IBM Quantum Challenge Fall 2022

Challenge 2 - Introduction to Primitives on Qiskit Runtime

Qiskit

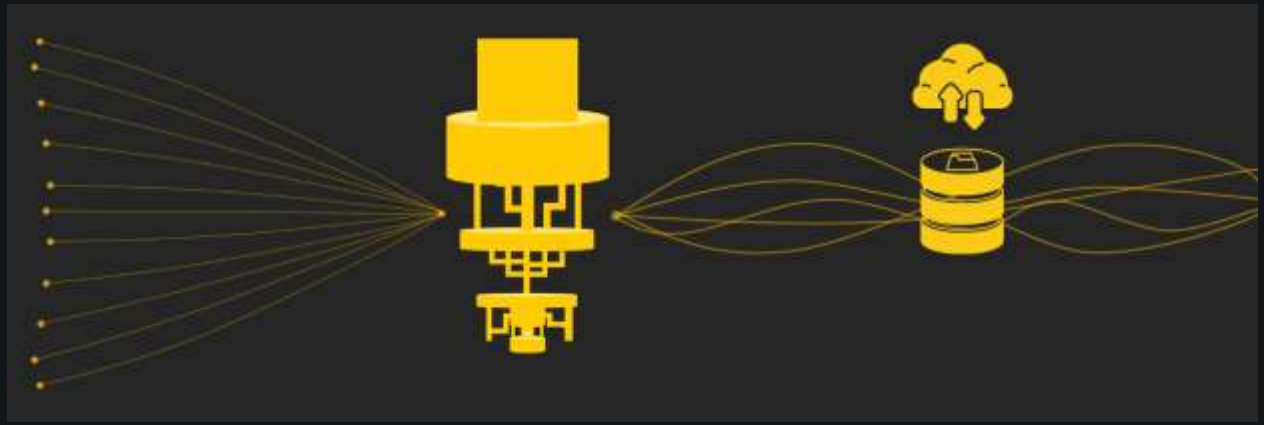
Table of Contents

Story

Quantum Machine Learning

Qiskit

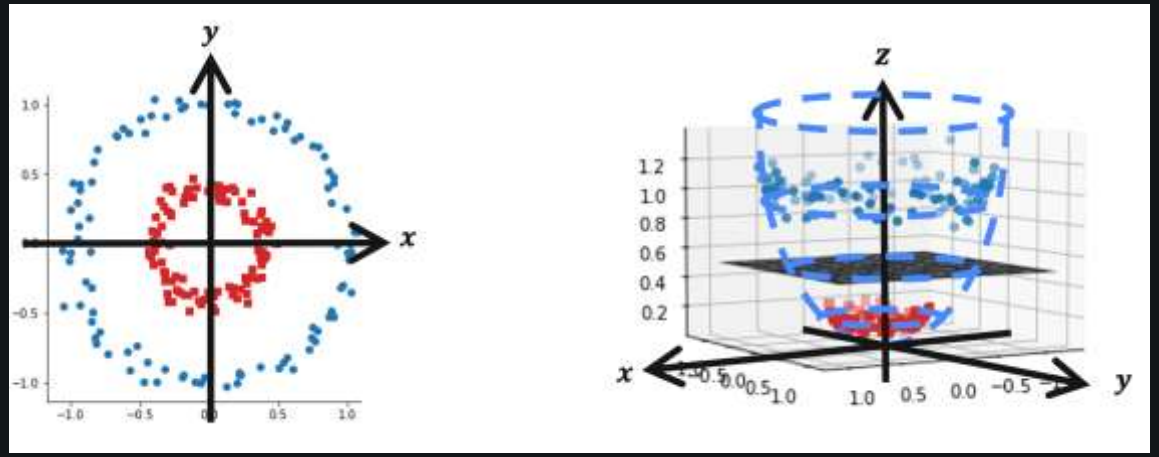
- Use the principles of quantum mechanics to enhance machine learning
- The purpose of quantum machine learning is to
  - lessen either in terms of sample complexity or amount of operations required to train a model, categorize a test vector, or create a novel example of a concept



Handwritten bit strings

Data Encoding

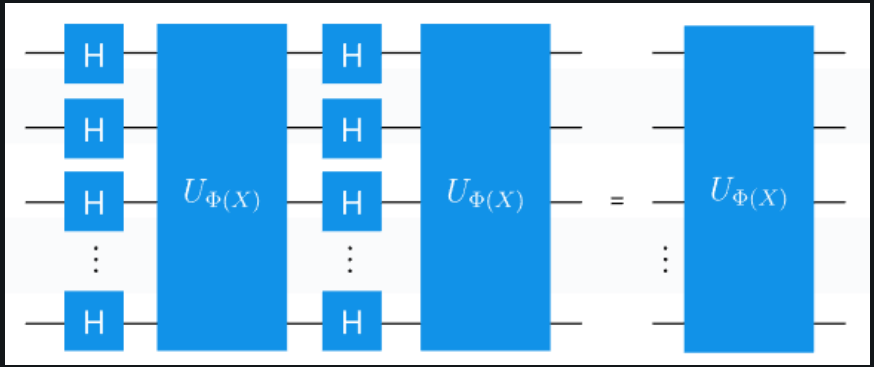
- Map input dataset to a higher dimensional feature space, through the use of a kernel function
  - To find and study pattern in data
- Transform the not linearly separable data from 2-dimensions to 3- dimensions, $z = x^2 + y^2$



Data Encoding

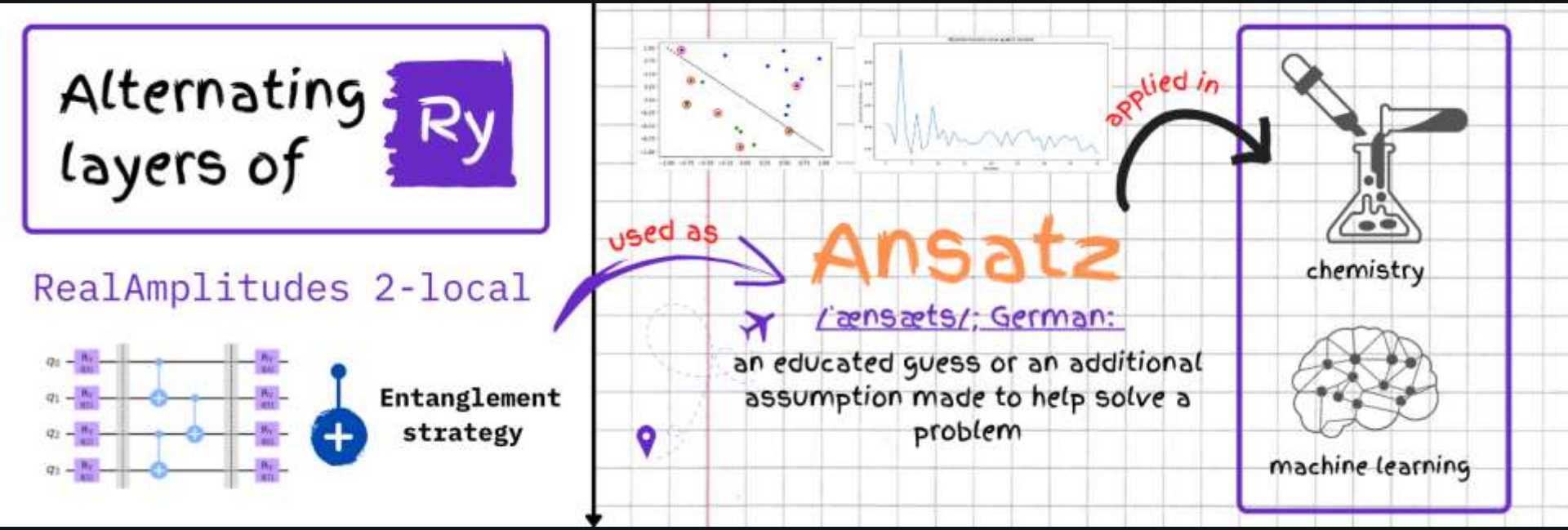
Qiskit

- In QML, unitary transformation transform quantum feature map
  - Parameterized quantum circuit – Encode data as quantum model
  - Parameters determined by data being encoded and optimization processes.
  - Generate key subset & used as machine learning model
- Construct quantum feature map that is hardly simulate classicaly but implementable in noisy quantum device



Exercise 1

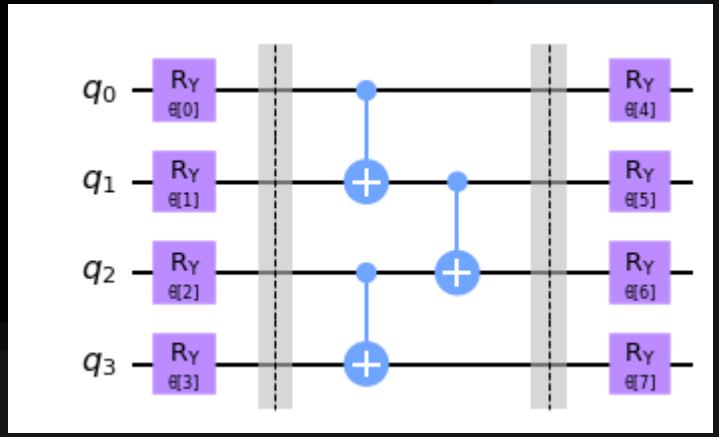
Qiskit



Exercise 1

Qiskit

```python
from qiskit.circuit.library import RealAmplitudes

def lab2_ex1():
    #
    #
    # build your code here
    #
    #
    return qc

qc = lab2_ex1()
qc.decompose().draw('mpl')
```



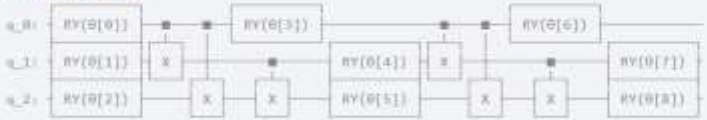
Exercise 1

Qiskit

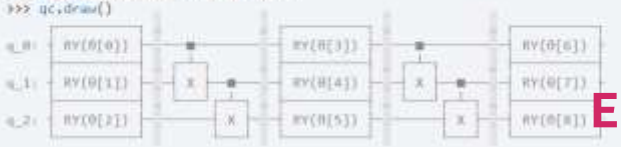
Examples

```
>>> ansatz = RealAmplitudes(3, reps=2)  # create the circuit on 3 qubits
>>> print(ansatz)
```


```
>>> ansatz = RealAmplitudes(3, entanglement='full', reps=2)  # it is the same unitary as above
>>> print(ansatz)
```

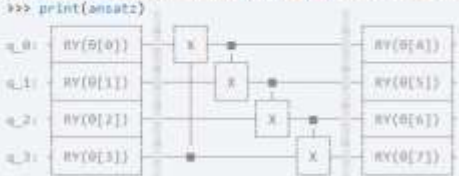



```
>>> ansatz = RealAmplitudes(3, entanglement='linear', reps=2, insert_barriers=True)
>>> qc = QuantumCircuit(3)  # create a circuit and append the RY variational form
>>> qc.compose(ansatz, inplace=True)
>>> qc.draw()
```



Entanglement strategy!!

```
>>> ansatz = RealAmplitudes(4, reps=1, entanglement='circular', insert_barriers=True)
>>> print(ansatz)
```



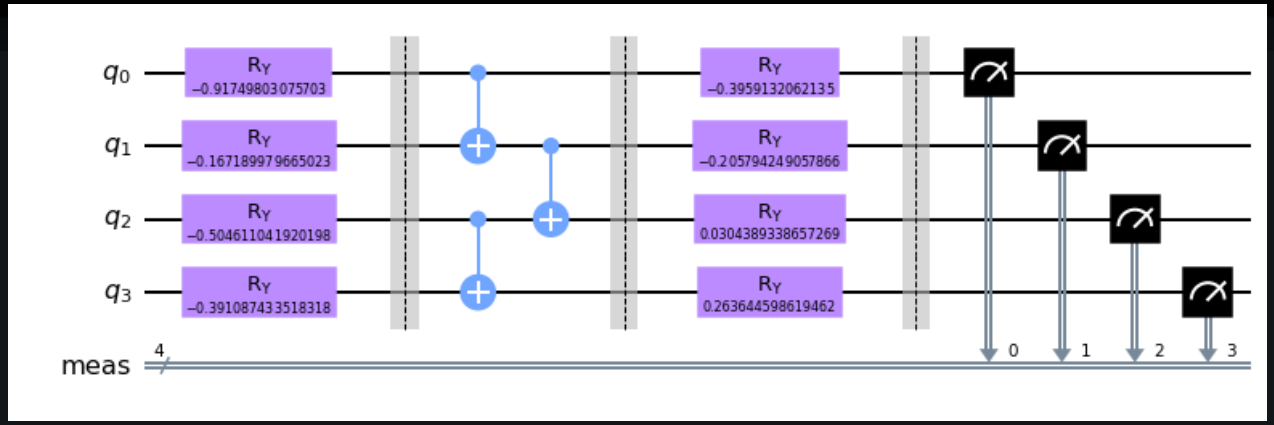
Exercise 2 – M3

Qiskit

True or False!!!

Input the sample_train[0] data into the parameters

```
encode = qc.bind_parameters(sample_train[0])
encode.measure_all()
encode.decompose().draw("mpl")
```

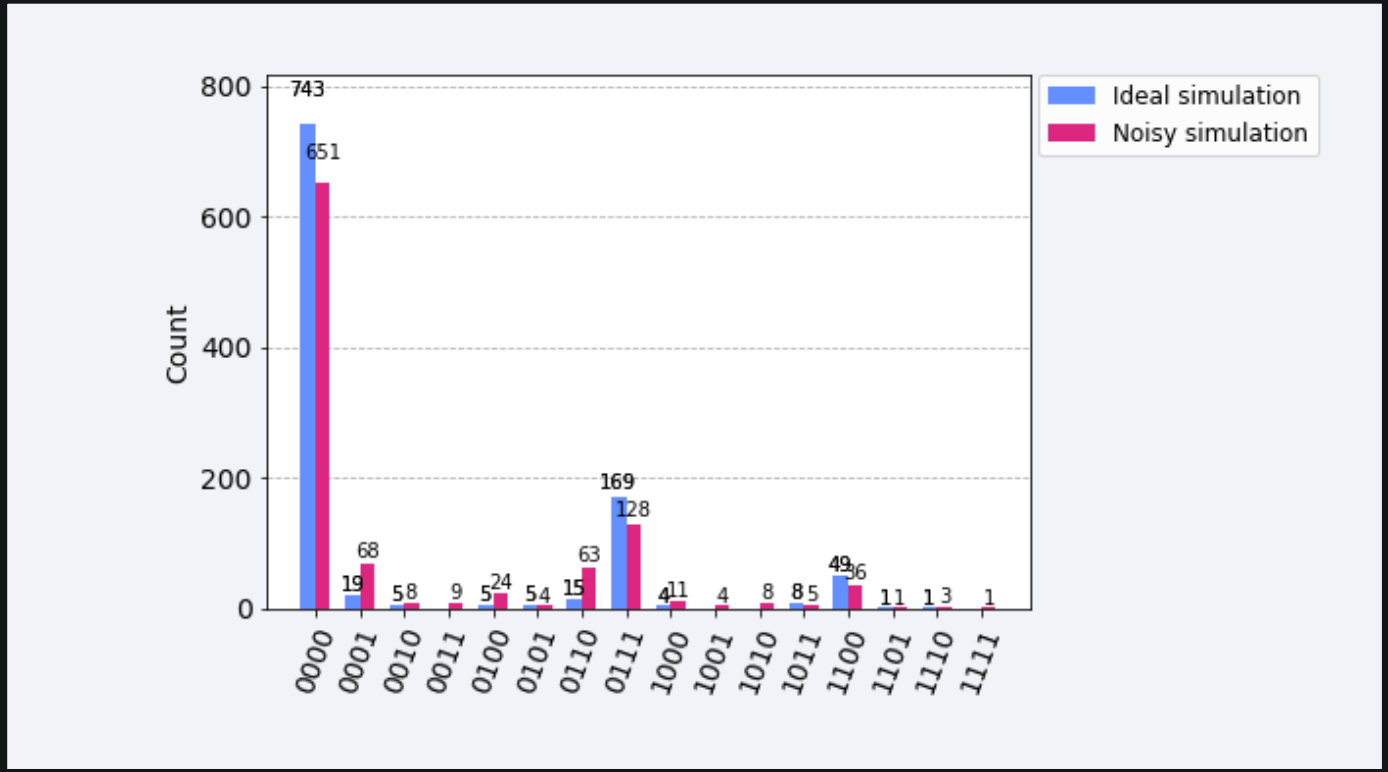


Mimic a real device

Qiskit

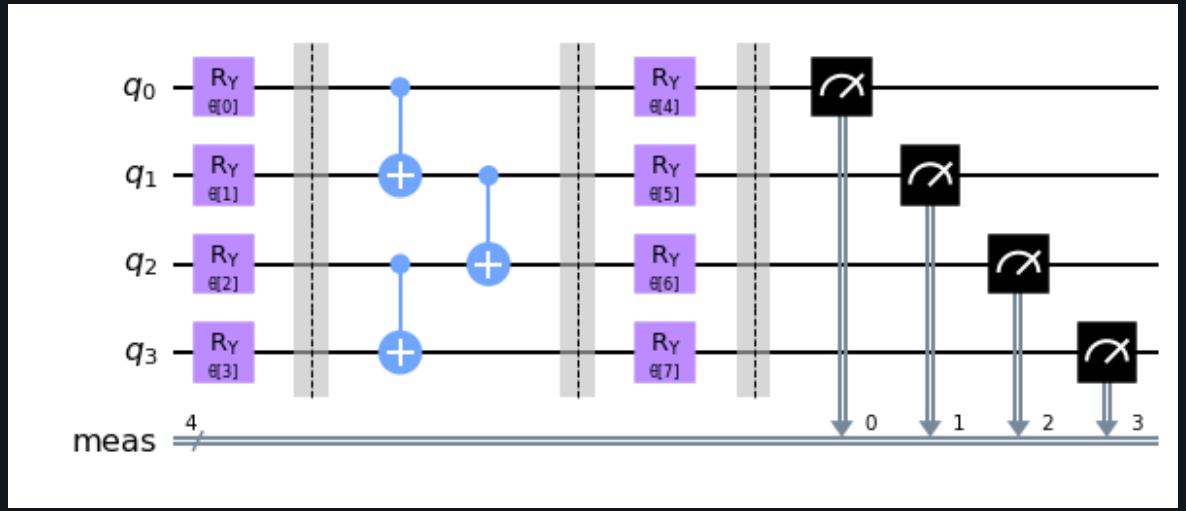
```
from qiskit.providers.fake_provider import FakeManilaV2
# Get a fake backend from the fake provider
noisy_backend = FakeManilaV2()

# Run the transpiled circuit using the simulated fake backend
job_noisy = noisy_backend.run(transpile(encode, noisy_backend), shots=1024)
result_noisy = job_noisy.result()
counts_noisy = result_noisy.get_counts(encode)
```

Mimic a real device

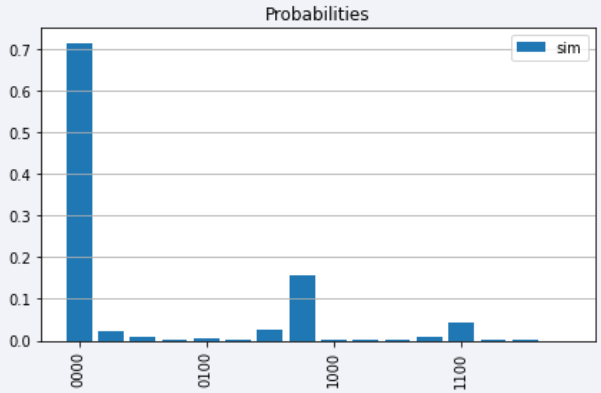


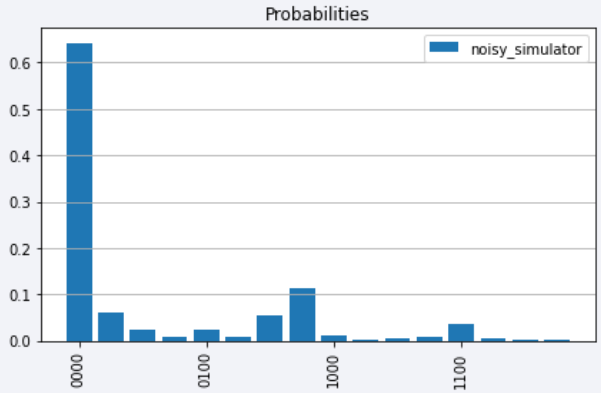
Implementation in Sampler

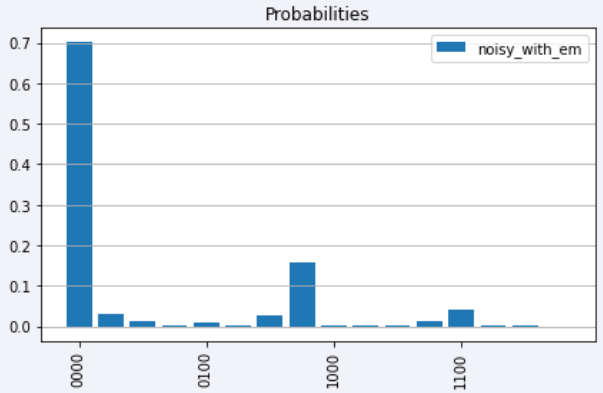
- Ideal Simulation
- Noisy Simulation
- Noisy Simulation with Error mitigation



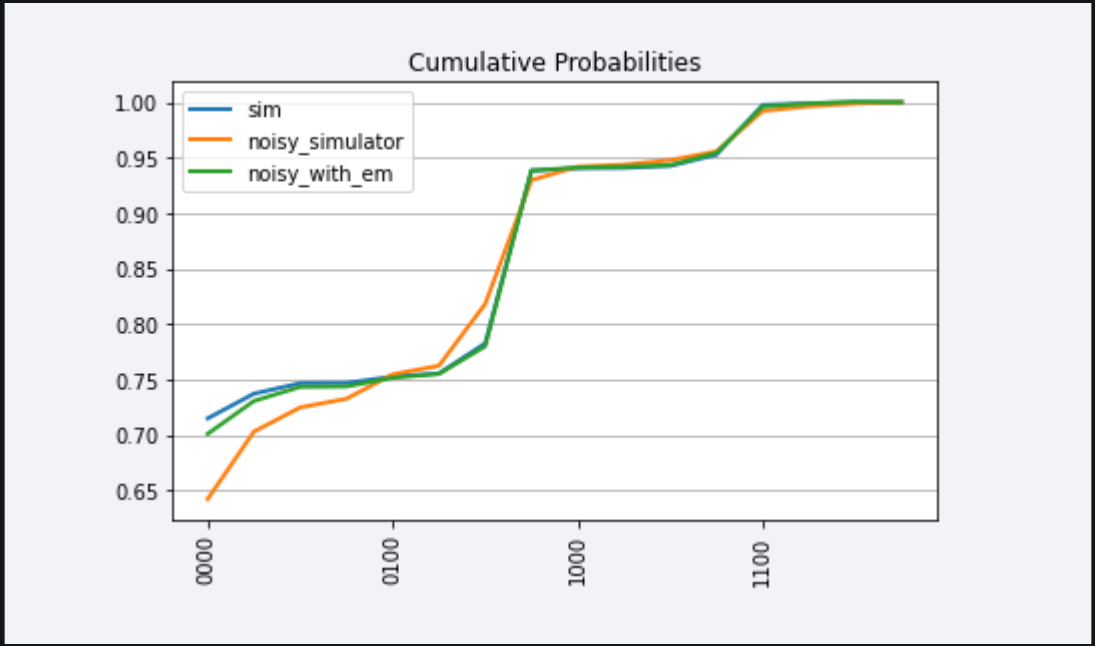
Implementation in Sampler







Implementation in Sampler



Estimating State Fidelities with a Sampler

Qiskit

- Frequent application of the sampler is the estimation of the **fidelity** of two quantum states.

Measurement of the closeness of two quantum state

- If fidelity = 1: Two states are exactly same
- If fidelity = 0: Two states are completely different

Estimating State Fidelities with a Sampler

- Frequent application of the sampler is the estimation of the **fidelity** of two quantum states.

Measurement of the closeness of two quantum state

- If fidelity = 1: Two states are exactly same
- If fidelity = 0: Two states are completely different

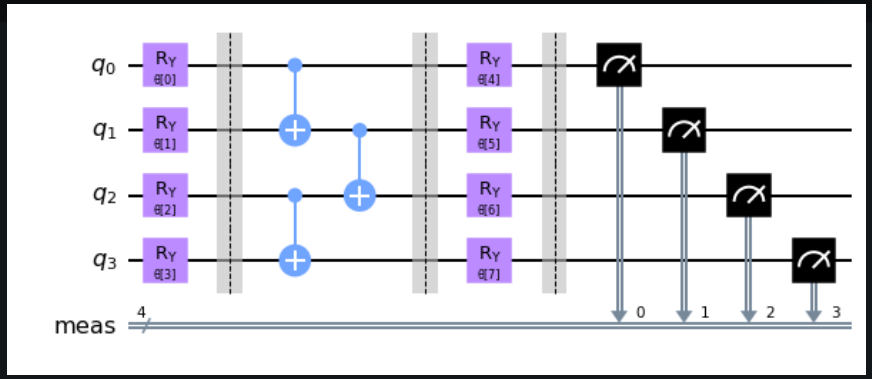
$$|\langle\psi \mid \varphi\rangle|^2 = \left|\langle 0 \mid U^\dagger V \mid 0\rangle\right|^2$$

$|\psi\rangle = U|0\rangle$ $\quad$ $|\varphi\rangle = V|0\rangle$

Fidelity Circuit

Qiskit

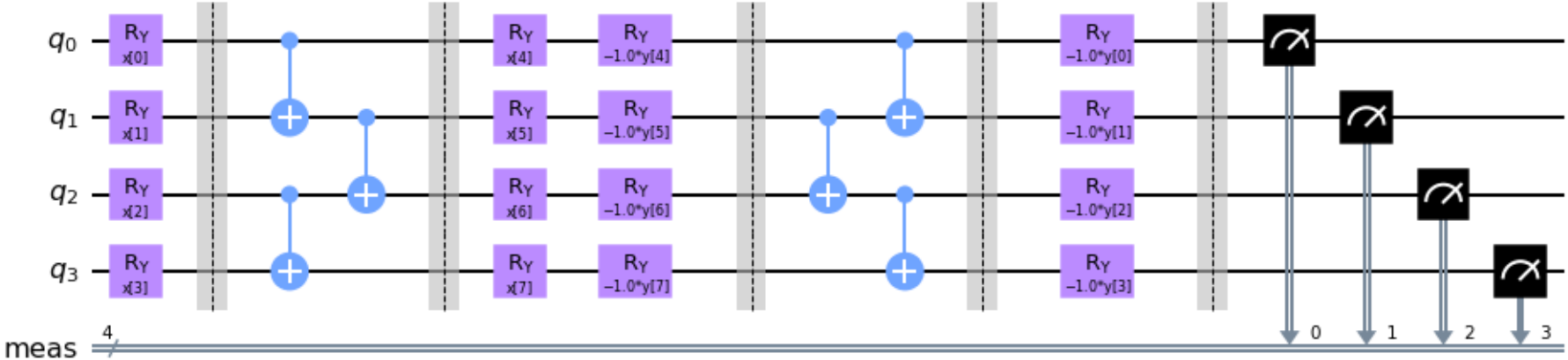
```
# parametrized circuit defining U first state
circuit_1 = RealAmplitudes(4, reps=1, entanglement='pairwise',
                           insert_barriers=True, parameter_prefix='x')

# parametrized circuit V defining second state
circuit_2 = RealAmplitudes(4, reps=1, entanglement='pairwise',
                           insert_barriers=True, parameter_prefix='y')
```



Fidelity Circuit

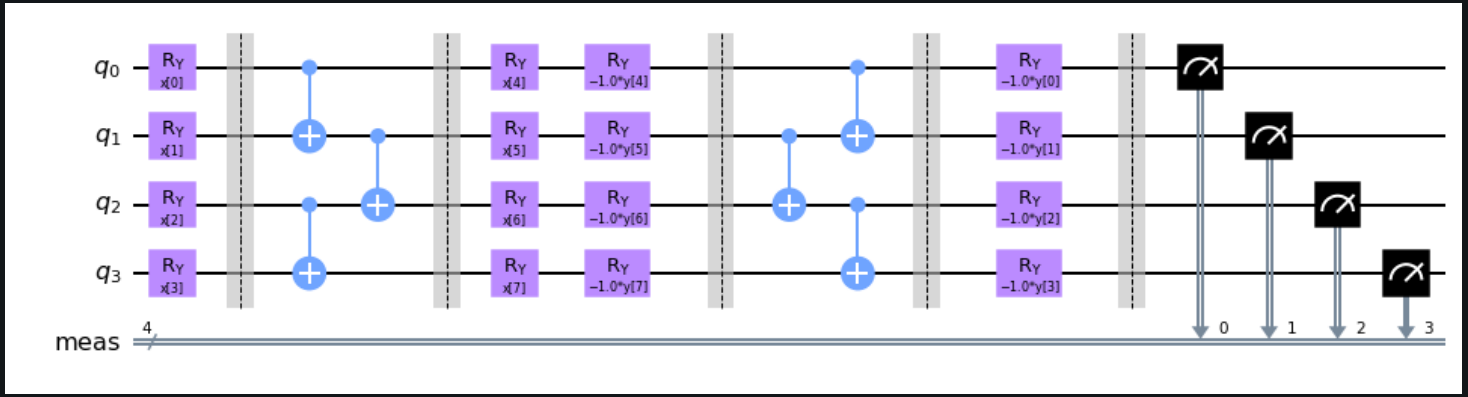
```
# combining circuits to evaluate U^dagger V
fidelity_circuit = circuit_1.copy()
fidelity_circuit.append(circuit_2.inverse().decompose(), range(fidelity_circuit.num_qubits))
fidelity_circuit.measure_all()

# drawing resulting circuit to estimate fidelity
fidelity_circuit.decompose().draw('mpl')
```

Fidelity Circuit



Fidelity Circuit

- θ_1,θ_2 parameters for two states fidelity estimation
- Calculate fidelity between state θ_1 and state $\theta_1 + c * (\theta_2 - \theta_1)$ for $c \in [0,1]$
  - We use both same parameterized circuits -> $c = 0$
    - Both states correspond to θ_1
    - Fidelity supposed to be 1



Input for θ_1, θ_2

```
# draw parameters for states 1 & 2
theta_1 = sample_train[0] # label"0"
theta_2 = sample_train[2] # label"1"

# linear combinations of theta_1 and theta_2
thetas = [theta_1 + c * (theta_2 - theta_1) for c in np.linspace(0, 1, 10)]

# list the parameters
theta_list = []
for theta_ in thetas:
    theta_list.append(list(theta_1) + list(theta_))
```

$\theta_1 + c * (\theta_2 - \theta_1)$

Exercise 3

Qiskit

Build the sampler codes to find the state fidelity under the following conditions

1. Ideal Simulation
2. Noisy devices simulator without error mitigation
3. Noisy devices simulator with error mitigation

The number of shots should be fixed to shots = 10000!!!

Exercise 3

```
# create sampler using simulator and sample fidelities
with Session(service=service, backend=backend):
    sampler = Sampler(options=options)
    job =  # build your code here
    fidelity_samples_sim = job.result()

    sampler = Sampler(options=options_noise)
    job =  # build your code here
    fidelity_samples_noise = job.result()

    sampler = Sampler(options=options_with_em)
    job =  # build your code here
    fidelity_samples_with_em = job.result()
```

Exercise 3

Qiskit

```
options = Options(simulator={"seed_simulator": 1234},resilience_level=0)
```

```
options_noise = Options(
    simulator={
        "noise_model": noise_model,
        "seed_simulator": 1234
    },
    resilience_level=0
)
```

```
options_with_em = Options(
    simulator={
        "noise_model": noise_model,
        "seed_simulator": 1234
    },
    resilience_level=1,
)
```

Exercise 3

Qiskit

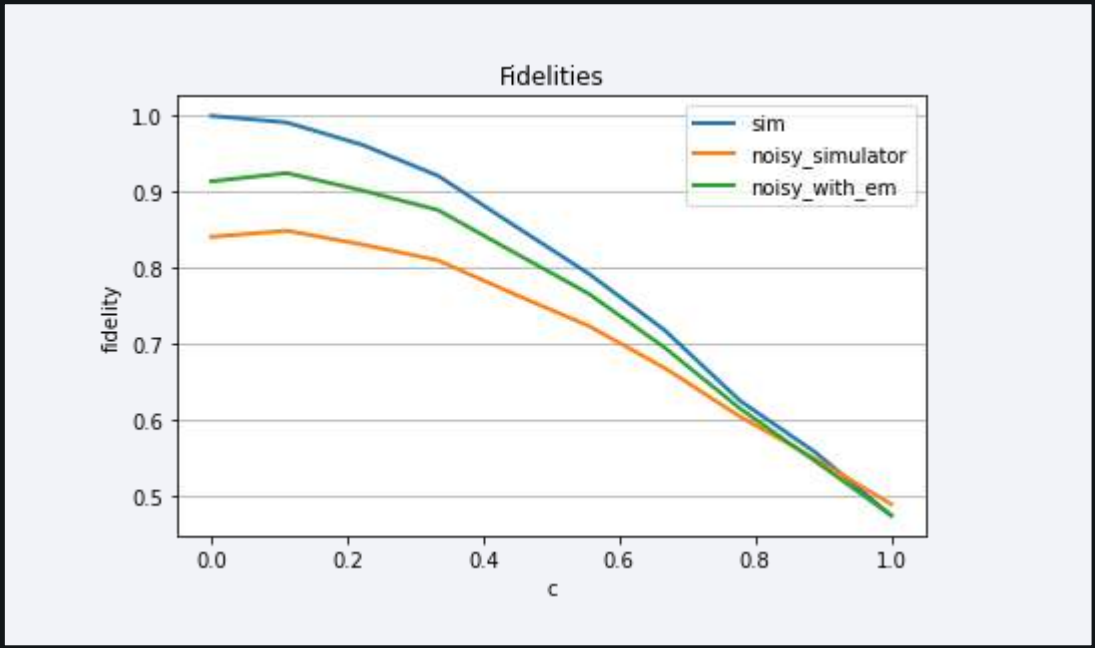
```
options = Options(simulator={"seed_simulator": 1234},resilience_level=0)
```

```
options_noise = Options(
    simulator={
        "noise_model": noise_model,
        "seed_simulator": 1234
    },
    resilience_level=0
)
```

```
fake_backend = FakeManila()
noise_model = NoiseModel.from_backend(fake_backend)
```

```
options_with_em = Options(
    simulator={
        "noise_model": noise_model,
        "seed_simulator": 1234
    },
    resilience_level=1,
)
```

Exercise 3


Fidelities
sim
noisy_simulator
noisy_with_em
fidelity
c
1.0
0.9
0.8
0.7
0.6
0.5
0.0
0.2
0.4
0.6
0.8
1.0

Quantum Kernels and Quantum Support Vector Machines

Qiskit

- QSVM is a classical support vector machine with a quantum kernel.
- Quantum kernel is defined by making the fidelity of two different input of feature map

$$K_{ij} = |\langle \phi^\dagger(\vec{x}_j)|\phi(\vec{x}_i)\rangle|^2$$

- Considering that the feature map is a parameterized quantum circuit we can calculate that kernel matrix with n qubits is $|\langle \phi^\dagger(\vec{x}_j)|\phi(\vec{x}_i)\rangle|^2 = |\langle 0^{\otimes n}|U^\dagger_{\phi(\vec{x}_j)}U_{\phi(\vec{x}_i)}|0^{\otimes n}\rangle|^2$

Quantum Kernels Support Vector Classification Algorithm

Qiskit

1. Build the train quantum kernel matrices
   - Apply the feature map and measure transition probability for each pair of data points in the training dataset $\vec{x}_i, \vec{x}_j$

Quantum Kernels Support Vector Classification Algorithm

Qiskit

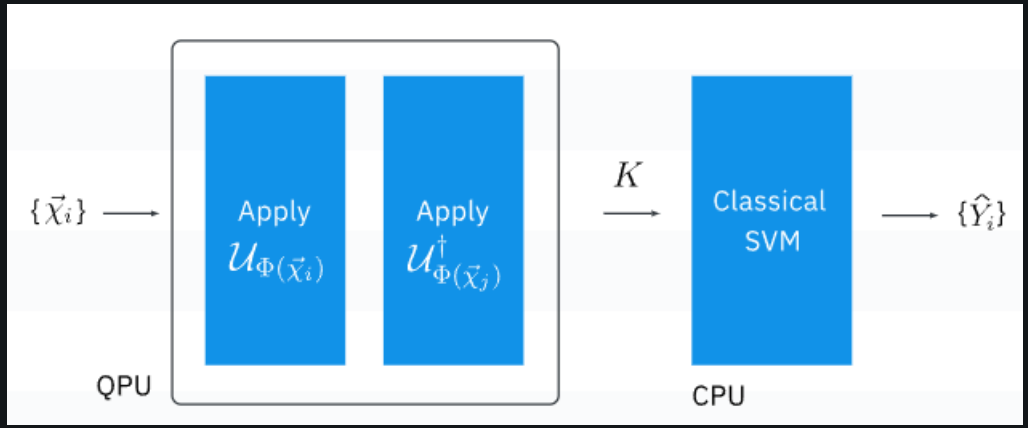
1. Build the train quantum kernel matrices
   - Apply the feature map and measure transition probability for each pair of data points in the training dataset $\vec{x}_i, \vec{x}_j$
2. Build the test quantum kernel matrices
   - Apply the feature map and measure transition probability for each training data point $\vec{x}_i$, and testing point $\vec{y}_i$

Quantum Kernels Support Vector Classification Algorithm

Qiskit

1. Build the train quantum kernel matrices
   - Apply the feature map and measure transition probability for each pair of data points in the training dataset $\vec{x}_i, \vec{x}_j$
2. Build the test quantum kernel matrices
   - Apply the feature map and measure transition probability for each training data point $\vec{x}_i$, and testing point $\vec{y}_i$
3. Classical support vector machine classification algorithm uses train and test quantum kernel matrices

Quantum Kernels Support Vector Classification Algorithm



Note for kernel matrix

- Each row / column represents the transition amplitude of a single data point, with all other data points in the dataset
- The transition amplitude of a data point with itself is 1, so the matrix has a unit diagonal
- The matrix is symmetric, the transition amplitude of $x \to y$ is the same as $y \to x$.

Number of kernel elements

- When the number of data is n, the kernel matrix is $n \times n$ and has n^2 elements.
- But it can be reduced to $(n-1)+(n-2)+\ldots+2+1$

Number of kernel elements

- When the number of data is n, the kernel matrix is $n \times n$ and has n^2 elements.
- But it can be reduced to $(n-1)+(n-2)+\ldots+2+1$

```
n = 25
circuit_num = np.arange(0,n).sum()
print(circuit_num)
```

```
300
```

Encodes data into the quantum kernel circuit

- n is number of data
- x_1 and x_2 are the lists of input data

```
def data_append(n, x1, x2):
    para_data = []
    #
    for i in range(n):
        for j in range(n):
            if i<j:
                para_data.append(list(x1[i])+list(x2[j]))
    #
    return para_data
```

Exercise 4

- Build the Quantum Kernel code using `circuit_num` and `data_append` with Sampler.
- As training data, use `sample_train` which you created in the earlier chapter, "Handwritten bit strings".
- You need to use the noisy simulator FakeManila with the error mitigation, `resilience_level=1`.

```
n = 25
circuit_num = np.arange(0,n).sum()
print(circuit_num)
```

```
def data_append(n, x1, x2):
    para_data = []
    #
    for i in range(n):
        for j in range(n):
            if i<j:
                para_data.append(list(x1[i])+list(x2[j]))
    #
    return para_data
```

Exercise 4

Qiskit

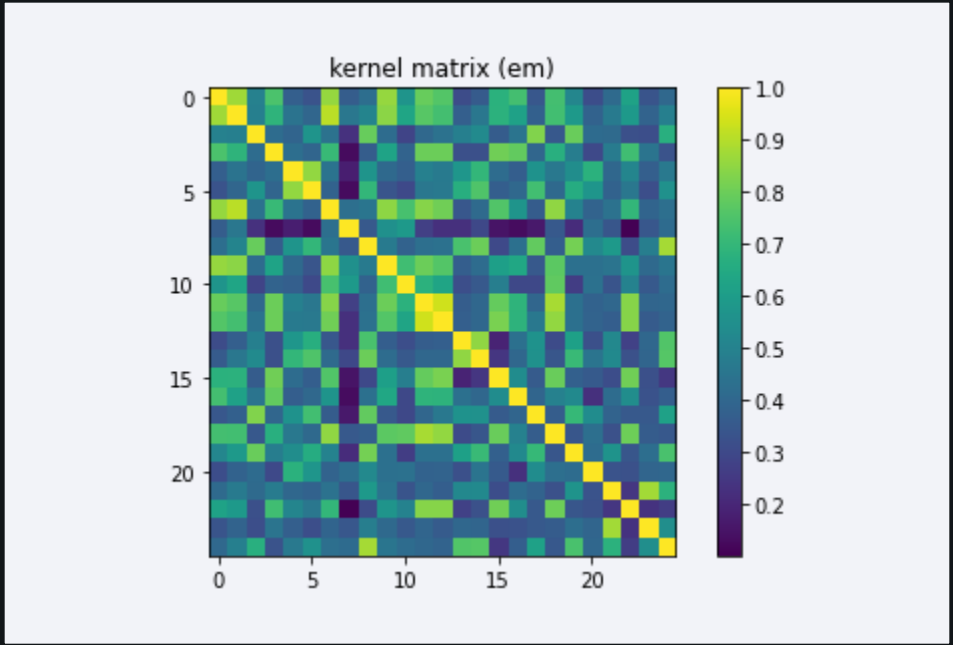
Review which environment defined option you should use:

- options
- options_noise
- options_with_em

number of shots should be fixed to shots = 10000!!!

```
with Session(service=service, backend=backend):
    sampler = # build your code here
    job =  # build your code here
    quantum_kernel_em = job.result()
```

Exercise 4



Train using SVM

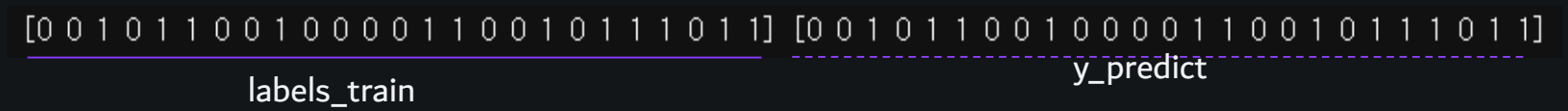
Qiskit

```
svc = SVC(kernel='precomputed')
# train SVM
svc.fit(K, labels_train)
```

```
# score SVM on training data
y_predict = svc.predict(K)
print('accuracy (sim):', sum(y_predict == labels_train)/len(labels_train))
```

```
accuracy (sim): 1.0
```

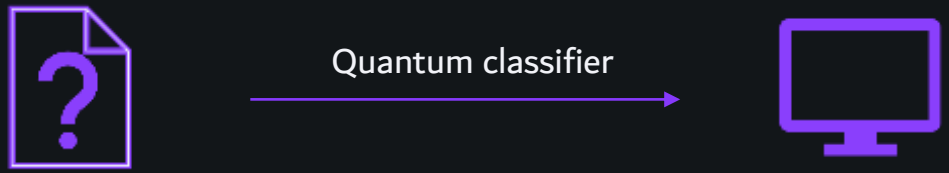
Train using SVM



Exercise 5

Qiskit

- Decode the unknown bitstring unknown_data to check if your quantum classifier is working properly.



Exercise 5

- Decode the unknown bitstring unknown_data to check if your quantum classifier is working properly.

```
n1 = len(unknown_data) # the number of unknown bit-string
n2 = 25 # the number of trained data
circuit_num_ex5 = # build your code here
print(circuit_num_ex5)
```

```
def data_append_ex5(n1, n2, x1, x2):
    para_data_ex5 = []
    #
    #
    # build your code here
    #
    #
    return para_data_ex5
```

Exercise 5

Qiskit

- Decode the unknown bitstring unknown_data to check if your quantum classifier is working properly.

```
with Session(service=service, backend=backend):
    sampler = # build your code here
    job =  # build your code here
    quantum_kernel_ex5 = job.result()
```

Exercise 5