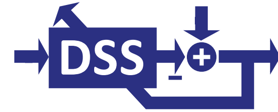
Lab

Machine Learning



Prof. Dr.-Ing. Gerhard Schmidt

Digital Signal Processing and System Theory
Christian-Albrechts-Universität zu Kiel

2025

Version: **1.0** (May 2025)

Contents

I Introduction 7

1 Introduction to Python 9

1.1 Development Environment . . . 9

1.1.1 Setup of Lab Environment . . . 10

1.1.2 Overview of common Libraries . . . 13

1.1.3 Juypter Notebook within VSCode . . . 14

1.2 Python Fundamentals . . . 14

1.3 Exercises . . . 19

2 Data Sets 23

2.1 Basics . . . 23

2.1.1 Introduction . . . 23

2.1.2 Confusion Matrix and Derived Performance Metrics . . . 24

2.1.3 Data Partitioning . . . 25

2.1.4 Code Reference . . . 26

2.2 Lab Exercises . . . 29

II Data Preprocessing 31

3 Linear Discriminant Analysis 33

3.1 Basics . . . 33

3.1.1 Principle Component Analysis . . . 33

3.1.2 Linear Discriminant Analysis . . . 35

3.1.3 Scikit-learn Classes . . . 36

3.2 Exercise . . . 36

4 Independent Component Analysis 39

4.1 Basics 39
4.1.1 Principles of ICA 40
4.1.2 Outline of FastICA 41
4.1.3 Scikit-learn Classes 44
4.2 Exercise 44

III Unsupervised Learning **47**

5 Autoencoders **49**
5.1 Tensorflow Basics 49
5.1.1 Important Commands in Tensorflow 50
5.1.2 Model Creation and Layers Addition 50
5.1.3 Network Training 50
5.1.4 Data Prediction 51
5.1.5 Subclassing 51
5.2 Basics 51
5.2.1 Encoder 51
5.2.2 Decoder 51
5.2.3 Mathematical Foundations 52
5.3 Simple Autoencoder Example 53
5.3.1 Problem 53
5.3.2 Dataset 53
5.3.3 Code 54
5.4 Exercise 55

IV Supervised Learning **59**

6 Decision Trees & Random Forests **61**
6.1 Basics 61
6.1.1 Decision Trees 61
6.1.2 Random Forests 64
6.2 Exercise 65

7 Support Vector Machines **69**
7.1 Basics 69

7.1.1 Linear SVM . . . 71
7.1.2 Nonlinear SVM . . . 73
7.1.3 Applications and Limitations . . . 74
7.2 Exercise . . . 75

8 Convolutional Neural Networks 81
8.1 Tensorboard Basics . . . 81
8.2 Basics . . . 81
8.2.1 Structure . . . 82
8.2.2 Complexity . . . 83
8.3 Exercise . . . 84

V System Aspects 89

9 Application Example 91
9.1 General Information . . . 91
9.2 Exercises . . . 92
9.3 Example Datasets . . . 93
9.3.1 CIFAR-10 and CIFAR-100 Dataset . . . 93
9.3.2 UCI Parkinsons Dataset . . . 93
9.3.3 German Traffic Sign Detection Benchmark Dataset . . . 94
9.3.4 German Traffic Sign Recognition Benchmark Dataset . . . 94
9.3.5 Boston Housing Dataset . . . 94

VI Authors 95

10 Authors 97

VII References 99

11 References 101

Part I

Introduction

Lab *Machine Learning*

Chapter 1

Introduction to Python

Within this chapter, the setup of an essential programming environment for this lab is presented. Furthermore, an insight into Python and other primary libraries are given to ensure working ability in this lab. Here, the difficulty level is chosen such that Python newbies can easily follow and get the required basics for the upcoming lab exercises in the field of machine learning.

1.1 **Development Environment** 9
- 1.1.1 Setup of Lab Environment 10
- 1.1.2 Library Overview 13
- 1.1.3 Juypter Notebook 14

1.2 **Python Fundamentals** 14

1.3 **Exercises** 19

1.1 Development Environment

Python has become one of the popular programming languages because the syntax is simple, many libraries are freely available, and it provides a good entry point for beginners. Numerous frameworks in machine learning are primarily based on Python, such as *TensorFlow* and *Keras*. Additionally, fundamental add-on Python libraries such as *NumPy*, *SciPy*, and *Scikit-learn* are easily accessible and provide excellent support in tackling challenging tasks in the field of machine learning. In addition, today many open-source libraries offer an additional Python Application Programming Interface (API) by default, such as the real-time optimized computer vision library *OpenCV* (Open Computer Vision - for more details see `https://opencv.org/`).

Especially in the field of digital signal processing and also system theory, the popularity of Python has increased rapidly over the last few years, primarily due to the usage of so-called *Jupyter notebooks*. These Python-based notebooks are comparable to *Live Scripts* in the commercial program *MATLAB* the name *Jupyter* arises from the first letters of the programming languages **Ju**lia, **Py**thon and **R**.

The following subsections explain how the programming environment required for this laboratory can be set up and put into operation. Furthermore, the necessary program libraries will be added, too. The installation procedure will be exemplarily provided for a *Windows* operating system. For

other major operating systems (*macOS* and *Linux*), you can find detailed installation instructions on the web and also on the Python website: https://www.python.org/.

1.1.1 Setup of Lab Environment

Within this lab, the standard distribution of Python version 3.10 is used combined with Visual Studio Code and Juypter Notebook. While the course content may also work with other Python versions, we cannot offer full support for these. First of all, a Python distribution is required and has to be installed. A suitable Version depending on the operating system can be directly downloaded from the Python download page: https://www.python.org/downloads/. The installation is performed by using the built-in installer (see Figure 1.1). There are other ways to install Python (installation via IDEs or the Microsoft App Store), but we advise against these for those new to Python. Not because these options are more difficult or inferior, but to keep potential installation issues consistent throughout this course.

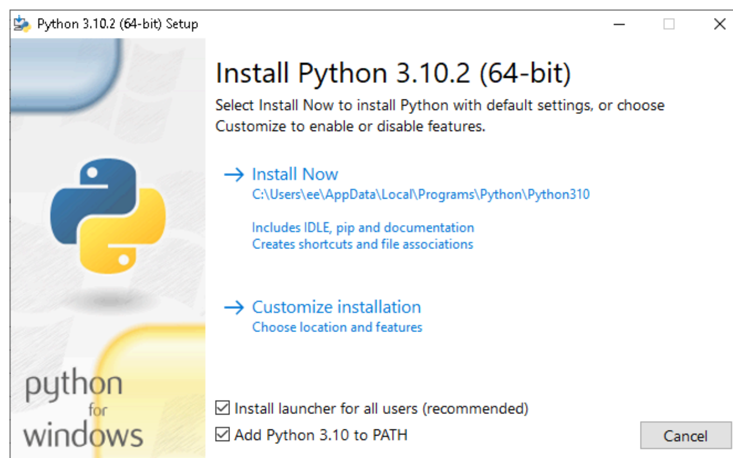


Figure 1.1: Python-Installer 3.10.2 (64-bit) on Microsoft Windows 10.

If the usual `python` command shall be linked to the current Python version, the option *'Add Python 3.10 to PATH'* should be selected during the installation process. **This is highly recommended!** After successfully finishing the installer, the installation can be checked using the Windows command prompt and the following command:

```
python --version
```

or

```
py --version
```

If different Python versions are displayed with these commands, this is an indication that multiple Python kernels are already installed on the computer. This isn't a problem in itself, but should be considered later. The current program version will be displayed in the Windows command prompt like:

```
C:\>python --version
Python 3.11.10
```

The advantage of Python is undoubtedly the possibility of easy code editing, such as using a simple text editor (e.g., Notepad++, Microsoft Editor) and performing code execution in a simple windows terminal. Nevertheless, using a programmer-friendly environment is quite helpful for this lab, also for debugging purposes. Therefore, the Integrated Development Environment (IDE) of Microsoft Visual Code (VSCode) will be used as an essential programming environment within this lab. This open-source code editor can be downloaded from `https://code.visualstudio.com/` and can be easily installed by the available installer. The feature of VSCode is that it is completely programmed in HTML and Node.js, which ensures the possibility of high configuration ability for the user. Additionally, this also allows supporting several programming languages and data formats. Due to numerous VSCode extensions, the development environment can be individually adapted to the required programming languages and can also be switched within one coding session.

Finally, to get VSCode working for Python, a particular extension is required, which is also accessible within VSCode. This enables VSCode to be a full-fledged Python editor with debugging functionalities, which is particularly helpful for this lab, especially the *Variable View* which enables checking values of variables or structures for correctness during run-time.

After the installation of VSCode, the Python extension can be easily installed after initial start-up. For doing this, it is necessary to make the extensions sideboard visible by applying the shortcuts (`Ctrl+Shift+X`) or clicking the corresponding program button on the left side. In the new available window the search term »**Python**« can be entered. This initiates an automatic search on the VSCode marketplace and displays all hits that are relevant for performing programming in Python. The first entry should be the required extension and could be directly installed by clicking the corresponding *install* button as shown in Figure 1.2.

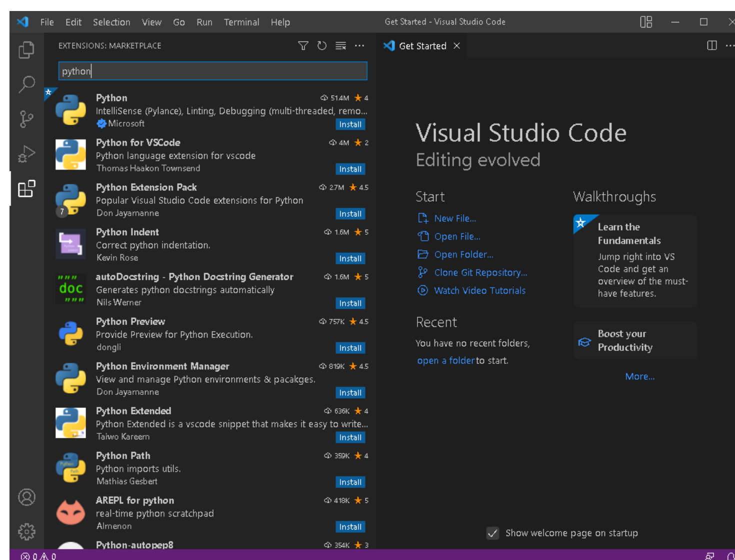


Figure 1.2: Marketplace within VSCode.

In order to take advantage of *Juypter Notebook* within VSCode, it is recommended to install also the Juypter extension. Therefore the VSCode marketplace has to be searched again and the corresponding »**Juypter**« package can be installed. Finally, this extension allows to save the Python source code very advantageously as a Juypter notebook and enables the possibility to export everything as PDF-File for documentation purposes. After the extensions have been successfully

installed, VSCode can be restarted. Finally, the fundamental programming environment is prepared. Nevertheless, further preparation is necessary to ensure a basic working capability within this lab. Therefore, the next steps are to create a working directory, a virtual Python environment and install required libraries.

To keep track of all created files through this lab, it is highly recommended to create a dedicated programming directory and use it constantly. Therefore, the creation of a new folder is helpful, e.g., `machine_learning_lab`. This folder shall be used as fundamental working directory.

The terminal integrated within VSCode allows to input common Python commands. Therefore, the working directory can be easily adjusted by the command:

```
cd <PATH>\textbackslash machine\_learning\_lab
```

Replace `<PATH>` with the corresponding desired path. At next, a new virtual environment is generated. The basic idea of a virtual environment is that the original Python installation remains untouched and is not affected by the installation of additional packages\libraries. For the generation of a new Python virtual environment, the VSCode terminal can be used again.

The command

```
python -m venv lab-env
```

will generate a new Python environment called `lab-env` in the current path of the terminal. Finally, this environment can be activated by applying the command

```
lab-env\Scripts\activate
```

Now `(lab-env)` appears in front of the actual pathname within the terminal. Finally, in the terminal this could be look like:

```
(lab-env) PS E:\machine_learning_lab>
```

A deactivation can be accomplished with the command `deactivate` so that `(lab-env)` disappears, e.g. like

```
PS E:\machine_learning_lab>
```

In the laboratory context, it is recommended to ensure the correct selection of the working directory and the activation of the created virtual environment each time. After finishing programming work, the environment should preferably be also deactivated. More Information about virtual Python environments can be found in the tutorial https://docs.python.org/3/tutorial/venv.html.

Another problem that can occur here or with other prompts is that running scripts is not allowed on the system. An error message like this appears:

```
...cannot be loaded because the execution of scripts is disabled on this system.
```

In this case, you must first allow the execution of external scripts. To do this, open the PowerShell console (important: not the normal command promptWindows PowerShell is different) and enter the following command:

```
Get-ExecutionPolicy
```

If something other than `RemoteSigned` is output, the following the command

```
Set-ExecutionPolicy RemoteSigned
```

must be executed and then `[J]` or `[Y]` – depending on your language settings.

In machine learning, reading, extracting, preparing, and manipulating data sets are fundamental. For this purpose, several useful functions for Python are freely available and accessible, which have already included basic operations. Furthermore, they also make the handling of extensive data sets manageable and make essential machine learning algorithms available. The libraries/frameworks required in the context of this lab can be installed advantageously with the terminal command

```
pip install numpy scipy pandas scikit-learn matplotlib tensorflow jupyter
```

within the virtual environment `(lab-env)`. After installing the Python libraries packages, a fully usable technical development environment for this lab is now ready. The following two sections give a short library overview and a short introduction using a Juypter Notebook Document within VSCode.

1.1.2 Overview of common Libraries

The most common libraries, which will be used throughout this laboratory, are briefly introduced in the following. Nevertheless, it must be pointed out that in the later experiments, libraries will find an application that are not presented explicitly or described more precisely. However, this working principle is common practice in Python projects because programmed and optimized libraries already exist for most machine learning tasks, which are simple to use and thus save self-programming work.

NumPy: The library Numerical Python (`NumPy`) enables the execution of numerical calculations on data structures (array, matrices, etc.) through numerous performant functions. Values can be searched, modified, and adjusted within a data structure with the help of a dedicated Python API. More information about NumPy can be found at https://numpy.org/.

Matplotlib: This library primarily allows the creation of diagrams and visualizations. Essential functions for line charts, histograms ,and scatter plots are provided. The particular advantage of this library is that in Juypter Notebook visualization can be represented directly. More information about Matplotlib can be found at https://matplotlib.org/.

Scikit-learn: The library scikit-learn, which is based on numby and matplotlib, is one of the most widely used Python packages in machine learning. It includes numerous data mining algorithms as well as machine learning algorithms for regression, classification, and data clustering. In addition, this library offers further functionalities regarding the preprocessing of data and the dimensionality reduction of data sets. Scikit-learn is constantly being enhanced and updated. More information about scikit-learn can be found at https://scikit-learn.org/.

Pandas: `Pandas` ensures data preprocessing and merging of data from different database sources. Data can be read in the common format structures, which are common to TensorFlow or Keras. Furthermore, pandas offer complementary possibilities to NumPy for the analysis and manipulation of data structures. Currently, TensorFlow relies more heavily on NumPy structures for data input. More information about pandas can be found at https://pandas.pydata.org/.

SciPy: Another Python library, namely Scientific Python (`SciPy`), integrates both NumPy and Pandas and extends them with additional functions, which are particularly helpful in the field of digital signal processing. This enables, for example, the Fourier transform in Python. In addition, SciPy offers a variety of other functions that can be used for dedicated machine learning tasks. More information about SciPy can be found at `https://scipy.org/`.

TensorFlow: `TensorFlow` is a framework that provides a whole collection of libraries in the field of machine learning. The calculation within TensorFlow takes place with so-called data flow graphs. This allows computation from a simple sum to highly complex matrix operations to be performed and represented. In the context of this lab, we use the version TensorFlow 2, which has a Keras library already integrated for easy handling of neural networks. More information about TensorFlow 2 can be found at `https://www.tensorflow.org`.

Keras: Keras is a standalone library that can handle various modules from other machine learning libraries, such as TensorFlow. Finally, since the release of TensorFlow 2, Keras has been fully integrated into the TensorFlow package, eliminating the need for a separate package installation. However, Keras will still be maintained as a standalone library. More information about Keras can be found at `https://keras.io/`.

1.1.3 Juypter Notebook within VSCode

In order to start with the first lines of code in VSCode, a new Juypter notebook document must be created. This can be done via *View* → *Command Plate* (Ctrl+Shift+P) → *Juypter: Create New Juypter Notebook*. Afterwards, an empty notebook will appear in VSCode. A notebook document consists of a list of consecutive cells, where a primary distinction is required between a *Markdown Cells* and *Code Cells*. Markdown cells allow formatting text to be entered, while code cells only allow Python program code. Furthermore, usual code comments can be used within a code cell, but they do not experience formatting capabilities. Both cell types have in common that either simple text or Python code is entered. The cell must be executed to display the content as formatted text or to execute the code; therefore, each cell must be successfully *executed*. This means that the compile process has already been executed with (Ctrl+Enter or Shift+Enter) or the *Run all* command has been selected via the corresponding button. Good laboratory documentation is achievable by using markdown cells. Consequently, this should be also used within the different lab experiments. An example of the different cell types within a Juypter document is given in Figure 1.3 for both cell types within VSCode. Finally a Juypter notebook can be saved in the *JSON format* with the corresponding file extension `.ipynb`. Furthermore, the notebook can be also exported as *HTML* or *PDF-File*.

1.2 Python Fundamentals

Creating a variable in Python is relatively straightforward: A variable name can be assigned to a dedicated value by using an equal sign `=`. Unlike other programming languages (e.g., C, C++), it is not necessary to explicitly ensure a data type declaration because a *dynamic typing* is automatically performed. The applied data type of a variable can be indicated very advantageously with the function `type()`.

```
1 a = 25
```

Code Cell

```
1  a = "This is a first test of using Juypter Notebooks within Visual Studio Code in DSS-Lab"
2  print(a)
```
[2] Python

```
... This is a first test of using Juypter Notebooks within Visual Studio Code in DSS-Lab
```

Mardown Cell

Below two identical Markdown cells are shown, once in source form and once complied.

```
1  This is a **text** with *formatting characters*.
2
3  1. Numbered lists start with a number
4  2. where the number can be chosen arbitrarily
5  1. namely like this - when translating it will be set correctly
6      1. sublists are created with indentation (tab)
7
8  Paragraphs are created by a blank line
9  Line breaks are created by two spaces at the end of a line
10
11 Web links can also be inserted quickly using round brackets
12 for example the link to the [DSS page](https://dss-kiel.de/).
```
Markdown

This is a **text** with *formatting characters*.

1. Numbered lists start with a number
2. where the number can be chosen arbitrarily
3. namely like this - when translating it will be set correctly
    1. sublists are created with indentation (tab)

Paragraphs are created by a blank line
Line breaks are created by two spaces at the end of a line

Web links can also be inserted quickly using round brackets
for example the link to the DSS page.

Figure 1.3: Celltypes within Juypter document (Markdown cell and Code cell).

User comments within program code are also useful in Python for an essential reading of program code. Comments are generally introduced with `#`. This means that text following `#` is not considered as instruction within one program line. For the output of texts, variables etc. the function `print()` is very helpful. Unlike other functions, any number of parameters can be passed directly through this function.

```
1 a = 25.125
2 print(type(a)) #Datatype of variable a = 25.125 is float!
```

Mathematical operands can be used for variables in the usual way. This applies to the operands `+`, `-`, `*` and `/`. Only the power operation takes a little getting used to, depending on the programming language commonly used. For example, if variable `b` to the square has to be calculated, the syntax `b**2` or `pow(b,2)` is used and not the symbol `^`.

```
1  b = a**2 #Result -> 631.266
```

The common bool values `True` and `False` are also coded as 1 and 0 within Python. Bool values arise, for example, as a result of comparison operators such as `==`, `<`, `<=`, `>` and `>=`. An inequality can be expressed using !=.

```
1  a == b #Result False
```

Strings are also present in Python, where they belong to the basic data types. The `+` operator can be overloaded advantageously to be able to combine strings by applying the operator in the common way. The string data type is defined by using single `' '` or double quotes `" "` and can then be assigned to a variable. Other operators are still available for strings, such as the comparison operator (`==`). In addition, there are useful string functions, such as `len()`, which makes it possible to determine the number of characters within a string. Furthermore, the string data type allows access to individual elements with square brackets `a[1]`, since a string array represents elements of characters. Moreover, Python allows to cut out whole text segments with e.g. `a[0:7]`, which is known as *slicing*.

```
1  a = "Digital Signal Processing"
2  b = 'and System Theory'
3  c = a + ' ' + b
4  a == 'Digital Signal Processing' #True
5  d = len(a) #25
6  e = a[1] #i
7  g = a[0:7] #Digital
```

Note

The element numbering for arrays, lists etc. always starts at 0 in Python and not at 1, as this is the case in C/C++.

The `f-strings` method, available for the string data type, ensures string composition and provided a useful formatting method for text output.

```
1  outputtext = f"The name of the group is {c} at {'CAU-Kiel'}"
2  print(outputtext) #The name of the Group is Digital Signal Processing and
     System Theory at CAU-Kiel.
```

Control structures (*if-statement*, *for-loop*, *while-loop*) also exist for a dedicated program control in Python. In the following, the usage of the different control structures will be briefly given based on simple examples. Here it must be noted that instructions are automatically summarized to blocks in Python, in which they are indented. In other programming languages, the use of brackets is common practice.

```
1  #Example of an if-statement
2  value = 20
3  if value < 19:
4      print('The value of the variable is less than 19!')
5  elif value == 19:
6      print('The value of the variable is equal 19!')
7  else:
8      print('The value of the variable is greater than 19!')
9  #Output: The value of the variable is greater than 19!
```

```
1 #Example of a for-loop
2  values = [0,2,4,6,8] # Serie/Sequence of objects
3  for x in values:
4      print(f"Value: {x}")
5  #Output: Value: 0 Value: 2 Value: 4
6  values_range = range(0,8,2) #range(startvalue,endvalue,interval)
7  for x in values_range:
8      print(f"Value: {x}")
9  #Output: Value: 0 Value: 2 Value: 4
10 objects = ["String1", 5, 5.5, "String2"] #different types possible
```

```
11  for typ in objects:
12      print(f"Content: {typ}, Data type: {type(typ)}")
13  #Output: Content: String1, Data type: <class 'str'>
14  #Content: 5, Data type: <class 'int'>
15  #Content: 5.5, Data type: <class 'float'>
16  #Content: String2, Data type: <class 'str'>
```

```
1  #Example of a while-loop
2  i = 1
3  sum = 0
4  noiterations = 15
5  while i <= noiterations:
6      sum = sum + i
7      i = i + 1
8  print("The calculated sum is", sum)
9  #Output: The calculated sum is 120
```

The definition of a function is instrumental in program code when individual program sections must be repeated. Furthermore, the use of functions also helps to achieve a good coding style. The definition of functions is introduced in Python with the keyword `def`, whereby a function does not need to take any arguments. The statements, which have to be executed by the function, are indented again. The function call is initiated by using the defined function accordingly by its name.

```
1  def myname(): # Function without an argument
2      print('My name is Max Mustermann.')
3  myname() # Only use name for call
4  #Output: My name is: Max Mustermann
5
6  def particularname(name = 'Max Mustermann'): # Function with an argument and
       Default
7      print(f"'My name is {name}")
8  particularname() # default name: Max Mustermann
9  particularname('Johnny Herbert')
10 particularname('Peter Pan')
11 #Output: My name is Max Mustermann.
12 #My name is Johnny Herbert.
13 #My name is Peter Pan.
14
15 def multiply_values(x,y,z):
16     return x*y*z
17 print(multiply_values(1,2,3))
18 #Output: 6
```

There are many functions that Python already has included in its large number of accessible libraries. They can be accessed as needed if the required libraries have been installed and also imported with an `import` statement within the program code.

For example, the basic mathematics library `math` can be used by applying the import command `import math as m`, whereby m represents an alias name for the corresponding library. Finally, the primary purpose of using an alias name is to reduce typing during programming. This procedure will be consistently utilized through this lab to include required data types, functions and classes from different libraries.

```
1  import math as m
2  m.sqrt(4) # Calculate square root of 4
3  #Output: 2.0
```

The `numpy` library provides the important `array` data type, which is particularly suitable for the representation of vectors and matrices. Furthermore, this library provides all necessary arithmetic operations that can be applied quickly and efficiently in machine learning. The appropriate library was already installed in subsection 1.1.1 in the virtual environment and can be used with the help of the import function. It has to be ensured that the python interpreter of the virtual environment is used. This can be identified in the upper corner of the current Juypter document. Otherwise the library is not available via `import` function!

Hint

The interpreter of a virtual environment can be added to VSCode by performing the following commands via terminal: `python -m pip install ipykernel` and `python -m ipykernel install -user -name=lab-env`. Afterwards the available interpreter of the environment `lab-env` could be added and selected via *View* → *Command Plate* (Ctrl+Shift+P) → *Python: Select Interpreter*.

The application of the `numpy` library is clarified by the following listing example. Furthermore, the library `Matplotlib` is used here for the graphic visualization.

```
# Lists as data type for the representation of vectors
# Generation of a Valuelist (Vector) by Corepython
# to represent a vector
values = [10, 20, 25, 30, 35, 40]

# add additional value to lisz
values.append(11)

# delete last entry within list
values.pop()

# Access to individual elements of List
print('values[0]: \t\t', values[0])

# Individual elements can be changed
values[1] = 15071988

# Slicing can be applied here as well, for
# example to display elements
print('values[1:3]: \t', values[1:3])
print('values[3:]: \t\t', values[3:])
# only every second entry
print('values[::2]: \t', values[::2])
# Rotate the list
print('values[::-1]: \t', values[::-1])
#*********************************************************************
#Import new lib
import numpy as np

#Convertion in numpy-array
myarray = np.array(values)
myarray = myarray*10
print(type(myarray))
print(myarray)

# Generate matrix within NumPy
A = np.array([ [1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(A)
```

```
39 # Creation of a special matrix with ones
40 B = np.ones((3,4))
41 print(B)
42 # Array operation elementwise subtraction
43 C = A-B
44 print(C)
45 # Transpose Matrix
46 D = C.T
47 print(D)
48 #Generate Matrix with random values
49 E = np.random.random((4,3))
50 E = E*3
51 F = np.matmul(A,E)
52 print(F)
53 #**************************************************************************
54 # Import matplotlib for visualization
55 import matplotlib.pyplot as plt
56 
57 # Make us of matplotlib
58 a = np.linspace(0, 8, 512)
59 b = np.exp(-a)
60 # Common Plot
61 plt.plot(a, b)
62 plt.show()
63 # Histogram
64 plt.hist(np.random.random(1024), bins=30, edgecolor='black')
65 plt.show()
66 # Scatter plot
67 plt.scatter(np.random.random(100), np.random.random(100), edgecolor='black')
68 plt.show()
```

1.3 Exercises

Task 1 - Put Programming Environment into Operation

First, the programming environment (Python and Visual Studio Code) must be put successfully into operation and checked for proper operation for the upcoming tasks.

In addition, create a new folder named `machine_learning_lab` in an arbitrary windows path and create within this folder a virtual Python environment called `lab_env`. Finally, activate this created virtual environment `lab_env` and check that it is also successfully activated in the terminal. Finally, install the required Python libs and check that the packages are successfully installed!

Task 2 - Familiarization with Programming Environment and Juypter Cells

Create a new Juypter notebook named `lab1_myfirstnotebook.ipynb` in Visual Studio Code and make yourself familiar with the given development environment. You could also adapt the layout of Visual Studio Code to your personal needs if required. Finally, try to rebuild the contents shown in Figure 1.3 to familiarize yourself with the primary cell types.

Task 3 - Basic Datatyps

a) First define a variable `a` and assign the value 15 to it. Write the data type that you expect behind this instruction as a comment. Afterwards output the variable `a`.

b) What is the data type of variable `a`? What result would you expect if you divide the variable by 5, especially if you think of other programming languages. Describe your assumption briefly.

c) Define a new variable `b` and assign the division from part b) to it accordingly. What result do you get, and can you explain it briefly by identifying the data type. Does the result correspond to your expectations?

d) Now assign the string `'This is text as a string'` to the already existing variable `a`. What result do you get, and can you explain it briefly by identifying the data type. Does the result correspond to your expectations? Finally, which advantages can be identified based on the given examples.

Task 4 - Additional Mathematical Operands

Execute the special operations `d = 10 // 4`, `e = -7 // 5`, and `f = 11 % 3` and identify which operations these are. Give a short explanation.

Task 5 - Boolean Operation

Execute the following instructions completely `x = 4 < 6`, `y = x + 2` and `y != 3` and give a short explanation about the final result.

Task 6 - String Data Typ - Slicing

a) Define the string `'This is the machine learning lab'` and output the individual words of the sentence using string slicing.

b) What happens when you access the string element at position 33 and at position -10. Briefly explain the behavior that occurs!

Task 7 - Control Structures

a) Output even numbers with `print()` from 0 to 8 inclusive using the `range()` function and a for-loop. Furthermore, try to stop the usual line break of print to output the numbers in one line!

b) Calculate the factorial value of 15 (15!), try to use a while loop for this and cause termination by an if condition at the end.

Task 8 - Functions

a) Define a new function `convertCelsiusToKelvin` that calculates the corresponding Kelvin value based on a given temperature value in degrees. The output should look like this: `0.0 řC = 273.15 K`.

b) Perform the calculation for the following temperatures -273.15 řC, 0 řC, 36 řC, 50 řC and 90 řC as efficiently as possible!

c) Import the math library into your program code and determine `sin(pi)` and `cos(pi/2)`. Do the results match your expectations? Briefly explain the given results or deviations.

Task 9 - Import and use Libraries `nunmy` and `matplot`

Familiarize yourself with the libraries `nunmy` and `matplot`. Therefore, make use of the given example listing from the lab describtion.

Task 10 - Simulation of a typical RC Circuit

Simulate the output voltage (*charging and discharging process*) of an RC circuit, where the applied DC voltage is 10 V and the time constant $\tau = 1\,\mathrm{s}$. The capacitor is completely discharged at $t \leq 0$! The voltage is applied to the circuit at $(t = 0)$ and disconnected at $t = 5 \cdot \tau$.
First start with the visualization of the charging process and then of the discharging process using `matplotlib`. Then try to combine both results as simply as possible and label them accordingly cf. Figure 1.4. There are several options possible for solving this task!

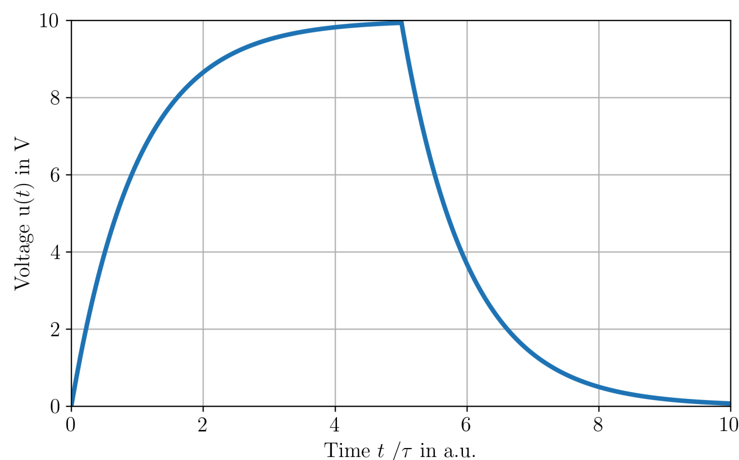


Figure 1.4: Typical voltage behavior of the output of an RC-System

Chapter 2

Data Sets

This chapter focuses on **working with data sets** in general. This includes reading data from files, transforming it, doing basic preprocessing operations on it and finally visualizing it. It also introduces important concepts such as data partitioning and confusion matrices.

2.1 **Basics** 23
2.1.1 Introduction 23
2.1.2 Confusion and Accuracy 24
2.1.3 Data Partitioning 25
2.1.4 Code Reference 26
2.2 **Lab Exercises** 29

2.1 Basics

2.1.1 Introduction

Take a look at this list of multiple cars from a card game:

Card index	Power	Top speed	Color	Price	Model
0	80 kW	180 km/h	Black	20 k	Sedan
1	140 kW	230 km/h	Silver	70 k	SUV
2	100 kW	240 km/h	Blue	40 k	Sedan
3	250 kW	300 km/h	Red	150 k	Roadster
4	90 kW	250 km/h	Green	60 k	Sedan
5	180 kW	210 km/h	Black	90 k	SUV
6	70 kW	190 km/h	White	30 k	Sedan

Each row represents a car (data point) with multiple features (power, top speed). The form of each feature can be quantitative (e.g., power) or qualitative (e.g., color, model) with various data

types and value ranges. While this chapter only uses artificially generated data, it is common to try new machine learning algorithms on open access (e.g., medical) databases.

We now consider a classifier algorithm to predict the correct model for each car presented to it based on power and top speed only. Therefore, we extract an $\boldsymbol{X}$ matrix containing both features for each car and a $\boldsymbol{y}$ vector with the corresponding model. The model acts as the label here and is represented by an integer value:

Index	Feature A	Feature B	Label
0	80	180	0
1	140	230	1
2	100	240	0
3	250	300	2
4	90	250	0
5	180	210	1
6	70	190	0

$$\boldsymbol{X} = \begin{bmatrix} 80 & 180 \\ 140 & 230 \\ 100 & 240 \\ 250 & 300 \\ 90 & 250 \\ 180 & 210 \\ 70 & 190 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 0, 1, 0, 2, 0, 1, 0 \end{bmatrix}^{\mathrm{T}}.$$

The classifier predicts the most likely class of the available discrete classes. When it comes to predicting a quantitative feature like the price, we might employ a regression algorithm instead. This would output a price estimation that is not limited to predefined discrete values.

2.1.2 Confusion Matrix and Derived Performance Metrics

The performance of an arbitrary classifier can be visualized in terms of a confusion matrix. For example, if we feed $\boldsymbol{X}$ into the classifier, we will yield a new vector $\hat{\boldsymbol{y}}$ with the corresponding predicted labels. We can now compare the true labels and the predicted ones. For simplicity, we assume 2 classes here with:

- Class A: Sedan, positive,
- Class B: Other, negative.

	Class A (pred.)	Class B (pred.)
Class A (true)	2 (29%, True positives)	1 (14%, False negatives)
Class B (true)	1 (14%, False positives)	3 (43%, True negatives)

The table above shows both the absolute frequencies as well as the more common relative frequencies (percentages). In this case, the relative frequencies are normalized to the total number of data points. If only two classes exist (e.g., COVID test) it is common practice to refer to the results with terms like `false positives`. The same is generally possible if we select one class and compare it to all other ones.

It is also possible to normalize for each row to get the share of the true labels that were predicted to be in a specific class.

	Class A (pred.)	Class B (pred.)
Class A (true)	2 (67%)	1 (33%)
Class B (true)	1 (25%)	3 (75%)

It is also possible to normalize each column to get the share of the predicted labels belonging to a specific true class.

	Class A (pred.)	Class B (pred.)
Class A (true)	2 (67%)	1 (25%)
Class B (true)	1 (33%)	3 (75%)

The main diagonal of the confusion matrix contains all labels that were correctly predicted. Summing its values up yields the accuracy score. In the example, 5 of all 7 predictions were correct (2 for class A and 3 for class B). This yields an accuracy score of 72%.

Precision (sensitivity) describes the ratio between samples that were correctly predicted to be in class A (true positives) and all samples predicted to be in class A (true and false positives). In the example, there are 2 samples correctly predicted to be in class A with 3 samples predicted to be in class A in total. This yields a precision score of $\frac{2}{3} = 66.7\%$.

Recall (specificity) describes the ratio between samples that were correctly predicted to be in class A (true positives) and all samples in class A (true positives and false negatives). In the example, there are 2 samples correctly predicted to be in class A with 3 samples in class A in total. This yields a recall score of $\frac{2}{3} = 66.7\%$.

The performance goals between precision and recall vary by application. In example, a COVID test must not produce many false negatives (infected people who are not detected) while false positives (non-infected people who are detected) are not as grave. So one would aim for a high precision.

2.1.3 Data Partitioning

We generally differentiate between `training` and `validation` or `test` data. Training data (features and the corresponding labels) are provided to a training routine. The algorithm will then learn to recognize underlying patterns between both. Validation means that we only feed the feature data into the algorithm and compare the predicted labels $\hat{y}$ to the true labels y.

Using training data for validation will usually (assuming an appropriate algorithm) yield a very good prediction of the labels. However, there might be overfitting, which means that we matched the algorithm so close to the training data that it performs significantly worse for slightly varying

input data as in an actual application. In consequence, one does partitioning into training and validation data. Therefore, the validation data must not be used for training and vice versa.

It is also essential to assure that all relevant relations between features and labels are contained in the training data. Otherwise, the algorithm is not able to learn them. Consequently, it is beneficial to randomly partition training and validation data from the overall data set.

K-fold cross-validation means that we (randomly) split the data into k equally sized subsets. We do k trainings with all subsets except for one as training data and the remaining one as validation data. The overall performance can then be assessed as the average error between all k validations.

2.1.4 Code Reference

Working with data sets requires the usage of multiple imported modules. This list is provided as an overview on the required modules for this chapter. Take a look at the specified functions and classes as a preparation for this lab.

Pandas

The quick starting guide "10 minutes to pandas" provides all the details needed for this lab. Take a closer look at `selection by label, viewing data, merging, plotting` and `getting data in/out (CSV)`.

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

Sklearn

The upcoming tasks require some very common methods from Sklearn. We will normalize data with `StandardScaler`, partition data with `train_test_split` and evaluate results with `confusion_matrix`, `ConfusionMatrixDisplay` and `accuracy_score`.

https://scikit-learn.org/stable/user_guide.html

Matplotlib

We need plots to visualize our data sets and classification results with regular and scatter plots. Take a look at both functions. A clear presentation of results is expected, so also take a look at subplots, titles and axes labeling.

https://matplotlib.org/stable/plot_types/index.html

NumPy

We also need various functions from NumPy, so have the user guide at hand.

https://numpy.org/doc/stable/user/index.html

Imports

Task 1

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
```

Task 2

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
    accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
import blackboxalgo as bb
```

2.2 Lab Exercises

Task 1 - Introducing pandas

In this exercise, we will apply some basic functionality of the pandas library to load, preprocess and visualize data sets.

a) Create a new notebook and setup the workspace by importing all required modules. Load the data frame from the file "df0.csv" into a corresponding variable of that name. Print `df0`.

b) Take a look at the `df0` output: What is a data point? What is a feature? How many of both do you find in the data frame? Take notes on the included features and their data types. What might be the purpose of the last labeled column?

c) Convert `df0` into a NumPy array (ndarray) `mat0` and print it again. Which information is lost in comparison to the original data frame? What about the data types?

d) We will now plot the first column of df0 as a (time) series. Access the first column of df0 by its name and call its plot method. Note that this is only possible for data frames. Plot the same data using mat0 separately as described in the python introduction. Use appropriate axis labels.

e) It is common for machine learning algorithms to require normalized input data with zero-mean and a standard deviation of 1. This normalization step requires stochastic parameters of the data set. Access the first column of `df0` and assign it to a new variable `df0_a`. Obtain mean value, max value, min value and standard deviation of `df0_a` with the corresponding NumPy functions.

f) It is also possible to get the stochastic parameters of `df0_a` directly from pandas. Store the output of the corresponding function in a variable `std_det` and normalize `df0_a` by subtracting the mean and dividing the rest by the standard deviation, resulting in a z-score normalization. Plot the unaltered as well as the normalized data series in a combined plot.

g) The StandardScaler class from sklearn offers a simpler way to normalize data. Initialize a StandardScaler object called scaler and use the `fit_transform` function to normalize `df0`, resulting in `df1`. Plot both the unaltered and the normalized version.

h) We will now visualize `df1` as a scatter plot. Set the first column of `df1` as the x value, the second one as the y value, and the third one as the color sequence. Use colormap "Set1" and vmax=9. What do the different colors mean in this context? Explain (the matplotlib docu might be helpful): What do the X and the Y values mean here? Clarify by labeling them. Why were the color parameters chosen in this way?

i) Read the data frame `df2` from the corresponding CSV file and concatenate it with `df0` to obtain the new data frame `df3`. Create a scatter plot of `df3`. Shortly describe the differences between `df0` and `df3` based on the scatter plots.

Task 2 - A blackbox classifier algorithm

In this exercise, we will train and validate a machine learning algorithm and learn tools to rate its performance.

a) Create a new notebook and setup the workspace by importing all required modules. Load the data frame from the file "df4.csv" into a corresponding variable of that name.

b) Create 2 NumPy arrays out of `df4`. NumPy array `X` shall contain the data of feature A and B, while `y` contains only C (classes). Create a scatter plot to visualize the data.

c) We will now feed the whole data frame as training data into an unknown (blackbox) classification algorithm. There is no normalization required here. Start the training by initiating an `algo` object of class BlackBoxAlgo by passing the data `X` and the labels `y`. Now validate the performance of the algorithm by calling its validate function. Pass `X` as validation data and store the result in `y_pred`. Create a scatter plot of `X`, but this time use `y_pred` as the color sequence. Compare the result to the previous plot. How would you rate the performance from a subjective point of view?

d) We will now take a look at a more objective performance metric. Compute the confusion matrix `cm`. Use ConfusionMatrixDisplay to visualize it. Explain each value in the matrix regarding the true and the predicted class. This confusion matrix uses absolute frequencies - name a disadvantage of this metric.

e) There are multiple ways to normalize the confusion matrix to obtain relative frequencies. Compute and plot all three. Use the corresponding normalized confusion matrix to answer the following questions:

   (i) Which share of all data points is in class B, but was falsely attributed to class A?

   (ii) Which share of all data points in class B was falsely attributed to class C?

   (iii) Which share of all data points attributed to class B was correctly predicted?

f) Accuracy, precision and recall score are additional metrics for the performance of the classifier. Compute all three metrics! Use the parameter `average=None` for both recall and precision. What do the results mean?

g) Load the data frame `df5` from the corresponding CSV file. Extract the arrays `X5` and `y5` and feed `X5` into the validation function of the blackbox algorithm to obtain `y_pred5`. Create a scatter plot of both the initial data and the prediction. Create a confusion matrix normalized to all. Compute the accuracy. Compare the results to e) and explain the differences in performance.

h) Create a concatenated data frame `df6` out of `df4` and `df5`. Extract `X6` and `y6`. Now use the train_test_split function from sklearn to partition the data randomly into two different data sets. Use a test size of 0.5 and a random state of 0. Use the train data to train the blackbox algorithm and the test data for validation. Produce both scatter plots, confusion matrix and accuracy. Explain!

Part II
Data Preprocessing

Lab *Machine Learning*

Chapter 3

Linear Discriminant Analysis

This chapter focuses on the linear transformation of data into a more compressed and discriminable form. Such transformations are given by the Principle Component Analysis and the Linear Discriminant Analysis. Both analyses are introduced in this chapter and by means of exercises their properties and capabilities are discussed.

3.1 **Basics** 33
- 3.1.1 Principle Components 33
- 3.1.2 Linear Discriminants 35
- 3.1.3 Scikit-learn Classes 36

3.2 **Exercise** 36

3.1 Basics

Commonly, classification problems take place in a multidimensional feature space. Classification models rely on this feature space to provide a high class separability at low feature space dimensionality. Consequently when designing a feature space, the discriminable information per dimension should be maximized. The methods discussed in this chapter transform an available feature space for space compression, easier class separation or both.

3.1.1 Principle Component Analysis

The Principle Component Analysis (PCA) aims at identifying the directions in the feature space which provide the most information in terms of feature variance. Notable references for this method are [1, 2, 3]. The PCA is computed on a dataset in which all classes are contained but no class distinction is introduced in the analysis. If Fig. 3.1 is considered as an example dataset in a 2D feature space, then the PCA should identify the two marked directions as directions with maximum variance. The transformation matrix obtained from a PCA then transforms the feature space in such a way that the directions of maximum variance in the dataset align with the coordinate axes. After the transformation, the dimensions no longer correspond to individual features but to a linear combination of all features. The obtained dimensions are refered to as principle components.

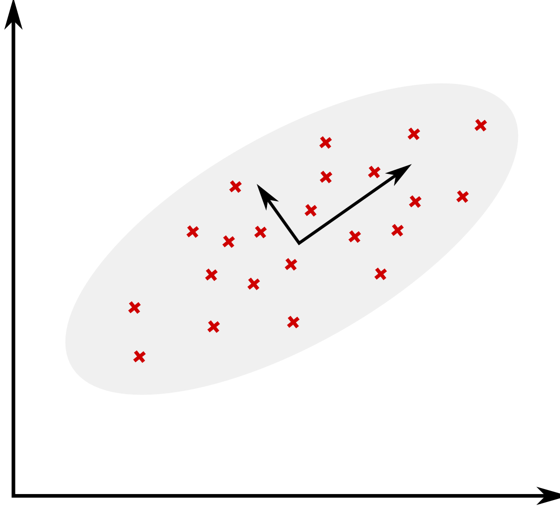


Figure 3.1: Example dataset where the small axis indicates what directions hold maximum variance.

If $\boldsymbol{X}$ is a $K \times N$ feature matrix where K is the number of features and N the number of data samples, then $\boldsymbol{W}$ is a $K \times K$ transformation matrix to obtain the data points in the principle component domain:

$$\boldsymbol{Y} = \boldsymbol{W}^{\mathrm{T}} \boldsymbol{X}. \tag{3.1}$$

The transformation matrix is concatenated from the eigenvectors $\boldsymbol{v}_k$ of a datatset's covariance matrix estimate. The magnitude of an eigenvector's corresponding eigenvalue λ_k is an indicator of the data's variance in the given principle component. Therefore, the eigenvectors are sorted in $\boldsymbol{W}$ such that the magnitude of the corresponding eigenvalues decrease monotonically from left to right:

$$\boldsymbol{W} = \left[\boldsymbol{v}_0, \boldsymbol{v}_1, ..., \boldsymbol{v}_k, ..., \boldsymbol{v}_{K-1}\right], \quad \lambda_k > \lambda_{k+1}. \tag{3.2}$$

If a dimension reduction is desired, the $R < K$ eigenvectors with the largest eigenvalues can be selected to form a transformation matrix $\boldsymbol{W}_{\text{trunc}}$ of dimensions $K \times R$. Thus, the last $K - R$ columns of the original transformation matrix are truncated:

$$\boldsymbol{W}_{\text{trunc}} = \left[\boldsymbol{v}_0, \boldsymbol{v}_1, ..., \boldsymbol{v}_k, ..., \boldsymbol{v}_{R-1}\right]. \tag{3.3}$$

The value of R can be a requirement in a given application or the value can be manually set by analytically looking at the data. A heuristic approach is to define a threshold for how lossy the compression is allowed to be. Remembering that eigenvalues are indicators of the data's variance in the principle components, a relative information content γ_k for each principle component can be defined as

$$\gamma_k = \frac{\lambda_k}{\sum\limits_{i=0}^{K-1} \lambda_i}. \tag{3.4}$$

The number of dimensions R can, hereby, be found as the smallest integer which satisfies

$$\sum_{k=0}^{R-1} \gamma_k \geq \Gamma, \tag{3.5}$$

where $\Gamma \in (0, 1]$ is a chosen threshold. It can, thus, be said that the compression retains at least $100 \cdot \Gamma$ percent of the original information content. However, since no class distinction was introduced for the computation of the PCA transformation matrix, no general statement can be given about the transformation's impact on class separability.

PCA Step-by-Step

1. Estimate the covariance matrix of the dataset.
2. On this matrix, compute the eigenvalues and the eigenvector matrix.
3. Rearrange the columns of the eigenvector matrix such that (3.2) is satisfied.
4. If applicable, truncate the eigenvector matrix according to (3.3).
5. Apply transformation matrix to data.

3.1.2 Linear Discriminant Analysis

Although the Principle Component Analysis is generally a good pre-processing tool, it is not the best method if the main focus is to be on an improved or maintained class separability. In this case, the Linear Discrimant Analysis (LDA) may be applied as it aims at minimizing the within-class scatter while also maximazing the so-called between-class scatter. This optimization criterion can be expressed as a matrix on which an analysis similar to the PCA is performed. Notable references for the LDA are [4, 5].

The within-class scatter $\mathbf{S}_{\mathrm{w}}$ can be understood as a (scaled) arithmetic mean of the covariance matrices of the individual classes. To mathematically define the $K \times K$ matrix $\boldsymbol{S}_{\mathrm{w}}$, the class scatters $\boldsymbol{S}_{\mathrm{w,l}}$ with $l \in \{0, 1, ..., L-1\}$, where L is the number of classes, are defined first:

$$\boldsymbol{S}_{\mathrm{w},l} = \sum_{\boldsymbol{x} \in C_l} (\boldsymbol{x} - \boldsymbol{m}_l)(\boldsymbol{x} - \boldsymbol{m}_l)^{\mathrm{T}}. \tag{3.6}$$

Here, C_l is the set of all data vectors in the l-th class and $\mathbf{m}_l$ is their corresponding mean vector. To obtain the within-class scatter matrix

$$\boldsymbol{S}_{\mathrm{w}} = \sum_{l=0}^{L-1} \boldsymbol{S}_{\mathrm{w},l}, \tag{3.7}$$

the individual class scatters are added. The computation of the between-class scatter matrix $\boldsymbol{S}_{\mathrm{b}}$, additionally, utilizes the global mean vector $\boldsymbol{m}$ of all available data vectors. Considering $|C_l|$ as the amount of data (cardinality) in each class, the between-class scatter is defined as

$$\boldsymbol{S}_{\mathrm{b}} = \sum_{l=0}^{L-1} |C_l| \, (\boldsymbol{m}_l - \boldsymbol{m})(\boldsymbol{m}_l - \boldsymbol{m})^{\mathrm{T}}. \tag{3.8}$$

The combination $\boldsymbol{S}_\mathrm{w}^{-1}\boldsymbol{S}_\mathrm{b}$ of the computed matrices is the basis for an eigenvector analysis as seen in the PCA. By sorting the obtained eigenvectors according to (3.2), the LDA transformation matrix of size $K \times P$ is created. It should be noted that $P < K$ holds, as at least one eigenvalue will be zero in the case of LDA. For a further data compression in terms of dimension reduction, the transformation matrix may be truncated to R columns by considering (3.5). The resulting dimensions are refered to as linear discriminants.

LDA Step-by-Step

1. Estimate the within-class scatter matrix $\boldsymbol{S}_\mathrm{w}$.
2. Estimate the between-class scatter matrix $\boldsymbol{S}_\mathrm{b}$.
3. Compute the eigenvalues and the eigenvector matrix of $\boldsymbol{S}_\mathrm{w}^{-1}\mathbf{S}_\mathrm{b}$.
4. Rearrange the columns of the eigenvector matrix such that (3.2) is satisfied.
5. If applicable, truncate the eigenvector matrix according to (3.3).
6. Apply transformation matrix to data.

3.1.3 Scikit-learn Classes

While both PCA and LDA can be computed in Python using Numpy functions only, it is easier to rely on libraries such as Scikit-learn for the direct application of PCA and LDA. The algorithms are provided as classes with a number of methods and attributes. An outline of the basic workflow with such a class is given below. For more information please refer to the online documentation of Scikit-learn.

```
# Imports
from sklearn.decomposition import PCA

# Create Object
pca_object = PCA()

# Compute parameters/transformation
pca_object.fit(X)

# Apply transformation
Y = pca_object.transform(X)
```

3.2 Exercise

There are three different exercises. By working with an artificial data set, you will first implement both PCA and LDA using Numpy functions. In a next step you will find out how to replace some of your Numpy code with the Scikit-learn classes. In the last task you will look at some real-world higher dimensional data.

Task 1 - LDA Basics

a) Start by importing Numpy, Pandas and Pyplot into a notebook.

b) Using Pandas, read in the file **lda_data.csv** as a dataframe.

c) Extract the features into a 2D float array. Extract the class labels into a 1D integer array.

d) Visualize the data in form of a scatter plot. Assign each data point a color to show its class affiliation.

e) For both features individually, estimate the feature distributions by plotting histograms (pyplot.hist) of both classes into the same figure. Adjust the histogram parameters to obtain a meaninful visualization. Tip: Use the **alpha** parameter.

f) What can you say about the separability of the two classes?

g) Compute the covariance matrix of the entire dataset.

h) Compute the eigenvector matrix of the covariance matrix.

i) Sort the eigenvectors in such a way, that their corresponding eigenvalues are sorted in decreasing order.

j) Use the rearranged eigenvector matrix to transform the features into principle components. Plot the results with class labels.

k) Repeat e) for the features in the principle component domain.

l) What effect does the transformation have? What can you say about the separability of the classes in the principle component domain?

m) What effect does a compression into the first principle component have?

n) Compute the within-class scatter matrix $\boldsymbol{S}_{\mathrm{w}}$.

o) Compute the between-class scatter matrix $\boldsymbol{S}_{\mathrm{b}}$.

p) Compute the eigenvector matrix of $\boldsymbol{S}_{\mathrm{w}}^{-1}\boldsymbol{S}_{\mathrm{b}}$. Are the eigenvectors influenced by the multiplicative bias of your scatter computations?

q) Use the eigenvector corresponding to the largest eigenvalue to perform a LDA data compression. Plot the results.

r) Repeat e) for the LDA compressed data.

s) What can you say about the separability of the classes in the LDA-compressed domain? How does this compare to the principle component domain?

Task 2 - Scikit-learn

Repeat Task 1. Shorten your code by using the PCA/LDA classes from Scikit-learn.

Task 3 - Real-world Data

a) Open the notebook **lda_iris.ipynb**. Here, the Iris dataset has already been imported for you. Refer to the scikit-learn online documentation to see what this dataset is about.

b) Plot the class histograms in all 4 feature dimensions. What do you observe? Tip: You can either use pyplot functions as in task 1 or the pairplot function from the seaborn library for more convenience.

c) Create an LDA object and fit it to the data.

d) Call on the class attribute **explained_variance_ratio_**. What does the output tell you? What is your suggestion for the size of the transformation matrix?

e) Create two LDA objects where you specify that you want to compress the data either into 1 or into 2 dimensions. Then fit the objects to your data.

f) Transform the data with the two LDA objects separately and plot the results. What do you observe? Do your plots agree with your interpretation of the **explained_variance_ratio_** attribute?

Chapter 4

Independent Component Analysis

Independent component analysis is a means of transforming data into a number of **independent** variables, or **components**. The goal is not so much to reduce the dimensionality, but rather to uncover more meaningful variables.

4.1 **Basics** 39
4.1.1 Principles of ICA 40
4.1.2 Outline of FastICA 41
4.1.3 Scikit-learn Classes 44
4.2 **Exercise** 44

4.1 Basics

Measurements often do not record only what they are intended to measure. Obviously, measurements are corrupted by noise, but that is not the only part of the story. Many times there are also signals from other, identifiable sources [6]. In general, a measurement will be a combination of many distinct sources. The broad topic of separating measurements into their underlying sources is called *blind source separation*. Independent Component Analysis (ICA) is a method for performing blind source separation, and probably the most widely used [7].

A good and often used example task for blind source separation is the magnetic heart measurement of a pregnant woman as schematically shown in Fig. 4.1. In addition to the measurement of the mother's heart signal, the heart signal of the fetus/feti is of course also measured by the magnetic sensors. Thus, a mixture of different heart signals is picked up by the sensors and this mixture usually cannot be disentangled by the naked eye. An ICA algorithm can now be used for this - this is to extract the source signals (heart signals) from the measured mixture.

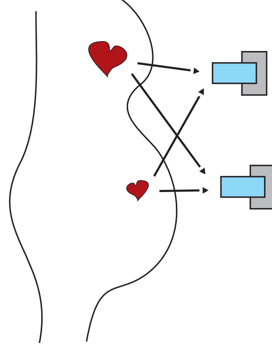


Figure 4.1: Schematic illustration of a magnetic measurement of a pregnant woman by two magnetic sensors. Each sensor records the heart signal of the mother as well as the heart signal of the fetus. The problem is how to identify the individual heart signals in the mixture of the signals.

4.1.1 Principles of ICA

The basis for ICA estimation is a generative model that describes how the M mixed signals are generated as a linear combination of the K source signals:

$$x_i(n) = a_{i1}s_1(n) + a_{i2}s_2(n) + \cdots + a_{iK}s_K(n) = \sum_{j=1}^{K} a_{ij}s_j(n) \qquad \forall i \in \{1, \cdots, M\} \tag{4.1}$$

where $x_i(n)$ are the observed signals obtained from sources $s_j(n)$ [7, 8]. We can write this in matrix-vector notation [7, 8] as:

$$\begin{bmatrix} x_1(n) \\ x_2(n) \\ \vdots \\ x_M(n) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1K} \\ a_{21} & a_{22} & \cdots & a_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MK} \end{bmatrix} \begin{bmatrix} s_1(n) \\ s_2(n) \\ \vdots \\ s_K(n) \end{bmatrix} \tag{4.2}$$

$$\boldsymbol{x}(n) = \boldsymbol{A}\,\boldsymbol{s}(n). \tag{4.3}$$

Note that it could be that $M > K$, i.e., there are more observed signals than sources. If this is the case, then you can and should use Principal Component Analysis (PCA) to reduce the number of signals [8].

The goal of ICA is to solve for the mixing matrix $\boldsymbol{A}$, such that the independent components, $\boldsymbol{s}(n)$, can be obtained, via:

$$\begin{aligned} \boldsymbol{s}(n) &= \boldsymbol{A}^{-1}\,\boldsymbol{x}(n) \\ &= \boldsymbol{W}\,\boldsymbol{x}(n). \end{aligned} \tag{4.4}$$

In order to estimate the mixing matrix, $\boldsymbol{A}$, we need to make two key assumptions:

(a) **Independence.** The components $s_j(n)$ should be statistically independent. Roughly speaking, two random variables are said to be independent if information on the value of one does

not give away any information on the value of the other. This means more concretely that the joint probability density of the two signals is equal to the product of both densities.

(b) **Non-Gaussianity.** The components $s_j(n)$ must have a non-Gaussian distribution. That is because a linear combination of two sources from Gaussian distributions would result in mixture signals that are Gaussian as well, hence we have no information on the directions of the columns of the mixing matrix $\boldsymbol{A}$ [6, 7] [1].

As a matter of fact, non-Gaussianity is actually the key to deriving the independent source signals, that is, if they follow the ICA model (Eq. (4.1)). To see why, let us consider the case where we only want to find one of the independent components, denoted by $y(n)$ [7]. As per our general assumption, $y(n)$ is a linear combination of the mixture variables: $y(n) = \boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x}(n)$. If $\boldsymbol{w}$ was one of the rows of $\boldsymbol{A}^{-1}$, then $y(n)$ would be exactly equal to the independent component. But how do we get there? How can we determine $\boldsymbol{w}$ such that it would equal one of the rows in $\boldsymbol{A}^{-1}$? We can make use of the Central Limit Theorem that tells us that the distribution of a sum of independent variables tends towards a Gaussian distribution in the limit where the number of independent variables approaches infinity [7]. Let us therefore make a change of variable: $\boldsymbol{z} = \boldsymbol{A}^{\mathrm{T}}\,\boldsymbol{w}$. We now have:

$$
\begin{aligned}
y(n) &= \boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x}(n) \\
&= \boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{A}\,\boldsymbol{s}(n) \\
&= \boldsymbol{z}^{\mathrm{T}}\,\boldsymbol{s}(n).
\end{aligned}
$$

We see that $y(n)$ is a linear combination of the $s_i(n)$'s with weights given by z_i. Since a sum of even two independent variables is more Gaussian than the original variables, $\boldsymbol{z}^{\mathrm{T}}\,\boldsymbol{s}(n)$ is more Gaussian than any of the individual $s_i(n)$'s, and is least Gaussian when it only contains exactly one $s_i(n)$. In the latter case, obviously only one of the z_i's is non-zero. Therefore, we could take as $\boldsymbol{w}$ the vector that maximizes the non-Gaussianity of $\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x}(n)$ [7].

4.1.2 Outline of FastICA

In the previous sections, we developed an intuition for when ICA could be used, and in a general sense, how we would approach this kind of problem. In this section, we will focus on how this translates into (pseudo-)code for an algorithm that solves it for us. More specifically, we will dive into the algorithm that is known as FastICA [9].

4.1.2.1 Preprocessing

Centering. The most basic and necessary preprocessing is to center the data [7]. This basically means that we subtract the mean from each variable

$$
\boldsymbol{x}_{\mathrm{c}}(n) = \boldsymbol{x}(n) - \frac{1}{N}\sum_{k=0}^{N-1}\boldsymbol{x}(k), \tag{4.5}
$$

with N being the number of considered samples of the signal.

Whitening. Another useful preprocessing step is to whiten the observed data [7]. In this step, the data is transformed linearly such that is becomes *white*, i.e., its components are uncorrelated and

[1] However, it is allowed that one component has a Gaussian distribution [7]

have unit variance [6, 7]. A popular way of whitening uses the eigenvalue decomposition of the covariance matrix [7]:

$$\begin{aligned} \boldsymbol{C}_{xx} &= \frac{1}{N-1}\boldsymbol{X}_{\mathrm{c}}\boldsymbol{X}_{\mathrm{c}}^{\boldsymbol{T}} \\ &= \boldsymbol{E}\boldsymbol{D}\boldsymbol{E}^{\mathrm{T}}, \end{aligned} \tag{4.6}$$

with $\boldsymbol{X}_{\mathrm{c}} = [\boldsymbol{x}_{\mathrm{c}}(0), \boldsymbol{x}_{\mathrm{c}}(1), \ldots, \boldsymbol{x}_{\mathrm{c}}(N-1)]$ and $\boldsymbol{C}_{xx}$ being the covariance matrix of the mixed input signals. The matrix $\boldsymbol{E}$ contains the eigenvectors of $\boldsymbol{C}_{xx}$ and the matrix $\boldsymbol{D}$ is a diagonal matrix containing the eigenvalues [7]. We can then obtain the whitened data using:

$$\boldsymbol{x}_{\mathrm{w}}(n) = \boldsymbol{E}\boldsymbol{D}^{-1/2}\boldsymbol{E}^{\mathrm{T}}\boldsymbol{x}_{\mathrm{c}}(n). \tag{4.7}$$

To anticipate the following block, we define

$$\boldsymbol{X}_{\mathrm{w}} = \Big[\boldsymbol{x}_{\mathrm{w}}(0),\, \boldsymbol{x}_{\mathrm{w}}(1),\, \ldots, \boldsymbol{x}_{\mathrm{w}}(N-1)\Big]. \tag{4.8}$$

4.1.2.2 Main Loop

Now, for estimating the weights (or unmixing) matrix $\boldsymbol{W}$ we go through the following main loop:

Let us walk through this block of pseudocode (Alg. 1). For each of the independent components, s_j, $j = 1, 2, \ldots, K$, we need to find a weight vector, and we will do this sequentially. For the jth component, we take as an initial guess a random weight vector. Then we iteratively improve by maximizing the negentropy $J(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}})$. In short, the negentropy is a measure for non-Gaussianity [7]. The higher its value, the less $\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}}$ looks like a Gaussian. Exactly computing the negentropy is difficult, however, we can approximate it with:

$$J(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}}) \propto \left[\mathrm{E}\left\{G(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}})\right\} - \mathrm{E}\left\{G(\boldsymbol{\nu})\right\}\right]^2, \tag{4.9}$$

where $G(\bullet)$ is a non-quadratic function, and $\boldsymbol{\nu}$ is a Gaussian random variable with zero mean and unit variance. Some good choices for $G(\bullet)$ are [7]:

$$G(u) = \frac{1}{\alpha}\log\left(\cosh\left(\alpha u\right)\right) \quad \text{or} \quad G(u) = -\exp\left(-u^2/2\right). \tag{4.10}$$

We want to maximize the negentropy $J(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}})$ (Eq. 4.9) for $\boldsymbol{w}$, and without going into details it turns out that we get as a solution [7, 10]:

$$\begin{aligned} &\max_{\boldsymbol{w}} J(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}}) \\ &\text{subject to } \|\boldsymbol{w}\|^2 = 1 \end{aligned} \tag{4.11}$$

$$\Rightarrow \boldsymbol{w} = \mathrm{E}\left\{g(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}})\boldsymbol{X}_{\mathrm{w}}\right\} - \mathrm{E}\left\{g'(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{X}_{\mathrm{w}}\right\}\boldsymbol{w}. \tag{4.12}$$

Here $g(\bullet)$ describes the derivative of the function $G(\bullet)$. This leads to [10]:

$$g(u) = \tanh\left(\alpha\, u\right) \quad \text{and} \quad g'(u) = \alpha\left(1 - \tanh^2\left(\alpha\, u\right)\right) \tag{4.13}$$

or

$$g(u) = u\,\exp\left(-u^2/2\right) \quad \text{and} \quad g'(u) = \left(1-u^2\right)\exp\left(-u^2/2\right). \tag{4.14}$$

Algorithm 1 FastICA for several units (see: [7])

1: **for** $j = 1, 2, \dots, K$ **do**
2: Choose an initial (e.g. random) weight vector $\boldsymbol{w}_j$
3: **for** $i = 1, 2, \dots, i_{\max}$ **do**
4: // Compute the weight vector $\boldsymbol{w}_j^+$ that maximizes the negentropy $J(\boldsymbol{w}_j^{\mathrm{T}} \boldsymbol{X}_{\mathrm{w}})$
5: $\boldsymbol{w}_j^+ = \frac{1}{N} \boldsymbol{X}_{\mathrm{w}}\, g(\boldsymbol{w}_j^{\mathrm{T}} \boldsymbol{X}_{\mathrm{w}})^{\mathrm{T}} - \frac{1}{N}\, g'(\boldsymbol{w}_j^{\mathrm{T}} \boldsymbol{X}_{\mathrm{w}}) \boldsymbol{1}_N\, \boldsymbol{w}_j$
6:
7: // Normalize $\boldsymbol{w}_j^+$, i.e.
8: $\boldsymbol{w}_j^+ = \boldsymbol{w}_j^+ / \|\boldsymbol{w}_j^+\|$
9:
10: // Decorrelate $\boldsymbol{w}_j^+$ with each of the previously computed weight vectors
11: $\boldsymbol{w}_j^+ = \boldsymbol{w}_j^+ - \sum_{l=1}^{j-1} \boldsymbol{w}_j^{+\mathrm{T}} \boldsymbol{w}_l \boldsymbol{w}_l$
12:
13: // Calculate how close $\boldsymbol{w}_j^{+\mathrm{T}} \boldsymbol{w}_j$ is to 1, i.e. δ
14: $\delta = |\left(|\boldsymbol{w}_j^{+\mathrm{T}} \boldsymbol{w}_j|\right) - 1|$
15:
16: // Update the weight vector, i.e.
17: $\boldsymbol{w}_j = \boldsymbol{w}_j^+$
18:
19: // If we are sufficiently close to a unit vector
20: **if** $\delta <$ tolerance **then**
21: // Continue with next independent component
22: break
23: **end if**
24: **end for**
25: // Store the weight vector in the weight matrix, i.e.
26: $\boldsymbol{W}[j, :] = \boldsymbol{w}_j$
27: **end for**

Once we have found the weight vector, we normalize, and then decorrelate it with each of the previously calculated weight vectors $\boldsymbol{w}_l$, $l = 1, 2, ..., j-1$. This prevents that we find the same weight vector twice. If the weight vector is not converged (i.e. the new values for $\boldsymbol{w}_j$ do not point in the same direction as the previous ones [7]) we continue with maximizing the negentropy. This is repeated a maximum of $i_{\max}$ times. The estimated source signals can be reconstructed by

$$\hat{\boldsymbol{s}}(n) = \boldsymbol{W}\boldsymbol{x}(n) \tag{4.15}$$

ICA Step-by-Step

1. Preprocess the data by centering and whitening using Eq. (4.5) and (4.7).
2. Compute the unmixing matrix $\boldsymbol{W}$ by using the Algorithm 1.
3. Apply unmixing matrix to data as in (4.15).

4.1.3 Scikit-learn Classes

While Fast ICA and the necessary preprocessing steps can be computed in Python using Numpy functions only, it is easier to rely on libraries such as Scikit-learn for the direct application of Fast ICA. The mode of operation here is similar to the one described in chapter 3.1.3. Again, for more information please refer to the online documentation of Scikit-learn.

```
# Imports
from sklearn.decomposition import FastICA

# Create Object
ica_object = FastICA(n_components = )

# Apply transformation
Y = ica_object.fit_transform(X)
```

4.2 Exercise

There are three different tasks. In the first task you have to program the preprocessing steps as well as the FastICA function for separating mixed signals using Numpy functions and apply them to given signals. In the second task, you will use Scikit-learn classes to unmix the same signals. The third task deals with 2D data clouds, where you have to explain the difference between ICA and PCA methods.

Task 1 - Fast ICA Basics

a) Open the notebook **ica_exercise_1.ipynb**. The needed packages are already imported.

b) Write a function that centers the data as described by (4.5).

c) Write a function to whiten the data as described by (4.7).

d) Write a preprocessing function including the centering and whitening functions.

e) Define the functions used to approximate the negentropy as defined in (4.14).

f) Write the FastICA function as defined in Alg. 1. The function shall thereby output the estimated source signals $\hat{\boldsymbol{s}}(n)$.

In your notebook three signals (stored in the variable S) have already been generated, which have been mixed to the signals in X via the mixing matrix A.

g) Plot the original signals stored in S as well as the mixed signals stored in X.

h) Center and whiten the data using the preprocessing funtion generated in d). Check if the whitening was successful by printing the covariance matrix of your data.

i) Estimate the original signals from your mixed (and preprocessed) data using your FastICA function.

j) Plot the estimated signals and compare them with the original signals. What are the differences between the original signals and the estimated signals?

Task 2 - Scikit-learn

From now on we want to use the Fast ICA class from Scikit-learn.

a) Open the notebook **ica_exercise_2.ipynb**. Import the FastICA class of Scikit-learn. All other needed packages are already imported.

b) Use the FastICA function to separate the mixed signals.

c) Plot the original, the mixed and estimated source signals.

Task 3 - 2D data clouds

a) Open the notebook **ica_exercise_3.ipynb**. Import Numpy, Pyplot, Pandas, and the FastICA and PCA classes of Scikit-learn.

b) Using Pandas, read the file **ica_data.csv** as a dataframe.

c) Visualize the data in form of a scatter plot.

d) Apply the FastICA algorithm to the data and plot the result in form of a scatter plot.

e) Apply the PCA algorithm to the data and plot the result in form of a scatter plot.

f) Where are the differences between PCA and ICA? Load the data **ica_data_original.csv** to compare the results with the original (unprocessed) data.

Part III
Unsupervised Learning

Lab *Machine Learning*

Chapter 5

Autoencoders

First, a brief introduction to Tensorflow is provided. Frequently used structures and basic API commands are illustrated. Afterwards, the basics of autoencoders are taught and deepened by means of an example. The example is based on efficient encoding of one-hot vectors. Subsequently, the laboratory exercise is presented, which is based on the less well-known but informative Fashion-MNIST.

5.1 **Tensorflow Basics** 49
5.2 **Autoencoder Basics** 51
5.2.1 Encoder 51
5.2.2 Decoder 51
5.2.3 Mathematical Structure 52
5.3 **Simple Autoencoder Example** 53
5.3.1 Problem 53
5.3.2 Dataset 53
5.3.3 Code 54
5.4 **Exercise** 55

5.1 Tensorflow Basics

A brief introduction to Tensorflow is provided in the following sections. [11]. There are some good tutorials on Tensorflow available [12] and especially this one should be completed before starting with the exercise: `https://www.tensorflow.org/tutorials/quickstart/beginner/`

Tensorflow itself is an open-source platform for machine learning. It provides a rich set of tools, libraries, and resources that are necessary to perform machine learning tasks. Sure, there are other alternatives like MATLAB or (Py)Torch, but we'll be focusing on Tensorflow in this lab. The name Tensorflow comes from mathematical operations on certain vector fields, which can be expressed as tensors (algebraic objects with multilinear relationships). Keras, on the other hand, is an API to Tensorflow that provides a more consistent experience for the user and was once used for many machine-learning backends. Since version 2.4, there is only the Tensorflow backend left.

5.1.1 Important Commands in Tensorflow

When you start using neural networks, the most important commands are the creation of models, adding layers, defining the structure and training the resulting network. We will cover these operations here briefly.

5.1.2 Model Creation and Layers Addition

The easiest way to create a model is by using the Keras sequential function, which allows you to create stacked layers:

```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(128, activation='relu')
])
```

As you can see, we created a model with an input layer that flattens an input (2D to 1D conversion) and uses a fully connected layer afterwards. The activation function is 'relu', which is the abbreviation for a rectified linear unit. A large number of different activation functions can be found in the Tensorflow documentation. You can simply add layers to your sequential model by inserting another layer:

```
model = tf.keras.models.Sequential([
tf.keras.layers.Flatten(input_shape=(28, 28)),
tf.keras.layers.Dense(128, activation='relu'),
tf.keras.layers.Dense(128, activation='tanh')
])
```

The new layer can have a different activation function(tanh, a hyperbolic tangent activation function in this example). You can find suitable activation functions here: [13]. Additionally, the different layers can be given names.

5.1.3 Network Training

In order to train the network, it has to be compiled first. Clever inheritance lets the user access methods on the created models. The first method to call is `compile`. Here, the user can set the optimizer for the network and the loss-function. There are many loss functions available, so feel free to check the documentation for details. Additionally, the user can set the desired metrics, e.g. accuracy. Here, we will use the `'adam'` optimizer and the mean-squared-error as a loss measure `'mse'`.

```
model.compile(optimizer='adam', loss='mse',...)
```

After compiling, the model needs to be fitted to the provided data. The inherited method `fit` provides this functionality. It takes the input data, the desired output, and some training parameters as input.

```
model.fit(inputs, outputs, epochs,...)
```

5.1.4 Data Prediction

To process data with the model, the `predict` method can be used. Keep in mind that the model must be trained before anything meaningful can be predicted. The syntax is as follows:

```
output = model.predict(input_data)
```

For example, class labels for the input data can be found in the `output`.

5.1.5 Subclassing

The Keras Subclassing API [14] is another approach to build a model, which is very handy when your models become more complex. This enables you to write custom functions for your model (e.g., special structures, custom training, custom cost functions) while also utilizing Tensorflow's functions. It works by employing the concept of super classes. In Python, a Subclass can inherit from multiple Superclasses, but don't worry, we won't go into detail here. Classes are required for the exercise, but a class skeleton will be provided.

5.2 Basics

An autoencoder is a type of neural network that tries to compress the input data and reconstruct it correctly using the compressed data in the output(unsupervised learning scheme that learns efficient codings of unlabeled data). The compression and reconstruction of the input data run consecutively in the autoencoder, which is why we can also consider the neural network in two sections. Usually, the autoencoder is capable of more than just compressing and decompressing data. Optimally, the autoencoder "learns" the underlying structure of the correlated input data. Thus, if a noisy input is fed into an autoencoder that was trained on clean input data, the data may be cleaned from the corruption of noise.

5.2.1 Encoder

The encoder, or coder, has the task of reducing the dimensions of the input data; this is also referred to as dimensional reduction. This reduction is achieved through compressing the data so that only the most significant or average of the data is forwarded. This method, like many other types of compression, has a loss. In a neural network, the encoding is realized by the hidden layers. This is accomplished by reducing the number of nodes in the subsequent hidden layers. You might see parallels here to earlier chapters dealing with dimension reduction techniques, which apply various forms of mathematical tricks, but in this case, the network itself learns "its" optimal dimension reduction algorithm. The dimensionality of the smallest layer in the network is called *latent dimension* and it shows how much the input has compressed.

5.2.2 Decoder

After the input signal has been encoded, the decoder comes into play. It has the task of reconstructing the original data from the compressed one. Optimally, the output data from the decoder is equivalent to the input data of the encoder. An encoder and a decoder are often symmetrical in their structure.

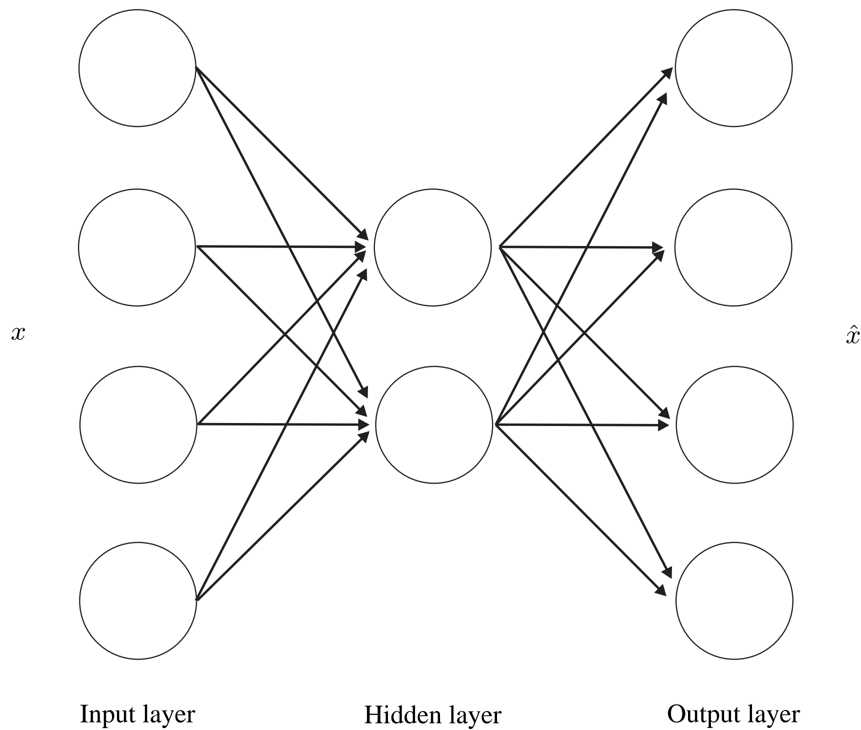


Figure 5.1: Simple autoencoder with one hidden layer.

5.2.3 Mathematical Foundations

The training target for autoencoders is that the input signal $\boldsymbol{x}$ is equivalent to the output signal $\hat{\boldsymbol{x}}$. Therefore, we can define a loss function whose input parameters are the autoencoder's input $\boldsymbol{x}$ and output $\hat{\boldsymbol{x}}$: $\mathcal{L}(\boldsymbol{x}, \hat{\boldsymbol{x}})$. This loss is minimized when input and output get closer. With the hidden layer $\boldsymbol{h}$, our encoder has the relationship $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x})$. The reconstruction in the decoder can be described as $\boldsymbol{r} = \boldsymbol{g}(\boldsymbol{h})$. Let's define some symbols to get closer to the mathematical structure of autoencoders:

Symbol	Meaning
$\boldsymbol{x}, \hat{\boldsymbol{x}}$	Input vector, output vector ,
$\boldsymbol{W}, \hat{\boldsymbol{W}}$	Weights for encoder and decoder,
$\boldsymbol{b}, \hat{\boldsymbol{b}}$	Bias for encoder and decoder ,
$\boldsymbol{\sigma}, \hat{\boldsymbol{\sigma}}$	Activation function for encoder and decoder,
$\mathcal{L}$	Loss function.

With the above defined symbols we can conclude:

$$\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{\sigma}(\boldsymbol{W}\,\boldsymbol{x} + \boldsymbol{b})$$
$$\hat{\boldsymbol{x}} = \boldsymbol{g}(\boldsymbol{h}) = \hat{\boldsymbol{\sigma}}(\hat{\boldsymbol{W}}\,\boldsymbol{h} + \hat{\boldsymbol{b}}),$$
$$\hat{\boldsymbol{x}} = \hat{\boldsymbol{\sigma}}\Big(\hat{\boldsymbol{W}}\big(\boldsymbol{\sigma}(\boldsymbol{W}\,\boldsymbol{x} + \boldsymbol{b}) + \hat{\boldsymbol{b}}\big)\Big).$$

An optimization following the mean squared error approach (MSE) would look like this:

$$\mathcal{L}(\boldsymbol{x},\, \hat{\boldsymbol{x}}) = \mathbf{MSE}(\boldsymbol{x},\, \hat{\boldsymbol{x}}) = \big\|\boldsymbol{x} - \hat{\boldsymbol{x}}\big\|^2 = \Big\|\boldsymbol{x} - \hat{\boldsymbol{\sigma}}\big(\hat{\boldsymbol{W}}\,[\boldsymbol{\sigma}(\boldsymbol{W}\,\boldsymbol{x} + \boldsymbol{b}) + \hat{\boldsymbol{b}}]\big)\Big\|^2.$$

5.3 Simple Autoencoder Example

In some cases, the best way to learn about a problem is to examine an example. Therefore, we will begin with a straightforward and easily-understood example.

5.3.1 Problem

Autoencoders can be used to compress data into a lower-dimensional output and reverse the compression afterwards. Here, we take a look at "One-Hot" encoded data, which means that exactly one element of a vector is a 1 and all other elements are zeros. It's obvious that this kind of data representation is not optimal, since even for low numbers, long vectors are required. We can intuitively think of more efficient coding types for representing numbers, such as binary, decimal, or hexadecimal representations. In this example, we will focus on a binary representation. You can find this example in **simple_autoencoder.ipynb**.

5.3.2 Dataset

We take a "One-Hot" coded dataset consisting of a representation of the numbers 0 to 3:

$$\big[1,\,0,\,0,\,0\big]^{\mathrm{T}} \mathrel{\hat{=}} 0$$
$$\big[0,\,1,\,0,\,0\big]^{\mathrm{T}} \mathrel{\hat{=}} 1$$
$$\big[0,\,0,\,1,\,0\big]^{\mathrm{T}} \mathrel{\hat{=}} 2$$
$$\big[0,\,0,\,0,\,1\big]^{\mathrm{T}} \mathrel{\hat{=}} 3$$

This *can* be encoded as

$$\big[1,\,0,\,0,\,0\big]^{\mathrm{T}} \mathrel{\hat{=}} 0 \mathrel{\hat{=}} \big[0,\,0\big]^{\mathrm{T}}$$
$$\big[0,\,1,\,0,\,0\big]^{\mathrm{T}} \mathrel{\hat{=}} 1 \mathrel{\hat{=}} \big[0,\,1\big]^{\mathrm{T}}$$
$$\big[0,\,0,\,1,\,0\big]^{\mathrm{T}} \mathrel{\hat{=}} 2 \mathrel{\hat{=}} \big[1,\,0\big]^{\mathrm{T}}$$
$$\big[0,\,0,\,0,\,1\big]^{\mathrm{T}} \mathrel{\hat{=}} 3 \mathrel{\hat{=}} \big[1\,,1\big]^{\mathrm{T}}$$

which is a simple binary representation and at the same time a reduction of dimensionality. Please remember, the last column is only one way of coding this. The order is arbitrary and might change during multiple training sessions!

5.3.3 Code

Our dataset is the above mentioned 4x4 matrix. First, we need to load the required modules (tensorflow, numpy, pandas) and for convenience some often used submodules:

```
# Imports
import tensorflow as tf
import numpy as np
import pandas as pd

from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
```

Enter the input data (numbers of 1 to 4):

```
# Dataset
X = np.array([
[[1., 0., 0., 0.]],
[[0., 1., 0., 0.]],
[[0., 0., 1., 0.]],
[[0., 0., 0., 1.]],
])
```

Tensorflow always requires an input layer that fits the shape of the input data. Therefore, we create an input layer first. Afterwards, we generate the encoder and decoder parts.

```
# Create models
inputs = Input(shape=(X.shape[1],X.shape[2]), name='Input')
encoded = Dense(2, activation='sigmoid', name='HiddenLayer')(inputs)
decoded = Dense(4, activation='sigmoid', name='Decoder')(encoded)

# separate model for autoencoder and encoder only
encoder = Model(inputs, encoded, name='Encoder_Only')
autoencoder = Model(inputs, decoded, name='Autoencoder')
```

We just use a single hidden layer for the encoder and the decoder. Additionally, we create a model that contains only the encoder. This can be used later to compress data. Please note that the creation of the encoder and decoder is done here in a hardcoded way. For further investigation and on more complex datasets, it's useful to wrap this into a function with the number of layers and neurons as parameters.

The model needs to be compiled and trained:

```
# Compile the model and print summary
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.summary()

# Train the model
autoencoder.fit(X, X, epochs=3_000, verbose=0)
```

After training, we can use the *predict* method to predict data with the trained neural network. Remember: a good autoencoder should show nearly the same data at the output as it was at the input. To mask smaller deviations, the *around* function of numpy is quite handy.

```
# Predict the train data with autoencoder
x_hat = np.around(autoencoder.predict(X))
x_hat
```

You should see that the output signal $\hat{\mathbf{X}}$ is exactly our input signal $\mathbf{X}$.

Now let's see what is actually going on in the encoder part. We take our input matrix **X** and run just the encoder part on it. This can be done by calling the *predict* method of the encoder on the input matrix:

```
1 # Predict the train data with encoder only
2 np.around(encoder.predict(X))
```

As predicted, the dimensionality is reduced. But: be careful, the *np.around* mask is not as clean as you might think. Due to training effects, the outputs are not really 0 or 1. Feel free to remove *np.around* and see for yourself.ă

In the next part, we will focus on using the encoder output to classify the input. Actually, this is what autoencoders are often used for: Reduction of dimensionality and classifying the input based on the dimension-reduced dataset. In order to do classification, we will extend the autoencoder with a classifier. This can be done in a simple manner by introducing a single layer with a suitable activation function. Since we want to differentiate between 4 classes, we will need 4 neurons in this layer. Furthermore, we can use a loss function, which emphasizes our desire to differentiate between classes.

```
1 # Encoder part with classifier
2 classify = Dense(4, activation='softmax', name='Classifier_Output')(encoded)
3
4 classifier = Model(inputs, classify, name='Encoder_with_Classifier')
5 classifier.compile(optimizer='adam',
6   loss='sparse_categorical_crossentropy',
7   metrics=['accuracy'])
8 classifier.summary()
```

Please note that our model uses *encoded* as input to the *classify* layer. That's the beauty of tensorflow: You're free to connect any layers you want, even the ones you previously defined. Be careful to keep track of the layers you introduce!

We need to fit the classifier to our input and output data. The second argument of this function call is now different from the autoencoder example since we now look for classes:

```
1   classifier.fit(X, np.array([0, 1, 2, 3]), epochs=3_000, verbose=0)
```

After fitting, we can predict and (hopefully) see the correct output.

```
1   # Predicted label of the encoder and classifier model
2   [np.argmax(classifier.predict(np.array([X[i]]))) for i in range(0,4)]
```

5.4 Exercise

For your own autoencoder exercise, we want to go a bit further. The dataset you will use is a lot more diverse and requires some minor preprocessing.

Task 1 - Set up a jupyter notebook for the upcoming tasks

Set up your notebook for this chapter. Prepare all the necessary steps to start working (i.e., imports and so on).

Hint: The necessary packages for this exercise are:

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf

from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from tensorflow.keras import layers, losses
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Model
```

Task 2 - Download the dataset

The dataset you will work on is the Fashion-MNIST [15]. Download the training and test data by running: `(x_train, x_trainlabels), (x_test, x_testlabels) = fashion_mnist.load_data()`

Task 3 - Preprocess and analyze the data

Take a look at the shape of the data. Scale it to a range between 0 and 1. Plot 10 random images from the dataset. What is the size of the images in pixels?

Task 4 - Write a basic autoencoder class

Define an autoencoder class with two "Dense" layers: an encoder, which compresses the image into a vector, and a decoder that reconstructs it from the latent space. The dimension of the latent vector should be an argument, so you can change it. To define your model, use the Keras Model Subclassing API. Provide a `call` method for the class that takes an input image, encodes it, decodes it, and outputs the result. Start with a latent dimension of 64.
Hint: A class skeleton is provided. Remember to flatten your images first and reshape them at the output, since "Dense" layers are 1D!ă

Task 5 - Compile andătrain the model

Compile the model with the same parameters as in the example `optimizer=adam, loss =MSE` You can train the model after compiling with the `fit` method on your class. Your training data is the input and output of the model (autoencoder!). Parameters: `epochs=10, shuffle=True, validation_data=(x_test, x_test)`

Task 6 - Display the results and tuning

Plot 10 original and reconstructed images side by side. Change the latent dimension and the layers activation functions, then examine the results. Write down or plot the different final losses and validation losses in your notebook.

Task 7 - Perform classification on latent space

Write a new class that performs classification on the autoencoded input data. Therefore, instead of a decoder, a classifier is needed. Replace the decoder with a classifier and use a dense layer with 10 neurons (since we have 10 different classes in the dataset). Put a softmax layer after the dense layer, since probabilities are easier to interpret than logits. Hint: Training should be done with a different cost function. `autoencoderClassifier.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])`
Additionally, your `fit` method needs the labels of the data, since we are trying to predict classes now.

Task 8 - Plot your classification results

Use the provided plot function to plot at least 10 examples. Vary the number of layers, the dimension of the latent space, and write down or plot your findings in your notebook. Please provide also a confusion matrix for your findings.

Part IV
Supervised Learning

Lab *Machine Learning*

Chapter 6

Decision Trees & Random Forests

This chapter will deal with decision trees and random forests. In a first introduction, the basic concepts and the structure of both models will be explained and a foundation for its usage will be given. Some examples will deepen the understanding and a subsequent laboratory experiment concludes the chapter.

6.1 **Basics** 61

6.1.1 Decision Trees 61

6.1.2 Random Forests 64

6.2 **Exercise** 65

6.1 Basics

Decision trees and random forests are a subset of supervised learning constructs used for data classification and regression problems. They perform best on any kind of linear or nonlinear data, but have trouble with extrapolating results out of the limitations of the training set.

6.1.1 Decision Trees

A decision tree is basically a set of questions that is asked for every data sample. It takes the input and compares it to a set of thresholds that are preliminary trained on a defined set of data. In that sense, decision trees are closely related to support vector machines (Chapter 7). Decision trees can easily be visualized using the form of an upside down tree with the trunk as a root node, branches as internal nodes and leaves as leaf nodes. Every node contains a condition and two resulting nodes, except for the leaf nodes. They mark the end of a path in the tree. Those trees are generally created starting from the root node. For construction, also known as training, the underlying dataset needs to be labeled, thus the method is being called a supervised learning method.

To find the first split or the first question to ask in the root node, the training algorithm relies on one of multiple possible criteria. Most commonly used nowadays are the *information gain* or the

gini index. For both criteria, the algorithm iteratively splits the dataset and determines the value of the split criterion for each. It then chooses the feature with the best criterion value as the split feature.

6.1.1.1 Information Gain

The goal of the information gain IG is to maximize the reduction of entropy E in each split. It is defined as

$$IG = E_{\text{parent}} - \sum_{i=1}^{N_{\text{children}}} \frac{N_{\text{child, i}}}{N_{\text{total}}} E_{\text{child}}, \tag{6.1}$$

with E_{child} as the entropy of a child node and E_{parent} as the entropy of the parent node, meaning the node, that is connected directly above the current child node. $N_{\text{child, }i}$ is the number of samples in the i_{th} child node, while N_{total} is the number of samples in all of the child nodes together. The entropy E can be calculated as

$$E = \sum_{i=1}^{N_{\text{classes}}} -p_i \log_2(p_i), \tag{6.2}$$

with N_{classes} being the number of classes in the current node and p_i as the share of the current class.

If for example there are $N_{\text{total}} = 8$ samples of $N_{\text{classes}} = 2$ classes in a node, with the first class having $N_{\text{child, 1}} = 3$ samples and the second class having $N_{\text{child, 2}} = 5$ samples, the entropy in that node would resolve to

$$E = -\frac{N_{\text{child, 1}}}{N_{\text{total}}} \log_2\left(\frac{N_{\text{child, 1}}}{N_{\text{total}}}\right) - \frac{N_{\text{child, 2}}}{N_{\text{total}}} \log_2\left(\frac{N_{\text{child, 2}}}{N_{\text{total}}}\right) \approx 0.95. \tag{6.3}$$

Say there is a feature that splits the samples perfectly into two child nodes with the first node containing all samples of the first class and the second node all samples of the second class. The resulting entropy for all of the nodes would equal to $E(\text{child}) = 0$, because each node contains samples of only one class. The information gain would consequently be $IG(\text{child}) \approx 0.95$ after Eqs. 6.1 and 6.2. If on the other hand a feature was chosen, that does not split the samples perfectly, the cumulative entropy of the resulting samples in each node would not equal zero and thus provide a lower information gain, making it the less preferred split.

6.1.1.2 Gini Index

The gini index works similar to the information gain. It is also calculated using the classes probabilities per node, but it does not require the computationally expensive logarithm. It is defined as

$$G = 1 - \sum_{i=1}^{N} p_i^2 \tag{6.4}$$

and represents the probability of labeling a random sample of that node wrong given the dominant label of that node. The weighted sum of gini indices of child nodes is to be minimized. If for example a decision has to be made between splitting the parent node with initially

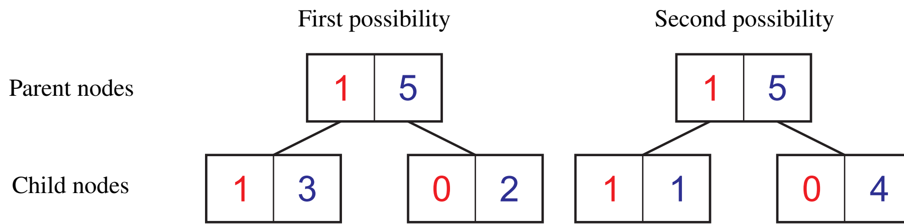


Figure 6.1: Two possibilities to split the dataset from the root node into child nodes with different gini indices.

- one of class A and five of class B into two nodes of either
  - one of class A and three of class B for the first node and
  - zero of class A and two of class B for the second node or

  - one of class A and one of class B for the first node and
  - zero of class A and four of class B for the second node

(as depicted in figure 6.1), the second option would be chosen regarding to the gini criterion because its gini index is smaller:

$$G_1 = \frac{4}{6} \cdot \left\{1 - \left(\frac{1}{4}^2 + \frac{3}{4}^2\right)\right\} + \frac{2}{6} \cdot \underbrace{\left\{1 - \left(\frac{0}{2}^2 + \frac{2}{2}^2\right)\right\}}_{=0} = \frac{1}{4} \tag{6.5}$$

$$G_2 = \frac{2}{6} \cdot \left\{1 - \left(\frac{1}{2}^2 + \frac{1}{2}^2\right)\right\} + \frac{4}{6} \cdot \underbrace{\left\{1 - \left(\frac{0}{4}^2 + \frac{4}{4}^2\right)\right\}}_{=0} = \frac{1}{6}. \tag{6.6}$$

6.1.1.3 Node Generation

After the best feature is chosen based on one of the criteria, the data is split based on the chosen feature and two nodes are created based on that split. This algorithm repeats the steps for every single node, until the maximum defined depth of the tree is reached or the homogenity in each node is perfect.

As an example there may be a dataset with two features and two classes as shown in Fig. 6.2 - a binary classification problem, and the split criterion is chosen to be the gini index.

Starting in the root node, there are 6 samples to be classified. As a single node without any restrictions, the node would output class 1 for every possible input, since the underlying dataset has more samples of class 1 than of class 0. To determine a good split, the algorithm tries to split the dataset based on the first feature, in this case for the threshold $f_1 > 0.5$. The gini index would resolve to

$$G_1(n) = \frac{3}{6} \cdot \left\{1 - \left(\frac{1}{3}^2 + \frac{2}{3}^2\right)\right\} + \frac{3}{6} \cdot \underbrace{\left\{1 - \left(\frac{0}{3}^2 + \frac{3}{3}^2\right)\right\}}_{=0} \approx 0.22, \tag{6.7}$$

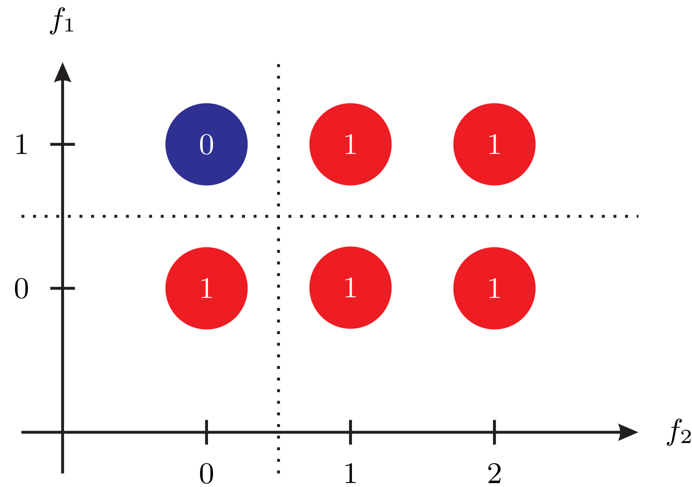


Figure 6.2: Dataset with two features, six samples and binary classification.

If instead using the second feature with its threshold $f_2 > 0.5$, the gini index would resolve to

$$G_1(n) = \frac{2}{6} \cdot \left\{1 - \left(\frac{1}{2}^2 + \frac{1}{2}^2\right)\right\} + \frac{4}{6} \cdot \underbrace{\left\{1 - \left(\frac{0}{4}^2 + \frac{4}{4}^2\right)\right\}}_{=0} \approx 0.17, \tag{6.8}$$

making it the better alternative due to the lower gini index. Based on that algorithm, the whole tree is build.

6.1.1.4 Pruning

The problem with decision trees lies in the fact, that they reach full accuracy on the given dataset but that most commonly leads to worse results with actual testing data. This is because the stopping criterion for the training of the tree is set to be an information gain or a gini index of zero, leading to perfect classification for the training dataset. A solution to that is called *pruning*. It resembles the pruning of actual trees by intentionally cutting and removing leaf nodes to reduce the depth of the tree. This can be done either after training by using cross-validation and removing the leaf nodes that do not contribute to the final accuracy or removing features that are not that important to the tree. Or it can be done indirectly before training, by restricting the growth of the tree initially. This can be done for example by defining a maximum tree depth or by lowering the requirements for the purity of a node, described by the entropy or the gini index. Another very effective method to prevent overfitting is to use multiple differently trained and connected decision trees, called random forests.

6.1.2 Random Forests

Random forests is an extension of decision trees. It addresses the problem of overfitting due to the exact mapping of the training dataset. The idea is to bootstrap the data, which takes only a random part of the dataset for each decision tree. Another method is to randomly restrict the features used to split the data in each decision tree. Due to the large amount of tweakable

parameters, hyperparameter tuning is often applied, which essentially trains the random forest multiple times using a set of random or predefined parameters to ultimately look for the best parametrisation.

To get a valid combined result from all of the used decision trees in a random forest, different compression functions based on what data you need can be used. If you are facing a regression problem, the way to go would probably be an average of all the outputs of the decision trees. If the problem is a based on classification, you may chose to use the most common resulting class given by the decision trees.

6.2 Exercise

Task 1 - Decision tree basics

a) Consider the dataset from figure 6.2, but replace all samples at $f_2 = 2$ with a sample of class zero.

b) Think of the options you have on splitting the dataset using only feature two (f_2). Without calculation, what do you guess will be the split that yields the smallest impurity and why?

c) Calculate the gini indices for splits at $f_2 = 0.5$ and $f_2 = 1.5$. Can you confirm your considerations by interpreting the indices?

d) After making this first split, you essentially created two nodes. What type of nodes are they and why?

e) Repeat making splits using the gini index as criterion until you end up with only leaf nodes. Write down the feature and threshold that you split at for each non-leaf node.

f) Using if-statements, write down the complete tree. Use a two-dimensional array as input vector with the first index representing the first feature value f_1 and the second index as the second feature value f_2.

g) After creating your own decision tree, test its classification capabilities with different feature values. What happens for values greater than $f_1 = 1$ or $f_2 = 2$ or values smaller than zero? What is the problem with those values?

Task 2 - Using a sophisticated library for the decision tree

a) Import the libraries *numpy* and *pandas*. Consider the dataset from task 1. Think of the data as a two-dimensional array with the columns 'f_1', 'f_2' and 'class' and the rows as samples, that represents the dataset, create it as a pandas dataframe using a numpy array with correct labels. Print the created dataframe and check, if the output concurs with the given dataset.

b) Now we want to use a sophisticated library to create a decision tree with our data automatically. Import the class tree from the library *sklearn* using `from sklearn import tree`. Create a new decision tree classifier using `tree.DecisionTreeClassifier()`.

c) Instead of calculating all the criteria and writing the condition structure, we want the class to do this for us using our little dataset. To do that, we first need to format the data such that the class can handle it properly. Create a feature array X with the two features as columns and the samples as rows. Create a complementary label array y with the column 'class' and the samples as rows.

Task 2 - Using a sophisticated library for the decision tree

d) Now to automatically train our model, use the function `fit` of the `DecisionTreeClassifier()` that you already created. To visualize the result, use the matplotlib (`import matplotlib.pyplot as plt`). You can plot the structure of the tree by using the `plot_tree(decision_tree)` function provided by the tree class and passing the trained model to it. Does the resulting plot match your decision tree from task 1f)? What does the information displayed in each box mean?

e) To use the trained decision tree, it provides the `predict(X)` function, with X being a vector of your features. Try different input combinations, does the output match the results of your own decision tree?

f) The library *sklearn* also provides multiple more real-world datasets than the one used so far. Import the dataset library (`from sklearn import datasets`) and load the iris data using `load_iris()`. Feel free to load any other classification dataset, an overview can be read at the documentation of the datasets class. Take a look at the data and print out the provided feature names and class names (`feature_names` and `target_names`). Load the data into a dataframe similar to task 2a), print out and understand the values in the rows. What does one sample tell you?

g) Now that we have a bigger dataset, we can split it into 70% training and 30% test sets to verify the accuracy of our model. Do so by using the function `train_test_split(X, y, test_size)` after importing it using `from sklearn.model_selection import train_test_split`. Create a new decision tree classifier and fit it with the training data. Predict the outcome with the test data. You can use the library *metrics* to easily calculate the accuracy. Import it by using `from sklearn import metrics`. It provides the function `accuracy_score(y_text, y_pred)`. Print the accuracy of your trained decision tree. Why does it change after running the cell again? Fix the randomness to make your future results comparable.

h) Another way to test your model is to print the confusion matrix. This can be done using the function `confusion_matrix(y_test, y_pred)` of the metrics class. What does the matrix reveal?

i) An important parameter of the model is the feature importance. It gives a measure of how much influence a single feature has in the dataset. Print out the importance parameter of the model (`feature_importances_`) and describe what you see. Can you improve your model by pruning the features? If not, why does it not work?

j) Try out different datasets, for example the digits dataset and do the pruning again. Also try to prune the tree by setting the *max_depth* parameter. Does it improve the accuracy?

Task 3 - Random forests

a) To decrease the risk of overfitting, use the random forest classifier instead of only a single decision tree, that is present in the *sklearn.ensemble* library. It can be initialized and trained with the same commands as the decision tree. Train it with the same dataset that you used in task 2j) and compare the results.

b) What are the advantages of using a random forest classifier above the decision tree classifier? What are the drawbacks?

c) Now we want to improve the random forest classifier by tweaking its parameters. The library *sklearn* also provides a function for that purpose. Import it using `from sklearn import model_selection`. Start with taking a look at the parameters that can be defined for the random forest classifier and picking at least `n_estimators` as the number of decision trees to be created in the model and `max_depth` as the maximum depth for each decision tree. Those are the variables we want to optimize. For each parameters think of values you want to test. The first step is to test a random subset of your given possible parameter values to narrow them down. The next step is to adapt the parameters according to the result of the search and perform another test using every single value that you defined. Start by creating a dictionary, that contains for each chosen parameter loosely a set of values as an array that you would like to test.

d) Use the `model_selection.RandomizedSearchCV(estimator=classifier, param_distributions=your_dictionary)` class to initialize a random search over your defined dictionary and fit it with the previously used training data. Print the set of best parameters (`best_params_`), that is part of the previously created class.

e) Look at the best parameters and choose values close to the best parameters in a finer resolution than previously. Update your dictionary and feed it to the class `model_selection.GridSearchCV(classifier, your_dictionary)`. Fit it and print the set of parameters. This can take a while depending on how many values you defined, since its training the classifier with each combination you provided. That is why you first use the random search, to narrow down the optimal parameters.

f) Apply your optimized parameters to your initial classifier and compare the results. Compare the parameters and make assumptions as to why the classifiers perform differently based on their values.

Chapter 7

Support Vector Machines

In this chapter Support Vector Machines (SVMs) are described. First, the theory and approaches of SVMs are presented. Subsequently, an own SVM is to be programmed and used in different tasks. Finally the powerful SVM of the Scikit-Learn libraries is used, in order to learn the possibilities and benefits of it in more complex datasets.

7.1 **Basics** 69
- 7.1.1 Linear SVM 71
- 7.1.2 Nonlinear SVM 73
- 7.1.3 Applications & Limitations 74

7.2 **Exercise** 75
- 7.2 Comprehensive Questions 75
- 7.2 Simple SVM 78
- 7.2 Sklearn SVM 80

7.1 Basics

Support vector machines (SVMs) are a form of supervised learning in the big field of machine learning. They are applied for data classification as well as for regression. However, they are mainly used in the classification of data, which is also the focus of this experiment. The original SVM algorithms were invented in 1963 by Vladimir N. Vapnik and Alexey Ya. Chervonenkis, but only became popular with the extension of nonlinear kernels in 1992 by Bernhard Boser, Isabelle Guyon and Vladimir Vapnik [16].

To understand the very basic principle we can look at an example. In Fig. 7.1 a dataset is plotted that contains both red triangles and green circles. Visually, we can see that both datasets are easily separated. As humans, we can do a very good job of separating the two classes. However, this becomes much more difficult as the feature size increases and the classes overlap to a greater extent. This is where machine learning now comes into play. The approach of a SVM is to separate the data using planes. In the example given above, a plane can be computed that optimally separates the dataset regarding the given classes. A solution for the example given above is shown in Fig. 7.2.

How these so-called hyperplanes are created mathematically is derived in the following. For the sake of simplicity, we will only look at binary datasets. Here $\boldsymbol{x}_i$ represents a point of the data set

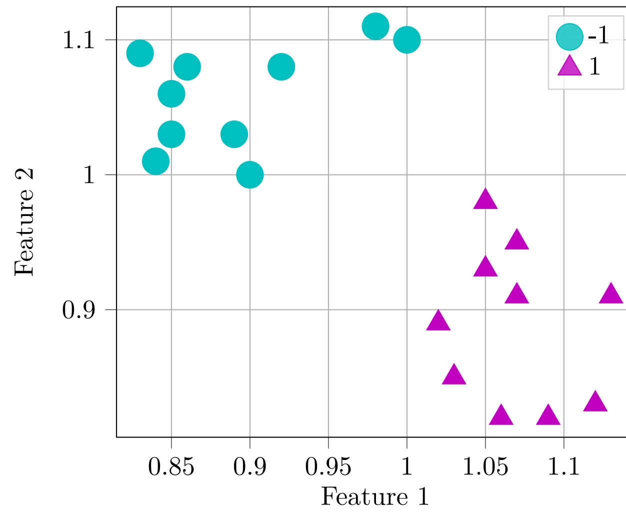


Figure 7.1: Example dataset with two classes we want to separate.

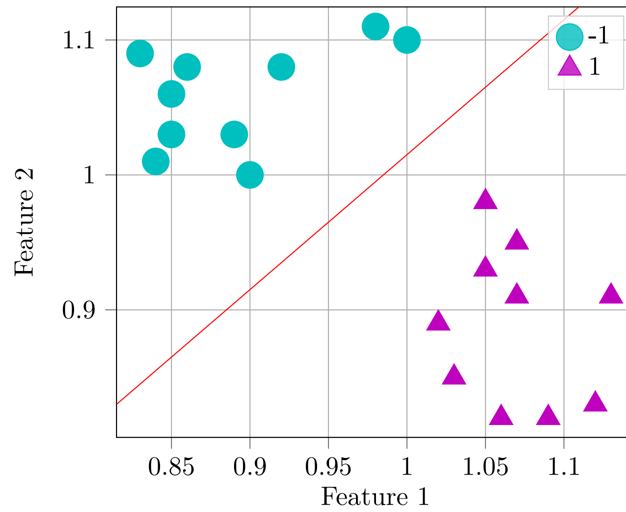


Figure 7.2: Example dataset a hyperplane separating the classes.

$\boldsymbol{X}$. The data set consists of N data points. For training, N labels are additionally needed, which describe the classification of the point $\boldsymbol{x}_i$ to the two classes in the form of $y_i \in \{-1, +1\}$.

7.1.1 Linear SVM

As shown in the example above, a SVM tries to find a hyperplane that separates the data as good as possible. In theory there are an infinite amount of hyperplanes that would completely separate our example dataset, but the goal of an SVM is to find the optimal hyperplane. Before we can find the optimal hyperplane we need to find a way of evaluating hyperplanes. For that we need to look at the mathematically definition of a plane. It's defined as

$$\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} + b = 0. \tag{7.1}$$

In a geometrical sense the weights $\boldsymbol{w}$ define our normal vector and b an offset to that plane. All points $\boldsymbol{X}$ that fulfill the formula (7.1) will lay on this plane. Therefore we can test if a point will be located on one side of a plane or the other by looking at the sign of the result of

$$h(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} + b. \tag{7.2}$$

Now, to find an optimal solution for a given dataset, we need to define a cost function that we can optimize. As an intuition we want the line to be in the middle of each class so that the hyperplane has the biggest margin for future unknown data and therefore outliers. The margin can be defined using the minimum distance from each observation to a given hyperplane. Therefore we want to maximize the margin of the hyperplane resulting in the optimal hyperplane with the biggest margin. The definition of the margin γ for a point $\boldsymbol{x}$ is given by

$$\gamma = y\left(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} + b\right). \tag{7.3}$$

Sometimes you will find the term $\|\boldsymbol{w}\|$ in Eqn. (7.3) as a normalization factor for neglecting scaling influences. You can imagine if you scale your dataset with a constant, everything will stay the same relative to each other, but the margin will grow due to a bigger distance of the points and the hyperplane. But for this implementation we'll neglect this factor. The variable y is our label of the given class and will either be one or minus one. Now we can calculate the minimum distance of all points $\boldsymbol{X}$ to the hyperplane according to [17] with

$$\gamma_{\min}(\boldsymbol{X}) = \min_{x \in \boldsymbol{X}} \left\{y(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b)\right\}. \tag{7.4}$$

resulting in the margin of the given hyperplane. The name support vector machine can be explained with Eqn. (7.4). We can see, that the performance of the hyperplane is measured on the distance from the hyperplane to the nearest points. Therefore those points are called support vectors and give the name to this whole concept.

To determine the weights for the optimal hyperplane $\boldsymbol{w}_{\mathrm{opt}}$, we need to feed our SVM with data and train it. In the process, a loss function L must be optimized. Usually the hinge loss is used for the optimization of SVMs with the gradient decent. This is defined according to [17] by

$$\zeta = \max\left\{0,\, 1 - y(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x} + b)\right\} \tag{7.5}$$

and represents a mathematical definition of the error. Subsequently, the loss function of the SVM can be formed from this, which results in

$$L(\boldsymbol{w}) = \frac{\|\boldsymbol{w}\|^2}{2} + C \sum_{i=0}^{N-1} \max\Big\{0,\, 1 - y_i\big(\boldsymbol{w}^{\mathrm{T}}\,\boldsymbol{x}_i + b\big)\Big\}. \tag{7.6}$$

The loss function consists of our weights $\boldsymbol{x}$, the bias b and the regularization constant C. The regularization parameter C is used as a tuning parameter between maximizing the margin and minimizing the loss function. To better understand the influence of the parameter, we will play around with it a bit in the exercise.

To obtain the update equations for the weights $\boldsymbol{x}$ and the bias b, the loss function must be derived for the respective parameters after [17] in the form of

$$\frac{\delta L}{\delta \boldsymbol{w}} = \boldsymbol{w} - C \sum_{i=0,\, \zeta_i \geq 0}^{N-1} y_i\, \boldsymbol{x}_i. \tag{7.7}$$

Afterwards the update of the weights according to

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - r\frac{\delta L}{\delta\, \boldsymbol{w}}. \tag{7.8}$$

The parameters r is called learning rate and determines the step size for each epoch. A high learning rate allows a fast convergence to the optimal solution, but also results in a less accurate solution. A low learning rate requires more epochs to reach the optimal solution, but can approximate it more accurately. The parameter n is representing the epoch dependency. Each iteration we need to calculate new values based on the value in the epoch before. Analogous to the weights, the gradient of the bias can also be calculated according to [17] with

$$\frac{\delta L}{\delta b} = -C \sum_{i=0,\, \zeta_i \geq 0}^{N-1} y_i. \tag{7.9}$$

Subsequently, the new bias values are calculated with

$$b(n+1) = b(n) - r\frac{\delta L}{\delta b}. \tag{7.10}$$

To summarize the training of a SVM, the following steps are necessary.

Training of soft-margin SVM

1. Calculate the margin for current hyperplane weights with Eqn. (7.3).
2. Calculate the loss for current margin with Eqn.,(7.2).
3. Identify misclassified points by comparing the output with real labels.
4. Update weights $\boldsymbol{w}$ with Eqs. (7.7) and (7.8).
5. Update bias b with Eqs. (7.9) and (7.10).
6. Repeat step 1.-5. for each epoch.

As a small remark, it should be noted that the calculation of the loss in the second step is not absolutely necessary for the training, but on the one hand can provide important information about the result of the training and on the other hand can be used as a termination criterion in a more sophisticated implementation.

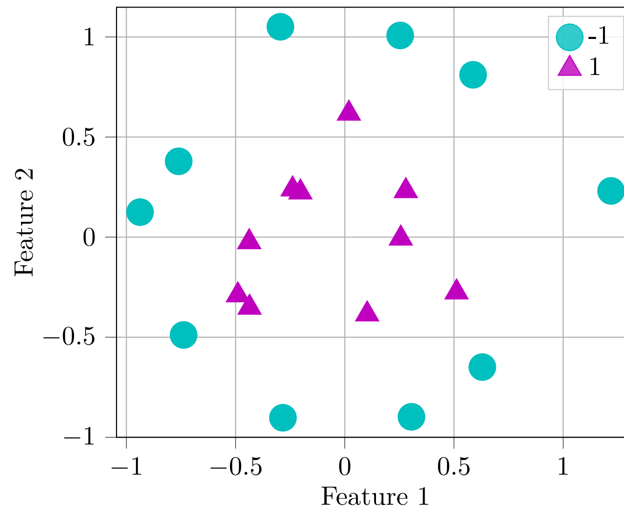


Figure 7.3: Example dataset where the small axis indicates what directions hold maximum variance.

7.1.2 Nonlinear SVM

In the previous section we have looked at quite detailed description and derivation of a simple SVM. Since besides the linear SVM especially the nonlinear SVM is commonly used, in this section we will look at the basic idea of the kernel method and the two main kernels used.

7.1.2.1 Kernel Method

$$\hat{y} = \operatorname{sign} \sum_{i=0}^{N-1} w_i \, y_i \, k(\boldsymbol{x}_i, \, \boldsymbol{y}_i) \tag{7.11}$$

The most prominent example for this kernel method is the the following dataset. We have a binary dataset in which on class is circular placed around a center point and a second class completely surrounding this circle. This is depicted in Fig. 7.3.

If we now apply a linear SVM to our dataset, we will only get an inadequate result. The reason for this is that we cannot put a hyperplane through our dataset that separates the data successfully or sufficiently. In this case a kernel is needed, which transforms the data depending on the radius to the center point. Mathematically this is given for example by the kernel

$$K_{\text{circ}}(\boldsymbol{x}_i, \, \boldsymbol{x}_j) = \boldsymbol{x}_i \, \boldsymbol{x}_j + \|\boldsymbol{x}_i\|^2 \, \|\boldsymbol{x}_j\|^2. \tag{7.12}$$

If we compute the new feature Z and display the data, as depicted in Fig. 7.4, we can clearly see that the data can easily be separated by a hyperplane.

Depending on the dataset and distribution, different kernels may have advantages or disadvantages. Two of the most commonly used kernels are presented below.

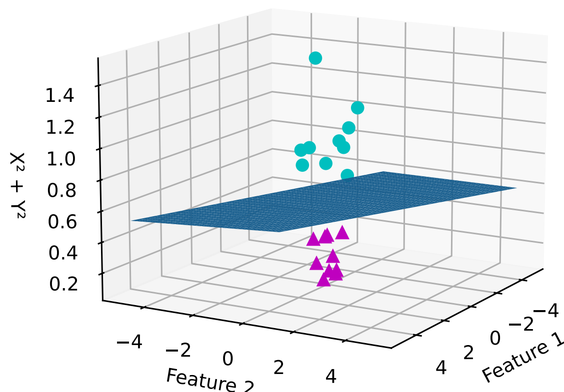


Figure 7.4: Example dataset where the small axis indicates what directions hold maximum variance.

7.1.2.2 Polynomial Kernel

The polynomial kernel is calculating the dot product by increasing the power d of the kernel. Often an additional bias term c is added inside. The kernel is defined after [18] by

$$K_{\text{pol}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(\boldsymbol{x}_i^{\mathrm{T}}\, \boldsymbol{x}_j + c\right)^d . \tag{7.13}$$

The polynomial kernel is for example often used in natural language processing. A disadvantage of the kernel is that it quickly tends to zero or infinity with increasing d depending on the dataset, which can lead to numerical instabilities.

7.1.2.3 Radial Basis Function Kernel

The radial basis function (RBF) kernel is the widest used kernel for SVMs and are generally often used in machine learning. The value of the RBF kernel depends of the distance from the origin or some point. σ is customizable parameter for shaping this function and controls the fitting. A higher σ will lead to overfitting, a lower σ to underfitting. The kernel is defined after [18] by

$$K_{\text{rbf}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}{2\sigma^2}\right) . \tag{7.14}$$

7.1.3 Applications and Limitations

SVMs in general are really effective with higher number of dimensions. As a rule of thumb the number of training data should be higher than the number of features. Due to the definition of the hyperplanes using only the support vectors, the performance of a SVM is very strongly dependent on the outliers. According to [18] the training of SVM is relatively slow, since many combinations have to be calculated. Furthermore, the performance tends to be poor for overlapping classes. The selection of the right kernel can also be difficult.

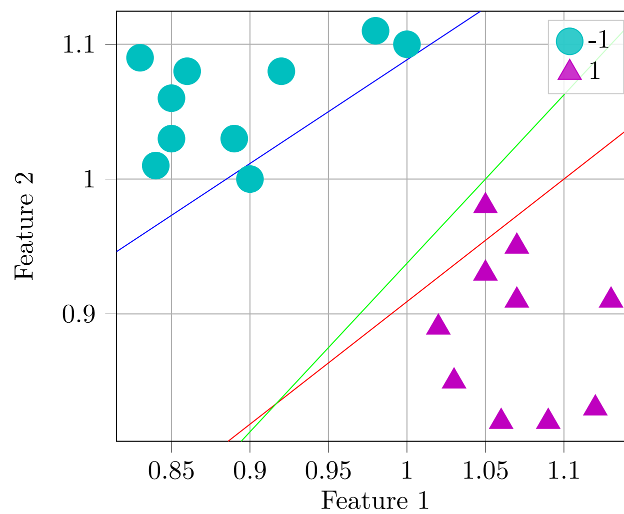


Figure 7.5: Figure for task 1a.

7.2 Exercise

Task 1 - Comprehensive Questions

Let's start with the first task, the purpose of which is to understand the basic properties of support vector machines. In this section you don't need to program anything yet, but only to use sharp thinking.

a) Look at Figure 7.5. In it, two data clouds are given, which are to be separated. Which of the SVMs presented above can be used? Additionally, three lines are drawn as possible hyperplanes. Which of the three lines separates the classes?

b) In Figure 7.6, several hyperplanes are now drawn, all correctly separating the data. Which of the three hyperplanes is likely to give the best result when classifying new data that was not yet available during training?

c) In the third scenario (Fig. 7.7), one point has been modified. Which of the drawn hyperplanes separates the data better now?

d) In Figure 7.8, one data point has been changed again. This time, however, the change results in the classes no longer being linearly separable. Which of the presented types of SVMs should be used now?

e) As a last scenario (Fig. 7.9), a more complicated distribution of classes is given. Think about how this would be separated by the different SVM approaches.

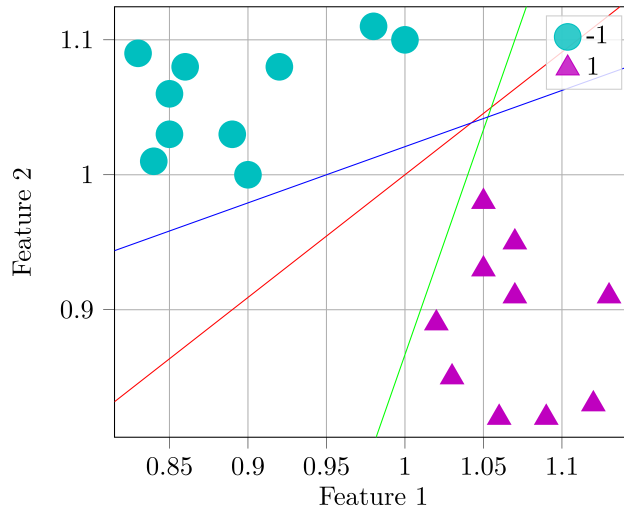


Figure 7.6: Figure for task 1b.

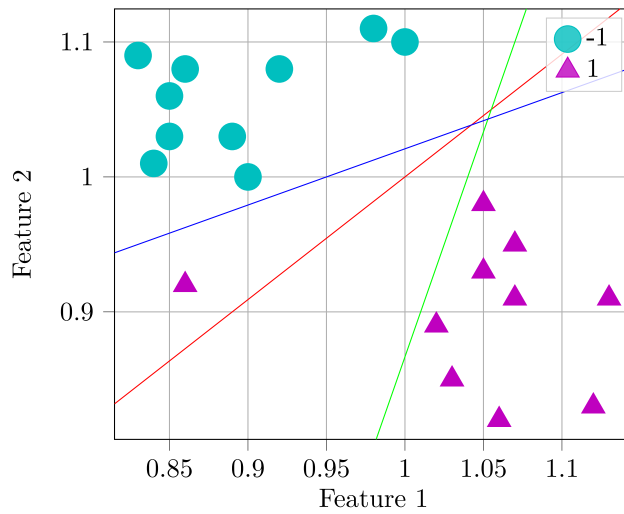


Figure 7.7: Figure for task 1c.

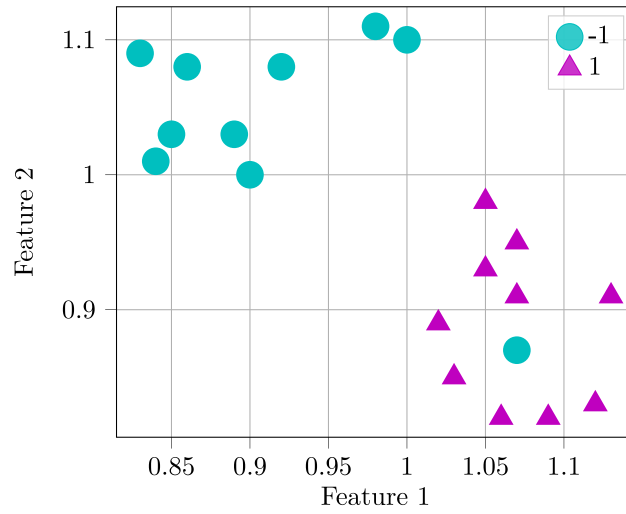


Figure 7.8: Figure for task 1d.

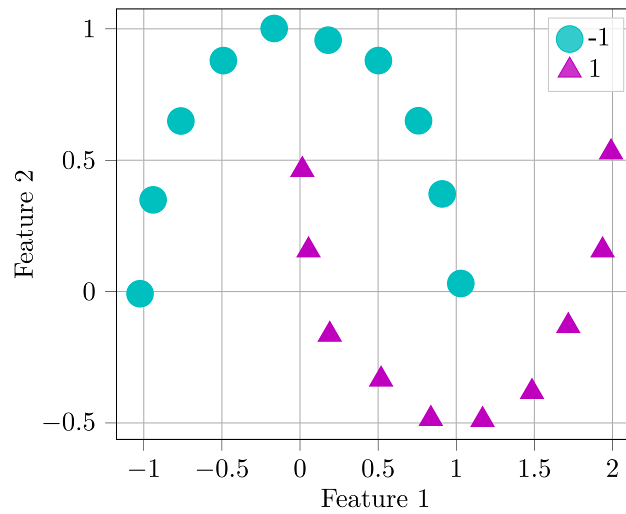


Figure 7.9: Figure for task 1e.

Task 2 - Simple SVM

In this task we will program our own support vector machine. The goal is to understand the basics behind it and the limitations of it. For that we will test it with several datasets.

a) Open the notebook *svm_task2.ipynb*. You will see, that there are already cells filled. We will go through this notebook and consecutively extend our support vector machine. Don't worry, at important steps we have implemented a small check to see if your functions work correctly.

b) Import the following packages:

   - `numpy` as `np`
   - `pyplot` as `plt` from the `matplotlib` package
   - `StandardScaler` from the `sklearn.preprocessing` package
   - `make_blobs` from the `sklearn.datasets` package

c) Now we can have a look at our support vector machine class. You'll see, that some functions are not implemented yet. Look at the `__init__` function of our class. There all important parameters get initialized. Make yourself familiar with them and try to associate each of them with the given formulas in 7.1.1.

d) Let's fill in some code. Implement the funcion `margin`. Orientate yourself on the formula 7.3. The input should be a dataset X containing multiple points and y which contains corresponding labels. The output should be a vector containing a margin for each of those points. You can test your implementation by executing the cell 4.

e) The second function we want to implement is `cost`. Refer to Eqn. (7.6). The input of this function should be the margin vector from above. As an output we expect a scalar, which is the loss. You can test your implementation by executing the cell 5.

f) The next function we need to complete is our fitting function. In there we train the model over our epochs. You'll find an incomplete for loop. We will complete it in the next steps. At first calculate the margin between our dataset X and our labels y.

g) For a better overview of the training we can determine the loss for each epoch. Therefore calculate the the cost of the margin and store it in the `loss_array`.

h) Now we can start to update our weights. For that we firstly need to check which points were misclassified. The easiest way is to extract the indices of all misclassified points. Extract them by looking at the margins.

i) Now we want to update our weights. For that we need to complete the function for calculating the derivation of our weights. Implement a function called `update_weights_derivation` that gets the dataset X, the labels y and the indices of the misclassified points as input. It should return a vector with the size of $\boldsymbol{x}$. For implementation orientate yourself on formula (7.7).

Task 2 - Simple SVM

j) Now update the weights in the loop with the learningrate r and and the derivation of the weights as in equation (7.8).

k) The second variable to update are the b's. Complete the function `update_bs_derivation`. For this we only need the labels y and the indices of the misclassified points as an input resulting in a scalar output. For orientation use equation (7.9).

l) Update the b's inside the loop with the derivation and the learningrate r according to equation (7.10).

m) The class should be nearly complete. For testing we need data though. So generate a dataset consisting of 2 classes with each 100 samples with `make_blobs`. For a better comparability use the seed 42.

n) Have a look at the generated data. You'll see that the labels consists of zeros and ones only. For our self-written SVM we need labels that are either minus one or one. So convert all zeros to minus ones.

o) A second preprocessing step is to scale the data to an appropriate input scale. Use the function `fit_transform` of the `StandardScaler` to scale our dataset. Note: Don't scale the labels, because they already have an appropriate format.

p) For a complete evaluation we need to check the SVM with unknown data. Import `train_test_split` from `sklearn.model_selection` and split the data in 70% training and 30% testing data.

q) Visualize the data in form of a scatter plot by completing the function `plot_data`. Color each class in a different color for a better overview.

r) Complete the function `classify`. Tip: Our labels are either plus or minus one, this may help you simplify the classification.

s) Complete the function `score`. This function should get an dataset set and corresponding labels as input and should compute the accuracy.

t) Now we are nearly finished. For a first test create an instance of the class and run it with created training dataset and labels. Afterwards you can check the implementation by running `plot_decision_boundary`. You should see the dataset is split by a hyperplane.

u) Run the `score` function and test the dataset. Look at the result and change the values of C and the learningrate r. What can you tell about the influence of the result? Visualize the loss.

Task 3 - Sklearn SVM

Now we can have a look at the SVM from the sklearn library so you learn how to use it. Open the notebook *svm_task3.ipynb*.

a) Import the following packages:
   - `numpy` as `np`
   - `pyplot` as `plt` from the `matplotlib` package
   - `train_test_split` from the `sklearn.model_selection` package
   - `StandardScaler` from the `sklearn.preprocessing` package
   - `make_blobs` from the `sklearn.datasets` package

b) Generate test and training data as in task 2.

c) Import svm from sklearn.

d) Create a svm with the linear kernel and a C of 15.

e) Fit the svm with the training data and labels with the function `fit` from the svm class.

f) Visualize the training data with the corresponding hyperplane using `plot_data`.

g) Test the svm by visualizing the test data.

h) Predict the response for test dataset with the function `predict` from the svm class.

i) To evaluate the training, load from sklearn the `metrics` package.

j) Calculate the accuracy score of the predicted labels using `accuracy_score`.

k) Now lets increase the overlapp over our classes by using the same parameters of `make_blobs`, but defining the center box from -3 to 3.

l) Create a new linear svm with $C = 15$ and fit the new training data.

m) Plot the data with test data and calculate the accuracy score.

n) Now import `make_moon` from sklearn datasets.

o) Create a new dataset with 100 samples and random state of 42.

p) Split this data into 70% training and 30% test data with the seed 2.

q) Create a linear svm with $C = 15$ and run the fitting with the training data.

r) Plot the test data and calculate the accuracy score.

s) Now Create a svm with an RBF kernel and $C = 15$ and fit the moon training data.

t) Plot the test data and calculate the accuracy score.

u) Now you can play around with different C values and blobs or moons.

Chapter 8

Convolutional Neural Networks

This chapter introduces Convolutional Neural Networks (CNN). First, the basics are presented and then, in the laboratory experiment, a Neural Network is trained on the basis of the MNIST digit dataset. Afterwards, a CNN is trained in order to compare the results and also the complexity of both networks.

8.1 **Tensorboard Basics** 81
8.2 **CNN Basics** 81
8.3 **Exercises** 84

8.1 Tensorboard Basics

Tensorboard is a tool provided by Tensorflow that can be used to visualize various metrics regarding the model and the training. For example, the graphs of trained networks can be visualized, but also the process of the training with the associated improvement in accuracy. In this lab, this tool is used by providing a ready-to-run python script with corresponding instructions. It will be used to visualize the trained networks and to analyse their complexity. If you want to get to know further features of this tool, you can do this in self-study with `https://www.tensorflow.org/tensorboard/get_started`.

8.2 Basics

A Convolutional Neural Network (CNN) is a type of Artifical Neural Networks (ANN) that is usually used to recognise patterns in images or audio files. Due to their architecture, these networks can process images with much less computational effort than neural networks that only rely on fully connected layers.

8.2.1 Structure

The aim of a CNN is to generate a vector with the dimensions $1 \times 1 \times n$ from an image with dimensions height $\times$ width $\times$ depth via different layers as shown in Fig. 8.1. For this case n corresponds to the number of classes to be classified and the resulting vector $\hat{\boldsymbol{y}}$ is usually one-hot coded. Three different types of layers are used for this purpose:

- Convolutional layer,
- Pooling layer,
- Fully-connected layer.

A basic structure of a CNN with a possible composition of these layers is shown in Fig. 8.1. Via the convolutional and pooling layer, pattern recognition is performed by the network learning existing structures in the image $\boldsymbol{x}(n)$. The fully-connected layer classifies the learned features to get the resulting output $\hat{\boldsymbol{y}}(n)$.

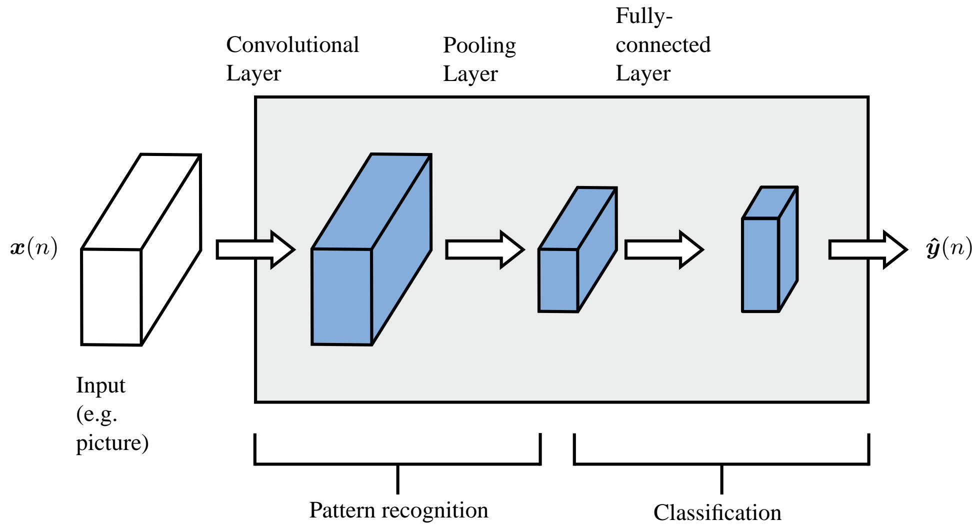


Figure 8.1: Basic structure of a CNN

8.2.1.1 Convolutional Layer

Together with the pooling layer, the convolutional layer is used to recognise patterns in the input images and to learn features. Each element of the image is processed by a discrete convolution

$$\boldsymbol{S}(i,j) = (\boldsymbol{K} * \boldsymbol{I})(i,j) = \sum_m \sum_n \boldsymbol{I}(i-m, j-n)\, \boldsymbol{K}(m,n). \tag{8.1}$$

$\boldsymbol{I}$ is the input image, $\boldsymbol{K}$ is a kernel and $\boldsymbol{S}$ the resulting image after convolution. Thus, a fixed kernel is multiplied with an smaller image section. The kernel can be initialised before and adjusted during training. Via this kernel, smaller sections of the image are viewed at a time and structures can be recognised. Different kernels can be trained for different layers so that different structures can be recognised.

The resulting image is processed with an element-wise *relu* activation function. The individual neurons therefore fire when the kernel recognises certain structures.

8.2.1.2 Pooling Layer

The pooling layer is the second part of the basic pattern recognition structure of a CNN shown in Fig. 8.1. This layer is used for downsampling in order to reduce the parameters for the next layer. For this purpose, a symmetrical matrix size is defined as pooling size. The image, which has been build by the previous convolutional layer, is processed section by section with this matrix size.

One pooling possibility is to search for the maximum within this matrix and to transfer the corresponding entry into the downsampled result. This variant is called *max-pooling* and is illustrated with an example in Fig. 8.2. In this example, the pooling size is chosen to be 2×2 and the image on the left side is divided into sections of this size. The respective maximum is transferred to the resulting image on the right and can be processed further. Here, it becomes clear that in a CNN the presence of certain structures or patterns should be recognized and the position of these within an image is not interesting for the classification.

Another alternative is that within the matrix the average value is formed and used for the resulting image.

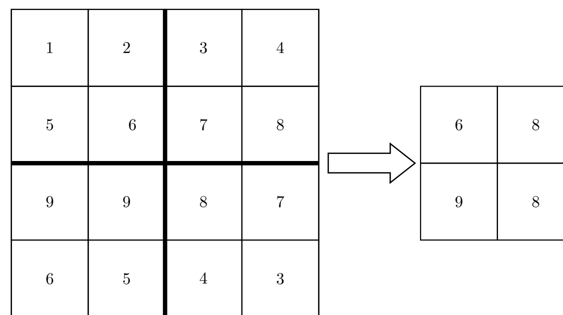


Figure 8.2: Example of max-pooling

8.2.1.3 Fully-connected Layer

With the help of fully-connected layers, the recognized patterns from the input image by the convolutional and pooling layers are classified as shown in Fig. 8.1. This type of layer is also the basis for previously mentioned ANNs. Any number of layers with different numbers of neurons can be used for classification in the CNN. The activation function can also be varied, whereby the *softmax* activation function is frequently used for the last layer, i.e. the output layer.

8.2.2 Complexity

For Artificial Neural Networks, the computational costs of processing images is very high. This is due to the fact that an image with a certain height and width is forwarded pixel-by-pixel from the input layer to the first hidden layer. This means that if a low-resolution image with, for example, a width and height of 32 pixels each is processed, $32 \cdot 32 = 1024$ weights per neuron of the first

hidden layer are required. Even if the complexity for the further processing of the network can be reduced with the appropriate choice of further hidden layers, the high computational effort for the first hidden layer remains. Further parameters of the depth, such as the colour or temporally framed images, lead to a further increase of the complexity. For the CNN, on the other hand, this computational effort is massively reduced due to the convolutional layer.

In addition, overfitting is reduced for CNNs. Overfitting arises because many free parameters cannot be adjusted in a generally valid way on the basis of training data and thus overfitting to these data arises. A general classification for independent test data is only possible with a significantly lower accuracy. For this reason, the influence of overfitting is therefore also reduced for CNNs in contrast to ANNs for the processing of images.

8.3 Exercise

Task 1 - Comprehensive Questions

In this task we will start with some basic properties of convolutional neural networks. In this section you don't need to program anything.

a) Convolve the image in Fig. 8.3 with the given kernel. For simplification, process the convolution only for complete overlap that the resulting image has a dimension of 2×2.
What is the effect of this kernel used? How has the resulting image changed in contrast to the input image?

b) How must the kernel look like in general in order to blur the image by convolution?

c) Use an average-pooling with a pooling size of 2×2 to downsample the image given in Fig. 8.3.

Image

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

Kernel

0	−1	0
−1	7	−1
0	−1	0

Figure 8.3: Convolution of image and kernel - Task 1

Task 2 - Analyse Dataset

In this task we will start with the analysis of the dataset that will be used for later task.

a) Open the notebook *cnn_task_2.ipynb*.

b) Import the following packages:

  – `tensorflow as tf`
  – `matplotlib.pyplot as plt`

c) Load the MNIST dataset provided by Keras[19] by `tensorflow.keras.datasets.mnist`. Then split the dataset into training and test data with `load_data()`.

d) Analyse the dataset.

  i) Determine and output how many training and test data are available each.
  ii) In which format and with which dimensions are the images and labels available?
  iii) How many different classes are available? Store this value while the initialization into a variable called `num_classes`.
  iv) Plot 9 examples in one figure with subplots. Use the given source code for this purpose. Complete the calls `plt.imshow(...)` and `plt.xlabel(...)` with the corresponding variables.

e) Normalise the images so that the colours are scaled to a maximum of 1. With which value do you have to scale? Plot the same examples as before with the given code. Add again the missing variables.

Task 3 - Train Neural Network

In this task we will train a neural network to classify the different digits based on the preprocessed images.

a) Open the notebook *cnn_task_3.ipynb*.

b) Import the following packages:

  – `tensorflow as tf`
  – `matplotlib.pyplot as plt`
  – `numpy as np`
  – `Sequential` from `tensorflow.keras.models`
  – `Dense` from `tensorflow.keras.layers`
  – `Flatten` from `tensorflow.keras.layers`
  – `convert_variables_to_constants_v2` from `tensorflow.python.framework.convert_to_constants`

Task 3 - Train Neural Network

c) Copy all code parts from task 2 that are necessary for loading and preprocessing the dataset used. You don't need the plots and outputs.

d) Define a neural network with one input, one hidden and one output layer. Use `model = tf.keras.Sequential()` for this. Add the individual layers with `model.add (...)`.

   i) For the input layer, use a flatten layer `Flatten(...)` with an `input_shape = ...` as you have determined in task 2.

   ii) For the hidden layer, use a dense layer `Dense(...)` with 256 neurons and *relu* as activation function with `activation = ...`.

   iii) Use a dense layer with *softmax* activation function as output layer. How many neurons do you need for this layer?

   iv) Describe mathematically the two different activation functions used. Describe in which cases each is used.

e) Compile the model using the given code.

f) Train the model with `model.fit(...)` with the training data and the initialised values for `batch_size = ...` and `epochs = ...`. Describe what these two parameters say and do.

g) Evaluate the model with `model.evaluate(...)` with the test data and output the values for loss and accuracy.

h) Store the model with the given code.

Task 4 - Train Convolutional Neural Network

In this task we will train a convolutional neural network to classify the different digits based on the preprocessed images.

a) Open the notebook *cnn_task_4.ipynb*.

b) Import the following packages:

- `tensorflow as tf`
- `matplotlib.pyplot as plt`
- `numpy as np`
- `Sequential` from `tensorflow.keras.models`
- `Dense` from `tensorflow.keras.layers`
- `Flatten` from `tensorflow.keras.layers`
- `Con2D` from `tensorflow.keras.layers`
- `MaxPooling2D` from `tensorflow.keras.layers`
- `convert_variables_to_constants_v2` from `tensorflow.python.framework.convert_to_constants`

c) Copy all code parts from task 2 that are necessary for loading and preprocessing the dataset used. You don't need the plots and outputs.

d) Define a convolutional neural network with two convolutional, two pooling and two fully-connected layers. Use `model = tf.keras.Sequential()` for this. Add the individual layers with `model.add(...)`.

   i) For the input layer, use a convolutional layer `Conv2D(...)` with an `input_shape = ...` as you have determined in task 2 and a `kernel_size = ...` of 3×3. Use *relu* as activation function with `activation = ...` and 32 neurons.

   ii) Use a max-pooling layer `MaxPooling2D(...)` with a `pool_size = ...` of 2×2.

   iii) Use a second convolutional layer with 64 neurons, a `kernel_size = ...` of 3×3 and *relu* as activation function.

   iv) Use a second max-pooling layer with the same settings like the first one.

   v) Define a flatten layer to flatten the output of the pooling layer.

   vi) Use a dense layer with 32 neurons and *relu* as activation function.

   vii) Use a dense layer as output layer with *softmax* as activation function.

   viii) Research which typical CNN architecture this network is based on. Name two other possible architectures and describe their basic structure.

e) Compile the model using the given code.

f) Train the model with `model.fit(...)` with the training data and the initialised values for `batch_size = ...` and `epochs = ...`.

g) Evaluate the model with `model.evaluate(...)` with the test data and output the values for loss and accuracy.

Task 4 - Train Convolutional Neural Network

h) Store the model with the given code.

Task 5 - Comparison of the Results

In this task we will compare the results and the complexity of the neural network and the convolutional neural network.

a) Compare the resulting accuracies for both networks.

b) Compare the structure of both networks. Use `model.summary()` and *Tensorboard* for this. Use the hint below to start *Tensorboard*.

c) Assess the complexity and computational effort in terms of the number of layers and the free parameters.

```
# Use these commands to start Tensorboard out of cmd from your lab directory
import_pb_to_tensorboard.py --model_dir .\result\trained_nn\frozen_models\
  simple_graph.pb --log_dir .\logs

tensorboard --logdir=.\logs

# Then copy http://localhost:6006/ to your browser to start tensorboard
```

Part V
System Aspects

Lab *Machine Learning*

Chapter 9

Application Example

In this chapter, students independently apply the concepts learned in the previous chapters to various machine learning problems. There are five problems to choose from, and students are expected to work on at least one of them. Working on a problem involves investigating and preprocessing the data set, selecting and training appropriate machine learning methods, and analyzing and comparing the results obtained. Finally, students will briefly present their results to share their experiences with their fellow students.

9.1 **General Information** 91

9.2 **Exercises** 92

9.3 **Example Datasets** 93

9.1 General Information

In the previous chapters, you learned about various machine learning and preprocessing algorithms. In this lab, you will have the opportunity to apply these concepts to known data sets to reinforce what you have learned.

This lab will cover two days. On the first day, you will examine datasets and train your algorithms. On the second day, you will be asked to give a short presentation on your results to share your experience with your fellow students.

To give you the freedom to explore the aspects of machine learning that are of particular interest to you, this lab includes mandatory and optional assignments. In addition, the datasets presented at the end of this chapter are only examples. If you would like to work on a different problem that is not listed here, please contact your lab supervisor.

9.2 Exercises

Mandatory Tasks

The following tasks must be completed by all students.

a) Choose at least one data set. This can be one of the examples below or a dataset and problem of your choice (assuming your supervisor agrees with your choice).

b) Choose at least one machine learning algorithm that is appropriate for your chosen problem.

c) Examine your data set. Calculate statistical values, try to find correlations between features, and visualize the data.

d) Train your algorithm on your data set:

   - Examine at least three different hyperparameters using grid search, random search, or Bayesian optimization. Report the training and testing results (the metric to report is listed in the appropriate problem chapter). Explain the impact of the hyperparameters studied.
   - Use different preprocessing methods, if applicable, and investigate their impact on performance.

e) Report on your work in a 10-minute presentation. Your presentation should include:

   - A brief introduction to the data set you are studying.
   - A brief introduction to the machine learning algorithm you used and why you chose it.
   - The results of your hyperparameter tuning including training and testing results.
   - Any preprocessing techniques you used and their impact on the algorithm's performance.
   - Your assessment of whether the algorithm is appropriate for the problem.
   - Your assessment of any further steps that could be taken to improve the performance of your algorithm for your dataset.

Optional Tasks

The following tasks are examples of what else you might investigate and report.

f) Choose a second data set and compare the performance of your algorithm on the two data sets.

g) Choose a second machine learning algorithm and compare the algorithms performance with your dataset.

h) Investigate more hyperparameters.

i) Try different data augmentation techniques and investigate their impact on the algorithms' performance.

9.3 Example Datasets

The following section contains a brief overview of the sample datasets, the tasks you are supposed to solve for them, and hints on what to look out for. You can find a more detailed description on the corresponding website. Please note that some of these tasks are more difficult than others.

9.3.1 CIFAR-10 and CIFAR-100 Dataset

The CIFAR-10 and CIFAR-100 Datasets [20] consist of labeled natural images.

The CIFAR-10 dataset contains 60000 32x32 pixel images of 10 classes, of which 50000 are training images and 10000 are test images. The CIFAR-100 dataset contains the same number of images but distributed among 100 different classes grouped into 20 superclasses.

When you train your machine learning algorithm on the training images, the goal is to achieve the highest possible accuracy on the test images.

Since this is a classification problem with multiple classes, looking at the confusion matrix can help you better understand the shortcomings of your algorithm. Also, looking directly at some of the misclassified images might help you improve the performance of your algorithm.

Depending on your algorithm, you may want to try different regularization techniques, such as weight decay, if you are having problems with overfitting.

Using data augmentation techniques, such as flipping, shifting, or cropping the training images, can also improve test accuracy.

9.3.2 UCI Parkinsons Dataset

The UCI Parkinsons Dataset [21] contains biomedical voice measurements from 31 individuals.

Its goal is to predict whether the speaker has Parkinson's disease or not based on the characteristics of 195 voice recordings. Since the dataset is relatively small, you can use cross-validation in hyperparameter tuning to obtain more meaningful results.

For medical applications, accuracy isn't the only thing that matters. You should also take a look

at the precision and recall of your algorithm. Which do you think is more important and why? Consider how you can tune your algorithm to favor one over the other.

Some of the features in this dataset may be familiar to you, while others may not. Make sure you understand all the features before you start using the data. How are they calculated and what is their significance?

Investigate whether all the features are equally important in predicting the speaker's health status. Are there correlations between them? Does the performance of your algorithm decrease if you omit some of the features?

9.3.3 German Traffic Sign Detection Benchmark Dataset

The GTSDB Dataset [22] contains 900 images, of which 600 are training images and 300 are test images. Each of these images contains zero to six traffic signs.

The goal of your algorithm is to find these traffic signs. The metric you want to maximize for this is *Itersection over Union* (IoU).

After you have found the road signs, you can optionally classify them using the provided training labels.

9.3.4 German Traffic Sign Recognition Benchmark Dataset

The GTSRB Dataset [23] contains 50000 images of 40 different traffic signs.

The objective for this dataset is very similar to that of CIFAR-10 / CIFAR-100, so all points mentioned there also apply to this dataset.

In addition, the resolution of the images varies from 15x15 to 250x250 pixels, so you have to find a way to deal with that.

Look at the confusion matrix and consider the significance of the misclassified images. Some misclassifications might be worse than others when automatically recognizing traffic signs in a car.

9.3.5 Boston Housing Dataset

The Boston Housing Dataset [24] contains 506 cases of housing information for the Boston Mass area. This information includes per capita crime rates, average number of rooms per dwelling, and student to teacher ratios by city among others.

There are two goals for this dataset. The first is to predict the mean value of homes and the second is to predict nitrogen oxide concentrations.

Since this data set is quite small, it is recommended to perform cross-validation to obtain more meaningful results.

Additionally, the same advice applies as for the UCI-Parkinsons Dataset. Look closely at the features. How are they calculated? How important are they? Are there any correlations?

Part VI
Authors

Lab *Machine Learning*

Chapter 10

Authors

The following people contributed to this lab:

- Gerhard Schmidt
- Rasool Al-Mafrachi
- Christin Bald
- Eric Elzenheimer
- Erik Engelhardt
- Johannes Hoffmann
- Tobias Hübschen
- Bastian Kaulen
- Frederik Kühne
- Patricia Piepjohn
- Robbin Romijnders
- Tim Owe Wisch

Part VII
References

Lab *Machine Learning*

Chapter 11

References

[1] F.R.S., Karl P.: LIII. On lines and planes of closest fit to systems of points in space. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2 (1901), Nr. 11, 559-572. http://dx.doi.org/10.1080/14786440109462720. – DOI 10.1080/14786440109462720

[2] Hotelling, Harold: Analysis of a complex of statistical variables into principal components. In: *Journal of educational psychology* 24 (1933), Nr. 6, S. 417

[3] Jolliffe, Ian T.: *Principal component analysis for special types of data*. Springer, 2002

[4] FISHER, R. A.: THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS. In: *Annals of Eugenics* 7 (1936), Nr. 2, 179-188. http://dx.doi.org/https://doi.org/10.1111/j.1469-1809.1936.tb02137.x. – DOI https://-doi.org/10.1111/j.1469–1809.1936.tb02137.x

[5] Rao, C. R.: The Utilization of Multiple Measurements in Problems of Biological Classification. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 10 (1948), Nr. 2, 159–203. http://www.jstor.org/stable/2983775. – ISSN 00359246

[6] Shlens, Jonathon: *A Tutorial on Independent Component Analysis*. 2014

[7] Hyvärinen, A. ; Oja, E.: Independent component analysis: algorithms and applications. In: *Neural Networks* 13 (2000), Nr. 4, S. 411–430. http://dx.doi.org/https://doi.org/10.1016/S0893-6080(00)00026-5. – DOI https://doi.org/10.1016/S0893–6080(00)00026–5. – ISSN 0893–6080

[8] Semmlow, J. L. ; Griffel, B.: *Biosignal and Medical Image Processing*. 3. CRC Press, 2014. http://dx.doi.org/https://doi.org/10.1201/b16584. http://dx.doi.org/https://doi.org/10.1201/b16584

[9] Hyvärinen, A.: Fast and robust fixed-point algorithms for independent component analysis. In: *IEEE Transactions on Neural Networks* 10 (1999), Nr. 3, S. 626–634. http://dx.doi.org/10.1109/72.761722. – DOI 10.1109/72.761722

[10] Hyvärinen, A. ; Karhunen, J. ; Oja, E.: *Independent component analysis*. John Wiley & Sons Inc, 2001

[11] *Tensorflow*. https://www.tensorflow.org/tutorials

[12] *Tutorials : Tensorflow Core*. https://www.tensorflow.org/tutorials

[13] *Tensorflow activation functions*. https://www.tensorflow.org/api_docs/python/tf/keras/activations

[14] *Keras: Making new Layers and Models via subclassing*. https://www.tensorflow.org/guide/keras/custom_layers_and_models

[15] Xiao, Han ; Rasul, Kashif ; Vollgraf, Roland: *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017

[16] Boser, Bernhard E. ; Guyon, Isabelle M. ; Vapnik, Vladimir N.: A Training Algorithm for Optimal Margin Classifiers. In: *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT'92)*, ACM Press, July 1992, S. 144–152

[17] Jana, Abhisek: *Support Vector Machines for Beginners Linear SVM*. http://www.adeveloperdiary.com/data-science/machine-learning/support-vector-machines-for-beginners-linear-svm/. Version: 2020

[18] Yadav, Ajay: *SUPPORT VECTOR MACHINES(SVM)*. https://towardsdatascience.com/support-vector-machines-svm-c9ef22815589. Version: 2018

[19] *MNIST digits classification dataset*. https://keras.io/api/datasets/mnist/

[20] Krizhevsky, Alex ; Hinton, Geoffrey u. a.: Learning multiple layers of features from tiny images. (2009)

[21] Little, Max ; Mcsharry, Patrick ; Roberts, Stephen ; Costello, Declan ; Moroz, Irene: Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. In: *Nature Precedings* (2007), S. 1–1

[22] Houben, Sebastian ; Stallkamp, Johannes ; Salmen, Jan ; Schlipsing, Marc ; Igel, Christian: Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark. In: *Proceedings of the International Joint Conference on Neural Networks* (2013), 08. http://dx.doi.org/10.1109/IJCNN.2013.6706807. – DOI 10.1109/IJCNN.2013.6706807

[23] Stallkamp, J. ; Schlipsing, M. ; Salmen, J. ; Igel, C.: Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. In: *Neural Networks* (2012), Nr. 0, -. http://dx.doi.org/10.1016/j.neunet.2012.02.016. – DOI 10.1016/j.neunet.2012.02.016. – ISSN 0893–6080

[24] Harrison Jr, David ; Rubinfeld, Daniel L.: Hedonic housing prices and the demand for clean air. In: *Journal of environmental economics and management* 5 (1978), Nr. 1, S. 81–102

[25] O'Shea, Keiron ; Nash, Ryan: An Introduction to Convolutional Neural Networks. In: *CoRR* abs/1511.08458 (2015). http://arxiv.org/abs/1511.08458

[26] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[27] Huber, P. J.: Projection Pursuit. In: *The Annals of Statistics* 13 (1985), Nr. 2, 435475. http://www.jstor.org/stable/2241175. – ISSN 0893–6080

[28] Hyvärinen, A.: New Approximations of Differential Entropy for Independent Component Analysis and Project Pursuit. In: *Advances in Neural Information Processing Systems*. Cambridge, MA : MIT Press, 1998, S. 273–279

[29] Karhunen, J. ; Oja, E. ; Wang, L. ; Vigario, R. ; Joutsensalo, J.: A class of neural networks for independent component analysis. In: *IEEE Transactions on Neural Networks* 8 (1997), Nr. 3, S. 486–504. http://dx.doi.org/10.1109/72.572090. – DOI 10.1109/72.572090

[30] Luenberger, D. G.: *Optimization by vector space methods*. New York : Wiley, 1969

[31] Schomer, Donald L. ; Silva, Fernando L.: *Niedermeyer's Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*. 6. Lippincott Williams & Wilkins, 2011