Quantum Reservoir Computing (QRC) Algorithms to Address Credit Risk Modeling

Brian Brueggert

October 21, 2025

Abstract

The focus of this paper is to describe the solution to a Project proposed by Bloq Quantum, namely; to build a AQSVM or Quantum Reservoir Computing (QRC) algorithm to address credit risk modeling, specifically using the Statlog German Credit Data dataset. Following an introduction, the basic principles of Quantum Reservoir Computing are discussed, and then the paper proceeds to cover the descriptions, algorithms and code implementations for each of the Classical Logistic Regression Model, introducing the Continuous and Binary Ising Models each with a linear, ridge and QUBO regression predictor version. It is intended to accompany the code for this Project, and contains a list of the prerequisites libraries in the Requirements section at the end.

1 Introduction

This work is the result of a Project proposed by Bloq Quantum to build a AQSVM or Quantum Reservoir Computing (QRC) algorithm to address credit risk modeling (a useful metric for estimating economic capital required for ...), specifically using the Statlog German Credit Data dataset. The Project encourages unconventional thinking, and seeks new and original ideas for implementing quantum reservoir systems or innovative algorithms, pushing the boundaries of what is possible.

The work here begins with a classical algorithm (Classical Logistic Regression Model.py), for a baseline comparison. Next, two variations of QRC algorithms are considered; named "Continuous Quadratic Ising Model" (CQIM) and "Binary Quadratic Ising Model" (BQIM). Each algorithm is explored for generating feature vectors, with two variants of each considered - both simple linear regression, as well as a more complex ridge regression, for making predictions. Finally, some Quadratic Unconstrained Binary Optimization (QUBO) algorithms are explored in conjunction with each of the feature vector extraction algorithms (CQIM and BQIM) to make predictions.

Some prerequisites to running the code in this Project include setting up a D-Wave Leap account, and leveraging the API token therein to activate the Cloud environment. In this case, this was configured via a terminal prompt inside of a Python 3.11.1 virtual environment, from where all of the code was run. The additional Python package/version requirements can be found in the requirements.txt document included in the Project, and are listed in the Requirements section at the end of this paper.

2 Basic Principles of Quantum Reservoir Computing

The basic principles of utilizing dissipation as a resource for QRC [QAISG [1]] involve, firstly, the association of physical points in a substrate x_k with states ρ_k in the reservoir.

Second, a spin state preparation (non-unitary transformation) must be performed on the substrate:

$$|\psi_{s_k}\rangle = \sqrt{1-s_k}|0\rangle + \sqrt{s_k}|1\rangle \xrightarrow{\text{(non-unitary)}} \rho' = |\psi_{s_k}\rangle\langle\psi_{s_k}| \otimes \text{Tr}^{(1)}\{\rho_{k-1}\} \tag{1}$$

Figure 1: N = 5 node clique spin network. (Image credit QAISG [1].)

Due to the disordered (spin) quantum ensemble dynamics of the reservoir interacting with the environment through dissipation, the states within the reservoir evolve unitarily as:

$$\begin{aligned}
\rho_k &= e^{-iH\Delta t}\rho' e^{iH\Delta t} \quad \text{(unitary)} \\
&= e^{-iH\Delta t}[|\psi_{s_k}\rangle\langle\psi_{s_k}| \otimes \text{Tr}^{(1)}\{\rho_{k-1}\}]e^{iH\Delta t}
\end{aligned} \tag{2}$$

Here the Hamiltonian provides the unitary evolution. The concept is that by putting the reservoir into a frustrated quantum superposition of states and allowing it to evolve unitarily via interaction with the environment through dissipation, we are essentially sampling low energy states representative of random feature vectors which capture quadratic relations or non-linear dynamics of the system. It's based on the idea that you can't contain neither heat nor energy, as it eventually dissipates into the environment. As such, it is noteworthy to mention that the information processing we seek should necessarily lie outside of the coherent quantum regime; past the boundary with the environment defined by extension of the Kibble-Zurick mechanism, with the information processing rate directly proportional to the rate of energy dissipation of the system.

3 Algorithm Overview

3.1 Classical Logistic Regression Model

3.1.1 Description

The data is briefly explored, processed, split and one hot encoded, as is standard practice. This model simply runs a classical logistic regression on the dataset, and is intended as a warm up to understanding the problem, as well as a baseline comparison.

3.1.2   Algorithm

Algorithm 1 Classical (Binary) Logistic Regression Model

Require: $n \geq 1$
 1: **for** $\forall n$ **do**
 2:      $d_i : i = 1...k$
 3:      **while** $i \leq k$ **do**
 4:          **for** each training data instance d_i **do** set the target value for the regression to

$$z_i = \frac{y_{i-P(1|d_j)}}{[P(1|d_j)(1 - P(1|d_j))]}$$

 5:              $d_j \leftarrow [P(1|d_j)(1 - P(1|d_j))]$
 6:              **for** $f(j)$ **do**

$$\text{class value} \leftarrow Z_j, \text{ weight } \leftarrow w_j$$

 7:              **end for**
 8:          **end for**
 9:      **end while**
10: **end for**
11: **return** classical label decision

3.1.3   Code Implementation

```
sudo python3 Classical_Logistic_Regression_Model.py
```

3.2   Continuous Quadratic Ising Model

3.2.1   Description

In this version of a QRC model, herein named the "Continuous Quadratic Ising Model", the logical structure of the problem is composed as follows: Consider a datum consisting of four attribute columns and a single class column (N = 5) as a training set schema for random feature vector extraction. In order to capture all of the quadratic terms (first-order non-linear dynamics), the logical structure of the problem assumes the topology of an N = 5 node clique graph, G (Fig.2).
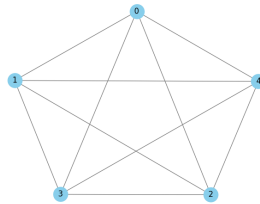

Figure 2: N = 5 clique graph structure.

Attribute columns that are continuous valued (non-binary) are then represented in this logical structure by splitting the representative node into k new nodes (four in this example), each retaining the clique structure with the original (non-replaced) graph nodes, and forming a bipartite group among themselves (Fig.3). The value of k is determined on a per training set basis, and distributed evenly among the available range. In practice, the implementation of a hybrid-quantum workflow would likely begin with data in one hot encoded format, as it was kept for use with classical systems. This means that we will begin by one hot un-encoding (decrypting) the one hot encoded data that we prepared in the Classical Logistic Regresison Model as a starting point, to recover categorical and integer (non-binary) attributes.
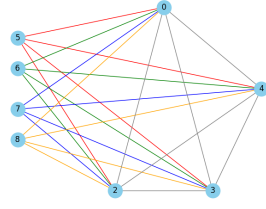
3

Figure 3: N = 5 clique with one node (Node 1) split into 4 bipartite nodes (Nodes 5, 6, 7, 8), each retaining clique connections with the rest of the original graph structure.

Logically, according to this model, a single datum row from the training set might be represented with its continuous valued attribute at node 5, and then the star clique shown in red in (Fig.4) would represent our problem.
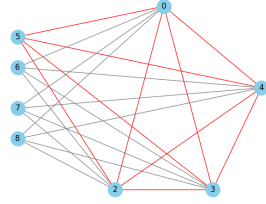

Figure 4: A single row of data might be represented as one of the replacement nodes forming the N = 5 clique with the original nodes.

A different datum row, with a different continuous value (node 6), might be represented as in (Fig.5), where the star clique shown in blue is our problem structure.
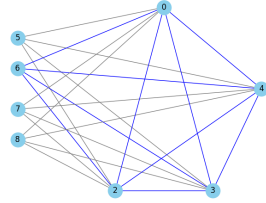

Figure 5: The structure of another row of datum with a different 'Node 1' value might be represented in this manner.

Note that any of the newly created 5-node clique structures is isomorphic to any other:

$$\forall G_j \in G : j \in [1, ..., k]; \qquad G_1 \cong G_2 \cong ... \cong G_k \tag{3}$$

with k equal to the number of nodes in the replacement set (four in this example); in this sense, indicative that they each represent the same topological structure. These mathematical results can be extended to values of N greater than five, but the current maximum inter-qubit connectivity available with the QPU's used here is 20-way connectivity. This, in turn, means that we can have a maximum of $N = 21$ nodes in our logical structure, or 20 attribute columns plus one class column in our training data schema (there are $N = 21$, $k_1 = X$ continuous variables with our specific dataset). This leads to N choose k (different $k_{\text{binary}} = 2$ this time) of:

$$\begin{aligned} C(N, k) &= \frac{N(N-1)}{k} \\ &= \frac{21(21-1)}{2} = 210 \text{ quadratic terms.} \end{aligned} \tag{4}$$

The D-Wave QPU's used in this Project are CC JJ niobium RF-SQUID's (Superconducting Quantum Interference Devices) that operate according to the transverse-field Ising model, to perform adiabatic quantum annealing.

Transverse-field Ising model:

$$H = h \sum_{i=1}^{N} \sigma_i^z + \sum_{i,j+1,i>j}^{N} J_{i,j} \sigma_i^x \sigma_j^x \qquad (5)$$

$$\mid h : \text{ transverse-field strength,}$$

$$J_{i,j} : \text{ inter-qubit coupling strength,}$$

$$\sigma_{i,j}^{x,z} : \text{ Pauli spin matrices (computational basis: } \sigma^x)$$

The dynamics of the Hamiltonian will generate the unitary evolution necessary in Eq.(2). The transverse-filed strength h is fed by the values in the training datum, thereby effectively setting the spin of a particular qubit to the datum column value (21 total), while the inter-qubit coupling strength $J_{i,j}$ is set to a constant value for all couplings. The assumption here is that every inter-attribute relationship among the datum (the data is in a state of quantum superposition) is of equal importance in determining a class label. As such, this becomes a hyper-parameter that will require tuning. (Not yet imlplemented in the code; will require a subscription for QPU time.)

A process of minor embedding is used to embed the logical structure of the problem graph G_j onto the specific QPU topology, which for this model is D-Wave's Advantage2 prototype2.6 (Zephyr QPU topology, 1,248 qubits) (Fig6). This embedding is reused for every row in the training data, consecutively, setting the h, $J_{i,j}$ values, and letting the Hamiltonian evolve unitarily ($t_{\text{anneal}} = 20 \ \mu\text{s}$), and then sampling the (low-energy) spin states of the system (10 samples each for testing mode code, and typically 1,024 samples each for non-test mode code.) The best of the samples is used as the resulting extracted random feature vector, and the next row is processed.
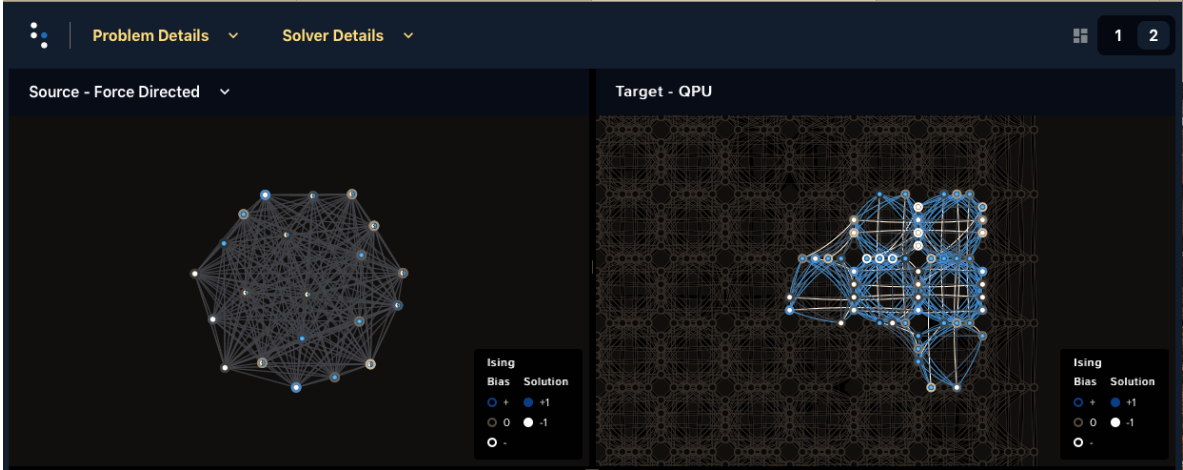

Figure 6: An $N = 21$ graph G_j example, brought up through a local host connection using D-Wave's Problem Inspector (post-processing), with the force directed logical graph topology displayed on the left, and the embedding on the target QPU, Advantage2 prototype2.6 (Zephyr QPU topology, 1,248 qubits) in this case, displayed on the right.

At the end of training, we have a set of random feature vectors, which we use to make predictions. First, in 'CQIM QRC LinearRegression test.py', we use a simple linear regressor as the predictor. Then, in 'CQIM QRC RidgeRegression test.py', we use a repeated 10-fold cross-validated, RidgeCV hyper-parameter tuned linear regressor as the predictor.

3.2.2 Algorithm

Algorithm 2 Continuous Quadratic Ising Model

Require: $1 \leq n \leq 21$ (current max.inter-qubit connectivity $= 20$)
Ensure: one hot encoded training data
 1: **for** one hot encoded training data **do**
 2:      decrypt one hot encoded data
 3:      **for** $\forall$ attribute $\in$ one hot encoded training data **do**
 4:          find attribute categories
 5:          **for** $\forall$ attribute categories **do**
 6:              find attribute subcategories
 7:              **for** $\forall$ attribute subcategories **do**
 8:                  find data type
 9:                  band gap $\leftarrow \frac{1}{(\text{num distinct values} \in \text{attribute})+2}$ $\triangleright$ +2: $\nexists$ val $\in [0,1]$
10:                  **if** data type is Categorical **then**
11:                      float64(data) $\leftarrow$ float64(data) $\times$ float64(band gap)
12:                  **else if** data type is Integer **then**
13:                      float64(data) $\leftarrow$ float64(data) $\times$ float64(band gap)
14:                  **else if** data type is Binary **then**
15:                      float64(data) $\leftarrow$ float64(data)
16:                  **end if**
17:              **end for**
18:          **end for**
19:          return new df col with combined data
20:      **end for**
21:      return decrypted one hot encoded training data
22: **end for**
23: $G_{CQIM} \leftarrow$ logical problem structure   (CQIM logic structure)
24: $qpu \leftarrow$ zephyr topology
25: $embedding \leftarrow$ minor embedding G_{CQIM} (map logical topology of G to qpu topology)
26: fixed embedding $\leftarrow embedding$
27: num samples $\leftarrow 1024$ (test mode: 10)
28: **if** test mode **then** $\triangleright \forall$ files ending in 'test'
29:      training data $\leftarrow$ first 2 rows of training data
30: **end if**
31: **for** $\forall$ rows $\in$ training data **do**
32:      $h \leftarrow$ spin: training data values
33:      $J_{i,j} \leftarrow$ transverse-filed: constant $\triangleright$ hard-coded until J hyper-parameter tuning implemented
34:      $\sigma_{i,j}^{x,z} \leftarrow$ fixed embedding
35:      spin state preparation:

$$|\psi_{s_k}\rangle = \sqrt{1-s_k}|0\rangle + \sqrt{s_k}|1\rangle \xrightarrow{\text{(non-unitary)}} \rho' = |\psi_{s_k}\rangle \langle\psi_{s_k}| \otimes \text{Tr}^{(1)} \{\rho_{k-1}\}$$

Algorithm 3 Continuous Quadratic Ising Model (continued)

36:     **for** $\forall$ samples $\in$ num samples **do**
37:         quantum annealing:

$$H = h \sum_{i=1}^{N} \sigma_i^z + \sum_{i,j+1,i>j}^{N} J_{i,j} \sigma_i^x \sigma_j^x$$

$$\rho_k = e^{-iH\Delta t} \rho' e^{iH\Delta t} \quad \text{(unitary)}$$

$$= e^{-iH\Delta t} [|\psi_{s_k}\rangle \langle \psi_{s_k}| \otimes \text{Tr}^{(1)} \{\rho_{k-1}\}] e^{iH\Delta t}$$

38:         sample $\leftarrow$ reservoir system spin measurement
39:     **end for**
40: **end for**
41: random feature vector $\leftarrow$ lowest energy sample
42: random feature vector list $\leftarrow$ random feature vector
43: X $\leftarrow$ attr cols $\in$ random feature vector list
44: y array $\leftarrow$ class col $\in$ random feature vector list
45: y $\leftarrow$ y array binary conversion
46: **for** X, y **do**
47:     **if** linear regression **then**
48:         prediction regressor $\leftarrow$ linear regression
49:     **else if** ridge regression **then**
50:         prediction regressor $\leftarrow$ ridge regression
51:     **end if**
52: **end for**
53: **for** $J \in$ J range **do**
54:     hyper-parameter tuning   ▷ not yet implemented
55: **end for**

3.2.3   Code Implementation

```
sudo python3 CQIM_QRC_LinearRegression_test.py
```

```
sudo python3 CQIM_QRC_RidgeRegression_test.py
```

```
sudo python3 CQIM_QRC_LinearRegression.py
```

```
sudo python3 CQIM_QRC_RidgeRegression.py
```

3.3   Binary Quadratic Ising Model

3.3.1   Description

This QRC model, introduced as the "Binary Quadratic Ising Model", begins with the one hot encoded training data, and applies a different problem logic. The entire topology can't be mapped in the same manner as in the CQIM, since there are too many columns present ($N = 1076 > 21$), and the current maximum inter-qubit connectivity is limited at 20. Here we will instead split the original one hot encoded training data into all permutation combinations of pairs of attribute columns,

$$
\begin{aligned}
C_k(N) &= \frac{N!}{(N-k)!k!} \\
&= \frac{(1076-1)!}{((1076-1)-2)!2!} = \frac{1075!}{1073! * 2} = \frac{1075 * 1074}{2} = 5.77275 \text{ x} 10^5 \text{ quadratic terms.}
\end{aligned}
\tag{6}
$$

and add the class column to each resulting pair in the combination (Fig.7). The model can be modified to run $m < \left\lfloor \frac{1248 \text{ qubits}}{3} \right\rfloor$ batches of triangle graphs G_j at a time.
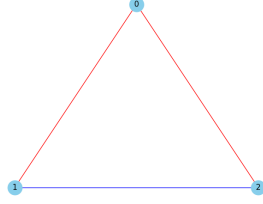

Figure 7: N = 3 clique graph structure $G_{j,m=1}$, where qubits 1 and 2 are every quadratic (non-self) combination of attribute column pairs and qubit 0 is the class column.

Each of the pairwise permutation combinations recombined with the class column is run through the process of setting the transverse-filed strength h to the values in the training datum, using the minor embedding process to embed the logical graph G_j onto the target QPU (again, D-Wave's Advantage2 prototype2.6 (Zephyr QPU topology, 1,248 qubits)), while the inter-qubit coupling strength $J_{i,j}$ is set to a constant value for all couplings. Again, the assumption being that every inter-attribute relationship among the datum is of equal importance in determining a class label, in addition to each inter-attribute relation being of equal importance. Again, this becomes a hyper-parameter that will require tuning, possibly on the triangle graph G_j level (i.e. varied $\forall$ permutation combinations). (Not yet imlplemented in the code; will require a subscription for QPU time.)

The embedding is reused for every attribute pair in every row in the training data, consecutively, setting the h, $J_{i,j}$ values, and letting the Hamiltonian evolve unitarily ($t_{\text{anneal}} = 20 \ \mu$s), and then sampling the (low-energy) spin states of the system (10 samples each for testing mode code, and typically 1,024 samples each for non-test mode code.) The best of the each of the samples are combined into a random feature vector, and the next row is processed.

Figure 8: An $N = 3$ graph G_j example, brought up through a local host connection using D-Wave's Problem Inspector (post-processing), with the force directed logical graph topology displayed on the left, and the embedding on the target QPU, Advantage2 prototype2.6 (Zephyr QPU topology, 1,248 qubits) in this case, displayed on the right.

3.3.2 Algorithm

Algorithm 4 Binary Quadratic Ising Model

Require: Enough QPU time to run
Ensure: one hot encoded training data
1: **if** test mode **then**   ▷ ∀ files ending in 'test'
2:      training data ← first 2 rows of training data
3: **end if**
4: find pairwise permutation combinations ∀ training data attribute cols
5: **for** each pairwise attribute permutation combination **do**
6:      datum ← pairwise attribute permutation combination + class col
7:      G_{BQIM} ← logical problem structure   (BQIM logic structure)
8:      $G_{\text{BQIM,m}} \leftarrow \{G_{j,1}, ..., G_{j,m}\}$ | $m \in \mathbb{N}^+, \{1, ..., \lfloor \frac{1248 \text{ qubits}}{3} \rfloor\}$ ▷ not yet implemented
9:      qpu ← zephyr topology
10:      $embedding$ ← minor embedding G_{BQIM} (map logical topology of G to qpu topology)
11:      fixed embedding ← $embedding$
12:      num samples ← 1024   (test mode: 10)
13:      **for** ∀ datum ∈ $G_{BQIM}/G_{\text{BQIM,m}}$ **do**
14:          h ← spin: training data values
15:          $J_{i,j}$ ← transverse-filed: constant ▷ hard-coded until J hyper-parameter tuning implemented
16:          $\sigma_{i,j}^{x,z}$ ← fixed embedding
17:          spin state preparation:

$$|\psi_{s_k}\rangle = \sqrt{1 - s_k}|0\rangle + \sqrt{s_k}|1\rangle \xrightarrow{\text{(non-unitary)}} \rho' = |\psi_{s_k}\rangle \langle\psi_{s_k}| \otimes \text{Tr}^{(1)}\{\rho_{k-1}\}$$

18:          **for** ∀ samples ∈ num samples **do**
19:              quantum annealing:

$$H = h \sum_{i=1}^{N} \sigma_i^z + \sum_{i,j+1,i>j}^{N} J_{i,j} \sigma_i^x \sigma_j^x$$

$$\rho_k = e^{-iH\Delta t} \rho' e^{iH\Delta t} \quad \text{(unitary)}$$
$$= e^{-iH\Delta t}[|\psi_{s_k}\rangle \langle\psi_{s_k}| \otimes \text{Tr}^{(1)}\{\rho_{k-1}\}]e^{iH\Delta t}$$

20:              sample ← reservoir system spin measurement
21:          **end for**
22:          sample ← separate $G_{\text{BQIM,m}}$ samples   ▷ not yet implemented
23:      **end for**
24:      random feature vector piece ← lowest energy sample
25:      random feature vector piece list ← random feature vector piece
26: **end for**

Algorithm 5 Binary Quadratic Ising Model (continued)
___
27: random feature vector ← random feature vector piece list
28: random feature vector list ← random feature vector
29: X ← attr cols ∈ random feature vector list
30: y array ← class col ∈ random feature vector list
31: y ← y array binary conversion
32: **for** X, y **do**
33:     **if** linear regression **then**
34:         prediction regressor ← linear regression
35:     **else if** ridge regression **then**
36:         prediction regressor ← ridge regression
37:     **end if**
38: **end for**
39: **for** $J \in$ J range **do**
40:     hyper-parameter tuning   ▷ not yet implemented
41: **end for**
___

3.3.3   Code Implementation

___
```
sudo python3 BQIM_QRC_LinearRegression_test.py
```
___

___
```
sudo python3 BQIM_QRC_RidgeRegression_test.py
```
___

* Beware: The following two programs will result in $5.77275 \text{ x}10^5 * 1024 \approx 5 \text{ x}10^8$ quantum anneal execution requests per row of data.

___
```
sudo python3 BQIM_QRC_LinearRegression.py
```
___

___
```
sudo python3 BQIM_QRC_RidgeRegression.py
```
___

3.4   Quadratic Unconstrained Binary Optimization (QUBO) Models

3.4.1   Description

In this section, we use adiabatic quantum linear regression as the model algorithm for our final analysis (classification/prediction) of the random feature vectors previously extracted via QRC, similar to how classical linear regression and ridge regression have been used as regressor options in this paper. The standard one hot encoded data is analyzed, along with a set of data augmented by a version of a SMOTE algorithm, as well as several sets of synthetically generated data. Here we utilize a quantum simulator by means of D-Wave's SimulatedAnnealingSampler (Metropolis-Hastings algorithm (Markov-chain Monte Carlo (MCMC))), due to limited QPU access. We compare the results to those obtained by using Scikit-Learn's LinearRegression with n-fold cross-validated hyper-parameter tuning, and then integrate the model for use into both the CQIM and BQIM QRC models.

In the blob data we examine pairwise linearly separable data with respect to class, meaning that the classes can be linearly separated from each other.

Classically, in the ordinary least squares (OLS) method of linear regression, we seek to minimize the sum of the squared errors (SSE), or differences between predicted $\hat{y}_i$ and actual y_i values:

$$min \ \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}_i - y_i \right)^2 \ . \tag{7}$$

We can write the Euclidean error function in matrix form as:

$$min_{w \in \mathbb{R}^{d+1}} \ E(w) = \| Xw - Y \|^2 \ , \tag{8}$$

where $X \in \mathbb{R}^{\mathbb{N} \times (d+1)}$ is the augmented training data set, $Y \in \mathbb{R}^{\mathbb{N}}$ are the (dependent variable) linear regression values/labels, and $w \in \mathbb{R}^{(d+1)}$ are the learned regression weights, following "Adiabatic Quantum Linear Regression", Date, P., Potok, T. [2]. Note that

$$Xw = \begin{bmatrix} \longleftarrow & x^1 & \longrightarrow \\ \longleftarrow & x^2 & \longrightarrow \\ \vdots & & \\ \longleftarrow & x^n_{n \times (d+1)} & \longrightarrow \end{bmatrix} \begin{bmatrix} \uparrow \\ w \\ \downarrow \end{bmatrix} = \begin{bmatrix} x^1 w \\ x^2 w \\ \vdots \\ x^n w \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^d x'_j w_j \\ \sum_{j=0}^d x^2_j w_j \\ \vdots \\ \sum_{j=0}^d x^n_j w_j \end{bmatrix} = \begin{bmatrix} \hat{y}^1 \\ \hat{y}^2 \\ \vdots \\ \hat{y}^n \end{bmatrix} = \hat{y} \ .$$

The minimization can be achieved via iterative approaches, such as gradient descent, or in this case via an analytical, or closed-form solution:

$$w = (X^T X)^{-1} X^T Y \ . \tag{9}$$

Rewriting Eq.(8) as:

$$min_{w \in \mathbb{R}^{d+1}} \ E(w) = w^T X^T X w - 2 w^T X^T Y + Y^T Y \ , \tag{10}$$

and, dropping the constant scalar term $Y^T Y$, which doesn't affect optimization, we have:

$$min_{w \in \mathbb{R}^{d+1}} \ E(w) = w^T X^T X w - 2 w^T X^T Y \ . \tag{11}$$

A precision vector is defined $P = [p_1, p_2, ..., p_K]^T$, $K \in \mathbb{N}$, as well as K binary variables $\hat{w}_{ik}$ ($w_i \in (d+1)$):

$$w_i = \sum_{k=1}^K p_k \hat{w}_{ik} \ \longrightarrow \ w = \mathscr{P} \hat{w} \tag{12}$$

where $\mathscr{P} = \mathbb{I}_{(d+1)} \otimes P^T$, and $\hat{w} = [\hat{w}_{11}, \hat{w}_{12}, ..., \hat{w}_{1K}, \hat{w}_{21}, \hat{w}_{22}, ..., \hat{w}_{2K}, ..., \hat{w}_{(d+1)k}, \hat{w}_{(d+1)2}, \hat{w}_{(d+1)K}] \in (d+1)K$. Substituting for w in Eq.(11) yields:

$$min_{\hat{w} \in \mathbb{R}^{(d+1)K}} \ E(\hat{w}) = \hat{w}^T \mathscr{P}^T X^T X \hat{w} - 2 \hat{w}^T \mathscr{P}^T X^T Y \ . \tag{13}$$

Next, we represent this objective function in the form of a Quadratic Unconstrained Binary Optimization (QUBO) problem, which assumes the general form:

$$min_{x \in [0,1]^n} \ x^T Q x + x^T B \ , \tag{14}$$

with $x = \hat{w}$, $Q = \mathscr{P}^T X^T X \mathscr{P}$ and $B = -2 \mathscr{P}^T X^T Y$, as in Eq.(13), using the minimizing function:

$$f(x) = \sum_i Q_i x_i + \sum_{i<j} Q_{i,j} x_i x_j \ . \tag{15}$$

Hyper-parameter tuning is implemented for both the permutation QUBO λ and adaptive precision vector parameters. The permutation QUBO is:

$$min_{x \in [0,1]^n} \ x^T Q x + \lambda x^T B \ . \tag{16}$$

For the adaptive precision vector tuning, we follow a procedure described in "Adaptive Learning for Quantum Linear Regression", C. Constantino, et al. [3], where we approximate each w_i by a tailored precision vector $\pi_i \in \mathbb{R}^K$, such that Eq.(12) becomes:

$$w_i = \sum_{k=1}^K \pi_{ik} \hat{w}_{ik} \ \longrightarrow \ w = \mathscr{P} \hat{w} \ , \tag{17}$$

where π_{ik} is the kth value of the precision vector for weight w_i, $\mathscr{P} = \sum_i \mathscr{I}^i \otimes \pi_i$, and π_i is the precision vector for the regression coefficient w_i.

The hyper-parameters are tuned for values of $\lambda \in [-2, 2]$ in steps of 0.2, with $K = 21$-bit precision vector, and rate $= 2$, rate-desc $= 2$ and rate-asc $= 4$, with cross-validation set at 5-fold for the adaptive

precision vector (due to 8GB RAM limitation). All augmented and synthetic data was also limited to 500 rows, for the same reason. The classical Scikit-Learn RidgeCV model was also set to 5-fold cross-validation, for comparison purposes, although with the benefit of 64-bit precision.

- rows of data, distributions, etc.

3.4.2   Algorithm

3.4.3   Code Implementation

```
sudo python3 CQIM_QRC_QUBO_1_test.py
```

---

4   Index

List of Algorithms

5   Requirements

The list below is in addition to a D-Wave Leap account with Cloud access.

Python 3.11.1, as well as the list below; information which is also contained in requirements.txt file included in Project.

```
Package Version ——————- ———
annotated-types 0.7.0
Authlib 1.3.2
blinker 1.9.0
certifi 2024.12.14
cffi 1.17.1
charset-normalizer 3.4.0
click 8.1.7
contourpy 1.3.1
cryptography 44.0.0
cycler 0.12.1
Deprecated 1.2.15
dimod 0.12.14
diskcache 5.6.3
dwave-cloud-client 0.11.3
dwave-greedy 0.3.0
dwave-hybrid 0.6.11
dwave-inspector 0.4.4
dwave-inspectorapp 0.3.1
dwave-neal 0.6.0
dwave-networkx 0.8.14
dwave-ocean-sdk 6.9.0
dwave-preprocessing 0.6.5
dwave-samplers 1.2.0
dwave-system 1.23.0
dwave-tabu 0.5.0
dwavebinarycsp 0.3.0
fasteners 0.19
Flask 3.1.0
fonttools 4.55.3
homebase 1.0.1
idna 3.10
imageio 2.36.1
importlib metadata 8.5.0
itsdangerous 2.2.0
Jinja2 3.1.4
joblib 1.4.2
kiwisolver 1.4.7
lazy loader 0.4
MarkupSafe 3.0.2
matplotlib 3.10.0
minorminer 0.2.13
networkx 3.4.2
numpy 1.26.4
packaging 24.2
```

6   Bibliography

[1] "Dissipation as a resource for quantum reservoir computing" YouTube, uploaded by QAISG, May 7, 2024. https://youtu.be/hxvTRJDj6ok?si=qD6N98Q8xB3UdU90

[2] Date, P., Potok, T. Adiabatic quantum linear regression. Sci Rep 11, 21905 (2021). https://doi.org/10.1038/s41598-021-01445-6

[3] Costantino, C., Dacrema, M F., Cremonesi, P. Adaptive Learning for Quantum Linear Regression. VTT Technical Research Center, Finland, Politecnico di Milano, Italy, arXiv, August 5, 2024 2408.02833v1 [quant-ph].