BitePulse AI - Feature Pipeline

Introduction

Our goal in this notebook is to convert labeled meals into model-ready inputs by:

- Cutting videos/poses into fixed windows,
- Extracting pose & motion features (hand→mouth distance, wrist speed, elbow angle, etc.), and
- Saving tensors + labels to disk for fast training. We'll also add a minimal metrics scaffold (precision/recall for intake events) to quickly sanity-check feature quality.

Data

We'll utilize the data we saved on our google drive from Label prep and task definition notebook - (label_v1):

Our inputs from labels_v1:

- manifest_with_split.parquet: video paths + split
- frames_idx.parquet: per-frame labels (label, time_sec, key, split)
- segs_idx.parquet: intake segments (start_sec, end_sec, label, key, split)
- subject_split.parquet: to avoid subject leakage
- true2d_parquet/.parquet: per-frame 2D joints

Imports and basic setup

```
# Mount Google Drive:
from google.colab import drive
drive.mount('/content/drive')
```

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

```
from pathlib import Path
from typing import Dict, Tuple, List, Optional
import json, re
import math
import warnings
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
import joblib
warnings.filterwarnings("ignore", category=FutureWarning)

# let's run below to customize notebook display:
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
pd.set_option('display.max_colwidth', 4000)
```

Paths setup

Here's let's define our exact folders on Drive.

```
# Our paths:
ROOT = Path("/content/drive/MyDrive/eatsense")      # project root
LABELS = ROOT / "labels_v1"                         # label artifacts
from previous notebook
POSE_PARQ_DIR = ROOT / "true2d_parquet"             # per-video 2D
pose
OUT_WINDOWS = ROOT / "windows"                      # output: windowed
features & labels

# First let's ensure output directory exists:
OUT_WINDOWS.mkdir(parents=True, exist_ok=True)


# Core config:
WIN_SEC = 2.0                  # window length (seconds)
STRIDE_SEC = 0.5               # window stride (seconds)
FPS_FALLBACK = 15.0            # fallback FPS when not found in map

WINDOW_POS_OVERLAP = 0.25      # IoU/overlap with an intake segment to
mark window as positive
USE_SUMMARY = False            # if True: aggregate per-window stats
instead of keeping sequences

# Keep in sync with labels_v1/label_config.json:
INTAKE_LABELS = {"Eat it"}     # which labels count as intake (positive
class)


# Joint naming:
RIGHT_CHAIN = ["Right-Shoulder", "Right-Elbow", "Right-Wrist"]
LEFT_CHAIN  = ["Left-shoulder", "Left-Elbow", "Left-Wrist"]
HEAD_NAME   = "head"

ALL_JOINTS = [HEAD_NAME] + RIGHT_CHAIN + LEFT_CHAIN
```

helpers

```
# helpers:
def load_json(path: Path) -> dict:
    if not path.exists():
        return {}
    with path.open("r") as f:
        return json.load(f)

def save_json(obj: dict, path: Path) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    with path.open("w") as f:
        json.dump(obj, f, indent=2)

def get_fps_map(root: Path, fallback: float) -> Dict[str, float]:
    """
    Load per-video FPS dict if available, else return empty dict.
    """
    fps_path = root / "fps_by_key.json"
    m = load_json(fps_path)
    return {str(k): float(v) for k, v in m.items()} if m else {}

FPS_BY_KEY: Dict[str, float] = get_fps_map(ROOT, FPS_FALLBACK)

def ts_key_from_path(p: str) -> Optional[str]:
    """
    Extract the timestamp key (YYYYMMDD_HHMMSS) from a path string.
    """
    m = re.search(r"(\d{8}_\d{6})", str(p))
    return m.group(1) if m else None

def fps_for_key(key: str) -> float:
    """
    Get FPS for a given key, falling back when missing.
    """
    return float(FPS_BY_KEY.get(str(key), FPS_FALLBACK))



# Quick sanity print:
print("ROOT:", ROOT)
print("Labels dir:", LABELS)
print("Pose parquet dir:", POSE_PARQ_DIR)
print("Windows out dir:", OUT_WINDOWS)
print("Config -> win:", WIN_SEC, "sec | stride:", STRIDE_SEC, "sec |
fallback FPS:", FPS_FALLBACK)
print("Intake labels:", INTAKE_LABELS)
```

```
ROOT: /content/drive/MyDrive/eatsense
Labels dir: /content/drive/MyDrive/eatsense/labels_v1
Pose parquet dir: /content/drive/MyDrive/eatsense/true2d_parquet
Windows out dir: /content/drive/MyDrive/eatsense/windows
Config -> win: 2.0 sec | stride: 0.5 sec | fallback FPS: 15.0
Intake labels: {'Eat it'}
```

Load label indices & manifest

Here let's pull in the artifacts from labels_v1 that we created earlier "manifest_with_split.parquet", "frames_idx.parquet", and "segs_idx.parquet" (plus subject_split.parquet).

This gives us:

- The video-to-file map with train/val/test split.
- Per-frame labels and times
- Merged action segments we'll use to tag windows as positive (intake) or negative.

```
# Required artifacts (under LABELS = ROOT / "labels_v1"):
MANIFEST_PATH   = LABELS / "manifest_with_split.parquet"
FRAMES_IDX_PATH = LABELS / "frames_idx.parquet"
SEGS_IDX_PATH   = LABELS / "segs_idx.parquet"          # segment-level
labels
SUBJ_SPLIT_PATH = LABELS / "subject_split.parquet"     # subject-wise
split
LABEL_CFG_PATH  = LABELS / "label_config.json"         # canonical
label groups

# Load core tables:
manifest   = pd.read_parquet(MANIFEST_PATH)[["key", "rgb",
"poses_true", "split"]]
frames_idx = pd.read_parquet(FRAMES_IDX_PATH)          # cols: key,
split, frame, time_sec, label
segs_idx   = pd.read_parquet(SEGS_IDX_PATH)            # cols: key,
split, start_sec, end_sec, label

label_cfg = {}
if LABEL_CFG_PATH.exists():
    with open(LABEL_CFG_PATH, "r") as f:
        label_cfg = json.load(f)

subjects = None
if SUBJ_SPLIT_PATH.exists():
    subjects = pd.read_parquet(SUBJ_SPLIT_PATH)

# Sanity checks:
assert {"key", "rgb", "split"}.issubset(manifest.columns)
assert {"key", "frame", "time_sec", "label",
"split"}.issubset(frames_idx.columns)
assert {"key", "start_sec", "end_sec", "label",
"split"}.issubset(segs_idx.columns)



# Intake labels:
if "INTAKE_LABELS" not in globals():
    INTAKE_LABELS = set(label_cfg.get("INTAKE", ["Eat it", "drink",
"sip"]))
NON_INTAKE_LABELS = set(label_cfg.get("NON_INTAKE", []))

# Quick lookups:
POSE_PATH_BY_KEY = dict(zip(manifest["key"], manifest["poses_true"]))
SPLIT_BY_KEY     = dict(zip(manifest["key"], manifest["split"]))

print("Loaded:")
print(f"  manifest:   {manifest.shape}  | splits ->
{manifest['split'].value_counts().to_dict()}")
print(f"  frames_idx: {frames_idx.shape} | splits ->
{frames_idx['split'].value_counts().to_dict()}")
print(f"  segs_idx:   {segs_idx.shape}   | splits ->
{segs_idx['split'].value_counts().to_dict()}")
if subjects is not None:
    print(f"  subject_split: {subjects.shape}")
print(f"INTAKE_LABELS = {sorted(map(str, INTAKE_LABELS))}")
```

```
Loaded:
  manifest:   (135, 4)  | splits -> {'train': 89, 'val': 25, 'test':
21}
  frames_idx: (742887, 5) | splits -> {'train': 471564, 'val': 159546,
'test': 111777}
  segs_idx:   (158370, 5)   | splits -> {'train': 100641, 'val':
33233, 'test': 24496}
  subject_split: (0, 2)
INTAKE_LABELS = ['Eat it']
```

Sliding windows & labels

Here, let's turn our continuous videos into small, fixed-length clips and give each clip a label. This way the Models learn better from uniform clips (e.g., 2 s) than entire videos. basically, for each video we'll slide a window of length WIN_SEC forward by STRIDE_SEC, producing many [start_sec, end_sec] intervals:

- Labeling rule: A window is positive (intake) if its temporal IoU with any intake segment (e.g., “Eat it”) is ≥ WINDOW_POS_OVERLAP; otherwise it's negative.
- Inputs:
    - frames_idx (per-frame times, to get each video's duration)
    - segs_idx (ground-truth intake segments with start/end times)
    - INTAKE_LABELS, WIN_SEC, STRIDE_SEC, WINDOW_POS_OVERLAP
- Output: windows_idx; one row per window with: key, split, win_id, start_sec, end_sec, max_iou, label.

This gives us a clean, model-ready index of training clips tied back to each source video.

```python
def key_durations(frames: pd.DataFrame) -> pd.Series:
    """
    Per-key duration in seconds (max time_sec seen in frames_idx).
    """
    return frames.groupby("key")["time_sec"].max().astype(float)

def windows_for_key(total_sec: float,
                    win_sec: float,
                    stride_sec: float) -> List[Tuple[float, float]]:
    """
    Produce [ (start_sec, end_sec), ... ] for a single video's
duration.
    """
    if total_sec <= 0 or win_sec <= 0:
        return []
    starts = np.arange(0.0, max(total_sec - win_sec, 0) + 1e-9,
stride_sec)
    return [(float(s), float(min(s + win_sec, total_sec))) for s in
starts]

def iou_1d(a0: float, a1: float, b0: float, b1: float) -> float:
    """Intersection-over-Union for 1D segments [a0,a1] and [b0,b1]."""
    inter = max(0.0, min(a1, b1) - max(a0, b0))
    union = max(1e-9, (a1 - a0) + (b1 - b0) - inter)
    return inter / union

def label_windows_for_key(k: str,
                          win_pairs: List[Tuple[float, float]],
                          intake_segs: pd.DataFrame,
                          pos_iou_thresh: float) -> Tuple[np.ndarray,
np.ndarray]:
    """
    For all windows of one key, compute:
      - max IoU to any intake segment
      - binary label (1 if max_iou >= pos_iou_thresh, else 0)
    Returns (max_iou_vec, label_vec).
    """
    max_iou = np.zeros(len(win_pairs), dtype=float)
    if len(intake_segs) == 0:
        return max_iou, (max_iou >= pos_iou_thresh).astype(int)

    starts = intake_segs["start_sec"].to_numpy(dtype=float)
    ends   = intake_segs["end_sec"].to_numpy(dtype=float)

    for i, (ws, we) in enumerate(win_pairs):
        # quick prune: segments overlapping time span:
        lo = (ends   > ws)
        hi = (starts < we)
        mask = lo & hi
        if not mask.any():
            continue
        ious = [iou_1d(ws, we, float(s), float(e)) for s, e in
zip(starts[mask], ends[mask])]
        if ious:
            max_iou[i] = max(ious)

    labels = (max_iou >= pos_iou_thresh).astype(int)
    return max_iou, labels

def build_window_index(frames_idx: pd.DataFrame,
                       segs_idx: pd.DataFrame,
                       win_sec: float,
                       stride_sec: float,
                       pos_iou_thresh: float,
                       intake_labels: set) -> pd.DataFrame:
    """
    Create a per-window index across all keys with columns:
      key, split, win_id, start_sec, end_sec, max_iou, label
    """
    vid_len = key_durations(frames_idx)   # seconds per key:
    # keep only intake segments:
    intake = segs_idx[segs_idx["label"].isin(intake_labels)].copy()
    intake.sort_values(["key", "start_sec", "end_sec"], inplace=True)

    out_rows = []
    for k, total_sec in vid_len.items():
        # windows for this key:
        ws = windows_for_key(total_sec, win_sec, stride_sec)
        if not ws:
            continue

        # all intake segments for this key:
        k_segs = intake[intake["key"] == k][["start_sec", "end_sec"]]
        max_iou, y = label_windows_for_key(k, ws, k_segs,
pos_iou_thresh)

        split = SPLIT_BY_KEY.get(k, "train")
        for wid, ((s, e), iou_val, lab) in enumerate(zip(ws, max_iou,
y)):
            out_rows.append((k, split, wid, s, e, float(iou_val),
int(lab)))

    win_df = pd.DataFrame(out_rows,
                          columns=["key", "split", "win_id",
                                   "start_sec", "end_sec",
                                   "max_iou", "label"])
    return win_df

# Build the window index using our config knobs:
windows_idx = build_window_index(
    frames_idx=frames_idx,
    segs_idx=segs_idx,
    win_sec=WIN_SEC,
    stride_sec=STRIDE_SEC,
    pos_iou_thresh=WINDOW_POS_OVERLAP,
    intake_labels=INTAKE_LABELS,
)

# Quick sanity: class balance and a few rows:
print("Windows index:", windows_idx.shape)
print("  by split:", windows_idx["split"].value_counts().to_dict())
print("  positives:", int(windows_idx["label"].sum()),
      " | negatives:", int((1 - windows_idx["label"]).sum()))
display(windows_idx.head())
```

```
Windows index: (98582, 7)
  by split: {'train': 62568, 'val': 21184, 'test': 14830}
  positives: 568  | negatives: 98014
```

{"summary":"{\n \"name\": \"display(windows_idx\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"key\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 1,\n   \"samples\": [\n   \"20210518_230219\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"split\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 1,\n   \"samples\": [\n   \"train\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"win_id\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 1,\n   \"min\": 0,\n   \"max\": 4,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   1\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"start_sec\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.7905694150420949,\n   \"min\": 0.0,\n   \"max\": 2.0,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.5\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"end_sec\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.7905694150420949,\n   \"min\": 2.0,\n   \"max\": 4.0,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   2.5\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"max_iou\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0.0,\n   \"min\": 0.0,\n   \"max\": 0.0,\n   \"num_unique_values\": 1,\n   \"samples\": [\n   0.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"label\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 0,\n

```
\"min\": 0,\n        \"max\": 0,\n        \"num_unique_values\": 1,\n
\"samples\": [\n          0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe"}
```

Summary Highlights:

- Rows: 98,582 windows total
- Splits: train/val/test ≈ 62,568 / 21,184 / 14,830 (mirrors video splits).
- Class balance: 568 positives vs 98,014 negatives → highly imbalanced (≈0.6% positive).

Pose to window features

Here, let's turn each time window into a compact set of motion features derived from the 2D joints (no pixels yet).

For every window (key, start_sec, end_sec) we'll compute things like:

- Wrist speed statistics (mean / max) for left & right
- Wrist↔head distance statistics (min / mean)
- Elbow angle statistics (mean / std / min)
- Wrist path length (how far the wrist moved in the window)

These lightweight features are a great baseline and also useful for debugging the labeling/timing before we try CNN/RNN models on raw frames.

```python
POSE_PATH_BY_KEY: Dict[str, str] = dict(zip(manifest["key"],
manifest["poses_true"]))

POSE_PARQ_DIR = ROOT / "true2d_parquet"
OUT_FEATS     = ROOT / "windows" / "pose_feats.parquet"
OUT_FEATS.parent.mkdir(parents=True, exist_ok=True)

# Joint naming we expect:
RIGHT = ["right-shoulder", "right-elbow", "right-wrist"]
LEFT  = ["left-shoulder",  "left-elbow",  "left-wrist"]
HEAD  = "head"

def _find_col(df: pd.DataFrame, base: str, suffix: str) -> str:
    """Locate a column by lowercase name match, tolerant to
capitalization like 'Left-shoulder'."""
    want = f"{base.lower()}_{suffix}"
    for c in df.columns:
        if c.lower() == want:
            return c
    raise KeyError(f"column '{want}' not found in pose dataframe")

def _speed(x: np.ndarray, y: np.ndarray, t: np.ndarray) -> np.ndarray:
    vx = np.gradient(x, t, edge_order=1)
    vy = np.gradient(y, t, edge_order=1)
    return np.hypot(vx, vy)

def _angle(ax, ay, bx, by, cx, cy) -> np.ndarray:
    """
    Elbow angle ABC in degrees, where B is the elbow.
    A=shoulder, B=elbow, C=wrist.
    """
    v1x, v1y = ax - bx, ay - by
    v2x, v2y = cx - bx, cy - by
    # normalize:
    n1 = np.hypot(v1x, v1y) + 1e-6
    n2 = np.hypot(v2x, v2y) + 1e-6
    dot = (v1x*v2x + v1y*v2y) / (n1*n2)
    dot = np.clip(dot, -1.0, 1.0)
    return np.degrees(np.arccos(dot))

def load_pose_df_for_key(key: str) -> pd.DataFrame:
    """
    Load per-frame pose for a video:
      1) try fast parquet:  ROOT/true2d_parquet/{key}.parquet
      2) fall back to CSV path from manifest['poses_true'] for this
key
    Expected columns after load: 'frame','time_sec' and
f'{joint}_x','{joint}_y'.
    """
    parq = POSE_PARQ_DIR / f"{key}.parquet"
    if parq.exists():
        df = pd.read_parquet(parq)
        return df

    # Fallback to CSV (slower) - parse "(x, y)" strings:
    csv_path = Path(POSE_PATH_BY_KEY[key])
    df = pd.read_csv(csv_path).copy()
    assert "image_id" in df.columns, "Expected 'image_id' column"
    df.rename(columns={"image_id": "frame"}, inplace=True)

    # If FPS mapping exists on disk, use it; else assume fallback
    fps_map_path = ROOT / "fps_by_key.json"
    fps = 15.0
    if fps_map_path.exists():
        try:

            with open(fps_map_path, "r") as f:
                m = json.load(f)
            # key like '20210518_230219':
            ts_key = key  # our key is already the timestamp token
            if ts_key in m:
                fps = float(m[ts_key])
        except Exception:
            pass
    df["time_sec"] = df["frame"] / float(fps)

    # Split "(x, y)" columns into numeric:
    import re
    pat = re.compile(r"\(?\s*([-+]?\d*\.?\d+(?:e[-+]?\d+)?)\s*[, ]\
s*([-+]?\d*\.?\d+(?:e[-+]?\d+)?)\s*\)?", re.I)
    meta = {"path","imgName","frame","date","time","Action"}
    joints = [c for c in df.columns if c not in meta]
    for j in joints:
        xy = df[j].map(lambda s: pat.match(str(s)).groups() if
pd.notna(s) and pat.match(str(s)) else (np.nan, np.nan))
        df[f"{j}_x"] = pd.to_numeric([p[0] for p in xy],
errors="coerce")
        df[f"{j}_y"] = pd.to_numeric([p[1] for p in xy],
errors="coerce")
    keep = ["frame","time_sec"] + [f"{j}_x" for j in joints] +
[f"{j}_y" for j in joints]
    return df[keep].reset_index(drop=True)

def slice_window(df: pd.DataFrame, start: float, end: float) ->
pd.DataFrame:
    m = (df["time_sec"] >= start) & (df["time_sec"] < end)
    return df.loc[m]

def wrist_path_len(x: np.ndarray, y: np.ndarray) -> float:
    if len(x) < 2:
        return 0.0
    return float(np.hypot(np.diff(x), np.diff(y)).sum())

def summarize_window(dfw: pd.DataFrame) -> Dict[str, float]:
    """Compute features for one window slice of the pose dataframe."""
    if dfw.empty:
        return {   # consistent NaNs so downstream can impute:
            "rw_speed_mean": np.nan, "rw_speed_max": np.nan,
"rw_head_min": np.nan, "rw_path": 0.0,
            "lw_speed_mean": np.nan, "lw_speed_max": np.nan,
"lw_head_min": np.nan, "lw_path": 0.0,
            "re_angle_mean": np.nan, "re_angle_std": np.nan,
            "le_angle_mean": np.nan, "le_angle_std": np.nan,
        }

    t  = dfw["time_sec"].to_numpy()

    # Resolve column names robustly:
    rx = dfw[_find_col(dfw, "Right-Wrist", "x")].to_numpy()
    ry = dfw[_find_col(dfw, "Right-Wrist", "y")].to_numpy()
    lx = dfw[_find_col(dfw, "Left-Wrist", "x")].to_numpy()
    ly = dfw[_find_col(dfw, "Left-Wrist", "y")].to_numpy()
```

```python
    hx = dfw[_find_col(dfw, HEAD, "x")].to_numpy()
    hy = dfw[_find_col(dfw, HEAD, "y")].to_numpy()

    rsx = dfw[_find_col(dfw, "Right-Shoulder", "x")].to_numpy()
    rsy = dfw[_find_col(dfw, "Right-Shoulder", "y")].to_numpy()
    rex = dfw[_find_col(dfw, "Right-Elbow", "x")].to_numpy()
    rey = dfw[_find_col(dfw, "Right-Elbow", "y")].to_numpy()

    lsx = dfw[_find_col(dfw, "Left-Shoulder", "x")].to_numpy()
    lsy = dfw[_find_col(dfw, "Left-Shoulder", "y")].to_numpy()
    lex = dfw[_find_col(dfw, "Left-Elbow", "x")].to_numpy()
    ley = dfw[_find_col(dfw, "Left-Elbow", "y")].to_numpy()

    # Speeds:
    r_speed = _speed(rx, ry, t)
    l_speed = _speed(lx, ly, t)

    # Distances to head:
    rw_head = np.hypot(rx - hx, ry - hy)
    lw_head = np.hypot(lx - hx, ly - hy)

    # Elbow angles:
    r_ang = _angle(rsx, rsy, rex, rey, rx, ry)
    l_ang = _angle(lsx, lsy, lex, ley, lx, ly)

    return {
        "rw_speed_mean": float(np.nanmean(r_speed)),
        "rw_speed_max":  float(np.nanmax(r_speed)),
        "rw_head_min":   float(np.nanmin(rw_head)),
        "rw_path":       wrist_path_len(rx, ry),

        "lw_speed_mean": float(np.nanmean(l_speed)),
        "lw_speed_max":  float(np.nanmax(l_speed)),
        "lw_head_min":   float(np.nanmin(lw_head)),
        "lw_path":       wrist_path_len(lx, ly),

        "re_angle_mean": float(np.nanmean(r_ang)),
        "re_angle_std":  float(np.nanstd(r_ang)),
        "le_angle_mean": float(np.nanmean(l_ang)),
        "le_angle_std":  float(np.nanstd(l_ang)),
    }

# Extract features for all windows:
def build_pose_features(windows: pd.DataFrame) -> pd.DataFrame:
    """
    For each key, load pose once, then compute features across that
key's windows.
    Returns a DataFrame aligned 1:1 with input `windows`.
    """
    rows = []
    for key, g in windows.groupby("key", sort=False):
        try:
            pose_df = load_pose_df_for_key(key)
        except Exception as e:
            print(f"[warn] failed to load pose for {key}: {e}")
            # Keep placeholder rows (NaNs) so downstream alignment
remains intact:
            for _, r in g.iterrows():
                rows.append({
                    "key": key, "split": r["split"], "win_id":
r["win_id"],
                    "start_sec": r["start_sec"], "end_sec":
r["end_sec"], "label": r["label"],
                    **summarize_window(pd.DataFrame())
                })
            continue

        for _, r in g.iterrows():
            dfw = slice_window(pose_df, r["start_sec"], r["end_sec"])
            feats = summarize_window(dfw)
            rows.append({
                "key": key, "split": r["split"], "win_id":
r["win_id"],
                "start_sec": r["start_sec"], "end_sec": r["end_sec"],
                "label": r["label"],     # 1 if positive (by IoU),
else 0
                **feats,
            })
    return pd.DataFrame(rows)



pose_feats = build_pose_features(windows_idx)
print("pose_feats:", pose_feats.shape)
display(pose_feats.head())
```

```
pose_feats: (98582, 18)
```

{"summary":"{\n \"name\": \"display(pose_feats\",\n \"rows\": 5,\n \"fields\": [\n   {\n   \"column\": \"key\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 1,\n   \"samples\": [\n   \"20210518_230219\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"split\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 1,\n   \"samples\": [\n   \"train\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"win_id\",\n   \"properties\": {\n   \"dtype\": \"number\",\n

```
\"std\": 1,\n        \"min\": 0,\n        \"max\": 4,\n
\"num_unique_values\": 5,\n        \"samples\": [\n          1\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"start_sec\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.7905694150420949,\n        \"min\": 0.0,\n        \"max\": 2.0,\n
\"num_unique_values\": 5,\n        \"samples\": [\n          0.5\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"end_sec\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.7905694150420949,\n        \"min\": 2.0,\n        \"max\": 4.0,\n
\"num_unique_values\": 5,\n        \"samples\": [\n          2.5\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"label\",\n      \"properties\":
{\n        \"dtype\": \"number\",\n        \"std\": 0,\n
\"min\": 0,\n        \"max\": 0,\n        \"num_unique_values\": 1,\n
\"samples\": [\n          0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"rw_speed_mean\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 40.55284680868913,\n
\"min\": 59.10183528139727,\n        \"max\": 145.00552829018767,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
144.75084389296774\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"rw_speed_max\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 17.32533336771692,\n        \"min\":
261.3234830323828,\n        \"max\": 292.95506937170114,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
261.3234830323828\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"rw_head_min\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 26.90420997287923,\n        \"min\":
16.572304338709174,\n        \"max\": 77.26076104637659,\n
\"num_unique_values\": 3,\n        \"samples\": [\n
16.572304338709174\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"rw_path\",\n      \"properties\": {\n        \"dtype\": \"number\",\
n        \"std\": 129.02697675796847,\n        \"min\":
120.85777254200542,\n        \"max\": 395.1912431818693,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
395.1912431818693\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"lw_speed_mean\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 15.96482050562836,\n        \"min\":
55.45386375428794,\n        \"max\": 93.77587305298204,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
86.06739412528454\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"lw_speed_max\",\n      \"properties\": {\n        \"dtype\":
```

```
\"number\",\n        \"std\": 36.993863499649514,\n        \"min\":
173.32421136702382,\n        \"max\": 240.86545646044064,\n
\"num_unique_values\": 2,\n        \"samples\": [\n
173.32421136702382\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"lw_head_min\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 26.382348697412805,\n        \"min\":
20.674799932865145,\n        \"max\": 70.18548657992622,\n
\"num_unique_values\": 3,\n        \"samples\": [\n
20.674799932865145\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"lw_path\",\n      \"properties\": {\n        \"dtype\": \"number\",\
n        \"std\": 63.23539989577235,\n        \"min\":
111.39329167331904,\n        \"max\": 254.55608605665088,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
242.44418653305277\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"re_angle_mean\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 15.086112161958045,\n        \"min\":
121.10032439180249,\n        \"max\": 156.60936481472726,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
154.72659853125575\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"re_angle_std\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 3.5490511254824337,\n        \"min\":
25.178434247606084,\n        \"max\": 32.854781954677016,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
25.178434247606084\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"le_angle_mean\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 17.132497547567432,\n        \"min\":
118.18239081866604,\n        \"max\": 159.40453146861384,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
156.09282903753862\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"le_angle_std\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 5.731014349222636,\n        \"min\":
12.925138836565326,\n        \"max\": 26.00308732956914,\n
\"num_unique_values\": 5,\n        \"samples\": [\n
12.925138836565326\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\n}","type":"dataframe"}
```

```python
# Save once built:
pose_feats.to_parquet(OUT_FEATS, index=False)
print(f"Saved pose features -> {OUT_FEATS}")
```

```
Saved pose features ->
/content/drive/MyDrive/eatsense/windows/pose_feats.parquet
```

Pack model-ready arrays (X, y) and exports

Now that we have pose_feats (one row per window with metadata + numeric features), let's:

- Select the numeric feature columns,
- Split into train/val/test using the split column,
- Standardize features using stats from the train set only,
- Export light-weight files we can feed into a baseline model quickly (.npz with X, y, and meta).

This keeps the "feature pipeline" self-contained and makes training notebooks tiny.

```
EXPORT_DIR = ROOT / "windows"
EXPORT_DIR.mkdir(parents=True, exist_ok=True)

# First, let's identify feature columns:
META_COLS = {"key","split","win_id","start_sec","end_sec","label"}
all_cols  = list(pose_feats.columns)
FEAT_COLS = [c for c in all_cols if c not in META_COLS]

# Small sanity:
assert len(FEAT_COLS) > 0, "No feature columns detected!"

# Split:
df_train = pose_feats.query("split == 'train'").reset_index(drop=True)
df_val   = pose_feats.query("split == 'val'").reset_index(drop=True)
df_test  = pose_feats.query("split == 'test'").reset_index(drop=True)

print("Rows:", {k: len(v) for k, v in
               {"train": df_train, "val": df_val, "test":
df_test}.items()})
print("Class balance (positives):",
      {"train": int(df_train["label"].sum()),
       "val":   int(df_val["label"].sum()),
       "test":  int(df_test["label"].sum())})
```

```
Rows: {'train': 62568, 'val': 21184, 'test': 14830}
Class balance (positives): {'train': 422, 'val': 57, 'test': 89}
```

```
# Standardize using TRAIN only:
scaler = StandardScaler()
X_train =
scaler.fit_transform(df_train[FEAT_COLS].values.astype(np.float32))
X_val   =
scaler.transform(df_val[FEAT_COLS].values.astype(np.float32))
X_test  =
scaler.transform(df_test[FEAT_COLS].values.astype(np.float32))

y_train = df_train["label"].values.astype(np.int64)
y_val   = df_val["label"].values.astype(np.int64)
```

```
y_test  = df_test["label"].values.astype(np.int64)

# Keep slim metadata for debugging/eval later:
meta_train = df_train[["key","win_id","start_sec","end_sec"]].copy()
meta_val   = df_val[["key","win_id","start_sec","end_sec"]].copy()
meta_test  = df_test[["key","win_id","start_sec","end_sec"]].copy()

# Save exports:
np.savez_compressed(EXPORT_DIR / "Xy_train_pose.npz",
                    X=X_train, y=y_train,
                    keys=meta_train["key"].to_numpy(),
                    win_id=meta_train["win_id"].to_numpy(),
                    start=meta_train["start_sec"].to_numpy(),
                    end=meta_train["end_sec"].to_numpy(),
                    feat_cols=np.array(FEAT_COLS))

np.savez_compressed(EXPORT_DIR / "Xy_val_pose.npz",
                    X=X_val, y=y_val,
                    keys=meta_val["key"].to_numpy(),
                    win_id=meta_val["win_id"].to_numpy(),
                    start=meta_val["start_sec"].to_numpy(),
                    end=meta_val["end_sec"].to_numpy(),
                    feat_cols=np.array(FEAT_COLS))

np.savez_compressed(EXPORT_DIR / "Xy_test_pose.npz",
                    X=X_test, y=y_test,
                    keys=meta_test["key"].to_numpy(),
                    win_id=meta_test["win_id"].to_numpy(),
                    start=meta_test["start_sec"].to_numpy(),
                    end=meta_test["end_sec"].to_numpy(),
                    feat_cols=np.array(FEAT_COLS))

# Save the scaler for reuse (so eval uses the exact same
normalization):
joblib.dump(scaler, EXPORT_DIR / "scaler_pose.joblib")

print("Saved:")
print(" -", EXPORT_DIR / "Xy_train_pose.npz")
print(" -", EXPORT_DIR / "Xy_val_pose.npz")
print(" -", EXPORT_DIR / "Xy_test_pose.npz")
print(" -", EXPORT_DIR / "scaler_pose.joblib")
```

```
Saved:
 - /content/drive/MyDrive/eatsense/windows/Xy_train_pose.npz
 - /content/drive/MyDrive/eatsense/windows/Xy_val_pose.npz
 - /content/drive/MyDrive/eatsense/windows/Xy_test_pose.npz
 - /content/drive/MyDrive/eatsense/windows/scaler_pose.joblib
```

Summary Highlights:

- .npz is tiny, loadable in one line, and framework-agnostic.

- Keeping feat_cols inside ensures feature order stays consistent.
- Saving the scaler prevents train/val/test leakage and makes downstream evaluation reproducible.

Metrics scaffold

Here, let's hook up a simple, fast baseline to verify the feature pipeline end-to-end:

- loads the saved NPZs (Xy_train_pose.npz, Xy_val_pose.npz, Xy_test_pose.npz)
- train a class-weighted neural network baseline (MLP for flat features or GRU head for sequences).
- finds a decision threshold that maximizes F1 on the validation set
- reports precision / recall / F1, ROC AUC, PR AUC, and a confusion matrix on val and test

The block below trains a tiny PyTorch model on the NPZ features we saved earlier. It auto-detects the feature shape:

- 2D (N, F) → a small MLP with dropout
- 3D (N, T, D) → a GRU followed by a linear head (uses the last hidden state)

It uses `BCEWithLogitsLoss` with class weighting, chooses a threshold that maximizes F1 on the validation set, and reports metrics on val and test.

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import (
    precision_recall_curve, average_precision_score, roc_auc_score,
    precision_score, recall_score, f1_score, confusion_matrix,
classification_report
)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
ROOT = Path("/content/drive/MyDrive/eatsense")
OUT_WINDOWS = ROOT / "windows"

# load NPZs:
def _load_npz(p: Path):
    z = np.load(p, allow_pickle=True)
    X = z["X"]
    y = z["y"].astype(np.float32)
    return X, y

Xtr, ytr = _load_npz(OUT_WINDOWS / "Xy_train_pose.npz")
Xva, yva = _load_npz(OUT_WINDOWS / "Xy_val_pose.npz")
Xte, yte = _load_npz(OUT_WINDOWS / "Xy_test_pose.npz")

print("train", Xtr.shape, "val", Xva.shape, "test", Xte.shape)

# standardize (fit on train only):
```

```python
def zscore_fit(X):
    if X.ndim == 2:
        mu = X.mean(0, keepdims=True)
        sd = X.std(0, keepdims=True) + 1e-8
    else:  # (N, T, D) -> stats over N,T for each D
        mu = X.reshape(-1, X.shape[-1]).mean(0, keepdims=True)
        sd = X.reshape(-1, X.shape[-1]).std(0, keepdims=True) + 1e-8
    return mu, sd

def zscore_apply(X, mu, sd):
    if X.ndim == 2:
        return (X - mu) / sd
    else:
        return (X - mu.reshape(1,1,-1)) / sd.reshape(1,1,-1)

mu, sd = zscore_fit(Xtr)
Xtr = zscore_apply(Xtr, mu, sd).astype(np.float32)
Xva = zscore_apply(Xva, mu, sd).astype(np.float32)
Xte = zscore_apply(Xte, mu, sd).astype(np.float32)
```

```
train (62568, 12) val (21184, 12) test (14830, 12)
```

```python
# datasets:
class NPZDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X)
        self.y = torch.from_numpy(y).float()
    def __len__(self): return len(self.y)
    def __getitem__(self, i): return self.X[i], self.y[i]

train_ds = NPZDataset(Xtr, ytr)
val_ds   = NPZDataset(Xva, yva)
test_ds  = NPZDataset(Xte, yte)

train_loader = DataLoader(train_ds, batch_size=512, shuffle=True,
num_workers=2, pin_memory=True)
val_loader   = DataLoader(val_ds,   batch_size=1024, shuffle=False,
num_workers=2, pin_memory=True)
test_loader  = DataLoader(test_ds,  batch_size=1024, shuffle=False,
num_workers=2, pin_memory=True)

# Model:
class MLP(nn.Module):
    def __init__(self, in_dim, hidden=256, p=0.2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, hidden),
            nn.ReLU(),
            nn.Dropout(p),
            nn.Linear(hidden, hidden//2),
            nn.ReLU(),
            nn.Dropout(p),
            nn.Linear(hidden//2, 1),
        )
    def forward(self, x):
        return self.net(x).squeeze(-1)

class GRUHead(nn.Module):
    def __init__(self, feat_dim, hid=128, layers=1, p=0.2,
bidir=False):
        super().__init__()
        self.gru = nn.GRU(
            input_size=feat_dim, hidden_size=hid,
            num_layers=layers, batch_first=True, bidirectional=bidir
        )
        out_dim = hid * (2 if bidir else 1)
        self.head = nn.Sequential(nn.Dropout(p), nn.Linear(out_dim,
1))
    def forward(self, x):          # x: (B, T, D)
        out, h = self.gru(x)       # last hidden state:
        if isinstance(h, tuple):   # (for LSTM compatibility)
            h = h[0]
        h_last = h[-1]             # (B, H)
        return self.head(h_last).squeeze(-1)

def make_model(X_sample):
    if X_sample.ndim == 2:
        return MLP(in_dim=X_sample.shape[1])
    elif X_sample.ndim == 3:
        return GRUHead(feat_dim=X_sample.shape[-1], hid=128, layers=1,
p=0.3, bidir=False)
    else:
        raise ValueError(f"Unsupported X shape: {X_sample.shape}")

model = make_model(Xtr)
model.to(DEVICE)
print(model)
```

```
MLP(
  (net): Sequential(
    (0): Linear(in_features=12, out_features=256, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=256, out_features=128, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=128, out_features=1, bias=True)
  )
)
```

```python
# loss, optimizer, class weighting:
pos = float(ytr.sum())
neg = float(len(ytr) - pos)
pos_weight = torch.tensor(neg / (pos + 1e-8), device=DEVICE)
criterion = nn.BCEWithLogitsLoss(pos_weight=pos_weight)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4,
weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=10)

#training / evaluation helpers:
@torch.no_grad()
def predict_proba(loader):
    model.eval()
    probs = []
    for xb, _ in loader:
        xb = xb.to(DEVICE, non_blocking=True)
        if xb.ndim == 2:
            logit = model(xb)
        else:
            logit = model(xb)
        probs.append(torch.sigmoid(logit).cpu().numpy())
    return np.concatenate(probs)

def train_epoch():
    model.train()
    total = 0.0
    for xb, yb in train_loader:
        xb = xb.to(DEVICE, non_blocking=True)
        yb = yb.to(DEVICE, non_blocking=True)
        optimizer.zero_grad(set_to_none=True)
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        total += float(loss.item()) * len(yb)
    scheduler.step()
    return total / len(train_ds)

def pick_thr_max_f1(y_true, y_prob):
    p, r, th = precision_recall_curve(y_true, y_prob)
    f1 = 2*p*r/(p+r+1e-12)
    i = np.nanargmax(f1[:-1])
    return float(th[i]), float(f1[i])

def eval_split(y_true, y_prob, thr):
    y_hat = (y_prob >= thr).astype(int)
    out = {
        "roc_auc": roc_auc_score(y_true, y_prob) if
len(np.unique(y_true)) > 1 else np.nan,
        "pr_auc": average_precision_score(y_true, y_prob) if
len(np.unique(y_true)) > 1 else np.nan,
        "precision": precision_score(y_true, y_hat, zero_division=0),
        "recall": recall_score(y_true, y_hat, zero_division=0),
        "f1": f1_score(y_true, y_hat, zero_division=0),
        "cm": confusion_matrix(y_true, y_hat),
        "report": classification_report(y_true, y_hat, digits=3,
zero_division=0),
    }
    return out

# train loop with simple early stopping:
best_val = -np.inf
best_state = None
patience = 5
pat_count = 0
EPOCHS = 25

for epoch in range(1, EPOCHS+1):
    tr_loss = train_epoch()
    # monitor val F1 at optimal thr each epoch:
    p_va = predict_proba(val_loader)
    thr, f1_val = pick_thr_max_f1(yva, p_va)
    print(f"epoch {epoch:02d} | train loss {tr_loss:.4f} | val F1*
{f1_val:.3f} @ thr {thr:.3f}")
    if f1_val > best_val + 1e-4:
        best_val = f1_val
        best_state = {k: v.detach().cpu().clone() for k, v in
model.state_dict().items()}
        pat_count = 0
    else:
        pat_count += 1
        if pat_count >= patience:
            print("Early stopping.")
            break
```

```
/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)

epoch 01 | train loss 0.8149 | val F1* 0.420 @ thr 0.911

/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)

epoch 02 | train loss 0.3583 | val F1* 0.414 @ thr 0.955
```

```
/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)

epoch 03 | train loss 0.2624 | val F1* 0.407 @ thr 0.975

/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)

epoch 04 | train loss 0.2502 | val F1* 0.406 @ thr 0.962

/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)

epoch 05 | train loss 0.2028 | val F1* 0.395 @ thr 0.967

/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)

epoch 06 | train loss 0.1997 | val F1* 0.414 @ thr 0.970
Early stopping.

# load best weights:
if best_state is not None:
    model.load_state_dict(best_state)

# final evaluation:
p_va = predict_proba(val_loader)
thr, f1_val = pick_thr_max_f1(yva, p_va)
va_metrics = eval_split(yva, p_va, thr)

p_te = predict_proba(test_loader)
te_metrics = eval_split(yte, p_te, thr)

print("\n=== Deep baseline (PyTorch) ===")
print(f"Chosen threshold (max F1 on val): {thr:.3f} | best val F1*:
{f1_val:.3f}")
print(f"[VAL]  ROC AUC: {va_metrics['roc_auc']:.3f}  PR AUC:
{va_metrics['pr_auc']:.3f}  "
      f"P: {va_metrics['precision']:.3f}  R:
{va_metrics['recall']:.3f}  F1: {va_metrics['f1']:.3f}")
print(va_metrics["cm"])
print(va_metrics["report"])

print(f"[TEST] ROC AUC: {te_metrics['roc_auc']:.3f}  PR AUC:
```

```
{te_metrics['pr_auc']:.3f}  "
      f"P: {te_metrics['precision']:.3f}  R:
{te_metrics['recall']:.3f}  F1: {te_metrics['f1']:.3f}")
print(te_metrics["cm"])
print(te_metrics["report"])
```

```
/usr/local/lib/python3.12/dist-packages/torch/utils/data/
dataloader.py:666: UserWarning: 'pin_memory' argument is set as true
but no accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)
/usr/local/lib/python3.12/dist-packages/torch/utils/data/dataloader.py
:666: UserWarning: 'pin_memory' argument is set as true but no
accelerator is found, then device pinned memory won't be used.
  warnings.warn(warn_msg)


=== Deep baseline (PyTorch) ===
Chosen threshold (max F1 on val): 0.911 | best val F1*: 0.420
[VAL]  ROC AUC: 0.992  PR AUC: 0.360  P: 0.403  R: 0.439  F1: 0.420
[[21090    37]
 [   32    25]]
              precision    recall  f1-score   support

         0.0      0.998     0.998     0.998     21127
         1.0      0.403     0.439     0.420        57

    accuracy                          0.997     21184
   macro avg      0.701     0.718     0.709     21184
weighted avg      0.997     0.997     0.997     21184

[TEST] ROC AUC: 0.895  PR AUC: 0.640  P: 0.981  R: 0.573  F1: 0.723
[[14740     1]
 [   38    51]]
              precision    recall  f1-score   support

         0.0      0.997     1.000     0.999     14741
         1.0      0.981     0.573     0.723        89

    accuracy                          0.997     14830
   macro avg      0.989     0.786     0.861     14830
weighted avg      0.997     0.997     0.997     14830
```

Summary Highlights:

This is a baseline sanity check to prove the pipeline, windows, labels, and training loop work end-to-end. It's doing its job: metrics make sense, confusion matrices look sane, and there's no obvious leakage:

- Works end-to-end: windows → features → training → eval are all wired correctly.

- Val vs. Test gap: test does better (higher PR AUC/F1). Likely due to:
  - Slightly higher positive rate in test, and/or
  - Test clips that are easier (cleaner movements). This isn't a red flag—just a reminder that PR AUC is sensitive to prevalence and difficulty.
- Thresholding: we're using a val-optimized threshold and applying it to test (good).
- No leakage signals: very low FP rates and reasonable recall given the scarcity of positives.

Save Artifacts

```
MODEL_DIR = OUT_WINDOWS / "artifacts"; MODEL_DIR.mkdir(parents=True, 
exist_ok=True)
torch.save(model.state_dict(), MODEL_DIR / "baseline_pose_gru.pt")
np.savez(MODEL_DIR / "zscore_pose.npz", mu=mu, sd=sd)
```