# === Real-HW Deep Insights (Brisbane) — Fast & Detailed ===

# New tests: McNemar, NRI/IDI, TOST, Bayesian bootstrap, Threshold stability,

# DET points, Decision Curve, Cost Curve, Calibration slope/intercept + HL,

# Wasserstein & Overlap, Shot-noise sensitivity.

#

# Hardware kept fast: class-balanced subsample + low shots + batching + caching.

# Outputs: log31/

import os, uuid, logging, math, warnings

from pathlib import Path

from datetime import datetime

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import roc\_curve

from sklearn.linear\_model import LogisticRegression

from scipy.stats import chi2, norm, rankdata, chisquare

from scipy.stats import wasserstein\_distance

warnings.filterwarnings("ignore", category=UserWarning)

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

from qiskit import QuantumCircuit, transpile

# ---------- Credentials / Config ----------

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"

# Hardware throttle

SHOTS = 512

BATCH\_SIZE = 32

ACTIVE\_FRAC = 0.50

SUBS\_POS = 20 # positives per dataset (adjust if dataset smaller)

SUBS\_NEG = 20 # negatives per dataset

# Bootstrap configs

N\_BOOT\_EER = 800 # for EER CI and stability (kept modest for speed)

N\_BB = 800 # Bayesian bootstrap resamples

# Data 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"

OUTDIR = Path("log31"); OUTDIR.mkdir(parents=True, exist\_ok=True)

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

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

logger.setLevel(logging.INFO)

fh = logging.FileHandler(OUTDIR / "deep\_insights.log", encoding="utf-8")

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

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

def log(msg, \*\*kw): logger.info(msg + (" | " + " ".join(f"{k}={v}" for k, v in kw.items()) if kw else ""))

# ---------- IBM helpers ----------

def ibm\_connect():

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

backend = service.backend(BACKEND\_NAME)

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

return backend

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

d = len(x\_vec)

qc = QuantumCircuit(d)

for i, val in enumerate(x\_vec):

v = float(np.clip(val, 1e-12, 1-1e-12))

theta = 2.0 \* np.arcsin(np.sqrt(v)) # sin^2(theta/2)=v

qc.ry(theta, i)

for i in range(d - 1):

qc.cx(i, i + 1)

qc.measure\_all()

return qc

def \_counts\_to\_p1(counts, d):

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

p1 = np.zeros(d, dtype=float)

for bitstr, cnt in counts.items():

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

if not set(s) <= {"0","1"}: continue

bits = s[::-1]

for q in range(min(d, len(bits))):

if bits[q] == "1":

p1[q] += cnt

p1 = p1 / total

return np.clip(p1, 1e-12, 1-1e-12)

def \_quasi\_to\_p1(qd, d):

p1 = np.zeros(d, dtype=float)

total = 0.0

for key, prob in dict(qd).items():

if isinstance(key, str):

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

bits = s[::-1].ljust(d, "0")[:d]

else:

bits = format(int(key), f"0{d}b")[::-1]

for q in range(d):

if bits[q] == "1":

p1[q] += float(prob)

total += float(prob)

if total > 1.0001: # counts-like

p1 = p1 / total

return np.clip(p1, 1e-12, 1-1e-12)

def \_extract\_p1\_list(result, d):

p1\_list = []

items = []

if hasattr(result, "results"):

items = list(result.results)

else:

try:

items = list(result)

except TypeError:

items = [result]

for item in items:

# 1) quasi

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

qd = item.quasi\_dists

if isinstance(qd, list):

for q in qd: p1\_list.append(\_quasi\_to\_p1(q, d))

else:

p1\_list.append(\_quasi\_to\_p1(qd, d))

continue

# 2) data.meas.get\_counts()

try:

if hasattr(item, "data") and hasattr(item.data, "meas") and hasattr(item.data.meas, "get\_counts"):

counts = item.data.meas.get\_counts()

if isinstance(counts, list):

for c in counts: p1\_list.append(\_counts\_to\_p1(c, d))

elif isinstance(counts, dict):

p1\_list.append(\_counts\_to\_p1(counts, d))

continue

except Exception:

pass

# 3) get\_counts()

try:

if hasattr(item, "get\_counts"):

c = item.get\_counts()

if isinstance(c, list):

for csub in c: p1\_list.append(\_counts\_to\_p1(csub, d))

elif isinstance(c, dict):

p1\_list.append(\_counts\_to\_p1(c, d))

continue

except Exception:

pass

# Last ditch: maybe result has quasi\_dists directly

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

qd = result.quasi\_dists

if isinstance(qd, list):

for q in qd: p1\_list.append(\_quasi\_to\_p1(q, d))

else:

p1\_list.append(\_quasi\_to\_p1(qd, d))

if not p1\_list:

raise RuntimeError("No quasi dists or counts found in SamplerV2 result.")

return p1\_list

def run\_hardware\_P1\_matrix(X, backend, shots=SHOTS, batch=BATCH\_SIZE):

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

n, d = X.shape

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

transpiled = [transpile(c, backend=backend, optimization\_level=2) for c in circuits]

sampler = Sampler(mode=backend)

P1 = np.zeros((n, d), dtype=float)

i = 0

while i < n:

j = min(n, i + batch)

job = sampler.run(transpiled[i:j], shots=shots)

res = job.result()

p1\_chunk = \_extract\_p1\_list(res, d)

if len(p1\_chunk) != (j - i):

if len(p1\_chunk) < (j - i):

raise RuntimeError("Sampler returned fewer results than circuits.")

p1\_chunk = p1\_chunk[:(j - i)]

for k, p in enumerate(p1\_chunk):

P1[i + k] = p

i = j

return P1

# ---------- Data / metrics ----------

def load\_dataset(path):

df = pd.read\_csv(path)

label\_col = next((c for c in ["label","y","target","Label","Target"] if c in df.columns), None)

if not label\_col:

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

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

if not feats:

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

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

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

Xn = MinMaxScaler().fit\_transform(X)

return Xn, y, feats

def class\_balanced\_subsample(X, y, pos\_target=SUBS\_POS, neg\_target=SUBS\_NEG, seed=7):

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

pos\_idx = np.where(y==1)[0]; neg\_idx = np.where(y==0)[0]

take\_pos = min(pos\_target, len(pos\_idx))

take\_neg = min(neg\_target, len(neg\_idx))

sel = np.concatenate([rng.choice(pos\_idx, take\_pos, replace=False),

rng.choice(neg\_idx, take\_neg, replace=False)])

sel.sort()

return X[sel], y[sel], sel

def apply\_active\_mask(X, active\_frac=ACTIVE\_FRAC):

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)

Xm = np.zeros\_like(X)

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

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

keep = idx\_sorted[:, :k]

Xm[rows, keep] = X[rows, keep]

return Xm

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 compute\_eer(scores, labels):

fpr, tpr, thr = roc\_curve(labels, -scores)

fnr = 1.0 - tpr

diff = fpr - fnr

idx = np.where(np.diff(np.sign(diff)) != 0)[0]

if len(idx) == 0:

j = int(np.argmin(np.abs(diff)))

return float(0.5\*(fpr[j]+fnr[j])), float(thr[j]), fpr, tpr

i = idx[0]

x0, x1 = fpr[i], fpr[i+1]

y0, y1 = diff[i], diff[i+1]

t = 0.0 if y1 == y0 else -y0/(y1 - y0)

eer = float(np.clip(x0 + t\*(x1-x0), 0.0, 1.0))

thr\_eer = float(thr[i] + t\*(thr[i+1]-thr[i]))

return eer, thr\_eer, fpr, tpr

def tpr\_at\_fpr(scores, labels, target\_fpr=0.01):

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

return float(np.interp(target\_fpr, fpr, tpr))

def logistic\_calibration(y, s):

# probs via Platt scaling (logistic on -scores)

X = (-s).reshape(-1,1)

clf = LogisticRegression(max\_iter=1000).fit(X, y)

p = clf.predict\_proba(X)[:,1]

# calibration slope/intercept from logistic of y on logit(p)

eps = 1e-8

logit\_p = np.log(np.clip(p,eps,1-eps)/(1-np.clip(p,eps,1-eps)))

lr = LogisticRegression(max\_iter=1000).fit(logit\_p.reshape(-1,1), y)

slope = float(lr.coef\_[0][0]); intercept = float(lr.intercept\_[0])

return p, slope, intercept

def hosmer\_lemeshow(y, p, bins=10):

# HL statistic using deciles of risk

edges = np.quantile(p, np.linspace(0,1,bins+1))

edges[0] -= 1e-12; edges[-1] += 1e-12

obs, exp = [], []

for i in range(bins):

mask = (p >= edges[i]) & (p < edges[i+1])

if not np.any(mask): continue

n = mask.sum()

o = y[mask].sum()

e = p[mask].sum()

obs.append(o); exp.append(e)

obs = np.array(obs, float); exp = np.array(exp, float)

if len(obs) < 5: # too few bins

return np.nan, np.nan

# Pearson chi-square with df = (bins-2)

chi = np.sum((obs - exp)\*\*2 / np.clip(exp\*(1 - (exp/np.clip(exp+1e-9,1e-9,None))), 1e-9, None))

df = max(len(obs)-2, 1)

pval = 1 - chi2.cdf(chi, df)

return float(chi), float(pval)

def mcnemar\_test(y\_true, y\_a, y\_b):

# y\_a, y\_b: binary predictions

disagree\_ab = np.logical\_and(y\_a==1, y\_b==0)

disagree\_ba = np.logical\_and(y\_a==0, y\_b==1)

b = int(disagree\_ab.sum()); c = int(disagree\_ba.sum())

# continuity-corrected McNemar (Edwards)

stat = (abs(b - c) - 1)\*\*2 / max(b + c, 1)

p = 1 - chi2.cdf(stat, 1)

return float(stat), float(p), b, c

def nri\_idi(y, p\_a, p\_b):

# NRI/IDI (binary). p\_a->old, p\_b->new

y = np.asarray(y)

up\_event = ((p\_b > p\_a) & (y==1)).mean()

down\_event = ((p\_b < p\_a) & (y==1)).mean()

up\_nonevt = ((p\_b < p\_a) & (y==0)).mean()

down\_nonevt= ((p\_b > p\_a) & (y==0)).mean()

nri = (up\_event - down\_event) - (up\_nonevt - down\_nonevt)

idi = (p\_b[y==1].mean() - p\_a[y==1].mean()) - (p\_b[y==0].mean() - p\_a[y==0].mean())

return float(nri), float(idi)

def tost\_equivalence(diff\_samples, low=-0.5, high=0.5):

# Two one-sided tests on mean(diff) in percentage points

d = np.asarray(diff\_samples)

m = d.mean(); s = d.std(ddof=1); n = len(d)

if n < 2 or s == 0: return np.nan, np.nan, float(m)

se = s/np.sqrt(n)

z1 = (m - low)/se

z2 = (high - m)/se

p1 = 1 - norm.cdf(z1)

p2 = 1 - norm.cdf(z2)

p\_equiv = max(p1, p2) # reject non-equivalence if both p<alpha

return float(p1), float(p2), float(m)

def bayesian\_bootstrap\_mean\_diff(a, b, n=N\_BB, seed=11):

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

a = np.asarray(a); b = np.asarray(b)

diffs = np.zeros(n)

for i in range(n):

w = rng.dirichlet(np.ones(len(a)))

diffs[i] = (w @ a) - (w @ b)

mean = float(np.mean(diffs))

lo, hi = np.percentile(diffs, [2.5, 97.5])

return mean, (float(lo), float(hi)), diffs

def decision\_curve(y, p, thresholds=np.linspace(0.01,0.99,99)):

# Net Benefit = TP/N - FP/N \* (t/(1-t))

y = np.asarray(y); p = np.asarray(p)

N = len(y)

nben = []

for t in thresholds:

pred = (p >= t).astype(int)

TP = np.logical\_and(pred==1, y==1).sum()

FP = np.logical\_and(pred==1, y==0).sum()

nb = (TP/N) - (FP/N) \* (t/(1-t))

nben.append(nb)

return thresholds, np.array(nben)

def cost\_curve(scores, y, cost\_fa=1.0, cost\_fr=1.0):

# For every threshold, expected cost = cost\_fa\*FAR\*P(neg) + cost\_fr\*FRR\*P(pos)

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

fnr = 1 - tpr

p\_pos = y.mean(); p\_neg = 1 - p\_pos

cost = cost\_fa \* fpr \* p\_neg + cost\_fr \* fnr \* p\_pos

return thr, cost

def overlap\_coefficient(a, b, bins=50):

hist\_a, edges = np.histogram(a, bins=bins, density=True)

hist\_b, \_ = np.histogram(b, bins=edges, density=True)

return float(np.sum(np.minimum(hist\_a, hist\_b) \* np.diff(edges)))

def shot\_noise\_variance(P1, shots=SHOTS, nsamp=200, seed=5):

# Binomial resample of P1 to assess measurement variance on errors

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

n, d = P1.shape

err\_vars = np.zeros(n)

for i in range(n):

p = P1[i]

xhats = []

for \_ in range(nsamp):

counts = rng.binomial(shots, p) / max(shots,1)

xhat = np.arcsin(np.sqrt(np.clip(counts,1e-12,1-1e-12)))

xhats.append(xhat)

xhats = np.stack(xhats, axis=0) # nsamp x d

err = ((xhats - xhats.mean(axis=0))\*\*2).mean(axis=(1,1)) if xhats.ndim==3 else ((xhats - xhats.mean(axis=0))\*\*2).mean(axis=1)

err\_vars[i] = np.var(err)

return float(np.mean(err\_vars))

# ---------- Run ----------

backend = ibm\_connect()

datasets = {

"gait": PATH\_GAIT,

"fixed": PATH\_FTXT,

"free": PATH\_FREET,

}

summary\_rows = []

for name, path in datasets.items():

Xfull, yfull, feats = load\_dataset(path)

X, y, sel = class\_balanced\_subsample(Xfull, yfull, SUBS\_POS, SUBS\_NEG, seed=13)

n, d = X.shape

log("dataset\_loaded\_subsample", name=name, path=path, n=n, d=d, pos=int(y.sum()), neg=int((1-y).sum()))

p1\_q\_path = OUTDIR / f"{name}\_P1\_qcaa\_fast.csv"

p1\_o\_path = OUTDIR / f"{name}\_P1\_opt\_fast.csv"

# QCAA (non-opt)

if p1\_q\_path.exists():

P1\_q = pd.read\_csv(p1\_q\_path).to\_numpy(dtype=float)

print(f"✅ Loaded cached P1 (QCAA) for {name}")

else:

print(f"⏳ HW run QCAA for {name} (n={n}, d={d}, shots={SHOTS})")

P1\_q = run\_hardware\_P1\_matrix(X, backend, shots=SHOTS, batch=BATCH\_SIZE)

pd.DataFrame(P1\_q, columns=[f"q{j}" for j in range(d)]).to\_csv(p1\_q\_path, index=False)

# QCAA-Optimized

X\_opt = apply\_active\_mask(X, ACTIVE\_FRAC)

if p1\_o\_path.exists():

P1\_o = pd.read\_csv(p1\_o\_path).to\_numpy(dtype=float)

print(f"✅ Loaded cached P1 (QCAA-Opt) for {name}")

else:

print(f"⏳ HW run QCAA-Opt for {name} (n={n}, d={d}, shots={SHOTS}, active≈{int(math.ceil(ACTIVE\_FRAC\*d))})")

P1\_o = run\_hardware\_P1\_matrix(X\_opt, backend, shots=SHOTS, batch=BATCH\_SIZE)

pd.DataFrame(P1\_o, columns=[f"q{j}" for j in range(d)]).to\_csv(p1\_o\_path, index=False)

# Errors & base thresholds

err\_q = recon\_error\_from\_P1(P1\_q, X)

err\_o = recon\_error\_from\_P1(P1\_o, X\_opt)

eer\_q, thr\_q, fpr\_q, tpr\_q = compute\_eer(err\_q, y)

eer\_o, thr\_o, fpr\_o, tpr\_o = compute\_eer(err\_o, y)

# Low-FPR diagnostics

tpr01\_q = tpr\_at\_fpr(err\_q, y, 0.001)

tpr1\_q = tpr\_at\_fpr(err\_q, y, 0.01)

tpr5\_q = tpr\_at\_fpr(err\_q, y, 0.05)

tpr01\_o = tpr\_at\_fpr(err\_o, y, 0.001)

tpr1\_o = tpr\_at\_fpr(err\_o, y, 0.01)

tpr5\_o = tpr\_at\_fpr(err\_o, y, 0.05)

# McNemar (paired at each model’s own EER threshold)

yhat\_q = (-err\_q >= thr\_q).astype(int)

yhat\_o = (-err\_o >= thr\_o).astype(int)

mc\_stat, mc\_p, mc\_b, mc\_c = mcnemar\_test(y, yhat\_q, yhat\_o)

# Calibration & HL

p\_q, slope\_q, int\_q = logistic\_calibration(y, err\_q)

p\_o, slope\_o, int\_o = logistic\_calibration(y, err\_o)

hl\_chi\_q, hl\_p\_q = hosmer\_lemeshow(y, p\_q, bins=10)

hl\_chi\_o, hl\_p\_o = hosmer\_lemeshow(y, p\_o, bins=10)

# NRI / IDI

nri, idi = nri\_idi(y, p\_q, p\_o)

# Bootstrap EER distributions + threshold stability

rng = np.random.default\_rng(17)

e\_q\_boot, e\_o\_boot, tau\_q, tau\_o = np.zeros(N\_BOOT\_EER), np.zeros(N\_BOOT\_EER), np.zeros(N\_BOOT\_EER), np.zeros(N\_BOOT\_EER)

for b in range(N\_BOOT\_EER):

idx = rng.integers(0, n, size=n)

eq, tq, \_, \_ = compute\_eer(err\_q[idx], y[idx])

eo, to, \_, \_ = compute\_eer(err\_o[idx], y[idx])

e\_q\_boot[b] = eq \* 100.0; e\_o\_boot[b] = eo \* 100.0

tau\_q[b] = tq; tau\_o[b] = to

# CI

def ci(x): return (float(np.mean(x)), float(np.std(x, ddof=1)), tuple(np.percentile(x, [2.5,97.5])))

mq, sdq, ciq = ci(e\_q\_boot); mo, sdo, cio = ci(e\_o\_boot)

tau\_q\_sd, tau\_o\_sd = float(np.std(tau\_q, ddof=1)), float(np.std(tau\_o, ddof=1))

tau\_q\_iqr, tau\_o\_iqr = float(np.subtract(\*np.percentile(tau\_q,[75,25]))), float(np.subtract(\*np.percentile(tau\_o,[75,25])))

# TOST on difference (Opt - QCAA) in pp, with ±0.5pp equivalence margin

diff\_pp = e\_o\_boot - e\_q\_boot

p1\_tost, p2\_tost, mean\_diff = tost\_equivalence(diff\_pp, low=-0.5, high=0.5)

# Bayesian bootstrap of mean difference

bb\_mean, bb\_ci, \_ = bayesian\_bootstrap\_mean\_diff(e\_o\_boot, e\_q\_boot, n=N\_BB, seed=19)

# Distances between EER distributions

wdist = float(wasserstein\_distance(e\_q\_boot, e\_o\_boot))

overlap = overlap\_coefficient(e\_q\_boot, e\_o\_boot, bins=30)

# Decision curve (using calibrated probs)

th, nb\_q = decision\_curve(y, p\_q)

\_, nb\_o = decision\_curve(y, p\_o)

pd.DataFrame({"threshold": th, "NB\_QCAA": nb\_q, "NB\_OPT": nb\_o}).to\_csv(OUTDIR / f"{name}\_decision\_curve.csv", index=False)

# Cost curve

thr\_qc, cost\_q = cost\_curve(err\_q, y, cost\_fa=1.0, cost\_fr=1.0)

thr\_oc, cost\_o = cost\_curve(err\_o, y, cost\_fa=1.0, cost\_fr=1.0)

pd.DataFrame({"thr": thr\_qc, "cost\_QCAA": cost\_q}).to\_csv(OUTDIR / f"{name}\_costcurve\_qcaa.csv", index=False)

pd.DataFrame({"thr": thr\_oc, "cost\_OPT": cost\_o}).to\_csv(OUTDIR / f"{name}\_costcurve\_opt.csv", index=False)

# Shot-noise sensitivity (fast offline binomial resample on P1)

sn\_var\_q = shot\_noise\_variance(P1\_q, shots=SHOTS, nsamp=120, seed=23)

sn\_var\_o = shot\_noise\_variance(P1\_o, shots=SHOTS, nsamp=120, seed=29)

# Save per-sample

pd.DataFrame({

"sel\_index": sel, "label": y, "err\_qcaa": err\_q, "err\_opt": err\_o,

"prob\_qcaa": p\_q, "prob\_opt": p\_o,

"pred\_qcaa@eer": yhat\_q, "pred\_opt@eer": yhat\_o

}).to\_csv(OUTDIR / f"{name}\_per\_sample\_fast.csv", index=False)

# Save EER boots + thresholds

pd.DataFrame({"EER\_QCAA(%)": e\_q\_boot, "EER\_OPT(%)": e\_o\_boot,

"tau\_QCAA": tau\_q, "tau\_OPT": tau\_o}).to\_csv(OUTDIR / f"{name}\_eer\_bootstrap\_fast.csv", index=False)

summary\_rows.append({

"dataset": name, "n": n, "d": d,

"EER\_QCAA\_point(%)": round(eer\_q\*100.0,3), "EER\_OPT\_point(%)": round(eer\_o\*100.0,3),

"EER\_QCAA\_mean(%)": round(mq,3), "EER\_OPT\_mean(%)": round(mo,3),

"EER\_QCAA\_SD(%)": round(sdq,3), "EER\_OPT\_SD(%)": round(sdo,3),

"EER\_QCAA\_95%CI": f"[{ciq[0]:.3f},{ciq[1]:.3f}]", "EER\_OPT\_95%CI": f"[{cio[0]:.3f},{cio[1]:.3f}]",

"tau\_QCAA\_SD": round(tau\_q\_sd,6), "tau\_OPT\_SD": round(tau\_o\_sd,6),

"tau\_QCAA\_IQR": round(tau\_q\_iqr,6), "tau\_OPT\_IQR": round(tau\_o\_iqr,6),

"TPR@0.1%FPR\_QCAA": round(tpr01\_q,3), "TPR@1%FPR\_QCAA": round(tpr1\_q,3), "TPR@5%FPR\_QCAA": round(tpr5\_q,3),

"TPR@0.1%FPR\_OPT": round(tpr01\_o,3), "TPR@1%FPR\_OPT": round(tpr1\_o,3), "TPR@5%FPR\_OPT": round(tpr5\_o,3),

"McNemar\_stat": round(mc\_stat,4), "McNemar\_p": round(mc\_p,6), "b(1,0)": mc\_b, "c(0,1)": mc\_c,

"Cal\_slope\_QCAA": round(slope\_q,4), "Cal\_int\_QCAA": round(int\_q,4),

"Cal\_slope\_OPT": round(slope\_o,4), "Cal\_int\_OPT": round(int\_o,4),

"HL\_chi\_QCAA": round(hl\_chi\_q,4), "HL\_p\_QCAA": round(hl\_p\_q,6),

"HL\_chi\_OPT": round(hl\_chi\_o,4), "HL\_p\_OPT": round(hl\_p\_o,6),

"NRI": round(nri,4), "IDI": round(idi,4),

"TOST\_p1": round(p1\_tost,6), "TOST\_p2": round(p2\_tost,6), "MeanDiff\_pp(OPT-QCAA)": round(mean\_diff,3),

"BB\_diff\_mean": round(bb\_mean,3), "BB\_diff\_95%CrI": f"[{bb\_ci[0]:.3f},{bb\_ci[1]:.3f}]",

"Wasserstein\_EER": round(wdist,4), "OverlapCoeff\_EER": round(overlap,4),

"ShotNoiseVar\_QCAA": round(sn\_var\_q,8), "ShotNoiseVar\_OPT": round(sn\_var\_o,8)

})

# Save summary & a brief report

summary\_df = pd.DataFrame(summary\_rows)

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

with open(OUTDIR / "analysis\_report.txt", "w", encoding="utf-8") as f:

f.write("Real-HW Deep Insights Report (Fast Mode)\n")

f.write(f"Backend: {BACKEND\_NAME}, shots={SHOTS}, batch={BATCH\_SIZE}\n")

f.write("Metrics: McNemar, NRI/IDI, TOST, Bayesian bootstrap, Threshold stability, DET, DCA, Cost curve, Calibration (slope/intercept + HL), Wasserstein, Overlap, Shot-noise variance.\n\n")

f.write(summary\_df.to\_string(index=False))

print("\n=== Deep Insights (Fast) — Summary ===")

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

print(f"\nSaved outputs to: {OUTDIR.resolve()}")