Data Analysis on Student Performance Dataset

Abstract

Variational Quantum Classifiers(VQC) are hybrid Quantum-Classical Machine Learning algorithms used for classification tasks using Quantum Computers. Despite Classical Machine Learning methods being very efficient in classification problems, a Quantum approach is explored here. In this review, we display and compare performances of both quantum classifier and classical methods. The review will also get insights on the dataset and reasons for varied accuracy for the models.

Introduction

Machine Learning is a process for machines to find patterns from existing data and produce output for new observations. There are various algorithms in Classical Machine Learning such as Logistic Regression, K-Nearest Neighbors, Support Vector Machines, Decision Tree, Neural Network based classifiers. All of them have similar processing methods, which is to get the classical data as input and reduce a cost function by optimizing the parameters of the model. Similarly, there are various methods for classifying data of a quantum computer. Here, we work on implementing Variational Quantum Classifier and Quantum Support Vector Machines.

Dataset used for the insights consists of 41 students and their academic performance in subject course C theory, Operating Systems and Programming C. Each dataset describes Course Outcomes attainment of students. In this review, we work on this dataset to perform analysis and understand the working of quantum approach for classification of student's course outcome attainment.

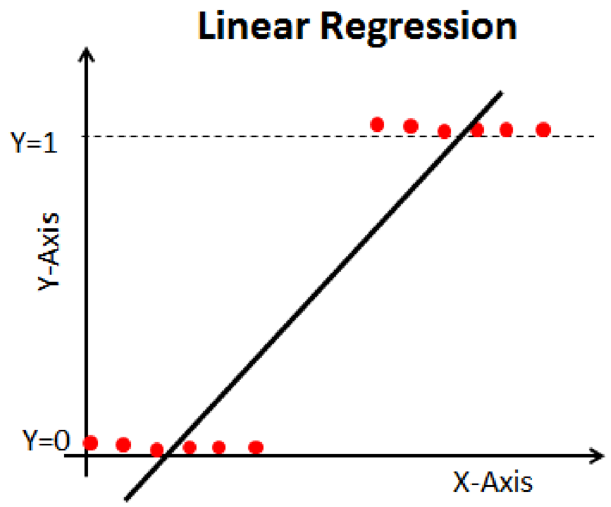
Classical Machine Learning

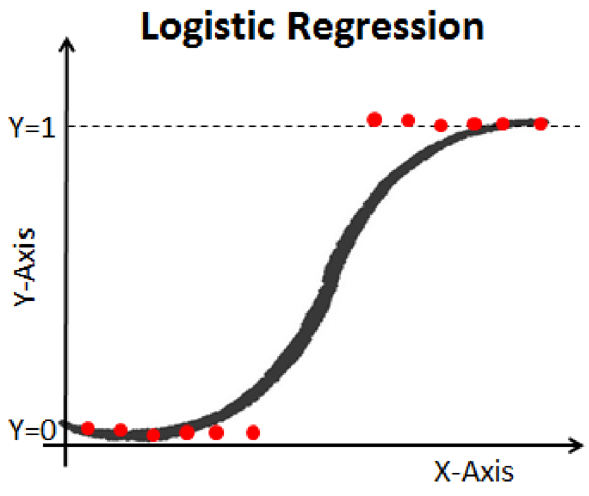
1) Logistic Regression

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). Like all regression analyses, logistic regression is a predictive analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

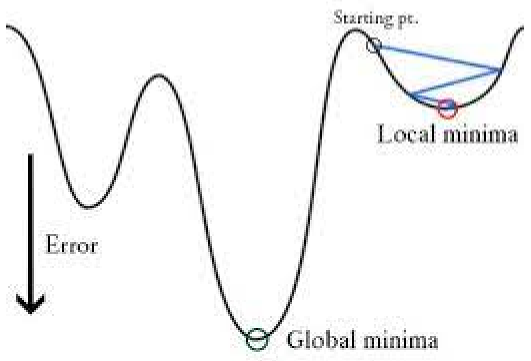
Logistic Function: $P = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$

Equation of best fit line: $y = \beta_0 + \beta_1 x$





In logistic regression Yi is a non-linear function (Ŷ=1/1+ e-z). If we use this in the above MSE equation then it will give a non-convex graph with many local minima as shown

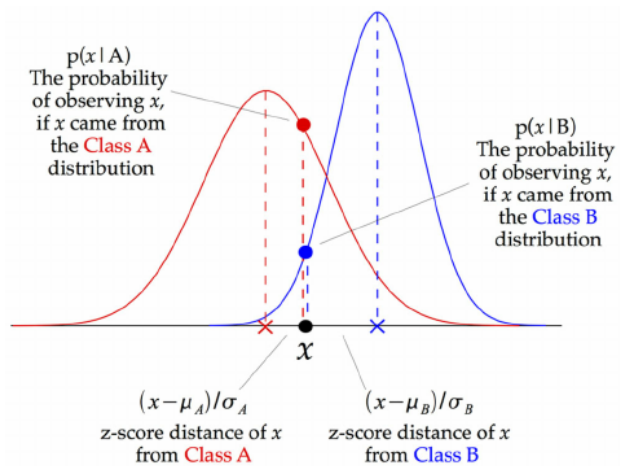


2) Gaussian Naive Bayes

Naive Bayes Classifiers are based on the Bayes Theorem. One assumption taken is the strong independence assumptions between the features. These classifiers assume that the value of a particular feature is independent of the value of any other feature. In a supervised learning situation, Naive Bayes Classifiers are trained very efficiently. Naive Bayes classifiers need a small training data to estimate the parameters needed for classification. Naive Bayes Classifiers have simple design and implementation and they can be applied to many real life situations.

Gaussian Naive Bayes supports continuous valued features and models each as conforming to a Gaussian (normal) distribution.

An approach to create a simple model is to assume that the data is described by a Gaussian distribution with no co-variance (independent dimensions) between dimensions.

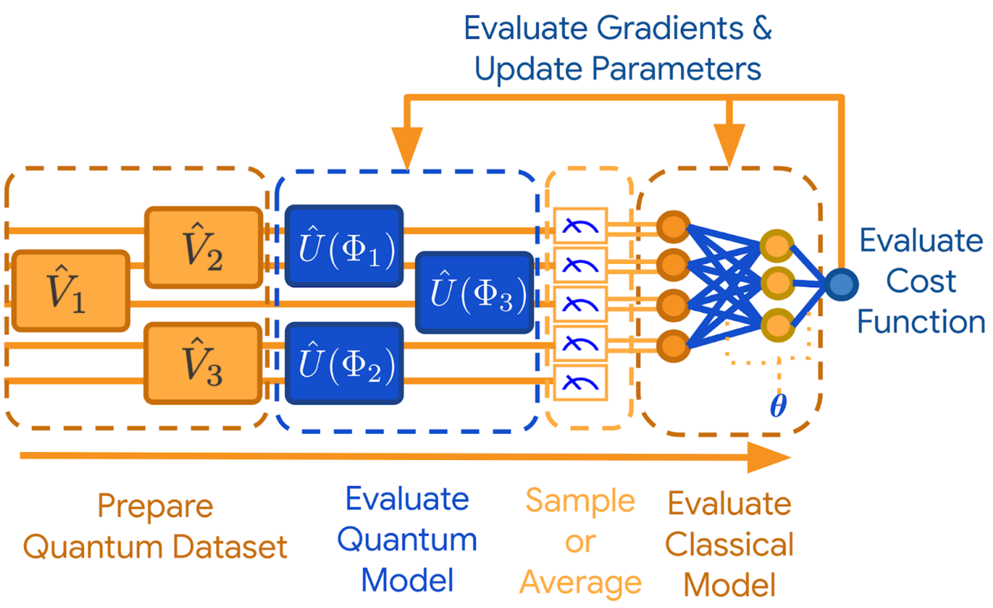

p(x | A)
The probability of observing x, if x came from the Class A distribution
p(x | B)
The probability of observing x, if x came from the Class B distribution
x
$(x-\mu_A)/\sigma_A$
z-score distance of x from Class A
$(x-\mu_B)/\sigma_B$
z-score distance of x from Class B

Quantum Machine Learning

Variational Quantum Classifier

Similar to classical machine learning, VQC has training dataset (consists of data points with label) and testing dataset (consists of data points without label).
VQC has four stages:

1. Feature Map: Loading Data into Quantum Computer
2. Variational Circuit: quantum classification circuit.
3. Measurement and assigning a binary label.
4. Classical optimization loop.



[HQC Machine Learning Algorithm Workflow].

1. Feature Map:

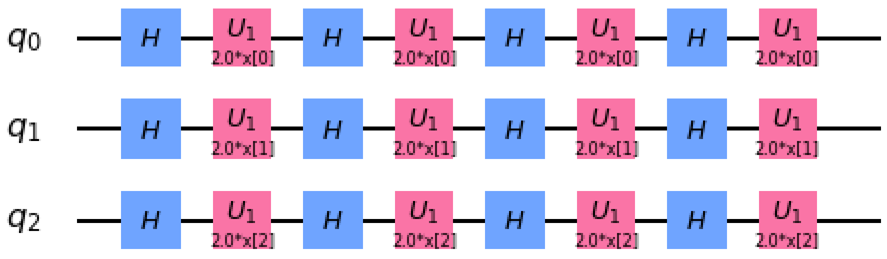
   A quantum feature map is a map from the classical feature vector to the quantum state, a vector in Hilbert space. By applying the unitary operation on the initial state, we have now blown up the dimension of our feature space and the task of our classifier is to find a separating hyperplane in this new space.

   The quantum feature map of depth d is implemented by the unitary operator :

$$\mathcal{U}_{\Phi(\mathbf{x})} = \prod_{d} U_{\Phi(\mathbf{x})} H^{\otimes n},\ U_{\Phi(\mathbf{x})} = \exp\left(i \sum_{S \subseteq [n]} \phi_S(\mathbf{x}) \prod_{k \in S} P_k \right)$$

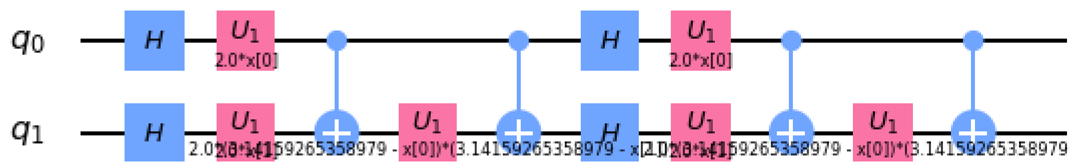
[https://www.researchgate.net/publication/338063278_Separation_of_Multi-mode_Surface_Waves_by_Supervised_Machine_Learning_Methods.]

Different types of feature map: ZfeatureMap, ZZfeatureMap, PaulifeatureMap, etc.
ZfeatureMap for depth = 2:



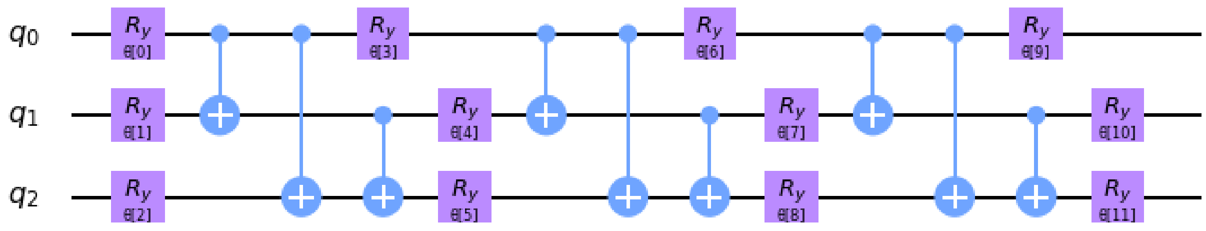
[Vojtech Havlicek, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, Jay M. Gambetta, “Supervised learning with quantum enhanced feature spaces”, https://arxiv.org/abs/1804.11326.]

ZZfeatureMap for depth 2:



[Vojtech Havlicek, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, Jay M. Gambetta, “Supervised learning with quantum enhanced feature spaces”, https://arxiv.org/abs/1804.11326.]

2. Variational Circuit:
   Variational Circuit is the part where the quantum classification takes place. We add a short depth variational circuit to the feature maps. The parameters of this variational circuit are then trained in the classical optimization loop until it classifies data points correctly. This is the learning stage of the algorithm. Accuracy of the model can vary based on the variational circuit one chooses.
   Here is an example of a parameterized variational circuit with interlinking parameterized Ry gates and entangling CNOT units:



[A variational quantum circuit.]

3. Measurement and assigning a binary label:
   An n bit classification string is obtained once we measure the variational quantum circuit using the Measurement operator. The n bit string is now assigned values of the classification classes. This is done with the help of a boolean function $f:0,1^n->0,1$.

4. Classical optimization loop:

The parameters of the quantum variational circuit are updated using a classical optimization routine once the measurements are ready. This is the classical loop that trains our parameters until the cost function’s value decreases. Commonly used optimization methods are as follows: **COBYLA** - Constrained Optimization By Linear Approximation optimizer. **SPSA** - Simultaneous Perturbation Stochastic Approximation (SPSA) optimizer. **SLSQP** - Sequential Least SQuares Programming optimizer, etc.

Quantum Support Vector Machines

In Classical SVMs, a set of data points that belong to one group or another can be classified based on finding a line or hyperplane for higher dimension that separates these two groups. But finding this line or plane can get much more complex with the use of kernels. Steps to perform QSVM:

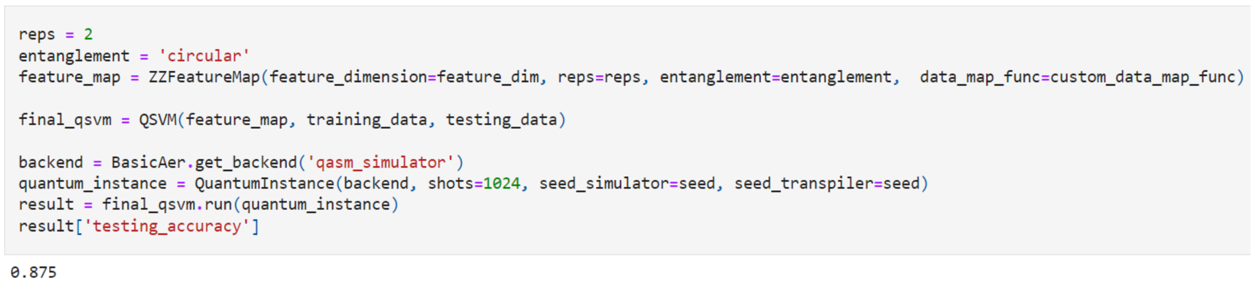
1. Translate the classical data point ***X*** into a quantum datapoint **|**$\Phi(\vec{x})\rangle$. Circuit **V***(*$\Phi(\vec{x})$*)* is used for this process. Where Φ() could be any classical function applied on the classical data $\vec{x}$.
2. Build parameterised quantum circuit ***W****(*Θ*) with parameters* Θ that processes the data.
3. Apply a measurement that returns a classical value ***-1*** or ***1*** for each classical input $\vec{x}$ that identifies the label of the classical data.

Results

C Theory:

QSVM Implementation -

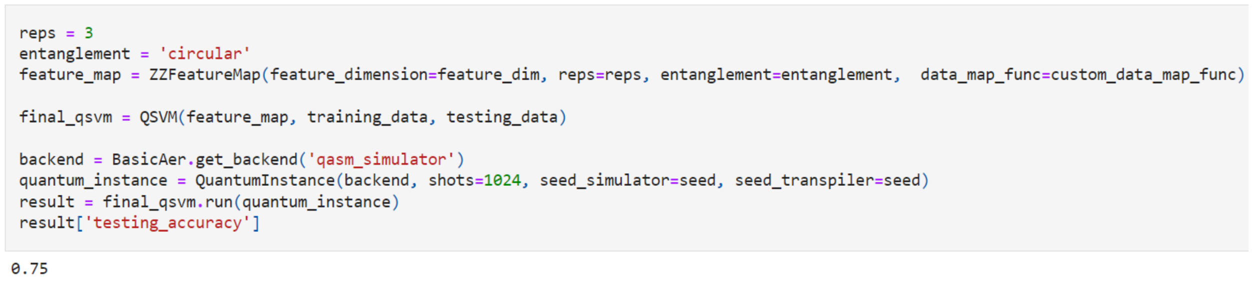
CO1:



```
reps = 2
entanglement = 'circular'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.875
```

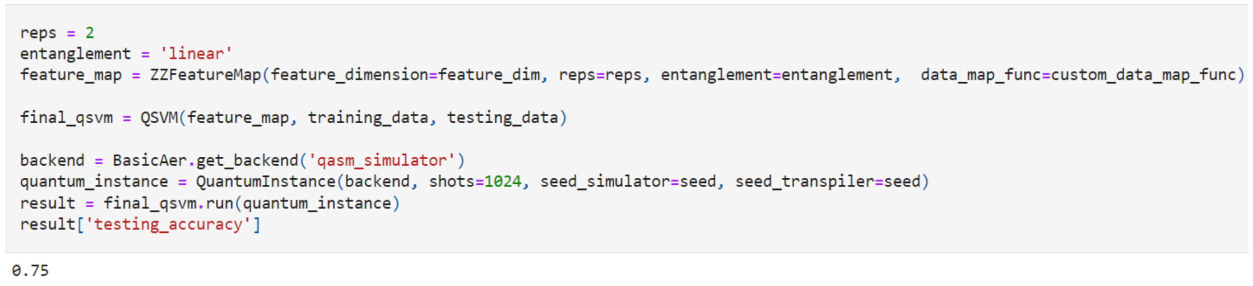
CO2:



```
reps = 3
entanglement = 'circular'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.75
```

CO3:



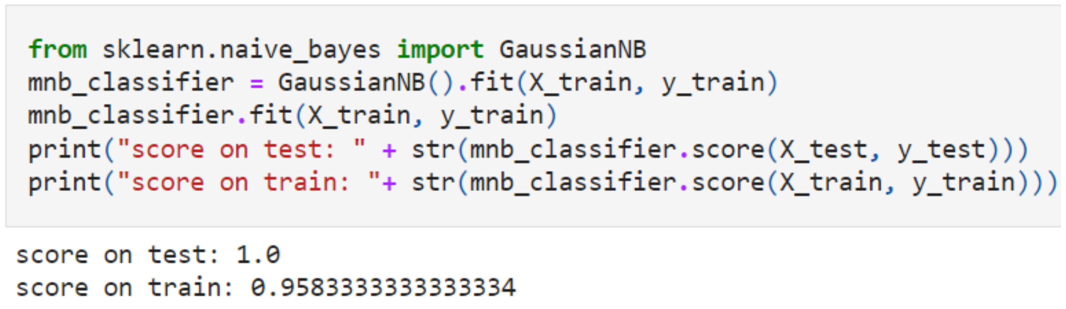
```
reps = 2
entanglement = 'linear'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.75
```

Operating System:

CO1:

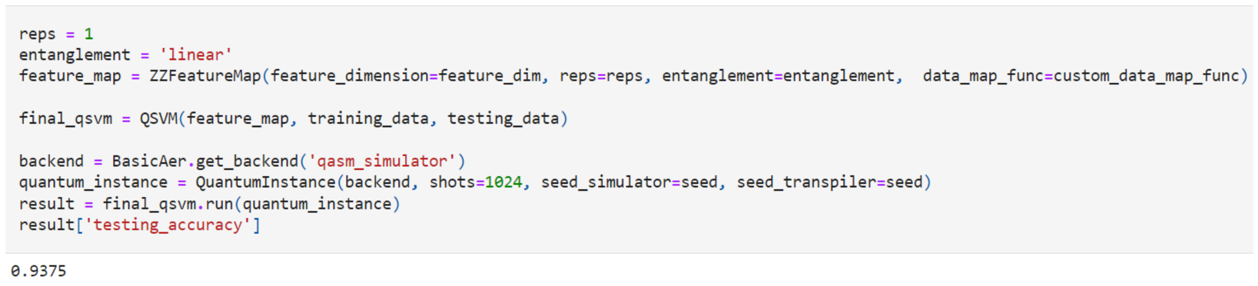
Classical:



```
from sklearn.naive_bayes import GaussianNB
mnb_classifier = GaussianNB().fit(X_train, y_train)
mnb_classifier.fit(X_train, y_train)
print("score on test:  " + str(mnb_classifier.score(X_test, y_test)))
print("score on train:  "+ str(mnb_classifier.score(X_train, y_train)))
```

```
score on test: 1.0
score on train: 0.9583333333333334
```

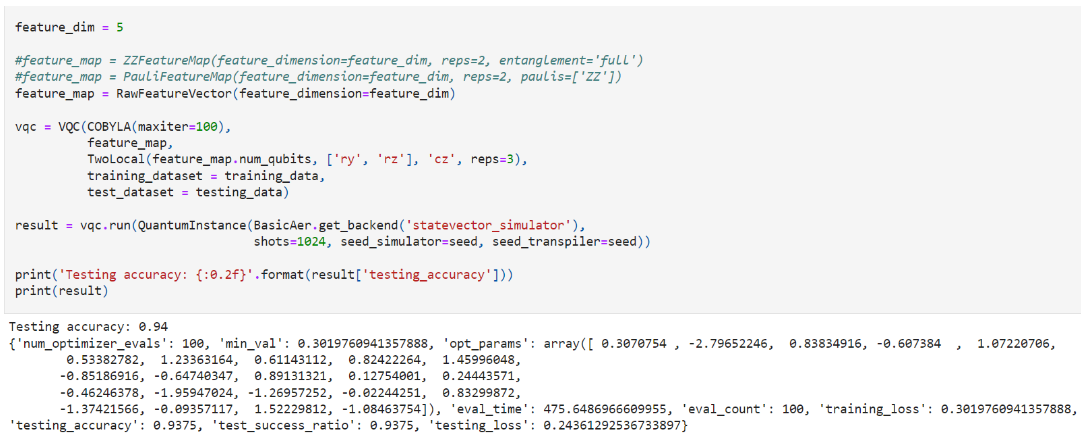
QSVM:



```
reps = 1
entanglement = 'linear'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.9375
```

VQC:

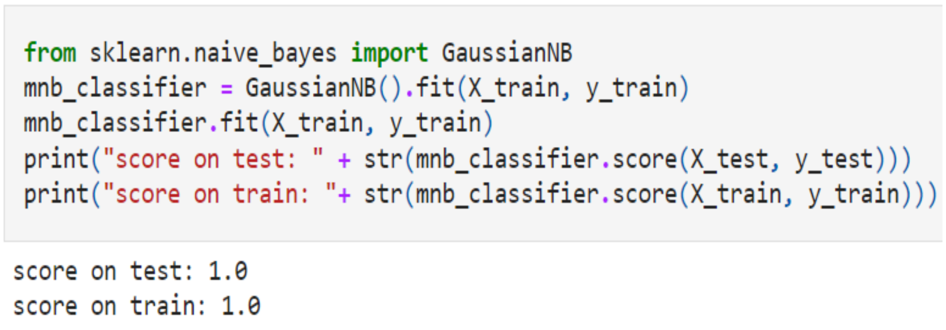


```
feature_dim = 5

#feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='full')
#feature_map = PauliFeatureMap(feature_dimension=feature_dim, reps=2, paulis=['ZZ'])
feature_map = RawFeatureVector(feature_dimension=feature_dim)

vqc = VQC(COBYLA(maxiter=100),
          feature_map,
          TwoLocal(feature_map.num_qubits, ['ry', 'rz'], 'cz', reps=3),
          training_dataset = training_data,
          test_dataset = testing_data)

result = vqc.run(QuantumInstance(BasicAer.get_backend('statevector_simulator'),
                                 shots=1024, seed_simulator=seed, seed_transpiler=seed))

print('Testing accuracy: {:0.2f}'.format(result['testing_accuracy']))
print(result)
```

```
Testing accuracy: 0.94
{'num_optimizer_evals': 100, 'min_val': 0.3019760941357888, 'opt_params': array([ 0.3070754 , -2.79652246,  0.83834916, -0.607384  ,  1.07220706,
        0.53382782,  1.23363164,  0.61143112,  0.82422264,  1.45996048,
       -0.85186916, -0.64740347,  0.89131321,  0.12754001,  0.24443571,
       -0.46246378, -1.95947024, -1.26957252, -0.02244251,  0.83299872,
       -1.37421566, -0.09357117,  1.52229812, -1.08463754]), 'eval_time': 475.6486966609955, 'eval_count': 100, 'training_loss': 0.3019760941357888,
'testing_accuracy': 0.9375, 'test_success_ratio': 0.9375, 'testing_loss': 0.24361292536733897}
```

Both QSVM and VQC performed equally well. Accuracy scores of QSVM and VQC are close to Gaussian Naive Bayes.

CO2:

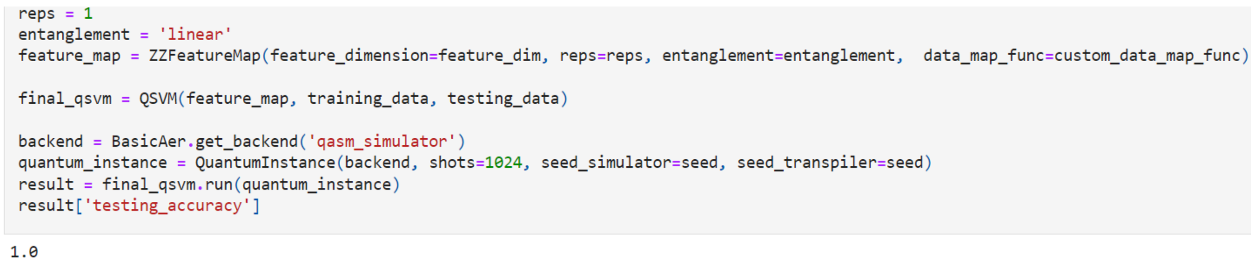


```
from sklearn.naive_bayes import GaussianNB
mnb_classifier = GaussianNB().fit(X_train, y_train)
mnb_classifier.fit(X_train, y_train)
print("score on test: " + str(mnb_classifier.score(X_test, y_test)))
print("score on train: "+ str(mnb_classifier.score(X_train, y_train)))
```

Classical:

```
score on test: 1.0
score on train: 1.0
```

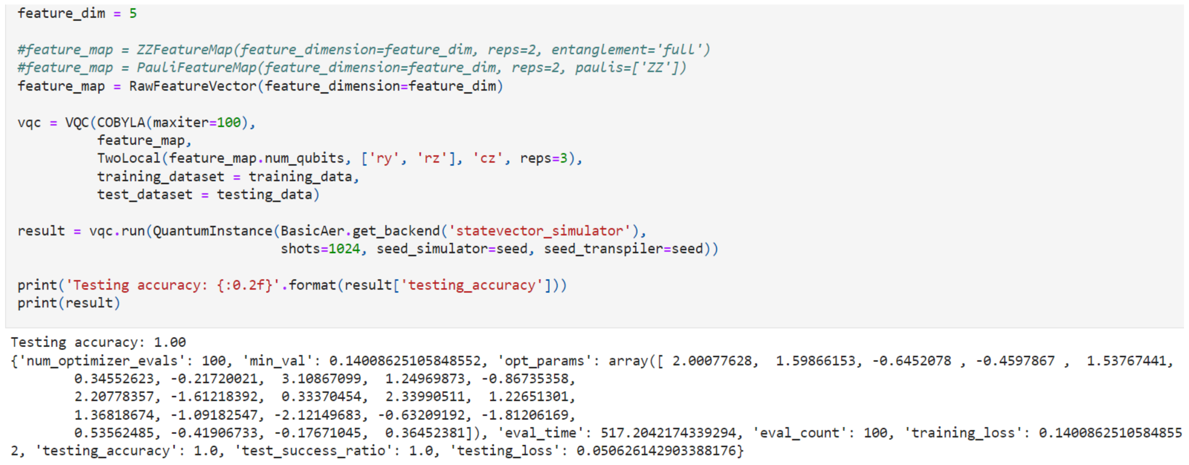
QSVM:



```
reps = 1
entanglement = 'linear'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
1.0
```

VQC:



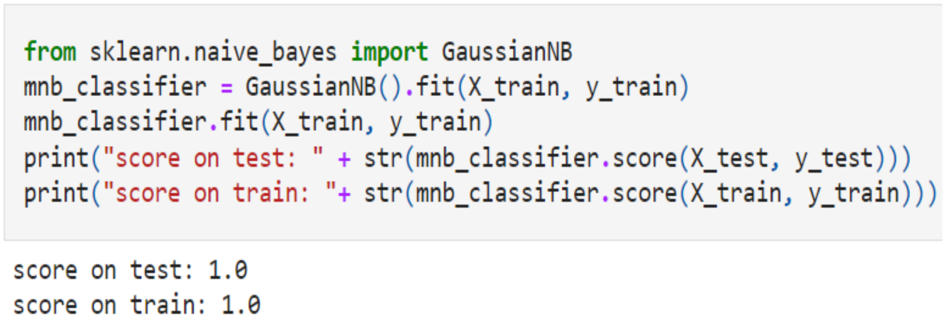
```
feature_dim = 5

#feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='full')
#feature_map = PauliFeatureMap(feature_dimension=feature_dim, reps=2, paulis=['ZZ'])
feature_map = RawFeatureVector(feature_dimension=feature_dim)

vqc = VQC(COBYLA(maxiter=100),
          feature_map,
          TwoLocal(feature_map.num_qubits, ['ry', 'rz'], 'cz', reps=3),
          training_dataset = training_data,
          test_dataset = testing_data)

result = vqc.run(QuantumInstance(BasicAer.get_backend('statevector_simulator'),
                                 shots=1024, seed_simulator=seed, seed_transpiler=seed))

print('Testing accuracy: {:0.2f}'.format(result['testing_accuracy']))
print(result)
```

```
Testing accuracy: 1.00
{'num_optimizer_evals': 100, 'min_val': 0.14008625105848552, 'opt_params': array([ 2.00077628,  1.59866153, -0.6452078 , -0.4597867 ,  1.53767441,
        0.34552623, -0.21720021,  3.10867099,  1.24969873, -0.86735358,
        2.20778357, -1.61218392,  0.33370454,  2.33990511,  1.22651301,
        1.36818674, -1.09182547, -2.12149683, -0.63209192, -1.81206169,
        0.53562485, -0.41906733, -0.17671045,  0.36452381]), 'eval_time': 517.2042174339294, 'eval_count': 100, 'training_loss': 0.1400862510584855
2, 'testing_accuracy': 1.0, 'test_success_ratio': 1.0, 'testing_loss': 0.050626142903388176}
```

Both QSVM and VQC performed equally well with accuracy scores equal to Gaussian Naive Bayes.

CO3:

Classical:



```
from sklearn.naive_bayes import GaussianNB
mnb_classifier = GaussianNB().fit(X_train, y_train)
mnb_classifier.fit(X_train, y_train)
print("score on test: " + str(mnb_classifier.score(X_test, y_test)))
print("score on train: "+ str(mnb_classifier.score(X_train, y_train)))
```

```
score on test: 1.0
score on train: 1.0
```

QSVM:

```
reps = 1
entanglement = 'linear'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.9375
```

VQC:

```
feature_dim = 6

#feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='full')
#feature_map = PauliFeatureMap(feature_dimension=feature_dim, reps=2, paulis=['ZZ'])
feature_map = RawFeatureVector(feature_dimension=feature_dim)

vqc = VQC(COBYLA(maxiter=100),
          feature_map,
          TwoLocal(feature_map.num_qubits, ['ry', 'rz'], 'cz', reps=3),
          training_dataset = training_data,
          test_dataset = testing_data)

result = vqc.run(QuantumInstance(BasicAer.get_backend('statevector_simulator'),
                                 shots=1024, seed_simulator=seed, seed_transpiler=seed))

print('Testing accuracy: {:0.2f}'.format(result['testing_accuracy']))
print(result)
```

```
Testing accuracy: 0.94
{'num_optimizer_evals': 100, 'min_val': 0.29874804010988604, 'opt_params': array([ 2.49944493,  1.95949774, -0.1355361 , -0.62188131,  0.34899018,
        0.17229119,  0.28769093,  1.94394102,  0.65302468,  0.13419275,
        2.82596775, -0.70602283,  0.99864894,  2.14563254,  0.71393098,
        1.45064493,  0.70813079, -1.99846896,  0.45510821, -2.15280316,
        0.94659234, -0.09729128, -0.86747909,  0.82808186]), 'eval_time': 162.50201058387756, 'eval_count': 100, 'training_loss': 0.2987480401098860
4, 'testing_accuracy': 0.9375, 'test_success_ratio': 0.9375, 'testing_loss': 0.293003667714507}
```

Both QSVM and VQC performed equally well. Accuracy scores of QSVM and VQC are close to Gaussian Naive Bayes.

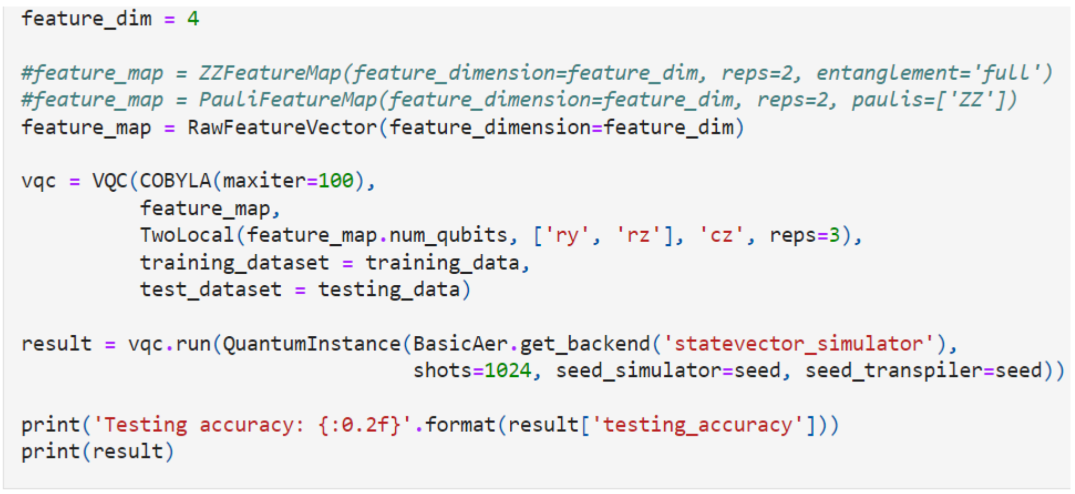
Programming C:

CO1:

QSVM:

```
reps = 1
entanglement = 'linear'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.75
```

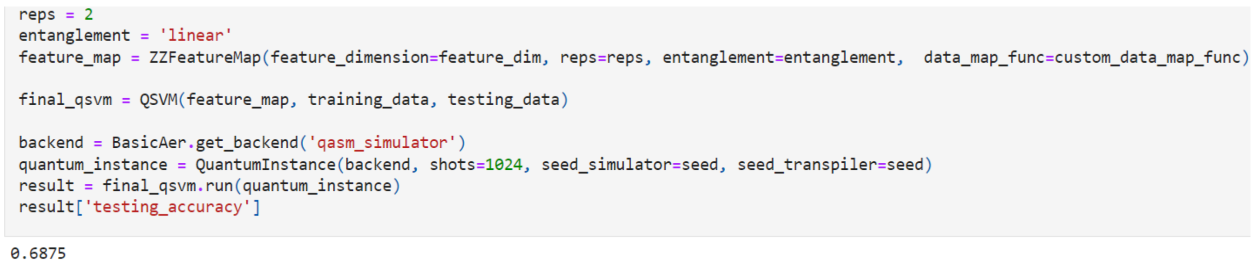


```python
feature_dim = 4

#feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='full')
#feature_map = PauliFeatureMap(feature_dimension=feature_dim, reps=2, paulis=['ZZ'])
feature_map = RawFeatureVector(feature_dimension=feature_dim)

vqc = VQC(COBYLA(maxiter=100),
          feature_map,
          TwoLocal(feature_map.num_qubits, ['ry', 'rz'], 'cz', reps=3),
          training_dataset = training_data,
          test_dataset = testing_data)

result = vqc.run(QuantumInstance(BasicAer.get_backend('statevector_simulator'),
                                 shots=1024, seed_simulator=seed, seed_transpiler=seed))

print('Testing accuracy: {:0.2f}'.format(result['testing_accuracy']))
print(result)
```

VQC:
```
Testing accuracy: 0.75
```

CO2:

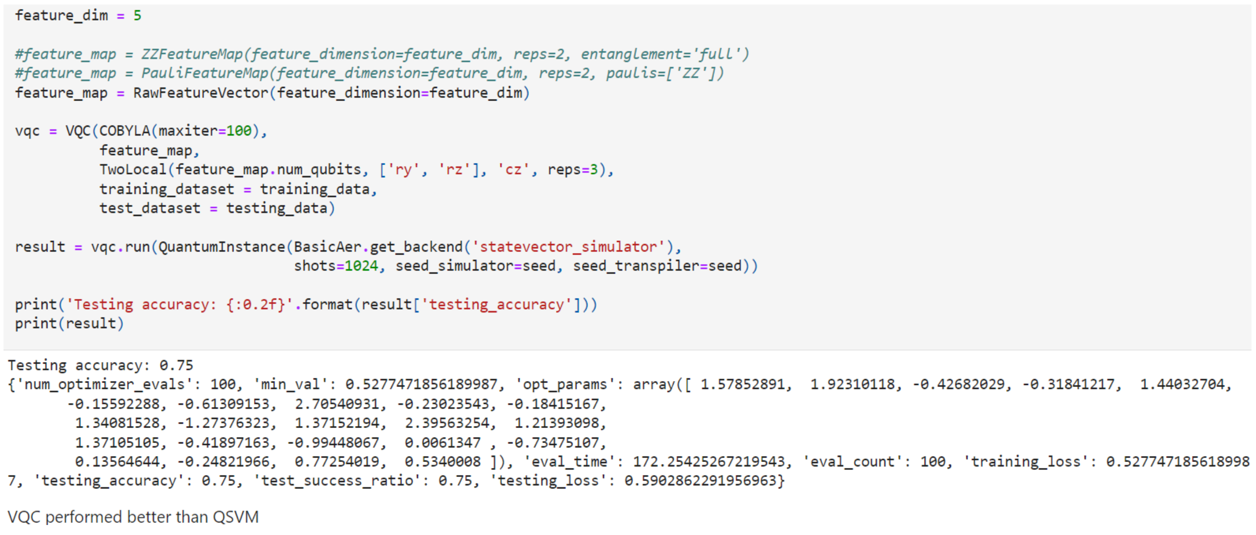
QSVM:



```python
reps = 2
entanglement = 'linear'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.6875
```

VQC:



```python
feature_dim = 5

#feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='full')
#feature_map = PauliFeatureMap(feature_dimension=feature_dim, reps=2, paulis=['ZZ'])
feature_map = RawFeatureVector(feature_dimension=feature_dim)

vqc = VQC(COBYLA(maxiter=100),
          feature_map,
          TwoLocal(feature_map.num_qubits, ['ry', 'rz'], 'cz', reps=3),
          training_dataset = training_data,
          test_dataset = testing_data)

result = vqc.run(QuantumInstance(BasicAer.get_backend('statevector_simulator'),
                                 shots=1024, seed_simulator=seed, seed_transpiler=seed))

print('Testing accuracy: {:0.2f}'.format(result['testing_accuracy']))
print(result)
```

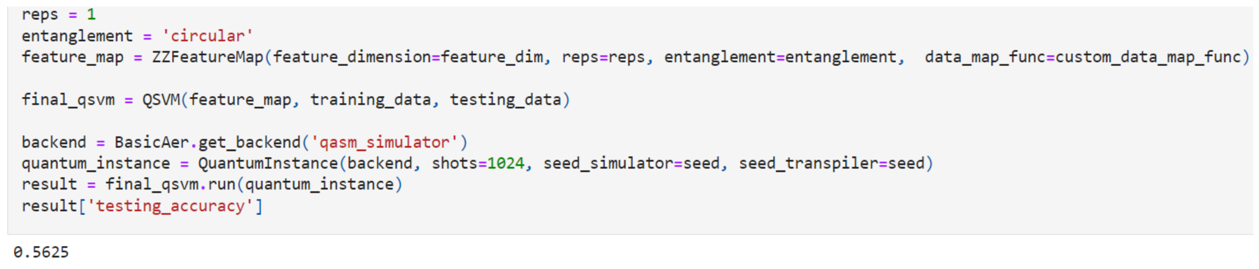
```
Testing accuracy: 0.75
{'num_optimizer_evals': 100, 'min_val': 0.5277471856189987, 'opt_params': array([ 1.57852891,  1.92310118, -0.42682029, -0.31841217,  1.44032704,
       -0.15592288, -0.61309153,  2.70540931, -0.23023543, -0.18415167,
        1.34081528, -1.27376323,  1.37152194,  2.39563254,  1.21393098,
        1.37105105, -0.41897163, -0.99448067,  0.0061347 , -0.73475107,
        0.13564644, -0.24821966,  0.77254019,  0.5340008 ]), 'eval_time': 172.25425267219543, 'eval_count': 100, 'training_loss': 0.527747185618998
7, 'testing_accuracy': 0.75, 'test_success_ratio': 0.75, 'testing_loss': 0.5902862291956963}
```

VQC performed better than QSVM

VQC performed better than QSVM.

CO3:

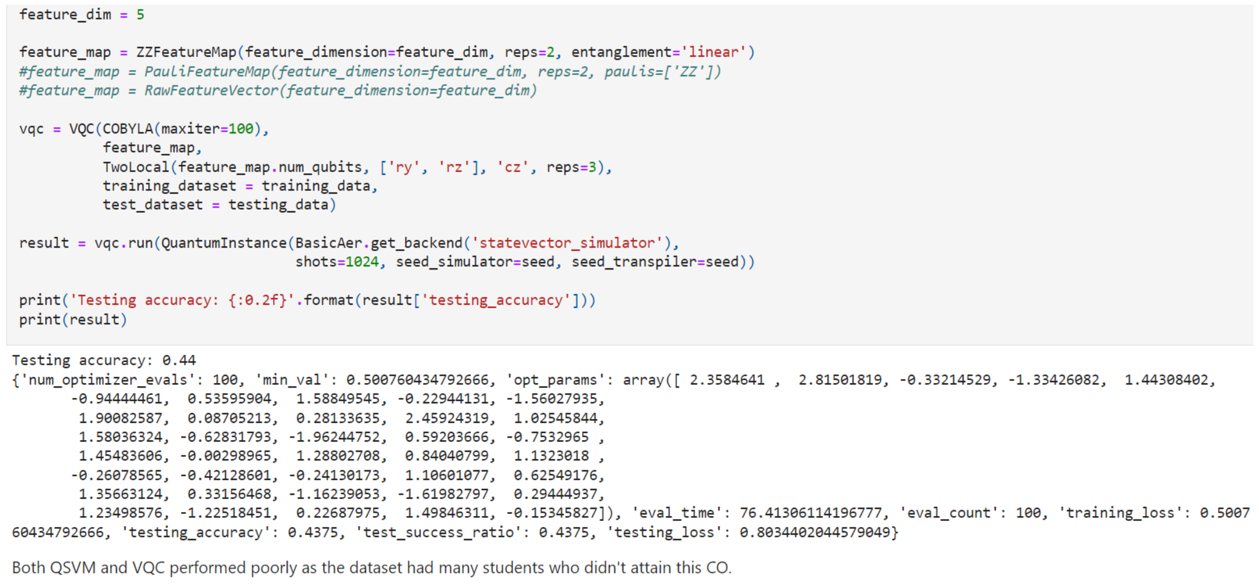
QSVM:



```
reps = 1
entanglement = 'circular'
feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=reps, entanglement=entanglement,  data_map_func=custom_data_map_func)

final_qsvm = QSVM(feature_map, training_data, testing_data)

backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024, seed_simulator=seed, seed_transpiler=seed)
result = final_qsvm.run(quantum_instance)
result['testing_accuracy']
```

```
0.5625
```

VQC:



```
feature_dim = 5

feature_map = ZZFeatureMap(feature_dimension=feature_dim, reps=2, entanglement='linear')
#feature_map = PauliFeatureMap(feature_dimension=feature_dim, reps=2, paulis=['ZZ'])
#feature_map = RawFeatureVector(feature_dimension=feature_dim)

vqc = VQC(COBYLA(maxiter=100),
          feature_map,
          TwoLocal(feature_map.num_qubits, ['ry', 'rz'], 'cz', reps=3),
          training_dataset = training_data,
          test_dataset = testing_data)

result = vqc.run(QuantumInstance(BasicAer.get_backend('statevector_simulator'),
                                 shots=1024, seed_simulator=seed, seed_transpiler=seed))

print('Testing accuracy: {:0.2f}'.format(result['testing_accuracy']))
print(result)
```

```
Testing accuracy: 0.44
{'num_optimizer_evals': 100, 'min_val': 0.500760434792666, 'opt_params': array([ 2.3584641 ,  2.81501819, -0.33214529, -1.33426082,  1.44308402,
       -0.94444461,  0.53595904,  1.58849545, -0.22944131, -1.56027935,
        1.90082587,  0.08705213,  0.28133635,  2.45924319,  1.02545844,
        1.58036324, -0.62831793, -1.96244752,  0.59203666, -0.7532965 ,
        1.45483606, -0.00298965,  1.28802708,  0.84040799,  1.1323018 ,
       -0.26078565, -0.42128601, -0.24130173,  1.10601077,  0.62549176,
        1.35663124,  0.33156468, -1.16239053, -1.61982797,  0.29444937,
        1.23498576, -1.22518451,  0.22687975,  1.49846311, -0.15345827]), 'eval_time': 76.41306114196777, 'eval_count': 100, 'training_loss': 0.5007
60434792666, 'testing_accuracy': 0.4375, 'test_success_ratio': 0.4375, 'testing_loss': 0.8034402044579049}
```

Both QSVM and VQC performed poorly as the dataset had many students who didn't attain this CO.

Future Scope/ Further Directions

- Implementation of custom VQC
- QSVM and VQC performed well for consistent dataset, whereas performed poorly on imbalanced data classification. So, finding a solution to this or using a different quantum approach for this case.
- Implementing and testing these algorithms on Big Data.
- Working on building quantum solutions for multi-class classification.

Conclusion

With the current quantum computers and its performance on real world dataset isn't the best. Although performance of QSVM and VQC in some cases was close to the classical ML model, it didn't surpass the classical ML model. There is no proof yet that QSVM brings quantum advantage. With the results obtained we observe that feature maps that are easy to simulate classically showed no advantage in quantum counterpart. There were cases in the results where both QSVM and VQC performed equally well to the classical ML model showing that quantum algorithms have potential in the future.