Performance of Variational Quantum Classifiers and Support Vector Machines in Binary Classification

To what extent can NISQ-based variational quantum classifiers outperform classical support vector machines in binary classification within higher-dimensional feature spaces?

Kenzo Eugenio Tjandra

Table of Contents

1. Introduction **3**
2. Theoretical Background **4**
2.1 Binary Classification and Machine Learning 4
2.2 Support Vector Machines 4
2.3 Variational Quantum Classifiers 8
3. Hypothesis **11**
4. Methodology **12**
4.1 MNIST Dataset 12
4.2 Variables 15
4.3. Procedure 19
5. Data **21**
6. Analysis **24**
6.1 Interpreting Training Time 24
6.2 Analyzing Training Time 28
6.3 Interpreting Accuracy Metrics 31
6.4 Analyzing Accuracy Metrics 33
7. Conclusion **35**
8. Evaluation **36**
8.1 Strengths of Methodology 36
8.2 Limitations of Methodology 36
8.3 Improvements and Further Research Opportunities 37
9. Bibliography **38**
10. Appendix **43**
10.1 Python Code for SVM Algorithm 43
10.2 Python Code for VQC Algorithm 45
10.3 Matplotlib Code to Generate Accuracy Metric Plots 49
10.4 MNIST Train CSV File 51
10.5 MNIST Test CSV File 52

1. Introduction

In recent years, quantum computing has emerged as a promising technology with the immense potential to revolutionize various fields, including machine learning. With the advent of Noisy Intermediate-Scale Quantum (NISQ) computers (Dargan, 2023), Variational Quantum Classifiers (VQCs) have become increasingly viable as substitutes for already existing, but powerful machine learning models—such as classical Support Vector Machines or SVMs. As VQCs leverage quantum mechanics to perform machine learning tasks, such as classification, at incredible speeds, numerous researchers are eager to delve deeper and demonstrate their quantum supremacy over classical models.

To this day, however, the practicality of NISQ-based VQCs to outperform SVMs in higher-dimensional feature spaces remains uncertain, despite its encouraging results in utilizing quantum computing concepts to more efficiently process and classify data (*paraphrased from ChatGPT* OpenAI, 2023). Quantum properties such as decoherence and noise sensitivity (Nguyen *et al.*, 2016) are a few among many that prevent VQCs from outright outperforming SVMs, which have been extensively studied and optimized over the years.

As a result, this essay aims to investigate how VQCs compare with SVMs when performing binary classification—particularly in incrementally higher dimensions, i.e. features or variables that are used to represent each data point in a dataset. Therefore, ***"To what extent can NISQ-based variational quantum classifiers outperform classical support vector machines in binary classification within higher-dimensional feature spaces?"*** will be answered.

By investigating the extent to which VQCs can (or cannot) outperform SVMs in binary classification, various insights can be gathered into the practical capabilities of quantum machine learning algorithms, in hopes of accelerating the development of powerful and safe artificial intelligence. Furthermore, the results of this study can have significant implications for industries that would, in the near future, heavily rely on machine learning and binary classification. For instance, if VQCs prove to be reliable, they could potentially revolutionize the healthcare industry by providing exponentially faster, life-saving diagnoses (Karabiber *et al.*, 2023). The anticipated outcomes of this study will not only shed light on the potential of VQCs, but also contribute to the advancement of adept quantum ML models for the second quantum revolution.

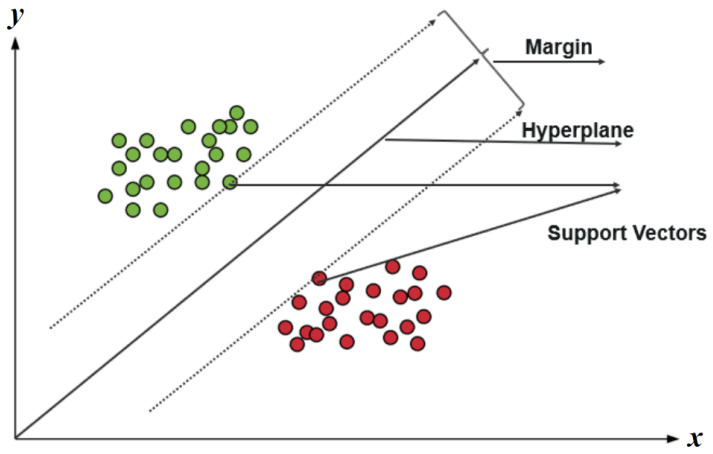
2. Theoretical Background

2.1 Binary Classification and Machine Learning

Binary classification refers to a fundamental task in machine learning, which involves categorizing instances into one of two distinct classes based on their features (Karabiber *et al.*, 2023). Being a form of **supervised learning**, binary classifying algorithms learn from labeled training data to build a model capable of predicting the class label of new, unseen instances.

2.2 Support Vector Machines

Support Vector Machines are a type of machine learning algorithm that is ubiquitously employed for binary classification tasks. These models aim to find the best possible decision boundary that separates two classes in a dataset by computing a **hyperplane** that maximizes the **margin** between the closest data points from distinct classes (Premanand, 2021). The data points that are closest to the decision boundary are called **support vectors** (see Figure 2.1).



▲ **Figure 2.1: Linear Support Vector Machine** (*adapted from* Waseem, 2023)

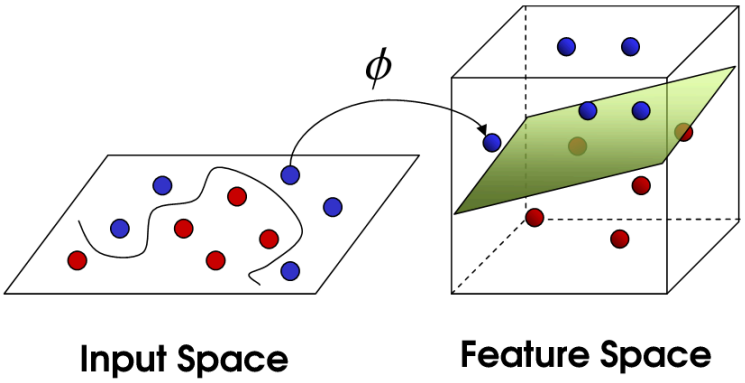
2.2.1 Nonlinear SVMs

Nonlinear support vector machines are an extension of the traditional linear SVMs that can handle non-linearly separable data. While linear SVMs find a linear decision boundary, nonlinear SVMs employ a technique called the **kernel trick** to implicitly map out the input data into a higher-dimensional feature space, where linear separation is possible (Kumar, 2020).

Consequently, the kernel trick enables nonlinear SVMs to effectively process complex data distributions without explicitly computing the coordinates of the transformed feature space—being more computationally efficient. A **kernel function** is therefore employed to compute the dot product between pairs of data points in the original input space. Equation 2.1 defines the kernel function $K(x, z)$ as the dot product between the features aggregated from $\phi(x)^T$ and $\phi(z)$, and all being transformed within a higher-dimensional feature space ϕ (Patel *et al.*, 2016).

$$K(x, z) = \phi(x)^T \cdot \phi(z)$$

▲ **Equation 2.1: Definition of a Kernel Function**

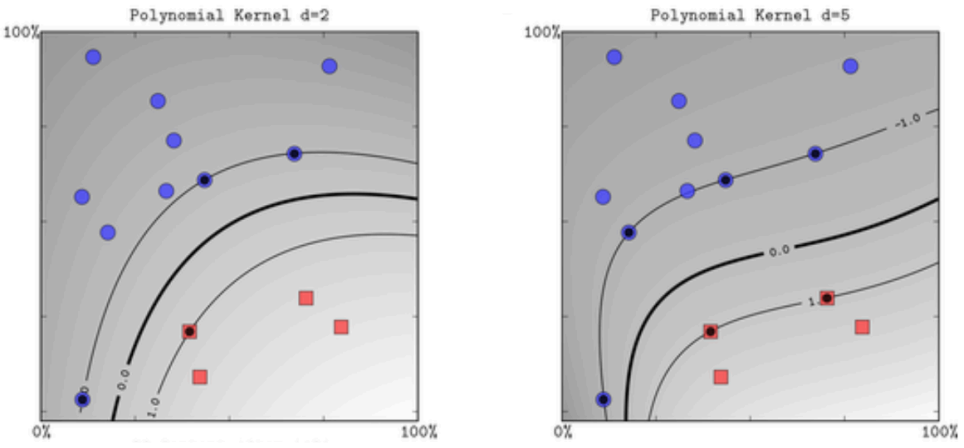


▲ **Figure 2.2: Transformation of Data Points x into Feature Space** ϕ (Wilimitis, 2018)

During training, nonlinear SVMs optimize the hyperplane parameters—such as **weights** and **biases**—and other kernel-specific parameters (refer to subsection 2.2.3) to maximize the margin between closest data points.

2.2.2 Polynomial Kernels

Polynomial kernels will be the kernel function of choice to be used in the investigation. To map optimize data in a higher-dimensional feature space, these kernels raise the transposed features x^T to various powers up to degree d (Sidharth, 2022). Figure 2.3 delineates the graphical contour of polynomial kernel functions at $d = 2$ and $d = 5$ respectively.



▲ **Figure 2.3: Linear Support Vector Machine** (Ben-Hur *et al.*, 2008)

$$K_{poly}(x, z) = (\alpha x^T \cdot z + c)^d$$

▲ Equation 2.2: Polynomial Kernel Function

When applied, the polynomial kernel computes the dot product between features x^T and z, all the while scaled by regularization coefficient α and constant c to provide control over the kernel's behavior (Berwick, n.d.). After then, d designates the extent of transformation towards the data points into higher dimensions. Polynomial kernels will be particularly useful in the experiment as its computational efficiency can potentially scale only **linearly** with the number of features.

2.2.3 Decision Function

$$\hat{y} = sign(\sum_{i=1}^{n_{sv}} w_i y_i K_{poly}(x_i, z) + b)$$

$$sign(z) = \{+1, if\ z \geq 0;\ -1, if\ z < 0\}$$

▲ Equation 2.3: Decision Function for Polynomial Kernels

During prediction, SVM with polynomial kernels predict novel data labels $\hat{y}$ via a decision function as denoted by Equation 2.3. From $i = 1$ to the number of support vectors n_{sv}, the function will iterate the products of weights w_i corresponding class labels y_i and $K_{poly}(x_i, z)$, where x_i is the corresponding feature and z is the input feature from the prediction dataset (Eq. 2.2), with the bias term b (Jäger, 20233). Subsequently, $\hat{y}$ will be classified into one of two classes, i.e. $(+1)$ or (-1) in the context of binary classification.

2.3 Variational Quantum Classifiers

Variational Quantum Classifiers are machine learning models that leverage the mechanics of quantum computing to perform classification. VQCs encode data points in quantum bits or qubits, such that they can exist in **superposition** and represent multiple states or labels simultaneously—enabling them to process and analyze data in parallel. By adjusting **trainable parameters**, encoded in **parameterized quantum circuits**, VQCs can exploit **entanglement**—where multiple qubits are "entangled" enabling measurements on one to convey information about others—and iteratively optimize the algorithm in training (Fuster, 2019).

2.3.1 Basis Encoding

Basis encoding refers to a method of representing classical data as quantum states in a quantum circuit—allowing them to be analyzed by quantum algorithms like VQCs. **Quantum gates** are primarily utilized to both initially induce a classical bit into superposition and manipulate qubits by performing operations, e.g. rotations, phase shifts, and entanglement (Ming *et al.*, 2023).

$$|\Psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$$

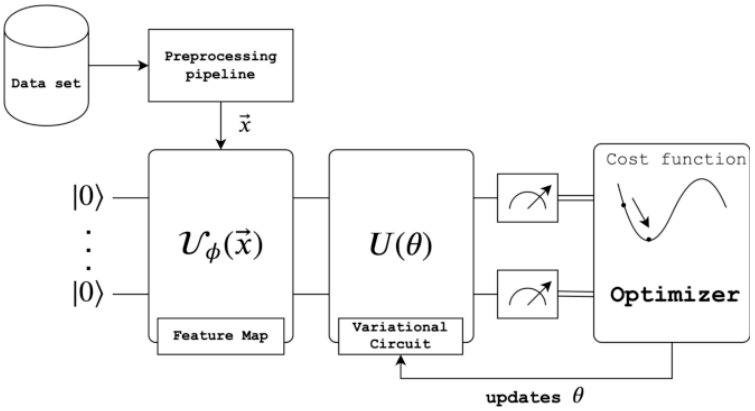
▲ Equation 2.4: Mathematical Representation of Qubits under Superposition

When basis encoding is performed on a classical bit via a **Hadamard gate**, they are represented as a probability distribution of state $|0\rangle$ and $|1\rangle$, as shown by Equation 3 (Hui, 2018). In binary classification, the probability at which some set of features x corresponds to states $|0\rangle$ and $|1\rangle$ can be interpreted as $|\alpha_0|^2$ and $|\alpha_1|^2$ respectively, where $|\alpha_0|^2 + |\alpha_1|^2 = 1$ and thus normalized. Both α_0 and α_1 make up a class's **probability amplitude** to capture complex relationships within data.

▲ **Figure 2.4: Basis Encoding with Hadamard Superposition** (IBM, 2023)

Figure 2.4 illustrates the process of basis encoding and other operations towards qubits q_0 and q_1 within a sequence of quantum gates, i.e. a **quantum circuit**.

2.3.2 Parameterized Quantum Circuits

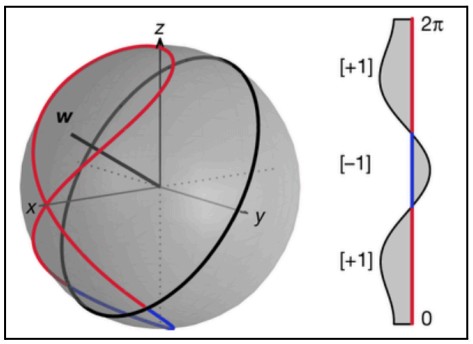


▲ **Figure 2.5: Architecture of PQCs** (Osodo, 2021)

Once superposition has been established, VQCs employ Parameterized Quantum Circuits or PQCs that perform various unitary operations U with respect to trainable parameters θ on qubits, i.e. the "variational circuit" or **ansatz** $U(\theta)$. These quantum data points are then collapsed to be evaluated classically and compared with the actual classes ($|+1\rangle$ or $|-1\rangle$ in binary classification). Finally, the PQC's parameters θ, which describe how the algorithm predicts novel data, are then adjusted iteratively which enable the VQC to simultaneously improve the prediction capability by means of minimizing some **cost function** (Maheshwari *et al.*, 2022).

2.3.3 Quantum Feature Maps

Unlike the kernel trick, basis encoded qubits are calibrated into quantum feature maps that enable the VQC to effectively exploit data points without necessarily raising dimensionality. In these feature maps, qubits are represented by the **Bloch Sphere** (see Figure 2.6). The poles are states $|0\rangle$ and $|1\rangle$ while located orthogonally and within the equator are classes $|+1\rangle$ and $|-1\rangle$. Based on a combination of the state spaces of entangled qubits within a feature map, the VQC can gauge educated predictions upon the respective classes.



▲ **Figure 2.6: Bloch Sphere** (Havlicek, 2020)

2.3.4 Types of Ansatzes

As established by Figure 2.5, the ansatz acts as the starting point for the VQC to classify data before optimization. Therefore, choosing an appropriate ansatz is crucial to ensure the performance of VQCs. In Qiskit, some of the most popular types of ansatzes are as follows:

- `RealAmplitudes`: simple; uses real-valued rather than complex-valued parameters.
- `EfficientSU2`: compact; combines single-qubit rotations and entangling gates.
- `RYRZ`: efficient; consists of alternating layers to represent many quantum states at once.

(Qiskit, 2023)

For binary classification, the `EfficientSU2` ansatz may be best for binary classification as it is most computationally efficient when describing qubits in two different states. It is also more suitable for problems with lower resources and dimensional complexity (Qiskit, 2023).

3. Hypothesis

NISQ-based VQCs will theoretically outperform SVMs in classification tasks especially as the number of features within a dataset becomes more dimensionally expansive, albeit only in specific aspects of the experiment. The term "performance" can be broadly categorized into two distinct aspects: "time efficiency" (1) and "metrics of accuracy" (2).

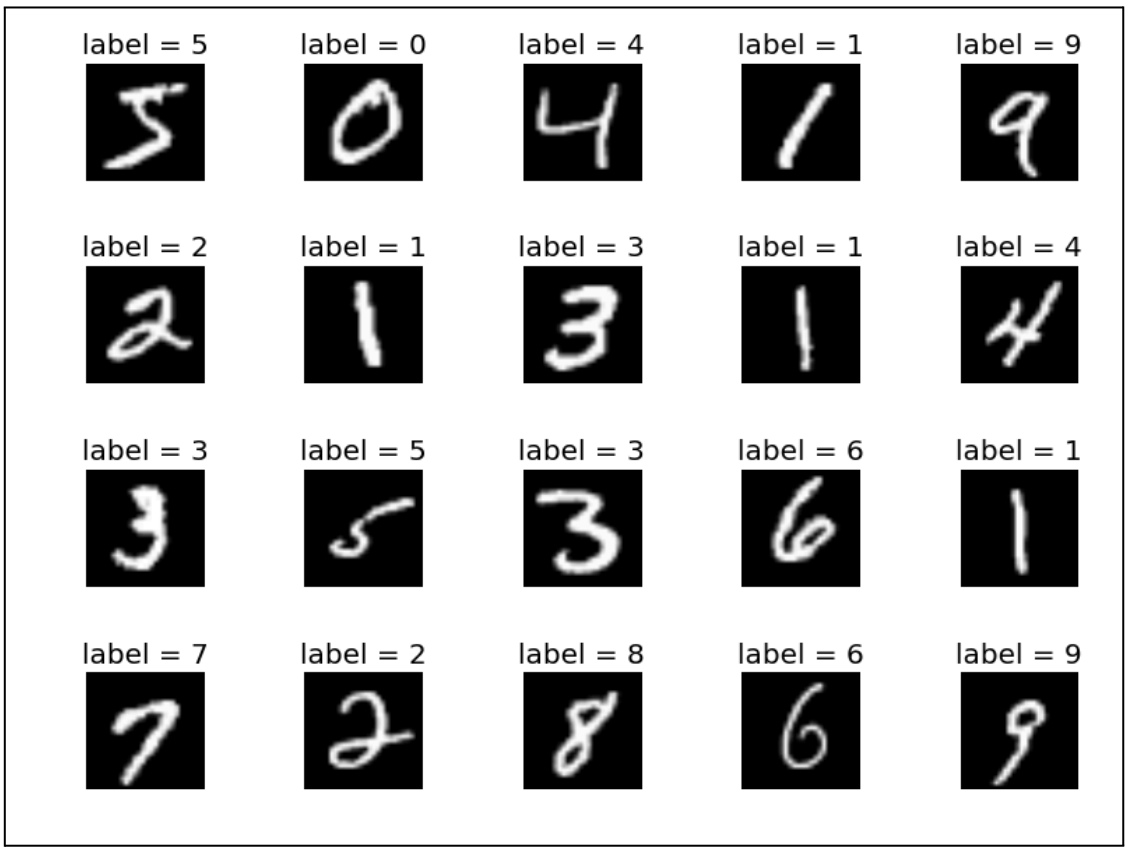
(1) VQCs and their ability to iteratively update trainable parameters in real-time enables them to be highly time efficient, where training time can potentially be **constant** (as qubits are processed in parallel) despite increasingly high dimensions. Meanwhile, SVMs with a polynomial kernel may not be as time efficient with added features where the training time would increase **linearly**, as the kernel computations of pairwise dot products between features in the training dataset is directly proportional to the number of features itself (see Eq. 2.2).

(2) VQCs would however **not classify the data as accurately** as SVMs given their current limitations, contrasting with the robust and well-established performance of classical models—as NISQ-based VQCs are susceptible to noise and decoherence. Furthermore, the hardware limitations of NISQ computers, particularly the low capacity of qubit count, can potentially **hinder the scalability** of VQCs in terms of achieving high accuracy, especially when tasked with classifying classes that possess features that are more indistinguishable.

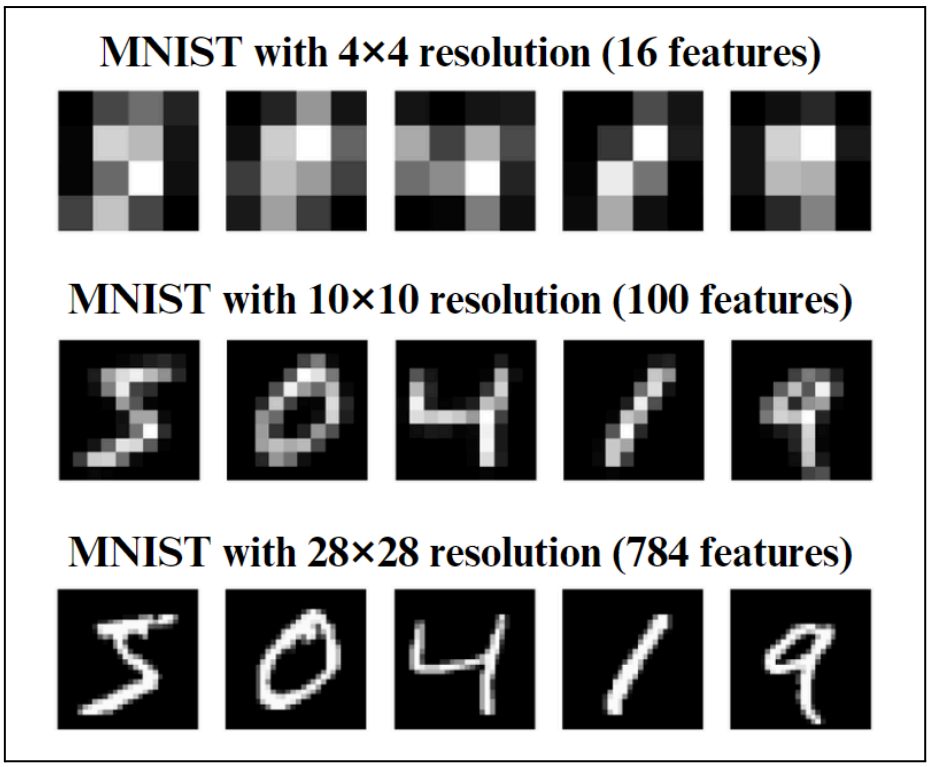
4. Methodology

4.1 MNIST Dataset

MNIST is a collection of handwritten digits ranging from 0 to 9 (TensorFlow, 2023). The training dataset consists of 60,000 data points each with a 28×28 image resolution, while the testing dataset possesses 10,000 images with similar characteristics but unique features. Each pixel has a grayscale integer value of 0 to 255. It is an effective dataset for the experiment as it has a **capacity of 784 distinct features** (or pixels) per image. To manipulate the number of features, the images will be resized into different resolutions.



▲ **Figure 4.1: MNIST Training Dataset Sample Labels** (Corochann, 2021)



▲ **Figure 4.2: Resized MNIST Images** (*adapted from* Matplotlib)

However, since MNIST possesses 10 classes, the training and testing datasets must be truncated to contain only two differing labels. For this experiment, a total of **3 trials** will be conducted with class differentiations based on varying levels of visual discernibility: rudimentary ("0" vs "1"), intermediate ("0" vs "8"), and advanced ("4" vs "9"). Visual discernibility is determined by the relative level of dissimilarity in **feature matrices**.

Table 4.1: Number of Data Points for Tested Classes in Training and Testing Datasets

Trial 1 (“0” vs “1”)			**Trial 2 (“0” vs “8”)**			**Trial 3 (“4” vs “9”)**		
Class	**Training**	**Testing**	**Class**	**Training**	**Testing**	**Class**	**Training**	**Testing**
“0”	5,923	980	**“0”**	5,923	980	**“4”**	5,842	982
“1”	6,742	1,135	**“8”**	5,851	974	**“9”**	5,949	1,009
Total	12,665	2,115	**Total**	11,774	1,954	**Total**	11,791	1,991

Equations 4.1, 4.2, and 4.3 aggregates the individual features of a sample data point—resized to 8×8—from the MNIST dataset and for the aforementioned classes, into a feature matrix where each element represents a grayscale integer from 0 (empty) to 255 (full).

$$x_{(0)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 6 & 6 & 0 & 0 \\ 0 & 0 & 0 & 23 & 188 & 168 & 9 & 0 \\ 0 & 0 & 23 & 209 & 255 & 202 & 70 & 0 \\ 0 & 6 & 158 & 135 & 27 & 96 & 149 & 1 \\ 0 & 40 & 188 & 9 & 5 & 120 & 115 & 1 \\ 0 & 49 & 178 & 54 & 137 & 120 & 10 & 0 \\ 0 & 25 & 213 & 213 & 84 & 7 & 0 & 0 \\ 0 & 0 & 9 & 9 & 1 & 0 & 0 & 0 \end{bmatrix} \quad x_{(1)} = \begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 162 & 49 & 0 & 0 & 0 \\ 0 & 0 & 2 & 229 & 158 & 1 & 0 & 0 \\ 0 & 0 & 1 & 197 & 250 & 2 & 0 & 0 \\ 0 & 0 & 1 & 196 & 235 & 2 & 0 & 0 \\ 0 & 0 & 1 & 192 & 213 & 1 & 0 & 0 \\ 0 & 0 & 0 & 133 & 255 & 2 & 0 & 0 \\ 0 & 0 & 0 & 13 & 35 & 0 & 0 & 0 \end{bmatrix}$$

▲ Equations 4.1: Feature Matrices for Sample Labels of Rudimentary Discernibility “0” vs “1”

$$x_{(0)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 6 & 6 & 0 & 0 \\ 0 & 0 & 0 & 23 & 188 & 168 & 9 & 0 \\ 0 & 0 & 23 & 209 & 255 & 202 & 70 & 0 \\ 0 & 6 & 158 & 135 & 27 & 96 & 149 & 1 \\ 0 & 40 & 188 & 9 & 5 & 120 & 115 & 1 \\ 0 & 49 & 178 & 54 & 137 & 120 & 10 & 0 \\ 0 & 25 & 213 & 213 & 84 & 7 & 0 & 0 \\ 0 & 0 & 9 & 9 & 1 & 0 & 0 & 0 \end{bmatrix} \quad x_{(8)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 14 & 32 & 73 & 50 & 0 \\ 0 & 0 & 17 & 194 & 181 & 247 & 48 & 0 \\ 0 & 0 & 17 & 223 & 207 & 121 & 4 & 0 \\ 0 & 0 & 36 & 255 & 191 & 9 & 0 & 0 \\ 0 & 9 & 175 & 167 & 131 & 1 & 0 & 0 \\ 0 & 15 & 215 & 183 & 29 & 0 & 0 & 0 \\ 0 & 1 & 32 & 18 & 0 & 0 & 0 & 0 \end{bmatrix}$$

▲ Equations 4.2: Feature Matrices for Sample Labels of Intermediate Discernibility “0” vs “8”

$$x_{(4)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 1 & 0 & 27 & 144 & 7 \\ 0 & 0 & 1 & 90 & 27 & 128 & 98 & 1 \\ 0 & 0 & 38 & 195 & 68 & 186 & 17 & 0 \\ 0 & 28 & 202 & 232 & 255 & 104 & 1 & 0 \\ 0 & 19 & 35 & 154 & 124 & 23 & 0 & 0 \\ 0 & 0 & 27 & 149 & 9 & 0 & 0 & 0 \\ 0 & 0 & 1 & 6 & 0 & 0 & 0 & 0 \end{bmatrix} \quad x_{(9)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 26 & 12 & 0 & 0 \\ 0 & 0 & 3 & 133 & 240 & 188 & 4 & 0 \\ 0 & 0 & 13 & 240 & 158 & 143 & 4 & 0 \\ 0 & 0 & 4 & 131 & 255 & 15 & 0 & 0 \\ 0 & 0 & 2 & 112 & 136 & 2 & 0 & 0 \\ 0 & 0 & 52 & 195 & 16 & 0 & 0 & 0 \\ 0 & 5 & 132 & 43 & 0 & 0 & 0 & 0 \end{bmatrix}$$

▲ Equations 4.3: Feature Matrices for Sample Labels of Advanced Discernibility “4” vs “9”

4.2 Variables

4.2.1 Independent Variable

Number of features is the independent variable in the experiment. To manipulate it, the image resolution will be varied from 4×4 (with 16 features) to 28×28 (with 784 features) at increments of 2×2. The number of features is varied exponentially to maximize range of values while minimizing instances of measurements—which can be unnecessary.

4.2.2 Dependent Variables

The **performance** of each classification model will be ascertained via "time efficiency" and "accuracy".

I. Time Efficiency

- **Training time** (in seconds) measures the duration it takes for the classification model to be trained on the training dataset, indicating how time efficient the models are in analyzing and processing the training data.

II. Metrics of Accuracy

- **Accuracy** (out of 1) quantifies the proportion of correctly classified instances out of the total instances in the test dataset, denoting how well the classification model performs in correctly predicting the labels.

$$Accuracy = \frac{Number\ of\ Correctly\ Classified\ Instances}{Total\ Number\ of\ Instances}$$

▲ Equation 4.4: Accuracy of a Binary Classifier

- **Precision** (out of 1) denotes the ratio of correctly predicted positive instances to all instances that were predicted correctly with respect to its class, as true positives; it is fundamentally the accuracy of positive predictions, calibrating each model's ability to distinguish true from false positives.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

▲ Equation 4.5: Precision of a Binary Classifier

- **Recall** (out of 1) or true positive rate calculates the ratio of correctly predicted positive instances to all actual positive instances, indicating each model's ability to correctly capture all instances of a certain class.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

▲ Equation 4.6: Recall Value of a Binary Classifier

- **F1 Score** (out of 1) is the harmonic mean of precision and recall; it provides a balanced measure that considers both false positives and false negatives, enabling it to be applicable in assessing each model's overall performance.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

▲ Equation 4.7: F1 Score of a Binary Classifier

These dependent variables will be measured with Python's `time` and `scikit-learn` packages.

4.2.3 Controlled Variables

Table 4.2: Universal Controlled Variables

Controlled Variable	How to Control	Specifications
Computer and Operating System	The classifier algorithms will be run on an MSI laptop with the Windows 11 Operating System.	**Processor:** 13th Gen Intel(R) Core(TM) i7-13620H 2.40 GHz **Installed RAM:** 16.0 GB **System Type:** 64-bit operating system, x64-based processor **OS Edition:** Windows 11 **OS Version:** 22H2
Programming Language and Shared Package Versions	The experiment will use Python 3.11 as the programming language, and libraries and packages will be kept at the same version throughout.	**Python Version:** 3.11.4 **Scikit Learn Version:** 1.3.0 **Matplotlib Version:** 3.7.0 **Seaborn Version:** 0.12.2 **NumPy Version:** 1.25.2 **Pandas Version:** 2.0.3
Integrated Development Environment	JetBrain's PyCharm integrated development environment will be utilized for algorithm development and execution.	**IDE:** PyCharm 2022.2 **Interpreter:** Python 3.11.4
Dataset and Compared Classes	The MNIST dataset will be used for training and testing, specifically focusing on three distinct class comparisons for both classifiers.	**Dataset:** MNIST **Trial 1:** "0" vs "1" **Trial 2:** "0" vs "8" **Trial 3:** "4" vs "9"
Quantity of Data Points	The quantity of data points in the training and testing datasets are kept constant (see Table 4.1).	**Dataset:** MNIST **Trial 1 Training:** 12,665 **Trial 1 Testing:** 2,115 **Trial 2 Training:** 11,774 **Trial 2 Testing:** 1,954 **Trial 3 Training:** 11,791 **Trial 3 Testing:** 1,991

Table 4.3: Controlled Variables for SVM

Controlled Variable	How to Control	Specifications
Scikit Learn Version	The SVM instance will be created by Scikit Learn, and therefore it is always kept at the same version.	**Scikit Learn Version:** 1.3.0
Kernel Type	The SVM will utilize the same polynomial kernel type for all measurement instances and trials.	**Kernel Type:** Polynomial
Hyperparameters	SVM Hyperparameters are kept the same throughout all measurement instances and trials.	**Regularization Parameter (C):** 1.0 **param_grid (degree):** 2.0–5.0

Table 4.4: Controlled Variables for VQC

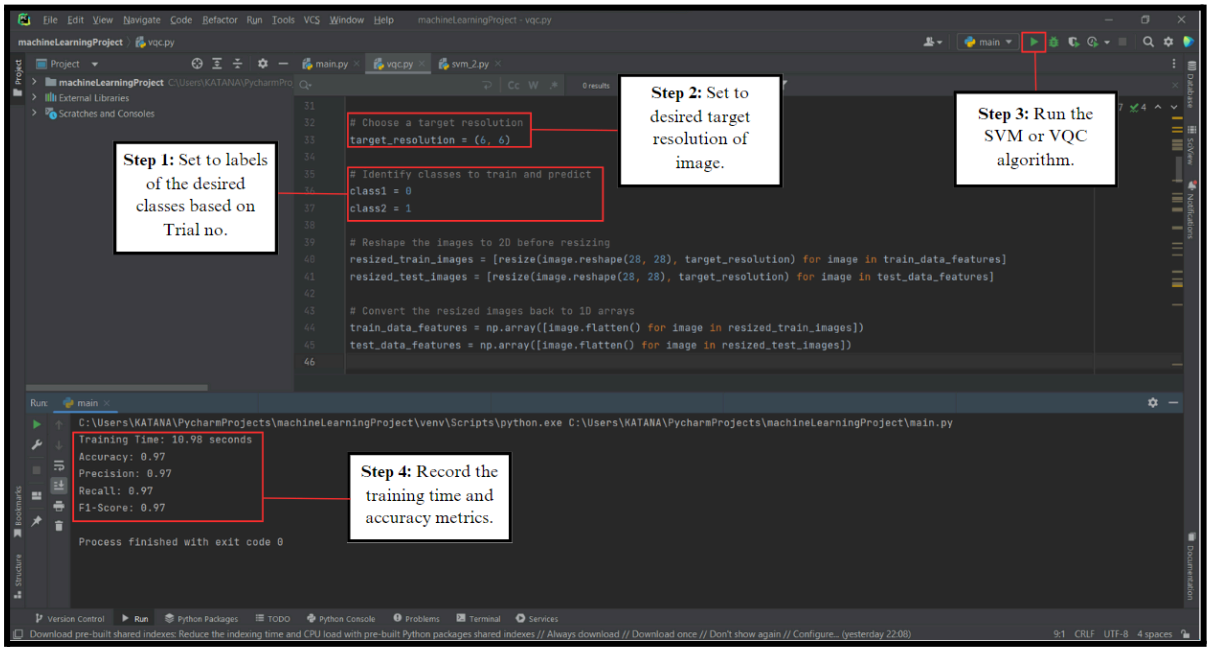
Controlled Variable	How to Control	Specifications
Qiskit Version	The VQC instance will be created with Qiskit, and therefore it is always kept at the same version.	**Qiskit Version:** 0.44.0
Encoding Method	Data points are encoded into quantum states using basis encoding.	**Method:** Basis encoding
Normalization Function	The normalization function applied to the data points will be consistent at range 0–1 (see Appendix 10.2).	**Normalization Function:** normalize(arr, max_val, n)
Feature Map	The quantum feature map applied to the encoded data points is the ZZFeatureMap.	**Type:** ZZFeatureMap
Ansatz	The parameterized quantum circuit (PQC) used as the ansatz in the VQC is an EfficientSU2 circuit.	**Type:** EfficientSU2
Hyperparameters	The VQC hyperparameters are kept constant for all measurement instances and trials.	**n_components (TruncatedSVD):** 10 **n_components (TSNE):** 2 **reps (ZZFeatureMap):** 2 **maxiter (COBYLA):** 500 **shots (QuantumInstance):** 1024 **seed:** 12345 **backend:** qasm_simulator

4.3 Procedure

The experiment requires the MNIST datasets in CSV format: `'mnist_train.csv'` (see Appendix 10.4) & `'mnist_test.csv'` (see Appendix 10.5), the Python code for the respective classification algorithms: `'svm.py'` (see Appendix 10.1) & `'vqc.py'` (see Appendix 10.2), run with the PyCharm IDE.

4.3.1 Data Collection

1. For respective trials, set `class_1` and `class_2` to their appointed labels based on Table 4.1 (e.g. Trial 1 will have `class_1 = 0` and `class_2 = 1`).
2. Vary the image resolution 13 times from 4×4 to 28×28 at increments of 2×2 by changing `target_resolution()`.
3. For each image resolution, run the algorithm once for SVM (`'svm.py'`) and another for VQC (`'vqc.py'`).
4. Record the training time and accuracy metrics shown on PyCharm's console view.
5. Repeat Steps 1 to 4 two more times for Trial 2 ("0" vs "8") and Trial 3 ("4" vs "9").



▲ Figure 4.3: Annotated Experimental Procedure for Data Collection in PyCharm IDE

4.3.2 Data Analysis for Training Time

1. Using a graphing software (such as Logger Pro 3), plot the training times of the SVM (T_{SVM}) and VQC (T_{VQC}) (y-axis) against the number of features n (x-axis) for respective trials (different graphs for each trial).
2. Apply a linear fit each for the SVM and VQC training times, i.e. (different linear functions for each classifier), repeating for each respective trial.
    - In Logger Pro 3, select "Analyze" then "Curve Fit" from the toolbar, choosing $y = mx + b$ as the General Equation, and press "Try Fit" and finally "OK".
3. Compute the gradient of the SVM and VQC linear functions (see Section 3: Hypothesis), which represents the change in training time per increase in one feature, for each respective trial by taking the quotient of ΔT from Δn.

$$gradient(T) = \frac{\Delta T}{\Delta n}$$

▲ Equation 4.7: Gradient of Training Time vs Number of Features Linear Function

- The gradient is automatically computed when creating a linear fit with Logger Pro 3, indicated by the coefficient m.

4.3.3 Data Analysis for Accuracy Metrics

1. Using a graphing software (such as Matplotlib), plot the accuracy, precision, recall, and F1 score of the SVM and VQC (y-axis) against the number of features n (x-axis) for respective trials (different plots for each classifier, different graphs for each trial).
2. Create a line chart that connects individual data points for each plot (6 plots in total).
    - Refer to the sample code to construct the line charts in Matplotlib (see Appendix 10.3).

5. Data

Table 5.1: SVM vs VQC Training Time and Accuracy Metrics for Classes “0” vs “1”

Resolution	Number of Features	Accuracy		Precision		Recall		F1 Score		Training Time (in seconds)	
		(out of 1)									
		SVM	VQC	SVM	VQC	SVM	VQC	SVM	VQC	SVM	VQC
4×4	16	0.98	0.65	0.98	0.59	0.98	1.00	0.98	0.74	10.54	91.79
6×6	36	0.98	0.97	0.98	0.96	0.98	0.98	0.98	0.97	6.50	82.74
8×8	64	0.99	0.99	0.99	0.98	0.99	1.00	0.99	0.99	16.47	70.71
10×10	100	0.99	1.00	1.00	1.00	0.99	1.00	0.99	1.00	23.63	62.79
12×12	144	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	30.74	69.66
14×14	196	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	40.57	62.29
16×16	256	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	50.81	67.86
18×18	324	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	61.55	70.29
20×20	400	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	62.41	62.43
22×22	484	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	81.78	58.68
24×24	576	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	91.06	53.46
26×26	676	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	119.33	58.33
28×28	784	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	148.51	59.57

Table 5.2: SVM vs VQC Training Time and Accuracy Metrics for Classes “0” vs “8”

Resolution	Number of Features	Accuracy		Precision		Recall		F1 Score		Training Time (in seconds)	
		(out of 1)									
		SVM	VQC	SVM	VQC	SVM	VQC	SVM	VQC	SVM	VQC
4×4	16	0.89	0.49	0.89	0.49	0.89	0.76	0.89	0.60	21.08	69.00
6×6	36	0.95	0.94	0.95	0.94	0.95	0.94	0.95	0.94	9.70	80.64
8×8	64	0.96	0.93	0.96	0.84	0.96	0.92	0.96	0.93	25.96	99.81
10×10	100	0.97	0.94	0.97	1.00	0.97	0.88	0.97	0.94	38.06	73.60
12×12	144	0.97	0.92	0.97	0.98	0.97	0.86	0.97	0.91	49.11	79.95
14×14	196	0.97	0.93	0.98	1.00	0.97	0.90	0.97	0.95	66.59	68.94
16×16	256	0.98	0.93	0.98	1.00	0.98	0.92	0.98	0.96	83.18	67.97
18×18	324	0.98	0.93	0.98	1.00	0.98	0.86	0.98	0.92	99.35	57.70
20×20	400	0.98	0.96	0.98	1.00	0.98	0.92	0.98	0.96	115.26	55.34
22×22	484	0.98	0.97	0.98	1.00	0.98	0.94	0.98	0.97	128.42	62.71
24×24	576	0.98	0.97	0.98	1.00	0.98	0.88	0.98	0.94	143.16	63.62
26×26	676	0.98	0.97	0.98	1.00	0.98	0.94	0.98	0.97	188.63	57.87
28×28	784	0.98	0.95	0.98	1.00	0.98	0.90	0.98	0.95	234.27	55.46

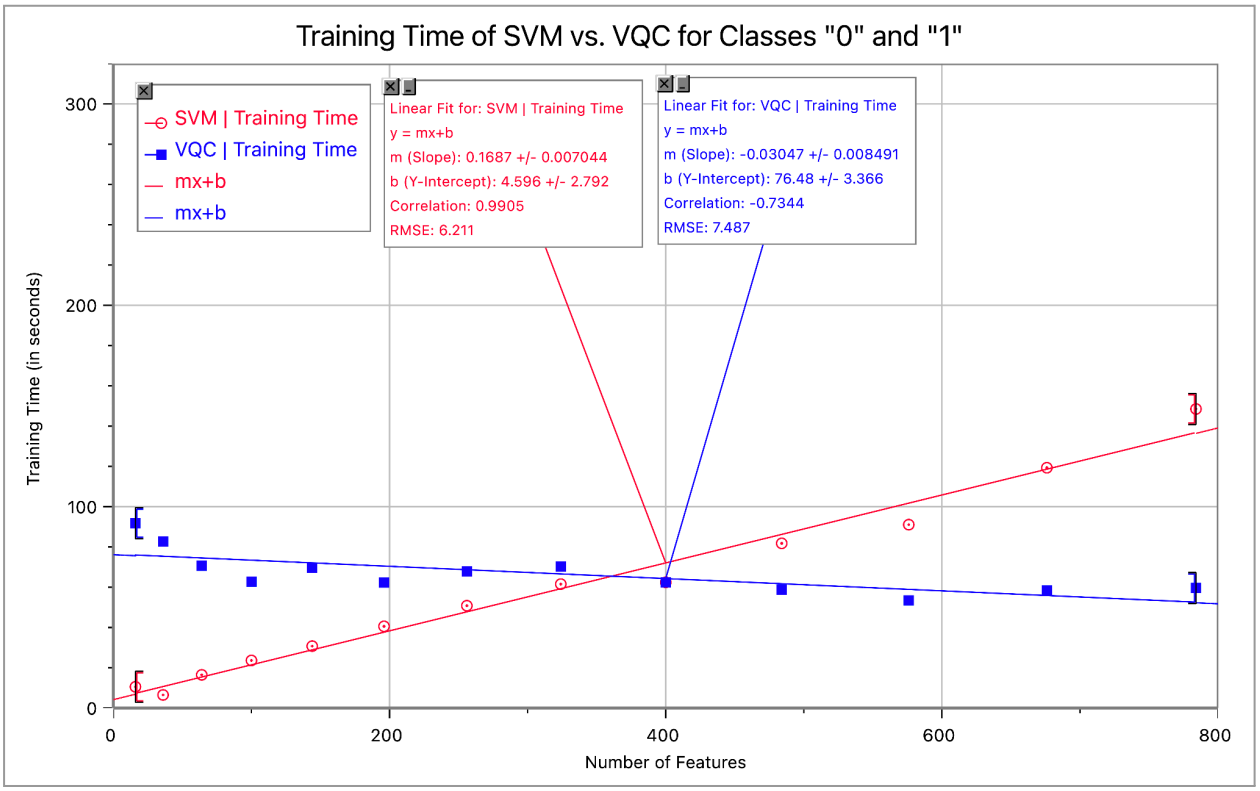
Table 5.3: SVM vs VQC Training Time and Accuracy Metrics for Classes “4” vs “9”

Resolution	Number of Features	Accuracy (out of 1)		Precision (out of 1)		Recall (out of 1)		F1 Score (out of 1)		Training Time (in seconds)	
		SVM	VQC	SVM	VQC	SVM	VQC	SVM	VQC	SVM	VQC
4×4	16	0.87	0.51	0.87	0.51	0.87	0.38	0.87	0.44	25.13	115.07
6×6	36	0.89	0.39	0.87	0.39	0.88	0.40	0.87	0.40	14.50	73.14
8×8	64	0.90	0.57	0.90	0.63	0.91	0.34	0.90	0.44	38.28	104.57
10×10	100	0.94	0.39	0.94	0.43	0.94	0.70	0.94	0.53	54.20	61.42
12×12	144	0.96	0.53	0.96	0.71	0.95	0.10	0.95	0.18	67.36	63.50
14×14	196	0.97	0.49	0.97	0.33	0.97	0.02	0.97	0.04	89.09	70.39
16×16	256	0.97	0.52	0.97	1.00	0.97	0.04	0.97	0.08	110.84	78.47
18×18	324	0.97	0.46	0.97	0.38	0.97	0.12	0.97	0.18	132.57	77.44
20×20	400	0.97	0.52	0.97	1.00	0.97	0.04	0.97	0.08	154.31	109.35
22×22	484	0.97	0.49	0.97	0.44	0.97	0.08	0.97	0.14	176.05	112.15
24×24	576	0.97	0.62	0.97	0.77	0.97	0.34	0.97	0.47	197.78	95.99
26×26	676	0.97	0.46	0.97	0.46	0.97	0.50	0.97	0.48	255.17	87.16
28×28	784	0.97	0.49	0.97	0.61	0.97	0.43	0.97	0.52	312.55	103.90

6. Analysis

6.1 Interpreting Training Time

Figures 6.1, 6.2, and 6.3 compares training times T_{SVM} and T_{VQC} with respect to the number of features n, the coefficient of determination (R^2) for each linear function, and the point in time $(T_{SVM} = T_{VQC})$ at which the training speed of VQC exceeds that of an SVM with more features:



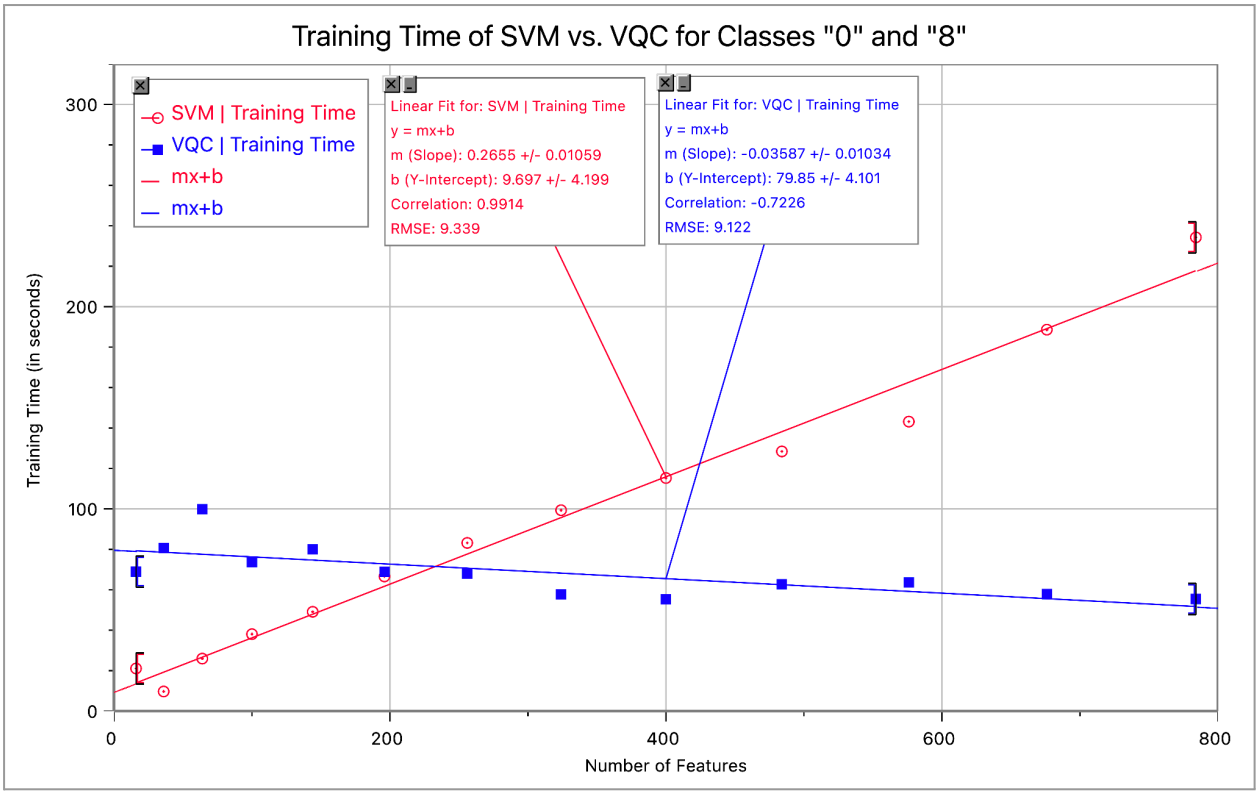
▲ Figure 6.1: Training Time for SVM vs VQC (Rudimentary Visual Discernibility "0" and "1")

$$T_{SVM(0,1)} = 0.1687n + 4.596;\ (R_{SVM(0,1)})^2 = 0.9811$$

$$T_{VQC(0,1)} = -0.03047n + 76.48;\ (R_{VQC(0,1)})^2 = 0.5393$$

$$T_{SVM(0,1)} = T_{VQC(0,1)} = 65.48\ s\ (n = 360)$$

▲ Equations 6.1: Training Time Summary (Rudimentary Visual Discernibility "0" and "1")



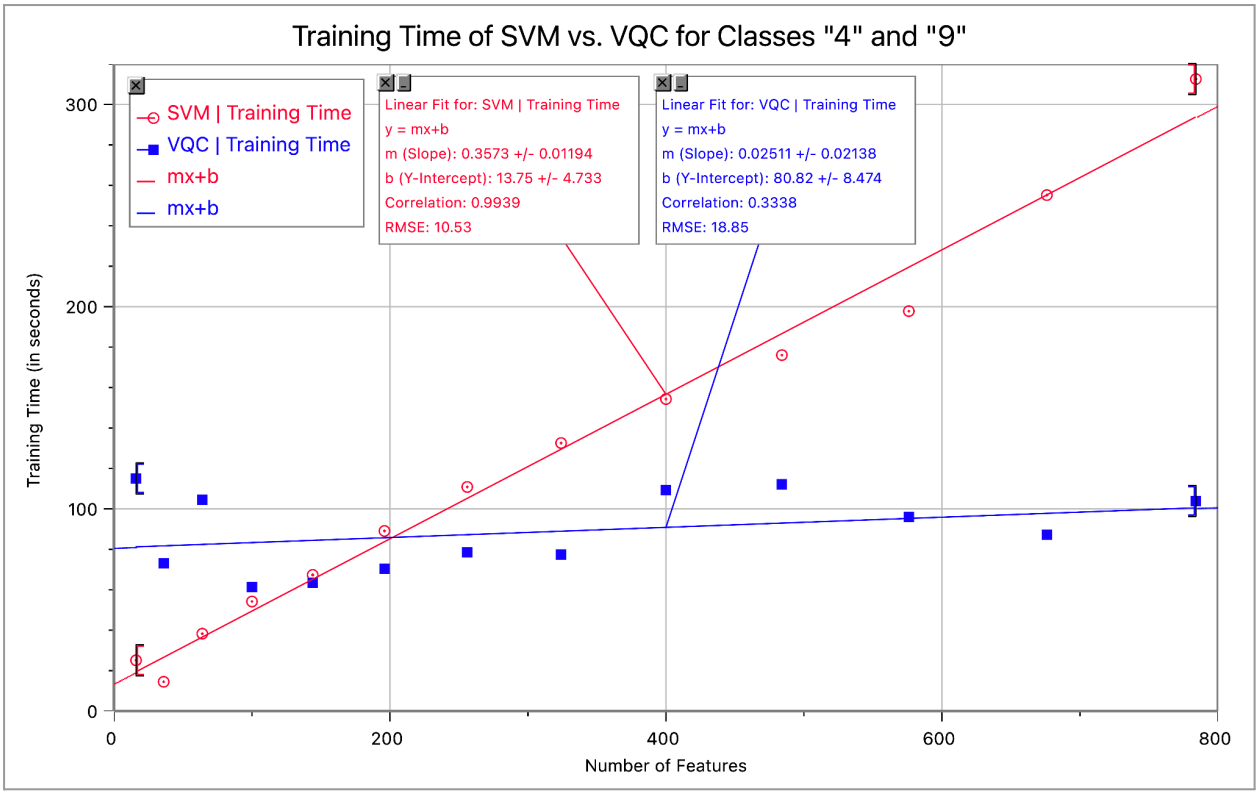
▲ Figure 6.2: Training Time for SVM vs VQC (Intermediate Visual Discernibility "0" and "8")

$$T_{SVM(0,8)} = 0.2665n + 9.697;\ (R_{SVM(0,8)})^2 = 0.9829$$

$$T_{VQC(0,8)} = -0.03587n + 79.85;\ (R_{VQC(0,8)})^2 = 0.5222$$

$$T_{SVM(0,8)} = T_{VQC(0,8)} = 71.53\ s\ (n = 232)$$

▲ Equations 6.2: Training Time Summary (Intermediate Visual Discernibility "0" and "8")



▲ Figure 6.3: Training Time for SVM and VQC (Advanced Visual Discernibility "4" and "9")

$$T_{SVM(4,9)} = 0.3573n + 13.75;\ (R_{SVM(4,9)})^2 = 0.9878$$

$$T_{VQC(4,9)} = 0.02511n + 80.82;\ (R_{VQC(4,9)})^2 = 0.1113$$

$$T_{SVM(4,9)} = T_{VQC(4,9)} = 85.89\ s\ (n = 202)$$

▲ Equations 6.3: Training Time Summary (Advanced Visual Discernibility "4" and "9")

Figures 6.1, 6.2, and 6.3 above delineates the variation of SVM training time (T_{SVM}) compared with VQC training time (T_{VQC}) with n, when classifying digits "0" vs "1", "0" vs "8", and "4" vs "9". The exhibited relationship between both classifiers can be represented **linearly**:

$$T = gradient(T) * n + y\,intercept(T)$$

▲ Equations 6.4: General Linear Equation for Training Time

For instances, from $gradient(T_{SVM(0,1)}) = 0.1687\,s$, the SVM's training time generally increases by **0.1687 seconds** for every additional feature in each data point; $gradient(T_{VQC(0,1)}) = -0.03047\,s$ indicates that the VQC's training time generally decreases by **0.03047 seconds** per added feature.

From $y\,intercept(T_{SVM(0,1)}) = 4.596\,s$, the SVM's base training time is designated at **4.596 seconds** while $y\,intercept(T_{SVM(0,1)}) = 76.48\,s$ suggests VQCs start at **76.48 seconds** when classifying these classes. The distinct values of $y\,intercept$ between differing class comparisons posits that feature discernibility affects the base training time of both classifiers.

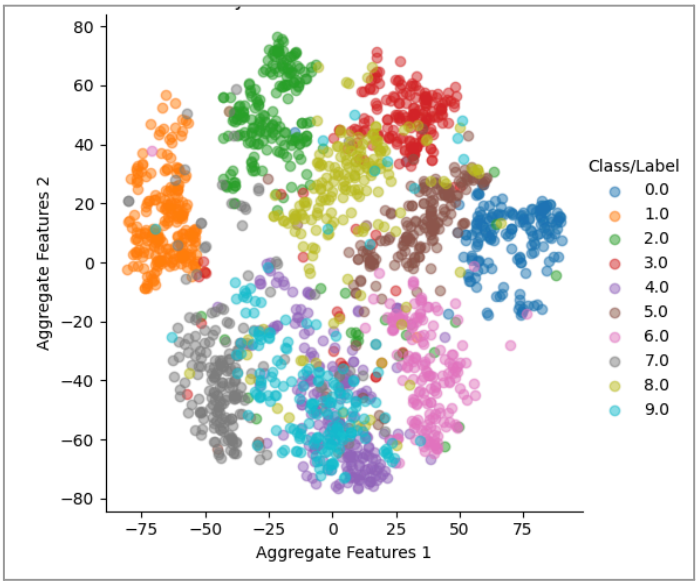
6.2 Analyzing Training Time

A strong linear relationship between the SVM's training time and number of features, indicated by $(R_{SVM})^2 > 0.98$ for all trials, suggest that the algorithm becomes more computationally expensive with **proportional** increases in features. Such an established trend aligns with Equation 2.2, as the implemented polynomial kernel processes the dot products of feature vectors x (for one class) and z (and another) with respect with class labels y embedded within K_{poly}). Therefore, the training speeds of SVMs are punished when classifying images of MNIST digits with increasingly high resolutions, given their need to conduct more iterations to transform the complex data points into a higher-dimensional feature space.

For VQCs, they do not exhibit an explicit linear trend, where the low values of $(R_{VQC})^2$ indicate that the training time of these algorithms do not correlate directly with increases in features. However, a general trend between can be ascertained: they are very low.

- $gradient(T_{VQC(0,1)}) = -0.03047\ s$
- $gradient(T_{VQC(0,8)}) = -0.03587\ s$,
- $gradient(T_{VQC(4,9)}) = 0.02511\ s$

This indicates that the number of features **do not directly affect** the time efficiency of VQCs, where the rate of change of training time is too insignificant—in terms of magnitude and positive/negative direction—to be considered as a consequence of a greater aggregation of features for every data point. As a result, the VQC algorithm seems to effectively perform qubit operations in parallel, such that an increase in the number of features does not affect the iterations performed by the PQC with the chosen `EfficientSU2` ansatz.



▲ **Figure 6.1: Segregation of MNIST Classes via t-SNE** (*adapted from* Seaborn & Q-munity, 2022)

Furthermore, when classifying digits of differing levels of visual discernibility due to relative differences in the feature matrix of each digit (see Equations 4.1, 4.2, and 4.3), both the SVMs and VQCs tend to be **more computationally expensive** as indicated by their general increase in training times with the number of features. Given how the gradient of T_{SVM} increases with respect to the level of visual discernibility:

$$gradient(T_{SVM(0,1)}) < gradient(T_{SVM(0,8)}) < gradient(T_{SVM(4,9)})$$

$$0.1687\ s < 0.2665\ s < 0.3573\ s$$

It is evident that the classification task's difficulty influences the computational demands of SVMs. Similarly, the y-intercept of T_{VQC} increases as visual discernibility becomes progressively advanced, where the VQC's gradients are not a gauging parameter as $gradient(T_{VQC}) \rightarrow 0\ s$ in general.

Since:

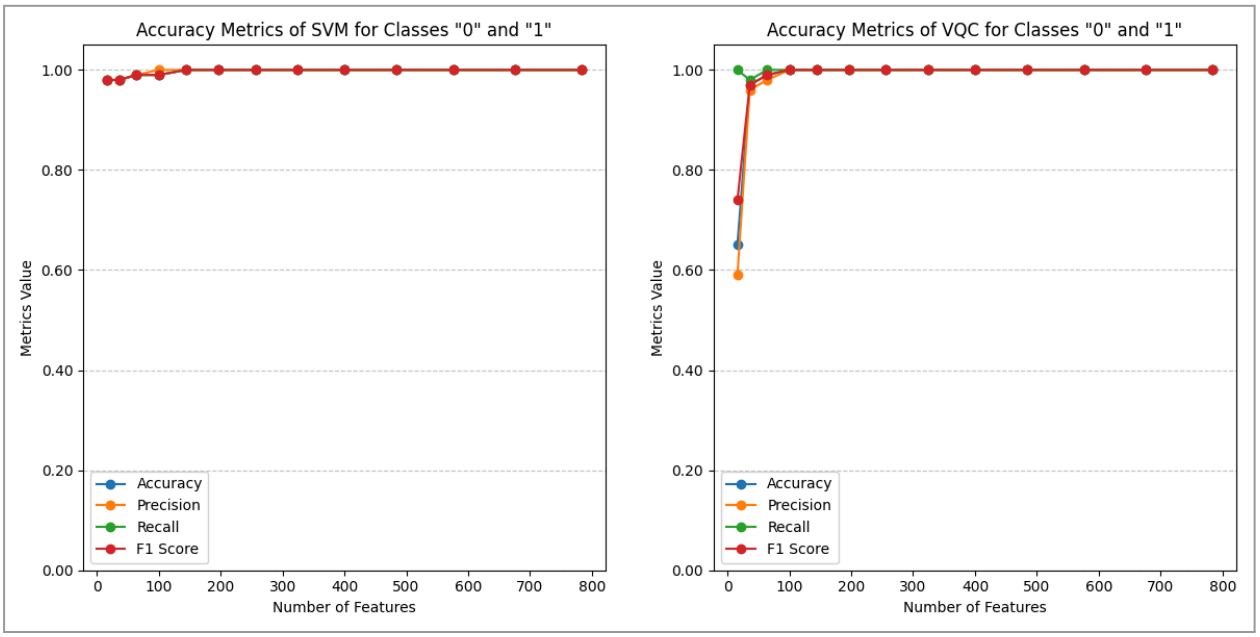
$$y\ intercept(T_{VQC(0,1)}) < y\ intercept(T_{VQC(0,8)}) < y\ intercept(T_{VQC(4,9)})$$

$$76.48\ s < 79.85\ s < 80.82\ s$$

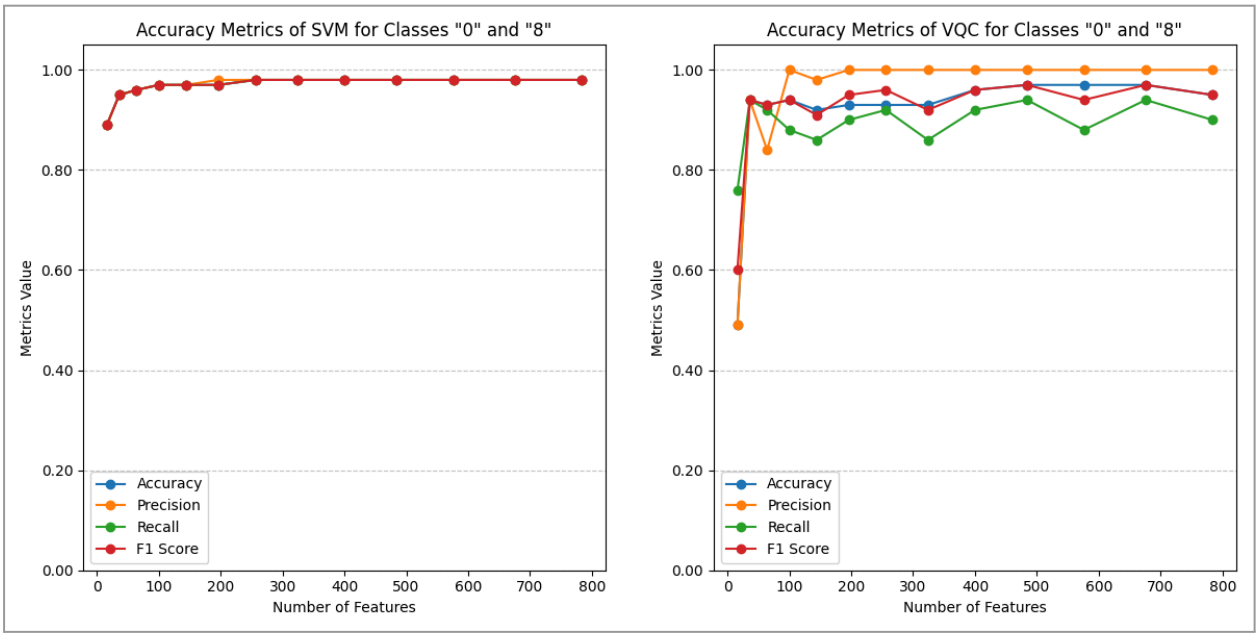
It may be inferred that the VQC would require to conduct more iterations with levels of visual discernibility, as a sufficient distinction between the two digit classes is absent (see Figure 6.1).

Nevertheless, it is noteworthy that VQCs consistently demonstrate superior time efficiency when compared to SVMs. While VQCs initially require more training time than SVMs—most probably due to insufficient qubit entanglements by `EfficientSU2`, there exists a threshold within the explored feature domain of 0 to 784, where the quantum classifier begins to surpass its classical counterpart. This juncture ($T_{SVM} = T_{VQC}$) is attributed to the VQC's consistent time efficiency across higher dimensions, unlike the SVM's much steeper linear relationship with number of features. Consequently, VQCs are **more time efficient in higher-dimensional feature spaces** when compared to SVMs as the number of features approaches infinity.

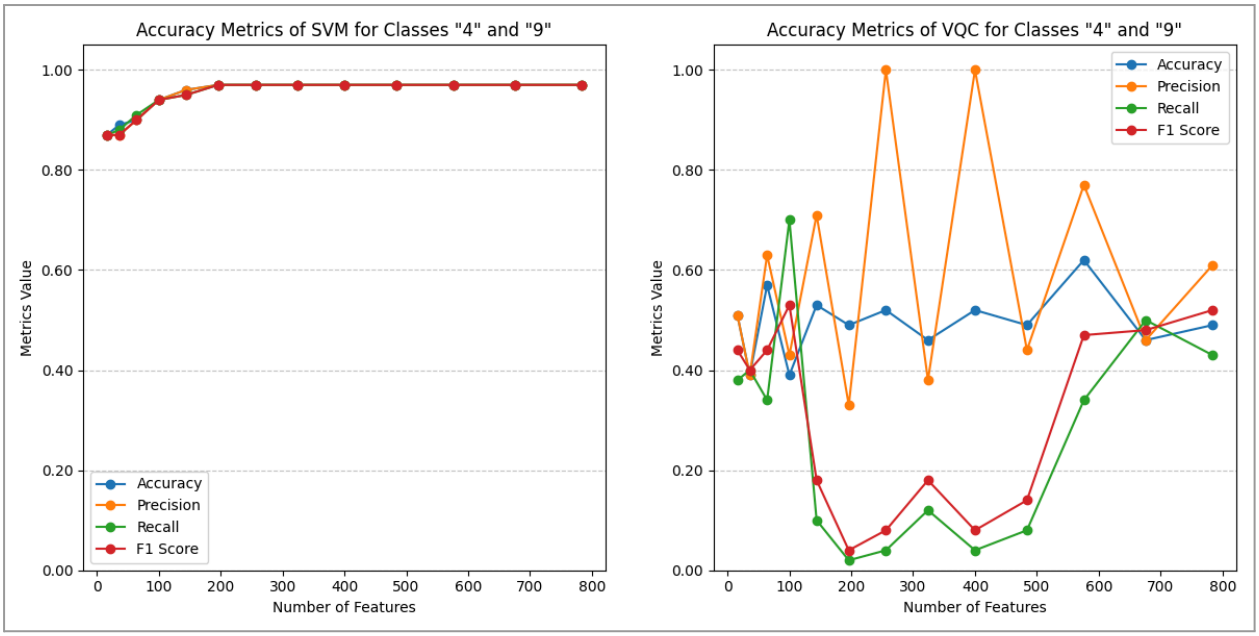
6.3 Interpreting Accuracy Metrics



▲ Figure 6.4: Accuracy Metrics for SVM and VQC (Rudimentary Discernibility “0” and “1”)



▲ Figure 6.5: Accuracy Metrics for SVM and VQC (Intermediate Discernibility “0” and “8”)



▲ Figure 6.6: Accuracy Metrics for SVM and VQC (Advanced Discernibility "4" and "9")

From Figure 6.4, both SVMs and VQCs, when classifying digits "0" and "1", **consistently maintain their accuracy metrics** as the number of features increases, albeit the VQC starts worse off, likely attributed to the same reason of initial entanglement insufficiency via the `EfficientSU2` ansatz. However, Figure 6.5 reveals that the intermediate visual discernibility leads to **significantly diminished accuracy metrics** for the VQC—albeit a consistently strong precision score, in stark contrast to the more stable performance of SVM in classifying digits "0" and "8". The compromise in VQC performance in accurately predicting labels is especially evident in the advanced discernibility classification task for digits "4" and "9", as shown by Figure 6.6. The quantum classifier performs **significantly worse** than the SVM, as all accuracy metrics aggressively fluctuate without any indication of a strong or persistent positive trend with respect to increasing features.

6.4 Analyzing Accuracy Metrics

In reference to all 3 trials, the combined accuracy of SVMs are shown to be resolute, where a lack of features—which initially may put the metrics slightly below 0.90 (see Figures 6.4 and 6.5)—has its adversity dampened. After approximately 225 features, all SVM algorithms: $SVM(0,1)$, $SVM(0,8)$, and $SVM(4,9)$, all maintain a weighted average of accuracy metrics close to a flawless 1.00. A uniformly high accuracy in binary classification tasks, regardless of the level of visual discernibility between classes, can be attributed to the suitable implementation of a polynomial kernel-based decision function (Eq. 2.3) with respect to the presented high-dimensional dataset at hand. As the polynomial degree is sustained at $2.0 \leq d \leq 4.0$ even within the maximum 784 features, the SVM model does not overfit and can therefore accurately predict novel data points.

As for VQCs, a perceptible interrelation exists between the model's accuracy metrics and visual discernibility level of classes with respect to increases in the number of features. When classifying rudimentary and intermediate classes, the models, i.e. $VQC(0,1)$ and $VQC(0,8)$, do improve in average scores given the PQC's capability to better optimize the ansatz $U(\theta)$, with respect to parameter values θ that do not overfit. Thus, with more features, the `EfficientSU2` ansatz could better establish complex relationships between high-dimensional data prior to their mapping in `ZZFeatureMap`, whilst preventing overfitting at the same time through circuit regularization such that unwanted noise is not memorized.

It should be noted that the precision score of $VQC(0, 8)$ seems to be more strongly consistent with the feature count, indicating that it is adept at instituting positive predictions. On the contrary, its lower recall score suggests that the said VQC might struggle to identify all instances of the positive class accurately. The **precision-recall tradeoff** implies cautious and focused qubit state preparation on a specific Bloch sphere point for confident positive label identification, but at the cost of potentially missing instances due to limited superposition utilization, driving inflexibility.

On the other hand, the unpromising accuracy metrics $VQC(4, 9)$ is most plausibly associated with the **rise in magnitude of noise** due to less discernible classes that caused the PQC to overfit the training data. During this specific trial, the noise in the VQC becomes more pronounced, attributed to data points sharing similar feature matrices and resulting in substantial overlap that amplifies interference and diminishes distinct peaks in the probability amplitude. Since the VQCs were instantiated via the backend `'qasm_simulator'` (see Appendix 10.2), designed to emulate a NISQ environment, the consequences of noise and errors would inadvertently undermine the performance of the VQC.

While the accuracy remains most consistent—as a perfectly inaccurate model approaches a score of 0.50 for binary classification, the precision-recall tradeoff is much more evident (see Figure 6.6). The precision score fluctuates obtrusively, at a range of **0.33–1.00**, between **200 to 500 features**, while the recall score exhibits a significant drop to **0.02** in this domain. Subsequently, the F1 score would reflect the aforementioned drop as well. These fluctuations in precision and recall ultimately reflect the sensitive nature of the NISQ-based VQC's decision boundary prior from a certain feature domain, where noise is most prominent.

7. Conclusion

The results of this study indicate that the proposed **hypothesis is valid** for both assumptions of time efficiency (1) and accuracy metrics (2).

VQCs have the potential to surpass SVMs in terms of time efficiency in increasingly high dimensional feature spaces, due to their parallel qubit processing capability. As indicated by how $gradient(T_{VQC}) \rightarrow 0\ s$ for all trials, the hypothesis in (1) correctly suggests that VQCs approach a constant training time. In contrast, SVM training times tend to exhibit a linear increase with features due to the computational nature of polynomial kernels, exhibiting a direct proportionality between them as also stated by the hypothesis in (1).

Nevertheless, the reliability of VQCs is only prominent when dealing with rudimentary levels of visual discernibility, as seen in binary classification tasks encompassing classes "0" and "1". This is evident through noticeably lower accuracy metrics compared to SVMs in more complex classification scenarios, i.e. Trials 2 and 3, underscoring their performance superiority only when distinguishing classes with distinct feature matrices—confirming that VQCs are not scalable in accuracy beyond superficial classification tasks as stated by the hypothesis in (2).

Therefore, VQCs **do not possess absolute leverage** over SVMs when performing binary classification, where quantum advantage is only applicable to less challenging classification tasks. In practical applications, these findings imply that while quantum computing can offer training speed advantages, it may not be as reliable and accurate as SVMs for complex binary classification tasks such as medical diagnoses among many others.

8. Evaluation

8.1 Strengths of Methodology

MNIST's expansiveness enables the methodology to explore a **substantial domain of feature dimensions**, ranging from 4×4 to 28×28, with varying levels of visual discernibility. By incorporating an extensive scope for testing, the methodology would enable the thorough assessment of SVM and VQC performances across complex feature spaces that better reflect data structures in more practical instances. Furthermore, the measurement of **multiple accuracy metrics**, including accuracy, precision, recall, and F1 score, enhances the comprehensiveness of the investigation. This was particularly evident in the metrics for $VQC(4, 9)$ shown by Figure 6.6, where each metric exhibits its own distinct pattern as the number of features increases—providing insight into its potential proficiency in identifying false positive instances (high precision) but not true positives (low recall).

8.2 Limitations of Methodology

8.2.1 Limitations for SVM

The kernel function was **fixed to polynomial**. Different kernel functions, such as radial basis function (RBF) or sigmoid kernels, may better capture various underlying patterns in the data, where firsthand testing may better ascertain SVM potential. In addition, since the **polynomial degree range was limited to 2 to 4**—albeit this was done to prevent overfitting—may limit the potential of maximally optimal SVM performance. Therefore, the SVM's ability to adapt to varying levels of data complexity is compromised, potentially leading to reduced performance on certain feature domains.

8.2.2 Limitations for VQC

The feature map was **fixed to** `ZZFeatureMap` **with the** `EfficientU2` **ansatz**. Although the usage of these PQC components are justified by extensive theoretical research (see subsections 2.3.3 and 2.3.4), testing with other feature maps such as `ZFeatureMap` or `PauliFeatureMap` and ansatzes including `RealAmplitudes` or `RYRZ` firsthand may retrieve better VQC performance, especially in terms of accuracy metrics. Moreover, **limiting the scope of the quantum backend** to `'qasm_simulator'` may not accurately reflect the noise and errors on real quantum computers, affected by various real external influences.

8.3 Improvements and Further Research Opportunities

To reduce the consequences of limitations in the methodology, improvements to both the SVM and VQC algorithms can be made. For SVM, different kernels (RBF and sigmoid) can be tested whilst a larger degree range can be optimized prior to the polynomial kernel. For VQCs, different feature maps (`ZFeatureMap`, `PauliFeatureMap`) and ansatzes (`RealAmplitudes`, `RYRZ`) can be tested. Additionally, as noise and interference are purely emulated, it may be wiser to investigate with multiple quantum backends (`santiago`, `lima`) as a controlled procedure to ensure robustness in a variety of quantum environments.

For further research opportunities, custom feature maps and ansatzes can be constructed for VQCs such that they are suitable to be tested in more complex ML problems, such as multinomial classification, unsupervised learning, or even reinforcement learning. Other classical ML algorithms can also be tested against the VQC, particularly Artificial Neural Networks for multinomial classification and Deep Q-Network for reinforcement learning, to better gauge the extent of quantum potential with a multitude of its classical counterparts.

9. Bibliography

Ben-Hur, A. (2008) *The effect of the degree of a polynomial kernel.* Available at: https://www.researchgate.net/figure/The-effect-of-the-degree-of-a-polynomial-kernel-The-polynomial-kernel-of-degree-1-leads_fig12_23442384 (Accessed: 30 August 2023).

Brandhofer, S. (2023) *Special Session: Noisy Intermediate-Scale Quantum (NISQ) Computers—How They Work, How They Fail, How to Test Them?* Available at: https://arxiv.org/pdf/2301.11739.pdf (Accessed: 30 August 2023).

Berwick, R. (no date) *An Idiot's Guide to support Vector Machines (svms)*, *MIT*. Available at: https://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf (Accessed: 30 August 2023).

Corochann (2021) *corochannNote - Deep learning, Machine learning, Android etc.* Available at: https://corochann.com/mnist-dataset-introduction-532/ (Accessed: 30 August 2023).

Dargan, J. (2023) *What is NISQ quantum computing?*, *The Quantum Insider*. Available at: https://thequantuminsider.com/2023/03/13/what-is-nisq-quantum-computing/ (Accessed: 30 August 2023).

Fuster, E.M.G. (2019) *Variational Quantum Classi er - diposit.ub.edu*. Available at: https://diposit.ub.edu/dspace/bitstream/2445/140318/1/GIL%20FUSTER%20Elies%20Miquel.pdf (Accessed: 30 August 2023).

Haque, A.K. (2022) 'Optimizing a variational quantum classifier through the behavior analysis of its components', *Journal of Ecology & Natural Resources*, 6(3). doi:10.23880/jenr-16000296.

Havlicek, V. (2020) *Quantum kernel functions A, feature map representation for a single ...*, *ResearchGate*. Available at: https://www.researchgate.net/figure/Quantum-kernel-functions-a-Feature-map-representation-for-a-single-qubit-A-classical_fig1_331719791 (Accessed: 30 August 2023).

Hui, J. (2022) *QC-what are qubits in quantum computing?*, *Medium*. Available at: https://jonathan-hui.medium.com/qc-what-are-qubits-in-quantum-computing-cdb3cb566595 (Accessed: 30 August 2023).

IBM (2023) *IBM Quantum*. Available at: https://quantum-computing.ibm.com/composer/files/new (Accessed: 30 August 2023).

Jäger, J. and Krems, R.V. (2023) 'Universal expressiveness of variational quantum classifiers and quantum kernels for support vector machines', *Nature Communications*, 14(1), pp. 1–2. doi:10.1038/s41467-023-36144-5.

Karabiber, F. *et al*. (2023) *Binary classification*, *Learn Data Science - Tutorials, Books, Courses, and More*. Available at: https://www.learndatasci.com/glossary/binary-classification/ (Accessed: 27 August 2023).

Kumar, S. (2020) *SVM: Difference between linear and non-linear models*, *AITUDE*. Available at:

https://www.aitude.com/svm-difference-between-linear-and-non-linear-models/ (Accessed: 30 August 2023).

Maheshwari, D., Sierra-Sosa, D. and Garcia-Zapirain, B. (2022) 'Variational quantum classifier for binary classification: Real vs synthetic dataset', *IEEE Access*, 10, pp. 1–2. doi:10.1109/access.2021.3139323.

Moguerza , J.M. and Muñoz, A. (2006) *Support Vector Machines with applications*. Available at: https://www.jstor.org/stable/pdf/27645765.pdf (Accessed: 30 August 2023).

Ming, D.L. and Chuanfeng, L. (2023) *Quantum Gate*, *Quantum Gate - an overview | ScienceDirect Topics*. Available at: https://www.sciencedirect.com/topics/mathematics/quantum-gate (Accessed: 30 August 2023).

Nguyen, N., Behrman, E.C. and Steck, J.E. (2016) *Quantum Learning with Noise and Decoherence: A Robust Quantum Neural Network*. Available at: https://arxiv.org/ftp/arxiv/papers/1612/1612.07593.pdf (Accessed: 30 August 2023).

OpenAI (2023) *ChatGPT*. Available at: https://openai.com/chatgpt (Accessed: 30 August 2023).

Osodo, R. (2023) *Qa2. explaining variational quantum classifiers*, *Medium*. Available at: https://medium.com/swlh/qa2-explaining-variational-quantum-classifiers-b584c3bd7849 (Accessed: 30 August 2023).

Park, S., Park, K. and Rhee, J.-K. (2022) *Variational quantum approximate support vector machine with inference transfer*, pp. 1–4. doi:10.21203/rs.3.rs-2136572/v1.

Patel, A. and Kalyani, T.V. (2016) 'Support Vector Machine with inverse fringe as feature for Mnist Dataset', *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, pp. 1–2. doi:10.1109/iacc.2016.32.

Piatrenka, I. and Rusek, M. (2022) 'Quantum variational multi-class classifier for the iris data set', *Computational Science – ICCS 2022*, pp. 3–7. doi:10.1007/978-3-031-08760-8_21.

Premanand, S. (2021) *The A-Z guide to support vector machine*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2021/06/support-vector-machine-better-understanding/ (Accessed: 30 August 2023).

Qiskit (2023) *RealAmplitudes*. Available at: https://qiskit.org/documentation/stubs/qiskit.circuit.library.RealAmplitudes.html (Accessed: 30 August 2023).

Q-munity (2022) *Building a variational quantum classifier*. Available at: https://www.qmunity.tech/tutorials/building-a-variational-quantum-classifier (Accessed: 30 August 2023).

Sidharth (2022) *SVM kernels: Polynomial kernel - from scratch using python.*, *PyCodeMates*. Available at: https://www.pycodemates.com/2022/10/svm-kernels-polynomial-kernel.html (Accessed: 30 August 2023).

Sweke, R. *et al.* (2020) 'Stochastic gradient descent for hybrid quantum-classical optimization', *Quantum*, 4, p. 5. doi:10.22331/q-2020-08-31-314.

TensorFlow (2023) *MNIST : tensorflow datasets*, *TensorFlow*. Available at: https://www.tensorflow.org/datasets/catalog/mnist (Accessed: 30 August 2023).

Waseem, M. (2023) *Support Vector Machine in python: Classification algorithms*, *Edureka*. Available at: https://www.edureka.co/blog/support-vector-machine-in-python/ (Accessed: 30 August 2023).

Wilimitis, D. (2018) *The kernel trick*, *Medium*. Available at: https://towardsdatascience.com/the-kernel-trick-c98cdbcaeb3f (Accessed: 30 August 2023).

10. Appendix

10.1 Python Code for SVM Algorithm

```
import pandas as pd
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
from skimage.transform import resize
import time

# Load the MNIST training dataset in CSV format
data_path = "./datasets/"
train_data = pd.read_csv(data_path + "mnist_train.csv",
delimiter=",")

# Choose the digits to classify
class_1 = 0
class_2 = 1

# Extract features and labels for the training dataset
X_train_data = train_data[train_data['label'].isin([class_1,
class_2])].drop(columns=['label']).values
y_train_data = train_data[train_data['label'].isin([class_1,
class_2])]['label'].values

# Choose a target resolution
target_resolution = (28, 28)

# Reshape the images to 2D before resizing
resized_train_images = [resize(image.reshape(28, 28),
target_resolution) for image in X_train_data]

# Flatten the resized images for SVM input
flattened_train_images = [image.flatten() for image in
resized_train_images]

# TESTING DATA
# Load the MNIST test dataset in CSV format (replace with your
dataset file paths)
test_data = pd.read_csv(data_path + "mnist_test.csv",
delimiter=",")

# Extract features and labels for the test dataset
X_test_data = test_data.drop(columns=['label']).values
y_test_data = test_data['label'].values

# Reshape the test images to 2D before resizing
resized_test_images = [resize(image.reshape(28, 28),
target_resolution) for image in X_test_data]

# Flatten the resized test images for SVM input
flattened_test_images = [image.flatten() for image in
resized_test_images]
```

```
# Create an SVM classifier with polynomial kernel
svm_classifier = SVC(kernel='poly', C=1.0, coef0=1.0, degree=4)

# Measure training time
start_time = time.time()

# Train the SVM classifier on the training dataset
svm_classifier.fit(flattened_train_images, y_train_data)

# Calculate training time
end_time = time.time()
training_time = end_time - start_time
print(f"Training Time: {training_time:.2f} seconds")

# Predict on the test set
y_pred = svm_classifier.predict(flattened_test_images)

print("Predicted Labels:", y_pred)

# Calculate accuracy
accuracy = accuracy_score(y_test_data, y_pred)
precision = precision_score(y_test_data, y_pred,
average='weighted')
recall = recall_score(y_test_data, y_pred, average='weighted')
f1 = f1_score(y_test_data, y_pred, average='weighted')

print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

10.2 Python Code for VQC Algorithm

```
import matplotlib.pyplot as plt
from sklearn.decomposition import TruncatedSVD
from sklearn.manifold import TSNE
from qiskit import *
import numpy as np
from qiskit.utils import QuantumInstance
from qiskit_machine_learning.algorithms.classifiers import VQC
from qiskit.circuit.library import ZZFeatureMap, ZFeatureMap,
PauliFeatureMap
import time
from sklearn.metrics import precision_score, recall_score,
f1_score
from sklearn.metrics import accuracy_score
from skimage.transform import resize

image_size = 28  # width and length are equal
data_path = "./datasets/"
train_data = np.loadtxt(data_path + "mnist_train.csv",
delimiter=",", skiprows=1)
test_data = np.loadtxt(data_path + "mnist_test.csv",
delimiter=",", skiprows=1)

train_data_features = train_data
train_data_labels = train_data.reshape(60000,)

test_data_features = test_data
test_data_labels = test_data.reshape(10000,)

target_resolution = (28, 28)

# Choose the digits to classify
class_1 = 0
class_2 = 1

resized_train_images = [resize(image.reshape(28, 28),
target_resolution) for image in train_data_features]
resized_test_images = [resize(image.reshape(28, 28),
target_resolution) for image in test_data_features]

train_data_features = np.array([image.flatten() for image in
resized_train_images])
test_data_features = np.array([image.flatten() for image in
resized_test_images])

tsvd = TruncatedSVD(n_components=10)
X_SVD = tsvd.fit_transform(train_data_features)
X_test_SVD = tsvd.fit_transform(test_data_features)

np.random.seed(0)
tsne = TSNE(n_components=2)
train_data_features_reduced = tsne.fit_transform(X_SVD)
test_data_features_reduced = tsne.fit_transform(X_test_SVD)
```

```
# Train data
zero_datapoints_array_train = []  # an array of the data points
containing value 0
one_datapoints_array_train = []   # an array of the data points
containing value 1

for i in range(60000):
    if train_data_labels[i] == class_1:  # extracting zeros

zero_datapoints_array_train.append(train_data_features_reduced[i])
    elif train_data_labels[i] == class_2:  # extracting ones

one_datapoints_array_train.append(train_data_features_reduced[i])

zero_datapoints_array_train =
np.array(zero_datapoints_array_train)
one_datapoints_array_train = np.array(one_datapoints_array_train)

# Test data
zero_datapoints_array_test = []  # an array of the data points
containing value 0
one_datapoints_array_test = []   # an array of the data points
containing value 1
for i in range(10000):
    if test_data_labels[i] == class_1:  # extracting zeros

zero_datapoints_array_test.append(test_data_features_reduced[i])
    elif test_data_labels[i] == class_2:  # extracting ones

one_datapoints_array_test.append(test_data_features_reduced[i])

zero_datapoints_array_test = np.array(zero_datapoints_array_test)
one_datapoints_array_test = np.array(one_datapoints_array_test)

def normalize(arr, max_val, n):
    a = np.divide(arr, max_val)
    return a + n

zero_datapoints_normalized_train =
normalize(zero_datapoints_array_train, 100, 1)
one_datapoints_normalized_train =
normalize(one_datapoints_array_train, 100, 1)
zero_datapoints_normalized_test =
normalize(zero_datapoints_array_test, 100, 1)
one_datapoints_normalized_test =
normalize(one_datapoints_array_test, 100, 1)

training_input = {'A': zero_datapoints_normalized_train, 'B':
one_datapoints_normalized_train}
test_input = {'A': zero_datapoints_normalized_test, 'B':
one_datapoints_normalized_test}

seed = 12345
feature_dim = zero_datapoints_normalized_train.shape[1]
```

```
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2,
entanglement='full')
feature_map.draw('mpl')

# Quantum Circuit Setup
from qiskit.algorithms.optimizers import COBYLA
from qiskit.circuit.library import EfficientSU2, RealAmplitudes

cobyla = COBYLA(maxiter=500, tol=0.001)

# Create an EfficientSU2 ansatz
var_circuit = EfficientSU2(feature_dim, reps=2)
var_circuit.draw('mpl')
plt.show()

# Initialize backend
backend = Aer.get_backend('qasm_simulator')
backend_options = {"method": "statevector"}

# Create a Quantum Instance
quantum_instance = QuantumInstance(backend, shots=1024,
seed_simulator=seed, seed_transpiler=seed,
backend_options=backend_options)

# Initialize VQC object
vqc = VQC(optimizer=cobyla, feature_map=feature_map,
ansatz=var_circuit, quantum_instance=quantum_instance)

# Concatenate data points from different classes
X_train = np.concatenate((training_input['A'],
training_input['B']))

# Create labels array for the concatenated data points
y_train = np.concatenate((np.zeros(training_input['A'].shape[0]),
np.ones(training_input['B'].shape[0])))

# Measure the training time
start_time = time.time()

# Now you can use X_train and y_train as inputs for vqc.fit
vqc.fit(X_train, y_train)

end_time = time.time()
training_time = end_time - start_time
print(f"Training Time: {training_time:.2f} seconds")

# Concatenate test data points from different classes
X_test = np.concatenate((test_input['A'], test_input['B']))

# Create labels array for the concatenated test data points
y_test = np.concatenate((np.zeros(test_input['A'].shape[0]),
np.ones(test_input['B'].shape[0])))

# Use the trained VQC model to predict labels for the test data
y_pred = vqc.predict(X_test)
```

```
print(f"Predicted labels: {y_pred}")
print(f"Ground truth:     {y_test}")

# Calculate metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

10.3 Matplotlib Code to Generate Accuracy Metric Plots

```
import matplotlib.pyplot as plt

# SVM Data
svm_num_features = [16, 36, 64, 100, 144, 196, 256, 324, 400, 484,
576, 676, 784]
svm_accuracy = [0.89, 0.95, 0.96, 0.97, 0.97, 0.97, 0.98, 0.98,
0.98, 0.98, 0.98, 0.98, 0.98]
svm_precision = [0.89, 0.95, 0.96, 0.97, 0.97, 0.98, 0.98, 0.98,
0.98, 0.98, 0.98, 0.98, 0.98]
svm_recall = [0.89, 0.95, 0.96, 0.97, 0.97, 0.97, 0.98, 0.98,
0.98, 0.98, 0.98, 0.98, 0.98]
svm_f1_score = [0.89, 0.95, 0.96, 0.97, 0.97, 0.97, 0.98, 0.98,
0.98, 0.98, 0.98, 0.98, 0.98]

# VQC Data
vqc_num_features = [16, 36, 64, 100, 144, 196, 256, 324, 400, 484,
576, 676, 784]
vqc_accuracy = [0.49, 0.94, 0.93, 0.94, 0.92, 0.93, 0.93, 0.93,
0.96, 0.97, 0.97, 0.97, 0.95]
vqc_precision = [0.49, 0.94, 0.84, 1.00, 0.98, 1.00, 1.00, 1.00,
1.00, 1.00, 1.00, 1.00, 1.00]
vqc_recall = [0.76, 0.94, 0.92, 0.88, 0.86, 0.90, 0.92, 0.86,
0.92, 0.94, 0.88, 0.94, 0.90]
vqc_f1_score = [0.60, 0.94, 0.93, 0.94, 0.91, 0.95, 0.96, 0.92,
0.96, 0.97, 0.94, 0.97, 0.95]

# Create a new figure with two subplots
plt.figure(figsize=(12, 6))

# Left subplot for SVM
plt.subplot(1, 2, 1)
plt.plot(svm_num_features, svm_accuracy, marker='o',
label='Accuracy')
plt.plot(svm_num_features, svm_precision, marker='o',
label='Precision')
plt.plot(svm_num_features, svm_recall, marker='o', label='Recall')
plt.plot(svm_num_features, svm_f1_score, marker='o', label='F1
Score')
plt.xlabel('Number of Features')
plt.ylabel('Metrics Value')
#plt.title('Accuracy Metrics of SVM for Classes "0" and "1"')
plt.title('Accuracy Metrics of SVM for Classes "0" and "8"')
#plt.title('Accuracy Metrics of SVM for Classes "4" and "9"')
plt.ylim(0, 1.05)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.gca().yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f')
)
plt.legend()

# Right subplot for VQC
plt.subplot(1, 2, 2)
plt.plot(vqc_num_features, vqc_accuracy, marker='o',
label='Accuracy')
```

```
plt.plot(vqc_num_features, vqc_precision, marker='o',
label='Precision')
plt.plot(vqc_num_features, vqc_recall, marker='o', label='Recall')
plt.plot(vqc_num_features, vqc_f1_score, marker='o', label='F1
Score')
plt.xlabel('Number of Features')
plt.ylabel('Metrics Value')
#plt.title('Accuracy Metrics of VQC for Classes "0" and "1"')
plt.title('Accuracy Metrics of VQC for Classes "0" and "8"')
#plt.title('Accuracy Metrics of VQC for Classes "4" and "9"')
plt.ylim(0, 1.05)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.gca().yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f')
)
plt.legend()

# Adjust space between subplots with hspace parameter
plt.tight_layout(w_pad=3)

# Show the plot
plt.show()
```

10.4 MNIST Train CSV File

No.	Label	1×1	1×2	…	14×14	14×15	14×16	14×17	…	28×27	28×28
1	5	0	0	…	241	225	160	108	…	0	0
2	0	0	0	…	0	0	0	0	…	0	0
3	4	0	0	…	0	0	14	96	…	0	0
4	1	0	0	…	104	251	253	184	…	0	0
5	9	0	0	…	36	201	252	252	…	0	0
6	2	0	0	…	110	121	122	121	…	0	0
…	…	…	…	…	…	…	…	…	…	…	…
30000	1	0	0	…	0	96	253	253	…	0	0
30001	3	0	0	…	253	254	238	225	…	0	0
30002	7	0	0	…	0	0	0	5	…	0	0
30003	3	0	0	…	227	107	23	6	…	0	0
30004	9	0	0	…	0	0	26	130	…	0	0
…	…	…	…	…	…	…	…	…	…	…	…
59995	1	0	0	…	59	248	197	14	…	0	0
59996	8	0	0	…	188	253	253	245	…	0	0
59997	3	0	0	…	252	252	253	252	…	0	0
59998	5	0	0	…	248	147	148	91	…	0	0
59999	6	0	0	…	0	0	0	0	…	0	0
60000	8	0	0	…	0	2	63	180	…	0	0

10.5 MNIST Test CSV File

No.	Label	1×1	1×2	…	14×14	14×15	14×16	14×17	…	28×27	28×28
1	7	0	0	…	0	0	0	0	…	0	0
2	2	0	0	…	233	35	0	0	…	0	0
3	1	0	0	…	57	237	205	8	…	0	0
4	0	0	0	…	0	0	0	0	…	0	0
5	4	0	0	…	0	0	0	0	…	0	0
6	1	0	0	…	111	254	254	132	…	0	0
…	…	…	…	…	…	…	…	…	…	…	…
5000	0	0	0	…	0	0	0	0	…	0	0
5001	3	0	0	…	134	73	0	0	…	0	0
5002	9	0	0	…	0	0	7	136	…	0	0
5003	9	0	0	…	0	101	234	252	…	0	0
5004	8	0	0	…	255	255	255	255	…	0	0
…	…	…	…	…	…	…	…	…	…	…	…
9995	1	0	0	…	235	253	253	172	…	0	0
9996	2	0	0	…	0	0	191	255	…	0	0
9997	3	0	0	…	77	185	230	254	…	0	0
9998	4	0	0	…	174	194	254	254	…	0	0
9999	5	0	0	…	248	133	51	0	…	0	0
10000	6	0	0	…	0	0	0	0	…	0	0