# === Noise Sensitivity on Real Hardware (Batched, Counts-Fallback) — Biometric Dataset ===

# Dataset: C:\Users\Sandip Dutta\OneDrive\Desktop\biometric\_detailed\_dataset.csv

# Backend: ibm\_brisbane (SamplerV2)

# Models: QCAA (Non-Opt.) and QCAA (Optimized)

# Noise: Gaussian @ 5%, 10%, 20%, 30% (in normalized space)

# Jobs: 2 total (one per model, batching clean+all noise levels)

# Output: log11/noise\_robustness\_biometric.csv

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

from pathlib import Path

from datetime import datetime

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from sklearn.model\_selection import StratifiedKFold

from sklearn.metrics import roc\_curve

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

from qiskit import QuantumCircuit, transpile

from neal import SimulatedAnnealingSampler

from dimod import BinaryQuadraticModel

# ---------------- Logging ----------------

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

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

LOG\_DIR = Path("log11"); 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 / "noise\_bio\_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)

def log\_json(event, \*\*kw):

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

with open(LOG\_DIR / "noise\_bio\_run.jsonl", "a", encoding="utf-8") as f:

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

def log\_kv(msg, \*\*kw):

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

log\_json(msg, \*\*kw)

# ---------------- Config (UPDATED CREDS) ----------------

IBM\_TOKEN = "ynRx1PCW4ipBRI-cZeL5LQp4VOh4YxAXjrmJDQHAr0XX"

IBM\_INSTANCE = "crn:v1:bluemix:public:quantum-computing:us-east:a/0a0a05e6d1a44757bc62c3e6b305f821:ac760e90-3c3e-41ed-8f33-a53cd7c2bf39::"

DATA\_PATH = r"C:\Users\Sandip Dutta\OneDrive\Desktop\biometric\_detailed\_dataset.csv"

LABEL\_CANDS = ["label", "y", "target", "Label", "Target"]

# Execution knobs (faster)

SHOTS = 2048 # lower for speed; raise to 3072+ for tighter stats

OPT\_LEVEL = 1 # faster transpile (was 3)

CV\_SPLITS = 5

NOISE\_LEVELS = [0.05, 0.10, 0.20, 0.30]

GAUSS\_SEED = 123

# SA masks for QCAA-Optimized

GLOBAL\_SA\_SEED = 7

SA\_READS = 50

MIN\_ON = 2

log\_kv("config\_set", data\_path=DATA\_PATH, shots=SHOTS, noise\_levels=",".join(map(str, NOISE\_LEVELS)))

# ---------------- Connect backend ----------------

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)

# ---------------- Load & normalize data ----------------

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

label\_col = next((c for c in LABEL\_CANDS if c in df.columns), None)

if label\_col is None: raise ValueError("No label column found. Include 'label' (or y/target).")

feature\_cols = [c for c in df.columns if c != label\_col and np.issubdtype(df[c].dtype, np.number)]

if not feature\_cols: raise ValueError("No numeric feature columns besides label.")

X\_raw = df[feature\_cols].to\_numpy(dtype=float)

y = df[label\_col].astype(int).to\_numpy()

scaler = MinMaxScaler()

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

log\_kv("dataset\_loaded", n\_samples=len(X\_norm), n\_features=X\_norm.shape[1],

n\_genuine=int(y.sum()), n\_impostor=int(len(y)-y.sum()), feature\_cols=",".join(feature\_cols))

# ---------------- Helpers ----------------

def fit\_threshold(train\_scores, train\_labels):

fpr, tpr, thr = roc\_curve(train\_labels, -train\_scores) # higher score = more genuine

J = tpr - fpr

return float(thr[int(np.argmax(J))])

def add\_noise(Xn, sigma, rng):

return np.clip(Xn + rng.normal(0.0, sigma, Xn.shape), 0.0, 1.0)

def recon\_error\_from\_P1(P1, Xref):

x\_hat = np.arcsin(np.sqrt(np.clip(P1, 1e-12, 1-1e-12)))

return ((Xref - x\_hat) \*\* 2).mean(axis=1)

def to\_bits\_any(k, width):

if isinstance(k, int):

return format(k, f"0{width}b")

if isinstance(k, str):

s = k.replace(" ", "")

if set(s) <= {"0","1"}:

return s.zfill(width)[-width:]

try:

return format(int(s, 2), f"0{width}b")

except Exception:

return s.zfill(width)[-width:]

if isinstance(k, tuple):

try:

return "".join("1" if bool(v) else "0" for v in k).zfill(width)[-width:]

except Exception:

return "".join(str(v) for v in k).zfill(width)[-width:]

return "0"\*width

# ------- Circuit builders (safe encoding with clipping) -------

def build\_circuit\_qcaa(x\_vec):

d = len(x\_vec); qc = QuantumCircuit(d)

for i in range(d):

xi = float(np.clip(x\_vec[i], 1e-9, 1-1e-9))

qc.ry(2.0 \* np.arcsin(np.sqrt(xi)), i)

for i in range(d-1): qc.cz(i, i+1)

qc.measure\_all()

return qc

def build\_circuit\_qcaa\_opt(x\_vec, mask):

d = len(x\_vec); qc = QuantumCircuit(d)

active = [i for i,m in enumerate(mask) if m==1]

for i in active:

xi = float(np.clip(x\_vec[i], 1e-9, 1-1e-9))

qc.ry(2.0 \* np.arcsin(np.sqrt(xi)), i)

for a,b in zip(active[:-1], active[1:]): qc.cz(a,b)

qc.measure\_all()

return qc

def sa\_mask(d, seed=None, reads=50, min\_on=2):

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

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

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

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

resp = SimulatedAnnealingSampler().sample(bqm, num\_reads=reads, seed=seed)

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

if sum(mask) < min\_on:

for i in sorted(range(d), key=lambda k: abs(linear[k]), reverse=True)[:min\_on]:

mask[i] = 1

return mask

# -------- Robust sampler: supports quasi\_dists OR raw counts ----------

def sampler\_prob\_ones(transpiled\_circuits, tag):

sampler = Sampler(mode=backend)

job = sampler.run(transpiled\_circuits, shots=SHOTS)

job\_id = job.job\_id()

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

res = job.result()

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

num\_qubits = len(feature\_cols)

N = len(transpiled\_circuits)

P1 = np.zeros((N, num\_qubits), dtype=float)

# Case A: modern quasi distributions

if hasattr(res, "quasi\_dists") and res.quasi\_dists:

qlist = res.quasi\_dists

for i, qdist in enumerate(qlist):

row = np.zeros(num\_qubits)

for k, prob in dict(qdist).items():

bits = to\_bits\_any(k, num\_qubits)[::-1]

for q in range(num\_qubits):

if bits[q] == "1":

row[q] += float(prob)

P1[i] = row

# Debug snapshot

try:

counts\_pub0 = {to\_bits\_any(k, num\_qubits): int(float(v)\*SHOTS) for k, v in dict(qlist[0]).items()}

with open(LOG\_DIR / f"counts\_pub0\_{tag}.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 / f"counts\_pub0\_{tag}.json").resolve()), tag=tag)

except Exception:

pass

return P1

# Case B: fallback to counts-like results

# Try to iterate result entries

try:

results\_iter = list(res)

except TypeError:

results\_iter = getattr(res, "results", None)

if results\_iter is None:

log\_kv("error", note="Unknown Sampler result shape; neither quasi\_dists nor iterable.")

return P1

def try\_get\_counts(r):

# try multiple patterns

for getter in (

lambda r: r.data.meas.get\_counts(),

lambda r: r.get\_counts(),

lambda r: r.data.counts,

lambda r: r.metadata.get("counts", None),

):

try:

c = getter(r)

if c:

return dict(c)

except Exception:

pass

return None

for i, circ\_res in enumerate(results\_iter):

counts = try\_get\_counts(circ\_res)

if not counts:

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

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

row = np.zeros(num\_qubits)

for bitstring, cnt in counts.items():

bits = to\_bits\_any(bitstring, num\_qubits)[::-1]

for q in range(num\_qubits):

if bits[q] == "1":

row[q] += int(cnt)

P1[i] = row / float(total)

# Snapshot of the first counts

try:

snap = try\_get\_counts(results\_iter[0])

if snap:

snap\_bits = {to\_bits\_any(k, num\_qubits): int(v) for k, v in snap.items()}

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

json.dump(snap\_bits, f, ensure\_ascii=False, indent=2)

log\_kv("counts\_pub0\_saved", path=str((LOG\_DIR / f"counts\_pub0\_{tag}.json").resolve()), tag=tag)

except Exception:

pass

return P1

# ---------------- Masks for QCAA-Optimized (fixed from clean) ----------------

masks = [sa\_mask(X\_norm.shape[1], seed=GLOBAL\_SA\_SEED+i, reads=SA\_READS, min\_on=MIN\_ON)

for i in range(len(X\_norm))]

pd.DataFrame(masks, columns=[f"f{j}" for j in range(X\_norm.shape[1])]).to\_csv(LOG\_DIR / "masks\_qcaa\_opt.csv", index=False)

# ---------------- Build batched circuits for BOTH algorithms ----------------

rng = np.random.default\_rng(GAUSS\_SEED)

all\_levels = [("clean", None)] + [(f"noise\_{int(p\*100)}", p) for p in NOISE\_LEVELS]

def make\_batched\_circuits(algorithm="qcaa"):

circuits = []; index\_map = {}; start = 0

for tag, sigma in all\_levels:

X\_use = X\_norm if sigma is None else add\_noise(X\_norm, sigma, rng)

if algorithm == "qcaa":

cs = [build\_circuit\_qcaa(x) for x in X\_use]

else:

cs = [build\_circuit\_qcaa\_opt(x, m) for x, m in zip(X\_use, masks)]

circuits.extend(cs)

end = start + len(cs)

index\_map[tag] = (start, end, sigma, X\_use)

start = end

circuits\_t = [transpile(c, backend=backend, optimization\_level=OPT\_LEVEL) for c in circuits]

return circuits\_t, index\_map

# ---------------- Run 2 jobs total (no session needed) ----------------

qcaa\_circuits\_t, qcaa\_map = make\_batched\_circuits("qcaa")

P1\_qcaa\_all = sampler\_prob\_ones(qcaa\_circuits\_t, "qcaa\_batch")

opt\_circuits\_t, opt\_map = make\_batched\_circuits("opt")

P1\_opt\_all = sampler\_prob\_ones(opt\_circuits\_t, "qcaa\_opt\_batch")

# ---------------- Split results, compute errors ----------------

def slice\_errors(P1\_all, idx\_map):

out = {}

for tag, (s,e, sigma, Xref) in idx\_map.items():

out[tag] = recon\_error\_from\_P1(P1\_all[s:e], Xref)

return out

RE\_qcaa = slice\_errors(P1\_qcaa\_all, qcaa\_map)

RE\_opt = slice\_errors(P1\_opt\_all, opt\_map)

# ---------------- Compute accuracies ----------------

def accuracy\_from\_clean\_and\_noisy(RE\_clean, RE\_noisy, labels, cv\_splits=5):

skf = StratifiedKFold(n\_splits=cv\_splits, shuffle=True, random\_state=42)

accs\_clean, accs\_noisy = [], []

for tr, te in skf.split(RE\_clean.reshape(-1,1), labels):

thr = fit\_threshold(RE\_clean[tr], labels[tr])

y\_pred\_clean = (-RE\_clean[te] >= thr).astype(int)

y\_pred\_noisy = (-RE\_noisy[te] >= thr).astype(int)

accs\_clean.append((y\_pred\_clean == labels[te]).mean())

accs\_noisy.append((y\_pred\_noisy == labels[te]).mean())

return 100.0 \* float(np.mean(accs\_clean)), 100.0 \* float(np.mean(accs\_noisy))

rows = []

for tag, sigma in all\_levels[1:]: # skip "clean"

clean\_acc\_qcaa, noisy\_acc\_qcaa = accuracy\_from\_clean\_and\_noisy(RE\_qcaa["clean"], RE\_qcaa[tag], y, CV\_SPLITS)

drop\_qcaa = clean\_acc\_qcaa - noisy\_acc\_qcaa

rec\_qcaa = (noisy\_acc\_qcaa / clean\_acc\_qcaa \* 100.0) if clean\_acc\_qcaa > 0 else 0.0

rows.append([int(sigma\*100), "QCAA (Non-Opt.)",

f"{clean\_acc\_qcaa:.1f}%", f"{noisy\_acc\_qcaa:.1f}%", f"{drop\_qcaa:.1f} pp", f"{rec\_qcaa:.1f}%"])

clean\_acc\_opt, noisy\_acc\_opt = accuracy\_from\_clean\_and\_noisy(RE\_opt["clean"], RE\_opt[tag], y, CV\_SPLITS)

drop\_opt = clean\_acc\_opt - noisy\_acc\_opt

rec\_opt = (noisy\_acc\_opt / clean\_acc\_opt \* 100.0) if clean\_acc\_opt > 0 else 0.0

rows.append([int(sigma\*100), "QCAA (Optimized)",

f"{clean\_acc\_opt:.1f}%", f"{noisy\_acc\_opt:.1f}%", f"{drop\_opt:.1f} pp", f"{rec\_opt:.1f}%"])

log\_kv("noise\_result",

noise\_pct=int(sigma\*100),

qcaa={"clean": round(clean\_acc\_qcaa,2), "noisy": round(noisy\_acc\_qcaa,2),

"drop": round(drop\_qcaa,2), "recovery": round(rec\_qcaa,2)},

qcaa\_opt={"clean": round(clean\_acc\_opt,2), "noisy": round(noisy\_acc\_opt,2),

"drop": round(drop\_opt,2), "recovery": round(rec\_opt,2)})

# ---------------- Save final table ----------------

out\_df = pd.DataFrame(rows, columns=[

"Noise Level (%)", "Model", "Clean Acc (%)", "Noisy Acc (%)", "Drop (pp)", "Recovery Rate (%)"

])

OUT\_CSV = LOG\_DIR / "noise\_robustness\_biometric.csv"

out\_df.to\_csv(OUT\_CSV, index=False)

print("\n=== Noise Robustness — Biometric Dataset (Batched, Real Hardware, Counts-Fallback) ===")

print(out\_df.to\_string(index=False))

print(f"\nSaved to: {OUT\_CSV.resolve()}")