pamap_project

February 21, 2026

1 Imports

[50]:
```python
import re
import numpy as np
import joblib
from pathlib import Path
import json
from sklearn.model_selection import GroupKFold
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.pipeline import Pipeline
from sklearn.metrics import f1_score, balanced_accuracy_score,␣
↪classification_report, confusion_matrix, accuracy_score,␣
↪ConfusionMatrixDisplay
from sklearn.model_selection import RandomizedSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.base import clone
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier,␣
↪HistGradientBoostingClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
```

2 EDA SECTION

2.0.1 Load PAMAP2 Merged Dataset

[2]:
```python
df = pd.read_parquet("E:/AAI/AAI530-Data Analytics and Internet of Things/
↪PAMAP2_Dataset/pamap2_merged.parquet")
```

[3]:
```python
# Confirm Dataset Structure
df.shape
```

[3]:
```
(2872533, 56)
```

[4]:
```python
df.columns.tolist()
```

[4]:
```
['subject_id',
 'timestamp',
 'activity_id_1',
 'activity_id_2',
 'heart_rate',
 'hand_temp',
 'hand_acc16_x',
 'hand_acc16_y',
 'hand_acc16_z',
 'hand_acc6_x',
 'hand_acc6_y',
 'hand_acc6_z',
 'hand_gyro_x',
 'hand_gyro_y',
 'hand_gyro_z',
 'hand_mag_x',
 'hand_mag_y',
 'hand_mag_z',
 'hand_orient_1',
 'hand_orient_2',
 'hand_orient_3',
 'hand_orient_4',
 'chest_temp',
 'chest_acc16_x',
 'chest_acc16_y',
 'chest_acc16_z',
 'chest_acc6_x',
 'chest_acc6_y',
 'chest_acc6_z',
 'chest_gyro_x',
 'chest_gyro_y',
 'chest_gyro_z',
 'chest_mag_x',
 'chest_mag_y',
 'chest_mag_z',
 'chest_orient_1',
 'chest_orient_2',
 'chest_orient_3',
 'chest_orient_4',
 'ankle_temp',
 'ankle_acc16_x',
 'ankle_acc16_y',
 'ankle_acc16_z',
 'ankle_acc6_x',
 'ankle_acc6_y',
```

```
 'ankle_acc6_z',
 'ankle_gyro_x',
 'ankle_gyro_y',
 'ankle_gyro_z',
 'ankle_mag_x',
 'ankle_mag_y',
 'ankle_mag_z',
 'ankle_orient_1',
 'ankle_orient_2',
 'ankle_orient_3',
 'ankle_orient_4']
```

2.0.2 Data Inspection

[5]:
```
df.head()
```

[5]:
```
   subject_id  timestamp  activity_id_1  activity_id_2  heart_rate  hand_temp  \
0         101       8.38              0          104.0        30.0    2.37223
1         101       8.39              0            NaN        30.0    2.18837
2         101       8.40              0            NaN        30.0    2.37357
3         101       8.41              0            NaN        30.0    2.07473
4         101       8.42              0            NaN        30.0    2.22936

   hand_acc16_x  hand_acc16_y  hand_acc16_z  hand_acc6_x  …  ankle_gyro_x  \
0       8.60074       3.51048       2.43954      8.76165  …      0.009250
1       8.56560       3.66179       2.39494      8.55081  …     -0.004638
2       8.60107       3.54898       2.30514      8.53644  …      0.000148
3       8.52853       3.66021       2.33528      8.53622  …     -0.020301
4       8.83122       3.70000       2.23055      8.59741  …     -0.014303

   ankle_gyro_y  ankle_gyro_z  ankle_mag_x  ankle_mag_y  ankle_mag_z  \
0     -0.017580      -61.1888     -38.9599     -58.1438          1.0
1      0.000368      -59.8479     -38.8919     -58.5253          1.0
2      0.022495      -60.7361     -39.4138     -58.3999          1.0
3      0.011275      -60.4091     -38.7635     -58.3956          1.0
4     -0.002823      -61.5199     -39.3879     -58.2694          1.0

   ankle_orient_1  ankle_orient_2  ankle_orient_3  ankle_orient_4
0             0.0             0.0             0.0             NaN
1             0.0             0.0             0.0             NaN
2             0.0             0.0             0.0             NaN
3             0.0             0.0             0.0             NaN
4             0.0             0.0             0.0             NaN

[5 rows x 56 columns]
```

[6]:
```
df.dtypes
```

[6]:
```
subject_id            int64
timestamp           float64
activity_id_1         int64
activity_id_2       float64
heart_rate          float64
hand_temp           float64
hand_acc16_x        float64
hand_acc16_y        float64
hand_acc16_z        float64
hand_acc6_x         float64
hand_acc6_y         float64
hand_acc6_z         float64
hand_gyro_x         float64
hand_gyro_y         float64
hand_gyro_z         float64
hand_mag_x          float64
hand_mag_y          float64
hand_mag_z          float64
hand_orient_1       float64
hand_orient_2       float64
hand_orient_3       float64
hand_orient_4       float64
chest_temp          float64
chest_acc16_x       float64
chest_acc16_y       float64
chest_acc16_z       float64
chest_acc6_x        float64
chest_acc6_y        float64
chest_acc6_z        float64
chest_gyro_x        float64
chest_gyro_y        float64
chest_gyro_z        float64
chest_mag_x         float64
chest_mag_y         float64
chest_mag_z         float64
chest_orient_1      float64
chest_orient_2      float64
chest_orient_3      float64
chest_orient_4      float64
ankle_temp          float64
ankle_acc16_x       float64
ankle_acc16_y       float64
ankle_acc16_z       float64
ankle_acc6_x        float64
ankle_acc6_y        float64
ankle_acc6_z        float64
ankle_gyro_x        float64
```

```
ankle_gyro_y      float64
ankle_gyro_z      float64
ankle_mag_x       float64
ankle_mag_y       float64
ankle_mag_z       float64
ankle_orient_1    float64
ankle_orient_2    float64
ankle_orient_3    float64
ankle_orient_4    float64
dtype: object
```

[7]:
```
# Create new datetime column (missing) because that's what Pandas expects
df['datetime'] = pd.to_datetime(df['timestamp'], unit='s')
```

[8]:
```
df[['timestamp','datetime']].head()
df.dtypes
```

[8]:
```
subject_id                  int64
timestamp                 float64
activity_id_1               int64
activity_id_2             float64
heart_rate                float64
hand_temp                 float64
hand_acc16_x              float64
hand_acc16_y              float64
hand_acc16_z              float64
hand_acc6_x               float64
hand_acc6_y               float64
hand_acc6_z               float64
hand_gyro_x               float64
hand_gyro_y               float64
hand_gyro_z               float64
hand_mag_x                float64
hand_mag_y                float64
hand_mag_z                float64
hand_orient_1             float64
hand_orient_2             float64
hand_orient_3             float64
hand_orient_4             float64
chest_temp                float64
chest_acc16_x             float64
chest_acc16_y             float64
chest_acc16_z             float64
chest_acc6_x              float64
chest_acc6_y              float64
chest_acc6_z              float64
chest_gyro_x              float64
```

```
chest_gyro_y              float64
chest_gyro_z              float64
chest_mag_x               float64
chest_mag_y               float64
chest_mag_z               float64
chest_orient_1            float64
chest_orient_2            float64
chest_orient_3            float64
chest_orient_4            float64
ankle_temp                float64
ankle_acc16_x             float64
ankle_acc16_y             float64
ankle_acc16_z             float64
ankle_acc6_x              float64
ankle_acc6_y              float64
ankle_acc6_z              float64
ankle_gyro_x              float64
ankle_gyro_y              float64
ankle_gyro_z              float64
ankle_mag_x               float64
ankle_mag_y               float64
ankle_mag_z               float64
ankle_orient_1            float64
ankle_orient_2            float64
ankle_orient_3            float64
ankle_orient_4            float64
datetime           datetime64[ns]
dtype: object
```

[9]:
```
# Sort by subject and time
df = df.sort_values(['subject_id', 'datetime']).reset_index(drop=True)
```

[10]:
```
df.groupby('subject_id')['timestamp'].apply(lambda x: x.
 ↪is_monotonic_increasing).head()
```

[10]:
```
subject_id
101    True
102    True
103    True
104    True
105    True
Name: timestamp, dtype: bool
```

2.0.3 Missing Values

[11]:
```
# Check missing values
df.isna().sum().sort_values(ascending=False).head(15)
```

[11]:
```
ankle_orient_4    2872533
activity_id_2     2610265
hand_acc16_y        13141
hand_acc16_x        13141
heart_rate          13141
hand_acc16_z        13141
hand_acc6_z         13141
hand_acc6_y         13141
hand_temp           13141
hand_mag_z          13141
hand_orient_1       13141
hand_orient_2       13141
hand_gyro_x         13141
hand_gyro_y         13141
hand_gyro_z         13141
dtype: int64
```

[12]:
```
# Forward-fill missing values per subject
numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns

df[numeric_cols] = df.groupby('subject_id')[numeric_cols].ffill()
```

[13]:
```
# Back-fill remaining leading NaNs per subject
df[numeric_cols] = df.groupby('subject_id')[numeric_cols].bfill()
```

[14]:
```
# Confirm missing values are gone
df.isna().sum().sum()
```

[14]:
```
np.int64(2872533)
```

[15]:
```
# Find which columns still contain NaNs
df.isna().sum().sort_values(ascending=False).head(10)
```

[15]:
```
ankle_orient_4    2872533
timestamp               0
activity_id_1           0
activity_id_2           0
heart_rate              0
hand_temp               0
hand_acc16_x            0
hand_acc16_y            0
subject_id              0
hand_acc6_x             0
```

dtype: int64

[16]:
```
df.isna().sum().sort_values(ascending=False).head(15)
```

[16]:
```
ankle_orient_4    2872533
timestamp               0
activity_id_1           0
activity_id_2           0
heart_rate              0
hand_temp               0
hand_acc16_x            0
hand_acc16_y            0
subject_id              0
hand_acc6_x             0
hand_acc6_y             0
hand_acc6_z             0
hand_gyro_x             0
hand_gyro_y             0
hand_gyro_z             0
dtype: int64
```

We can see that ankle_orient_4 is 100% missing. This is normal and documented behavior for the PAMAP2 dataset. So let's drop it entirely.

[17]:
```
df = df.drop(columns=['ankle_orient_4'])
```

[18]:
```
df.isna().sum().sum()
```

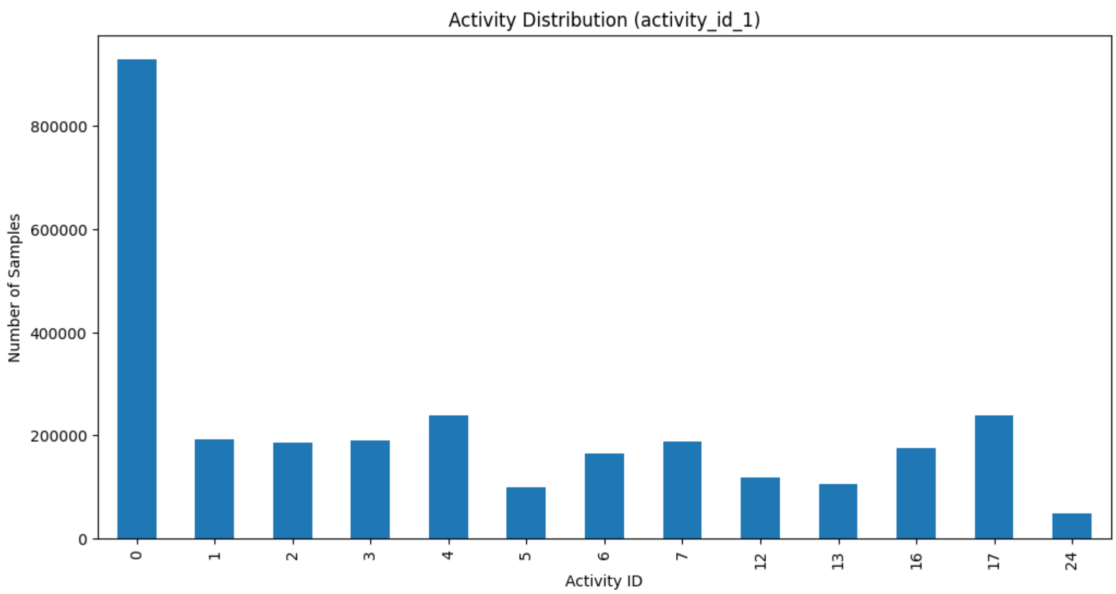
[18]:
```
np.int64(0)
```

Perfect, nothing missing now!

2.1 Start of EDA...

2.1.1 Class Imbalance

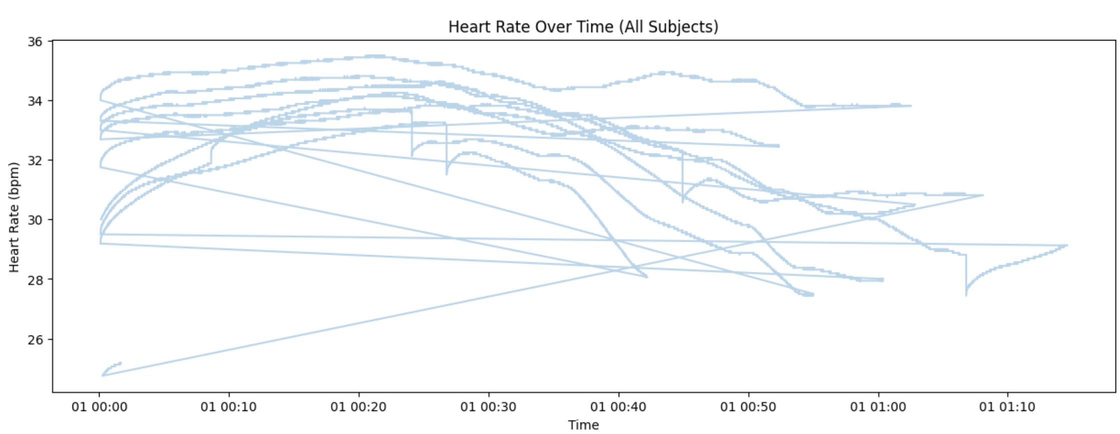
[19]:
```
plt.figure(figsize=(12,6))
df['activity_id_1'].value_counts().sort_index().plot(kind='bar')
plt.title("Activity Distribution (activity_id_1)")
plt.xlabel("Activity ID")
plt.ylabel("Number of Samples")
plt.show()
```



EDA conclusion so far is that dataset is highly imbalanced, dominated by 0 (unknown).

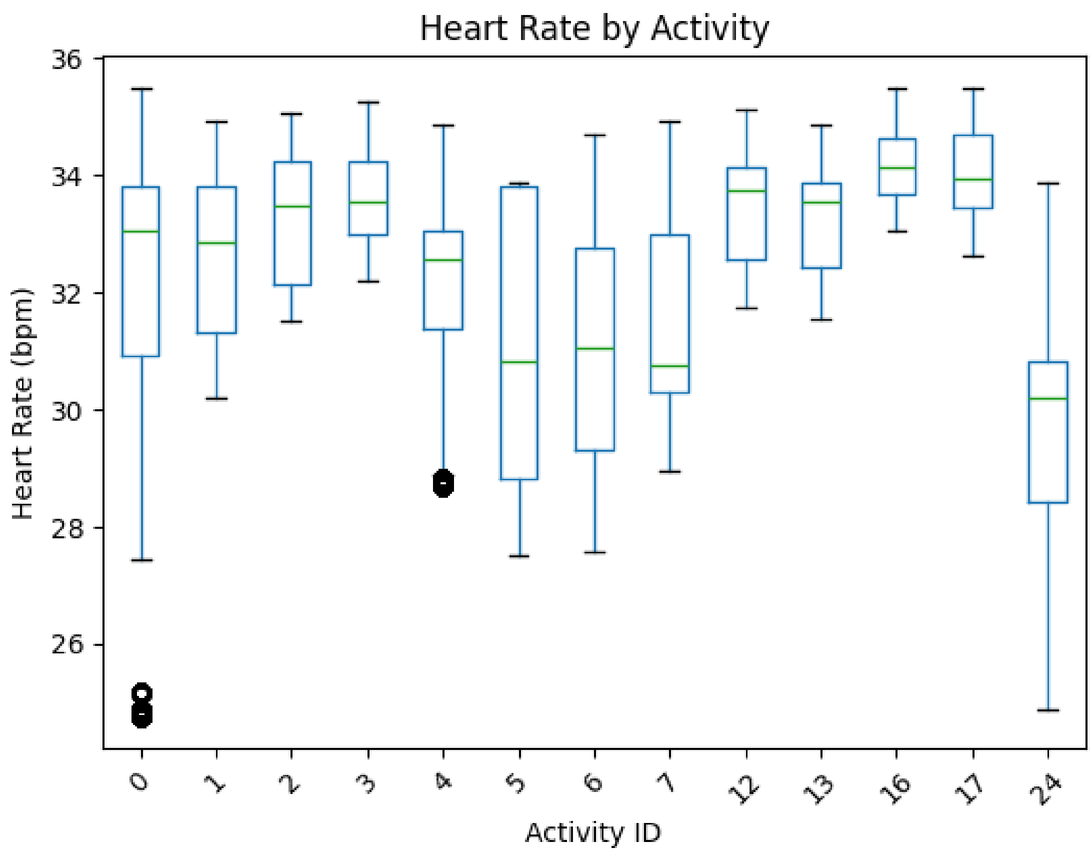
2.1.2 Heart Rate Analysis

[20]:
```python
# Plot heart rate over time
plt.figure(figsize=(15,5))
plt.plot(df['datetime'], df['heart_rate'], alpha=0.3)
plt.title("Heart Rate Over Time (All Subjects)")
plt.xlabel("Time")
plt.ylabel("Heart Rate (bpm)")
plt.show()
```



[21]:
```python
plt.figure(figsize=(12,6))
df.boxplot(column='heart_rate', by='activity_id_1', grid=False, rot=45)
plt.title("Heart Rate by Activity")
plt.suptitle("")  # removes extra automatic title
plt.xlabel("Activity ID")
plt.ylabel("Heart Rate (bpm)")
plt.show()
```

```
<Figure size 1200x600 with 0 Axes>
```



Conclusion for Heart-Rate EDA:

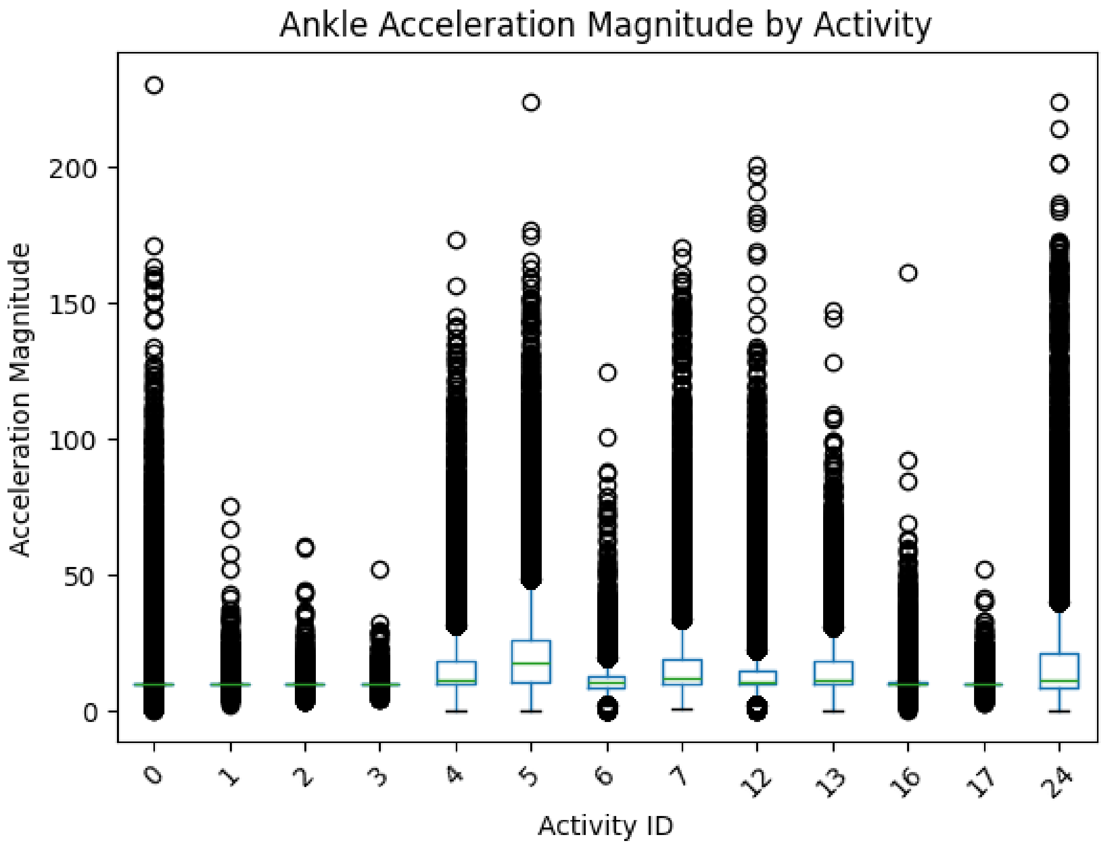
- Activity labels are valid
- Activity 24 is a "mixed behavior" label
- HR separates clearly for major activities
- Activity 0 and 24 behave like "background/transition"
- Nothing appears strange or erroneous

2.1.3 Acceleration Magnitude

[22]:
```python
# Compute Acceleration Magnitude
df['hand_acc_mag'] = np.sqrt(
    df['hand_acc16_x']**2 +
    df['hand_acc16_y']**2 +
    df['hand_acc16_z']**2
)

df['chest_acc_mag'] = np.sqrt(
    df['chest_acc16_x']**2 +
    df['chest_acc16_y']**2 +
    df['chest_acc16_z']**2
)

df['ankle_acc_mag'] = np.sqrt(
    df['ankle_acc16_x']**2 +
    df['ankle_acc16_y']**2 +
    df['ankle_acc16_z']**2
)
```

[23]:
```python
plt.figure(figsize=(12,6))
df.boxplot(column='ankle_acc_mag', by='activity_id_1', grid=False, rot=45)
plt.title("Ankle Acceleration Magnitude by Activity")
plt.suptitle("")
plt.xlabel("Activity ID")
plt.ylabel("Acceleration Magnitude")
plt.show()
```

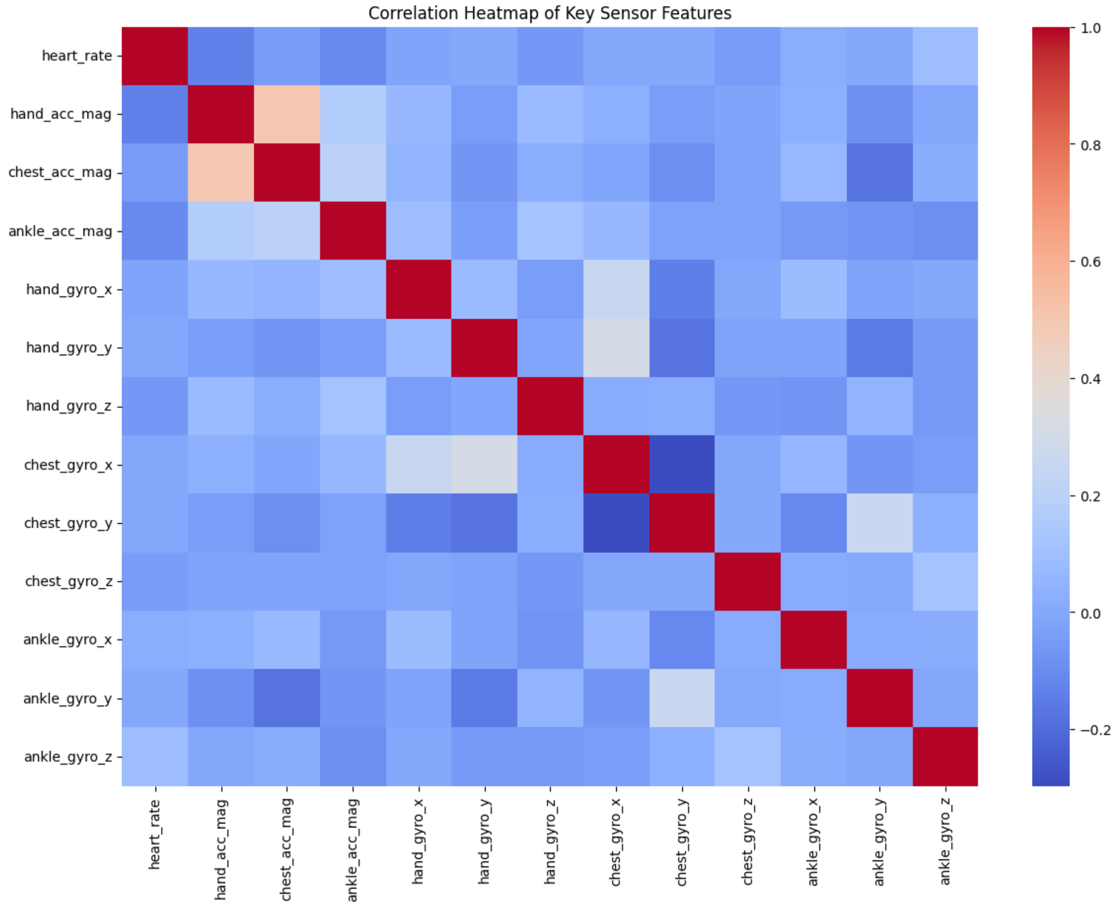
```
<Figure size 1200x600 with 0 Axes>
```



Conclusion for Acceleration Magnitude: - Activities have clear separability - Activity 0 is noisy and non-specific

2.1.4 Correlation Heatmap

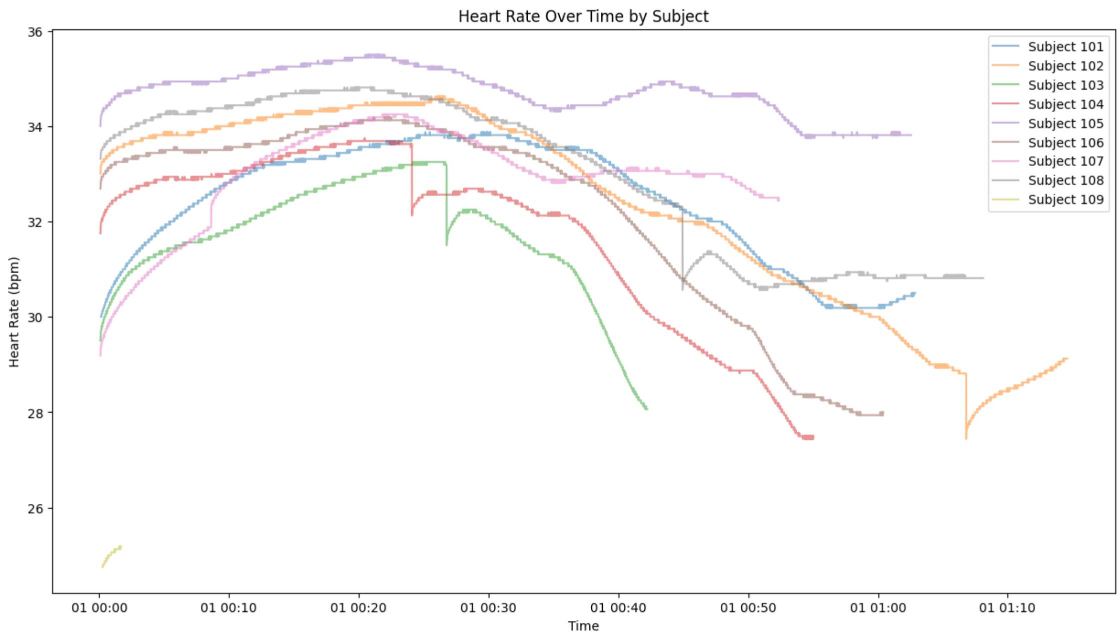
[24]:
```
# Correlation Heatmap
plt.figure(figsize=(14,10))

# Use only a subset to avoid memory overload
subset = df[['heart_rate',
             'hand_acc_mag', 'chest_acc_mag', 'ankle_acc_mag',
             'hand_gyro_x', 'hand_gyro_y', 'hand_gyro_z',
             'chest_gyro_x', 'chest_gyro_y', 'chest_gyro_z',
             'ankle_gyro_x', 'ankle_gyro_y', 'ankle_gyro_z']]

corr = subset.corr()

sns.heatmap(corr, annot=False, cmap="coolwarm")
plt.title("Correlation Heatmap of Key Sensor Features")
plt.show()
```



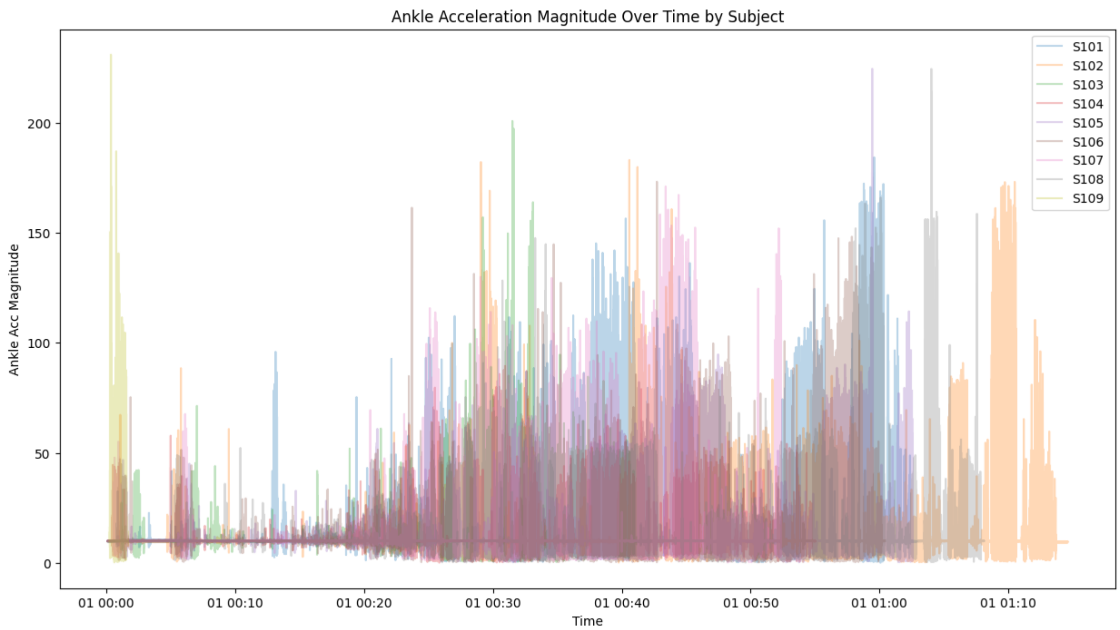
Conclusion for Heat Map: - Hand_acc_mag has the highest correlation with heart rate (but still weak) - Ankle_acc_mag has the weakest (slight negative) - Weak HR correlations

2.1.5 Subject Drift & Individual Differences

[25]:
```
# Subject Drift & Individual Differences
# Heart Rate Drift Per Subject
plt.figure(figsize=(15,8))

for sid in df['subject_id'].unique():
    sub = df[df['subject_id'] == sid]
    plt.plot(sub['datetime'], sub['heart_rate'], alpha=0.5, label=f"Subject␣
 ↪{sid}")

plt.title("Heart Rate Over Time by Subject")
plt.xlabel("Time")
plt.ylabel("Heart Rate (bpm)")
plt.legend()
plt.show()
```



[26]:
```python
# Accelerometer Drift (Ankle)
plt.figure(figsize=(15,8))

for sid in df['subject_id'].unique():
    sub = df[df['subject_id'] == sid]
    plt.plot(sub['datetime'], sub['ankle_acc_mag'], alpha=0.3, label=f"S{sid}")

plt.title("Ankle Acceleration Magnitude Over Time by Subject")
plt.xlabel("Time")
plt.ylabel("Ankle Acc Magnitude")
plt.legend()
plt.show()
```



[27]:
```python
# Subject-Level Summary Statistics
subject_summary = df.groupby('subject_id').agg({
    'heart_rate': ['mean','std','min','max'],
    'ankle_acc_mag': ['mean','std'],
    'chest_acc_mag': ['mean','std'],
    'hand_acc_mag': ['mean','std']
})

subject_summary
```

[27]:

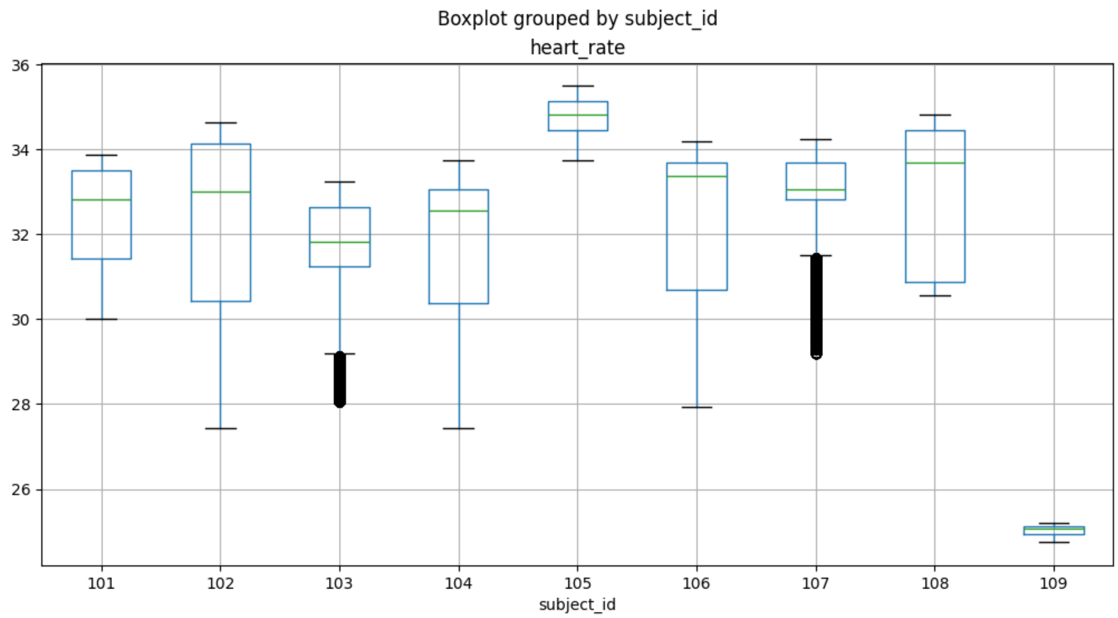
	heart_rate				ankle_acc_mag		\
	mean	std	min	max	mean	std	
subject_id							
101	32.428912	1.236819	30.0000	33.8750	12.029074	6.996151	
102	32.241076	2.103383	27.4375	34.6250	11.596222	6.191083	
103	31.694351	1.191588	28.0625	33.2500	11.656435	5.894083	
104	31.793707	1.795173	27.4375	33.7500	11.362009	4.521276	
105	34.726806	0.488918	33.7500	35.5000	12.271062	6.666630	
106	32.235847	2.042913	27.9375	34.1875	11.968420	6.701648	
107	32.942459	1.031391	29.1875	34.2500	12.109440	7.133284	
108	32.964541	1.618991	30.5625	34.8125	12.129373	7.237526	
109	25.017555	0.115631	24.7500	25.1875	16.828030	14.224496	

	chest_acc_mag		hand_acc_mag	
	mean	std	mean	std
subject_id				

```
101            10.191324   3.975436    10.773256  5.023797
102            10.152105   2.994420    10.555947  3.118693
103             9.922849   2.177478    10.529987  2.480054
104             9.955185   2.126014    10.182421  2.487374
105            10.190863   3.600012    10.731602  4.985392
106            10.109868   3.362223    11.390808  8.119901
107             9.982908   2.292810    10.213617  3.864200
108            10.150360   4.057130    10.759876  5.116739
109            11.906719  10.657747    13.779229  8.777914
```

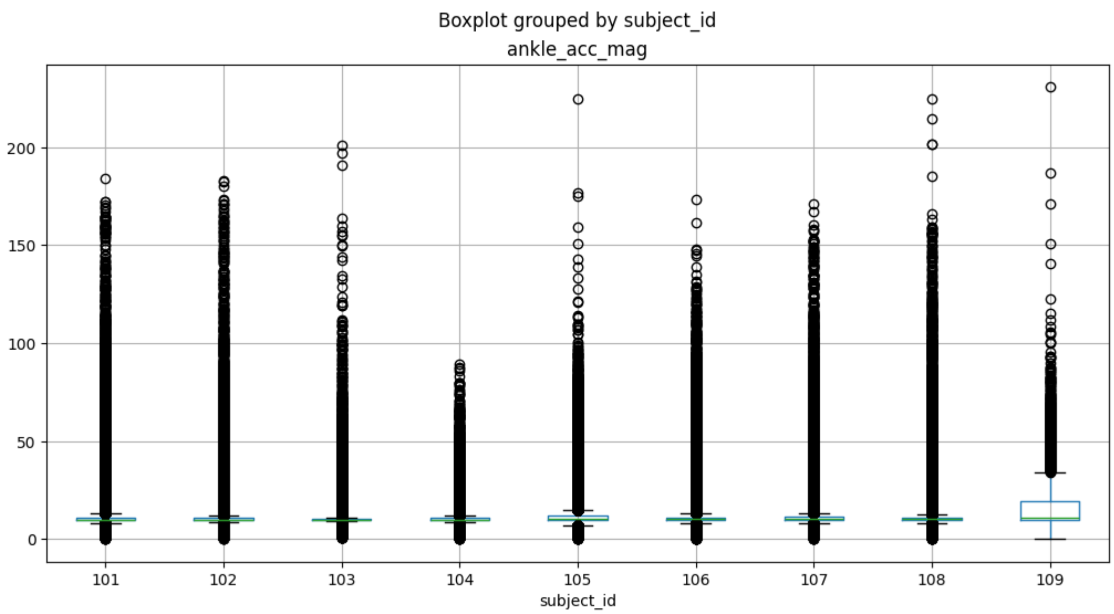
[28]:
```
# Box Plots Per Subject
# Heart Rate Boxplot
df.boxplot(column='heart_rate', by='subject_id', figsize=(12,6))
```

[28]: <Axes: title={'center': 'heart_rate'}, xlabel='subject_id'>



[29]:
```
# Acceleration Boxplot
df.boxplot(column='ankle_acc_mag', by='subject_id', figsize=(12,6))
```

[29]: <Axes: title={'center': 'ankle_acc_mag'}, xlabel='subject_id'>



Conclusion: - Some subjects have higher HR baselines - Some subjects have stronger movement patterns - There is drift over time

All are normal and expected.

2.1.6 EDA SUMMARY FOR PAMAP2 WEARABLE SENSOR DATASET

1. Dataset Overview

The merged dataset contains 2,872,533 observations and 55 usable columns after dropping the all-NaN ankle_orient_4 field. It includes:

Heart rate

IMU data from hand, chest, and ankle sensors (accelerometer, gyroscope, magnetometer, orientation)

Activity labels

Subject identifiers

A timestamp and derived datetime column

The dataset represents nine participants performing various daily-life and fitness activities, sampled at 100 Hz.

2. Data Cleaning Steps

The following cleaning operations were completed:

Combined all nine subject files into one dataset with a subject_id column

Converted timestamp into a usable datetime column

Sorted data by subject_id and datetime

Identified and handled missing values using forward fill followed by backward fill within each subject

Dropped ankle_orient_4 because it contained 100% missing values

Verified no remaining NaNs in the dataset

The cleaned dataset is complete, chronologically ordered, and free of missing sensor values.

3. Activity Distribution

The activity labels present in the dataset were:

0, 1, 2, 3, 4, 5, 6, 7, 12, 13, 16, 17, 24

Mapped to:

ID Activity 0 Unlabeled / transition activity 1 Lying 2 Sitting 3 Standing 4 Walking 5 Running 6 Cycling 7 Nordic Walking 12 Vacuum Cleaning 13 Ironing 16 Rope Jumping 17 Misc. household activity 24 Sensor warm-up / transition Key Findings:

Activity 0 dominates the dataset, representing long transitional or unlabeled periods.

Structured activities (walking, running, vacuuming, etc.) are present but imbalanced.

This imbalance will require attention during ML model preparation.

4. Heart Rate Analysis

Two heart rate EDA plots were generated:

Findings:

Heart rate shows clear separation between activities, with higher-intensity activities (running, rope jumping) producing higher HR, and low-intensity activities (lying, sitting) producing lower HR.

Activity 0 has a wide HR range, confirming it mixes multiple behavioral states.

Activity 24 shows an unusually wide HR range, consistent with warm-up/movement-between-tasks behavior.

This confirms the dataset's physiological signals are valid and activity-dependent.

5. Acceleration Magnitude Analysis

Acceleration magnitude (vector norms) was computed for hand, chest, and ankle sensors.

Findings:

Activities show clear separability based on acceleration magnitude.

Walking, running, cycling, and rope jumping produce high ankle acceleration.

Vacuum cleaning and ironing produce strong hand acceleration.

Chest sensor captures full-body movement with smoother patterns.

Activity 0 again shows a wide range, reinforcing its transitional nature.

These patterns support use of accelerometers for activity recognition.

6. Correlation Heatmap Analysis Findings:

Heart rate shows only weak correlations with motion features, as expected:

Small positive correlation with hand movement

Smaller or slightly negative correlation with ankle movement

Strong correlations exist within each sensor group (hand, chest, ankle), especially among gyro axes.

Low correlations between sensors (hand chest ankle), which is expected since they track different body parts.

This confirms the signals are independent, well-behaved, and provide unique information.

7. Subject Drift and Individual Differences

Plots were generated for heart rate and ankle acceleration per subject.

Findings:

Subjects have different HR baselines, consistent with individual physiology.

Subjects display different movement intensities, reflecting natural biomechanical differences.

Some subjects show drift over time in HR or movement, consistent with:

sensor warming

fatigue accumulation

strap adjustments

No major noise, artifacts, or sensor failures were detected.

These findings suggest that per-subject normalization may be useful for ML modeling, especially for heart rate.

Notes:

- Some subjects have higher HR baselines. So be sure to normalize heart rate per subject. Here's the code for feature engineering:

df['hr_norm'] = df.groupby('subject_id')['heart_rate'].transform(lambda x: (x - x.mean()) / x.std())

3 MODELING SECTION

3.1 Train/Test Split and Validation

[30]:
```python
#Filter out transition labels for a clean activity classifier
ACTIVITIES_TO_USE = sorted([a for a in df['activity_id_1'].unique() if a not in (0, 24)])
df_model = df[df['activity_id_1'].isin(ACTIVITIES_TO_USE)].copy()
df_model['activity_id_1'] = df_model['activity_id_1'].astype(int)

#Choose test subjects (20-30% of subjects)
```

```python
rng = np.random.RandomState(42)
subjects = df_model['subject_id'].dropna().unique()
subjects = np.array(sorted(subjects))

test_frac = 0.2
n_test = max(1, int(round(len(subjects) * test_frac)))
test_subjects = rng.choice(subjects, size=n_test, replace=False)

train_subjects = np.array([s for s in subjects if s not in test_subjects])

df_train = df_model[df_model['subject_id'].isin(train_subjects)].copy()
df_test  = df_model[df_model['subject_id'].isin(test_subjects)].copy()

print("All subjects:", subjects)
print("Train subjects:", train_subjects, "(", len(train_subjects), ")")
print("Test subjects :", test_subjects, "(", len(test_subjects), ")")
print("Train rows:", len(df_train), "Test rows:", len(df_test))
print("\nTrain activity distribution (top 10):")
print(df_train['activity_id_1'].value_counts().head(10))
print("\nTest activity distribution (top 10):")
print(df_test['activity_id_1'].value_counts().head(10))
```

```
All subjects: [101 102 103 104 105 106 107 108]
Train subjects: [101 103 104 105 107 108] ( 6 )
Test subjects : [102 106] ( 2 )
Train rows: 1393585 Test rows: 499927

Train activity distribution (top 10):
activity_id_1
4     180507
17    172066
1     145753
3     139999
2     139802
16    133592
7     131682
6     119006
12     86583
13     78459
Name: count, dtype: int64

Test activity distribution (top 10):
activity_id_1
17    66624
4     58254
7     56425
3     49932
1     46770
```

```
6     45594
2     45386
16    41761
5     32063
12    30633
Name: count, dtype: int64
```

[31]:

```python
if 'activity_id_1' in df_train.columns: # Remove any remaining transition labels
    df_train = df_train[~df_train['activity_id_1'].isin([0, 24])].copy()
    df_test  = df_test[~df_test['activity_id_1'].isin([0, 24])].copy()
    df_train['activity_id_1'] = df_train['activity_id_1'].astype(int)
    df_test['activity_id_1']  = df_test['activity_id_1'].astype(int)

for c in ['subject_id','activity_id_1','datetime']: # Ensure key columns exist
    assert c in df_train.columns and c in df_test.columns, f"Missing {c}"

print("Train subjects:", sorted(df_train['subject_id'].unique()))
print("Test subjects :", sorted(df_test['subject_id'].unique()))
print("Train rows:", len(df_train), "Test rows:", len(df_test))
print("Train activity counts (top 10):")
print(df_train['activity_id_1'].value_counts().head(10))
print("Test activity counts (top 10):")
print(df_test['activity_id_1'].value_counts().head(10))
```

```
Train subjects: [np.int64(101), np.int64(103), np.int64(104), np.int64(105),
np.int64(107), np.int64(108)]
Test subjects : [np.int64(102), np.int64(106)]
Train rows: 1393585 Test rows: 499927
Train activity counts (top 10):
activity_id_1
4     180507
17    172066
1     145753
3     139999
2     139802
16    133592
7     131682
6     119006
12     86583
13     78459
Name: count, dtype: int64
Test activity counts (top 10):
activity_id_1
17    66624
4     58254
7     56425
3     49932
1     46770
```

```
6     45594
2     45386
16    41761
5     32063
12    30633
Name: count, dtype: int64
```

3.2 Features

[32]:
```
# Ensure acceleration magnitudes exist
def ensure_acc_magnitudes(dfx: pd.DataFrame) -> pd.DataFrame:
    dfx = dfx.copy()
    needed = [
        ('hand_acc16_x','hand_acc16_y','hand_acc16_z','hand_acc_mag'),
        ('chest_acc16_x','chest_acc16_y','chest_acc16_z','chest_acc_mag'),
        ('ankle_acc16_x','ankle_acc16_y','ankle_acc16_z','ankle_acc_mag')
    ]
    for ax, ay, az, mag in needed: # Compute magnitude if missing
        if mag not in dfx.columns and all(c in dfx.columns for c in [ax, ay,␣
↪az]):
            dfx[mag] = np.sqrt(dfx[ax]**2 + dfx[ay]**2 + dfx[az]**2)
    return dfx

# Add normalized heart rate per subject
def add_hr_norm(dfx: pd.DataFrame, hr_col='heart_rate') -> pd.DataFrame:
    dfx = dfx.copy()
    if hr_col in dfx.columns:
        dfx['hr_norm'] = dfx.groupby('subject_id')[hr_col].transform(
            lambda x: (x - x.mean()) / (x.std() + 1e-8)
        ) # avoid division by zero
    else: # heart_rate column missing
        dfx['hr_norm'] = np.nan
        print("WARNING: heart_rate not found; hr_norm will be NaN")
    return dfx

df_train = ensure_acc_magnitudes(df_train)
df_test  = ensure_acc_magnitudes(df_test)

df_train = add_hr_norm(df_train)
df_test  = add_hr_norm(df_test)

df_train = df_train.sort_values(['subject_id','datetime']).
↪reset_index(drop=True)
df_test  = df_test.sort_values(['subject_id','datetime']).reset_index(drop=True)

print("Added magnitudes + hr_norm. Train/Test shapes:", df_train.shape, df_test.
↪shape)
```

```
Added magnitudes + hr_norm. Train/Test shapes: (1393585, 60) (499927, 60)
```

[33]:

```python
# Select Feature Columns
def pick_sensor_columns(df):
    base = ['hr_norm', 'hand_acc_mag', 'chest_acc_mag', 'ankle_acc_mag']

    acc_axes = [c for c in df.columns if re.
↪search(r'(hand|chest|ankle)_acc16_[xyz]$', c)]
    gyro_axes = [c for c in df.columns if re.
↪search(r'(hand|chest|ankle)_gyro_[xyz]$', c)]
    mag_axes  = [c for c in df.columns if re.
↪search(r'(hand|chest|ankle)_mag_[xyz]$', c)]
    temp_cols = [c for c in df.columns if re.
↪search(r'(hand|chest|ankle)_temp$', c)]

    cols = []
    for lst in [base, acc_axes, gyro_axes, mag_axes, temp_cols]:
        cols.extend([c for c in lst if c in df.columns])

    # remove duplicates preserving order
    seen = set()
    cols = [c for c in cols if not (c in seen or seen.add(c))]
    return cols

feature_cols = pick_sensor_columns(df_train)
feature_cols = [c for c in feature_cols if c in df_train.columns and c in␣
↪df_test.columns]

print("Feature columns used:", len(feature_cols))
print(feature_cols[:40])

# drop NaNs on selected features
df_train = df_train.dropna(subset=feature_cols +␣
↪['activity_id_1','subject_id']).copy()
df_test  = df_test.dropna(subset=feature_cols + ['activity_id_1','subject_id']).
↪copy()

print("After dropna -> Train rows:", len(df_train), "Test rows:", len(df_test))
```

```
Feature columns used: 34
['hr_norm', 'hand_acc_mag', 'chest_acc_mag', 'ankle_acc_mag', 'hand_acc16_x',
'hand_acc16_y', 'hand_acc16_z', 'chest_acc16_x', 'chest_acc16_y',
'chest_acc16_z', 'ankle_acc16_x', 'ankle_acc16_y', 'ankle_acc16_z',
'hand_gyro_x', 'hand_gyro_y', 'hand_gyro_z', 'chest_gyro_x', 'chest_gyro_y',
'chest_gyro_z', 'ankle_gyro_x', 'ankle_gyro_y', 'ankle_gyro_z', 'hand_mag_x',
'hand_mag_y', 'hand_mag_z', 'chest_mag_x', 'chest_mag_y', 'chest_mag_z',
'ankle_mag_x', 'ankle_mag_y', 'ankle_mag_z', 'hand_temp', 'chest_temp',
'ankle_temp']
```

```
After dropna -> Train rows: 1393585 Test rows: 499927
```

[34]:
```python
'''
Downsampling: Makes window feature generation computationally feasible.
'''
DOWNSAMPLE_FACTOR = 10
# Downsample per subject to preserve class distribution
def downsample_per_subject(dfx, factor):
    return (dfx.groupby('subject_id', group_keys=False)
              .apply(lambda g: g.iloc[::factor])
              .reset_index(drop=True))

df_train_ds = downsample_per_subject(df_train, DOWNSAMPLE_FACTOR)
df_test_ds  = downsample_per_subject(df_test,  DOWNSAMPLE_FACTOR)

print("Downsampled rows: train", len(df_train_ds), "test", len(df_test_ds))
```

```
Downsampled rows: train 139361 test 49993
```

```
C:\Users\andre\AppData\Local\Temp\ipykernel_35428\1455445310.py:8:
DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns.
This behavior is deprecated, and in a future version of pandas the grouping
columns will be excluded from the operation. Either pass `include_groups=False`
to exclude the groupings or explicitly select the grouping columns after groupby
to silence this warning.
  .apply(lambda g: g.iloc[::factor])
C:\Users\andre\AppData\Local\Temp\ipykernel_35428\1455445310.py:8:
DeprecationWarning: DataFrameGroupBy.apply operated on the grouping columns.
This behavior is deprecated, and in a future version of pandas the grouping
columns will be excluded from the operation. Either pass `include_groups=False`
to exclude the groupings or explicitly select the grouping columns after groupby
to silence this warning.
  .apply(lambda g: g.iloc[::factor])
```

[35]:
```python
'''
We transformed the downsampled wearable time series into supervised learning␣
↪examples using sliding windows (10s, 50% overlap).
For each window we extracted time-domain statistics, energy, jerk features, and␣
↪coarse frequency-band energy from FFT, producing a fixed-length feature␣
↪vector per window.
We built windows only within constant activity segments to avoid label␣
↪contamination, and stored the subject ID per window to enable GroupKFold␣
↪cross-validation without subject leakage.
'''
# Window Feature Extraction
def window_features(w):
    mean = w.mean(axis=0)
    std  = w.std(axis=0)
```

```
    mn   = w.min(axis=0)
    mx   = w.max(axis=0)
    med  = np.median(w, axis=0)
    q75  = np.percentile(w, 75, axis=0)
    q25  = np.percentile(w, 25, axis=0)
    iqr  = q75 - q25

    energy = (w**2).mean(axis=0)

    jw = np.diff(w, axis=0)
    jerk_mean = jw.mean(axis=0)
    jerk_std  = jw.std(axis=0)
    jerk_energy = (jw**2).mean(axis=0)

    # FFT band energies
    wd = w - mean
    fft = np.fft.rfft(wd, axis=0)
    psd = (np.abs(fft)**2)
    mid = max(1, psd.shape[0] // 2)
    low_band = psd[:mid].mean(axis=0)
    high_band = psd[mid:].mean(axis=0)

    return np.concatenate([
        mean, std, mn, mx, med, iqr,
        energy,
        jerk_mean, jerk_std, jerk_energy,
        low_band, high_band
    ], axis=0)

def make_windows_no_crossing(df_in, feature_cols,
                             label_col='activity_id_1', group_col='subject_id',
                             win_size=100, step=50, max_windows_per_subject=None):
    X_list, y_list, g_list = [], [], []

    for sid, g in df_in.groupby(group_col, sort=False):
        g = g.sort_values('datetime')
        arr = g[feature_cols].to_numpy(dtype=np.float32)
        labels = g[label_col].to_numpy(dtype=np.int32)

        change_idx = np.where(labels[:-1] != labels[1:])[0] + 1
        starts = np.r_[0, change_idx]
        ends   = np.r_[change_idx, len(g)]

        n_added = 0
        for s, e in zip(starts, ends):
            seg_len = e - s
```

```
            if seg_len < win_size:
                continue

            seg_arr = arr[s:e]
            seg_label = labels[s]

            for st in range(0, seg_len - win_size + 1, step):
                w = seg_arr[st:st + win_size]
                X_list.append(window_features(w))
                y_list.append(seg_label)
                g_list.append(sid)

                n_added += 1
                if max_windows_per_subject is not None and n_added >=␣
↪max_windows_per_subject:
                    break

            if max_windows_per_subject is not None and n_added >=␣
↪max_windows_per_subject:
                break

    X = np.vstack(X_list) if X_list else np.empty((0, 1), dtype=np.float32)
    y = np.array(y_list, dtype=np.int32)
    groups = np.array(g_list, dtype=np.int32)
    return X, y, groups

SAMPLE_RATE = 100 // DOWNSAMPLE_FACTOR  # 10 Hz
WIN_SECONDS = 10
WIN_SIZE = WIN_SECONDS * SAMPLE_RATE    # 100 samples
STEP = WIN_SIZE // 2                    # 50% overlap

MAX_WINDOWS_PER_SUBJECT = 15000  # reduce if memory/time is an issue

X_train, y_train, g_train = make_windows_no_crossing(
    df_train_ds, feature_cols, win_size=WIN_SIZE, step=STEP,␣
↪max_windows_per_subject=MAX_WINDOWS_PER_SUBJECT
)
X_test, y_test, g_test = make_windows_no_crossing(
    df_test_ds, feature_cols, win_size=WIN_SIZE, step=STEP,␣
↪max_windows_per_subject=MAX_WINDOWS_PER_SUBJECT
)

print("X_train:", X_train.shape, "X_test:", X_test.shape)
print("Train label dist (top 10):")
print(pd.Series(y_train).value_counts().head(10))
```

```
X_train: (2676, 408) X_test: (962, 408)
```

```
Train label dist (top 10):
4     353
17    335
1     284
3     272
2     269
16    258
7     256
6     231
12    156
13    135
Name: count, dtype: int64
```

3.3 Models and Validation

[36]:
```python
# Model Evaluation with GroupKFold because of subject differences
def evaluate_model_cv(name, model, X, y, groups):
    n_splits = min(5, len(np.unique(groups)))
    gkf = GroupKFold(n_splits=n_splits)

    f1s, bals = [], []
    for fold, (tr, va) in enumerate(gkf.split(X, y, groups), 1):
        model.fit(X[tr], y[tr])
        pred = model.predict(X[va])
        f1s.append(f1_score(y[va], pred, average='macro'))
        bals.append(balanced_accuracy_score(y[va], pred))

    return np.mean(f1s), np.std(f1s), np.mean(bals), np.std(bals)

models = {
    "LogReg": Pipeline([
        ("scaler", StandardScaler()),
        ("clf", LogisticRegression(max_iter=2000, class_weight="balanced",␣
 ↪n_jobs=-1))
    ]),
    "LinearSVC": Pipeline([
        ("scaler", StandardScaler()),
        ("clf", LinearSVC(class_weight="balanced"))
    ]),
    "RandomForest": RandomForestClassifier(
        n_estimators=300, n_jobs=-1, class_weight="balanced_subsample",
        min_samples_leaf=2, random_state=42
    ),
    "ExtraTrees": ExtraTreesClassifier(
        n_estimators=400, n_jobs=-1, class_weight="balanced_subsample",
        min_samples_leaf=1, random_state=42
    ),
```

```
    "HistGradientBoosting": HistGradientBoostingClassifier(
        max_depth=8, learning_rate=0.1, max_iter=300, random_state=42
    )
}

rows = []
for name, m in models.items():
    mf1, sf1, mbal, sbal = evaluate_model_cv(name, m, X_train, y_train, g_train)
    rows.append((name, mf1, sf1, mbal, sbal))

results_df = pd.DataFrame(rows,␣
 ↪columns=["model","macro_f1_mean","macro_f1_std","bal_acc_mean","bal_acc_std"])␣
 ↪\
             .sort_values("macro_f1_mean", ascending=False)
results_df
```

```
C:\Users\andre\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\svm\_base.py:1249: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  warnings.warn(
C:\Users\andre\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\metrics\_classification.py:2524: UserWarning: y_pred contains
classes not in y_true
  warnings.warn("y_pred contains classes not in y_true")
```

[36]:

```
                  model  macro_f1_mean  macro_f1_std  bal_acc_mean  \
3            ExtraTrees       0.949646      0.028989      0.946196
2          RandomForest       0.917240      0.070296      0.933758
4  HistGradientBoosting       0.861851      0.112168      0.872557
0                LogReg       0.853238      0.117535      0.876152
1             LinearSVC       0.818779      0.133356      0.846958

   bal_acc_std
3     0.028260
2     0.043935
4     0.101874
0     0.086248
1     0.098783
```

[37]:

```
# pick best model name from your results table
best_name = results_df.iloc[0]["model"]
best_model = models[best_name]

# fit on full train set (if not already fitted)
best_model.fit(X_train, y_train)
```

```
# save to disk
out_dir = Path("artifacts")
out_dir.mkdir(exist_ok=True)

model_path = out_dir / f"{best_name}.joblib"
joblib.dump(best_model, model_path)

print("Saved model to:", model_path.resolve())


feat_path = out_dir / "feature_cols.json"
with open(feat_path, "w", encoding="utf-8") as f:
    json.dump(feature_cols, f, indent=2)

print("Saved feature columns to:", feat_path.resolve())
```

```
Saved model to: E:\AAI\AAI530-Data Analytics and Internet of
Things\project\artifacts\ExtraTrees.joblib
Saved feature columns to: E:\AAI\AAI530-Data Analytics and Internet of
Things\project\artifacts\feature_cols.json
```

[38]:
```
gkf = GroupKFold(n_splits=min(5, len(np.unique(g_train)))) # for use in CV␣
 ↪searches

svc_pipe = Pipeline([
    ("scaler", StandardScaler()),
    ("clf", LinearSVC(class_weight="balanced"))
])

svc_params = {"clf__C": np.logspace(-3, 2, 20)}  # 0.001..100

svc_search = RandomizedSearchCV(
    svc_pipe,
    svc_params,
    n_iter=12,
    scoring="f1_macro",
    cv=gkf.split(X_train, y_train, g_train),
    n_jobs=-1,
    random_state=42,
    verbose=1
)
svc_search.fit(X_train, y_train)

print("Best SVC params:", svc_search.best_params_)
print("Best SVC CV macro F1:", svc_search.best_score_)
best_svc = svc_search.best_estimator_
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
```

```
Best SVC params: {'clf__C': np.float64(0.003359818286283781)}
Best SVC CV macro F1: 0.8827800834693544
```

3.3.1 Models Summary

```
[ ]: best_name = results_df.iloc[0]["model"]
best_model = models[best_name]

best_model.fit(X_train, y_train)
pred_best = best_model.predict(X_test)

best_svc.fit(X_train, y_train)
pred_svc = best_svc.predict(X_test)

def summarize(name, y_true, y_pred):
    print(f"\n{name}")
    print("TEST macro F1:", f1_score(y_true, y_pred, average="macro"))
    print("TEST balanced acc:", balanced_accuracy_score(y_true, y_pred))
    print(classification_report(y_true, y_pred, digits=4))

summarize(f"best model: {best_name}", y_test, pred_best)
summarize("Tuned LinearSVC", y_test, pred_svc)

f1_best = f1_score(y_test, pred_best, average="macro")
f1_svc  = f1_score(y_test, pred_svc, average="macro")

use_name = f"Tuned LinearSVC" if f1_svc >= f1_best else f"best model:␣
↪{best_name}"
use_pred = pred_svc if f1_svc >= f1_best else pred_best

labels_sorted = np.sort(np.unique(np.concatenate([y_train, y_test])))
cm = confusion_matrix(y_test, use_pred, labels=labels_sorted)
cm_counts_df = pd.DataFrame(cm, index=labels_sorted, columns=labels_sorted)
print("\nCounts (rows=true, cols=pred):")
display(cm_counts_df)

#normalized by true class (recall per class)
cm_norm = confusion_matrix(y_test, use_pred, labels=labels_sorted,␣
↪normalize="true")
cm_norm_df = pd.DataFrame(cm_norm, index=labels_sorted, columns=labels_sorted)
print("\nNormalized by true class:")
display(cm_norm_df.round(3))
```

```
best model: ExtraTrees
TEST macro F1: 0.9476640061215146
TEST balanced acc: 0.946905599798313
              precision    recall  f1-score   support
```

	precision	recall	f1-score	support
1	1.0000	0.9778	0.9888	90
2	0.9610	0.8409	0.8970	88
3	0.9881	0.8557	0.9171	97
4	0.8636	1.0000	0.9268	114
5	1.0000	1.0000	1.0000	61
6	0.9888	1.0000	0.9944	88
7	1.0000	0.8273	0.9055	110
12	0.9492	1.0000	0.9739	56
13	1.0000	0.9574	0.9783	47
16	0.8602	0.9877	0.9195	81
17	0.8811	0.9692	0.9231	130
accuracy			0.9418	962
macro avg	0.9538	0.9469	0.9477	962
weighted avg	0.9473	0.9418	0.9414	962

Tuned LinearSVC
TEST macro F1: 0.9492797701230294
TEST balanced acc: 0.9470850119256202

	precision	recall	f1-score	support
1	1.0000	0.9889	0.9944	90
2	0.8730	0.6250	0.7285	88
3	0.9762	0.8454	0.9061	97
4	1.0000	1.0000	1.0000	114
5	1.0000	1.0000	1.0000	61
6	1.0000	1.0000	1.0000	88
7	1.0000	1.0000	1.0000	110
12	0.9655	1.0000	0.9825	56
13	1.0000	0.9787	0.9892	47
16	0.9412	0.9877	0.9639	81
17	0.7866	0.9923	0.8776	130
accuracy			0.9459	962
macro avg	0.9584	0.9471	0.9493	962
weighted avg	0.9502	0.9459	0.9440	962

Confusion matrix for: Tuned LinearSVC

Counts (rows=true, cols=pred):

	1	2	3	4	5	6	7	12	13	16	17
1	89	0	1	0	0	0	0	0	0	0	0
2	0	55	1	0	0	0	0	0	0	4	28
3	0	8	82	0	0	0	0	0	0	0	7

```
4     0    0    0  114    0    0     0    0    0    0     0
5     0    0    0    0   61    0     0    0    0    0     0
6     0    0    0    0    0   88     0    0    0    0     0
7     0    0    0    0    0    0   110    0    0    0     0
12    0    0    0    0    0    0     0   56    0    0     0
13    0    0    0    0    0    0     0    1   46    0     0
16    0    0    0    0    0    0     0    1    0   80     0
17    0    0    0    0    0    0     0    0    0    1   129
```

```
Normalized by true class:
```

```
        1      2      3    4    5    6    7     12     13     16     17
1   0.989  0.000  0.011  0.0  0.0  0.0  0.0  0.000  0.000  0.000  0.000
2   0.000  0.625  0.011  0.0  0.0  0.0  0.0  0.000  0.000  0.045  0.318
3   0.000  0.082  0.845  0.0  0.0  0.0  0.0  0.000  0.000  0.000  0.072
4   0.000  0.000  0.000  1.0  0.0  0.0  0.0  0.000  0.000  0.000  0.000
5   0.000  0.000  0.000  0.0  1.0  0.0  0.0  0.000  0.000  0.000  0.000
6   0.000  0.000  0.000  0.0  0.0  1.0  0.0  0.000  0.000  0.000  0.000
7   0.000  0.000  0.000  0.0  0.0  0.0  1.0  0.000  0.000  0.000  0.000
12  0.000  0.000  0.000  0.0  0.0  0.0  0.0  1.000  0.000  0.000  0.000
13  0.000  0.000  0.000  0.0  0.0  0.0  0.0  0.021  0.979  0.000  0.000
16  0.000  0.000  0.000  0.0  0.0  0.0  0.0  0.012  0.000  0.988  0.000
17  0.000  0.000  0.000  0.0  0.0  0.0  0.0  0.000  0.000  0.008  0.992
```

3.4 Deep Learning

3.4.1 Raw windows with no crossing label boundaries

[40]:
```python
'''
This deep learning section uses the same time windowing concept above,
but instead of summarizing each window into statistics,
it feeds the raw multi channel sequence into a 1D CNN so the network learns temporal patterns directly.
That makes the deep learning choice true and consistent with the other machine learning approach, and allows us to compare them more fairly.
'''
def make_raw_windows_no_crossing(
    df_in,
    feature_cols,
    label_col="activity_id_1",
    group_col="subject_id",
    win_size=100,
    step=50,
    max_windows_per_subject=None
):
    X_list, y_list, g_list = [], [], []

    for sid, g in df_in.groupby(group_col, sort=False):
```

```
        g = g.sort_values("datetime")
        arr = g[feature_cols].to_numpy(dtype=np.float32)
        labels = g[label_col].to_numpy(dtype=np.int32)

        change_idx = np.where(labels[:-1] != labels[1:])[0] + 1
        starts = np.r_[0, change_idx]
        ends = np.r_[change_idx, len(g)]

        n_added = 0
        for s, e in zip(starts, ends):
            seg_len = e - s
            if seg_len < win_size:
                continue

            seg_arr = arr[s:e]
            seg_label = labels[s]

            for st in range(0, seg_len - win_size + 1, step):
                w = seg_arr[st:st + win_size]
                X_list.append(w)
                y_list.append(seg_label)
                g_list.append(sid)

                n_added += 1
                if max_windows_per_subject is not None and n_added >=␣
↪max_windows_per_subject:
                    break

            if max_windows_per_subject is not None and n_added >=␣
↪max_windows_per_subject:
                break

    X = np.stack(X_list, axis=0) if X_list else np.empty((0, win_size,␣
↪len(feature_cols)), dtype=np.float32)
    y = np.array(y_list, dtype=np.int32)
    groups = np.array(g_list, dtype=np.int32)
    return X, y, groups
```

3.4.2 Train and test raw windows

[41]:
```
SAMPLE_RATE = 100 // DOWNSAMPLE_FACTOR
WIN_SECONDS = 10
WIN_SIZE = WIN_SECONDS * SAMPLE_RATE
STEP = WIN_SIZE // 2

MAX_WINDOWS_PER_SUBJECT_DL = 12000
```

```python
Xtr_raw, ytr_raw, gtr_raw = make_raw_windows_no_crossing(
    df_train_ds, feature_cols,
    win_size=WIN_SIZE, step=STEP,
    max_windows_per_subject=MAX_WINDOWS_PER_SUBJECT_DL
)

Xte_raw, yte_raw, gte_raw = make_raw_windows_no_crossing(
    df_test_ds, feature_cols,
    win_size=WIN_SIZE, step=STEP,
    max_windows_per_subject=MAX_WINDOWS_PER_SUBJECT_DL
)

print("Xtr_raw:", Xtr_raw.shape, "Xte_raw:", Xte_raw.shape)
print("Unique train labels:", np.unique(ytr_raw))

le = LabelEncoder()
ytr = le.fit_transform(ytr_raw)
yte = le.transform(yte_raw)

K = len(le.classes_)
T = Xtr_raw.shape[1]
C = Xtr_raw.shape[2]

mu = Xtr_raw.mean(axis=(0, 1), keepdims=True)
sd = Xtr_raw.std(axis=(0, 1), keepdims=True) + 1e-8

Xtr = (Xtr_raw - mu) / sd
Xte = (Xte_raw - mu) / sd

print("Classes:", K, "Time steps:", T, "Channels:", C)
print("Label example mapping:", dict(list(zip(le.classes_[:10], range(min(10,
 ↪K))))))

# Map labels to 0..K-1 and standardize per channel using train statistics only
le = LabelEncoder()
ytr = le.fit_transform(ytr_raw)
yte = le.transform(yte_raw)

K = len(le.classes_)
T = Xtr_raw.shape[1]
C = Xtr_raw.shape[2]

mu = Xtr_raw.mean(axis=(0, 1), keepdims=True)
sd = Xtr_raw.std(axis=(0, 1), keepdims=True) + 1e-8

Xtr = (Xtr_raw - mu) / sd
Xte = (Xte_raw - mu) / sd
```

```
print("Classes:", K, "Time steps:", T, "Channels:", C)
print("Label example mapping:", dict(list(zip(le.classes_[:10], range(min(10,␣
↪K))))))
```

```
Xtr_raw: (2676, 100, 34) Xte_raw: (962, 100, 34)
Unique train labels: [ 1  2  3  4  5  6  7 12 13 16 17]
Classes: 11 Time steps: 100 Channels: 34
Label example mapping: {np.int32(1): 0, np.int32(2): 1, np.int32(3): 2,
np.int32(4): 3, np.int32(5): 4, np.int32(6): 5, np.int32(7): 6, np.int32(12): 7,
np.int32(13): 8, np.int32(16): 9}
Classes: 11 Time steps: 100 Channels: 34
Label example mapping: {np.int32(1): 0, np.int32(2): 1, np.int32(3): 2,
np.int32(4): 3, np.int32(5): 4, np.int32(6): 5, np.int32(7): 6, np.int32(12): 7,
np.int32(13): 8, np.int32(16): 9}
```

3.4.3 1D CNN model and training utilities

[42]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Device:", device)

class WindowDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.tensor(X, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.long)

    def __len__(self):
        return self.X.shape[0]

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

train_ds = WindowDataset(Xtr, ytr)
test_ds = WindowDataset(Xte, yte)

batch_size = 256
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,␣
↪num_workers=0)
test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False,␣
↪num_workers=0)

class CNN1D(nn.Module):
    def __init__(self, channels, num_classes):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv1d(channels, 128, kernel_size=7, padding=3),
            nn.BatchNorm1d(128),
```

```
            nn.ReLU(),
            nn.Dropout(0.2),

            nn.Conv1d(128, 128, kernel_size=5, padding=2),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(0.2),

            nn.Conv1d(128, 256, kernel_size=5, padding=2),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Dropout(0.3),

            nn.AdaptiveAvgPool1d(1)
        )
        self.fc = nn.Linear(256, num_classes)

    def forward(self, x):
        x = x.transpose(1, 2)  # N T C to N C T
        x = self.net(x).squeeze(-1)
        return self.fc(x)

model = CNN1D(C, K).to(device)

# Class weights help imbalance
counts = np.bincount(ytr, minlength=K).astype(np.float32)
weights = (counts.sum() / (counts + 1e-6))
weights = weights / weights.mean()
class_weights = torch.tensor(weights, dtype=torch.float32).to(device)

criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
```

```
Device: cuda
```

3.4.4 Train and evaluate

[43]:
```
def eval_model(model, loader):
    model.eval()
    ys, ps = [], []
    with torch.no_grad():
        for xb, yb in loader:
            xb = xb.to(device)
            yb = yb.to(device)
            logits = model(xb)
            pred = torch.argmax(logits, dim=1)
            ys.append(yb.cpu().numpy())
            ps.append(pred.cpu().numpy())
```

```python
    y_true = np.concatenate(ys)
    y_pred = np.concatenate(ps)
    return y_true, y_pred

epochs = 100

for ep in range(1, epochs + 1):
    model.train()
    total_loss = 0.0
    for xb, yb in train_loader:
        xb = xb.to(device)
        yb = yb.to(device)

        optimizer.zero_grad()
        logits = model(xb)
        loss = criterion(logits, yb)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * xb.size(0)

    avg_loss = total_loss / len(train_ds)

    y_true, y_pred = eval_model(model, test_loader)
    macro_f1 = f1_score(y_true, y_pred, average="macro")
    bal_acc = balanced_accuracy_score(y_true, y_pred)

    print(f"Epoch {ep} loss {avg_loss:.4f} test macroF1 {macro_f1:.4f} test␣
 ↪bal_acc {bal_acc:.4f}")

y_true, y_pred = eval_model(model, test_loader)
print("Final test macroF1:", f1_score(y_true, y_pred, average="macro"))
print("Final test bal_acc:", balanced_accuracy_score(y_true, y_pred))

# Report using original activity ids
print(classification_report(le.inverse_transform(y_true), le.
 ↪inverse_transform(y_pred), digits=4))
```

```
Epoch 1 loss 1.1963 test macroF1 0.6414 test bal_acc 0.6994
Epoch 2 loss 0.3300 test macroF1 0.8000 test bal_acc 0.8163
Epoch 3 loss 0.1566 test macroF1 0.8392 test bal_acc 0.8417
Epoch 4 loss 0.0955 test macroF1 0.8331 test bal_acc 0.8288
Epoch 5 loss 0.0621 test macroF1 0.8396 test bal_acc 0.8327
Epoch 6 loss 0.0447 test macroF1 0.8460 test bal_acc 0.8413
Epoch 7 loss 0.0357 test macroF1 0.8370 test bal_acc 0.8331
Epoch 8 loss 0.0299 test macroF1 0.8540 test bal_acc 0.8488
Epoch 9 loss 0.0233 test macroF1 0.8535 test bal_acc 0.8499
Epoch 10 loss 0.0169 test macroF1 0.8585 test bal_acc 0.8552
Epoch 11 loss 0.0141 test macroF1 0.8463 test bal_acc 0.8440
```

```
Epoch 12 loss 0.0118 test macroF1 0.8651 test bal_acc 0.8599
Epoch 13 loss 0.0118 test macroF1 0.8572 test bal_acc 0.8520
Epoch 14 loss 0.0101 test macroF1 0.8490 test bal_acc 0.8492
Epoch 15 loss 0.0086 test macroF1 0.8969 test bal_acc 0.8915
Epoch 16 loss 0.0078 test macroF1 0.8420 test bal_acc 0.8425
Epoch 17 loss 0.0062 test macroF1 0.8609 test bal_acc 0.8580
Epoch 18 loss 0.0062 test macroF1 0.9106 test bal_acc 0.9054
Epoch 19 loss 0.0062 test macroF1 0.8871 test bal_acc 0.8802
Epoch 20 loss 0.0055 test macroF1 0.9139 test bal_acc 0.9077
Epoch 21 loss 0.0043 test macroF1 0.9284 test bal_acc 0.9224
Epoch 22 loss 0.0038 test macroF1 0.8814 test bal_acc 0.8793
Epoch 23 loss 0.0031 test macroF1 0.9243 test bal_acc 0.9186
Epoch 24 loss 0.0037 test macroF1 0.8991 test bal_acc 0.8928
Epoch 25 loss 0.0029 test macroF1 0.8929 test bal_acc 0.8863
Epoch 26 loss 0.0024 test macroF1 0.9232 test bal_acc 0.9169
Epoch 27 loss 0.0026 test macroF1 0.9153 test bal_acc 0.9094
Epoch 28 loss 0.0022 test macroF1 0.9198 test bal_acc 0.9138
Epoch 29 loss 0.0024 test macroF1 0.8967 test bal_acc 0.8900
Epoch 30 loss 0.0020 test macroF1 0.9289 test bal_acc 0.9235
Epoch 31 loss 0.0019 test macroF1 0.9217 test bal_acc 0.9158
Epoch 32 loss 0.0017 test macroF1 0.9212 test bal_acc 0.9150
Epoch 33 loss 0.0019 test macroF1 0.8859 test bal_acc 0.8797
Epoch 34 loss 0.0020 test macroF1 0.9188 test bal_acc 0.9129
Epoch 35 loss 0.0018 test macroF1 0.9240 test bal_acc 0.9178
Epoch 36 loss 0.0014 test macroF1 0.9200 test bal_acc 0.9145
Epoch 37 loss 0.0015 test macroF1 0.9295 test bal_acc 0.9243
Epoch 38 loss 0.0016 test macroF1 0.9293 test bal_acc 0.9236
Epoch 39 loss 0.0012 test macroF1 0.9173 test bal_acc 0.9112
Epoch 40 loss 0.0012 test macroF1 0.9238 test bal_acc 0.9179
Epoch 41 loss 0.0011 test macroF1 0.9223 test bal_acc 0.9160
Epoch 42 loss 0.0011 test macroF1 0.9270 test bal_acc 0.9208
Epoch 43 loss 0.0013 test macroF1 0.9189 test bal_acc 0.9129
Epoch 44 loss 0.0012 test macroF1 0.9227 test bal_acc 0.9168
Epoch 45 loss 0.0013 test macroF1 0.8944 test bal_acc 0.8887
Epoch 46 loss 0.0011 test macroF1 0.9311 test bal_acc 0.9254
Epoch 47 loss 0.0010 test macroF1 0.9343 test bal_acc 0.9293
Epoch 48 loss 0.0009 test macroF1 0.9313 test bal_acc 0.9256
Epoch 49 loss 0.0009 test macroF1 0.9418 test bal_acc 0.9371
Epoch 50 loss 0.0009 test macroF1 0.9135 test bal_acc 0.9077
Epoch 51 loss 0.0009 test macroF1 0.9276 test bal_acc 0.9215
Epoch 52 loss 0.0009 test macroF1 0.9215 test bal_acc 0.9151
Epoch 53 loss 0.0008 test macroF1 0.9341 test bal_acc 0.9286
Epoch 54 loss 0.0008 test macroF1 0.9332 test bal_acc 0.9277
Epoch 55 loss 0.0009 test macroF1 0.9148 test bal_acc 0.9110
Epoch 56 loss 0.0009 test macroF1 0.9214 test bal_acc 0.9149
Epoch 57 loss 0.0008 test macroF1 0.9216 test bal_acc 0.9148
Epoch 58 loss 0.0008 test macroF1 0.9105 test bal_acc 0.9035
Epoch 59 loss 0.0008 test macroF1 0.9249 test bal_acc 0.9182
```

```
Epoch 60 loss 0.0009 test macroF1 0.9356 test bal_acc 0.9301
Epoch 61 loss 0.0009 test macroF1 0.9153 test bal_acc 0.9092
Epoch 62 loss 0.0010 test macroF1 0.8850 test bal_acc 0.8828
Epoch 63 loss 0.0008 test macroF1 0.9310 test bal_acc 0.9253
Epoch 64 loss 0.0009 test macroF1 0.9390 test bal_acc 0.9342
Epoch 65 loss 0.0008 test macroF1 0.9310 test bal_acc 0.9254
Epoch 66 loss 0.0007 test macroF1 0.8977 test bal_acc 0.8911
Epoch 67 loss 0.0007 test macroF1 0.9358 test bal_acc 0.9304
Epoch 68 loss 0.0006 test macroF1 0.9087 test bal_acc 0.9073
Epoch 69 loss 0.0006 test macroF1 0.9252 test bal_acc 0.9189
Epoch 70 loss 0.0007 test macroF1 0.9272 test bal_acc 0.9208
Epoch 71 loss 0.0006 test macroF1 0.8962 test bal_acc 0.8890
Epoch 72 loss 0.0006 test macroF1 0.9288 test bal_acc 0.9218
Epoch 73 loss 0.0006 test macroF1 0.9282 test bal_acc 0.9219
Epoch 74 loss 0.0006 test macroF1 0.8924 test bal_acc 0.8908
Epoch 75 loss 0.0007 test macroF1 0.9319 test bal_acc 0.9264
Epoch 76 loss 0.0040 test macroF1 0.9301 test bal_acc 0.9244
Epoch 77 loss 0.0124 test macroF1 0.8966 test bal_acc 0.9031
Epoch 78 loss 0.0539 test macroF1 0.8265 test bal_acc 0.8339
Epoch 79 loss 0.0372 test macroF1 0.8564 test bal_acc 0.8536
Epoch 80 loss 0.0169 test macroF1 0.8992 test bal_acc 0.8898
Epoch 81 loss 0.0080 test macroF1 0.9127 test bal_acc 0.9053
Epoch 82 loss 0.0045 test macroF1 0.9103 test bal_acc 0.9036
Epoch 83 loss 0.0030 test macroF1 0.8779 test bal_acc 0.8698
Epoch 84 loss 0.0031 test macroF1 0.8621 test bal_acc 0.8561
Epoch 85 loss 0.0021 test macroF1 0.8997 test bal_acc 0.8902
Epoch 86 loss 0.0018 test macroF1 0.9251 test bal_acc 0.9171
Epoch 87 loss 0.0014 test macroF1 0.9396 test bal_acc 0.9322
Epoch 88 loss 0.0013 test macroF1 0.9382 test bal_acc 0.9319
Epoch 89 loss 0.0011 test macroF1 0.9324 test bal_acc 0.9256
Epoch 90 loss 0.0011 test macroF1 0.9267 test bal_acc 0.9203
Epoch 91 loss 0.0010 test macroF1 0.9273 test bal_acc 0.9198
Epoch 92 loss 0.0009 test macroF1 0.9281 test bal_acc 0.9210
Epoch 93 loss 0.0009 test macroF1 0.9337 test bal_acc 0.9270
Epoch 94 loss 0.0008 test macroF1 0.9323 test bal_acc 0.9252
Epoch 95 loss 0.0010 test macroF1 0.9090 test bal_acc 0.9009
Epoch 96 loss 0.0010 test macroF1 0.9226 test bal_acc 0.9172
Epoch 97 loss 0.0008 test macroF1 0.9239 test bal_acc 0.9177
Epoch 98 loss 0.0008 test macroF1 0.9264 test bal_acc 0.9198
Epoch 99 loss 0.0007 test macroF1 0.9319 test bal_acc 0.9250
Epoch 100 loss 0.0008 test macroF1 0.9307 test bal_acc 0.9235
Final test macroF1: 0.9306669249239762
Final test bal_acc: 0.9235021976673522
              precision    recall  f1-score   support

           1     1.0000    0.9667    0.9831        90
           2     0.9508    0.6591    0.7785        88
           3     0.9524    0.8247    0.8840        97
```

```
           4     1.0000    1.0000    1.0000       114
           5     1.0000    1.0000    1.0000        61
           6     0.9263    1.0000    0.9617        88
           7     1.0000    1.0000    1.0000       110
          12     0.9649    0.9821    0.9735        56
          13     0.9730    0.7660    0.8571        47
          16     0.9186    0.9753    0.9461        81
          17     0.7529    0.9846    0.8533       130

    accuracy                         0.9314       962
   macro avg     0.9490    0.9235    0.9307       962
weighted avg     0.9404    0.9314    0.9301       962
```

[57]:
```
def _cls_metrics(y_true, y_pred):
    return {
        "Accuracy": accuracy_score(y_true, y_pred),
        "BalancedAcc": balanced_accuracy_score(y_true, y_pred),
        "MacroF1": f1_score(y_true, y_pred, average="macro"),
    }

rows = []

if "results_df" in globals() and "models" in globals():
    for name in results_df["model"].tolist():
        if name not in models:
            continue

        est = clone(models[name])
        est.fit(X_train, y_train)

        pred_tr = est.predict(X_train)
        pred_te = est.predict(X_test)

        tr = _cls_metrics(y_train, pred_tr)
        te = _cls_metrics(y_test, pred_te)

        r = results_df.loc[results_df["model"] == name].iloc[0]
        rows.append({
            "Model": name,
            "Validation BalancedAcc": float(r.get("bal_acc_mean", np.nan)),
            "Validation MacroF1": float(r.get("macro_f1_mean", np.nan)),
            "Test Accuracy": te["Accuracy"],
            "Test BalancedAcc": te["BalancedAcc"],
            "Test MacroF1": te["MacroF1"],
        })
```

```python
def eval_all(ds, batch_size=256):
    from torch.utils.data import DataLoader
    loader = DataLoader(ds, batch_size=batch_size, shuffle=False, num_workers=0)
    y_true, y_pred = eval_model(model, loader)
    return y_true, y_pred

if all(k in globals() for k in ["model", "device", "train_ds", "test_ds",␣
↪"eval_model"]):
    try:
        y_tr_true, y_tr_pred = eval_all(train_ds, batch_size=256)
        y_te_true, y_te_pred = eval_all(test_ds, batch_size=256)

        tr = _cls_metrics(y_tr_true, y_tr_pred)
        te = _cls_metrics(y_te_true, y_te_pred)

        rows.append({
            "Model": "CNN 1D",
            "Validation BalancedAcc": np.nan,
            "Validation MacroF1": np.nan,
            "Test Accuracy": te["Accuracy"],
            "Test BalancedAcc": te["BalancedAcc"],
            "Test MacroF1": te["MacroF1"],
        })
    except Exception as e:
        print("CNN compare row skipped due to error:", e)

compare_df = pd.DataFrame(rows)

compare_df = compare_df.sort_values("Test MacroF1", ascending=False).
↪reset_index(drop=True)

best_val = compare_df["Test MacroF1"].max()

def _highlight_best_row(row):
    return ["background-color: #808070" if row["Test MacroF1"] == best_val else␣
↪"" for _ in row]

display(
    compare_df.style
    .format({c: "{:.4f}" for c in compare_df.columns if c != "Model"})
    .apply(_highlight_best_row, axis=1)
)
```

```
C:\Users\andre\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\sklearn\svm\_base.py:1249: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
```

```
  warnings.warn(
```

```
<pandas.io.formats.style.Styler at 0x2547cc5ebd0>
```