Introduction to Quantum Computing Final Project: Quantum Machine Learning

Jacob Emerson, Ourania Glezakou-Elbert, Elif Kozanoglu

1 Introduction

Quantum computers have long presented the potential to complete problems that would otherwise require immense computational power in classical computing. Though a generalized quantum computer is far in the future, mixed classical and quantum methods can provide efficient approaches to particular problems that are suited to the logic of quantum systems. One such case is the kernel method in machine learning.

In classical machine learning, when presented with a complicated training data set and labels $\mathcal{X} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, it may not be linearly separable, where you can draw a hyperplane separating the different labeled values. One way to get around this is to use a feature map where you send the data into higher dimensional spaces, and more aspects of the shape can be used to make the data linearly separable in feature space.

The kernel $k(\mathbf{x}, \mathbf{x}') := \langle \phi(\mathbf{x}), \phi(\mathbf{x}') \rangle$ for $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$ is the inner product between the feature maps and is used to construct the decision function $f(x) = \sum_{i=1}^{N} \alpha_i y_i k(x_i, x) + b$, which predicts the label of some $\mathbf{x}$ which may not have been in the original training data. However, these higher dimensional inner products are very computationally costly at times, and this leaves room for quantum computers to come in and provide an advantage. If we map each individual feature map $\phi(\mathbf{x})$ to a unitary gate $\mathcal{U}_{\phi(\mathbf{x})}$ then $|\langle 0|\mathcal{U}^{\dagger}_{\phi(\mathbf{y})}\mathcal{U}_{\phi(\mathbf{x})}|0\rangle|^2$ can be calculated with a simple quantum circuit with n qubits, where $n = \dim(\mathrm{Im}(U_\phi))$.

Quantum kernel methods function in one of two ways: through implicit methods and explicit methods (Fig. 1). Implicit kernel methods simply leverage quantum computers to evaluate a kernel for a given choice of feature map. This kernel is then fed into a classical support vector machine (SVM) that is used for prediction. Explicit kernel methods incorporate the model implementation in the quantum circuit. In addition to the encoding circuit, $\mathcal{U}_{\phi(\mathbf{x})}$, the circuit includes a variational circuit, $W(\theta)$. This project considers implementations of both forms and compares them to their classical equivalents.

2 Datasets

This report makes use of the n-sphere datasets to test the performance of these algorithms. For a reasonable training data size, we chose n=5 with using 50 total data points for the implicit algorithm, and 100 points for the explicit algorithm. By the generation of the dataset we see that we have one 5-sphere with radius 0.5

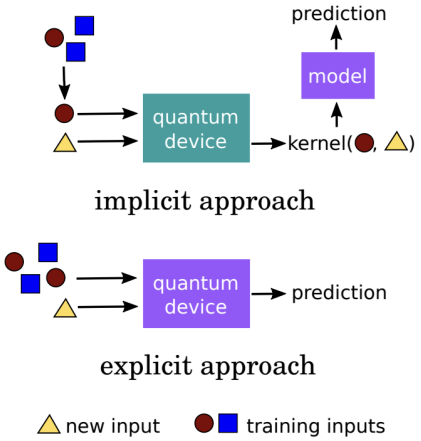

Figure 1: Figure taken from [8]. A diagram representing the two possible quantum kernel method approaches. The implicit approach trains the model using kernels calculated by a quantum circuit whereas the explicit approach trains a quantum model to perform label predictions or regressions.

and variance 0.2, so the data points vary between 0.3-0.7 distance from the origin, and then a second 5-sphere with radius 1 with variance 0.2 so the data points vary between 0.8-1.2 distance from the origin. From this we construct a decision boundary at 0.75 and break down the inner points into class 0 and the outer points into class 1. This is now what we would like to apply our machine learning algorithms to classify.

In order to ensure that the data can be expressed in the circuit, all data is normalized by the largest norm of any point in the set, ensuring that the largest possible value seen by the circuit is 1. Training and test data should be generated at the same time and normalized by the same value before being portioned off into separate sets.

3 Implicit Quantum Kernel Method

Classical kernel methods used for SVMs are already classically optimal. Quantum kernel methods use a quantum circuit design to calculate the kernel to be used in a classical SVM setting. These methods provide a quantum advantage when the kernel cannot be efficiently computed on a classical computer [4]. In our implementation of this algorithm, we calculate the kernel using a quantum circuit and apply SVM classically using this kernel. To start, we first map the data points we're looking to calculate the kernel of $\mathbf{x}$ and $\mathbf{y}$ to a radius or magnitude, which in this case is our feature map. The reason why this is sufficient to determine data points and their classification is because we are working in a spherically symmetric dataset and our decision boundary is a 5-sphere of radius 0.75.

One common method for implementing the encoding of classical data into quantum states is angle encoding, wherein features are encoded through the application of a rotation proportional to the classical data value. For data with multiple features, each feature corresponds to a different qubit in the circuit [1, 2, 7, 8]. The rotations may include some set of tunable parameters, ϵ_i, that varies the weight of each feature of the data. In the case of the n-sphere data, we map the magnitudes of each data point to rotation unitary matrices about the Y-axis. From this we can draw a circuit to estimate the kernel between every training data point through measuring the probability of a $|0\rangle$ outcome after applying the unitary corresponding to one point, $\mathbf{x}$, and the Hermitian of the unitary corresponding to another point, $\mathbf{y}$. This gives us a measure of how similar these data points are, and thus how likely they are to be of the same label.

3.1 Circuit Description

The circuit we chose for the encoding applies a y axis rotation to each qubit in the circuit initialized at $|0\rangle$ based on the normalized magnitude of each coordinate of $\mathbf{x} \in \mathcal{X}$, where the rotation can go from 0 to π which is $\mathcal{U}_{\phi(\mathbf{x})}$ and then $\mathcal{U}^{\dagger}_{\phi(\mathbf{y})}$ is applied which is just the negative of the rotation from each of the magnitudes derived from another point $\mathbf{y} \in \mathcal{X}$ (Fig. 3). The rotation is scaled by a single tunable parameter that was adjusted in order to provide the optimal kernel. This value was around 0.01 (as the actual rotation ends up being halved), and generated a kernel as seen in Figure 2.

The circuit is then measured a large number of times and the percentage that return $|0\rangle$ is the kernel $k(\mathbf{x}, \mathbf{y}) := \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle$. An example of such a circuit for dimension 5 feature space is shown below where $\mathcal{U}_{\phi(\mathbf{x})}$ is the first column of rotation gates, and $\mathcal{U}^{\dagger}_{\phi(\mathbf{y})}$ is the second column.

Listing 1: Quantum Kernel Evaluation function for n-sphere Dataset

```
# takes in ndim x N array
def eval_kernel_nsphere(data, val):
  print(data.shape)
  N = 1
  kernel = np.zeros((data.shape[1], data.shape[1]))
  cq = QuantumRegister(N, 'encoding_qubit')
  cb = ClassicalRegister(N, 'encoding_bit')

  init_circ = QuantumCircuit(cq, cb)

  kernel = np.zeros((data.shape[1], data.shape[1]))
  for i in range(data.shape[1]):
    kernel[i, i] = 1
    norm = np.linalg.norm(data[:, i].T)
```

```
    encoding_circ = QuantumCircuit(cq, cb)

    encoding_circ.ry(norm*np.pi*val, cq[0])

    for j in range(i+1, data.shape[1]):
      norm_prime = np.linalg.norm(data[:,j].T)
      encoding_circ_prime = QuantumCircuit(cq, cb)

      encoding_circ_prime.ry(-np.pi*norm_prime*val, cq[0])

      final_circ = init_circ.compose(encoding_circ)
      final_circ = final_circ.compose(encoding_circ_prime)

      for k in range(N):
        final_circ.measure(cq[k], cb[k])
      final_circ.draw('mpl')

      # measure a bunch to extract K(x', x) by getting prob
         outcome is 0^n
      transpiled_circuit = transpile(final_circ, backend_qasm)
      job = backend_qasm.run(transpiled_circuit, shots = 2**13)
      counts = job.result().get_counts()
      vals = list(counts.values())
      prob = vals[0]/np.sum(vals)

      kernel[i, j] = prob
      kernel[j, i] = prob

  return kernel
```

3.2 Classical Prediction

To actually implement the machine learning model, the calculated kernel is fed into a classical SVM. This project uses the scikit-learn Python library [6] to implement all classical algorithms. To properly train and utilize the SVM, both the kernel of every training point against every other training point is required, as is the kernel of every test data point with every training data point. Such calculations can be somewhat arduous to carry out, particularly when dealing with large data sets. Indeed, this requirement made the evaluation of the implicit quantum kernel method somewhat arduous. While there exist methods such as the Nyström method [2, 3] for extrapolating the kernel between any two points, such methods are beyond the scope of this project.

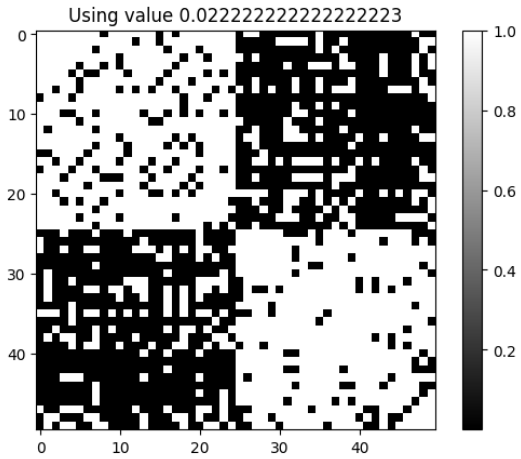

Figure 2: Heat map of the kernel calculated for 50 5D sphere points. Points below the 25th index correspond to those with label 0, i.e. those from the sphere of radius 0.5; those above the 25th index have the label 1 and are from the sphere of radius 1.

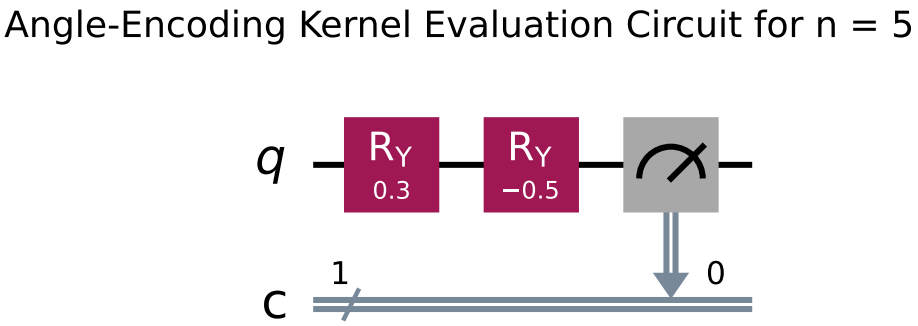

Figure 3: Angle-encoded feature map circuit encoding example vectors x and y.

4 Variational Quantum Algorithm

Rather than allow a classical algorithm to perform the classification, variational quantum circuits, otherwise known as explicit kernel methods, implement a circuit dependent upon some variational parameter, θ, which is updated in order to improve classification accuracy. The circuit, as described previously, consists of an encoding circuit that represents the classical data as a quantum state, and a variational circuit that classifies the data point.

As with the SVM model, the variational quantum algorithm (VQA) optimizes θ by minimizing a loss function, $\mathcal{L}(\theta)$. In most cases, such as in [5], the loss function is chosen to be the mean squared error (MSE),

$$\textbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

.

where n is the number of training data points, y_i is the actual label or value, and $\hat{y}_i$ is the predicted value. This project makes use of this loss function for the implementation of the VQA, based on the work of [5].

In order to efficiently minimize the loss function, gradient-descent is used to optimize the value of θ. Since loss function is convex, its gradient will point along the path of steepest slope. Thus, by adjusting θ to $\theta - \eta\nabla\mathcal{L}$, where η is a tunable parameter known as the learning rate, it is efficiently moved towards a local minimum of the loss function [1]. (In the case where this process accidentally maximizes the loss function, one should instead add the second term.)

Thus, in order to optimize θ, one must calculate the gradient. After each iteration of the model, the gradient is estimated using Spall's simultaneous perturbation stochastic approximation method [1, 9, 10]. To estimate the gradient of the loss function, a random perturbation is generated, Δ. Each partial derivative of the loss function is then estimated to be

$$\frac{\partial\mathcal{L}}{\partial\theta_j} = \frac{\mathcal{L}(\theta + c\Delta) - \mathcal{L}(\theta - c\Delta)}{2c\Delta_j}$$

where c is some small parameter.

Once the gradient drops sufficiently close to zero, the algorithm is considered to have been trained and may be used to predict test data.

4.1 Circuit Description

The actual implementation of the variational circuit, $W(\theta)$, can follow from numerous possible ansatzes. This project follows the ansatz provided in [1], [2], and [7]. In its most general form, the variational circuit applies a rotation θ_j along either $\hat{X}$, $\hat{Y}$, or $\hat{Z}$ on each qubit. It then applies some number of CNOT gates between each encoding qubit in order to allow for more expressibility, which allows for the model to decipher what features of the data are correlated and in what manner depending on the data's label [7]. In order to predict the label, there is some allowance of what metric may be used. For instance, it is perfectly acceptable to either consider the parity of the outcome results and use an even parity to indicate a 0 label and an odd parity to indicate a 1. It is also possible to use the result of the first qubit to determine the label of the data point [7].

This project implements a simple variational circuit where each θ_i in θ corresponds to a rotation about Y of θ_i for each qubit, i. After the rotation, a CNOT gate is

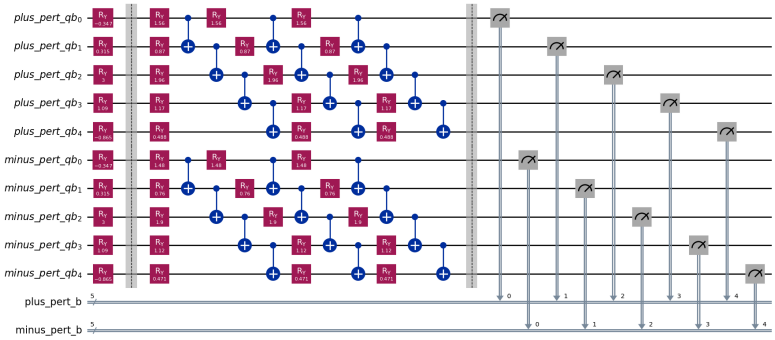

Figure 4: Example of the Variational Quantum Circuit for the 5D N-sphere with the complex encoding circuit. A series of rotations

taken between each qubit and its neighbor. It is possible to apply this circuit several times; this project applies the variational circuit three times to attempt to allow for a bit more expressibility without causing the runtime to get ridiculously inefficient.

Two attempts at the encoding circuit were tested. The first simply made use of the encoding algorithm presented in the implicit quantum kernel method implementation. The second encoding circuit consisted of a rotation of 2π times the classical data value on each qubit. After this, a CNOT gate was applied between each qubit pair, (i, j), alongside a second rotation of 2π times the product of the ith and jth data values (Fig. 4). The label of the data point is taken to be the relative probability of measuring $|0\rangle$ on qubit 1, with 0 probability corresponding to the label 1, and 1 corresponding to the label 0.

In the training process, the circuit consisted of $2N$ qubits, with N being the number of features of each data point. This allows for the concurrent evaluations of $W(\theta + c\Delta)$ and $W(\theta - c\Delta)$ to estimate the gradient. The initial value of θ was taken to be a random N-dimensional vector with components ranging from 0 to 1.

The algorithm requires a few hyperparameters, that being the learning rate, which should be relatively small, no more than 0.1, the scaling of the perturbation for calculation of the gradient, likewise small, and the cutoff for the loss function improvement, which is also a small value. For the circuit, these values were chosen to be 0.01 for the learning rate, 0.05 for the perturbation scaling, and 0.02 for the cutoff to ensure reasonable convergence of the gradient.

It is also possible to implement an algorithm with a fourth set of parameters to control the weighting of the encoding circuit rotations, however, for simplicity, this project makes use of the two relatively bare-bones encoding methods described above. The actual code of the implementation of this is at the bottom of the attached .ipnb file due to the length of this algorithm.

5 Comparison With Classical Methods

The classical methods we will compare our results to are classical SVM and feed-forward neural networks, implemented using the scikit-learn library [6].

5.1 Implicit Method against the Classical SVM

The following code was used on the n-sphere data to test the standard classical SVM method for datasets like this with the RBF Kernel.

Listing 2: Classical SVM training on concentric circles

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score
import time
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.8, random_state=42
)

svm_model = Pipeline([
    ("scaler", StandardScaler()),
    ("svm", SVC(kernel="rbf", C=10.0, gamma="scale"))
])
start = time.perf_counter()
svm_model.fit(X_train, y_train)
svm_train_time = time.perf_counter() - start

start = time.perf_counter()
y_pred_svm = svm_model.predict(X_test)
svm_test_time = time.perf_counter() - start

svm_acc = accuracy_score(y_test, y_pred_svm)

print("SVM RESULTS")
print("-----------")
print(f"Training time   : {svm_train_time:.4f} s")
print(f"Testing time  : {svm_test_time:.4f} s")
print(f"Test accuracy   : {svm_acc:.3f}")
print(f"# Support vecs  : {svm_model.named_steps['svm'].
    n_support_}")
```

The results from this classical SVM were very fast and effective with an almost perfect test accuracy in under 0.01 seconds for 1000 training data points. As training data size scales the efficiency theoretically should be beaten by a quantum computer, but trying 500,000 training data points and seeing it finish in under 6 seconds doesn't make it seem very achievable. The following were the results for the 1000 training point case.

Listing 3: Output of Classical SVM

```
SVM RESULTS
-----------
Training time   : 0.0044 s
Testing time  : 0.0051 s
Test accuracy   : 0.989
# Support vecs  : [16 26]
```

While the implicit quantum kernel method was able to reproduce a comparable accuracy of 96.7%, this was done after over an hour of evaluating kernels for a 50 point data set. While it is possible that the circuit design was not optimal, and some unnecessary inner-products were certainly calculated, even if the design had been somewhat streamlined, the time it would have taken to run the algorithm would still have been much longer than that of the classical SVM. One reason is the briefly mentioned previously is the fact that the evaluation of the kernel took a considerable amount of time, largely due to the fact that Qiskit is simply a classical simulation of quantum computers. Thus, it is not possible to compare the amount of time that classical and quantum methods take, particularly when the quantum simulations perform an egregious number of matrix multiplications to implement the circuit. In general, quantum kernels may allow for the realizing of complex kernels that might otherwise be intractable in classical computing [1], however, the dataset used in this project did not require any particularly exotic kernels. Thus, there was no real benefit in applying the quantum kernel over the relatively efficient RBF.

Aside from the potential time benefits in bizarre data cases, the selection of an appropriate feature map based on the particularities data is also not quite as necessary when using the quantum kernel. Regardless of the data, angle encoding with some tunable parameter is acceptable for defining the inner product of any two data points, which is all that is needed for the evaluation of the kernel. Explicit knowledge of what form the feature map must take or knowledge of the kernel's form is thus less important for these quantum methods. Thus, the kernel method retains this benefit.

5.2 VQA/Explicit Method against a Feed-Forward Neural Network

The classical counterpart to a VQA would be a Feed-forward Neural Network, where the neural network is analogous to the sequence of iterations that the circuit in the VQA is ran to find the correct parameters, just in this case classical parameters are varied, and gradient descent is just used to minimize a standard loss function. The following code is a classical example of this on 1000 training data points and shows the run-time, accuracy, log loss and other parameters.

Listing 4: Classical Feed-Forward Neural Network

```
from sklearn.metrics import log_loss

# Build model
nn_model = Pipeline([
    ("scaler", StandardScaler()),
    ("mlp", MLPClassifier(
        hidden_layer_sizes=(64, 64),
        activation="relu",
        solver="adam",
        max_iter=2000,
        random_state=42
    ))
])

t0 = time.time()
nn_model.fit(X_train, y_train)
train_time_nn = time.time() - t0


t0 = time.time()
y_pred_nn = nn_model.predict(X_test)
infer_time_nn = time.time() - t0

nn_acc = accuracy_score(y_test, y_pred_nn)

y_prob = nn_model.predict_proba(X_test)
nn_loss = log_loss(y_test, y_prob)

mlp = nn_model.named_steps["mlp"]
n_params = sum(w.size for w in mlp.coefs_) + sum(b.size for b
    in mlp.intercepts_)


print("Neural Network Results")
print("-" * 40)
```

```
print(f"Training time         : {train_time_nn:.4f} s")
print(f"Testing time        : {infer_time_nn:.4f} s")
print(f"Inference / sample    : {infer_time_nn / len(X_test):.2e} s")
print(f"Accuracy              : {nn_acc:.4f}")
print(f"Log loss              : {nn_loss:.4f}")
print(f"Number of parameters : {n_params}")
```

Listing 5: Neural Network Output

```
Neural Network Results
----------------------------------------
Training time         : 0.5481 s
Testing time        : 0.0026 s
Inference / sample    : 3.21e-06 s
Accuracy              : 0.9812
Log loss              : 0.0609
Number of parameters : 4609
```

As can be seen above, the training time was a little longer than the SVM classical SVM because neural networks are a bit more complicated, and work better for more complicated data sets, but compared to the quantum algorithm this is an enormous improvement in every category. The accuracy is 98% and the log loss is 0.06 meaning that the confidence in the results is also very high.

Because of the large hyperparameter space, from the choice of gates for encoding and variational circuits, to the choice of learning rate and perturbation strength, the VQA was difficult to implement and optimize. Ultimately, its prediction accuracy seldom went above 55%, despite following the recommended ansatz, but it typically sat around 40% or lower. The loss function sat around 0.2 in most cases, though it did occasionally drift up to 0.4. This is relatively large, as it means that the average distance from the true label was at least 0.5, placing it right on the decision boundary. This may be because of a poor choice of estimation of the gradient, poor choice of loss function, poor choice of hyperparameters, or poor choice of gates in implementing the variational circuit. We regrettably lack deep enough knowledge of machine learning at this time to better diagnose the issue with the method. Additionally, the model was incredibly slow to train and thus difficult to debug and optimize, often taking at least an hour to run when training on only 100 points.

With regards to the timing, it is an unfair comparison to consider the efficiency of this algorithm against the neural network, as both are run on classical computers. The Qiskit simulation is incredibly inefficient for heavy-duty computational tasks. It is however, more fair to compare the accuracy and loss, and in this case the quantum circuit method lacks a lot of accuracy compared to the classical method, and this is without even considering the physical implementation into the quantum computer and

the extra level of error that would provide. The black-box nature of the variational circuit in combination with the large parameter space and long run times simply limit the maximal accuracy possible for this project.

A final note regarding the choice of encoding circuit for the VQA. The single value encoding function was able to produce a 100% accuracy classification with a final θ of around 0.65 radians for the 2D data set. This accuracy marginally out-competes the neural network accuracy, though it did still require a few minutes to train and classify, and this was only for a total of 300 data points with a 1:2 training to test split.

It should also be noted that running the VQA code for the 5D n-spheres dataset, we have found an accuracy as low as 53.5% for a learning rate of 0.005. A few models with different hyperparameters such as circuit depth and learning rate were tested. Although the accuracy increased with lower learning rates, it did not improve significantly for different hyperparameters. We believe this could be due to how features of the n-spheres dataset were encoded. The encoding through y rotations in the 2D data works well: we differentiate encoded quantum states through theta values on the Bloch sphere, with one degree of freedom, which matches the one degree of freedom the 2D n-spheres dataset has (since classification ultimately only depends on r, the distance of the point from the origin). However, when we move to a higher dimensional space, like 5D, we have more degrees of freedom which means that they should be encoded in an effective way to account for this. As future work, this could point to using an encoding function that takes rotations by squares of the individual data points' components.

6 Conclusion

Given the current limitations to lifetimes, gate fidelity, and scalability of quantum computers, these noisy intermediate-scale quantum (NISQ) algorithms provide the best present application of quantum technology to everyday problems. While this promise was not entirely realized in this project, some hope still remains for quantum computers to aid in the solving of new, difficult problems. Whether this future will ever be realized is a task left up to electrical engineers and physicists. As of now, classical methods provide an extremely quick, accurate, and confident result, while the simulation performed in this project of the quantum kernel method had accurate but horrendously slow results, and the simulation of the VQA struggled to get an accuracy over 50% in addition to its hour-long run-time. These simulations should not be taken as indicative of the potential efficiency offered by NISQ algorithms.

References

[1] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4(4):043001, November 2019.

[2] Rodrigo Coelho, Georg Kruse, and Andreas Rosskopf. Quantum-efficient kernel target alignment, 2025.

[3] Petros Drineas and Michael W. Mahoney. Approximating a gram matrix for improved kernel-based learning. In *Proceedings of the 18th Annual Conference on Learning Theory*, COLT'05, page 323–337, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, March 2019.

[5] Annie E. Paine, Vincent E. Elfving, and Oleksandr Kyriienko. Quantum kernel methods for solving regression problems and differential equations. *Phys. Rev. A*, 107:032428, Mar 2023.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] Qiskit Global Summer School and Amira Abbas. Lecture 5.1 - building a quantum classifier.

[8] Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Phys. Rev. Lett.*, 122:040504, Feb 2019.

[9] James C. Spall. A one-measurement form of simultaneous perturbation stochastic approximation. *Automatica*, 33(1):109–112, 1997.

[10] J.C. Spall. Adaptive stochastic approximation by the simultaneous perturbation method. In *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, volume 4, pages 3872–3879 vol.4, 1998.