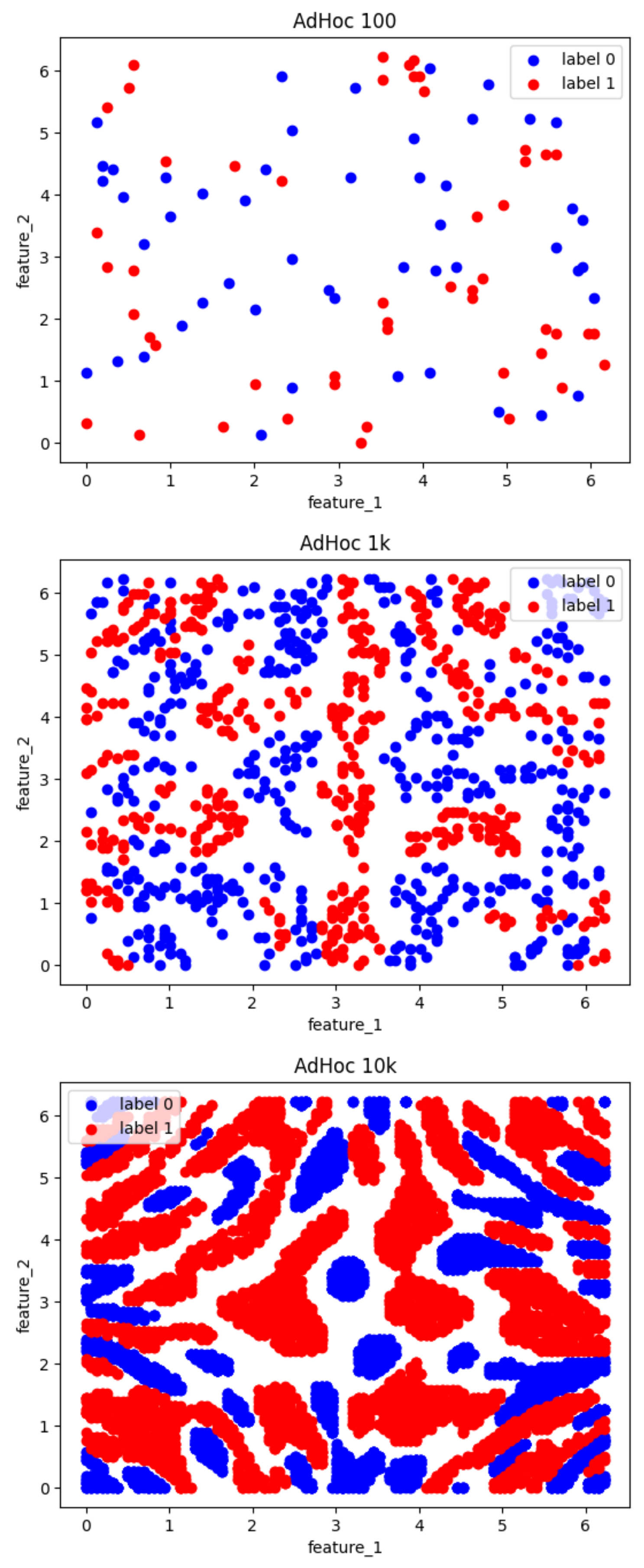
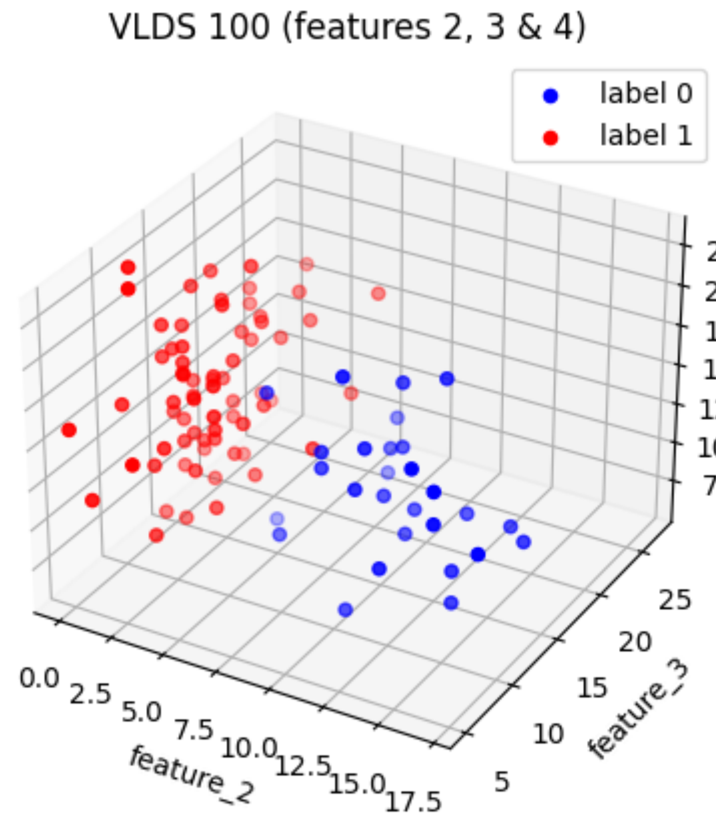
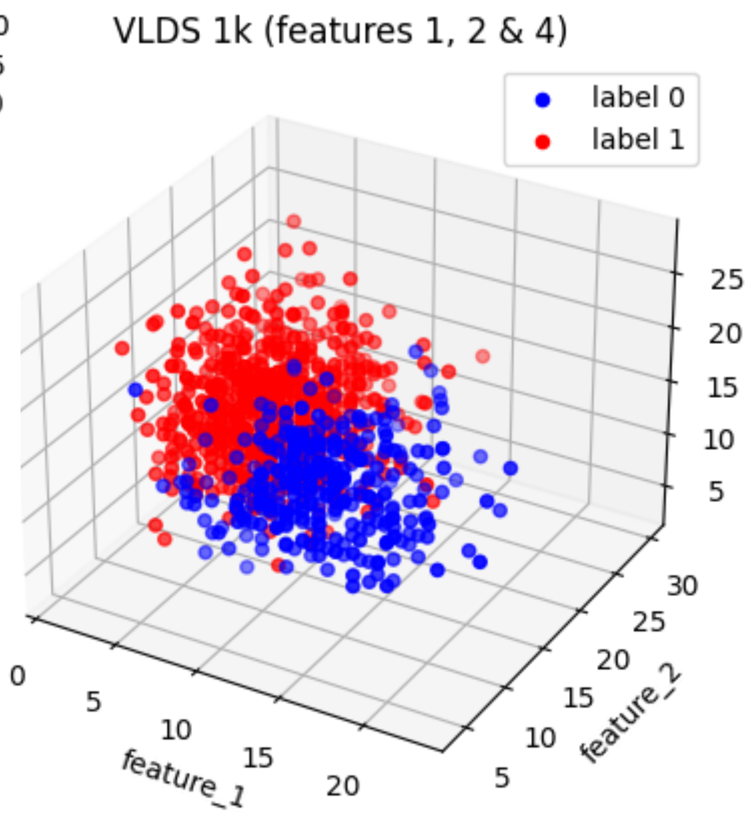
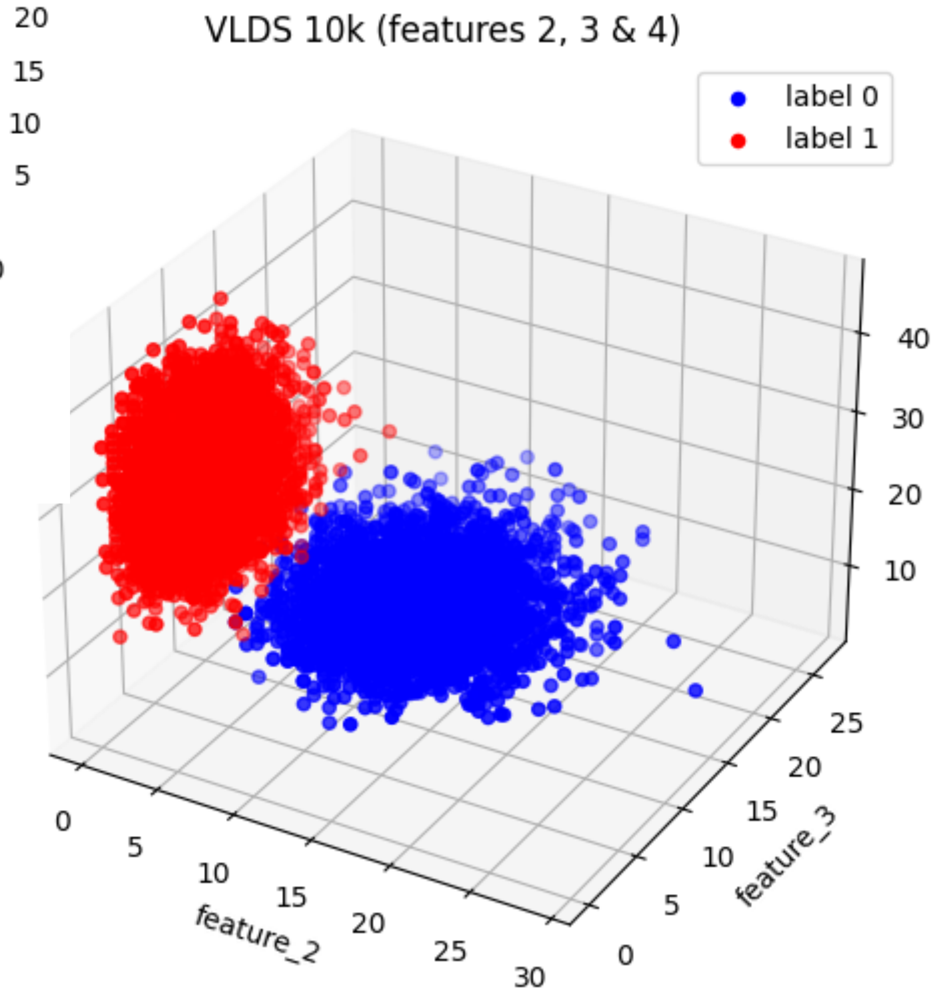
# Evaluation of QML algorithms

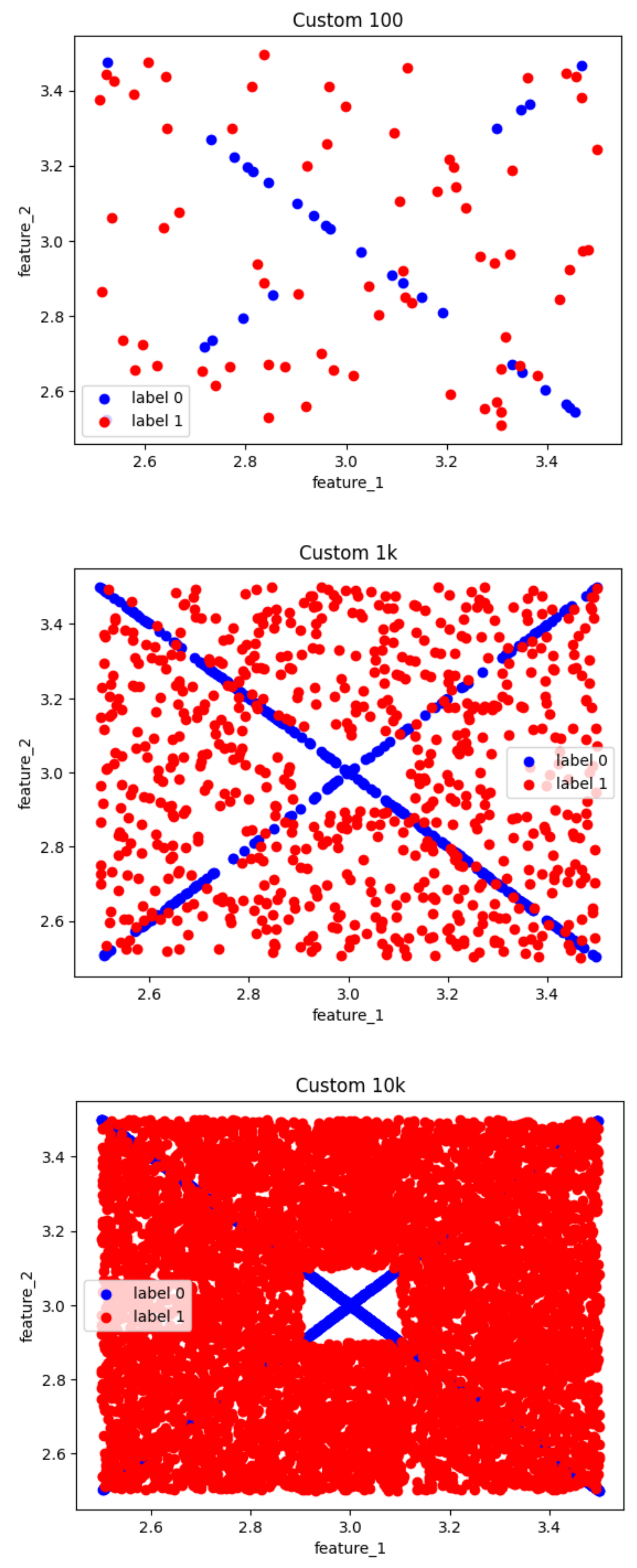
## Using the following datasets (100, 1k, 10k): Adhoc, VLDS, Custom











**Lukas Bischof, Stefan Teodoropol, Karin Birle – ZHAW Quantum Information Science – Semester Project 2022/2023**

*Code available on Github: [https://github.com/lukasbischof/qi-project](https://github.com/lukasbischof/qi-project)*

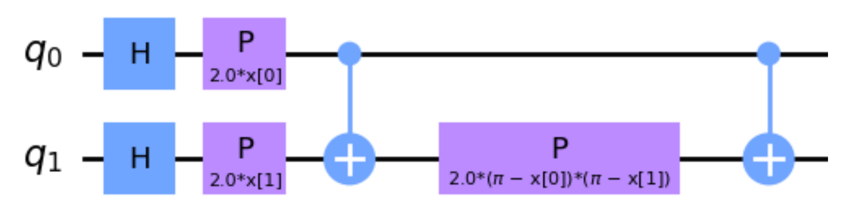
# Quantum Support Vector Machine

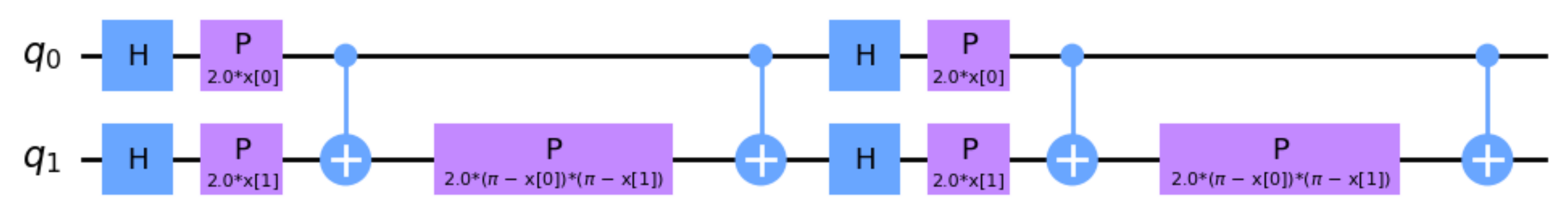
## Implementation

```python
kernels = [
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=1, entanglement='linear')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=2, entanglement='linear')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=3, entanglement='linear')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=4, entanglement='linear')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=1, entanglement='sca')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=2, entanglement='sca')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=3, entanglement='sca')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=(ZZFeatureMap(feature_dimension=feature_dimension, reps=4, entanglement='sca')),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=PauliFeatureMap(feature_dimension=feature_dimension, reps=1, entanglement='linear', paulis=['ZZ']),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=PauliFeatureMap(feature_dimension=feature_dimension, reps=1, entanglement='linear', paulis=['Z', 'XX']),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=PauliFeatureMap(feature_dimension=feature_dimension, reps=2, entanglement='linear', paulis=['ZZ']),
                  quantum_instance=quantum_instance),
    QuantumKernel(feature_map=PauliFeatureMap(feature_dimension=feature_dimension, reps=2, entanglement='linear', paulis=['Z', 'XX']),
                  quantum_instance=quantum_instance),
    FidelityQuantumKernel(feature_map=ZZFeatureMap(feature_dimension=feature_dimension, reps=2, entanglement='linear'),
                          fidelity=ComputeUncompute(sampler=Sampler())),
    FidelityQuantumKernel(feature_map=ZZFeatureMap(feature_dimension=feature_dimension, reps=3, entanglement='linear'),
                          fidelity=ComputeUncompute(sampler=Sampler())),
    FidelityQuantumKernel(feature_map=PauliFeatureMap(feature_dimension=feature_dimension, reps=1, entanglement='linear', paulis=['Z', 'XX']),
                          fidelity=ComputeUncompute(sampler=Sampler())),
    FidelityQuantumKernel(feature_map=PauliFeatureMap(feature_dimension=feature_dimension, reps=1, entanglement='linear', paulis=['Z', 'XX']),
                          fidelity=ComputeUncompute(sampler=Sampler()))
]
```

```python
svc = SVC()
search = RandomizedSearchCV(svc, cv=10, n_iter=16, n_jobs=-1, refit=True,
                            param_distributions={'kernel': [kernel.evaluate for kernel in kernels]})
search.fit(train_features, train_labels)
```

- We used ***RandomizedSearchCV*** to find the best suited kernel for each data set containing 100 entries.
- For the subsequent training periods, we used the same kernel again. It turned out, that the normal QuantumKernel using a ZZFeatureMap worked best for all datasets - however the entanglement and the repetitions varied:
- 2 reps' and ***linear*** entanglement for Adhoc,
  4 reps' and '***sca***' entanglement for VLDS,
  1 rep' with ***linear*** entanglement for the custom dataset.
- The custom dataset kernel (left) and the adhoc kernel (right) are displayed below. The VLDS kernel was too complicated to include in this presentation, but it's plotted in the notebook and can be found in the GIT repository.





The data was split into 75% training data, 25% test data. Training was then done using ***cross_validate(...)*** with a split of 10 (10-fold cross validation) on the training data. The scores were finally determined using the 25% unseen test data.
This step was the same for all QM algorithms (QSVC, HQNN and QNN).

```python
best_kernel = kernels[0] # Index from output above
svc = SVC(kernel=best_kernel.evaluate)

results = cross_validate(svc, train_features, train_labels, cv=10, n_jobs=-1, return_estimator=True, return_train_score=True)
```

```python
from qiskit import IBMQ

# Best kernel should already be evaluated and set (in the above code cell)

IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q-education', group='zhaw-1')
backend = least_busy(provider.backends(simulator=False))
print(f"Chosen backend: {backend}")
best_model = resulting_models[np.argmax(accuracies)]
best_model.kernel.__self__.quantum_instance = QuantumInstance(backend, shots=1024)
score = best_model.score(test_features, test_labels)
print(f"Quantum score: {score}")

```

For the evaluation on the quantum computer, we switched the backend provider of the kernel before running the scoring method.

# Quantum Support Vector Machine

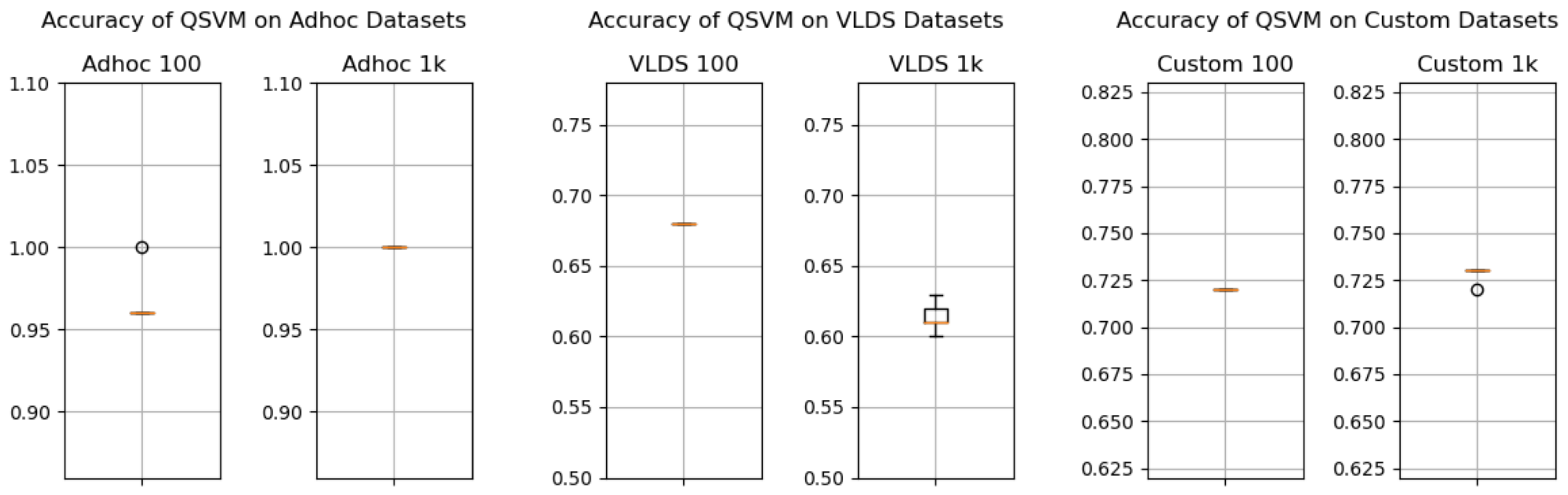
## Accuracies and Results using 10-fold cross validation

| Accuracy | AER Simulator | Quantum Computer (*) | Classical SVM |
|---|---|---|---|
| **Adhoc 100** | 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 1.00, 0.96, 0.96, 0.96 mean: 0.96, std: 0.01, mean training time: 6s | 0.96, 0.96, 0.96, 0.96, 0.96 mean: 0.96, std: 0.00, mean training time: 187s (***) | 0.52, 0.56, 0.56, 0.44, 0.60, 0.44, 0.52, 0.52, 0.60, 0.60 mean: 0.54, std: 0.06, mean training time: 6s |
| **Adhoc 1k** | 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00 mean: 1.00, std: 0.00, mean training time: 15028s | 0.98 | 0.57, 0.60, 0.56, 0.59, 0.56, 0.61, 0.60, 0.60, 0.56, 0.56 mean: 0.58, std: 0.02, mean training time: 7s |
| **Adhoc 10k** | n/a (**) | n/a (**) | 0.66, 0.66, 0.66, 0.66, 0.66, 0.67, 0.67, 0.66, 0.66, 0.66 mean: 0.66, std: 0.00, mean training time: 1s |
| **VLDS 100** | 0.68, 0.68, 0.68, 0.68, 0.68, 0.68, 0.68, 0.68, 0.68, 0.68 mean: 0.68, std: 0.00, mean training time: 16s | 0.68 | 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96 mean: 0.96, std: 0.00, mean training time: 6s |
| **VLDS 1k** | 0.61, 0.62, 0.63, 0.63, 0.62, 0.60, 0.61, 0.61, 0.61, 0.60 mean: 0.61, std: 0.01, mean training time: 2087s | 0.636 | 0.85, 0.85, 0.85, 0.86, 0.85, 0.86, 0.85, 0.86, 0.84, 0.87 mean: 0.85, std: 0.01, mean training time: 1s |
| **VLDS 10k** | n/a (**) | n/a (**) | 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00 mean: 1.00, std: 0.00, mean training time: 1s |
| **Custom 100** | 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72 mean: 0.72, std: 0.00, mean training time: 6s | 0.72 | 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72 mean: 0.72, std: 0.00, mean training time: 6s |
| **Custom 1k** | 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.72, 0.73, 0.73 mean: 0.73, std: 0.00, mean training time: 6384s | 0.728 | 0.72, 0.72, 0.70, 0.72, 0.70, 0.72, 0.70, 0.71, 0.72, 0.72 mean: 0.71, std: 0.01, mean training time: 7s |
| **Custom 10k** | n/a (**) | n/a (**) | 0.78, 0.78, 0.78, 0.78, 0.78, 0.78, 0.79, 0.79, 0.78, 0.78 mean: 0.78, std: 0.00, mean training time: 1s |

(*) We evaluated the best performing simulator-trained model on the quantum computer using the 25% test data.
(**) We were unable to train any 10k dataset as the execution took too long. The longest attempt for a 10k data set was on the custom 10k set where the training algorithm was running for 16h without completing and since it used up all CPU cores, the computer was unusable during training so we couldn't train for longer than a night.
(***) This specific model was trained and evaluated on a quantum computer (using 5-fold cross validation to speed up validation).

# Quantum Support Vector Machine

## Accuracy Boxplots



**Results QSVM:**

**Adhoc:** The QSVM worked really well on the Adhoc dataset compared to the classical approach. This probably can be explained by the “island”-style data distribution of the dataset. A classical kernel has a hard time to extend this to multiple dimensions in order to learn a linear separation. The quantum kernel reflected the data much better and helped to improve accuracy significantly over the classical approach.
**VLDS, Custom:** For the other datasets, we found no real benefit of using a quantum approach over the classical approach, especially considering the much longer training and evaluation times. For the custom dataset, the accuracy was on par with the classical SVM and for the VLDS dataset it was even worse. Looking at the 3D-plot of the VLDS dataset, we can assume a possible explanation for this: When reducing the dataset to three dimensions, it can be easily separated using a plane. Here, the classical kernel approach had an advantage over the stochastic, entangled approach of the quantum kernel.

# Hybrid Neural Network

## Implementation

```
class Net(nn.Module):
    def __init__(self, feature_dimension, backend):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(feature_dimension, 64)
        self.fc2 = nn.Linear(64, 1)
        self.hybrid = Hybrid(backend, shots=100, shift=np.pi / 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.hybrid(x)
        return torch.cat((x, 1 - x), -1)
```

The hybrid neural network (***Net*** class on the left) consisted of classical layers (in this case, two dense layers which had a single output) and a quantum layer which in the end decided the output.

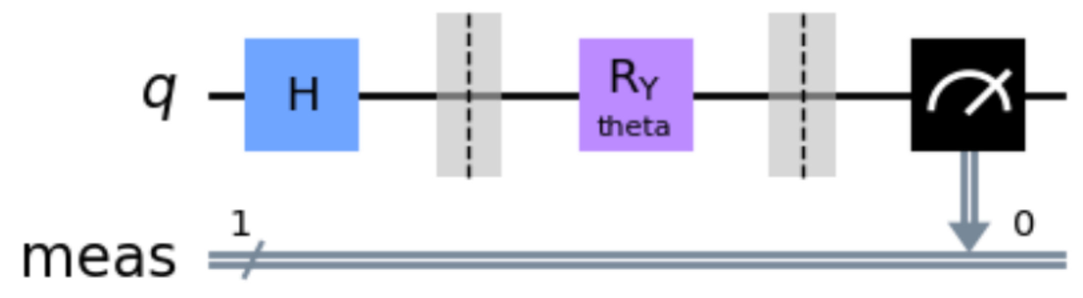
The quantum layer (code on the right, image below) was in this case a simple variational Quantum Circuit which provided a rotation parameter. The implementation of more parameters would have been interesting but required a much better understanding of Pytorch tensors and learning methods.

```
self._circuit = qiskit.QuantumCircuit(n_qubits)

all_qubits = [i for i in range(n_qubits)]
self.theta = qiskit.circuit.Parameter('theta')

self._circuit.h(all_qubits)
self._circuit.barrier()
self._circuit.ry(self.theta, all_qubits)

self._circuit.measure_all()
```

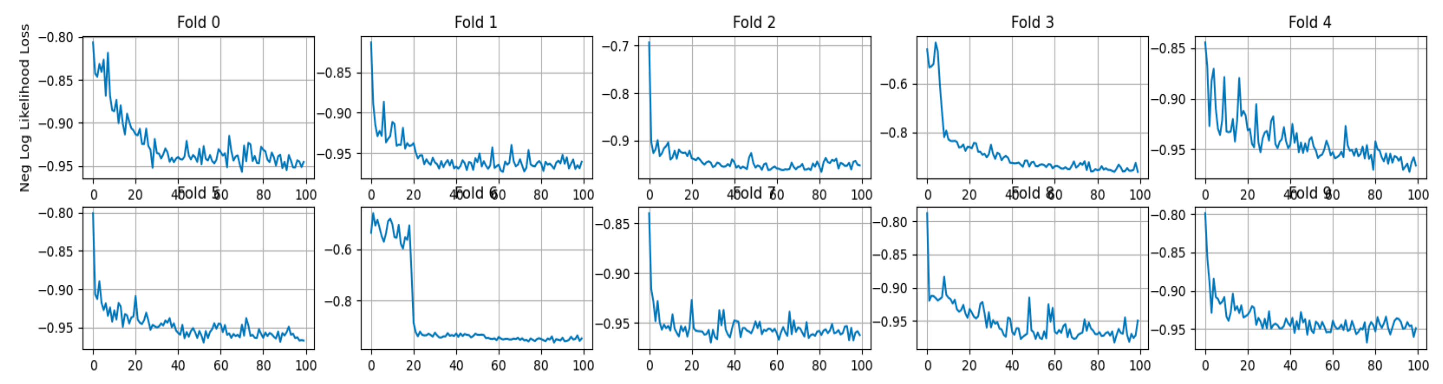


```
if not all(model.loaded for model in models):
    k_fold = KFold(n_splits=10)
    with concurrent.futures.ProcessPoolExecutor() as executor:
        futures = []
        for i, (train_index, test_index) in enumerate(k_fold.split(train_features)):
            model = models[i]
            if model.loaded:
                continue

            current_train_features = train_features.iloc[train_index]
            current_train_labels = train_labels.iloc[train_index]
            train_loader = torch.utils.data.DataLoader(MyDataset(current_train_features, current_train_labels),
                                                       batch_size=1, shuffle=True)
            futures.append(executor.submit(model.fit, train_loader))
        executor.shutdown(wait=True)
        models = [future.result() for future in futures]
```

The model was then trained using 10-fold cross validation on the CPU. The folds were chosen randomly so running the training methods caused slight fluctuations in the mean accuracy over all folds. Probably some fluctuations are also due to the stochastic nature of the quantum layer.

Loss function plot for adhoc 100 dataset



# Hybrid Neural Network

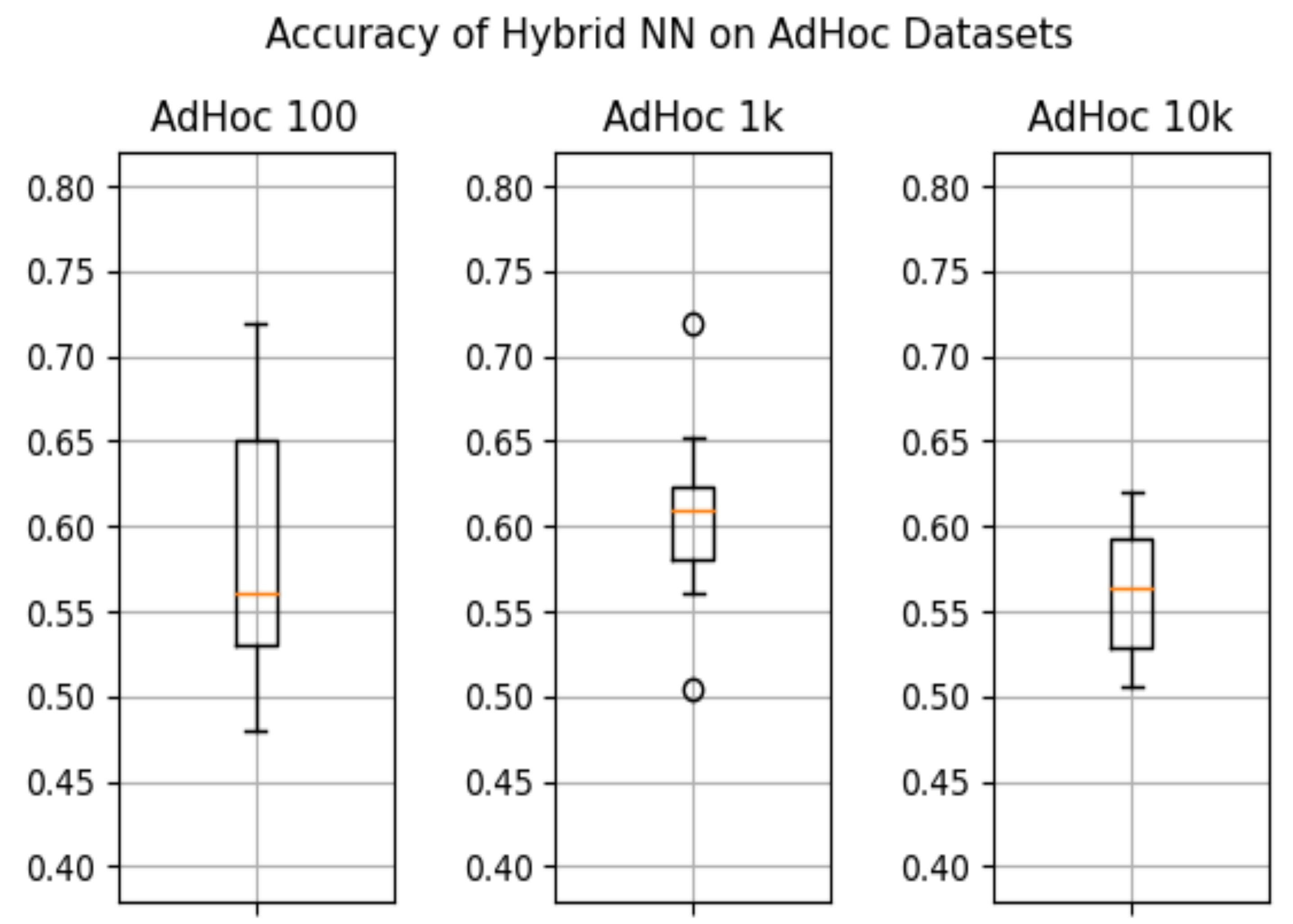
## Accuracies and Results using 10-fold cross validation

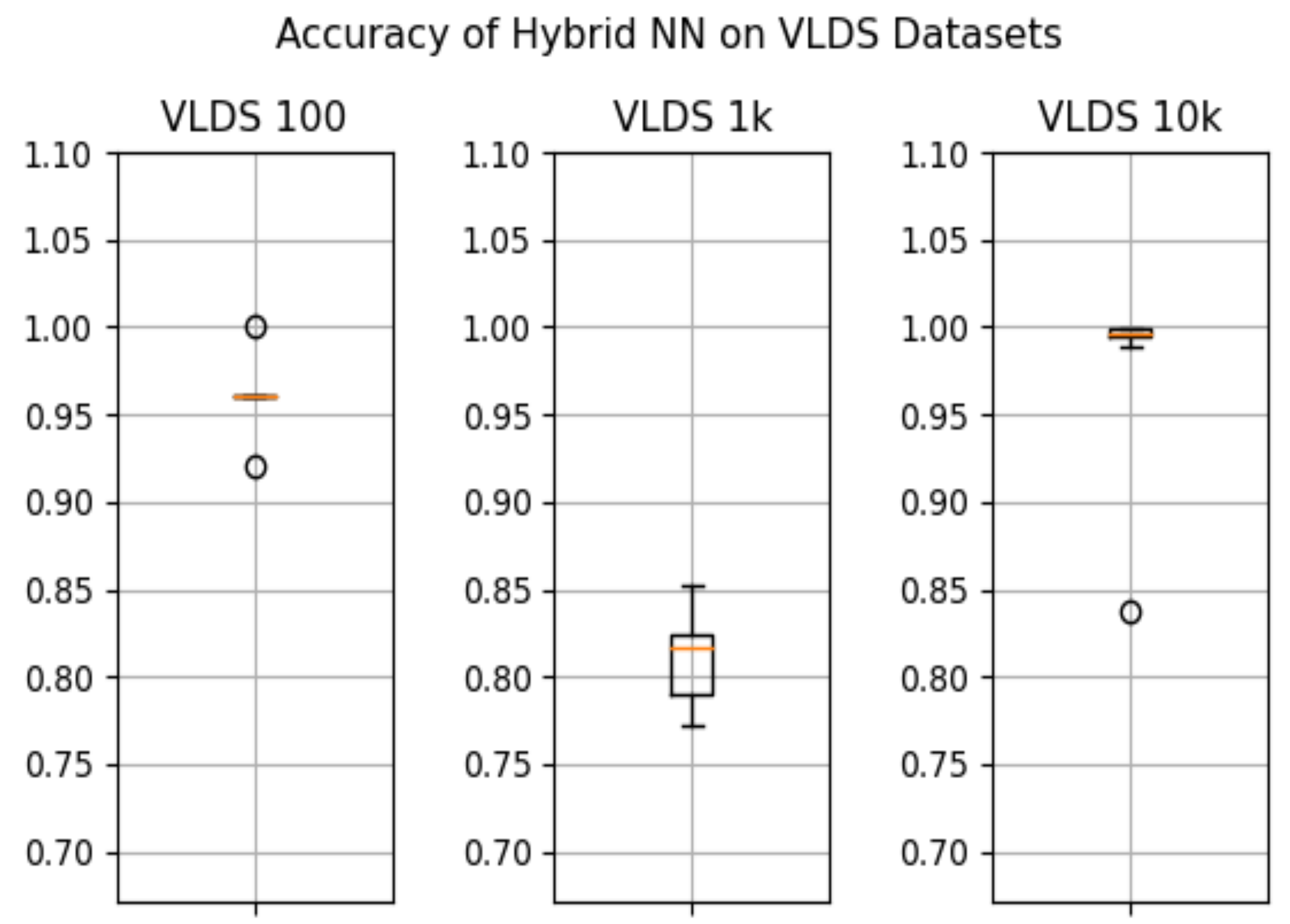
| Accuracy | AER Simulator | Quantum Computer (*) | Classical Neural Network |
|---|---|---|---|
| **Adhoc 100** | 0.56, 0.52, 0.68, 0.48, 0.72, 0.56, 0.56, 0.52, 0.68, 0.56<br>mean: 0.58, std: 7.6, mean training time: 167s | 0.9 | 0.56, 0.56, 0.56, 0.52, 0.56, 0.56, 0.52, 0.56, 0.60, 0.60<br>mean: 0.56, std: 0.02, mean training time: 0.55 |
| **Adhoc 1k** | 0.62, 0.56, 0.62, 0.592, 0.576, 0.504, 0.60, 0.72, 0.652, 0.624<br>mean: 0.606, std: 5.43, mean training time: 1459s | 0.8 | 0.54, 0.54, 0.55, 0.57, 0.56, 0.55, 0.54, 0.54, 0.54, 0.51<br>mean: 0.54, std: 0.01, mean training time: 2s |
| **Adhoc 10k** | 0.62, 0.538, 0.576, 0.593, 0.506, 0.527, 0.552, 0.603, 0.523, 0.59<br>mean: 0.562, std: 3.7, mean training time: 6763s | 0.8 | 0.65, 0.62, 0.64, 0.64, 0.63, 0.63, 0.66, 0.60, 0.65, 0.62<br>mean: 0.64, std: 0.02, mean training time: 17s |
| **VLDS 100** | 1.00, 0.96, 0.96, 0.96, 0.96, 0.92, 0.96, 0.96, 0.90, 0.96<br>mean: 0.96, std: 1.79, mean training time: 77s | 0.8 | 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96<br>mean: 0.96, std: 0.00, mean training time: 1s |
| **VLDS 1k** | 0.816, 0.784, 0.824, 0.852, 0.808, 0.816, 0.824, 0.7840, 0.836, 0.772<br>mean: 0.811, std: 2.38, mean training time: 960s | 0.8 | 0.85, 0.86, 0.85, 0.86, 0.84, 0.85, 0.86, 0.84, 0.86, 0.86<br>mean: 0.85, std: 0.01, mean training time: 2s |
| **VLDS 10k** | 0.998, 0.994, 0.836, 0.989, 0.994, 0.995, 0.998, 0.998, 0.998, 0.998<br>mean: 0.98, std: 4.79, mean training time: 6656s | 0.9 | 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00<br>mean: 1.00, std: 0.0, mean training time: 17s |
| **Custom 100** | 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72<br>mean: 0.72, std: 0.0, mean training time: 77s | 0.8 | 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72<br>mean: 0.72, std: 0.0, mean training time: 1s |
| **Custom 1k** | 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728<br>mean: 0.728, std: 0.0, mean training time: 958s | 0.7 | 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73<br>mean: 0.73, std: 0.00, mean training time: 2s |
| **Custom 10k** | 0.7032, 0.7032, 0.7032, 0.7032, 0.7032, 0.7032, 0.7032, 0.7032, 0.7032, 0.7032<br>mean: 0.7032, std: 0.0, mean training time: 9622s | 0.6 | 0.70, 0.72, 0.72, 0.72, 0.71, 0.71, 0.71, 0.72, 0.71, 0.71<br>mean: 0.71, std: 0.006, mean training time: 18s |

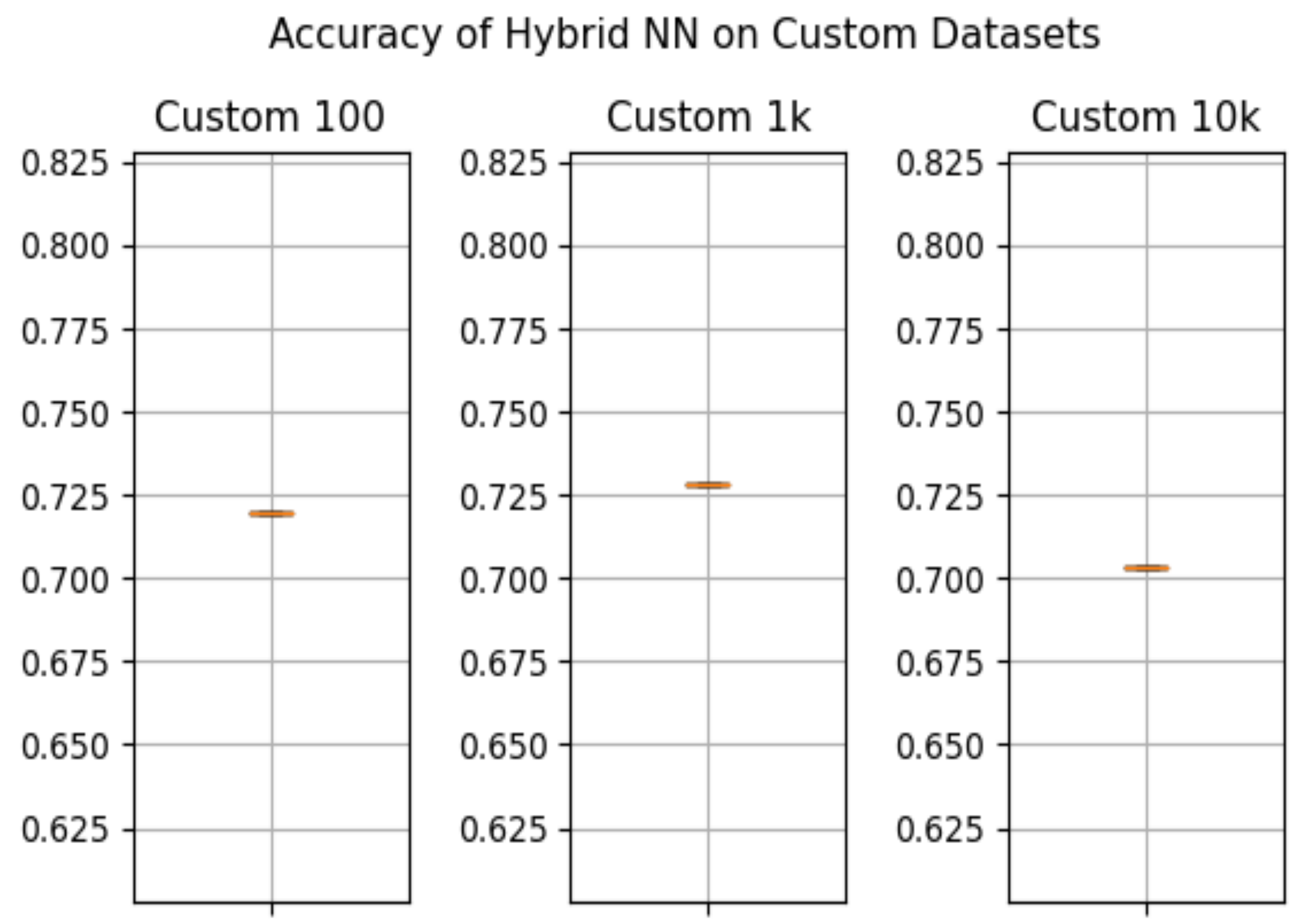
(*) Due to the serial nature of the Pytorch model, we couldn't submit batches of jobs. We only used 10 test rows to run on the quantum computer, because for everything above, we had to wait way too long since each new job was appended to the very end of the queue. Hence, the quantum computer accuracy values aren't really realistic (and if it's significantly better, then probably because of a lucky 10-rows subset).

# Hybrid Neural Network

## Accuracy Boxplots







**Results HQNN:**

**VLDS:** The hybrid NN worked really well on the VLDS dataset, which was to be expected since the classical NN already performed well on the VLDS dataset.
**Adhoc:** The Adhoc dataset was much more difficult for the settings we chose (classical layer first and then a quantum layer). It would've been interesting to know how a reversed setup performs (using a quantum layer first and then a classical layer to do the decision), mainly because the quantum layer is much more capable for representing (all) the data at once. However, this isn't trivial to switch since it requires a lot of knowledge for Pytorch setups as well as for efficient training, which we didn't have.
**Custom:** The custom dataset was surprisingly stable to learn, it might have helped to provide more trainable parameters also on the quantum layer.

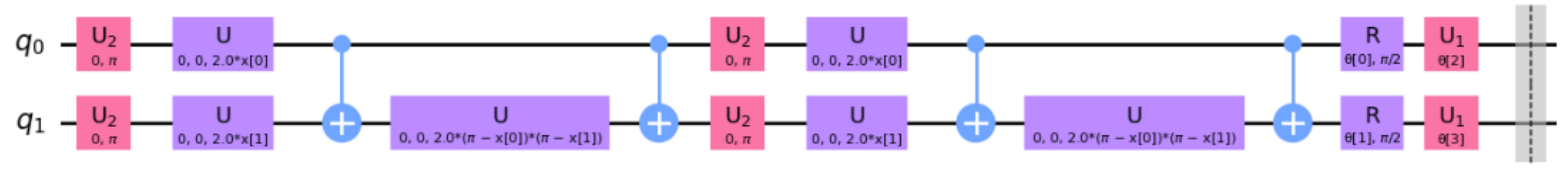
**Notes:** In general, it's hard to tell if the quantum layer made the network better. It performed better for some cases but not a lot better that one could justify the enormous increase in testing and validation time (the largest difference was for the 10k dataset where we had 9622s vs 18s). This difference was mainly due to the fact, that we could train the purely classical NN on the GPU whereas the hybrid NN had to be trained on the CPU and the layer needed to do 1024 shots before returning a result. Additionally, to run the network efficiently on a real quantum computer, it would require a more efficient and more complex setup to batch the quantum jobs together, which would also require changing the way the NN trains on the classical data part.

# Quantum Neural Network

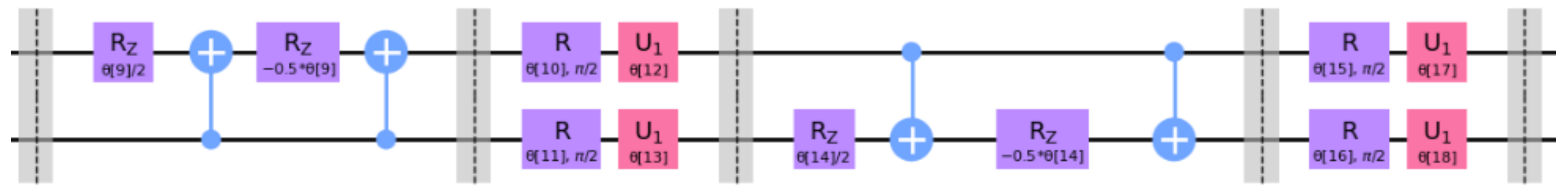
## Implementation

The quantum neural network used the "***ZZFeatureMap***" with ***'linear'*** entanglement and the "***TwoLocal***" Ansatz with ***'crz'*** entanglement blocks, ***'sca'*** entanglement and 5 repetitions. The Feature Maps and snippets from the *Ansätze* can be seen below for the QNNs with two input Qubits (Adhoc and Custom datasets) and four input Qubits (VLDS dataset).

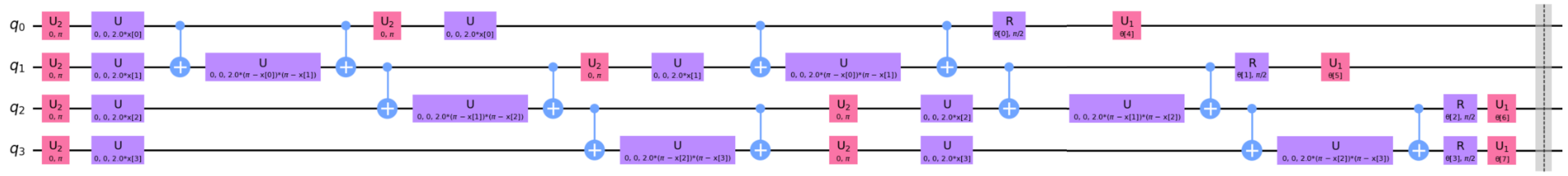
```
feature_map =
ZZFeatureMap(feature_dimension=feature_dimension,
entanglement='linear')
```



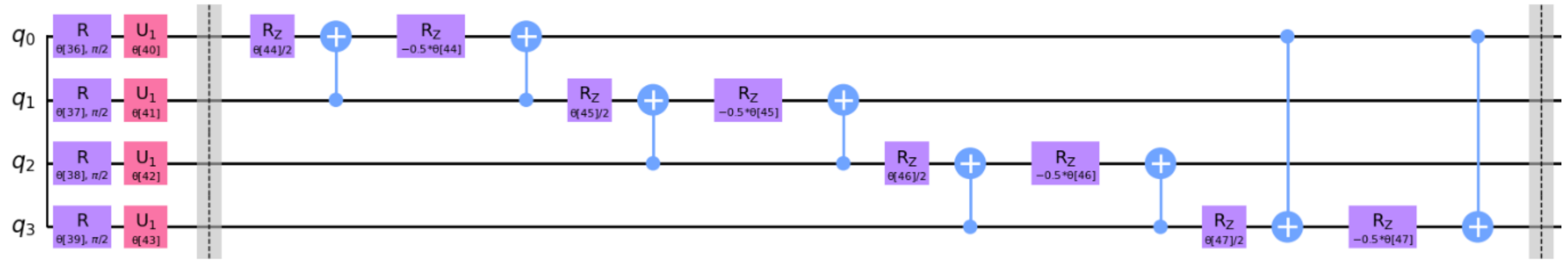
```
ansatz =
TwoLocal(num_qubits=feature_map.num_qubits,  reps=5,
rotation_blocks=['ry', 'rz'],
    entanglement_blocks='crz', entanglement= 'sca',
insert_barriers=True, skip_final_rotation_layer=True)
```



The circuits above and to the right with two qubits were used for the Adhoc and Custom datasets.



The circuits above and to the right with four qubits were used for the VLDS datasets.



# Quantum Neural Network

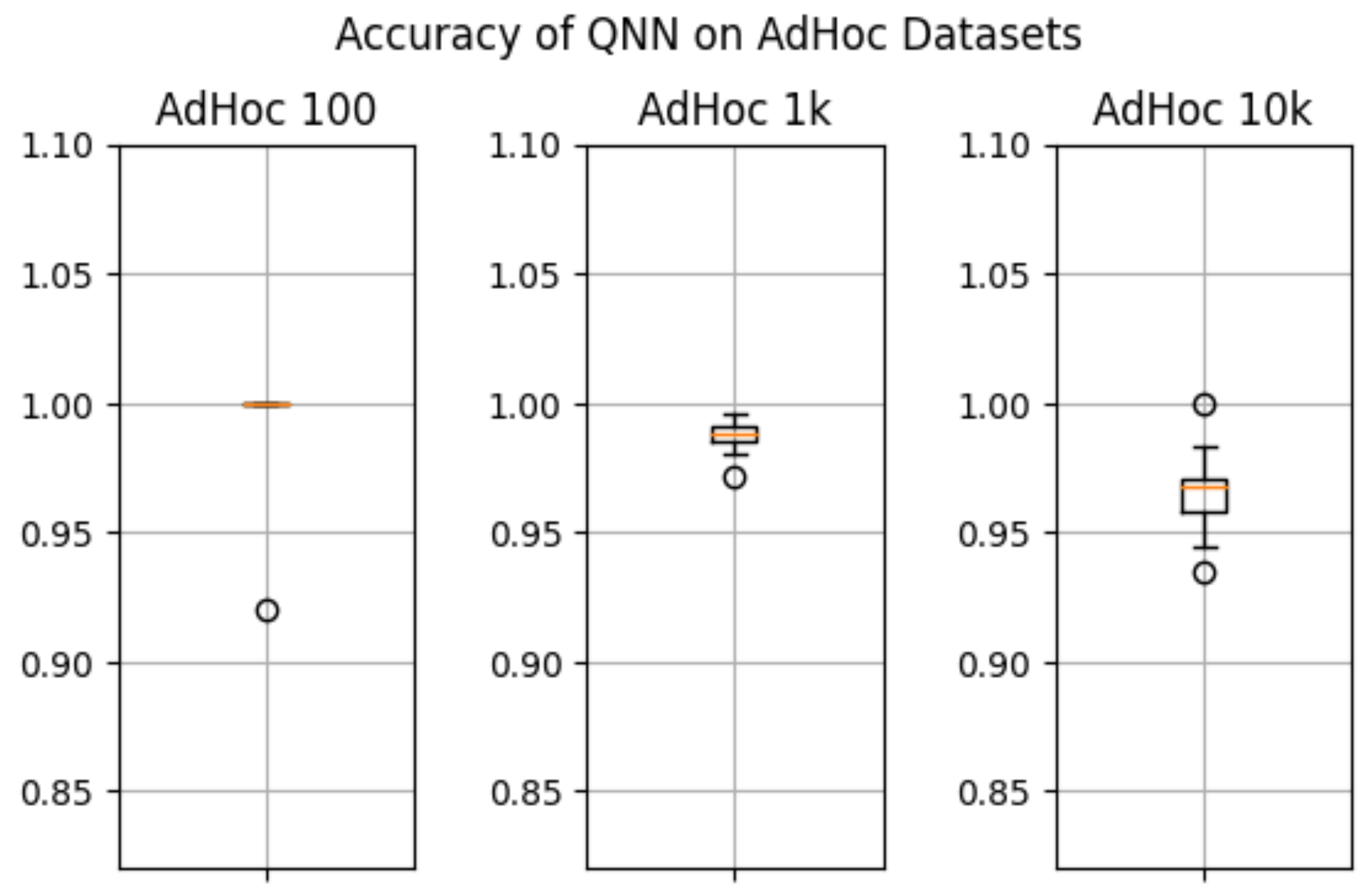
## Accuracies and Results using 10-fold cross validation

| Accuracy | AER Simulator | Quantum Computer | Classical Neural Network* |
|---|---|---|---|
| **Adhoc 100** | 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.92, 1.0, 1.0, 1.0<br>mean: 0.992, std: 0.025, mean training time: 48.69s | 1.0 | 0.56, 0.56, 0.56, 0.52, 0.56, 0.56, 0.52, 0.56, 0.60, 0.60<br>mean: 0.56, std: 0.02, mean training time: 0.55 |
| **Adhoc 1k** | 0.996, 0.988, 0.984, 0.992, 0.988, 0.992, 0.98, 0.988, 0.988, 0.972<br>mean: 0.9868, std: 0.0068, mean training time: 553.72s | 0.98 | 0.54, 0.54, 0.55, 0.57, 0.56, 0.55, 0.54, 0.54, 0.54, 0.51<br>mean: 0.54, std: 0.01, mean training time: 2s |
| **Adhoc 10k** | 0.9712, 0.9584, 0.9444, 0.9648, 0.9696, 0.9704, 0.9348, 0.9996, 0.9836, 0.9572<br>mean: 0.9654, std: 0.0184, mean training time: 5490.89s | 0.91 | 0.65, 0.62, 0.64, 0.64, 0.63, 0.63, 0.66, 0.60, 0.65, 0.62<br>mean: 0.64, std: 0.02, mean training time: 17s |
| **VLDS 100** | 0.56, 0.48, 0.44, 0.48, 0.44, 0.44, 0.64, 0.4, 0.44, 0.36<br>mean: 0.468, std: 0.0801, mean training time: 118.75s | 0.42 | 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96, 0.96<br>mean: 0.96, std: 0.00, mean training time: 1s |
| **VLDS 1k** | 0.54, 0.488, 0.556, 0.504, 0.536, 0.56, 0.532, 0.52, 0.512, 0.472<br>mean: 0.522, std: 0.0285, mean training time: 1222.29s | 0.5 | 0.85, 0.86, 0.85, 0.86, 0.84, 0.85, 0.86, 0.84, 0.86, 0.86<br>mean: 0.85, std: 0.01, mean training time: 2s |
| **VLDS 10k** | 0.5384, 0.5528, 0.5436, 0.5468, 0.5344, 0.5572, 0.5564, 0.5496, 0.5584, 0.5368<br>mean: 0.547, std: 0.0088, mean training time: 15788.8s | 0.53 | 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00<br>mean: 1.00, std: 0.00, mean training time: 17s |
| **Custom 100** | 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72,<br>mean: 0.72, std: 0.00, mean training time: 51.54s | 0.71 | 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72, 0.72<br>mean: 0.72, std: 0.0, mean training time: 1s |
| **Custom 1k** | 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728, 0.728<br>mean: 0.728, std: 0.0, mean training time: 696.82s | 0.72 | 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73<br>mean: 0.73, std: 0.00, mean training time: 2s |
| **Custom 10k** | 0.703, 0.703, 0.703, 0.703, 0.703, 0.703, 0.703, 0.703, 0.703, 0.703<br>mean: 0.703, std: 0.00, mean training time: 7422.2s | 0.7 | 0.70, 0.72, 0.72, 0.72, 0.71, 0.71, 0.71, 0.72, 0.71, 0.71<br>mean: 0.71, std: 0.006, mean training time: 18s |

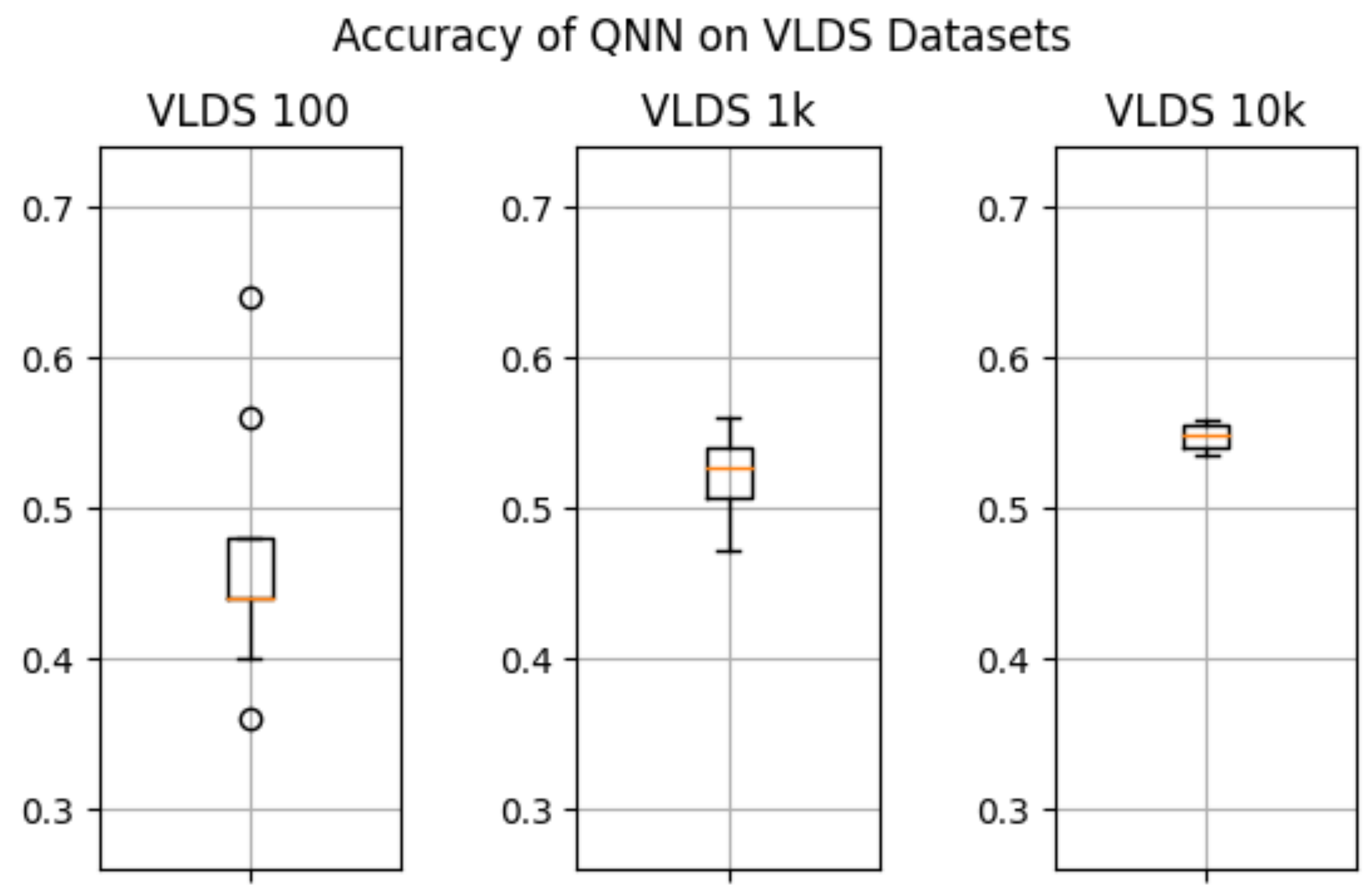
(*) These are the same values as in the previous slide, since the same classical NN was used as a benchmark for both the HNN and the QNN.

# Quantum Neural Network

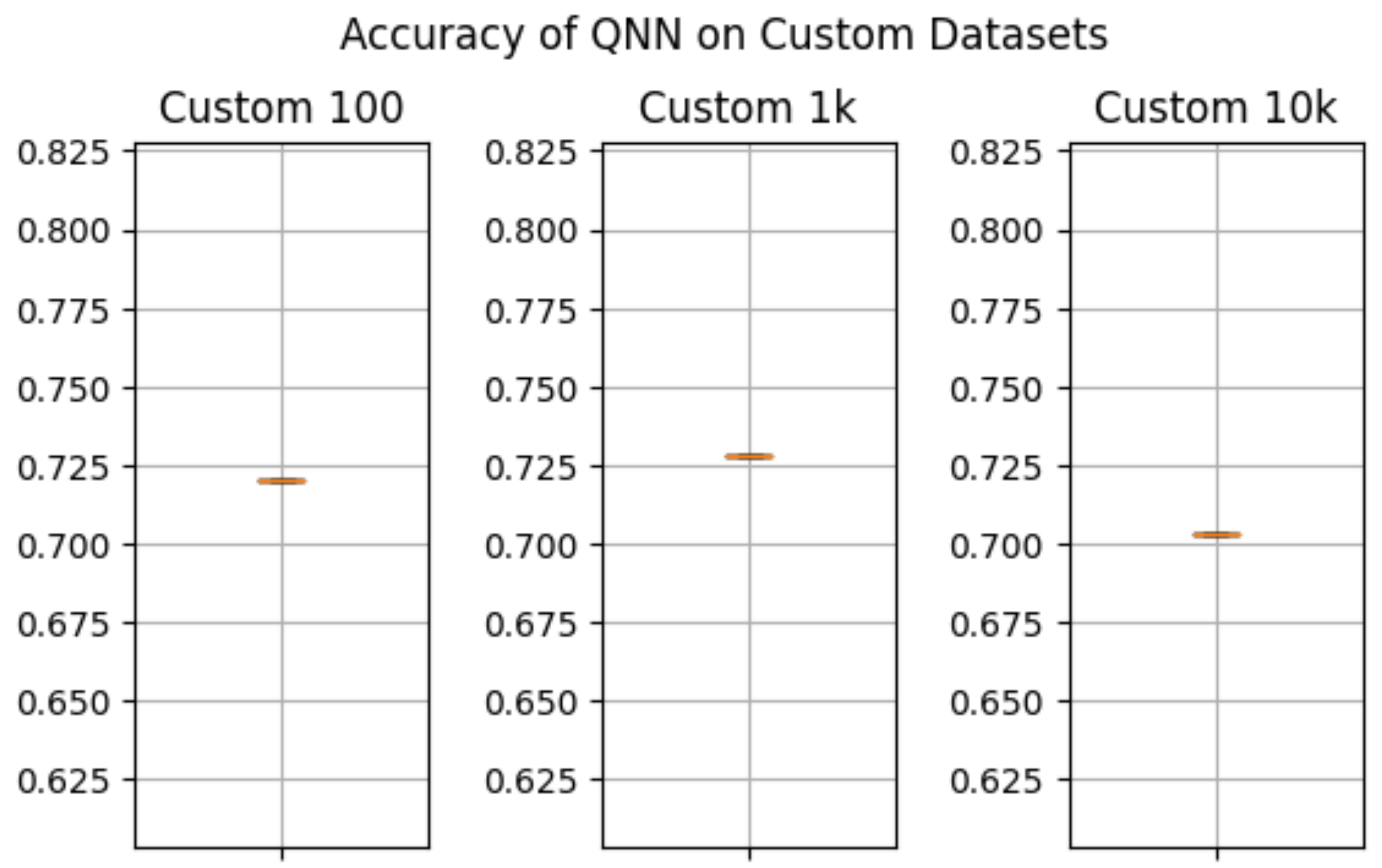
## Accuracy Boxplots



Avg. training time per fold (100, 1k, 10k):
48.69s, 553.71s, 5490.89s



Avg. training time per fold (100, 1k, 10k):
118.75s, 1222.29s, 15788.83s



Avg. training time per fold (100, 1k, 10k):
51.53s, 696.82s, 7422.18s

**Results QNN:**

**AdHoc:** The QNN worked really well on the Adhoc dataset, and also much better than the classical NN. This was somewhat expected, since classical NNs don't perform well with the Adhoc dataset.
**VLDS:** On the VLDS dataset the performance was less stable and also slightly worse compared to the classical NN.
**Custom:** On the custom dataset the results were surprisingly stable and surprisingly similar to the HQNN and the classical NN.

**Notes:**

- Design: Designing the quantum neural network, i.e. picking feature map and the entanglement *ansatz* proved to be the challenging portion of this task. It is generally not known what feature map and what ansatz will yield good results for QNNs. It would be interesting to use a more systematic and combinatoricly exhaustive approach, however a lot of time would be required to analyse the results.
- Waiting times: The training times for the QNNs grew roughly linearly with the input sizes. For the 1k datasets, training took ~5h+. For the larger (10k) datasets it took roughly 10x as long.
- Custom Dataset: When looking at the datasets directly (ie. scatter-plot), a clear pattern is visible (two diagonal lines) – however some points that fall very close to that line have the same label as points not on that line. If we wanted to train the network to recognize the lines, we would have to feed it more precise data, and maybe remove any points on the diagonals not labelled as such.
- VLDS Dataset: The VLDS datasets were a bit more tricky due to there being 4 features instead of 2. However when viewed directly (ie scatter-plot) from the right angle, it is very easy to see a linear plane that could identify two clusters representing the two labels. This made this dataset a very good candidate for classical approaches.

# Retrospective

## What we would do differently and what we learned

***"Which algorithm performs best?"***

- In general one can say, that the QSVM performed best for the Adhoc dataset as it's quantum kernel was suited best for the unusual distribution whereas the QNN and Hybrid NN were best on the VLDS dataset. The custom dataset doesn't really have a winning algorithm, due to its ambiguous distribution.

***"How do the results compare to those from the bachelor thesis?"***

- In comparison to the bachelor thesis, the accuracies fluctuate sometimes quite a bit. For example, the Adhoc dataset performed better for the QSVM in our evaluation whereas the VLDS dataset performed better in the evaluation described in the bachelor thesis.

***"Which lessons did you learn?"***

- Some datasets are better suited for the classical ML approach, some are better suited for a quantum-based approach.
- Plan enough time for training, maybe even prepare a machine that works overnight on its own. The laptop was unusable during training which lead to problems since we had to use it also for different tasks.
- Also, schedule quantum computer runs and be prepared to be stuck in the queue even for hours.
- We wasted quite some time to first find an all-fitting model for each dataset but it would have been better to fine-tune the parameters for each dataset type (e.g. change the kernel per dataset type and not trying to find a universal one).

***"What would you do differently if you could start all over again?"***

- Focus more on simpler algorithms and don't lose too much time in fine-tuning the parameters since it mostly didn't significantly affect the result.
- Focus more on the dataset where classical neural networks perform the worst.
- Focus on systematically introducing more complexity into the QSVM, HNN or QNN in order to get better results - without *unnecessary* complexity.
- It would've been much more interesting to test the algorithms on different datasets. Especially the VLDS and Custom dataset were not really suited (for quantum ML) as the VLDS dataset revealed a linear separation between the two features, hence it was to be expected that a classical SVM with a linear kernel would perform better. The custom dataset on the other side produced indistinguishable data for larger datasets, as the features overlapped each other (also clearly visible in data plots) so that the models (quantum and classical) had no other option than to guess the feature on the overlapping area. The only dataset which was actually interesting for using on a quantum approach was the Adhoc dataset, since there was no clear, linear separation of the data.