load libraries and data

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!pip install mne
```

```
Collecting mne
  Downloading mne-1.10.1-py3-none-any.whl.metadata (20 kB)
Requirement already satisfied: decorator in /usr/local/lib/python3.12/dist-packages (from mne) (4.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.12/dist-packages (from mne) (3.1.6)
Requirement already satisfied: lazy-loader>=0.3 in /usr/local/lib/python3.12/dist-packages (from mne) (0.4)
Requirement already satisfied: matplotlib>=3.7 in /usr/local/lib/python3.12/dist-packages (from mne) (3.10.0)
Requirement already satisfied: numpy<3,>=1.25 in /usr/local/lib/python3.12/dist-packages (from mne) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-packages (from mne) (25.0)
Requirement already satisfied: pooch>=1.5 in /usr/local/lib/python3.12/dist-packages (from mne) (1.8.2)
Requirement already satisfied: scipy>=1.11 in /usr/local/lib/python3.12/dist-packages (from mne) (1.16.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from mne) (4.67.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (4.60.1)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (1.4.9)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (11.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (3.2.5)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib>=3.7->mne) (2.9.0.p
Requirement already satisfied: platformdirs>=2.5.0 in /usr/local/lib/python3.12/dist-packages (from pooch>=1.5->mne) (4.4.0)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.12/dist-packages (from pooch>=1.5->mne) (2.32.4)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.12/dist-packages (from jinja2->mne) (3.0.3)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplotlib>=3.7->mne
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0->pooch>=1
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0->pooch>=1.5->mne) (3.
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0->pooch>=1.5->mn
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests>=2.19.0->pooch>=1.5->mn
Downloading mne-1.10.1-py3-none-any.whl (7.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.4/7.4 MB 46.8 MB/s eta 0:00:00
Installing collected packages: mne
Successfully installed mne-1.10.1
```

```
# === SETUP ===
import os
import numpy as np
import mne
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.metrics import (
    classification_report, confusion_matrix,
    average_precision_score, precision_recall_curve
)
#helper
from sklearn.metrics import precision_recall_curve, f1_score
import numpy as np

def best_threshold_from_validation(y_true_val, y_proba_val):
    """Return threshold that maximizes F1 for class=1 on the validation set."""
    prec, rec, th = precision_recall_curve(y_true_val, y_proba_val)
    # note: precision_recall_curve gives len(th)+1 points; align sizes for F1 at each threshold
    f1s = []
    for t in th:
        y_pred = (y_proba_val >= t).astype(int)
        f1s.append(f1_score(y_true_val, y_pred, zero_division=0))
    best_idx = int(np.argmax(f1s))
    return float(th[best_idx]), float(f1s[best_idx])

# === CONFIGURATION ===
DATA_FOLDER = "/content/drive/MyDrive/my final project/database"  # ✅ <-- change to your path if needed
CHANNELS = ['Fp1.', 'Fp2.']  # Use both Fp1 and Fp2
Z_THRESH = 6.0               # Dynamic threshold
WINDOW_SEC = 1               # Window duration in seconds
LOGIC = 'either'             # Logic for labeling: 'either', 'both', or 'single'
SUBJECTS = range(1, 30)      # You can expand this when m  data available
```

```
# === FUNCTION: Extract windows and label ===
def extract_eeg_windows(
    filepath,
    channels=['Fp1.','Fp2.'],
    z_thresh=6.0,
    logic='either',
    window_sec=0.5,
    overlap=0.750,          # <-- NEW: 0.50 = 50% overlap (try 0.75 later)
    max_windows=None       # <-- optional cap (None = use all)
):
    raw = mne.io.read_raw_edf(filepath, preload=True, verbose=False)
    sfreq = int(raw.info['sfreq'])
    win = int(sfreq * window_sec)
    hop = max(1, int(win * (1 - overlap)))   # e.g., 50% overlap -> hop = win/2

    # get channel data in µV
    data = [raw.copy().pick(ch).get_data()[0] * 1e6 for ch in channels]
    n = min(len(d) for d in data)            # align to shortest channel

    # sliding starts with overlap
    starts = list(range(0, n - win + 1, hop))

    X, y = [], []
    for s in starts:
        wins = [d[s:s+win] for d in data]
        # z per window (add epsilon for flat windows)
        z_scores = []
        for w in wins:
            sd = np.std(w)
            sd = sd if sd > 0 else 1e-9
            z_scores.append((np.max(np.abs(w)) - np.mean(w)) / sd)

        if logic == 'either':
            label = 1 if max(z_scores) > z_thresh else 0
        elif logic == 'both':
            label = 1 if sum(z_scores) > z_thresh else 0
        elif logic == 'single':
            label = 1 if z_scores[0] > z_thresh else 0
        else:
            raise ValueError("logic must be 'either', 'both', or 'single'")

        X.append(np.stack(wins))  # (channels, samples)
        y.append(label)

    X = np.asarray(X)
    y = np.asarray(y, dtype=int)

    # optional: cap how many windows you keep (useful during experiments)
    if max_windows is not None and len(X) > max_windows:
        idx = np.random.choice(len(X), max_windows, replace=False)
        X, y = X[idx], y[idx]

    return X, y



# === LOAD DATA ===
all_X, all_y, groups = [], [], []
for subject in SUBJECTS:
    for run in [1, 2]:
        fpath = os.path.join(DATA_FOLDER, f"S{subject:03d}R{run:02d}.edf")
        if not os.path.exists(fpath):
            continue
        try:
            X, y = extract_eeg_windows(
    fpath, channels=CHANNELS,
    z_thresh=Z_THRESH, logic=LOGIC,
    window_sec=WINDOW_SEC, overlap=0.75,  # try 0.75 for more recall
    max_windows=None                      # or an int (e.g., 120) if you want a cap
)


            if len(X) > 0:
                all_X.append(X)
                all_y.append(y)
                groups.extend([subject] * len(y))
        except Exception as e:
```

```
            print(f"❌ Error in {fpath}: {e}")

if not all_X:
    raise RuntimeError("No data loaded. Check DATA_FOLDER and EDF file paths.")

X_all = np.concatenate(all_X)
y_all = np.concatenate(all_y)
groups = np.array(groups)

print(f"📦 Total samples: {len(y_all)}, Blinks: {np.sum(y_all)}")

# === SUBJECT-WISE CROSS-VALIDATION ===
logo = LeaveOneGroupOut()
y_true, y_pred, y_proba = [], [], []

for train_idx, test_idx in logo.split(X_all, y_all, groups):
    clf = RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)
    X_train = X_all[train_idx].reshape(len(train_idx), -1)
    X_test  = X_all[test_idx].reshape(len(test_idx), -1)
    clf.fit(X_train, y_all[train_idx])

    y_p = clf.predict(X_test)
    y_pr = clf.predict_proba(X_test)[:, 1]

    y_true.extend(y_all[test_idx])
    y_pred.extend(y_p)
    y_proba.extend(y_pr)

# === METRICS ===
print("\n📊 Classification Report:")
print(classification_report(y_true, y_pred))
print("🧩 Confusion Matrix:")
print(confusion_matrix(y_true, y_pred))
print("🎯 AUC-PR Score:", average_precision_score(y_true, y_proba))

# === PLOTS ===
# Precision-Recall Curve
precision, recall, _ = precision_recall_curve(y_true, y_proba)
plt.figure(figsize=(6, 4))
plt.plot(recall, precision, marker='o')
plt.title("📈 Precision-Recall Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.grid(True)
plt.show()

# Feature Importances (based on full data)
#clf_final = RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)
#clf_final.fit(X_all.reshape(len(X_all), -1), y_all)
#importances = clf_final.feature_importances_

#plt.figure(figsize=(12, 3))
#plt.plot(importances)
#plt.title("🧠 Feature Importances (Random Forest)")
#plt.xlabel("Feature Index (flattened channels)")
#plt.ylabel("Importance")
#plt.grid(True)
#plt.show()
```

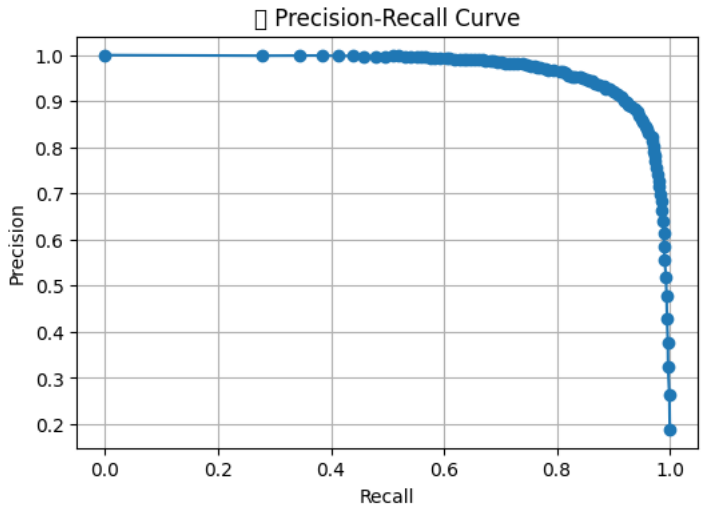
```
📦 Total samples: 13974, Blinks: 2642

📊 Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.99      0.98     11332
           1       0.96      0.82      0.88      2642

    accuracy                           0.96     13974
   macro avg       0.96      0.91      0.93     13974
weighted avg       0.96      0.96      0.96     13974

🧩 Confusion Matrix:
[[11231   101]
 [  473  2169]]
🎯 AUC-PR Score: 0.9693774044019361
/usr/local/lib/python3.12/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128200 (\N{CHART WITH UPWARDS TREND}) mi
  fig.canvas.print_figure(bytes_io, **kw)
```



*numbera are good! but are they meaningfull? *

```
from sklearn.utils import shuffle
from sklearn.metrics import average_precision_score

# Shuffle labels as a baseline
y_shuf = shuffle(y_all, random_state=42)
logo = LeaveOneGroupOut()
y_true_null, y_proba_null = [], []
clf_null = RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)

for tr, te in logo.split(X_all, y_shuf, groups):
    clf_null.fit(X_all[tr].reshape(len(tr), -1), y_shuf[tr])
    y_true_null.extend(y_shuf[te])
    y_proba_null.extend(clf_null.predict_proba(X_all[te].reshape(len(te), -1))[:,1])

print("Null AUC-PR:", average_precision_score(y_true_null, y_proba_null))
```

```
Null AUC-PR: 0.18622245528502576
```

overlap check

```
import numpy as np
from collections import defaultdict

def _zscore_rows(X):
    mu = X.mean(axis=1, keepdims=True)
    sd = X.std(axis=1, keepdims=True) + 1e-8
    return (X - mu) / sd

def mean_adjacent_correlation(X, groups, order=None):
    """
```

```
    Estimate the mean correlation between *adjacent* windows within each subject.
    - X: array (n_windows, n_features)
    - groups: array-like (n_windows,) subject IDs (used by your LeaveOneGroupOut)
    - order: optional array-like (n_windows,) with time indices if your rows are NOT already
             in temporal order per subject. If None, we assume current row order is temporal.

    Returns: (rho_mean, rho_median, n_pairs)
    """
    X = np.asarray(X)
    if X.ndim > 2:  # in case your features are 3D (e.g., channels × features)
        X = X.reshape(X.shape[0], -1)

    groups = np.asarray(groups)

    # group indices
    idx_by_group = defaultdict(list)
    for i, g in enumerate(groups):
        idx_by_group[g].append(i)

    # if time order provided, sort within group; else keep current order
    if order is not None:
        order = np.asarray(order)

    # z-score each row for Pearson-like similarity
    Xz = _zscore_rows(X)

    adj_corrs = []
    for g, idxs in idx_by_group.items():
        if order is not None:
            idxs = sorted(idxs, key=lambda i: order[i])

        # consecutive pairs (i, i+1) within this subject
        for a, b in zip(idxs[:-1], idxs[1:]):
            # cosine similarity of z-scored rows == Pearson corr between feature vectors
            num = (Xz[a] * Xz[b]).sum()
            den = np.linalg.norm(Xz[a]) * np.linalg.norm(Xz[b]) + 1e-12
            adj_corrs.append(num / den)

    if len(adj_corrs) == 0:
        return np.nan, np.nan, 0

    adj_corrs = np.array(adj_corrs, dtype=float)
    return float(np.nanmean(adj_corrs)), float(np.nanmedian(adj_corrs)), int(adj_corrs.size)

def effective_sample_size(N, rho):
    return int(N * (1 - rho) / (1 + rho))

# ---- RUN IT ON YOUR DATA ----
# If your rows are already in time order per subject, leave order=None.
# If you have a time index per window, pass it as 'order=time_index_array'.
rho_mean, rho_median, n_pairs = mean_adjacent_correlation(X_all, groups, order=None)

N = len(y_all)
Neff = effective_sample_size(N, rho_mean) if np.isfinite(rho_mean) else None
rel_eff = Neff / N if Neff is not None else None

print(f"Windows (N): {N}")
print(f"Adjacent correlation ρ (mean): {rho_mean:.3f} | median: {rho_median:.3f} | pairs: {n_pairs}")
if Neff is not None:
    print(f"Neff: {Neff}   |   Relative efficiency: {rel_eff:.2%}")
else:
    print("Neff: N/A (ρ not finite)")
```

```
Windows (N): 13974
Adjacent correlation ρ (mean): 0.195 | median: 0.132 | pairs: 13945
Neff: 9408   |   Relative efficiency: 67.33%
```

stress-test with held-out subjects

```
# ---------------- FINAL HELD-OUT TEST ----------------
heldout_subjects = [5, 12, 19]   # freeze your held-out set
mask_test = np.isin(groups, heldout_subjects)
```

```
mask_dev  = ~mask_test

X_dev, y_dev, g_dev = X_all[mask_dev], y_all[mask_dev], groups[mask_dev]
X_tst, y_tst = X_all[mask_test], y_all[mask_test]

clf = RandomForestClassifier(n_estimators=400, class_weight="balanced", random_state=42, n_jobs=-1)

# OOF predictions on DEV to choose threshold
logo = LeaveOneGroupOut()
y_dev_true, y_dev_proba = [], []
for tr, te in logo.split(X_dev.reshape(len(X_dev), -1), y_dev, g_dev):
    clf.fit(X_dev[tr].reshape(len(tr), -1), y_dev[tr])
    y_dev_true.extend(y_dev[te])
    y_dev_proba.extend(clf.predict_proba(X_dev[te].reshape(len(te), -1))[:,1])

prec, rec, th = precision_recall_curve(y_dev_true, y_dev_proba)
f1s = [f1_score(y_dev_true, (np.array(y_dev_proba) >= t).astype(int), zero_division=0) for t in th]
t_best = th[int(np.argmax(f1s))]
print("Best threshold on DEV:", t_best)

# Retrain on full DEV and test on held-out
clf.fit(X_dev.reshape(len(X_dev), -1), y_dev)
p_test = clf.predict_proba(X_tst.reshape(len(X_tst), -1))[:,1]
from sklearn.metrics import roc_auc_score # Import roc_auc_score
auprc_test = average_precision_score(y_tst, p_test)
auroc_test = roc_auc_score(y_tst, p_test)

y_pred_test = (p_test >= t_best).astype(int)

print("\n=== FINAL HELD-OUT TEST ===")
print("Held-out subjects:", heldout_subjects)
print("AUC-PR:", auprc_test, "| AUC-ROC:", auroc_test)
print(classification_report(y_tst, y_pred_test, digits=3))
print("Confusion matrix:\n", confusion_matrix(y_tst, y_pred_test))
print("Null AUC-PR (≈ prevalence):", y_tst.mean())
```

```
Best threshold on DEV: 0.3525

=== FINAL HELD-OUT TEST ===
Held-out subjects: [5, 12, 19]
AUC-PR: 0.8744937598729543 | AUC-ROC: 0.9923625981441827
              precision    recall  f1-score   support

           0      0.993     0.991     0.992      1401
           1      0.745     0.778     0.761        45

    accuracy                          0.985      1446
   macro avg      0.869     0.885     0.877      1446
weighted avg      0.985     0.985     0.985      1446

Confusion matrix:
 [[1389   12]
 [  10   35]]
Null AUC-PR (≈ prevalence): 0.03112033195020747
```

IETHER \ BOTH \ SINGLE CHANNEL

```
# === TEST ALL LOGIC OPTIONS ===
logic_options = ["either", "both", "single"]
results = {}

for logic in logic_options:
    print(f"\n🔎 Testing logic = {logic.upper()} ...")

    # --- reload all data with this labeling logic ---
    all_X, all_y, groups = [], [], []
    for subject in SUBJECTS:
        for run in [1, 2]:
            fpath = os.path.join(DATA_FOLDER, f"S{subject:03d}R{run:02d}.edf")
            if not os.path.exists(fpath):
                continue
            try:
                X, y = extract_eeg_windows(
                    fpath, channels=CHANNELS,
                    z_thresh=Z_THRESH, logic=logic,
                    window_sec=WINDOW_SEC, overlap=0.75,
                    max_windows=None
```

```
                )
                if len(X) > 0:
                    all_X.append(X)
                    all_y.append(y)
                    groups.extend([subject] * len(y))
            except Exception as e:
                print(f"❌ Error in {fpath}: {e}")

    if not all_X:
        print(f"⚠️ No data for logic={logic}, skipping.")
        continue

    X_all = np.concatenate(all_X)
    y_all = np.concatenate(all_y)
    groups = np.array(groups)

    # --- subject-wise cross-validation ---
    logo = LeaveOneGroupOut()
    y_true, y_pred, y_proba = [], [], []

    for train_idx, test_idx in logo.split(X_all, y_all, groups):
        clf = RandomForestClassifier(n_estimators=100, class_weight='balanced', random_state=42)
        X_train = X_all[train_idx].reshape(len(train_idx), -1)
        X_test  = X_all[test_idx].reshape(len(test_idx), -1)
        clf.fit(X_train, y_all[train_idx])

        y_p = clf.predict(X_test)
        y_pr = clf.predict_proba(X_test)[:, 1]

        y_true.extend(y_all[test_idx])
        y_pred.extend(y_p)
        y_proba.extend(y_pr)

    # --- evaluate ---
    auc_pr = average_precision_score(y_true, y_proba)
    print("📊 Classification Report:")
    print(classification_report(y_true, y_pred))
    print("🎯 AUC-PR Score:", auc_pr)

    results[logic] = auc_pr

# === SUMMARY ===
print("\n✅ Comparison of logics (by AUC-PR):")
for logic, auc in results.items():
    print(f"{logic}: {auc:.3f}")

best_logic = max(results, key=results.get)
print(f"\n⏳ Best logic is: {best_logic.upper()} (AUC-PR={results[best_logic]:.3f})")
```

```
🔎 Testing logic = EITHER ...
📊 Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.99      0.98     11332
           1       0.96      0.82      0.88      2642

    accuracy                           0.96     13974
   macro avg       0.96      0.91      0.93     13974
weighted avg       0.96      0.96      0.96     13974

🎯 AUC-PR Score: 0.9693774044019361

🔎 Testing logic = BOTH ...
📊 Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.82      0.83      5430
           1       0.89      0.91      0.90      8544

    accuracy                           0.87     13974
   macro avg       0.87      0.86      0.87     13974
weighted avg       0.87      0.87      0.87     13974

🎯 AUC-PR Score: 0.9680801204618499

🔎 Testing logic = SINGLE ...
📊 Classification Report:
              precision    recall  f1-score   support
```

```
           0       0.97      0.99      0.98     11890
           1       0.95      0.82      0.88      2084

    accuracy                           0.97     13974
   macro avg       0.96      0.91      0.93     13974
weighted avg       0.97      0.97      0.97     13974

🎯 AUC-PR Score: 0.9685013873919293

✅ Comparison of logics (by AUC-PR):
either: 0.969
both: 0.968
single: 0.969

⏳ Best logic is: EITHER (AUC-PR=0.969)
```

```
# Find F1-max operating point on the concatenated out-of-fold predictions
best_t, best_f1 = best_threshold_from_validation(np.array(y_true), np.array(y_proba))
print(f"⚙️ Best F1 threshold for logic={logic}: {best_t:.3f} (F1={best_f1:.3f})")

# Optional: show confusion matrix at that threshold
y_hat = (np.array(y_proba) >= best_t).astype(int)
print("🧩 Confusion Matrix @ best F1 threshold:")
print(confusion_matrix(np.array(y_true), y_hat))
```

```
⚙️ Best F1 threshold for logic=single: 0.350 (F1=0.911)
🧩 Confusion Matrix @ best F1 threshold:
[[11701   189]
 [  184  1900]]
```