Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Quantum Computing Research Project 2020-2021

D'Angelo Stefano
10607664
Rivera Olimpia
10617517
Sanvito Alessandro
10578314

July 16, 2021

POLITECNICO
MILANO 1863

Contents

1. Abstract

Thanks to significant speedups compared with classical algorithms, quantum algorithms promise to revolutionise many application domains, from cryptography to machine learning. However, their integration in classical applications is a relevant challenge due to complex pre- and post-processing tasks. A possible solution to this problem comes from workflows, orchestrated and repeatable patterns of activity.

In this report, we give an overview of two workflow engines to support the execution of quantum workflows: Orquestra and Apache Airflow. Furthermore, we extend the concept of quantum workflows in the case of quantum annealing, providing a case study on linear regression.

2. Introduction

The emerging paradigm of quantum computing could significantly impact many different fields, unlocking efficient solutions to various problems not practically solvable on classical computers. Several hardware providers, such as IBM or Rigetti, recently developed quantum computers, which became publicly accessible. However, today's quantum computers are error-prone and have only limited capabilities. Significant mathematical and technical knowledge is required to perform the complex pre-and post-processing tasks needed to execute a quantum circuit. For instance, an initialisation step has to be added at the beginning of the quantum circuits as today's quantum computers do not allow to load arbitrary data into their registers. Moreover, the small number of qubits of such quantum computers limits the size of the input data. Lastly, calculations on quantum computers are often affected by noise.

Workflows technologies are a promising solution to model quantum computations in workflow models, enabling the integration of quantum computations in classical scenarios and facilitating the orchestration of classical applications and quantum circuits. Such benefits are brought by the following advantageous features characterising such technologies: scalability, reliability, robustness, and transactional processing. Thanks to workflow models, the previously mentioned pre- and post-processing tasks can be easily included and automatically executed by a workflow engine.

3. Quantum Software Lifecycle

The ongoing research focuses on improving existing quantum algorithms, increasing the capabilities of available quantum computers, and developing required software tooling support. For these purposes, a basic understanding of the lifecycle comprising all relevant phases a quantum software application should go through is needed. In many fields of computer science, lifecycles document the different stages a software artefact goes through. The Quantum software lifecycle provides a unified overview of the development of quantum applications [1]. For what concerns the execution of a quantum circuit, several pre -and post-processing tasks, e.g., data preparation or oracle expansion, have to be performed.

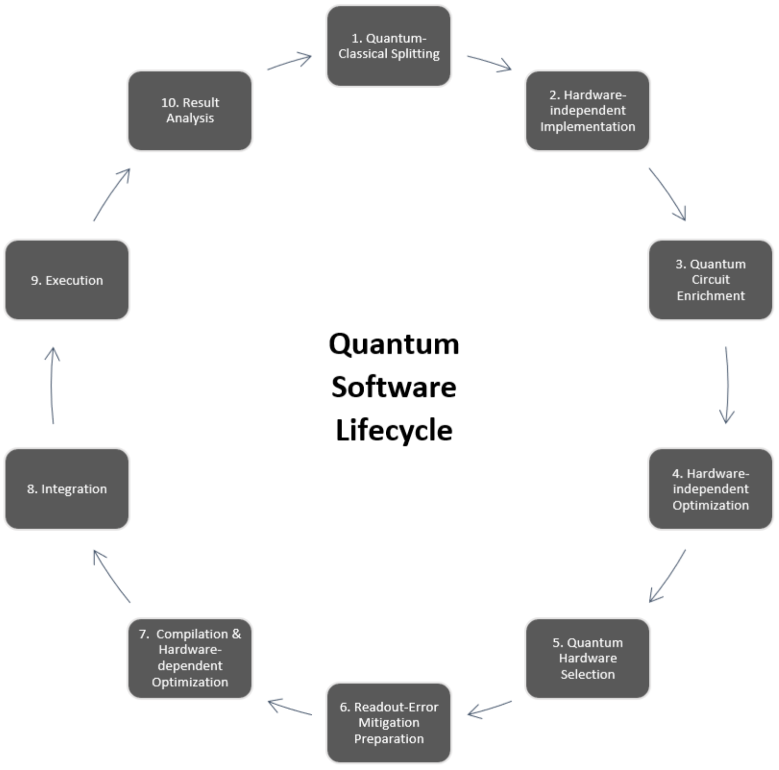

1. Quantum-Classical Splitting
2. Hardware-independent Implementation
3. Quantum Circuit Enrichment
4. Hardware-independent Optimization
5. Quantum Hardware Selection
6. Readout-Error Mitigation Preparation
7. Compilation & Hardware-dependent Optimization
8. Integration
9. Execution
10. Result Analysis
Quantum Software Lifecycle

Figure 3.1: Quantum Software Lifecycle

Consequently, the integration of classical applications with quantum circuits becomes a significant challenge and, although workflow technologies provide the necessary basis for integrating heterogeneous applications, all phases of the quantum software lifecycle, shown in figure 3.1, still need to be implemented as activities in workflow models [2].

4. Workflow Model

Workflows are a set of activities that have to be performed to achieve a goal. The functionality of these activities can be implemented in different ways. Processes are usually specified using standardised workflow languages, such as the Business Process Model and Notation (BPMN) or the Business Process Execution Language (BPEL), making workflow models more portable across different workflow engines.

One advantage of exploiting workflow technologies is the automation of the execution of workflow models using a workflow engine.

Unfortunately, there is currently no means to explicitly model quantum circuit executions in workflow languages and existing workflow engines do not support the invocation of quantum circuits [2].

4.1 Orquestra

Orquestra is a software platform recently developed by Zapata Computing, Inc., a software company specialised in quantum technologies for executing the so-called quantum-enabled workflows. In particular, Orquestra offers the opportunity to "*compose quantum-enabled workflows and execute freely across the full range of quantum devices*"[1]. The platform enables the creation of workflows in a YAML-compatible language named Zapata Quantum Workflow Language (ZQWL). In this way, Orquestra offers solutions to reproduce experiments, repurpose code, share data, and solve dependency problems in projects.

Workflows provide the Quantum Engine step-by-step instructions that describe which actions to perform on which data. Once submitted to the Orquestra Quantum Engine (OQE) servers via a REST API, workflows can be conducted across quantum computers from different providers, such as IBM, Rigetti, and Atos, quantum gate model computers, quantum annealers, and quantum-inspired classical devices.

Using the metaphor of a real orchestra, the workflow model provided by Orquestra can be seen as follows:

- A **workflow template**, that represents the director dictating what happens in the workflow.
- The **steps**, representing the musicians who play parts of the melody. They comprise functions, or tasks, that perform a particular step.
- The **artefacts**, that are either parameters or data passed from the workflow template to the steps or between steps themselves. Once the workflow ends, it outputs an artefact as a result.

When building a workflow, a list of **components** that have to be imported is required. To correctly import them, the developer needs to specify the type of each component. The specified type of each component in the workflow determines how the bits are retrieved, and where they are placed into the run-time environment of the workflow. Orquestra provides a default type **git**, which allows specifying a Github repository where the components are located.

[1]https://orquestra.io/post/workflows-and-quantum-go-together-like-peanut-butter-jelly

Components represent reusable bits within a step of a workflow. When a workflow step begins, Orquestra retrieves each component and ensures it is placed into the run-time environment of the workflow, so that it can be invoked by the steps. An Orquestra workflow executes a collection of **steps**. Examples of workflows can be found in the last chapter (see: Annexes).

Furthermore, Orquestra provides several open-source components for common quantum algorithms and machine-learning tasks that can be used within the workflows.

4.2 Apache Airflow

Apache Airflow is an open-source workflow management platform initially developed by Airbnb. This solution allows to *programmatically author, schedule and monitor workflows* in distributed settings [3].

Airflow represents workflows as directed acyclic graphs (DAGs), in which each node is an instance of an operator, an (often) atomic task that does not need to share resources with other operators. Airflow offers many readily available operators to execute various tasks, from python scripts, to bash scripts to operations interfacing to cloud platforms. Dependencies between tasks are graphically represented as directed lines, and no cycle is allowed.

Tasks can exchange information in two ways:

- by relying on external data storage services
- through XCOMs

XCOMs are a native communication facility that allows exchanging messages identified by a string key. Any serializable object can become a payload of an XCOM, which also contains information about the generation timestamp and the origin task. Airflow persists every XCOM exchanged during the execution of a DAG in a local database.

In Airflow, developers can specify a DAG as a pure python script, parametrized through templating. In standard Airflow, the script instantiates each operator with the correct parameters, then it specifies the dependencies between the tasks explicitly.

If the tasks are mostly python scripts, a developer can leverage the Taskflow API instead. Through the API, a developer only needs to define functions and call them in the desired sequence. Functions are then automatically converted into tasks by Taskflow, which also recognises the dependencies between them, and moves inputs and outputs between them as XCOMs.

Finally, a GUI is available to visualise and run DAGs and their results.

5. Quantum Annealing

The application of workflows to algorithms running on Quantum Annealears is still quite an unexplored topic in research. To properly understand the differences between the gate model and quantum annealers in terms of workflow models, it is first of all necessary to understand how quantum annealing works.

Quantum Annealing is a process similar to Simulated Annealing, an optimisation technique inspired by the **Annealing process**. Annealing is typically used in metallurgy and it involves heating a metal and then slowly cooling it down. Because of the heat, the atoms in the metal can break previous bounds and reorganise themselves in a lower energy state changing the properties of the solid. If the initial temperature is high enough and the cooling process adequately slow, the metal reaches the so-called **ground state**, i.e. the crystalline structure.

In the **Simulated Annealing** technique, the atomic structure becomes the variable assignment, the solution, while the energy of the solid represents the fitness of the solution. In practice, the solution is initialised and evaluated at a random high temperature. Then random changes are applied to it, like atoms rearranging themselves, and the new solution is accepted if it is a better fit than the previous one, otherwise, it is accepted with a certain probability, defined as a function of the temperature. Then the temperature is reduced and the steps are repeated until the temperature reaches a certain ϵ. At that point, the algorithm outputs the best solution.

Quantum Annealing, on the other hand, exploits the in-hardware tunnelling effect instead of thermal jumps, therefore yielding better results faster, as shown in Figure 5.1. To solve an optimisation problem with Quantum Annealing, the problem needs to be reformulated as the energy of a quantum system, either in the Ising formulation $H(\sigma) = -\sum_{<ij>} J_{ij}\sigma_i\sigma_j - \mu \sum_j h_j\sigma_j$ or in the QUBO (Quadratic Unconstrained Binary Optimisation) formulation $y = x^T Q x$, where the state of minimum energy represents the solution to the problem under scrutiny. The quantum system is then evolved from an initial configuration to the configuration representing the problem, keeping it in the minimum energy state. Once the transition is completed, the measured state should be the solution to the optimisation problem or an approximation of it.

This approach is substantially different from the gate model, and it needs different workflow models to be fully exploited.

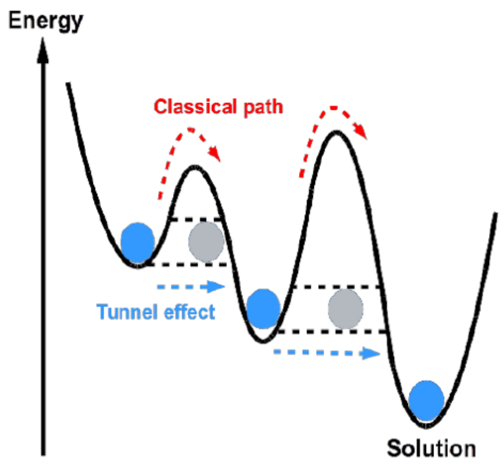


Figure 5.1: Quantum tunnelling process

6. Algorithms

To properly test quantum workflows on both Orquestra and Apache Airflow, we selected the following quantum algorithms: Grover's algorithm and Quantum Linear Regression algorithm. While the former represents an example of a quantum algorithm implemented on the gate model, promising to solve search problems more efficiently than possible on classical computers, the latter is a well-known algorithm from the machine learning field that can be described in terms of **QUBO** (Quadratic Unconstrained Binary Optimisation) problem and hence adapted to the *quantum annealing* model. Our purpose is to show how such distinct algorithms that exploit different quantum computational paradigms can be transformed into workflows and run on different platforms.

6.1 Grover's Algorithm

The purpose of Grover's algorithm is to search the index w of an item in an unsorted list of N items. It has a complexity of $\mathcal{O}(\sqrt{N})$ oracle executions, leading to a quadratic speed-up compared to classical unstructured search, which performs the same task in $\mathcal{O}(N)$.
The algorithm consists of the following steps:

1. Initialise the circuit to the uniform superposition $|\boldsymbol{s}\rangle$ of the input qubits, which is easily prepared by applying a set of Hadamard gates
$$|\boldsymbol{s}\rangle = \boldsymbol{H}^{\otimes n}\,|\boldsymbol{0}^n\rangle$$
2. Apply an **oracle reflection** U_f to the state $|\boldsymbol{s}\rangle$
3. Apply an additional reflection U_s to the state $|\boldsymbol{s}\rangle$ in the form:
$$U_s = 2\,|\boldsymbol{s}\rangle\,\langle\boldsymbol{s}| - 1$$
4. Measure the results

The transformation $U_s U_f|\boldsymbol{s}\rangle$ geometrically applies a rotation of the initial state $|\boldsymbol{s}\rangle$ towards the searched index $|\boldsymbol{w}\rangle$. In order for the algorithm to produce better results, that is to find $\boldsymbol{w}$ with a higher probability, this transformation has to be applied t times, with $t \sim\sqrt{N}$, where N is the number of qubits[4]. A two qubits implementation of the algorithms could be the one presented in Figure 6.1. Our implementation of the algorithm is inspired by the one provided in qiskit's textbook [5] and it is meant to adapt to the workflow model proposed by Weder et al. [2], also supporting a generic number of qubits N.

The first stage of Grover's workflow consists of building the parts of the circuit not dependent on the oracle or external data. The reflection operator is therefore represented as a black box in the first stage. Then the workflow applies the Oracle Expansion, where the oracle is inserted into the circuit. Figure 6.2 shows the structure of the oracle when the element to search is $\boldsymbol{w}$ = 01. The oracle is realized by applying the following mapping:

- $0 \rightarrow q_1 \quad \Rightarrow$ X gate
- $1 \rightarrow q_0 \quad \Rightarrow$ no gate

In general, the oracle block can be divided into an X-gated layer, a multi-controlled Z-gate applied to the last qubit and another X-gated layer, equal to the first one. Both the X-gated layers contain an X gate for each 0 in the element to search, starting from the last qubit [6].

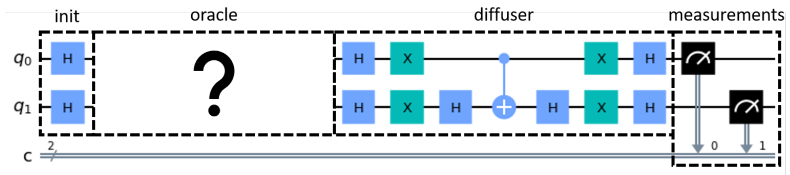


Figure 6.1: Generic quantum circuit with two qubits

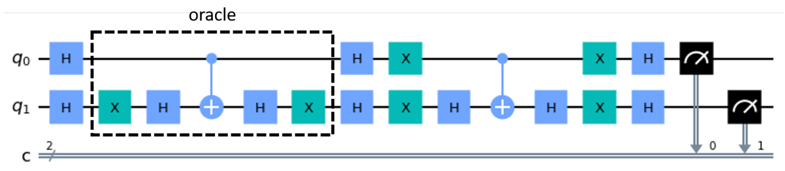


Figure 6.2: Oracle expansion of the circuit with $w = 01$

Finally, the circuit is executed on a specific backend ("ibm" or "qasm_simulator"), as in the Execution stage of the simplified version of the Quantum Software Lifecycle proposed by Weder et al. [2], shown in Figure 6.3. Our workflow model for Grover's algorithm covers all these stages except the Data Preparation, since there is no data to integrate to the circuit, and the Readout-Error Mitigation, not included for simplicity.

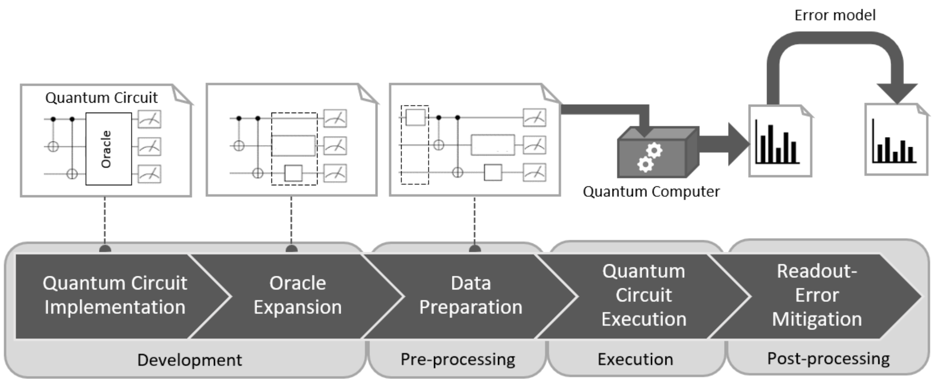


Figure 6.3: Simplified development and execution process of a quantum circuit

6.2 Linear Regression

Linear regression is a popular machine learning problem, in which real processes are approximated with linear models and the goal is to learn a mapping from input X to a continuous output Y. Given the following inputs:

$$N \in \mathbb{N}\ samples,\ d \in \mathbb{N}\ features,\ X \in \mathbb{R}^{N\times(d+1)},\ Y \in \mathbb{R}^N$$

the regression problem's output is the vector of weights:

$$w \in \mathbb{R}^{d+1}\ such\ that\ Y \approx Xw$$

The LR minimisation problem is usually written as:

$$\min_{\forall w \in \mathbb{R}^d} ||Xw - Y||_2^2$$

which is the Euclidean error function, where the Euclidean distance between the two vectors Xw and Y is expressed using the l_2 norm, which is calculated as the square root of the sum of the squared vector values. The regression problem has an analytical solution, given by:

$$w = (X^T X)^{-1} X^T Y$$

The time complexity of this classical closed-form linear regression is $\mathcal{O}(Nd^2 + d^3)$, because of the matrix inversion.

Linear regression is one of the machine learning models for which non-conventional computing paradigms like quantum computing can be leveraged to solve the training problem more efficiently. In particular, adiabatic quantum computers, that exploit quantum annealing's fluctuations to find the global minimum of a given objective function, claim to solve optimisation problems faster than classical computers[7]. To exploit adiabatic quantum computers like D-Wave 2000Q, training problems of machine learning models must be formulated as QUBO problems. In the particular case of linear regression, the minimisation problem defined above can be rewritten as:

$$\min_{\forall w \in \mathbb{R}^d} w^T X^T X w - 2 w^T X^T Y + Y^T Y$$

which yields the same result as:

$$\min_{\forall w \in \mathbb{R}^d} w^T X^T X w - 2 w^T X^T Y$$

Next, a precision matrix $P \in \mathbb{R}^{(d+1)\times K(d+1)}$ and a binary vector $\tilde{w} \in \mathbb{B}^{K(d+1)}$ with arbitrary precision $K \in \mathbb{N}$ such that:

$$w \approx P\tilde{w}$$

are introduced and the minimisation problem can be rewritten as follows:

$$\min_{\forall w \in \mathbb{R}^d} \tilde{w}^T P^T X^T X P \tilde{w} - 2 \tilde{w}^T P^T X^T Y$$

The optimisation problem can now be solved using an adiabatic quantum computer since it has been obtained an equation having the form of a quadratic unconstrained binary optimisation. From this very last equation, it can also be inferred that considering $\mathcal{O}(1)$ annealing time, the QUBO formulation has a complexity of $\mathcal{O}(Nd^2K^2)$, in which K is the number of binary variables introduced when converting the d + 1 weights in binary form. Therefore, the complexity has a quadratic dependency from K, which was assumed to be a variable in the above analysis. However, by assuming a fixed precision of quantum computers, which is also the case in classical computers, and considering K as a constant, time complexity becomes $\mathcal{O}(Nd^2)$ [8].

Figure 6.4 presents a possible workflow model we have identified to support QUBO formulations for training machine learning models on quantum annealers.

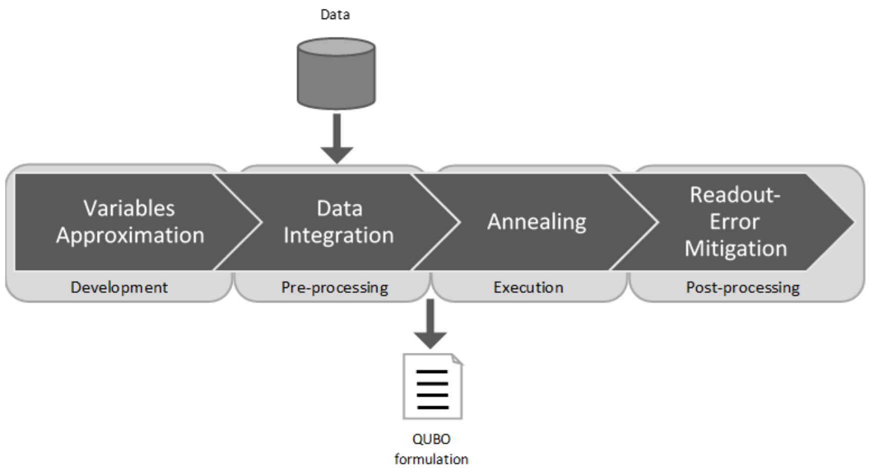


Figure 6.4: Quantum annealing ML workflow.

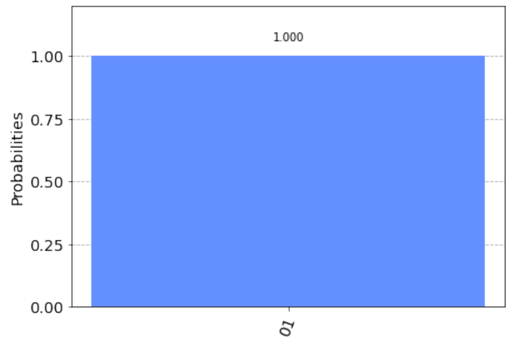
The Variable Approximation phase, in which the precision matrix P is built, is independent of the dataset and may not be always performed since the variable conversion from a binary form into a continuous one is not necessary for all problems. In Data Integration we build the components of the QUBO problem that depend on the dataset and we output the matrix formulation of the problem. The data integration phase takes the data as input and outputs the QUBO formulation, which is then used in the Annealing phase to solve the problem. In our specific case, we did not measure much noise on the problem's output and therefore we decided not to add the Readout-Error Mitigation part in our workflow.

The workflow model has been built with an analogy with the one snow in Figure 6.3. The Variable Approximation phase can be matched with the Quantum Circuit Implementation one, whereas the Data Integration step can be seen somehow as a combination of the Oracle Expansions and the Data Preparation phases. Finally, the remaining phases are the equivalent of the two workflows.

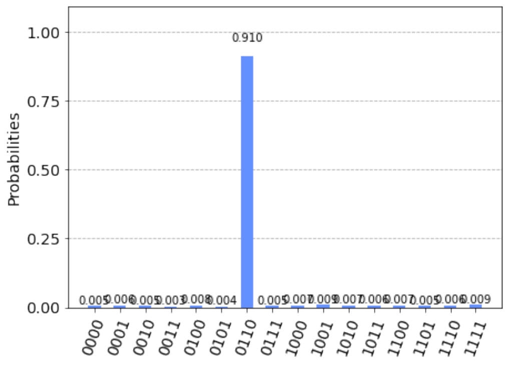
7. Results

7.1 Grover's Algorithm

Grover's algorithm has been tested both with the *IBM* backend and the *qasm_simulator*. For each backend, a 2-qubit circuit and a 4-qubit one have been run and measured.

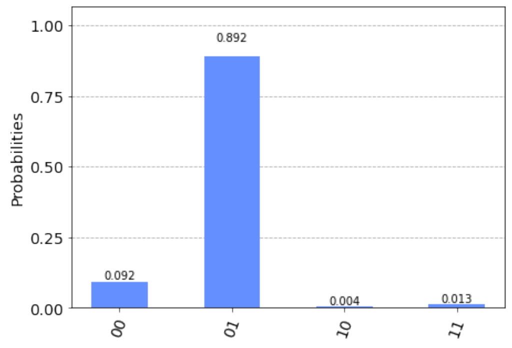


(a) Two qubits with searched element $w = 01$

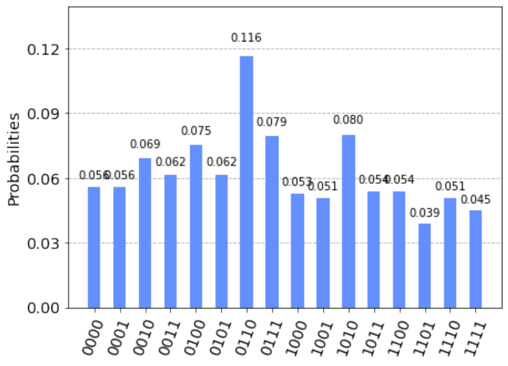


(b) Four qubits with searched element $w = 0110$

Figure 7.1: *qasm_simulator* results



(a) Two qubits with searched element $w = 01$



(b) Four qubits with searched element $w = 0110$

Figure 7.2: *IBMQ* backend results

Figure 7.1 and Figure 7.2 show the results of the runs. Results are correct in all of them since the elements with the highest probabilities are the searched elements specified at the beginning of the workflows. However, the *IBM* backend yields worse results as the number of qubits in the circuit increases. This is because the circuit itself becomes deeper, amplifying the noise present in real quantum computers, such as those owned by IBM. In Grover's case, with four or more qubits, as shown in Figure 7.2(b), the circuit will be deeper, compared to the two qubits case, because of the repetition of the two oracle and diffuser blocks (see Section 6.1). As a consequence, some runs can lead to incorrect results.

7.2 Linear Regression

To properly test quantum linear regression, we have chosen three popular datasets provided by the ScikitLearn library [9]: the Iris dataset, the Boston dataset, and the diabetes dataset.

We have chosen the feature-target combinations to represent scenarios with either a positive, a negative, or a loose statistical correlation between the variables. From the data available, we trained the closed-form linear regression from the *LinearRegression* model available in ScikitLearn.

Quantum linear regression, on the other hand, was infeasible on a purely quantum device due to the density of the problems, which led to poor results. Therefore, we relied on Leap's quantum-classical hybrid solver, which in part uses classical hardware to make dense problems tractable while trying to achieve a quantum speedup [10]. Hence, the results presented in Table 7.1 are a pessimistic estimate of what future, denser quantum annealers might achieve. The variable to optimise was approximated as a 32-bit two-complement fixed-point binary number.

To measure the percentage error of quantum linear regression compared to a closed-form solution, we rely on the l_2, l_1 and l_∞ norms, measured as

$$||w||_2 = \sqrt{\sum_{n=1}^{len(w)} |w|_n^2}$$

$$||w||_1 = \sum_{n=1}^{len(w)} |w|_n$$

$$||w||_\infty = \max_n |w|_n$$

The results show that, in any case, quantum linear regression reaches the correct solution with minimal error.

A further visual comparison between the closed-form and the quantum linear regression is available in the Annexes.

Table 7.1: Percentage of error between the closed form result w and the quantum result $\tilde{w}$ on different datasets in terms of l_2, l_1 and l_∞ norms.

Dataset	**Feature**	**Target**	w	$\tilde{w}$	$\frac{\|\|w-\tilde{w}\|\|_2}{\|\|w\|\|_2}$	$\frac{\|\|w-\tilde{w}\|\|_1}{\|\|w\|\|_1}$	$\frac{\|\|w-\tilde{w}\|\|_\infty}{\|\|w\|\|_\infty}$
Iris plants	Petal length	Petal width	$\begin{pmatrix}0.00\\0.96\end{pmatrix}$	$\begin{pmatrix}0.00\\1.00\end{pmatrix}$	3.8684%	3.8699%	3.8684%
Boston house prices	CRIM	MEDV	$\begin{pmatrix}0.00\\-0.39\end{pmatrix}$	$\begin{pmatrix}0.00\\-0.3750\end{pmatrix}$	3.4263%	3.4303%	3.4263%
Diabetes	BMI	Diabetes	$\begin{pmatrix}0.00\\0.59\end{pmatrix}$	$\begin{pmatrix}0.00\\0.59\end{pmatrix}$	0.0900%	0.0900%	0.0900%

7.3 Orquestra vs Apache Airflow

Throughout the project, we had the opportunity to implement various quantum algorithms on Orquestra and Apache Airflow. We present the most relevant differences in Table 7.2.

On the one hand, Orquestra offers many off-the-shelf algorithms and libraries to create quantum workflows based on Zapata Computing's domain expertise. Although components specifically targeting quantum annealers are not yet fully available, the company plans to support the technology soon.

On the other hand, Airflow offers a less specialised environment with no already implemented quantum operator. It will be up to the community to develop new operators to support quantum tasks. The great advantage of Airflow over Orquestra is in the definition of workflows. Python scripts are more manageable than YAML-like files to define workflows, and the Taskflow API is particularly suitable for quantum algorithms because most of the quantum libraries are Python libraries. Moreover, Airflow provides a GUI to manage workflows, which makes for a better user experience.

Both features might be implemented in further iterations of Orquestra.

Table 7.2: Comparison between Orquestra and Apache Airflow.

	Orquestra	**Apache Airflow**
Company	Zapata Computing	Apache Software Foundation
Open source	No	Yes
Free	No	Yes
Workflow language	YAML-like	Python
GUI	Soon	Yes
Dedicated quantum tasks	Yes	Implementable
Quantum tasks libraries	Available	N.A.

8. Conclusions

Our work was mainly focused on the analysis of **Orquestra**, a platform to execute quantum **workflows**. To experiment with its functionalities, we adapted two algorithms to this engine: Grover's algorithm and Quantum Linear Regression. The main guideline throughout our work was the **Quantum Software Lifecycle**, which was deeply reflected into Grover's workflow due to the presence of the *oracle expansion* step.

Then, we extended the concept of quantum workflows from the gate model to the quantum annealing paradigm, exploring a standard Machine Learning algorithm, which is Linear Regression. Consequently, we formulated this training problem as a QUBO problem to embed it onto the D-Wave 2000Q quantum machine, proposing a workflow model also for Quantum Linear Regression.

As a next step, we measured the results of both the algorithms, explaining the main metrics and setups used in the evaluation of various runs and highlighting the existing differences between the two engines used to run the workflows.

References

[1] B. Weder et al. "The Quantum Software Lifecycle". In: *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS '20)* (Nov. 2020). DOI: `https://dl.acm.org/doi/10.1145/3412451.3428497`.

[2] B. Weder et al. "Integrating Quantum Computing into Workflow Modeling and Execution". In: *Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2020)* (Dec. 2020), pp. 279–291. DOI: `10.1109/UCC48980.2020.00046`.

[3] Apache Airflow Development Team. *Apache Airflow Documentation*. 2021. URL: `https://airflow.apache.org/docs/apache-airflow/stable/index.html`.

[4] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, 212–219. ISBN: 0897917855. DOI: `10.1145/237814.237866`. URL: `https://doi.org/10.1145/237814.237866`.

[5] Qiskit Development Team. *Qiskit Textbook*. 2021. URL: `https://qiskit.org/textbook/ch-algorithms/grover.html`.

[6] V. B. Karlsson and P. Strömberg. "4-qubit Grover's algorithm implemented for the ibmqx5 architecture". In: (2018), p. 34.

[7] D-Wave. *White paper: Computational Power Consumption and Speedup*. Tech. rep. 14-1005A-D. D-Wave, 2017, p. 2.

[8] Prasanna Date, Davis Arthur, and Lauren Pusey-Nazzaro. *QUBO Formulations for Training Machine Learning Models*. 2020. arXiv: `2008.02369 [cs.LG]`.

[9] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project". In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[10] D-Wave Systems Inc. *D-Wave System Documentation*. 2021. URL: `https://docs.dwavesys.com/docs/latest/doc_getting_started.html`.

Annexes

Grover's algorithm workflow

Orquestra

```yaml
# Workflow API version
apiVersion: io.orquestra.workflow/1.0.0

# Prefix for workflow ID
name: grover

# List of components needed by workflow
imports:
- name: main-repository
  type: git
  parameters:
    repository: "git@github.com:stefanodangelo/grover-orquestra.git"
    branch: "main"
- name: z-quantum-core
  type: git
  parameters:
    repository: "git@github.com:zapatacomputing/z-quantum-core.git"
    branch: "0.5.0"

steps:
# Step 1
- name: circuit-preparation
  config:
    runtime:
      language: python3
      imports: [main-repository, z-quantum-core]
      parameters:
        file: main-repository/steps/build_circuit.py
        function: build_circuit
  inputs:
  - n_qubits: 2
    type: int
  outputs:
  - name: circuit
    type: circuit
```

```
# Step 2
- name: oracle-expansion
  passed: [circuit-preparation]
  config:
    runtime:
      language: python3
      imports: [main-repository, z-quantum-core]
      parameters:
        file: main-repository/steps/oracle_expansion.py
        function: expand_oracle
  inputs:
  - circuit: ((circuit-preparation.circuit))
    type: circuit
  - element_to_search: "01"
    type: string
  outputs:
  - name: expanded-circuit
    type: circuit

# Step 3
- name: execute
  passed: [oracle-expansion]
  config:
    runtime:
      language: python3
      imports: [main-repository, z-quantum-core]
      parameters:
        file: main-repository/steps/measure.py
        function: run_and_measure
  inputs:
  - circuit: ((oracle-expansion.expanded-circuit))
    type: circuit
  - backend: "ibm"
    type: string
  outputs:
  - name: measurements
    type: json

types:
- circuit
- json
```

Airflow

```
from math import sqrt

from airflow.decorators import dag, task
```

```python
from airflow.utils.dates import days_ago
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister

from src.functions import find_all_mcz, remove_x_gates
from src.functions import get_results_from_IBM, get_simulation_results
from src.functions import init, oracle, diffuser, add_measurements
from src.utils import from_json
from src.utils import to_zap

# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'airflow',
}


@dag(default_args=default_args, start_date=days_ago(0))
def grover():
    @task()
    def circuit_preparation(n_qubits: int = 4) -> dict:
        # Define registers
        qr = QuantumRegister(n_qubits)
        cr = ClassicalRegister(n_qubits)

        # Create circuit
        qc = QuantumCircuit(qr, cr)

        # Build circuit
        qc = init(qc, qr)
        for i in range(round(sqrt(n_qubits))):
            qc = oracle(qc, qr)
            qc.barrier(qr)
            qc.barrier(qr)
            qc = diffuser(qc, qr)
            qc.barrier(qr)
        qc = add_measurements(qc, qr, cr)

        return to_zap(qc)

    @task()
    def oracle_expansion(circuit: dict, element_to_search: str = "0110") -> dict:
        element_to_search = element_to_search[::-1]  # reverse the string in order to
         ↪  correctly apply the oracle function

        # load circuit data
        metadata = circuit

        # remove X gates where not needed
        mcz_positions = find_all_mcz(metadata)
        for position in mcz_positions:
            remove_x_gates(metadata['gates'], position, element_to_search,
             ↪  len(element_to_search))

        return metadata

    @task()
    def execute(
            circuit:dict,
            backend: str = "qasm_simulator",
            token: str = "f1b53e81357761bafbbb2d7a71e"
                        + "aa26cfc0d9df4eeb53513096c5cee9"
                        + "0ec01bcc8ba6fa547ea4988a1b9"
                        + "136f2abf8090a133546389581b3c51a1471dad8749e6"
    ):
        print(circuit)
        qc = from_json(circuit)

        if backend == 'ibm':
            results = get_results_from_IBM(qc, token)
            measurements = results.get_counts(qc)
        else:
            results = get_simulation_results(qc, backend)

            # <--- code used only for plot task --->
            old_keys = list(results.keys())
            new_keys = list(
                map(lambda x: x[-len(qc.qubits):], old_keys))  # needed because otherwise
                 ↪  keys would be repated

            measurements = {new_keys[i]: results[old_keys[i]] for i in
             ↪  range(len(new_keys))}

        return measurements

    incremental_circuit = circuit_preparation()
    final_circuit = oracle_expansion(incremental_circuit)
    measurement = execute(circuit=final_circuit)


grover_dag = grover()
```

Quantum linear regression workflow

Orquestra

```
# Workflow API version
apiVersion: io.orquestra.workflow/1.0.0

# Prefix for workflow ID
name: sa-linear-regression

imports:
- name: z-quantum-core
  type: git
  parameters:
    repository: "git@github.com:zapatacomputing/z-quantum-core.git"
    branch: "master"
- name: z-quantum-qubo
  type: git
  parameters:
    repository: "git@github.com:zapatacomputing/z-quantum-qubo.git"
    branch: "master"
- name: linear-regression-orquestra
  type: git
  parameters:
    repository: "git@github.com:Alexdruso/linear-regression-orquestra.git"
    branch: "main"


steps:
- name: load-dataset
  config:
    runtime:
      language: python3
      imports: [linear-regression-orquestra]
      parameters:
        file: linear-regression-orquestra/steps/dataset_preprocessing.py
        function: load_dataset
  inputs:
  - url: 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
    type: str
  - names: ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
    type: List[str]

  outputs:
  - name: dataset
    type: dataset

- name: preprocess-dataset
  passed: [load-dataset]
```

```yaml
  config:
    runtime:
      language: python3
      imports: [linear-regression-orquestra]
      parameters:
        file: linear-regression-orquestra/steps/dataset_preprocessing.py
        function: preprocess_dataset
  inputs:
  - dataset: ((load-dataset.dataset))
    type: dataset
  - features: ['petal-length']
    type: List[str]
  - target: 'petal-width'
    type: str

  outputs:
  - name: preprocessed_dataset
    type: dataset

- name: get-precision
  config:
    runtime:
      language: python3
      imports: [linear-regression-orquestra]
      parameters:
        file: linear-regression-orquestra/steps/qubo.py
        function: get_precision
  inputs:
  - num_bits: 32
    type: int
  - num_weights: 2
    type: int
  - point_position: 0.5
    type: float

  outputs:
  - name: precision_matrix
    type: matrix

- name: get-problem-q
  passed: [get-precision, preprocess-dataset]
  config:
    runtime:
      language: python3
      imports: [linear-regression-orquestra]
      parameters:
        file: linear-regression-orquestra/steps/qubo.py
        function: get_problem_matrix
  inputs:
```

```
  - precision_matrix: ((get-precision.precision_matrix))
    type: matrix
  - preprocessed_dataset: ((preprocess-dataset.preprocessed_dataset))
    type: dataset

  outputs:
  - name: q
    type: matrix

- name: solve-qubo
  passed: [get-problem-q]
  config:
    runtime:
      language: python3
      imports: [linear-regression-orquestra]
      parameters:
        file: linear-regression-orquestra/steps/qubo.py
        function: anneal
  inputs:
  - q: ((get-problem-q.q))
    type: matrix
  - precision_matrix: ((get-precision.precision_matrix))
    type: matrix

  outputs:
  - name: solution
    type: solution

types:
- dataset
- str
- List[str]
- matrix
- solution
```

Airflow

```
import dimod
from airflow.decorators import dag, task
from airflow.models import Variable
from airflow.utils.dates import days_ago

from src.data import *
from src.linear_regression import *

from dwave.system.samplers import LeapHybridSampler
from dwave.system.composites import EmbeddingComposite

```

```python
# These args will get passed on to each operator
# You can override them on a per-task basis during operator initialization
default_args = {
    'owner': 'airflow',
}


@dag(default_args=default_args, start_date=days_ago(0))
def quantum_annealing_linear_regression():
    @task()
    def load_dataset() -> dict:
        url = Variable.get('url')
        names = [name for name in Variable.get('names').split(', ')]
        return get_dataset(url=url, names=names).to_dict()

    @task(multiple_outputs=True)
    def preprocess_dataset(dataset: dict) -> dict:
        features = [name for name in Variable.get('features').split(', ')]
        target = Variable.get('target')

        df = pd.DataFrame.from_dict(dataset)

        x, y = preprocess_data(dataset=df, features=features, target=target)

        return {
            'x': x.tolist(),
            'y': y.tolist()
        }

    @task()
    def get_precision() -> list:
        num_bits = int(Variable.get('num_bits'))
        num_weights = int(Variable.get('num_weights'))
        point_position = float(Variable.get('point_position'))

        return build_precision_matrix(
            num_weights=num_weights,
            precision=build_precision_two_complement_fixed_point(
                num_bits=num_bits,
                point_position=point_position
            )
        ).tolist()

    @task()
    def get_problem_matrix(precision: list, x: list, y: list) -> list:
        precision = np.asarray(precision)
        x = np.asarray(x)
        y = np.asarray(y)
        return build_Q(
            precision_matrix=precision,
            feature_matrix=x,
            target_matrix=y
        ).tolist()

    @task()
    def anneal(q: list, precision: list) -> list:
        q = np.asarray(q)
        precision = np.asarray(precision)
        array_bqm = dimod.AdjArrayBQM(q, 'BINARY')

        # Run the QUBO on the solver from your config file
        sampler = LeapHybridSampler()

        response_QPU = sampler.sample(array_bqm)

        w_binary = np.asarray(list(response_QPU.first.sample.values()))

        w_approx = precision.dot(w_binary)

        for index in range(len(w_approx)):
            print('Weight number {} is {}'.format(index, w_approx[index]))

        return w_approx.tolist()

    ds = load_dataset()
    preprocessed_ds = preprocess_dataset(ds)
    precision_matrix = get_precision()
    problem_matrix = get_problem_matrix(precision=precision_matrix,
     ↪  x=preprocessed_ds['x'], y=preprocessed_ds['y'])
    weights = anneal(q=problem_matrix, precision=precision_matrix)


quantum_annealing_linear_regression_dag = quantum_annealing_linear_regression()
```

Quantum linear regression results

Iris dataset

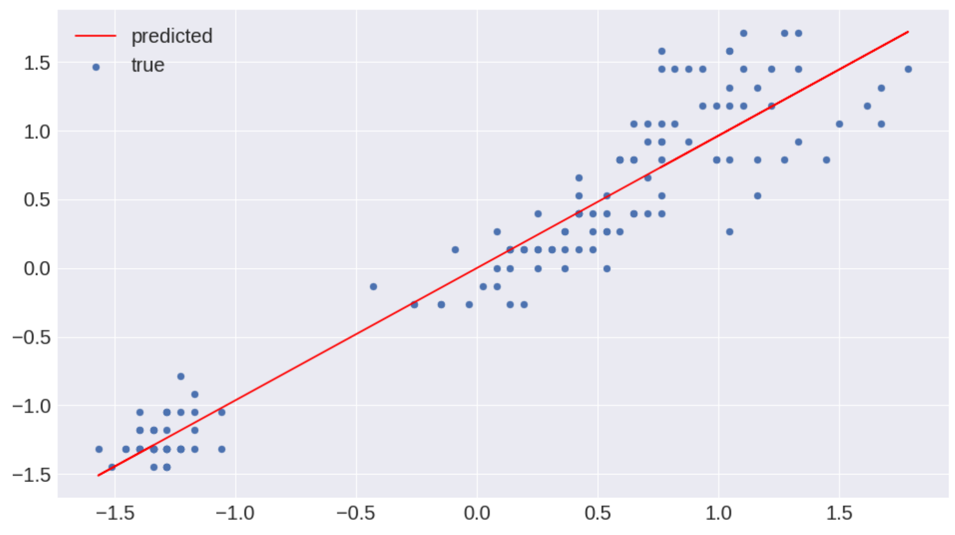


Figure 8.1: Closed form solution.

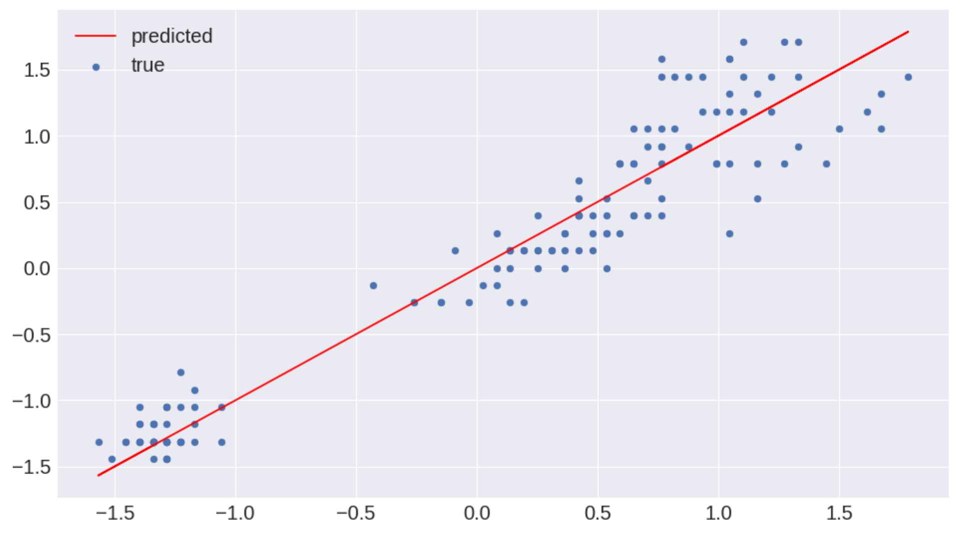


Figure 8.2: Quantum solution.

Boston dataset

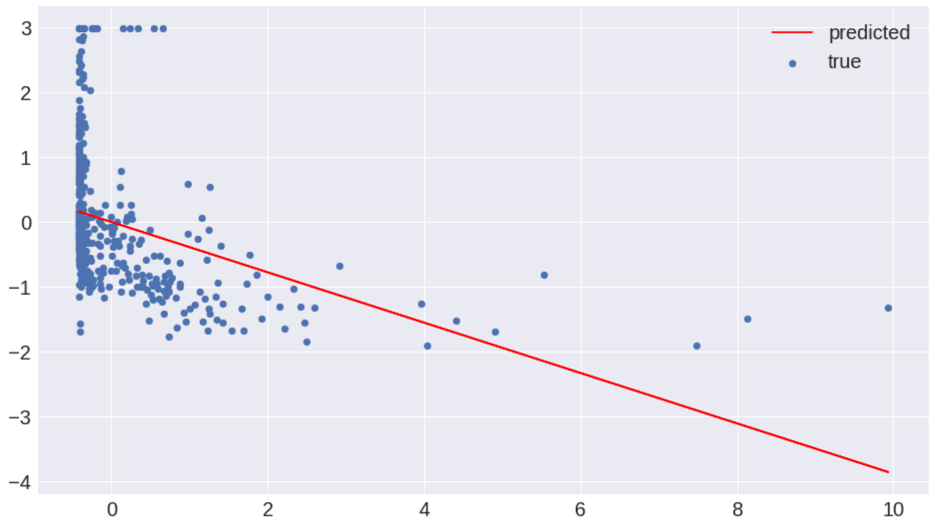


Figure 8.3: Closed form solution.

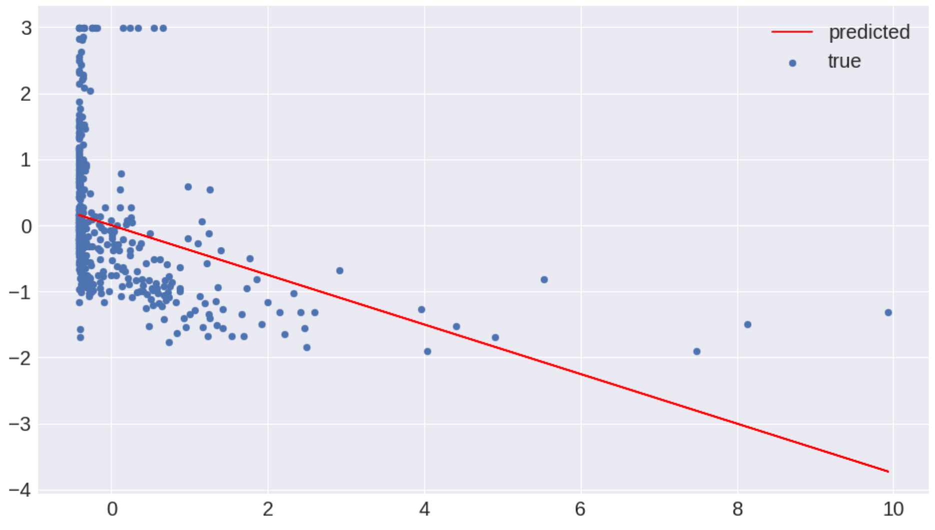


Figure 8.4: Quantum solution.

Diabetes dataset

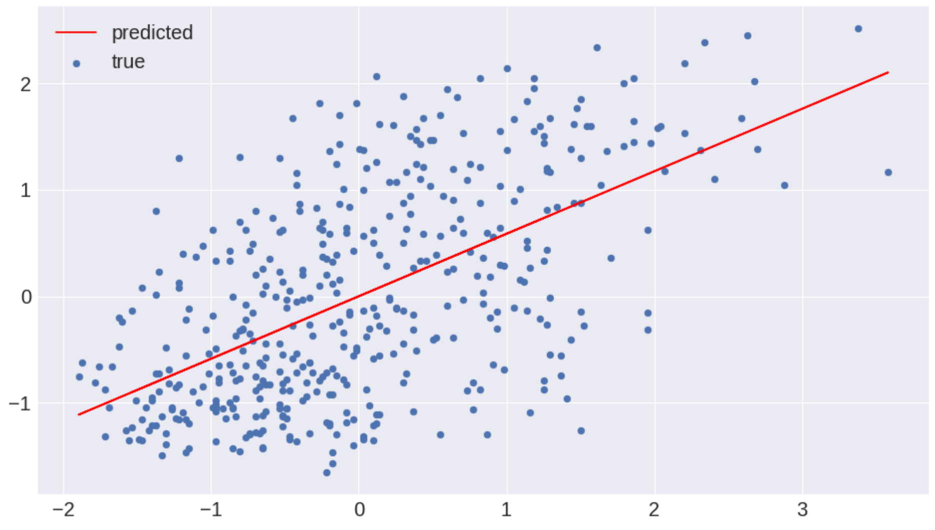


Figure 8.5: Closed form solution.

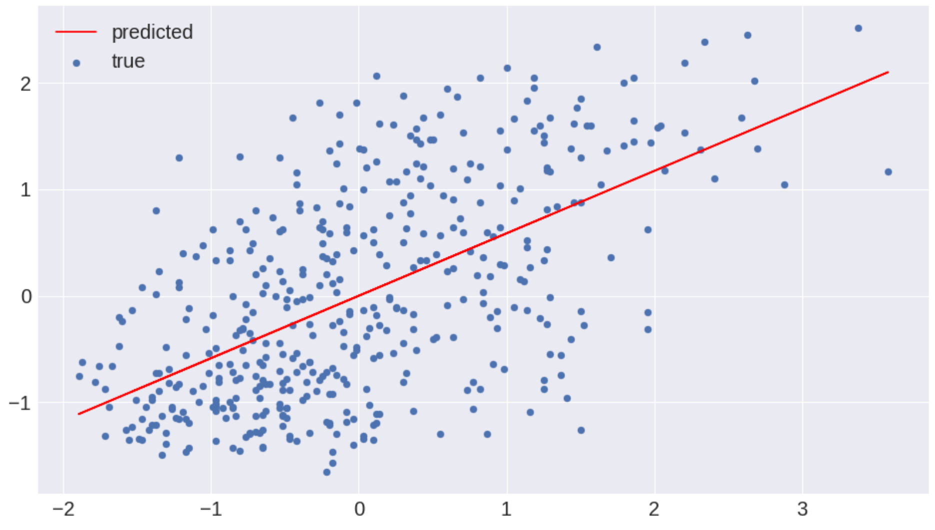


Figure 8.6: Quantum solution.