# === QCAA vs QCAA-Optimized: 15-in-1 HYBRID Analysis (Fast) ===

# - Uses real hardware minimally (ibm\_brisbane) for baseline P(1) snapshots

# - Runs analyses classically for speed

# - Saves everything into log23/

# - Your credentials embedded per request

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

from pathlib import Path

from datetime import datetime

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, confusion\_matrix, precision\_recall\_fscore\_support

from sklearn.model\_selection import StratifiedShuffleSplit, StratifiedKFold

# ---------- USER PATHS ----------

PATH\_GAIT = r"C:\Users\Sandip Dutta\OneDrive\Desktop\data\gait\_data.csv"

PATH\_FTXT = r"C:\Users\Sandip Dutta\OneDrive\Desktop\data\fixed\_text\_keystroke.csv"

PATH\_FREET = r"C:\Users\Sandip Dutta\OneDrive\Desktop\data\free\_text\_keystroke.csv"

# ---------- IBM QUANTUM CREDS (as provided) ----------

IBM\_TOKEN = "isTxH69BGxixH7QohOX\_F8Zxm9fvMY4FP4ZET6F9xjTZ"

IBM\_INSTANCE = "crn:v1:bluemix:public:quantum-computing:us-east:a/34961d67783d401f880bc62b6543135b:a4d04a9f-4d50-445b-b363-7db4050ad8c2::"

BACKEND\_NAME = "ibm\_brisbane"

# ---------- OUTPUT / LOGGING ----------

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

OUTDIR = Path("log23"); OUTDIR.mkdir(parents=True, exist\_ok=True)

LOG\_JSONL = OUTDIR / "run.jsonl"

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

logger.setLevel(logging.INFO)

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

fh = logging.FileHandler(OUTDIR / "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\_kv(event, \*\*kw):

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

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

f.write(json.dumps({"ts": datetime.now().isoformat(), "event": event, \*\*kw}, ensure\_ascii=False) + "\n")

log\_kv("config\_set", outdir=str(OUTDIR.resolve()), backend=BACKEND\_NAME)

# ---------- OPTIONAL QUANTUM IMPORTS (lazy) ----------

QUANTUM\_OK = True

try:

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

from qiskit import QuantumCircuit, transpile

except Exception as e:

QUANTUM\_OK = False

log\_kv("quantum\_disabled", reason="qiskit imports failed", error=str(e))

# ---------- Utility / Common ----------

np.random.seed(42)

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

ENCODINGS = ["angle", "amplitude", "hybrid"] # for simulation sweeps

def load\_dataset(path):

df = pd.read\_csv(path)

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

if not label\_col:

raise ValueError(f"No label column in {path}")

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(f"No numeric feature columns in {path}")

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

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

# Preserve chronological order if a session/time column exists

session\_col = next((c for c in ["session","timestamp","time","date","Session","Timestamp"] if c in df.columns), None)

order = np.argsort(df[session\_col].values) if session\_col else np.arange(len(df))

X, y = X[order], y[order]

scaler = MinMaxScaler()

Xn = scaler.fit\_transform(X)

return {"X": Xn, "y": y, "features": feature\_cols, "label": label\_col, "scaler": scaler}

DATASETS = {

"gait": load\_dataset(PATH\_GAIT),

"fixed": load\_dataset(PATH\_FTXT),

"free": load\_dataset(PATH\_FREET),

}

for name, d in DATASETS.items():

log\_kv("dataset\_loaded", name=name, n=len(d["X"]), d=d["X"].shape[1], positives=int(d["y"].sum()), negatives=int((1-d["y"]).sum()))

# ---------- QCAA / QCAA-Opt Surrogates (fast simulation) ----------

def qcaa\_simulate\_P1(X, depol=0.08, crosstalk=0.10):

X = np.asarray(X, dtype=float)

left = np.concatenate([X[:, :1], X[:, :-1]], axis=1)

right = np.concatenate([X[:, 1:], X[:, -1:]], axis=1)

mix = (left + right) / 2.0

P = (1 - crosstalk) \* X + crosstalk \* mix

P = (1 - depol) \* P + depol \* 0.5

return np.clip(P, 1e-9, 1-1e-9)

def qcaa\_opt\_simulate\_P1(X, depol=0.06, crosstalk=0.08, active\_frac=0.5):

X = np.asarray(X, dtype=float)

n, d = X.shape

k = max(2, int(np.ceil(active\_frac \* d)))

mu = X.mean(axis=0, keepdims=True)

z = np.abs(X - mu)

P0 = np.zeros\_like(X)

idx\_sorted = np.argsort(-z, axis=1)

rows = np.arange(n)[:, None]

active\_cols = idx\_sorted[:, :k]

P0[rows, active\_cols] = X[rows, active\_cols]

left = np.concatenate([P0[:, :1], P0[:, :-1]], axis=1)

right = np.concatenate([P0[:, 1:], P0[:, -1:]], axis=1)

mix = (left + right) / 2.0

P = (1 - crosstalk) \* P0 + crosstalk \* mix

P = (1 - depol) \* P + depol \* 0.5

return np.clip(P, 1e-9, 1-1e-9)

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 fit\_threshold(scores, labels):

fpr, tpr, thr = roc\_curve(labels, -scores) # lower error => genuine

J = tpr - fpr

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

def metrics\_from\_scores(scores, y, thr):

y\_pred = (scores <= thr).astype(int)

acc = (y\_pred == y).mean()

p, r, f1, \_ = precision\_recall\_fscore\_support(y, y\_pred, average="binary", zero\_division=0)

fpr, tpr, \_ = roc\_curve(y, -scores)

return {

"acc": acc,

"precision": p,

"recall": r,

"f1": f1,

"auc": auc(fpr, tpr)

}

# ---------- Minimal REAL HARDWARE SNAPSHOT (fast) ----------

def run\_hardware\_snapshot(name, Xref, features, max\_samples=8, shots=512, opt\_level=1):

"""

Angle-encoding QCAA & mask-based QCAA-Opt on a tiny subset to anchor results.

Returns dict: { "P1\_qcaa": array, "P1\_opt": array, "err\_qcaa": array, "err\_opt": array }

"""

if not QUANTUM\_OK:

log\_kv("hardware\_skip", reason="qiskit not available")

return None

# Connect

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

backend = service.backend(BACKEND\_NAME)

log\_kv("backend\_connected", backend=BACKEND\_NAME, name=name)

# Pick subset indices stratified

y\_dummy = np.array([1]\*len(Xref)) # label not used here; just to pick a balanced subset by index

sel = np.linspace(0, len(Xref)-1, num=min(max\_samples, len(Xref)), dtype=int)

Xsub = Xref[sel]

d = Xsub.shape[1]

# Builders

def build\_qcaa(x):

qc = QuantumCircuit(d)

for i in range(d):

xi = float(np.clip(x[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

# simple mask: top-|active| by deviation from mean in subset

mu = Xsub.mean(axis=0)

k = max(2, int(np.ceil(0.5\*d)))

def mask\_for\_row(x):

idx = np.argsort(-np.abs(x-mu))[:k]

m = np.zeros(d, dtype=int); m[idx] = 1

return m

def build\_qcaa\_opt(x, m):

qc = QuantumCircuit(d)

active = [i for i in range(d) if m[i]==1]

for i in active:

xi = float(np.clip(x[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

# Build batched circuits

circ\_qcaa = [build\_qcaa(x) for x in Xsub]

masks = [mask\_for\_row(x) for x in Xsub]

circ\_qcaaopt = [build\_qcaa\_opt(x, m) for x, m in zip(Xsub, masks)]

circ\_qcaa\_t = [transpile(c, backend=backend, optimization\_level=opt\_level) for c in circ\_qcaa]

circ\_qcaaopt\_t = [transpile(c, backend=backend, optimization\_level=opt\_level) for c in circ\_qcaaopt]

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

def P1\_from\_result(res, width, tag):

# Try quasi\_dists first

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

P1 = np.zeros((len(res.quasi\_dists), width), dtype=float)

for i, qdist in enumerate(res.quasi\_dists):

row = np.zeros(width)

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

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

for q in range(width):

if bits[q]=="1": row[q] += float(p)

P1[i] = row

# snapshot

try:

snap = {to\_bits\_any(k, width): int(float(v)\*shots) for k,v in dict(res.quasi\_dists[0]).items()}

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

json.dump(snap, f, indent=2)

except Exception: pass

return P1

# Fallback to counts

try:

results\_iter = list(res)

except TypeError:

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

if results\_iter is None: return None

def try\_counts(r):

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

P1 = np.zeros((len(results\_iter), width), dtype=float)

for i, r in enumerate(results\_iter):

counts = try\_counts(r)

if not counts: continue

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

row = np.zeros(width)

for bitstring, cnt in counts.items():

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

for q in range(width):

if bits[q]=="1": row[q] += int(cnt)

P1[i] = row/float(total)

# snapshot

try:

c0 = try\_counts(results\_iter[0])

if c0:

snap = {to\_bits\_any(k, d): int(v) for k,v in c0.items()}

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

json.dump(snap, f, indent=2)

except Exception: pass

return P1

sampler = Sampler(mode=backend)

# QCAA

job = sampler.run(circ\_qcaa\_t, shots=shots); log\_kv("job\_submitted", name=name, tag="qcaa", n=len(circ\_qcaa\_t), shots=shots)

res = job.result(); log\_kv("job\_completed", name=name, tag="qcaa")

P1\_qcaa = P1\_from\_result(res, d, "qcaa")

# QCAA-Opt

job2 = sampler.run(circ\_qcaaopt\_t, shots=shots); log\_kv("job\_submitted", name=name, tag="qcaa\_opt", n=len(circ\_qcaaopt\_t), shots=shots)

res2 = job2.result(); log\_kv("job\_completed", name=name, tag="qcaa\_opt")

P1\_opt = P1\_from\_result(res2, d, "qcaa\_opt")

err\_qcaa = recon\_error\_from\_P1(P1\_qcaa, Xsub) if P1\_qcaa is not None else None

err\_opt = recon\_error\_from\_P1(P1\_opt, Xsub) if P1\_opt is not None else None

# Save quick csvs

if err\_qcaa is not None:

pd.DataFrame({"idx": sel, "mse": err\_qcaa}).to\_csv(OUTDIR / f"hw\_errors\_{name}\_qcaa.csv", index=False)

if err\_opt is not None:

pd.DataFrame({"idx": sel, "mse": err\_opt}).to\_csv(OUTDIR / f"hw\_errors\_{name}\_qcaa\_opt.csv", index=False)

return {"P1\_qcaa": P1\_qcaa, "P1\_opt": P1\_opt, "err\_qcaa": err\_qcaa, "err\_opt": err\_opt, "indices": sel}

# ---------- 15 Analyses Driver ----------

ALL\_ROWS = []

def analysis\_core\_metrics(name, X, y, tag):

# Simulate full-set errors for both models

P1a = qcaa\_simulate\_P1(X); P1b = qcaa\_opt\_simulate\_P1(X)

err\_a = recon\_error\_from\_P1(P1a, X); err\_b = recon\_error\_from\_P1(P1b, X)

thr\_a = fit\_threshold(err\_a, y); thr\_b = fit\_threshold(err\_b, y)

m\_a = metrics\_from\_scores(err\_a, y, thr\_a)

m\_b = metrics\_from\_scores(err\_b, y, thr\_b)

row\_a = {"dataset": name, "model": "QCAA (Non-Opt.)", \*\*m\_a}

row\_b = {"dataset": name, "model": "QCAA (Optimized)", \*\*m\_b}

ALL\_ROWS.extend([row\_a, row\_b])

# Save curves

for model, scores, thr in [("qcaa", err\_a, thr\_a), ("qcaa\_opt", err\_b, thr\_b)]:

fpr, tpr, \_ = roc\_curve(y, -scores)

pd.DataFrame({"fpr":fpr,"tpr":tpr}).to\_csv(OUTDIR / f"roc\_{name}\_{model}.csv", index=False)

return err\_a, err\_b, thr\_a, thr\_b

def analysis\_noise(name, X, y):

res = []

for nl in [0.05, 0.10, 0.20, 0.30]:

noise = np.random.normal(0, nl, size=X.shape)

Xn = np.clip(X + noise, 0.0, 1.0)

for mdl, gen in [("QCAA (Non-Opt.)", qcaa\_simulate\_P1), ("QCAA (Optimized)", qcaa\_opt\_simulate\_P1)]:

P1 = gen(Xn); err = recon\_error\_from\_P1(P1, Xn); thr = fit\_threshold(err, y)

m = metrics\_from\_scores(err, y, thr)

res.append({"dataset":name,"noise":nl,"model":mdl,\*\*m})

df = pd.DataFrame(res); df.to\_csv(OUTDIR / f"noise\_{name}.csv", index=False)

return df

def analysis\_missing\_features(name, X, y):

res = []

d = X.shape[1]

drop\_counts = [1, 2, min(3, d-1)]

for k in drop\_counts:

cols = np.random.choice(d, size=k, replace=False)

Xmiss = X.copy(); Xmiss[:, cols] = 0.0

for mdl, gen in [("QCAA (Non-Opt.)", qcaa\_simulate\_P1), ("QCAA (Optimized)", qcaa\_opt\_simulate\_P1)]:

P1 = gen(Xmiss); err = recon\_error\_from\_P1(P1, Xmiss); thr = fit\_threshold(err, y)

m = metrics\_from\_scores(err, y, thr)

res.append({"dataset":name,"dropped":k,"model":mdl,\*\*m})

df = pd.DataFrame(res); df.to\_csv(OUTDIR / f"missingfeat\_{name}.csv", index=False)

return df

def analysis\_cross\_dataset():

# Train threshold on source, test on target (scores computed on target)

res = []

keys = list(DATASETS.keys())

for src, tgt in itertools.permutations(keys, 2):

Xs, ys = DATASETS[src]["X"], DATASETS[src]["y"]

Xt, yt = DATASETS[tgt]["X"], DATASETS[tgt]["y"]

for mdl, gen in [("QCAA (Non-Opt.)", qcaa\_simulate\_P1), ("QCAA (Optimized)", qcaa\_opt\_simulate\_P1)]:

P1\_src = gen(Xs); err\_src = recon\_error\_from\_P1(P1\_src, Xs); thr = fit\_threshold(err\_src, ys)

P1\_tgt = gen(Xt); err\_tgt = recon\_error\_from\_P1(P1\_tgt, Xt)

m = metrics\_from\_scores(err\_tgt, yt, thr)

res.append({"train":src,"test":tgt,"model":mdl,\*\*m})

df = pd.DataFrame(res); df.to\_csv(OUTDIR / f"cross\_dataset.csv", index=False)

return df

def analysis\_encoding\_compare(name, X, y):

# Simple encoding variants via surrogate knobs

res = []

configs = {

"angle": dict(depol=0.08, crosstalk=0.10, active\_frac=1.0, opt=False),

"amplitude": dict(depol=0.10, crosstalk=0.12, active\_frac=1.0, opt=False),

"hybrid": dict(depol=0.06, crosstalk=0.08, active\_frac=0.5, opt=True),

}

for enc, cfg in configs.items():

if cfg["opt"]:

P1 = qcaa\_opt\_simulate\_P1(X, depol=cfg["depol"], crosstalk=cfg["crosstalk"], active\_frac=cfg["active\_frac"])

model\_name = "QCAA (Optimized)"

else:

P1 = qcaa\_simulate\_P1(X, depol=cfg["depol"], crosstalk=cfg["crosstalk"])

model\_name = "QCAA (Non-Opt.)"

err = recon\_error\_from\_P1(P1, X); thr = fit\_threshold(err, y)

m = metrics\_from\_scores(err, y, thr)

res.append({"dataset":name,"encoding":enc,"model":model\_name,\*\*m})

df = pd.DataFrame(res); df.to\_csv(OUTDIR / f"encoding\_{name}.csv", index=False)

return df

def analysis\_feature\_importance\_opt(name, X):

# Average activation frequency under opt surrogate as importance proxy

n, d = X.shape

mu = X.mean(axis=0, keepdims=True)

z = np.abs(X - mu)

k = max(2, int(np.ceil(0.5\*d)))

idx\_sorted = np.argsort(-z, axis=1)

rows = np.arange(n)[:, None]

active\_cols = idx\_sorted[:, :k]

counts = np.zeros(d, dtype=int)

for r in range(n):

counts[active\_cols[r]] += 1

imp = counts / n

pd.DataFrame({"feature": np.arange(d), "importance": imp}).to\_csv(OUTDIR / f"feat\_importance\_{name}.csv", index=False)

return imp

def analysis\_feature\_sensitivity(name, X, y):

# Perturb one feature at a time

d = X.shape[1]; res=[]

for j in range(d):

Xp = X.copy(); Xp[:, j] = np.clip(Xp[:, j] + 0.05, 0.0, 1.0)

for mdl, gen in [("QCAA (Non-Opt.)", qcaa\_simulate\_P1), ("QCAA (Optimized)", qcaa\_opt\_simulate\_P1)]:

P1 = gen(Xp); err = recon\_error\_from\_P1(P1, Xp); thr = fit\_threshold(err, y)

m = metrics\_from\_scores(err, y, thr)

res.append({"dataset":name,"feature\_idx":j,"model":mdl,\*\*m})

df = pd.DataFrame(res); df.to\_csv(OUTDIR / f"feature\_sensitivity\_{name}.csv", index=False)

return df

def analysis\_threshold\_adaptation(name, X, y, batch\_size=8, alpha=0.35):

n = len(X); s1=int(0.6\*n); s2=int(0.8\*n)

X1, y1 = X[:s1], y[:s1]

X2, y2 = X[s1:s2], y[s1:s2]

X3, y3 = X[s2:], y[s2:]

def adapt(err1, err2, err3, y1, y2, y3, label):

thr = fit\_threshold(err1, y1); rec=[]

for (tag, E, Y) in [("s2", err2, y2), ("s3", err3, y3)]:

for i in range(0, len(E), batch\_size):

e\_b = E[i:i+batch\_size]; y\_b = Y[i:i+batch\_size]

acc\_before = ( (e\_b <= thr).astype(int) == y\_b ).mean()

thr\_batch = fit\_threshold(e\_b, y\_b)

thr = (1-alpha)\*thr + alpha\*thr\_batch

acc\_after = ( (e\_b <= thr).astype(int) == y\_b ).mean()

rec.append({"dataset":name,"model":label,"session":tag,"batch\_start":i,"acc\_before":acc\_before,"acc\_after":acc\_after,"thr":thr})

return rec

# compute errs

Pa1 = qcaa\_simulate\_P1(X1); Pb1 = qcaa\_opt\_simulate\_P1(X1)

Pa2 = qcaa\_simulate\_P1(X2); Pb2 = qcaa\_opt\_simulate\_P1(X2)

Pa3 = qcaa\_simulate\_P1(X3); Pb3 = qcaa\_opt\_simulate\_P1(X3)

Ea1 = recon\_error\_from\_P1(Pa1, X1); Eb1 = recon\_error\_from\_P1(Pb1, X1)

Ea2 = recon\_error\_from\_P1(Pa2, X2); Eb2 = recon\_error\_from\_P1(Pb2, X2)

Ea3 = recon\_error\_from\_P1(Pa3, X3); Eb3 = recon\_error\_from\_P1(Pb3, X3)

recs = adapt(Ea1,Ea2,Ea3,y1,y2,y3,"QCAA (Non-Opt.)") + adapt(Eb1,Eb2,Eb3,y1,y2,y3,"QCAA (Optimized)")

pd.DataFrame(recs).to\_csv(OUTDIR / f"adaptation\_{name}.csv", index=False)

return recs

def analysis\_training\_size(name, X, y):

sizes = [10,15,20,30,40,50,60,70,80,90,100]

res=[]

for s in sizes:

if s<100:

accs\_a, accs\_b = [], []

for seed in [0,1,2,3,4]:

sss = StratifiedShuffleSplit(n\_splits=1, train\_size=s/100.0, random\_state=seed)

(tr, te), = sss.split(X, y)

for mdl, gen, accs in [("A", qcaa\_simulate\_P1, accs\_a), ("B", qcaa\_opt\_simulate\_P1, accs\_b)]:

P1\_tr = gen(X[tr]); E\_tr = recon\_error\_from\_P1(P1\_tr, X[tr]); thr = fit\_threshold(E\_tr, y[tr])

P1\_te = gen(X[te]); E\_te = recon\_error\_from\_P1(P1\_te, X[te])

y\_pred = (E\_te <= thr).astype(int)

accs.append((y\_pred==y[te]).mean()\*100.0)

res.append({"dataset":name,"size":s,"model":"QCAA (Non-Opt.)","acc\_mean":np.mean(accs\_a),"acc\_std":np.std(accs\_a, ddof=1)})

res.append({"dataset":name,"size":s,"model":"QCAA (Optimized)","acc\_mean":np.mean(accs\_b),"acc\_std":np.std(accs\_b, ddof=1)})

else:

skf = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

accs\_a, accs\_b = [], []

for tr, te in skf.split(X,y):

for mdl, gen, accs in [("A", qcaa\_simulate\_P1, accs\_a), ("B", qcaa\_opt\_simulate\_P1, accs\_b)]:

P1\_tr = gen(X[tr]); E\_tr = recon\_error\_from\_P1(P1\_tr, X[tr]); thr = fit\_threshold(E\_tr, y[tr])

P1\_te = gen(X[te]); E\_te = recon\_error\_from\_P1(P1\_te, X[te]); y\_pred=(E\_te<=thr).astype(int)

accs.append((y\_pred==y[te]).mean()\*100.0)

res.append({"dataset":name,"size":100,"model":"QCAA (Non-Opt.)","acc\_mean":np.mean(accs\_a),"acc\_std":np.std(accs\_a, ddof=1)})

res.append({"dataset":name,"size":100,"model":"QCAA (Optimized)","acc\_mean":np.mean(accs\_b),"acc\_std":np.std(accs\_b, ddof=1)})

df = pd.DataFrame(res); df.to\_csv(OUTDIR / f"train\_size\_{name}.csv", index=False)

return df

# ---------- RUN ----------

summary\_rows = []

hw\_snapshots = {}

# 0) Minimal real-hardware run per dataset (fast anchor)

if QUANTUM\_OK:

for ds\_name in DATASETS:

X = DATASETS[ds\_name]["X"]; hw = run\_hardware\_snapshot(ds\_name, X, DATASETS[ds\_name]["features"], max\_samples=6, shots=512, opt\_level=1)

hw\_snapshots[ds\_name] = hw

if hw is not None: log\_kv("hardware\_anchor\_saved", dataset=ds\_name, n=len(hw["indices"]))

# 1–5) Core metrics, ROC/AUC/EER, confusion, distributions (per dataset)

for ds\_name in DATASETS:

X = DATASETS[ds\_name]["X"]; y = DATASETS[ds\_name]["y"]

errA, errB, thrA, thrB = analysis\_core\_metrics(ds\_name, X, y, "core")

# Confusion matrices & score distributions

for tag, E, thr in [("qcaa", errA, thrA), ("qcaa\_opt", errB, thrB)]:

y\_pred = (E <= thr).astype(int)

cm = confusion\_matrix(y, y\_pred)

pd.DataFrame(cm, index=["Impostor","Genuine"], columns=["Pred Imp","Pred Gen"]).to\_csv(OUTDIR / f"cm\_{ds\_name}\_{tag}.csv")

# Histogram plot

plt.figure()

plt.hist(E[y==1], bins=30, alpha=0.7, label="Genuine")

plt.hist(E[y==0], bins=30, alpha=0.7, label="Impostor")

plt.title(f"Score Dist — {ds\_name} — {tag}")

plt.xlabel("Reconstruction Error"); plt.ylabel("Count"); plt.legend(); plt.tight\_layout()

plt.savefig(OUTDIR / f"dist\_{ds\_name}\_{tag}.png"); plt.close()

# 6) Noise robustness per dataset

for ds\_name in DATASETS:

analysis\_noise(ds\_name, DATASETS[ds\_name]["X"], DATASETS[ds\_name]["y"])

# 7) Missing-feature robustness

for ds\_name in DATASETS:

analysis\_missing\_features(ds\_name, DATASETS[ds\_name]["X"], DATASETS[ds\_name]["y"])

# 8) Cross-dataset generalization

analysis\_cross\_dataset()

# 9) Encoding strategy comparison (simulated)

for ds\_name in DATASETS:

analysis\_encoding\_compare(ds\_name, DATASETS[ds\_name]["X"], DATASETS[ds\_name]["y"])

# 10) Feature importance via opt mask frequency

for ds\_name in DATASETS:

analysis\_feature\_importance\_opt(ds\_name, DATASETS[ds\_name]["X"])

# 11) Per-feature sensitivity

for ds\_name in DATASETS:

analysis\_feature\_sensitivity(ds\_name, DATASETS[ds\_name]["X"], DATASETS[ds\_name]["y"])

# 12) Threshold adaptation / concept drift

for ds\_name in DATASETS:

analysis\_threshold\_adaptation(ds\_name, DATASETS[ds\_name]["X"], DATASETS[ds\_name]["y"])

# 13) Accuracy vs training size

for ds\_name in DATASETS:

analysis\_training\_size(ds\_name, DATASETS[ds\_name]["X"], DATASETS[ds\_name]["y"])

# 14–15) Composite summary table from ALL\_ROWS

summary\_df = pd.DataFrame(ALL\_ROWS)

summary\_df.to\_csv(OUTDIR / "summary\_core\_metrics.csv", index=False)

# Quick aggregate

agg = summary\_df.groupby(["dataset","model"]).agg({"acc":"mean","auc":"mean","f1":"mean"}).reset\_index()

agg.to\_csv(OUTDIR / "summary\_aggregate.csv", index=False)

log\_kv("done", outdir=str(OUTDIR.resolve()))

print(f"\n✅ All analyses completed. Results saved to: {OUTDIR.resolve()}")