Hands-On Quantum Machine Learning With Python

Dr. Frank Zickert

PyQML

PUBLISHED BY PYQML

www.pyqml.com

Pre-Release 0.1, September 2020

Contents

1. Introduction

Welcome to **Hands-On Quantum Machine Learning With Python**. This book is your comprehensive guide to get started with "*Quantum Machine Learning*" – the use of **quantum computing** for computation of **machine learning** algorithms.

Hands-On Quantum Machine Learning With Python strives to be the perfect balance between theory taught in a textbook and the actual hands-on knowledge you'll need to implement real-world solutions.

Inside this book, you will learn the basics of quantum computing and machine learning in a practical and applied manner. And you will learn to use state-of-the-art quantum machine learning algorithms.

By the time you finish this book, you'll be well equipped to apply quantum machine learning to your projects. You will be in the pole position to become a "*Quantum Machine Learning Engineer*" – the job to become the sexiest job of the 2020s.

1.1 Who This Book Is For

This book is for **developers**, **programmers**, **students**, and **researchers** who have at least some programming experience and who want to become proficient in quantum machine learning.

Don't worry if you're just getting started with quantum computing and machine learning. We will begin with the very basics. We don't assume prior knowledge of machine learning or quantum computing. You will not get left behind.

If you are already experienced in machine learning or quantum computing, the respective parts may be a repetition of concepts you're already familiar with. However, this may make learning the respective new topic easier and provide a slightly different angle to the known.

This book offers a practical, hands-on exploration of quantum machine learning. Rather than working through tons of theory, we will build up practical intuition about the core concepts. We will acquire the exact knowledge we need to solve practical examples with lots of code. Step by step, you will extend your knowledge and learn how to solve new problems.

Of course, we will do some math. Of course, we will cover a little physics. But I don't expect you to hold a degree in any of these two fields. We will go through all the concepts we need. While this includes some mathematical notation and formulae, we keep it at the minimum required to solve our practical problems.

The theoretical foundation of quantum machine learning may appear overwhelming at first sight. Be assured, when put into the right context and when explained conceptually, it is not harder than learning a new programming language. And this is what's inside **Hands-On Quantum Machine Learning With Python**.

Of course, we will write code. A lot of code, actually. If you know a little Python, great! If you don't know Python but another language, such as Java, Javascript, or PHP, you'll be fine, too. If you know programming concepts (such as `if then else`-constructs and loops) then learning the syntax is a piece of cake. If you're familiar with functional programming constructs, such as `map`, `filter`, and `reduce`, you're already well equipped. If not, don't worry, we will get you started with these constructs, too. We don't expect you to be a senior software developer. We will go through all the code. Line by line.

By the time you finish the first few chapters of this book, you will be proficient with doing the math, understanding the physics, and writing the code you need to graduate to the more advanced content.

This book is not just for beginners. There is a lot of advanced content in here, too. Many chapters of **Hands-On Quantum Machine Learning With Python** cover, explain, and apply quantum machine learning algorithms developed in the last two years. The insights this book provides can be directly applied in your job and research. The time you'll save by reading through **Hands-On Quantum Machine Learning With Python** will more than pay for itself.

1.2 Book Organization

Machine learning and quantum computing rely on math, statistics, physics, and computer science. This a lot of theory. Covering it all upfront would be quite exhaustive and fill at least one book without any practical insight.

However, without at least some understanding of the underlying theoretical concepts, the code examples on their own do not provide many practical insights, either. While libraries free you from tedious implementation details, the code, even though short, does not explain the core concepts.

This book provides the theory needed to understand the code we're writing to solve a problem. We cover the theory when it applies and when we need the background to understand what we are doing. We will embed the theory into solving a practical problem and thus, directly see it in action.

As a result, the theory spreads among all the chapters. From simple to complex. You may skip certain examples if you like. But you should have a look at the theoretical concepts discussed in each chapter.

In this pre-release, we start with a **Variational Hybrid Quantum-Classical Algorithm** to solve a binary classification task. First, we have a detailed look at binary classification in chapter 2. Then, in chapter 3, we introduce the basic concept of the quantum bit, the quantum state, and how measurement affects it. Based on these concepts, we build our first **Parameterized Quantum Circuit** and use it to solve our binary classification task. Such a hybrid algorithm combines the quantum state preparation and measurement with classical optimization.

1.3 Why Should I Bother With Quantum Machine Learning?

In the recent past, we have witnessed how algorithms learned to drive cars and beat world champions in chess and Go. Machine learning is being applied to virtually every imaginable sector, from military to aerospace, from agriculture to manufacturing, and from finance to healthcare.

But these algorithms become increasingly hard to train because they consist of billions of parameters. Quantum computers promise to solve such problems intractable with current computing technologies. Their ability to compute multiple states simultaneously enables them to perform an indefinite number of superposed tasks in parallel. An ability that promises to improve and to expedite machine learning techniques.

Unlike classical computers that are based on sequential information processing, quantum computing makes use of the properties of quantum physics. Superposition, entanglement, and interference. But rather than increasing the available computing capacity, it reduces the capacity needed to solve a problem.

But quantum computing requires us to change the way we think about computers. It requires a whole new set of algorithms. Algorithms that encode and use quantum information. This includes machine learning algorithms.

And it requires a new set of developers. Developers who understand machine learning and quantum computing. Developers capable to solve practical problems that have not been solved before. A rare type of developer. The ability to solve quantum machine learning problems today already sets you apart from all the others.

Quantum machine learning promises to be disruptive. Although this merger of machine learning and quantum computing, both areas of active research, is largely in the conceptual domain, there are already a number of examples where it is being applied to solve real-life problems. Google, Amazon, IBM, Microsoft, and a whole fleet of high-tech startups strive to be the first to build and sell quantum machine learning systems.

The opportunity to study a technology right at the moment when it is about to prove its supremacy is a unique opportunity. Don't miss it.

1.4 Quantum Machine Learning - Beyond The Hype

If there were two terms in computer science that I would describe as overly hyped and poorly understood, I would say *machine learning* and *quantum computing*.

That said, **Quantum Machine Learning** is the use of *quantum computing* for computation of *machine learning* algorithms. Could it be any worse?

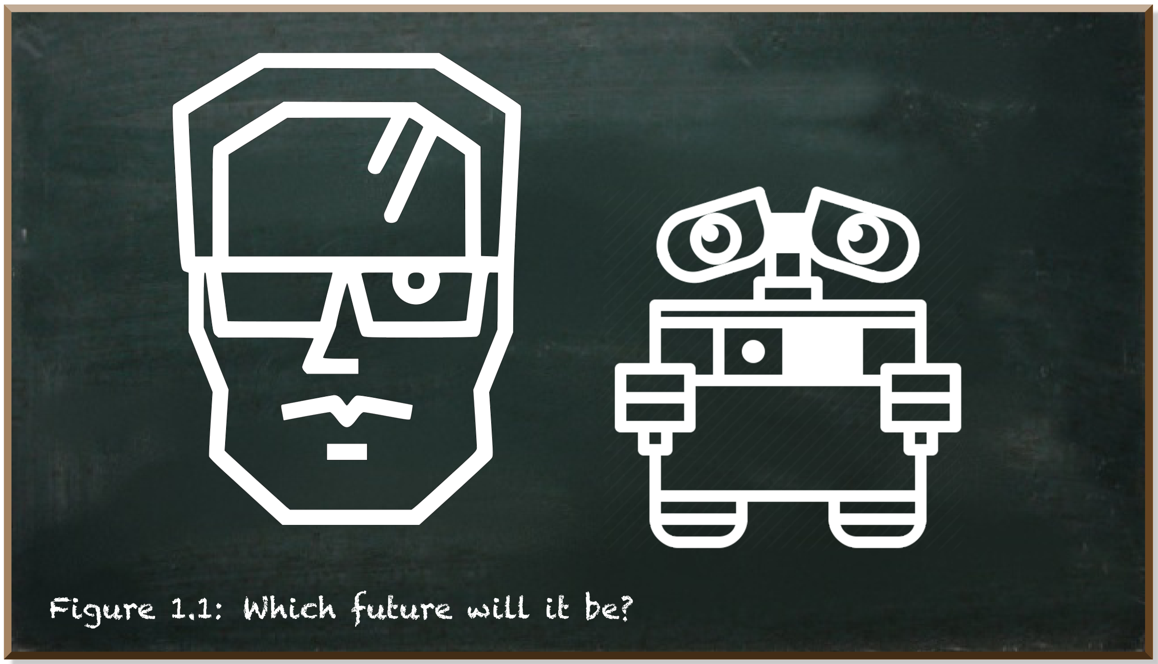


Figure 1.1: Which future will it be?

There are a lot of anecdotes on these two technologies. They start at machines that understand the natural language of us humans. And they end at the advent of the Artificial General Intelligence that either manifests as the Terminator-like apocalypse or the Wall-E-like utopia.

Don't fall for the hype! An unbiased and detailed look at a technology helps not to fall for the hype and the folklore. Let's start with machine learning.

1.4.1 What is Machine Learning?

"Machine learning is a thing-labeler, essentially"

– Cassie Kozyrkov, Chief Decision Scientist at Google, source –

With machine learning, we aim to put a label onto a yet unlabeled thing. And there are three main ways of doing it: classification, regression, and segmentation.

In **classification**, we try to predict the discrete label of an instance. Given the input and a set of possible labels, which one is it? Here's a picture. Is it a cat or a dog?

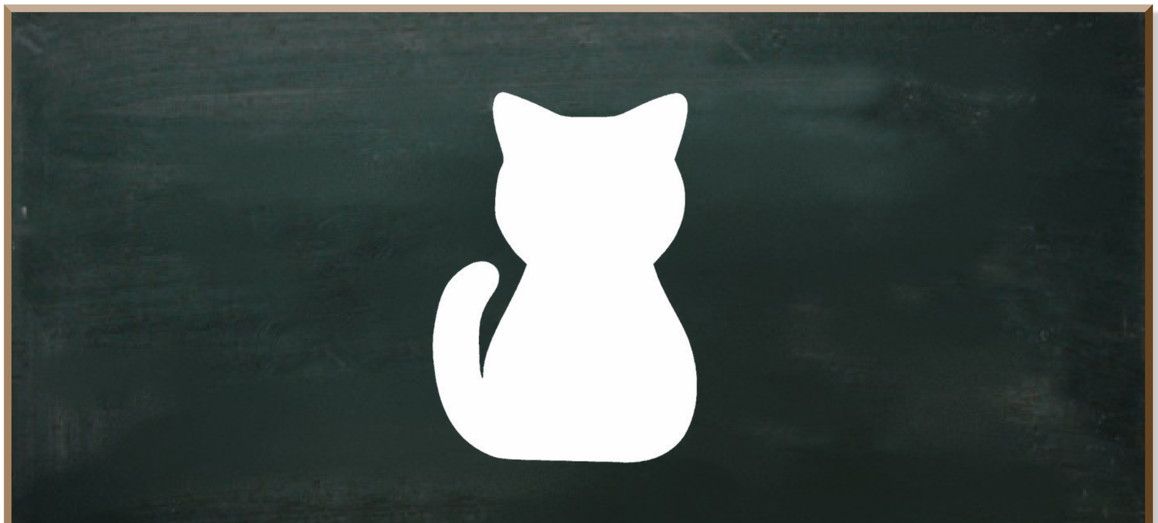


Figure 1.2: Is it a cat or a dog?

Regression is about finding a function to predict the relationship between some input and the dependent continuous output value. Given you know the income and the effective tax rates of your friends, can you estimate your tax rate given your income even though you don't know the actual calculation?

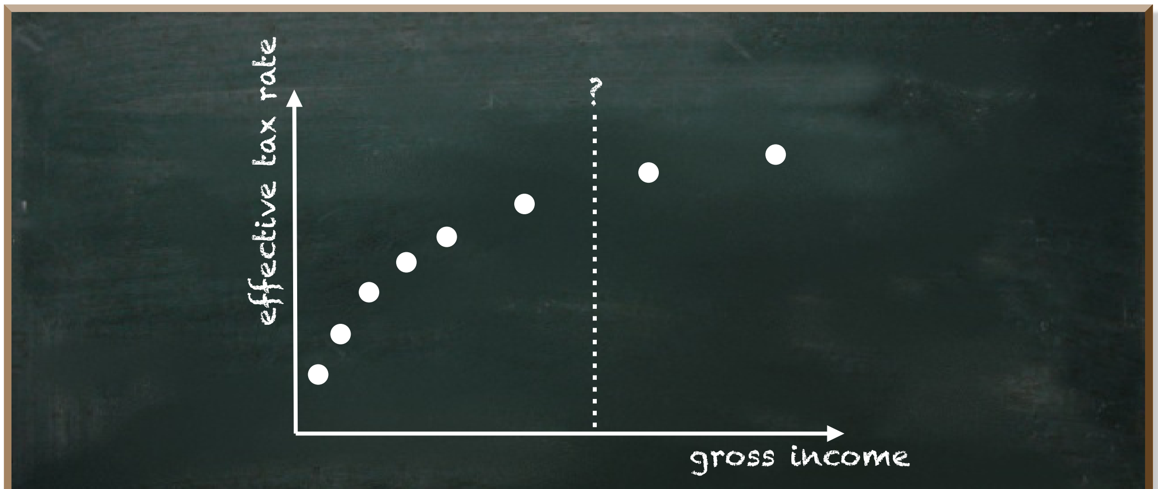


Figure 1.3: Effective tax rate by gross income

And **segmentation** is the process of partitioning the population into groups with similar characteristics who are thus likely to exhibit similar behavior. Given you produce an expensive product, such as yachts, and a population of potential customers, who do you want to try to sell to?

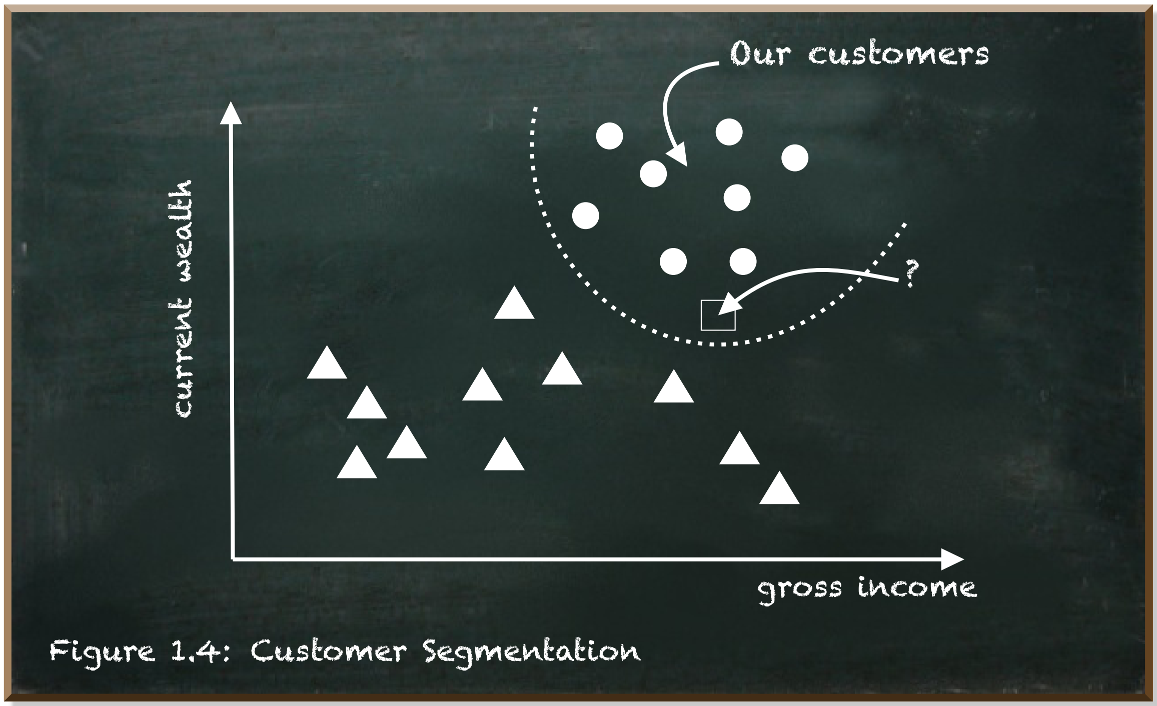


Figure 1.4: Customer Segmentation

1.4.2 What is Quantum Computing?

Quantum computing is a different form of computation. It uses three fundamental properties of quantum physics: superposition, interference, and entanglement.

Superposition refers to the quantum phenomenon where a quantum system can exist in multiple states concurrently.

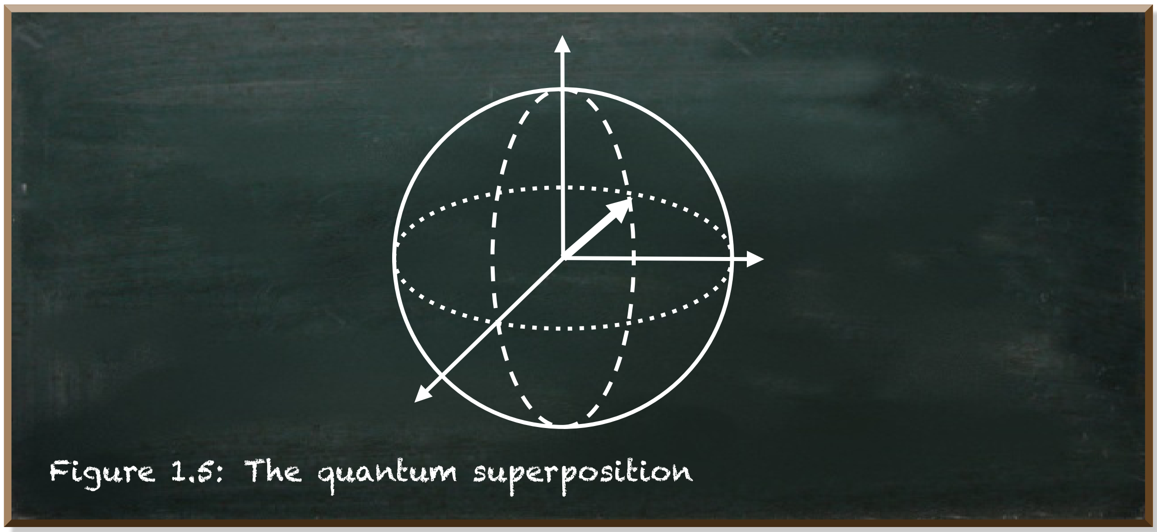


Figure 1.5: The quantum superposition

Actually, the quantum system does not exist in multiple states concurrently. It exists in a complex linear combination of a state 0 and a state 1. It is a different kind of combination that is neither ”or” nor is it ”and”. We will explore this state in depth in this book.

Quantum **interference** is what allows us to bias quantum systems toward the desired state. The idea is to create a pattern of interference where the paths leading to wrong answers interfere destructively and cancel out but the paths leading to the right answer reinforce each other.

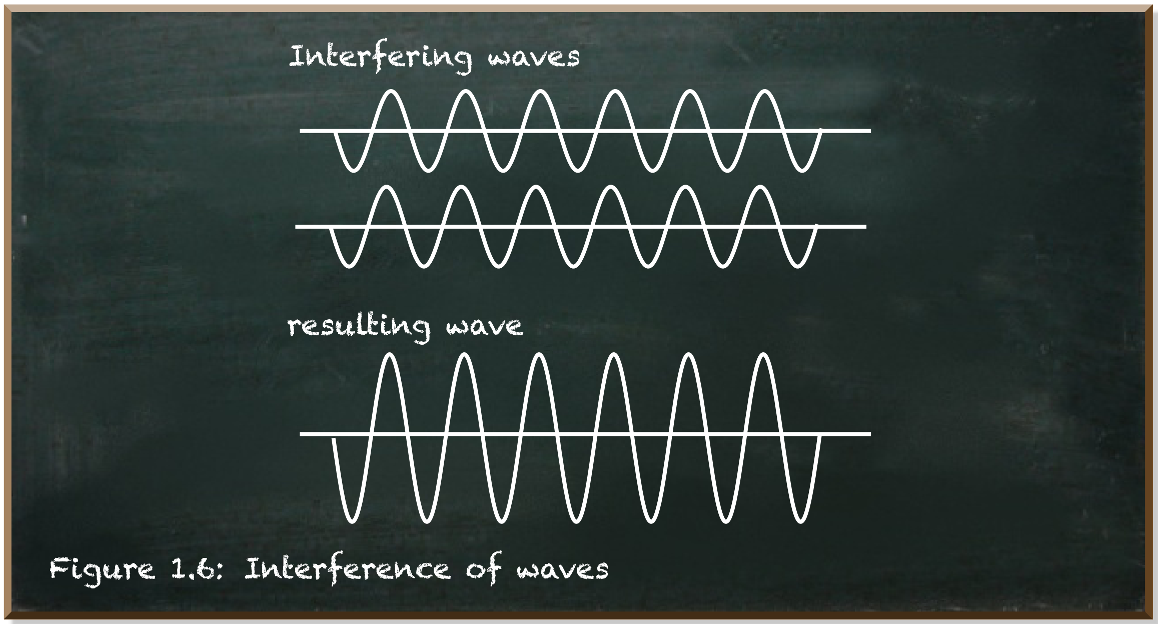


Figure 1.6: Interference of waves

Entanglement is an extremely strong correlation between quantum particles. Entangled particles remain perfectly correlated even if separated by great distances.

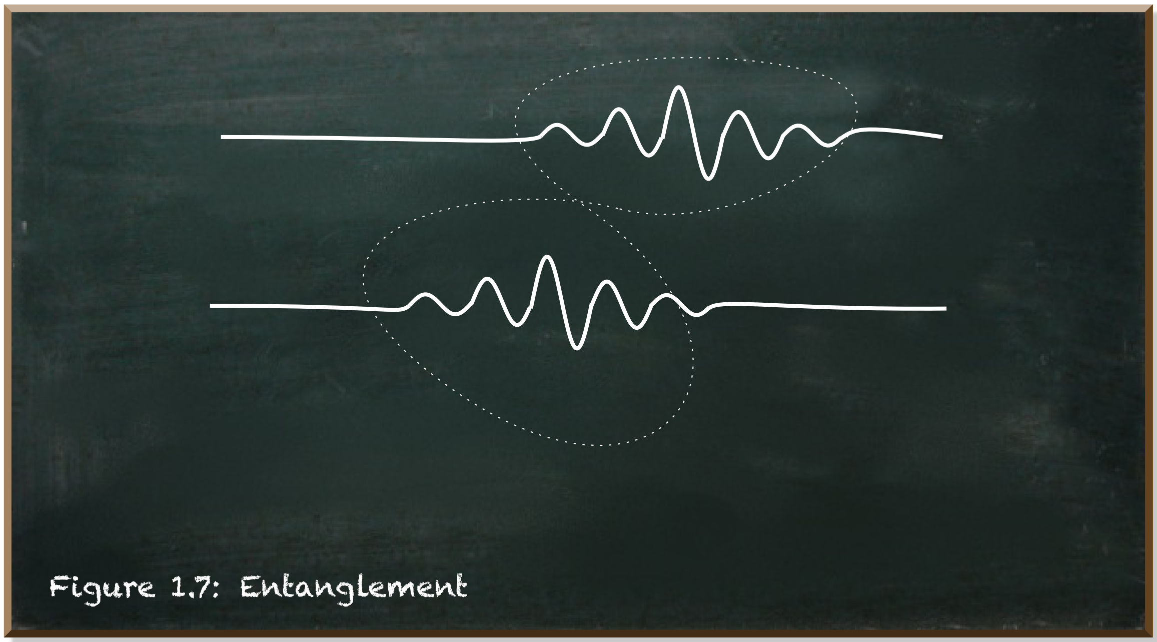


Figure 1.7: Entanglement

Do you see the Terminator already? No?

Maybe Wall-E? No *again*?

Maybe it helps to look at how these things work.

1.4.3 How Does Machine Learning Work?

There are myriads of machine learning algorithms out there. But every one of these algorithms has three components:

- The **Representation** depicts the inner architecture the algorithm uses to represent the knowledge. It may consist of a set of rules, instances, decision trees, support vector machines, neural networks, and others.
- The **Evaluation** is a function to evaluate candidate algorithm parameterizations. Examples include accuracy, prediction and recall, squared error, posterior probability, cost, margin, entropy, and others.
- The **Optimization** describes the way of generating candidate algorithm parameterizations. It is known as the search process. For instance, combinatorial optimization, convex optimization, and constrained optimization.

The first step of machine learning is the development of an architecture, the representation. The architecture specifies the parameters whose values hold the representation of the knowledge. This step determines how suited the solution will be to solve a certain problem. More parameters is not always better. If our problem can be solved by a linear function, trying to solve it with a solution that consists of millions of parameters is likely to fail. On the other hand, an architecture with very few parameters may be insufficient to solve complex problems such as natural language understanding.

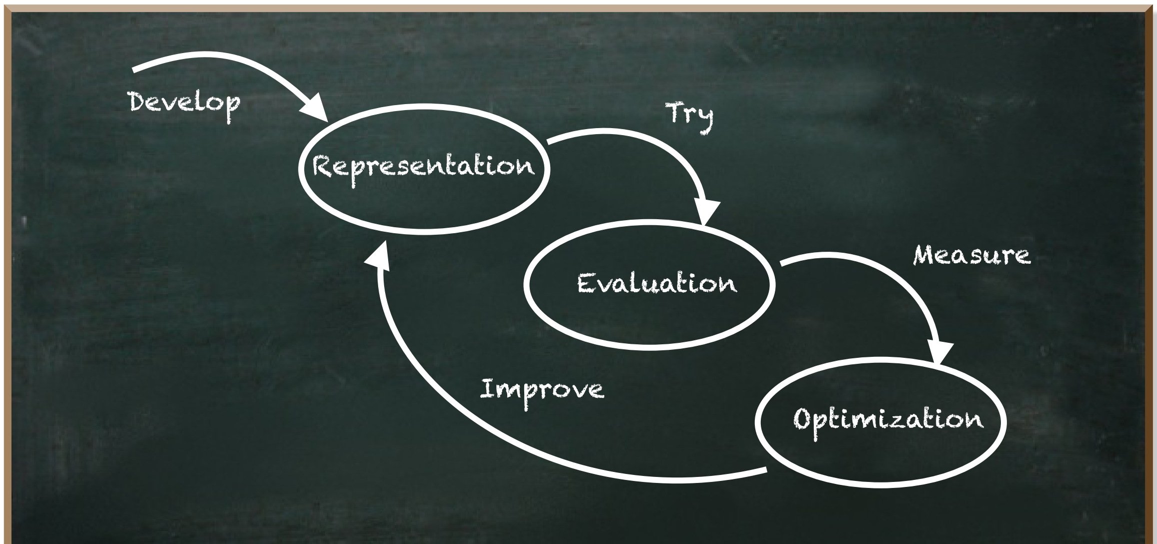


Figure 1.8: A generalized notion of machine learning

Once we settled for an architecture to represent the knowledge, we train our machine learning algorithm with examples. Depending on the number of parameters, we need a lot of examples. The algorithm tries to predict the label of each example. We use the evaluation function to measure how the algorithm performed.

Finally, the optimizer adjusts the representation parameters in a way that promises better performance with regard to the measured evaluation. It may even involve changing the architecture of the representation.

Learning does not happen in giant leaps. Rather in tiny steps. In order to yield a good performance and depending on the complexity of the problem, it takes several iterations of this general process until the machine is able to put the correct label at a thing.

1.4.4 What Tasks Are Quantum Computers Good At?

The world of quantum mechanics is different from the physics we experience in our everyday situations. So is the world of quantum computing different from classical (digital) computing.

What makes quantum computing so powerful isn't its processing speed. In fact, it is rather slow. What makes quantum computing so powerful isn't its memory, either. In fact, it is absurdly tiny. We're talking about a few quantum bits.

What makes quantum computing so powerful is the algorithms it makes possible. These algorithms exhibit different complexity characteristics than their classical equivalents.

In order to understand what that means, let's have a brief look at complexity theory. Complexity theory is the study of the computational effort required to run an algorithm.

For instance, the computational effort of addition is $\mathcal{O}(n)$. This means that the effort of adding two numbers increases linearly with the size (digits) of the number. The computational effort of multiplication is $\mathcal{O}(n^2)$. The effort increases by the square of the number size. These algorithms are said to be solvable in polynomial time.

But these problems are comparably simple. The best algorithm solving the problem of factorization, that is finding the prime factors of an n-digit number, is $\mathcal{O}(e^{n^{1/3}})$. It means that the effort increases exponentially with the number of digits.

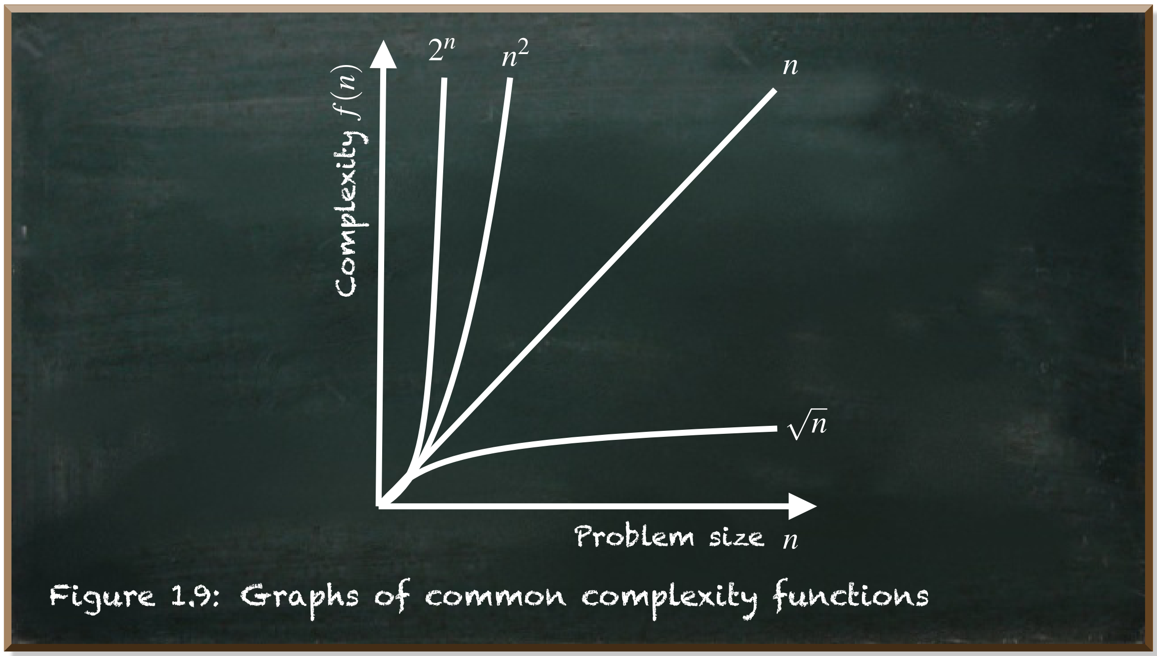


Figure 1.9: Graphs of common complexity functions

The difference between $\mathcal{O}(n^2)$ and $\mathcal{O}(e^{n^{1/3}})$ complexity must not be underestimated. While your smartphone is able to multiply numbers with 800 digits in a few seconds, the factorization of such numbers takes about 2,000 years on a supercomputer.

A savvy quantum algorithm (such as Shor's algorithm) can use superposition to evaluate all possible factors of a number simultaneously. And rather than calculating the result, it uses interference to combine all possible answers in a way that yields a correct answer. This algorithm solves a factorization problem with $\mathcal{O}\left((\log n)^2(\log\log n)(\log\log\log n)\right)$ complexity. This is a polynomial complexity! So is multiplication.

Quantum computing is powerful because it promises to solve certain types of mathematical calculations with reduced complexity.

Do you see the Terminator or Wall-E now? Not *yet*?

1.4.5 The Case For Quantum Machine Learning

Quantum machine learning is the use of quantum computing for computation of machine learning algorithms.

We have learned that machine learning algorithms contain three components: representation, evaluation, and optimization.

When we look at the representation, current machine learning algorithms, such as the *Generative Pre-trained Transformer* 3 (GPT-3) network, published in 2020, come to mind. GPT-3 produces human-like text. It has 175 billion parameters. The IBM Q quantum computer has 27 quantum bits. Even though quantum bits store a lot more information than a classical bit does (because it is not either `0` or `1`), quantum computers are far away from advancing machine learning for their representation ability.

During the evaluation, the machine learning algorithm tries to predict the label of a thing. Classically, this involves measuring and transforming data points. For instance, neural networks rely on matrix multiplications. These are tasks classical computers are good at. However, if you have 175 billion parameters, then calculating the resulting prediction takes quite a lot of matrix multiplications.

Finally, the algorithm needs to improve the parameters in a meaningful way. The problem is to find a set of parameter values that result in better performance. With 175 billions of parameters, the number of combinations is endless.

Classical machine learning employs heuristics that exploit the structure of the problem to converge to a sufficient solution within a reasonable time. Despite the use of even advanced heuristics, training the GPT-3 would require 355 years to train on a single GPU (Graphics Processing Unit) and cost $4.6

million. Just to get a feeling of what reasonable means in this context.

The main characteristic of quantum computing is the ability to compute multiple states concurrently. A quantum optimization algorithm can combine all possible candidates and yield those that promise good results. Therefore, quantum computing promises to be exponentially faster than classical computers in the optimization of the algorithm.

But this does not mean we only look at the optimization. Because the optimization builds upon running an evaluation. And the evaluation builds upon the representation. Thus, tapping the full potential of quantum computing to solve the machine learning optimization problem requires the evaluation and the representation to integrate with the quantum optimizer.

Having in mind what classical machine learning algorithms can do today, and if we expect quantum computing to reduce the complexity of training such algorithms by magnitudes, then the hype becomes understandable. Because we are "only" magnitudes away from things like Artificial General Intelligence.

But of course, building an Artificial General Intelligence requires more than the computation. It needs data. And it needs the algorithms.

The development of such algorithms is one of the current challenges in quantum machine learning. But there's another aspect to cope with in that challenge. That aspect is we are in the NISQ era.

1.5 Quantum Machine Learning In The NISQ Era

Quantum computing is a different form of computation. A form that, as we just learned, can change the complexity of solving problems making them tractable. But this different form of computation brings its own challenges.

Digital computers need to distinguish between two states: 0 and 1. The circuits need to tell the difference between high voltage and low voltage. Whenever there is high voltage, it is 1 and if there is lower voltage it is 0. This discretization means that errors must be relatively large to be noticeable and methods for detecting and correcting such errors can then be implemented.

Unlike digital computers, quantum computers need to be very precise. They keep a continuous quantum state. And quantum algorithms base on precise manipulations of continuously varying parameters. In quantum computers, however, errors can be arbitrarily small and impossible to detect, but still their effects can build up to ruin a computation.

This fragile quantum state is very vulnerable to the noise coming from the environment around the quantum bit. Noise can arise from control electronics, heat, or impurities in the quantum computer's material itself, and can also cause serious computing errors that may be difficult to correct.

But to keep the promises quantum computers make, we need fault-tolerant devices. We need devices to compute Shor's algorithm for factoring. We need devices to execute all the other algorithms that have been developed in theory that solve problems intractable for digital computers.

But such devices require millions of quantum bits. This overhead is required for error correction since most of these sophisticated algorithms are extremely sensitive to noise.

Current quantum computers have up to 27 quantum bits. Even though IBM strives for a 1000-quantum bits computer by 2023, the quantum processors we expect in the near-term will have between 50 and 100 quantum bits. Even if they exceed these numbers, they remain relatively small and noisy. These computers can only execute short programs since the longer the program is the more noise-related output errors will occur.

Nevertheless, programs that run on devices beyond 50 quantum bits become extremely difficult to simulate on classical computers already. These devices can do things infeasible for a classical computer.

And this is the era we're about to enter. The era when we can build quantum computers that, while not being fault-tolerant, can do things classical computers can't. The era is described by the term "Noisy Intermediate-Scale Quantum" - **NISQ**.

Noisy because we don't have enough qubits to spare for error correction. And "Intermediate-Scale" because of the number of quantum bits is too small to compute sophisticated quantum algorithms, but large enough to show quantum advantage or even supremacy.

The current era of NISQ-devices requires a different set of algorithms, tools, and strategies.

For instance, **Variational Quantum-Classical Algorithms** have become a popular way to think about quantum algorithms for near-term quantum devices. In these algorithms, classical computers perform the overall machine learning task on information they acquire from running certain hard-to-compute calculations on a quantum computer.

The quantum algorithm produces information based on a set of parameters provided by the classical algorithm. Therefore, they are called **Parameterized Quantum Circuits (PQCs)**. They are relatively small, short-lived, and thus suited for NISQ-devices.

Before we build our first **Variational Quantum-Classical Algorithm** that uses a **Parameterized Quantum Circuit**, let's configure our workstation.

1.6 Configuring Your Quantum Machine Learning Workstation

Even though this book is about quantum machine learning, I don't expect you to have a quantum computer at your disposal. Thus, we will run most of the code examples in a simulated environment on your local machine. But we will need to compile and install some dependencies first.

We will use the following software stack:

- Unix-based operating system (not required but recommended)
- Python, including pip
- Jupyter (not required but recommended)

1.6.1 Python

For all examples inside **Hands-On Quantum Machine Learning With Python**, we use Python as our programming language. Python is easy to learn. Its simple syntax allows you to concentrate on learning quantum machine learning, rather than spending your time with the specificities of the language.

Most importantly, machine learning tools, such as PyTorch and Tensorflow, as well as quantum computing tools, such as Qiskit and Cirq, are available as Python SDKs.

1.6.2 Jupyter

Jupyter notebooks are a great way to run quantum machine learning experiments. They are a de facto standard in the machine-learning and quantum computing communities.

A notebook is a file format (`.ipynb`). The Jupyter Notebook app lets you edit your file in the browser while running the Python code in interactive Python kernels. The kernel keeps the state in memory until it is terminated or restarted. This state contains the variables defined during the evaluation of code.

A notebook allows you to break up long experiments into smaller pieces you can execute independently. You don't need to rerun all the code every time you make a change. But you can interact with it.

1.6.3 Libraries and Packages

We will use the following libraries and packages:

- Scikit-learn
- PyTorch or Tensorflow
- Qiskit or Cirq

Scikit-learn is the most useful library for machine learning in Python. It contains a range of supervised and unsupervised learning algorithms. Scikit-learn builds upon a range of other very useful libraries, such as:

- NumPy: Work with n-dimensional arrays
- SciPy: Fundamental library for scientific computing
- Matplotlib: Comprehensive 2D/3D plotting
- IPython: Enhanced interactive console
- Sympy: Symbolic mathematics
- Pandas: Data structures and analysis

PyTorch and Tensorflow are the major libraries when it comes to deep neural networks. Qiskit is IBM's quantum computing SDK and Cirq is Google's.

1.6.4 Virtual Environment

Like most programming languages, Python has its own package installer. This is `pip`. It installs packages from the Python Package Index (PyPI) and other indexes.

By default, it installs the packages in the same base directory that is shared among all your Python projects. It makes an installed package available to all your projects. This seems to be good because you don't need to install the same packages over and over again.

However, if any two of your projects require different versions of a package, you'll be in trouble. Because there is no differentiation between versions. You would need to uninstall the one version and install the other whenever you switch working on either one of the projects.

This is where virtual environments come into play. Their purpose is to create an isolated environment for each of your Python projects. It's no surprise, using Python virtual environments is best practice.

1.6.5 Configuring Ubuntu For Quantum Machine Learning with Python

An Ubuntu Linux environment is highly recommended when working with quantum machine learning and Python because all the tools you need can be installed and configured easily.

Other Linux distributions (such as Debian) or MacOS (that also builds upon Unix) are also ok. But there are a few more aspects to consider.

All the code should work on Windows, too. However, the configuration of a Windows working environment can be a challenge on its own.

In this section, we go through the installation on Ubuntu Linux. If you're interested in a guide for another operating system, please have a look at my blog for I plan to add these guides.

We accomplish all steps by using the terminal. To start, open up your command line and update the `apt-get` package manager.

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ apt-get install -y build-essential wget python3-dev
```

The next step downloads and installs Python 3.8.5 (the latest stable release at the time of writing).

```
$ mkdir /tmp/Python38
$ cd /tmp/Python38
$ wget https://www.python.org/ftp/python/3.8.5/Python-3.8.5.tar.xz
$ tar xvf Python-3.8.5.tar.xz
$ cd /tmp/Python38/Python-3.8.5
$ ./configure
$ make altinstall
```

If you want to have this Python version as the default, run

```
$ ln -s /usr/local/bin/python3.8 /usr/bin/python
```

Python is ready to work. Let's now install and update the Python package manager pip:

```
$ wget https://bootstrap.pypa.io/get-pip.py && python get-pip.py
$ pip install --upgrade pip
```

As mentioned, we install all the Python packages in a virtual environment. So, we need to install `virtualenv`:

```
$ python -m pip install --user virtualenv
```

To create a virtual environment, go to your project directory and run `venv`. The following parameter (here `env`) specifies the name of your environment

```
$ python3 -m venv env
```

You'll need to activate your environment before you can start installing or using packages.

```
$ source env/bin/activate
```

When you're done working on this project, you can leave your virtual environment by running the command `deactivate`. If you want to reenter, simply call `source env/bin/activate` again.

We're now ready to install the packages we need.

Install Jupyter:

```
$ pip install jupyter notebook jupyterlab --upgrade
```

Install PyTorch:

```
$ pip install torch
```

Install TensorFlow

```
$ pip install tensorflow
```

Install Qiskit

```
$ pip install qiskit --user
```

Install Cirq

```
$ pip install cirq
```

Install further dependencies required of Qiskit and Scikit-Learn:

```
$ pip install numpy scipy matplotlib ipython pandas sympy nose seaborn --user
```

Install Scikit-Learn:

```
$ pip install scikit-learn --user
```

Install drawing libraries:

```
$ pip install pylatexenc ipywidgets
```

You're now ready to start. Open up JupyterLab with

```
$ jupyter lab
```

2. Binary Classification

2.1 Predicting Survival On The Titanic

The sinking of the Titanic is one of the most infamous shipwrecks in history.

On April 15, 1912, the Titanic sank after colliding with an iceberg. Being considered unsinkable, there weren't enough lifeboats for everyone onboard. 1502 out of 2224 passengers and crew members died that night.

Of course, the 722 survivors must have had some luck. But it seems as if certain groups of people had better chances to survive than others.

Therefore, the sinking of the Titanic has also become a famous starting point for anyone interested in machine learning.

If you have some experience with machine learning, you'll probably know the legendary Titanic ML competition provided by Kaggle.

If you don't know Kaggle yet, Kaggle is among the world's largest data science communities. It offers many interesting datasets and therefore, it is a good place to get started.

The problem to be solved is simple. Use machine learning to create a model that, given the passenger data, predicts which passengers survived the Titanic shipwreck.

2.2 Get the Dataset

In order to get the dataset, you'll need to create a Kaggle account (it's free) and join the competition. Even though Kaggle is all about competitions, you don't need to take part in them actively by uploading your solution.

Figure 2.1: The Titanic Shipwreck

When you join a competition, you need to accept and abide by the rules that govern how many submissions you can make per day, the maximum team size, and other competition-specific details.

You'll find the competition data in the Data tab at the top of the competition page. Then, scroll down to find the list of files.

There are three files in the data:

- `train.csv`
- `test.csv`
- `gender_submission.csv`

The file `train.csv` contains the data of a subset of the Titanic's passengers. This file is supposed to serve your algorithm as basis to learn whether a passenger survived or not.

The file `test.csv` contains the data of another subset of passengers. It serves to determine how good your algorithm performs.

The gender_submission.csv file is an example that shows how you should structure your predictions if you plan to submit them to Kaggle. Since we're here to start learning and not yet be ready to compete, we'll skip this file.

Download the files train.csv and test.csv.

2.3 Look at the data

The first thing we need to do is to load the data. We use *Pandas* for that. It is renowned in the machine learning community for data processing. It offers a variety of useful functions, such as a function to load .csv-files: read_csv.

Listing 2.1: Load the data from the csv-files

```python
import pandas as pd

train = pd.read_csv('./data/train.csv')
test = pd.read_csv('./data/test.csv')
```

We loaded our data into train and test. These are *Pandas* DataFrames.

A DataFrame keeps the data in a two-dimensional structure with labels. Such as a database table or a spreadsheet. It provides a lot of useful attributes and functions out of the box.

For instance, the DataFrame's attribute shape provides a tuple of two integers that denote the number of rows and the number of columns.

Let's have a look:

Listing 2.2: The shapes of the Titanic datasets

```python
print('train has {} rows and {} columns'.format(*train.shape))
print('test has {} rows and {} columns'.format(*test.shape))
```

```
train has 891 rows and 12 columns
test has 418 rows and 11 columns
```

We can see we have 891 training and 418 testing entries. More interestingly, the train dataset has one more column than the test dataset.

The DataFrame's info() method shows some more detailed information. Have a look at the train dataset.

Listing 2.3: The structure of the `train` dataset

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

The info method returns a list of the columns: their index, their names, how many entries have actual values (are not null), and the type of the values.

Let's have a look at the test dataset, too.

Listing 2.4: The structure of the `test` dataset

```
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  418 non-null    int64
 1   Pclass       418 non-null    int64
 2   Name         418 non-null    object
 3   Sex          418 non-null    object
 4   Age          332 non-null    float64
 5   SibSp        418 non-null    int64
 6   Parch        418 non-null    int64
 7   Ticket       418 non-null    object
 8   Fare         417 non-null    float64
 9   Cabin        91 non-null     object
 10  Embarked     418 non-null    object
dtypes: float64(2), int64(4), object(5)
memory usage: 36.0+ KB
```

When comparing both infos, we can see the `test` dataset misses the column `Survived` that indicates whether a passenger survived or died.

As Kaggle notes, they use the `test` dataset to evaluate the submissions. If they provided the correct answer, it wouldn't be much of a competition anymore, would it? It is our task to predict the correct label.

Since we do not plan to submit our predictions to Kaggle to get an evaluation of how our algorithm performed, the `test` dataset is quite useless for us.

So, we concentrate on the `train` dataset.

The `info` output is quite abstract. Wouldn't it be good to see some actual data? No problem. That's what the `head`-method is for.

The `head` method shows the column heads and the first five rows. With this impression, let's go through the columns. You can read an explanation on the Kaggle page, too.

Listing 2.5: Look at the data

```
train.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Each column represents one feature of our data. The `PassengerId` is a consecutive number identifying each row. `Survived` is the indicator on whether the passenger survived (0 = No, 1 = Yes). `Pclass` is the ticket class (1 = 1st, 2 = 2nd, 3 = 3rd). Then we have self-explanatory `Name`, `Sex`, and `Age`.

`SibSp` denotes the number of this passenger's siblings or spouses aboard the Titanic. `Parch` denotes the number of this passenger's parents or children aboard the Titanic.

Then, there are the `Fare` the passenger paid, the `Cabin` number and the port of embarkation (`embarked`) (C = Cherbourg, Q = Queenstown, S = Southampton).

2.4 Data Preparation and Cleaning

Our data has different types. There is numerical data, such as the `Age`, `SibSp`, `Parch`, and the `Fare`. There is categorial data. Some of the categories are represented by numbers (`Survived`, `Pclass`). Some are represented by text (`Sex` and `Embarked`). And there is textual data (`Name`, `Ticket`, and `Cabin`).

This is quite a mess. For a computer. Furthermore, when having another look at the result of `train.info()`, you can see that the counts vary for different columns. While we have 891 values for most columns, we only have 714 for `Age`, 204 for `Cabin`, and 889 for `Embarked`.

Before we can feed our data into any machine learning algorithm, we need to clean up.

2.4.1 Missing Values

Most machine learning algorithms don't work well with missing values. There are three options of how we can fix this:

- Get rid of the corresponding rows (removing the passengers from consideration)
- Get rid of the whole column (remove the whole feature for all passengers)
- Fill the missing values (for example with zero, the mean, or the median)

Listing 2.6: Cope with missing values

```
# option 1
# We only have two passengers without it. This is bearable
train = train.dropna(subset=["Embarked"])

# option 2
# We only have very few information about the cabin, let's drop it
train = train.drop("Cabin", axis=1)

# option 3
# The age misses quite a few times. But intuition
# says it might be important for someone's chance to survive.
mean = train["Age"].mean()
train["Age"] = train["Age"].fillna(mean)

train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 889 entries, 0 to 890
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  889 non-null    int64  
 1   Survived     889 non-null    int64  
 2   Pclass       889 non-null    int64  
 3   Name         889 non-null    object 
 4   Sex          889 non-null    object 
 5   Age          889 non-null    float64
 6   SibSp        889 non-null    int64  
 7   Parch        889 non-null    int64  
 8   Ticket       889 non-null    object 
 9   Fare         889 non-null    float64
 10  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(4)
memory usage: 83.3+ KB
```

We can accomplish these things easily using `DataFrame`'s `dropna()`, `drop()`, and `fillna()` methods. There is not a best option in general. But you should carefully consider the specific context.

There are only two passengers whose port of embarkation we don't know.

These account for less than 1% of our data. If we completely disregard these two passengers, we won't see completely different results. Thus, we drop these rows (line 3) with the `dropna`-method.

The `dropna`-method takes the column (`"Embarked"`) as named parameter `subset`. This parameter specifies the columns that determine whether to remove the row (passenger). If at least one value of these columns is missing, the row gets removed.

The situation is different with regard to the `Cabin`. We only have this information for 204 out of 991 passengers. It is questionable if this is enough to draw any information from. We don't know why these values miss. Even if we found the `Cabin` to be highly correlated with the survival of a passenger, we wouldn't know whether this correlation can be generalized to all passengers or whether there is a selection bias meaning that the fact that we know the `Cabin` depends on some other aspect.

We drop the whole column with the method `drop`. We provide the column (`Cabin`) we want to remove as positioned argument. The value `1` we provide as named argument `axis` specifies that we want to remove the whole column.

Next, we know the `Age` of 714 passengers. Removing all the passengers from consideration whose `Age` we don't know doesn't seem to be an option because they account for about 22% of our data, quite a significant portion. Removing the whole column doesn't seem to be a good option either. First, we know the `Age` of most of the passengers and intuition suggests that the `Age` might be important for someone's chance to survive.

We fill the missing values with the `fillna` method (line 13). Since we want to fill only the missing values in the `Age` column, we call this function on this column and not the whole `DataFrame`. We provide as argument the value we want to set. This is the mean age of all passengers we calculated before (line 12).

Great. We now have 889 rows, 10 columns, and no missing data anymore.

2.4.2 Identifiers

The goal of machine learning is to create an algorithm that is able to predict data. Or, as we said before: to put a label on a thing. While we use already labeled data when building our algorithm, the goal is to predict labels we don't know yet.

We don't tell our algorithm how it can decide which label to select. Rather, we tell the algorithm, "here is the data, figure it out yourself". That being said, a savvy algorithm may be able to memorize all the data you provide it with. This is referred to as overfitting. The result is an algorithm performing well on known data, but poorly on unknown data.

If our goal was to only predict labels we know already, the best thing we could do is to memorize all passengers and whether they survived. But if we want to create an algorithm that performs well even on unknown data, we need to prevent from memorization.

We have not even started building our algorithm. Yet, the bare features we provide our algorithm with affect whether the algorithm will be able to memorize data. Because we have potential identifiers in our data.

When looking at the first five entries of the dataset, three columns appear suspicious: the `PassengerId`, the `Name`, and the `Ticket`.

The `PassengerId` is a subsequent number. There should be no connection between how big the number is and whether a passenger survived.

Neither should the name of a passenger or the number on a ticket be a decisive factor for survival. Rather, these are data identifying single passengers. Let’s validate this assumption.

Let’s have a look at the how many unique values are in these columns.

Listing 2.7: Unique values in columns

```python
print('There are {} different (unique) PassengerIds in the data'.format(
    train["PassengerId"].nunique()))
print('There are {} different (unique) names in the data'.format(train["
    Name"].nunique()))
print('There are {} different (unique) ticket numbers in the data'.format
    (train["Ticket"].nunique()))
```

```
There are 889 different (unique) PassengerIds in the data
There are 889 different (unique) names in the data
There are 680 different (unique) ticket numbers in the data
```

`Name` and `PassengerId` are perfect identifiers. Each of the 889 rows in our dataset has a uniqe value.

And there are 680 different `Ticket` numbers. A possible explanation for the `Ticket` not to be a perfect identifier may be family tickets. Yet, a prediction based on this data appears to support memorization rather than learning transferable insights.

We remove these columns.

Listing 2.8: Remove identifying data

```
train = train.drop("PassengerId", axis=1)
train = train.drop("Name", axis=1)
train = train.drop("Ticket", axis=1)

train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 889 entries, 0 to 890
Data columns (total 8 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Survived  889 non-null    int64
 1   Pclass    889 non-null    int64
 2   Sex       889 non-null    object
 3   Age       889 non-null    float64
 4   SibSp     889 non-null    int64
 5   Parch     889 non-null    int64
 6   Fare      889 non-null    float64
 7   Embarked  889 non-null    object
dtypes: float64(2), int64(4), object(2)
memory usage: 62.5+ KB
```

2.4.3 Handling Text and Categorical Attributes

As we will see throughout this book, all the algorithms, both classic and quantum algorithms, work with numbers. Nothing but numbers. If we want to use textual data, we need to translate it into numbers.

Scikit-Learn provides a transformer for this task called `LabelEncoder`.

Listing 2.9: Transforming textual data into numbers

```
1 from sklearn.preprocessing import LabelEncoder
2 le = LabelEncoder()
3
4 for col in ['Sex', 'Embarked']:
5   le.fit(train[col])
6   train[col] = le.transform(train[col])
7
8 train.head()
```

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	1	22.0	1	0	7.2500	2
1	1	1	0	38.0	1	0	71.2833	0
2	1	3	0	26.0	0	0	7.9250	2
3	1	1	0	35.0	1	0	53.1000	2
4	0	3	1	35.0	0	0	8.0500	2

First, we import the `LabelEncoder` (line 1) and initialize an instance (line 2). We loop through the columns with textual data (`Sex` and `Embarked`) (line 4). For each column, we need to `fit` the encoder to the data in the column (line 5) before we can transform the values (line 6).

Finally, let's have another look at our `DataFrame`. You can see that both, `Sex` and `Embarked` are now numbers (`int64`). In our case, `0` denotes male and `1` dentotes female passengers. But when you run the transformation again you may yield different assignments.

2.4.4 Feature Scaling

Machine learning algorithms only work with numbers. Moreover, they usually work with numbers with identical scales. If numbers have different scales, the algorithm may consider those with higher scales to be more important.

Even though all our data is numerical, it is not yet uniformly scaled. The values of most of the columns range between `0` and `3`. But `Age` and `Fare` have far bigger scales.

The `max` method returns the maximum value in a column. As we can see, the oldest passenger was 80 years old and the highest fare was about 512.

Listing 2.10: The maximum values

```python
print('The maximum age is {}'.format(train["Age"].max()))
print('The maximum fare is {}'.format(train["Fare"].max()))
```

```
The maximum age is 80.0
The maximum fare is 512.3292
```

A common way to cope with data of different scales is min-max-scaling (also known as normalization). This process shifts and rescales values so that they end up ranging from `0` to `1`. It subtracts the minimum value from each value and divides it by the maximum minus the minum value.

`Scikit-Learn` provides the `MinMaxScaler` transformer to do this for us.

Listing 2.11: Normalization of the data.

```python
1 from sklearn.preprocessing import MinMaxScaler
2
3 scaler = MinMaxScaler()
4 scaler.fit(train)
5 train = scaler.transform(train)
6
7 print('The minimum value is {} and the maximum value is {}'.format(train.
      min(), train.max()))
```

```
The minimum value is 0.0 and the maximum value is 1.0
```

Again, we first import the transformer (line 1) and initialize it (line 3). Then, we `fit` the transformer to our data (line 4) and transform it (line 5).

As result, all the data in our dataset range between `0.0` and `1.0`.

The scaler returns a *Numpy*-array rather than a Pandas *DataFrame*.

2.4.5 Training and Testing

We already mentioned the goal of building an algorithm that does not only perform well on data it already knows but one that also predicts the labels of yet unknown data.

That's why, it is important to separate the data into a training and a testing set. We use the training set to build our algorithm. And we use the testing set to validate its performance.

Even though Kaggle provides a testing set, we skipped it for not including the `Survived` column. We would need to ask Kaggle every time we wanted to validate it. To keep things simple and be able to do the validation on our own, we rather spare some rows from the Kaggle training set for testing.

Separating a `test` set is quite simple. *Scikit-learn* provides a useful method for that, too. This is `train_test_split`.

Further, we need to separate the input data from the resulting label we want to predict.

Listing 2.12: Separating input from labels and training from testing sets

```
1 from sklearn.model_selection import train_test_split
2
3 input = train[:, 1:8]
4 labels = train[:, 0]
5
6 train_input, test_input, train_labels, test_labels = train_test_split(
      input, labels, test_size = 0.2)
7
8 print('We have {} training and {} testing rows'.format(train_input.shape
      [0], test_input.shape[0]))
9 print('There are {} input columns'.format(train_input.shape[1]))
```

```
We have 711 training and 178 testing rows
There are 7 input columns
```

We separate the input columns from the labels with Python array indices (lines 3-4). The first colum (position `0`) contains the `Survived` flag we want to predict. The other columns contain the data we use as input.

`train_test_split` separates the training from the testing data set. The parameter `test_size = 0.2` (= 20%) specifies the portion we want the testing set to have.

We can see our training data set consists of 711 entries. Accordingly, our testing set consists of 178 entries. We have input 7 columns and a single column output.

Let's save our prepared data so that we can use it in the future without needing to repeat all these steps.

Listing 2.13: Save the data to the filesystem

```python
import numpy as np

with open('data/train.npy', 'wb') as f:
  np.save(f, train_input)
  np.save(f, train_labels)

with open('data/test.npy', 'wb') as f:
  np.save(f, test_input)
  np.save(f, test_labels)
```

2.5 Baseline

Now, we have our input data and the resulting labels. And we have it separated into a training and a testing set. The only thing left is our algorithm.

Our algorithm should predict whether a passenger survived the Titanic shipwreck. This is a classification task, since there are distinct outcome values. Specifically, it is a binary classification task because there are exactly two possible predictions (`survived` or `died`).

Before we develop a quantum machine learning algorithm, let's implement the simplest algorithm we can imagine. A classifier that guesses.

Listing 2.14: A random classifier

```python
1 import random
2 random.seed(a=None, version=2)
3
4 def classify(passenger):
5   return random.randint(0, 1)
```

We import the random number generator (line 1) and initialize it (line 2).

Our classifier is a function that takes passenger data as input and returns either `0` or `1` as output. Similar to our data, `0` indicates the passenger died and `1` the passenger survived.

In order to use the classifier, we write a Python function that runs our classifier for each item in the training set.

Listing 2.15: The classification runner

```python
1 def run(f_classify, x):
2   return list(map(f_classify, x))
```

This function takes the classifier-function as the first argument (we can replace the classifier later) and the input data (as `x`) as the second parameter (line 1).

It uses Python's `map` function to call the classifier with each item in `x` and return an array of the results.

Let's run it.

Listing 2.16: Run the classifier

```python
result = run(classify, train_input)
```

```
[0, 1, 0, ... 0, 1, 1]
```

When we run the classifier with our `train_input` we recevie a list of predictions.

Since our goal is to predict the actual result correctly, we need to evaluate whether the prediction matches the actual result.

Let's have a look at the accuracy of our predictions.

Listing 2.17: Evaluate the classifier

```
1 def evaluate(predictions, actual):
2   correct = list(filter(
3     lambda item: item[0] == item[1],
4     list(zip(predictions,actual))
5   ))
6   return '{} correct predictions out of {}. Accuracy {:.0f} %' \
7     .format(len(correct), len(actual), 100*len(correct)/len(actual))
8
9 print(evaluate(run(classify, train_input), train_labels))
```

```
350 correct predictions out of 711. Accuracy 49 %
```

We define another function `evaluate`. It takes the predictions of our algorithm and the acutual results as parameters (line 1).

The term `list(zip(predictions,actual))` (line 4) creates a list of 2-item lists. The 2-item lists are pairs of a prediction and the corresponding actual result.

We `filter` these `items` from the list where the prediction matches the actual result (`lambda item: item[0] == item[1]`) (line 3). These are correct predictions. The length of the list of correct predictions divided by the total number of passengers is our `Accuracy`.

Great! We are already correct in half of the cases (more or less). This is not a surprise when guessing one out of two possible labels.

But maybe we can do even better? I mean without any effort. We know that more people died than survived. What if we always predicted the death of a passenger?

Listing 2.18: Always predict a passenger died

```
def predict_death(item):
  return 0

print(evaluate(run(predict_death, train_input), train_labels))
```

```
437 correct predictions out of 711. Accuracy 61 %
```

We're up to an accuracy of 61% of our predictions. Not too bad, is it? This value, that is the ratio between the two possible actual value is the prevalence.

Let's consider a different task for a moment. Let's say you're a doctor and your task is to predict whether a patient has cancer. Only 1% of your patients actually have cancer. If you predicted no-cancer all the time, your accuracy would be astonishing 99%! But you would falsely diagnose the patients that actually have cancer. And for the resulting lack of treatment they're going to die.

Maybe the accuracy of the predictions alone is not a good measure to evaluate the performance of our algorithm.

2.6 Classifier Evaluation and Measures

As we mentioned in section 1.4.3, the evaluation is one main part of every machine learning algorithm. It may seem trivial at first sight. Yet, deciding on the right measure is a very important step. When you optimize your algorithm towards better performance, you will inevitably optimize towards better scores in your evaluation function.

We will get to know more sophisticated evaluation functions in this book. But right now, we keep it simple. A better way to evaluate the performance of a classifier is to look at the confusion matrix.

The general idea is to compare the predictions with the actual values. In a binary classification, there are two possible actual values: `true` or `false`. And there are two possible prediction: `true` or `false`.

There are four possibilites:

- **True Negatives** (TN): a passenger who died was correctly predicted
- **False Positives** (FN): a passenger who died was wrongly predicted to survive
- **False Negatives** (FP): a passenger who survived was wrongly predicted to die
- **True Positive** (TP): a passenger who survived was correctly predicted

Let's have a look at the confustion matrix of the `predict death` classifier.

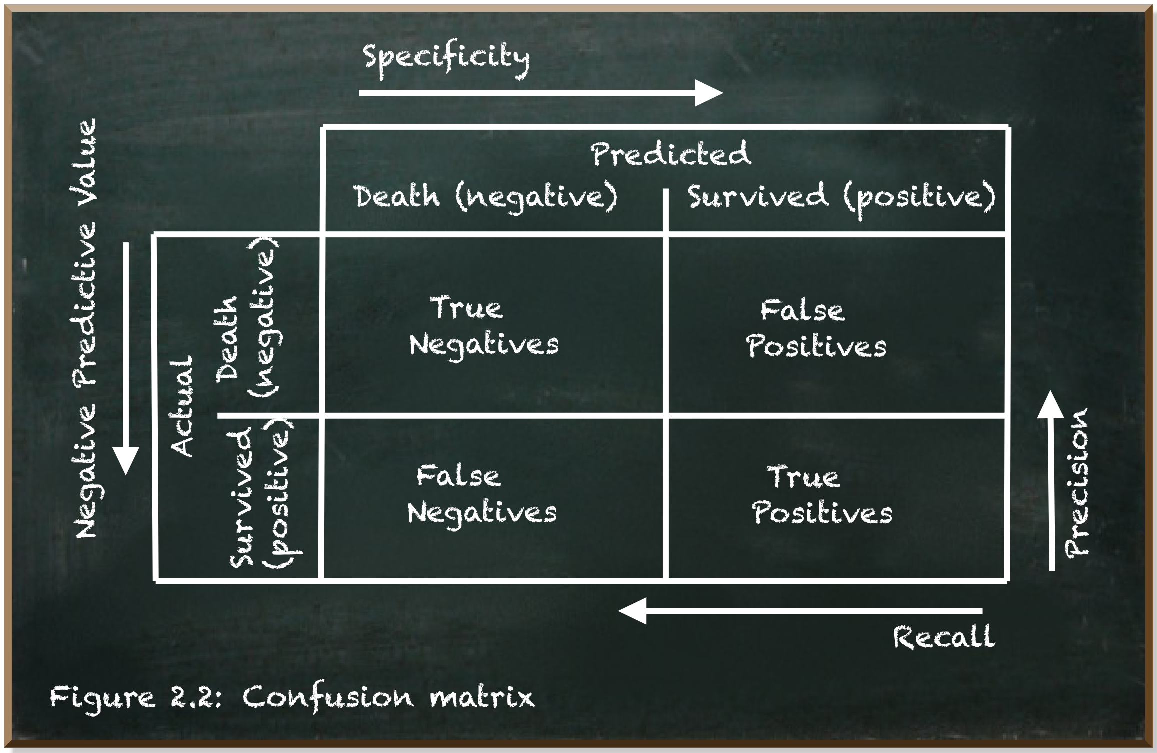


Figure 2.2: Confusion matrix

Listing 2.19: Confustion matrix of the predict death classifier

```python
1 from sklearn.metrics import confusion_matrix
2
3 predictions = run(predict_death, train_input)
4 confusion_matrix(train_labels, predictions)
```

```
array([[437,   0],
       [274,   0]])
```

Scikit-Learn provides the `confusion_matrix` method that we import (line 1). It takes the actual values as first and the predictions as second parameter (line 3).

It returns a two-dimensional array. In the first row, it shows the true negatives (TN) and the false positives (FP). In the second row it shows the false negatives (FN) and the true positives (TP).

We can define the accuracy we measured thus far as:

$$Accuracy = \frac{\sum TruePositives + \sum TrueNegatives}{TotalPopulation} \tag{2.1}$$

It does not care whether there is a systematic error, such as the algorithm's inability to correctly predict a passenger who survived (true positives) as long as it performs well in correctly predicting passengers who dies (true negatives).

The confusion matrix offers us more detailed measures of our classifier performance. These are:

- precision
- recall
- specificity
- negative predictive value (NPV)

The precision is the "accuracy of the positive predictions". It only looks at the positive predictions, these are predictions that the passenger survived.

$$Precision = \frac{\sum TruePositives}{\sum AllPredictedPositives} \tag{2.2}$$

Let's have a look at the code:

Listing 2.20: The precision score

```
from sklearn.metrics import precision_score
print('The precision score of the predict_death classifier is {}'.format(
    precision_score(train_labels, predictions)))
```

```
The precision score of the predict_death classifier is 0.0
```

Scikit-Learn provides a function to calculate the `precision_score`. It takes the list of actual values and the list of predicted values as input.

Since we did not have a single positive prediction, our precision is not defined. *Scikit-Learn* interprets this as a score of `0.0`.

Recall is the "accuracy of the actual positives". It only looks at the actual positives.

$$Recall = \frac{\sum TruePositives}{\sum AllActualPositives} \tag{2.3}$$

In Python, it is:

Listing 2.21: The recall score

```python
from sklearn.metrics import recall_score
print('The recall score of the predict_death classifier is {}'.format(
    recall_score(train_labels, predictions)))
```

```
The recall score of the predict_death classifier is 0.0
```

This time, even though `recall` is defined (the number of actual positives is greater than 0), the score is `0.0` again, because our classifier did not predict a single survival correctly. Not a suprise when it always predicts death.

The specificity is the "accuracy of the actual negatives". It only looks at actual negatives (deaths).

$$Specificity = \frac{\sum TrueNegatives}{\sum AllActualNegatives} \tag{2.4}$$

And the 'negative predictive value' (NPV) is the "accuracy of the negative predictions".

$$NegativePredictiveValue(NPV) = \frac{\sum TrueNegatives}{\sum AllPredictedNegatives} \tag{2.5}$$

These two functions are not provided out of the box. But with the values we get from the confusion matrix, we can calculate them easily:

Listing 2.22: The specificity and the npv

```
 1 def specificity(matrix):
 2   return matrix[0][0]/(matrix[0][0]+matrix[0][1]) if (matrix[0][0]+matrix
     [0][1] > 0) else 0
 3 
 4 def npv(matrix):
 5   return matrix[0][0]/(matrix[0][0]+matrix[1][0]) if (matrix[0][0]+matrix
     [1][0] > 0) else 0
 6 
 7 cm = confusion_matrix(train_labels, predictions)
 8 
 9 print('The specificity score of the predict_death classifier is {:.2f}'.
     format(specificity(cm)))
10 print('The npv score of the predict_death classifier is {:.2f}'.format(
     npv(cm)))
```

```
The specificity score of the predict_death classifier is 1.00
The npv score of the predict_death classifier is 0.61
```

The function `specificity` takes the confusion matrix as parameter (line 1). It divides the true negatives (`matrix[0][0]`) by the sum of the true negatives and the false positives (`matrix[0][1]`) (line 2).

The function `npv` takes the confusion matrix as a parameter (line 4) and divides the true negatives by the sum of the true negatives and the false negatives (`matrix[1][0]`).

These four scores provide a more detailed view on the performance of our classifiers.

Let's calculate these scores for our random classifier as well:

Listing 2.23: The scores of the random classifier

```python
random_predictions = run(classify, train_input)
random_cm = confusion_matrix(train_labels, random_predictions)

print('The precision score of the random classifier is {:.2f}'.format(
    precision_score(train_labels, random_predictions)))
print('The recall score of the random classifier is {:.2f}'.format(
    recall_score(train_labels, random_predictions)))
print('The specificity score of the random classifier is {:.2f}'.format(
    specificity(random_cm)))
print('The npv score of the random classifier is {:.2f}'.format(npv(
    random_cm)))
```

```
The precision score of the random classifier is 0.40
The recall score of the random classifier is 0.52
The specificity score of the random classifier is 0.51
The npv score of the random classifier is 0.63
```

While the `predict death` classifier exhibits a complete absence of precision and recall, it has perfect specificity and reaches an NPV score that matches the percentage of negatives in our test dataset (the prevalence).

The random classifier yields more balanced scores. You'll get a little bit different scores everytime you run the classifier. But the values seem to stay in certain ranges. While the precision of this classifier is usually below `0.4` the npv is above `0.6`.

The confusion matrix and related measures give you a lot of information. But sometimes, you need a more concise metric. In fact, the evaluation function in a machine learning algorithm must return a single measure it can optimize.

And this single measure should unmask a classifier that does not really add any value.

2.7 Unmask the Hypocrite Classifier

Even though the `predict death` classifier does not add any insight, it outperforms the random classifier with regard to overall accuracy. It exploits the prevalence, the ratio between the two possible values, not being 0.5.

The confusion matrix reveals more details on certain areas. It shows the `predict death` classifier lacks any recall, the accuracy of predicting actual positives. This is no surprise, since it always predicts death.

But having a whole set of metrics makes it difficult to measure real progress. How do we recognize that one classifier is better than another? How do we even identify a classifier that adds no value at all? How do we identify such a hypocrite classifier?

Let's write a generalized hypocrite classifier and see how we can unmask it.

Listing 2.24: A hypocrite classifier

```python
def hypocrite(passenger, weight):
  return round(min(1,max(0,weight*0.5+random.uniform(0, 1))))
```

The `hypocrite` classifier takes the `passenger` data and a `weight`. The `weight` is a number between `-1` and `1`. It denotes the classifier's tendency to predict death (negative values) or survival (positive values).

The formula `weight*0.5+random.uniform(0, 1)` generates numbers between `-0.5` and `1.5`. The `min` and `max` functions ensure the result to be between `0` and `1`. The `round` function returns either `0` (death) or `1` (survival).

Depending on the weight, the chances to return the one or the other prediction differs.

If `weight` is `-1`, it returns `-1*0.5+random.uniform(0, 1)`, a number between `-0.5` and `0.5`. A number almost always rounding to `0` (predicted death).

If `weight` is `0`, the formula returns `-1*0+random.uniform(0, 1)`. This is our random classifier.

If `weight` is `1`, it returns `1*0.5+random.uniform(0, 1)`, a number that is always greater than `0.5` and thus, rounding to `1`(predicted survival).

We can choose the tendency from `-1` to `1`. `-1` always predicts death, `0` is completely random, `1` always predicts survival.

Let's have a look at how the predictions vary. We pass the weight as a hyperparameter. Try different values, if you like.

Listing 2.25: The scores of the hypocrite classifier

```python
w_predictions = run(lambda passenger: hypocrite(passenger, -0.5),
    train_input)
w_cm = confusion_matrix(train_labels, w_predictions)

print('The precision score of the hypocrite classifier is {:.2f}'.format(
    precision_score(train_labels, w_predictions)))
print('The recall score of the hypocrite classifier is {:.2f}'.format(
    recall_score(train_labels, w_predictions)))
print('The specificity score of the hypocrite classifier is {:.2f}'.
    format(specificity(w_cm)))
print('The npv score of the hypocrite classifier is {:.2f}'.format(npv(
    w_cm)))
```

```
The precision score of the hypocrite classifier is 0.38
The recall score of the hypocrite classifier is 0.26
The specificity score of the hypocrite classifier is 0.73
The npv score of the hypocrite classifier is 0.61
```

If you run the `hypocrite` classifier a few times, you may get a feeling for its performance. But let's create a visualization of it.

The following code runs the hypocrite classifier for different values of `weight`.

Listing 2.26: Run the hypocrite classifiers

```
import numpy as np

# number of steps to consider between -1 and 1
cnt_steps = 40

# a list of the step numbers [0, 1, ..., 38, 39]
steps = np.arange(0, cnt_steps, 1).tolist()

# list of the weights at every step [-1, -0.95, ... 0.9, 0.95, 1.0]
weights = list(map(
  lambda weight: round(weight, 2),
  np.arange(-1, 1+2/(cnt_steps-1), 2/(cnt_steps-1)).tolist()
))

# list of predictions at every step
l_predictions = list(map(
  lambda step: run(
    lambda passenger: hypocrite(passenger, weights[step]),
    train_input
  ),
  steps
))

# list of confusion matrices at every steo
l_cm = list(map(
  lambda step: confusion_matrix(train_labels, l_predictions[step]),
  steps
))
```

The range of allowed `weights` is between `-1` and `1`. We divide this range into 40 (`cnt_steps`) steps (line 4). We create lists of the indices (`steps=[0, 1, ..., 38, 39]`, line 7) and of the `weights` at every step (`weights=[-1, -0.95, ... 0.9, 0.95, 1.0]`, lines 10-13). We run the `hypocrite` classifier for every step (lines 17-19) and put the results into `l_predictions` (line 16). Based on the predictions and the actual results, we calculate the confustion matrix for every step (line 26) and store them in `l_cm` (line 25).

The next piece of code takes care of rendering the two graphs.

The green graph depicts the number of predicted survivals at a step. The red graph depicts the number of predicted deaths.

Listing 2.27: Plot the distribution of predictions

```python
import matplotlib.pyplot as plt
import matplotlib

# create a graph for the number of predicted deaths
deaths, = plt.plot(
  weights, # point at x-axis
  list(map(lambda cur: l_cm[cur][0][0]+l_cm[cur][1][0], steps)),
  'lightsalmon', # color of the graph
  label='Predicted death'
)

# create a graph for the number of predicted survivals
survivals, = plt.plot(
  weights, # point at x-axis
  list(map(lambda cur: l_cm[cur][0][1]+l_cm[cur][1][1], steps)),
  'lightgreen', # color of the graph
  label='Predicted survival'
)

plt.legend(handles=[deaths, survivals],loc='upper center',
  bbox_to_anchor=(0.5, -0.15), framealpha=0.0, ncol=2)
plt.xlabel("Weight")
plt.ylabel("Number of predictions")
plt.show()
```

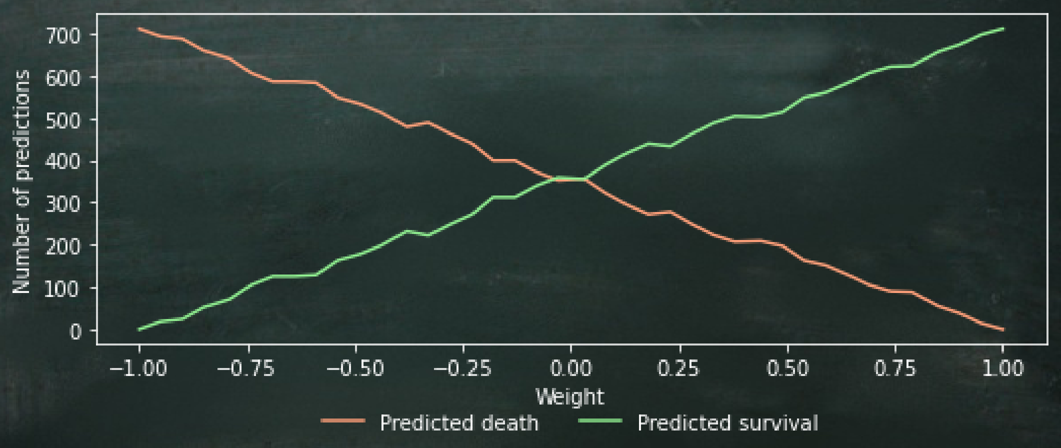


Figure 2.3: Distribution of predictions based on the weight

We can see that the `hypocrite` classifier generates the expected tendency in its predictions. At `weight=-1`, it always predicts death, at `weight=0` it is 50:50, and at `weight=1` it always predicts survival.

Let's see how the different `hypocrite` classifiers perform at the four metrics depending on the `weight`.

Listing 2.28

```python
l_precision = list(map(lambda step: precision_score(train_labels,
    l_predictions[step]),steps))
l_recall = list(map(lambda step: recall_score(train_labels, l_predictions
    [step]),steps))
l_specificity = list(map(lambda step: specificity(l_cm[step]),steps))
l_npv = list(map(lambda step: npv(l_cm[step]),steps))
```

In these four lines, we calculate the four metrics at each step. Let's visualize them.

Listing 2.29: Plot the performance measures

```python
m_precision, = plt.plot(weights, l_precision, 'pink', label="precision")
m_recall, = plt.plot(weights, l_recall, 'cyan', label="recall")
m_specificity, = plt.plot(weights, l_specificity, 'gold', label="
    specificity")
m_npv, = plt.plot(weights, l_npv, 'coral', label="npv")

plt.legend(
  handles=[m_precision, m_recall, m_specificity, m_npv],
  loc='upper center',
  bbox_to_anchor=(0.5, -0.15),
  framealpha=0.0,
  ncol=4)

plt.xlabel("Weight")
plt.ylabel("Number of predictions")
plt.show()
```

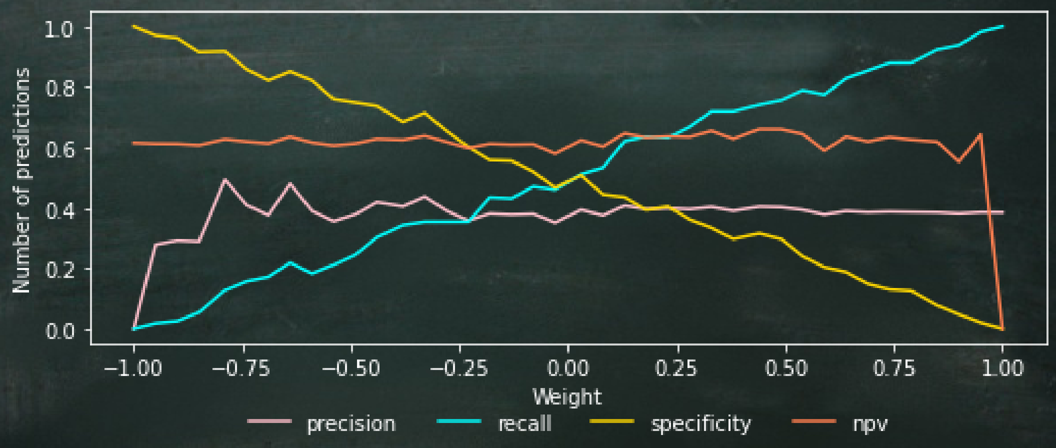


Figure 2.4: Performance measures of the hypocrite classifier

These graphs show some interesting characteristics. `specificity` and `recall` are directly related to the classifier's tendency either towards predicting death (higher `specificity`) or towards predicting survival (higher `recall`).

Except for the edge cases where all predictions are death or all are survival, the values for `precision` and `npv` seem to be horizontal lines. In fact, `precision` relates to the prevalance of 39% survivals in our data and `npv` to the prevalance of 61% deaths.

Listing 2.30: Calculating the mean of the measures

```
l_mean = list(map(lambda step: sum(step)*0.25, zip(l_precision, l_recall,
     l_specificity, l_npv)))
m_mean, = plt.plot(weights, l_mean, 'pink', label="Mean of the measures")

plt.legend(handles=[m_mean],loc='upper center',
  bbox_to_anchor=(0.5, -0.15),framealpha=0.0)
plt.ylim(0, 1)
plt.show()
```

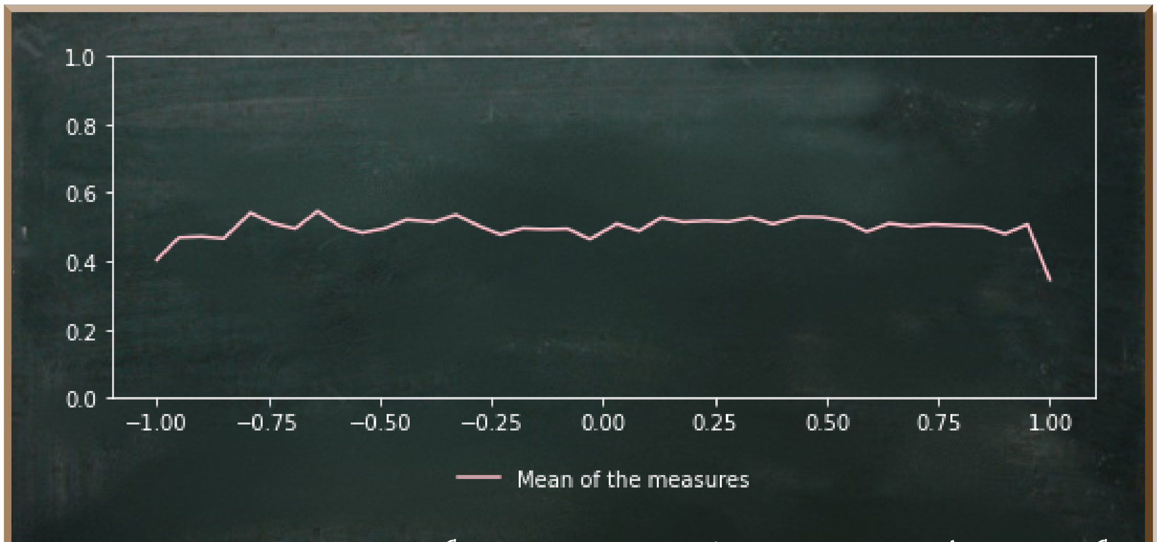


Figure 2.5: The mean of the measures discloses the absence of any information gain

When looking at the mean of all four measures, we see an almost flat line. Its drops at the edges are due to `precision` and `npv` being `0` there because there are no predicted survivals (left edge) respectively no predicted deaths (right edge) to calculate some measures.

This line indicates the overall level of information provided by all hypocrite classifiers is equal. And the level is about 0.5. That is the baseline for a binary classifier for there are only two possible outcomes.

Even though specific types of `hypocrite` classifiers are able to trick a single measure (like `accuracy`, `recall`, `precision`, or `npv`) by exploiting the prevalence, when looking at all complementary measures at once, we can unmask the `hypocrite` classifier.

However, this does not imply that the mean of these measures is the best measure to evaluate the performance of your classifier with. Depending on your task at hand, you may, for instance, favor `precision` over `recall`. Rather, the implication is that you should look at the overall level of information provided by the classifier, too. You should not let yourself be dazzled by the classifier's performance at a single measure.

Finally, let's create a resuable function that calculates the measures for us and displays the results.

Listing 2.31: A reusable function to unmask the hypocrite classifier

```python
def classifier_report(name, classify, input, labels):
    cr_predictions = run(classify, input)
    cr_cm = confusion_matrix(labels, cr_predictions)

    cr_precision = precision_score(labels, cr_predictions)
    cr_recall = recall_score(labels, cr_predictions)
    cr_specificity = specificity(cr_cm)
    cr_npv = npv(cr_cm)
    cr_level = 0.25*(cr_precision + cr_recall + cr_specificity + cr_npv)

    print('The precision score of the {} classifier is {:.2f}'.format(name,
       cr_precision))
    print('The recall score of the {} classifier is {:.2f}'.format(name,
      cr_recall))
    print('The specificity score of the {} classifier is {:.2f}'.format(
      name, cr_specificity))
    print('The npv score of the {} classifier is {:.2f}'.format(name,
      cr_npv))
    print('The information level is: {:.2f}'.format(cr_level))
```

Let’s use this function to get a report of our random classifier.

Listing 2.32

```python
classifier_report(
    "Random PQC",
    classify,
    train_input,
    train_labels)
```

```
The precision score of the Random PQC classifier is 0.36
The recall score of the Random PQC classifier is 0.48
The specificity score of the Random PQC classifier is 0.47
The npv score of the Random PQC classifier is 0.59
The information level is: 0.48
```

3. The Qubit and Quantum States

In this chapter, we start with the very basics of quantum computing. The quantum bit. And we will write our first quantum circuit. A quantum circuit is a sequence of quantum bit transformations. The quantum program. Let's start with the basics.

3.1 Exploring Quantum States

The world of quantum mechanics is... different. A quantum system can be in a state of superposition. A popular notion of superposition is that the system is in different states concurrently, unless you measure it.

For instance, the spin of a particle is not up **or** down but it is up **and** down at the same time. But when you look at it, you find it either up **or** down.

Or, let's say you flip a quantum coin. In the air, it has both values heads **and** tails. If and only if you catch it and look at it, it decides for a value. Once landed, it is a normal coin with heads up **or** tails up.

Another notion of superposition is that the system is truly random and therefore distinguishes from the systems we know. Tossing a (normal) coin, for instance, seems random, because whenever you do it, the conditions are slightly different. And even tiny differences can change the outcome from heads to tails. The coin is *sensitive dependent to initial conditions*.

If we were able to measure all conditions precisely, we could tell the outcome. In classic mechanics, there is no randomness. Things in our everyday world, such as the coin, seem random. But they are not. If measured with infinite precision, randomness would dissappear. By contrast, a quantum system is truly random.

Maybe you wonder: *Ok, it's random. Where's the big deal?*

The big thing is the consequences. In a classic system, a system *sensitive dependent to initial conditions*, the answer to a question is already determined before we ask it.

Rather than watching the baseball match tonight, you spend the evening with your friends. When you return home, even though you don't know the results, the match is over and there is a definite result. There could be different results, but you simply don't know the result until you look at it.

Contrarily, in a quantum system the answer to a question is not determined up until the time you ask it. And since it is not determined yet, you still can change the probabilities of measuring distinct states.

If you have doubts, good! Not even Einstein liked this notion. It led him to his famous statement of *God does not play dice*.

Figure 3.1: Albert Einstein

Many physicists, including Einstein, proposed the quantum state, though hidden, to be a well-defined state. This is known as the *hidden variable theory*.

There are statistically distinct behaviors between a system following the hidden variable theory and a quantum system following the superposition principle. And experiments showed that the quantum mechanical predictions were correct.

For now, let's accept the quantum state is something different. Later in this book, we will have a closer look at it. And its consequences. But this requires a little more theory and math.

We turn to the quantum computer. Let's say you have a quantum bit. We call it qubit. Unless you observe its value, it is in a superposition state of `0` and `1`. Once you observe its value, you'll get `0` or `1`.

The chances of a qubit to result in either one value don't need to be 50:50. It can be 25:75, 67:33, or even 100:0. It can be any weighted probability distribution.

The probability distribution a qubit has when observed depends on its state. The quantum state.

In quantum mechanics, we use vectors to describe the quantum state. A popular way of representing quantum state vectors is the **Dirac** notation's "ket"-construct that looks like $|\psi\rangle$. In Python, we don't have vectors. But we have arrays. Luckily, their structures are similar.

Let's have a look. We start with the simplest case. Let's say, we have a qubit that, when observed, always has the value `0`. If you argued this qubit must have the value `0` even before it is observed, you wouldn't be completely wrong. Yet, you'd be imprecise. Before it is observed, this qubit has the probability of 1 $(= 100\%)$ to have the value `0` when observed.

These are the equivalent representations (ket, vector, array) of a qubit that always results in `0` when observed:

$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and in Python `[1, 0]`.

Accordingly, the following representations depict a qubit that always results in `1` when observed:

$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and in Python `[0, 1]`.

Ok, enough with the theory for now. Let's have a look at the code of such a qubit.

If you haven't configured your workstation yet, have a look at the brief explanation of how to setup the working environment (section 1.6).

Now, open the *Jupyter* notebook and test whether *Qiskit* works.

Listing 3.1

```python
import qiskit
qiskit.__qiskit_version__
```

```
{'qiskit-terra': '0.15.1',
 'qiskit-aer': '0.6.1',
 'qiskit-ignis': '0.4.0',
 'qiskit-ibmq-provider': '0.8.0',
 'qiskit-aqua': '0.7.5',
 'qiskit': '0.20.0'}
```

If you get a response like this, *Qiskit* works. Great! We're ready to create our first qubit.

Listing 3.2: The first qubit

```python
1  from qiskit import QuantumCircuit
2
3  # Create a quantum circuit with one qubit
4  qc = QuantumCircuit(1)
5
6  # Define initial_state as |1>
7  initial_state = [0,1]
8
9  # Apply initialization operation to the qubit at position 0
10 qc.initialize(initial_state, 0)
```

The fundamental unit of *Qiskit* is the quantum circuit. A quantum circuit is a model for quantum computation. The program, if you will. Our circuit consists of a single one qubit (line 4).

We define `[0,1]` as the `initial_state` of our qubit (line 7) and initialize the first and only qubit (at position `0` of the array) of our quantum circuit with it (line 10).

Remember `[0,1]`? This is the equivalent to $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$. And in plain English it is a qubit resulting in the value `1` when observed.

This is it. It's now time to boot our quantum computer. In case you don't have one, no problem. We can simulate it. (In case you have one: "Cool, *let me know*").

Listing 3.3: Prepare the simulation backend

```
1 from qiskit import execute, Aer
2
3 # Tell Qiskit how to simulate our circuit
4 backend = Aer.get_backend('statevector_simulator')
5
6 # Do the simulation, returning the result
7 result = execute(qc,backend).result()
```

Qiskit provides the `Aer` package (that we import in line 1). It provides different backends for simulating quantum circuits. The most common backend is the `statevector_simulator` (line 4).

The `execute`-function (that we import in line 1, too) runs our quantum circuit (`qc`) at the specified `backend`. It returns a `job`-object that has the useful method `job.result()`. This returns the `result` object once our program completes.

Let's have a look at our qubit in action.

Qiskit uses Matplotlib to provide useful visualizations. A simple histogram will do. The `result` object provides the `get_counts` method to obtain the histogram data of an executed circuit (line 5).

The method `plot_histogram` returns a *Matplotlib* figure that Jupyter draws automatically (line 8).

We see we have a 100% chance of observing the value `1`.

Listing 3.4: The measured qubit

```
1 from qiskit.visualization import plot_histogram
2 import matplotlib.pyplot as plt
3
4 # get the probability distribution
5 counts = result.get_counts()
6
7 # Show the histogram
8 plot_histogram(counts)
```

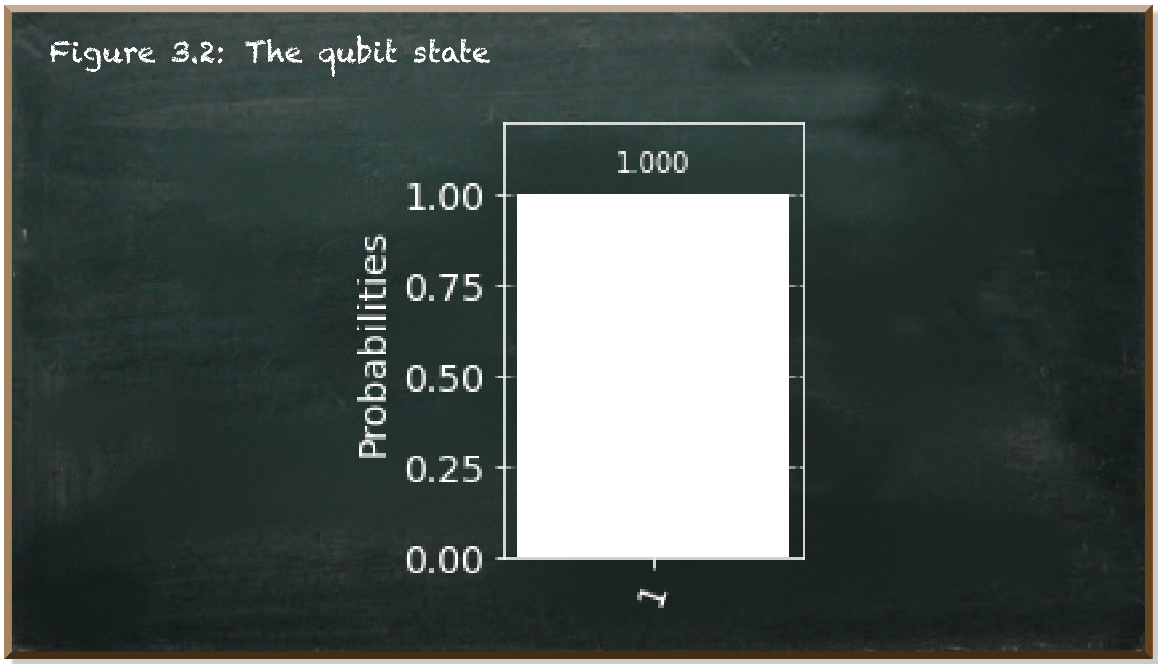


Figure 3.2: The qubit state

Now, let's move on to a more advanced case. Say, we want our qubit to result in either `0` or `1` with the same probability (50:50).

In quantum mechanics, there is the fundamental principle superposition. It says any two (or more) quantum states can be added together ("superposed") and the result will be another valid quantum state.

Wait! We already know two quantum states, $|0\rangle$ and $|1\rangle$. Why don't we add them? $|0\rangle$ and $|1\rangle$ are vectors. Adding two vectors is straight forward.

A vector is a geometric object that has a magnitude (or length) and a direction. Usually, they are represented by straight arrows, starting at one point on a coordinate axis and ending at a different point.

You can add two vectors by placing one vector with its tail at the others vector's head. The straight line between the yet unconnected tail and the yet unconnected tail is the sum of both vectors. Have a look at figure 3.3.

Mathematically, it is as easy.

Let $\vec{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ and $\vec{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$ be two vectors.

The sum of $\vec{u}$ and $\vec{v}$ is:

$$\vec{u} + \vec{v} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \end{bmatrix} \tag{3.1}$$

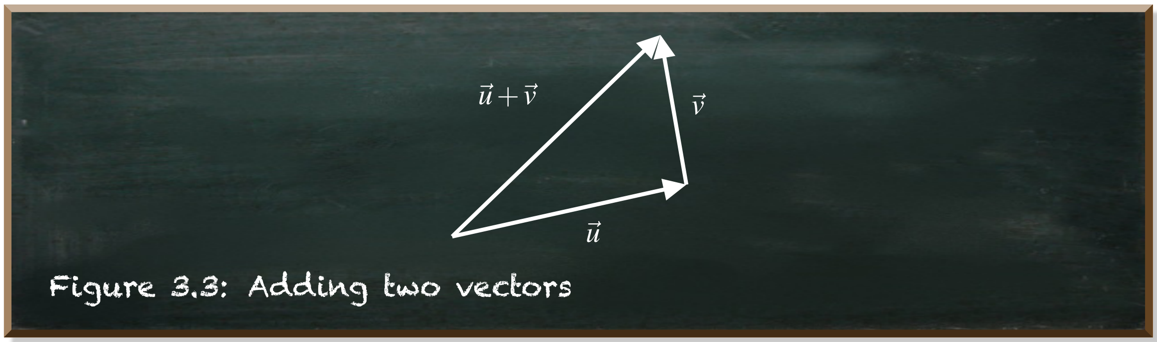


Figure 3.3: Adding two vectors

Accordingly, our superposed state should be ψ^*:

$$|\psi\rangle = \underbrace{|0\rangle + |1\rangle}_{superposition} = \begin{bmatrix}1+0\\0+1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix} \tag{3.2}$$

$^*\psi$ (“psi”) is a common symbol used for the state of a quantum system.

We have a computer at our hand. Why don’t we try it?

Listing 3.5: First attempt to superpose two states

```python
# Define state |psi>
initial_state = [1, 1]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```

```
QiskitError: 'Sum of amplitudes-squared does not equal one.'
```

It didn’t quite work. It tells us: `QiskitError: 'Sum of amplitudes-squared does not equal one.'`.

The amplitudes are the values in our array. They are proportionals to probabilities. And all the probabilites should add up to exactly 1 (100%). We need to add weights to the quantum states $|0\rangle$ and $|1\rangle$. Let's call them α and β.

We weight $|0\rangle$ with α and $|1\rangle$ with β. Like this:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} 1\cdot\alpha + 0\cdot\beta \\ 0\cdot\alpha + 1\cdot\beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Amplitudes are proportionals to probabilities. We need to normalize them so that:

$$\alpha^2 + \beta^2 = 1 \tag{3.3}$$

If both states $|0\rangle$ and $|1\rangle$ should have the same weight, then

$$\alpha = \beta$$

And therefore, we can solve our equation to α:

$$\alpha^2 + \alpha^2 = 1 \Leftrightarrow 2\cdot\alpha^2 = 1 \Leftrightarrow \alpha^2 = \frac{1}{2} \Leftrightarrow \alpha = \frac{1}{\sqrt{2}}$$

And we insert the value for both α and β (both are equal). Let's try this quantum state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

The corresponding array in Python is: `[1/sqrt(2), 1/sqrt(2)]`. Don't forget to import `sqrt`.

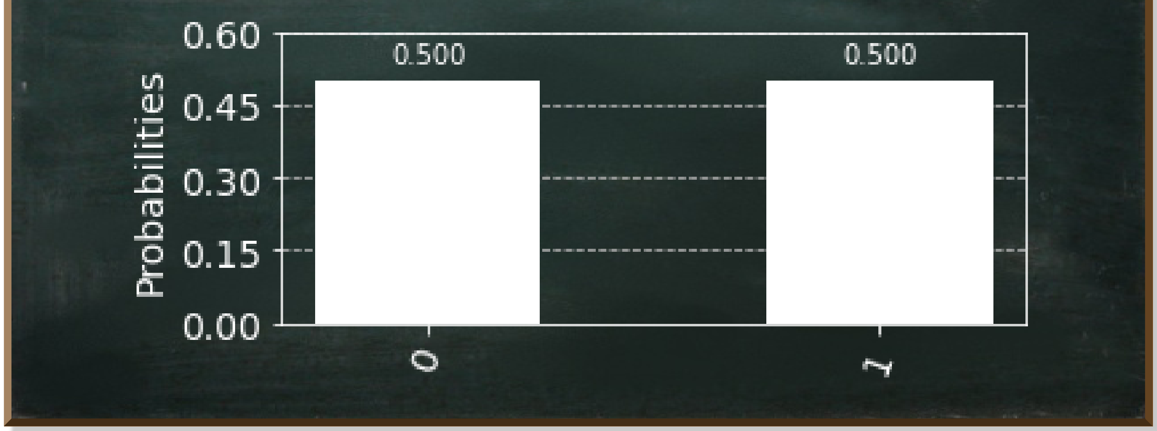
Exercise 3.1: Calculate the qubit state

What is the state of qubit that has a 25% chance of resulting in `0` and 75% of resulting in `1`?
See section 5.1 for the solution.

Listing 3.6: Weighted initial state

```python
from math import sqrt

# Define state |psi>
initial_state = [1/sqrt(2), 1/sqrt(2)]

# Redefine the quantum circuit
qc = QuantumCircuit(1)

# Initialise the 0th qubit in the state `initial_state`
qc.initialize(initial_state, 0)

# execute the qc
results = execute(qc,backend).result().get_counts()

# plot the results
plot_histogram(results)
```

Figure 3.4: The qubit state



Phew. In this chapter, we introduced quite a few terms and equations just to scratch on the surface of quantum mechanics. But the actual source code is pretty neat, isn't it?

We introduced the notion of the quantum state. In particular, the state of a binary quantum system. The quantum bit or qubit.

Until we observe a qubit, it is in superposition. Contrary to a classical bit that can be either `0` or `1`, a qubit is in a superposition of both states. But once you observe it, there are distinct probabilities of measuring `0` or `1`.

This means that multiple measurements made on multiple qubits in identical states will not always give the same result. The equivalent representations of a quantum bit that, when observed, has the probability of α^2 to result in `0` and β^2 to result in `1` are:

$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha \\ \beta\end{bmatrix}$, with $\alpha^2 + \beta^2 = 1$. In Python the array `[alpha, beta]` denotes this state.

3.2 Visual Exploration Of The Qubit State

The qubit is a two-dimensional quantum system. Each dimension is denoted by a standard basis vector:

$|0\rangle = \begin{bmatrix}1 \\ 0\end{bmatrix}$ in Python `[1, 0]` and

$|1\rangle = \begin{bmatrix}0 \\ 1\end{bmatrix}$, in Python `[0, 1]`.

The state of the qubit is represented by the superposition of both dimensions. This is the qubit state vector $|\psi\rangle$ ("psi").

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha \\ \beta\end{bmatrix} \tag{3.4}$$

In Python, $|\psi\rangle$ is the array `[alpha, beta]`.

But $|\psi\rangle$ must be normalized by:

$$\alpha^2 + \beta^2 = 1 \tag{3.5}$$

Altough normalizing the qubit state vector is not a difficult task, doing the math over and over again is quite cumbersome.

But maybe, there's another way. An easy way. Let's first have a look at a graphical representation of the qubit state $|\psi\rangle$ in the following figure 3.5.

In this representation, both dimensions reside at the vertical axis but in opposite directions. The top and the bottom of the system correspond to the standard basis vectors $|0\rangle$ and $|1\rangle$, respectively.

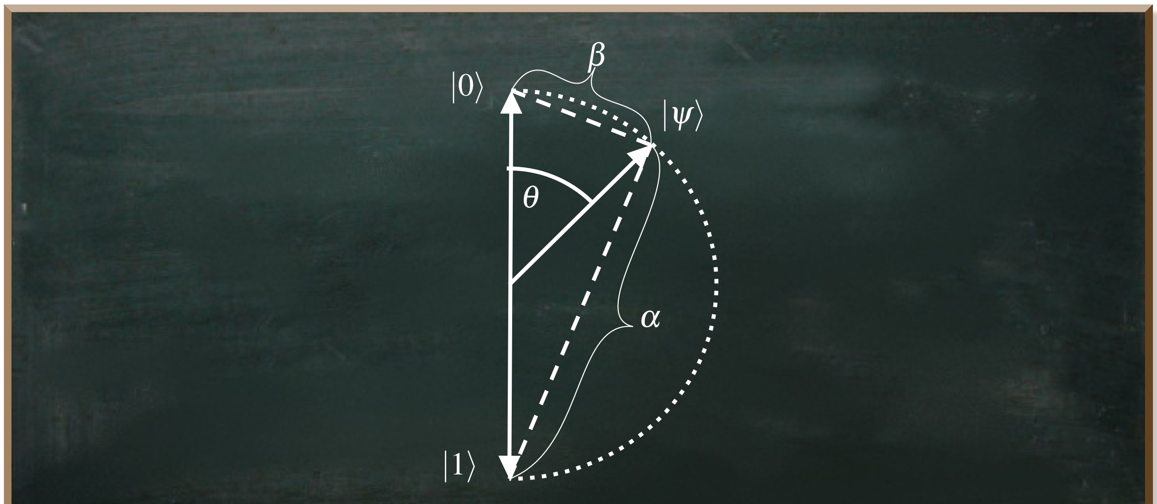


Figure 3.5: 2-dimensional qubit system

When there are two dimensions, the usual way is to put the two dimensions orthogonal to each other. While using one axis to represent both dimensions is rather an unusual representation for a two-dimensional system, it is well suited for a quantum system. But more on this later.

Let's have a look at the arbitrary qubit state vector $|\psi\rangle$ in this figure 3.5.

Since qubit state vectors are normalized, $|\psi\rangle$ originates in the center and has the magnitude (length) of $\frac{1}{2}$. Due to this equal magnitude, all state vectors end at the pointed circle. So does $|\psi\rangle$.

The angle between the state vector $|0\rangle$ and $|\psi\rangle$, named θ ("theta"), controls the proximities of the vector head to the top and the bottom of the system (dashed lines).

These proximities represent the probabilities of

- α^2 of measuring $|\psi\rangle$ as `0`
- and β^2 of measuring it as `1`.

The proximities α and β are at the opposite sides of the state's probability ($|\psi\rangle$) they describe. α is the proximity (or distance) to $|1\rangle$ because with increasing distance to $|1\rangle$ the probability of measuring ‘0‘ increases.

Thus, by controlling the proximities, the angle θ also controls the probabilities of measuring the qubit in either state `0` or `1`.

Rather than specifying the relation between α and β and then coping with normalizing their values, we can specify the angle θ and use the required normalization to derive α and β from it.

We can deduct the values of α and β and thus the state $|\psi\rangle$:

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix} \tag{3.6}$$

In Python the two-field array `[cos(theta/2), sin(theta/2)]` denotes this state.

Exercise 3.2: Derive The Proof Of The Theta-Formula

Can you mathematically derive the formula:

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix}$$

If you need a little hint: Thales’ and the Pythagorean theorem help you here.
See section 5.2 for the solution.

There’s one problem left. For $\theta \in \mathbb{R}$, what if $\pi < \theta < 2\pi$? Or in plain English, what if the θ denotes a vector pointing to the left side of the vertical axis?

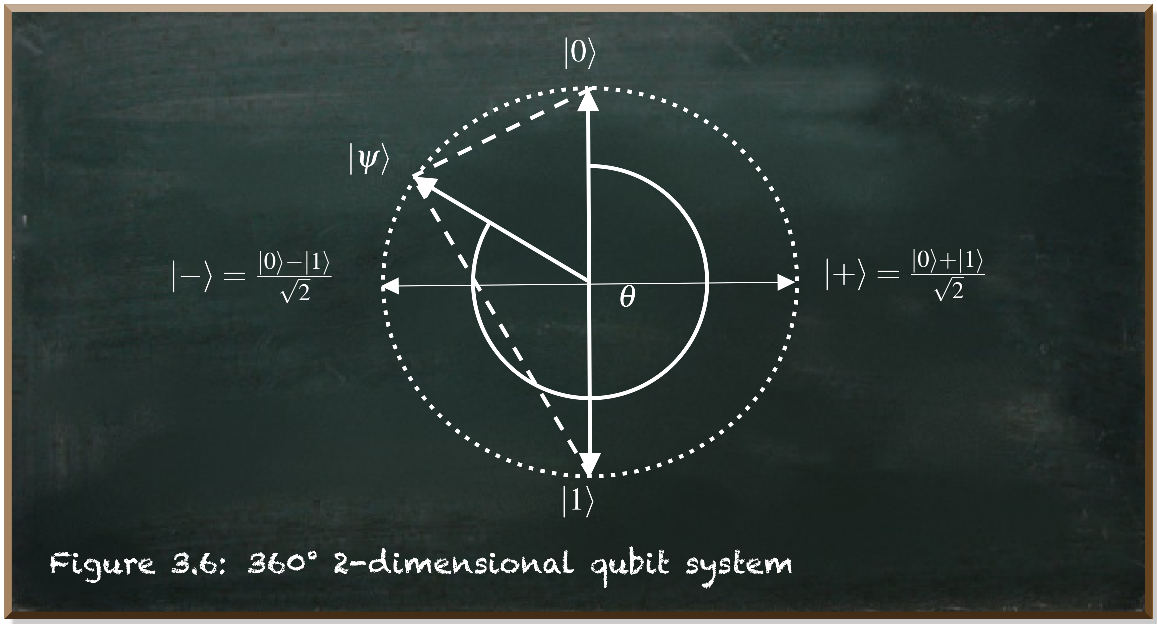
Figure 3.6 shows this situation.

Mathematically, we don’t have a problem. Since we square α and β, their signs ($+$ or $-$) are irrelevant for the resulting probabilities.

But what does it mean? How can either α^2 or β^2 be negative, as the figure indicates? The answer is i. i is a complex number whose square is negative: $i^2 = -1$.

And if α and β are complex numbers ($\alpha, \beta \in \mathbb{C}$), their squares can be negative.

This entails a lot of consequences. And it raises a lot of questions. We will unravel them one by one in following sections. For now, we interpret all vec-


$|0\rangle$
$|\psi\rangle$
$|-\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$
$|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$
θ
$|1\rangle$
Figure 3.6: 360° 2-dimensional qubit system

tors on the left hand side of the vertical axis to have a negative value for β^2 ($\beta^2 < 0$).

While such a value lets us distinguish the qubit state vectors on both sides of the vertical axis, it does not matter for the resulting probabilities.

For instance, the state $|\psi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ yields the same probability of measuring `0` or `1`. It resides at the horizontal axis. And so does $|\psi\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$

Although, these states share the same probabilities, they are different. And the angle θ differentiates between them.

$\theta = \frac{\pi}{2}$ specifies $|\psi\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ that is also known as $|+\rangle$.

And $\theta = \frac{3}{2}\pi$ or $\theta = -\frac{\pi}{2}$ specifies $|\psi\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$ that is also known as $|-\rangle$.

One of the consequences mentioned above of α^2 or β^2 being negative is that our normalization rule needs some adjustments.

We need to change equation 3.5 to:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{3.7}$$

This section contained a lot of formulae. The important takeaway is we can specify quantum states that yield certain probabilities of measuring `0` and `1` by an angle θ. It saves us doing the normalization manually.

Let's have a look.

Listing 3.7: Using theta to specify the quantum state vector

```
 1 from math import pi, cos, sin
 2 from qiskit import QuantumCircuit, Aer, execute
 3 from qiskit.visualization import plot_histogram
 4
 5 def get_state (theta):
 6   """returns a valid state vector"""
 7   return [cos(theta/2), sin(theta/2)]
 8
 9 # play with the values for theta to get a feeling
10 theta = -pi/2 # affects the probabilities
11
12
13 # create, initialize, and execute the quantum circuit
14 qc = QuantumCircuit(1)
15 qc.initialize(get_state(theta), 0)
16 backend = Aer.get_backend('statevector_simulator')
17 result = execute(qc,backend).result()
18 counts = result.get_counts()
19
20 # Show the histogram
21 plot_histogram(counts)
```

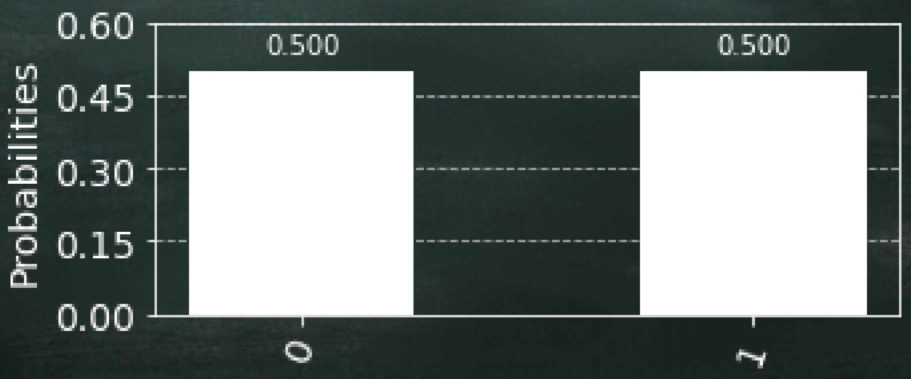


Figure 3.7: Theta specifies the qubit state |->

In this piece of code, we introduced the function `getState` (line 5). It takes `theta` as a parameter and returns the array `[cos(theta/2), sin(theta/2)]`. This is the vector we specified in equation 3.6.

3.3 Exploring The Observer Effect

A qubit is a two-level quantum system that is in superposition of the quantum states $|0\rangle$ and $|1\rangle$ unless you observe it. Once you observe it, there are distinct probabilities of measuring `0` or `1`. In physics, this is known as the observer effect. It says the mere observation of a phenomenon inevitably changes that phenomenon itself. For instance, if you're measuring the temperature in your room, you're taking away a little bit of the energy to heat up the mercury in the thermometer. This loss of energy cools down the rest of your room. In the world we experience, the effects of observation are often negligible.

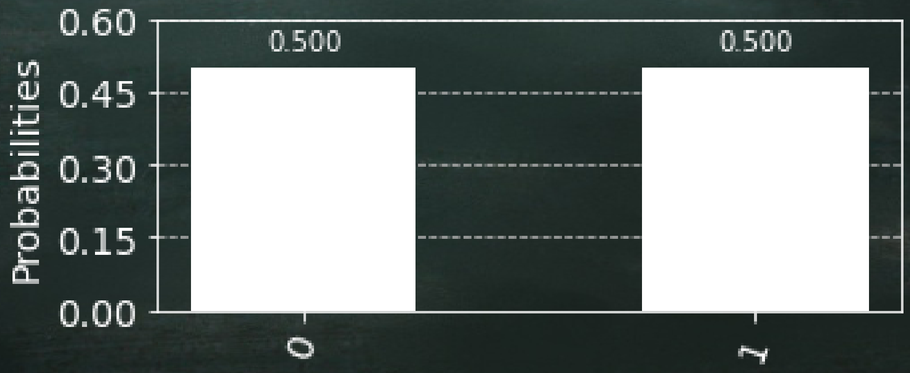
But in the sub-atomic world of quantum-mechanics, these effects matter. They matter a lot. The mere observation of a quantum bit changes its state from a superposition of the states $|0\rangle$ and $|1\rangle$ to either one value. Thus, even the observation is a manipulation of the system we need to consider when developing a quantum circuit.

Let's revisit the quantum circuit from section 3.1. Here's the code and the result if you run it:

Listing 3.8: Weighted initial state

```python
from qiskit import QuantumCircuit, execute, Aer
from qiskit.visualization import plot_histogram
from math import sqrt

# Create a quantum circuit with one qubit
qc = QuantumCircuit(1)

# Define state |Psi>
initial_state = [1/sqrt(2), 1/sqrt(2)]

# Apply initialization operation to the qubit at position 0
qc.initialize(initial_state, 0)

# Tell Qiskit how to simulate our circuit
backend = Aer.get_backend('statevector_simulator')

# Do the simulation, returning the result
result = execute(qc,backend).result()

# Get the data and display histogram
counts = result.get_counts()
plot_histogram(counts)
```

Figure 3.8: Probabilities of measuring a qubit



Our circuit consists of a single one qubit (line 6). It has the initial state `[1/sqrt(2), 1/sqrt(2)]` (line 9) that we initialize our quantum circuit with (line 12).

Here are the *Dirac* and the vector notation of this state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

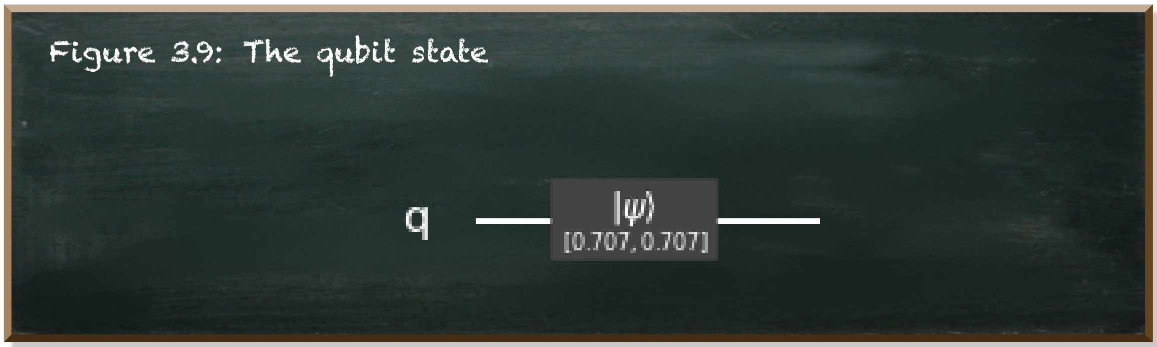
We add a simulation backend (line 15), execute the circuit and obtain the result (line 18). The `result` object provides the `get_counts` function that provides the probabilities for the resulting (observed) state our qubit.

Let's have a look at our circuit. The `QuantumCircuit` provides the `draw` function that renders an image of the circuit diagram. Provide `output=text` as named parameter to get an ASCII art version of the image.

Listing 3.9: Draw the circuit

```
qc.draw(output='text')
```

Figure 3.9: The qubit state



This drawing shows the inputs on the left, outputs on the right, and operations in between.

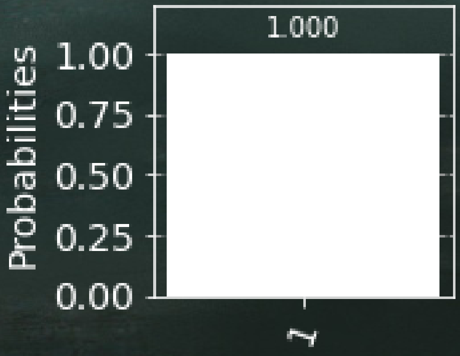
What we see here is our single qubit (`q`) and its initialization values ($\frac{1}{\sqrt{2}} = 0.707$). These values are both, the input and the output of our circuit. When we execute this circuit, our `result`-function evaluates the quantum bit in the superposition state of $|0\rangle$ and $|1\rangle$. Thus, we have a 50:50 chance to catch our qubit in either one state.

Let's see what happens if we observe our qubit as part of the circuit.

Listing 3.10: Weighted initial state

```
qc = QuantumCircuit(1)
qc.initialize(initial_state, 0)

# observe the qubit
qc.measure_all()

# Do the simulation, returning the result
result = execute(qc,backend).result()
counts = result.get_counts()
plot_histogram(counts)
```

Figure 3.10: Measuring the qubit inside the circuit



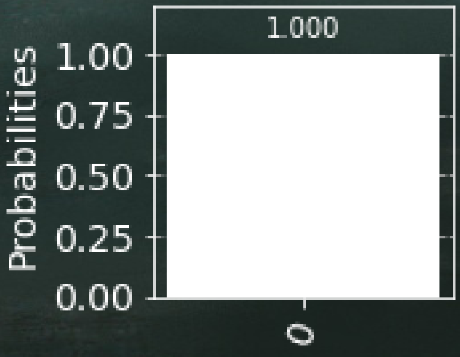
"*Whoa?!*"

We get a 100% probability of resulting state 1. That can't be true. Let's rerun the code... (I know, doing the same things and expecting different results is a sign of insanity)

Listing 3.11: Weighted initial state

```
qc = QuantumCircuit(1)
qc.initialize(initial_state, 0)
qc.measure_all()
result = execute(qc,backend).result()
counts = result.get_counts()
plot_histogram(counts)
```

Figure 3.11: Measuring the qubit inside the cicuit, agian



Again. 100% probability of measuring ... wait ... it's state `0`.

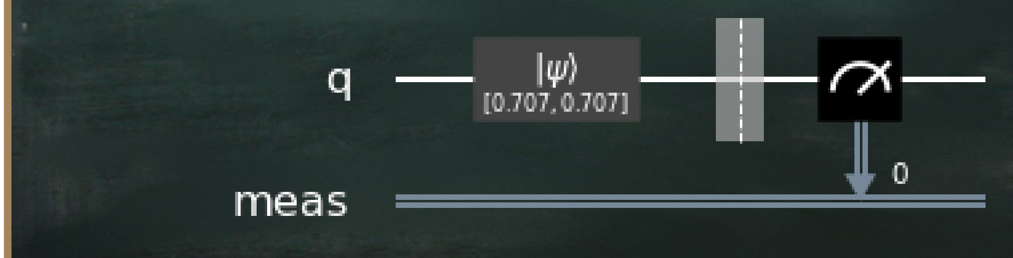
No matter how often you run this code, you'll always get 100% probability of either `0` or `1`. In fact, if you reran the code many, many times and counted the results, you'd see a 50:50 distribution.

Sounds suspicious? Yes, you're right. Let's have a look at our ciruit.

Listing 3.12: Weighted initial state

```
qc.draw(output='text')
```

Figure 3.12: The qubit state



Our circuit now contains a measurement. That is an observation. It pulls our qubit out of a superposition state and lets it collapse into either `0` **or** `1`. When we obtain the result afterwards, there's nothing quantumic anymore. It is a distinct value. And this is the output (to the right) of the circuit.

Whether we observe a `0` or a `1` is now part of our quantum circuit.

The small number at the bottom measurement line does not depict a qubit's value. It is the measurement's index. It starts counting at 0. The next measurements will have the numbers 1, 2, etc.

Sometimes, we refer to measurement as collapsing the state of the qubit. This notion emphasizes the effect a measurement has. Unlike classical programming, where you can inspect, print, and show values of your bits as often as you like, in quantum programming, it has an effect on your results.

If we constantly measured our qubit to keep track of its value, we would keep it in a well-defined state, either `0` or `1`. Such a qubit wouldn't be different from a classical bit. Our computation could be easily replaced by a classical computation. In a quantum computation, we must allow the qubits to explore more complex states. Measurements are therefore only used when we need to extract an output. This means that we often place the all measurements at the end of our quantum circuit.

In this section, we had a look at the simplest quantum circuit. We initialize a single qubit and observe it. But it effectively demonstrates the observer effect in quantum computing. It is something we need to keep in mind, when we start manipulating our qubits.

3.4 Parameterized Quantum Circuit

In chapter 2, we created different hypocrite classifiers. These are classifiers solely building upon chance when predicting the label of a thing. While such a classifier can yield seemingly good performance in a single measure, such as "precision", it does not reach an average far beyond 0.5 four the four measures that directly result from the confusion matrix (precision, recall, specificity, and NPV).

In this section, we use a quantum circuit as to solve our binary classification task. This quantum circuit is a **Parameterized Quantum Circuit** (PQC). A PQC is a quantum circuit that takes all data it needs as input paramters. Therefore it has its name "parameterized". It predicts the label of the thing based on these parameters.

The following image 3.13 depicts the simple PQC we are about to build in this section.

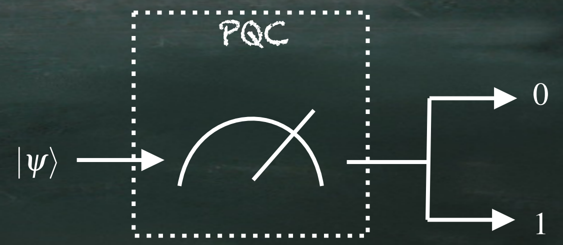


Figure 3.13: A PQC binary classifier

This PQC takes a single quantum state (ψ) as its input. It measures the state and provides its prediction as output.

We created such a quantum circuit in the last section 3.3, already.

Here's the source code.

Listing 3.13: A simple PQC binary classifier

```
qc = QuantumCircuit(1)
initial_state = [1/sqrt(2), 1/sqrt(2)]
qc.initialize(initial_state, 0)
qc.measure_all()
```

In fact, this circuit outputs either `0` or `1`, each with a probability of 50%. It sounds a lot like the random classifier we created in section 2.5.

Let's wrap this circuit into a function we can use with the `run` and `evaluate` functions we created in that section to see whether it behaves similarly.

Listing 3.14: The parameterized quantum circuit classifier

```
from qiskit import execute, Aer, QuantumCircuit
from math import sqrt
from sklearn.metrics import recall_score, precision_score,
    confusion_matrix

def pqc_classify(backend, passenger_state):
  """backend -- a qiskit backend to run the quantum circuit at
  passenger_state -- a valid quantum state vector"""

  # Create a quantum circuit with one qubit
  qc = QuantumCircuit(1)

  # Define state |Psi> and initialize the circuit
  qc.initialize(passenger_state, 0)

  # Measure the qubit
  qc.measure_all()

  # run the quantum circuit
  result=execute(qc,backend).result()

  # get the counts, these are either {'0': 1} or {'1': 1}
  counts=result.get_counts(qc)

  # get the bit 0 or 1
  return int(list(map(lambda item: item[0], counts.items()))[0])
```

The first difference to notice is the function takes two parameters instead of one (line 5). The first parameter is a *Qiskit* `backend`. Since the classifier will run a lot of times in a row, it makes sense to reuse all we can. And we can reuse the `backend`.

The second parameter differs from the classifiers thus far. It does not take the passenger data, but a quantum state vector (`passenger_state`) as input . This is not a problem right now, since all the hypocrite classifiers we developed so far ignored the data anyway.

The function creates a quantum circuit with one qubit (line 12), initializes it with the `passenger_state` (line 15), measures the qubit (line 18), executes the quantum circuit (line 21) and retrieves the counts from the `result` (line 24). All these steps did not change.

But how we return the counts is new (line 27). `counts` is a Python dictionary. It contains the measurement result (either `0` or `1`) as a key and the probability as the associated value. Since our quantum circuit measures the qubit, it collapsed to a finite value. Thus, the measurement probability is always `1`. Consequently, `counts` is either `{'0': 1}` or `{'1': 1}`.

All we're interested in here is the key. And this is what we return.

We start (from inner to outer) with the term `counts.items()`. It transforms the Python dictionary into a list of tuples, like `[('0', 1)]`. Since we only have one key in the dictionary, there is only one tuple in the list. The important point is to get the tuple rather than the dictionary's key-value construct because we can access a tuple's elements thorugh the index.

This is what we do in the function `lambda: item: item[0]`. It takes a tuple and returns its first element. We do this for every item in the list (even though there is only one item) by using `list(map(...))`. From this list, we take the first (and only) item (either `'0'` or `'1'`) and transform it into a number (`int(...)`).

Before we can run it, we need to load the prepared passenger data.

Listing 3.15: Load the data

```python
import numpy as np

with open('data/train.npy', 'rb') as f:
  train_input = np.load(f)
  train_labels = np.load(f)

with open('data/test.npy', 'rb') as f:
  test_input = np.load(f)
  test_labels = np.load(f)
```

The following code runs the `pqc_classifier` with the initial state with a probability of 0.5 to measure `0` or `1`, respectively (line 5).

Further, we create a backend (line 2) and provide it as a parameter to be reused (line 8).

Listing 3.16: The scores of the random quantum classifier

```python
 1 # Tell Qiskit how to simulate our circuit
 2 backend = Aer.get_backend('statevector_simulator')
 3 
 4 # Specify the quantum state that results in either 0 or 1
 5 initial_state = [1/sqrt(2), 1/sqrt(2)]
 6 
 7 classifier_report("Random PQC",
 8   lambda passenger: pqc_classify(backend, initial_state),
 9   train_input,
10   train_labels)
```

```
The precision score of the Random PQC classifier is 0.35
The recall score of the Random PQC classifier is 0.46
The specificity score of the Random PQC classifier is 0.46
The npv score of the Random PQC classifier is 0.57
The information level is: 0.46
```

When we run the `pqc_classify` classifier with the initial state, we can see that it yields the same scores as the random classifier did.

But how these two classifiers create the results is completely different.

The classic "random" classifier uses the function `random` and initializes it, as depicted by the following code snippet.

Listing 3.17: Initialization of classical (pseudo-)random

```python
import random
random.seed(a=None, version=2)
```

We provide `None` as the randomness source (`a`). This implies the function takes a value from the operating system. It appears random, but it is not. If we knew the value it gets from the operating system or if we specified a distinct value ourselves, we could reproduce the exact predictions.

That's why Python's `random`-function generates pseudo-random (see Python-docs) numbers.

By contrast, the PQC generates truly random results (when being run on a real quantum computer). This is in accordance with one of the interpretations of the quantum state of superposition. we discussed in section (3.1).

Nevertheless, we have not used anything quantumic yet which would make us see the difference between classical pseudo-random and quantumic truly random.

3.5 Variational Hybrid Quantum-Classical Algorithm

The PQC binary classifer we created in the previous section 3.4 is as good as the random classifier. Or as poor as. Because it does not provide any increase in the information level.

This is going to change now. So far, we always feed the PQC with the same initial state: $|\psi\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, with the corresponding array in Python: `[1/sqrt(2), 1/sqrt(2)]`.

This state does not take into account the passenger data at all. It is a hypocrite classifier, such as the classifiers we build in section 2.7.Hypocrite classifiers solely use chance when predicting the label of a thing. While such a classifier can yield seemingly good performance in a single measures, such as "precision", it does not reach an average above 0.5 for the four measures that directly result from the confusion matrix (precision, recall, specificity, and NPV). It does not provide any information gain.

In order to improve our classifier, we need to use the passenger data. Even though we prepared the passenger data into normalized numerical data, it does not fit the quantum state vector we need to feed into our PQC. We need to pre-process our passenger data to be computable by a quantum computer.

In fact, we also need to post-process it. We implicitly post-processed the results as part of the `return` statement, as shown in the following snippet.

Listing 3.18

```python
def pqc_classify(backend, passenger_state):
  # ...

  # get the bit 0 or 1
  return int(list(map(lambda item: item[0], counts.items()))[0])
```

Since we have a binary classification task, our prediction is in fact `0` or `1`. Our post-processing is limited to transforming the output format. But in any other setting, post-processing may involve translation from the output of the quantum circuit into a useable prediction.

Altogether, we wrap the PQC into a process of classical pre- and post-processing. This is an algorithm with an outer structure running at a classical computer and an inner component running on a quantum computer. It is a **Variational Hybrid Quantum-Classical Algorithm** and it is a popular approach for near-term quantum devices.

Figure 3.14 shows the overall architecture of our simple Variational Hybrid Quantum-Classical Algorithm.

The data is pre-processed on a classical computer to determine a set of parameters for the PQC. In our simple case, this is the quantum state vector $|\psi\rangle$.

The quantum hardware uses the intitial quantum state, works with it and performs measurements. All its calculations are parameterized. So, they are relatively small and short-lived. In our case, we only measure the quantum state. We do not use any further parameters beyond $|\psi\rangle$.

Finally, the measurement outcomes are post-processed by the classical computer to generate a prediction.

The overall algorithm consists of a closed loop between the classical and

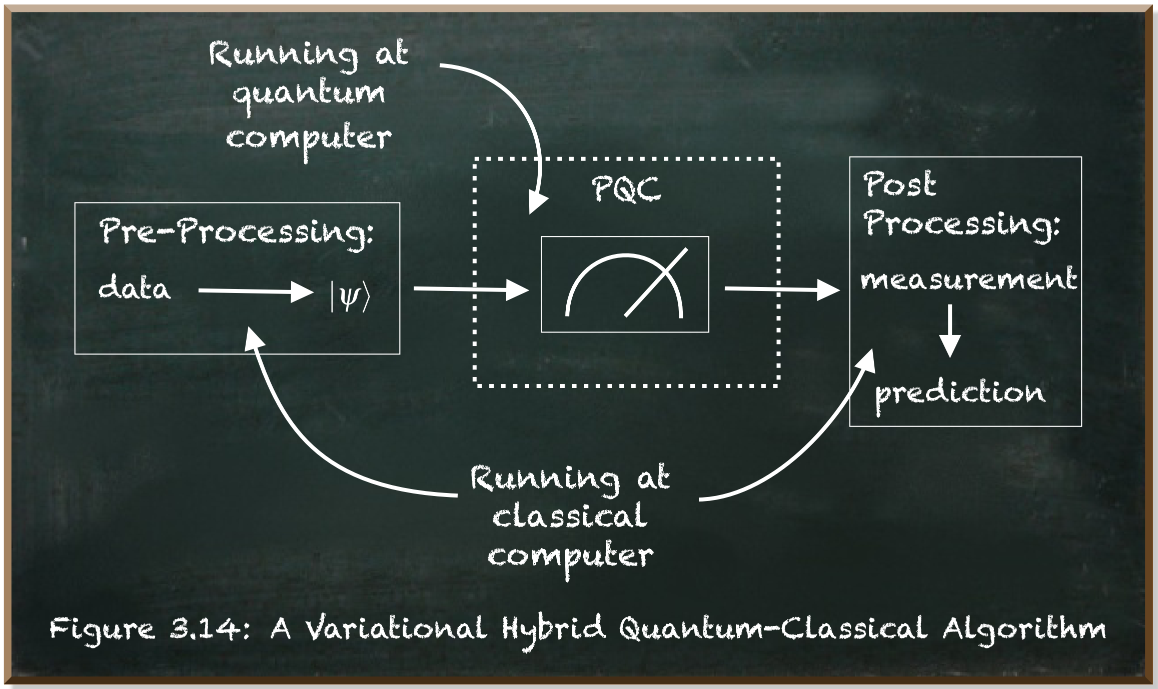


Figure 3.14: A Variational Hybrid Quantum-Classical Algorithm

quantum components.

Let's separate our code thus far into the three parts:

- Pre-processing
- PQC
- Post-processing

Listing 3.19: Pre-processing template

```python
def pre_process(passenger):
  """
  passenger -- the normalized (array of numeric data) passenger data
  returns a valid quantum state
  """
  quantum_state = [1/sqrt(2), 1/sqrt(2)]
  return quantum_state
```

The function `pre_process` takes the passenger data as an array of numeric data.

It returns a valid quantum state vector. In this first version, it returns the balanced state of measuring `0` or `1` with equal probabilities.

Listing 3.20: The parameterized quantum circuit

```python
def pqc(backend, quantum_state):
    """
    backend -- a qiskit backend to run the quantum circuit at
    quantum_state -- a valid quantum state vector
    returns the counts of the measurement
    """

    # Create a quantum circuit with one qubit
    qc = QuantumCircuit(1)

    # Define state |Psi> and initialize the circuit
    qc.initialize(quantum_state, 0)

    # Measure the qubit
    qc.measure_all()

    # run the quantum circuit
    result=execute(qc,backend).result()

    # get the counts, these are either {'0': 1} or {'1': 1}
    counts=result.get_counts(qc)

    return counts
```

The function `pqc` is the PQC. It takes a quantum `backend` and a valid `quantum_state` as input parameters.

It prepares and runs the quantum circuit before it returns the `counts` of its measurements.

Listing 3.21: Post-processing

```python
def post_process(counts):
    """
    counts -- the result of the quantum circuit execution
    returns the prediction
    """
    return int(list(map(lambda item: item[0], counts.items()))[0])
```

The function `post_process` takes the `counts` as input and returns the prediction (see section 3.4 for the detailed explanation of how to transform the `counts` dictionary into the prediction).

Let's put it all together.

Listing 3.22: The scores of the random quantum classifier

```
1 # Tell Qiskit how to simulate our circuit
2 backend = Aer.get_backend('statevector_simulator')
3
4 classifier_report(
5   "Variational",
6   lambda passenger: post_process(pqc(backend, pre_process(passenger))),
7   train_input,
8   train_labels)
```

```
The precision score of the Variational classifier is 0.43
The recall score of the Variational classifier is 0.54
The specificity score of the Variational classifier is 0.51
The npv score of the Variational classifier is 0.62
The information level is: 0.52
```

We first create the `statevector_simulator` backend we can reuse for all our predictions (line 2).

We use the `classifier_report` wrapping function we developed in section 2.7.

Besides an arbitrary name it uses in the output (line 5) the main input is the classifier we provide (line 6).

We provide an anonymous (`lambda`) function (a function without a name) as our classifier. It takes a single parameter `passenger` and runs (from inner to outer) the `pre_process` function with the `passenger` as parameter. We put the result alongside the `backend` into the `pqc` function whose result we put into the `post_process` function.

When we run the pqc classifier with the initial state, we can see that it yields the same scores as the random classifier.

Now, it's finally time to build a real classifier. One that uses the actual passenger data to predict whether the passenger survived the Titanic shipwreck or not.

Let's start at the end. The current post-processing already returns either `0` or `1`. This fits our required output, since `0` represents the passenger died and `1` represents the passenger survived.

The current PQC measures the provided quantum state vector and returns the `counts`. We could leave it unchanged, if we provided as input a vector whose probability corresponds to the passenger's actual probability to survive.

The passenger data consists of an array of seven features. We already transformed all features into numbers between `0` and `1` (section 2.4).

Thus, it is the task of the pre-processing to translate these seven numbers into a quantum state vector whose probability corresponds to the passenger's actual probability to survive.

Finding such a probability is the innate objective of any machine learning algorithm.

Our data consists of seven features. The main assumption is these features determine or at least affected whether a passenger survived or not. If that wasn't the case, we wouldn't be able to predict anything reliably. Let's assume the features determine survival.

The question then is, **how do these seven features determine survival?** Is one feature more important than another? Is there a direct relationship between a feature and survival? Are there any interdependencies between the features, such as if A then B indicates survival. But if not A, then B is irrelevant but C is important.

But before we use sophisticated tools (such as neural networks) that are able to discover complex structures of how the features determine the outcome, we start simple.

We assume all features are independent from each other and each features contributes more or less to the survival or death of the passenger.

Therefore, we say the overall probability of survival $P(survival)$ is the sum of each feature's value F times the feature's weight μ_F ("mu").

$$P(survival) = \sum (F \cdot \mu_F) \tag{3.8}$$

Let's have a look at what this means in Python.

Listing 3.23: weight a passenger's feature

```python
1  def weight_feature(feature, weight):
2    """
3    feature -- the single value of a passenger's feature
4    weight -- the overall weight of this feature
5    returns the weighted feature
6    """
7    return feature*weight
```

The `weight_feature` function calculates and returns the term $F \cdot \mu_F$. This function calculates how much a passenger's feature, the `age` for instance, contributes to this passenger's overall probability to survive. The higher the weighted value, the higher the probability.

Next, we need to add all the weighted features to calculate the overall probability.

Listing 3.24: Calculate the overall probability

```python
1  from functools import reduce
2  
3  def get_overall_probability(features, weights):
4    """
5    features -- list of the features of a passenger
6    weights -- list of all features' weights
7    """
8    return reduce(
9      lambda result, data: result + weight_feature(*data),
10     zip(features, weights),
11     0
12   )
```

The function `get_overall_probability` takes two parameters. First, the list of a passenger's feature values. This is a passenger's data. Second, the list of the feature weights.

We construct a list of tuples for each feature (line 10) containing the feature and its weight. Python's `zip`-function takes two separate lists and creates the respective tuple for each two elements in the lists.

We `reduce` this list of `(feature, weight)` into a single number (line 8). We call the `weight_feature`-function for each of the tuples and add up the results (line 9), starting with the value `0` (line 11).

Now, we need to calculate the weights of the features. These are similar across all passengers. We build the weights upon the correlation coefficients.

The correlation coefficient is a measure of the relationship between two variables. Each variable is a list of values. It denotes how much the value in one list increases as the value of the other list increases. The correlation coefficient can take values between `-1` and `1`.

- A correlation coefficient of `1` means that for every **increase** in one variable, there is a proportional **increase** in the other.
- A correlation coefficient of `-1` means that for every **increase** in one variable, there is a proportional **decrease** in the other.
- A correlation coefficient of `0` means that the two variables are not linearly related.

We calculate the correlation coefficient for each feature in our dataset in relation to the list of `labels`. In the following code, we thus separate our dataset into a list of the columns (line 4).

The term `list(map(lambda passenger: passenger[i], train_input` transforms each passenger's data into its value at the position `i`. And we do this `for i in range(0,7)`. It means we do this for each column.

Listing 3.25: Calculate the correlation coefficients

```python
1 from scipy.stats import spearmanr
2 
3 # separate the training data into a list of the columns
4 columns = [list(map(lambda passenger: passenger[i], train_input)) for i
      in range(0,7)]
5 
6 # calculate the correlation coefficient for each column
7 correlations = list(map(lambda col: spearmanr(col, train_labels)[0],
      columns))
8 correlations
```

```
[-0.3197157323427042,
 -0.578307132534222,
 -0.07119068845990383,
 0.09087327268445101,
 0.13761444319293503,
 0.31592964952566327,
 -0.1745346616390969]
```

There are different types of correlation coefficients. The most frequently used are the Pearson and Spearman correlation methods.

The Pearson correlation is best suited for linear continuous variables whereas the Spearman correlation also works for monotonic ordinal variables. Since we have some categorial data (`Plass`, `Sex`, and `Embarked`), we use the Spearman method to calculate the correlation coefficient.

Scipy provides the function `spearmanr` for us. We call this function for each column and the `train_labels` (line 7). The function returns two values, the correlation coefficient and the p-value. We're only interested in the first (at index `0`).

The correlation coefficients range from `-0.58` to `0.32`.

Let's put this all together in the pre-processing.

Listing 3.26: The weighting pre-processing

```
 1 from math import pi, sin, cos
 2 
 3 def get_state (theta):
 4   """returns a valid state vector from angle theta"""
 5   return [cos(theta/2), sin(theta/2)]
 6 
 7 def pre_process_weighted(passenger):
 8   """
 9   passenger -- the normalized (array of numeric data) passenger data
10   returns a valid quantum state
11   """
12 
13   # caluclate the overall probability
14   mu = get_overall_probability(passenger, correlations)
15 
16   # theta between 0 (|0>) and pi (|1>)
17   quantum_state = get_state((1-mu)*pi)
18 
19   return quantum_state
```

We use the function `get_state` from section 3.2. It takes the angle `theta` and returns a valid quantum state. An angle of 0 denotes the state $|0\rangle$ that is the probability of 100% of measuring `0`. An angle of π denotes the state $|1\rangle$ that is the probability of 100% of measuring `1`.

Accordingly, we multiply the overall probability we calculate at line 14 with `pi` to specify an angle up to π (line 17). Since the correlation coefficients are between -1 and 1 and most of our coefficients are negative, a value of μ towards -1 implies the passenger actually died. Thus, we reverse the angles by calculating `(1-mu)*pi`.

Now, we're ready to run the classifier. Let's feed it into the `classifier_report` wrapping function.

Listing 3.27: Run the PQC with the weighted pre-processing

```python
backend = Aer.get_backend('statevector_simulator')

classifier_report("Variational",
  lambda passenger: post_process(pqc(backend, pre_process_weighted(
    passenger))),
  train_input,
  train_labels)
```

```
The precision score of the Variational classifier is 0.77
The recall score of the Variational classifier is 0.61
The specificity score of the Variational classifier is 0.88
The npv score of the Variational classifier is 0.77
The information level is: 0.76
```

We achieve an overall information level of about 0.73 to 0.77. Not too bad, is it? But before we're start to party, we need to test our classifier. We "trained" the classifier with the training data. So it has seen the data before we just used.

Let's run the classifier with the test dataset.

Listing 3.28: Test the PQC-based classifier on data it has not seen before

```python
classifier_report("Variational-Test",
  lambda passenger: post_process(pqc(backend, pre_process_weighted(
    passenger))),
  test_input,
  test_labels)
```

```
The precision score of the Variational-Test classifier is 0.60
The recall score of the Variational-Test classifier is 0.55
The specificity score of the Variational-Test classifier is 0.85
The npv score of the Variational-Test classifier is 0.82
The information level is: 0.70
```

The overall information level is somewhere between 0.66 and 0.71. This is only a little bit lower than the value we get when running it on the training data. The algorithm seems to generalize (to a certain extent).

Most importantly, in comparison to the hypocrite classifiers, we see a significant increase in the information level. This classifier provides some actual insights. Not too many. But some.

It is our first working Variational Hybrid Quantum-Classical Classifier.

4. What's Next?

You've reached the end of the preview of **Hands-On Quantum Machine Learning with Python**. I hope you enjoyed reading it thus far. And I hope you stay with me in the journey on learning about quantum machine learning.

We have just scratched the very surface of quantum machine learning. We created a simple **Variational Hybrid Quantum-Classical Classification Algorithm**.

But this algorithm is only the beginning. It is among the simplest classifiers we can imagine. Even though we use a **Parameterized Quantum Circuit**, we didn't do anything classical computers can't do.

In the upcoming chapters, we will explore how to create more sophisticated quantum circuits. And we will learn how to use entanglement and interference to reduce the complexity of a problem at hand.

There is a lot to discover in quantum machine learning. And with **Hands-On Quantum Machine Learning With Python**, you will continue discovering it in a practical and easy to follow manner. You'll get all the knowledge you need to implement quantum-based machine learning algorithms.

In this book, we will go far beyond the basics. We will learn how to create state-of-the-art quantum machine learning algorithms.

Hands-On Quantum Machine Learning With Python will equip you with all you need to become a "*Quantum Machine Learning Engineer*" – the job to become the sexiest job of the 2020s.

Hands-On Quantum Machine Learning With Python is currently work in process. While I continue working on the book, I'd appreciate your interest and I'll be happy to provide you with the newest insights I gathered.

Don't miss the regular updates on Substack, Medium, and `www.pyqml.com`.

If you like, please provide me with feedback at `mail@pyqml.com`.

You can also mail me if you have any other question regarding quantum machine learning in general. I'll strive to reply. Most likely, others will have the same question, so I'll be happy to include the answers in the book once it is ready.

Thank you for reading.

5. Appendix A: Solutions

5.1 Solution: Calculate The Qubit State

In exercise ??, we asked for the state of qubit that has a 25% chance of resulting in 0 and 75% of resulting in 1.

The solution is solving the following equation system.

Equation 5.1. This is the definition of a qubit in superposition. This qubit, when observed, has the probability of α^2 to result in 0 and β^2 to result in 1.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix} \tag{5.1}$$

Equation 5.2. This is the required normalization. It requires the sum of the squared amplitudes (α and β) to equal 1.

$$\alpha^2 + \beta^2 = 1 \tag{5.2}$$

Let's regard the probabilities 25% and 75% as fractions and equate them to α^2 and β^2, respectively.

$$\alpha^2 = \frac{1}{4} \Leftrightarrow \alpha = \frac{1}{2} \tag{5.3}$$

and

$$\beta^2 = \frac{3}{4} \Leftrightarrow \beta = \frac{\sqrt{3}}{2} \tag{5.4}$$

Now, we insert 5.3 and 5.4 into equation 5.1:

$$|\psi\rangle = \frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle = \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix}$$

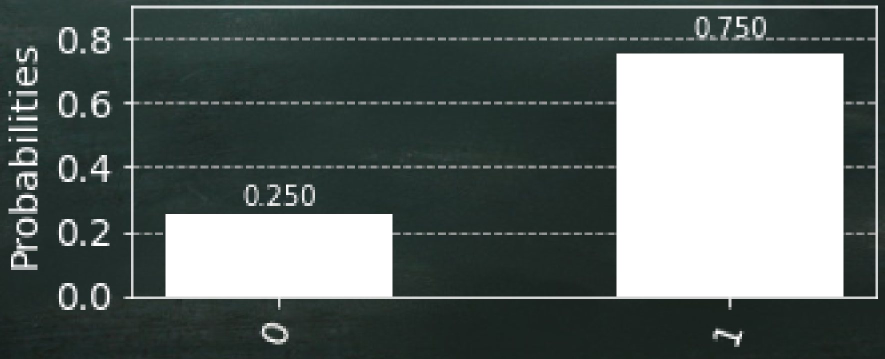
In Python, the array `[1/2, sqrt(3)/2]` represents the vector $\begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix}$

Now, let's open our Jupyter notebook and test our calculation.

Listing 5.1: The qubit with a probability of 0.25 to result in 0

```
from qiskit import QuantumCircuit, execute, Aer
from qiskit.visualization import plot_histogram
from math import sqrt

qc = QuantumCircuit(1)
initial_state = [1/2, sqrt(3)/2] # Here, we insert the state
qc.initialize(initial_state, 0)
backend = Aer.get_backend('statevector_simulator')
result = execute(qc,backend).result()
counts = result.get_counts()
plot_histogram(counts)
```

Figure 5.1: The qubit measurement probabilities



5.2 Solution: Derive The Proof Of The Theta-Formula

In exercise ??, we asked for the proof of the formula: 3.6:

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} cos\frac{\theta}{2} \\ sin\frac{\theta}{2} \end{bmatrix}$$

The structure of this formula derives directly from the definition of the qubit superposition state.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{5.5}$$

To prove equation 3.6, we need to prove the following two terms

$$\alpha = \cos\frac{\theta}{2} \tag{5.6}$$

and

$$\beta = \sin\frac{\theta}{2} \tag{5.7}$$

Let's have a look at our graphical definition of α and β. We already saw this figure 3.5 in section 3.2.

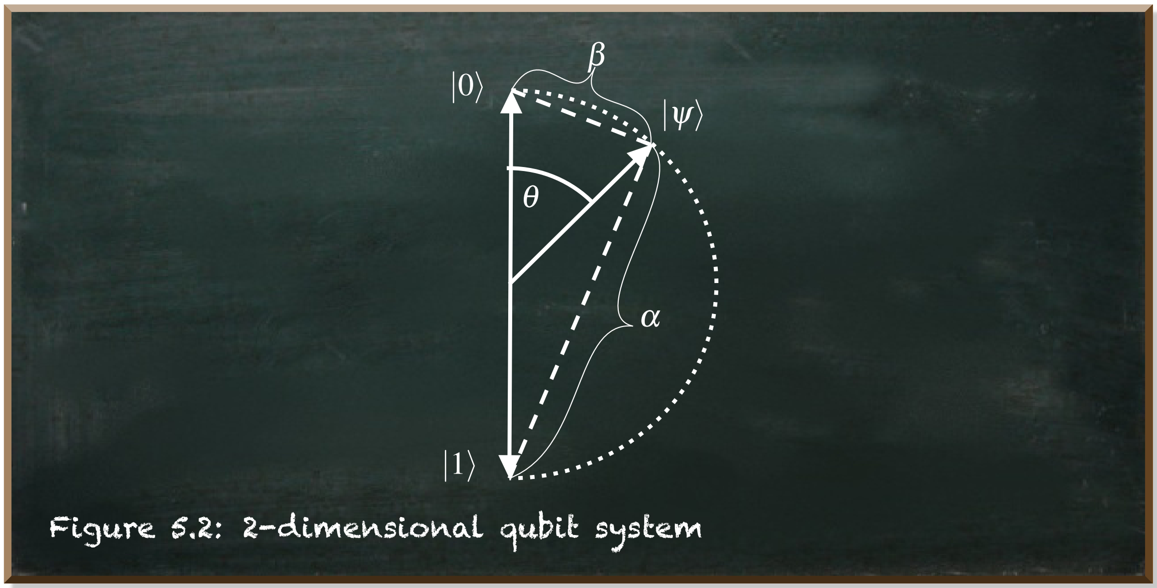


Figure 5.2: 2-dimensional qubit system

α and β describe the proximity to the top and the bottom of the system, respectively. θ is the angle between the standard basis vector: $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and the qubit state vector $|\psi\rangle$ it represents.

Any valid qubit state vector must be normalized:

$$\alpha^2 + \beta^2 = 1 \tag{5.8}$$

This implies all qubit state vectors have the same magnitude (length). Since they all originate in the center, they form a circle with the radius of their magnitude (that is half of the circle diameter).

In such a situation, **Thales' theorem** states, if

- A, B, and C are distinct points on a circle (condition 1)
- where the line AC is a diameter (condition 2)
- then the angle $\angle ABC$ (the angle at point B) is a right angle.

In our case, the heads of $|0\rangle$, $|1\rangle$, and $|\psi\rangle$ represent the points A, B, and C, respectively (satisfy condition 1). The line between $|0\rangle$ and $|1\rangle$ is the diameter (satisfy condition 2). Therefore, the angle at the head of $|\psi\rangle$ is a right angle.

Now, the **Pythagorean theorem** states the area of the square whose side is opposite the right angle (hypotenuse, c) is equal to the sum of the areas of the squares on the other two sides (legs a, b).

$$c^2 = a^2 + b^2 \tag{5.9}$$

When looking at figure 5.2, again, we can see that α and β are the two legs of the rectangular triangle and the diameter of the circle is the hypotenuse. Therefore, we can insert the normalization equation 5.8

$$c = \sqrt{\alpha^2 + \beta^2} = \sqrt{1} = 1 \tag{5.10}$$

The diameter c is two times the radius, thus two times the magnitude of any vector $|\psi\rangle$. The length of $|\psi\rangle$ is thus $\frac{c}{2} = \frac{1}{2}$.

Since all qubit state vectors have the same length, including $|0\rangle$ and $|1\rangle$, there are two isosceles triangles ($\triangle M|0\rangle|\psi\rangle$ and $\triangle M|\psi\rangle|1\rangle$).

Have a look at the following figure 5.3.

You can see the two isosceles triangles. The angles in isosceles triangles at the equal legs are equal, as denoted by γ and δ.

Further, the sum of all three angles in a triangle is 180^o. Therefore,

$$\theta + 2\gamma = 180^o \tag{5.11}$$

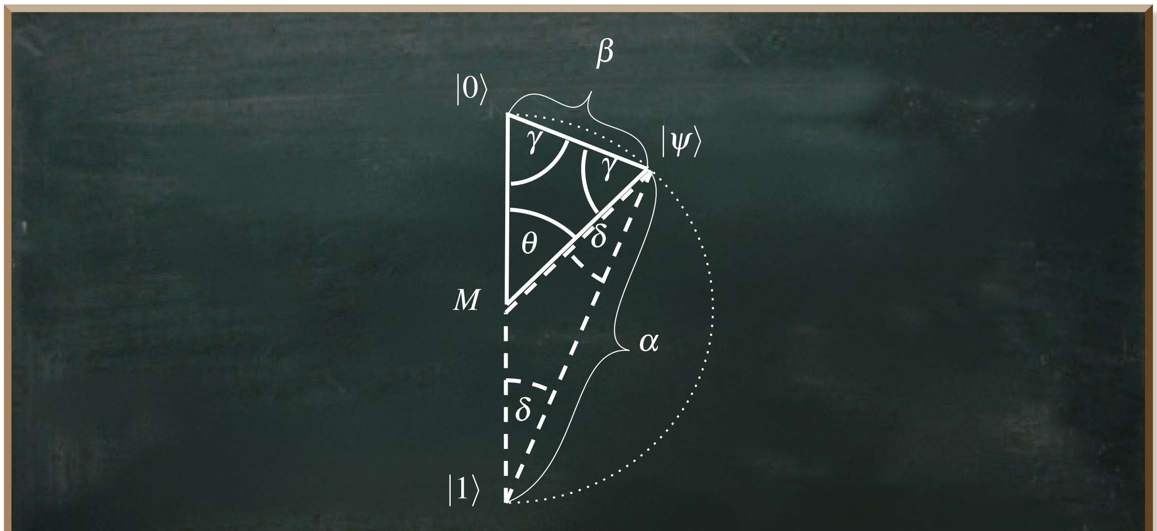


Figure 5.3: Two inner isosceles triangles and an outer rectangular triangle

Let's solve this after γ

$$\gamma = \frac{180^o - \theta}{2} = 90^o - \frac{\theta}{2} \tag{5.12}$$

In a rectangular triangle (the outer one), trigonometric identity says the sine of an angle is the length of the opposite leg divided by the length of the hypotenuse. In our case, this means:

$$\sin\gamma = \frac{\alpha}{1} = \alpha \tag{5.13}$$

Now, we insert equation 5.12:

$$\sin\left(90^o - \frac{\theta}{2}\right) = \alpha \tag{5.14}$$

With $sin(90^o - x) = \cos x$, we can see:

$$\alpha = \cos\frac{\theta}{2}$$

This is the first equation 5.6 to prove.

The further proof works accordingly and is straight forward. At the center (M), the (unnamed) angle inside the dashed triangle is $180^o - \theta$.

$$(180^o - \theta) + 2\delta = 180^o \Leftrightarrow \delta = \frac{\theta}{2} \tag{5.15}$$

Again, we use the trigonometric identity. This time it implies:

$$\sin \delta = \frac{\beta}{1} = \beta \tag{5.16}$$

Finally, we insert 5.15:

$$\sin \frac{\theta}{2} = \beta \tag{5.17}$$

This is the second equation to prove.