Comparisons of Performance between Quantum and Classical Machine Learning

Christopher Havenstein
Southern Methodist University, chavenstein@smu.edu

Damarcus Thomas
Southern Methodist University, dtthomas@smu.edu

Swami Chandrasekaran
swamichandra@gmail.com

Follow this and additional works at: https://scholar.smu.edu/datasciencereview

Part of the Categorical Data Analysis Commons, Other Computer Sciences Commons, and the Quantum Physics Commons

Comparisons of Performance between Quantum and Classical Machine Learning

Christopher L. Havenstein[1], Damarcus T. Thomas[1], and
Swami Chandrasekaran[2]

[1] Master of Science in Data Science, Southern Methodist University,
Dallas, TX 75275 USA
[2] Managing Director, Innovation & Enterprise Solutions,
KPMG
{chavenstein, dtthomas}@smu.edu
swamichandra@gmail.com

Abstract. In this paper, we present a performance comparison of machine learning algorithms executed on traditional and quantum computers. Quantum computing has potential of achieving incredible results for certain types of problems [1], and we explore if it can be applied to machine learning. First, we identified quantum machine learning algorithms with reproducible code and had classical machine learning counterparts. Then, we found relevant data sets with which we tested the comparable quantum and classical machine learning algorithms performance. We evaluated performance with algorithm execution time and accuracy. We found that quantum variational support vector machines in some cases had higher accuracy than classical support vector machines on multi-class classification problems. The main conclusion was that quantum multi-class SVM classifiers have the potential to be useful in the future as quantum computer's available number of qubits increases.

1 Introduction

Quantum computing is an area of computing touted with achieving incredible results for factoring and unordered search problems [1]. Due to a phenomenon called quantum parallelism, which occurs because of superposition, quadratic or exponential increases in solution speed are possible with quantum computers when compared with classical computers [1]. However, quantum computing does not provide such a speedup for all problems and researchers are still learning in what situations it is best applied. The question we have is when quantum computing is combined with machine learning, do we still see these solution speedups or perhaps does the machine learning solution accuracy increase? Researchers have already shown that quantum computing and machine learning can be combined. Through the concept of quantum parallelism, quantum computing can be combined with machine learning when the machine learning algorithm can be sufficiently parallelized [2]. The problem we are trying to solve is how does quantum machine learning compare to classical machine learning and can their relative performance be measured?

First, to solve this problem, we have performed a thorough search of available quantum machine learning algorithms in research literature with reproducible code. Next, we required that the quantum machine learning algorithm to have a comparable classical machine learning algorithm available for comparison. Once identifying reproducible code to apply quantum machine learning algorithms, we identified data sets which would be suitable for the type of quantum machine learning algorithm—regression or classification. Through this process, we were introduced to quantum support vector machine (SVM) algorithms by a paper on "quantum enhanced feature spaces" that are applicable for classification machine learning problems [3].

Second, solving our problem required access to a quantum computer to run quantum machine learning algorithms on. We were introduced to the IBM® Quantum Experience [4], which can be combined with IBM® application programming interfaces (APIs) to interact with quantum computers. This IBM® Quantum Experience allows a user to make an account, be granted an API key, and an allotment of credits to use. This API key uniquely identifies a user and their available credits to access quantum compute. Through IBM® Qiskit Aqua API, quantum machine learning algorithms were available to be used on IBM® quantum computers [5]. Last, to compare quantum and classical machine learning algorithm performance, we decided on evaluation metrics to evaluate their relative performance. To determine if there was a quantum speedup over the relevant classical machine learning algorithm, we utilized wall time, which measures the program execution time on the classical or quantum hardware in microseconds. We additionally used accuracy as an evaluation metric to find the relative algorithm performance on the data sets selected.

Then, to describe our main results, an overview of our experiments are provided. Through our experiments, we compared two types of quantum SVMs each against a classical SVM. For the quantum SVMs compared, a kernel-based SVM and a variational SVM were both independently compared to a classical SVM for a binary classification problem. Then, we compared the quantum variational SVM to a classical SVM for a multi-class classification problem with three classes. We found that for binary classification problems, there was no significant improvement over the classical SVM. However, for the multi-class classification problem, the quantum variational SVM had higher accuracy than the classical SVM. Our main conclusions were that quantum variational SVMs performed better in our experiments than classical SVMs for multi-class classification problems. However, we observed no quantum speedups when comparing wall times for algorithm executions. We also experienced long queuing times before we could run our quantum algorithms on an IBM® quantum computer.

This paper is structured in a way to provide the reader with relevant background information and intuition to understand the key concepts. We begin by providing a conceptual overview of classical computing and machine learning algorithms—focused on areas that are relevant to quantum computing. An overview of quantum machine learning follows, concluding with a summary on when quantum machine learning should be considered for use and when it should

not. Next, descriptions of the binary classification data set and the multi-class classification data set follow. Subsequently, the methods used and experiments that were performed are defined. After providing the methods and experiments, the results of our experiments are given in more detail. After providing an analysis of the experiments, a descriptive analysis of the results is delivered. The authors then key ethical issues and relevant ethical principles are explored. Last, the authors supply their conclusions.

2   Classical Computing and Machine Learning

In classical computing, at the lowest level, data is stored with bits. Bits can take on only one of two possible values based on whether an electron charge exists [6]. If no electron charge exists, the value of a bit is 0. Whereas, if there is an electron charge, the value of a bit is 1. A sequence of bits is known as a bitstream [7]. Bitstreams store information which can be later analyzed. Table 1 on the next page shows a sampling of different configurations of bits. It should also be noted as an aside that one letter can be stored in a byte [8].
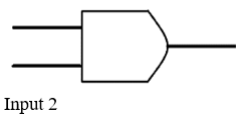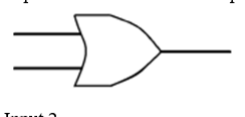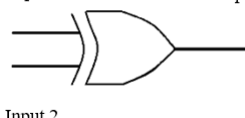
Table 1. Bit configurations

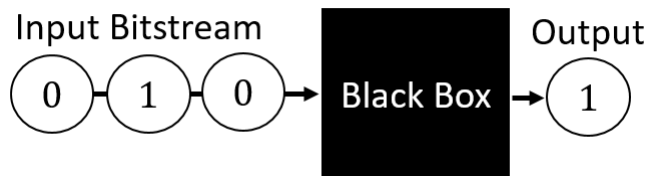| Number of bits	Example	Name
1	0	Bit
4	0, 1, 0, 0	Nibble
8	0, 0, 1, 1, 0, 0, 1, 0	Byte

One or more input bitstreams are operated on by using logic gates. The Oxford Dictionary of Computing states that a logic gate is "a device, usually but not exclusively electronic, that implements an elementary logic function; examples include AND, OR, NAND, and NOR gates, and inverters [9]." Logic gates tend to make more sense when shown pictorially. Please view table 2 for examples of input bit values and their output value after passing through various logic gates. Please note that table two spans across two pages. We created this table after learning concepts found in [10].

Next, bitstreams can be processed in serial or parallel. In serial processing, only one input bitstream can be processed at a given time to generate output. Please refer to figure 1 for a pictorial example of serial processing.

Next, parallel processing can utilize multiple input bitstreams to generate multiple outputs at a given time, which is a crucial distinction when evaluating machine learning within quantum computing. Please refer to figure 2 for a pictorial example of parallel processing. Later, in the quantum computing and machine learning section, we discuss why parallel processing is more important than serial processing for quantum computers.

Table 2. Logic gate examples

Type of logic gate	Logic gate diagram	Table representation		

		Input 1	Input 2	Output
And gate	Input 1 — Output / Input 2	0	0	0
		1	0	0
		0	1	0
		1	1	1

		Input 1	Input 2	Output
Or gate	Input 1 — Output / Input 2	0	0	0
		1	0	1
		0	1	1
		1	1	1

		Input 1	Input 2	Output
XOR gate	Input 1 — Output / Input 2	0	0	0
		1	0	1
		0	1	1
		1	1	0

Fig. 1. A simple figure illustrating serial processing. Notice an input bitstream feeds into a "black box" which abstracts a series of logic gates or other processing. Then, a bit is output from the black box.
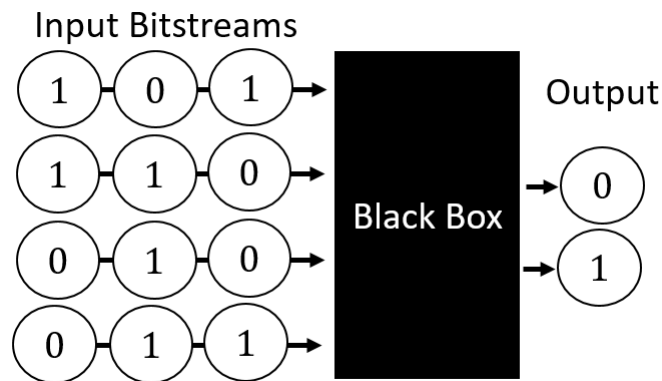
Fig. 2. A simple figure illustrating parallel processing. Notice there are four input bitstreams that feed into a "black box" which abstracts a series of logic gates or other processing. Then, two bits are output from the black box simultaneously.

Thankfully, we can abstract these concepts generally and not be bothered with them. However, they will later become relevant when describing quantum computing and how it pertains to machine learning. Though, we do have some examples of machine learning algorithms that do have components that are similar to logic gates. One such example are activation functions in neural networks. We do not expect the reader to fully understand neural networks. Conceptually, the activation function takes a series of weighted inputs to produce an output within a range defined by the type of activation function. There are various types of activation functions, but one example is a sigmoid function that restricts the output to either a 0 or 1. An example of a neural network is provided in figure 3.

As we alluded to, it is more typical in machine learning to think at an algorithmic level. Algorithms generally are comprised of a series of steps and may have iteration [11] and/or recursion [12]. In algorithms, the order of these steps is imperative, and sometimes the algorithm will not work if the steps are performed in the wrong order. Algorithms can be implemented using serial or parallel processing. However, a serial algorithm may not work well or at all in parallel. The reverse is also possible.

Machine learning algorithms are broadly described as applicable to classification or regression problems. First, we begin describing classification problems. In classification problems, the objective for the machine learning algorithm is to train a classifier to predict a label of a class. To do this, the machine learning algorithm learns from patterns in the distribution of the data for input variables, also known as features. The label being predicted, also known as a target, must be provided for training purposes. Overall, the data used to train a machine learning algorithm is called training data. Then, the classifier uses the input fea-
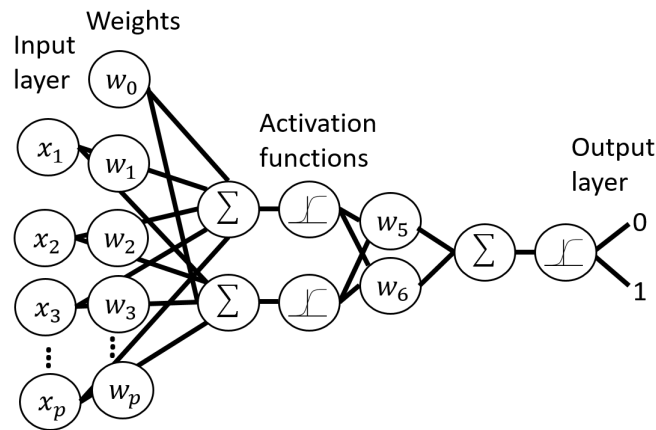
Fig. 3. A figure of a neural network architecture. Activation functions are similar to logic gates since they take a number of inputs to generate outputs.

tures for new unseen data, known as test data, to predict an unknown label. For instance, the classifier would be trained to predict given input features to predict the target label of "this" or "that" (binary classification), or perhaps "this" or "that" or "the other thing" and etc. (which is multi-class classification). Second, for regression machine learning problems, we still have training and test data and input features, but the predicted target is a real number—often known as a continuous value.

In this paper, we do not explore all of the available classification and regression machine learning algorithms that are usable on a classical computer. Regardless, the reader should be aware that machine learning algorithms typically fall into one of these two categories. Next, an overview of quantum computing and the concepts relevant to quantum machine learning is presented.

3   Quantum Computing and Machine Learning

Quantum computers look much different in person than the computers we use every day. The writers believe quantum computers look similar to metal chandeliers. For copyright purposes, we cannot provide a picture of a quantum computer in-line, but the writers suggest that you refer to the link in [13]. To paraphrase IBM® Research division, quantum computers require an extremely cold environment and containment from outside "electromagnetic radiation" to guard against the introduction of error into the quantum processing chip [13]. The quantum processing chip must be kept at "[approximately] -459.67 degrees Fahrenheit", according to IBM® [13].

While classical computers operate on bitstreams where each bit can take on one of two values each (0 or 1), quantum computers have quantum bits with

unique properties. First, we describe quantum bits, which are most commonly known as qubits. Qubits can be in multiple states at once, which can be easiest explained as spinning in the electron. If the electron spin is in the "up" position, then the qubit would equal 1 [14]. Whereas, if the electron spin were in the down position, the qubit would equal 0 [14]. However, in quantum computing, we have no way of knowing at a given time whether the qubit is in the up or down position. Therefore, a superposition (i.e. a sphere) of all potential values is created [14]. To compare bits to qubits visually, please refer to figure 4. The sphere created by the spinning of the qubit is commonly known in quantum computing as a "Bloch sphere [14]."
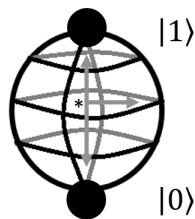
Fig. 4. A figure of a qubit. The notation used for the 1 and 0 values comes from Dirac, and is known as a "ket [14]." A series of qubit values are notated within a ket. The red arrows represent the spinning of the qubit, which creates a sphere pattern. This spinning represents the superposition of values available to the qubit.

In quantum computing, we still use logic gates, but the logic gates have different purposes. Quantum logic gates are commonly known as quantum gates, and they are essentially a transformation on one or more qubits. In table 3, we have provided a sample of quantum gate types from [15].

Table 3. Quantum gate examples

Quantum gate	Input	Classical gate	Short description
Hadamard gate	1 qubit	None	Creates superposition
Pauli-X gate	1 qubit	Not gate	Creates x rotation
Pauli-Y gate	1 qubit	None	Creates y rotation
Controlled not gate	2 or more qubits	Not gate	Used to measure 2nd qubit

For this paper, the reader does not need to become intimately familiar with quantum gates. However, the reader should know quantum gates are impor-

tant because they transform input qubits with transformations. These quantum transformations include but are not limited to establishing the quantum state (e.g., superposition) and to measure qubits (e.g., with the CNOT gate) [15]. However, if the reader would like a more in-depth mathematical explanation of quantum gates, please refer to [16].

Quantum computing becomes much more interesting when multiple qubits are entangled together [16]. In [16], entanglement is described as having "deep connections between spatially separated entities." When qubits are entangled, the superposition of values also increases. When some qubits, n, are entangled, the superposition of values is 2^n [16]. For two entangled qubits, we have represented the available values in the superposition in figure 5.
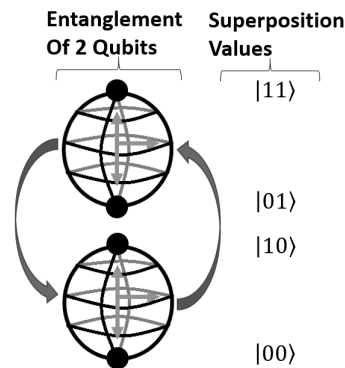
Fig. 5. A figure of a two entangled qubits. Notice that there are four unique values in the superposition of values.

At this juncture, the reader may intuit that a large number of qubits entangled leads to a large superposition of values available to perform operations on a quantum computer. At the authors time of writing this paper, IBM® currently has 5 qubit ("IBM® Q 5 Tenerife"), 14 qubit ("IBM® Q 14 Melbourn"), and 16 qubit ("IBM® Q 16 Rschlikon") quantum computers available for public use [17]. Also, IBM® has a 20 qubit quantum computer available for private use ("IBM® Q 20 Tokyo") [17]. Current quantum computing researchers in [16] suggest that the current highest performance supercomputers are only able to represent the superposition of values in a quantum computer up to 50 qubits. At 50 qubits, the superposition of values available to the quantum computer at a given time would be 1,125,899,906,842,624, or 2^{50}.

Regarding machine learning, it is beneficial to know the circumstances of when a quantum computer should be used. When a machine learning algorithm can be parallelized, "quantum parallelism" may be possible to apply, according to [2]. According to [18], when you have a quantum computer with large super-

position of values due to many entangled qubits, you can measure the function output for many values simultaneously. However, for a serial machine learning algorithm, a quantum computer's large superposition of values is wasted, since only a small portion of the superposition is utilized.

To reinforce the idea of when machine learning may be able to be applied on a quantum computer, we provide a conceptual example of a good potential quantum machine learning algorithm candidate. Random Forests are a type of machine learning algorithm that can be run in parallel and may make a good quantum machine learning algorithm. Leo Breiman introduced the random forest machine learning algorithm in his 2001 paper, which can be used for classification or regression problems [19]. For Random Forests, the overall data set (as in most machine learning algorithms) is broken into a training and test data set. However, from the training data set, approximately one third is set aside as an "out of bag" set. The out of bag set is used to evaluate the performance of the Random Forest model created from the training data set. Random Forests utilize "bagging" which is short for bootstrap aggregating, and we describe next [19]. Then, the n bootstrap samples are selected from the training data with replacement to create trees in the random forest. With replacement means that once an observation is selected, it is put back into the pool to be potentially sampled again. Then, for each bootstrapped sample, the process of creating a tree begins. For each bootstrap sample with a predetermined sample size, a random feature (e.g. a variable) is chosen to create a binary split from the training data [19]. This process of creating random binary splits continues until a user-defined maximum tree depth is reached. After all n trees are created in this way (which can be in parallel), the tree results are aggregated together to create the fitted random forest model. These aggregations generally are votes for classification problems and averages for regression problems. The fitted random forest model then predicts the target variable from the out of bag set to report the training error. Then, the model can be used to predict the target on the test data set. An example of a Random Forest is provided in figure 6.

While a Random Forest is a good potential candidate for a quantum machine learning algorithm, a Boosted Trees model is a bad example of a quantum machine learning algorithm. This is because boosted trees are fit in serial. XG-Boost is a type of Boosted Trees machine learning algorithm that is currently heavily used on classical computers [21]. Without going into as much depth as with Random Forest models, Boosted Tree models are required to fit a series of trees. After each tree is fit, the prediction errors are up-weighted and the correct predictions are down-weighted. Then, the next tree is fit after incorporating the new weights. This process continues until a number of trees are fit. Then, the results are aggregated together and weighted by the trees that had the best individual tree accuracy. An example of part of the boosted tree fitting process is provided in figure 7.

In summary, quantum machine learning should be considered when a particular machine learning algorithm candidate can be parallelized—to benefit from quantum parallelism. However, serial machine learning algorithms should be kept
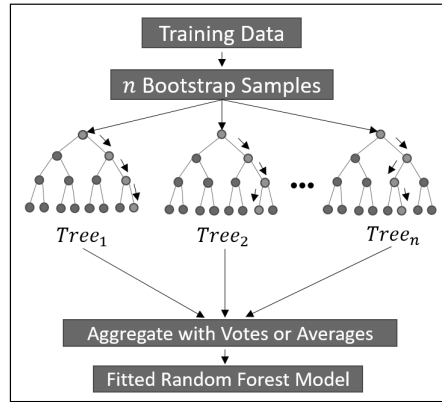
Fig. 6. A figure showing the Random Forest algorithm fitting process.
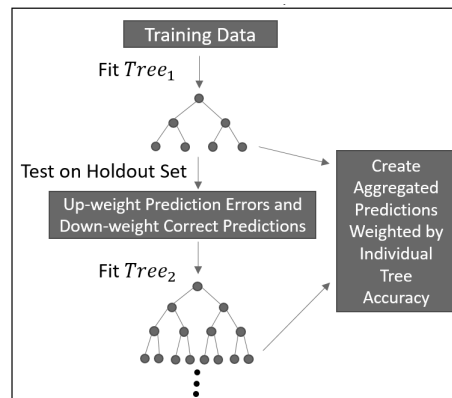
Fig. 7. A figure showing the Boosted Trees algorithm fitting process.

on a classical computer. Next, we began searching for viable quantum machine learning algorithms. We found that there was a common algorithm used as a foundation for many of these quantum machine learning algorithms, Grover's Search. Grover's Search is a quantum algorithm that has been shown to be useful for clustering, Support Vector Machines, and Quantum Neural Networks [1]. For data that is in unordered sets, Grover's Search can find a globally optimum value extremely quickly and has been shown very valuable for machine learning [20]. For quantum clustering problems, Grover's Search is used almost exclusively [21].

The following observation we made is that many of the types of quantum machine learning algorithms found in our literary review did not have reproducible code that we could find. We considered this a serious issue. There was only one type of quantum machine learning algorithm with reproducible code that we found, for quantum support vector machines. In table 4, we provide: (1) the types of quantum machine learning algorithms we found; (2) the type of machine learning problem the algorithm can be applied to; and (3) the citations found. We were only able to find reproducible code for quantum support vector machines. At this point, we proceed to our descriptions of the data sets used to solve our problem.

Table 4. Quantum machine learning algorithms found through literary review

Quantum ML algorithm	Applicable ML problems	Citations
Quantum annealing	Regression	[1], [22], [23]
Quantum adiabatic algorithms	Classification	[24], [25], [26]
Quantum neural networks	Regression or Classification	[27]
Quantum process tomography	Regression	[1], [28]
Quantum support vector machines	Classification	[3]

4 Data Sets

There were two data sets used to solve our problem. The first data set that was chosen is the UCI ML Breast Cancer Wisconsin (Diagnosis) data set that is available from scikit learn [29]. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. This data set has 569 observations with 30 columns that are used as features to predict the diagnosis value of a benign breast cancer cell nuclei, 1. The default diagnosis value, 0, indicates that a breast cancer cell nuclei is malignant. The breast cancer data set is suitable for quantum machine learning algorithms that solve supervised binary classification problems. For more in-depth explanation of the columns and their values in the breast cancer data set, please refer to Appendix I: Data Set Descriptions. The quoted

column descriptions in the breast cancer data dictionary came from a Kaggle entry for the breast cancer data set [29].

The second data set that was chosen is the UCI ML Wine data set. The features are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines [30]. The data set has 178 observations and the class identifier are 1, 2, 3. For more in depth explanation of the columns and their values in the wine data set, please refer to Appendix I: Data Set Descriptions. The quoted column descriptions in the breast cancer data dictionary came from the UCI data repository. The Wine data set is suitable for multi-classification quantum machine learning algorithms.

5 Methods and Experiments

We utilized Qiskit Aqua for quantum machine learning. We used both a local quantum computer simulator and a real quantum chip as our backend. We applied two quantum machine learning algorithms, Support Vector Machine's variational method and the Support Vector Machine's quantum kernel-based method for supervised classification. The direct kernel-based method for SVM was run on a classical computer that utilized a regular CPU.

We used the methods discovered by Vojtech Havlicek et al. [3] to apply two quantum machine learning methods to classification problems. We have applied two quantum support vector machine learning methods for binary classification problems, using the breast cancer data set. These two methods are the quantum variational method and the quantum kernel-based method.

In [3], the authors suggest that their quantum support vector machine algorithms could be applied in the near future on quantum architectures. They have even provided an example Jupyter notebook to show how one of their algorithms works [31]. We utilized their Jupyter notebook as a baseline for this analysis, since the reader will likely have the aptitude to apply quantum machine learning in Python with such a Jupyter notebook—after downloading the required package libraries. The authors in [3] state that their methods were run on a quantum computer, but they have not shown this in their Jupyter notebook. Since, their code in their notebook is rerunning with a local quantum simulator in Qiskit Aqua. Next, we describe the quantum variational method and the quantum kernel-based method, beginning with the quantum variational method.

With the quantum variational method, hyperplane(s) are calculated and used for classification of new test data. In the simplest sense, if the training data are used and plotted in two dimensions, similar to a scatterplot, a hyperplane is the line (or curve in higher dimensions) that separates the class values that the training data belongs to (often denoted as the yi values). The main advantage of the quantum variational method is that it can handle classification of more than two classes for the response variable. The disadvantage of the quantum variational method is that it requires two sequential quantum algorithms to be run, which is much more computationally intensive than the quantum kernel-based

method. The first algorithm is used when trying to compute the hyperplane(s) with the training data. This first algorithm can be found in figure 8. The second algorithm runs after the hyperplane(s) from the first algorithm are calculated and are used to begin classifying the new test data. You can find the second algorithm in figure 9 [3].

Algorithm 1 Quantum variational classification: the training phase

1: **Input** Labeled training samples $T = \{\vec{x} \in \Omega \subset \mathbb{R}^n\} \times \{y \in C\}$, Optimization routine,
2: **Parameters** Number of measurement shots R, and initial parameter $\vec{\theta}_0$.
3: Calibrate the quantum Hardware to generate short depth trial circuits.
4: Set initial values of the variational parameters $\vec{\theta} = \vec{\theta}_0$ for the short-depth circuit $W(\vec{\theta})$
5: **while** Optimization (e.g. SPSA) of $R_{\text{emp}}(\vec{\theta})$ has not converged **do**
6:     **for** $i = 1$ **to** $|T|$ **do**
7:         Set the counter $r_y = 0$ for every $y \in C$.
8:         **for** $shot = 1$ **to** R **do**
9:             Use $\mathcal{U}_{\Phi(\vec{x}_i)}$ to prepare initial feature-map state $|\Phi(\vec{x}_i)\rangle\langle\Phi(\vec{x}_i)|$
10:             Apply discriminator circuit $W(\vec{\theta})$ to the initial feature-map state .
11:             Apply $|C|$ - outcome measurement $\{M_y\}_{y \in C}$
12:             Record measurement outcome label y by setting $r_y \to r_y + 1$
13:         **end for**
14:         Construct empirical distribution $\hat{p}_y(\vec{x}_i) = r_y R^{-1}$.
15:         Evaluate $\Pr(\tilde{m}(\vec{x}_i) \neq y_i | m(\vec{x}) = y_i)$ with $\hat{p}_y(\vec{x}_i)$ and y_i
16:         Add contribution $\Pr(\tilde{m}(\vec{x}_i) \neq y_i | m(\vec{x}) = y_i)$ to cost function $R_{\text{emp}}(\vec{\theta})$.
17:     **end for**
18:     Use optimization routine to propose new $\vec{\theta}$ with information from $R_{\text{emp}}(\vec{\theta})$
19: **end while**
20: **return** the final parameter $\vec{\theta}^*$ and value of the cost function $R_{\text{emp}}(\theta^*)$

Fig. 8. A figure showing the first algorithm in the quantum variational SVM method. This algorithm was found in [3].

The second quantum support vector method, is the quantum kernel-based method, is restricted to only binary classification problems. However, it is much quicker to complete since the quantum computer is only used for one algorithm. Whereas, the quantum variational method has to run two algorithms, one to calculate the hyperplane(s) and another to classify the test data. According to the authors of [32], the kernel approach used by the quantum kernel-based method "cannot be estimated classically." They also mention in their paper, that there won't be an advantage to using quantum kernel-based Support Vector Machines (SVM) if the data are not already complex to fit on classical silicon-based computers.

The steps for the quantum kernel-based SVM algorithm follow. As stated previously, with the quantum kernel-based method, we are restricted to only having two labels to classify (e.g. binary classification). The authors stated in [3] that it is technically possible to classify more than two labels, but partitioning would have to occur into two label sets, and the developer would require a deeper understanding of quantum hardware. First, a kernel matrix is estimated with the quantum computer by using all of the training data rows. Then, the classical computer takes this quantum kernel matrix of the training data to calculate

---

Algorithm 2 Quantum variational classification: the classification phase

1: **Input** An unlabeled sample from the test set $\vec{s} \in S$, optimal parameters $\vec{\theta}^*$ for the discriminator circuit.
2: **Parameters** Number of measurement shots R
3: Calibrate the quantum Hardware to generate short depth trial circuits.
4: Set the counter $\mathrm{r}_y = 0$ for every $y \in C$.
5: **for** $shot = 1$ **to** R **do**
6:     Use $\mathcal{U}_{\Phi(\vec{s})}$ to prepare initial feature-map state $|\Phi(\vec{s})\rangle\langle\Phi(\vec{s})|$
7:     Apply optimal discriminator circuit $W(\vec{\theta^*})$ to the initial feature-map state .
8:     Apply $|C|$ - outcome measurement $\{M_y\}_{y \in C}$
9:     Record measurement outcome label y by setting $\mathrm{r}_y \to \mathrm{r}_y + 1$
10: **end for**
11: Construct empirical distribution $\hat{p}_y(\vec{s}) = \mathrm{r}_y R^{-1}$.
12: Set label $= \mathrm{argmax}_y\{\hat{p}_y(\vec{s})\}$
13: **return** label

Fig. 9. A figure showing the first algorithm in the quantum variational support vector machine method. This algorithm was found in [3].

the support vectors with the classical computer. After the support vectors are calculated with the classical computer, classification can begin by the classical computer to predict the labels for the test data set. Because the support vectors were already calculated and the quantum kernel matrix from the training data is available, the label can be directly calculated for each of the test data rows.

The classical support vector machine method used for comparative purposes is the "Support Vector Machine Radial Basis Function Kernel (SVM RBF Kernel)" which is made available in Qiskit Aqua and is briefly described in their documentation [32].

For our experiments, we used code from our Jupyter Notebook which can be found in Appendix II: Python Code from Quantum SVM Analysis Jupyter Notebook. Table 5 lists out the complete experimental configurations. Please note that the runtime and accuracy percentage will be compared amongst the two different algorithms. We also utilized four different types of back-ends, which were: (1) the local CPU; (2) the quantum simulator; (3) the state vector simulator; and (4) a real quantum chip. In the first approach, we use a variational circuit as given in [33] [34] that generates a separating hyperplane in the quantum feature space. In the second approach, we use the quantum computer to estimate the kernel function of the quantum feature space directly and implement a conventional SVM.

6   Analysis

After deciding on how to setup the experiments, we began the analysis. We tried applying the quantum variational SVM model to the same training and test split for the breast cancer data set (with 20 training and 10 test observations). As expected, this model took too long to fit, with a total approximate run time of 25 minutes. However, the accuracy when we use the quantum simulator for the quantum variational SVM model completed was 95%, as we show in Table 6. The classic SVM had a slower wall time with only an 85% accuracy. The

Table 5. Experimental setup configurations

Algorithm compared	Quantum or classical	Back-end	Runtime	Data set	Evaluation metric
Kernel-based SVM	Quantum	Quantum back-end (ibmqx4)	Wall time (μs)	UCI Breast Cancer Wisconsin	Accuracy
Kernel-based SVM	Quantum	Quantum simulator	Wall time (μs)	UCI Breast Cancer Wisconsin	Accuracy
Kernel-based SVM	Quantum	Statevector simulator	Wall time (μs)	UCI Breast Cancer Wisconsin	Accuracy
Classic SVM	Classical	Local CPU environment	Wall time (μs)	UCI Breast Cancer Wisconsin	Accuracy
Variational SVM	Quantum	Quantum simulator	Wall time (μs)	UCI Breast Cancer Wisconsin	Accuracy
Variational (multi-class) SVM	Quantum	Quantum back-end (ibmqx4)	Wall time (μs)	Wine	Accuracy
Variational (multi-class) SVM	Quantum	Quantum simulator	Wall time (μs)	Wine	Accuracy
Variational (multi-class) SVM	Classical	Local CPU environment	Wall time (μs)	Wine	Accuracy

simulated variational model outperformed the local CPU as it should have since such variational circuit classifiers are directly related to conventional SVMs [35].

Table 6. Quantum variational SVM results on the Breast Cancer Wisconsin data set

Classifiers	Back-end	Algorithm	Wall Time	Accuracy
Quantum SVM	Quantum simulator	Variational SVM	5.96 μs	95.00%
Classic SVM	Local CPU environment	SVM	6.20 μs	85.00%

Next, as our control, we applied the classical "Support Vector Machine Radial Basis Function Kernel (SVM RBF Kernel)" to the same training and test split for the breast cancer data set (with 20 training and 10 test observations) [32]. Table 7 summarizes the kernel-based binary classification results. The quantum state vector simulator had 100% accuracy in approximately 6 microseconds. The classic SVM run on a local CPU had 85% accuracy.

Table 7. Quantum kernel-based SVM results on the Breast Cancer Wisconsin data set

Classifiers	Back-end	Algorithm	Wall Time	Accuracy
Quantum SVM	ibmqx4	SVM RBF Kernel	6.91 μs	80.00%
Quantum SVM	Statevector simulator	SVM RBF Kernel	5.96 μs	100.00%
Classic SVM	Local CPU environment	SVM	6.20 μs	85.00%

We ran the classical SVM RBF Kernel model a few more times, and received the same result, of 85% prediction accuracy each time. It was after this we realized that the quantum SVM methods appeared to have more variability during the model fit process than the classical SVM model did. We were also

impressed with how easy it was after running through one example dataset to fit quantum machine learning models. On multi-class variational SVM comparisons as noted in Table 8, the quantum simulator had the best accuracy (100%). The quantum SVM run on a quantum chip had an accuracy of 93%. The local CPU SVM had 90% accuracy.

Table 8. Quantum multi-class variational SVM results on the Wine data set

Classifiers	Back-end	Algorithm	Wall Time	Accuracy
Quantum Multi-class SVM	ibmqx4	Variational SVM	6.44 μs	93.33%
Quantum Multi-class SVM	Statevector simulator	Variational SVM	5.96 μs	100.00%
Classic SVM	Local CPU environment	SVM	6.20 μs	90.00%

7   Results

The results on the Breast Cancer Wisconsin data set are reviewed first. On the Breast Cancer Wisconsin data set, only simulator results were obtained for the quantum variational SVM while quantum computer results were obtained for the quantum kernel-based SVM. The quantum variational SVM results were not obtained after waiting for approximately a day for the results to return. It was unclear if this wait was due to queuing issues or the run time of the quantum variational SVM on the IBM® quantum computer. For the quantum variational SVM simulator results, the accuracy was 95.00% when compared to 85.00% with the classical SVM to predict benign or malignant breast cancer tumors. For the quantum kernel-based SVM, the best results were obtained by the quantum SVM simulator with 100.00% accuracy. The classical SVM had the next highest accuracy at 85.00%. Then last, the quantum kernel-based SVM run on a quantum computer performed the worst with 80.00% accuracy on the Breast Cancer Wisconsin data set.

The results for the Wine data set are reviewed next. The quantum multi-class variational SVM run on the simulator had the best accuracy at 100.00% to predict the three types of wine. Next, the quantum multi-class variational SVM run on the quantum computer had 93.33% accuracy. Last, the classic SVM had 90.00% accuracy on the Wine data set.

There were a few key insights gained after reviewing these results on both data sets. First, fitting quantum SVM models on large data sets for binary classification problems appeared initially to be computationally inefficient. The authors believe that using 20 training and 10 test observations isn't enough for practical use, and is primarily useful only as a proof of concept. The authors would like to see if the performance improves with a quantum computer with more qubits. The authors will also state that the most likely explanation for long runtime of the SVMs on IBM⒭ quantum computers was due to long queuing times. The justification for this statement is due to the low wall-times (measured in microseconds) returned from the quantum computer, while the overall runtime was typically long (greater than 20 minutes).

Second, the difference in predictive accuracy between the simulated quantum SVM and the actual quantum SVM suggests that the simulator tends to overstate the accuracy of quantum SVMs run on quantum computers.

Finally, The most interesting results were found from the quantum variational SVM to perform multi-class classification on the Wine Data Set. Since, the quantum variational SVM for the multi-class classification problem had higher accuracy than the classical SVM (93.33% compared to 90.00%).

8 Ethics

During our analysis, we discovered a potential ethical issue that we deemed worthy of discussion. Here, we provide some background to introduce the potential ethical issue. While the wall times we observed to fit the quantum SVMs were short—typically taking a few microseconds—the queuing times were long in comparison. We do acknowledge that in our python code we initially did not think to include a runtime timer in addition to the wall time timer. If we did, we would have recorded from our experiences very long code run times. For the kernel-based quantum SVM on the Breast Cancer Wisconsin data set, the code execution time from running the line to receiving results back from the quantum computer was approximately 20 to 30 minutes. Then, with the variational quantum SVM on the Wine data set, we received results back from the quantum computer in approximately 5 to 6 hours. This queueing time for the variational SVM on the Wine multi-class classification data set was unfortunate since the results were the most promising.

The experience of waiting in a slow queue to access the IBM⒭ quantum compute led us to wonder, what if clients were provided unequal access to the quantum computers queue? This is a potential ethical issue, which right now doesn't have much impact other than being an annoyance, but in the future could become a serious ethical issue. We believe that access to quantum computers shouldn't be restricted.

As a framework to evaluate the ethical issue we are exploring—access to quantum computers shouldn't be restricted—we used two ethical principles. The first ethical principle we used for evaluation of this ethical issue was beneficence accessed from the Stanford Encyclopedia of Philosophy [36]. Then, we used the

ethical principle Be fair and take action not to discriminate from the ACM code of ethics [37].

Initially, we evaluated the ethical issue, access to quantum computers should not be restricted, with the ethical principle "beneficence [36]." According to [36], the ethical principle of beneficence alludes to a moral obligation to provide some service or to perform an action to help others. If IBM® intends to provide a free quantum computing service, then that service should be provided with equal access to all requesting clients. If IBM® is not providing equal access to their free quantum computing service, then beneficence is violated. From the authors' perspective, IBM® is doing a great job with providing access to quantum compute currently. In the probable future where quantum computers become more impactful, we believe that IBM® should continue to provide this free quantum computing service for educational purposes.

Last, we evaluated this potential ethical issue with the ethical principle "Be fair and take action not to discriminate [37]." This ethical principle is similar to beneficence, but also focuses on equal treatment of all people. For IBM® not to be discriminating with access to their free quantum compute, all clients would have to be treated equally. To not violate this ethical principle, no discriminating on clients on any basis can occur when requesting access with a valid API key to the free IBM® quantum compute. If any discriminating or unequal treatment occurs where some clients are more equal than others, then the ACM ethical principle "be fair and take action not to discriminate" is violated. We challenge IBM® in the future to keep comparable free access available to their quantum computers as is available now for educational purposes.

9   Conclusions and Future Work

Experimentally we have shown that a classifier can exploit a quantum feature space. The kernel of this feature space has been conjectured to be hard to estimate classically. In the experiment we find that even in the presence of noise, we are capable of achieving success rates up to 93.00% with quantum compute. In the future it becomes intriguing to find suitable feature maps for this technique with provable quantum advantages while providing significant improvement on real world data sets. With the ubiquity in machine learning, we are optimistic that our technique will find application beyond with SVM classifiers. Two major issues should be addressed, for one, the quantum simulator was executed on a local classical machine. Despite the efforts of the backend, it is not feasible for a classical computer to imitate a quantum computer without some type of high-performance computing power, and so the simulator results are purely observational. Furthermore, the presence of noise diminishes the results on a real quantum chip especially with the limited number of qubits. Small quantum computers and larger special purpose quantum simulators, annealers, etc., exhibit promising applications in machine learning and data analysis [38]. However we are in a certain quantum era called, the Noisy Intermediate-Scale Quantum (NISQ) era. Here "intermediate scale" refers to the size of quantum computers

which will be available in the next few years. NISQ tends to refer to quantum computers having number of qubits ranging from 50 to a few hundred 50 qubits as a significant milestone. This is because that's beyond what can be simulated by brute force using the most powerful existing digital supercomputers. "Noisy" emphasizes that we will have imperfect control over those qubits; the noise will place serious limitations on what quantum devices can achieve in the near term [39]. It is unknown whether the NISQ era will yield great developments and technologies that speed up the time to solutions. Nonetheless, we are living in exciting times where experimentation on real quantum chips are possible, and a new quantum community is developing and trickling over to the curiosity of the general public minds.

References

1. Peter Wittek. *Quantum machine learning: what quantum computing means to data mining.* Academic Press, 2014.
2. Daoyi Dong, Chunlin Chen, Hanxiong Li, and Tzyh-Jong Tarn. Quantum reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 38(5):1207–1220, 2008.
3. Vojtech Havlicek, Antonio D Córcoles, Kristan Temme, Aram W Harrow, Jerry M Chow, and Jay M Gambetta. Supervised learning with quantum enhanced feature spaces. *arXiv preprint arXiv:1804.11326*, 2018.
4. IBM Staff. Ibm quantum experience. `https://quantumexperience.ng.bluemix.net/qx/experience`.
5. IBM Staff. Quantum information science kit. `https://qiskit.org/aqua`.
6. David Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A*, 400(1818):97–117, 1985.
7. Community Editor. Bitstream. `https://en.wikipedia.org/wiki/Bitstream`.
8. Ashley Taylor. Bits bytes. `https://web.stanford.edu/class/cs101/bits-bytes.html`.
9. Reference Staff. Logic gates quick reference. `http://www.oxfordreference.com/view/10.1093/oi/authority.20110810105307777`.
10. Margaret Rouse. logic gate (and, or, xor, not, nand, nor and xnor). `https://whatis.techtarget.com/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR`.
11. Reference Staff. Iteration. `https://www.oxfordlearnersdictionaries.com/us/definition/english/iteration`.
12. Chris Alvin. Recursive function. `http://pages.cs.wisc.edu/~calvin/cs110/RECURSION.html`.
13. IBM Staff. Inside look: Quantum computer. `https://www.research.ibm.com/ibm-q/learn/what-is-ibm-q/images/infographic-inside.jpg`.
14. Eleanor G Rieffel and Wolfgang H Polak. *Quantum computing: A gentle introduction.* MIT Press, 2011.
15. Community Editor. Quantum logic gates. `https://en.wikipedia.org/wiki/Quantum_logic_gate`.
16. Krysta M Svore and Matthias Troyer. The quantum future of computation. *Computer*, 49(9):21–30, 2016.
17. IBM Staff. Quantum devices simulators. `https://www.research.ibm.com/ibm-q/technology/devices/`.

18. J. Lanzagorta, M. Uhlmann. Quantum parallel real? `https://www.researchgate.net/profile/Marco_Lanzagorta/publication/252477910_Is_quantum_parallelism_real/links/54982fc80cf2eeefc30f7e2c/Is-quantum-parallelism-real.pdf`.

19. L. Breiman. Random forest. `https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf`.

20. Christoph Durr and Peter Hoyer. A quantum algorithm for finding the minimum. *arXiv preprint quant-ph/9607014*, 1996.

21. Esma Aïmeur, Gilles Brassard, and Sébastien Gambs. Quantum speed-up for unsupervised learning. *Machine Learning*, 90(2):261–287, 2013.

22. Emile HL Aarts et al. Simulated annealing: Theory and applications. 1987.

23. AB Finnila, MA Gomez, C Sebenik, C Stenson, and JD Doll. Quantum annealing: a new method for minimizing multidimensional functions. *Chemical physics letters*, 219(5-6):343–348, 1994.

24. Hartmut Neven, Vasil S Denchev, Geordie Rose, and William G Macready. Training a binary classifier with the quantum adiabatic algorithm. *arXiv preprint arXiv:0811.0416*, 2008.

25. Wim Van Dam, Michele Mosca, and Umesh Vazirani. How powerful is adiabatic quantum computation? In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 279–287. IEEE, 2001.

26. David DiVincenzo. Quantum information processing. lecture notes.

27. Sanjay Gupta and RKP Zia. Quantum neural networks. *Journal of Computer and System Sciences*, 63(3):355–383, 2001.

28. Isaac L Chuang and Michael A Nielsen. Prescription for experimental determination of the dynamics of a quantum black box. *Journal of Modern Optics*, 44(11-12):2455–2467, 1997.

29. Nick Street. Scikit learn dataset load. `http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html`.

30. Nick Street. Breast cancer wisconsin (diagnostic) data set. `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data/version/2`.

31. Qiskit Community. Qiskit aqua-tutorials. `https://github.com/Qiskit/aqua-tutorials/blob/master/artificial_intelligence/svm_qkernel.ipynb`.

32. Qiskit Community. Support vector machine radial basis function kernel svm rbf kernel. `https://qiskit.org/documentation/aqua/algorithms.html#support-vector-machine-radial-basis-function-kernel-svm-rbf-kernel`.

33. Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242, 2017.

34. Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Hartmut Neven. Quantum algorithms for fixed qubit architectures. *arXiv preprint arXiv:1703.06199*, 2017.

35. Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

36. T Beauchamp. The principle of beneficence in applied ethics. `https://plato.stanford.edu/entries/principle-beneficence/`.

37. Committee Editor. Acm code of ethics. `https://www.acm.org/code-of-ethics`.

38. Rodion Neigovzen, Jorge L Neves, Rudolf Sollacher, and Steffen J Glaser. Quantum pattern recognition with liquid-state nuclear magnetic resonance. *Physical Review A*, 79(4):042321, 2009.

39. John Preskill. Quantum computing in the nisq era and beyond. *arXiv preprint arXiv:1801.00862*, 2018.

Appendix I: Data Set Descriptions

First, information about the Breast Cancer Wisconsin (Diagnostic) Data Set was acquired from the UCI Machine Learning Repository of data sets. The data set had 32 features, 569 rows, and a diagnosis/classification for each row of either a malignant breast cancer tumor or a benign breast cancer tumor. In other words, this is a binary classification data set. Then, three values were provided for each of the features in table 9.

Table 9. Breast Cancer Wisconsin (Diagnosis) feature names acquired from the UCI Machine Learning Repository. Note that there are three values captured for each feature below for each row in the data set.

Feature number	Feature name
1	Radius
2	Texture
3	Perimeter
4	Area
5	Smoothness
6	Compactness
7	Concavity
8	Concave points
9	Symmetry
10	Fractal dimension

Second, information about the Wine data set was acquired from the UCI Machine Learning Repository of data sets. The dataset had 13 attributes, 178 rows, and a classification of one of three types of wine. In other words, this is a multi-class classification data set. The feature names are listed in table 10.

Table 10. Wine data set feature names acquired from the UCI Machine Learning Repository.

Feature number	Feature name
1	Alcohol
2	Malic Acid
3	Ash
4	Alcalinity of ash
5	Magnesium
6	Total phenols
7	Flavanoids
8	Nonflavanoid phenols
9	Proanthocyanins
10	Color intensity
11	Hue
12	OD280/OD315 of diluted wines
13	Proline

Appendix II: Python Code from Quantum SVM Analysis Jupyter Notebook

```python
#Breast Cancer on a Real Quantum chip Kernel Direct Method
from datasets import *
from qiskit_aqua.utils import split_dataset_to_data_and_labels
from qiskit_aqua.input import get_input_instance
from qiskit_aqua import run_algorithm
from qiskit import execute, register

import getpass
try:
    APItoken = getpass.getpass('Please input your token and hit enter:
        ')
    qx_config = {
        "APItoken": APItoken,
        "url":"https://quantumexperience.ng.bluemix.net/api"}
except (ConnectionError, ValueError, RuntimeError, TypeError,
    NameError):
    print("That was not a valid token. Try again...")

register(qx_config['APItoken'], qx_config['url'])

sample_Total, training_input, test_input, class_labels = \
    Breast_cancer(training_size=10, test_size=10, n=2, # 2 is the
        dimension of each data point
    gap=0.3, PLOT_DATA=False)
```

```python
datapoints, class_to_label =
    split_dataset_to_data_and_labels(test_input)

params = {
    'problem': {'name': 'svm_classification', 'random_seed': 10598},
    'algorithm': {
        'name': 'QSVM.Kernel'
    },
    'backend': {'name': 'ibmqx4', 'shots': 1024},
    'feature_map': {'name': 'SecondOrderExpansion', 'depth': 2,
        'entanglement': 'linear'}
}

algo_input = get_input_instance('SVMInput')
algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
algo_input.datapoints = datapoints[0] # 0 is data, 1 is labels

result = run_algorithm(params, algo_input)
print(result)

#Breast Cancer Data set Variational method on QASM simulator
from datasets import *
from qiskit_aqua.utils import split_dataset_to_data_and_labels,
    map_label_to_class_name
from qiskit_aqua.input import get_input_instance
from qiskit_aqua import run_algorithm

n = 2 # dimension of each data point

sample_Total, training_input, test_input, class_labels =
    Breast_cancer(training_size=10, test_size=10, n=n, gap=0.3,
    PLOT_DATA=False)

datapoints, class_to_label =
    split_dataset_to_data_and_labels(test_input)

params = {
    'problem': {'name': 'svm_classification', 'random_seed': 10598},
    'algorithm': {'name': 'QSVM.Variational', 'override_SPSA_params':
        True},
    'backend': {'name': 'qasm_simulator', 'shots': 1024},
    'optimizer': {'name': 'SPSA', 'max_trials': 200, 'save_steps': 1},
    'variational_form': {'name': 'RYRZ', 'depth': 3},
    'feature_map': {'name': 'SecondOrderExpansion', 'depth': 2}
}

algo_input = get_input_instance('SVMInput')
algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
```

```python
algo_input.datapoints = datapoints[0]

result = run_algorithm(params, algo_input)
print(result)

#Breast Cancer on QASM simulator Kernel Direct Method
from datasets import *
from qiskit_aqua.utils import split_dataset_to_data_and_labels
from qiskit_aqua.input import get_input_instance
from qiskit_aqua import run_algorithm

sample_Total, training_input, test_input, class_labels = \
    Breast_cancer(training_size=10, test_size=10, n=2, # 2 is the
        dimension of each data point
    gap=0.3, PLOT_DATA=False)

datapoints, class_to_label = 
    split_dataset_to_data_and_labels(test_input)

params = {
    'problem': {'name': 'svm_classification', 'random_seed': 10598},
    'algorithm': {
        'name': 'QSVM.Kernel'
    },
    'backend': {'name': 'qasm_simulator', 'shots': 1024},
    'feature_map': {'name': 'SecondOrderExpansion', 'depth': 2,
        'entanglement': 'linear'}
}

algo_input = get_input_instance('SVMInput')
algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
algo_input.datapoints = datapoints[0] # 0 is data, 1 is labels

result = run_algorithm(params, algo_input)
print(result)

#Wine Data set QVSM kernel direct method
from datasets import *
from qiskit_aqua.utils import split_dataset_to_data_and_labels
from qiskit_aqua.input import get_input_instance
from qiskit_aqua import run_algorithm
import numpy as np

n = 2 # dimension of each data point

sample_Total, training_input, test_input, class_labels =
    Wine(training_size=40, test_size=10, n=n, PLOT_DATA=False)

temp = [test_input[k] for k in test_input]
```

```python
total_array = np.concatenate(temp)

params = {
    'problem': {'name': 'svm_classification', 'random_seed': 10598},
    'algorithm': {
        'name': 'QSVM.Kernel',
    },
    'backend': {'name': 'qasm_simulator', 'shots': 1024},
    # 'multiclass_extension': {'name': 'OneAgainstRest'},
    'multiclass_extension': {'name': 'AllPairs'},
    # 'multiclass_extension': {'name': 'ErrorCorrectingCode',
        'code_size': 5},
    'feature_map': {'name': 'SecondOrderExpansion', 'depth': 2,
        'entangler_map': {0: [1]}}
}

algo_input = get_input_instance('SVMInput')
algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
algo_input.datapoints = total_array

result = run_algorithm(params, algo_input)
print(result)

#SVM Classical algorithm Code
# load required libraries
import numpy as np
import scipy
from scipy.linalg import expm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA
from qiskit_aqua.svm.data_preprocess import *
from qiskit_aqua.input import get_input_instance
from qiskit_aqua import run_algorithm
from qiskit import register, available_backends

import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) # uncomment it to see detailed logging

### import Qconfig and set APIToken and API url and prepare backends

### Used https://hk.saowen.com/a/
## 5070648b000bbba96268d1f35ba20d241fbccff1e520590f8a3327f2c5c15f0a
    to set up Qconfig.py
```

```python
### get Qconfig set up for API token
try:
    import sys
    sys.path.append("../../") # go to parent dir
    import Qconfig
except Exception as e:
    print(e)
#set api
APItoken=getattr(Qconfig, 'APItoken', None)
url = Qconfig.config.get('url', None)
#hub = Qconfig.config.get('hub', None)
#group = Qconfig.config.get('group', None)
#project = Qconfig.config.get('project', None)
try:
    #register(APItoken, url, hub, group, project)
    register(APItoken)
except Exception as e:
    print(e)

print("Backends: {}".format(available_backends()))

# Create Breast_cancer data preparation function
def Breast_cancer(training_size, test_size, n, PLOT_DATA):
    class_labels = [r'Malignant', r'Benign']
    data, target = datasets.load_breast_cancer(True)
    sample_train, sample_test, label_train, label_test = \
        train_test_split(data, target, test_size=0.3, random_state=12)

    # Now we standarize for gaussian around 0 with unit variance
    std_scale = StandardScaler().fit(sample_train)
    sample_train = std_scale.transform(sample_train)
    sample_test = std_scale.transform(sample_test)

    # Now reduce number of features to number of qubits
    pca = PCA(n_components=n).fit(sample_train)
    sample_train = pca.transform(sample_train)
    sample_test = pca.transform(sample_test)

    # Scale to the range (-1,+1)
    samples = np.append(sample_train, sample_test, axis=0)
    minmax_scale = MinMaxScaler((-1, 1)).fit(samples)
    sample_train = minmax_scale.transform(sample_train)
    sample_test = minmax_scale.transform(sample_test)

    # Pick training size number of samples from each distro
    training_input = {key: (sample_train[label_train == k,
        :])[:training_size] for k, key in enumerate(class_labels)}
    test_input = {key: (sample_train[label_train == k,
        :])[training_size:(
    training_size+test_size)] for k, key in enumerate(class_labels)}
```

```python
    if PLOT_DATA:
        for k in range(0, 2):
            plt.scatter(sample_train[label_train == k,
                0][:training_size], sample_train[label_train == k,
                1][:training_size])

        plt.title("PCA dim. reduced Breast Cancer dataset")
        plt.show()

    return sample_train, training_input, test_input, class_labels

#Call the Breast Cancer function to prepare the data
sample_train, training_input, test_input, class_labels = \
    Breast_cancer(training_size=20, test_size=10, n=2, PLOT_DATA=True)
#sample_train, training_input, test_input, class_labels = \
    Breast_cancer(training_size=105, test_size=45, n=2,
    PLOT_DATA=True)
#sample_Total, training_input, test_input, class_labels = \
    Breast_cancer(training_size=210, test_size=90, n=2,
    PLOT_DATA=True)

print(class_labels)

total_array, label_to_labelclass = get_points(test_input,
    class_labels)

#Set SVM_Variational parameter JSON string
params = {
    'problem': {'name': 'svm_classification'},
    'backend': {'name': 'local_qasm_simulator', 'shots':1000},
    'algorithm': {
        'name': 'SVM_Variational', #SVM_RBF_Kernel for classical,
            SVM_QKernel
        'print_info' : True
    }
}


algo_input = get_input_instance('SVMInput')

algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
algo_input.datapoints = total_array

#Run Quantum Variational SVM model
result = run_algorithm(params,algo_input)

print("testing success ratio: ", result['test_success_ratio'])
print("predicted labels:", result['predicted_labels'])
```

```python
#Set SVM_QKernel parameter JSON string
params = {
    'problem': {'name': 'svm_classification'},
    'backend': {'name': 'local_qasm_simulator', 'shots':1000},
    'algorithm': {
        'name': 'SVM_QKernel', #SVM_RBF_Kernel for classical,
            SVM_Variational
        'print_info' : True
    }
}

algo_input = get_input_instance('SVMInput')

algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
algo_input.datapoints = total_array

#Run Quantum Kernel-based SVM model
result = run_algorithm(params,algo_input)

print("testing success ratio: ", result['test_success_ratio'])
print("predicted labels:", result['predicted_labels'])


#Set SVM_RBF_Kernel parameter JSON string
params = {
    'problem': {'name': 'svm_classification'},
    'backend': {'name': 'local_qasm_simulator', 'shots':1000},
    'algorithm': {
        'name': 'SVM_RBF_Kernel', #SVM_QKernel, SVM_Variational
        'print_info' : True
    }
}

algo_input = get_input_instance('SVMInput')

algo_input.training_dataset = training_input
algo_input.test_dataset = test_input
algo_input.datapoints = total_array

#Run Classical Radial Basis Function Kernel SVM model
result = run_algorithm(params,algo_input)

print("testing success ratio: ", result['test_success_ratio'])
print("predicted labels:", result['predicted_labels'])
```