Quantum Support Vector Machines

Introduction to the programming of Quantum Computers project

Enrico Buratto

University of Helsinki

Faculty of Science

Academic Year 2021-2022

Contents

1 Introduction **2**
1.1 Conventional SVMs 2
1.1.1 Least Squares SVMs (LS-SVM) 3

2 Algorithm description **3**
2.1 Summary 3
2.2 Detailed description 4
2.2.1 Deal with real data 5
2.2.2 Solving systems of linear equations 6
2.2.3 Back on QSVMs 7

3 Conclusion **8**
3.1 Requirements 8
3.2 Future research 8
3.3 Further reading 9

1 Introduction

This short document describes the operation of Quantum Support Vector Machines (QSVMs), which is the quantum equivalent for a famous machine learning algorithm called Support Vector Machines (SVMs). As for the traditional version, this algorithm can be used for supervised learning tasks, the most important being classification.

Before starting to explain how QSVMs works, an introduction on standard SVMs must be done. Even if the quantum version of the algorithm deals with quantum-related issues, such as how to represent data inside a quantum computer, it finds its foundations in the theory developed for conventional SVMs.

Note, however, that the next subsection is not meant to be a complete description, but rather just some essential theory to better explain how QSVMs work; for the same reason, it can be regarded as not part of the essay itself, but just a facilitation for the reader.

1.1 Conventional SVMs

As already mentioned, SVMs are a class of machine learning model algorithms; as we will see, they can be considered either a linear classifier as well as a non-linear classifier with a kernel function. In short, the main idea behind them is to find hyperplanes in the feature space such that they separate different output classes, *i.e.* different space regions. After having defined these hyperplanes, the classification of a new sample is an easy task: we just infer the class based on the region of the space it lies.

In order to learn the hyperplane, the process considers two parallel hyperplanes separating the classes, and then tries to maximize the distance between them. The result is then a third hyperplane described by a vector $\mathbf{w}$ and a bias term b; the classification of a new sample $\mathbf{x}$ can then be easily done computing $sign(\mathbf{w}\mathbf{x} - b)$.

An important property of SVM, which will come in handy for the description of QSVMs, is the **duality**: every SVM problem can be formulated as another optimization problem, *i.e.* its dual form. Instead of finding the optimal hyperplane, then, we can try to solve the following maximization problem:

$$\begin{aligned}\max_{\alpha} \sum_i \alpha_i y_i - \frac{1}{2}\sum_i \sum_j \alpha_i \mathbf{x}_i \cdot \mathbf{x}_j \alpha_j \\ s.t. \quad \sum_i \alpha_i = 0 \\ y_i \alpha_i \geqslant 0 \ \forall i\end{aligned}$$

where:

- $\vec{\alpha} = [\alpha_1, ..., \alpha m]$ are the parameters;
- $\mathbf{x}_i$ is the i^{th} training data point in the feature space;
- y_i is the class of the i^{th} point.

With this dual formulation, the classification of a new sample can be done computing

$$sign(\sum_i \alpha_i \mathbf{x}_i \cdot \mathbf{x} - b)$$

A final definition to give here is the one of the *kernel matrix*; this is defined as the set of inner products $K_{ij} = \mathbf{x}_i \cdot \mathbf{x}_j$, and can be extended in order to learn the parameters in a non-linear separably data scenario; this, however, is only a generalization of SVM and it's not crucial for our purposes.

1.1.1 Least Squares SVMs (LS-SVM)

A final theoretical result we need to keep in mind before starting describing how QSVMs work is Least Squares SVMs. Since solving the maximization problem reported above with a quantum computer can be tricky, we can instead solve this reformulation, which consists in solving a linear system of equations:

$$\begin{bmatrix} b \\ \vec{\alpha} \end{bmatrix} = \mathbf{F}^{-1} \begin{bmatrix} 0 \\ \mathbf{y} \end{bmatrix}$$

with

$$\mathbf{F} = \begin{bmatrix} 0 & 1^\top \\ \mathbf{1} & K + \frac{1}{\gamma}\mathbb{1} \end{bmatrix}$$

where:

- $\mathbf{y}$ is a vector containing the data classes for training data;
- γ is a hyperparameter determined by cross validation;
- $\mathbf{1}$ is a vector of ones;
- $\mathbb{1}$ is the identity matrix.

2 Algorithm description

2.1 Summary

As for traditional SVMs, the algorithm can be divided into two main parts:

- A **training** phase, where known data samples and their classes are used to learn the parameters for the model;
- A **classification** phase, where the parameters are used to infer the classes from new data samples.

Unlike the conventional algorithm, however, we have now to deal with the quantum nature of quantum computers. This means that we have now some new aspects to take into account; these are:

- How to store and access data samples: unlike binary computers, we now need to store data to be accessible in superposition. As we will see in a more detailed fashion later, this means that we must use QRAM as a physical component, quantum simulation to represent Hermitian matrices and amplitude encoding to represent vectors;
- How to store the parameters we learnt for subsequent classification.

The main idea of the algorithm is then to use the known data samples and target classes, stored in QRAM, to solve a linear problem with the HHL algorithm. This linear problem is exactly the LS-SVM described in the previous section, and it will give us the parameters as a result. We then can use these parameters (alongside with new data samples) for the classification task; the details for both these phases are provided in the next sections.

2.2 Detailed description

In order to better analyze the functioning of the **training phase** of QSVM, a scheme of it is here reported; the source of this is Johnston et. al book, which is also the most consistent reference for the other details.[1]

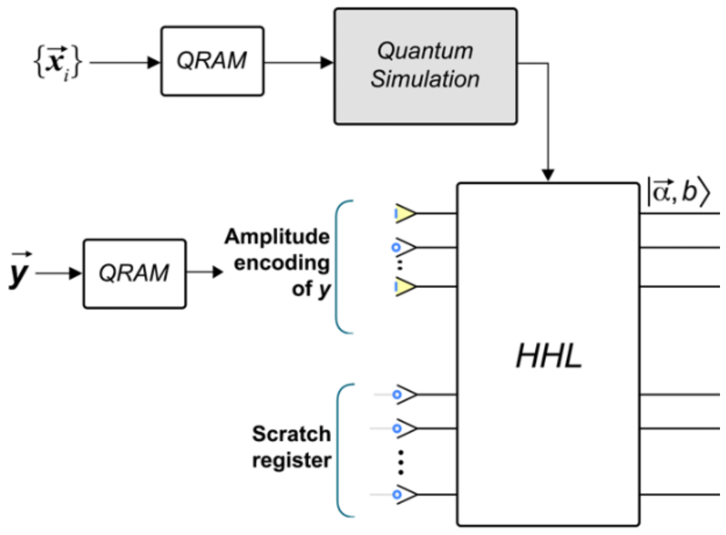

Figure 1: Training phase scheme for QSVM.

This part of the algorithm, then, works as follows:

1. The data samples $\mathbf{x}_i$, with their features, are loaded in superposition into QRAM; since these are Hermitian matrices[1], we can use quantum simulation to efficiently represent them as QPU operations;
2. The target variable is loaded in superposition into QRAM, and is then encoded with amplitude encoding;
3. The HHL algorithm is then used in order to solve the linear system of LS-SVM reported above, using the quantum simulation representation of data samples, the amplitude encoding of the target variable and some scratch qubits. The output of this algorithm are the two components of the model: the vector α containing the parameters and the bias term b.

Until now we described how the algorithm works, but we kind of ignored the *elephants* in the room:

- What does it mean *amplitude encoding*?
- What does it mean *quantum simulation*?
- Why is it important that $\mathbf{x}_i$ are Hermitian matrices?
- What *actually* is QRAM?
- What is the HHL algorithm?

In the next two subsections these concepts are explained with regards to this specific problem: the first fourth questions are answered in the first paragraph, while the last one is answered

[1]A matrix is Hermitian if it is equal to its own conjugate transpose, that is, given a matrix A, A is Hermitian iff $A = \bar{A}^{\top}$.

in the second paragraph. This will lead to a better understanding of the algorithm itself, as the entirety of it is based on these concepts. Note that what follows is just a brief explanation needed to understand how QSVMs work, as a more detailed exposition is out of the scope of this document.

2.2.1 Deal with real data

First of all, a small introduction on **real data** should be made. Until now, in fact, we only worked with integer values, which as we know can easily be represented with a binary encoding. It is not plausible, however, that the data for a machine learning task is composed by integer numbers; with a high probability, we would have real numbers instead. We then need to find a way to represent this data in a *quantum way*. The easiest abstraction we can find for this is fixed-point notation: a register is split into two sections, one for the integer part and one for the fractional part. In the former, integers are expressed as usual as powers of 2, while in the latter fractional numbers are expressed as powers of $\frac{1}{2}$; this works, but as we will see later in this section there is a more convenient way to represent them for the task we are trying to solve.

Now we might ask ourselves what is the best way to store the data needed for the task: maybe we could use a traditional RAM, read values from them and writing them into registers. This approach, however, has a limit: how do we store data in superposition? The answer to this is **QRAM**, which is also needed for QSVMs (as the reader may notice in the scheme above). This structure works, in brief, like standard RAM: it takes two qubit registers as input, the first being an address QPU register (which specifies a memory address) and an output register (which returns the value stored in the specified address); the fact that these registers are qubit registers means that we can specify a superposition of locations in the address, and this leads to the capability of receiving a superposition of the values in the output register.

After having described how to represent and store real values, it's now time to define convenient ways to represent data for the QSVMs tasks. As already explained above in this document, we need to represent the target variable using amplitude encodings, and the data samples using quantum simulation: let's then start with the former.

As we already said, the target variable is a vector of size n, with n being the number of samples in our dataset: we then need a convenient way to represent a vector inside a quantum computer. A first approach, as also described by Johnston et. al, is to use state encodings: for each element of a vector we use a QPU register and we binary encode the value. This is, however, a really inefficient way to represent this kind of data, since it can be really heavy on qubits; for instance, let's say that we have a 4-elements vector with the biggest one being 10: we then need 4 qubit registers (one for each element), each of which with four qubits (the binary representation of 10 is in fact 1010).
A better idea (and, actually, what we use for QSVMs) is then to use **amplitude encodings**: instead of having huge qubit registers, we can just encode the values with amplitudes. A QPU register with n qubits, in fact, can exist in a superposition with 2^n amplitudes, therefore we can encode a vector of size n in a QPU register of $ceil(log(n))$ qubits. An example of this encoding follows; note that this encoding comes in handy also for real values representation: we can, in fact, represent fractionary data with amplitudes (after careful normalization of all the values), and indeed this is what we do with target data for the QSVM algorithm. Moreover, we can also represent negative values using a 180° phase shift.

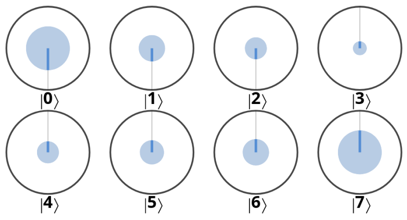

Figure 2: Amplitude encoding for $(-10, -6, -5, 3.14, 5, 5.5, 6, 10)$.

Finally, let's briefly see **quantum simulation**. As already said, the data samples $\mathbf{x}_i$ are a matrix; therefore, we need a way to represent them in order for them to be convenient for our linear system solving task. The first idea we could have is to represent a m x n matrix as m qubit registers, each of which represents a row of the matrix using amplitude encoding. However, a smarter idea can be implemented: we can think of matrices not as sets of vectors, but as QPU operations; in fact, a QPU operation can be entirely described by its eigenstates and eigenphases, and the eigendecomposition of a matrix characterizes it completely. The eigenstates of a QPU operation are the amplitude encoding of the eigenvectors of the corresponding matrix, and its eigenphases are related to the matrix's eigenvalues. Therefore, even if we cannot state (yet) that the operation corresponds to the matrix, we can at least intuitively say that the former acts the same way of the matrix on another amplitude encoded vector, which is what we need for our QSVMs.
Quantum simulation is, then, a procedure to represent a matrix with QPU operations in an efficient way. Even though this is beyond the scope of discussion, the main idea is that (unfortunately) not all the matrices correspond to a buildable set of QPU operations; valid QPU operations are described by unitary matrices. Fortunately, we can use Hermitian matrices to construct a unitary matrix; therefore, the set of representable matrices extends also to Hermitian ones. As we can prove, $\mathbf{F}$ is Hermitian, and therefore representable.

2.2.2 Solving systems of linear equations

The second obscure part of the QSVM's training phase was the HHL algorithm; even though this topic should take an entire document, I will here briefly explain the basic idea in order to understand how QSVMs algorithm solves the linear system (and therefore finds the optimal parameters).

The main idea behind solving a system of linear equations is to take advantage of the system itself: a linear system can be written as

$$A\mathbf{x} = \mathbf{b}$$

where:

- A is the coefficient matrix;
- $\mathbf{x}$ is the vector of unknowns, *i.e.* the quantities we have to find;
- $\mathbf{b}$ is the vector of constants of the equations.

The solutions of this system can then be computed using the matrix inversion properties:

$$\mathbf{x} = A^{-1}\mathbf{b}$$

Therefore, if we are able to find the inversion of matrix A we can compute the solutions just by performing a matrix multiplication between A^{-1} and $\mathbf{b}$; this is exactly what the HHL algorithm does. The general scheme for it is the following (image retrieved from Johnston et. al):

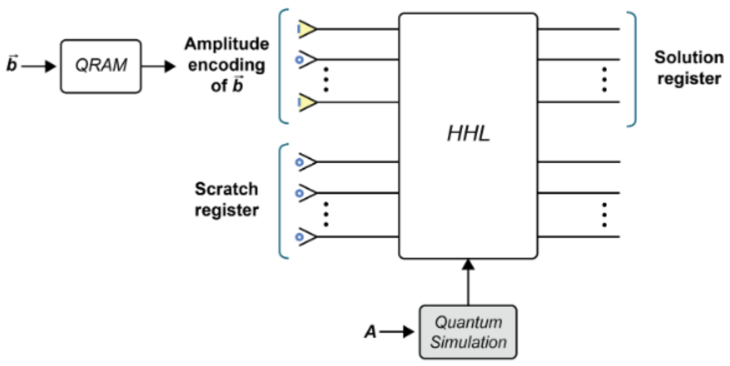

Figure 3: Scheme for the HHL algorithm.

Now, the reader may notice that this scheme looks very similar to the QSVM scheme reported above. This is not a coincidence: in fact, what the QSVMs algorithm does is exactly the same, as already argued previously in this document. We will stop here with this, because as already mentioned a detailed description of this algorithm needs a research on its own; the only other thing left to point out is that the solutions of the system are returned in the solution register. This returned vector is amplitude-encoded, and we cannot access the solutions since they are hidden in the amplitude; fortunately, for our case we don't need to read them, as they are just the parameters that will subsequently used for the classification task of the QSVM algorithm. However, this is better described in the next section.

2.2.3 Back on QSVMs

Now that we described the general training phase of the algorithm and its components, the operation of it should be clear: we just need to represent the data in order for it to be usable by the HHL algorithm, and we solve the linear system using the HHL algorithm, which is equivalent to find the inverse of the $\mathbf{F}$ matrix and then act, *i.e.* multiply, it with the $\begin{bmatrix}0 & \mathbf{y}\end{bmatrix}$ vector.

The results are then the parameters vector and the bias term, stored in the output registers of HHL, and the scratch qubits, which are reset to $|0\rangle$ (because of the definition of the algorithm itself). These parameters are then used for the classification task, which is now reported; what follows is the scheme for it (retrieved from Johnston et. al as well).

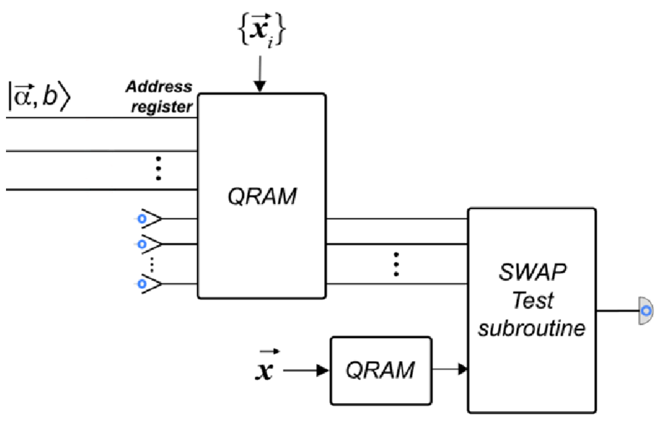

Figure 4: Classification phase scheme for QSVM.

This last part of the algorithm, then, works as follows:

1. The parameters are loaded into QRAM; better, they are already in QRAM because of the quantum behavior of HHL algorithm;
2. The data samples from the training phase are loaded into QRAM as well; the algorithm needs these in order to perform the classification, as explained below;
3. A new data point $\mathbf{x}$, on which we need to perform the classification, is loaded into QRAM;
4. The classification is then performed using the *sign* formula reported in §1. In order to do this, the inner product between $\mathbf{x}$ and the previous data samples is computed using the parameters in α and the bias term b (computed during the training phase) as weights;
5. We now have a superposition of the training data with amplitudes containing the α_i. The algorithm then executes a SWAP test subroutine, which combines the states into an entangled superposition and performs a swap test. This swap test is in turn build so that the probability of reading a 1 corresponds to the sign: when the probability is less than $\frac{1}{2}$ the sign is positive, and when it is greater than $\frac{1}{2}$ the sign is negative;
6. The swap test is performed several times and the results are averaged; the higher the number of tests, the higher the precision will be.

3 Conclusion

In conclusion, we can say that QSVM is a promising algorithm in the Quantum Machine Learning field: its quantum nature, in fact, enables to speedily computing operations which are slow to compute with traditional, binary CPUs. The most evident fact is the matrix inversion: as reported[2], in fact, HHL algorithm takes $O(Ns\kappa log(\frac{1}{\epsilon}))$ on a binary CPU, while it takes $O(\log(N)s^2\frac{\kappa^2}{\epsilon})$ on a QPU, and this is due to the matrix inversion complexity. However, as the section below reports, some requirements and precautions are needed.

3.1 Requirements

As the reader may noticed reading this document, an important hardware requirement for this algorithm is QRAM: in fact, as already explained earlier, this structure is needed in order to store data samples in superposition. Without it, *i.e.* using only traditional RAM, the quantum nature of the algorithm cannot be fully expressed, making it almost useless. This, of course, brings with itself another issue, which could be the biggest limit of the algorithm so far: we need, in fact, a huge amount of qubits to store all the data for a real-life classification scenario; nowadays, quantum computers have only a number of qubits in the order of thousands, and this limits the number of problems which can be solved with QSVMs.

Another observation we could do is that the number of operations involved in this algorithm is enormous: this eight pages work, in fact, only scratches the surface of the subject, and just topics like quantum simulation and HHL algorithm deserves an in-depth discussion. We could say, then, that a software requirement (more a convenience than a real requirement) is to have a library that enables the usage of the algorithm on an high level fashion. This, indeed, already exists, and is developed by IBM Qiskit[3]. Since this has nothing to do with the quantum nature of the algorithm, however, is just cited.

3.2 Future research

Even knowing the limits of QSVMs, it is difficult to state which could be the next steps in their development. This document is, in fact, just a brief summary of years and years of research on the topic; thus, any consideration that could be made on this are just personal speculations.

An interesting aspect to perform research could be a smarter way to load data: QRAM is, as already said, limited due to the scarce amount of available qubits. Therefore, it could be interesting to investigate a way to dynamically load data into QRAM from traditional RAM; for instance, an equivalent of batch training for traditional SVMs[4] could be explored.

3.3 Further reading

The material used for this paper mainly comes from Johnston et. al[1], but other resources have been used to better understand the topic. However, these have not been used consistently, hence the fact that they are not mentioned elsewhere in the document; therefore, they are left to the interested readers to learn more. A more in-depth discussion on the approximation of traditional SVMs by least squares is provided by Ye et. al[5]. A description of how HHL algorithm works under the hood is given (besides Johnston et. al) by IBM Qiskit[2]. A really theoretical approach to Quantum supervised Machine Learning is reported in Havlicek et. al[6]. Finally, a practical approach to QSVM and a proof that the performances can be equated to classical SVMs are described in Park et. al[7].

References

[1] Johnston E.R, Harrigan N., Gimeno-Segovia M., *Programming Quantum Computers*, O'Reilly, July 2009.

[2] IBM Qiskit, *Solving Linear Systems of Equations using HHL*, retrieved from https://qiskit.org/textbook/ch-applications/hhl_tutorial.html.

[3] IBM Qiskit, *Quantum-enhanced Support Vector Machine (QSVM)*, retrieved from https://qiskit.org/documentation/stable/0.24/tutorials/machine_learning/01 _qsvm_classification.html.

[4] Yangguang L., Qinming H., Qi C., *Incremental batch learning with support vector machines*, Fifth World Congress on Intelligent Control and Automation, 2004, pp. 1857-1861 Vol.2, doi: 10.1109/WCICA.2004.1340997.

[5] Ye, J., Xiong, T., *SVM versus Least Squares SVM*, Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics, in Proceedings of Machine Learning Research 2:644-651, 2007, retrieved from https://proceedings.mlr.press/v2/ye07a.html.

[6] Havlíček, V., Córcoles, A.D., Temme, K. et al, *Supervised learning with quantum-enhanced feature spaces*, 2019, Nature 567, pp.209–212. doi: 10.1038/s41586-019-0980-2.

[7] Park J.-E., Quanz B., Wood S., Higgins H., Harishankar R., *Practical application improvement to Quantum SVM: theory to practice*, 2020, doi: 10.48550/arXiv.2012.07725.