# === Threshold Sensitivity Analysis (Angle / Amplitude / Hybrid, Single Job) — ibm\_brisbane ===

# Output dir: log22/

# What you get:

# - Per-encoding score vectors (S1 and All Sessions)

# - Threshold sweeps (FAR, FRR, FPR, TPR) over a dense grid

# - EER point, Youden-J optimum, and thresholds matching FAR targets (5%, 1%, 0.1%)

# - All with ONE hardware job to keep runtime low

#

# Artifacts:

# - log22/threshold\_sweep\_ANGLE\_S1.csv

# - log22/threshold\_sweep\_ANGLE\_ALL.csv

# - log22/threshold\_sweep\_AMPLITUDE\_S1.csv

# - log22/threshold\_sweep\_AMPLITUDE\_ALL.csv

# - log22/threshold\_sweep\_HYBRID\_S1.csv

# - log22/threshold\_sweep\_HYBRID\_ALL.csv

# - log22/summary.json

# - log22/run.log, log22/run.jsonl

# - log22/snap\_angle.json, log22/snap\_amp.json, log22/snap\_hyb.json

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

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 qiskit\_ibm\_runtime import QiskitRuntimeService, SamplerV2 as Sampler

from qiskit import QuantumCircuit, transpile

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

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

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

LOG\_DIR = Path("log22"); 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 / "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 / "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)

# ---------------- Credentials (UPDATED) ----------------

IBM\_TOKEN = "isTxH69BGxixH7QohOX\_F8Zxm9fvMY4FP4ZET6F9xjTZ"

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

# ---------------- Config ----------------

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

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

SHOTS = 1024 # fidelity

OPT\_LEVEL = 0 # fastest transpile

AMP\_Q\_MAX = 3 # ≤8 amplitude features

# Create drifted sessions (same as earlier)

DRIFT\_S2 = (0.03, 1.03)

DRIFT\_S3 = (0.06, 1.06)

# FAR targets for deployment suggestions

FAR\_TARGETS = [0.05, 0.01, 0.001] # 5%, 1%, 0.1%

np.random.seed(42)

log\_kv("config\_set", data\_path=DATA\_PATH, shots=SHOTS, opt\_level=OPT\_LEVEL, amp\_q\_max=AMP\_Q\_MAX)

# ---------------- 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)

# ---------------- 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 (expected one of label/y/target/Label/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()

# chronological if present

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\_raw, y = X\_raw[order], y[order]

# normalize to [0,1]

scaler = MinMaxScaler()

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

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

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

feature\_cols=",".join(feature\_cols))

# ---------------- Sessions ----------------

n = len(X)

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

idx\_s1, idx\_s2, idx\_s3 = np.arange(0, s1\_end), np.arange(s1\_end, s2\_end), np.arange(s2\_end, n)

def driftify(Xpart, mean\_shift, scale): return np.clip(Xpart \* scale + mean\_shift, 0.0, 1.0)

X\_s1, y\_s1 = X[idx\_s1], y[idx\_s1]

X\_s2, y\_s2 = driftify(X[idx\_s2], \*DRIFT\_S2), y[idx\_s2]

X\_s3, y\_s3 = driftify(X[idx\_s3], \*DRIFT\_S3), y[idx\_s3]

labels\_by\_sess = {"s1": y\_s1, "s2": y\_s2, "s3": y\_s3}

# ---------------- Utils ----------------

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\_quasi\_list(quasi\_list, logical\_width):

N = len(quasi\_list)

P1 = np.zeros((N, logical\_width), dtype=float)

for i, qdist in enumerate(quasi\_list):

row = np.zeros(logical\_width)

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

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

for q in range(logical\_width):

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

P1[i] = row

return P1

# ---------------- Threshold math ----------------

def compute\_eer(scores, labels):

# scores: smaller => more genuine (error), accept iff score <= thr

fpr, tpr, thr = roc\_curve(labels, -scores) # use -scores so "higher is positive" for ROC

fnr = 1 - tpr

idx = int(np.argmin(np.abs(fpr - fnr)))

eer = float((fpr[idx] + fnr[idx]) / 2.0)

thr\_at\_eer = float(thr[idx])

# Convert ROC threshold (on -scores) back to "score threshold"

score\_thr\_eer = -thr\_at\_eer

return eer, score\_thr\_eer

def sweep\_thresholds(scores, labels, grid\_size=501):

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

labels = np.asarray(labels, dtype=int)

if len(scores) == 0: return pd.DataFrame(columns=["threshold","FAR","FRR","TPR","FPR","ACC","POS","NEG"])

lo, hi = float(scores.min()), float(scores.max())

if lo == hi:

thrs = np.array([lo])

else:

thrs = np.linspace(lo, hi, grid\_size)

pos = (labels == 1); neg = ~pos

n\_pos = max(int(pos.sum()), 1)

n\_neg = max(int(neg.sum()), 1)

rows = []

for t in thrs:

# accept if score <= t

acc\_pos = np.sum((scores[pos] <= t))

acc\_neg = np.sum((scores[neg] <= t))

# FAR: impostors accepted / impostors

FAR = float(acc\_neg / n\_neg)

# FRR: genuines rejected / genuines

FRR = float((n\_pos - acc\_pos) / n\_pos)

# For completeness, TPR = 1 - FRR; FPR = FAR

TPR = 1.0 - FRR

FPR = FAR

ACC = float((acc\_pos + (n\_neg - acc\_neg)) / (n\_pos + n\_neg))

rows.append({"threshold": float(t), "FAR": FAR, "FRR": FRR, "TPR": TPR, "FPR": FPR, "ACC": ACC,

"POS": int(n\_pos), "NEG": int(n\_neg)})

return pd.DataFrame(rows)

def threshold\_for\_far(scores, labels, far\_target):

df = sweep\_thresholds(scores, labels, grid\_size=2001) # dense grid for interpolation

# pick threshold with FAR closest to target (tie-breaker: lower FAR)

df["diff"] = (df["FAR"] - far\_target).abs()

j = int(df["diff"].idxmin())

return float(df.loc[j, "threshold"]), float(df.loc[j, "FAR"]), float(df.loc[j, "FRR"])

# ---------------- Encoding circuit builders ----------------

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

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

return qc

def build\_amplitude\_circuit(x\_vec, q\_max=AMP\_Q\_MAX):

m = len(x\_vec)

if m <= 1:

q = 1; size = 1

else:

q = min(q\_max, int(math.floor(math.log2(m))))

q = max(q, 1)

size = min(2\*\*q, m)

sub = np.array(x\_vec[:size], dtype=float)

norm = np.linalg.norm(sub)

sub = (sub / norm) if norm != 0 else (np.ones\_like(sub)/math.sqrt(size))

amps = np.zeros(2\*\*q, dtype=complex); amps[:size] = sub.astype(complex)

qc = QuantumCircuit(q, q)

if q > 0: qc.initialize(amps.tolist(), list(range(q)))

qc.measure(range(q), range(q))

return qc, q, size

def build\_hybrid\_circuit(x\_vec, q\_max=AMP\_Q\_MAX):

m = len(x\_vec)

if m <= 1:

q = 1; size = 1

else:

q = min(q\_max, int(math.floor(math.log2(m))))

q = max(q, 1)

size = min(2\*\*q, m)

sub = np.array(x\_vec[:size], dtype=float)

norm = np.linalg.norm(sub)

sub = (sub / norm) if norm != 0 else (np.ones\_like(sub)/math.sqrt(size))

amps = np.zeros(2\*\*q, dtype=complex); amps[:size] = sub.astype(complex)

rest = x\_vec[size:]; d\_angle = len(rest)

qc = QuantumCircuit(q + d\_angle, q + d\_angle)

if q > 0: qc.initialize(amps.tolist(), list(range(q)))

for i in range(d\_angle):

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

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

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

qc.measure(range(q + d\_angle), range(q + d\_angle))

return qc, q, size, d\_angle

# ---------------- Build all circuits (single job) ----------------

def batch\_angle(X1, X2, X3):

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

for tag, Xref in [("s1",X1),("s2",X2),("s3",X3)]:

cs=[build\_angle\_circuit(x) for x in Xref]

circuits.extend(cs); end=start+len(cs); idx\_map[tag]=(start,end,Xref); start=end

logical\_width = X1.shape[1] if circuits else 1

return circuits, idx\_map, logical\_width

def batch\_amplitude(X1, X2, X3):

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

for tag, Xref in [("s1",X1),("s2",X2),("s3",X3)]:

cs=[]; meta=[]

for x in Xref:

c, q, size = build\_amplitude\_circuit(x, q\_max=AMP\_Q\_MAX)

cs.append(c); meta.append((q,size))

circuits.extend(cs); end=start+len(cs); idx\_map[tag]=(start,end,Xref,meta); start=end

return circuits, idx\_map

def batch\_hybrid(X1, X2, X3):

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

for tag, Xref in [("s1",X1),("s2",X2),("s3",X3)]:

cs=[]; meta=[]

for x in Xref:

c, q, size, d\_ang = build\_hybrid\_circuit(x, q\_max=AMP\_Q\_MAX)

cs.append(c); meta.append((q,size,d\_ang))

circuits.extend(cs); end=start+len(cs); idx\_map[tag]=(start,end,Xref,meta); start=end

return circuits, idx\_map

# ---------------- Run one job ----------------

sampler = Sampler(mode=backend)

angle\_circs, angle\_map\_local, angle\_w = batch\_angle(X\_s1, X\_s2, X\_s3)

amp\_circs, amp\_map\_local = batch\_amplitude(X\_s1, X\_s2, X\_s3)

hyb\_circs, hyb\_map\_local = batch\_hybrid(X\_s1, X\_s2, X\_s3)

n\_angle = len(angle\_circs); n\_amp = len(amp\_circs); n\_hyb = len(hyb\_circs)

def offset\_map(local\_map, base):

adj = {}

for tag,(s,e,\*rest) in local\_map.items():

adj[tag] = (s+base, e+base, \*rest)

return adj

angle\_map = offset\_map(angle\_map\_local, 0)

amp\_map = offset\_map(amp\_map\_local, n\_angle)

hyb\_map = offset\_map(hyb\_map\_local, n\_angle + n\_amp)

all\_circs = angle\_circs + amp\_circs + hyb\_circs

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

job = sampler.run(all\_circs\_t, shots=SHOTS)

jid = job.job\_id(); log\_kv("job\_submitted", model="ENC\_ALL\_SINGLE\_JOB", n\_circuits=len(all\_circs\_t), job\_id=jid)

res = job.result(); log\_kv("job\_completed", model="ENC\_ALL\_SINGLE\_JOB", job\_id=jid)

# quasi list (dict per circuit)

quasi\_all = getattr(res, "quasi\_dists", None)

if quasi\_all is None:

try: results\_iter = list(res)

except TypeError: results\_iter = getattr(res, "results", [])

quasi\_all = []

for r in results\_iter:

qd = {}

got = None

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:

got = getter(r)

if got: break

except Exception:

pass

got = dict(got or {})

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

for k,v in got.items(): qd[k] = v/total

quasi\_all.append(qd)

# snapshots for sanity

try:

with open(LOG\_DIR/"snap\_angle.json","w",encoding="utf-8") as f: json.dump(quasi\_all[0], f, indent=2)

with open(LOG\_DIR/"snap\_amp.json","w",encoding="utf-8") as f: json.dump(quasi\_all[n\_angle], f, indent=2)

with open(LOG\_DIR/"snap\_hyb.json","w",encoding="utf-8") as f: json.dump(quasi\_all[n\_angle+n\_amp], f, indent=2)

log\_kv("snapshots\_saved", angle="snap\_angle.json", amp="snap\_amp.json", hyb="snap\_hyb.json")

except Exception:

pass

# ---------------- Reconstruct scores per encoding ----------------

def slice\_quasi(quasi, s, e): return quasi[s:e]

# ANGLE

angle\_quasi = slice\_quasi(quasi\_all, 0, n\_angle)

P1\_angle\_all = P1\_from\_quasi\_list(angle\_quasi, angle\_w)

def score\_angle\_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 score\_amplitude(qdist, q, size, X\_row):

# normalize target subvector

Xn = np.array(X\_row[0, :size], dtype=float)

norm = np.linalg.norm(Xn)

Xn = (Xn/norm) if norm!=0 else (np.ones\_like(Xn)/math.sqrt(len(Xn))) if size>0 else Xn

probs = np.zeros(2\*\*q, dtype=float) if q>0 else np.array([1.0], dtype=float)

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

idx = int(to\_bits\_any(k, q), 2) if q>0 else 0

probs[idx] += float(prob)

return float(np.mean((Xn - probs[:size])\*\*2)) if size>0 else float(0.0)

def score\_hybrid(qdist, q, size, d\_ang, X\_row):

# amplitude part

probs = np.zeros(2\*\*q, dtype=float) if q>0 else np.array([1.0], dtype=float)

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

bits = to\_bits\_any(k, q + d\_ang)

idx = int(bits[:q], 2) if q>0 else 0

probs[idx] += float(prob)

amp\_err = 0.0

if size>0:

Xn = np.array(X\_row[0, :size], dtype=float)

norm = np.linalg.norm(Xn); Xn = (Xn/norm) if norm!=0 else (np.ones\_like(Xn)/math.sqrt(len(Xn)))

amp\_err = float(np.mean((Xn - probs[:size])\*\*2))

# angle part

ang\_err = 0.0

if d\_ang>0:

P1\_ang = np.zeros(d\_ang, dtype=float)

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

bits = to\_bits\_any(k, q + d\_ang)[::-1]

for aq in range(d\_ang):

if bits[aq] == "1": P1\_ang[aq] += float(prob)

rest = X\_row[0, size:size+d\_ang]

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

ang\_err = float(np.mean((rest - x\_hat)\*\*2))

tot = max(size + d\_ang, 1)

return (size\*amp\_err + d\_ang\*ang\_err)/tot

def gather\_angle\_scores(P1\_all, idx\_map):

out = {}

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

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

return out

def gather\_amp\_scores(quasi, idx\_map):

out = {}

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

qd = quasi[s:e]; err = np.zeros(len(qd), dtype=float)

for i,(q,size) in enumerate(meta):

err[i] = score\_amplitude(qd[i], q, size, Xref[i:i+1,:size])

out[tag] = err

return out

def gather\_hyb\_scores(quasi, idx\_map):

out = {}

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

qd = quasi[s:e]; err = np.zeros(len(qd), dtype=float)

for i,(q,size,d\_ang) in enumerate(meta):

err[i] = score\_hybrid(qd[i], q, size, d\_ang, Xref[i:i+1])

out[tag] = err

return out

scores\_angle = gather\_angle\_scores(P1\_angle\_all, angle\_map)

scores\_amp = gather\_amp\_scores(quasi\_all, amp\_map)

scores\_hyb = gather\_hyb\_scores(quasi\_all, hyb\_map)

# S1 and ALL score/label packs

def pack(scope, scores\_by\_sess):

if scope=="S1":

return scores\_by\_sess["s1"], labels\_by\_sess["s1"]

all\_scores = np.concatenate([scores\_by\_sess["s1"], scores\_by\_sess["s2"], scores\_by\_sess["s3"]], axis=0)

all\_labels = np.concatenate([labels\_by\_sess["s1"], labels\_by\_sess["s2"], labels\_by\_sess["s3"]], axis=0)

return all\_scores, all\_labels

# ---------------- Threshold sweeps & recommendations ----------------

def analyze\_encoding(name, scores\_by\_sess):

out = {}

for scope in ("S1","ALL"):

scores, labels = pack(scope, scores\_by\_sess)

# EER

eer, thr\_eer = compute\_eer(scores, labels)

# Youden-J optimum threshold (maximize TPR-FPR)

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

J = tpr - fpr

j\_idx = int(np.argmax(J))

thr\_j = float(-thr[j\_idx]) # convert back to score threshold

# FAR targets

far\_recs = []

for ft in FAR\_TARGETS:

t\_star, far\_star, frr\_star = threshold\_for\_far(scores, labels, ft)

far\_recs.append({"target\_FAR": ft, "thr": t\_star, "achieved\_FAR": far\_star, "FRR": frr\_star})

# full sweep

sweep\_df = sweep\_thresholds(scores, labels, grid\_size=801)

sweep\_path = LOG\_DIR / f"threshold\_sweep\_{name}\_{scope}.csv"

sweep\_df.to\_csv(sweep\_path, index=False)

out[scope] = {

"EER": float(eer), "thr\_at\_EER": float(thr\_eer),

"thr\_at\_YoudenJ": float(thr\_j),

"FAR\_targets": far\_recs,

"sweep\_csv": str(sweep\_path.resolve())

}

log\_kv("threshold\_sweep\_saved", encoding=name, scope=scope, path=str(sweep\_path.resolve()))

return out

summary\_blocks = {

"ANGLE": analyze\_encoding("ANGLE", scores\_angle),

"AMPLITUDE": analyze\_encoding("AMPLITUDE", scores\_amp),

"HYBRID": analyze\_encoding("HYBRID", scores\_hyb),

}

# ---------------- Summary JSON (safe types) ----------------

def \_to\_builtin(x):

if isinstance(x, dict): return {k:\_to\_builtin(v) for k,v in x.items()}

if isinstance(x, list): return [\_to\_builtin(v) for v in x]

if isinstance(x, tuple): return tuple(\_to\_builtin(v) for v in x)

if isinstance(x, np.ndarray): return x.tolist()

if isinstance(x, (np.integer,)): return int(x)

if isinstance(x, (np.floating,)): return float(x)

return x

summary = {

"RUN\_ID": RUN\_ID,

"backend": backend.name,

"shots": SHOTS,

"opt\_level": OPT\_LEVEL,

"dataset": DATA\_PATH,

"n\_samples": int(n),

"n\_session1": int(len(X\_s1)),

"n\_session2": int(len(X\_s2)),

"n\_session3": int(len(X\_s3)),

"features": feature\_cols,

"amp\_q\_max": AMP\_Q\_MAX,

"far\_targets": FAR\_TARGETS,

"encodings": summary\_blocks

}

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

json.dump(\_to\_builtin(summary), f, indent=2)

print("\n✅ Threshold Sensitivity Analysis completed. Results saved to:", str(LOG\_DIR.resolve()))

print("Overview (EER / Youden-J / FAR targets) per encoding:")

for enc in ["ANGLE","AMPLITUDE","HYBRID"]:

blk = summary\_blocks[enc]

print(f"\n[{enc}]")

for scope in ("S1","ALL"):

b = blk[scope]

print(f" {scope}: EER={b['EER']:.4f}, thr@EER={b['thr\_at\_EER']:.6f}, thr@J={b['thr\_at\_YoudenJ']:.6f}")

for r in b["FAR\_targets"]:

print(f" FAR≈{r['target\_FAR']\*100:.2f}% -> thr={r['thr']:.6f} (FAR={r['achieved\_FAR']:.4f}, FRR={r['FRR']:.4f})")