**Driver Identification from CAN Bus Data**

**Dataset Overview**

This project is based on the publicly available Driver Identification Dataset hosted by Oak Ridge National Laboratory (ORNL) (<https://doi.ccs.ornl.gov/dataset/8e55a1d4-95fb-5f77-952f-c889db9fa539>).

The dataset contains driving behavior data from 50 drivers operating a 2014 Kenworth T270 Class 6 truck around Fort Collins, Colorado. Data sources include CAN bus signals (SAE J1939 standard), GPS, inertial measurements, and biometric heart rate monitoring.

During each drive, a cyberattack was triggered, forcing dashboard warning lights and resetting speed/tachometer displays. Drivers were grouped by prior knowledge of the attack (Group 1 = unaware, Group 2 = warned, Group 3 = warned with pull-over instructions).

The dataset is structured into three folders:

1. DriverIDDataDownsampled – 1-second aggregated data with derived variables.
2. DriverIDDataHighResolution – high-frequency raw signals (hundredths of a second) provided in CSV and SQLite formats.
3. DriverIDDataRaw – original device-level data (CAN logs, GPS, heart rate monitor files).

Each folder comes with its own data dictionary. The original dataset also provides a detailed README file describing devices, preprocessing, and known issues (e.g., missing VBOX data for some drivers).

For this project, we selected the DriverIDDataHighResolution folder and downloaded all 50 driver CSV files. These files contain the full-resolution CAN bus streams required for building driver identification models. Other data modalities (GPS, heart rate, inertial sensors) were not used in this study.

**Data Preprocessing**

**Step 1 — Extracting CAN Bus Signals**

Each raw driver file from the DriverIDDataHighResolution folder contained multiple modalities (CAN, GPS, inertial, biometric). For this project, we restricted analysis to the **CAN bus signals only**.

* A cutoff variable (84 Wheel-Based Vehicle Speed) was used to identify the last valid CAN column.
* All columns from the first CAN feature up to this cutoff were retained, while non-CAN data (e.g., GPS, biometrics) were discarded.
* Processed CAN-only files were exported into a dedicated CAN\_Data/ folder with filenames suffixed as \_CANdata.csv.

This step standardized the input so that each driver’s file contained only CAN-based features, which are the basis of all subsequent windowing, feature selection, and modeling stages.

**Step 2 — Downsampling and Aggregation**

The raw CAN bus signals in the high-resolution files were sampled at very high and irregular frequencies (hundredths of a second). To obtain consistent inputs suitable for machine learning, we performed **downsampling and statistical aggregation**:

* Signals were paired with their associated time columns using identifier tags in the dataset.
* For each signal, timestamps were floored to the nearest second, producing a uniform **1 Hz (1 sample per second)** representation.
* Within each 1-second interval, the following statistical features were computed:
  + **min, max, mean, standard deviation, and median**.
* These aggregate features capture short-term variability in each signal while dramatically reducing file size and noise.
* The downsampled files were exported into a dedicated Downsampled\_data/ folder, with filenames suffixed as \_downsampled\_1Hz.csv.

This step ensured that all drivers’ data streams were synchronized at the same temporal resolution and encoded both central tendency and variation of sensor readings, which are critical for distinguishing individual driving behavior.

**Step 3 — Sparsity Analysis**

To evaluate the completeness of the downsampled CAN features, a **sparsity table** was generated across all driver files:

* For each driver’s \_downsampled\_1Hz.csv, the percentage of missing values was calculated per column.
* Results were consolidated into a driver–feature matrix.
* This table allowed us to systematically identify **sparse columns** (features with high proportions of null values) that may need removal or imputation.

This diagnostic step was crucial to ensure only reliable, consistently recorded CAN signals were carried forward into feature selection and modeling.

**Step 3 (continued)— Sparsity Analysis and Visualization**

After downsampling, we examined the completeness of each CAN signal by constructing a sparsity table for all drivers. For every downsampled file, the percentage of missing values was computed per column. This produced a driver–by–signal matrix of sparsity, which we further visualized using a color-coded HTML heatmap:

* Blue (<20%) = low sparsity, reliable columns
* Yellow (20–50%) = moderate sparsity
* Orange (50–80%) = high sparsity
* Red (>80%) = very sparse, unreliable columns

The visualization provided an at-a-glance profile of which signals were consistently reliable and which were problematic across drivers.

A key observation from this step was that Driver G1\_Subject4 exhibited abnormally sparse data across most signals, with extended sequences of missing values. This issue was already noted in the dataset’s official README documentation, which flagged this driver as having reduced data quality. Based on this evidence, G1\_Subject4 was excluded from subsequent modeling to avoid biasing feature selection and classification results.

**Step 4 — Dropping Highly Sparse Features**

Following the sparsity analysis and visualization, we cleaned each downsampled driver file by removing features with excessive missingness:

* A threshold of **80% missing values** was applied.
* Columns meeting or exceeding this threshold were dropped from each driver’s file.
* Cleaned files were saved with the suffix \_g80.csv in a dedicated output folder.

This pruning step removed unreliable or incomplete signals, ensuring that downstream feature selection and modeling were based only on robust CAN variables. By enforcing a strict threshold, we balanced retaining enough signals for discriminative modeling while discarding those that would contribute noise or bias. **This step is followed by identification and dropping of other un-necessary columns manually.**

**Step 5 — Imputation of Remaining Missing Values**

After removing highly sparse columns (≥80% missing), the remaining CAN signals still contained occasional gaps. To ensure continuity of features for modeling, we applied forward–backward filling (ffill + bfill) imputation:

* For each cleaned driver file, all numeric columns were scanned for missing values.
* Missing entries were filled by propagating the last valid observation forward; if the first values in a series were still missing, they were filled using the next valid value backward.
* This two-pass approach guaranteed that no Nan’s remained while preserving realistic temporal consistency in sensor readings.

Cleaned and imputed driver-level datasets were saved with the suffix \_imputed.csv. Post-processing checks confirmed that all Nan values were resolved, providing a fully usable dataset for downstream merging, normalization, and windowing steps.

**Step 6 — Adding Target Labels**

To enable supervised learning, each driver’s imputed CAN dataset was annotated with a **target column** representing the driver ID.

* The driver ID was extracted directly from the filename (e.g., G1\_Subject5\_CANdata\_imputed.csv → G1\_Subject5).
* A new column target was appended to every row of the corresponding driver’s data.
* The updated files were saved in a dedicated folder (Imputed\_Data\_WithTarget).

This step ensured that all samples carried their driver labels, preparing the data for merging into a single dataset and enabling downstream classification tasks.

**Step 7 — Constructing a Normalized Time Column**

After labeling individual driver datasets, all files were merged into a single consolidated dataset (combined\_CAN\_dataset\_cleaned.csv). To ensure comparability across drivers, a **time index** was generated as follows:

* For each driver (grouped by target), rows were enumerated sequentially using cumcount().
* This produced a monotonic time column starting from 0 for each driver, effectively acting as a normalized timeline.
* The final dataset was saved as combined\_CAN\_dataset\_with\_time.csv.

This step standardized the temporal structure across drivers, enabling downstream operations such as windowing, temporal feature extraction, and fair cross-driver comparisons.

**Step 8 — Windowing and Chronological Train/Test Split**

To convert the continuous CAN streams into fixed-length learning units, we applied an **overlapping sliding window** over each driver’s time-normalized sequence:

* **Window length:** WINDOW\_SIZE = 120 rows (≈ 120 seconds at 1 Hz)
* **Overlap:** OVERLAP = 30 rows → **stride = 60**
* **Per-driver split:** first **30%** of windows reserved for **test** (TEST\_FRAC = 0.30)
* **Temporal gap:** **1 window** between the last test window and the first train window (GAP\_WINDOWS = 1) to reduce leakage across adjacent windows

**Procedure (per driver):**

1. Sort by time, then slide a 120-second window with 60-second stride; discard incomplete windows.
2. Assign a monotonically increasing window\_id within each driver.
3. Chronologically split windows: the earliest 30% go to **test**; skip one **gap** window; the remainder go to **train**.
4. Concatenate per-driver segments into global train\_df and test\_df, then sort by ['target', 'window\_id', 'time'].

**Outputs:**

* train\_70.csv — training windows (≈ 70% per driver, after the gap)
* test\_30.csv — earliest 30% windows per driver
* Each row retains target, time, window\_id, and all CAN features

**Leakage checks:**

* We verify that the set of (target, window\_id) pairs in train and test are **disjoint** (intersection size = 0).
* The 1-window **temporal gap** further reduces overlap-induced leakage due to the 50% window overlap.

**Why this design?**

* Overlapping windows preserve short-horizon dynamics while increasing the number of samples.
* A chronological split reflects realistic deployment (predicting on earlier segments without using future information).
* The enforced gap provides conservative separation between train and test contexts.

**Step 9 — Feature Ranking on Windowed Data (120s, 50% overlap)**

To identify the most informative CAN variables for driver identification, we computed **model-based importances** using a Random Forest trained on **flattened 120-second windows** (built in Step 8; 120 rows per window, 30-row stride). Each window was converted to a fixed-length vector by stacking all sensor values in time order, preserving short-horizon dynamics.

**Setup**

* **Train/Test inputs:** train\_70\_diff.csv, test\_30\_diff.csv (120s windows, per-driver chronological split with a 1-window gap).
* **Non-feature columns excluded:** {"target", "window\_id"} (the time column remains and contributes temporally ordered values).
* **Model:** RandomForestClassifier(n\_estimators=24, max\_depth=None, min\_samples\_leaf=2, max\_features="sqrt", class\_weight="balanced\_subsample", random\_state=903100, n\_jobs=-1).
* **Labels:** encoded with LabelEncoder.

**Procedure**

1. **Window preparation:** Keep only full windows of length 120; flatten each window (sensor × time → 1D).
2. **Training:** Fit the RF on training windows; evaluate on temporally separated test windows (accuracy and macro-F1 printed for reference).
3. **Flat feature naming:** Create names of the form sensor\_tXXX for each time index (e.g., speed\_t045).
4. **Importance extraction:** Use rf.feature\_importances\_ to rank flat features (sensor@time).
5. **Aggregation by sensor:** Sum importances across time for each sensor to get **per-sensor importance**.
6. **Exports:**
   * importances.csv — sensor-level importances (summed over 120 timesteps)

**Why this helps**

* Flat (sensor@time) importances highlight **temporally localized** patterns that separate drivers.
* Aggregated (per-sensor) importances reveal **which CAN variables matter most overall**, independent of time index.
* These rankings guide dimensionality reduction (e.g., selecting the **top-k sensors**) while keeping the most discriminative temporal context.

*(Later sections leverage these rankings to justify the six-sensor subset used by all downstream models.)*

**Step 10 — Selecting Top-6 Features**

Based on the Random Forest–based feature ranking (Step 9), we observed that the **top six CAN signals** accounted for the majority of discriminative importance, with a sharp drop in contribution after the sixth ranked feature. To streamline modeling while avoiding redundancy, we constructed reduced datasets retaining only these **six most informative sensors**, along with:

* the **normalized time column**,
* the **target column** (driver ID),
* the **window\_id** column (to preserve window alignment).

**Procedure:**

* From train\_70\_diff.csv and test\_30\_diff.csv, we subset only the top-6 selected features plus time, target, and window\_id.
* Reduced datasets were saved as:
  + train\_70\_diff\_6+t.csv
  + test\_30\_diff\_6+t.csv

This dimensionality reduction step ensured that subsequent models were trained and evaluated on the most informative subset of signals, simplifying interpretability and reducing the risk of overfitting.

**Modeling**

**Decision Tree Classifier**

As a first baseline, we trained a **DecisionTreeClassifier** on flattened **2-minute window vectors** (120×features) constructed from the six selected CAN signals plus the normalized time feature.

**Configuration:**

* criterion="entropy" (information gain)
* max\_depth=7 (limits tree depth, controls overfitting)
* min\_samples\_split=2, min\_samples\_leaf=1 (allow fine partitions at leaves)
* max\_features=0.02 (~2% of features considered per split to encourage diversity)
* class\_weight="balanced" (mitigates class imbalance across drivers)
* random\_state=903100 (ensures reproducibility)

**Results:**  
The trained tree achieved approximately **94.9% accuracy** and **~0.93 macro-F1** on the test set (643 windows). While most subjects were classified perfectly, a few classes exhibited reduced recall—consistent with the high variance and axis-aligned splits typical of single decision trees.

This model is therefore treated as an interpretable but relatively fragile baseline, against which more robust ensemble and non-linear models (Random Forest, KNN, MLP) are compared in subsequent sections.

**Random Forest Classifier**

Building on the single-tree baseline, we trained a **RandomForestClassifier** on the same flattened **2-minute window vectors** (120×features) derived from the six selected CAN signals plus the normalized time feature.

**Configuration:**  
A small **grid-search-lite** was run (fixing values that performed well in prior sweeps). The best configuration was:

* n\_estimators=8 (number of trees)
* max\_depth=None (trees grown fully, limited by other parameters)
* min\_samples\_leaf=4 (regularization at leaf level)
* max\_features=0.07 (~7% of features considered at each split → promotes decorrelated trees)
* class\_weight="balanced\_subsample" (mitigates class imbalance on a per-bootstrap basis)
* random\_state=903100

**Results:**  
The refit Random Forest achieved **Accuracy ≈ 0.9798** and **Macro-F1 ≈ 0.9787** on the test set (643 windows). The per-class classification report showed near-perfect precision and recall across almost all subjects, with no classes completely missed.

**Discussion:**  
Compared to the single decision tree, the Random Forest benefited from **variance reduction via ensembling** and **decorrelation of features across trees**, which stabilized performance across drivers. This ensemble model established a **strong benchmark**, outperforming the interpretable but fragile decision tree.

**K-Nearest Neighbors (KNN)**

We evaluated a **KNN classifier** on flattened **2-minute window vectors** (120×features) derived from the six selected CAN signals plus the normalized time feature. Because KNN is distance-based, we wrapped the classifier in a **Pipeline(StandardScaler → KNeighborsClassifier)** to ensure that all features contributed equally to the distance metric.

**Configuration (best from grid search):**

* n\_neighbors=1 (nearest-neighbor rule)
* weights="uniform" (equal weight for neighbors)
* metric="minkowski" with p=1 (Manhattan/L1 distance)
* algorithm="brute" (exact search, efficient at this scale)
* leaf\_size=30 (default, not critical here)

**Results:**  
The best configuration achieved **Accuracy ≈ 0.9938** and **Macro-F1 ≈ 0.9935** on the test set (643 windows). Nearly all subjects were classified with perfect precision and recall, with only one or two slightly below 1.0.

**Discussion:**  
The excellent performance with **k=1** and **L1 distance** indicates that driver-specific window vectors form highly separable neighborhoods in feature space. Unlike tree-based models, KNN does not construct decision boundaries; instead, it relies directly on the distribution of windows. This explains both its speed in training (no model fitting beyond storing data) and its high test performance. However, inference cost scales with dataset size, which could become a limitation in deployment compared to Random Forests or MLPs

**Multi-Layer Perceptron (MLP)**

We also trained a **fully connected MLP** on flattened **2-minute window vectors** (120×features) using the six selected CAN signals plus the normalized time feature. Inputs were standardized (StandardScaler) and labels were integer-encoded.

**Architecture & training**

* Hidden blocks: **[512 → 256 → 128]** dense layers
  + He initialization → BatchNorm → ReLU → Dropout(**0.45**) after each block
* Output: softmax over all driver classes
* Optimizer: **Adam (1e-3)**
* Loss: **SparseCategoricalCrossentropy**
* Callbacks:
  + **EarlyStopping**(monitor=loss, patience=50, restore\_best\_weights=True)
  + **ReduceLROnPlateau**(monitor=loss, factor=0.7, patience=28, min\_lr=1e-5)
* Seeds fixed for reproducibility (numpy, tensorflow, PYTHONHASHSEED)

**Results**  
On the 643 temporally separated test windows, the MLP achieved **Accuracy ≈ 0.9129** and **Macro-F1 ≈ 0.8979**. Many subjects reached perfect precision/recall, with a subset showing reduced recall—consistent with (i) relatively strong regularization (Dropout 0.45), (ii) conservative LR scheduling and early stopping, and (iii) the challenge of learning from high-dimensional flattened temporal vectors without an explicit sequence model.

**Methodological note (early stopping without a validation set)**  
For the MLP, early stopping and LR scheduling were monitored on **training loss** rather than a separate validation set. This was a deliberate choice to maintain strict train/test separation and to keep results consistent with the project’s stated evaluation policy (reporting **best-case results on this test split**). While monitoring training loss cannot fully detect overfitting, the combination of BatchNorm, substantial Dropout, capped epochs, and transparent reporting makes this setup acceptable for **exploratory benchmarking**. Future work can introduce a validation split or cross-validation and/or sequence-aware models (e.g., 1-D CNNs, Transformers) to probe further gains.