import os, sys, json, time, uuid, logging

from pathlib import Path

from datetime import datetime

RUN\_TS = datetime.now().strftime("%Y%m%d\_%H%M%S")

RUN\_ID = f"qcaa\_{RUN\_TS}\_{uuid.uuid4().hex[:6]}"

LOG\_DIR = Path("logs"); LOG\_DIR.mkdir(parents=True, exist\_ok=True)

logger = logging.getLogger(RUN\_ID)

logger.setLevel(logging.INFO)

fmt = logging.Formatter("[%(asctime)s] %(levelname)s - %(message)s", "%Y-%m-%d %H:%M:%S")

fh = logging.FileHandler(LOG\_DIR / "qcaa\_run.log", encoding="utf-8")

fh.setFormatter(fmt); fh.setLevel(logging.INFO); logger.addHandler(fh)

sh = logging.StreamHandler(sys.stdout)

sh.setFormatter(fmt); sh.setLevel(logging.INFO); logger.addHandler(sh)

JSONL\_PATH = LOG\_DIR / "qcaa\_run.jsonl"

def log\_json(event: str, \*\*kwargs):

rec = {"ts": datetime.now().isoformat(), "run\_id": RUN\_ID, "event": event, \*\*kwargs}

with open(JSONL\_PATH, "a", encoding="utf-8") as f:

f.write(json.dumps(rec, ensure\_ascii=False) + "\n")

def log\_kv(message: str, \*\*kwargs):

logger.info(f"{message} | " + " ".join(f"{k}={v}" for k,v in kwargs.items()))

log\_json(message, \*\*kwargs)

log\_kv("logging\_initialized", run\_id=RUN\_ID, log\_dir=str(LOG\_DIR.resolve()))

from qiskit\_ibm\_runtime import QiskitRuntimeService, SamplerV2 as Sampler

from qiskit import QuantumCircuit, transpile

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import roc\_curve, auc

from scipy.optimize import brentq

from scipy.interpolate import interp1d

from neal import SimulatedAnnealingSampler

from dimod import BinaryQuadraticModel

IBM\_TOKEN = "wZdn6wKKMk4l3XICKoXmAheXQWSsu5JbbBA9Wrh9vpBc"

IBM\_INSTANCE = "crn:v1:bluemix:public:quantum-computing:us-east:a/6148794ccc8942b0b186858407d6ee44:afcd21b9-6c50-40a2-abe9-de5231e2324f::"

DATA\_PATH = r"C:\Users\Sandip Dutta\Downloads\dataset.csv"

SELECTED\_COLS = ['flight\_time\_mean', 'hold\_time\_std', 'gyro\_alpha', 'accel\_y']

SHOTS = 8192

GLOBAL\_SA\_SEED = 7

log\_kv("config\_set", data\_path=DATA\_PATH, shots=SHOTS, cols=",".join(SELECTED\_COLS))

t0 = time.time()

service = QiskitRuntimeService(channel="ibm\_cloud", token=IBM\_TOKEN, instance=IBM\_INSTANCE)

backend = service.backend("ibm\_brisbane")

print(f"✅ Connected to backend: {backend.name}")

log\_kv("backend\_connected", backend=backend.name, seconds=round(time.time()-t0, 3))

df = pd.read\_csv(DATA\_PATH)

X = df[SELECTED\_COLS].values

y = df['label'].astype(int).values

scaler = MinMaxScaler()

X\_norm = scaler.fit\_transform(X)

print("📊 Dataset Info:")

print(f"Total samples: {len(X)}")

print(f"Feature shape: {X\_norm.shape}")

print(f"Genuine: {int(y.sum())}, Impostor: {len(y) - int(y.sum())}")

log\_json("dataset\_loaded",

path=DATA\_PATH, n\_samples=len(X), n\_features=X.shape[1],

n\_genuine=int(y.sum()), n\_impostor=int(len(y)-y.sum()),

feature\_cols=SELECTED\_COLS)

def get\_feature\_mask\_sa(d, seed=None):

rng = np.random.default\_rng(seed)

linear = {i: float(rng.uniform(-1, 1)) for i in range(d)}

quadratic = {(i, j): float(rng.uniform(-1, 1)) for i in range(d) for j in range(i+1, d)}

bqm = BinaryQuadraticModel(linear, quadratic, 0.0, vartype='BINARY')

response = SimulatedAnnealingSampler().sample(bqm, num\_reads=50)

mask = list(response.first.sample.values())

return mask

def build\_qcaa\_optimized\_circuit(x, w):

d = len(x)

qc = QuantumCircuit(d)

for i in range(d):

if w[i] == 1:

xi = np.clip(x[i], 0.001, 0.999)

theta = 2 \* np.arccos(np.sqrt(1 - xi))

qc.ry(theta, i)

for i in range(d - 1):

qc.cz(i, i + 1)

qc.measure\_all()

return qc

circuits = []

feature\_masks = []

for idx, x in enumerate(X\_norm):

w = get\_feature\_mask\_sa(len(x), seed=GLOBAL\_SA\_SEED + idx)

if sum(w) < 2:

on\_idxs = np.random.choice(len(w), size=2, replace=False)

for j in on\_idxs: w[j] = 1

feature\_masks.append(w)

qc = build\_qcaa\_optimized\_circuit(x, w)

circuits.append(transpile(qc, backend=backend, optimization\_level=3))

pd.DataFrame(feature\_masks, columns=[f"f{j}" for j in range(len(SELECTED\_COLS))])\

.to\_csv(LOG\_DIR / "feature\_masks.csv", index=False)

log\_kv("circuits\_compiled", n\_circuits=len(circuits), feature\_masks=str((LOG\_DIR/"feature\_masks.csv").resolve()))

sampler = Sampler(mode=backend)

print("🚀 Submitting job to real quantum backend...")

job = sampler.run(circuits, shots=SHOTS)

job\_id = job.job\_id()

print(f"🆔 Job ID: {job\_id}")

print("⏳ Waiting for result...")

log\_kv("job\_submitted", job\_id=job\_id, shots=SHOTS, n\_circuits=len(circuits))

result = job.result()

print("✅ Job completed.")

log\_kv("job\_completed", job\_id=job\_id)

with open(LOG\_DIR / "job\_meta.txt", "w", encoding="utf-8") as f:

f.write(f"RUN\_ID={RUN\_ID}\njob\_id={job\_id}\nbackend={backend.name}\ninstance={IBM\_INSTANCE}\n")

log\_kv("job\_meta\_saved", path=str((LOG\_DIR / "job\_meta.txt").resolve()))

reconstructed = []

reconstruction\_errors = []

counts\_pub0 = None

def quasis\_or\_counts(res\_obj):

# SamplerV2 often returns quasi distributions; try both patterns

if hasattr(result, "quasi\_dists"):

return "quasi", result.quasi\_dists

try:

return "listlike", result # iterable of results with get\_counts()

except Exception:

return "unknown", None

mode, payload = quasis\_or\_counts(result)

if mode == "quasi":

# payload is a list of dict-like quasi distributions per circuit

for i, qdist in enumerate(payload):

num\_qubits = len(SELECTED\_COLS)

prob\_1 = np.zeros(num\_qubits)

for bitstring, prob in qdist.items():

b = format(bitstring, f"0{num\_qubits}b")[::-1]

for q in range(num\_qubits):

if b[q] == '1':

prob\_1[q] += prob

if i == 0:

# convert to pseudo-counts snapshot for log (scaled by SHOTS)

counts\_pub0 = {format(k, f"0{num\_qubits}b"): int(v\*SHOTS) for k, v in qdist.items()}

x\_hat = np.arcsin(np.sqrt(np.clip(prob\_1, 1e-10, 1-1e-10)))

reconstructed.append(x\_hat)

reconstruction\_errors.append(float(np.mean((X\_norm[i] - x\_hat) \*\* 2)))

else:

# fallback: iterable with get\_counts()

for i, circ\_res in enumerate(payload):

num\_qubits = len(SELECTED\_COLS)

prob\_1 = np.zeros(num\_qubits)

counts = None

try:

counts = circ\_res.data.meas.get\_counts()

except Exception:

try:

counts = circ\_res.get\_counts()

except Exception:

pass

if counts is None:

log\_kv("counts\_unavailable", circuit=i); continue

if i == 0:

counts\_pub0 = counts

total = max(sum(counts.values()), 1)

for bitstring, cnt in counts.items():

bits = bitstring[::-1]

for q in range(min(num\_qubits, len(bits))):

if bits[q] == '1':

prob\_1[q] += cnt

prob\_1 = prob\_1 / total

x\_hat = np.arcsin(np.sqrt(np.clip(prob\_1, 1e-10, 1-1e-10)))

reconstructed.append(x\_hat)

reconstruction\_errors.append(float(np.mean((X\_norm[i] - x\_hat) \*\* 2)))

reconstruction\_errors = np.array(reconstruction\_errors)

pd.DataFrame({

"index": np.arange(len(reconstruction\_errors)),

"label": y[:len(reconstruction\_errors)],

"mse": reconstruction\_errors

}).to\_csv(LOG\_DIR / "recon\_errors.csv", index=False)

log\_kv("recon\_errors\_saved", path=str((LOG\_DIR / "recon\_errors.csv").resolve()), n=len(reconstruction\_errors))

if counts\_pub0 is not None:

with open(LOG\_DIR / "counts\_pub0.json", "w", encoding="utf-8") as f:

json.dump(counts\_pub0, f, ensure\_ascii=False, indent=2)

log\_kv("counts\_pub0\_saved", path=str((LOG\_DIR / "counts\_pub0.json").resolve()))

def calculate\_eer(fpr, tpr):

return float(brentq(lambda x: 1. - x - interp1d(fpr, tpr)(x), 0., 1.))

valid\_n = min(len(reconstruction\_errors), len(y))

fpr, tpr, thr = roc\_curve(y[:valid\_n], -reconstruction\_errors[:valid\_n])

auc\_val = float(auc(fpr, tpr))

eer\_val = calculate\_eer(fpr, tpr)

pd.DataFrame({"fpr": fpr, "tpr": tpr, "thr": thr}).to\_csv(LOG\_DIR / "roc\_points.csv", index=False)

log\_kv("roc\_points\_saved", path=str((LOG\_DIR / "roc\_points.csv").resolve()))

with open(LOG\_DIR / "summary.txt", "w", encoding="utf-8") as f:

f.write(f"RUN\_ID: {RUN\_ID}\n")

f.write(f"Backend: {backend.name}\n")

f.write(f"Job ID: {job\_id}\n")

f.write(f"Samples: {valid\_n} / {len(X\_norm)}\n")

f.write(f"Average Reconstruction MSE: {np.mean(reconstruction\_errors):.4f}\n")

f.write(f"AUC: {auc\_val:.4f}\n")

f.write(f"EER: {eer\_val:.4f}\n")

log\_kv("summary\_saved",

avg\_mse=round(float(np.mean(reconstruction\_errors)), 6),

auc=round(auc\_val, 6), eer=round(eer\_val, 6))

print("\n📊 Performance Summary:")

print(f"- Average Reconstruction MSE: {np.mean(reconstruction\_errors):.4f}")

print(f"- AUC: {auc\_val:.4f}")

print(f"- Equal Error Rate (EER): {eer\_val:.4f}")

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, label=f"AUC={auc\_val:.2f}, EER={eer\_val:.2f}")

plt.plot([0, 1], [0, 1], linestyle="--")

plt.xlim([0, 1]); plt.ylim([0, 1])

plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")

plt.title("Optimized QCAA ROC")

plt.legend(loc="lower right"); plt.grid(True)

plt.show()