Secrets revealed in this session:

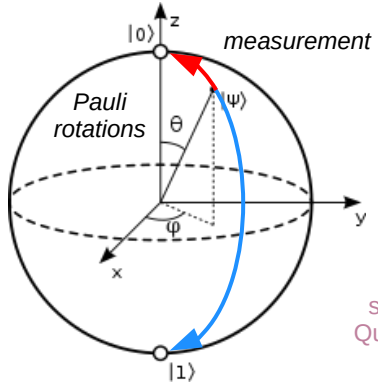
To look behind the doors of quantum classification!

QML classification models
Classical vs quantum classification
Class imbalance in quantum
PennyLane meets PyTorch
Wrapping quantum models in PyTorch layers
PennyLane / PyTorch seamless integration
Grid search
Quantum vs classical results
Adjusting the threshold
Barriers to quantum classification
PenyLane demo

Quantum Machine Learning

Quantum classification with a gentle introduction to PennyLane + PyTorch integration

Jacob L. Cybulski
Enquanted, Melbourne, Australia



We will assume
some knowledge of
Quantum Computing
ML and Python

Classical classification / Quantum classification

Class of a data sample is a group defined by a unique nominal value of one of its attributes. For instance vehicles can be grouped by:

- ***Colour***, such as “red”, “yellow” or “green”
- ***Size***, e.g. “small”, “medium” or “large”

Classification is the process of classifying data samples based on their attribute values, i.e. deciding what class value should be given to their label attribute, with a view to determine the membership of a sample in a particular group:

- ***Vehicles type***, such as “sedan” or “truck”, which can be predicted from other attributes, e.g. size and colour

Classifier is a model predicting the class of a sample, and capable of automating classification of data recorded in the future.

Classical machine learning offers numerous models and algorithms for highly efficient classification. Their quantum counterparts are still in research phase.

Quantum Classification Models vs. Classical Counterparts

Classical Model	Quantum Version	Status	Key Challenges
Logistic Regression	Quantum Variational Classifier	Implemented (Qiskit, PennyLane)	Limited qubit scalability
SVM	Quantum Kernel SVM	Working prototypes (QSVM)	Kernel computation on quantum hardware
Decision Trees	Quantum Decision Trees	Theoretical (noisy, hard to train)	Noisy intermediate-scale quantum (NISQ) limitations
Random Forest	Not directly applicable	N/A (ensemble methods not portable)	Quantum parallelism ≠ classical boosting
Neural Networks (NN)	Quantum Neural Networks (QNN)	Early-stage (e.g., QCNNs)	Barren plateaus, training difficulties
k-NN	Quantum k-NN (distance-based)	Proof-of-concept (small datasets)	Requires QRAM (not yet practical)
Naïve Bayes	Quantum Bayesian Networks	Theoretical	Probabilistic circuits are complex
Transformers	Quantum Attention Mechanisms	Speculative research	Noisy hardware, coherence time limits

Examples where quantum ML outperformed classical ML:
(1) high dimensional feature spaces, (2) dimensionality reduction, (3) sampling from complex distributions, (4) simulation of chemical properties and reactions, (5) quantum Monte Carlo methods, etc.

Data preparation for classification

Class imbalance and preparation of predictors

Due to limited qubit resources, quantum classifiers are sensitive to class imbalance!

Often we have a *minority class* (very small number) of positive examples (important to us).

In cases of the class imbalance, we cannot trust accuracy as it can be high even though the most, or all, positive examples are misclassified. What can we do?

We cannot rely on the accuracy alone, we must also analyse our failures, e.g. check

- *Sensitivity:* the proportion of correctly identified true positives
- *Specificity:* the proportion of correctly identified true negatives

We can also use *Cohen kappa statistic*, which adjusts accuracy based on the distribution of class values.

- Kappa > 0.6 is considered good!
- High accuracy but low kappa is poor!

In cases of class imbalance, some classifiers learn to produce *bias towards the majority class*.

The solution may involve rebalancing data sample by either:

- *over-sampling the minority class or*
- *under-sampling the majority class.*

Balancing of training data may lead to a better model.

However, balancing test data may lead to incorrect accuracy.

It is best to balance training data and to use the unbalanced data for validation and testing.

All previously mentioned concerns about analysis and preparation of predictors are still applicable to quantum classification.

SMOTE or *Synthetic Minority Oversampling Technique* is one of the most commonly used technique for class oversampling.

SMOTE creates synthetic (not real) data points in the smallest label class. Then you perform SMOTE balancing iteratively for all classes.

However, in some circumstances, certain types of models can deal with unbalanced samples.

In which case, by balancing data the model performance may drop!

Always test if sample balancing actually improves the model performance.

Weighing data examples (by inverse of class frequency) is an alternative approach in classical ML. In training with weighing, e.g. weights are used to penalise majority class examples.

PyTorch-Ignite, “CohenKappa”, https://pytorch.org/ignite/generated/ignite.metrics.CohenKappa.html

Imbalanced-learn, “SMOTE”, https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html

Priors and classification

Another way is to balance all data, for training and validation, but then *recalculate validation results*.

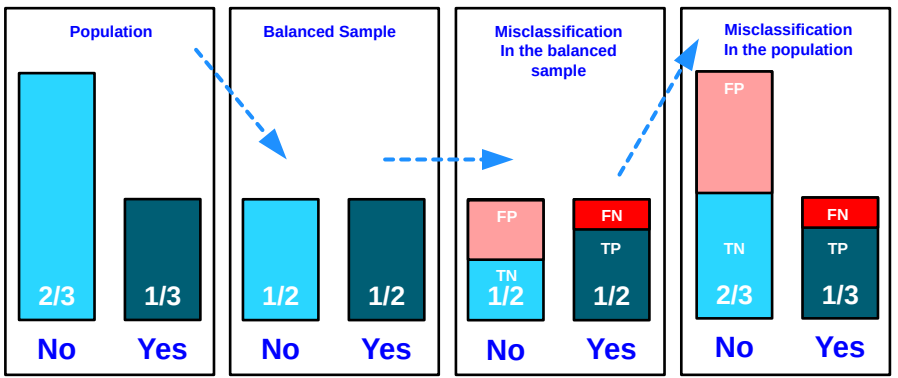
The class probability distribution in the population is called *prior probability* (or *priors*).

Let's say we have 900 examples, split 2:1 between negative vs positive cases, we are interested in positives.

If we trained a model on this data, it will favour the negative cases, it will over-train on them.

We can resample data (e.g. under-sample the negatives).

Now we have balanced data, better for model training.



We then validate the model and let us say we found 50% of negatives and 25% of positives to be misclassified.

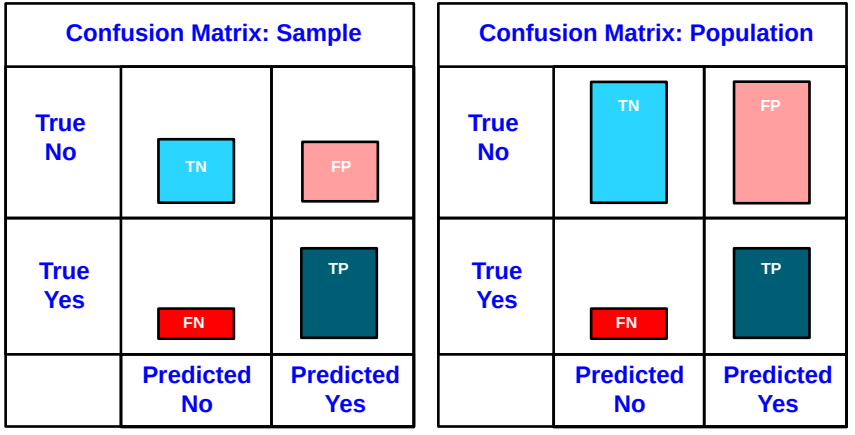
Our misclassification rate is: 0.25+0.125=0.375

However, if we deploy the model to work with the population data we can expect a very different result – we need to scale this result to reflect the proportions In the population, i.e. 2:1 (not 1:1).

The new misclassification rate is 1/3+0.25x1/3=0.417

Detection rate of positive cases TP/FN is the same.

The cost of handling negative cases went up.



Some data mining software can perform these calculations automatically,

PennyLane / PyTorch

Neural network structure

```python
### Classic classifier
class Classic_Auto(nn.Module):

    def __init__(self, in_shape, out_shape):
        super(Classic_Auto, self).__init__()

        layers = self.layers(in_shape, out_shape)
        self.model = nn.Sequential(*layers)

    def layers(self, in_shape, out_shape):

        clayer_0 = torch.nn.Linear(in_shape, 32)
        clayer_1 = torch.nn.ReLU()
        clayer_2 = torch.nn.Linear(32, 64)
        clayer_3 = torch.nn.ReLU()
        clayer_4 = torch.nn.Linear(64, 32)
        clayer_5 = torch.nn.ReLU()
        clayer_6 = torch.nn.Linear(32, 8)
        clayer_7 = torch.nn.ReLU()
        clayer_8 = torch.nn.Linear(8, out_shape)

        layers = [clayer_0, clayer_1, clayer_2, clayer_3, clayer_4,
                  clayer_5, clayer_6, clayer_7, clayer_8]
        return layers

    def forward(self, x):
        x = self.model(x)
        return x
```

Pure PyTorch

Both models are designed as *PyTorch neural net classes*.

Both initialise their model instances by saving the most important parameters (self), and then creating and saving their models (self.model) as *sequences of layers*.

Both define and create neural network *layers* in PyTorch nn.

The quantum model's QNode with its shape details was wrapped in a *TorchLayer*. The QNode was created with a *"torch" interface* and a device specified via parameters.

Both models identify the *forward* function, which provides a method of calculating outputs from inputs.

```python
### Create a PyTorch model with a PennyLane circuit within

class Quantum_Auto(nn.Module):

    def __init__(self, sim, n_wires, n_layers=1, shots=None):
        super(Quantum_Auto, self).__init__()

        self.sim = sim
        self.n_wires = n_wires
        self.n_layers = n_layers
        self.shots = shots

        # Wrap a torch layer around the PennyLane model
        layers = [self.layers()]
        self.model_pt = nn.Sequential(*layers)

    ### Define a quantum layer
    def layers(self):

        # Specify a device
        dev = qml.device(self.sim, wires=self.n_wires, shots=self.shots)

        # Define the quantum model and its circuit (or node, save it for later)
        model_pl = qmodel(self.n_wires)
        self.model_qc = qml.QNode(model_pl, dev, interface='torch')

        # Define the shape of the model weight parameters
        # Note that the name "weights" must match the param name defined in function
        # "model_pl" which in our case is _qmodel(inputs, weights)
        weights_shapes = {"weights": qshape(self.n_wires, n_layers=self.n_layers)}

        # Turn the circuit into a Torch-compatible quantum layer
        qlayer = qml.qnn.TorchLayer(self.model_qc, weight_shapes=weights_shapes)
        return qlayer

    ### Return the quantum model circuit
    def qmodel_qc(self):
        return self.model_qc

    ### Apply the model to data (forward step)
    def forward(self, x):
        y = self.model_pt(x)
        return y
```

PyTorch with PennyLane layer

The structure of a PyTorch classic model is almost identical to that of a PyTorch model with a quantum model as it layer!

What is different is the specification of their layers.

PennyLane / PyTorch
Model training

```
### Create a model
q_auto = Quantum_Auto(sim, X_train_tens.shape[1],
    n_layers=n_layers, shots=shots).double().to(torch_device)

### Loss and optimiser
cost_fun = torch.nn.MSELoss()
opt = torch.optim.NAdam(q_auto.parameters(), lr=0.01)

### Train the model
train_mse_hist, train_acc_hist, opt_params, hist_params, opt_point = \
    train_model(q_auto, X_train_tens, y_train_tens, cost_fun, accuracy, opt,
                epochs=50, log_interv=1, prompt_fract=0.1, acc_prec=0.5)
```

Invocation of the PennyLane model training

```
### Trains a pure PyTorch model

def train_model(model, X, y, cost_fun, acc_fun, optimizer, epochs,
                log_interv=100, prompt_fract=0.1, acc_prec=0.5, start_time=0):

    history = []
    acc_hist = []
    opt_params = {}
    hist_params = []
    min_epoch = 0
    min_cost = 1000
    max_acc = -1000
    if start_time == 0: start_time = time.time()

    model.train()
    for epoch in range(epochs):

        optimizer.zero_grad()
        output = model(X)
        cost = cost_fun(output, y)
        acc = acc_fun(output, y, prec=acc_prec)
        cost.backward()
        optimizer.step()

        curr_cost = cost.item()
        curr_acc = acc
        if curr_cost < min_cost:
            min_cost = curr_cost
            min_epoch = epoch
            opt_params = copy.deepcopy(model.state_dict())

        if curr_acc > max_acc:
            max_acc = curr_acc

        if epoch % log_interv == 0:
            history.append(curr_cost)
            acc_hist.append(curr_acc)
            hist_params.append(copy.deepcopy(model.state_dict()))

        elapsed = time.time() - start_time
        if (prompt_fract == 0) or (epoch % int(prompt_fract*epochs) == 0):
            print(f'{epoch: 5d} '+ \
                  f'({elapsed:06.0f} sec): '+ \
                  f'Cost {curr_cost:6.4g}   '+ \
                  f'Acc {curr_acc:6.4g}')

    return history, acc_hist, opt_params, hist_params, (min_epoch, min_cost)
```

Training of Pure PyTorch NN

Some optimisers, e.g. LBFGS, perform additional inner steps and need a "closure" to be passed to their step() function.

Both models are trained in the same way. PyTorch has no knowledge of the two models differences!

The training function receives data (X and y), cost and accuracy calculating functions, the optimiser, the required number of training epochs, and other variables used during the process.

Model training starts with the initialisation of lists to collect training cost and accuracy at each optimisation step.

Then a training loop starts.

Training is conducted in a loop utilising gradients in the model weights.

First, the optimiser is instructed to *reinitialise gradients.*

Second, it performs the *forward step* by applying the model to data (X) to receive the model predictions on output.

Third, the predictions are compared with the expected values (y) and the *cost is calculated*.

Fourth, a *backward step* is taken to recalculate all model weights.

Fifth, the optimiser then prepares for the next optimisation *step*.

Finally, we *collect performance indicators* and print the partial results.

```
### Trains a PennyLane+PyTorch model

def train_model(model, X, y, cost_fun, acc_fun, optimizer, epochs,
                log_interv=100, prompt_fract=0.1, acc_prec=0.5, start_time=0):

    history = []
    acc_hist = []
    opt_params = {}
    hist_params = []
    min_epoch = 0
    min_cost = 1000
    max_acc = -1000
    if start_time == 0: start_time = time.time()

    model.train()
    for epoch in range(epochs):

        optimizer.zero_grad()
        output = model(X)
        cost = cost_fun(output, y)
        acc = acc_fun(output, y, prec=acc_prec)
        cost.backward()
        optimizer.step()

        curr_cost = cost.item()
        curr_acc = acc
        if curr_cost < min_cost:
            min_cost = curr_cost
            min_epoch = epoch
            opt_params = copy.deepcopy(model.state_dict())

        if curr_acc > max_acc:
            max_acc = curr_acc

        if epoch % log_interv == 0:
            history.append(curr_cost)
            acc_hist.append(curr_acc)
            hist_params.append(copy.deepcopy(model.state_dict()))

        elapsed = time.time() - start_time
        if (prompt_fract == 0) or (epoch % int(prompt_fract*epochs) == 0):
            print(f'{epoch: 5d} '+ \
                  f'({elapsed:06.0f} sec): '+ \
                  f'Cost {curr_cost:6.4g}   '+ \
                  f'Acc {curr_acc:6.4g}')

    return history, acc_hist, opt_params, hist_params, (min_epoch, min_cost)
```

Training of PyTorch NN with a PennyLane layer

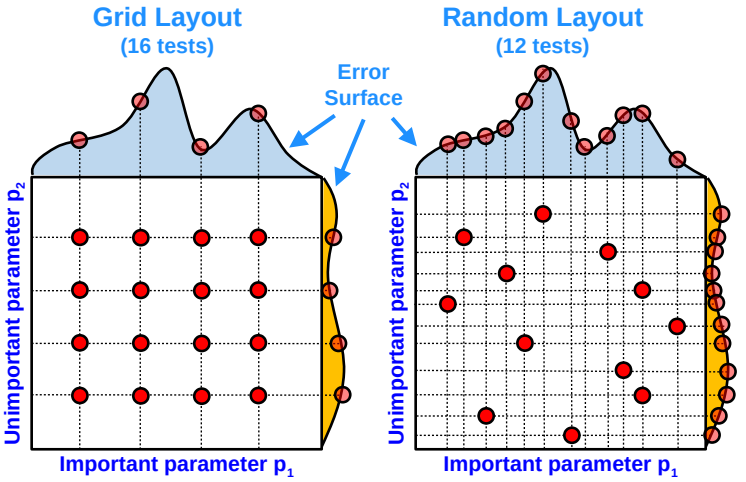
In search of the optimum
Grid search

- The performance of a quantum classifier depends on the combination of its circuit characteristics, e.g.
  - the *number of qubits* and the *number of layers*
- as well as the hyper-parameters of the optimiser and its training process, e.g.
  - the *learning rate* and the *number of epochs*
  - the *optimiser* in use, e.g. SGD, RMSprop, Adam
- We *tune the model* by experimenting with all of these training process hyper-parameters.
- Trial and error is a possible approach!
  However, a systematic approach is always preferred!
- For a single parameter, a feasible approach is to construct a *loop over a list of hyper-parameter values* and then log, chart and review the performance indicators.
- Scikit-learn and PyTorch (via skorch) provide support (via its operators) for the systematic exploration of multiple model parameters in a *grid search*.

Scikit Learn, “Tuning the hyper-parameters of an estimator”,
https://scikit-learn.org/stable/modules/grid_search.html

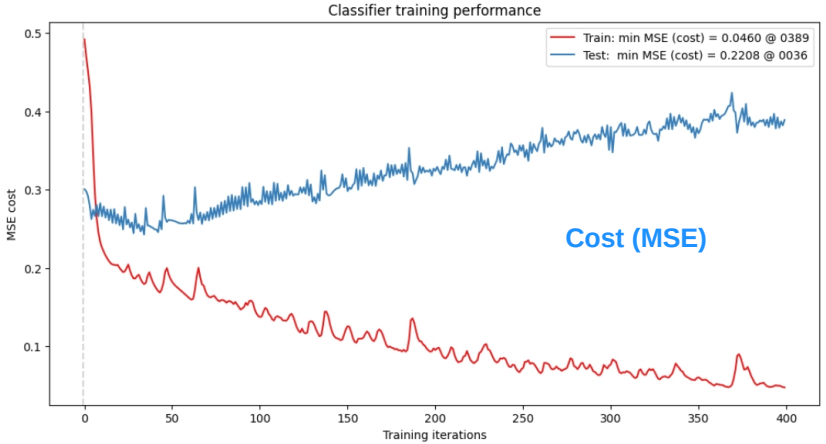
Adrian Tam, “How to Grid Search Hyperparameters for PyTorch Models”,
https://machinelearningmastery.com/how-to-grid-search-hyperparameters-for-pytorch-models/

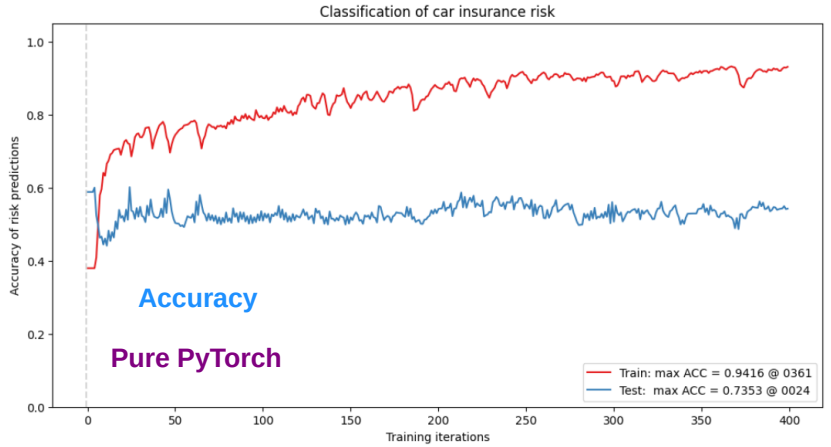
- There are two possible ways of exploring multiple parameter values, i.e. with:
  - *Grid search* of parameter values, where for each parameter we supply a list of its possible values and we test the model on all their combinations – *most commonly used and easy to implement*;
  - *Random grid search*, where test points are generated randomly, each having a combination of (most likely) unique parameter values – *fewer tests and yet better coverage of parameter values*.



The collection of parameters may include the more important parameters, which may be better at identifying distinguishing features of the error surface than unimportant parameters.

PennyLane / PyTorch
Results





Model scoring and plotting of results is also identical!

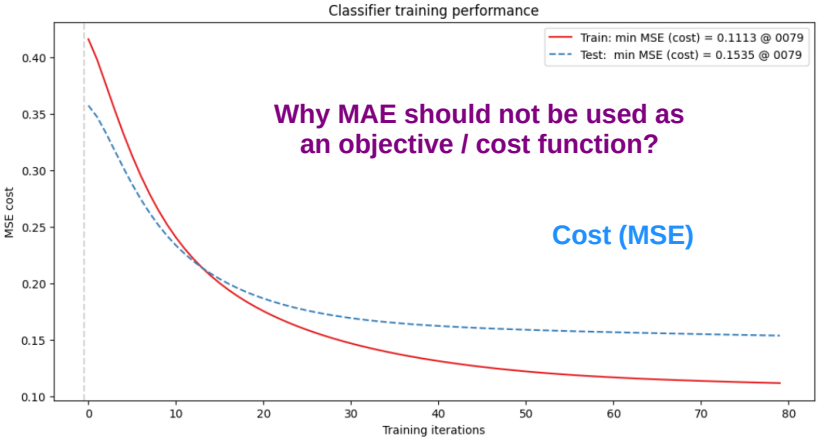
As we are training with small data sets, the classical "plain vanilla" PyTorch models tend to quickly overtrain.

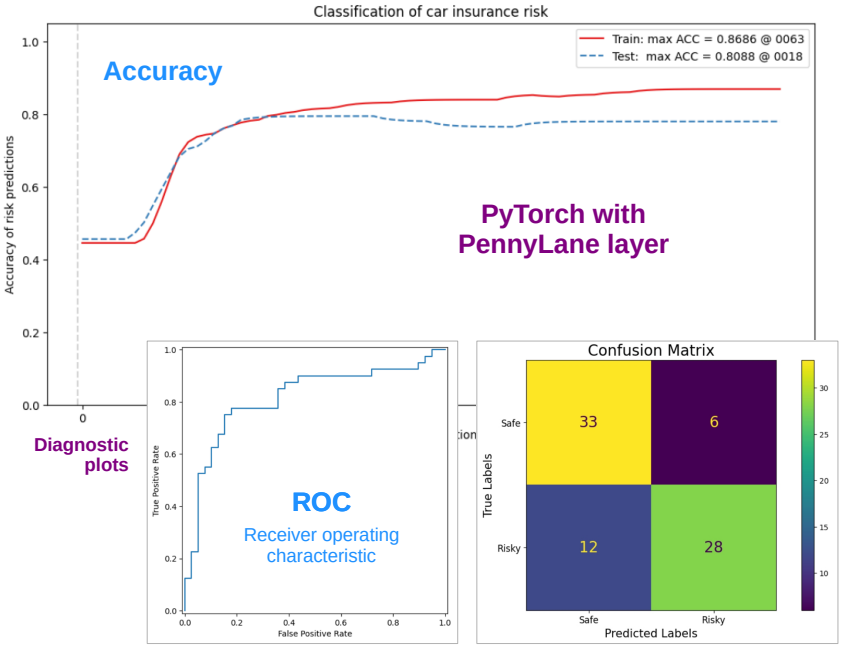
There is a lot more tuning required of the quantum model but its plots are also more beautiful!

Performance plots from training classical models look as if they were produced by a very nervous but very successful gambler!

Note that IMHO if a quantum model on simple data exceeds the performance of a classical model, it usually means that you have not tuned it properly!

Sadly, this is the case here!

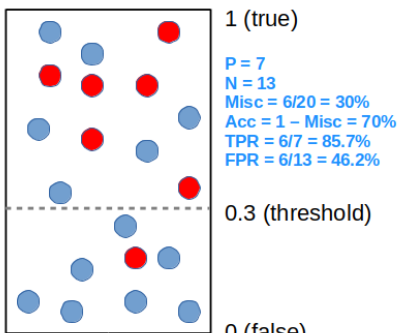


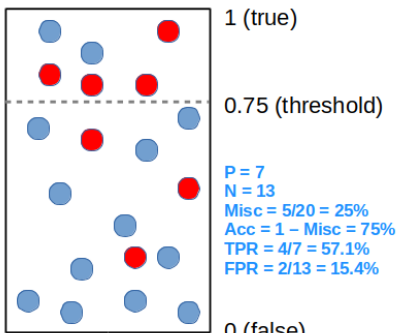


Binomial classification
Adjusting the threshold

- Assume we have two classes - positive and negative.
- When a positive class is correctly classified, it is called *true-positive* (TP), the negative class is called *true-negative* (TN).

Consider these 20 data points, classified with different confidence factors from 0 to 1. Depending on the threshold they will be classified differently and the classifier performance will also be different.

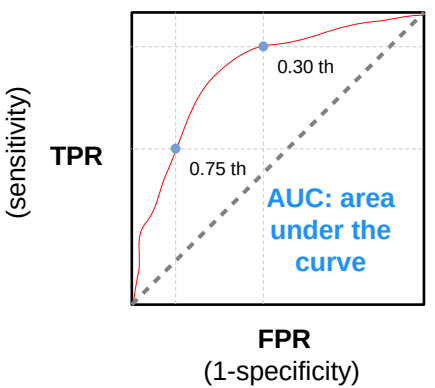




TPR = TP / P
FPR = FP / N

- When a positive class is incorrectly classified it is called *false-negative* (FN), the incorrectly classified negative class is called *false-positive* (FP).
- The prediction is defined by confidence factors, i.e. when the confidence of a positive class is greater than a certain threshold, (e.g. 0.5), it is concluded that classification is positive, else it is negative.
- The threshold can be changed to maximise some performance indicator (e.g. the car is risky).

We can visualise the classifier performance by plotting an ROC (Receiver Operating Characteristic) chart of all possible pairs of FP rate vs TP rate when varying the threshold (see below). It is often considered that the “best” classifier has the largest area under the curve (AUC).



However, AUC should not be used as a sole criterion for classifier selection. From a decision making viewpoint, we are not interested in a model which is good for all possible thresholds. Instead, we are interested in a model with a single specific threshold that supports the most effective decision making!

Scikit Learn, “Tuning the decision threshold for class prediction”, https://scikit-learn.org/stable/modules/classification_threshold.html

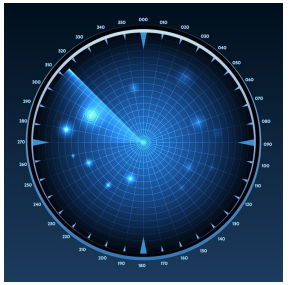
Barriers to quantum classification

Some Problems	Possible Solutions
Poorly chosen embeddings can lead to *loss of information* or *biases*.	• *Avoid simple embeddings* for complex data (e.g., basic angle encoding), as they may not capture nonlinear relationships. • Consider *trainable embeddings* (e.g., quantum neural networks).
Quantum models are sensitive to *imbalanced data*, due to limited qubit resources.	• *Oversampling* or *hybrid classical-quantum approaches* may be needed.
Since classifiers rely on discrete decision boundaries, barren plateaus can lead to *random guessing behaviour*.	• *Regularisation* may help (via classical post-processing).
Quantum classifiers can *overfit* due to limited training data or excessive circuit expressivity.	• Use *dimensionality reduction* (e.g., PCA) before encoding or employ classical regularisation (e.g., dropout in hybrid models).
Quantum classifiers might appear to work well on training data but fail on test data due to *quantum-specific artefacts*,	• Always *compare against classical models* and their performance (e.g., SVM, neural networks). Quantum advantage is rare!

PennyLane Demo

Engineer quantum solutions!

Dataset 2: Sonar



Dataset 1: Automobile risk assessment

	price	highway-mpg	city-mpg	peak-rpm	horsepower	compression-ratio	stroke	bore	fuel-system	engine-size	...	length	wheel-base	engine-location	drive-wheels	body-style	num-of-doors	aspiration	fuel-type	make	normalized-losses
0	13495.0	27	21	5000.0	111.0	9.0	2.68	3.47	mpfi	130	...	168.8	88.6	front	rwd	convertible	2.0	std	gas	alfa-romero	NaN
1	16500.0	27	21	5000.0	111.0	9.0	2.68	3.47	mpfi	130	...	168.8	88.6	front	rwd	convertible	2.0	std	gas	alfa-romero	NaN
2	16500.0	26	19	5000.0	154.0	9.0	3.47	2.68	mpfi	152	...	171.2	94.5	front	rwd	hatchback	2.0	std	gas	alfa-romero	NaN
3	13950.0	30	24	5500.0	102.0	10.0	3.40	3.19	mpfi	109	...	176.6	99.8	front	fwd	sedan	4.0	std	gas	audi	164.0
4	17450.0	22	18	5500.0	115.0	8.0	3.40	3.19	mpfi	136	...	176.6	99.4	front	4wd	sedan	4.0	std	gas	audi	164.0

5 rows × 25 columns

Image source: Unsplash and Freepik

PennyLane Demo:

- Explore insurance risk data
- Consider class order and the need for shuffling
- Reduce your data dimensionality (test and compare)
- Play with hyper-parameters to improve performance
- Apply the best model to new data

Key takeaways:

- Quantum modelling is an engineering task
- There is more to success than a clever model
- Data encoding is (again) crucial to performance
- Dimensionality reduction is crucial to performance
- Design your model tests – use grid search!
- Experiment with the ansatz parameters
- Learn from classical ML how to measure accuracy
- Think about class imbalance
- Once model is trained, you can still improve accuracy!

Thank you!

Any questions?

Photos from Unsplash

Enquanted is being somewhere in-between Enchanted and Entangled