Case study: binary classification problem of entanglement detection

Ana Martínez Sabiote

The quantum separability problem consists in deciding whether a bipartite density matrix is entangled or separable. Finding the Schmidt decomposition of a state to determine if its separable or not is an NP-hard problem. In this notebook we will propose QSVM and QNN methods to tackle problem as a binary classification task using the data produced by https://gitlab.lis-lab.fr/balthazar.casale/ML-Quant-Sep

This repository publishes two fully labeled datasets of 6,000 bipartite density matrices. The first contains density matrices of dimension 9×9 of a bipartite quantum system $H = H_a \otimes H_b$, with $\rho\_a = \dim(H_a) = 3$ and $\rho_b = \dim(H_b) = 3$. The second is composed of density matrices of size 49×49, thus $\rho_a = \rho_b = 7$. Each dataset is a collection of pairs of input density matrices and labels indicating whether the corresponding density matrix is separable or entangled, and contains separable (SEP), PPT entangled (PPT-ENT) and non-PPT (NPPT-ENT) density matrices with 2,000 examples each.

We will constraint our experimentation to the first dataset for a bipartite system of $\rho_a = \rho_b = 3$.

We will start taking 100 samples of each label to form our training dataset, resulting in a perfectly balanced dataset. Using the code from the repository, we have read the separable data, PPT entangled data and NPPT entangled data and joined them to make a np.ndarray of 200 density matrix of dimensions 9 x 9. Then, the representations.py module has been used to transform the np.ndarray of density matrices into a real-valued vector of 200 samples and 80 attributes. We have saved this arrays into a csv file. An analogous process has been performed for creating the test set.

We can deal with 200 data instances, but the challenging aspect of this problem is the amount of attributes. For applying the quantum machine learning techniques we have studied, we firstly need to encode this great number of features.

1 Work setup

[ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[ ]:

```
pip install pennylane==0.26
```

```
Collecting pennylane==0.26
  Downloading PennyLane-0.26.0-py3-none-any.whl (1.0 MB)
```

[ ]:

```
pip install tensorflow==2.9.1
```

```
Collecting tensorflow==2.9.1
  Downloading
tensorflow-2.9.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(511.7 MB)
```

[ ]:
```
pip install pyyaml h5py
```

[ ]:
```
import pandas as pd
import numpy as np
import pennylane as qml
import tensorflow as tf
import matplotlib.pyplot as plt

from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.preprocessing import MaxAbsScaler
from sklearn.model_selection import KFold

import seaborn as sns

import joblib
```

[ ]:
```
# pennylane works with doubles and tensorflow works with floats.
# We ask tensorflow to work with doubles

tf.keras.backend.set_floatx('float64')
```

2 Pre-work to get started

2.1 Data preprocessing

Overview of the dataset description. Should we normalise or scale the values? they are all really close to 0.

[ ]:
```
training_data = pd.read_csv("/content/drive/MyDrive/tfg/x_train_1.csv", header=None)
training_data.head()
```

[ ]:
```
          0         1         2         3         4         5         6   ...
0  0.002405 -0.013551  0.014313  0.007182 -0.008649 -0.003936 -0.002174
1 -0.006249  0.010679 -0.000343 -0.003259 -0.001029 -0.008732 -0.002129
2  0.012010  0.006747  0.006879  0.000910 -0.001017 -0.003304 -0.003681
3 -0.147372 -0.009079 -0.008336 -0.021830 -0.012960 -0.004989  0.054233
4 -0.010161 -0.021211 -0.009640 -0.004726  0.000182 -0.013206 -0.001762
```

```
         74        75        76        77        78        79
0  0.045722  0.076358  0.083020  0.093225  0.105654  0.109198
1  0.061884  0.080446  0.095130  0.103344  0.103996  0.110223
2  0.028129  0.060973  0.073608  0.078672  0.095634  0.102708
3  0.105213  0.087479  0.105942  0.130937  0.111882  0.118560
4  0.055780  0.075803  0.095085  0.105444  0.107857  0.114460

[5 rows x 80 columns]
```

[ ]:
```
training_data.describe()
```

[ ]:
```
                0           1           2           3           4           5  ...
count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
mean     0.004797    0.000064    0.000108   -0.000144    0.000498    0.001381
std      0.028860    0.029620    0.036638    0.025325    0.026439    0.032549
min     -0.147372   -0.081901   -0.187091   -0.069290   -0.080485   -0.159721
25%     -0.011422   -0.019337   -0.015301   -0.011488   -0.010467   -0.014435
50%      0.005595    0.000423    0.000219   -0.000011   -0.000762    0.001258
75%      0.024328    0.016108    0.018386    0.007331    0.010476    0.018243
max      0.078962    0.099702    0.170843    0.073404    0.087689    0.114851


               77          78          79
count  200.000000  200.000000  200.000000
mean     0.097276    0.106195    0.112494
std      0.013057    0.010411    0.008404
min      0.057148    0.059018    0.070539
25%      0.091099    0.100307    0.108088
50%      0.096756    0.107006    0.112603
75%      0.104528    0.111522    0.116679
max      0.162879    0.162101    0.152014

[8 rows x 80 columns]
```

[ ]:
```
x_train = np.genfromtxt("/content/drive/MyDrive/tfg/x_train_1.csv",␣
 ↪delimiter=",",dtype=None)
y_train = np.genfromtxt("/content/drive/MyDrive/tfg/y_train_1.csv",␣
 ↪delimiter=",",dtype=None)

x_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/x_test_small.csv",␣
 ↪delimiter=",",dtype=None)
y_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/y_test_small.csv",␣
 ↪delimiter=",",dtype=None)
```

[ ]:
```
print(type(x_train))
print(type(y_train))
print(x_train.shape)
print(y_train.shape)

print(type(x_test_small))
```

```
print(type(y_test_small))
print(x_test_small.shape)
print(y_test_small.shape)
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
(200, 80)
(200,)
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
(300, 80)
(300,)
```

2.2 Quantum support vector machines

[ ]:
```
# Amplitude encoding of 80 variables using 7 qubits (can encode up to 128 inputs)

# Number of qubits of the system
nqubits = 7
# We define a device
dev = qml.device("lightning.qubit", wires = nqubits)

# We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
# operation equivalent to amplitude encoding of the first argument
# Since the vector has 80 components which is not a power of 2, we extend the vector
# to 128 components using padding with 0.
@qml.qnode(dev)
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
 ↪normalize=True))    # We return an array with the probabilities fo measuring each␣
 ↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

[ ]:
```
# Check that the circuit works as expected
k=kernel_circ(x_train[0], x_train[1])
print(k.shape)
```

```
(128,)
```

[ ]:
```
# Checking we get a 1 in the first entry of the returned array when the two arguments␣
 ↪are the same
kernel_circ(x_train[0], x_train[0])
```

[ ]:
```
tensor([1.00000000e+00, 7.40406749e-32, 1.66038413e-32, 1.38834029e-32,
        4.92602587e-33, 1.42263375e-32, 1.88747586e-31, 4.36044173e-32,
        ...,
        2.52937313e-34, 2.07789578e-33, 5.23592488e-35, 1.95060277e-34],
requires_grad=True)
```

Even if amplitude encoding can be computed, then it takes more than 1h execture the following cell to train the SVM (16384 iterations).

```
[ ]: """
     from sklearn.svm import SVC
     def qkernel(A, B):
       return np.array([[kernel_circ(a,b)[0] for b in B] for a in A])

     svm = SVC(kernel = qkernel).fit(x_train, y_train)
     """
```

```
[ ]: """
     from sklearn.metrics import accuracy_score
     print(accuracy_score(svm.predict(x_test), y_test))
     """
```

```
[ ]: '\nfrom sklearn.metrics import
     accuracy_score\nprint(accuracy_score(svm.predict(x_test), y_test))\n'
```

Therefore, its needed to apply some dimensionality reduction techniques to work with less attributes such as principal component analysis or autoencoding.

Then, regarding embeddings (encoding input features into the quantum state of the circuit) - Amplitude encoding: we need n qubits for $N \leq 2^n$ attributes. - Angle enconding: it encodes N features into the rotation angles of n qubits, where $N \leq n$. The other built in encodings that PennyLane offers need a number of qubits greater or equal to the number of features. Since we have so many attributes, even if we reduce its dimensionality, I think we would need too many qubits for applying this kind of encoding.

2.2.1 PCA (64) with amplitude encoding

Next, we are goint to try a different approach: reducing the dimensionality of a dataset while minimizing information loss. We will use principal component analysis to reduce the number of variables in our dataset from 80 to 64, to apply amplitude enconding on 6 qubits instead of 7.

```
[ ]: pca = PCA(n_components = 64)

     xs_train = pca.fit_transform(x_train)
     xs_test = pca.transform(x_test)
```

```
[ ]: print(xs_train.shape)
```

```
(200, 64)
```

```
[ ]: # Amplitude encoding of 64 variables using 6 qubits (can encode up to 64 inputs)

     # Number of qubits of the system
     nqubits = 6
     # We define a device
     dev = qml.device("lightning.qubit", wires = nqubits)

     # We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
     # operation equivalent to amplitude encoding of the first argument
```

```
@qml.qnode(dev)
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
 ↪normalize=True))    # We return an array with the probabilities fo measuring each␣
 ↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

[ ]:
```
# Check that the circuit works as expected
ks=kernel_circ(xs_train[0], xs_train[1])
print(ks.shape)
```

```
(64,)
```

[ ]:
```
def qkernel(A, B):
  return np.array([[kernel_circ(a,b)[0] for b in B] for a in A])
```

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
xs_test_small = pca.transform(x_test_small)
print(accuracy_score(svm.predict(xs_test_small), y_test_small))
```

```
1.0
```

Summary of this case

PCA 64 attributes, amplitude encoding, lightning qubit device, 6 qubits

Accuracy on a test set of 300 instances is 1.0. Suspicious?

Training took 28 min

Prediction took 44 min

2.2.2 PCA (32) with amplitude encoding

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train)
xs_test_small = pca.transform(x_test_small)
```

[ ]:
```
# Amplitude encoding of 64 variables using 6 qubits (can encode up to 64 inputs)

# Number of qubits of the system
nqubits = 5
# We define a device
dev = qml.device("lightning.qubit", wires = nqubits)

# We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
# operation equivalent to amplitude encoding of the first argument
@qml.qnode(dev)
```

```
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
 ↪normalize=True))    # We return an array with the probabilities fo measuring each␣
 ↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
print(accuracy_score(svm.predict(xs_test_small), y_test_small))
```

```
0.9666666666666667
```

Summary of this case

PCA 32 attributes, amplitude encoding, lightning qubit device, 5 qubits

Accuracy on a test set of 300 instances is 0.9666

Training took 15 min

Prediction took 23 min

2.3 Quantum neural networks

[ ]:
```
# We set a seed for the packages so the results are reproducible
seed=4321
np.random.seed(seed)
tf.random.set_seed(seed)
```

[ ]:
```
def plot_losses(history):
  tr_loss = history.history["loss"]
  epochs = np.array(range(len(tr_loss))) + 1
  plt.plot(epochs, tr_loss, label="Training loss")
  plt.xlabel("Epoch")
  plt.show()
```

2.3.1 PCA (32) with amplitude encoding, TwoLocal variational form, 5 qubits, default.qubit

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train)
xs_test_small = pca.transform(x_test_small)
```

[ ]:
```
# Two local variational form
def TwoLocal(nqubits, theta, reps=1):
  for r in range(reps):
    for i in range(nqubits):
      qml.RY(theta[r*nqubits+i], wires=i)
    for i in range(nqubits-1):
```

```
      qml.CNOT(wires=[i,i+1])

  for i in range(nqubits):
    qml.RY(theta[reps*nqubits+i], wires=i)
```

```
[ ]: # Hermitian matrix
     state_0 = [[1], [0]]
     M = state_0 * np.conj(state_0).T
```

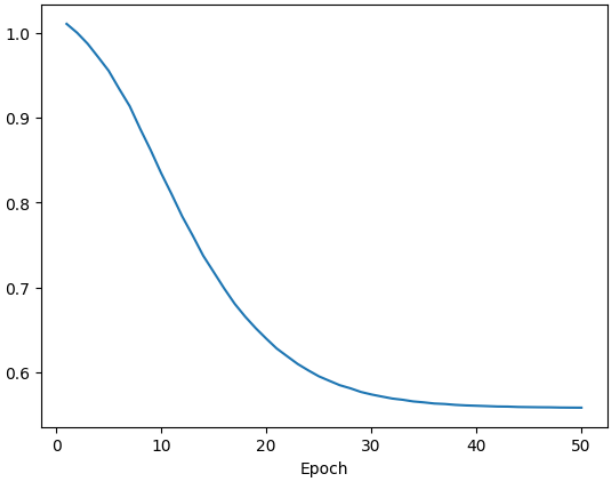
```
[ ]: nqubits=5
     dev=qml.device("default.qubit", wires=nqubits)

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=1)
       return qml.expval(qml.Hermitian(M, wires=[0]))

     qnn = qml.QNode(qnn_circuit, dev, interface="tf")
```

```
[ ]: weights={"theta": 10}
     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 27s 3s/step - loss: 1.0103
Epoch 2/50
10/10 [==============================] - 35s 3s/step - loss: 0.9998
.
.
.
Epoch 49/50
10/10 [==============================] - 15s 2s/step - loss: 0.5584
Epoch 50/50
10/10 [==============================] - 15s 1s/step - loss: 0.5583
```

```
[ ]: plot_losses(history)
```



[ ]:
```
# Check accuracy

tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 10s 1s/step
10/10 [==============================] - 13s 1s/step
```

[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.715
Test accuracy:  0.5966666666666667
```

Test accuracy is not great.

Aspects to consider - Training time 15 min - We can try other NN architectures, looking at paper Entanglement detection using Deep Neural networks - We don't have validation set!

2.3.2 PCA (32) with amplitude encoding, StronglyEntanglingLayers, default.qubit

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train)
xs_test_small = pca.transform(x_test_small)
```

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

[ ]:
```
nqubits=5
dev = qml.device("default.qubit", wires=nqubits)
```

```
# number of repetitions that we want in each instance of the variational form
nreps = 2
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

qnn_strong = qml.QNode(qnn_circuit_strong, dev)

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:

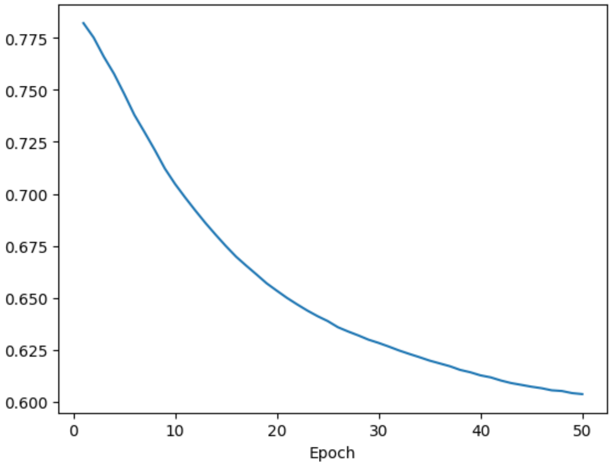
```
# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:

```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 31s 3s/step - loss: 0.7821
Epoch 2/50
10/10 [==============================] - 31s 3s/step - loss: 0.7751
.
.
.
Epoch 49/50
10/10 [==============================] - 31s 3s/step - loss: 0.6042
Epoch 50/50
10/10 [==============================] - 31s 3s/step - loss: 0.6037
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 22s 2s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.715
Test accuracy:  0.5733333333333334
```

Using StronglyEntanglingLayers the accuracy is basically identical to the previous case.

Observation, lightning.qubit in this case raises unknown error in tensorflow.

Training 25 minutes

2.3.3 PCA (32) with amplitude encoding, StronglyEntanglingLayers, lightning.qubit and adjoind differentiation method

```
[ ]: nqubits=5
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 2
     #  dimensions of the input that the variational form expects
     weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
     #  number of inputs that each instance of the variational form will take
     nweights = 3*nreps*nqubits
```

```
def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))


# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

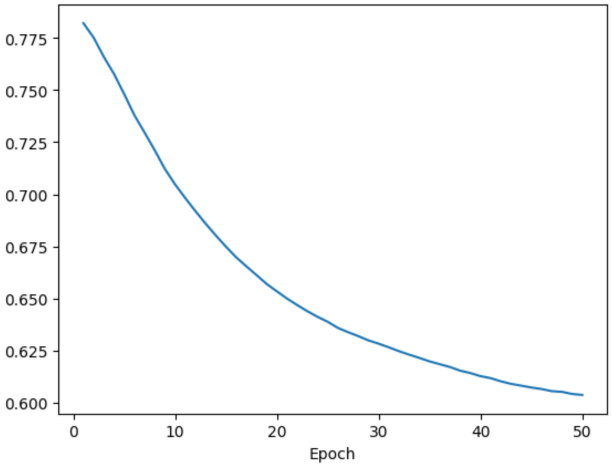
[ ]:

```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 15s 2s/step - loss: 0.7821
Epoch 2/50
10/10 [==============================] - 8s 835ms/step - loss: 0.7751
.
.
.
Epoch 49/50
10/10 [==============================] - 14s 1s/step - loss: 0.6042
Epoch 50/50
10/10 [==============================] - 13s 1s/step - loss: 0.6037
```

[ ]:

```
plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 5s 660ms/step
10/10 [==============================] - 12s 1s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.715
Test accuracy:  0.5733333333333334
```

No improvements on accuracy, exact performance, which makes sense because we are running the exact same configuration. Although using lightning.qubit with adjoint differenciation method increases significantly the speed (averagely per epoch 30s -> 14s)

Training 11 minutes

2.3.4 PCA (64) with amplitude encoding, StronglyEntanglingLayers, lightning.qubit and adjoint differentiation method

```
[ ]: pca = PCA(n_components = 64)

     xs_train = pca.fit_transform(x_train)
     xs_test_small = pca.transform(x_test_small)
```

```
[ ]: # Hermitian matrix
     state_0 = [[1], [0]]
     M = state_0 * np.conj(state_0).T
```

[ ]:
```
nqubits=6
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 2
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

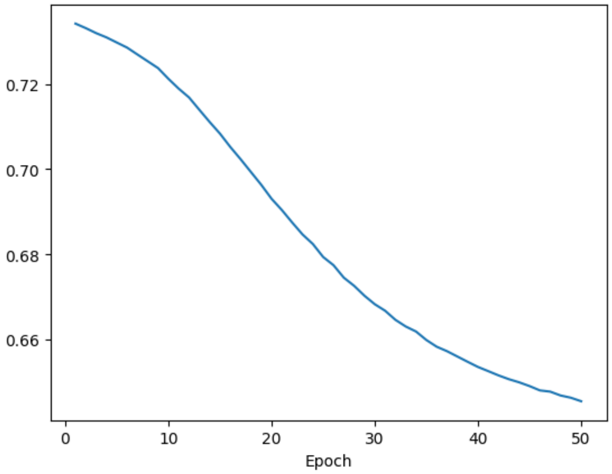
[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 18s 2s/step - loss: 0.7342
Epoch 2/50
10/10 [==============================] - 16s 2s/step - loss: 0.7331
.
.
```

```
.
Epoch 49/50
10/10 [==============================] - 16s 2s/step - loss: 0.6463
Epoch 50/50
10/10 [==============================] - 16s 2s/step - loss: 0.6455
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 11s 1s/step
10/10 [==============================] - 15s 2s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.66
Test accuracy:  0.63
```

Still not great accuracies even if we are working with more attributes (64). Since the training time was only 13 min thanks to the configuration of lightning.qubit + adjoint differentiation method, we can afford to try with 7 qubits and the original attributes, without reducing dimensionality with PCA.

2.3.5 80 attributes, 7 qubits, lightning.qubit and adjoint differentiation method

```
[ ]: nqubits=7
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 2
```

```
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:

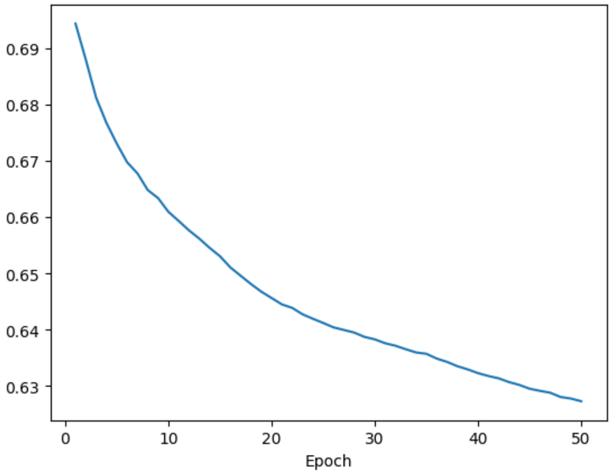
```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:

```
# Training our model
history = model.fit(x_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 17s 2s/step - loss: 0.6943
Epoch 2/50
10/10 [==============================] - 17s 2s/step - loss: 0.6879
.
.
.
Epoch 49/50
10/10 [==============================] - 17s 2s/step - loss: 0.6278
Epoch 50/50
10/10 [==============================] - 22s 2s/step - loss: 0.6273
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 10s 1s/step
10/10 [==============================] - 16s 2s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.475
Test accuracy:  0.48333333333333334
```

Terrible results

Possible directions to continue
- Incorporating data preprocessing, since all the attributes have values really really close to 0 - Using other possible feature maps and dimensionality reduction techniques - Study the accuracy of each type of data (SEP, PPT-ENT and NPPT-ENT) - Generate dataset with different proportions of PPT-ENT and NPPT-ENT as done in the paper, train svm's with those variations of the dataset and compare the accuracy of those svm's. - another test_small dataset because it is not balanced. We are training with a perfectly balanced set and testing with a 1:2 dataset. - a different QNN architecture

```
[ ]: scaler = MaxAbsScaler()
     x_train_norm = scaler.fit_transform(x_train)

     x_test_norm = scaler.transform(x_test_small)

     # Restrict all the values to be between 0 and 1
     x_test_norm = np.clip(x_test_norm,0,1)
```

2.3.6 Normalized data, 80 attributes, 7 qubits, lightning.qubit and adjoint differentiation method

[ ]:

```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

[ ]:

```
nqubits=7
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 2
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

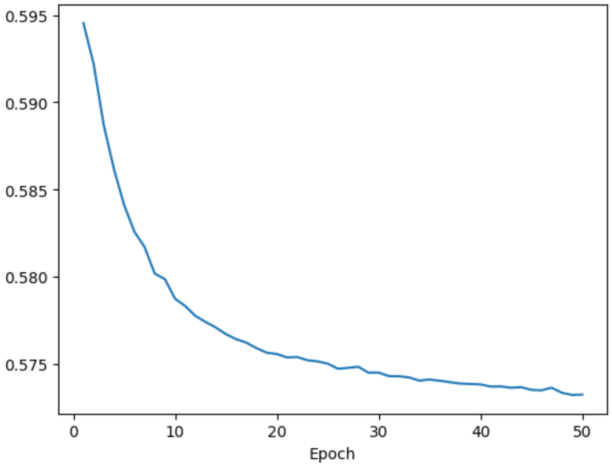
[ ]:

```
# Training our model
history = model.fit(x_train_norm, y_train, epochs = 50, shuffle = True,
```

```
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 81s 8s/step - loss: 0.5945
Epoch 2/50
10/10 [==============================] - 16s 2s/step - loss: 0.5922
.
.
.
Epoch 49/50
10/10 [==============================] - 16s 2s/step - loss: 0.5732
Epoch 50/50
10/10 [==============================] - 16s 2s/step - loss: 0.5732
```

[ ]:
```
plot_losses(history)
```



[ ]:
```
# Check accuracy

tr_acc = accuracy_score(model.predict(x_train_norm) >= 0.5, y_train)
test_acc = accuracy_score(model.predict(x_test_norm) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 12s 1s/step
10/10 [==============================] - 13s 1s/step
```

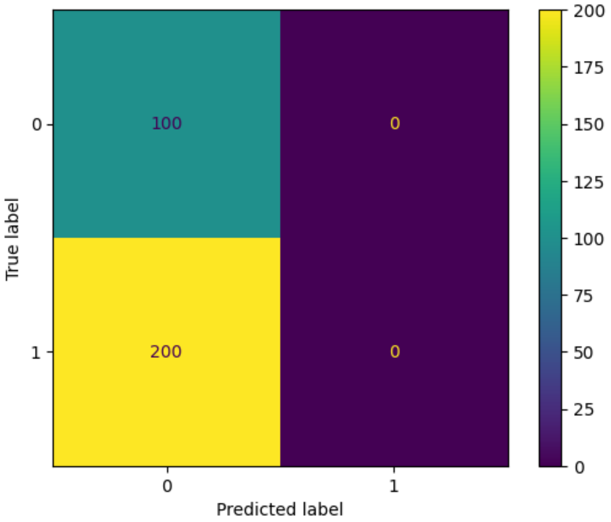
[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.545
Test accuracy:  0.3333333333333333
```

Even worst!

[ ]:
```
cm = confusion_matrix(y_test_small, model.predict(x_test_norm) >= 0.5)

cm_display = ConfusionMatrixDisplay(cm).plot()
```

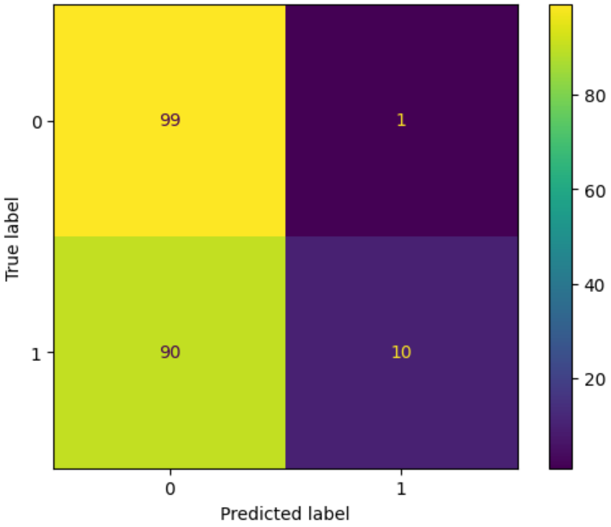
```
10/10 [==============================] - 16s 1s/step
```



It predicts all states separable, it doesn't detect entanglement at all. Let's see it it is because overfitting

[ ]:
```
cm = confusion_matrix(y_train, model.predict(x_train_norm) >= 0.5)

cm_display = ConfusionMatrixDisplay(cm).plot()
```

```
7/7 [==============================] - 8s 1s/step
```



With the training set it does not predict entanglement either

2.3.7 Normalized data, 80 attributes, 7 qubits, lightning.qubit,adjoint differentiation method and 4 repetitions in each instance of the variational form

[ ]:

```
# number of repetitions that we want in each instance of the variational form
nreps = 4
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

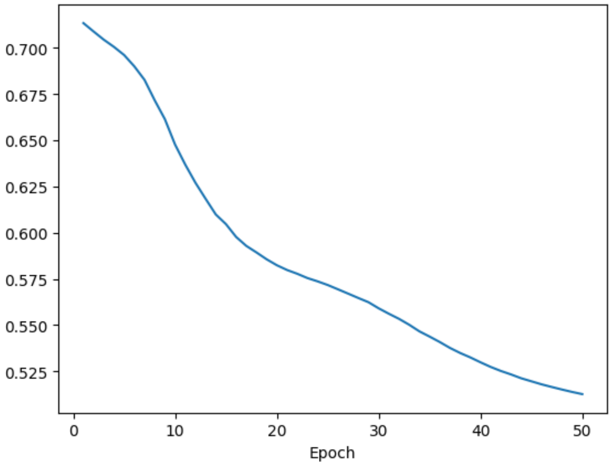
[ ]:

```
# Training our model
history = model.fit(x_train_norm, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
```

```
10/10 [==============================] - 28s 3s/step - loss: 0.7134
Epoch 2/50
10/10 [==============================] - 27s 3s/step - loss: 0.7089
.
.
.
Epoch 49/50
10/10 [==============================] - 27s 3s/step - loss: 0.5139
Epoch 50/50
10/10 [==============================] - 22s 2s/step - loss: 0.5127
```

```
[ ]: plot_losses(history)
```



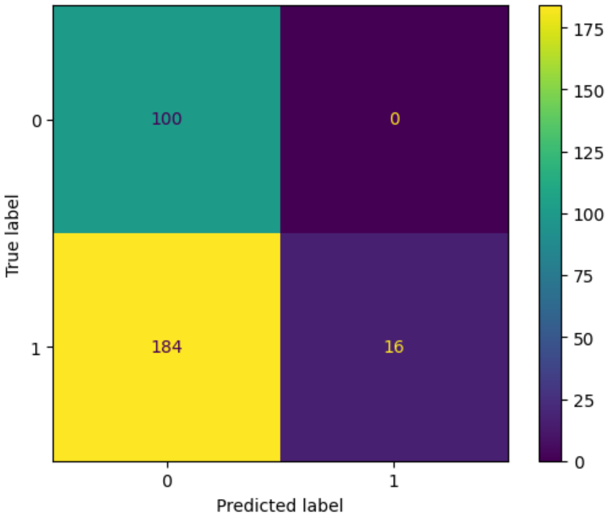
```
[ ]: # Check accuracy
     y_train_pred=model.predict(x_train_norm) >= 0.5
     y_test_pred=model.predict(x_test_norm) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 12s 1s/step
10/10 [==============================] - 17s 2s/step
```

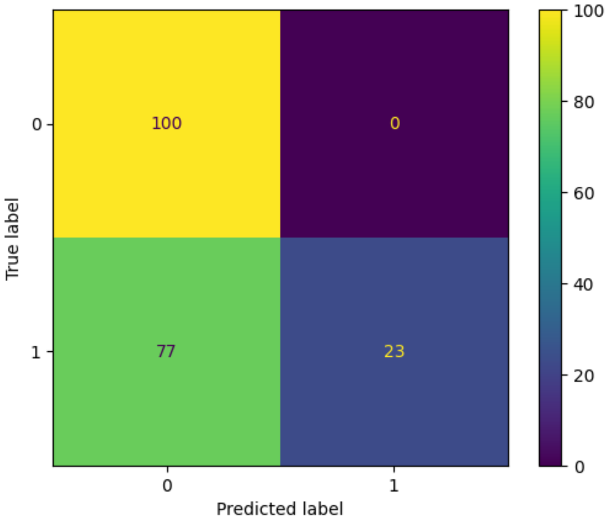
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.615
Test accuracy:  0.38666666666666666
```

```
[ ]: cm = confusion_matrix(y_test_small, y_test_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: cm = confusion_matrix(y_train, y_train_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



Not much improvement, still absolutely terrible accuracy. Let's discard this QNN with 80 attributes and 7 qubits

2.3.8 PCA (32) with amplitude encoding, Twolocal variational form, 5 qubits, lightning.qubit, adjoint differentiation method

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train_norm)
xs_test_small = pca.transform(x_test_norm)
```

[ ]:
```
# Two local variational form
def TwoLocal(nqubits, theta, reps=1):
  for r in range(reps):
    for i in range(nqubits):
      qml.RY(theta[r*nqubits+i], wires=i)
    for i in range(nqubits-1):
      qml.CNOT(wires=[i,i+1])

  for i in range(nqubits):
    qml.RY(theta[reps*nqubits+i], wires=i)
```

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

[ ]:
```
nqubits=5
dev=qml.device("lightning.qubit", wires=nqubits)

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=1)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

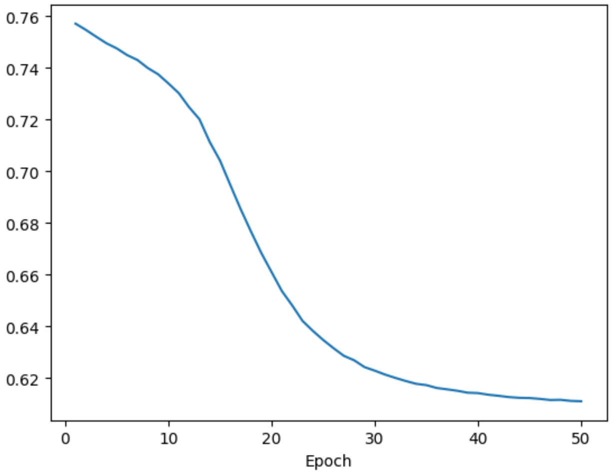
[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

weights={"theta": 10}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
```

```
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 8s 657ms/step - loss: 0.7572
Epoch 2/50
10/10 [==============================] - 8s 892ms/step - loss: 0.7548
.
.
.
Epoch 49/50
10/10 [==============================] - 6s 439ms/step - loss: 0.6113
Epoch 50/50
10/10 [==============================] - 4s 441ms/step - loss: 0.6111
```

[ ]:
```
plot_losses(history)
```



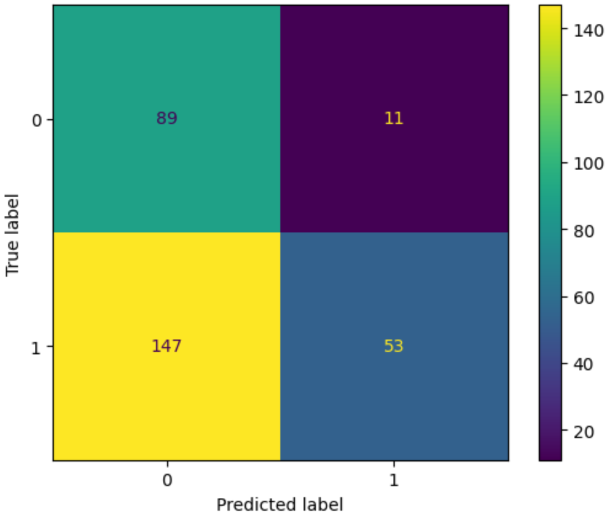
[ ]:
```
# Check accuracy
y_train_pred=model.predict(xs_train) >= 0.5
y_test_pred=model.predict(xs_test_small) >= 0.5

tr_acc = accuracy_score(y_train_pred, y_train)
test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 3s 369ms/step
10/10 [==============================] - 4s 391ms/step
```

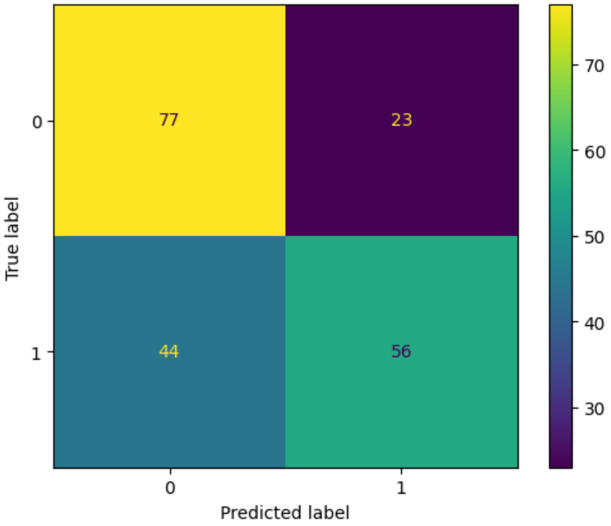
[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.665
Test accuracy:  0.47333333333333333
```

[ ]:
```
cm = confusion_matrix(y_test_small, y_test_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
cm = confusion_matrix(y_train, y_train_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



Normalization doesn't improve the original

2.3.9 PCA (32) with amplitude encoding, Twolocal variational form 5 REPS, 5 qubits, lightning.qubit, adjoint differentiation method

[ ]:

```
nqubits=5
dev=qml.device("lightning.qubit", wires=nqubits)

nreps=5

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

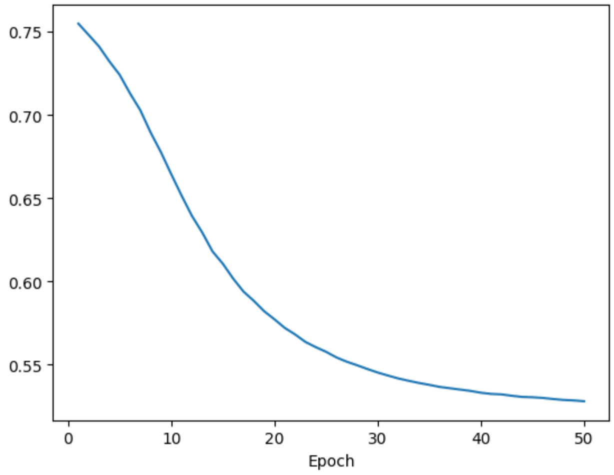
[ ]:

```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 72s 8s/step - loss: 0.7548
Epoch 2/50
10/10 [==============================] - 13s 1s/step - loss: 0.7480
.
.
.
Epoch 49/50
10/10 [==============================] - 13s 1s/step - loss: 0.5286
```

```
Epoch 50/50
10/10 [==============================] - 13s 1s/step - loss: 0.5281
```

[ ]:
```
plot_losses(history)
```



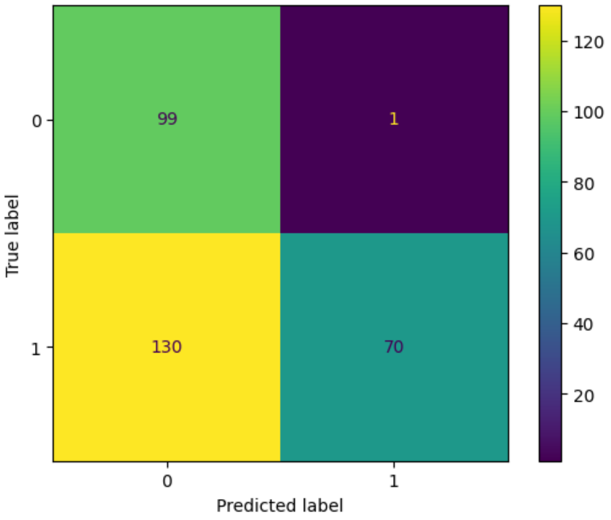
[ ]:
```
# Check accuracy
y_train_pred=model.predict(xs_train) >= 0.5
y_test_pred=model.predict(xs_test_small) >= 0.5

tr_acc = accuracy_score(y_train_pred, y_train)
test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 9s 1s/step
10/10 [==============================] - 14s 1s/step
```

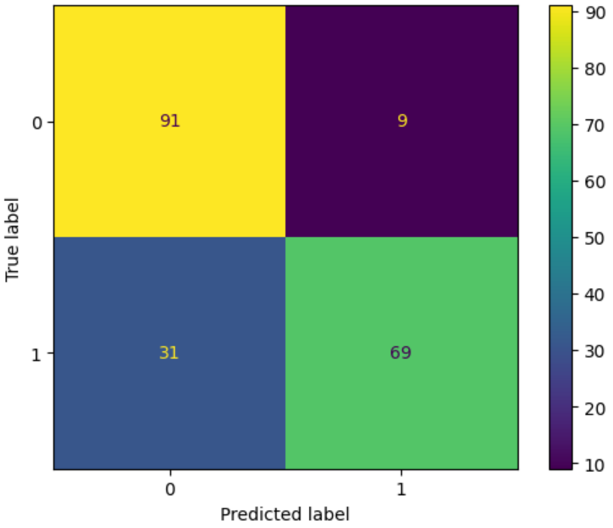
[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.8
Test accuracy:  0.5633333333333334
```

[ ]:
```
cm = confusion_matrix(y_test_small, y_test_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: cm = confusion_matrix(y_train, y_train_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



Increase in the number of repetitions produces an increase in the accuracy of the model. Let's try with more repetitions

2.3.10 PCA (32) with amplitude encoding, Twolocal variational form 10 REPS, 5 qubits, lightning.qubit, adjoint differentiation method

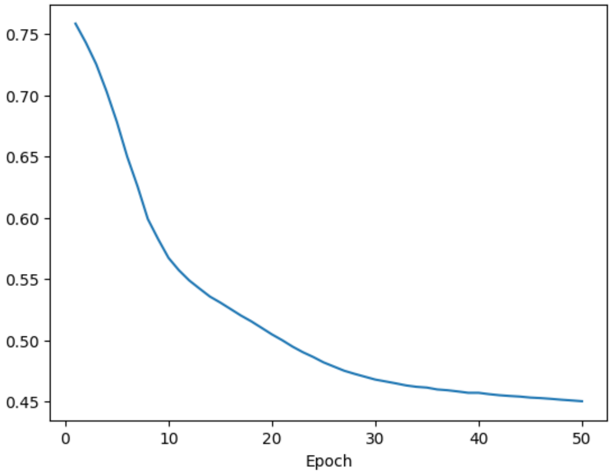
```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)

     nreps=10

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
       return qml.expval(qml.Hermitian(M, wires=[0]))
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 18s 2s/step - loss: 0.7584
Epoch 2/50
10/10 [==============================] - 18s 2s/step - loss: 0.7429
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4507
Epoch 50/50
10/10 [==============================] - 18s 2s/step - loss: 0.4501
```

[ ]:
```
plot_losses(history)
```



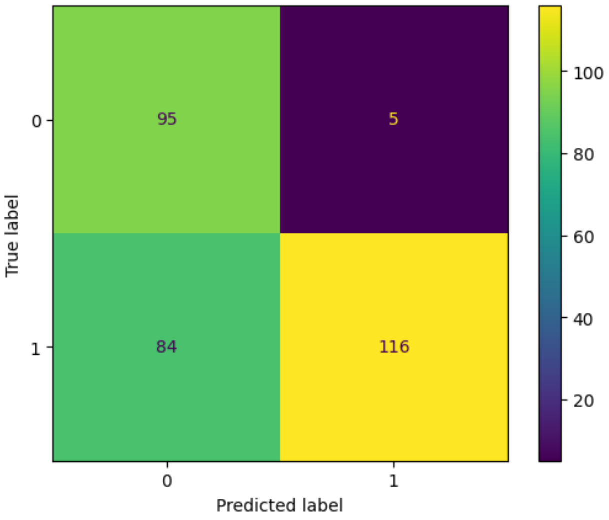
[ ]:
```
# Check accuracy
y_train_pred=model.predict(xs_train) >= 0.5
y_test_pred=model.predict(xs_test_small) >= 0.5

tr_acc = accuracy_score(y_train_pred, y_train)
test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 6s 846ms/step
10/10 [==============================] - 13s 1s/step
```

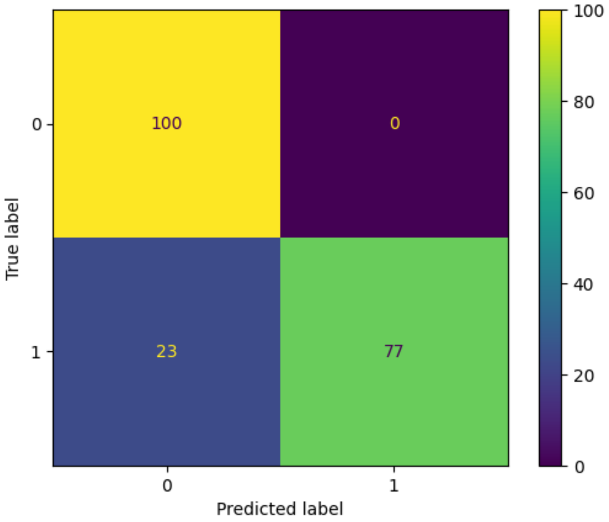
[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.885
Test accuracy:  0.7033333333333334
```

[ ]:
```
cm = confusion_matrix(y_test_small, y_test_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
cm = confusion_matrix(y_train, y_train_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
test_size=100
pred_split=np.array_split(y_test_pred,3)
y_sep_pred=pred_split[0]
y_ppt_pred=pred_split[1]
y_nppt_pred=pred_split[2]
```

```
score_sep = accuracy_score(y_sep_pred, np.full(test_size, 0))
score_ppt= accuracy_score(y_ppt_pred, np.full(test_size,1))
score_nppt= accuracy_score(y_nppt_pred, np.full(test_size,1))

print("SEP accuracy: ", score_sep)
print("PPT accuracy: ", score_ppt)
print("NPPT accuracy: ", score_nppt)
```

```
SEP accuracy:  0.95
PPT accuracy:  0.57
NPPT accuracy:  0.59
```

GREAT IMPROVEMENT!!! Increasing the number of repetitions of TwoLocal variational form produces better accuracy. In this case with 10 repetitions we obtain 70% of accuracy on the test set, the highest so far for QNN.

Observing the confussion matrix of the test set, there is almost a 1:1 proportion of true possitives to false negatives. Remembering that 1 represents entanglement and those samples come from PPT-ENT and NPPT-ENT we can investigate if that amount of false negatives comes from our model making errors when predicting one of the types of entanglement

2.3.11 PCA (32) with amplitude encoding, StrongEntanglingLayers 8 reps, lightning.qubit, 5 qubits, adjoint diff method

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train_norm)
xs_test_small = pca.transform(x_test_norm)
```

[ ]:
```
nqubits=5
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 8
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))
```

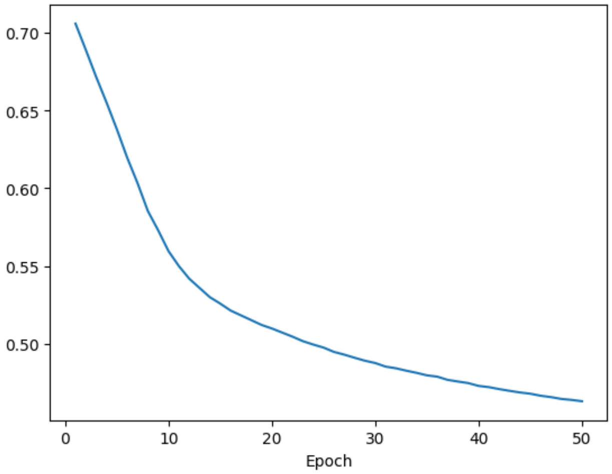
```
  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 39s 4s/step - loss: 0.7059
Epoch 2/50
10/10 [==============================] - 33s 3s/step - loss: 0.6890
.
.
.
Epoch 49/50
10/10 [==============================] - 32s 3s/step - loss: 0.4639
Epoch 50/50
10/10 [==============================] - 32s 3s/step - loss: 0.4631
```

[ ]:
```
plot_losses(history)
```



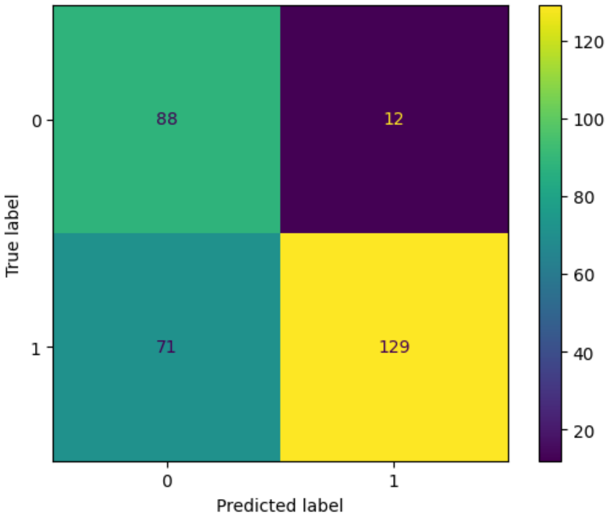
```
[ ]: # Check accuracy
     y_train_pred=model.predict(xs_train) >= 0.5
     y_test_pred=model.predict(xs_test_small) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 24s 3s/step
10/10 [==============================] - 20s 2s/step
```

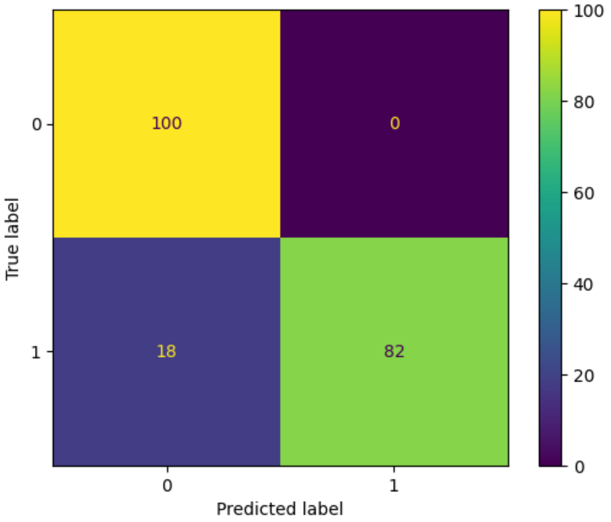
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.91
Test accuracy:  0.7233333333333334
```

```
[ ]: cm = confusion_matrix(y_test_small, y_test_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
cm = confusion_matrix(y_train, y_train_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



Improvement with more repetitions (8) of strong entangling layers with normalized data PCA 32 attributes Training 28 min

[ ]:
```
test_size=100
pred_split=np.array_split(y_test_pred,3)
y_sep_pred=pred_split[0]
```

```
y_ppt_pred=pred_split[1]
y_nppt_pred=pred_split[2]

score_sep = accuracy_score(y_sep_pred, np.full(test_size, 0))
score_ppt= accuracy_score(y_ppt_pred, np.full(test_size,1))
score_nppt= accuracy_score(y_nppt_pred, np.full(test_size,1))

print("SEP accuracy: ", score_sep)
print("PPT accuracy: ", score_ppt)
print("NPPT accuracy: ", score_nppt)
```

```
SEP accuracy:  0.88
PPT accuracy:  0.69
NPPT accuracy:  0.6
```

3 Case study: dataset with 0.5 PPT ratio

Replicating the working conditions from paper LARGE-SCALE QUANTUM SEPARABILITY THROUGH A REPRODUCIBLE MACHINE LEARNING LENS, we will work with - A training set of 200 samples: 100 separable and 100 entangled (with 0.5 ppt ratio, thus 50 samples ppt-ent and 50 samples nppt-ent). File *train_set_05.csv* - A test set of 300 samples, 100 separable, 100 ppt-entangled and 100 nppt-entangled. It is a reduced version of the papers test set consisting in 1000 samples per type. File *test_set_small.csv*

Additionally, we will use 5 fold cross validation for the training process of the models.

```
[ ]: training_data = pd.read_csv("/content/drive/MyDrive/tfg/x_train_05.csv", header=None)
     training_data.head()
```

```
[ ]:         0         1         2         3         4         5         6   ...
     0  0.139035  0.008456 -0.082087 -0.019300  0.009335  0.050326  0.010913
     1  0.027434 -0.024593 -0.004040 -0.009824  0.003811  0.012059  0.013763
     2  0.009734  0.003539  0.016788  0.005305  0.014389  0.013836  0.011906
     3  0.007171  0.000200  0.006113 -0.011893 -0.008216  0.047268 -0.002918
     4  0.016754  0.088911  0.170843  0.007160  0.037591 -0.159721 -0.014413


             74        75        76        77        78        79
     0  0.061386  0.077893  0.107214  0.120851  0.119377  0.126888
     1  0.045268  0.065257  0.075945  0.086621  0.088141  0.100372
     2  0.017251  0.065321  0.076891  0.090936  0.113257  0.113306
     3  0.043498  0.080326  0.100310  0.101018  0.113511  0.120475
     4  0.052332  0.105790  0.112462  0.076749  0.107003  0.119324

     [5 rows x 80 columns]
```

```
[ ]: training_data.describe()
```

```
[ ]:                0           1           2           3           4           5   ...
     count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
     mean     0.003763   -0.002582   -0.001872    0.003203   -0.002515   -0.001205
     std      0.035739    0.033113    0.033377    0.026297    0.026577    0.035588
```

```
min     -0.170321   -0.166935   -0.136005   -0.078350   -0.080244   -0.159721
25%     -0.016411   -0.022544   -0.016625   -0.007574   -0.014764   -0.015894
50%      0.001116    0.000261    0.000438    0.001154   -0.000650   -0.002426
75%      0.022831    0.013867    0.016098    0.011541    0.007547    0.017720
max      0.139035    0.096240    0.170843    0.102711    0.077133    0.117559


               77          78          79
count  200.000000  200.000000  200.000000
mean     0.097140    0.105141    0.111583
std      0.012903    0.011354    0.008890
min      0.064312    0.083020    0.092286
25%      0.088503    0.098364    0.105646
50%      0.096861    0.104572    0.111667
75%      0.104330    0.111644    0.116109
max      0.162879    0.162430    0.152799

[8 rows x 80 columns]
```

[ ]:

```
x_train = np.genfromtxt("/content/drive/MyDrive/tfg/x_train_05.csv",␣
 ↪delimiter=",",dtype=None)
y_train = np.genfromtxt("/content/drive/MyDrive/tfg/y_train_05.csv",␣
 ↪delimiter=",",dtype=None)

x_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/x_test_small.csv",␣
 ↪delimiter=",",dtype=None)
y_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/y_test_small.csv",␣
 ↪delimiter=",",dtype=None)
```

[ ]:

```
print("Training set:")
print("Size of the training set ", x_train.shape)
print("Size of the corresponding labels ", y_train.shape)

print("\nTest set:")
print("Size of the test set ", x_test_small.shape)
print("Size of the corresponding labels ", y_test_small.shape)
```

```
Training set:
Size of the training set  (200, 80)
Size of the corresponding labels  (200,)

Test set:
Size of the test set  (300, 80)
Size of the corresponding labels  (300,)
```

3.1 Exploratory data analysis

We will use Principal Component Analysis to reduce the dimensionality of the dataset. Since we will use quantum machine learning techniques to build a classifier on this data, our first step in that process will be data embedding. There are different data encoding built-in functions in Pennylane, however, the only suitable one for our case is Amplitude embedding due to the constraint in the number of qubits we can use in the current simulators. Amplitude encoding allows us to codify $N \leq 2^n$ attributes with n qubits. Our dataset has 80 attributes, with 7 qubits amplitude encoding could codify up to 128 attributes. In the initial testing, we have obtained best performance reducing the dimensionality of the dataset to 32 attributes using principal component analysis. This allows us to reduce to 5 qubits.

We will also check the effect of normalization on our dataset.

We will use the class *sklearn.decomposition.PCA*. First we will work with direclty with our data (no normalization)

```
[ ]: pca = PCA(n_components = 32)

     xs_train = pca.fit_transform(x_train)
     xs_test = pca.transform(x_test_small)
```

Once trained, the PCA object contains all the information about the components. We fixed the number of components to 32. Next, we will analyse the three main components.

These three rows correspond to the three principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by decreasing *explained_ variance_* (the amount of variance explained by each of the selected components)

```
[ ]: print(pca.components_.shape)
```

```
(32, 80)
```

```
[ ]: three_pc = pd.DataFrame(
         data=pca.components_[0:3],
         index = ['PC1', 'PC2', 'PC3']
     )
```

```
[ ]: three_pc
```

```
[ ]:            0         1         2         3         4         5         6   ...
     PC1 -0.078923  0.138744  0.347397 -0.082156 -0.010084 -0.157959  0.018403
     PC2 -0.046127  0.056054 -0.006340  0.038842  0.018734  0.008609 -0.047358
     PC3 -0.261291 -0.176917  0.008989  0.012620 -0.057868  0.357191  0.029919


               73        74        75        76        77        78        79
     PC1 -0.052625 -0.025488  0.013009 -0.000893 -0.013329 -0.002820 -0.008094
     PC2 -0.040785 -0.016024  0.008447 -0.013036 -0.008851  0.007211 -0.004051
     PC3  0.003536 -0.072748 -0.062288 -0.055809 -0.027001 -0.009286 -0.014112

     [3 rows x 80 columns]
```

```
[ ]: print('Porcentaje de varianza explicada por cada componente')
     print(pca.explained_variance_ratio_)
```

```
Porcentaje de varianza explicada por cada componente
[0.05263381 0.0494832  0.04331597 0.03946737 0.03424476 0.03411636
 0.03268133 0.02931714 0.02839679 0.02712453 0.02636718 0.02459969
 0.02388274 0.02320542 0.02227055 0.02117816 0.02100179 0.02023511
 0.01923234 0.01888048 0.01842044 0.01773041 0.01685939 0.0162822
```

```
0.01530907 0.01474871 0.01440209 0.01387777 0.01355994 0.01286114
0.01243935 0.01218028]
```

[ ]:

```
pca_df = pd.DataFrame(
    data = xs_train[:,0:3],
    columns = ['PC1', 'PC2', 'PC3']
)
pca_df = pd.concat([pca_df, pd.DataFrame(y_train, columns =['target'])[['target']]],␣
 ↪axis=1)
```

[ ]:

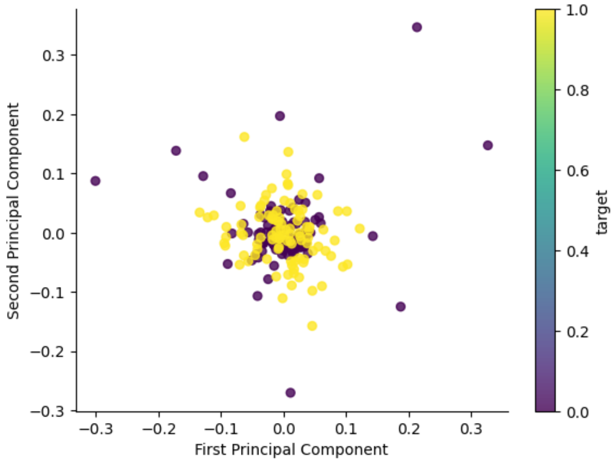
```
pca_df
```

[ ]:

```
          PC1       PC2       PC3  target
0   -0.089289 -0.052919 -0.008266     0.0
1   -0.016367 -0.006689 -0.012467     0.0
2    0.014952  0.042296 -0.025777     0.0
3   -0.031557 -0.017658  0.018171     0.0
4    0.326876  0.147360 -0.167084     0.0
..        ...       ...       ...     ...
195  0.032934 -0.044987 -0.048340     1.0
196  0.030219  0.039094 -0.039554     1.0
197  0.067205 -0.030824  0.026076     1.0
198  0.011818 -0.048264 -0.068982     1.0
199 -0.041580 -0.027801  0.058748     1.0

[200 rows x 4 columns]
```

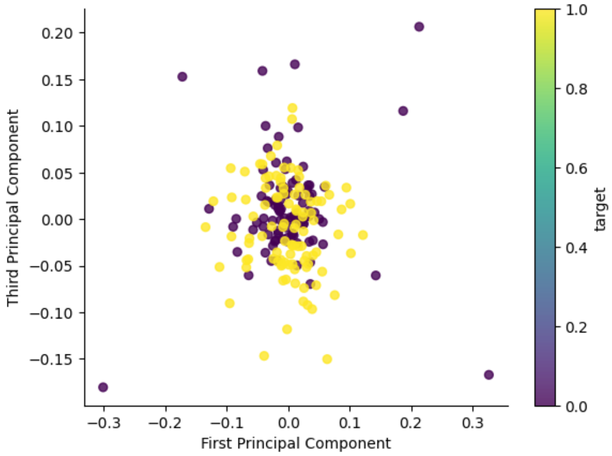
[ ]:

```
pca_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



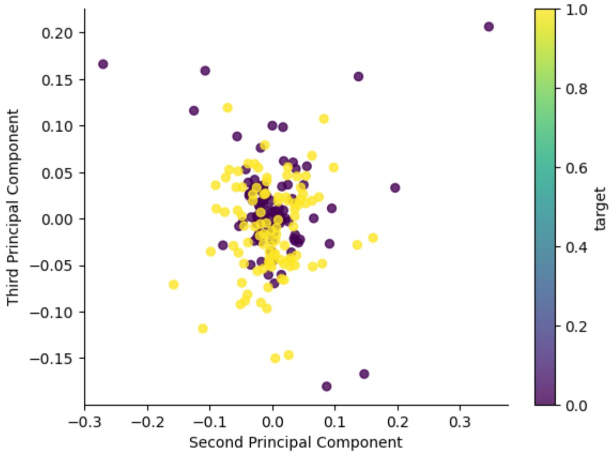
[ ]:

```
pca_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



[ ]:

```
pca_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8)
plt.xlabel('Second Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```

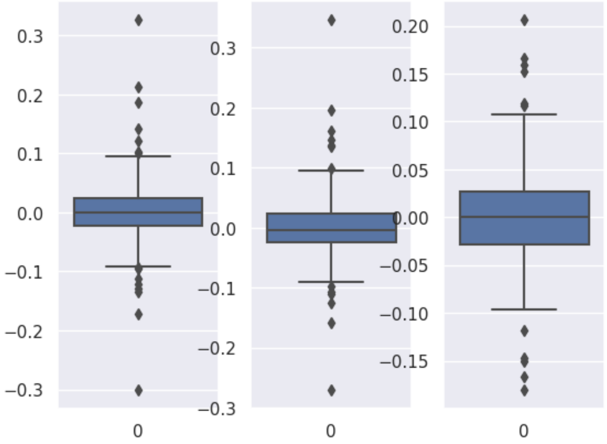


[ ]:

```
# Box Plot

#set seaborn plotting aesthetics as default
sns.set()

#define plotting region (2 rows, 2 columns)
fig, axes = plt.subplots(1, 3)

#create boxplot in each subplot
sns.boxplot(data=pca_df['PC1'], ax=axes[0])
sns.boxplot(data=pca_df['PC2'], ax=axes[1])
sns.boxplot(data=pca_df['PC3'], ax=axes[2])
```

[ ]:

```
<Axes: >
```



3.1.1 MaxAbsScaler normalization

[ ]:

```
scaler = MaxAbsScaler()
x_train_norm = scaler.fit_transform(x_train)

x_test_norm = scaler.transform(x_test_small)

# Restrict all the values to be between 0 and 1
x_test_norm = np.clip(x_test_norm,0,1)
```

[ ]:

```
pca = PCA(n_components = 32)

xs_train_norm = pca.fit_transform(x_train_norm)
xs_test_norm = pca.transform(x_test_norm)
```

[ ]:

```
three_pc_norm = pd.DataFrame(
    data=pca.components_[0:3],
```

```
    index = ['PC1', 'PC2', 'PC3']
)
```

[ ]:

```
three_pc_norm
```

[ ]:

```
            0         1         2         3         4         5         6   ...
PC1  0.024188  0.056195  0.037531  0.013884 -0.109444 -0.014466 -0.072613
PC2  0.000420  0.083452  0.119313 -0.104092  0.167000 -0.124731  0.058960
PC3 -0.123593 -0.040699 -0.073086  0.014512  0.088001  0.059851 -0.071889


           73        74        75        76        77        78        79
PC1 -0.085012 -0.015312  0.021536 -0.008675 -0.020402 -0.003547 -0.002765
PC2 -0.077326 -0.000914  0.024644  0.020783  0.008202  0.004571 -0.003491
PC3  0.005548 -0.023724 -0.019626 -0.016679  0.010794  0.012590 -0.003202

[3 rows x 80 columns]
```

[ ]:

```
print('Porcentaje de varianza explicada por cada componente')
print(pca.explained_variance_ratio_)
```

```
Porcentaje de varianza explicada por cada componente
[0.04667584 0.04370644 0.038514   0.03810517 0.03644829 0.03399039
 0.03291459 0.03232812 0.03079872 0.02836721 0.02774176 0.02559811
 0.02475009 0.02444303 0.02398773 0.02280793 0.02158439 0.02110549
 0.02043725 0.0198008  0.01908729 0.0179154  0.01718435 0.01674891
 0.01561015 0.01516731 0.01468464 0.01395486 0.01385951 0.01316067
 0.01292767 0.01250232]
```

[ ]:

```
pca_norm_df = pd.DataFrame(
    data = xs_train_norm[:,0:3],
    columns = ['PC1', 'PC2', 'PC3']
)
pca_norm_df = pd.concat([pca_norm_df, pd.DataFrame(y_train, columns␣
 ↪=['target'])[['target']]], axis=1)
```

[ ]:

```
pca_norm_df
```

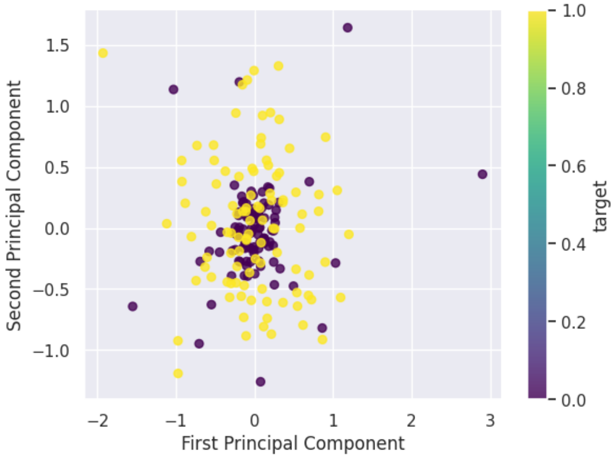
[ ]:

```
          PC1       PC2       PC3  target
0   -0.128141 -0.391311 -0.041418     0.0
1   -0.043548  0.094400  0.170526     0.0
2    0.180909  0.189010  0.086033     0.0
3   -0.198316 -0.271490 -0.151484     0.0
4    1.186498  1.642871 -0.648775     0.0
..        ...       ...       ...     ...
195 -0.028142  0.488250 -0.709229     1.0
196  0.580187 -0.000676 -0.809034     1.0
197  0.176704  0.513842 -0.930971     1.0
198 -0.520711  0.679900 -0.256616     1.0
199 -0.629642 -0.320575 -0.166607     1.0
```

```
[200 rows x 4 columns]
```

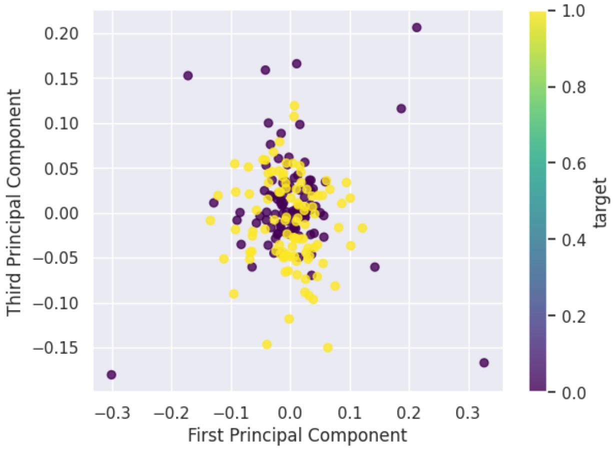
[ ]:

```
pca_norm_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8,␣
↪cmap='viridis')
plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



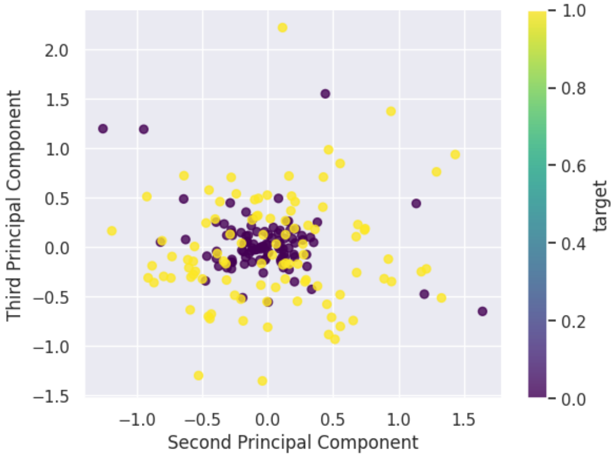
[ ]:

```
pca_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8,␣
↪cmap='viridis')
plt.xlabel('First Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



[ ]:

```
pca_norm_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8,␣
 ↪cmap='viridis')
plt.xlabel('Second Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```

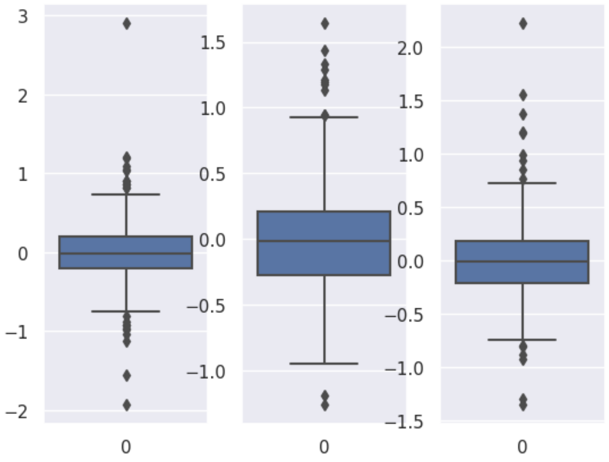


[ ]:

```
# Box Plot

#set seaborn plotting aesthetics as default
sns.set()

#define plotting region (2 rows, 2 columns)
fig, axes = plt.subplots(1, 3)

#create boxplot in each subplot
sns.boxplot(data=pca_norm_df['PC1'], ax=axes[0])
sns.boxplot(data=pca_norm_df['PC2'], ax=axes[1])
sns.boxplot(data=pca_norm_df['PC3'], ax=axes[2])
```

[ ]:

```
<Axes: >
```



3.2 Quantum Support Vector Machines

[ ]:

```
def detailed_accuracy(y_pred, size):
  test_size=size
  pred_split=np.array_split(y_test_pred,3)
  y_sep_pred=pred_split[0]
  y_ppt_pred=pred_split[1]
  y_nppt_pred=pred_split[2]

  score_sep = accuracy_score(y_sep_pred, np.full(test_size, 0))
  score_ppt= accuracy_score(y_ppt_pred, np.full(test_size,1))
  score_nppt= accuracy_score(y_nppt_pred, np.full(test_size,1))

  print("SEP accuracy: ", score_sep)
  print("PPT accuracy: ", score_ppt)
  print("NPPT accuracy: ", score_nppt)
```

[ ]:

```
def save_model(model, filename):
  dir="/content/drive/MyDrive/tfg/"+filename
  joblib.dump(model, dir)
  print("Model saved")
```

[ ]:

```
def load_model(model, filename):
  dir="/content/drive/MyDrive/tfg/"+filename
  return joblib.load(dir)
```

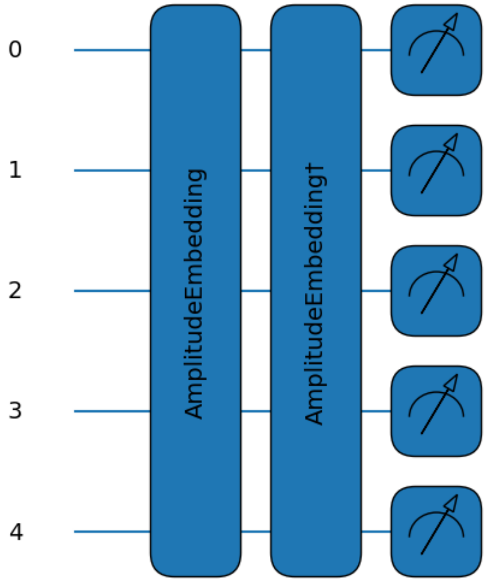
[ ]:

```
# Amplitude encoding of 64 variables using 5 qubits (can encode up to 32 inputs)

# Number of qubits of the system
nqubits = 5
# We define a device
dev = qml.device("lightning.qubit", wires = nqubits)
```

```
# We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
# operation equivalent to amplitude encoding of the first argument
@qml.qnode(dev)
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
 ↪normalize=True))    # We return an array with the probabilities fo measuring each␣
 ↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

[ ]:

```
fig, ax = qml.draw_mpl(kernel_circ)([1],[1])
fig.show()
```



[ ]:

```
def qkernel(A, B):
  return np.array([[kernel_circ(a,b)[0] for b in B] for a in A])
```

Now, we will activate GPU acceleration and check it

[ ]:

```
gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
```

```
  print('Not connected to a GPU')
else:
  print(gpu_info)
```

```
Mon Nov 13 14:59:13 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   47C    P8    10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
y_train_pred=svm.predict(xs_train)
```

[ ]:
```
tr_acc=accuracy_score(y_train_pred, y_train)
print("Train accuracy: ", tr_acc)
```

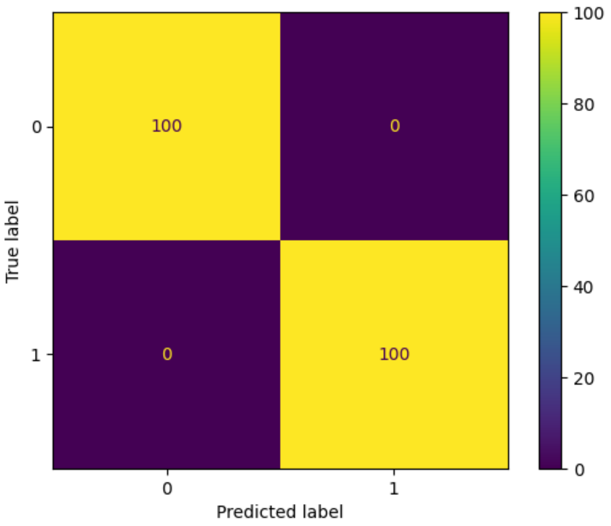
```
Train accuracy:  1.0
```

[ ]:
```
y_test_pred=svm.predict(xs_test)
```

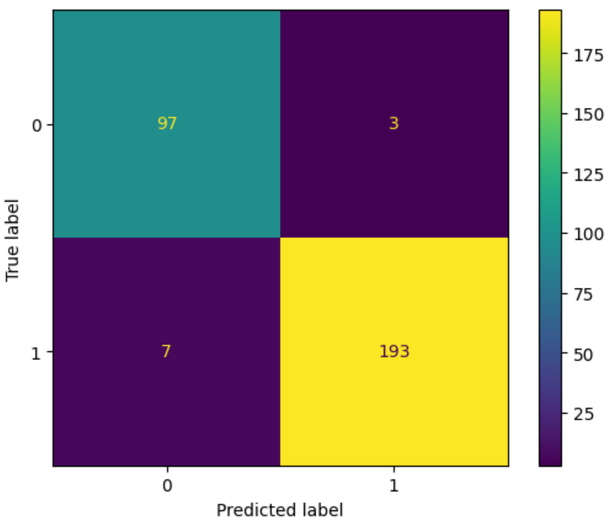
[ ]:
```
test_acc=accuracy_score(y_test_pred, y_test_small)
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.9666666666666667
```

[ ]:
```
# Train confussion matrix
cm = confusion_matrix(y_train, y_train_pred)
cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test confussion matrix
     cm = confusion_matrix(y_test_small, y_test_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test accuracy per type
     detailed_accuracy(y_test_pred, 100)
```

```
SEP accuracy:  0.97
PPT accuracy:  0.95
NPPT accuracy:  0.98
```

```
[ ]: from sklearn.model_selection import GridSearchCV, StratifiedKFold
```

```
[ ]: model_type = SVC()
     model_params = [{'kernel': [qkernel]}]
     model = GridSearchCV(model_type, model_params, cv=StratifiedKFold(shuffle=True)).
      ↪fit(xs_train, y_train)
     print('Training results :')
     print(model.best_params_)
     print(model.best_score_)
```

```
Training results :
{'kernel': <function qkernel at 0x788877ed0dc0>}
0.9650000000000001
```

Using a Support Vector Machine with a quantum kernel induced by Amplitude encoding with - 5 qubits - 32 attributes We have run with GPU acceleration the training in 12 minutes and predictions on the training set in 13 minutes. We have obtained an accuracy of 1.0 on the training set

Then, we have performed the predictions on the test set, which took 19 minutes. We have obtained an accuracy of 0.9667 on the test set.

With the predictions of the test set, we have also studied the accuracy on the predictions of each type of data: Separable, PPT-entangled and NPPT-entangled. - Accuracy of 0.97 on the separable data samples - Accuracy of 0.95 on the PPT entangled data samples - Accuracy of 0.98 on the NPPT entangled data samples.

Therefore, we can conclude that this QSVM performs great and is able to correctly clasify separable and entangled data, specially PPT entangled data (which is the "hardest" to distinguish from separable) with a great accuracy.

Above we can see the confussion matrices for classification of the training set and the test set.

Finally, we have run trained this model using GridSearchCV (just with this model exact configuration) with five fold cross validation using. This took more than 50 minutes to execute. We obtained 0.965 as best accuracy. We need to discard using GridSeachCV to perform an exhaustive search over differnet parameter values for a model due to the long executing time obtained just for one configuration and because the different configurations of a QSVM are not externally parametrised, but defined in the quantum circuit

Next we will train and predict this same model configuration on our normalized dataset to check if there is any difference in the accuracy

```
[ ]: svm = SVC(kernel = qkernel).fit(xs_train_norm, y_train)
```

```
[ ]: y_train_pred=svm.predict(xs_train_norm)
```

```
[ ]: tr_acc=accuracy_score(y_train_pred, y_train)
     print("Train accuracy: ", tr_acc)
```

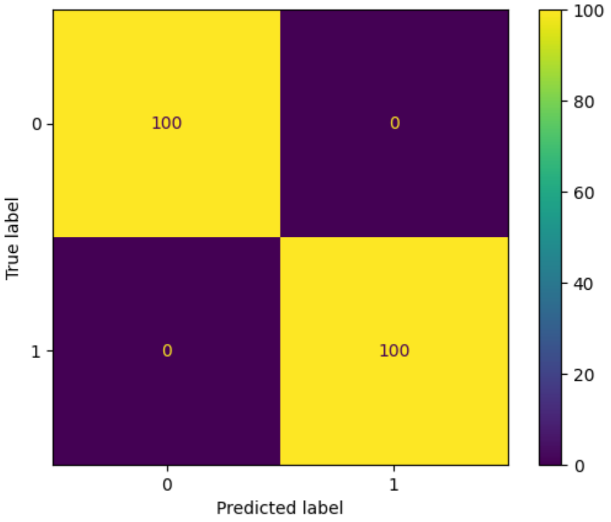
```
Train accuracy:  1.0
```

```
[ ]: y_test_pred=svm.predict(xs_test_norm)
```

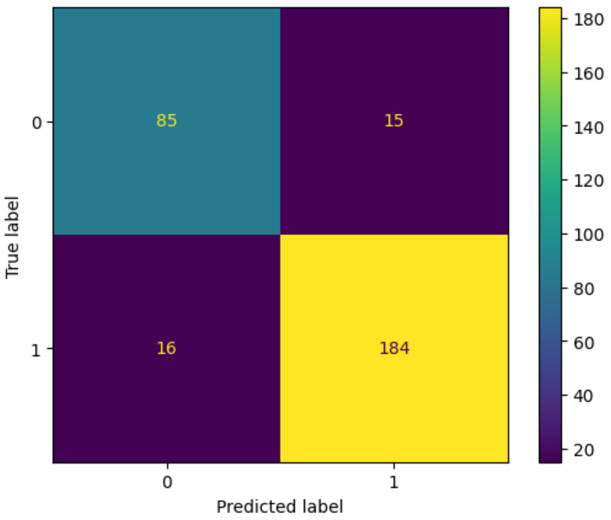
```
[ ]: test_acc=accuracy_score(y_test_pred, y_test_small)
     print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.8966666666666666
```

```
[ ]: # Train confussion matrix
     cm = confusion_matrix(y_train, y_train_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test confussion matrix
     cm = confusion_matrix(y_test_small, y_test_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test accuracy per type
     detailed_accuracy(y_test_pred, 100)
```

```
SEP accuracy:  0.85
PPT accuracy:  0.94
NPPT accuracy:  0.9
```

The use of normalized data (with MaxAbsScaler) does not improve the model neither the execution time. Indeed, the accuracy worsens from 0.9667 to 0.8967. From now on, we will continue working only with the not normalized dataset

3.3 3. Quantum Neural Networks

We will try different configurations of neural networks based on the promising results obtained in the first phase of trials. However, we cannot use GridSeachCV to select the best model because the parameters and changes that we apply in each configuration are performed in the quantum neural network circuit definition. We have created the function *fivefoldCV_qnn(circuit)* to perform five fold cross validation for every QNN circuit that we will test. The five fold cross validation code is inspired by https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-keras.md

```
[ ]: # We set a seed for the packages so the results are reproducible
     seed=4321
     np.random.seed(seed)
     tf.random.set_seed(seed)
```

```
[ ]: # pennylane works with doubles and tensorflow works with floats.
     # We ask tensorflow to work with doubles

     tf.keras.backend.set_floatx('float64')
```

```
[ ]: def fivefoldCV_qnn(circuit, x, y):
       # Define the K-fold Cross Validator
       kfold = KFold(n_splits=5, shuffle=True)
       acc_per_fold = []
       loss_per_fold = []

       # K-fold Cross Validation model evaluation
       fold_no = 1
       for train, test in kfold.split(x, y):

         # Define the model architecture
         method= "adjoint"
         tf.random.set_seed(seed)
         qnn = qml.QNode(circuit, dev, interface="tf", diff_method=method)
         nweights = 3*nreps*nqubits
         weights={"theta": nweights}
         # Keras layer containing qnn
         qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)
         # keras model
         model = tf.keras.models.Sequential([qlayer])
```

```
    # we choose adam optimizer with a learning rate of 0.005
    opt = tf.keras.optimizers.Adam(learning_rate=0.005)


    # binary cross entropy loss, because we are training a binary classifier
    model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Training our model
    history = model.fit(x[train], y[train], epochs = 50, shuffle = True,
                      validation_data = None, batch_size = 20, verbose=1)

    # Generate generalization metrics
    score = model.evaluate(x[test], y[test], verbose=0)
    # Check accuracy
    y_test_pred=model.predict(x[test]) >= 0.5
    test_acc = accuracy_score(y_test_pred, y[test])

    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {score}; accuracy of␣
 ↪{test_acc}')
    loss_per_fold.append(score)
    acc_per_fold.append(test_acc)


    # Increase fold number
    fold_no = fold_no + 1

  # == Provide average scores ==
  print('------------------------------------------------------------------------')
  print('Score per fold')
  for i in range(0, len(loss_per_fold)):
    print('------------------------------------------------------------------------')
    print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}')
  print('------------------------------------------------------------------------')
  print('Average scores for all folds:')
  print(f'> Loss: {np.mean(loss_per_fold)}')
  print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
  print('------------------------------------------------------------------------')
```

[ ]:

```
def performance(y_train_pred, y_train, y_test_pred, y_test):
  tr_acc = accuracy_score(y_train_pred, y_train)
  test_acc = accuracy_score(y_test_pred, y_test_small)

  tr_f1 = f1_score(y_train, y_train_pred)
  test_f1 = f1_score(y_test, y_test_pred)

  print("Train accuracy: ", tr_acc)
```

```
  print("Train F-1 score: ", tr_f1)

  print("\nTest accuracy: ", test_acc)
  print("Test F-1 score: ", test_f1)

  # Test accuracy per type
  print("\nTest accuracy broken down per type")
  detailed_accuracy(y_test_pred, 100)

  print("\n")
  # Train confusion matrix
  cm = confusion_matrix(y_train, y_train_pred)
  cm_display = ConfusionMatrixDisplay(cm).plot()
  cm_display.ax_.set_title("Train confusion matrix")

  # Test confusion matrix
  cm = confusion_matrix(y_test, y_test_pred)
  cm_display = ConfusionMatrixDisplay(cm).plot()
  cm_display.ax_.set_title("Test confusion matrix")
```

3.3.1 Twolocal variational form

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

[ ]:
```
# Two local variational form
def TwoLocal(nqubits, theta, reps=1):
  for r in range(reps):
    for i in range(nqubits):
      qml.RY(theta[r*nqubits+i], wires=i)
    for i in range(nqubits-1):
      qml.CNOT(wires=[i,i+1])

  for i in range(nqubits):
    qml.RY(theta[reps*nqubits+i], wires=i)
```

[ ]:
```
nqubits=5
dev=qml.device("lightning.qubit", wires=nqubits)

nreps=10

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

[ ]:
```
# Model 1 five fold cross validation
fivefoldCV_qnn(qnn_circuit,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.5885045654400779; accuracy of 0.675
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5742032743541083; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5726296517871493; accuracy of 0.675
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.6207334711746321; accuracy of 0.65
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5579251616057197; accuracy of 0.825
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5885045654400779 - Accuracy: 0.675
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5742032743541083 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5726296517871493 - Accuracy: 0.675
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6207334711746321 - Accuracy: 0.65
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5579251616057197 - Accuracy: 0.825
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5827992248723375
> Accuracy: 0.7150000000000001 (+- 0.06442049363362559)
------------------------------------------------------------------------
```

Since the performance we have obtained in the five fold cross validation process looks promising, we will finalise it by re-training our model with all the training data to make predictions on the test set.

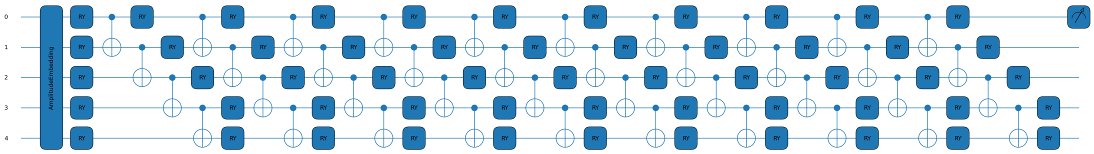
[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}
```

```python
# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_1 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_1.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:

```python
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
fig.show()
```



[ ]:

```python
history = model_1.fit(xs_train, y_train, epochs = 50, shuffle = True,
                      validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 21s 2s/step - loss: 0.9890
Epoch 2/50
10/10 [==============================] - 19s 2s/step - loss: 0.9460
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4731
Epoch 50/50
10/10 [==============================] - 16s 2s/step - loss: 0.4723
```

[ ]:

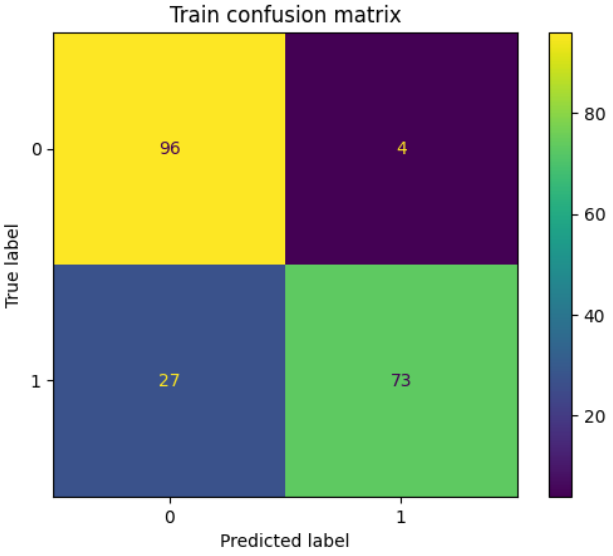
```python
# Compute predictions
y_train_pred=model_1.predict(xs_train) >= 0.5
y_test_pred=model_1.predict(xs_test) >= 0.5
```

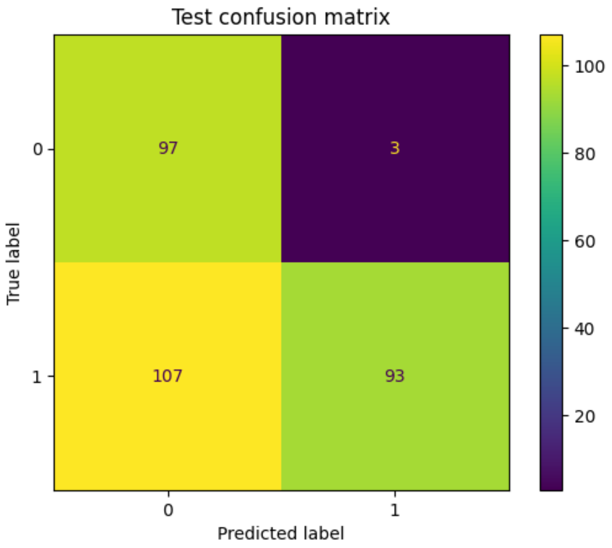
```
7/7 [==============================] - 8s 1s/step
10/10 [==============================] - 10s 960ms/step
```

[ ]:

```python
performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.845
Train F-1 score:  0.824858757062147

Test accuracy:  0.6333333333333333
Test F-1 score:  0.6283783783783784

Test accuracy broken down per type
SEP accuracy:  0.97
PPT accuracy:  0.45
NPPT accuracy:  0.48
```





Training full dataset 17 min

Normalized data

```
[ ]: # Model 2 five fold cross validation
     fivefoldCV_qnn(qnn_circuit,xs_train_norm, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.4644733953246071; accuracy of 0.875
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5745343208440845; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5817230830156055; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5524740872067613; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5461485356295814; accuracy of 0.825
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.4644733953246071 - Accuracy: 0.875
```

```
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5745343208440845 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5817230830156055 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5524740872067613 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5461485356295814 - Accuracy: 0.825
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5438706844041279
> Accuracy: 0.78 (+- 0.062048368229954284)
------------------------------------------------------------------------
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_2 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_2.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
history = model_2.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

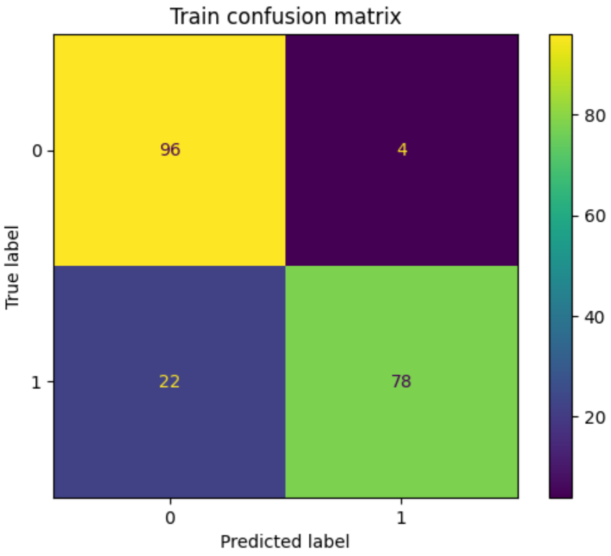
```
Epoch 1/50
10/10 [==============================] - 19s 2s/step - loss: 0.7627
Epoch 2/50
10/10 [==============================] - 28s 3s/step - loss: 0.7409
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4778
Epoch 50/50
10/10 [==============================] - 18s 2s/step - loss: 0.4776
```

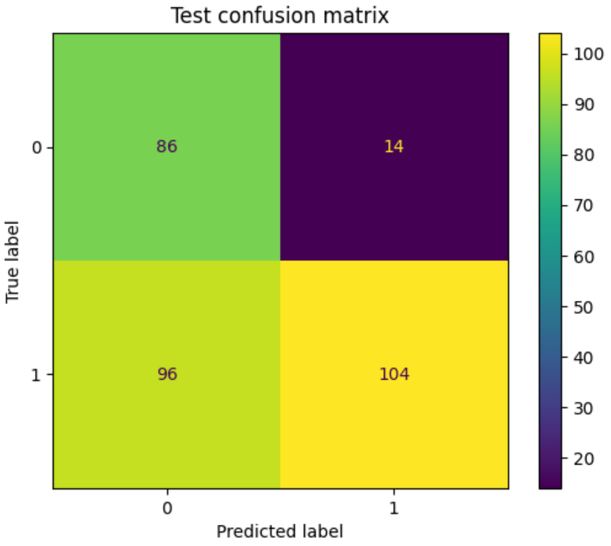
```
[ ]: # Check accuracy
     y_train_pred=model_2.predict(xs_train_norm) >= 0.5
     y_test_pred=model_2.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 9s 1s/step
10/10 [==============================] - 11s 1s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.87
Train F-1 score:  0.8571428571428571

Test accuracy:  0.6333333333333333
Test F-1 score:  0.6540880503144655

Test accuracy broken down per type
SEP accuracy:  0.86
PPT accuracy:  0.49
NPPT accuracy:  0.55
```





Training full dataset 15 min

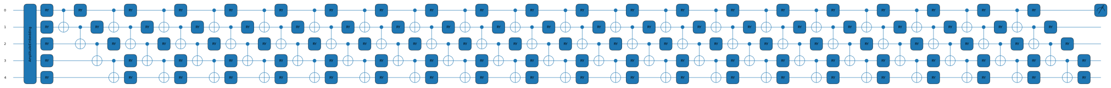
20 repetitions Next, we will try again with this variational form seeking for better results by increasing the number of repetitions. We need to improve our QNN to detect entanglement better, because in both previous experiments there is a high number of false negatives.

[ ]:
```
nqubits=5
dev=qml.device("lightning.qubit", wires=nqubits)

nreps=20

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)
```

```
# keras model
model_3 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_3.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
fig.show()
```



[ ]:
```
# Training our model
history = model_3.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 29s 3s/step - loss: 0.9886
Epoch 2/50
10/10 [==============================] - 25s 3s/step - loss: 0.9227
.
.
.
Epoch 49/50
10/10 [==============================] - 25s 3s/step - loss: 0.4093
Epoch 50/50
10/10 [==============================] - 30s 3s/step - loss: 0.4074
```

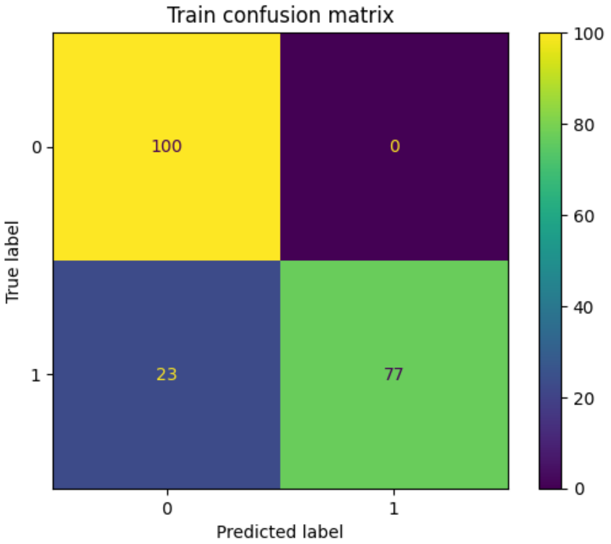
[ ]:
```
# Check accuracy
y_train_pred=model_3.predict(xs_train) >= 0.5
y_test_pred=model_3.predict(xs_test) >= 0.5
```

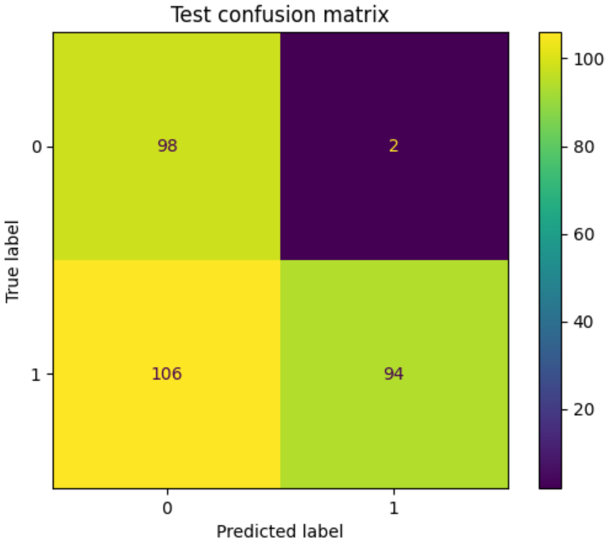
```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 19s 2s/step
```

[ ]:
```
performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.885
Train F-1 score:  0.8700564971751412

Test accuracy:  0.64
Test F-1 score:  0.6351351351351351
```

```
Test accuracy broken down per type
SEP accuracy:  0.98
PPT accuracy:  0.44
NPPT accuracy:  0.5
```





Training full dataset 24 min

We have tried with 15 and 20 repetitions and while it is true that the increase in the numer of repetitions slightly increases the accuracy, the tradeoff performance-executing time is not worth. The accuracy on the test set, specially detecting entanglement is really poor and the training time doubles, making it really costly to perform five fold cross validation.

3.3.2 StrongEntanglingLayers

[ ]:

```
nqubits=5
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 8
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))
```

[ ]:

```
# Model 4 five fold cross validation
fivefoldCV_qnn(qnn_circuit_strong,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.5192811509641961; accuracy of 0.725
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5206793387352479; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.502250655284547; accuracy of 0.8
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5195175620979109; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5569835664692083; accuracy of 0.725
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
```

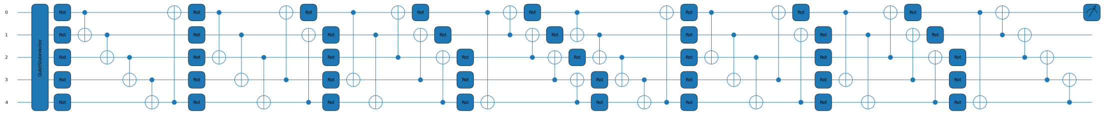
```
> Fold 1 - Loss: 0.5192811509641961 - Accuracy: 0.725
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5206793387352479 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 3 - Loss: 0.502250655284547 - Accuracy: 0.8
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5195175620979109 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5569835664692083 - Accuracy: 0.725
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5237424547102221
> Accuracy: 0.76 (+- 0.030000000000000023)
------------------------------------------------------------------------
```

[ ]:

```python
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_4 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_4.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:

```python
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn,expansion_strategy="device")(xs_train,theta)
fig.show()
```



[ ]:

```
# Training our model
history = model_4.fit(xs_train, y_train, epochs = 50, shuffle = True,
                      validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 36s 4s/step - loss: 0.7253
Epoch 2/50
10/10 [==============================] - 28s 3s/step - loss: 0.7125
.
.
.
Epoch 49/50
10/10 [==============================] - 33s 3s/step - loss: 0.4707
Epoch 50/50
10/10 [==============================] - 33s 3s/step - loss: 0.4697
```

[ ]:

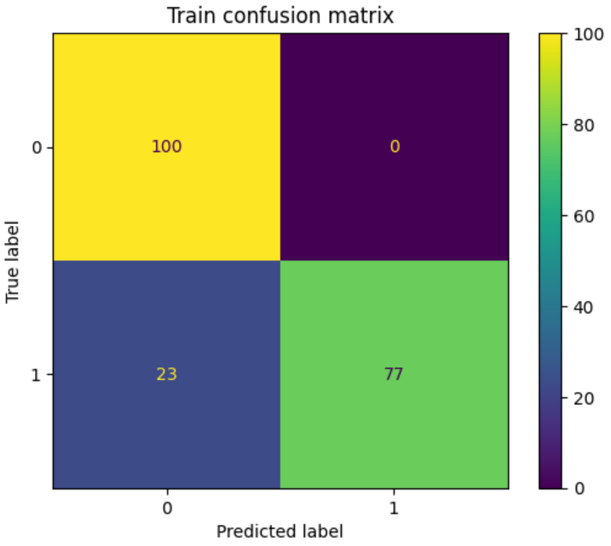
```
# Check accuracy
y_train_pred=model_4.predict(xs_train) >= 0.5
y_test_pred=model_4.predict(xs_test) >= 0.5
```

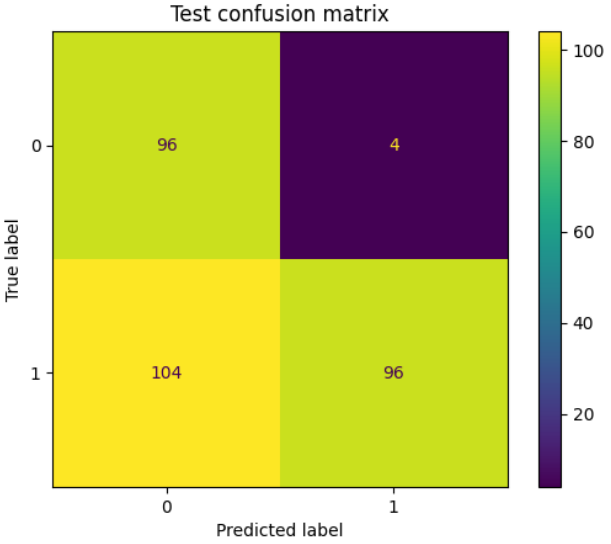
```
7/7 [==============================] - 15s 2s/step
10/10 [==============================] - 20s 2s/step
```

[ ]:

```
performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.885
Train F-1 score:  0.8700564971751412

Test accuracy:  0.64
Test F-1 score:  0.64

Test accuracy broken down per type
SEP accuracy:  0.96
PPT accuracy:  0.46
NPPT accuracy:  0.5
```





Training time full dataset 26 min

StrongEntanglingLayers variational form training time increases as the numper of repetitions increases. We have observed in previous tests that the increase in the number of repetitions improves the performance of the QNN. For 8 repetitions, it took 2h for training with five fold cross validation and 26 minutes for training with the entire training set. Therefore, we cannot try with a higher number of repetitions because of the high training time. With this model we have still obtained a poor entanglement detection.

Normalized data

```
[ ]: # Model 4n five fold cross validation (normalized data)
     fivefoldCV_qnn(qnn_circuit_strong,xs_train_norm, y_train)
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_4n = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_4n.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_4n.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 33s 3s/step - loss: 0.7903
Epoch 2/50
10/10 [==============================] - 34s 4s/step - loss: 0.7656
.
.
.
Epoch 49/50
10/10 [==============================] - 33s 3s/step - loss: 0.4556
Epoch 50/50
10/10 [==============================] - 28s 3s/step - loss: 0.4550
```

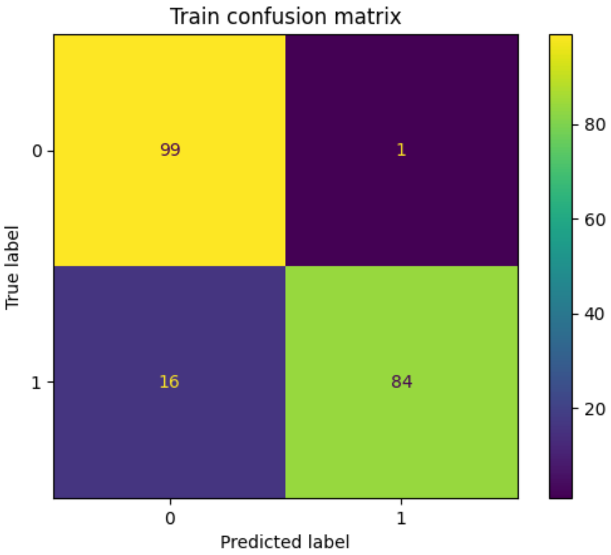
```
[ ]: # Check accuracy
     y_train_pred=model_4n.predict(xs_train_norm) >= 0.5
     y_test_pred=model_4n.predict(xs_test_norm) >= 0.5
```

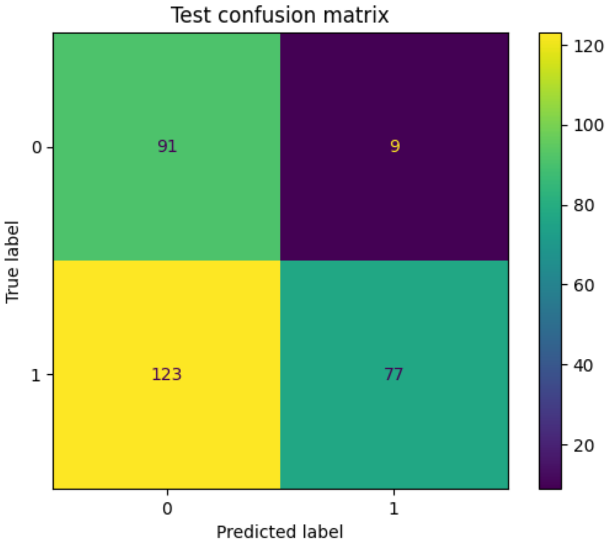
```
7/7 [==============================] - 12s 2s/step
10/10 [==============================] - 20s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.915
Train F-1 score:  0.9081081081081082
```

```
Test accuracy:  0.56
Test F-1 score:  0.5384615384615384

Test accuracy broken down per type
SEP accuracy:  0.91
PPT accuracy:  0.42
NPPT accuracy:  0.35
```





Training full dataset 26 min

3.3.3 BasicEntanglerLayers

It uses by default the single qubit X rotation gate

[ ]:

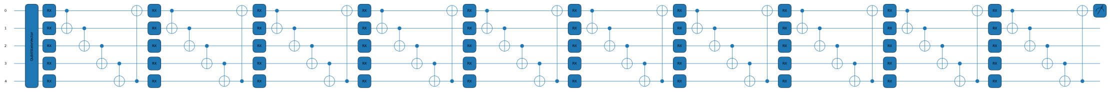
```
nqubits=5
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 10


def qnn_circuit_basic(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
↪normalize=True)
  qml.BasicEntanglerLayers(weights=theta, wires=range(nqubits))
  return qml.expval(qml.Hermitian(M, wires = [0]))

n_layers = 6
weights = {"theta": (nreps, nqubits)}
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit_basic, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_5 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_5.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:

```
theta=np.random.rand(nreps,nqubits)

fig, ax = qml.draw_mpl(qnn,expansion_strategy="device")(xs_train,theta)
fig.show()
```



```
[ ]: # Training our model
     history = model_5.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

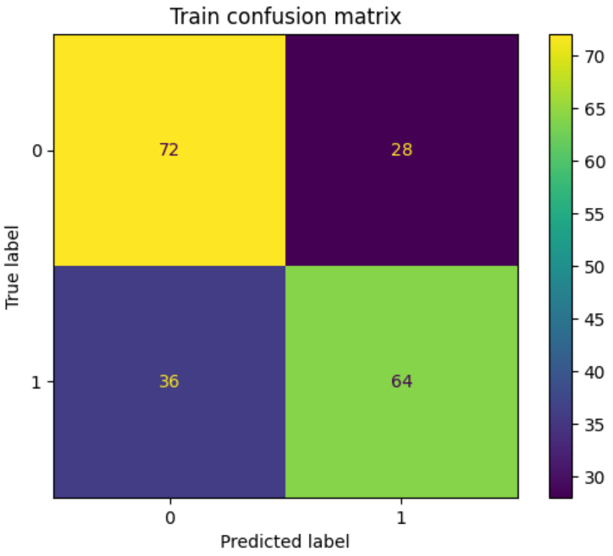
```
Epoch 1/50
10/10 [==============================] - 23s 2s/step - loss: 0.6810
Epoch 2/50
10/10 [==============================] - 27s 3s/step - loss: 0.6728
.
.
.
Epoch 49/50
10/10 [==============================] - 15s 2s/step - loss: 0.6456
Epoch 50/50
10/10 [==============================] - 16s 1s/step - loss: 0.6458
```

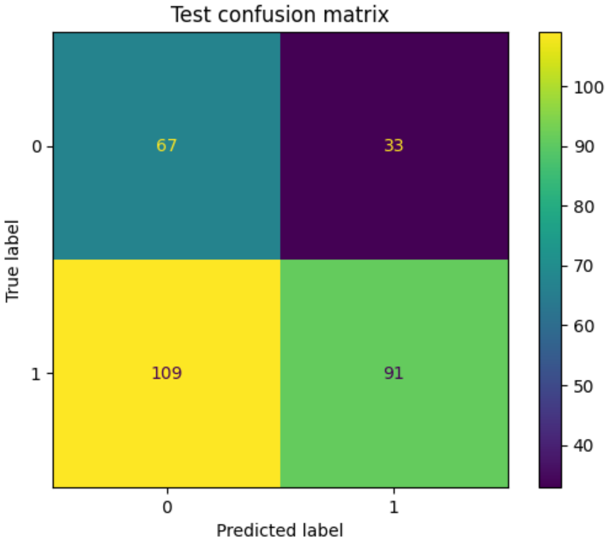
```
[ ]: y_train_pred=model_5.predict(xs_train) >= 0.5
     y_test_pred=model_5.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 7s 966ms/step
10/10 [==============================] - 10s 938ms/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.68
Train F-1 score:  0.6666666666666666

Test accuracy:  0.5266666666666666
Test F-1 score:  0.5617283950617283

Test accuracy broken down per type
SEP accuracy:  0.67
PPT accuracy:  0.48
NPPT accuracy:  0.43
```





Training full dataset 20 min

We cannot apply our five fold cross validation function because of the shape of the weights of this variational form. Anyway, this is the worst accuracy

3.3.4 Hybrid model

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit_basic, dev, interface="tf", diff_method=method)

     clayer1 = tf.keras.layers.Input(32)
     clayer2 = tf.keras.layers.Dense(32, activation="sigmoid")

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_6 = tf.keras.models.Sequential([clayer1, clayer2, qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_6.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_6.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

```
Epoch 1/50

WARNING:tensorflow:Gradients do not exist for variables ['dense_2/kernel:0',
'dense_2/bias:0'] when minimizing the loss. If you're using `model.compile()`,
did you forget to provide a `loss`argument?

 1/10 [==>...] - ETA: 1:33 - loss: 0.6914

WARNING:tensorflow:Gradients do not exist for variables ['dense_2/kernel:0',
'dense_2/bias:0'] when minimizing the loss. If you're using `model.compile()`,
did you forget to provide a `loss`argument?

 2/10 [=====>...] - ETA: 15s - loss: 0.6921

WARNING:tensorflow:Gradients do not exist for variables ['dense_2/kernel:0',
'dense_2/bias:0'] when minimizing the loss. If you're using `model.compile()`,
did you forget to provide a `loss`argument?
```

```
[ ]: # Check accuracy
     y_train_pred=model_6.predict(xs_train) >= 0.5
     y_test_pred=model_6.predict(xs_test) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 10s 1s/step
10/10 [==============================] - 9s 884ms/step
```

[ ]:

```python
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.555
Test accuracy:  0.46
```

[ ]:

```python
nqubits=4
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 2
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:

```python
tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit_strong, dev, interface="tf")

input = tf.keras.Input(shape=(32,))
clayer = tf.keras.layers.Dense(16, use_bias=False)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights_strong, output_dim=1)

# keras model
model_7 = tf.keras.models.Sequential([input, clayer, qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_7.compile(opt, loss='binary_crossentropy')

model_7.summary()
```

```
Model: "sequential_10"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 16)                512

 keras_layer_12 (KerasLayer)  (None, 1)                0 (unused)

=================================================================
Total params: 512
Trainable params: 512
Non-trainable params: 0
_________________________________________________________________
```

[ ]:

```python
# Training our model
history = model_7.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50

WARNING:tensorflow:Gradients do not exist for variables ['dense_16/kernel:0']
when minimizing the loss. If you're using `model.compile()`, did you forget to
provide a `loss`argument?

 1/10 [==>...] - ETA: 4:20 - loss: 0.7640

WARNING:tensorflow:Gradients do not exist for variables ['dense_16/kernel:0']
when minimizing the loss. If you're using `model.compile()`, did you forget to
provide a `loss`argument?

 2/10 [=====>...] - ETA: 26s - loss: 0.7501
```

[ ]:

```python
# Check accuracy
y_train_pred=model_7.predict(xs_train) >= 0.5
y_test_pred=model_7.predict(xs_test) >= 0.5

tr_acc = accuracy_score(y_train_pred, y_train)
test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 6s 620ms/step
```

[ ]:

```python
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.445
Test accuracy:  0.4666666666666667
```

4 Case study: dataset with 1.0 PPT ratio

- A training set of 200 samples: 100 separable and 100 entangled (with 1.0 ppt ratio, thus 100 samples ppt-ent and 0 samples nppt-ent). File *train_ set.csv*
- A test set of 300 samples, 100 separable, 100 ppt-entangled and 100 nppt-entangled. It is a reduced version of the papers test set consisting in 1000 samples per type. File *test_ set_ small.csv*

Additionally, we will use 5 fold cross validation for the training process of the models.

[ ]:

```
training_data = pd.read_csv("/content/drive/MyDrive/tfg/x_train_1.csv", header=None)
training_data.head()
```

[ ]:

```
          0         1         2         3         4         5         6   ...
0  0.002405 -0.013551  0.014313  0.007182 -0.008649 -0.003936 -0.002174
1 -0.006249  0.010679 -0.000343 -0.003259 -0.001029 -0.008732 -0.002129
2  0.012010  0.006747  0.006879  0.000910 -0.001017 -0.003304 -0.003681
3 -0.147372 -0.009079 -0.008336 -0.021830 -0.012960 -0.004989  0.054233
4 -0.010161 -0.021211 -0.009640 -0.004726  0.000182 -0.013206 -0.001762


         74        75        76        77        78        79
0  0.045722  0.076358  0.083020  0.093225  0.105654  0.109198
1  0.061884  0.080446  0.095130  0.103344  0.103996  0.110223
2  0.028129  0.060973  0.073608  0.078672  0.095634  0.102708
3  0.105213  0.087479  0.105942  0.130937  0.111882  0.118560
4  0.055780  0.075803  0.095085  0.105444  0.107857  0.114460

[5 rows x 80 columns]
```

[ ]:

```
training_data.describe()
```

[ ]:

```
                0           1           2           3           4           5   ...
count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
mean     0.004797    0.000064    0.000108   -0.000144    0.000498    0.001381
std      0.028860    0.029620    0.036638    0.025325    0.026439    0.032549
min     -0.147372   -0.081901   -0.187091   -0.069290   -0.080485   -0.159721
25%     -0.011422   -0.019337   -0.015301   -0.011488   -0.010467   -0.014435
50%      0.005595    0.000423    0.000219   -0.000011   -0.000762    0.001258
75%      0.024328    0.016108    0.018386    0.007331    0.010476    0.018243
max      0.078962    0.099702    0.170843    0.073404    0.087689    0.114851


               77          78          79
count  200.000000  200.000000  200.000000
mean     0.097276    0.106195    0.112494
std      0.013057    0.010411    0.008404
min      0.057148    0.059018    0.070539
25%      0.091099    0.100307    0.108088
50%      0.096756    0.107006    0.112603
75%      0.104528    0.111522    0.116679
max      0.162879    0.162101    0.152014

[8 rows x 80 columns]
```

[ ]:

```
x_train = np.genfromtxt("/content/drive/MyDrive/tfg/x_train_1.csv",␣
 ↪delimiter=",",dtype=None)
y_train = np.genfromtxt("/content/drive/MyDrive/tfg/y_train_1.csv",␣
 ↪delimiter=",",dtype=None)
```

4.1 Exploratory data analysis

```
[ ]: pca = PCA(n_components = 32)

     xs_train = pca.fit_transform(x_train)
     xs_test = pca.transform(x_test_small)
```

```
[ ]: three_pc = pd.DataFrame(
         data=pca.components_[0:3],
         index = ['PC1', 'PC2', 'PC3']
     )
```

```
[ ]: three_pc
```

```
[ ]:            0         1         2         3         4         5         6   ...
     PC1 -0.161845 -0.022361 -0.034527  0.045538  0.044388  0.110899  0.046586
     PC2 -0.041790  0.087543 -0.464655  0.028212 -0.038200  0.201786  0.081180
     PC3 -0.124399  0.134612 -0.014800  0.033860 -0.003348 -0.382316  0.007777


              73        74        75        76        77        78        79
     PC1  0.057525  0.096779  0.060026  0.031667  0.003717 -0.010086  0.007219
     PC2  0.033076  0.007223  0.007607  0.009609  0.023536  0.023750  0.023123
     PC3  0.035481  0.079102  0.033847  0.058791  0.098434  0.067680  0.049390

     [3 rows x 80 columns]
```

```
[ ]: print('Porcentaje de varianza explicada por cada componente')
     print(pca.explained_variance_ratio_)
```

```
Porcentaje de varianza explicada por cada componente
[0.05654583 0.05482265 0.04150263 0.03946556 0.03682448 0.03483101
 0.0316728  0.02962571 0.0272871  0.02612473 0.0254304  0.02540558
 0.02431726 0.02338352 0.02163313 0.02123271 0.02030116 0.01983655
 0.01857565 0.01796784 0.01784044 0.01704973 0.01620598 0.01576475
 0.01498603 0.01467931 0.01376626 0.01358834 0.01350699 0.01288878
 0.01268894 0.01245    ]
```

```
[ ]: pca_df = pd.DataFrame(
         data = xs_train[:,0:3],
         columns = ['PC1', 'PC2', 'PC3']
     )
     pca_df = pd.concat([pca_df, pd.DataFrame(y_train, columns =['target'])[['target']]],␣
      ↪axis=1)
```

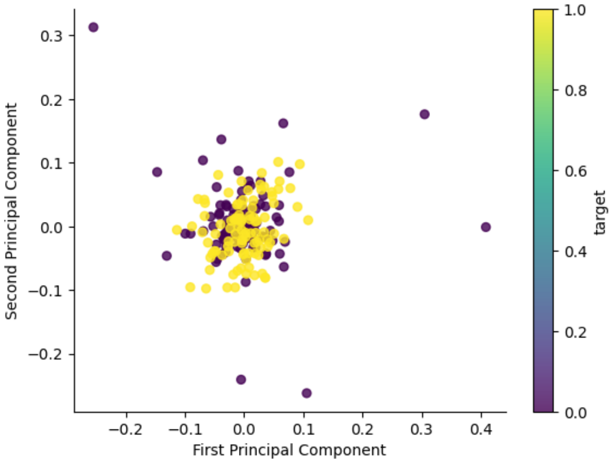
```
[ ]: pca_df
```

```
[ ]:           PC1       PC2       PC3  target
     0   -0.033562 -0.021375  0.001327     0.0
     1   -0.007987 -0.004189  0.006105     0.0
     2   -0.025172 -0.000218 -0.001940     0.0
```

```
3    0.408645 -0.001203  0.173139     0.0
4    0.019979  0.027201  0.054462     0.0
..        ...       ...       ...     ...
195  0.024591 -0.045316 -0.063818     1.0
196 -0.015872 -0.024501  0.051496     1.0
197  0.009933  0.010972  0.049546     1.0
198 -0.065871  0.036739 -0.000013     1.0
199 -0.025628 -0.039407 -0.033502     1.0

[200 rows x 4 columns]
```

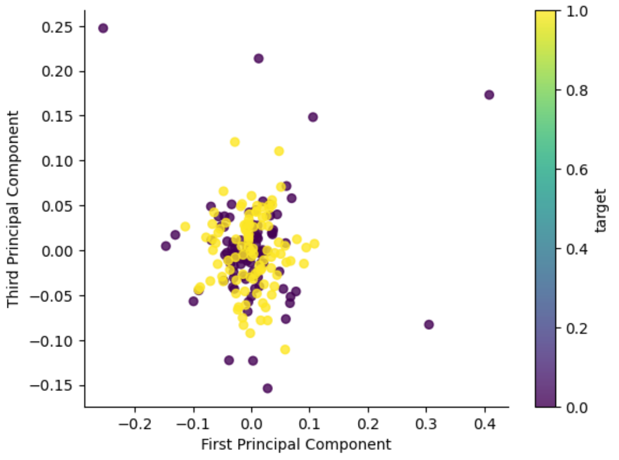
[ ]:

```
pca_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



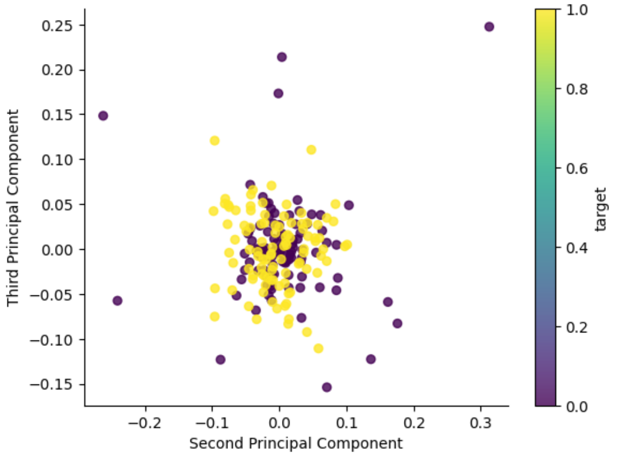
[ ]:

```
pca_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



[ ]:

```python
pca_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8)
plt.xlabel('Second Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```

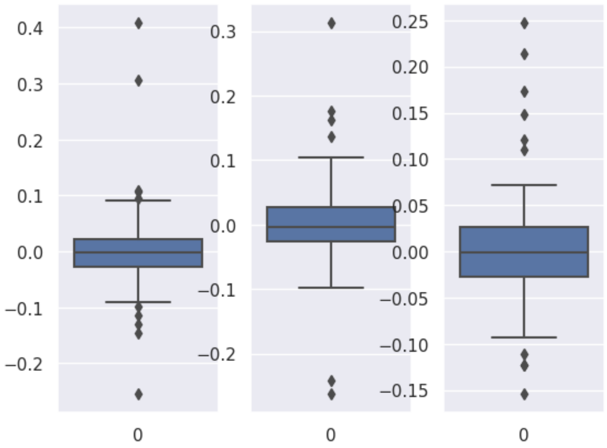


[ ]:

```python
# Box Plot
import seaborn as sns

#set seaborn plotting aesthetics as default
sns.set()

#define plotting region (2 rows, 2 columns)
```

```python
fig, axes = plt.subplots(1, 3)

#create boxplot in each subplot
sns.boxplot(data=pca_df['PC1'], ax=axes[0])
sns.boxplot(data=pca_df['PC2'], ax=axes[1])
sns.boxplot(data=pca_df['PC3'], ax=axes[2])
```

[ ]:
```
<Axes: >
```



MaxAbsScaler normalization

[ ]:
```python
scaler = MaxAbsScaler()
x_train_norm = scaler.fit_transform(x_train)

x_test_norm = scaler.transform(x_test_small)

# Restrict all the values to be between 0 and 1
x_test_norm = np.clip(x_test_norm,0,1)
```

[ ]:
```python
pca = PCA(n_components = 32)

xs_train_norm = pca.fit_transform(x_train_norm)
xs_test_norm = pca.transform(x_test_norm)
```

[ ]:
```python
three_pc_norm = pd.DataFrame(
    data=pca.components_[0:3],
    index = ['PC1', 'PC2', 'PC3']
)
```

[ ]:
```python
three_pc_norm
```

[ ]:
```
            0         1         2         3         4         5         6  ...
PC1 -0.088289  0.179963 -0.152478  0.172708  0.004456  0.057968  0.148815
```

```
PC2  0.037075  0.007439  0.065940  0.042914  0.202427  0.005615 -0.033994
PC3  0.027372  0.114360 -0.066136  0.126286  0.053287 -0.102089 -0.061799


           73        74        75        76        77        78        79
PC1  0.059820  0.079378  0.056832  0.027311  0.021976  0.013616  0.018079
PC2 -0.005179 -0.017332  0.010894  0.011215 -0.009000 -0.010088 -0.007949
PC3 -0.006481  0.000265  0.009991  0.015081  0.023333  0.024105  0.017907

[3 rows x 80 columns]
```

[ ]:

```python
print('Porcentaje de varianza explicada por cada componente')
print(pca.explained_variance_ratio_)
```

```
Porcentaje de varianza explicada por cada componente
[0.04519081 0.04305087 0.04040429 0.03852013 0.0358103  0.03383065
 0.03208468 0.03121227 0.03024195 0.02948633 0.02856054 0.02657779
 0.02529048 0.02499907 0.02368502 0.02338174 0.02270439 0.02130465
 0.02036222 0.01929434 0.01862555 0.01833084 0.0176221  0.01709922
 0.01623152 0.015521   0.01503582 0.01450377 0.01365743 0.01317983
 0.01273295 0.01217936]
```

[ ]:

```python
pca_norm_df = pd.DataFrame(
    data = xs_train_norm[:,0:3],
    columns = ['PC1', 'PC2', 'PC3']
)
pca_norm_df = pd.concat([pca_norm_df, pd.DataFrame(y_train, columns␣
 ↪=['target'])[['target']]], axis=1)
```

[ ]:

```python
pca_norm_df
```

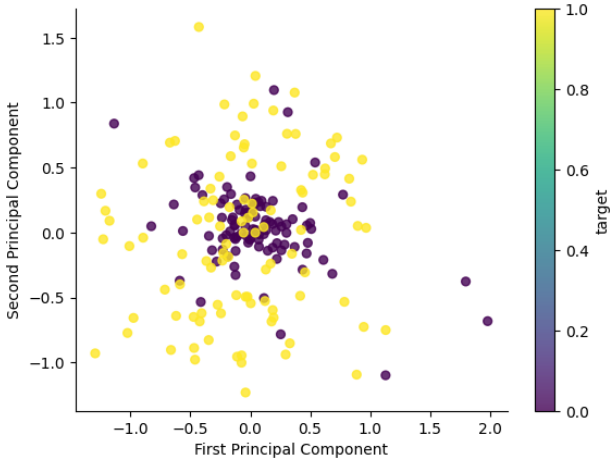
[ ]:

```
          PC1       PC2       PC3  target
0   -0.230297 -0.045281  0.067306     0.0
1   -0.016277  0.117728 -0.071714     0.0
2   -0.122600 -0.141607  0.226814     0.0
3    1.981290 -0.681880 -0.798570     0.0
4    0.114172 -0.504560  0.411078     0.0
..        ...       ...       ...     ...
195 -0.215890  0.986376 -0.324199     1.0
196  0.295155 -0.938925  0.137995     1.0
197  0.456633 -0.306292  0.033884     1.0
198 -0.662429 -0.903509  0.070498     1.0
199 -0.461377 -0.978318 -0.719641     1.0

[200 rows x 4 columns]
```

[ ]:

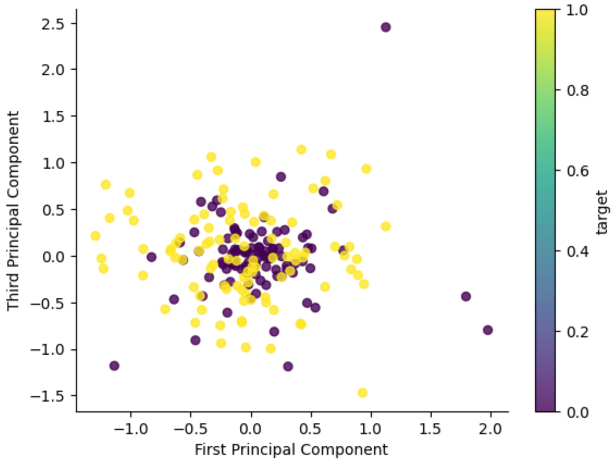
```python
pca_norm_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8,␣
 ↪cmap='viridis')
plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')
```

```python
plt.gca().spines[['top', 'right',]].set_visible(False)
```



[ ]:

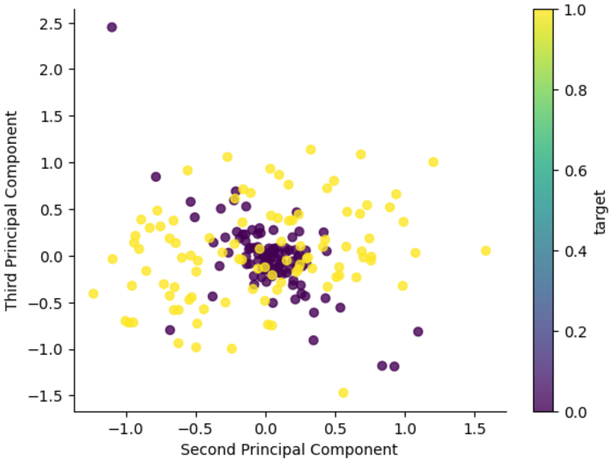
```python
pca_norm_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8, cmap='viridis')
plt.xlabel('First Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



[ ]:

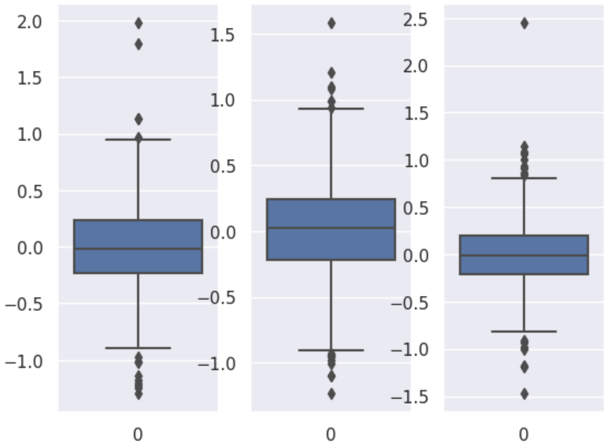
```python
pca_norm_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8, cmap='viridis')
```

```
plt.xlabel('Second Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: # Box Plot

     #set seaborn plotting aesthetics as default
     sns.set()

     #define plotting region (2 rows, 2 columns)
     fig, axes = plt.subplots(1, 3)

     #create boxplot in each subplot
     sns.boxplot(data=pca_norm_df['PC1'], ax=axes[0])
     sns.boxplot(data=pca_norm_df['PC2'], ax=axes[1])
     sns.boxplot(data=pca_norm_df['PC3'], ax=axes[2])
```

```
[ ]: <Axes: >
```



4.2 Quantum Support Vector Machines

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
y_train_pred=svm.predict(xs_train)
```

[ ]:
```
tr_acc=accuracy_score(y_train_pred, y_train)
print("Train accuracy: ", tr_acc)
```

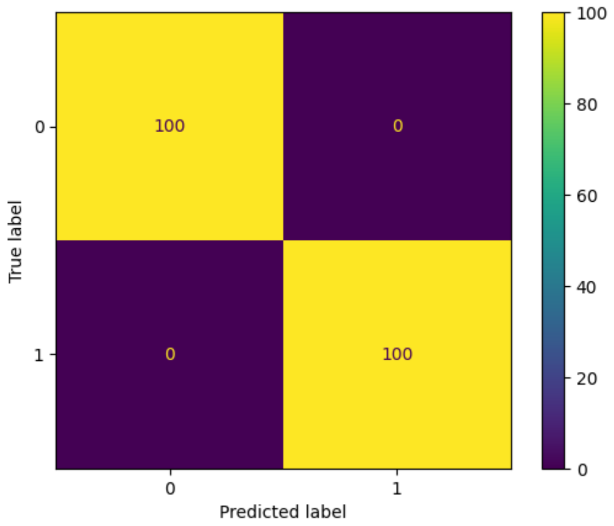
```
Train accuracy:  1.0
```

[ ]:
```
y_test_pred=svm.predict(xs_test)
```

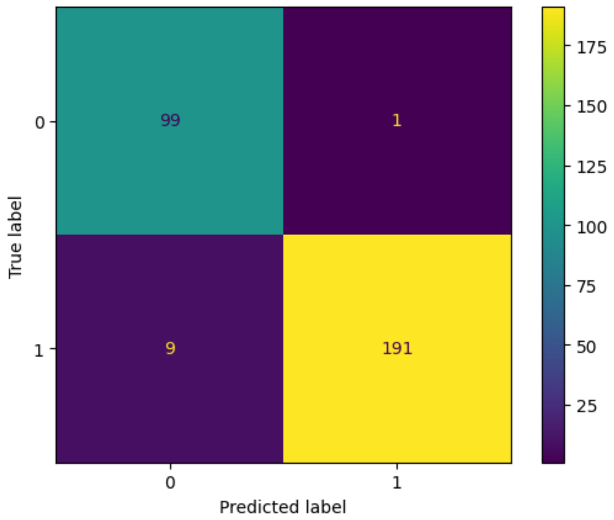
[ ]:
```
test_acc=accuracy_score(y_test_pred, y_test_small)
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.9666666666666667
```

[ ]:
```
# Train confussion matrix
cm = confusion_matrix(y_train, y_train_pred)
cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
# Test confussion matrix
cm = confusion_matrix(y_test_small, y_test_pred)
cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
# Test accuracy per type
detailed_accuracy(y_test_pred, 100)
```

```
SEP accuracy:  0.99
PPT accuracy:  0.93
NPPT accuracy:  0.98
```

```
[ ]: model_type = SVC()
     model_params = [{'kernel': [qkernel]}]
     model = GridSearchCV(model_type, model_params, cv=StratifiedKFold(shuffle=True)).
      ↪fit(xs_train, y_train)
     print('Training results :')
     print(model.best_params_)
     print(model.best_score_)
```

```
Training results :
{'kernel': <function qkernel at 0x7cd0d8e4b6d0>}
0.985
```

Training time 12 minutes

Train Prediction time 11 minutes

Test prediction time 18 min

Five fold cross validation 1 hour

4.3 Quantum Neural Networks

```
[ ]: def save_modelkeras(model,filename):
       dir="/content/drive/MyDrive/tfg/models/"+filename
       model.save(dir)
       print("Keras model saved")
```

```
[ ]: def load_modelkeras(filename):
       # Recreate the exact same model, including its weights and the optimizer
       dir="/content/drive/MyDrive/tfg/models/"+filename
       new_model = tf.keras.models.load_model(dir)
       return new_model
```

4.3.1 TwoLocal variational form

```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)

     nreps=10

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
       return qml.expval(qml.Hermitian(M, wires=[0]))
```

```
[ ]: # Model 1 five fold cross validation
     fivefoldCV_qnn(qnn_circuit,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.4691675038124646; accuracy of 0.9
```

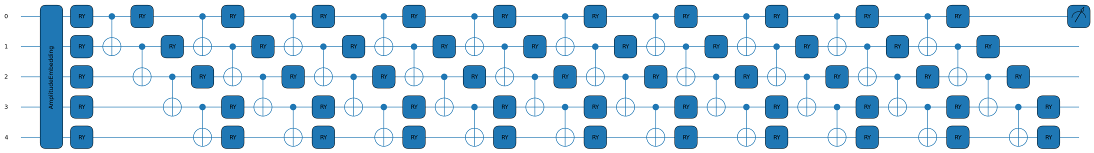
```
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5366220189702486; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5642844875142827; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.53884107944004; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.6026014746873601; accuracy of 0.65
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.4691675038124646 - Accuracy: 0.9
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5366220189702486 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5642844875142827 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 4 - Loss: 0.53884107944004 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6026014746873601 - Accuracy: 0.65
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5423033128848792
> Accuracy: 0.74 (+- 0.08602325267042628)
------------------------------------------------------------------------
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_1 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
```

```
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_1.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
fig.show()
```



[ ]:
```
history = model_1.fit(xs_train, y_train, epochs = 50, shuffle = True,
                      validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 15s 1s/step - loss: 0.9957
Epoch 2/50
10/10 [==============================] - 17s 1s/step - loss: 0.9520
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4449
Epoch 50/50
10/10 [==============================] - 19s 2s/step - loss: 0.4435
```

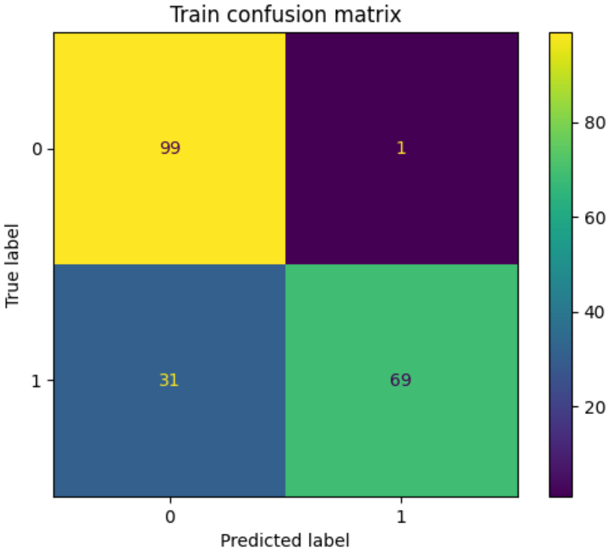
[ ]:
```
y_train_pred=model_1.predict(xs_train) >= 0.5
y_test_pred=model_1.predict(xs_test) >= 0.5
```

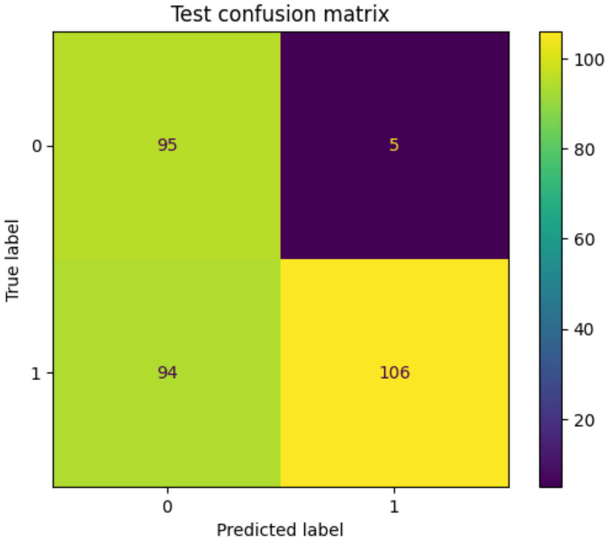
```
7/7 [==============================] - 9s 1s/step
10/10 [==============================] - 15s 2s/step
```

[ ]:
```
performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.84
Train F-1 score:  0.8117647058823529

Test accuracy:  0.67
Test F-1 score:  0.6816720257234727

Test accuracy broken down per type
SEP accuracy:  0.95
```

```
PPT accuracy:  0.52
NPPT accuracy:  0.54
```





Training full dataset 14 min

Normalized data

```
[ ]: # Model 2 five fold cross validation (normalized data)
     fivefoldCV_qnn(qnn_circuit,xs_train_norm, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.4813411885921739; accuracy of 0.875
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.4781813496467676; accuracy of 0.85
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5783924202387007; accuracy of 0.725
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5303979587338639; accuracy of 0.8
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5760894933973543; accuracy of 0.675
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.4813411885921739 - Accuracy: 0.875
------------------------------------------------------------------------
> Fold 2 - Loss: 0.4781813496467676 - Accuracy: 0.85
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5783924202387007 - Accuracy: 0.725
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5303979587338639 - Accuracy: 0.8
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5760894933973543 - Accuracy: 0.675
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5288804821217721
> Accuracy: 0.7849999999999999 (+- 0.07516648189186453)
------------------------------------------------------------------------
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}
```

```
# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_2 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_2.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: history = model_2.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                          validation_data = None, batch_size = 20)
```

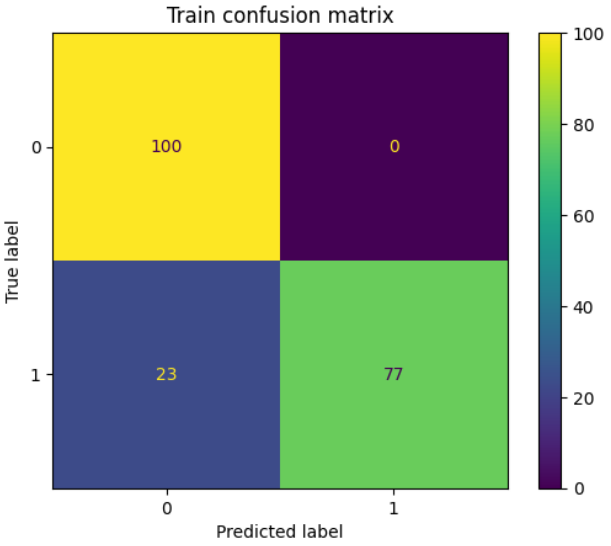
```
Epoch 1/50
10/10 [==============================] - 17s 1s/step - loss: 0.7584
Epoch 2/50
10/10 [==============================] - 18s 2s/step - loss: 0.7429
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4507
Epoch 50/50
10/10 [==============================] - 18s 2s/step - loss: 0.4501
```

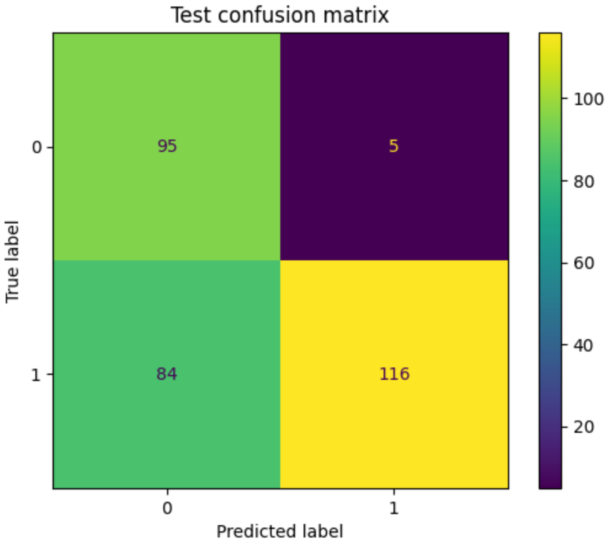
```
[ ]: y_train_pred=model_2.predict(xs_train_norm) >= 0.5
     y_test_pred=model_2.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 17s 3s/step
10/10 [==============================] - 13s 1s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.885
Train F-1 score:  0.8700564971751412

Test accuracy:  0.7033333333333334
Test F-1 score:  0.7227414330218069

Test accuracy broken down per type
SEP accuracy:  0.95
PPT accuracy:  0.57
NPPT accuracy:  0.59
```





```
[ ]: save_modelkeras(model_3,'model_twolocal10reps_norm.h5')
```

```
Keras model saved
```

15 min training whole dataset

20 repetitions

```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)
```

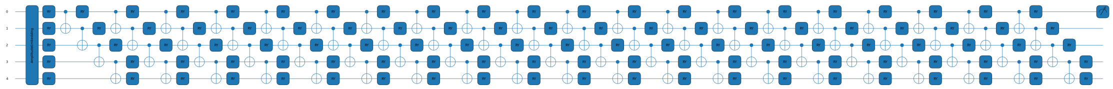
```
nreps=20

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_3 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_3.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:

```
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
fig.show()
```



[ ]:

```
# Training our model
history = model_3.fit(xs_train, y_train, epochs = 50, shuffle = True,
                      validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 28s 3s/step - loss: 0.9938
Epoch 2/50
```

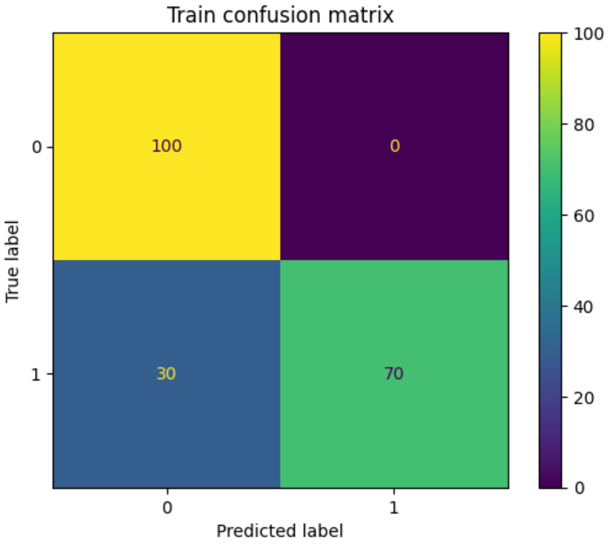
```
10/10 [==============================] - 32s 3s/step - loss: 0.9200
.
.
.
Epoch 49/50
10/10 [==============================] - 29s 3s/step - loss: 0.4082
Epoch 50/50
10/10 [==============================] - 27s 3s/step - loss: 0.4066
```

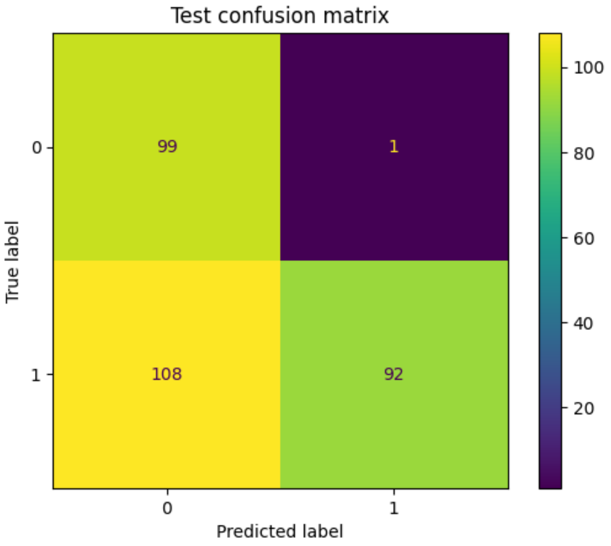
[ ]:
```
y_train_pred=model_3.predict(xs_train) >= 0.5
y_test_pred=model_3.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 19s 2s/step
```

[ ]:
```
performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.85
Train F-1 score:  0.8235294117647058

Test accuracy:  0.6366666666666667
Test F-1 score:  0.6279863481228669

Test accuracy broken down per type
SEP accuracy:  0.99
PPT accuracy:  0.44
NPPT accuracy:  0.48
```





Training time 24 minutes

20 repetitions, normalized data

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_3n = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_3n.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_3n.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                            validation_data = None, batch_size = 20)
```

```
Epoch 1/50
```

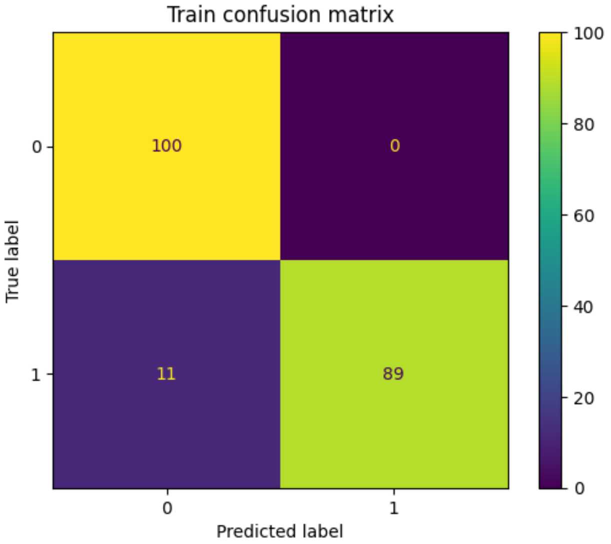
```
10/10 [==============================] - 32s 3s/step - loss: 0.7562
Epoch 2/50
10/10 [==============================] - 26s 3s/step - loss: 0.7098
.
.
.
Epoch 49/50
10/10 [==============================] - 31s 3s/step - loss: 0.3772
Epoch 50/50
10/10 [==============================] - 26s 3s/step - loss: 0.3763
```

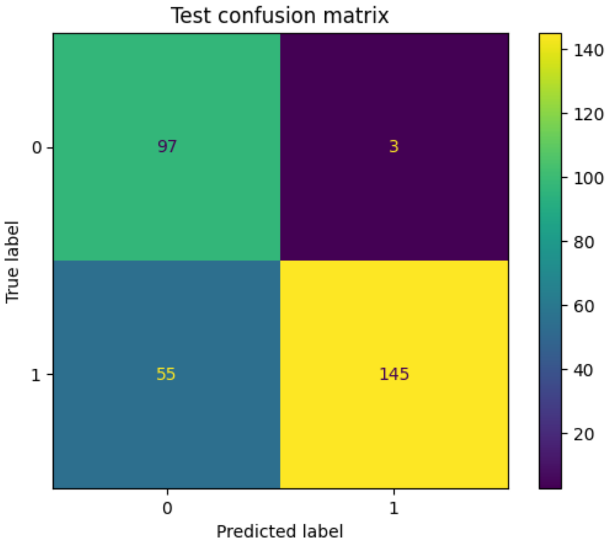
```
[ ]: y_train_pred=model_3n.predict(xs_train_norm) >= 0.5
     y_test_pred=model_3n.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 19s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.945
Train F-1 score:  0.9417989417989417

Test accuracy:  0.8066666666666666
Test F-1 score:  0.8333333333333334

Test accuracy broken down per type
SEP accuracy:  0.97
PPT accuracy:  0.76
NPPT accuracy:  0.69
```





```
[ ]: save_modelkeras(model_3n,'model_twolocal20reps_norm.h5')
```

```
Keras model saved
```

Training time 24 minutes

4.3.2 StrongEntanglingLayers

```
[ ]: nqubits=5
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 8
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

```
[ ]: # Model 4 five fold cross validation
     fivefoldCV_qnn(qnn_circuit_strong,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.5199374013200586; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5743251901298562; accuracy of 0.675
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.6113768385617595; accuracy of 0.65
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.49767407140878933; accuracy of 0.725
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.6144068848443738; accuracy of 0.675
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5199374013200586 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5743251901298562 - Accuracy: 0.675
------------------------------------------------------------------------
> Fold 3 - Loss: 0.6113768385617595 - Accuracy: 0.65
------------------------------------------------------------------------
> Fold 4 - Loss: 0.49767407140878933 - Accuracy: 0.725
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6144068848443738 - Accuracy: 0.675
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5635440772529675
> Accuracy: 0.7 (+- 0.044721359549995579)
------------------------------------------------------------------------
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

     # keras model
```

```
model_4 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_4.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_4.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

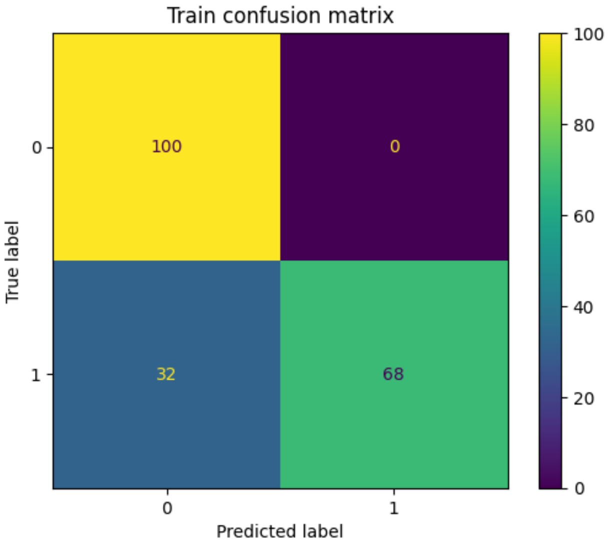
```
Epoch 1/50
10/10 [==============================] - 44s 5s/step - loss: 0.7748
Epoch 2/50
10/10 [==============================] - 60s 6s/step - loss: 0.7602
.
.
.
Epoch 49/50
10/10 [==============================] - 29s 3s/step - loss: 0.4824
Epoch 50/50
10/10 [==============================] - 33s 3s/step - loss: 0.4819
```

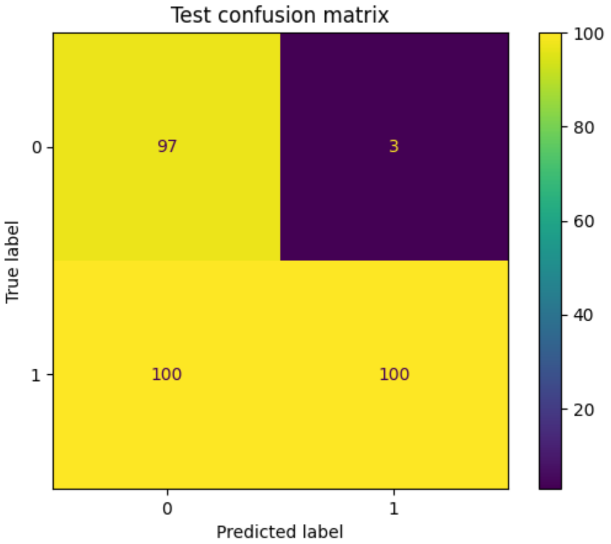
```
[ ]: y_train_pred=model_4.predict(xs_train) >= 0.5
     y_test_pred=model_4.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 15s 2s/step
10/10 [==============================] - 20s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.84
Train F-1 score:  0.8095238095238095

Test accuracy:  0.6566666666666666
Test F-1 score:  0.6600660066006601

Test accuracy broken down per type
SEP accuracy:  0.97
PPT accuracy:  0.5
NPPT accuracy:  0.5
```





five fold cross validation 2h

28 min training full training set

Normalized data

```
[ ]: # Model 4n five fold cross validation (normalized data)
fivefoldCV_qnn(qnn_circuit_strong,xs_train_norm, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...
```

```
Score for fold 1: loss of 0.5191338239214546; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.575823766732432; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5508142099930897; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5358043361919507; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5572398545035854; accuracy of 0.8
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5191338239214546 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 2 - Loss: 0.575823766732432 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5508142099930897 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5358043361919507 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5572398545035854 - Accuracy: 0.8
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5477631982685025
> Accuracy: 0.7550000000000001 (+- 0.033166247903554026)
------------------------------------------------------------------------
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model_4n = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)
```

```
# binary cross entropy loss, because we are training a binary classifier
model_4n.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: history = model_4n.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                          validation_data = None, batch_size = 20)
```

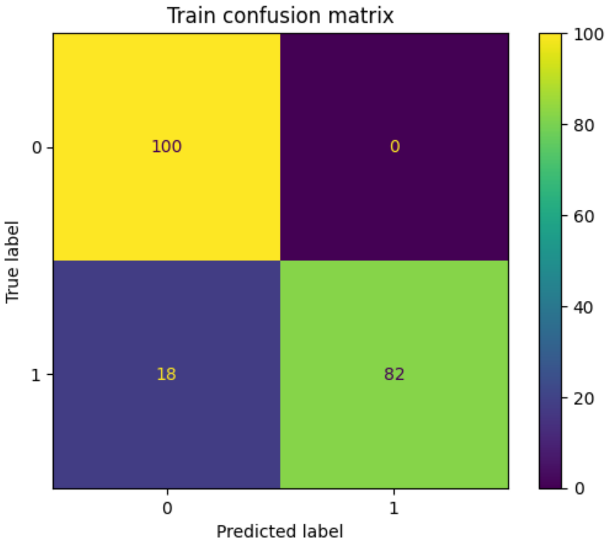
```
Epoch 1/50
10/10 [==============================] - 31s 3s/step - loss: 0.7059
Epoch 2/50
10/10 [==============================] - 34s 4s/step - loss: 0.6890
.
.
.
Epoch 49/50
10/10 [==============================] - 33s 3s/step - loss: 0.4639
Epoch 50/50
10/10 [==============================] - 30s 3s/step - loss: 0.4631
```

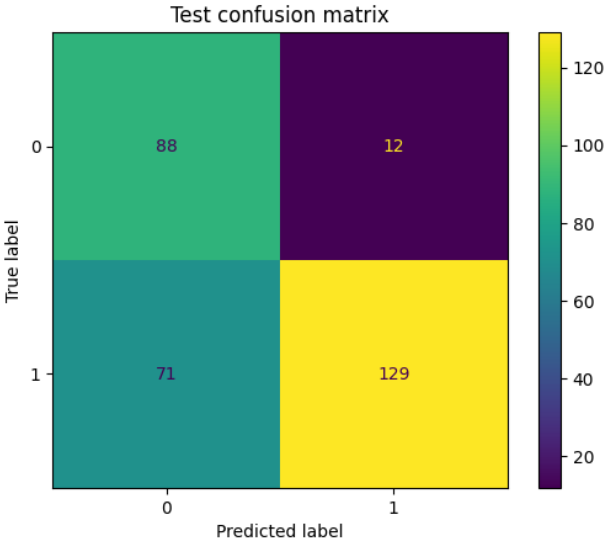
```
[ ]: y_train_pred=model_4n.predict(xs_train_norm) >= 0.5
     y_test_pred=model_4n.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 13s 2s/step
10/10 [==============================] - 21s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.91
Train F-1 score:  0.9010989010989011

Test accuracy:  0.7233333333333334
Test F-1 score:  0.7565982404692083

Test accuracy broken down per type
SEP accuracy:  0.88
PPT accuracy:  0.69
NPPT accuracy:  0.6
```





[ ]:
```
save_modelkeras(model_4n,'model_strongentangling8reps_norm.h5')
```

```
Keras model saved
```

Five fold cross validation 2h

Training time full training set 24 min