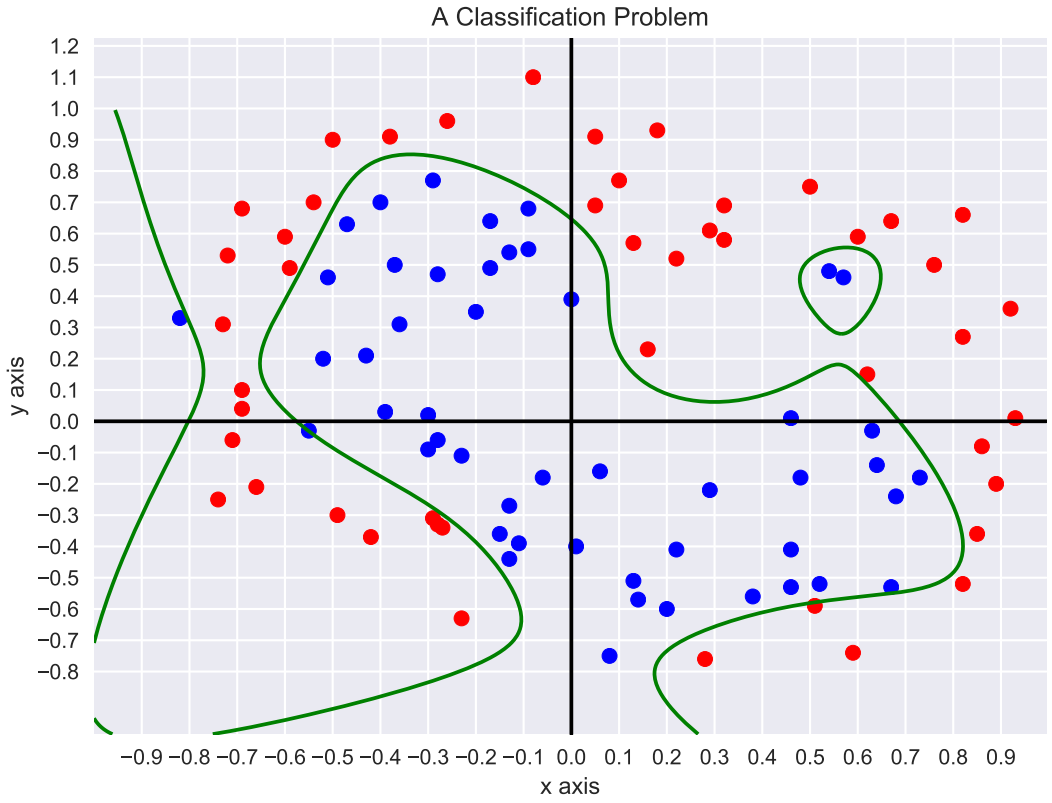

Figure 6.18: Fake data with a decision boundary of order 14.

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 $k_1 + k_2 + k_3 \leq n$. 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, $\cdots$, 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

$$x_1^{k_1} \cdot x_2^{k_2} \cdot \ldots \cdot x_d^{k_d} \quad \text{such that } k_1 + k_2 + \cdots + k_d \leq n.$$

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. ◇

6.5 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

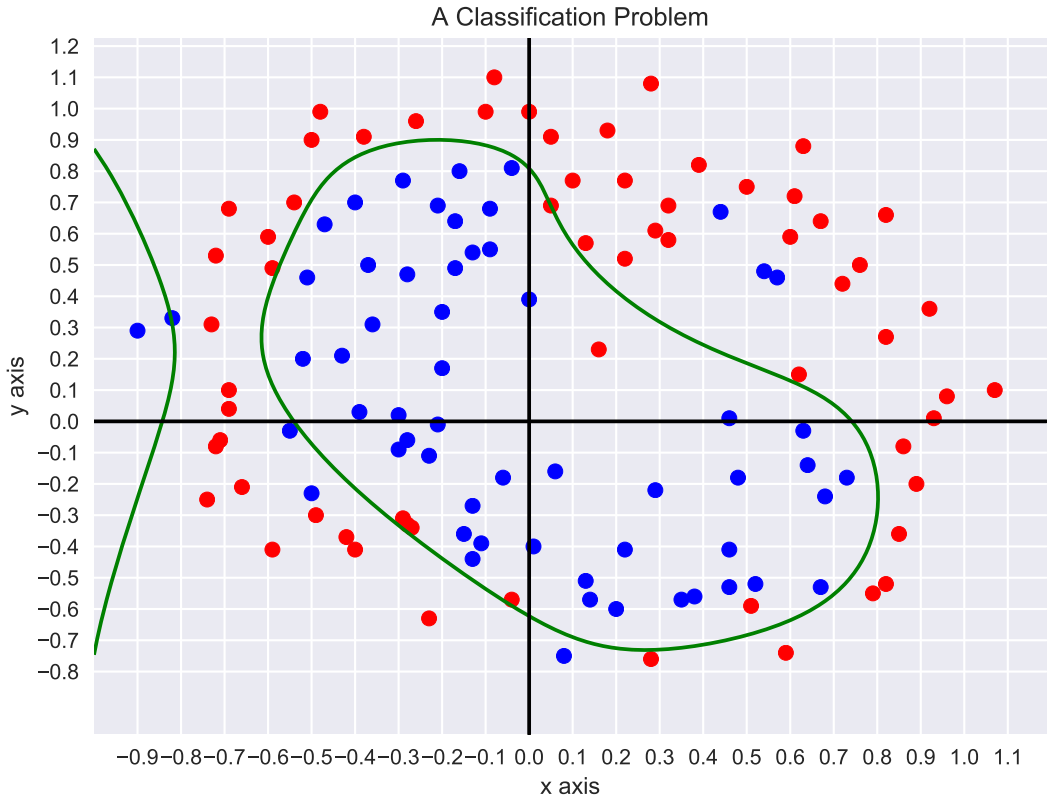

Figure 6.19: Fake data with a decision boundary of order 14, regularized.

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 $f_1, \cdots, f_m$ 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:

$$\begin{aligned} P(A_1 \wedge \cdots \wedge A_m \mid C) &= P(A_1 \wedge \cdots \wedge A_{m-1} \mid A_m \wedge C) \cdot P(A_m \mid C) \\ &= P(A_1 \wedge \cdots \wedge A_{m-2} \mid A_{m-1} \wedge A_m \wedge C) \cdot P(A_{m-1} \mid A_m \wedge C) \cdot P(A_m \mid C) \\ &= \cdots \\ &= P(A_1 \mid A_2 \wedge \cdots \wedge A_m \wedge C) \cdot \ldots \cdot P(A_{m-1} \mid A_m \wedge C) \cdot P(A_m \mid C) \\ &= \prod_{i=1}^{m} P(A_i \mid A_{i+1} \wedge \cdots \wedge A_m \wedge C) \end{aligned}$$

Two events A and B are defined to be conditional independent given an event C if and only if we have

$$P(A \mid C) = P(A \mid B \wedge C).$$

To put this equation differently, once we known 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 $f_1, \cdots, f_m$ it is assumed that the features $f_1, \cdots, f_m$ 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 weal 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 $P(C \mid f_1 \wedge \cdots \wedge f_m)$ of an object o with features $f_1, \cdots, f_m$ to be of class C is given as

$$\begin{aligned} P(C \mid f_1 \wedge \cdots \wedge f_m) &= \frac{P(f_1 \wedge \cdots \wedge f_m \mid C) \cdot P(C)}{P(f_1 \wedge \cdots \wedge f_m)} \cdot P(C) \\ &= \frac{\prod_{i=1}^{m} P(f_i \mid f_{i+1} \wedge \cdots \wedge f_m \wedge C)}{P(f_1 \wedge \cdots \wedge f_m)} \cdot P(C) \\ &= \frac{\prod_{i=1}^{m} P(f_i \mid C)}{P(f_1 \wedge \cdots \wedge f_m)} \cdot P(C) \end{aligned}$$

In the last line of the previous chain of equations we have used the fact that the features $f_1, \cdots, f_m$ are conditionally independent given C. Now a naive Bayes classifier works as follows: Assume we have a set of n classes $\mathcal{C} = \{C_1, \cdots, C_n\}$ from which we have to choose the class of an object o given the features $f_1, \cdots, f_m$. We assume that o has class C_k if and only if the probability $P(C_k \mid f_1 \wedge \cdots \wedge f_m)$ 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

$$\arg\max_{x\in S} f(x) := \texttt{arb}\Big(\big\{x \in S \mid \forall y \in S : f(y) \leq f(x)\big\}\Big),$$

where the function $\texttt{arb}(M)$ 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 $f_1, \cdots, f_m$, the naive Bayes classifier computes the most probable class as follows:

$$\texttt{NaiveBayes}(f_1, \cdots, f_m) := \arg\max_{C\in\mathcal{C}} \frac{\prod_{i=1}^{m} P(f_i \mid C)}{P(f_1 \wedge \cdots \wedge f_m)} \cdot P(C)$$

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:

$$\texttt{NaiveBayes}(f_1, \cdots, f_m) := \arg\max_{C\in\mathcal{C}} \left(\prod_{i=1}^{m} P(f_i \mid C)\right) \cdot P(C)$$

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 know about this object. $P(C)$ is estimated as the proportion of objects in T that are of class C:

$$P(C) \approx \frac{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C\}\big)}{\texttt{card}(T)}.$$

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 $\texttt{class}(t)$ determines the class of the object t, while $\texttt{card}(M)$ 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:

$$P(f \mid C) \approx \frac{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C \wedge \texttt{has}(t, f)\}\big)}{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C\}\big)}$$

Here, for an object t and a feature f the expression $\texttt{has}(t, f)$ is true if and only if t has the feature f.

6.5.1 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 6.20 on page 151 and 6.21 on page 152 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 a dictionary that is well suited for counting objects.
3. We set some variables that would need to be adapted if this notebook were to 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

   ```
   r"[\w']+"
   ```

   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

   $$C = \texttt{Counter}(\{\texttt{a}: 1, \texttt{b}: 2, \texttt{c}: 3\}),$$

   then the call C.`update`$(\{\texttt{a}, \texttt{d}\})$ changes the counter C such that afterwards

   $$C = \texttt{Counter}(\{\texttt{a}: 2, \texttt{b}: 2, \texttt{c}: 3, \texttt{d}: 1\}).$$

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 6.21 on page 152 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 $w \in$ `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

   $$\texttt{Spam\_Probability}[w] = P(w \mid C = \texttt{spam}).$$

   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

   $$P(w \mid C = \texttt{spam}) = \frac{\texttt{Spam\_Counter}[w]}{N}, \quad \text{where } N \text{ is the number of all spam training mails.}$$

   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 $P(w \mid C = \texttt{spam})$ 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 $P(w \mid C = \texttt{spam})$ is changed as follows:

   $$P(w \mid C = \texttt{spam}) = \frac{\texttt{Spam\_Counter}[w] + 1}{N + 1}, \quad \text{where } N \text{ is the number of all spam training mails.}$$

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:

$$\prod_{i=1}^{m} P(f_i \mid C) = \exp\left(\alpha + \sum_{i=1}^{m} \ln\big(P(f_i \mid C)\big)\right) \cdot \exp(-\alpha)$$

Here, the constant α has to chosen such that the application of the exponential function to the value

$$\alpha + \sum_{i=1}^{m} \ln\big(P(f_i \mid C)\big)$$

does not lead to an underflow error.

3. In order to evalate 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

   $$\texttt{precision} = \frac{\text{number of true positives}}{\text{number of true positives+number of false positives}}$$

   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

   $$\texttt{recall} = \frac{\text{number of true positives}}{\text{number of true positives+number of false negatives}}$$

   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 loose 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 loosing 10% of our ham emails because they are incorrectly specified as spam.

6.5.2 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

$$P(f = x \mid C) = \frac{1}{\sqrt{2 \cdot \pi} \cdot \sigma_{f,C}} \cdot \exp\left(-\frac{\big(x - \mu_{f,C}\big)^2}{2 \cdot \sigma_{f,C}^2}\right),$$

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 15: 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%. ◇

6.5.3 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 6.22 on page 153 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 $P(\texttt{Female})$ and $P(\texttt{Male})$ 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:

   $$P(C) \approx \frac{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C\}\big)}{\texttt{card}(T)}.$$

   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:

   $$P(f \mid C) \approx \frac{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C \wedge \texttt{has}(t, f)\}\big)}{\texttt{card}\big(\{t \in T \mid \texttt{class}(t) = C\}\big)}$$

   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 $g \in \{\texttt{'f'}, \texttt{'m'}\}$, 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.

6.6 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.

6.6.1 Non-Optimality of Logistic Regression

Figure 6.23 on page 154 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 6.24 on page 155 shows how logistic regression deals with with this problem.

A close inspection of Figure 6.24 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 6.25 on page 156, 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.

6.6.2 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

$$\mathbf{w} \cdot \mathbf{x}^{(i)} + b \geq 0.$$

Instead, we demand that

$$\mathbf{w} \cdot \mathbf{x}^{(i)} + b \geq 1 \tag{6.3}$$

holds. Similarly, if $\mathbf{x}^{(i)}$ is a negative example, we want to have that

$$\mathbf{w} \cdot \mathbf{x}^{(i)} + b \leq -1 \tag{6.4}$$

holds. Let us define the 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 6.3 by $y_i = 1$. Unsurprisingly, we get

$$y_i \cdot \left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right) \geq 1. \tag{6.5}$$

Similarly, let us multiply equation 6.3 by $y_i = -1$. This time, things get more interesting as the direction of the inequality is flipped:

$$y_i \cdot \left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right) \geq 1. \tag{6.6}$$

Notice that the equations 6.5 and 6.6 are the same! Hence inequation 6.5 holds for both positive and negative examples. We rewrite the last inequation as

$$y_i \cdot \left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right) - 1 \geq 0. \tag{6.7}$$

Those vectors $\mathbf{x}^{(i)}$ that satisfy the equality

$$y_i \cdot \left(\mathbf{w} \cdot \mathbf{x}^{(i)} + b\right) - 1 = 0$$

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

$$\mathbf{w} \cdot \mathbf{x}_+ + b = 1, \tag{6.8}$$

while $\mathbf{x}_-$ is a negative support vector and therefore satisfies

$$\mathbf{w} \cdot \mathbf{x}_- + b = -1. \tag{6.9}$$

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 $\mathbf{w}$:

$$\texttt{width} = \left(\mathbf{x}_+ - \mathbf{x}_-\right) \cdot \frac{\mathbf{w}}{||\mathbf{w}||} = \left(\mathbf{x}_+ \cdot \mathbf{w} - \mathbf{x}_- \cdot \mathbf{w}\right) \cdot \frac{1}{||\mathbf{w}||} \tag{6.10}$$

If we subtract equation 6.9 from equation 6.8, 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 6.10 yields the equation

$$\texttt{width} = \frac{2}{||\mathbf{w}||}.$$

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 6.7. 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 $\{\mathbf{x}^{(1)}, \cdots \mathbf{x}^{(n)}\}$, we define the Lagrangian $\mathcal{L}(\mathbf{w}, b, \alpha_1, \cdots, \alpha_n)$ as follows:

$$\mathcal{L}(\mathbf{w}, b, \alpha_1, \cdots, \alpha_n) := \frac{1}{2} \cdot ||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i \cdot \big(y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1\big).$$

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 $i \in \{1, \cdots, \}$ where we only have the inequality

$$y_i \cdot (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0$$

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

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i \cdot y_i \cdot \mathbf{x}^{(i)} \tag{6.11}$$

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

$$\sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \tag{6.12}$$

Let us rewrite the Lagrangian by substituting $\mathbf{w}$ with the right hand side of equation 6.11:

$$\begin{aligned}
\mathcal{L} &= \frac{1}{2} \cdot \left(\sum_{i=1}^{n} \alpha_i \cdot y_i \cdot \mathbf{x}^{(i)} \right) \cdot \left(\sum_{j=1}^{n} \alpha_j \cdot y_j \cdot \mathbf{x}^{(j)} \right) - \sum_{i=1}^{n} \alpha_i \cdot \left(y_i \cdot \left(\mathbf{x}^{(i)} \cdot \left(\sum_{j=1}^{n} \alpha_j \cdot y_j \cdot \mathbf{x}^{(j)} \right) + b \right) - 1 \right) \\
&= \frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j \cdot \left(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right) - \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j \cdot \left(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right) - b \cdot \underbrace{\sum_{i=1}^{n} \alpha_i \cdot y_i}_{=0} + \sum_{i=1}^{n} \alpha_i \\
&= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j \cdot \left(\mathbf{x}^{(i)} \cdot \mathbf{x}^{(j)} \right)
\end{aligned}$$

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 $\left\{ \Phi(\mathbf{x}^{(1)}), \cdots, \Phi(\mathbf{x}^{(n)}) \right\}$ 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

$$\Phi\big(\mathbf{x}\big) \cdot \Phi\big(\mathbf{y}\big).$$

This is done with the help of so called kernel functions: We define the dot product $\Phi\big(\mathbf{x}\big) \cdot \Phi\big(\mathbf{y}\big)$ as

$$\Phi\big(\mathbf{x}\big) \cdot \Phi\big(\mathbf{y}\big) := k(\mathbf{x}, \mathbf{y})$$

where k is called a kernel function. There are two kernel functions that are quite popular:

1. Polynomial kernels have the form

   $$k(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y} + c)^n,$$

   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 6.26 on page 157 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.

```
import os
import re
import numpy as np
import math

from collections import Counter

spam_dir_train = 'EmailData/spam-train/'
ham__dir_train = 'EmailData/ham-train/'
spam_dir_test  = 'EmailData/spam-test/'
ham__dir_test  = 'EmailData/ham-test/'
Directories    = [spam_dir_train, ham__dir_train, spam_dir_test, ham__dir_test]

no_spam    = len(os.listdir(spam_dir_train))
no_ham     = len(os.listdir(ham__dir_train))
spam_prior = no_spam / (no_spam + no_ham)
ham__prior = no_ham  / (no_spam + no_ham)

def get_words(fn):
    file = open(fn)
    text = file.read()
    text = text.lower()
    return set(re.findall(r"[\w']+", text))

def read_all_files(Directories):
    Words = Counter()
    for directory in Directories:
        for file_name in os.listdir(directory):
            Words.update(get_words(directory + file_name))
    return Words

Word_Counter = read_all_files(Directories)
Common_Words = { w for w, _ in Word_Counter.most_common(2500) }

def get_common_words(fn):
    return get_words(fn) & Common_Words

def count_commmon_words(directory):
    Words = Counter()
    for file_name in os.listdir(directory):
        Words.update(get_common_words(directory + file_name))
    return Words

spam_counter = count_commmon_words(spam_dir_train)
ham__counter = count_commmon_words(ham__dir_train)
```

Figure 6.20: A Naive Bayes Classifier for Spam Detection: Part I

```
Spam_Probability = {}
Ham__Probability = {}
for w in Common_Words:
    Spam_Probability[w] = (Spam_Counter[w] + 1) / (no_spam + 1)
    Ham__Probability[w] = (Ham__Counter[w] + 1) / (no_ham  + 1)

def spam_probability(fn):
    log_p_spam = 0.0
    log_p_ham  = 0.0
    words = get_common_words(fn)
    for w in Common_Words:
        if w in words:
            log_p_spam += math.log(Spam_Probability[w])
            log_p_ham  += math.log(Ham__Probability[w])
        else:
            log_p_spam += math.log(1.0 - Spam_Probability[w])
            log_p_ham  += math.log(1.0 - Ham__Probability[w])
    alpha = abs(min(log_p_spam, log_p_ham))
    if alpha > 400: # avoid overflow
        if log_p_spam < log_p_ham:
            return 0
        else:
            return 1
    p_spam = math.exp(log_p_spam + alpha) * spam_prior
    p_ham  = math.exp(log_p_ham  + alpha) * ham__prior
    return p_spam / (p_spam + p_ham)

def precission_recall(spam_dir, ham_dir):
    TN = 0 # true negatives
    FP = 0 # false positives
    for email in os.listdir(spam_dir):
        if spam_probability(spam_dir + email) > 0.5:
            TN += 1
        else:
            FP += 1
    FN = 0 # false negatives
    TP = 0 # true positives
    for email in os.listdir(ham_dir):
        if spam_probability(ham_dir + email) > 0.5:
            FN += 1
        else:
            TP += 1
    precision = TP / (TP + FP)
    recall    = TP / (TP + FN)
    accuracy  = (TN + TP) / (TN + TP + FN + FP)
    return precision, recall, accuracy
```


Figure 6.21: A Naive Bayes Classifier for Spam Detection: Part II

```
def read_names(file_name):
    Result = []
    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')
MaleNames   = read_names('names-male.txt'  )
pFemale     = len(FemaleNames) / (len(FemaleNames) + len(MaleNames))
pMale       = len(MaleNames)   / (len(FemaleNames) + len(MaleNames))

def conditional_prop(c, g):
    if g == 'f':
        return len([n for n in FemaleNames if n[-1] == c]) / len(FemaleNames)
    else:
        return len([n for n in MaleNames   if n[-1] == c]) / len(MaleNames)

Conditional_Probability = {}
for c in 'abcdefghijklmnopqrstuvwxyz':
    for g in ['f', 'm']:
        Conditional_Probability[(c, g)] = conditional_prop(c, g)

def classify(name):
    last   = name[-1]
    female = Conditional_Probability[(last, 'f')] / pFemale
    male   = Conditional_Probability[(last, 'm')] / pMale
    if female >= male:
        return 'f'
    else:
        return 'm'

total, correct = 0, 0
for n in FemaleNames:
    if classify(n) == 'f':
        correct += 1
    total += 1
for n in MaleNames:
    if classify(n) == 'm':
        correct += 1
    total += 1
accuracy = correct / total
print(f'The accuracy of our estimator is {accuracy}.')
```

Figure 6.22: A naive Bayes classifier for predicting the gender of a name.

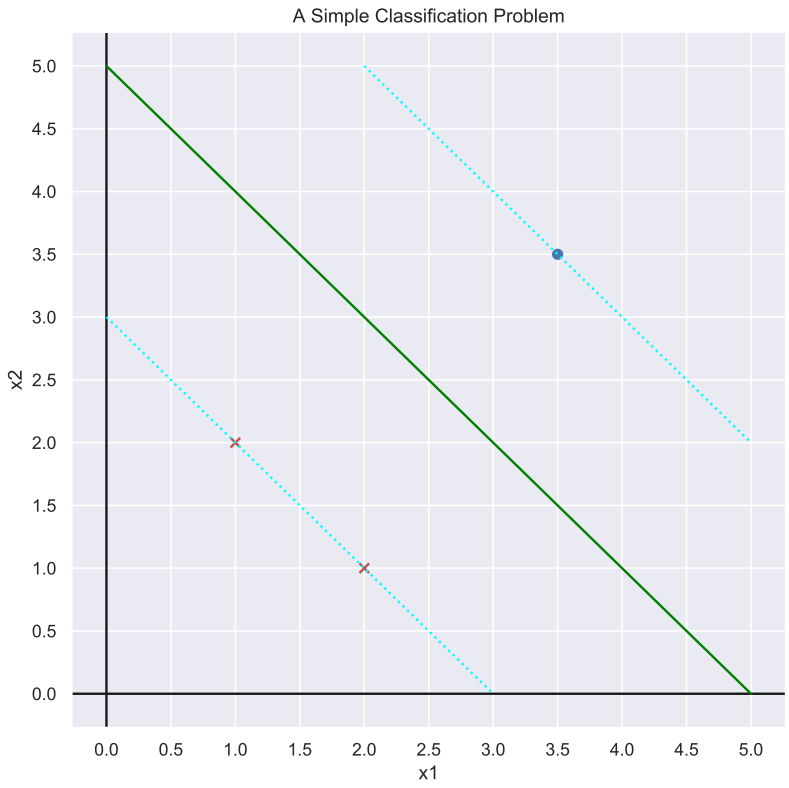

Figure 6.23: Three points to separate.

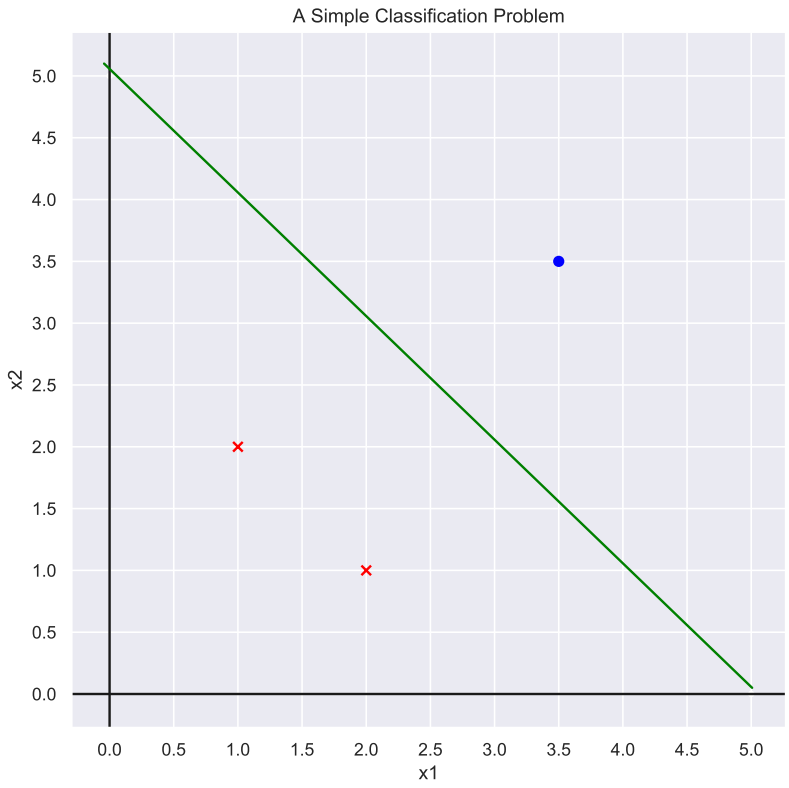

Figure 6.24: Three points separated by logistic regression.

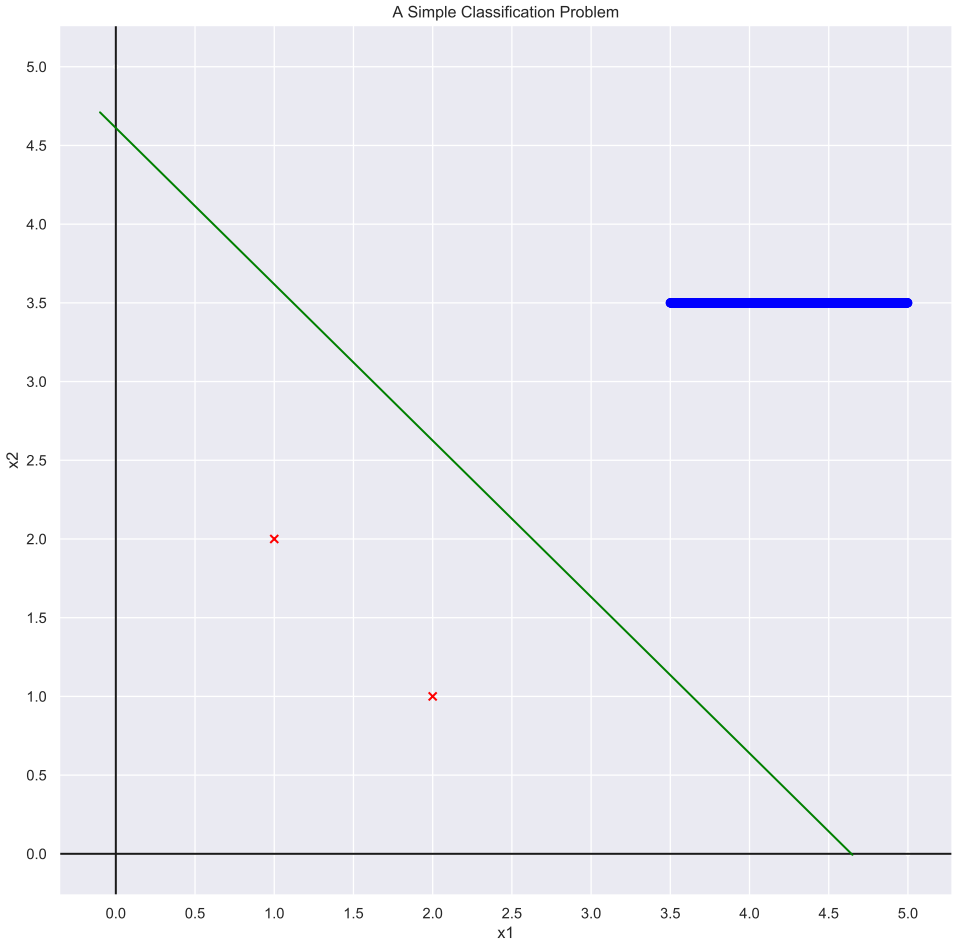

Figure 6.25: Points separated by logistic regression.

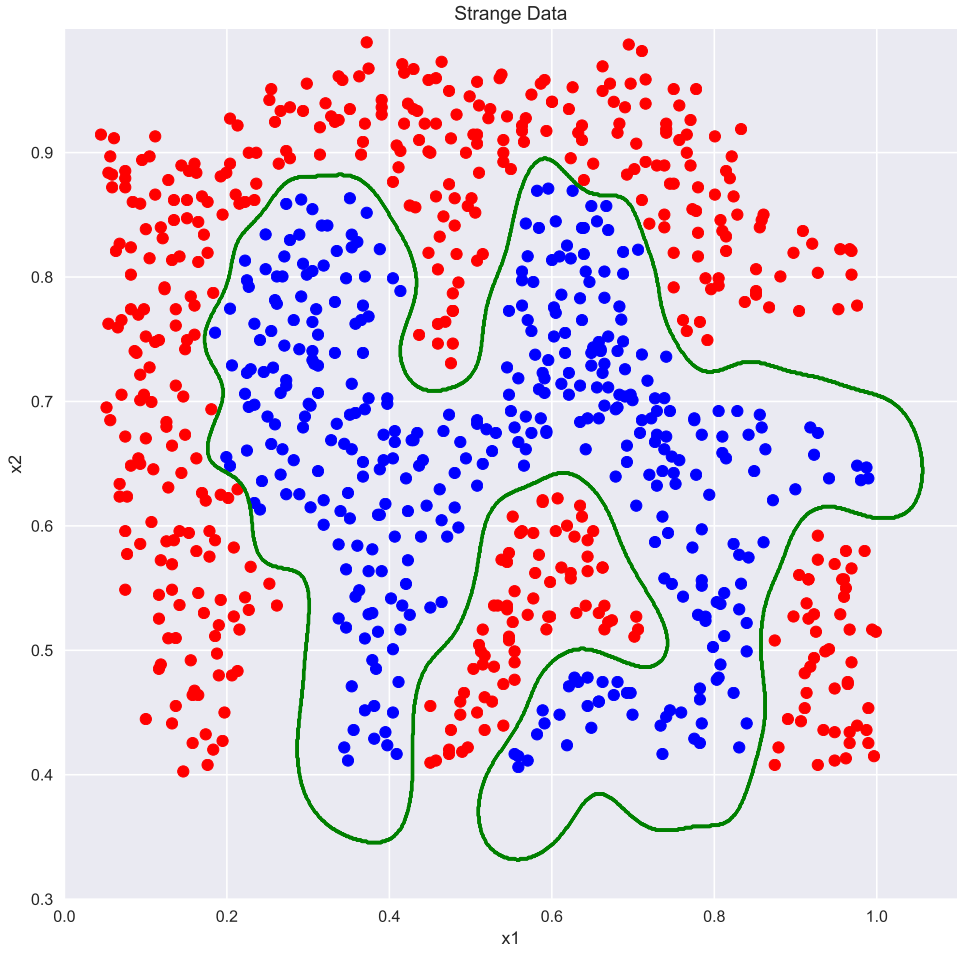

Figure 6.26: Points separated by a support vector machine.

Chapter 7

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 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 [Nie15]. 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. Furthermore, the online learning platform Coursera offers a specialization called Deep Learning that is exclusively devoted to 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 back propagation.

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.

7.1 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, work[1]. 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 looking at neural networks, a single neuron with n 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 6 on page 119 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, 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 $[m(1), \cdots, m(L)]$ of L natural numbers. The number

[1] Actually, when it comes to brains, although there are many speculations, surprisingly little is known for a fact.

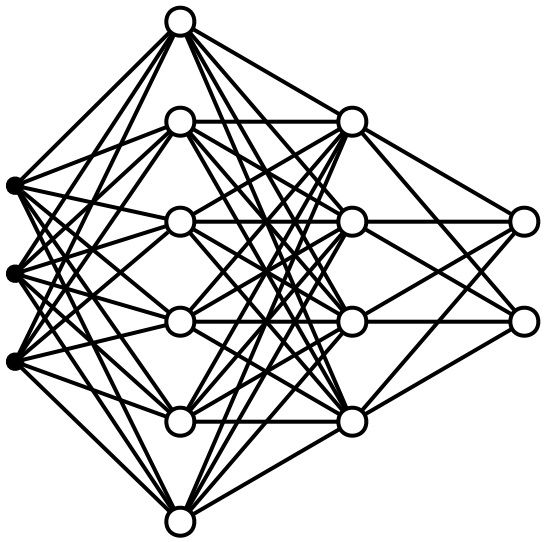

Figure 7.1: A neural network with topology $[3, 6, 4, 2]$.

L is called the number of layers and for $i \in \{2, \cdots, L\}$ the number $m(i)$ 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 neural network is called a deep neural network. Figure 7.1 on page 160 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 7.2 on page 161. 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 $[m(1), \cdots, m(L)]$, 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

$$W^{(l)} := \big(w_{j,k}^{(l)}\big).$$

Note that $W^{(l)}$ is an $m(l) \times m(l-1)$ matrix, i.e. we have

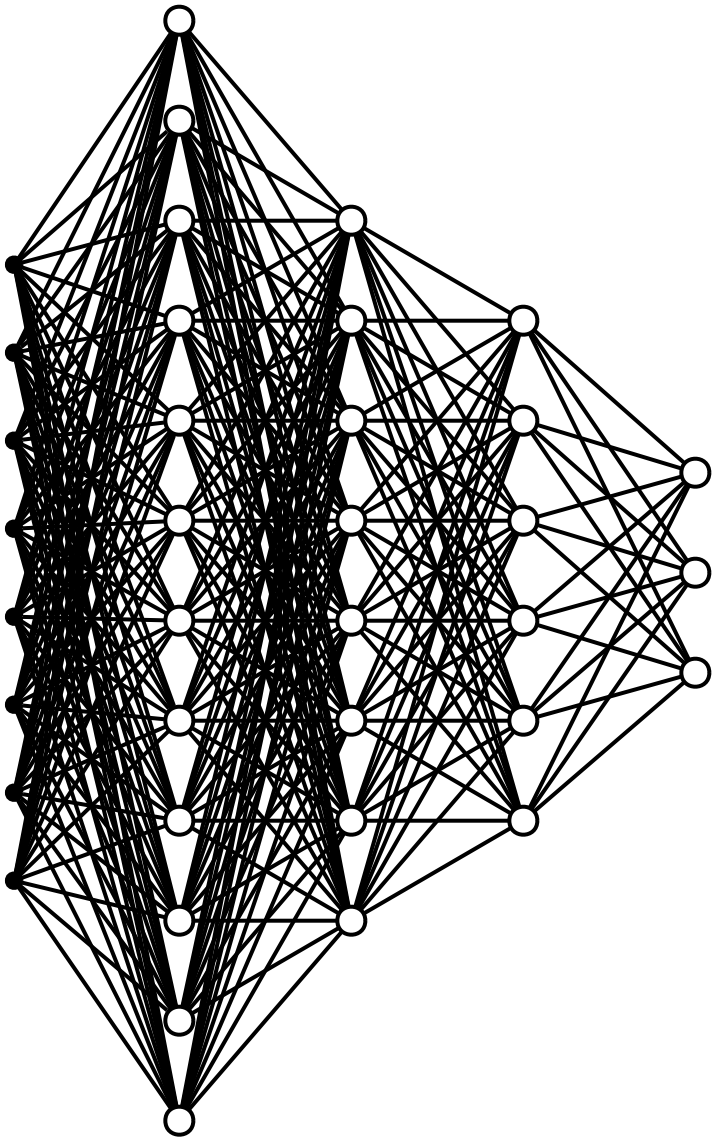

Figure 7.2: A neural network with topology $[8, 12, 8, 6, 3]$.

$W^{(l)} \in \mathbb{R}^{m(l) \times m(l-1)}$.

The j-th neuron in layer l has the bias $b_j^{(l)}$. These biases of layer l are combined into the bias vector

$$\mathbf{b}^{(l)} := \langle b_1^{(l)}, \cdots, b_{m(l)}^{(l)} \rangle^\top.$$

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

$$a_j^{(l)}(\mathbf{x}) := S\left(\left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)}(\mathbf{x})\right) + b_j^{(l)}\right) \quad \text{for all } l \in \{2, \cdots, L\}. \tag{FF2}$$

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:

$$\mathbf{a}^{(1)}(\mathbf{x}) := \mathbf{x}, \tag{FF1v}$$

$$\mathbf{a}^{(l)}(\mathbf{x}) := S\Big(W^{(l)} \cdot \mathbf{a}^{(l-1)}(\mathbf{x})\Big) + \mathbf{b}^{(l)}\Big) \quad \text{for all } l \in \{2, \cdots, L\}. \tag{FF2v}$$

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

$$\mathbf{o}(\mathbf{x}) := \langle a_1^{(L)}(\mathbf{x}), \cdots, a_{m(L)}^{(L)}(\mathbf{x}) \rangle^\top = \mathbf{a}^{(L)}(\mathbf{x}).$$

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

$$\mathbf{a}^{(2)}(\mathbf{x}) := S\big(W^{(2)} \cdot \mathbf{a}^{(1)}(\mathbf{x}) + \mathbf{b}^{(2)}\big) = S\big(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}\big).$$

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

$$\mathbf{a}^{(3)}(\mathbf{x}) := S\big(W^{(3)} \cdot \mathbf{a}^{(2)}(\mathbf{x}) + \mathbf{b}^{(3)}\big) = S\Big(W^{(3)} \cdot S\big(W^{(2)} \cdot \mathbf{x} + \mathbf{b}^{(2)}\big) + \mathbf{b}^{(3)}\Big)$$

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

$$\langle \mathbf{x}^{(i)}, \mathbf{y}^{(i)} \rangle \quad \text{for } i = 1, \cdots, n$$

such that

$$\mathbf{x}^{(i)} \in \mathbb{R}^{m(1)} \text{ and } \mathbf{y}^{(i)} \in \mathbb{R}^{m(L)}.$$

Our goal is to choose the weight matrices $W^{(l)}$ and the bias vectors $\mathbf{b}^{(l)}$ in a way such that

$$\mathbf{o}\big(\mathbf{x}^{(i)}\big) = \mathbf{y}^{(i)} \quad \text{for all } i \in \{1, \cdots, n\}.$$

Unfortunately, in general we will not be able to achieve equality for all $i \in \{1, \cdots, n\}$. Therefore, our goal is to minimize the error instead. To be more precise, the quadratic error cost function is defined as

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \Big(\mathbf{o}\big(\mathbf{x}^{(i)}\big) - \mathbf{y}^{(i)}\Big)^2.$$

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

$$C_{\mathbf{x},\mathbf{y}}\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big) := \frac{1}{2} \cdot \Big(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}\Big)^2,$$

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

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{n} \cdot \sum_{i=1}^{n} C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}\Big(W^{(2)}, \cdots W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big).$$

As the notation

$$C_{\mathbf{x},\mathbf{y}}\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}\Big)$$

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 minimum[2]. 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.

7.2 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 AMD Radeon Vega `VII` graphic card offers about 13.8 teraflops in single precision performance. It consumes about 300 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. It needed 6 megawatt to operate. The cost to build ASCI White was about $110,000,000\,\$$. To compare, the AMD RX Vega `VII` costs $699\,\$$.

[2] In 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*.

The Nvidia Titan V comes at 2,999 $ and offers a stunning 110 teraflops for deep learning via the CUDA API.

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.

7.2.1 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:

$$z_j^{(l)} := \left(\sum_{k=1}^{m(l-1)} w_{j,k}^{(l)} \cdot a_k^{(l-1)} \right) + b_j^{(l)} \quad \text{for all } j \in \{1, \cdots, m(l)\} \text{ and } l \in \{2, \cdots, L\}.$$

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 formulæ get too cluttered. Essentially, $z_j^{(l)}$ is the input to the sigmoid function when the activation $a_j^{(l)}$ 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 $z_j^{(l)}$. 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 $\varepsilon_j^{(l)}$ does depend 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:

$$\boldsymbol{\varepsilon}^{(l)} := \begin{pmatrix} \varepsilon_1^{(l)} \\ \vdots \\ \varepsilon_{m(l)}^{(l)} \end{pmatrix}.$$

The vector $\boldsymbol{\varepsilon}^{(l)}$ is called the error in layer l.

7.2.2 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 `x` and `y` is computed by the expression `x * y`.

7.2.3 Backpropagation: The Equations

Now we are ready to state the backpropagation equations. The first of these four equations reads as follows:

$$\varepsilon_j^{(L)} = (a_j^{(L)} - y_j) \cdot S'\big(z_j^{(L)}\big) \quad \text{for all } j \in \{1, \cdots, m(L)\}, \tag{BP1}$$

where $S'(x)$ denotes the derivative of the sigmoid function. We have shown in Chapter 6 that this derivative satisfies the equation

$$S'(t) = \big(1 - S(t)\big) \cdot S(t).$$

The equation (BP1) can also be written in vectorized form using the Hadamard product:

$$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'\big(\mathbf{z}^{(L)}\big) \tag{BP1v}$$

Here, we have vectorized the application of the function S' to the vector $\mathbf{z}^{(L)}$, i.e. the expression $S'\big(\mathbf{z}^{(L)}\big)$ is defined as follows:

$$S'\begin{pmatrix} z_1^{(L)} \\ \vdots \\ z_{m(L)}^{(L)} \end{pmatrix} := \begin{pmatrix} S'\big(z_1^{(L)}\big) \\ \vdots \\ S'\big(z_{m(L)}^{(L)}\big) \end{pmatrix}.$$

The next equation computes $\varepsilon_j^{(l)}$ for $l < L$.

$$\varepsilon_j^{(l)} = \sum_{i=1}^{m(l+1)} w_{i,j}^{(l+1)} \cdot \varepsilon_i^{(l+1)} \cdot S'\big(z_j^{(l)}\big) \quad \text{for all } j \in \{1, \cdots, m(l)\} \text{ and } l \in \{2, \cdots, L-1\}. \tag{BP2}$$

This equation is more succinct in vectorized notation:

$$\boldsymbol{\varepsilon}^{(l)} = \Big(\big(W^{(l+1)}\big)^\top \cdot \boldsymbol{\varepsilon}^{(l+1)}\Big) \odot S'\big(\mathbf{z}^{(l)}\big) \quad \text{for all } l \in \{2, \cdots, L-1\}. \tag{BP2v}$$

Note that this equation computes the error in layer l for $l < L$ in terms of the error in layer $l+1$: The error $\boldsymbol{\varepsilon}^{(l+1)}$ at layer $l+1$ is propagated backwards through the neural network to produce the error $\boldsymbol{\varepsilon}^{(l)}$ 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 $b_j^{(l)}$. We have

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial b_j^{(l)}} = \varepsilon_j^{(l)} \quad \text{for all } j \in \{1, \cdots, m(l)\} \text{ and } l \in \{2, \cdots, L\}. \tag{BP3}$$

This equation shows the reason for defining the error terms $\varepsilon_j^{(l)}$. In vectorized notation, this equation takes the following form:

$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \quad \text{for all } l \in \{2, \cdots, L\}. \tag{BP3v}$$

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 $w_{j,k}^{(l)}$:

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial w_{j,k}^{(l)}} = \varepsilon_j^{(l)} \cdot a_k^{(l-1)} \quad \text{for all } j \in \{1, \cdots, m(l)\},\ k \in \{1, \cdots, m(l-1)\}, \text{ and } l \in \{2, \cdots, L\}. \tag{BP4}$$

In vectorized notation, this equation can be written as:

$$\nabla_{W^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^\top \quad \text{for all } l \in \{2, \cdots, L\}. \tag{BP4v}$$

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:

$$\boldsymbol{\varepsilon}^{(L)} = (\mathbf{a}^{(L)} - \mathbf{y}) \odot S'(\mathbf{z}^{(L)}) \tag{BP1v}$$

$$\boldsymbol{\varepsilon}^{(l)} = \left(\left(W^{(l+1)}\right)^\top \cdot \boldsymbol{\varepsilon}^{(l+1)}\right) \odot S'(\mathbf{z}^{(l)}) \quad \text{for all } l \in \{2, \cdots, L-1\} \tag{BP2v}$$

$$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \quad \text{for all } l \in \{2, \cdots, L\} \tag{BP3v}$$

$$\nabla_{W^{(l)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^\top \quad \text{for all } l \in \{2, \cdots, L\} \tag{BP4v}$$

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.

7.2.4 A Proof of the Backpropagation Equations

Since we are living in a time of fake news and alternative facts where truth isn't truth and not even a president can be trusted, you should not lightly believe the validity of the backpropagation equations. Instead, we have 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 multivariate calculus.

Lets us start with the proof of equations BP1. Remember that we have defined the numbers $\varepsilon_j^{(l)}$ as

$$\varepsilon_j^{(l)} = \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(l)}},$$

while the numbers $z_j^{(l)}$ 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 $\mathbf{a}^{(L)}(\mathbf{x}) = S\left(\mathbf{z}^{(L)}\right)$, the chain rule of calculus tells us that $\varepsilon_j^{(L)}$ can be computed as follows:

$$
\begin{aligned}
\varepsilon_j^{(L)} &= \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(L)}} \\
&= \frac{\partial}{\partial z_j^{(L)}} \frac{1}{2} \cdot \big(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}\big)^2 \\
&= \frac{1}{2} \cdot \frac{\partial}{\partial z_j^{(L)}} \sum_{i=1}^{m(L)} \Big(a_i^{(L)}(\mathbf{x}) - y_i\Big)^2 \\
&= \frac{1}{2} \cdot \frac{\partial}{\partial z_j^{(L)}} \sum_{i=1}^{m(L)} \Big(S\big(z_i^{(L)}\big) - y_i\Big)^2 \\
&= \frac{1}{2} \cdot \sum_{i=1}^{m(L)} 2 \cdot \Big(S\big(z_i^{(L)}\big) - y_i\Big) \cdot \frac{\partial}{\partial z_j^{(L)}} S\big(z_i^{(L)}\big) \\
&= \sum_{i=1}^{m(L)} \Big(S\big(z_i^{(L)}\big) - y_i\Big) \cdot S'\big(z_i^{(L)}\big) \cdot \frac{\partial z_i^{(L)}}{\partial z_j^{(L)}} \\
&= \sum_{i=1}^{m(L)} \Big(S\big(z_i^{(L)}\big) - y_i\Big) \cdot S'\big(z_i^{(L)}\big) \cdot \delta_{i,j} \qquad \delta_{i,j} \text{ denotes the Kronecker delta} \\
&= \Big(S\big(z_j^{(L)}\big) - y_j\Big) \cdot S'\big(z_j^{(L)}\big) \\
&= \Big(a_j^{(L)} - y_j\Big) \cdot S'\big(z_j^{(L)}\big)
\end{aligned}
$$

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

$$f : \mathbb{R}^k \to \mathbb{R} \quad \text{and} \quad g : \mathbb{R}^n \to \mathbb{R}^k.$$

are differentiable[3]. If the function $h : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$h(\mathbf{x}) := f\big(g(\mathbf{x})\big) \quad \text{for all } \mathbf{x} \in \mathbb{R}^n,$$

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

[3] If this text had been written in German, I would have said that f and g are "total differenzierbar".

$$\begin{aligned}
\varepsilon_j^{(l)} &= \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(l)}} \\
&= \sum_{i=1}^{m(l+1)} \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} && \text{using the chain rule of multivariate calculus} \\
&= \sum_{i=1}^{m(l+1)} \varepsilon_i^{(l+1)} \cdot \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} && \text{using the definition of } \varepsilon_i^{(l+1)}
\end{aligned}$$

In order to proceed, we have to remember the definition of $z_i^{(l+1)}$. We have

$$z_i^{(l+1)} = \left(\sum_{k=1}^{m(l)} w_{i,k}^{(l+1)} \cdot S\big(z_k^{(l)}\big) \right) + b_i^{(l+1)}$$

Therefore, the partial derivatives $\dfrac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}}$ can be computed as follows:

$$\begin{aligned}
\frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} &= \sum_{k=1}^{m(l)} w_{i,k}^{(l+1)} \cdot S'\big(z_k^{(l)}\big) \cdot \frac{\partial z_k^{(l)}}{\partial z_j^{(l)}} \\
&= \sum_{k=1}^{m(l)} w_{i,k}^{(l+1)} \cdot S'\big(z_k^{(l)}\big) \cdot \delta_{k,j} \\
&= w_{i,j}^{(l+1)} \cdot S'\big(z_j^{(l)}\big)
\end{aligned}$$

If we substitute this expression back into the result we got for $\varepsilon_j^{(l)}$ we have shown the following:

$$\begin{aligned}
\varepsilon_j^{(l)} &= \sum_{i=1}^{m(l+1)} \varepsilon_i^{(l+1)} \cdot \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \\
&= \sum_{i=1}^{m(l+1)} \varepsilon_i^{(l+1)} \cdot w_{i,j}^{(l+1)} \cdot S'\big(z_j^{(l)}\big) \\
&= \sum_{i=1}^{m(l+1)} w_{i,j}^{(l+1)} \cdot \varepsilon_i^{(l+1)} \cdot S'\big(z_j^{(l)}\big)
\end{aligned}$$

Therefore, we have now proven equation (BP2).

We proceed to prove equation (BP4). According to the chain rule we have

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial w_{j,k}^{(l)}} = \frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{j,k}^{(l)}}$$

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 $\dfrac{\partial z_j^{(L)}}{\partial w_{j,k}^{(l)}}$. The term $z_j^{(l)}$ has been defined as follows:

$$z_j^{(l)} = \left(\sum_{k=1}^{m(l)} w_{j,k}^{(l)} \cdot S\big(z_k^{(l-1)}\big) \right) + b_j^{(l)}.$$

Hence we have

$$\frac{\partial z_j^{(l)}}{\partial w_{j,k}^{(l)}} = S\big(z_k^{(l-1)}\big) = a_k^{(l-1)}.$$

Combining these equations we arrive at

$$\frac{\partial C_{\mathbf{x},\mathbf{y}}}{\partial w_{j,k}^{(l)}} = a_k^{(l-1)} \cdot \varepsilon_j^{(l)}.$$

Therefore, equation (BP4) has been verified.

Exercise 16: Prove equation (BP3). ◇

7.3 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

$$\langle \mathbf{x}^{(1)}, \mathbf{y}^{(1)} \rangle\rangle, \langle \mathbf{x}^{(2)}, \mathbf{y}^{(2)} \rangle\rangle, \cdots, \langle \mathbf{x}^{(n)}, \mathbf{y}^{(n)} \rangle\rangle,$$

then the quadratic error cost function has been previously defined as the sum

$$C\Big(W^{(2)}, \cdots, W^{(L)}, \mathbf{b}^{(2)}, \cdots, \mathbf{b}^{(L)}; \mathbf{x}^{(1)}, \mathbf{y}^{(1)}, \cdots, \mathbf{x}^{(n)}, \mathbf{y}^{(n)}\Big) := \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \Big(\mathbf{o}\big(\mathbf{x}^{(i)}\big) - \mathbf{y}^{(i)} \Big)^2.$$

In practical applications of neural networks, the number of training examples is usually big. For example, when we later develop a neural network to classify handwritten digits, we will have $60,000$ training examples. More ambitious projects that use neural networks to classify 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

$$\frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial w_{j,k}^{(l)}} \quad \text{and} \quad \frac{1}{2 \cdot n} \cdot \sum_{i=1}^{n} \frac{\partial C_{\mathbf{x}^{(i)},\mathbf{y}^{(i)}}}{\partial b_j^{(l)}}$$

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.

7.4 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 7.3 shows the first 18 images.

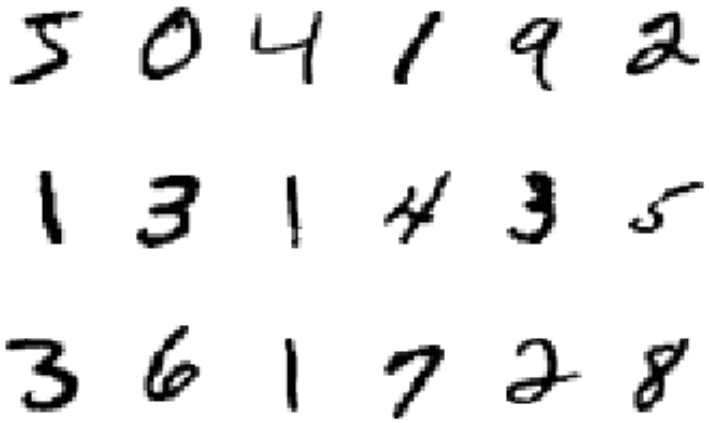

Figure 7.3: The first 18 images of the Mnist dataset.

We will use the first 60 000 of these images to train a neural network, while the remaining 10 000 images will be used to check the accuracy of the trained network. 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

We will next 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 7.4, 7.5, 7.6, and 7.7.

The code on Figure 7.4 on page 172 shows the code to load the image files. We discuss it line by line.

1. Since the images are have been compressed as a `.gz` file, we need the module `gzip` to uncompress the file.

```
import gzip
import pickle
import numpy              as np
import matplotlib.pyplot as plt
import random

def vectorized_result(d):
    e    = np.zeros((10, 1), dtype=np.float32)
    e[d] = 1.0
    return e

def load_data():
    with gzip.open('mnist.pkl.gz', 'rb') as f:
        train, validate, test = pickle.load(f, encoding="latin1")
    training_inputs   = [np.reshape(x, (784, 1)) for x in train[0]]
    training_results  = [vectorized_result(y) for y in train[1]]
    training_data     = zip(training_inputs, training_results)
    validation_inputs = [np.reshape(x, (784, 1)) for x in validate[0]]
    validation_data   = zip(validation_inputs, validate[1])
    test_inputs       = [np.reshape(x, (784, 1)) for x in test[0]]
    test_data         = zip(test_inputs, test[1])
    training_data     = list(training_data)
    validation_data   = list(validation_data)
    test_data         = list(test_data)
    return (training_data, validation_data, test_data)

(training_data, validation_data, test_data) = load_data()
```

Figure 7.4: Code to load the image files.

2. The format that has been used to store the images is called pickle. This 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`.

3. The images of the handwritten digits that we are going to import have a size of 28×28 pixels. In this program, we will store the images 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 $d \in \{0, \cdots, 0\}$. 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 one 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 $d \in \{0, \cdots, 9\}$ and returns a `numpy` 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

```
vectorized_result(2) = array([[0.],
                              [0.],
                              [1.],
                              [0.],
                              [0.],
                              [0.],
                              [0.],
                              [0.],
                              [0.],
                              [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,
   (b) `validation_data` holds 10 000 images that can be used for the validation of hyper-parameters,
   (c) `test_data` holds the remaining 10 000 images.

   The images are stored as pairs (x, y): x is a `numyp` array of shape $(784, 1)$ and y is a `numpy` array of shape $(10, 1)$.

```
28  def rndMatrix(rows, cols):
29      return np.random.randn(rows, cols) / np.sqrt(cols)
30
31  def sigmoid(x):
32      return 1.0 / (1.0 + np.exp(-x))
33
34  def sigmoid_prime(x):
35      s = sigmoid(x)
36      return s * (1 - s)
```

Figure 7.5: The constructor of the class `network`.

Figure 7.5 on page 173 shows the implementation of three utility functions.

1. The function `rndMatrix`(r, c) creates a matrix of shape $r, c)$ that is filled with random numbers. These numbers have 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 to big for otherwise the associated neurons would saturate and their learning would be very slow.

2. The function `sigmoid`(x) computes the sigmoid function. If x is a number, `sigmoid`(x) computes

$$S(x) := \frac{1}{1 + \texttt{exp}(-x)}$$

If $\mathbf{x}$ is a vector of the form $\mathbf{x} = (x_1, \cdots, x_n)^\top$, we have

$$S(\mathbf{x}) = \big(S(x_1), \cdots, S(x_n)\big)^\top.$$

3. The function `sigmoid_prime`(x) computes the derivative of the sigmoid function. The implementation is based on the equation:

   $$S'(x) = S(x) \cdot \big(1 - S(x)\big)$$

   x can either be a number or a vector.

Figure 7.6 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, 60 neurons and 100 neurons.

   (a) For 30 neurons, the computation took about 1 minute and 58 seconds and the trained neural network achieved an accuracy of 94.8%.

   (b) For 60 neurons, the computation took 2 minutes and 51 seconds the network achieved an accuracy of 96.1%.

   (c) If there are 100 neurons in the hidden layer, the computation took 3 minutes and 42 secondsseed and achieved an accuracy of 97.8%.

   For 100 neurons, the number of weights in the hidden layer is $784 \cdot 100 = 78\,400$. 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) `mWeightsH` 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 (FF1) and (FF2).

6. The method `sgd` implements stochastic gradient descent. It receives 5 arguments.

```
37  class Network(object):
38      def __init__(self, hiddenSize):
39          self.mInputSize  = 28 * 28
40          self.mHiddenSize = hiddenSize
41          self.mOutputSize = 10
42          self.mBiasesH    = np.zeros((self.mHiddenSize, 1))
43          self.mBiasesO    = np.zeros((self.mOutputSize, 1))
44          self.mWeightsH   = rndMatrix(self.mHiddenSize, self.mInputSize)
45          self.mWeightsO   = rndMatrix(self.mOutputSize, self.mHiddenSize)
46
47  def feedforward(self, x):
48      AH = sigmoid(self.mWeightsH @ x  + self.mBiasesH)
49      AO = sigmoid(self.mWeightsO @ AH + self.mBiasesO)
50      return AO
51
52  def sgd(self, training_data, epochs, mbs, eta, test_data):
53      n_test = len(test_data)
54      n      = len(training_data)
55      for j in range(epochs):
56          random.shuffle(training_data)
57          mini_batches = [training_data[k : k+mbs] for k in range(0, n, mbs)]
58          for mini_batch in mini_batches:
59              self.update_mini_batch(mini_batch, eta)
60          print('Epoch %2d: %d / %d' % (j, self.evaluate(test_data), n_test))
61
62  def update_mini_batch(self, mini_batch, eta):
63      nabla_BH = np.zeros((self.mHiddenSize, 1))
64      nabla_BO = np.zeros((self.mOutputSize, 1))
65      nabla_WH = np.zeros((self.mHiddenSize, self.mInputSize))
66      nabla_WO = np.zeros((self.mOutputSize, self.mHiddenSize))
67      for x, y in mini_batch:
68          dltNbl_BH, dltNbl_BO, dltNbl_WH, dltNbl_WO = self.backprop(x, y)
69          nabla_BH += dltNbl_BH
70          nabla_BO += dltNbl_BO
71          nabla_WH += dltNbl_WH
72          nabla_WO += dltNbl_WO
73      alpha = eta / len(mini_batch)
74      self.mBiasesH  -= alpha * nabla_BH
75      self.mBiasesO  -= alpha * nabla_BO
76      self.mWeightsH -= alpha * nabla_WH
77      self.mWeightsO -= alpha * nabla_WO
```

Figure 7.6: The class `Network`, part `I`.

(a) `training_data` is a list of pairs of the form $\big(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\big)$. 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. In order to train the neural network to recognize handwritten digits, we will use 30 iterations.

(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 to be quite uncritical.

(d) `eta` is the learning rate.

(e) `test_data` is the list of test data. These data are only used to check the accuracy, 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 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

   $$\nabla_{\mathbf{b}^{(l)}} C_{\mathbf{x},\mathbf{y}} \quad \text{and} \quad \nabla_{W^{(l)}} C_{\mathbf{x},\mathbf{y}}$$

   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 7.7 computes the gradients of the bias vectors and the weight matrices with respect to a single training example $\langle \mathtt{x}, \mathtt{y} \rangle$. The implementation of `backprop` proceeds as follows:

```
78      def backprop(self, x, y):
79          # feedforward pass
80          ZH = self.mWeightsH @ x  + self.mBiasesH
81          AH = sigmoid(ZH)
82          ZO = self.mWeightsO @ AH + self.mBiasesO
83          AO = sigmoid(ZO)
84          # backwards pass, output layer
85          epsilonO = (AO - y) # * sigmoid_prime(ZO)
86          nabla_BO = epsilonO
87          nabla_WO = epsilonO @ AH.transpose()
88          # backwards pass, hidden layer
89          epsilonH = (self.mWeightsO.transpose() @ epsilonO) * sigmoid_prime(ZH)
90          nabla_BH = epsilonH
91          nabla_WH = epsilonH @ x.transpose()
92          return (nabla_BH, nabla_BO, nabla_WH, nabla_WO)
93
94      def evaluate(self, test_data):
95          test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]
96          return sum(int(y1 == y2) for (y1, y2) in test_results)
97      # end of class Network
98
99  net = Network(60)
100 net.sgd(training_data, 30, 20, 0.3, test_data)
```

Figure 7.7: The class `Network`, part `II`.

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`.

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 `epsilonO` 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

   $$\nabla_{\mathbf{b}^{(3)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(3)}.$$

   This gradient is stored in the variable `nabla_B0`.

7. According to equation (BP4v), the gradient of the cost function with respect to the weight matrix of the output layer is given as

   $$\nabla_{W^{(3)}} C_{\mathbf{x},\mathbf{y}} = \boldsymbol{\varepsilon}^{(3)} \cdot \left(\mathbf{a}^{(2)}\right)^\top.$$

   This gradient is stored in the variable `nabla_W0`.

8. Next, the error in the hidden layer `epsilonH` is computed using the backpropagation equation (BP2v)

   $$\boldsymbol{\varepsilon}^{(2)} = \left(\left(W^{(3)}\right)^\top \cdot \boldsymbol{\varepsilon}^{(3)}\right) \odot S'(\mathbf{z}^{(2)}).$$

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 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 address:

https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Python/Digit-Recognition.ipynb.

Bibliography

[BH69] Arthur Earl Bryson, Jr. and Yu-Chi Ho. *Applied Optimal Control: Optimization, Estimation, and Control*. Blaisdell Pub., Waltham, Massachusetts, 1969.

[Cho17] Francois Chollet. *Deep Learning with Python*. Manning Publications, Greenwich, CT, USA, 1st edition, 2017.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[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.

[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.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[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.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, 3rd edition, 2009.

[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.

List of Figures

Index