Bachelor Thesis

Machine Learning-Based State Classification in Memristive IV Traces from Mechanically Controlled Break Junctions

Max Doornbos

TU Delft

March 13, 2026

Supervisors:
Prof.dr.ir. H.S.J. van der Zant
Prof.dr.ir. H.X. Lin
Riccardo Conte

Abstract

This bachelor thesis develops a four stage machine-learning pipeline for pointwise conductance state classification in memristive current-voltage traces from mechanically controlled break junction experiments. The aim of this thesis is to improve data retention of hysteretic traces from the raw data set for downstream physics analysis despite an absence of ground truth pointwise state classifications and heterogeneous traces. The pipeline consists of an initial data filter, pseudo-labelling with a simple linear regression Hidden Markov Model (teacher), a Temporal Convolutional Network (student) capable of large-scale pointwise conductance classification and a final hysteretic filter. The methodology is developed with data from three reference molecules expected to exhibit two-state behaviour, while two additional target molecules that are expected to show occasional three conductance state behaviour are used for comparison. On the test–set of unseen data, the temporal convolutional network reproduces the teacher pseudo-labels with 98% pointwise agreement. The pipeline also identifies 8% of all traces in the available dataset as hysteretic, compared with the 5% resulting from the methodology used in previous work. Manual audit and plausibility analyses indicate that the increase in retained traces is achieved without losing physical plausibility for most molecules. Limitations include a lack of definitive ground truth and a reduced robustness on complex three conductance state target molecules.

Contents

Chapter 1

Introduction

1.1 Background and motivation

The focus of this thesis is to examine memristive switching in current-voltage data from a mechanically controlled break junction experiment. The data for the experiment is collected by carrying out voltage sweeps on a single molecule in a junction. The current is then measured to obtain a current-voltage curve. The important parameter obtained from the data is the conductance G, which is given by $G = \frac{I(V)}{V}$. There is a small set of data in the IV curve for which the conductance at a given voltage depends on the history of the voltage applied. This indicates that the molecule has memory effects [1].

The data used in this thesis was originally collected and first studied in the master thesis of van der Geest [2]. In that work, elementary methods were employed to classify the points in each IV trace as belonging to a particular conductance-state. These clustering/classification methods proved sufficient, but in the context of this work, they are too limited for robust large-scale implementation. This is because they led to a large fraction of otherwise useful IVs being discarded due to misclassification of points. As a result, downstream analyses were computed from a reduced subset of the available data, limiting the robustness of the aggregated results. The central motivation of this thesis is to improve pointwise state classification such that more hysteretic IV traces can be retained and used for downstream physics analysis.

The aim is to extract switching characteristics from the available datasets, in particular conductance ratios (the ratio between the average conductances of the states present), threshold voltages (the bias voltage at which a molecule switches from one state to another) and conductance state distributions (the distribution of conductances of a given conductance state in a molecule). These quantities are only informative when they are computed for many traces and aggregated per molecule. The available dataset contains measurements for five molecules. Three of these molecules are called reference molecules and are expected to exhibit mainly two conductance-states. The last two molecules are target molecules that may, in principle, exhibit proton hopping and therefore result in a more complex conductance-state structure. In this dataset of all five molecules, there are in total 128,848 IV files, of which 71,177 are measurements on the reference molecules, and 57,671 are measurements of the target molecules. Each of these IV files contains 200 data points each, meaning that consistent pointwise classification is required at large scale.

Extracting threshold voltages, conductances and conductance ratios requires discretely as-

signing each point in each IV trace to a conductance-state. However, real IV data are noisy and heterogeneous: most IVs do not exhibit coherent switching, conductance levels can drift or overlap, and many points do not clearly belong to any single state [2].

For this reason, the bottleneck of the MCBJ experiment is not measurement, but robust and reproducible clustering of IV points into coherent conductance-states. This thesis employs machine learning techniques to build a clustering pipeline that retains a larger set of hysteretic IVs for analysis, enabling more robust conductance ratio and threshold voltage statistics. The resulting pipeline is also intended to be deployable during future measurements, so that newly acquired data can be screened more effectively for usefulness in downstream analysis.

1.2 Problem statement and objectives

The input to this work is a dataset consisting of conductance-voltage measurements of an MCBJ experiment. The dataset consists of measurements of five different molecules, of which three reference molecules that are expected to exhibit two-state behaviour and two target molecules expected to exhibit more complex behaviour. Per molecule, thousands of IV measurements are available, where each measurement consists of around 200 current-voltage data points (and associated measurement metadata).

Although the full dataset considered consists of five molecules, only the data from the reference molecules will be used for model development and training. This is because the pipeline is designed as a classifier of two-state IVs, and the two target molecules can, in principle, exhibit proton-hopping-related three-state conductance behaviour. By excluding the target molecule data, possible three-state structures cannot interfere with model development. The target molecules are, however, kept in the dataset for comparative analyses, particularly those involving conductance state and conductance ratio results.

The central task is pointwise state assignment. Formally, given a sequence of current-voltage points in an IV trace, the goal is to classify each point as belonging to one of a small number of conductance-states. These pointwise labels are then used to compute switching characteristics, specifically threshold voltages and conductance ratios. The main research question is: To what extent can machine learning techniques be used to produce a robust, well-performing, and practically useful model for pointwise classification of IV points into conductance-states, despite noisy measurements, heterogeneous trace quality, and the absence of ground-truth labels?

The main contributions of this thesis are:

- The development of filtering and preprocessing to obtain high-quality two-state IVs from the reference molecules, suitable for model development;
- The implementation of a teacher model (SLR-HMM) that incorporates state persistence and generates pseudo-labels;
- The training of a student model (TCN) on these pseudo-labels to produce scalable per-point classifications;
- The construction of a hysteretic filter that selects traces used for downstream analysis
- The evaluation of the resulting pipeline using student-teacher agreement, manual audit, physical plausibility metrics and downstream physical statistics

The scope of the final neural network is not to achieve perfect pointwise accuracy, which cannot be established in the absence of ground truth, but rather to increase the amount of data that can be retained for downstream physics analysis.

1.3 Approach

The approach in this thesis is a four-stage pipeline that turns raw IV files into a trained neural network capable of producing reliable state labels and, from these labels, improved switching statistics.

The training and development of the pipeline will only be done with reference molecule data, since these reference molecules are not likely to exhibit complex conductance behaviour like proton hopping, that may result in more than two conductance-states. The data of the two target molecules are retained for later comparative analysis.

The first phase is an initial filter that removes IVs that clearly do not exhibit coherent behaviour and are very unlikely to be relevant in later pipeline stages. This reduces the dataset to a subset of relevant IVs on which models can be trained.

The second phase is a teacher model that generates pseudo-labels for supervised learning. Fitting two RANSAC lines onto an IV can yield a reasonable initial classification, but this classification is often wrong because it does not encode state persistence. By combining a Hidden Markov Model with the RANSAC-derived plateau structure, the teacher model incorporates knowledge of infrequent switching between states. Modelling state persistence is essential, since a purely geometric assignment based on the RANSAC lines frequently fails in switching regions and noisy segments.

Third, the pseudo-labels produced by the SLR-HMM are used to train a student model. The student model chosen is a temporal convolutional network (TCN). While the teacher can produce high-quality classifications, it is computationally expensive and requires multiple tuning choices. In contrast, a trained TCN can reproduce the teacher-style classifications without per-trace fine-tuning and can be applied efficiently once trained.

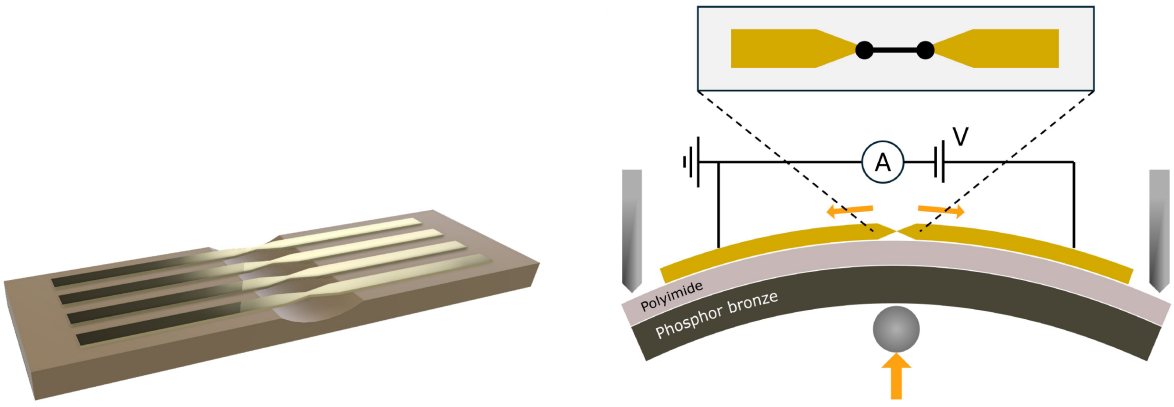
Lastly, a hysteresis filter is applied to ensure that downstream physics analysis is only performed on IVs that exhibit a true hysteretic structure. This hysteresis filter is largely based on the memristivity-related rules presented in the thesis by van der Geest [2]. The criteria ensure that multiple states are present, switching occurs across the voltage domain, and that states are sufficiently visited. The filter is used in combination with the labels produced by the TCN, yielding a final set of well-classified hysteretic IVs appropriate for conductance ratio and threshold voltage analysis.

Chapter 2

Background

2.1 Mechanically controlled break junction measurements

A mechanically controlled break junction (MCBJ) is a common experiment in the field of molecular electronics that is used to probe charge transport through single molecules [3]. In MCBJ experiments, four gold wires connected to a circuit are placed on a flexible surface consisting of polyimide and phosphor bronze. On these gold wires, a solution containing the target molecule is drop cast. Then, by applying pressure to the surface from below, the wires break causing their broken ends to act as electrodes. In the process of breaking, a single molecule is trapped between the electrodes, thereby completing the circuit. A bias voltage is applied across the electrodes, resulting in a measured current flowing across the single molecule. By changing the applied bias voltage V, the current $I(V)$ flowing across the single molecule can be measured. This measurement results in current-voltage traces (IVs), showing how $I(V)$ varies with V. Due to the fact that the nanoscopic junction configuration can change between molecules, IV traces exhibit very significant variability, even when measurement settings are identical.



(a) (b)

Figure 2.1: **A Mechanically Controlled Break Junction diagram** [2]. In a MCBJ experiment, four golden wires (a) are fixed on a flexible surface (b), on which a solution containing the target molecule is drop cast. As the surface is bent and the golden wires break, turning them into sets of electrodes between which a single molecule from the solution can be trapped. The molecule acts as a bridge between electrodes, whereby a bias voltage is applied across the electrodes, and the resulting current is measured.

In order to make an IV trace measurement, the voltage across the electrodes must be varied. In MCBJ experiments, this is done by *sweeping* the voltage across a certain set voltage range. A voltage sweep is when the voltage is varied between two values, in order to measure the resulting current $I(V)$ across the molecule. In the IV datasets considered in this thesis, one IV measurement consists of a half sweep starting at zero bias, then two full sweeps, and then another half sweep back to zero bias. To be precise, these sweeps are in the form:

$$0 \to \pm V_{\max} \to \mp V_{\max} \to \pm V_{\max} \to 0,$$

with $V_{\max} \approx 1000\,\mathrm{mV}$. [2]

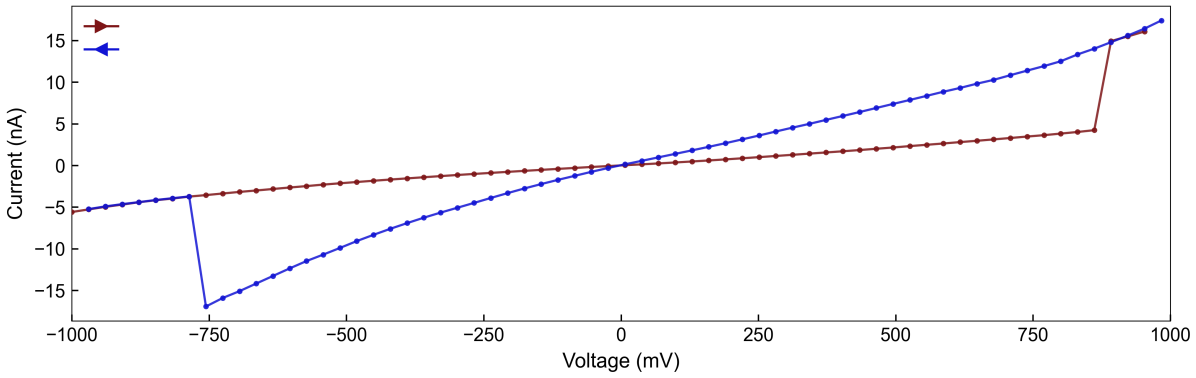


Figure 2.2: **A hysteretic IV measurement.** A coherent example of an IV measurement from the MCBJ experiment with minimal noise. The voltage up and down full sweeps denoted in red and blue respectively.

2.2 Conductance and common representations of IV data

Generally, when considering IV data produced by MCBJ systems, it is most relevant to calculate the conductance of the molecule at each point in the voltage sweep. Conductance is conceptually defined as:

$$G(V) = \frac{I(V)}{V}. \tag{2.1}$$

However, in practice, directly computing this is numerically unstable near $V = 0$, motivating the use of a voltage cutoff around zero bias [4].

With conductance appropriately calculated, it is often convenient to proceed with data-analysis in the logarithmic conductance domain[2],

$$x_t = \log_{10}\big(\max(|G_t|, \varepsilon)\big), \tag{2.2}$$

where $\varepsilon > 0$ prevents $\log(0)$. By implementing log-conductance, dynamic range is reduced thereby resulting in GV plots of IV traces where conductance-states appear as approximately separated clusters.

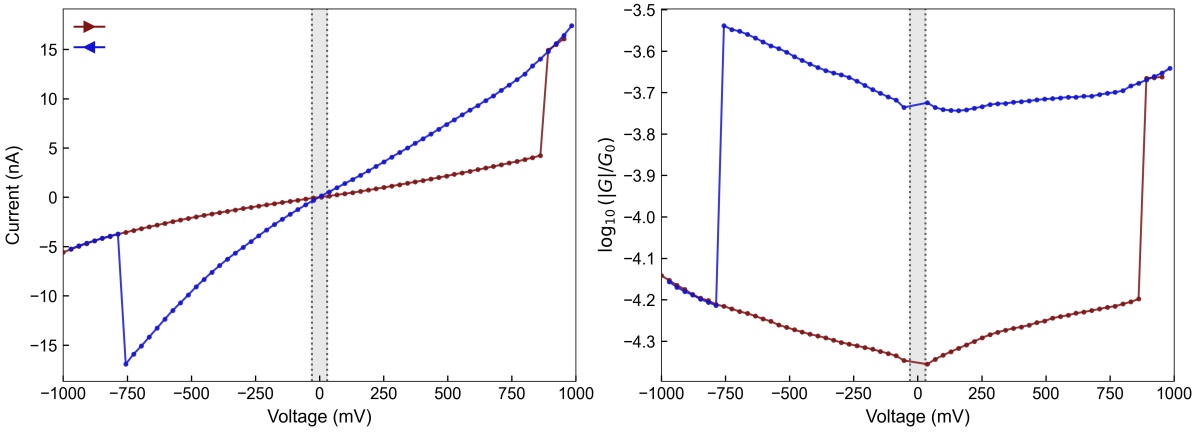


Figure 2.3: **A hysteretic IV and its associated GV. Left:** A hysteretic IV trace. **Right:** The GV plot that results from calculating the conductance at V of the IV. This GV plot is clearly also hysteretic. Both the IV and GV plots consist of only the full voltage sweeps, with the voltage down sweep in blue, and the voltage up sweep in red.

2.3 Single-molecule transport and conductance-states

Having defined the conductance G as a measured quantity that can be derived pointwise from IV traces, it is now useful to motivate the existence of discrete conductance levels in single-molecule MCBJ experiments. Discrete conductance levels can be understood as the consequence of the high sensitivity of molecular-junction transport to specific nanoscopic configuration of a molecule. A widely understood conceptual framework for coherent and elastic transport in MCBJ experiments is the Landauer picture, where the conductance is related to the electronic transmission of the junction [5]. For small voltage biases, one obtains

$$G \approx G_0\, T(E_F), \qquad G_0 = \frac{2e^2}{h}, \tag{2.3}$$

where $T(E_F)$ is the transmission evaluated at the relevant Fermi energy, and G_0 is the conductance quantum constant. This relation is not implemented in this thesis to model conductance-states, but provides a qualitative point: if the configuration of the molecule/junction changes (e.g. by contact geometry, coupling strength or local electrostatics), then $T(E_F)$ can change significantly, thereby causing the measured conductance to shift by many orders of magnitude [5].

This sensitivity alone does not explain why only a small number of conductance values are observed repeatedly within a single measurement. At the low temperatures relevant for the present MCBJ experiments ($\sim 6K$), the junction is expected to be stable, giving well-defined IVs that remain approximately constant in time. This means that, for extended parts of a voltage sweep, the molecule remains in one particular junction configuration, which causes the transmission $T(E_F)$ and subsequently the conductance G also remain approximately constant. Occasionally however, the molecular configuration in the junction changes due to thermal activation or as a result of the applied bias voltage modifying the stability of the junction. When a configuration like this occurs, the transmission $T(E_F)$ changes abruptly, leading to a jump to a different conductance level. This two-state behaviour can be understood as repeated switching between a

very small number of relatively stable molecule configurations in the junction.

A commonly used simplified example of single-molecule MCBJ transport is the single-level model [5]. At zero-bias ($V = 0$), the transmission as a function of the electron energy E depends on both the molecular level position ε_0 and the couplings to the left and right electrodes, Γ_L and Γ_R. In a standard convention, this transmission is written in Lorentzian (Breit–Wigner) form:

$$T(E) \approx \frac{\Gamma_L \Gamma_R}{(E - \varepsilon_0)^2 + \left(\frac{\Gamma_L + \Gamma_R}{2}\right)^2}, \tag{2.4}$$

This illustrates how small shifts in ε_0 or in the couplings $\Gamma_{L,R}$ can strongly influence $T(E_F)$, and thereby G.

The above relations are merely included as a qualitative motivation for the extreme sensitivity of conductance to junction configuration. The hysteretic memory effect studied in this thesis requires significant additional explanation on other internal degrees of freedom such as structural rearrangements, ionic motion and charge trapping. These are not explained by coherent transport alone [2][6].

2.4 Hysteresis and memristive switching

When analysing IV and GV data, it is clear that in a small subset of the available data the aforementioned conductance plateaus/levels are visited for extended periods of time, until a switch occurs. Due to this, it can be said that the conductance-state in which a molecule presides at a given moment during a set of voltage sweeps, depends on the history of the voltage bias applied across this molecule. The resulting IV and GV plots are therefore considered as hysteretics, since a hysteretic system is a system whose state is defined by the history of the system. More specifically, any two-terminal electrical component that exhibits such hysteretic behaviour in the flow of electrical current is called a *memristor*, which is a conjunction of the words memory and resistor [1]. In this thesis, the term "memristive switching" is used to describe this experimentally observed memory effect.

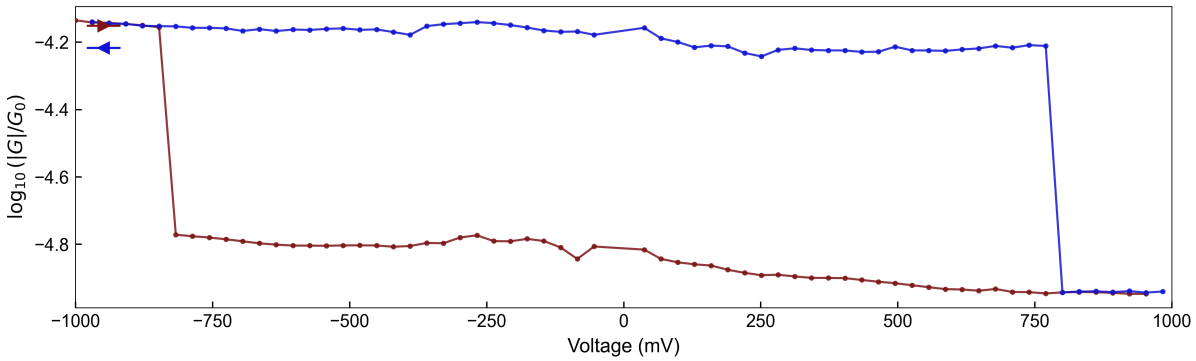


Figure 2.4: **A hysteretic GV plot.** With the $+V$ and $-V$ sweeps marked in red and blue respectively. A clear hysteretic structure is visible as the conductance of the system depends on its history.

A convenient way to describe the memory of a system that observes memristive switching is to

introduce an internal state variable $w(t)$ that changes on experimental timescales and influences conductance [1]. One common form is

$$I(t) = G\big(V(t), w(t)\big)\, V(t), \qquad \dot{w}(t) = f\big(w(t), V(t)\big), \tag{2.5}$$

where f describes how the internal state changes under applied bias. When w evolves slowly compared to the sampling rate, the system generally remains in a quasi-stable conductance-state for many points, and it only transitions to another state occasionally, producing what is called *state-persistence* and clear hysteresis in observed conductance.

From a physical point of view, the switching behavior between conductance-states can be summarized by the analysis of two quantities used throughout this thesis: threshold voltages and conductance ratios. The *conductance ratio* simply quantifies the separation between two conductance-states and is defined as [2]

$$R_G = \frac{G_{\text{high}}}{G_{\text{low}}}, \tag{2.6}$$

where G_{high} and G_{low} are the conductance levels for the high and low conductance-states. The *threshold voltage* V_{th} is the bias voltage at which the system transitions from one conductance-state to another during a sweep. In systems observing memristive switching, the threshold voltages can depend on polarity and sweep direction [2].

These switch characteristics are only informative when aggregated over many IV traces for a given molecule; thereby producing distributions rather than single values. These distributions provide a compact way to compare switching behaviour across molecules.

2.5 Possible switching mechanisms

In order to compare switching behaviour across molecules, it is important to understand that not all switching is the same.

In general, a change in conductance of a molecule means that something in the molecular junction has changed. Examples of this include the molecular configuration, the way the molecule is in contact with the electrodes or the electronic state of the system. Different physical changes in the junction give rise to different switching mechanisms, which are generally associated with differing conductance ratios. Different physical mechanisms can lead to different typical conductance ratios, although these typical conductance ratios can overlap [2]. This is why analysing the conductance ratio distributions and the state conductance distributions are useful for understanding the underlying mechanisms behind the changes in the junction.

Switching mechanisms can be divided into two categories, *extrinsic* and *intrinsic* mechanisms. Extrinsic mechanisms correspond to switching caused by changes in the junction configuration that are external to the internal structure of the molecule itself such as contact geometry or molecular orientation. Intrinsic mechanisms correspond to changes in the molecule itself, such as isomerization or charge state alternation [2]. In practice, conductance ratios cannot be used to uniquely identify which switching mechanisms are being observed, but they can give an indication as to which mechanisms are plausible.

Table 2.1 below summarizes which conductance ratio ranges are associated with differing switching mechanisms, as based on earlier work [2]. This table is given as an overview, in order to support the interpretation of conductance ratio distributions later in this thesis.

Table 2.1: **Approximate conductance-ratio ranges for several switching mechanisms** [2].

Switch mechanism	**Typical ratio R_G**	**Interpretation**
Contact geometry change	$\sim$ 2–4	Small changes in molecule-electrode coupling.
Isomerization	$\sim$ 2–100	Structural rearrangement within the molecule.
Injection point change	$\sim$ 10–100	Change in effective transport pathway.
Open-close switching	$\gtrsim 100$	Partial attachment or detachment of the junction.
π–π stacking / parallel-molecule transport	~ 100	Additional transport pathway through multiple molecules.
Charge-state alternation	$\sim$ 100–1000	Change in molecular charge state causing a strong conductance shift.

2.6 Proton hopping

A specific switching mechanism that is relevant to the target molecules considered in this thesis is *proton hopping*. Proton hopping is when a proton within the molecule transfers between different positions, due to the applied electric field. Because a proton hop changes local electronic structure, it can also change the measured conductance [2].

In the simplest case, proton hopping can produce switching between two states. However, in certain molecules like the target molecules, protons can hop to two different positions in the molecule. For this reason, proton hopping can, in principle, lead to three state conductance behaviour in the molecular junction, instead of simply two states [2]. This is relevant for interpreting the conductance and conductance ratio results of the target molecules. An example of a hysteretic three state GV plot is given below in figure 2.5.

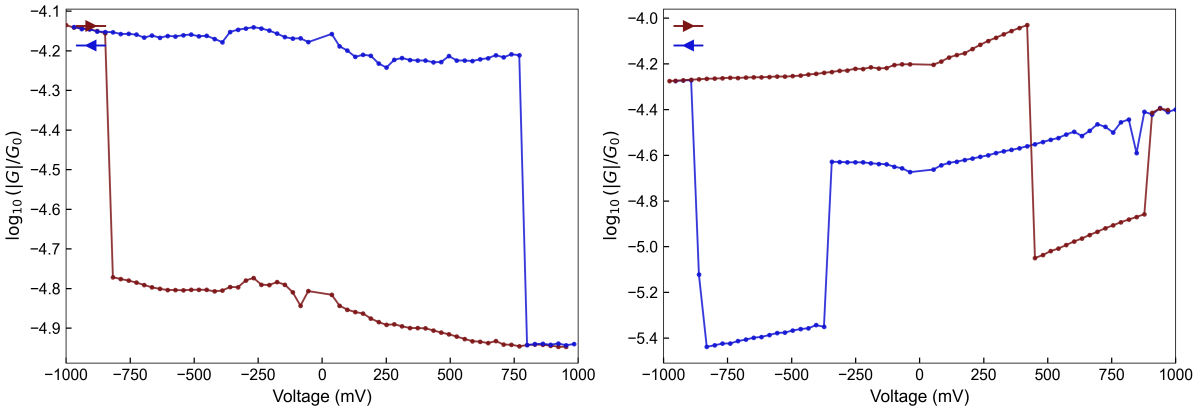


Figure 2.5: **Example GVs of a two- and three-conductance-state system.** Two conductance-voltage plots resulting from IV measurements in a MCBJ. **Left:** An ideal hysteretic two state GV from a reference molecule. **Right:** An ideal hysteretic GV plot from a target molecule that contains three different conductance-states.

In this thesis, an understanding of proton hopping is relevant as an interpretive mechanism for the conductance results of the target molecules. However, observing a three conductance-state structure that corresponds to proton hopping in measurements is expected to be rare. For this reason, it cannot be established that proton hopping occurs from conductance statistics alone. Other mechanisms can produce similar conductance results as proton hopping would, which means that any data interpretation regarding proton hopping remains tentative [2].

Due to the fact that the target molecules in dataset may exhibit proton hopping, and therefore a three-state-conductance structure, the data from these molecules are not used to train the two-state classification pipeline developed in this thesis.

2.7 Robust estimation and RANSAC

Almost all model-fitting tasks start by using the same ingredients: a set of measured data points $\{(u_i, y_i)\}_{i=1}^n$, a model $m_\theta(\cdot)$ with parameters θ and residuals that can quantify the mismatch between model and data,

$$r_i(\theta) = y_i - m_\theta(u_i). \tag{2.7}$$

A common approach is to choose θ by minimizing the total loss over these residuals. The classical approach is least squares, where

$$\hat{\theta}_{\text{LS}} = \arg\min_\theta \sum_{i=1}^{n} r_i(\theta)^2, \tag{2.8}$$

The least squares method is optimal under the assumption of independent Gaussian noise. However, due to the nature of θ, this method is severely influenced by outliers. A single large residual contributes quadratically to θ and can therefore dominate the fit [7].

RANSAC (Random Sample Consensus) fitting takes a fundamentally different approach: it explicitly assumes that the data consists of two sets, an *inlier set* consistent with the model and an *outlier set* that may be arbitrary [7]. For a predetermined residual threshold $\tau > 0$, a point is

called an inlier when

$$|r_i(\theta)| \leq \tau. \tag{2.9}$$

What RANSAC does is repeatedly sample minimal subsets of points, fits candidate models on that subset and computes the corresponding inlier set using (2.9). By doing this process for a large quantity of randomly sampled subsets of the data, and calculating the error repeatedly, the RANSAC fit selects the model with the largest consensus set (or best inlier fit).

The key difference between the least squares method and RANSAC is that RANSAC is designed to ignore any points that do not agree with the most dominant structure in the data. Therefore, despite a particular data set being very noisy, a random sample consensus fit can still model the part of the data that does not appear to be noise, while ignoring the rest by putting other data points in the *outlier set*.

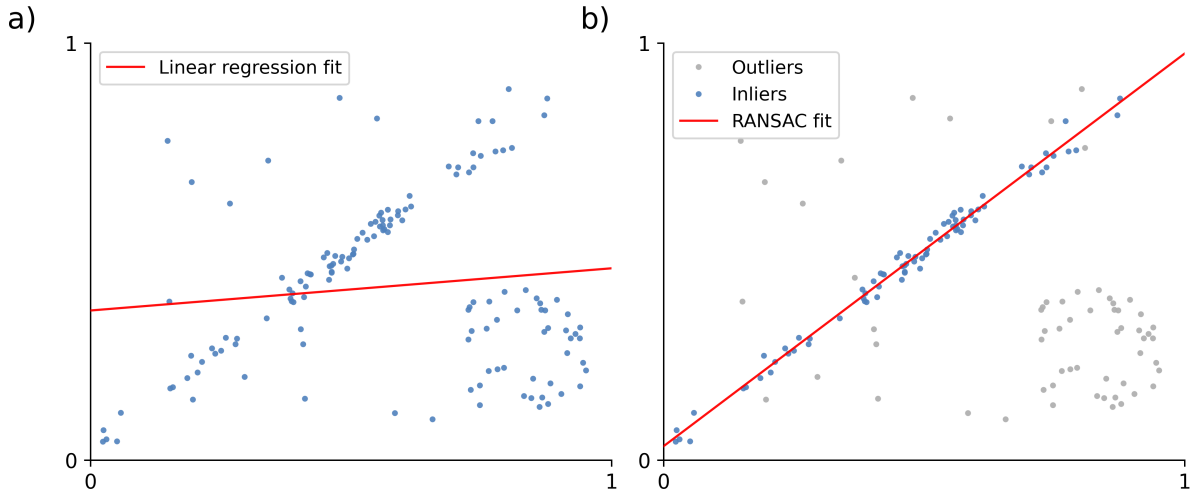


Figure 2.6: **A comparison between linear regression and RANSAC. a)** A linear regression fitted on a dataset that contains a clear linear structure. Due to the presence of a dense set of outliers, the linear regression fails to accurately model the linear structure. **b)** A RANSAC fitted line where the dense set of outliers are all considered outliers, and therefore do not contribute error to the fit. The RANSAC line perfectly fits the prevalent linear structure.

2.8 Clustering methods used in earlier work

In the earlier work by van der Geest, on which this thesis is based, three pointwise clustering approaches were used to label IV points without modelling temporal persistence: Gaussian mixture models in one and two dimensions, and agglomerative clustering in two dimensions [2]. These approaches form the baseline against which the classification model pipeline in this thesis is compared.

Gaussian mixture models (GMMs)

In a Gaussian mixture model, it is assumed that the density of a scalar feature x can be written as

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x \,|\, \mu_k, \sigma_k^2), \qquad \sum_{k=1}^{K} \pi_k = 1, \tag{2.10}$$

where π_k are mixture weights and (μ_k, σ_k^2) are component parameters [8]. In the two dimensional case, the the scalar feature is expanded to a two dimensional vector, and the Gaussian components are generalized to multivariate normal distributions.

The parameters used in the Gaussian mixture models are fitted by expectation-maximization (EM) [8]. After this fitting, the posterior responsibility of component k for a point t is given by

$$\gamma_{t,k} = \mathbb{P}(z_t = k \mid x_t) = \frac{\pi_k \mathcal{N}(x_t \mid \mu_k, \sigma_k^2)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_t \mid \mu_j, \sigma_j^2)}. \tag{2.11}$$

Then, a hard label is assigned to each point by $\hat{z}_t = \arg\max_k \gamma_{t,k}$.

The process of selecting the number of mixture components is a model selection problem for which a common criterion is the Bayesian Information Criterion (BIC) [8]:

$$\mathrm{BIC} = -2 \log \hat{L} + p \log n, \tag{2.12}$$

where $\hat{L}$ is the maximised likelihood, p is the number of free parameters, and n is the number of data points.

Agglomerative clustering in 2D

Hierarchical agglomerative clustering starts by treating each point as its own cluster, and then it merges clusters iteratively according to a certain linkage criterion [9]. In something called *Ward's method*, merges are chosen in order to minimise the increase in variance within clusters. Given clusters C_a and C_b with sizes $|C_a|$ and $|C_b|$ and centroids $\boldsymbol{\mu}_a$ and $\boldsymbol{\mu}_b$, the Ward merge cost can be written as

$$\Delta(C_a, C_b) = \frac{|C_a||C_b|}{|C_a| + |C_b|} \|\boldsymbol{\mu}_a - \boldsymbol{\mu}_b\|^2. \tag{2.13}$$

With this cost, merging is done repeatedly until two clusters remain that form a two-state partition of the data [9].

How earlier work used these methods

Earlier work by van der Geest applies: (i) a 1D GMM on a log-conductance-derived feature, (ii) a 2D GMM on (V, x), and (iii) 2D agglomerative clustering on (V, x) to obtain two pointwise clusters [2]. After labelling all points, the cluster identities were ordered by average conductance. For the resulting downstream analysis, conductance curves per cluster were constructed by the interpolation of the clusters on a fixed bias grid, and a *threshold curve* was defined by the midpoint between the two cluster curves. Based on this threshold curve, the switching thresholds were extracted. Subsequently, a set of rules was used to decide whether or not an IV trace was sufficiently memristive for use in the downstream statistical analysis. Results were combined across methods by majority vote.

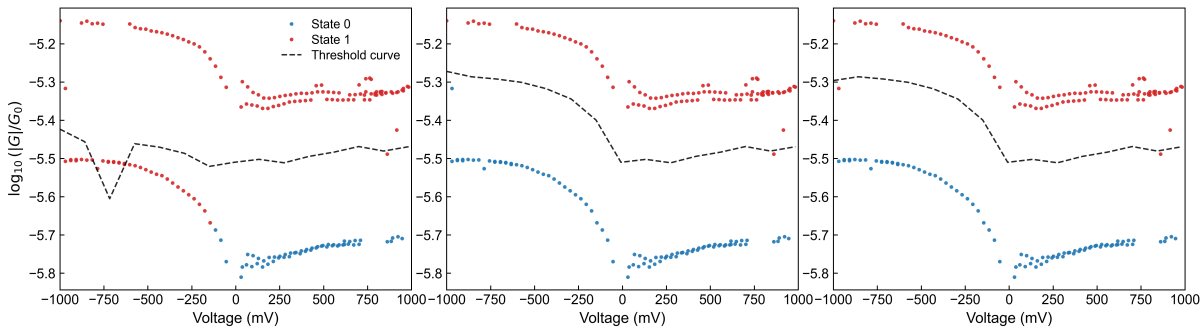


Figure 2.7: **Example of effective pointwise clustering results using three baseline methods.** Panels show **Left:** Gaussian Mixture Clustering in the one-dimensional conductance space. **Center:** Gaussian Mixture Clustering in the two dimensional $(V, \log_{10} G)$ space. **Right:** Agglomerative clustering in the two dimensional $(V, \log_{10} G)$ space. Points are coloured according to their assigned conductance-state, while the dashed curve indicates the midpoint threshold separating the two states.

Due to the fact that these three methods do not encode any state-persistence, they are sensitive to noise and ambiguous switching regions, and can therefore result in rapid or incorrect label switching that is inconsistent with the physical expectation of state-persistence, such as in figure 2.8. According to the majority vote rule, the below example would not be characterised as a useful hysteretic to use in downstream analysis due to wrong clustering.

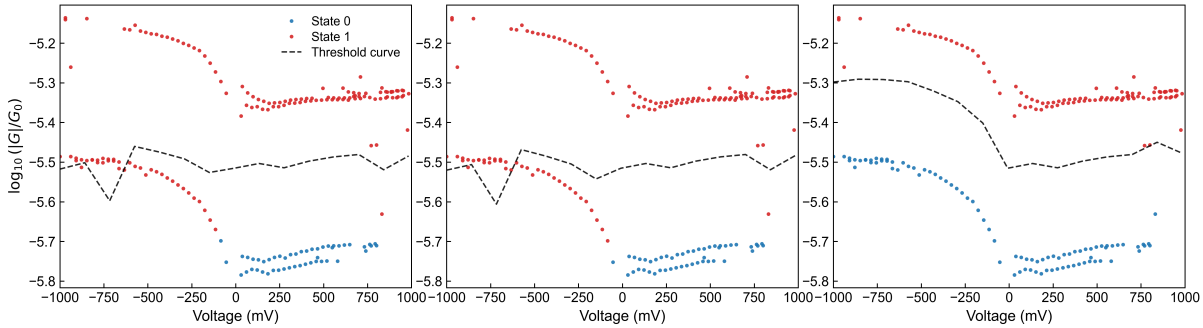


Figure 2.8: **Example of ineffective pointwise clustering results using three baseline methods.** Panels showing the performance of the three baselines on another GV from the exact same measurement set (with the same measurement setup) as in 2.7. Panels show **Left:** Gaussian Mixture Clustering in the one-dimensional conductance space. **Center:** Gaussian Mixture Clustering in the two dimensional $(V, \log_{10} G)$ space. **Right:** Agglomerative clustering in the two dimensional $(V, \log_{10} G)$ space. Points are coloured according to their assigned conductance-state, while the dashed curve indicates the midpoint theshold seperating the two states.

In addition to this, the GVs in figures 2.7 and 2.8 are from the same measurement set, and therefore represent the exact same two state structure. However, as is clear, in the measurements shown in figure 2.7, two of the three baseline clustering methods perform well whereas in figure 2.8 only one in three baseline methods performs well. The differences between the GV plots themselves are minute, with the general structure of the GVs being identical. Despite this nearly identical shape, the majority vote rule would misclassify the GV in figure 2.8. This demonstrates that, along with high sensitivity to noise, high frequency switching regions, and shifting conductance-states, these methods used in previous work are also high in variability

across very similar measurements.

These three clustering methods operate pointwise and therefore ignore temporal structure. This motivates sequential models such as Hidden Markov Models.

2.9 Hidden Markov Models for sequential labelling

There are many sequential datasets that exhibit the same property: the system occupies a small number of regimes for extended periods of time, with occasional transitions between them. A Hidden Markov Model (HMM) is a probabilistic model that is designed to capture the logic behind datasets such as these. The core idea behind an HMM is to separate an unobserved discrete state labelling from noisy observations generated by those states [10]. In the context of MCBJ data, the hidden state can be interpreted as the conductance-state, and the observed signal is a conductance derived feature, such as $x_t = \log_{10}(|G_t|)$.

Formally, an HMM consists of:

- hidden states $z_t \in \{1, \ldots, K\}$,
- an initial distribution $\pi_k = \mathbb{P}(z_1 = k)$,
- transition probabilities $A_{ij} = \mathbb{P}(z_t = j \mid z_{t-1} = i)$,
- emission models $p(x_t \mid z_t = k)$ describing observations in each state.

The Markov property is

$$\mathbb{P}(z_t \mid z_{1:t-1}) = \mathbb{P}(z_t \mid z_{t-1}), \tag{2.14}$$

and conditional independence of emissions gives

$$\mathbb{P}(x_{1:T} \mid z_{1:T}) = \prod_{t=1}^{T} p(x_t \mid z_t). \tag{2.15}$$

In many applications, including conductance-state labeling, a common choice is a Gaussian emission model:

$$x_t \mid (z_t = k) \sim \mathcal{N}(\mu_k, \sigma_k^2), \tag{2.16}$$

which says that each state corresponds to a characteristic mean level with noise [10]. In an HMM, state persistence is naturally encoded by large self-transition probabilities A_{kk}, which make remaining in the same state likely and switching relatively rare.

A central inference problem is estimating the most likely state sequence given observations:

$$\hat{z}_{1:T} = \arg\max_{z_{1:T}} \; \pi_{z_1}\, p(x_1|z_1) \prod_{t=2}^{T} A_{z_{t-1},z_t}\, p(x_t|z_t), \tag{2.17}$$

This is computed efficiently with use of dynamic programming (the Viterbi algorithm) [10]. When parameters are unknown, they can be learned from the data by use of expectation-maximisation, which is known as the Baum–Welch algorithm in the context of HMMs.

2.10 Neural networks and Temporal Convolutional Networks

A neural network is, in essence, a function that maps inputs to outputs through layers of simple calculations [11]. By composing linear transformations with nonlinear activation functions, a neural network can represent complex nonlinear relationships. For some input vector $\mathbf{u} \in \mathbb{R}^d$, a single (fully connected) layer computes

$$\mathbf{h} = \phi(W\mathbf{u} + \mathbf{b}), \tag{2.18}$$

where W is a weight matrix, $\mathbf{b}$ is a bias vector, and $\phi(\cdot)$ is a non-linear activation function. In a deep neural network, multiple layers are stacked:

$$\mathbf{h}^{(\ell)} = \phi\Big(W^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\Big), \tag{2.19}$$

starting from $\mathbf{h}^{(0)} = \mathbf{u}$ [11].

When a neural network is trained on training data, it is constantly fine tuning its choices for the parameters $\theta = \{W^{(\ell)}, \mathbf{b}^{(\ell)}\}_\ell$ in order to minimise a loss over a dataset [11]. When applying neural networks to classification into K classes, the network produces logits $\mathbf{s} \in \mathbb{R}^K$ which are then converted to probabilities by softmax:

$$p(k \mid \mathbf{u}) = \frac{\exp(s_k)}{\sum_{j=1}^{K} \exp(s_j)}. \tag{2.20}$$

Given a target label $\tilde{z}$, a standard loss is cross-entropy loss:

$$\mathcal{L} = -\log p(\tilde{z} \mid \mathbf{u}), \tag{2.21}$$

and subsequent optimization is done by gradient-based methods as backpropagation is used to compute gradients and an optimizer is used to update θ.

An important concept in applied machine learning is the *feature representation*. A feature vector $\mathbf{u}$ is a set of quantities derived from raw measurements that is provided to the nueral network for it to learn from. Good feature selection reduces irrelevant variability and highlights the structure relevant to the task. Overfitting occurs in a neural nework when a model learns the noise and idiosyncrasies of the training data, that do not generalise to the dataset on which the neural network is tested on.

For sequential data, a type of neural network called a Temporal Convolution Network (TCN) is relevant because it uses local context in a data sequence to learn [12]. TCNs achieve this by implementing one dimensional convolutions. A 1D convolution computes features at time t by combining inputs from a local window:

$$(\mathbf{u} * \mathbf{w})_t = \sum_{i=0}^{m-1} w_i \, \mathbf{u}_{t-i}, \tag{2.22}$$

where $\mathbf{w}$ is a filter of length m. TCN models use dilated convolutions to increase the receptive field of the context window, allowing for the network to learn context. A dilated convolution with

a dilation d uses inputs spaced apart by:

$$(\mathbf{u} *_d \mathbf{w})_t = \sum_{i=0}^{m-1} w_i \, \mathbf{u}_{t-id}. \tag{2.23}$$

By stacking layers with increasing dilation, the receptive field, or *context window* grows efficiently, allowing the model to use a wider temporal context [12].

For K-state sequence classification, a TCN produces logits $\mathbf{s}_t \in \mathbb{R}^K$ at each time index t, which are converted to probabilities by softmax 2.20. A standard training objective is the sequence cross-entropy loss

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log p_\theta(z_t = \tilde{z}_t \mid \mathbf{u}_{1:T}), \tag{2.24}$$

optionally with masking when certain indices are excluded [12].

Chapter 3

Data & Problem Formulation

3.1 Dataset

The dataset in this thesis consists of the IV traces obtained from 5 different molecules subjected to the MCBJ experiment [2]. Three of these molecules are called reference molecules, and the last two are called target molecules. This is because the reference molecules are expected to exhibit only two conductance-states, whereas the target molecules may exhibit three conductance-states. An overview is given in Table 3.1.

Table 3.1: **Overview of the molecular datasets used in this thesis.** Reference molecules are expected to exhibit predominantly two-state switching behaviour, whereas target molecules may exhibit proton-hopping-related three-state switching behaviour.

Molecule	**Number of traces**	**Expected behaviour**
1_MSH0443	17,984	Reference molecule, expected two-state system
11_MSH0477	14,333	Reference molecule, expected two-state system
12_MSH0501	38,860	Reference molecule, expected two-state system
id4_MSH0028	26,051	Target molecule, may exhibit proton hopping / three-state behaviour
10_MMA1944-10D	31,620	Target molecule, may exhibit proton hopping / three-state behaviour

In total, this dataset contains 128,848 IV traces, each of which contains approximately 195 data points.

Each trace spans approximately $\pm 1000\,\mathrm{mV}$ and approximately half of the voltage sweeps begin with the positive-going sweep, and the other half with the negative-going sweep [2].

Despite the overwhelming number of measurements, the number of relevant measurements is very small. To illustrate this point, figure 3.1 below shows 6 randomly sampled GV plots.

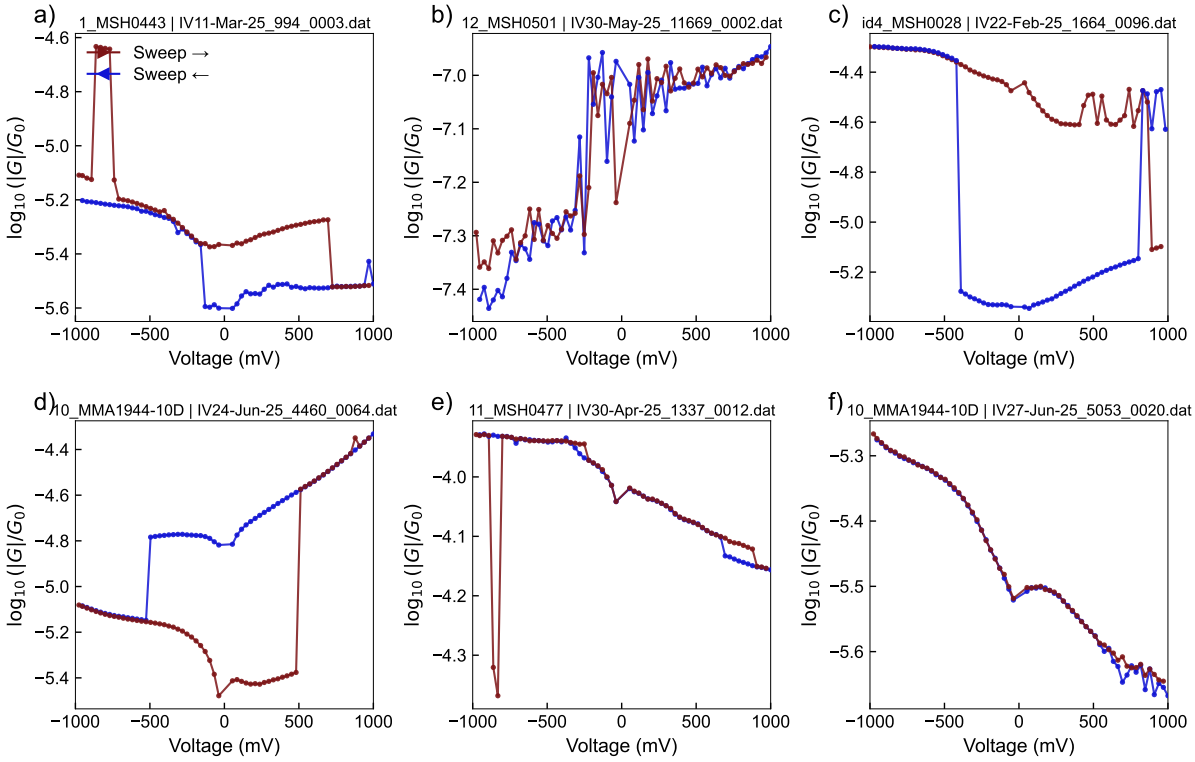


Figure 3.1: **Random sample of GV plots from the dataset.** Six GV plots randomly selected from the five molecule dataset used in this work. There is a clear variation in trace shape, and switching behaviour, which illustrates the heterogeneity of the dataset and thereby motivates the necessity for automated high-quality analysis methods.

3.2 Representation of an IV trace

Every IV trace measurement is an ordered sequence of measured voltage-current pairs

$$(V_t, I_t), \qquad t = 1, \ldots, T, \tag{3.1}$$

where T is the number of data points.

Conductance is computed using a `calculate_conductance()` function from the existing project framework [2]. This function partially removes a measurement artefact around zero bias that is present in the raw data. In order to fully remove the effect of the measurement artefact, a voltage bias cutoff at

$$|V| < 30\,\mathrm{mV} \tag{3.2}$$

is applied to all data. These points are excluded in all further analysis.

All the models used in this thesis treat logarithmic conductance as the main feature of the data, instead of simply current. Logarithmic conductance, $x_t = \log_{10}(|G_t|)$, is used as the primary feature of all the models because it compresses the large dynamic range of conductance values and thereby makes state conductance values easier to separate [2].

In the legacy clustering methods, 1D GMM, 2D GMM and agglomerative clustering, one of the following representations are used:

- one-dimensional clustering: $x_t = \log_{10}(|G_t|)$,
- two-dimensional clustering: (V_t, x_t).

The models in the classification pipeline employ richer feature representations that are introduced in Chapter 4.

3.3 Pointwise state assignment problem

The main task of this thesis is to assign individual points in an GV plot a discrete conductance-state. For each measurement index t, a label z_t is assigned. The pipeline is designed to classify datapoints of two-state systems, so $z_t \in \{0, 1\}$.

The objective is therefore to infer the full label sequence

$$z_{1:T} = (z_1, z_2, \ldots, z_T) \tag{3.3}$$

for every hysteretic IV trace.

3.4 Main challenges of the dataset

The process of building a model that performs simple pointwise state assignment is significantly complicated by several characteristics of the data.

Firstly, only a small fraction of the available IV traces exhibit clearly identifiable hysteretic or two state behaviour. In practice, based on qualitative observation, roughly 5–10% of all data shows clear hysteresis, with most traces containing little or no useful switching information. Because of this, any attempt to cluster IVs will have to be preceded by extensive filtering. In figure 3.2 below are three example GV plots that clearly do not exhibit any relevant switching information.

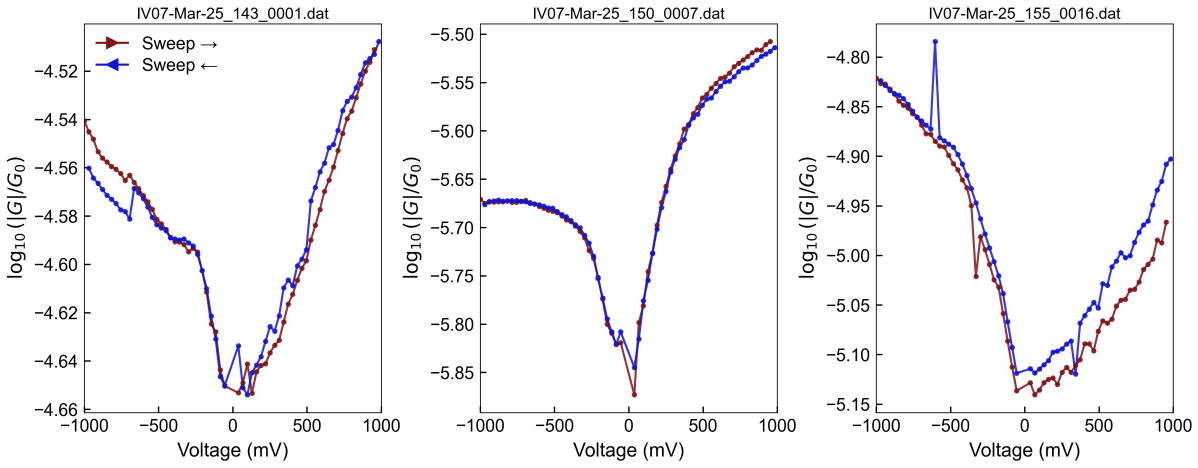


Figure 3.2: **Example GV traces that do not exhibit an useful switching structure.** A sample of three conductance plots that contain no relevant switching or conductance information. Traces like these correspond to parts of the breaking process in the MCBJ experiment where no relevant molecular switching occurs.

The second factor that makes effective pointwise classification difficult is that conductance

plateaus are often not flat. Instead, conductance can drift with voltage, which causes parts of one state to overlap in $\log_{10}|G|$ with parts of another. For this reason, one dimensional clustering methods fitted on logarithmic conductance, such as 1D Gaussian Mixture clustering, often fail in these instances [2]. In figure 3.3 below, there are three examples of GV in which it is clear that conductance plateaus can shift significantly with bias.

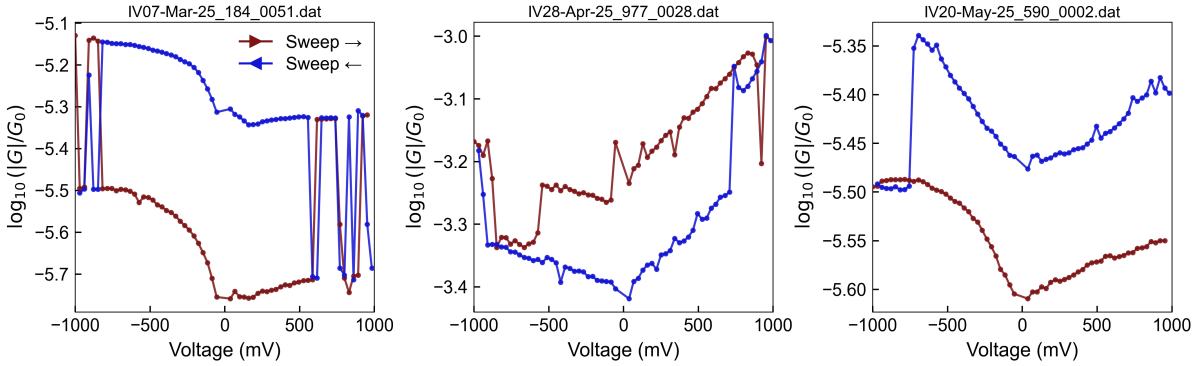


Figure 3.3: **Example GV plots with shifting conductance plateaus.** A selection of three GV plots that each exhibit hysteretic behaviour between conductance plateaus that are not constant, but shifted based on the voltage applied across the molecule.

Thirdly, many measurements seem to have a two state structure but exhibit a significant number of switches between these two states. All these frequent switches together give an idea of what an upper conductance-state may look like, but these frequent switches make it very difficult to, with the same model, incorporate correct state persistence as well as accurate switching in high frequency switching regimes, like those in the GV plots in figure.

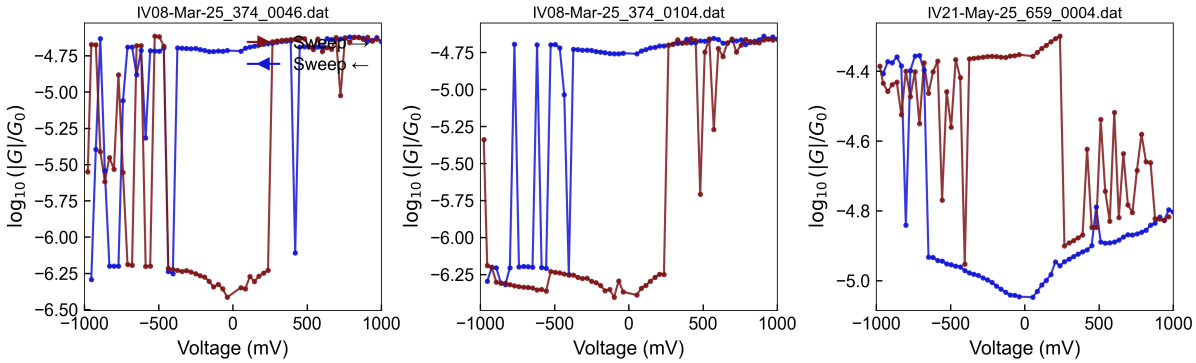


Figure 3.4: **Example GV plots with frequent state switches.** three GV plots that clearly exhibit a two state structure, but also has very frequent switching between these states, making temporal classification difficult.

3.5 Learning setting and objective

Generally, when training a neural network a dataset for this purpose is used that contains a ground truth from which the neural network can learn [11]. However, in this aim posed in this thesis, there does not exist a ground truth pointwise classification. The classification problem, therefore, cannot be treated as a standard supervised classification task.

Instead, in this thesis a multi-stage pipeline is implemented in order to prepare data on which a neural network *can* be trained on. To do this, firstly a filter must be implemented in order to remove all plots that certainly will not be able to contribute to the training dataset. This filter removes measurements such as those in figure 3.2. Next, a probabilistic teacher model generates psuedo-labels for the traces that remain. These psuedo-labels are subsequently used to train a student neural network for efficient large-scale pointwise classification.

Finally, in order to ensure that downstream physics analysis is performed on purely hysteretic traces, the neural network classifier is applied on the full dataset, and then a filter is built that enforces hysteretic behaviour. This results in a set of true hysteretic IVs, with high-quality pointwise conductance-state classifications on which further statistical analyses can be performed.

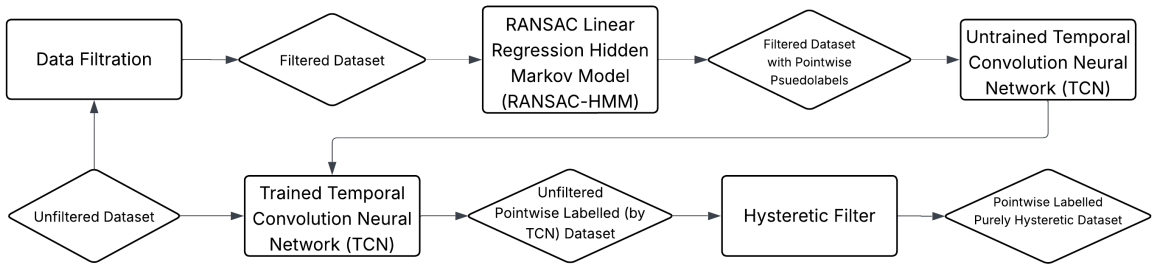


Figure 3.5: **Two-state classification pipeline flowchart.** A flowchart of the structure of the modelling pipeline used for pointwise classification of two-conductance-state molecules in a MCBJ.

Due to the lack of a ground-truth, the performance of the models is evaluated indirectly. The goal is not perfect pointwise accuracy in the standard supervised-learning sense, but rather to produce state assignments that:

- identify switching behaviour more reliably,
- retain a larger set of useful IV traces for downstream analysis, and
- improve the extraction of switching statistics such as conductance ratios and threshold voltages.

The classification pipeline is therefore not an end in itself, but a tool for improved downstream physics analysis.

Chapter 4

Methods

4.1 Method overview

The overall model pipeline and its motivation were introduced in section 3.5. Therefore, in this chapter, the individual stages of that pipeline are described in technical detail. The chosen methodology consists of three parts: Filtering to remove plots unlikely to contain relevant switching behaviour, the subsequent generation of pseudo-labels with a Hidden Markov Model, and then the training of a temporal convolutional network on the resulting labels. These stages are described in the following sections.

All methods in this chapter are developed and trained on only the three reference molecules, because these molecules define a two-state classification problem.

4.2 Data Filtering

Data filtration is the very first step in the process, with the goal of removing the vast majority of traces that are noisy, do not exhibit switching behaviour or do not exhibit any conductance plateau persistence. In order to filter for two state switching behaviour, traces are required to satisfy all criteria discussed below. All filtering operates on the log-conductance representation of the data, $x_t = \log_{10}(\max(|G_t|, \epsilon_G))$. The scope of this filter is not to perfectly remove all irrelevant data, but rather to filter until all data remaining is of sufficient high quality that it can be used in the teacher and student models.

4.2.1 Bimodality Test

Upon elementary analysis of the data, it is immediately evident that the majority of GV plots contain no relevant information due to their clear lack of switching structure. This is outlined in section 3.4, and in particular, figure 3.2. In these GV plots, both sweeps follow the exact same path in the log-conductance space. Therefore, the Bimodality Test determines whether the log-conductance values of a measurement are statistically better described by two conductance levels than by one single noisy plateau.

In order to determine whether a GV plot exhibits a 2 state structure, 2 gaussians are fitted

onto the sequence $x_{1:T} = [x_1, x_2, ..., x_T]$. The first fit is a one component Gaussian in the form of

$$p(x) = \mathcal{N}(\mu_1, \sigma_1^2) \tag{4.1}$$

The second fitted Gaussian is a two-component Gaussian mixture in the form of

$$p(x) = \pi_1 \mathcal{N}(\mu_{2,1}, \sigma_{2,1}^2) + \pi_2 \mathcal{N}(\mu_{2,2}, \sigma_{2,2}^2) \tag{4.2}$$

This two-component Gaussian mixture fit, fits two separate Gaussian distributions on sequence x.

In order to determine whether or not a given GV plot exhibits two-state behaviour, the results of the above two Gaussian fits must be compared. To compare two fits with a different number of parameters, the Bayesian Information Criterion (BIC) is used:

$$\text{BIC} = -2\log(L) + klog(T)$$

Where, L is the Likelihood of the model, k is the number of parameters used, and T is the number of data points. The likelihood of the one-dimensional Gaussian model is calculated by equation 4.3

$$L_1 = \prod_{t=1}^{T} \mathcal{N}(x_t|\mu_1, \sigma_1^2) \tag{4.3}$$

The likelihood of the two-component Gaussian Mixture is calculated by equation 4.4

$$L_2 = \prod_{t=1}^{T} [\pi_1 \mathcal{N}(\mu_{2,1}, \sigma_{2,1}^2) + \pi_2 \mathcal{N}(\mu_{2,2}, \sigma_{2,2}^2)] \tag{4.4}$$

As the fit improves, its likelihood increases. Since fits often improve with complexity, the BIC penalises the 2-component Gaussian fit with the k term, which is the number of parameters of a model. A lower BIC value means that something is a better model after penalising for complexity.

To compare the BIC values, the difference is calculated

$$\Delta\text{BIC} = \text{BIC}_1 - \text{BIC}_2 \tag{4.5}$$

By evaluating this ΔBIC for all GV plots, by visual inspection, it is determined that the plots for which $\Delta\text{BIC} < 10$ are almost all measurements lacking any switching behaviour to a possible second state, such as figures in 3.2. Therefore, this Bimodality test enforces $\Delta\text{BIC} \geq 10$, and any plots for which this is not the case, are filtered out.

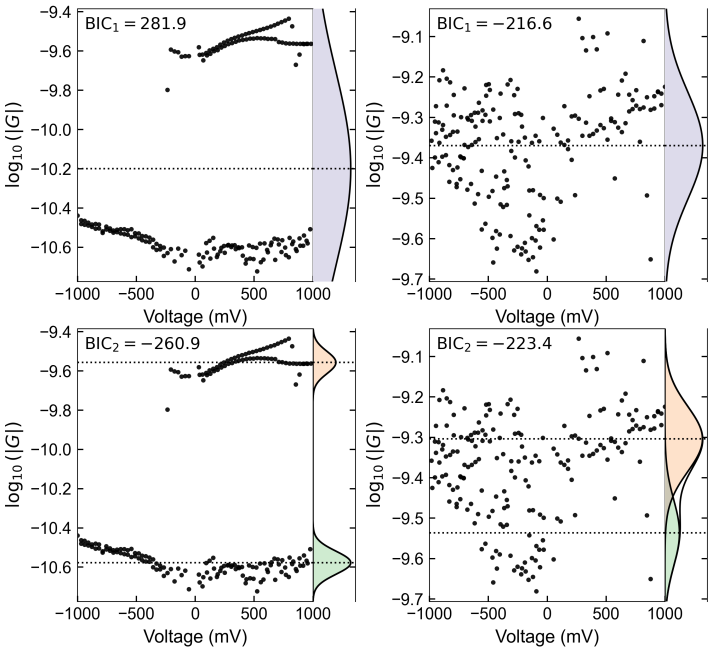


Figure 4.1: **Two examples of the bimodality Test.** Two example measurements (Left and Right) on which the Bimodality Test is applied, with its 1-component Gaussian fit (top) and its 2-component Gaussian fit (bottom). **Left:** The fit of the 1-component Gaussian is much worse, resulting in a very large BIC_1 value, causing the Bimodality test to pass. **Right:** The GV is very noisy, causing the 1-component Gaussian to outperform its counterpart enough cause failure of the Bimodality test.

Lastly, in order to ensure that the two-component Gaussian mixture fits are relevant, three additional requirements are added:

- Each Gaussian component must have $\pi_k \geq 0.15$, thereby preventing situations where one component dominates the fit. Without this, the GMM could incorrectly interpret noise as a second state.
- The two Gaussian means must have a minimum separation of

$$\frac{\max(\mu_{2,1}, \mu_{2,2})}{\min(\mu_{2,1}, \mu_{2,2})} > 1.3$$

  which ensures both Gaussians are not mapped atop each other.

If a GV plot performs well enough to pass $\Delta\text{BIC} \geq 10$, but its 2-component Gaussian fit does not pass one of the above 2 supplementary rules, it does not pass the Bimodality Filter. Out of all IV files belonging to reference molecules, this Bimodality filter removes **51%** of all IV files.

4.2.2 Plateau-Jump Filter

Whereas the bimodality filter tests whether two conductance-states are present, the plateau-jump filter tests for the existence of a plateau-like temporal structure, and at least one switching event.

The goal of this trace is to remove plots that:

- Have a significant amount of noise, which causes there to be no recognizable flat region, and therefore no clear conductance plateau

- Exhibit no switching events.

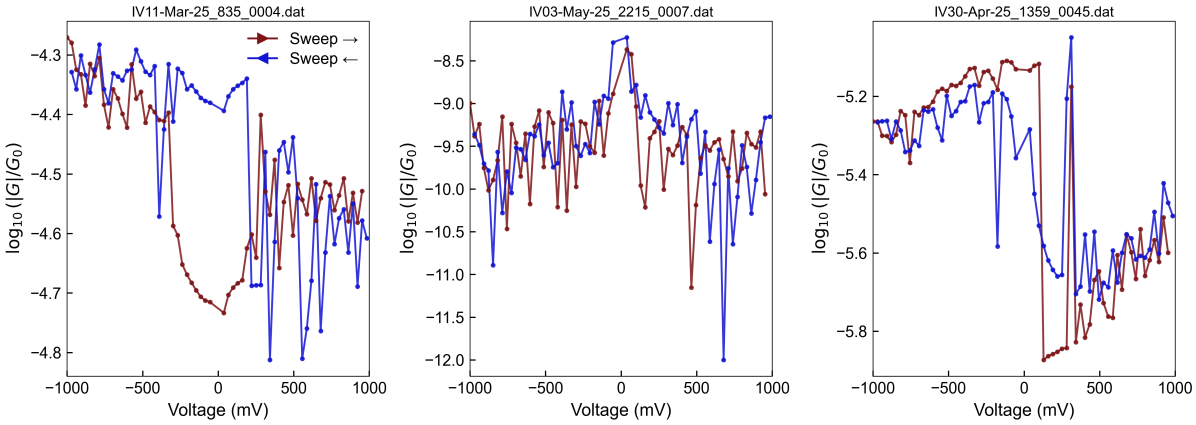


Figure 4.2: **Example GV plots that fail plateau-jump filter.** A selection of 3 GV plots that pass all other filters, but fail due to the plateau-jump filter.**Left:** This measurement fails the plateau-jump filter due to it failing the minimum jump condition. **Center and Right:** These measurements fail the plateau-jump filter due to the lack of a consistent plateau of data points.

In order to filter out high-noise GV data, the plateau-jump requires for at least 60% of the GV plot to be locally flat. When a part of a GV plot is considered locally flat, this segment must consist of at least 6 points, and there must be at least 2 of these plateaus present in the GV plot. In order for a segment of points in conductance-voltage space to be considered as locally flat, the local change between two points, $\Delta x_t = x_{t+1} - x_t$, is calculated, and when any two points satisfy the condition

$$|\Delta x_t| < 0.015 \tag{4.6}$$

then that step in the trace is considered flat. This value of flatness was chosen as to still allow most drifting conductance plateaus.

Upon calculating this, the plateau fraction criterion is applied:

$$p_{\text{flat}} = \frac{|\{t \in \{1, \ldots, T-1\} : |\Delta x_t| < 0.015\}|}{T-1}. \tag{4.7}$$

Equation 4.7 above calculates p_{flat}, the percentage of the GV that is flat. An IV trace is accepted under the plateau-fraction criterion when

$$p_{\text{flat}} \geq 0.6. \tag{4.8}$$

This removes all plots that high frequency oscillations/jumps, which are treated as unusable noise.

The plateau-jump filter also contains a jump-criterion that requires a measurement to exhibit at least one local jump of at least a conductance ratio of 1.6. This is enforced as a filter by computing the strongest local jump in the measurement and requiring this jump to be at least of a ratio of 1.6:

$$J = \max_t |\Delta x_t|. \tag{4.9}$$

The jump criterion then requires

$$J \geq \log_{10}(1.6). \tag{4.10}$$

By filtering out IV measurements that do not satisfy this criterion, GV plots such as those in figure 3.2 are further removed from the dataset.

To summarize, the plateau-jump criterion removes plots that

- Are too noisy
- Exhibit a plateau-like structure without exhibiting a real jump
- Have only one plateau-like region
- Have sufficient flatness, but this only occurs in scattered isolated points

The plateau jump filter passes only **34%** of all data, across the three reference molecules.

4.2.3 Window-Contrast Filter

The final filter ensures that the remaining measurements are sufficiently hysteretic by imposing a weak requirement on hysteretic structure. To do this, a requirement is imposed that states that a GV plot that passes this filter must, for small windows around zero voltage bias, exhibit both an upper and lower conductance level.

To this end, the window-contrast filter imposes that within both a negative window $-250\text{mV} \leq V \leq -150\text{mV}$ and a positive window $150\text{mV} \leq V \leq 250\text{mV}$, there must be at least two points within each of these windows that are separated by a conductance ratio of 1.5. The requirement then becomes that, there must exist at least two values of x such that:

$$\max(x_t) - \min(x_t) \geq \log_{10}(1.5) \tag{4.11}$$

for some t such that x_t is in a given window.

A trace measurement is only accepted by this filter if the contrast criterion holds in both the negative and the positive window, for at least two points.

The essence of this filter is that it requires somewhat meaningful conductance plateau separation. The structural effect of this filter is that it removes traces that are

- Almost flat or constant.
- clearly not hysteretic, or one-sided hysteretic.
- hysteretic but only contain small fluctuations.

.

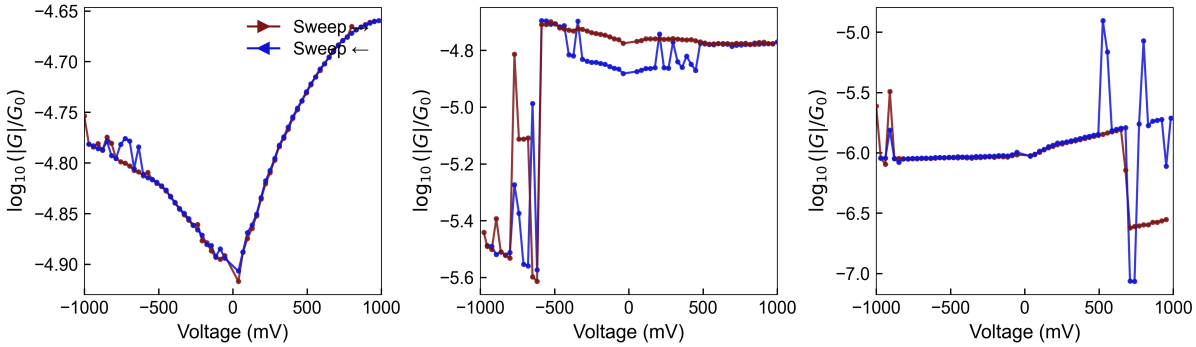


Figure 4.3: **Example GV plots that fail the window-contrast filter.** A selection of 3 GV plots that all fail on only the window-contrast filter. **Left:** This GV fails due to it being nearly constant. This has not failed the bimodality test because two Gaussians were fit sufficiently well on the top and bottom part of this trace. **Center and Left:** These measurements fail on the window-contrast filter because they only switch far from zero bias. These plots do not exhibit any hysteretic structure.

This is a weak way of enforcing hysteresis, which is acceptable as the scope of this data filter is not to filter out *all* traces that are not perfect hysteretics, but rather to ensure that the traces that remain exhibit a sufficiently convincing 2 state structure to train downstream models. This filter fails **67%** of all IV traces.

4.2.4 Filtering Statistics

After applying this comprehensive filter, all remaining GV plots exhibit each of the following:

- A somewhat hysteretic structure, across large parts of the bias domain.
- two conductance-state structure and switching.
- Sufficiently large switching conductance ratios.
- Near noiseless measurements without high-frequency switching regions

After applying this comprehensive filter, only 11.6% of all IV traces from the reference molecules remain. In the below figure 4.4, a comprehensive overview of the performance of each sub-filter is given.

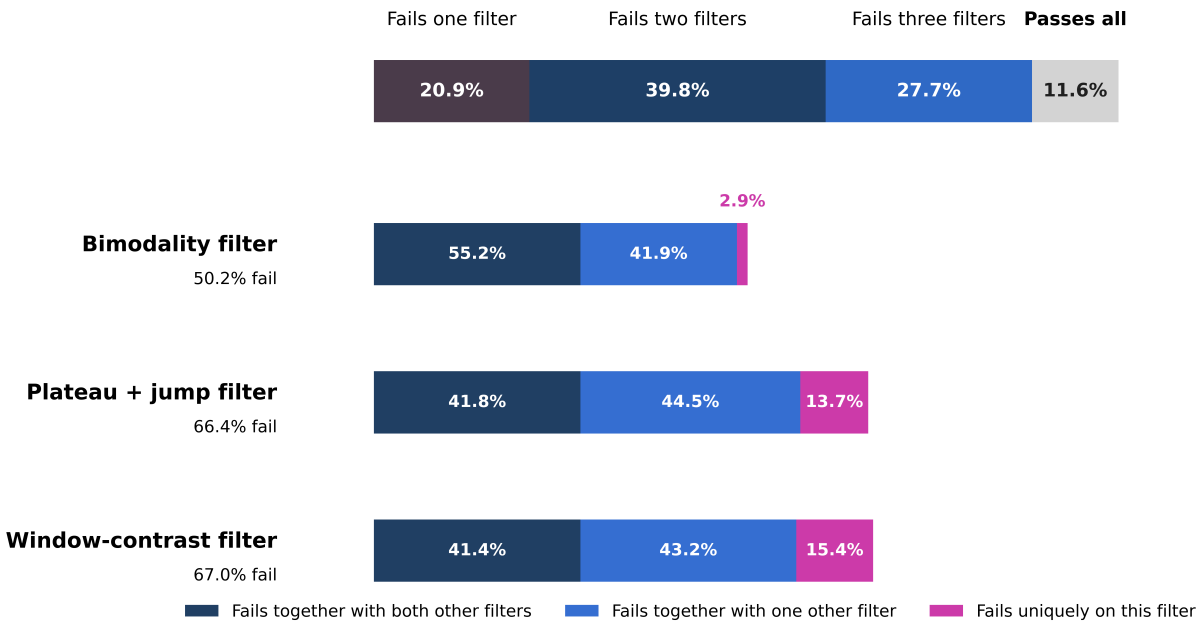


Figure 4.4: **Overview of filtering statistics.** A comprehensive overview of how each filter performs, their resulting pass/fail percentages and the extent to which certain filters overlap with each other regarding rejecting IV measurements.

4.3 Simple Linear Regression RANSAC Hidden Markov Model

Following the initial filtering stage outlined in section 4.2, the trace measurements that remain are given pseudo-labels by implementing a two-state Hidden Markov Model with state-dependent linear-Gaussian emissions. Since the general framework of Hidden Markov Models, state persistence and Viterbi decoding have been previously introduced in section 2.9, this section focuses merely on specific model implementation and the estimation procedure used in this work.

As is conform with the general approach to data preprocessing throughout this pipeline, the necessary zero-bias handling is implemented, after which all points with $|V| < 30\,\mathrm{mV}$ are removed from the analysed data. The resulting conductance sequence is therefore again transformed into the scalar sequence outlined in equation 2.2. This scalar sequence serves as the observation variable in this teacher model. Also, the corresponding bias voltage V_t is used as a regressor. The Hidden Markov Model therefore operates on the ordered sequence $\{(V_t, x_t)\}_{t=1}^{T}$, instead of on x_t alone. This is necessary due to the fact that the conductance plateaus are often not flat in bias, as is demonstrated in figure 3.3 and discussed in section 3.4.

Since the teacher HMM is designed for two-state traces, let the hidden state sequence be denoted by $z_t \in \{0, 1\}$, where these labels correspond to the low- and high-conductance-states after ordering. Conditional on the hidden state, the observation model of the HMM is taken to be a linear regression in voltage,

$$x_t \mid (z_t = k, V_t) \sim \mathcal{N}(a_k V_t + b_k, \sigma_k^2), \qquad k \in \{0, 1\}. \tag{4.12}$$

Where, a_k and b_k are considered to be the slope and intercept of the state plateaus in (V, x) space.

Here, σ_k^2 captures the residual variance around these plateaus. In contrast to standard Gaussian-emission HMM with state-dependent constants μ_k, this model allows the expected log-conductance levels to vary with an applied bias voltage inside each state, reflecting conductance-state shifting.

The transition matrix of the Hidden Markov Model is fixed, sticky and symmetric,

$$A = \begin{pmatrix} a & 1-a \\ 1-a & a \end{pmatrix}, \tag{4.13}$$

with $a = 0.995$. This a value is the probability of the next point in the sequence retaining the same state as its preceding point. This value is high, thereby naturally encoding a high level of state persistence consistent with the physical system. This also makes rapid state switching very unlikely, unless the emission likelihood strongly supports such frequent transitions. The initial state distributions are set by the initialization procedure described below.

4.3.1 Initialization

The parameter initialization of this Hidden Markov Model is performed in two stages. Firstly, a one-dimensional two-component Gaussian mixture model like described in equation 4.2, is fitted to the full sequence $x_{1:T}$. This produces an initial partition of the trace into two conductance levels. From this initial partition, a hard initial classification is obtained by maximum posterior assignment. These hard labels are then reordered such that the higher and lower conductance-states correspond to $z = 1$ and $z = 0$ respectively.

This Gaussian Mixture step is used only as a very rough initialization. Since this GMM ignores all voltage dependence and temporal structure in conductance plateaus, it is clearly not suitable as a final classification model. Its purpose is merely to provide an initial partition from which state-specific regression lines can be estimated.

In the following stage, two global regression lines are fitted onto the GV using an alternating two-line RANSAC procedure. Starting from the GMM classifications, two RANSAC lines are fitted, one to each provisional state. Each point in the GV is subsequently reassigned to the RANSAC line with the smaller absolute residual. Then, the two lines are refitted, and this process repeats itself. This alternating fit-reassign algorithm is repeated until, either the maximum number of iterations is reached, or the assignments and the lines stabilize. The resulting stable lines are fitted by

$$x = a_0 V + b_0, \qquad x = a_1 V + b_1, \tag{4.14}$$

This procedure produces both an initial geometric partition and initial values for the HMM emission parameters this produces both an initial geometric partition of the points and initial values for the SLR-HMM emission parameters.

By implementing RANSAC fitting in this stage, the resulting fits are less sensitive to isolated outlying points and therefore more accurately model the prevalent 2 state structure present. In contrast to ordinary least squares fitting, this process yields more stable initial plateau estimates.

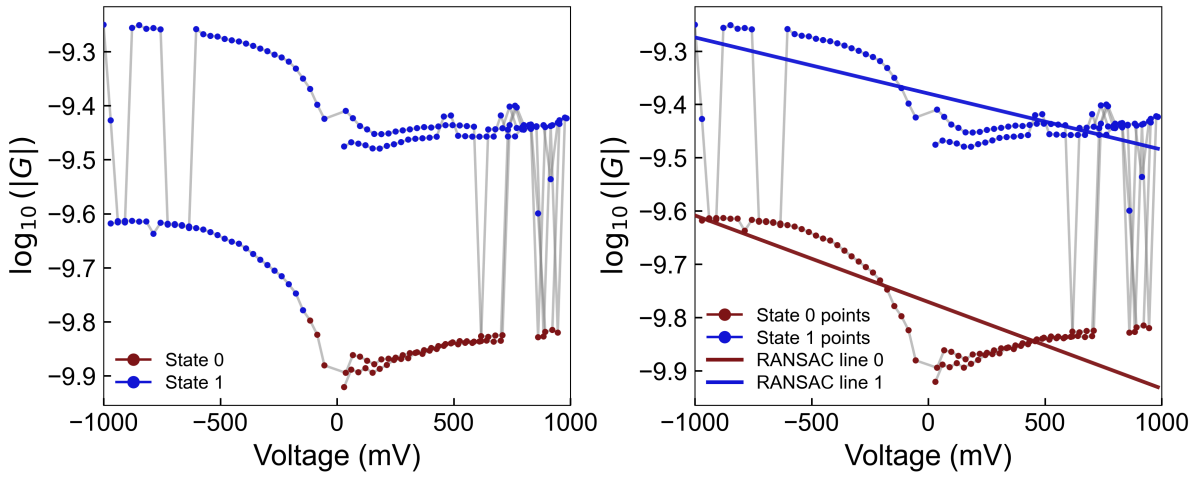


Figure 4.5: **GMM initialization compared to RANSAC procedure result.** Two classifications of the same GV. **Left:**Classification of a GV based in its GMM initialization **Right:** Classification of a GV based on the RANSAC procedure applied after the GMM initialization. The resulting RANSAC lines demonstrate an understanding of conductance level shifting.

4.3.2 SLR-HMM inference and optional EM refinement

The RANSAC procedure results in two initial lines, and based on these two lines, the state-dependent emission log-likelihoods are given by

$$\log p(x_t \mid z_t = k, V_t) = -\frac{1}{2}\left(\frac{(x_t - (a_k V_t + b_k))^2}{\sigma_k^2} + \log \sigma_k^2\right) + C, \tag{4.15}$$

where C is a constant that is state-independent and does not affect interference. For the fixed parameters, posterior state probabilities are computed by using the forward-backward algorithm, and the final hard labels are obtained with Viterbi decoding [10].

In the vast majority of traces, this process already yields high-quality labels that result from a useful combination of geometric separation and temporal smoothing. However, in rare cases, the RANSAC lines fitted to the data are poor estimations of conductance plateaus, causing the RANSAC lines to intersect in the voltage domain, which is physically infeasible. When this occurs, an expectation-maximization refinement process is enabled that updates the regression parameters.

Concretely, if the two RANSAC lines are written as

$$x = a_0 V + b_0, \qquad x = a_1 V + b_1, \tag{4.16}$$

then, if they intersect, they intersect at

$$V^* = \frac{b_1 - b_0}{a_0 - a_1}. \tag{4.17}$$

Therefore, EM refinement occurs when $V^* \in [-1100, 1100]$ mV. The process of this EM refinement works by using the posterior responsibility of a particular state to update the line parameters and residual variances. Concretely, the posterior responsibility of state k at index t is $\gamma_{t,k} = P(z_t =$

$k \mid x_{1:T}, V_{1:T}$), and therefore, the line parameters are updated by weighted linear regression given by

$$(a_k, b_k) = \arg\min_{a,b} \sum_{t=1}^{T} \gamma_{t,k} \big(x_t - (aV_t + b)\big)^2, \tag{4.18}$$

The residual variances are updated by

$$\sigma_k^2 = \max\left(\frac{\sum_{t=1}^{T} \gamma_{t,k} \big(x_t - (a_k V_t + b_k)\big)^2}{\sum_{t=1}^{T} \gamma_{t,k}}, \sigma_{\min}^2 \right), \tag{4.19}$$

with a small variance floor $\sigma_{\min}^2$ for numerical stability.

The reasoning behind this conditional implementation of EM refinement of the line parameters is practical. When RANSAC lines intersect, the geometric separation of the states present is ambiguous, and posterior-weighted refinement allows for a chance are improving the fit. When $V^* \notin [-1100, 1100]$ mV, the RANSAC lines are sufficiently separated and likely model conductance shifting correctly. In these cases, updating the line parameters is unnecessary and can cause the lines to drift away from the robust RANSAC fit.

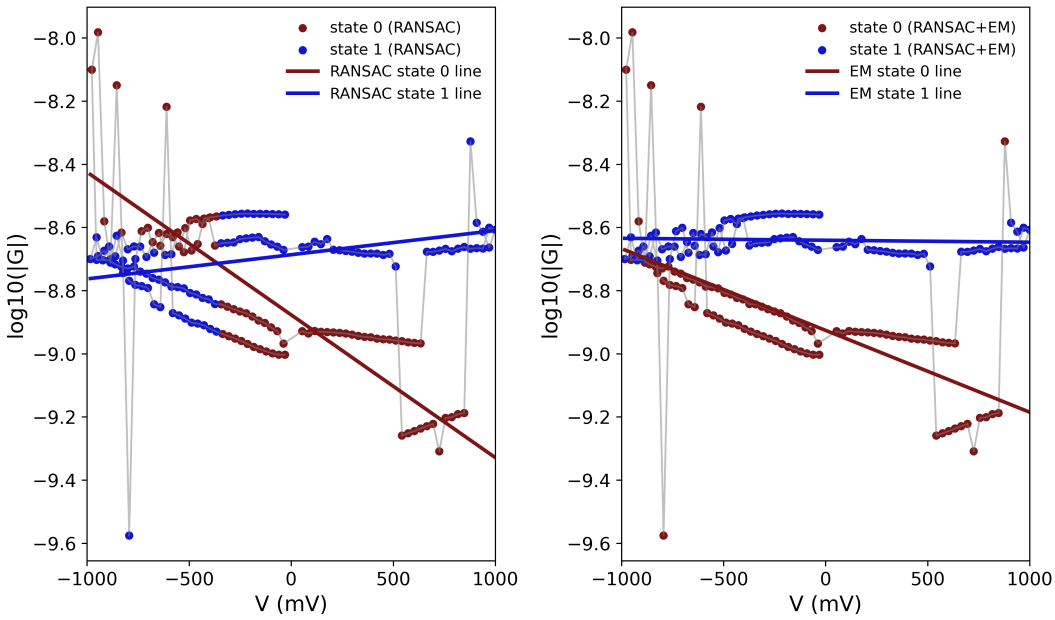


Figure 4.6: **Initial RANSAC lines v. EM updated lines.** An example of the EM process significantly improving the classification of a plot. **Left:** A poor initial RANSAC fit on an GV, resulting in poor classification. **Right:** The same GV, with lines that result from EM refinement, resulting in an improved classification.

Finally, the state classifications are ordered and reassigned to ensure that state 0 and 1 correspond to the lower and higher conductance-states.

4.3.3 Output pseudo-labels & computation time

The output of this SLR-HMM teacher model is a label sequence $\hat{z}_{1:T}$ for every IV that passed the filter in section 4.2. In addition to storing these label sequences, the model also stores the

GMM and RANSAC initializations, the final fitted line parameters, the soft labels (certainties per classification), as well as the processed voltage and conductance arrays. These outputs are saved separately for every trace, and subsequently used as pseudo-labels to be used for training the temporal convolutional network to be introduced next.

This Hidden Markov Teacher serves a crucial role. It produces a classification that combines the voltage-dependent state geometry with sequential state persistence to produce a high quality classification. However, the initialization of this combination of Gaussian Mixture Modelling, iterative RANSAC fitting, occasional EM refinement and HMM calculations provides combines to form a multi-step model that can be computationally intense.

The computation time of this model scales approximately linearly with the number of measurements in an IV, T. However, there is a large multiplicative constant due to frequent repeated optimization procedures. The GMM initialization requires multiple EM iterations and several random restarts, thereby contributing a cost $O(N_{\text{init}} I_{\text{GMM}} T)$, where $N_{\text{init}} = 50$ is the number of random initializations used when fitting the GMM and $I_{\text{GMM}} \leq 5000$ is the number EM iterations per GMM initialization. The subsequent alternating RANSAC line fitting results in roughly $O(I_R J_R T)$ operations, where $I_R \leq 25$ is the number of alternating assignment iterations in the RANSAC procedure. J_R is the number of internal RANSAC trials used, which typically lies in the order of tens to hundreds of trials. The HMM inference itself scales with $O(Tk^2)$ which reduces to $O(T)$ for $k = 2$. In some cases, additional EM refinement is used, resulting in a $O(I_H T)$ contribution to computation time, where $I_H \leq 40$ is the number of EM iterations used to refine the SLR-HMM parameters. Combined, the total computation time per IV trace can be approximated as

$$O(T\,[1 + N_{\text{init}} I_{\text{GMM}} + I_R J_R + I_H]) \tag{4.20}$$

This means that although the algorithm is linear in length, the repeated iterative fitting procedure makes the constant factor large. Consequently, robustly applying this pipeline to large collections of IV traces becomes computationally expensive. This, along with the fine-tuning necessary in this model, motivates the necessity to use this model as a teacher model in order to use the pseudo-labels to train a neural network that, after being trained, allows for light computation to result in high-quality classification.

4.4 Temporal convolutional network student model

The student temporal convolutional network model is trained on the pseudo-labels generated by the Hidden Markov Model, and its aim is not to recover a ground-truth classifier but rather to reproduce or improve on the teacher classifications efficiently.

4.4.1 Training objective and role of the student model

The teacher model produces sequences of hard state pseudo-labels and their confidences. The student model is introduced as an approximation map

$$f_\theta : X_{1:T} \mapsto \hat{p}_{1:T}, \tag{4.21}$$

where $X_{1:T}$ denotes a sequence of feature vectors derived from a single IV trace, θ the trainable parameters, and $\hat{p}_t \in [0, 1]$ the predicted probability that point t belongs to state 1. The hard classification resulting from the TCN is obtained through

$$\hat{z}_t = \mathbb{1}[\hat{p}_t \geq 0.5]\,. \tag{4.22}$$

This TCN is therefore trained as a pointwise sequence labeller, not a trace-level classifier.

One interpretation of this student model is a distillation model trained from a probabilistic teacher. It learns based on inferred state geometry and the temporal persistence structure imposed by the teacher. The TCN can, once trained, assign labels in a single forward pass.

4.4.2 Training data and pseudo-label targets

For each valid point t, the training target of the TCN is the soft teacher probability

$$\tilde{p}_t = p_t^{(\text{teacher})}, \tag{4.23}$$

with $\tilde{p}_t \in [0, 1]$. The TCN is thus not trained on the hard pseudo-labels, but rather their certainties.

Plateau regions are typically assigned soft labels close to 0 or 1, whereas ambiguous points can take intermediate values. The student model approximates thus

$$\hat{p}_t \approx \mathbb{P}_{\text{teacher}}(z_t = 1 \mid \text{trace}). \tag{4.24}$$

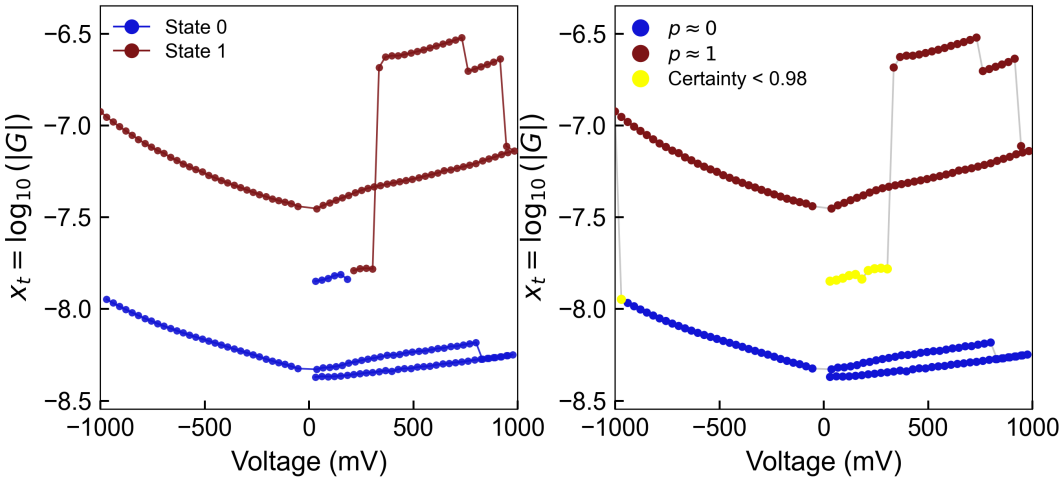


Figure 4.7: **Example of soft labelling of a GV. Left:** GV plot with classified by hard pseudo-labels **Right:** The same GV plot, but with all data point with a certainty of less than 0.98 plotted in yellow. This shows clearly that the ambiguous points between states are considered less certain than those clearly belonging to a particular conductance-state.

4.4.3 Input representation and feature construction

The neural network is not trained on raw current values, but is instead trained on a five-dimensional feature vector that combines conductance information with sweep structure information. After

conductance computation, the input sequence is given by:

$$X_{1:T} = [u_1, \ldots, u_T], \qquad u_t \in \mathbb{R}^5. \tag{4.25}$$

The five features of the TCN are:

1. normalized voltage

$$u_{t,1} = v_t^{\text{norm}} = \frac{V_t^{(\text{mV})}}{V_{\text{scale}}}, \tag{4.26}$$

   with $V_{\text{scale}} = 1000$ mV;
2. standardized log-conductance

$$x_t = \log_{10}(\max(|G_t|, \epsilon_G)), \qquad u_{t,2} = x_t^{(z)} = \frac{x_t - \mu_x}{\sigma_x}; \tag{4.27}$$

3. gradient of the standardized conductance feature

$$u_{t,3} = \Delta x_t^{(z)}; \tag{4.28}$$

4. sweep-direction indicator

$$u_{t,4} \in \{-1, +1\}; \tag{4.29}$$

5. full-sweep indicator

$$u_{t,5} \in \{0, 1\}. \tag{4.30}$$

Therefore, the final feature vector is

$$u_t = \begin{bmatrix} v_t^{\text{norm}} \\ x_t^{(z)} \\ \Delta x_t^{(z)} \\ d_t \\ s_t \end{bmatrix} \in \mathbb{R}^5. \tag{4.31}$$

This feature design allows the network access to directional and sweep-context information, as well as local conductance behaviour. The sweep-context information is directly relevant for hysteretic classification, as the trained TCN will be run on true hysteretic GV curves, and in these cases, sweep direction is a good indicator for a points conductance-state. Additionally, the gradient of the standardized conductance feature is a strong feature due to the fact that this gradient value is a good indicator of a conductance-state switch.

4.4.4 Variable-length batching and masked training

After preprocessing, different IV traces may contain different number of valid points, thereby causing not all sequences to have the same length. During training, the neural network processes several traces together, as part of the same mini-batch, which requires all traces in each mini-batch to contain the same sequence length. The implementation of the TCN therefore pads each trace with dummy entries until it reaches the length of the longest trace in the mini-batch.

If a mini-batch contains B traces, and if trace b contains T_b points, then the longest trace has

length

$$T_{\max} = \max_{1 \le b \le B} T_b. \tag{4.32}$$

All traces in a given mini-batch are then stored as if they had length $T_{\max}$.

The batched data is represented by four tensors:

$$X \in \mathbb{R}^{B \times T_{\max} \times 5}, \tag{4.33}$$

$$P \in \mathbb{R}^{B \times T_{\max}}, \tag{4.34}$$

$$W \in \mathbb{R}^{B \times T_{\max}}, \tag{4.35}$$

$$M \in \{0,1\}^{B \times T_{\max}}. \tag{4.36}$$

Where X contains the five input features for each point, P contains the soft teacher targets, W contains the confidence weights of the targets and M is a binary mask.The mask differentiates real data points from padded dummy entries:

$$M_{b,t} = \begin{cases} 1, & \text{if the point is real,} \\ 0, & \text{if the point is padding.} \end{cases} \tag{4.37}$$

The mask ensures that padded entries do not contribute to loss. As a result, all traces of different lengths can be processed in one mini-batch without padding affecting training.

4.4.5 Network architecture

The student model is a one-dimensional TCN that assigned pointwise probabilities in each IV trace. The input feature sequence is

$$X \in \mathbb{R}^{T \times 5}, \tag{4.38}$$

The network transforms this input sequence into the sequence

$$\hat{p}_{1:T}, \tag{4.39}$$

where $\hat{p}_t$ is the predicted probability that point t belongs to state 1.

The first operation in the network is a 1×1 convolution. When implemented, this does not combine data points that neighbor each other, but mixes the five input features at each point into a higher-dimensional representation. This causes each point to be mapped from a five dimensional feature vector to a 128 dimensional feature vector:

$$H^{(0)} = \mathrm{Conv}_{1\times 1}(X), \qquad H^{(0)} \in \mathbb{R}^{T \times 128}. \tag{4.40}$$

After this first layer, every point in the sequence retains its trace-position, however, each point is now described by a rich internal representation.

The main computation of the model consists of seven successive residual convolution blocks where each block takes a sequence of hidden feature vectors and updates it. The purpose of these blocks is letting the network combine information from points around the currently targetted point, so that the classification of the target point obtains a dependence on its surrounding

context.

In each convolution block, the TCN applies two one-dimensional convolutions with kernel size 5. This implies that the convolution combines information from a local window around the target point of five positions. These five positions that form this context are not all neighbouring, but instead the context window is dilated with a dilation factor d. This means that the convolution uses a sample of points with spacing d around the target point to form the context. For example:

- dilation $d = 1$ corresponds to an ordinary local convolution,
- dilation $d = 2$ skips every second point,
- dilation $d = 4$ looks over a wider range still.

Larger dilations allow for the network to incorporate information from a larger context window, without requiring a deep network.

The seven residual blocks used in this TCN use dilation factors

$$d_l = 2^{l-1}, \qquad l = 1, \ldots, 7, \tag{4.41}$$

which make the dilation factors

$$1,\ 2,\ 4,\ 8,\ 16,\ 32,\ 64. \tag{4.42}$$

Each residual block has the same structure. Starting from its input $H^{(l-1)}$, each residual block applies a dilated convolution, a normalization, a ReLu nonlinearity and then a dropout. This process is repeated twice. The transformed output is then added to the original input:

$$H^{(l)} = H^{(l-1)} + \text{transformed version of } H^{(l-1)}. \tag{4.43}$$

This is called residual connection. The purpose of it is to simplify the optimization of each block, and make it more stable. Each block now only learns from a correction of its input, rather than having to output an entirely new representation. The network therefore gradually refines its outputs in this process, instead of overwriting all previous information in every repetition.

After all residual blocks are computed, the output representation still has the same shape, $T \times 128$. Therefore, a final 1×1 convolution maps this representation back to a single scalar per point:

$$s_{1:T} = \text{Conv}_{1\times 1}(H^{(7)}). \tag{4.44}$$

Where these s_t values are called logits. By applying the sigmoid function to these logits, the final state probabilities are obtained:

$$\hat{p}_t = \sigma(s_t). \tag{4.45}$$

The full network therefore maps a sequence of five dimension feature vectors to a sequence of probabilities.

4.4.6 Loss function and confidence-weighted learning

In order to train this neural network, the loss function must incorporate the mask, and be a confidence-weighted binary cross-entropy loss computed on logits, s_t. Let $\hat{p}_t = \sigma(s_t)$ be the predicted probability, $\tilde{p}_t$ be the teacher target, w_t be the confidence weight and $m_t \in \{0, 1\}$ be

Table 4.1: Hyperparameters of the final TCN student model.

Hyperparameter	**Value**
Number of input features	5
Input projection width	128
Number of residual blocks	7
Dilations	$1, 2, 4, 8, 16, 32, 64$
Kernel size	5
Dropout rate	0.1
Epochs	40
Batch size	32
Learning rate	2×10^{-3}
Optimizer	AdamW
Weight decay	10^{-4}
Gradient clipping norm	1.0
Voltage scaling constant V_{scale}	1000 mV
Zero-voltage cutoff	30 mV
Conductance floor ϵ_G	10^{-20}
Test fraction	0.20
Validation fraction of remaining data	0.15

the padding mask. The pointwise loss is then

$$\ell_t = -\tilde{p}_t \log \hat{p}_t - (1 - \tilde{p}_t) \log(1 - \hat{p}_t), \tag{4.46}$$

and the batch loss is

$$\mathcal{L} = \frac{\sum_{b=1}^{B} \sum_{t=1}^{T_{\max}} m_{b,t} w_{b,t} \, \ell_{b,t}}{\max\left(\sum_{b=1}^{B} \sum_{t=1}^{T_{\max}} m_{b,t} w_{b,t},\, 1\right)}. \tag{4.47}$$

This formulation is crucial due to the fact that the student is not trained from a ground truth. In this loss function, high-certainty pseudo-labels contribute much more strongly to the optimization, whereas low-certainty pseudo-labels only play a weak supervisory role. In this TCN, the confidence-based learning is done by using the teacher provided certainty value of each point directly as the weight w_t. This means that points for which the teacher is highly certain contribute more to the loss, but points of low certainty are down weighted.

4.4.7 Optimization procedure and model selection

The model is trained for 40 epochs with a batch size 32, by using the common gradient optimizer, AdamW [13]. The hidden width of the TCN is 128 and the dropout rate is 0.1. The hyperparameters are summarized in table 4.1 below

The selection of the final model after is based on validation loss in that, whenever the validation loss improves those particular weights are saved as the current best model.

4.4.8 Train, validation, and test split

The TCN input files are split randomly in order to form a training set, a validation set and a test set split. 20% of the files are assigned to the test set used to evaluate the performance of the

TCN. The remaining files are split further, with 15% belonging to the validation set, and the remainder being used for training.

Due to the fact that the input files of this TCN span all three reference molecules, the final trained TCN does not learn molecule-specific behavior, but rather trains on global two-state behavior across all molecules.

4.4.9 Finalized Labels & Computation Time

After this training process, the best model is run on all available IV traces, including the IV traces of the target molecules, to produce final TCN based labels. For each raw IV data file, the trained TCN is run to compute

$$\hat{p}_t = \sigma(s_t), \qquad \hat{z}_t = \mathbb{1}[\hat{p}_t \geq 0.5]\,. \tag{4.48}$$

for each point in each IV. The output of running the trained TCN is both the predicted soft labels as well as their hard labels.

Applying the trained TCN to IV traces is computationally inexpensive. For an IV trace with T points, this computation cost per IV trace is simply $O(T)$, because all heavy computation is done in the training phase. This allows large collections of traces to be labelled rapidly without expensive optimization steps required by the teacher model.

4.5 Hysteretic filter

4.5.1 Purpose of the filter

After the teacher model has produced pointwise state labels for all IVs of the three reference molecules, a final hysteretic filter must be applied to these measurements to enforce that all downstream physics statistical analysis is based on strictly hysteretic data.

The reason that the filter discussed in section 4.2 is not used is because that filter does not enforce hysteresis strongly. The purpose of that filter was to remove IV data that was of such low quality that it would not be able to be used to train the neural network. Therefore, that filter is less strict than this hysteretic filter should be, as this hysteretic filter strongly enforces hysteresis.

The design of this filter is fully based on the memristivity-criteria used in the thesis by van der Geest [2]. These criteria are only slightly adapted for improved performance.

4.5.2 Rule-based hysteretic criterion

This filter works by accepting an IV trace if it satisfies four rules that are intended to enforce basic characteristics expected of memristive switching: Opposite low-bias states on the two full sweeps, switching at both ends of the voltage domain and sufficient separation between conductance-states.

Rule 1: low-bias state separation. The first rule requires the two full sweeps in each IV measurement to exhibit points in both conductance-states in the low-bias region $|V| \leq 100\,\mathrm{mV}$. This rule is only passed if, in this region, each of the two full sweeps largely occupy different

states. If, in the low-bias region, the full sweeps both occupy the same state, the GV plot is clearly not hysteretic, and therefore fails this rule.

Rule 2: high-bias switching at both polarities. The second rule requires that at least one of the two full sweeps exhibits a true switching event at both positive and negative bias. In this rule, a switch is only counted if a state change happens beyond a set high bias threshold. Also, in order to improve the implementation of this rule compared to its implementation by van der Geest, a true switch only occurs when a switch occurs that is not directly followed by a switch back. By doing this, one point switches are not counted, as many plots have noise that manifests itself as one point switches.

Rule 3: switching in both sweep directions. The third rule is directly mirrors rule 5 of the memristive criterion outlined in the thesis by van der Geest. This rule requires at least one which to be present in the sweep to the left, and one switch to be present in the sweep to the right. Switches that occur close to $V = 0$ are ignored, as this region is often unstable. This rule ensures that all accepted IV traces exhibit bidirectional switching.

Rule 4: element-wise conductance-ratio criterion. The final rule of the hysteretic filter enforces an element-wise conductance ratio. In this rule, the state-separated conductance ratio is calculated in multiple voltage bins. In other words, the voltage spectrum is split into 15 bins and for each bin, the average conductance ratio is calculated between the upper and lower conductance-states present in that bin. This method of calculating conductance ratio performs better than a global average per state, as it somewhat corrects for shifting conductance plateaus. An IV passes this rule when its average conductance ratio across the bins is greater than 1.5. This is the same value used by van der Geest to enforce conductance-state separation.

4.5.3 Final decision and relation to earlier work

An IV trace is only passed as hysteretic after passing all four rules defined above. The core difference between this hysteretic filter and the hysteretic filter from previous work is that this hysteretic filter is run on data labelled by the trained TCN, instead of the elementary clustering methods used in the previous work [2]. An improved classification improves the performance of this hysteretic filter significantly, because poorly classified points result in state switches being detected at incorrect places, and improve the accuracy of calculating conductance ratios that are based on this classification.

4.6 Evaluation metrics

In this section, three methods of evaluation used in this work are introduced; The first is a trigger-based evaluation that relates to the original measurement protocol followed when producing the dataset. The second is a set of metrics that can indicate whether or not a hysteretic is physically plausible, which is inferred from the idea of a consistent conductance plateau separation. The last is a simple manual audit evaluation, that consists of manually labelling GVs as hysteretic, and evaluating performance by comparison.

4.6.1 Trigger-based evaluation.

In the original MCBJ experiment that resulted in the datasets used in this work, the decision whether or not to re-measure the same molecule with the same experimental setup was not done arbitrarily. In the original measurement workflow, follow-up measurements with the same experimental setup were only performed when the first measurement was identified to be a hysteretic measurement. This is because the vast majority of rotor positions in the MCBJ only give data that clearly has no hysteretic structure.

As a result of this measurement methodology, if a first measurement at a new rotor position is misclassified to be non-hysteretic, then the rotor position is changed and the next measurement is made. Therefore, a way in which the performance of the hysteretic filter outlined in section 4.5 can be evaluated is by calculating how many times the measurement setup was changed despite the fact that the first measurement in that measurement set is in fact hysteretic (but skipped due to poor classification). This performance metric of the hysteretic filter is important, as it directly related to how many possible extra measurements could have been made if state classification, and therefore hysteretic classification, is more accurate.

4.6.2 Physical plausibility.

In order to evaluate the quality and accuracy of the hysteretic filter at the end of the pipeline, it is important to ascertain whether or not these plots labelled as hysteretic are physically plausible.

Physical plausibility was evaluated based on the idea that, when a plot is a clear two-state hysteretic, then the ratio between the two state plateaus at different voltage biases should remain relatively similar. To quantify this, and implement it in a methodology for testing physical plausibility, each hysteretic IV is split into many voltage bins. In other words, the voltage range $[-1000\text{mV}, 1000\text{mV}]$ is split into many different intervals/bins, that together make up the full interval. Then, per bin, the logarithmic conductance ratio (referred to here as the log-separation) between the high conductance-state and low conductance-state points within that bin is calculated. In order to do that, the median of each is calculated, and then these conductances are divided by each other to provide a conductance ratio. In order to ensure that this calculation is not being done for bins that only contain one conductance state, it is required, before calculation, that the bin must have at least two data points per state. By making this calculation for all IVs considered hysteretic, a distribution of bin-wise conductance ratios per GV is obtained. From this distribution and bin-wise separations of the voltage range, three summary metrics were computed:

The first is **coverage**, which is defined as the fraction of bins for which it was possible to calculate the conductance ratios. A higher coverage implies that a valid two-state structure persists across a large part of the voltage range.

The second is **standard deviation of the bin-wise log-separation**. This is computed from the distributions per GV of the bin-wise log-separations. Lower values indicate a more uniform state separation across the GV.

The last is **interquartile range (IQR) of the bin-wise log-separation.** This is very similar to the standard deviation of bin-wise log-separation, but provides a more robust measure of spread. In this metric, lower values also characterise a more consistent conductance separation between conductance states.

In tandem, these three metrics help evaluate whether or not the GVs classified as hysteretic by the hysteretic filter are actually physically plausible.

4.6.3 Manual audit.

Due to the absence of ground truth, it is impossible to say whether or not the hysteretic filter is genuinely producing pure hysteretic GVs, without manual auditing. For this reason, a manual audit is done by labelling a random set of 200 GVs from the all molecules by hand as hysteretic or not, and testing the extent to which the hysteretic filter, based on the TCN pointwise classifications, labels the random sample of GVs the same way as the manual labels given.

Chapter 5

Results

5.1 SLR-HMM teacher model results

The first output of the pipeline is the pseudo-labels created by the teacher SLR-HMM model. Since target molecules were not used in the development of the SLR-HMM model, their data does not have pseudo-labels. For this reason, all results presented in this subsection consider only the data from the reference molecules. The SLR-HMM model produced pointwise labels by combining voltage bias state geometry with temporal persistence by implementing the Hidden Markov Model. Generating a final labelling of data points is not the main purpose of this state. The main purpose is to generate pseudo-labels of sufficient quality to serve as supervision for the TCN student model. Since there does not exist a ground truth classification, the quality of these results is evaluated by their physical plausibility and temporal consistency.

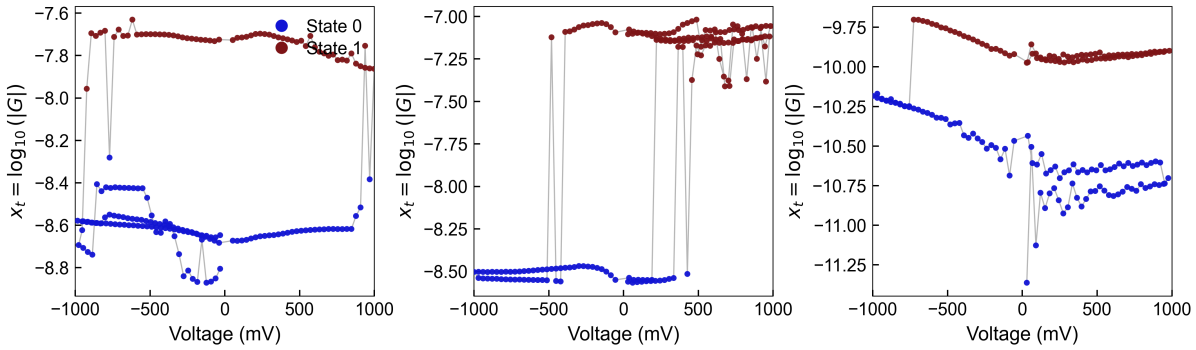


Figure 5.1: **Example pseudo-labelled reference molecule GV plots.** A random selection of 3 plots with their HMM pseudo-labels. These GV measurements are all from the filtered dataset.

To place the pseudo-labels generated by the SLR-HMM in context, figure 5.2 compares them to legacy classification methods used in earlier work [2].

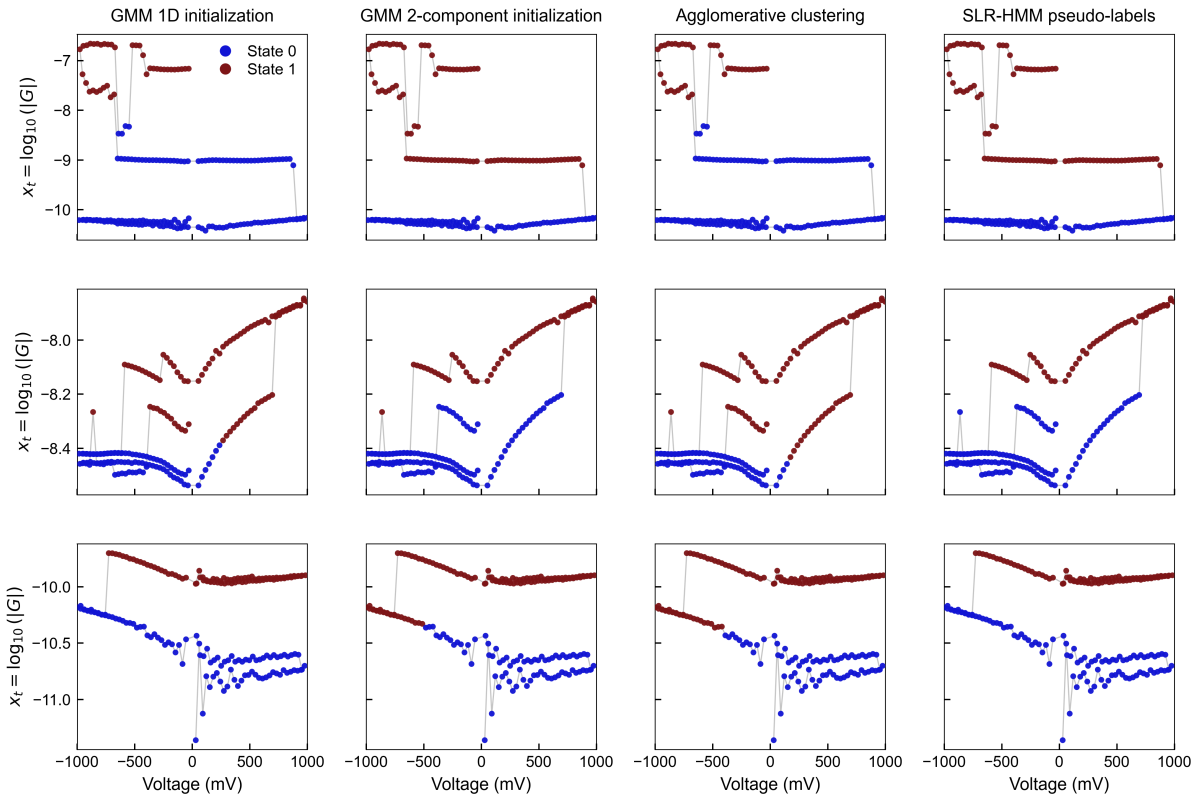


Figure 5.2: **Full comparison between legacy classification and SLR-HMM classification.** A full comparison of the legacy classification methods used in previous work, with the SLR-HMM pseudo-labels produced. Each row is a measurement from a different molecule.

In the figure above, the top row shows a case where the first half sweep of a measurement is misclassified as a state on its own by two of the three legacy methods, because its conductance is different to the conductances of the full sweeps and the half sweep below it. In the rest of the trace, a clear two state structure is evident. Relative to the legacy methods, the SLR-HMM pseudo-labels follow this two state structure more closely.

In the middle and bottom row of figure 5.2, we see examples of legacy classification techniques misclassifying the opposite ends of the lower conductance-state due to their drift. Again, two of the three methods do this wrong, whereas the SLR-HMM pseudo-labels perform well.

Table 5.1: **High-Disagreement GV classifications rates.** A table showing the percentages of GVs that disagree at least 10% pointwise with the SLR-HMM pseudo-labels per legacy classification method.

Legacy clustering method	**Percentage of GVs with $\geq 10\%$ disagreement**
GMM 1D initialization	11.54%
GMM 2-component initialization	10.84%
Agglomerative clustering	10.11%

In table 5.1, we see that out of all the GVs for which there exist pseudo-labels, 10% of these contain a high rate of pointwise classification disagreement between the legacy classification methods and the pseudo-labels produced by the Hidden Markov Model [2].

Figure 5.3 shows the impact the HMM stage has on pointwise labels. In the example, the initial RANSAC classification contains some inconsistencies but the final pseudo-labels are more temporally consistent.

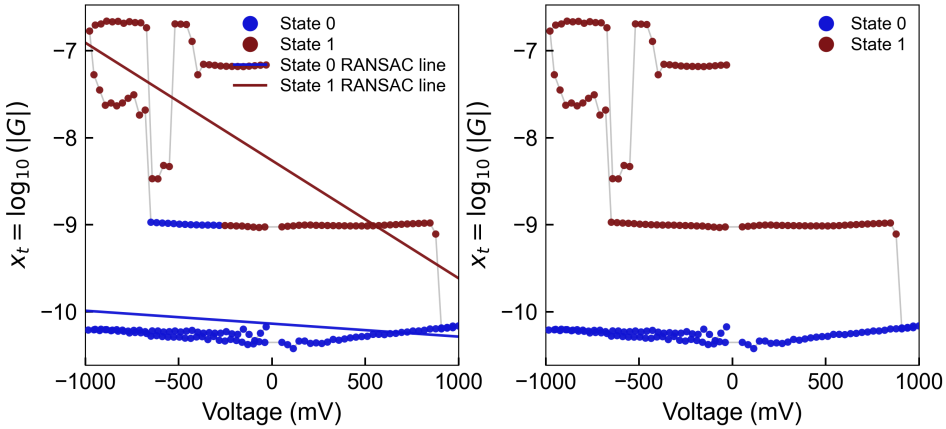


Figure 5.3: **Example of HMM temporal coherence.** An example of how the Hidden Markov Model encodes state-persistence and therefore can significantly improve labels after a RANSAC Fit. **Left:** RANSAC fitted on a GV, resulting in a poor initial classification. **Right:** HMM pseudo-labels of the same GV, showing that the upper conductance-state is classified correctly.

From the figures above, the SLR-HMM stage seems to provide high-accuracy teacher pseudo-labels. The pseudo-labels produced contain more temporal persistence and therefore align better with the natural temporal persistence present in MCBJ GV traces.

5.2 TCN student model results

In this section, the performance of the TCN student model is evaluated. The purpose of this evaluation is not to compare the student model against an external ground truth, as it does not exist, but rather to determine the extent to which the student successfully reproduces the pseudo-label behaviour of the teacher model.

5.2.1 Training behaviour

The first aspect to evaluate is the training behaviour of the student model.

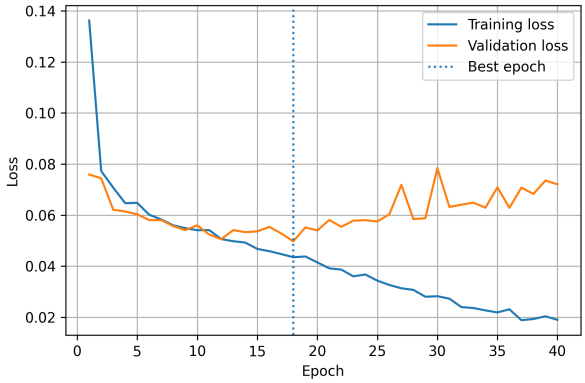


Figure 5.4: **TCN validation and training loss curves.**A figure showing the loss curves of the student model across the 40 epochs it was trained, showing significant divergence between the training and the validation loss.

Figure 5.4, the training and validation loss of the temporal convolutional network are depicted as a function of epoch. In the initial stage of training, both losses decrease rapidly, indicating that the network learns the main structure of the pseudo-labels quickly. The validation loss reaches a minimum at epoch 18, whereafter it diverges significantly from the training loss. This behaviour is indicative of over-fitting: after the initial convergence, the network fits training-specific detail more strongly than it improves its overall performance on the validation set.

Despite this over-fitting, the final model that is used for the remainder of the TCN pipeline is based on the model with the lowest validation loss. This was at epoch 18. Therefore, the over-fitting that occurred between epoch 18 and 40 did not have any impact on the final model used for pointwise labelling.

To conclude, the loss curves of this TCN imply that the structure of the pseudo-labelling was learnt quickly and early in the training process. It is crucial to note here that this diverging validation loss curve has no influence on the final model used, since the final model used was that of epoch 18.

5.2.2 Agreement with the teacher model

In order to evaluate the extent to which the TCN labels agree with the pseudo-labels, it is imperative to use only IV traces that the TCN has not trained on. However, it is also the case that not all IV traces have a pseudo-label, because only the traces that passed the filter from section 4.2 have pseudo-labels. It is important to note here that only the IV/GV traces belonging to the reference molecules have pseudo-labels, which is why all analysis on the agreement with the teacher model done here is based on only the reference molecules. There are 1,653 files that are part of the TCN test set (and therefore unseen), and have pseudo-labels, thus the evaluation of the extent to which the teacher model and the student model agree will be evaluated on this subset of traces.

The first way in which we evaluate the agreement between the student and the teacher model is by figure 5.5, in which the percentage disagreement of each of the 1653 GV files is shown on a histogram.

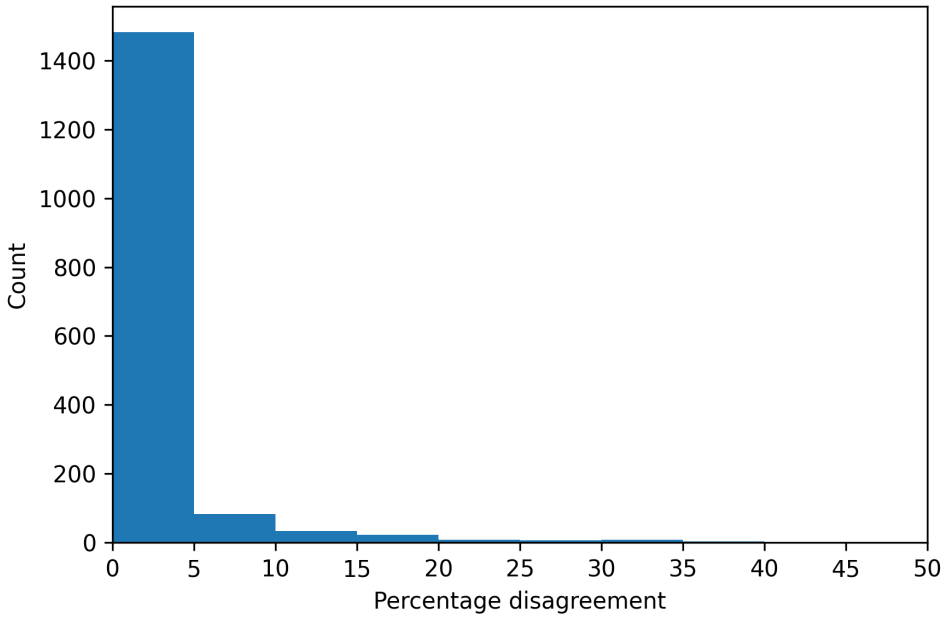


Figure 5.5: **Student-Teacher disagreement histogram.** A histogram showing the number of files that have a certain percentage pointwise disagreement between the student and the teacher labels. The histogram excludes empty bins above 50%

In figure 5.5 it is clear that the vast majority of GV traces have less than 5% disagreement. In fact, 60.44% of the GV traces have no pointwise classification disagreement between the pointwise TCN classifications and the pointwise pseudo-labels. Additionally, 89.72% of the 1,653 GV traces have less than 6% pointwise disagreement between the pseudo-labels and the TCN labels.

Another way of gaining understanding of the nature of the agreements and disagreements between the student and the teacher model for this set of GVs is by a contingency table of all the points in the 1,653 GV files.

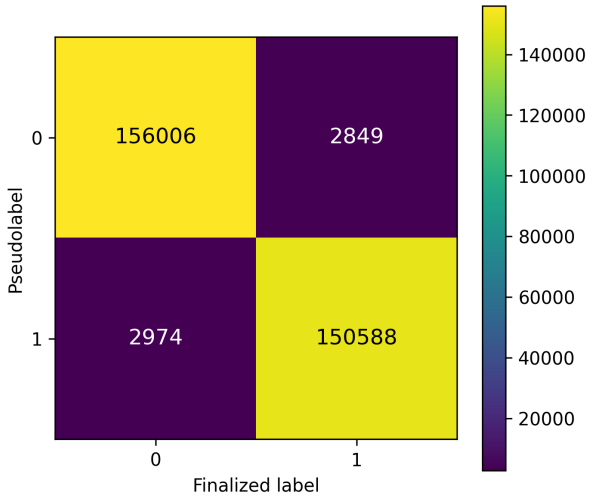


Figure 5.6: **Contingency table of pointwise pseudo-labels v. finalized TCN labels.** A contingency table of all points in the 1653 GV traces (312,417 total points) illustrating the frequency of different types of disagreements between the student and the teacher model. The values inside the array are the number of points in all the traces that are classified accordingly.

From figure 5.6 above, we conclude that the nature of the disagreements between the student and teacher models are symmetrical. This means that the number of points considered by the student model to be in state 0 and by the teacher model to be in state 1 is roughly equal to the number of points considered by the student to be in state 1 and by the teacher in state 0. Also, from this contingency table we can conclude that the percentage of pointwise agreement is 98.14%, which indicates very high fidelity.

There are, however, a few GV traces in which there is a high percentage disagreement, as seen in the histogram in figure 5.5.

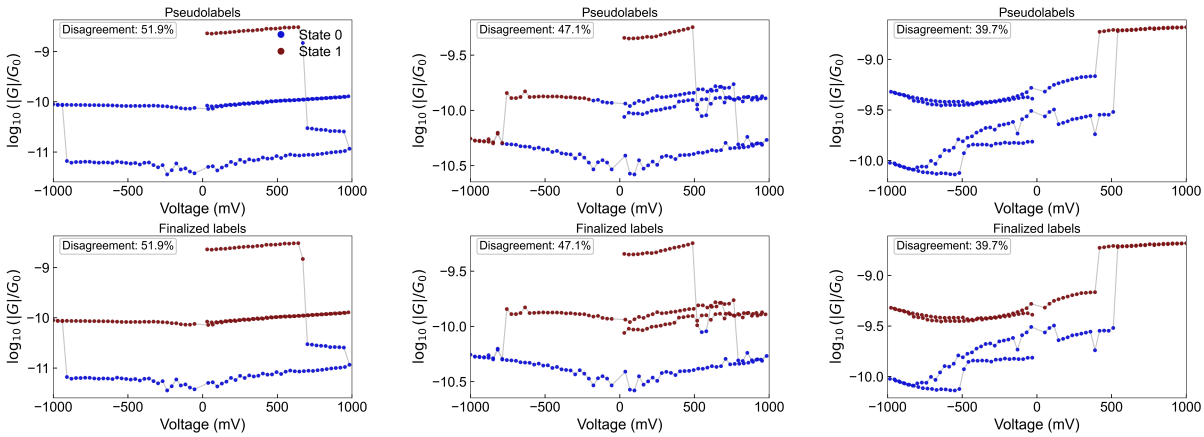


Figure 5.7: **GV traces with the 3 highest student-teacher disagreement rates.** The 3 GV traces with the highest disagreement rates out of the TCN unseen GV trace set. These traces have (from left to right) disagreements of 51.9%, 47.1% and 39.7%, but are all cases where the student TCN finalized label is clearly an improvement on the teacher HMM pseudo-labels.

The figure above shows three unseen GV traces that have the highest rates of disagreement between the student and teacher model. These disagreement rates are 51.9%, 47.1%, and 39.7%, respectively. Based on a visual inspection, it is fair to say that in these cases the student outperforms the teacher by following the apparent structure more closely.

5.3 Hysteretic filter results

The final stage in the pipeline is to run the trained TCN on all IVs of the three reference molecules, and then run these pointwise classified IVs through the hysteretic filter in order to end up with a set of true hysteretic, well classified IVs.

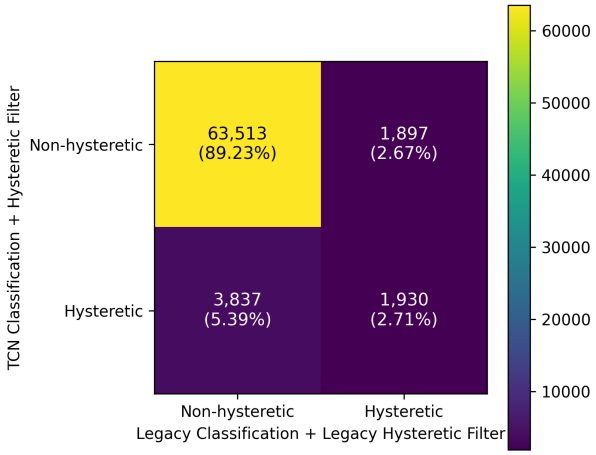


Figure 5.8: **Confusion matrix of legacy classification & legacy hysteretic filter v. TCN classification & hysteretic filter.** A contingency matrix comparing the number of IV traces the methodology in previous work considers hysteretic, to the number of IV traces considered hysteretic with this HMM TCN classification pipeline.

In figure 5.8, we see that the machine learning pipeline, in total, considers 8.1% of all IV traces hysteretic, whereas the classification and filtering implemented in previous work only considers 5.38% hysteretic [2]. In addition to this, there is only a small overlap of 33% between the two. Therefore, if the traces considered hysteretic by either one of the two methods are combined, that accounts for 10.77%, which is twice that of the legacy methodology in previous work.

5.3.1 Additional measurement trigger results

The measurement protocol of the MCBJ experiment from which the dataset in this thesis originates is based on an additional measurement trigger, as is outlined in subsection 4.6.1 [2]. To summarize; in order to determine whether or not a particular measurement setup, and therefore measurement set, can result in a high number of relevant hysteretic IV/GV traces, the first measurement of a given setup is evaluated to determine whether or not it is hysteretic. In order to determine this, the legacy pointwise classification methods are used in tandem with the legacy hysteretic filter is used. This methodology was chosen because, when the first measurement in a measurement set is hysteretic, there is a significant chance that when keeping the measurement setup the same, very similar high quality additional measurements can be made in order to increase the fraction of hysteretic IVs in the dataset. This measurement protocol, however, presents the problem in that, when the legacy hysteretic classification labels the first measurement of a given measurement set as not hysteretic despite it being so, a measurement set that could have produced relevant results is skipped.

In the interest of obtaining more usable measurement data, which is a central aim of this thesis, it is therefore imperative to calculate whether or not the pipeline outlined in this thesis would have triggered measurement sets resulting in additional relevant measurements, that the legacy data-analysis methodology would not have. To calculate this, all first measurements of the whole dataset were segregated from the rest of the dataset. On this set of first measurements, the legacy methodology and the methodology outlined in this thesis were applied on this set, to determine the extent to which this new methodology would have triggered additional measurements in more

measurements sets, resulting in a higher fraction of high quality hysteretic measurements.

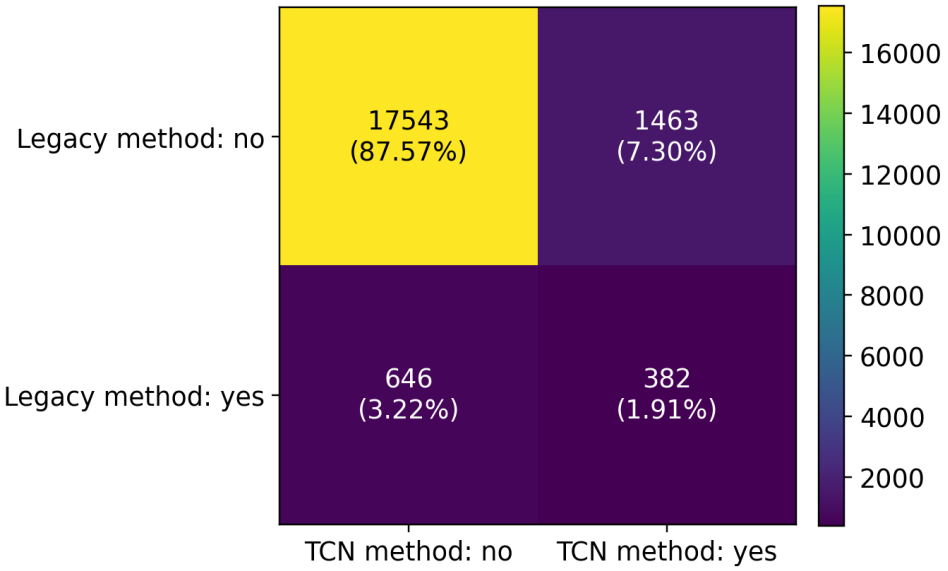


Figure 5.9: **Hysteretic classification contingency matrix comparison between legacy methodology and TCN pipeline methodology of first measurements.** This contingency matrix depicts the rates of hysteretic classification of IVs when applying the legacy and TCN pipeline methods for hysteretic classification on all first measurements in measurement sets. The calculations on which this contingency matrix is based included all first measurements from all five molecules in the full dataset. It is evident that, in total, the TCN methodology for hysteretic classification resulted in more than 9% of first measurements being classified as hysteretic, in contrast to the legacy methodology, which classified only around 5% as hysteretic.

In figure 5.9, the number of times additional measurements are triggers are compared between the TCN and the legacy methods. This trigger decision is based on determining whether or not the first measurement in a measurement set is hysteretic. The TCN method classifies around 9% of first measurements as hysteretic, but the legacy method only classifies 5% as hysteretic. The matrix also shows a significant disagreement between the two methods.

5.3.2 Manual audit results

In the hysteretics that result from Mechanically Controlled Break Junction experiments, the observation of clear conductance states is a well documented phenomenon. However, in these datasets, there is a high level of complexity, and there does not exist a full understanding of the physics behind these conductance states to the extent to which they can be classified by implementing known physical equations and mathematical understanding. This results in the fact that there is a fundamental lack of ground truth regarding whether or not a measurement is hysteretic or not.

In order to evaluate whether or not GVs classified as hysteretic, are actually hysteretic, a manual audit is done. A manual audit like this allows for a benchmark of accuracy that is measured against a true ground truth, namely, the manual classification of GVs as hysteretic or not.

To this end, 200 GVs were randomly sampled from all molecules, and labelled as hysteretic

or not, by hand. Then, both the legacy and the TCN methods for hysteretic classification are implemented on this set of 200 GVs, and their rates of agreement and disagreement are summarized in the figure below.

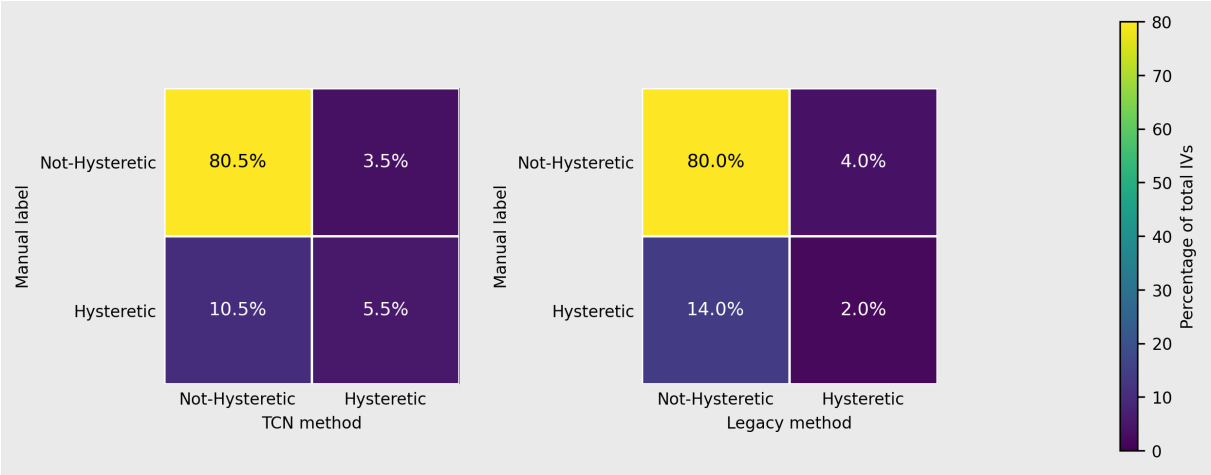


Figure 5.10: **Two contingency matrices of the agreement between the hysteretic classification of the manual labels, and both the legacy and TCN methods.** In this figure there are two contingency matrices depicting the rates of agreement and disagreement between the manual labelling of GVs as hysteretic or not, and both the legacy method of hysteretic classification and the TCN method of classification. It is clear that the performance of the TCN method is better than the legacy method. However, the TCN method still classifies 10.5% as not hysteretic despite them being hysteretic. This contingency matrix is based on a random sample of 200 GVs from all molecules.

In figure 5.10 above, the classifications of GVs from the randomly sampled set of 200 GVs are compared by contingency matrix. On the left of the figure, the contingency matrix comparing the TCN method's results to the manual audit labels indicates that 5.5% of the 16.0% manually labelled as hysteretic, are in fact correctly classified as hysteretic by the TCN method. This indicates that the TCN method still produces a significant fraction of false negatives. However, this is also the case for the legacy method, which only detects 2.0% of the 16.0% as hysteretic.

5.3.3 Physical Plausibility Results

It has been established, based on a small random sample of 200 GVs, that the TCN method correctly classifies GVs much more often than the legacy method. However, this manual audit is performed based on only a small sample of measurements, when considering that the full set of measurements is more than 100,000. The random set of 200 GVs therefore only accounts for less than 0.2% of the full dataset. Therefore, the extent to which the GVs considered hysteretic by the TCN method are physically plausible is evaluated, as this gives a large-scale validation criterion to measure the success of the TCN method and compare it to the legacy method.

This test of physical plausibility is done based on the assumption that a GV considered hysteretic should exhibit a relatively consistent conductance ratio across the voltage domain. This is quantified further by splitting the voltage domain into many bins, and for each bin calculating the conductance of the high and low conductance-states present in that bin, and calculating the logarithmic conductance ratio, referred to here as the log-separation. Then, for each GV, there is

a distribution of the log-separations present in the bins of that GV. From these distributions, the two important metrics that measure the consistency of log-separation across bins are the standard deviation and the interquartile range. The last metric that is also used below is the coverage, which is the fraction of bins that contain two points per state in each bin. This methodology is outlined in more detail in subsection 4.6.2.

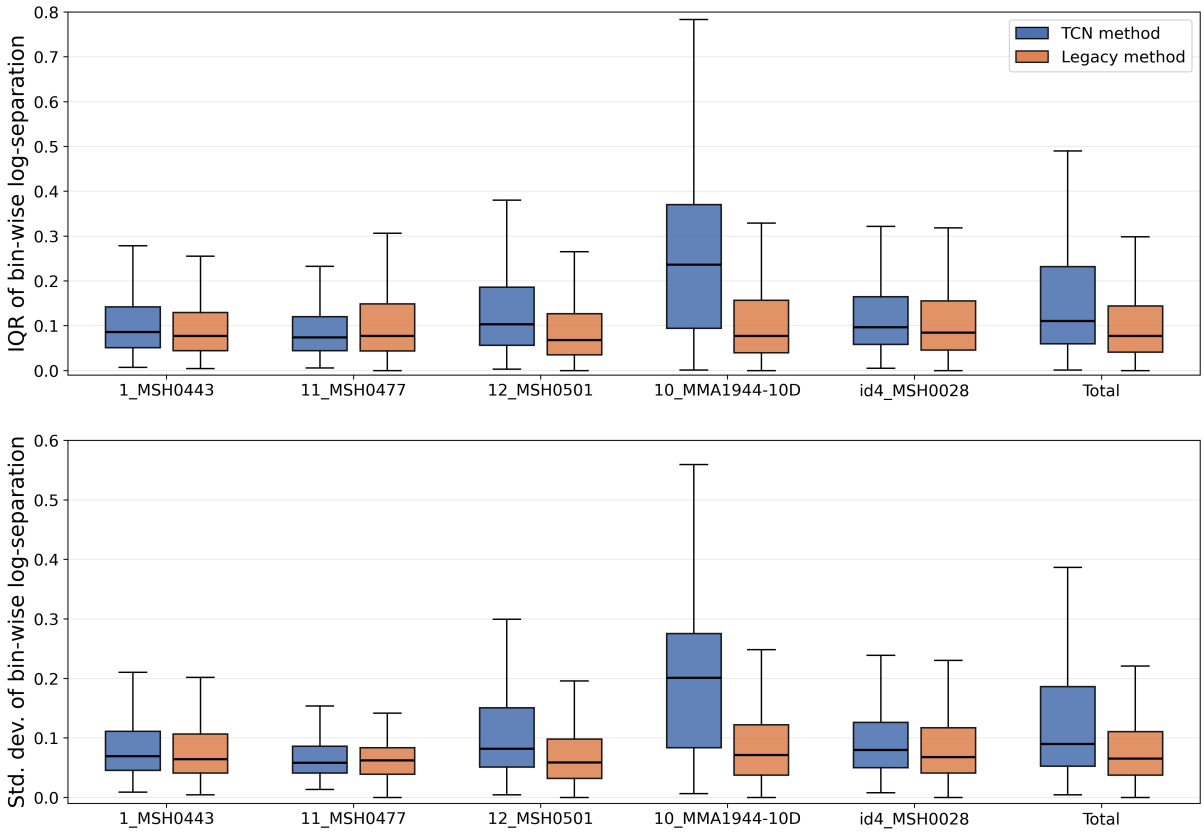


Figure 5.11: **Box plots of interquartile ranges and standard deviations of bin-wise log-separation distributions per GV.** These box plots depict the extent to which log-separation varies across the bins within each GV. **Upper:** The upper subplot shows box plots of the interquartile ranges of distributions of bin-wise log-separations per GV. **Lower:** This subplot depicts the same as the upper plot, but for the standard deviation of the distributions, instead of the IQR.

In figure 5.11 above, the box plots of each the interquartile range and the standard deviation of the per GV bin-wise log-separation distributions are shown. Both the interquartile ranges and the standard deviations of these per GV distributions are shown because the standard deviation measures the overall spread of the bin-wise log-separation whereas the IQR measures only the spread of the middle 50% of those values. However, both of these metrics are indicators of the extent to which log-separation remains consistent across bins.

From figure 5.11, it is evident that for almost all molecules the IQR and standard-deviation distributions of the bin-wise log-separations are very similar for the TCN and legacy methodologies. The two exceptions to this are molecules 12_MSH0501 and 10_MMA1944-10D (target molecule)

In the case of molecule 12_MSH0501, both the interquartile range and the standard deviation means are very similar, but their spreads towards high standard deviation and high IQR is

significantly larger than that of the legacy method. This is an indicator that, for this molecule, the TCN method classified GVs as hysteretic despite them having a relatively significant variation in log-separation across bins. This may be due to the fact that this molecule exhibits more conductance state shifting, but it can also be an indicator that the TCN method produced many false positives for this molecule, resulting in a less physically-plausible result than the legacy method.

The case of molecule 10_MMA1944-10D is very different in magnitude. For this target molecule, the IQR and standard deviation distributions have much higher means than that of the legacy method, and also exhibit a much more significant interquartile range of the box plots.

Besides those two molecules, both the standard deviations and the interquartile ranges of the different methods across the molecules are very consistent. Also, the total boxes on the right include molecule 10_MMA1944-10D, and are therefore highly influenced by the abnormalities of that molecule. This is clear in the fact that the means of the total boxes are very close between methods, but the third quartile is much larger.

In addition to the interquartile ranges and standard deviations providing for a measure of physical plausibility, the coverage of a GV is also a metric for physical plausibility. This is because a truly hysteretic GV should have both conductance states across large parts of the voltage domain. The coverage is the percentage of bins in the voltage domain that contain at least two points per conductance state, and is therefore a good measure for the extent to which conductance states are persistent in bias.

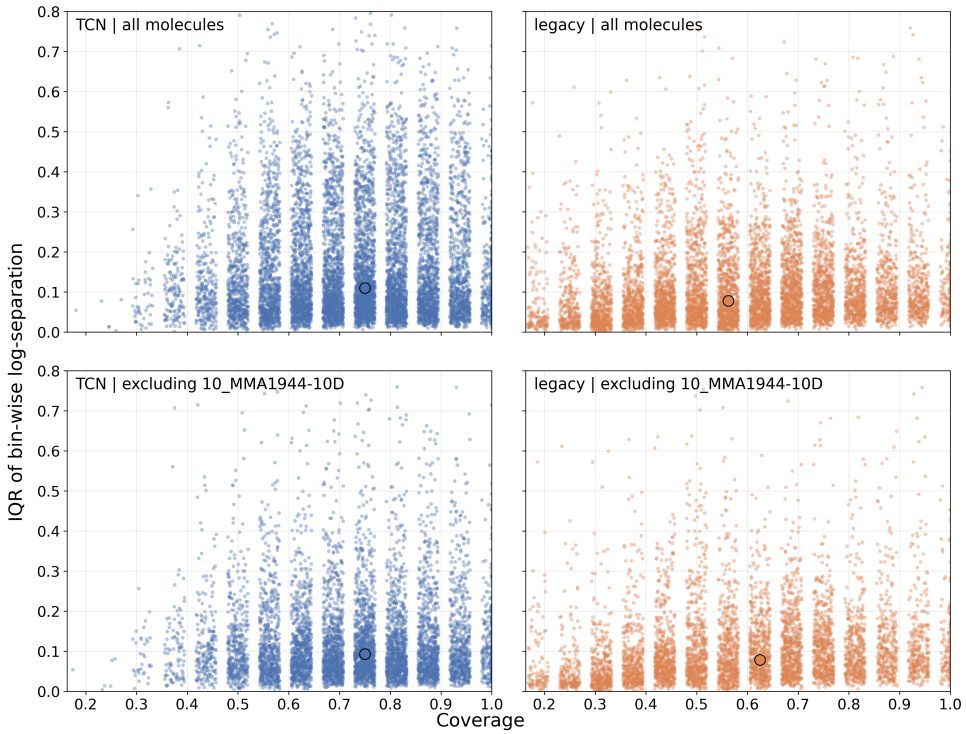


Figure 5.12: **Scatterplot of interquartile range and coverage of all molecules and of all molecules excluding molecule 10_MMA1944-10D, for both the TCN method and the legacy method.** This figure consists of scatter plots that depict the interquartile ranges of the GVs against the coverage of that GV. GVs that are more physically plausible should have a high rate of coverage, and a low interquartile range. Therefore, the bottom-right quadrant of each scatter plot is the region in which GVs can be considered more physically plausible. A small jitter is added to the x axis for better visualisation. The large points with black outline indicate the medians for each scatter plot.

In figure 5.12 above, the interquartile ranges are plotted against the coverages of each GV. Like explained above, high coverage corresponds to a sustained conductance plateau that spans across a large part of the voltage domain. True hysteretic plots should exhibit a high coverage rate. Therefore, it is ideal for a GV to be situated in the bottom right of the scatter plots, as that also corresponds to low interquartile range of bin-wise log separation.

In the figure, it is evident that, for the TCN method there is a significant difference between the scatter plot that includes the problematic molecule 10_MMA1944-10D and the one that excludes it. Furthermore, it is also apparent that the TCN method has a much higher coverage rate on average than the legacy method. This is evident both by visual inspection, but also by the medians in the scatter plots, with the mean of the legacy method excluding the problematic molecule being around 0.6 coverage, and the mean of the legacy method being at around 0.75 coverage. This is an indicator of the GVs being classified as hysteretic by the TCN method being more physically plausible than those classified by the legacy method.

To summarize these results, the plausibility analysis shows that for all molecules besides the outlier molecule, 10_MMA1944-10D, the TCN hysteretic traces are very similar to the legacy method hysteretic traces in terms of log-separation, but exhibit higher coverage.

5.4 Conductance and threshold voltage results

There are three key statistics that can be calculated from the final set of well-classified hysteretic GVs, that make for relevant downstream analysis. The first is the conductance-ratio distributions, which are the distributions of how frequent certain conductance ratios are present in different molecules. The second is conductance-state distributions, which are the distributions per molecule of the conductances of the high-conductance state and the low-conductance state. The last key statistic is the threshold voltages of each IV of each molecule. The threshold voltage is simply the voltage at which a state-switch occurs. This section shows these results as based on the TCN pipeline, and compares them to the results based on the legacy methodology. In this section, statistics are computed on the full dataset, but in accordance with memristivity rules outlined in previous work, the conductance ratio thresholds of the hysteretic filter, for analysis, are increased by 0.5, resulting in a smaller hysteretic subset on which the TCN results are computed [2].

In each subsection, the results will be split into the results for the reference molecules for which the pipeline in this thesis is designed, and the results for the target molecules that may exhibit three conductance states.

5.4.1 Conductance ratio results

Conductance ratio distributions of reference molecules

In figure 5.13 below, the conductance ratio distributions of each of the three molecules is calculated according to the IVs considered hysteretic by the TCN method (left), by the legacy method (center) and by the TCN method but not the legacy method (right).

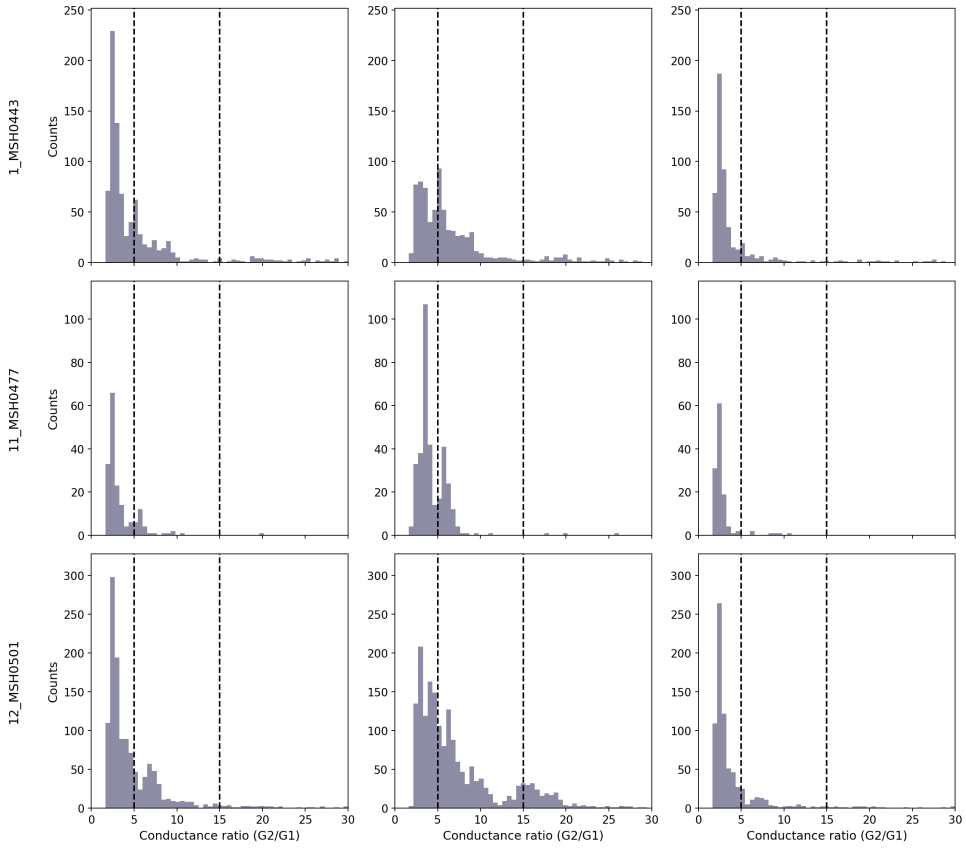


Figure 5.13: **Conductance ratio histogram comparison of reference molecules. Left:** The conductance-ratio distributions for each of the reference molecules when calculated based on all IV traces considered hysteretic by the TCN method. **Center:** The conductance-ratio distribution for each of the reference molecules when calculated based on all IV traces considered hysteretic by the legacy method. **Right:** The conductance-ratio distributions for each of the reference molecules for the set of IVs considered hysteretic by the TCN method but not considered hysteretic by the legacy method.

In figure 5.13 above it is evident that, with the TCN method, there are many more hysteretic plots passed that have a much lower conductance ratio. For the TCN method distributions, all three molecules exhibit much higher counts of conductance ratios under $R_G = 5$. Outside this low-ratio part of the histograms, there is no clear difference between the legacy and TCN methods that result in a significant change in conductance ratio distribution.

Conductance ratio distributions of target molecules

In the figure below, the conductance ratio distributions are depicted of the target molecules, in the same way was done in figure 5.13 above. It is important to note here that the TCN method is trained and intended for two-state classification.

In figure 5.14 above, the distributions of the conductance ratios of the target molecules are presented. In these distributions, there is no clear pattern of note besides the very high count of low conductance ratios in the molecule 10_MMA1944-10D.

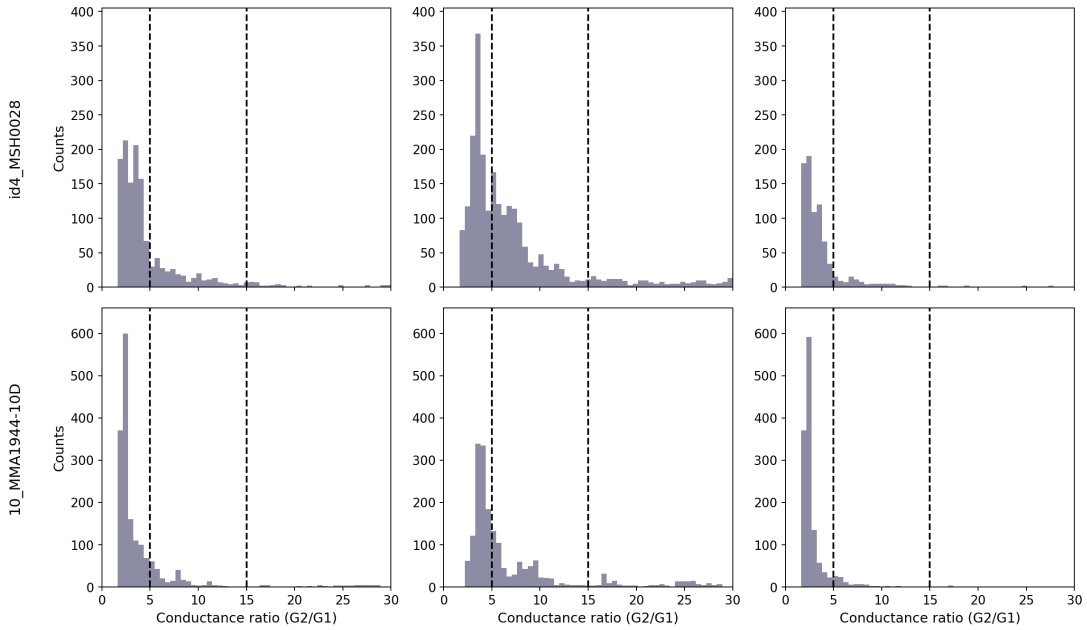


Figure 5.14: **Conductance ratio histogram comparison of target molecules. Left:** The conductance-ratio distributions for each of the target molecules when calculated based on all IV traces considered hysteretic by the TCN method. **Center:** The conductance-ratio distribution for each of the target molecules when calculated based on all IV traces considered hysteretic by the legacy method. **Right:** The conductance-ratio distributions for each of the target molecules for the set of IVs considered hysteretic by the TCN method but not considered hysteretic by the legacy method.

5.4.2 Conductance distribution results

Calculating the conductance ratio of an IV is difficult, because, as is discussed in the physical plausibility subsection above, these conductance ratios can vary across the voltage domain. For this reason, it is important to also consider the distributions of the conductances of each state, per molecule. By analysing the conductance distributions of the two states, it is possible to gain a more comprehensive understanding of the behaviour of the molecule than purely by considering the conductance ratio distribution.

Conductance state distributions of reference molecules

In figure 5.15 below, the conductance distributions per state per reference molecule are depicted as histograms.

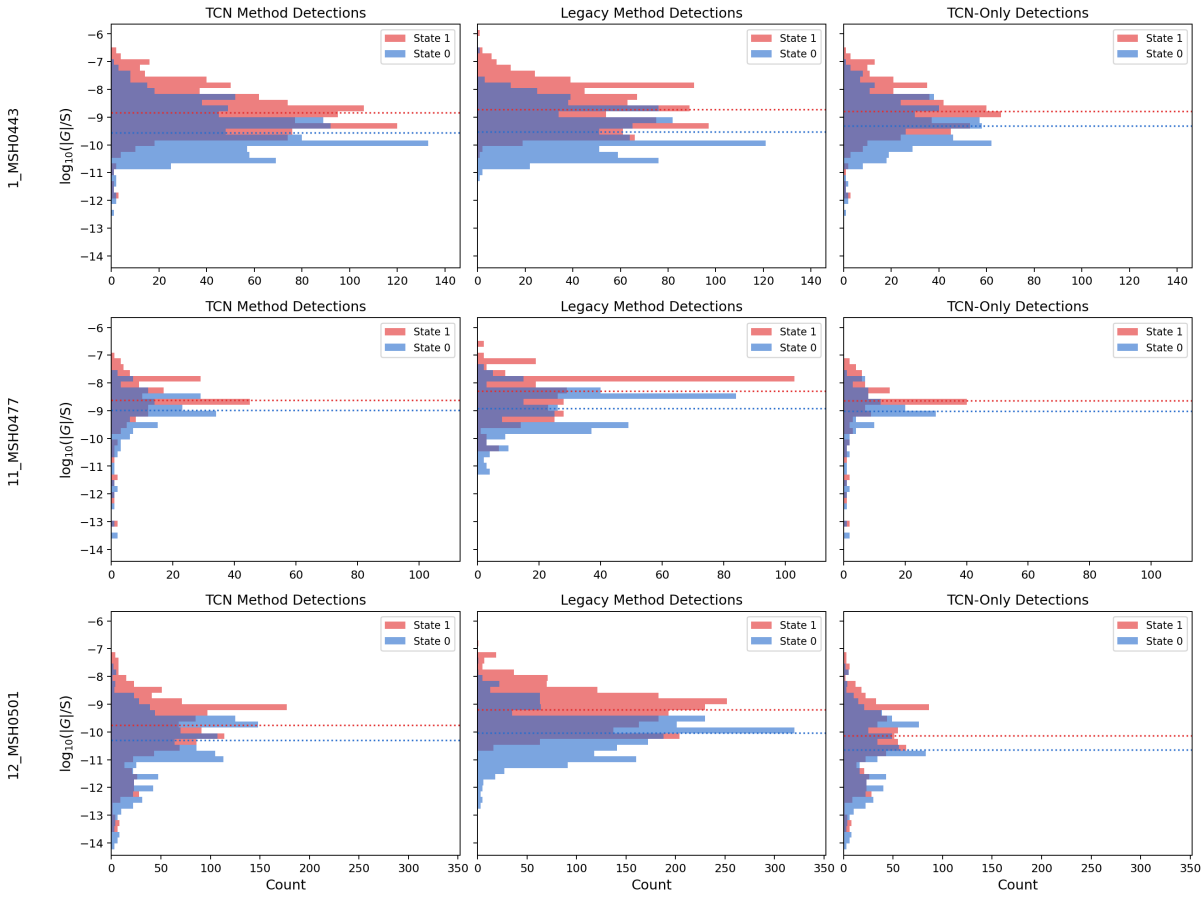


Figure 5.15: **Conductance state distribution histograms of reference molecules.** In this figure, each row corresponds to a different molecule and each column corresponds to hysteretic IV detections/classifications based on either the TCN method (left), the legacy method (centre) or the difference between the two (right).

In figure 5.15 above, it is clear that the distributions, when compared between the TCN method and the legacy method, of all three molecules take very similar shapes. However, in all three molecules it is evident that the TCN method has a fraction median conductances are considered to be part of in state 1 despite being below the rest of the distribution. This is physically implausible because, by definition, if a data point, and therefore a median, in a GV is below the lower conductance state, it should never belong to the higher conductance state.

In addition to this observation, it is also evident that the TCN method detections contain more overlap than the legacy method detections. This observation can be directly linked with the fact that in figure 5.13, low conductance ratios are much more prevalent.

Conductance state distributions of target molecules

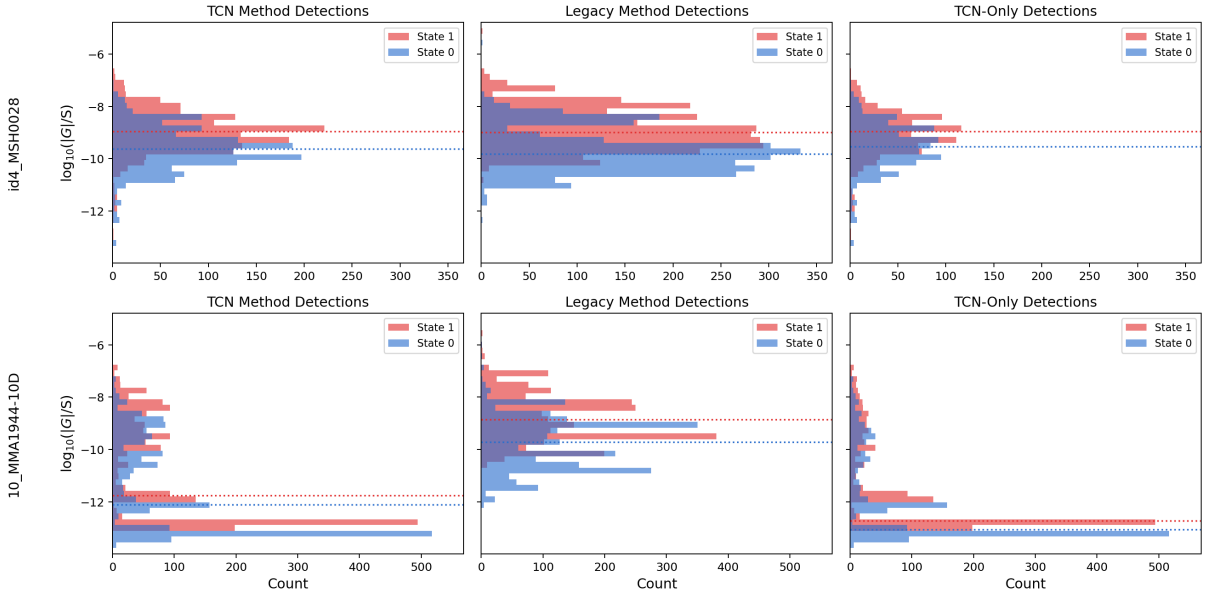


Figure 5.16: **Conductance state distribution histograms of target molecules.** In this figure, each row corresponds to a different target molecule and each column corresponds to hysteretic IV detections/classifications based on either the TCN method (left), the legacy method (centre) or the difference between the two (right).

Figure 5.16 shows a pronounced anomaly for 10_MMA1944-10D, where the TCN-selected conductance histograms contain strong spikes at specific conductance levels. No similarly strong feature is observed for id4_MSH0028, whose conductance distributions are more similar to those seen in the reference molecules.

Figure 5.16 shows the conductance state distributions plotted as histograms per target molecule and per methodology. The histogram for molecule 10_MMA1944-10D contains a very pronounced anomaly for the TCN method because the histogram spikes significantly at very low conductances. These spikes also correspond with two different states, which is also very abnormal. There is no feature like that visible in the histograms for molecule id4_MSH0028. The conductance distributions per method of molecule id4_MSH0028 are broadly similar.

5.4.3 Threshold voltage results

The final physical statistic calculated in previous work on this dataset is the threshold voltages of the IVs. A threshold voltage is simply the voltage at which a switch occurs. In purely hysteretic GVs, these threshold voltages should somewhat symmetrical. Below are the threshold voltage distributions for each of the five molecules. There are three ways in which threshold voltage statistics can be presented; the first is an overall histogram where all threshold voltages are plotted together as one, the second is a state dependant histogram, where bars are colour-coded based on which conductance state is being switched to, and the last is sweep direction histogram where bars are colour-coded based on the sweep direction of the sweep in which the sweep takes place. These statistics are all presented and discussed in this subsection.

Threshold voltage statistics of reference molecules

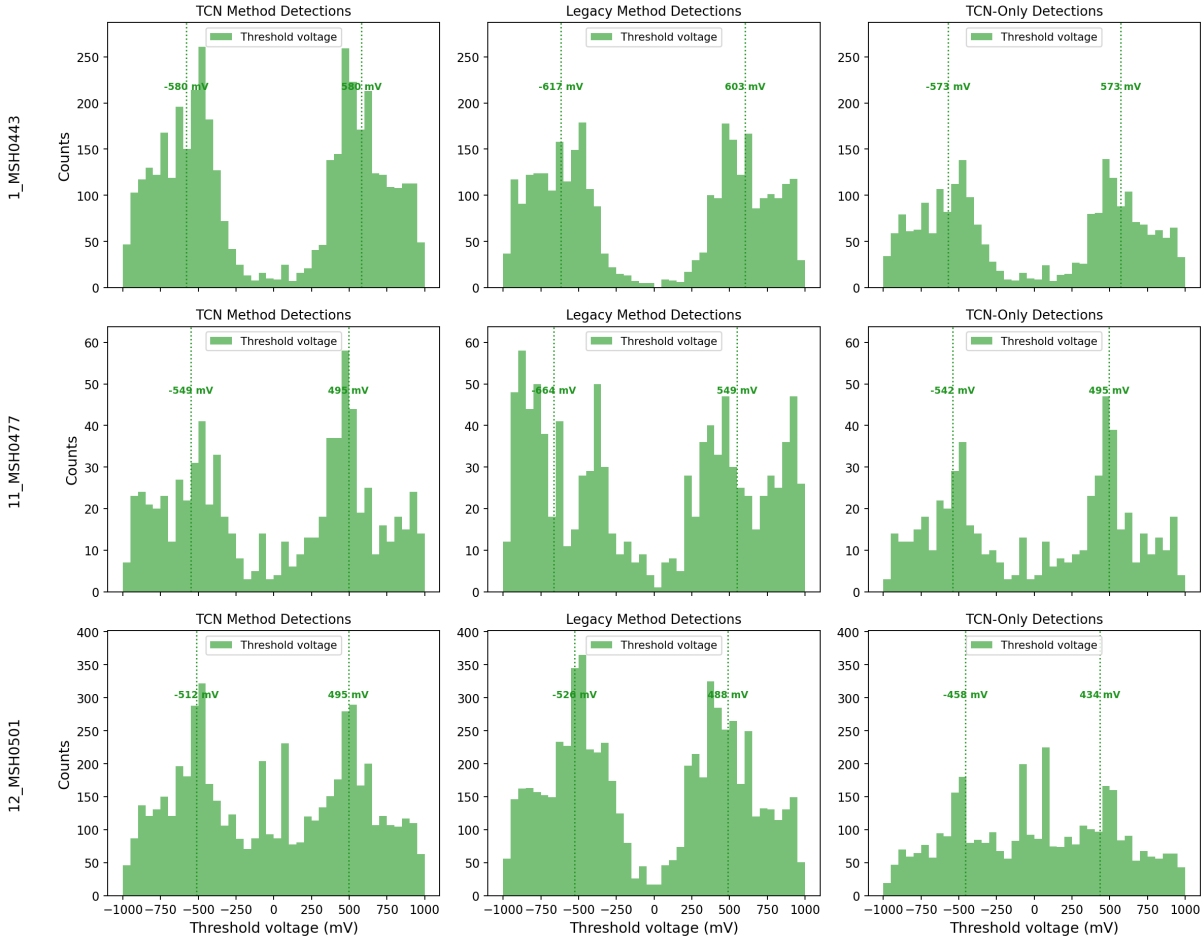


Figure 5.17: **Threshold voltage histogram comparison between the TCN method and the legacy method for reference molecules.** A subplot of histograms of the number of switches observed per voltage. Each column represents a different subset of IVs. The green dotted vertical lines represent the medians per side of the voltage domain. **Left:** switching statistics based on the TCN method **Centre:** switching statistics based on the legacy method. **Right:** switching statistics based on the the subset of IVs that the TCN method determined to be hysteretic and the legacy method did not.

In figure 5.17 above it is evident that the added information of including the hysteretic definition based on this thesis is evenly spread for molecule 1_MSH_0443, however for molecule 11_MSH_0477 it is clear in the TCN-only detections plot that the net effect of adding the data considered hysteretic by this pipeline is not evenly centred on each side of the domain around the average threshold voltages (green dotted lines), but rather very spread out with peaks either sides of the voltage domain. It is also clear to see that in molecule 12_MSH0501, the TCN method detects a significant number of switches near in the low-bias region. The two other reference molecules (11_MSH0477 and 1_MSH0443) have a more distinct bimodal structure for the TCN method than the legacy method.

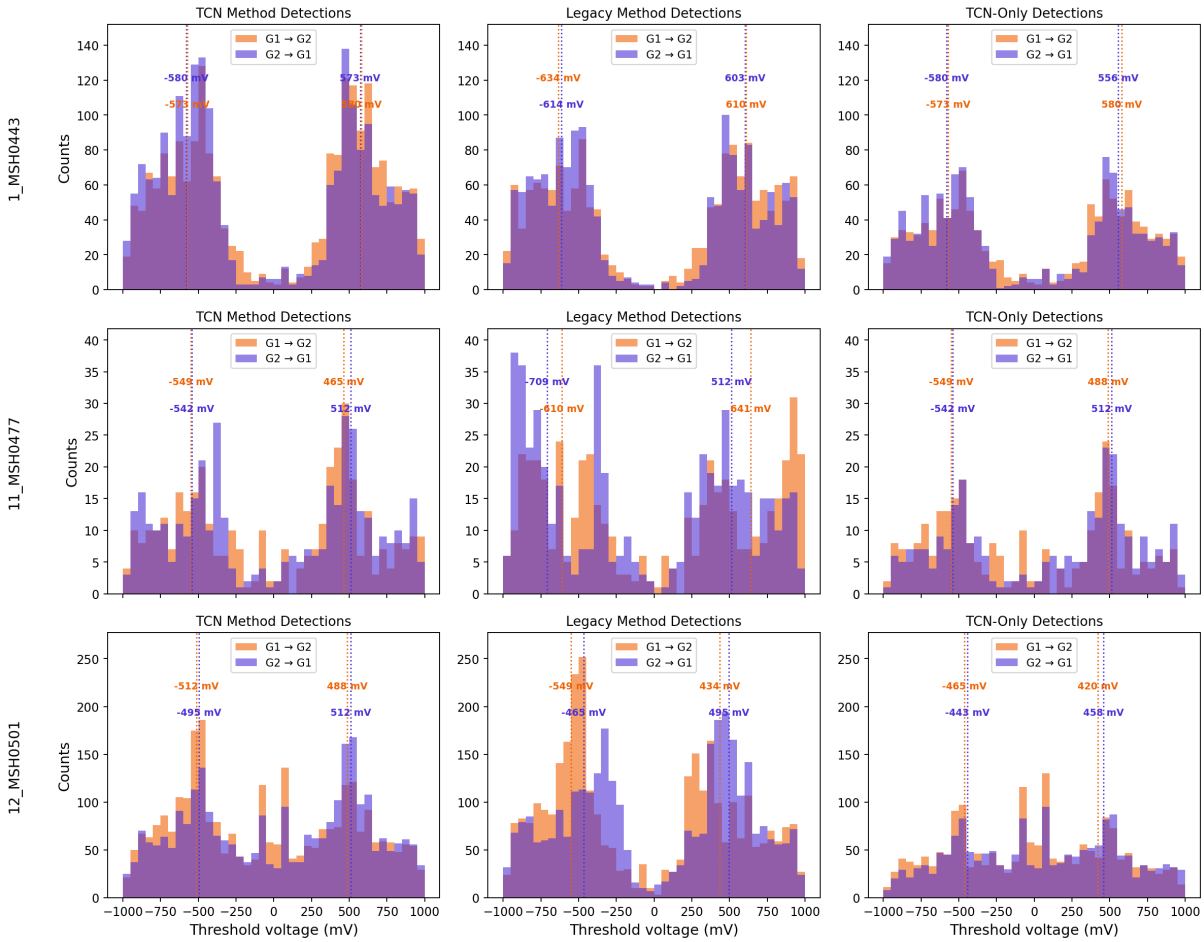


Figure 5.18: **Threshold voltage histogram comparison between the TCN method and the legacy method, colour-coded based on switch direction for reference molecules.** A subplot of histograms of the number of switches observed per voltage, with switches color-coded based on the direction of the switch. Each column represents a different subset of IVs. The orange and purple dotted vertical lines represent the medians per switching direction per side of the voltage domain. **Left:** switching statistics based on the TCN method **Centre:** switching statistics based on the legacy method. **Right:** switching statistics based on the the subset of IVs that the TCN method determined to be hysteretic and the legacy method did not.

From figure 5.18 it is clear that state switch direction statistics are relatively similar between the TCN method and the legacy method. However, there is one notable difference. In the legacy method histogram for molecules 11_MSH0477 and 12_MSH0501, both sides of the histograms exhibit significant peaks of switching both directions. In contrast to the legacy method, the TCN method does not have these unidirectional peaks on both sides of the domain.

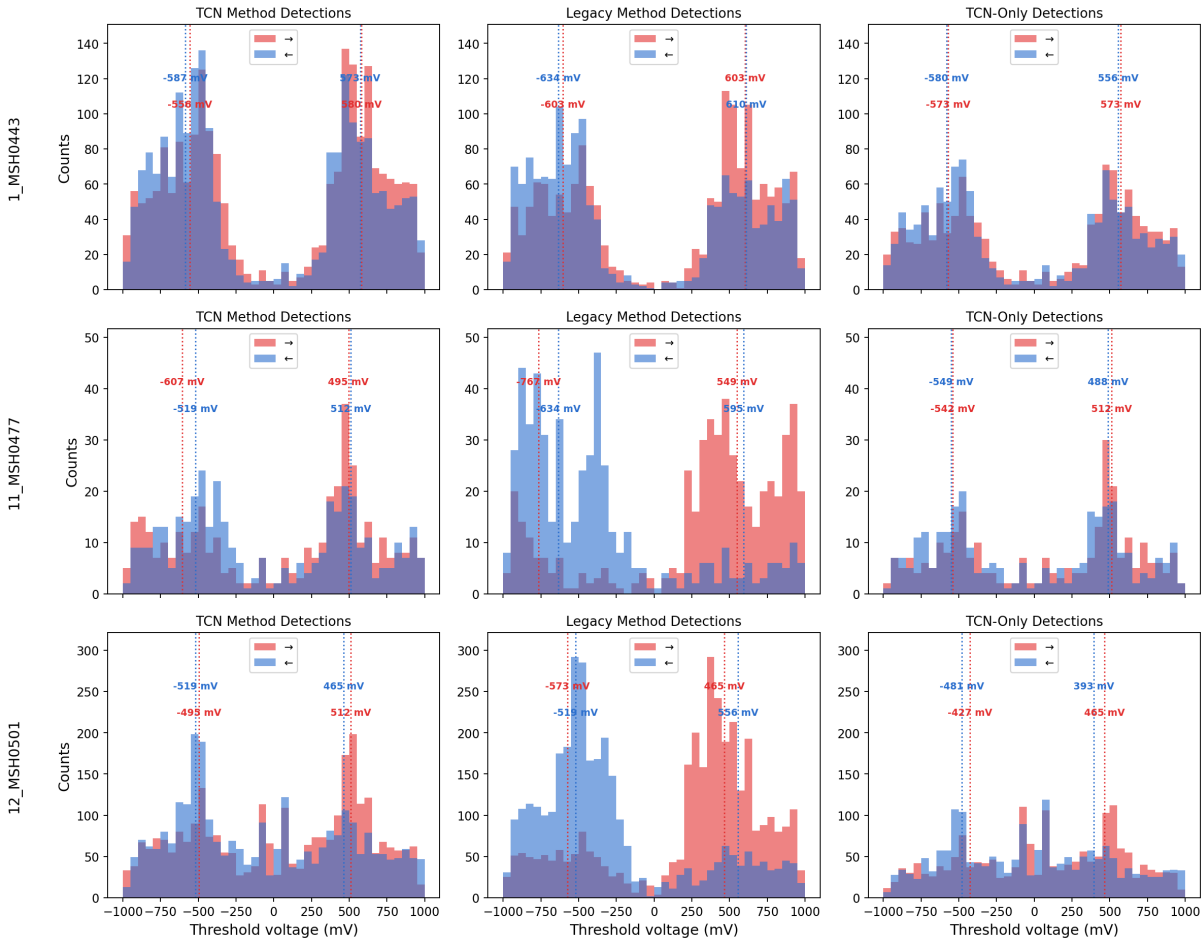


Figure 5.19: **Threshold voltage histogram comparison between the TCN method and the legacy method, colour-coded based on sweep direction for reference molecules.** A subplot of histograms of the number of switches observed per voltage, with switches colour-coded based on the sweep direction of the sweep in which the switch took place. Each column represents a different subset of IVs. The red and blue dotted vertical lines represent the medians per sweep direction per side of the voltage domain. **Left:** switching statistics based on the TCN method **Centre:** switching statistics based on the legacy method. **Right:** switching statistics based on the the subset of IVs that the TCN method determined to be hysteretic and the legacy method did not.

In figure 5.19 above, there is one clear statistical difference between the legacy and TCN methods. Namely, for the molecules 11_MSH0477 and 12_MSH0501, the histograms of the legacy method have a much better directional split in voltage threshold. More specifically, for $V < 0$, the vast majority of switches occurred in the left moving sweep, and the opposite is true for $V > 0$. In contrast to this, the TCN method does not have such a clear split.

Threshold voltage statistics of target molecules

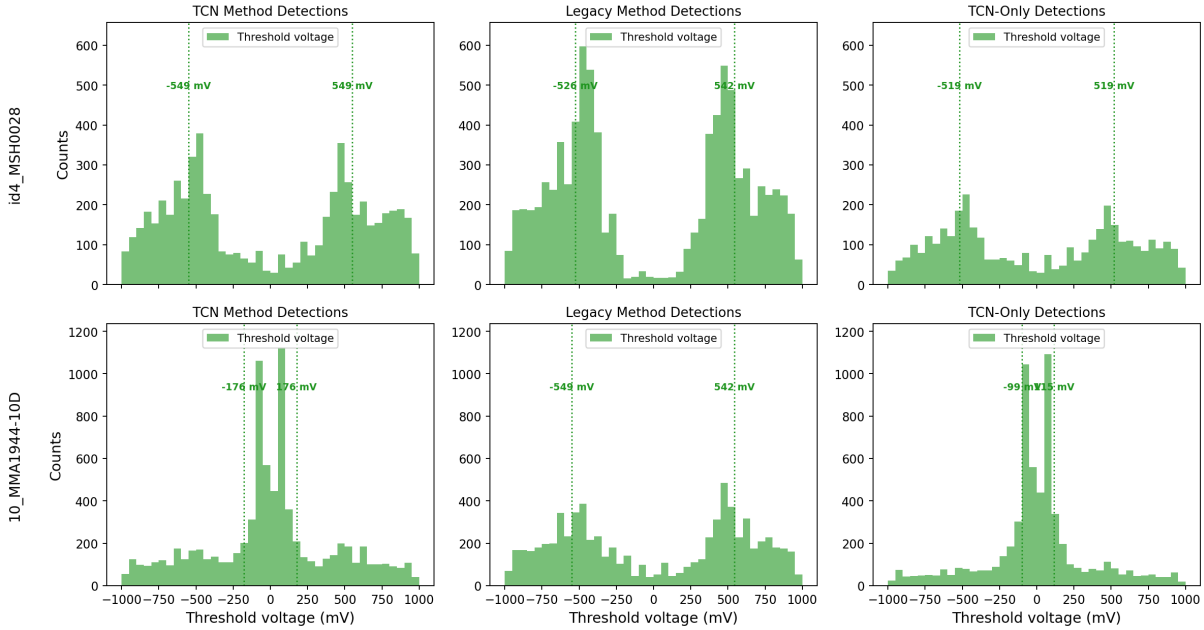


Figure 5.20: **Threshold voltage histogram comparison between the TCN method and the legacy method for target molecules.** A subplot of histograms of the number of switches observed per voltage. Each column represents a different subset of IVs. The green dotted vertical lines represent the medians per side of the voltage domain. **Left:** switching statistics based on the TCN method **Centre:** switching statistics based on the legacy method. **Right:** switching statistics based on the the subset of IVs that the TCN method determined to be hysteretic and the legacy method did not.

In figure 5.20 above the overall switching histogram is given. In this histogram, there is only one artefact that warrants discussion; the fact that TCN histogram for molecule 10_MMA1944-10D has a unimodal structure. The fact that this molecule exhibits abnormal switching behaviour is not surprising given the number of previous statistics on this molecule that also do not conform with reasonable expectations.

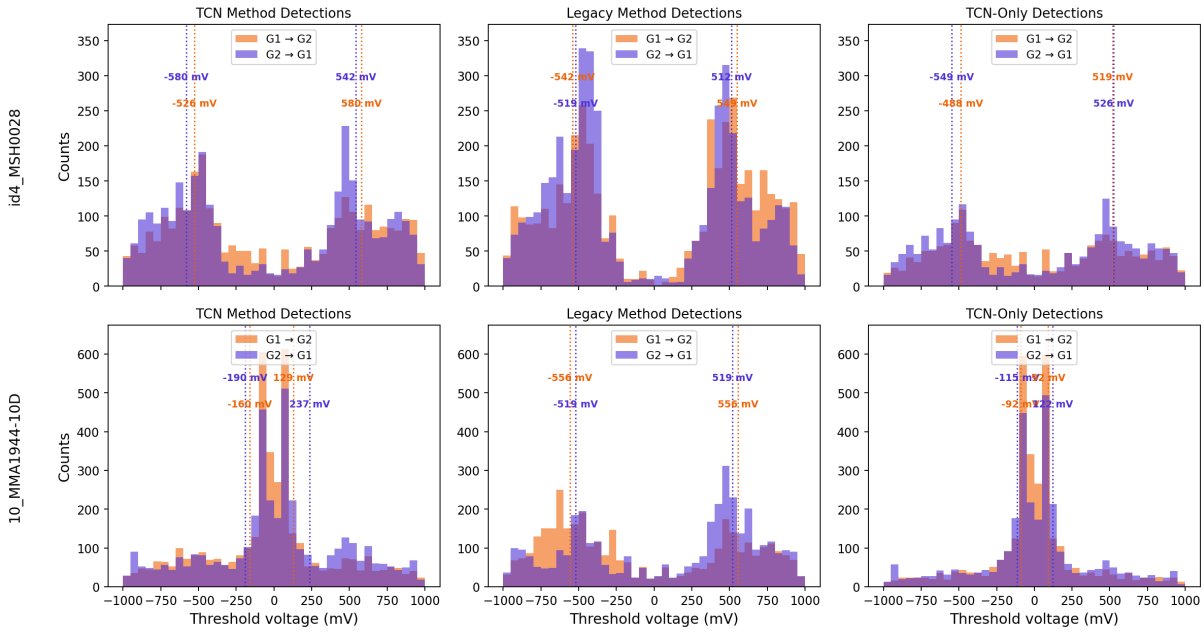


Figure 5.21: **Threshold voltage histogram comparison between the TCN method and the legacy method, colour-coded based on switch direction for target molecules.** A subplot of histograms of the number of switches observed per voltage, with switches color-coded based on the direction of the switch. Each column represents a different subset of IVs. The orange and purple dotted vertical lines represent the medians per switching direction per side of the voltage domain. **Left:** switching statistics based on the TCN method **Centre:** switching statistics based on the legacy method. **Right:** switching statistics based on the the subset of IVs that the TCN method determined to be hysteretic and the legacy method did not.

In figure 5.21 above, the histograms of the switching statistics (colour-coded by switch direction) of the target molecules is given. In these histograms, there is no discernable difference between the TCN and legacy methods besides the previously discussed differences for molecule 10_MMA1944-10D. The TCN and legacy histograms of the other molecule, molecule id4_MSH0028, are very similar.

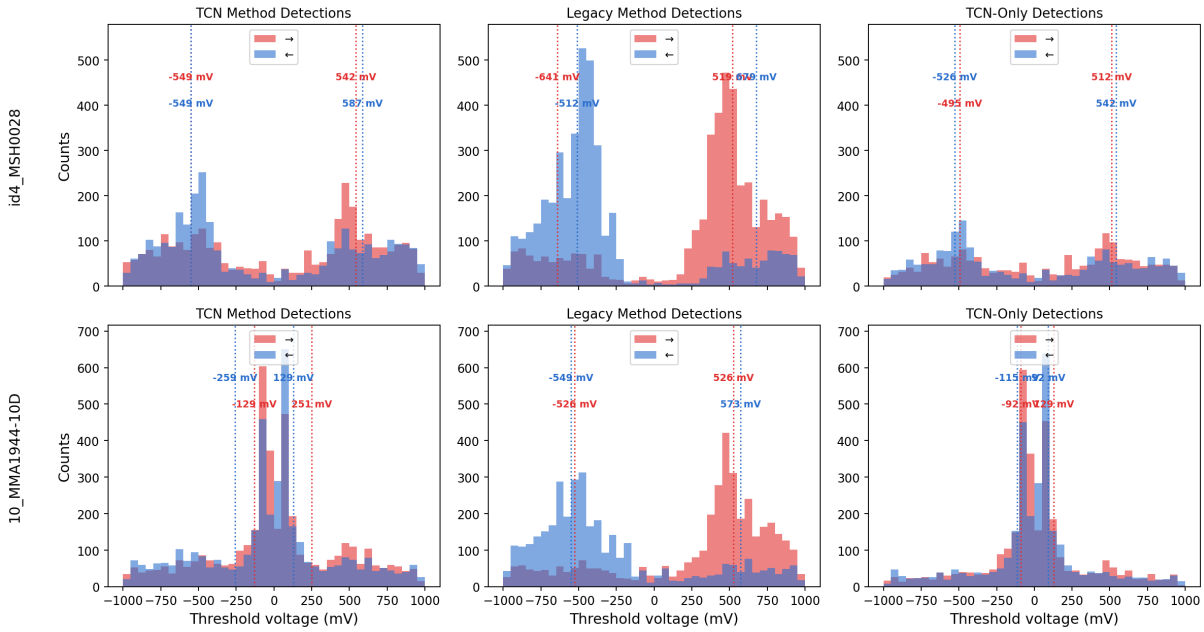


Figure 5.22: **Threshold voltage histogram comparison between the TCN method and the legacy method, colour-coded based on sweep direction for target molecules.** A subplot of histograms of the number of switches observed per voltage, with switches colour-coded based on the sweep direction of the sweep in which the switch took place. Each column represents a different subset of IVs. The red and blue dotted vertical lines represent the medians per sweep direction per side of the voltage domain. **Left:** switching statistics based on the TCN method **Centre:** switching statistics based on the legacy method. **Right:** switching statistics based on the the subset of IVs that the TCN method determined to be hysteretic and the legacy method did not.

In the above figure of switch statistics histograms with sweep direction colour-coded, the abnormalities of molecule 10_MMA1944-10D remain unchanged. However, in molecule id4_MSH0028, there is a clear difference between the legacy and the TCN method histograms. Namely, like in figure 5.19 above, there is a very clear directional split in the legacy method histogram, whereas this is not the case for the TCN method.

Chapter 6

Discussion & Conclusion

6.1 Discussion

6.1.1 Main findings in relation to the thesis objective

The central aim of this thesis was developing a machine-learning pipeline that could be used in order to accurately perform pointwise state classification of data points in GV plots that could result in a larger fraction of the data being used for downstream physical analysis. The results presented in this thesis suggest that this aim was, for the most part, accomplished for the scope of the modelling pipeline, namely, two conductance state systems.

The most important contribution in this thesis are the SLR-HMM that understands temporal persistence, a scalable student TCN neural network that can reproduce pseudo-labels results well and the hysteretic filter that follows this pipeline that is used to decide whether or not a GV is hysteretic. When comparing the legacy methodology with this machine-learning pipeline, the data retention of this pipeline is higher, without sacrificing the physical plausibility (for all molecules except one, which is a target molecule). However, results across molecules vary, indicating that the pipeline is not optimized on a per-molecule basis. Also, the pipeline is shown to still miss a portion of all hysteretic GVs in the available set.

6.1.2 Role of the SLR-HMM teacher model

The simple linear regression Hidden Markov model in this pipeline improves on the legacy classification techniques used in previous work because it is much less sensitive to outliers/points that do not conform to the overall two state structure of the GV. Examples of this are the half-sweeps that do not conform with the prevalent two state structure in a GV, as is shown in figure 5.2. This lack of sensitivity is because of the RANSAC line-fitting implementation within the hidden Markov model. The Hidden Markov Model also implements a strong form of temporal persistence that encodes the understanding of the fact that in hysteretic GV traces, states do not switch very often, as can be seen in figure 5.3. The purely geometric classifications from the previous work assign states point by point, but the HMM favours sequences of the same state which improves pseudo-labels in noisy regions. The pseudo-labels produced by the Hidden Markov Model are more contiguous and also more consistent with the physical understanding of state persistence within conductance plateaus. This is especially relevant when dealing with

drifting conductance plateaus. In summary, the SLR-HMM results support the implementation of this model as a teacher model, intended to teach the TCN state persistence and general accurate labelling.

6.1.3 What the TCN student model learned

Training behaviour

The loss curves of the TCN in figure 5.4 suggest that the structure of the pseudo-labels was learned very early in the training process. This is evident due to the rapid drop in validation loss before epoch 12, and the subsequent high reproducibility of pseudo-labels.

There are three likely reasons why the over-fitting after epoch 18 may have occurred. The first is that the teacher pseudo-labels are not necessarily very complex in and of themselves. In the end, a two state labelling can be very easy to understand given the fact that the GVs on which the TCN was trained were already heavily filtered. The second possible reason for over-fitting can be because the training set does not vary significantly in structure. Since most irrelevant plots were already filtered, the structure of the remaining GVs are mostly hysteretic and therefore not very complex which allows for easier and faster learning. The last possible reason is that the model may have had a larger capacity than was necessary. The TCN had an input projection width of 128, 7 residual blocks and a very wide context window. It is likely that a combination of these hyper-parameters could have been changed to make the TCN simpler, which could have prevented the over-fitting but still retained the quality of the resulting model.

The final model was selected to be the model that had the smallest validation loss, which means that all overfitting that occurred after epoch 18 did not have any effect on the final TCN used for pointwise classification.

Agreement with the teacher model

There was a very high rate of agreement between the student and teacher models when calculated based on the test-set (unseen by TCN). Namely, 60% of the GV traces in the test set had no pointwise disagreement, and 90% had less than 6% disagreement. This indicates strongly that the student was successful in learning the teacher signal.

This is very important because the practical value of introducing the TCN into the pipeline is to replace the computationally heavy teacher model. The TCN can, quickly and effectively, without per-IV fine-tuning, produce an accurate classification that is very high in agreement with the teacher model. The TCN therefore appears to be able to work as a scalable replacement for the teacher model. The high agreement rates between the student and teacher classifications also mean that any limitations later in the pipeline are likely to be caused by teacher-label quality and hysteretic-filter design, not the neural network itself.

High-disagreement cases

As is evident in figure 5.5, not all GVs have a very low disagreement rate. Most notably, there are a few GVs in the test set that have disagreement rates above 30%. The three GVs with the highest disagreement with the teacher are shown in figure 5.7, where it is clear that the student model is

capable of smoothing out local errors made by the teacher pseudo-labels. A likely explanation for this phenomenon is that when the student trained on the large set of pseudo-labelled traces, it learnt its dominant regularities but was less sensitive to rare imperfections in the teacher output. This may also have to do with the very wide context window of the TCN, and that due to this large context window, single point and few-point irregularities were compensated for by the context around them. This qualification is qualitative and it cannot be said for certain that this is the case.

6.1.4 Hysteretic detection as a trade-off between retention and purity

Increased retention and experimental relevance

The most important result of this thesis is the difference in retained hysteretic traces relative to the legacy methodology after the final hysteretic filter. The results presented in figure 5.8 provide strong evidence that the TCN pipeline has higher data-retention. This is directly relevant to the stated aim of increased data-retention in order to have more data that can be used for downstream analysis.

The results from the trigger plot, figure 5.9 reinforces this point, but in a different way. Namely that, if the TCN methodology were used during experimentation, then around twice as many measurement settings and sets would have triggered additional measurements.

Evidence from the manual audit

The absence of a ground truth of pointwise state classification is the most fundamental limitation of the problem presented in this thesis. In order to, despite this fact, be able to evaluate the results effectively, it is necessary to perform a manual audit. The results of the manual audit are presented in figure 5.10, which clearly demonstrate that the TCN method classified more than twice as many GVs as hysteretic correctly than the legacy method. This serves as strong external evidence that the pipeline presented in this thesis is capable of classifying GVs as hysteretic or not much better than the existing methodology.

An important secondary result to come from the manual audit is the fact that out of the 16% of GVs manually labelled as hysteretic, the TCN method only classified 5.5% correctly. This shows that there are still very many missed GVs, and that the pipeline can still be improved and optimized further.

Finally, it is also worth noting that the TCN method also produced a relatively significant number of false positives. Out of the 9% classified as hysteretic by the TCN method, 3.5% were false positives. This also indicates that the TCN pipeline can still definitely be improved upon.

6.1.5 Physical plausibility and molecule-specific behaviour

Most molecules

The plausibility analysis provides a complement to the manual audit. This is because the manual audit only concerns a small random sample from the set of all GVs. In contrast to this, the plausibility analysis allows for a broad check as to whether or not the GVs considered hysteretic by the TCN are actually physically plausible. For all molecules besides the molecules 12_MSH0501

and 10_MMA1944-10D, there was no discernable difference between the legacy and the TCN model in the box plot in figure 5.11. Because of this fact that the standard deviations and interquartile ranges of the bin-wise log-separations of these three molecules are almost identical to the legacy model, it is reasonable to deduce that the hysteretic filter produced more GVs considered hysteretic, but the physical plausibility of these added GVs was not sufficiently significant to change the bin-wise log-separation distributions of the TCN method.

In figure 5.12, it is clear that the TCN methodology, when excluding molecule 10_MMA1944-10D, has a much higher rate of coverage. This would imply that both conductance states are more often present in a higher percentage of the voltage bins. Higher coverage is consistent with more persistent two state behaviour across the voltage range, and therefore more ideal hysteretic GVs. This suggests that the TCN method results in GVs of higher quality due to the higher state persistence present throughout the bins.

When considered together, the results from the physics plausibility test support the conclusion that the pipeline retains a higher portion of traces, without seeing a change in the physical plausibility of these traces, for most molecules

Molecule 12_MSH0501

One of the two important exceptions to the discussion on plausibility results above is molecule 12_MSH0501. For this molecule, the standard deviation and interquartile range box plots for the TCN method in figure 5.11 showed a longer upper tail, a higher median, and a wider third quartile than the other two reference molecules. This can both have to do with the number/quality of hysteretic GVs retained for this molecule by the TCN method, but also have to do with the molecule data set itself. In this way, this difference in behaviour is ambiguous because it can reflect both of these options.

The fact that this molecule does not conform with the other two reference molecules is also prevalent in the fact that its threshold voltage results are also less clean and ideal than those of the other reference molecules. It is, however, important to note that both in the case of figure 5.11 and the threshold voltage results, the differences between this molecule and the other two molecules are not incredibly significant.

Molecule 10_MMA1944-10D

The most prevalent failure of this current pipeline is the molecule 10_MMA1944-10D. This molecule performs very poorly on all plausibility metrics, and also has incredibly different distributions when it comes to the downstream physics analysis. This molecule clearly behaves as an outlier when compared to the four other molecules.

In the physics plausibility box plot results in figure 5.11, for both the IQR and std. dev. boxplots, this molecule exhibits a much higher rate of bin-wise log-separation deviation. This is clear from the figure by the fact that in both subfigures, the full IQR of the molecule is above the medians of all other molecules. Also, its upper tail spans to more than twice that of the other molecules.

The clearest evidence of this molecule being an outlier is by its conductance state distributions and threshold voltage statistics. In figure 5.16, there are two clear spikes, of different states in

the low conductance region. Not only are these two spikes in the histogram very large, but also, they are attributed to different conductance states. This is very abnormal behaviour because the conductances of the conductance states that are calculated for figure 5.16 are based on the median conductance of all data points in that state. In that figure, one of the two spikes is red, and therefore corresponds to the higher conductance state. For this to happen, there must be a large number of GVs for which the median upper conductance state exist within this low region, which is very unlikely. In addition to this, the voltage threshold plots based on the TCN method in figure 5.20 show that a very high number of switches occur in the low bias region, which also corresponds to very unusual behaviour.

There are two possible explanations for this. The first is that this molecule exhibits significant three-state behaviour that is not compatible with the two state classification pipeline in this thesis. If this is the case, then the potential third state gets classified as belonging to one of the two other states. This would explain the high rates of bin-wise log-separation, because for one portion of the GV the conductance ratio would be calculated binwise between state 0 and state 1, whereas in another part of the GV the conductance ratio would be calculated between state 0 and a potential state 2 (third state) that exists on a whole different conductance plateau.

The second possible explanation for this is that the hysteretic filter is poorly calibrated for this molecule. More specifically, in all four rules of the hysteretic filter, that depend on conductance ratio, that stays the same across molecules. If this molecule typically exhibits much higher conductance ratios, then it is possible that a significant fraction of non-hysteretic IVs passed the filter.

6.1.6 Interpretation of downstream physical statistics

Conductance-ratio and conductance-state distributions

One of the most visible effects of the new pipeline is in the conductance ratio histograms in figures 5.15 and 5.13. In both figures there is a significant increase in low conductance-ratio traces from the legacy method to the TCN method. This could be due to a higher portion of the IVs in the TCN method exhibiting switching due to a low conductance ratio switching mechanism such as contact-geometry. However because conductance ratio ranges of different switching mechanisms overlap, this is insufficient evidence to conclude anything.

The conductance state distributions also, generally, have much more overlap between the two states for the TCN method, possibly also alluding to the workings of low conductance ratio switching mechanisms. This lack of state separation can also be a price associated with higher GV retention.

Threshold-voltage structure

Threshold voltage statistics provide an additional measure on the quality of the retained hysteretic set in that a true hysteretic set should have a relatively symmetrical threshold voltage distribution.

Across all molecules, there is no significant difference in the extent to which the histograms are symmetrical between the TCN method and the legacy method. However, there are a few relevant individual results:

The most important result from this data is from figure 5.19. In this figure, the lower two molecules see a big difference between the histograms between the methods. The legacy method seems to have much more switches on each side belonging to one sweep direction. This is an indicator that the legacy method produces a set of GVs with a low amount of noisy switching regions. In contrast to the legacy method, the TCN method has much more overlap on both sides of the domain. This suggests that the TCN method produces hysteretic GVs with more noise in switching regions, than the legacy method, for those two molecules. The exact same phenomenon occurs in figure 5.22 for molecule id4_MSH0028.

These figures suggest that despite the new pipeline improving retention, these GVs can contain noisy switching regions that are revealed by the threshold voltage distributions.

6.1.7 Limitations and Future Work

The central limitation of this thesis is the fact that there does not exist a ground truth for the pointwise state classification and final hysteretic classification. Despite this being partially addressed in the manual audit, that is still a human benchmark based on only a very small portion of all GVs. The second main limitation is the student teacher model structure. This is because the student model cannot prevent learning the biases present in the pseudo-labels produced by the teacher. A third limitation is the scope of the current pipeline. This pipeline is designed for two state systems, and can therefore perform poorly on three state systems like the target molecules. In order to perform analysis on three state molecular systems, a fully different pipeline is necessary. This is well exemplified by the performance of molecule 10_MMA1944-10D. Building a three state classification pipeline is a fully different question, and is therefore a good topic for future work on this data set.

Another limitation is that the hysteretic filter at the end of the pipeline remains inconsistent, and when tested against the manual benchmark, it resulted in a high number of false negatives. The task of decreasing the rates of false negatives is a problem in and of itself, as it would require very specific and difficult fine tuning of all variables used in this pipeline. Therefore, optimizing this pipeline to reduce the number of false negatives is also a good topic for future work on this pipeline.

Lastly, it is crucial to note that all physics downstream analysis could not be done without running the TCN model partially on data that it had seen in training. Since the teacher-student disagreement rates are so low, this is unlikely to result in vastly different results as if the TCN had not seen a part of the data set, but it remains a limitation.

6.2 Conclusion

The central aim presented in this thesis was the development of a pipeline that implemented machine learning techniques in order to improve the accuracy of point-wise state assignments in order to retain a higher portion of hysteretic IV traces that can be used for further physical analysis. This aim exists within the scope of building the machine learning pipeline such that it is designed for two-state reference-molecules. The aim is largely achieved through a four-state pipeline consisting of initial filtering, a Simple Linear Regression Hidden Markov Model

teacher, a Temporal Convolutional Neural Network and a final hysteretic filter. The teacher model implemented in this pipeline was capable of producing pseudo-labels that exhibited temporal persistence, which were used in order to train the TCN. The neural network was able to reproduce the teacher pseudo-label signal with very high agreement, which enabled the TCN to act as an efficient large-scale surrogate for the teacher model, capable of replacing the relatively computationally heavy teacher model. For most molecules, the increased hysteretic GV retention did not correspond to an evident loss of physical plausibility. However, the pipeline did show limitations with regards to certain molecules, that resulted in abnormalities in the downstream physics analysis. The main bottlenecks of the problem presented in this thesis are the absence of a ground truth pointwise state classification, the significant portion of false negatives produced by the final hysteretic filter and the ineffectiveness of this two state pipeline on more complex target molecules. In general, this thesis demonstrates that a teacher-student pipeline is a viable way to do large scale data analysis on two state memristive IV datasets without a ground truth. This, while providing a basis for future analysis on this dataset, and providing the foundations for future work into improved filtering and an extension to three state systems.

Bibliography

[1] Leon O. Chua. "Memristor—The Missing Circuit Element". In: *IEEE Transactions on Circuit Theory* 18.5 (1971), pp. 507–519. DOI: 10.1109/TCT.1971.1083337.

[2] Lucienne van der Geest. "Toward Disentangling Intrinsic and Extrinsic Single-Molecule Memristive Switches". MA thesis. Delft University of Technology, 2025.

[3] Dong Xiang, Xiaonan Wang, Chenguang Jia, Taehun Lee, and Xin Guo. "Mechanically Controllable Break Junctions for Molecular Electronics". In: *Advanced Materials* 25.35 (2013), pp. 4845–4867. DOI: 10.1002/adma.201301589.

[4] Srinivasan V. Aradhya and Latha Venkataraman. "Single-Molecule Junctions Beyond Electronic Transport". In: *Nature Nanotechnology* 8 (2013), pp. 399–410. DOI: 10.1038/nnano.2013.91.

[5] Juan Carlos Cuevas and Elke Scheer. *Molecular Electronics: An Introduction to Theory and Experiment*. World Scientific, 2010.

[6] Xiaona Xu, Chunyan Gao, Ramya Emusani, Chuancheng Jia, and Dong Xiang. "Toward Practical Single-Molecule/Atom Switches". In: *Advanced Science* 11.29 (2024), p. 2400877. DOI: https://doi.org/10.1002/advs.202400877. eprint: https://advanced.onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202400877. URL: https://advanced.onlinelibrary.wiley.com/doi/abs/10.1002/advs.202400877.

[7] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Communications of the ACM* 24.6 (1981), pp. 381–395. DOI: 10.1145/358669.358692.

[8] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. 2nd ed. Springer, 2009.

[10] Lawrence R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition". In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286. DOI: 10.1109/5.18626.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[12] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *arXiv preprint arXiv:1803.01271* (2018).

[13] Ilya Loshchilov and Frank Hutter. “Decoupled Weight Decay Regularization”. In: *International Conference on Learning Representations (ICLR)* (2019).

Appendix A

Code Appendix

A.1 Filtering

```
#!/usr/bin/env python3

from __future__ import annotations

import os
import re
import math
import glob
from dataclasses import dataclass
from typing import List, Tuple, Dict, Optional, Any

import numpy as np

# user settings

DATA_DIR = r"C:\Users\maxdo\nn_clustering\data\raw\11_MSH0477"

FILE_GLOB_PRIMARY = "IV*.dat"
FILE_GLOB_FALLBACK = "*.dat"

V_CUTOFF = 30.0
EPS_G = 1e-20

# manifest output

WRITE_MANIFEST = False
MANIFEST_BASENAME = "filter_manifest"

# random sampling

SAMPLE_ON = False
SAMPLE_PERCENT = 10.0
SAMPLE_SEED = 166574123645678893

# filter toggles
```

```
36
37 FILTER_GMM_BIC_ON = True
38 FILTER_PLATEAU_JUMP_ON = True
39 FILTER_WINDOW_CONTRAST_ON = True
40
41 # GMM / BIC settings
42
43 GMM_MIN_RATIO = 1.3
44 GMM_MIN_WEIGHT = 0.15
45 GMM_BIC_IMPROVEMENT = 10.0
46 GMM_MIN_N = 20
47
48 # GMM solver settings
49 GMM_RANDOM_STATE = 0
50 GMM_N_INIT = 2
51 GMM_MAX_ITER = 200
52 GMM_COV_TYPE = "full"
53 GMM_REG_COVAR = 1e-6
54
55 # plateau + jump settings
56
57 DY_FLAT = 0.015
58 P_FLAT_MIN = 0.60
59
60 # jump threshold
61 JUMP_RATIO_MIN = 1.6
62
63 LMIN = 6
64 MIN_PLATEAU_RUNS = 2
65
66 # window contrast settings
67
68 WIN_LO = 150.0
69 WIN_HI = 250.0
70 WINDOW_RATIO_MIN = 1.5
71
72 # viewer defaults
73
74 START_INDEX = 0
75
76 # import IV class
77
78 try:
79     from mechanical_switches.utils import IV  # type: ignore
80 except Exception:
81     from iv import IV  # type: ignore
82
83 # summaries
84
85 @dataclass
86 class GMMSummary:
87     ok: bool
```

```
    bic1: float
    bic2: float
    dbic: float
    mu1: float
    mu2: float
    ratio: float
    w1: float
    w2: float
    reason: str

# load data and build IV

PAT = re.compile(r"^IV.*_(\d+)_(\d{4})\.dat$", re.IGNORECASE)

def read_iv_dat(path: str) -> Tuple[np.ndarray, np.ndarray]:
    # read V and I from file
    V, I = [], []
    with open(path, "r", encoding="utf-8", errors="ignore") as f:
        for line in f:
            s = line.strip()
            if not s or s.startswith("#"):
                continue
            parts = re.split(r"[,\s;]+", s)
            if len(parts) < 2:
                continue
            try:
                v = float(parts[0])
                i = float(parts[1])
            except ValueError:
                continue
            if math.isfinite(v) and math.isfinite(i):
                V.append(v)
                I.append(i)
    if not V:
        raise ValueError("No numeric V,I data found.")
    return np.asarray(V, float), np.asarray(I, float)

def _parse_ids_and_date_from_name(name: str) -> Tuple[int, int, str]:
    trace_id = 0
    iv_id = 0
    m = PAT.match(name)
    if m:
        trace_id = int(m.group(1))
        iv_id = int(m.group(2))
    date_match = re.search(r"\d{2}-[A-Za-z]{3}-\d{2}", name)
    date = date_match.group() if date_match else ""
    return trace_id, iv_id, date

def load_iv(path: str) -> IV:
    # make IV object
    V_mV, I = read_iv_dat(path)
    V_V = V_mV * 1e-3
```

```
140
141     base = os.path.basename(path)
142     trace_id, iv_id, date = _parse_ids_and_date_from_name(base)
143
144     iv = IV(
145         file=base,
146         date=date,
147         trace_id=trace_id,
148         id=iv_id,
149         vbias=V_V,
150         current=I,
151         motor_position=0,
152         config_id=None,
153         magnetic_field=0,
154         settings={},
155     )
156     return iv
157
158 def build_logG_from_IV(iv: IV) -> Tuple[np.ndarray, np.ndarray]:
159     # build log|G| after the voltage cutoff
160     # same cutoff near zero when conductance is computed
161     zero_thr_V = float(V_CUTOFF) * 1e-3
162
163     G, Vbias = iv.calculate_conductance(zero_threshold=zero_thr_V, offsetted=True)
164
165     G = np.asarray(G, float)
166     Vbias = np.asarray(Vbias, float)
167
168     if G.size == 0 or Vbias.size == 0:
169         return np.array([], dtype=float), np.array([], dtype=float)
170
171     n = min(G.size, Vbias.size)
172     G = G[:n]
173     Vbias = Vbias[:n]
174
175     # back to mV for the rest of the script
176     Vbias_mV = Vbias * 1e3
177
178     # apply voltage cutoff
179     mask = np.isfinite(Vbias_mV) & np.isfinite(G) & (np.abs(Vbias_mV) >= float(
            V_CUTOFF))
180     Vv = Vbias_mV[mask]
181     Gv = G[mask]
182
183     if Vv.size == 0:
184         return Vv, np.array([], dtype=float)
185
186     y = np.log10(np.maximum(np.abs(Gv), float(EPS_G)))
187     return Vv, y
188
189 # GMM / BIC filter
190
```

```
def gmm_bic_summary(y: np.ndarray) -> GMMSummary:
    if y.size < int(GMM_MIN_N):
        return GMMSummary(
            ok=False, bic1=np.nan, bic2=np.nan, dbic=np.nan,
            mu1=np.nan, mu2=np.nan, ratio=np.nan, w1=np.nan, w2=np.nan,
            reason=f"GMM: too few points (N={y.size} < GMM_MIN_N={GMM_MIN_N})"
        )

    try:
        from sklearn.mixture import GaussianMixture
    except Exception:
        return GMMSummary(
            ok=False, bic1=np.nan, bic2=np.nan, dbic=np.nan,
            mu1=np.nan, mu2=np.nan, ratio=np.nan, w1=np.nan, w2=np.nan,
            reason="GMM: scikit-learn not available (install scikit-learn)"
        )

    X = y.reshape(-1, 1)

    g1 = GaussianMixture(
        n_components=1,
        covariance_type=GMM_COV_TYPE,
        reg_covar=GMM_REG_COVAR,
        max_iter=GMM_MAX_ITER,
        n_init=GMM_N_INIT,
        random_state=GMM_RANDOM_STATE,
    )
    g2 = GaussianMixture(
        n_components=2,
        covariance_type=GMM_COV_TYPE,
        reg_covar=GMM_REG_COVAR,
        max_iter=GMM_MAX_ITER,
        n_init=GMM_N_INIT,
        random_state=GMM_RANDOM_STATE,
    )

    g1.fit(X)
    g2.fit(X)

    bic1 = float(g1.bic(X))
    bic2 = float(g2.bic(X))
    dbic = bic1 - bic2

    mus = g2.means_.reshape(-1).astype(float)
    ws = g2.weights_.reshape(-1).astype(float)

    order = np.argsort(mus)
    mu1, mu2 = float(mus[order[0]]), float(mus[order[1]])
    w1, w2 = float(ws[order[0]]), float(ws[order[1]])

    ratio = 10.0 ** (mu2 - mu1)

```

```
243     return GMMSummary(
244         ok=True,
245         bic1=bic1, bic2=bic2, dbic=dbic,
246         mu1=mu1, mu2=mu2, ratio=float(ratio),
247         w1=w1, w2=w2,
248         reason="",
249     )
250
251 def filter_gmm_bic(iv: IV) -> Tuple[bool, str, Optional[GMMSummary]]:
252     _, y = build_logG_from_IV(iv)
253     s = gmm_bic_summary(y)
254     if not s.ok:
255         return False, f"GMM_BIC failed: {s.reason}", s
256
257     reasons = []
258     if s.dbic < float(GMM_BIC_IMPROVEMENT):
259         reasons.append(f"ÎŤBIC={s.dbic:.1f} < {GMM_BIC_IMPROVEMENT:.1f}")
260     if s.ratio < float(GMM_MIN_RATIO):
261         reasons.append(f"ratio={s.ratio:.2f} < {GMM_MIN_RATIO:.2f}")
262     if min(s.w1, s.w2) < float(GMM_MIN_WEIGHT):
263         reasons.append(f"min_w={min(s.w1, s.w2):.2f} < {GMM_MIN_WEIGHT:.2f}")
264
265     if reasons:
266         return False, "GMM_BIC failed: " + " | ".join(reasons), s
267
268     return True, "", s
269
270 # plateau + jump filter
271
272 def _count_plateau_runs(flat_edge_mask: np.ndarray, *, lmin_points: int) -> Tuple[
    int, int]:
273     if flat_edge_mask.size == 0:
274         return 0, 0
275
276     runs = 0
277     max_len_points = 0
278     i = 0
279     n = flat_edge_mask.size
280     while i < n:
281         if not flat_edge_mask[i]:
282             i += 1
283             continue
284         j = i
285         while j < n and flat_edge_mask[j]:
286             j += 1
287         run_len_edges = j - i
288         run_len_points = run_len_edges + 1
289         max_len_points = max(max_len_points, run_len_points)
290         if run_len_points >= lmin_points:
291             runs += 1
292         i = j
293     return runs, max_len_points
```

```
294
295 def filter_plateau_jump(iv: IV) -> Tuple[bool, str, float]:
296     _, y = build_logG_from_IV(iv)
297     if y.size < max(5, int(LMIN)):
298         return False, f"PLATEAU_JUMP failed: too few points (N={y.size})", np.nan
299
300     dy = np.diff(y)
301     abs_dy = np.abs(dy)
302
303     flat_edges = abs_dy < float(DY_FLAT)
304     plateau_fraction = float(np.mean(flat_edges)) if flat_edges.size else 0.0
305     jump_strength = float(np.max(abs_dy)) if abs_dy.size else 0.0
306
307     jr = float(JUMP_RATIO_MIN)
308     if jr <= 1.0:
309         jr = 1.0000001
310     dy_jump_min = math.log10(jr)
311
312     n_runs, max_run_pts = _count_plateau_runs(flat_edges, lmin_points=int(LMIN))
313
314     ok = True
315     reasons = []
316
317     if plateau_fraction < float(P_FLAT_MIN):
318         ok = False
319         reasons.append(f"plateau_frac={plateau_fraction:.2f} < {P_FLAT_MIN:.2f} (
            DY_FLAT={DY_FLAT:.3f})")
320
321     if jump_strength < float(dy_jump_min):
322         ok = False
323         reasons.append(f"max|dy|={jump_strength:.3f} < log10({jr:.3g})={
            dy_jump_min:.3f}")
324
325     if n_runs < int(MIN_PLATEAU_RUNS):
326         ok = False
327         reasons.append(f"plateau_runs={n_runs} < {MIN_PLATEAU_RUNS} (LMIN={LMIN}
            pts, max_run={max_run_pts} pts)")
328
329     if ok:
330         return True, "", dy_jump_min
331     return False, "PLATEAU_JUMP failed: " + " | ".join(reasons), dy_jump_min
332
333 # window contrast filter
334
335 def _window_contrast_ok(Vv_mV: np.ndarray, y: np.ndarray, *, side: str) -> Tuple[
    bool, str, Dict[str, float]]:
336     if WINDOW_RATIO_MIN <= 1.0:
337         dy_req = 0.0
338     else:
339         dy_req = float(math.log10(float(WINDOW_RATIO_MIN)))
340
341     absV = np.abs(Vv_mV)
```

```
    in_abs = (absV >= float(WIN_LO)) & (absV <= float(WIN_HI))
    if side == "neg":
        in_side = in_abs & (Vv_mV < 0.0)
        tag = "NEG"
    else:
        in_side = in_abs & (Vv_mV > 0.0)
        tag = "POS"

    yy = y[in_side]
    n = int(yy.size)
    if n < 2:
        return False, f"{tag} window: n={n} < 2 in |V|âĹĹ[{WIN_LO:g},{WIN_HI:g}]", {
            "n": float(n),
            "dy_req": float(dy_req),
            "dy_span": float("nan"),
            "y_min": float("nan"),
            "y_max": float("nan"),
        }

    y_min = float(np.min(yy))
    y_max = float(np.max(yy))
    dy_span = float(y_max - y_min)

    if dy_span + 1e-12 < dy_req:
        return False, f"{tag} window: ÎŤy={dy_span:.3f} < log10({WINDOW_RATIO_MIN:g})={dy_req:.3f}", {
            "n": float(n),
            "dy_req": float(dy_req),
            "dy_span": float(dy_span),
            "y_min": float(y_min),
            "y_max": float(y_max),
        }

    return True, "", {
        "n": float(n),
        "dy_req": float(dy_req),
        "dy_span": float(dy_span),
        "y_min": float(y_min),
        "y_max": float(y_max),
    }

def filter_window_contrast(iv: IV) -> Tuple[bool, str, Dict[str, float]]:
    Vv, y = build_logG_from_IV(iv)

    ok_neg, r_neg, meta_neg = _window_contrast_ok(Vv, y, side="neg")
    ok_pos, r_pos, meta_pos = _window_contrast_ok(Vv, y, side="pos")

    meta = {
        "neg_n": meta_neg.get("n", float("nan")),
        "neg_dy_span": meta_neg.get("dy_span", float("nan")),
        "pos_n": meta_pos.get("n", float("nan")),
```

```
392         "pos_dy_span": meta_pos.get("dy_span", float("nan")),
393         "dy_req": meta_neg.get("dy_req", float(math.log10(WINDOW_RATIO_MIN))) if
                WINDOW_RATIO_MIN > 1.0 else 0.0,
394     }
395
396     if ok_neg and ok_pos:
397         return True, "", meta
398
399     reasons = []
400     if not ok_neg:
401         reasons.append(r_neg)
402     if not ok_pos:
403         reasons.append(r_pos)
404     return False, "WINDOW_CONTRAST failed: " + " | ".join([x for x in reasons if x
            ]), meta
405
406 # run filters and store pass/fail
407
408 TESTS = ["GMM_BIC", "PLATEAU_JUMP", "WINDOW_CONTRAST"]
409
410 def apply_filters(iv: IV) -> Tuple[bool, List[str], Dict[str, object]]:
411     reasons: List[str] = []
412     meta: Dict[str, object] = {"test_results": {}}
413
414     if FILTER_GMM_BIC_ON:
415         ok, reason, summ = filter_gmm_bic(iv)
416         meta["test_results"]["GMM_BIC"] = bool(ok)
417         if not ok:
418             reasons.append(reason)
419         if summ is not None:
420             meta["gmm"] = summ
421     else:
422         meta["test_results"]["GMM_BIC"] = True
423
424     if FILTER_PLATEAU_JUMP_ON:
425         ok, reason, dy_jump_min = filter_plateau_jump(iv)
426         meta["test_results"]["PLATEAU_JUMP"] = bool(ok)
427         if not ok:
428             reasons.append(reason)
429         meta["dy_jump_min"] = float(dy_jump_min) if math.isfinite(float(
                dy_jump_min)) else np.nan
430     else:
431         meta["test_results"]["PLATEAU_JUMP"] = True
432
433     if FILTER_WINDOW_CONTRAST_ON:
434         ok, reason, wmeta = filter_window_contrast(iv)
435         meta["test_results"]["WINDOW_CONTRAST"] = bool(ok)
436         if not ok:
437             reasons.append(reason)
438         meta["window_meta"] = wmeta
439     else:
440         meta["test_results"]["WINDOW_CONTRAST"] = True
```

```
441
442     return (len(reasons) == 0), reasons, meta
443
444 # file collection and sampling
445
446 def collect_files() -> List[str]:
447     files = sorted(glob.glob(os.path.join(DATA_DIR, FILE_GLOB_PRIMARY)))
448     if not files:
449         all_dat = sorted(glob.glob(os.path.join(DATA_DIR, FILE_GLOB_FALLBACK)))
450         files = [f for f in all_dat if os.path.basename(f).lower().startswith("iv"
                )]
451     return [os.path.abspath(f) for f in files]
452
453 def sample_files(files: List[str]) -> List[str]:
454     if not SAMPLE_ON:
455         return files
456     pct = float(SAMPLE_PERCENT)
457     pct = max(0.0, min(100.0, pct))
458     if pct <= 0.0:
459         return []
460     if pct >= 100.0:
461         return files
462     n = len(files)
463     k = max(1, int(round(n * (pct / 100.0))))
464     rng = np.random.default_rng(int(SAMPLE_SEED))
465     idx = rng.choice(n, size=k, replace=False)
466     idx = np.sort(idx)
467     return [files[int(i)] for i in idx]
468
469 # progress UI
470
471 def build_pass_fail_lists_with_progress(
472     files: List[str],
473 ) -> Tuple[List[str], List[str], Dict[str, str], Dict[str, Optional[IV]], Dict[str
    , Dict[str, object]]]:
474     import tkinter as tk
475     from tkinter import ttk
476
477     passed: List[str] = []
478     failed: List[str] = []
479     fail_reason: Dict[str, str] = {}
480     cache: Dict[str, Optional[IV]] = {}
481     meta_cache: Dict[str, Dict[str, object]] = {}
482
483     prog = tk.Tk()
484     prog.title("Filtering progress")
485     prog.resizable(False, False)
486
487     frm = ttk.Frame(prog, padding=12)
488     frm.grid(row=0, column=0, sticky="nsew")
489
490     label = ttk.Label(frm, text="Starting...")
```

```
491     label.grid(row=0, column=0, sticky="w")
492
493     pvar = tk.DoubleVar(value=0.0)
494     bar = ttk.Progressbar(frm, length=420, mode="determinate", maximum=max(1, len(
            files)), variable=pvar)
495     bar.grid(row=1, column=0, pady=(8, 0), sticky="ew")
496
497     sub = ttk.Label(frm, text="")
498     sub.grid(row=2, column=0, sticky="w", pady=(8, 0))
499
500     prog.update_idletasks()
501
502     total = len(files)
503     for i, p in enumerate(files, start=1):
504         base = os.path.basename(p)
505         label.config(text=f"{i}/{total}: {base}")
506         pvar.set(i)
507
508         try:
509             iv = load_iv(p)
510             cache[p] = iv
511             ok, reasons, meta = apply_filters(iv)
512             meta_cache[p] = meta
513             if ok:
514                 passed.append(p)
515             else:
516                 failed.append(p)
517                 fail_reason[p] = " | ".join(reasons)
518         except Exception as e:
519             cache[p] = None
520             meta_cache[p] = {"test_results": {"GMM_BIC": False, "PLATEAU_JUMP":
                    False, "WINDOW_CONTRAST": False}}
521             failed.append(p)
522             fail_reason[p] = f"LOAD/PARSE error: {type(e).__name__}: {e}"
523
524         sub.config(text=f"PASS: {len(passed)}   FAIL: {len(failed)}")
525         prog.update_idletasks()
526         prog.update()
527
528     prog.destroy()
529     return passed, failed, fail_reason, cache, meta_cache
530
531 # diagnostics
532
533 def compute_diagnostics(files: List[str], meta_cache: Dict[str, Dict[str, object
        ]]) -> Dict[str, object]:
534     enabled_tests: List[str] = []
535     if FILTER_GMM_BIC_ON:
536         enabled_tests.append("GMM_BIC")
537     if FILTER_PLATEAU_JUMP_ON:
538         enabled_tests.append("PLATEAU_JUMP")
539     if FILTER_WINDOW_CONTRAST_ON:
```

```
        enabled_tests.append("WINDOW_CONTRAST")

    per_file: Dict[str, Dict[str, bool]] = {}
    for p in files:
        tr = (meta_cache.get(p, {}).get("test_results", {}) or {})
        per_file[p] = {t: bool(tr.get(t, True)) for t in TESTS}

    N = len(files)
    out: Dict[str, object] = {"enabled_tests": enabled_tests, "N": N}

    per_test: Dict[str, Dict[str, float]] = {}
    for t in enabled_tests:
        n_pass = sum(1 for p in files if per_file[p].get(t, True))
        per_test[t] = {"pass": n_pass, "fail": N - n_pass, "pass_pct": (100.0 *
            n_pass / N) if N else 0.0}
    out["per_test"] = per_test

    patt_counts: Dict[Tuple[int, ...], int] = {}
    for p in files:
        key = tuple(1 if per_file[p].get(t, True) else 0 for t in enabled_tests)
        patt_counts[key] = patt_counts.get(key, 0) + 1
    out["pattern_counts"] = patt_counts

    pairwise: Dict[Tuple[str, str], Dict[str, float]] = {}
    for i in range(len(enabled_tests)):
        for j in range(i + 1, len(enabled_tests)):
            a = enabled_tests[i]
            b = enabled_tests[j]
            both_pass = 0
            both_fail = 0
            for p in files:
                pa = per_file[p][a]
                pb = per_file[p][b]
                if pa and pb:
                    both_pass += 1
                elif (not pa) and (not pb):
                    both_fail += 1
            pairwise[(a, b)] = {
                "both_pass": both_pass,
                "both_fail": both_fail,
                "agree_pct": (100.0 * (both_pass + both_fail) / N) if N else 0.0,
            }
    out["pairwise"] = pairwise
    return out

def print_diagnostics(diag: Dict[str, object]) -> None:
    enabled = diag["enabled_tests"]
    N = diag["N"]
    per_test = diag["per_test"]
    patt = diag["pattern_counts"]
    pair = diag["pairwise"]

```

```
591     print("\n==== DIAGNOSTICS ====")
592     print(f"Enabled tests: {enabled}")
593     print(f"Files used: {N}")
594     print("\nPer-test pass rates:")
595     for t in enabled:
596         d = per_test[t]
597         print(f"  {t:15s} pass {d['pass']:5d} ({d['pass_pct']:.1f}%)   fail {d['
                fail']:5d} ({100.0-d['pass_pct']:.1f}%)")
598 
599     print("\nPairwise agreement (both pass + both fail):")
600     for (a, b), d in pair.items():
601         print(f"  ({a},{b})  both_pass={d['both_pass']:5d}  both_fail={d['
                both_fail']:5d}  agree={d['agree_pct']:.1f}%")
602 
603     print("\nPass/fail patterns over enabled tests (1=pass,0=fail):")
604     items = sorted(patt.items(), key=lambda kv: kv[1], reverse=True)
605     for key, cnt in items:
606         pct = (100.0 * cnt / N) if N else 0.0
607         label = " ".join(f"{enabled[i]}={'PASS' if key[i] else 'FAIL'}" for i in
                range(len(enabled)))
608         print(f"  {cnt:5d} ({pct:5.1f}%)  {label}")
609     print("====================\n")
610 
611 # viewer
612 
613 def launch_viewer(
614     files_all: List[str],
615     passed_all: List[str],
616     failed_all: List[str],
617     fail_reason: Dict[str, str],
618     cache: Dict[str, Optional[IV]],
619     meta_cache: Dict[str, Dict[str, object]],
620     diag: Dict[str, object],
621     summary_text: str,
622 ) -> None:
623     import tkinter as tk
624     from tkinter import ttk, messagebox
625 
626     import matplotlib
627     matplotlib.use("TkAgg")
628     from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
629     from matplotlib.figure import Figure
630 
631     root = tk.Tk()
632     root.title("IV Filter Viewer")
633 
634     root.columnconfigure(0, weight=0)
635     root.columnconfigure(1, weight=1)
636     root.rowconfigure(0, weight=1)
637 
638     left = ttk.Frame(root, padding=8)
639     left.grid(row=0, column=0, sticky="nsw")
```

```
640
641     main = ttk.Frame(root, padding=8)
642     main.grid(row=0, column=1, sticky="nsew")
643     main.columnconfigure(0, weight=1)
644     main.rowconfigure(0, weight=1)
645
646     ttk.Label(left, text="Show files matching:", font=("TkDefaultFont", 10, "bold"
            )).grid(row=0, column=0, sticky="w")
647
648     enabled_tests = diag.get("enabled_tests", [])
649     pass_vars: Dict[str, tk.BooleanVar] = {}
650     fail_vars: Dict[str, tk.BooleanVar] = {}
651
652     row = 1
653     for t in enabled_tests:
654         frm = ttk.Frame(left)
655         frm.grid(row=row, column=0, sticky="w", pady=(2, 2))
656         ttk.Label(frm, text=f"{t}:", width=16).grid(row=0, column=0, sticky="w")
657         pv = tk.BooleanVar(value=False)
658         fv = tk.BooleanVar(value=False)
659         pass_vars[t] = pv
660         fail_vars[t] = fv
661         ttk.Checkbutton(frm, text="PASS", variable=pv).grid(row=0, column=1, padx
                =(6, 6))
662         ttk.Checkbutton(frm, text="FAIL", variable=fv).grid(row=0, column=2, padx
                =(0, 6))
663         row += 1
664
665     ttk.Separator(left, orient="horizontal").grid(row=row, column=0, sticky="ew",
            pady=(6, 6))
666     row += 1
667
668     ttk.Label(left, text="Files").grid(row=row, column=0, sticky="w")
669     row += 1
670
671     lb = tk.Listbox(left, width=62, height=26)
672     lb.grid(row=row, column=0, sticky="nsw")
673     row += 1
674
675     btns = ttk.Frame(left)
676     btns.grid(row=row, column=0, sticky="ew", pady=(8, 0))
677     row += 1
678
679     jump = ttk.Frame(left)
680     jump.grid(row=row, column=0, sticky="ew", pady=(10, 0))
681     ttk.Label(jump, text="Jump (1..N):").grid(row=0, column=0, sticky="w")
682     entry_jump = ttk.Entry(jump, width=10)
683     entry_jump.grid(row=0, column=1, padx=(6, 6))
684     ttk.Button(jump, text="Go").grid(row=0, column=2)
685
686     plot_frame = ttk.Frame(main)
687     plot_frame.grid(row=0, column=0, sticky="nsew")
```

```
    plot_frame.columnconfigure(0, weight=1)
    plot_frame.rowconfigure(0, weight=1)

    fig = Figure(figsize=(10.0, 6.2), dpi=110)
    ax = fig.add_subplot(1, 1, 1)
    fig.tight_layout()

    canvas = FigureCanvasTkAgg(fig, master=plot_frame)
    canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")

    diag_frame = ttk.Frame(main, padding=(8, 0, 0, 0))
    diag_frame.grid(row=0, column=1, sticky="nsew")
    diag_frame.columnconfigure(0, weight=1)
    diag_frame.rowconfigure(0, weight=1)

    diag_text = tk.Text(diag_frame, width=44, height=35, wrap="none")
    diag_text.grid(row=0, column=0, sticky="nsew")
    diag_text.configure(state="disabled")

    status = ttk.Label(main, text=summary_text, anchor="w", justify="left")
    status.grid(row=1, column=0, columnspan=2, sticky="ew", pady=(6, 0))

    current = {"idx": max(0, int(START_INDEX))}
    visible_files: List[str] = []

    def file_label(path: str) -> str:
        base = os.path.basename(path)
        return f"[FAIL] {base}" if path in fail_reason else f"[PASS] {base}"

    def get_test_result(path: str, test: str) -> Optional[bool]:
        tr = meta_cache.get(path, {}).get("test_results", {})
        if test not in tr:
            return None
        return bool(tr[test])

    def matches_checkbox_filter(path: str) -> bool:
        for t in enabled_tests:
            want_pass = bool(pass_vars[t].get())
            want_fail = bool(fail_vars[t].get())
            if (want_pass and want_fail) or ((not want_pass) and (not want_fail)):
                continue
            res = get_test_result(path, t)
            if res is None:
                return False
            if want_pass and (not res):
                return False
            if want_fail and res:
                return False
        return True

    def rebuild_visible_list(keep_path: Optional[str] = None) -> None:
        nonlocal visible_files
```

```
        visible_files = [p for p in files_all if matches_checkbox_filter(p)]
        lb.delete(0, tk.END)
        for p in visible_files:
            lb.insert(tk.END, file_label(p))

        if not visible_files:
            current["idx"] = 0
            return

        if keep_path is not None and keep_path in visible_files:
            current["idx"] = visible_files.index(keep_path)
        else:
            current["idx"] = max(0, min(current["idx"], len(visible_files) - 1))

        lb.selection_clear(0, tk.END)
        lb.selection_set(current["idx"])
        lb.see(current["idx"])

    def draw_dyflat_reference_lines(Vv: np.ndarray, y: np.ndarray) -> None:
        if Vv.size < 2:
            return
        dV = np.diff(Vv)
        dVabs = np.abs(dV[np.isfinite(dV)])
        if dVabs.size == 0:
            return
        dV_med = float(np.median(dVabs))
        if dV_med <= 0:
            return

        slope_dex_per_mV = float(DY_FLAT) / dV_med
        x0 = float(np.median(Vv))
        y0 = float(np.median(y))
        x_min = float(np.min(Vv))
        x_max = float(np.max(Vv))
        xx = np.array([x_min, x_max], dtype=float)

        yy_pos = y0 + slope_dex_per_mV * (xx - x0)
        yy_neg = y0 - slope_dex_per_mV * (xx - x0)

        ax.plot(xx, yy_pos, linestyle="--", linewidth=1.6)
        ax.plot(xx, yy_neg, linestyle="--", linewidth=1.6)

        ax.text(
            0.02, 0.98,
            f"DY_FLAT={DY_FLAT:.3f} dex/step | median|dV|={dV_med:.3g} mV => ~{
                slope_dex_per_mV:.3g} dex/mV",
            transform=ax.transAxes,
            va="top",
            fontsize=9,
        )

    def draw_windows_and_errorbars(Vv: np.ndarray, y: np.ndarray) -> None:
```

```
791         ax.axvspan(-WIN_HI, -WIN_LO, alpha=0.10)
792         ax.axvspan(WIN_LO, WIN_HI, alpha=0.10)
793 
794         if WINDOW_RATIO_MIN <= 1.0:
795             return
796         dy_req = float(math.log10(float(WINDOW_RATIO_MIN)))
797 
798         absV = np.abs(Vv)
799         in_abs = (absV >= float(WIN_LO)) & (absV <= float(WIN_HI))
800 
801         for side_mask in [in_abs & (Vv < 0), in_abs & (Vv > 0)]:
802             idx = np.where(side_mask)[0]
803             if idx.size == 0:
804                 continue
805             k = min(12, int(idx.size))
806             pick = np.linspace(0, idx.size - 1, k).round().astype(int)
807             idx_pick = idx[pick]
808             ax.errorbar(
809                 Vv[idx_pick],
810                 y[idx_pick],
811                 yerr=dy_req,
812                 fmt="none",
813                 elinewidth=1.0,
814                 capsize=2,
815             )
816 
817         ax.text(
818             0.02, 0.90,
819             f"Window: |V|âĹĹ[{WIN_LO:g},{WIN_HI:g}] mV, require ratioâĽě{
                WINDOW_RATIO_MIN:g} (ÎŤyâĽě{dy_req:.3f} dex) in BOTH windows",
820             transform=ax.transAxes,
821             va="top",
822             fontsize=9,
823         )
824 
825     def format_right_panel(current_path: Optional[str]) -> str:
826         enabled = diag["enabled_tests"]
827         N = diag["N"]
828         per_test = diag["per_test"]
829         pair = diag["pairwise"]
830         patt = diag["pattern_counts"]
831 
832         lines = []
833         lines.append("==== DIAGNOSTICS ====")
834         lines.append(f"Files used: {N}")
835         lines.append("")
836         lines.append("Per-test pass rates:")
837         for t in enabled:
838             d = per_test[t]
839             lines.append(f"  {t:15s}: PASS {d['pass']:5d} ({d['pass_pct']:.1f}%) |
                FAIL {d['fail']:5d} ({100.0-d['pass_pct']:.1f}%)")
840         lines.append("")
```

```
841         lines.append("Pairwise agreement (both pass + both fail):")
842         for (a, b), d in pair.items():
843             lines.append(f"  ({a},{b}) agree={d['agree_pct']:.1f}%  both_pass={d['
                both_pass']}  both_fail={d['both_fail']}")
844         lines.append("")
845         lines.append("Pass/fail patterns (enabled tests):")
846         items = sorted(patt.items(), key=lambda kv: kv[1], reverse=True)[:12]
847         for key, cnt in items:
848             pct = (100.0 * cnt / N) if N else 0.0
849             label = ", ".join(f"{enabled[i]}={'PASS' if key[i] else 'FAIL'}" for i
                    in range(len(enabled)))
850             lines.append(f"  {cnt:5d} ({pct:5.1f}%)  {label}")
851 
852         lines.append("")
853         lines.append("==== CURRENT VIEW ====")
854         lines.append(f"Visible files: {len(visible_files)}")
855         constraints = []
856         for t in enabled_tests:
857             wp = bool(pass_vars[t].get())
858             wf = bool(fail_vars[t].get())
859             if (wp and wf) or ((not wp) and (not wf)):
860                 constraints.append(f"{t}: any")
861             elif wp:
862                 constraints.append(f"{t}: PASS")
863             else:
864                 constraints.append(f"{t}: FAIL")
865         lines.append("Constraints:")
866         for c in constraints:
867             lines.append(f"  {c}")
868 
869         if visible_files:
870             lines.append("")
871             lines.append("Among visible files:")
872             for t in enabled_tests:
873                 n_pass = sum(1 for p in visible_files if get_test_result(p, t) is
                        True)
874                 lines.append(f"  {t:15s}: PASS {100.0*n_pass/len(visible_files):.1
                        f}%")
875 
876         if current_path is not None:
877             lines.append("")
878             lines.append("==== CURRENT FILE ====")
879             base = os.path.basename(current_path)
880             lines.append(base)
881             tr = meta_cache.get(current_path, {}).get("test_results", {})
882             for t in enabled_tests:
883                 if t in tr:
884                     lines.append(f"  {t:15s}: {'PASS' if tr[t] else 'FAIL'}")
885             if current_path in fail_reason:
886                 lines.append("")
887                 lines.append("Fail reasons:")
888                 lines.append(fail_reason[current_path])
```

```
889
890         return "\n".join(lines)
891
892     def set_diag_text(s: str) -> None:
893         diag_text.configure(state="normal")
894         diag_text.delete("1.0", tk.END)
895         diag_text.insert(tk.END, s)
896         diag_text.configure(state="disabled")
897
898     def redraw() -> None:
899         ax.clear()
900         if not visible_files:
901             ax.set_title("No files match the selected PASS/FAIL checkbox
                constraints.")
902             canvas.draw_idle()
903             set_diag_text(format_right_panel(None))
904             return
905
906         idx = max(0, min(current["idx"], len(visible_files) - 1))
907         current["idx"] = idx
908         path = visible_files[idx]
909         base = os.path.basename(path)
910
911         iv = cache.get(path, None)
912         if iv is None:
913             ax.set_title(f"{base} (load failed)")
914             ax.text(0.02, 0.5, fail_reason.get(path, "Unknown error"), transform=
                ax.transAxes)
915             canvas.draw_idle()
916             set_diag_text(format_right_panel(path))
917             return
918
919         Vv, y = build_logG_from_IV(iv)
920         if y.size == 0:
921             ax.set_title(f"{base} (no data after |V| >= {V_CUTOFF:g} mV)")
922             canvas.draw_idle()
923             set_diag_text(format_right_panel(path))
924             return
925
926         ax.plot(Vv, y, linewidth=1.0)
927         ax.scatter(Vv, y, s=10)
928
929         if FILTER_PLATEAU_JUMP_ON:
930             draw_dyflat_reference_lines(Vv, y)
931         if FILTER_WINDOW_CONTRAST_ON:
932             draw_windows_and_errorbars(Vv, y)
933
934         meta = meta_cache.get(path, {})
935         tr = meta.get("test_results", {})
936         gmm = meta.get("gmm", None)
937
938         title_parts = [base, f"N={len(y)}", f"|V|cut={V_CUTOFF:g}mV"]
```

```
939
940         if FILTER_GMM_BIC_ON and isinstance(gmm, GMMSummary) and gmm.ok:
941             title_parts += [f"ratio={gmm.ratio:.2f}", f"BIC1={gmm.bic1:.1f}", f"
                    BIC2={gmm.bic2:.1f}", f"ΔTBIC={gmm.dbic:.1f}"]
942
943         for t in enabled_tests:
944             if t in tr:
945                 title_parts.append(f"{t}:{'PASS' if tr[t] else 'FAIL'}")
946
947         ax.set_title(" | ".join(title_parts))
948         ax.set_xlabel("V (mV)")
949         ax.set_ylabel("log10(|G|)")
950
951         if path in fail_reason:
952             ax.text(0.02, 0.02, fail_reason[path], transform=ax.transAxes,
                    fontsize=9, va="bottom")
953
954         canvas.draw_idle()
955
956         lb.selection_clear(0, tk.END)
957         lb.selection_set(idx)
958         lb.see(idx)
959
960         set_diag_text(format_right_panel(path))
961
962     def on_prev():
963         current["idx"] -= 1
964         redraw()
965
966     def on_next():
967         current["idx"] += 1
968         redraw()
969
970     def on_list_select(_event):
971         sel = lb.curselection()
972         if not sel:
973             return
974         current["idx"] = int(sel[0])
975         redraw()
976
977     lb.bind("<<ListboxSelect>>", on_list_select)
978
979     ttk.Button(btns, text="Prev", command=on_prev).grid(row=0, column=0, padx=(0,
        6))
980     ttk.Button(btns, text="Next", command=on_next).grid(row=0, column=1)
981
982     def on_jump():
983         try:
984             i = int(entry_jump.get()) - 1
985         except Exception:
986             messagebox.showerror("Error", "Enter an integer index (1-based).")
987             return
```

```
            current["idx"] = i
            redraw()

    for child in jump.winfo_children():
        if isinstance(child, ttk.Button) and child.cget("text") == "Go":
            child.configure(command=on_jump)

    def on_constraints_change(*_):
        keep_path = None
        if visible_files:
            idx0 = max(0, min(current["idx"], len(visible_files) - 1))
            keep_path = visible_files[idx0]
        rebuild_visible_list(keep_path=keep_path)
        redraw()

    for t in enabled_tests:
        pass_vars[t].trace_add("write", on_constraints_change)
        fail_vars[t].trace_add("write", on_constraints_change)

    rebuild_visible_list()
    redraw()
    root.mainloop()

# write manifest

def write_manifest(
    *,
    files_used: List[str],
    passed: List[str],
    fail_reason: Dict[str, str],
    meta_cache: Dict[str, Dict[str, object]],
    out_dir: str,
    basename: str,
) -> Tuple[str, str]:
    import csv
    import json
    from datetime import datetime

    os.makedirs(out_dir, exist_ok=True)
    passed_set = set(passed)

    csv_path = os.path.join(out_dir, f"{basename}.csv")
    json_path = os.path.join(out_dir, f"{basename}.json")

    fieldnames = [
        "file_abs",
        "file_base",
        "passed_all",
        "fail_reason",
        "GMM_BIC",
        "PLATEAU_JUMP",
        "WINDOW_CONTRAST",
```

```
        "gmm_ratio",
        "gmm_dbic",
        "gmm_w1",
        "gmm_w2",
        "window_neg_n",
        "window_pos_n",
        "window_neg_dy_span",
        "window_pos_dy_span",
        "window_dy_req",
        "dy_jump_min",
    ]

    with open(csv_path, "w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=fieldnames)
        w.writeheader()

        for p in files_used:
            base = os.path.basename(p)
            meta = meta_cache.get(p, {}) or {}
            tr = meta.get("test_results", {}) or {}

            passed_all = (p in passed_set)

            gmm = meta.get("gmm", None)
            gmm_ratio = ""
            gmm_dbic = ""
            gmm_w1 = ""
            gmm_w2 = ""
            try:
                if gmm is not None and getattr(gmm, "ok", False):
                    gmm_ratio = getattr(gmm, "ratio", "")
                    gmm_dbic = getattr(gmm, "dbic", "")
                    gmm_w1 = getattr(gmm, "w1", "")
                    gmm_w2 = getattr(gmm, "w2", "")
            except Exception:
                pass

            wmeta = meta.get("window_meta", {}) or {}
            row = {
                "file_abs": p,
                "file_base": base,
                "passed_all": int(bool(passed_all)),
                "fail_reason": fail_reason.get(p, ""),
                "GMM_BIC": int(bool(tr.get("GMM_BIC", True))),
                "PLATEAU_JUMP": int(bool(tr.get("PLATEAU_JUMP", True))),
                "WINDOW_CONTRAST": int(bool(tr.get("WINDOW_CONTRAST", True))),
                "gmm_ratio": gmm_ratio,
                "gmm_dbic": gmm_dbic,
                "gmm_w1": gmm_w1,
                "gmm_w2": gmm_w2,
                "window_neg_n": wmeta.get("neg_n", ""),
                "window_pos_n": wmeta.get("pos_n", ""),
```

```
                "window_neg_dy_span": wmeta.get("neg_dy_span", ""),
                "window_pos_dy_span": wmeta.get("pos_dy_span", ""),
                "window_dy_req": wmeta.get("dy_req", ""),
                "dy_jump_min": meta.get("dy_jump_min", ""),
            }
            w.writerow(row)

    run = {
        "created_at": datetime.now().isoformat(timespec="seconds"),
        "data_dir": DATA_DIR,
        "n_files_used": len(files_used),
        "n_pass": len(passed),
        "n_fail": len(files_used) - len(passed),
        "sampling": {"SAMPLE_ON": SAMPLE_ON, "SAMPLE_PERCENT": SAMPLE_PERCENT, "
            SAMPLE_SEED": int(SAMPLE_SEED)},
        "filters_on": {
            "FILTER_GMM_BIC_ON": FILTER_GMM_BIC_ON,
            "FILTER_PLATEAU_JUMP_ON": FILTER_PLATEAU_JUMP_ON,
            "FILTER_WINDOW_CONTRAST_ON": FILTER_WINDOW_CONTRAST_ON,
        },
        "constants": {
            "V_CUTOFF_mV": V_CUTOFF,
            "EPS_G": EPS_G,
            "GMM_MIN_RATIO": GMM_MIN_RATIO,
            "GMM_MIN_WEIGHT": GMM_MIN_WEIGHT,
            "GMM_BIC_IMPROVEMENT": GMM_BIC_IMPROVEMENT,
            "GMM_MIN_N": GMM_MIN_N,
            "DY_FLAT": DY_FLAT,
            "P_FLAT_MIN": P_FLAT_MIN,
            "JUMP_RATIO_MIN": JUMP_RATIO_MIN,
            "LMIN": LMIN,
            "MIN_PLATEAU_RUNS": MIN_PLATEAU_RUNS,
            "WIN_LO": WIN_LO,
            "WIN_HI": WIN_HI,
            "WINDOW_RATIO_MIN": WINDOW_RATIO_MIN,
        },
        "outputs": {"csv": os.path.abspath(csv_path), "json": os.path.abspath(
            json_path)},
    }

    with open(json_path, "w", encoding="utf-8") as f:
        json.dump(run, f, indent=2)

    return csv_path, json_path

# main

def main() -> None:
    files_all_full = collect_files()
    if not files_all_full:
        raise SystemExit(f"No IV files found in: {DATA_DIR}")

```

```
1142    files_used = sample_files(files_all_full)
1143
1144    passed, failed, fail_reason, cache, meta_cache =
            build_pass_fail_lists_with_progress(files_used)
1145
1146    total_full = len(files_all_full)
1147    total_used = len(files_used)
1148    n_pass = len(passed)
1149    n_fail = len(failed)
1150    pct = (100.0 * n_pass / total_used) if total_used > 0 else 0.0
1151
1152    print("==== FILTER SUMMARY ====")
1153    print(f"Folder: {DATA_DIR}")
1154    print(f"Total files in folder: {total_full}")
1155    if SAMPLE_ON:
1156        print(f"Sampling: ON ({SAMPLE_PERCENT:.1f}% => {total_used}/{total_full},
                seed={SAMPLE_SEED})")
1157    else:
1158        print(f"Sampling: OFF (using all {total_used}/{total_full})")
1159    print(f"Used:   {total_used}")
1160    print(f"Passed: {n_pass} ({pct:.1f}%)")
1161    print(f"Failed: {n_fail} ({100.0 - pct:.1f}%)")
1162    print("========================")
1163
1164    diag = compute_diagnostics(files_used, meta_cache)
1165    print_diagnostics(diag)
1166
1167    if WRITE_MANIFEST:
1168        csv_path, json_path = write_manifest(
1169            files_used=files_used,
1170            passed=passed,
1171            fail_reason=fail_reason,
1172            meta_cache=meta_cache,
1173            out_dir=DATA_DIR,
1174            basename=MANIFEST_BASENAME,
1175        )
1176        print(f"\nManifest written:\n  {csv_path}\n  {json_path}\n")
1177
1178    summary_text = (
1179        f"Folder total: {total_full} | Used: {total_used} | Passed(all enabled): {
                n_pass} ({pct:.1f}%) | "
1180        f"Failed: {n_fail} ({100.0 - pct:.1f}%)\n"
1181        f"Sampling: {'ON' if SAMPLE_ON else 'OFF'} ({SAMPLE_PERCENT:.1f}% seed={
                SAMPLE_SEED})\n"
1182        f"Conductance computed via IV.calculate_conductance(); y=log10(|G|) after
                |V| >= {V_CUTOFF:g} mV."
1183    )
1184
1185    launch_viewer(
1186        files_all=files_used,
1187        passed_all=passed,
1188        failed_all=failed,
```

```
            fail_reason=fail_reason,
            cache=cache,
            meta_cache=meta_cache,
            diag=diag,
            summary_text=summary_text,
        )

if __name__ == "__main__":
    main()
```

Listing A.1: Filtering script

A.2 SLR-HMM

```python
#!/usr/bin/env python3
from __future__ import annotations

import os
import csv
import json
import math
import threading
from dataclasses import dataclass
from typing import Dict, List, Optional, Tuple

import numpy as np
import sys
from pathlib import Path

# paths
REPO_ROOT = Path(__file__).resolve().parents[2]
SRC_ROOT = REPO_ROOT / "src"
if str(SRC_ROOT) not in sys.path:
    sys.path.insert(0, str(SRC_ROOT))

# settings

DATA_DIR = r"C:\Users\maxdo\nn_clustering\data\raw\12_MSH0501"
FILTER_MANIFEST_NAME = "filter_manifest.csv"
ONLY_USE_PASSED_ALL = True

V_CUTOFF_mV = 30.0
EPS_G = 1e-20

GMM_N_INIT = 50
GMM_MAX_ITER = 5000
GMM_REG_COVAR = 1e-6
GMM_RANDOM_STATE = 0

RANSAC_RESIDUAL_THRESHOLD = 4.0      # ransac threshold
RANSAC_GLOBAL2_MAX_ITER = 25         # alternating assignment iterations
RANSAC_MIN_SAMPLES_FRAC = 0.5
RANSAC_RANDOM_STATE = 0

HMM_N_ITER = 40                      # EM iterations for refining line params
A_STICKY = 0.995                     # stickiness for state transitions
MIN_SIGMA2 = 1e-5                    # variance floor

USE_VITERBI_DECODE = True            # True = viterbi; False = argmax posterior

UPDATE_LINES_WITH_EM = False          # True = update (a,b,sigma) with EM; False = fixed RANSAC + HMM smoothing only

PRECOMPUTE_IN_BACKGROUND = True
```

```
50
51  SAVE_PSEUDOLABELS = True
52  PSEUDOLABEL_DIRNAME = "pseudolabels_slr_hmm_2_state"
53  PSEUDOLABEL_TAG = "slr_hmm_ransac_global2"
54
55  # plot settings
56  POINT_SIZE = 14
57  LINE_WIDTH = 0.8
58
59  INTERSECTION_WINDOW_mV = 1100.0
60
61  # iv import
62  try:
63      from mechanical_switches.utils import IV  # type: ignore
64  except Exception:
65      from iv import IV  # type: ignore
66
67  # data structures
68  @dataclass
69  class IVSeries:
70      Vv_mV: np.ndarray
71      Vv_V: np.ndarray     # V
72      x: np.ndarray        # log10(|G|)
73      G: np.ndarray        # conductance (for saving/debug)
74
75  def _parse_iv_file(path: str) -> Tuple[np.ndarray, np.ndarray]:
76      arr = np.loadtxt(path, dtype=float)
77      if arr.ndim == 1 or arr.shape[1] < 2:
78          raise ValueError(f"Bad format (need >=2 cols): {path}")
79      return np.asarray(arr[:, 0], float), np.asarray(arr[:, 1], float)
80
81  def build_series_with_IV(path: str) -> Tuple[IV, IVSeries]:
82      V_mV_raw, I = _parse_iv_file(path)
83
84      iv_obj = IV(
85          file=os.path.basename(path),
86          date="",
87          trace_id=0,
88          id=0,
89          vbias=V_mV_raw * 1e-3,
90          current=I,
91          motor_position=0,
92          settings={},
93      )
94
95  # requirement: use calculate_conductance(..., offsetted=true)
96      G = iv_obj.calculate_conductance(zero_threshold=(V_CUTOFF_mV * 1e-3),
            offsetted=True)[0]
97      G = np.asarray(G, float)
98
99      Vv_V = np.asarray(iv_obj.conductance_bias, float)
100     Vv_mV = Vv_V * 1e3
```

```
101
102     n = min(Vv_mV.size, G.size)
103     Vv_mV = Vv_mV[:n]
104     Vv_V = Vv_V[:n]
105     G = G[:n]
106
107     x = np.log10(np.maximum(np.abs(G), float(EPS_G)))
108
109     m = np.isfinite(Vv_V) & np.isfinite(x) & np.isfinite(G)
110     Vv_mV = Vv_mV[m]
111     Vv_V = Vv_V[m]
112     x = x[m]
113     G = G[m]
114
115     return iv_obj, IVSeries(Vv_mV=Vv_mV, Vv_V=Vv_V, x=x, G=G)
116
117 def read_filter_manifest(manifest_path: str) -> List[Dict[str, str]]:
118     rows: List[Dict[str, str]] = []
119     with open(manifest_path, "r", encoding="utf-8") as f:
120         for r in csv.DictReader(f):
121             rows.append(r)
122     return rows
123
124 def select_files_from_manifest(rows: List[Dict[str, str]]) -> List[str]:
125     files: List[str] = []
126     for r in rows:
127         p = (r.get("file_abs", "") or "").strip()
128         if not p:
129             continue
130         try:
131             passed_all = int(float(r.get("passed_all", "0")))
132         except Exception:
133             passed_all = 0
134         if ONLY_USE_PASSED_ALL and passed_all != 1:
135             continue
136         files.append(os.path.abspath(p))
137     return files
138
139 # gmm init
140 def fit_gmm_baseline_1d(x: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
141     x = np.asarray(x, float)
142     m = np.isfinite(x)
143     x2 = x[m]
144     if x2.size < 6:
145         thr = float(np.nanmedian(x2)) if x2.size else 0.0
146         z = (x >= thr).astype(int)
147         mu0 = float(np.nanmean(x[z == 0])) if np.any(z == 0) else float("nan")
148         mu1 = float(np.nanmean(x[z == 1])) if np.any(z == 1) else float("nan")
149         if math.isfinite(mu0) and math.isfinite(mu1) and mu0 > mu1:
150             z = 1 - z
151             mu0, mu1 = mu1, mu0
152         return z.astype(int), np.array([mu0, mu1], float)
```

```
153
154     from sklearn.mixture import GaussianMixture
155     X = x2.reshape(-1, 1)
156     gmm = GaussianMixture(
157         n_components=2,
158         n_init=int(GMM_N_INIT),
159         max_iter=int(GMM_MAX_ITER),
160         reg_covar=float(GMM_REG_COVAR),
161         random_state=int(GMM_RANDOM_STATE),
162     )
163     gmm.fit(X)
164     z_valid = gmm.predict(X).astype(int)
165     means = gmm.means_.reshape(-1).astype(float)
166
167     order = np.argsort(means)  # low mean -> label 0
168     means = means[order]
169     z_valid = np.array([np.where(order == zi)[0][0] for zi in z_valid], dtype=int)
170
171     z = np.zeros_like(x, dtype=int)
172     z[m] = z_valid
173     z[~m] = int(np.nanmedian(z_valid)) if z_valid.size else 0
174     return z, means
175
176 # ransac line
177 def fit_line_ransac(Vseg: np.ndarray, yseg: np.ndarray, residual_threshold: float)
        -> Tuple[float, float]:
178     try:
179         from sklearn.linear_model import RANSACRegressor, LinearRegression
180     except Exception as e:
181         raise RuntimeError("scikit-learn is required for RANSAC. Install sklearn."
              ) from e
182
183     Vseg = np.asarray(Vseg, float)
184     yseg = np.asarray(yseg, float)
185     X = Vseg.reshape(-1, 1)
186
187     min_samples = max(10, int(RANSAC_MIN_SAMPLES_FRAC * len(Vseg)))
188     model = RANSACRegressor(
189         estimator=LinearRegression(),
190         min_samples=min_samples,
191         residual_threshold=float(residual_threshold),
192         random_state=int(RANSAC_RANDOM_STATE),
193     )
194     model.fit(X, yseg)
195     a = float(model.estimator_.coef_[0])
196     b = float(model.estimator_.intercept_)
197     return a, b
198
199 def global2_ransac_lines(V: np.ndarray, x: np.ndarray, residual_threshold: float,
200                          max_iter: int = 25, z0: Optional[np.ndarray] = None) ->
                             Tuple[np.ndarray, Tuple[float, float], Tuple[float,
                             float]]:
```

```
    V = np.asarray(V, float)
    x = np.asarray(x, float)
    n = V.size

    if z0 is None:
        z = (x >= np.nanmedian(x)).astype(int)
    else:
        z = np.asarray(z0, int).copy()
        if z.size != n:
            z = (x >= np.nanmedian(x)).astype(int)

    if np.all(z == 0) or np.all(z == 1):
        z = (x >= np.nanmedian(x)).astype(int)

    a0 = b0 = a1 = b1 = 0.0

    for _ in range(int(max_iter)):
        z_prev = z.copy()

        m0 = (z == 0)
        if m0.sum() >= 10:
            a0, b0 = fit_line_ransac(V[m0], x[m0], residual_threshold)
        else:
            a0, b0 = np.polyfit(V, x, 1)

        m1 = (z == 1)
        if m1.sum() >= 10:
            a1, b1 = fit_line_ransac(V[m1], x[m1], residual_threshold)
        else:
            a1, b1 = np.polyfit(V, x, 1)

        r0 = np.abs(x - (a0 * V + b0))
        r1 = np.abs(x - (a1 * V + b1))
        z = (r1 < r0).astype(int)

        if np.array_equal(z, z_prev):
            break

    mu0 = float(np.median(x[z == 0])) if np.any(z == 0) else float("inf")
    mu1 = float(np.median(x[z == 1])) if np.any(z == 1) else float("inf")
    if mu0 > mu1:
        z = 1 - z
        (a0, b0, a1, b1) = (a1, b1, a0, b0)

    return z.astype(int), (float(a0), float(b0)), (float(a1), float(b1))

# intersection check
def ransac_lines_intersect_in_domain_mV(a0: float, b0: float, a1: float, b1: float,
                                        domain_mV: float) -> Tuple[bool, float]:
    # Lines are in x = a*V + b, with V in Volts
    da = float(a0) - float(a1)
```

```
252     if abs(da) < 1e-12:
253         return False, float("nan")
254     V_star = (float(b1) - float(b0)) / da
255     V_star_mV = 1e3 * V_star
256     inside = (V_star_mV >= -float(domain_mV)) and (V_star_mV <= float(domain_mV))
257     return bool(inside), float(V_star_mV)
258
259 # slr-hmm
260 def _viterbi_decode(logB: np.ndarray, A: np.ndarray, startprob: np.ndarray) -> np.
        ndarray:
261     T, K = logB.shape
262     logA = np.log(A + 1e-300)
263     logpi = np.log(startprob + 1e-300)
264
265     delta = np.zeros((T, K), float)
266     psi = np.zeros((T, K), int)
267
268     delta[0] = logpi + logB[0]
269     psi[0] = 0
270
271     for t in range(1, T):
272         for j in range(K):
273             vals = delta[t - 1] + logA[:, j]
274             psi[t, j] = int(np.argmax(vals))
275             delta[t, j] = float(np.max(vals) + logB[t, j])
276
277     z = np.zeros(T, int)
278     z[-1] = int(np.argmax(delta[-1]))
279     for t in range(T - 2, -1, -1):
280         z[t] = psi[t + 1, z[t + 1]]
281     return z
282
283 def _forward_backward(logB: np.ndarray, A: np.ndarray, startprob: np.ndarray) ->
        Tuple[np.ndarray, float]:
284     T, K = logB.shape
285     logA = np.log(A + 1e-300)
286     logpi = np.log(startprob + 1e-300)
287
288     alpha = np.zeros((T, K), float)
289     beta = np.zeros((T, K), float)
290
291     alpha[0] = logpi + logB[0]
292     for t in range(1, T):
293         for j in range(K):
294             alpha[t, j] = logB[t, j] + np.logaddexp.reduce(alpha[t - 1] + logA[:,
                    j])
295
296     ll = float(np.logaddexp.reduce(alpha[-1]))
297
298     beta[-1] = 0.0
299     for t in range(T - 2, -1, -1):
300         for i in range(K):
```

```
            beta[t, i] = np.logaddexp.reduce(logA[i, :] + logB[t + 1, :] + beta[t
                + 1, :])

    loggamma = alpha + beta
    loggamma = loggamma - np.logaddexp.reduce(loggamma, axis=1, keepdims=True)
    gamma = np.exp(loggamma)
    return gamma, ll

def _weighted_linreg(V: np.ndarray, x: np.ndarray, w: np.ndarray) -> Tuple[float,
    float]:
    V = np.asarray(V, float)
    x = np.asarray(x, float)
    w = np.asarray(w, float)
    w = np.maximum(w, 0.0)

    sw = float(np.sum(w)) + 1e-12
    Vbar = float(np.sum(w * V) / sw)
    xbar = float(np.sum(w * x) / sw)

    num = float(np.sum(w * (V - Vbar) * (x - xbar)))
    den = float(np.sum(w * (V - Vbar) ** 2)) + 1e-12

    a = num / den
    b = xbar - a * Vbar
    return float(a), float(b)

def slr_hmm_fit_predict(
    V: np.ndarray,
    x: np.ndarray,
    a0b0: Tuple[float, float],
    a1b1: Tuple[float, float],
    z_init: np.ndarray,
    *,
    update_lines_with_em_override: Optional[bool] = None,
) -> Tuple[np.ndarray, Dict[str, float], np.ndarray]:
    # allows per-IV override of UPDATE_LINES_WITH_EM
    V = np.asarray(V, float)
    x = np.asarray(x, float)

# decide em usage for this iv
    update_lines = UPDATE_LINES_WITH_EM if update_lines_with_em_override is None
        else bool(update_lines_with_em_override)

    a = float(A_STICKY)
    A = np.array([[a, 1.0 - a], [1.0 - a, a]], float)

    z_init = np.asarray(z_init, int)
    p0 = float(np.mean(z_init == 0))
    p1 = 1.0 - p0
    startprob = np.array([max(p0, 1e-3), max(p1, 1e-3)], float)
    startprob /= float(np.sum(startprob))

```

```
    a0, b0 = map(float, a0b0)
    a1, b1 = map(float, a1b1)

    def _sigma2_from_label(k: int) -> float:
        mk = (z_init == k)
        if mk.sum() < 5:
            r = x - (a0 * V + b0) if k == 0 else x - (a1 * V + b1)
            return float(max(np.var(r), MIN_SIGMA2))
        r = x[mk] - ((a0 * V[mk] + b0) if k == 0 else (a1 * V[mk] + b1))
        return float(max(np.mean(r * r), MIN_SIGMA2))

    sig20 = _sigma2_from_label(0)
    sig21 = _sigma2_from_label(1)

    def _logB() -> np.ndarray:
        r0 = x - (a0 * V + b0)
        r1 = x - (a1 * V + b1)
        s0 = float(max(sig20, MIN_SIGMA2))
        s1 = float(max(sig21, MIN_SIGMA2))
        lb0 = -0.5 * (r0 * r0 / s0 + np.log(s0))
        lb1 = -0.5 * (r1 * r1 / s1 + np.log(s1))
        return np.column_stack([lb0, lb1])

    if not update_lines:
        logB = _logB()
        gamma, _ll = _forward_backward(logB, A, startprob)
        z = _viterbi_decode(logB, A, startprob) if USE_VITERBI_DECODE else np.
            argmax(gamma, axis=1).astype(int)
        params = dict(a0=a0, b0=b0, a1=a1, b1=b1, sig20=sig20, sig21=sig21)
        return z, params, gamma

# em refine
    last_ll = -1e300
    for _ in range(int(HMM_N_ITER)):
        logB = _logB()
        gamma, ll = _forward_backward(logB, A, startprob)

        w0 = gamma[:, 0]
        w1 = gamma[:, 1]
        a0, b0 = _weighted_linreg(V, x, w0)
        a1, b1 = _weighted_linreg(V, x, w1)

        r0 = x - (a0 * V + b0)
        r1 = x - (a1 * V + b1)
        sig20 = float(max(np.sum(w0 * r0 * r0) / (float(np.sum(w0)) + 1e-12),
            MIN_SIGMA2))
        sig21 = float(max(np.sum(w1 * r1 * r1) / (float(np.sum(w1)) + 1e-12),
            MIN_SIGMA2))

        if abs(ll - last_ll) < 1e-6 * (1.0 + abs(last_ll)):
            break
        last_ll = ll
```

```
399
400     logB = _logB()
401     gamma, _ll = _forward_backward(logB, A, startprob)
402     z = _viterbi_decode(logB, A, startprob) if USE_VITERBI_DECODE else np.argmax(
            gamma, axis=1).astype(int)
403
404     mu0 = float(np.median(x[z == 0])) if np.any(z == 0) else float("inf")
405     mu1 = float(np.median(x[z == 1])) if np.any(z == 1) else float("inf")
406     if mu0 > mu1:
407         z = 1 - z
408         gamma = gamma[:, ::-1]
409         (a0, b0, a1, b1) = (a1, b1, a0, b0)
410         (sig20, sig21) = (sig21, sig20)
411
412     params = dict(a0=a0, b0=b0, a1=a1, b1=b1, sig20=sig20, sig21=sig21)
413     return z.astype(int), params, gamma
414
415 # save labels
416 def _ensure_pseudolabel_dir() -> str:
417     out_dir = os.path.join(DATA_DIR, PSEUDOLABEL_DIRNAME)
418     os.makedirs(out_dir, exist_ok=True)
419     return out_dir
420
421 def save_npz(iv_path: str, series: IVSeries,
422              z_gmm: np.ndarray, z_ransac: np.ndarray, z_hmm: np.ndarray,
423              lines_ransac: Dict[str, float], lines_hmm: Dict[str, float],
424              settings: Dict) -> None:
425     out_dir = _ensure_pseudolabel_dir()
426     base = os.path.basename(iv_path)
427     stem = os.path.splitext(base)[0]
428     out_path = os.path.join(out_dir, f"{stem}.npz")
429
430     np.savez_compressed(
431         out_path,
432         iv_file=base,
433         V_mV=series.Vv_mV.astype(float),
434         V_V=series.Vv_V.astype(float),
435         x=series.x.astype(float),
436         G=series.G.astype(float),
437         z_gmm=z_gmm.astype(int),
438         z_ransac=z_ransac.astype(int),
439         z_slr_hmm=z_hmm.astype(int),
440         ransac_lines=json.dumps(lines_ransac, indent=2),
441         hmm_lines=json.dumps(lines_hmm, indent=2),
442         settings=json.dumps(settings, indent=2),
443     )
444
445 # cache
446 @dataclass
447 class CacheEntry:
448     ok: bool
449     err: str
```

```
    series: Optional[IVSeries]
    z_gmm: Optional[np.ndarray]
    means_gmm: Optional[np.ndarray]
    z_ransac: Optional[np.ndarray]
    ransac_line0: Optional[Tuple[float, float]]
    ransac_line1: Optional[Tuple[float, float]]
    z_hmm: Optional[np.ndarray]
    hmm_params: Optional[Dict[str, float]]
    gamma: Optional[np.ndarray]
    saved_npz: bool
    em_used: bool
    v_intersect_mV: float

def compute_entry(path: str) -> CacheEntry:
    try:
        _iv, series = build_series_with_IV(path)
        if series.x.size < 12:
            return CacheEntry(False, "too few points after cutoff", None, None,
                None, None, None, None, None, None, None, False, False, float("nan
                "))

        z_gmm, means_gmm = fit_gmm_baseline_1d(series.x)

        z_ransac, (a0, b0), (a1, b1) = global2_ransac_lines(
            series.Vv_V, series.x,
            residual_threshold=float(RANSAC_RESIDUAL_THRESHOLD),
            max_iter=int(RANSAC_GLOBAL2_MAX_ITER),
            z0=z_gmm,
        )

# new rule: enable em only if lines intersect within [-1100, 1100] mv
        intersects, v_star_mV = ransac_lines_intersect_in_domain_mV(a0, b0, a1, b1
            , domain_mV=INTERSECTION_WINDOW_mV)
        em_used = bool(intersects)

        z_hmm, hmm_params, gamma = slr_hmm_fit_predict(
            series.Vv_V, series.x,
            (a0, b0), (a1, b1),
            z_init=z_ransac,
            update_lines_with_em_override=em_used,
        )

        saved = False
        if SAVE_PSEUDOLABELS:
            settings = dict(
                tag=PSEUDOLABEL_TAG,
                V_CUTOFF_mV=float(V_CUTOFF_mV),
                EPS_G=float(EPS_G),
                RANSAC_RESIDUAL_THRESHOLD=float(RANSAC_RESIDUAL_THRESHOLD),
                RANSAC_GLOBAL2_MAX_ITER=int(RANSAC_GLOBAL2_MAX_ITER),
                A_STICKY=float(A_STICKY),
                HMM_N_ITER=int(HMM_N_ITER),
```

```
499                     USE_VITERBI_DECODE=bool(USE_VITERBI_DECODE),
500                     UPDATE_LINES_WITH_EM_GLOBAL=bool(UPDATE_LINES_WITH_EM),
501                     UPDATE_LINES_WITH_EM_USED_THIS_IV=bool(em_used),
502                     INTERSECTION_WINDOW_mV=float(INTERSECTION_WINDOW_mV),
503                     V_INTERSECT_mV=(None if not math.isfinite(v_star_mV) else float(
                            v_star_mV)),
504                     MIN_SIGMA2=float(MIN_SIGMA2),
505                 )
506                 ransac_lines = dict(a0=a0, b0=b0, a1=a1, b1=b1)
507                 save_npz(path, series, z_gmm, z_ransac, z_hmm, ransac_lines,
                        hmm_params, settings)
508                 saved = True
509
510             return CacheEntry(
511                 ok=True,
512                 err="",
513                 series=series,
514                 z_gmm=z_gmm,
515                 means_gmm=means_gmm,
516                 z_ransac=z_ransac,
517                 ransac_line0=(a0, b0),
518                 ransac_line1=(a1, b1),
519                 z_hmm=z_hmm,
520                 hmm_params=hmm_params,
521                 gamma=gamma,
522                 saved_npz=saved,
523                 em_used=em_used,
524                 v_intersect_mV=v_star_mV,
525             )
526
527         except Exception as e:
528             return CacheEntry(
529                 ok=False,
530                 err=f"{type(e).__name__}: {e}",
531                 series=None,
532                 z_gmm=None,
533                 means_gmm=None,
534                 z_ransac=None,
535                 ransac_line0=None,
536                 ransac_line1=None,
537                 z_hmm=None,
538                 hmm_params=None,
539                 gamma=None,
540                 saved_npz=False,
541                 em_used=False,
542                 v_intersect_mV=float("nan"),
543             )
544
545     # viewer
546     def launch_viewer(files: List[str]) -> None:
547         import tkinter as tk
548         from tkinter import ttk
```

```
549
550     import matplotlib
551     matplotlib.use("TkAgg")
552     from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
553     from matplotlib.figure import Figure
554
555     root = tk.Tk()
556     root.title("SLR-HMM viewer (1D GMM init | SLR-HMM | RANSAC 2 global lines)")
557
558     root.columnconfigure(0, weight=0)
559     root.columnconfigure(1, weight=1)
560     root.rowconfigure(0, weight=1)
561
562     left = ttk.Frame(root, padding=8)
563     left.grid(row=0, column=0, sticky="nsw")
564     left.columnconfigure(0, weight=1)
565
566     main = ttk.Frame(root, padding=8)
567     main.grid(row=0, column=1, sticky="nsew")
568     main.columnconfigure(0, weight=1)
569     main.rowconfigure(0, weight=1)
570
571     ttk.Label(left, text="IV files (from manifest)", font=("TkDefaultFont", 10, "
            bold")).grid(row=0, column=0, sticky="w")
572     lb = tk.Listbox(left, width=62, height=28)
573     lb.grid(row=1, column=0, sticky="nsw", pady=(6, 6))
574
575     stats = ttk.LabelFrame(left, text="Stats", padding=8)
576     stats.grid(row=2, column=0, sticky="ew", pady=(0, 8))
577     stats_text = tk.StringVar(value="(select a file)")
578     ttk.Label(stats, textvariable=stats_text, justify="left", anchor="w").grid(row
            =0, column=0, sticky="w")
579
580     fig = Figure(figsize=(11.0, 8.5), dpi=110)
581     canvas = FigureCanvasTkAgg(fig, master=main)
582     canvas.get_tk_widget().grid(row=0, column=0, sticky="nsew")
583
584     status = ttk.Label(main, text="", anchor="w")
585     status.grid(row=1, column=0, sticky="ew", pady=(6, 0))
586
587     cache: Dict[str, CacheEntry] = {}
588     cache_lock = threading.Lock()
589     stop_flag = {"stop": False}
590
591     def _get_or_compute(p: str) -> CacheEntry:
592         with cache_lock:
593             if p in cache:
594                 return cache[p]
595         e = compute_entry(p)
596         with cache_lock:
597             cache[p] = e
598         return e
```

```
599
600     def _background_precompute():
601         for p in files:
602             print(f"background precompute: {p}")
603             if stop_flag["stop"]:
604                 print("stopped background precompute")
605                 return
606             _get_or_compute(p)
607
608     if PRECOMPUTE_IN_BACKGROUND:
609         threading.Thread(target=_background_precompute, daemon=True).start()
610
611     def _scatter_states(ax, V_mV: np.ndarray, x: np.ndarray, z: np.ndarray, title:
            str) -> None:
612         z = np.asarray(z, int)
613         m0 = z == 0
614         m1 = z == 1
615         ax.plot(V_mV, x, linewidth=LINE_WIDTH)
616         ax.scatter(V_mV[m0], x[m0], s=POINT_SIZE, label="state 0 (low)")
617         ax.scatter(V_mV[m1], x[m1], s=POINT_SIZE, label="state 1 (high)")
618         ax.set_ylabel("log10(|G|)")
619         ax.set_title(title)
620         ax.legend(loc="best", fontsize=9)
621
622     def redraw(*_):
623         sel = lb.curselection()
624         if not sel:
625             return
626         idx = int(sel[0])
627         path = files[idx]
628         base = os.path.basename(path)
629
630         e = _get_or_compute(path)
631
632         fig.clf()
633         axs = [fig.add_subplot(3, 1, i + 1) for i in range(3)]
634         for ax in axs[:-1]:
635             ax.tick_params(labelbottom=False)
636
637         if not e.ok or e.series is None:
638             axs[0].text(0.1, 0.5, f"ERROR: {e.err}", transform=axs[0].transAxes)
639             canvas.draw_idle()
640             stats_text.set(f"ERROR: {e.err}")
641             status.configure(text=f"{idx+1}/{len(files)} | {path}")
642             return
643
644         s = e.series
645
646         if e.z_gmm is not None and e.means_gmm is not None:
647             _scatter_states(
648                 axs[0], s.Vv_mV, s.x, e.z_gmm,
```

```
649                 f"(1) 1D GMM init on x | means [{e.means_gmm[0]:.3f}, {e.means_gmm
                        [1]:.3f}] | {base}"
650             )
651
652         if e.z_hmm is not None and e.hmm_params is not None:
653             p = e.hmm_params
654             _scatter_states(
655                 axs[1], s.Vv_mV, s.x, e.z_hmm,
656                 f"(2) SLR-HMM | A_STICKY={A_STICKY:.3f} | sig2=[{p['sig20']:.3g},{
                        p['sig21']:.3g}] | {base}"
657             )
658
659         if e.z_ransac is not None and e.ransac_line0 is not None and e.
                ransac_line1 is not None:
660             (a0, b0) = e.ransac_line0
661             (a1, b1) = e.ransac_line1
662
663             _scatter_states(
664                 axs[2], s.Vv_mV, s.x, e.z_ransac,
665                 f"(3) RANSAC 2 global lines | thr={RANSAC_RESIDUAL_THRESHOLD:g} |
                        {base}"
666             )
667
668             Vgrid_mV = np.linspace(float(np.min(s.Vv_mV)), float(np.max(s.Vv_mV)),
                    200)
669             Vgrid_V = Vgrid_mV * 1e-3
670             y0 = a0 * Vgrid_V + b0
671             y1 = a1 * Vgrid_V + b1
672             axs[2].plot(Vgrid_mV, y0, linewidth=2.0)
673             axs[2].plot(Vgrid_mV, y1, linewidth=2.0)
674
675         axs[-1].set_xlabel("V (mV)")
676         fig.tight_layout()
677         canvas.draw_idle()
678
679 # stats (now correctly shows per-iv em usage)
680         vstar = e.v_intersect_mV
681         vstar_txt = "NA" if not math.isfinite(vstar) else f"{vstar:.1f} mV"
682         txt = (
683             f"{base}\n\n"
684             f"RANSAC thr={RANSAC_RESIDUAL_THRESHOLD:g} | global2 iters={
                    RANSAC_GLOBAL2_MAX_ITER}\n"
685             f"HMM: A_STICKY={A_STICKY:.3f} | EM iters={HMM_N_ITER} | viterbi={int(
                    USE_VITERBI_DECODE)}\n"
686             f"Per-IV EM used: {int(e.em_used)} | intersection V*={vstar_txt} |
                    window=Âś{INTERSECTION_WINDOW_mV:.0f} mV\n"
687         )
688         if e.hmm_params is not None:
689             p = e.hmm_params
690             txt += (
691                 f"\nHMM lines:\n"
```

```
692                 f"  line0: a={p['a0']:.4g}, b={p['b0']:.4g}, sig2={p['sig20']:.3g}\n"
693                 f"  line1: a={p['a1']:.4g}, b={p['b1']:.4g}, sig2={p['sig21']:.3g}\n"
694             )
695         stats_text.set(txt)
696
697         saved_tag = " | SAVED" if e.saved_npz else ""
698         status.configure(text=f"{idx+1}/{len(files)} | {path}{saved_tag}")
699
700     def on_select(_event):
701         redraw()
702
703     lb.bind("<<ListboxSelect>>", on_select)
704
705     for p in files:
706         lb.insert("end", os.path.basename(p))
707
708     if files:
709         lb.selection_set(0)
710         lb.see(0)
711         redraw()
712
713     def on_close():
714         stop_flag["stop"] = True
715         root.destroy()
716
717     root.protocol("WM_DELETE_WINDOW", on_close)
718     root.mainloop()
719
720 # main
721 def main() -> None:
722     manifest_path = os.path.join(DATA_DIR, FILTER_MANIFEST_NAME)
723     if not os.path.exists(manifest_path):
724         raise FileNotFoundError(f"Missing manifest: {manifest_path}")
725
726     rows = read_filter_manifest(manifest_path)
727     files = select_files_from_manifest(rows)
728     if not files:
729         raise RuntimeError("No files selected (manifest filters / ONLY_USE_PASSED_ALL).")
730
731     print("Files:", len(files))
732     print("SAVE_PSEUDOLABELS:", SAVE_PSEUDOLABELS, "| dir:", os.path.join(DATA_DIR, PSEUDOLABEL_DIRNAME))
733     print("RANSAC thr:", RANSAC_RESIDUAL_THRESHOLD, "| global2 iters:", RANSAC_GLOBAL2_MAX_ITER)
734     print("HMM: A_STICKY:", A_STICKY, "| HMM_N_ITER:", HMM_N_ITER, "| viterbi:", USE_VITERBI_DECODE)
735     print("Per-IV EM rule: EM only if intersection in Âś", INTERSECTION_WINDOW_mV, "mV")
736
```

```
737         launch_viewer(files)
738
739 if __name__ == "__main__":
740         main()
```

Listing A.2: SLR-HMM script

A.3 Temporal Convolution Network

A.3.1 Training

```
#!/usr/bin/env python3
from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path
from typing import Dict, List, Optional, Tuple

import math
import re
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader, Subset

from sklearn.mixture import GaussianMixture

# settings

DATA_DIRS = [
    r"C:\Users\maxdo\nn_clustering\data\raw\1_MSH0443\pseudolabels_slr_hmm_2_state_soft",
    r"C:\Users\maxdo\nn_clustering\data\raw\11_MSH0477\pseudolabels_slr_hmm_2_state_soft",
    r"C:\Users\maxdo\nn_clustering\data\raw\12_MSH0501\pseudolabels_slr_hmm_2_state_soft",
]

EPOCHS = 40
BATCH_SIZE = 32
LR = 2e-3
MODEL_WIDTH = 128
N_BLOCKS = 7
DROPOUT = 0.1
KERNEL_SIZE = 5
USE_SOFT_TARGETS = True

SAVE_PATH = Path(r"C:\Users\maxdo\nn_clustering\TCN_Model_Outputs\tcn_pointwise.pt")

# split settings
TEST_FRACTION = 0.20
VAL_FRACTION_OF_REMAIN = 0.15
SPLIT_SEED = 68934062

# feature settings
V_SCALE_MV = 1000.0
EPS = 1e-8
```

```
45
46  # conductance recompute settings
47  ZERO_V_CUTOFF_MV = 30.0
48  EPS_G = 1e-20
49
50  PREFER_RECOMPUTE_FROM_SOURCE = True
51
52  # gmm teacher
53  USE_GMM_1D_TEACHER = False
54  ALPHA_MIN = 0.60
55  ALPHA_MAX = 0.95
56
57  GMM_N_COMPONENTS = 2
58  GMM_N_INIT = 5
59  GMM_REG_COVAR = 1e-6
60  GMM_MAX_ITER = 200
61  GMM_RANDOM_STATE = 0
62  GMM_MIN_POINTS = 20
63
64  # speed knobs
65  NUM_WORKERS = 0
66  PIN_MEMORY = False
67
68  # import the project's iv class (iv.py)
69  try:
70      from mechanical_switches.utils import IV  # type: ignore
71  except Exception:
72      from iv import IV  # type: ignore
73
74  # sweep helpers
75
76  def _split_by_direction_changes(Vv: np.ndarray) -> List[Tuple[int, int]]:
77      Vv = np.asarray(Vv, float)
78      n = Vv.size
79      if n < 4:
80          return [(0, n)] if n > 0 else []
81
82      d = np.diff(Vv)
83      s = np.sign(d)
84
85      # fill zero directions
86      last = 0.0
87      for i in range(s.size):
88          if s[i] == 0:
89              s[i] = last if last != 0 else 1.0
90          else:
91              last = s[i]
92
93      change = np.where(np.diff(s) != 0)[0] + 1
94      cuts = np.concatenate(([0], change, [n]))
95
96      segs: List[Tuple[int, int]] = []
```

```
    for i in range(len(cuts) - 1):
        a = int(cuts[i])
        b = int(cuts[i + 1])
        if b - a >= 2:
            segs.append((a, b))
    return segs

def split_into_four_segments(Vv: np.ndarray, y: np.ndarray) -> List[Dict[str, object]]:
    Vv = np.asarray(Vv, float)
    y = np.asarray(y, float)
    segs = _split_by_direction_changes(Vv)

    def seg_len(ab: Tuple[int, int]) -> int:
        return ab[1] - ab[0]

    while len(segs) > 4:
        lengths = [seg_len(s) for s in segs]
        k = int(np.argmin(lengths))

        if k == 0:
            segs[1] = (segs[0][0], segs[1][1])
            segs.pop(0)
        elif k == len(segs) - 1:
            segs[-2] = (segs[-2][0], segs[-1][1])
            segs.pop(-1)
        else:
            a0, a1 = segs[k]
            left0, left1 = segs[k - 1]
            right0, right1 = segs[k + 1]
            jump_left = abs(Vv[a0] - Vv[left1 - 1])
            jump_right = abs(Vv[right0] - Vv[a1 - 1])
            if jump_left <= jump_right:
                segs[k - 1] = (left0, a1)
                segs.pop(k)
            else:
                segs[k + 1] = (a0, right1)
                segs.pop(k)

    out: List[Dict[str, object]] = []
    for (a, b) in segs:
        Vseg = Vv[a:b]
        yseg = y[a:b]
        direction = "inc" if (Vseg[-1] > Vseg[0]) else "dec"
        out.append({"i0": a, "i1": b, "V": Vseg, "y": yseg, "direction": direction
            })
    return out

def _full_sweep_mask_from_segments(segments: List[Dict[str, object]]) -> np.
    ndarray:
    if not segments:
        return np.array([], dtype=np.float32)
```

```
146
147     n = int(max(int(s["i1"]) for s in segments))
148     is_full = np.zeros((n,), dtype=np.float32)
149
150     if len(segments) == 4:
151         full = [segments[1], segments[2]]
152     else:
153         full = sorted(segments, key=lambda s: int(s["i1"]) - int(s["i0"]), reverse
                =True)[:2]
154
155     for s in full:
156         a = int(s["i0"])
157         b = int(s["i1"])
158         is_full[a:b] = 1.0
159
160     return is_full
161
162 def _direction_feature_from_segments(segments: List[Dict[str, object]]) -> np.
    ndarray:
163     if not segments:
164         return np.array([], dtype=np.float32)
165
166     n = int(max(int(s["i1"]) for s in segments))
167     dfeat = np.zeros((n,), dtype=np.float32)
168
169     for s in segments:
170         a = int(s["i0"])
171         b = int(s["i1"])
172         dfeat[a:b] = 1.0 if (s["direction"] == "inc") else -1.0
173
174     return dfeat
175
176 # gmm teacher
177
178 def _gmm_1d_teacher_from_x(x_log10_absG: np.ndarray) -> Tuple[np.ndarray, Dict[str
    , float]]:
179     x = np.asarray(x_log10_absG, dtype=np.float64)
180     T = x.size
181     if T < GMM_MIN_POINTS:
182         raise ValueError(f"Too few points for GMM teacher (T={T} < {GMM_MIN_POINTS
                })")
183
184     g_feat = (x * 10.0).reshape(-1, 1)
185
186     gmm = GaussianMixture(
187         n_components=GMM_N_COMPONENTS,
188         n_init=GMM_N_INIT,
189         reg_covar=GMM_REG_COVAR,
190         max_iter=GMM_MAX_ITER,
191         random_state=GMM_RANDOM_STATE,
192     ).fit(g_feat)
193
```

```
    proba = gmm.predict_proba(g_feat)
    means = gmm.means_.reshape(-1)

    hi = int(np.argmax(means))
    lo = 1 - hi

    p_hi = proba[:, hi].astype(np.float32)
    info = {"gmm_mean_low": float(means[lo]), "gmm_mean_high": float(means[hi])}
    return p_hi, info

def _conf_to_alpha(conf: np.ndarray) -> np.ndarray:
    c = np.asarray(conf, dtype=np.float32)
    c = np.clip(c, 0.0, 1.0)
    return (ALPHA_MIN + (ALPHA_MAX - ALPHA_MIN) * c).astype(np.float32)

# npz helpers

def _npz_get_scalar_str(d: np.lib.npyio.NpzFile, key: str) -> Optional[str]:
    if key not in d.files:
        return None
    try:
        v = d[key]
# could be array(['...'], dtype='<u...')
        if isinstance(v, np.ndarray) and v.size >= 1:
            return str(v.flat[0])
        return str(v)
    except Exception:
        return None

def _load_voltage_from_npz(d: np.lib.npyio.NpzFile) -> Tuple[np.ndarray, str]:
    # voltage in mV and a label of which key was used
    if "V_mV" in d.files:
        V = np.asarray(d["V_mV"], dtype=np.float32)
        return V, "V_mV"
    if "V_V" in d.files:
        V = np.asarray(d["V_V"], dtype=np.float32) * 1e3
        return V.astype(np.float32), "V_V"
    if "Vv" in d.files:
        V = np.asarray(d["Vv"], dtype=np.float32)
        return V, "Vv"
    raise KeyError("Voltage key missing: expected one of {'V_mV','V_V','Vv'}")

def _load_x_from_npz(d: np.lib.npyio.NpzFile) -> np.ndarray:
    if "x" in d.files:
        return np.asarray(d["x"], dtype=np.float32)
    raise KeyError("Feature key missing: expected 'x' in .npz")

def _load_primary_target(d: np.lib.npyio.NpzFile, use_soft_targets: bool) -> Tuple
    [np.ndarray, str]:
    # Preferred:
    if use_soft_targets and ("p_state1" in d.files):
        p = np.asarray(d["p_state1"], dtype=np.float32)
```

```
        return np.clip(p, 0.0, 1.0), "p_state1"

    if "z" in d.files:
        z = np.asarray(d["z"], dtype=np.float32)
        return np.clip(z, 0.0, 1.0), "z"

# new slr-hmm exports:
    if "z_slr_hmm" in d.files:
        z = np.asarray(d["z_slr_hmm"], dtype=np.float32)
        return np.clip(z, 0.0, 1.0), "z_slr_hmm"

# extra fallbacks (in case you want to train on other teachers)
    for k in ("z_ransac", "z_gmm"):
        if k in d.files:
            z = np.asarray(d[k], dtype=np.float32)
            return np.clip(z, 0.0, 1.0), k

    raise KeyError("Need one of {'p_state1','z','z_slr_hmm','z_ransac','z_gmm'} in
        .npz")

def _load_confidence(d: np.lib.npyio.NpzFile, T: int) -> Tuple[np.ndarray, str]:
    # Uses conf if present; otherwise all-ones
    if "conf" in d.files:
        w = np.asarray(d["conf"], dtype=np.float32)
        w = np.clip(w, 0.0, 1.0)
        if w.shape[0] != T:
            raise ValueError("conf length mismatch")
        return w, "conf"
    return np.ones((T,), dtype=np.float32), "ones"

# recompute from source

def _read_iv_dat_two_col(path: Path) -> Tuple[np.ndarray, np.ndarray]:
    V, I = [], []
    with open(path, "r", encoding="utf-8", errors="ignore") as f:
        for line in f:
            s = line.strip()
            if (not s) or s.startswith("#"):
                continue
            parts = re.split(r"[,\s;]+", s)
            if len(parts) < 2:
                continue
            try:
                v = float(parts[0])
                i = float(parts[1])
            except ValueError:
                continue
            if math.isfinite(v) and math.isfinite(i):
                V.append(v)
                I.append(i)
    if not V:
        raise ValueError(f"No numeric V,I rows in: {path}")
```

```
296         return np.asarray(V, float), np.asarray(I, float)
297
298 PAT = re.compile(r"^IV.*_(\d+)_(\d{4})\.dat$", re.IGNORECASE)
299
300 def _parse_ids_and_date_from_name(name: str) -> Tuple[int, int, str]:
301     trace_id = 0
302     iv_id = 0
303     m = PAT.match(name)
304     if m:
305         trace_id = int(m.group(1))
306         iv_id = int(m.group(2))
307     date_match = re.search(r"\d{2}-[A-Za-z]{3}-\d{2}", name)
308     date = date_match.group() if date_match else ""
309     return trace_id, iv_id, date
310
311 def _recompute_VmV_x_from_source_file(source_file: str) -> Tuple[np.ndarray, np.
    ndarray]:
312     p = Path(source_file)
313     if not p.exists():
314         raise FileNotFoundError(f"source_file does not exist: {p}")
315
316     V_mV, I = _read_iv_dat_two_col(p)
317     V_V = V_mV * 1e-3
318
319     trace_id, iv_id, date = _parse_ids_and_date_from_name(p.name)
320
321     iv = IV(
322         file=p.name,
323         date=date,
324         trace_id=trace_id,
325         id=iv_id,
326         vbias=V_V,
327         current=I,
328         motor_position=0,
329         config_id=None,
330         magnetic_field=0,
331         settings={},
332     )
333
334     zero_thr_V = float(ZERO_V_CUTOFF_MV) * 1e-3
335     G, Vbias = iv.calculate_conductance(zero_threshold=zero_thr_V, offsetted=True)
336
337     G = np.asarray(G, dtype=np.float32)
338     Vbias = np.asarray(Vbias, dtype=np.float32)
339     if G.size == 0 or Vbias.size == 0:
340         return np.array([], dtype=np.float32), np.array([], dtype=np.float32)
341
342     n = min(G.size, Vbias.size)
343     G = G[:n]
344     Vbias = Vbias[:n]
345
346     V_mV = (Vbias * 1e3).astype(np.float32)
```

```
    mask = np.isfinite(V_mV) & np.isfinite(G)
    V_mV = V_mV[mask]
    G = G[mask]

    x = np.log10(np.maximum(np.abs(G), float(EPS_G))).astype(np.float32)
    return V_mV, x

# dataset

@dataclass
class Sample:
    x: torch.Tensor
    p_teacher: torch.Tensor
    w: torch.Tensor
    mask: torch.Tensor
    meta: Dict

class MultiRootIVPointDataset(Dataset):
    # Supports new SLR-HMM soft pseudolabel files:
    def __init__(
        self,
        roots: List[Path],
        *,
        use_soft_targets: bool = True,
        v_scale_mv: float = 1000.0,
        eps: float = 1e-8,
        prefer_recompute_from_source: bool = True,
    ):
        self.roots = [Path(r) for r in roots]
        self.files: List[Path] = []

# robust discovery:
        for r in self.roots:
            if r.exists():
                self.files.extend(sorted(r.rglob("*.npz")))

        if not self.files:
            raise FileNotFoundError(f"No .npz files found under: {self.roots}")

        self.use_soft_targets = bool(use_soft_targets)
        self.v_scale_mv = float(v_scale_mv)
        self.eps = float(eps)
        self.prefer_recompute_from_source = bool(prefer_recompute_from_source)

    def __len__(self) -> int:
        return len(self.files)

    def __getitem__(self, idx: int) -> Sample:
        p = self.files[idx]
        d = np.load(p, allow_pickle=True)

```

```
398         p_primary, target_key = _load_primary_target(d, use_soft_targets=self.
                use_soft_targets)
399
400         V_mV_stored, v_key = _load_voltage_from_npz(d)
401         x_stored = _load_x_from_npz(d)
402
403         T = int(V_mV_stored.shape[0])
404         if x_stored.shape[0] != T:
405             raise ValueError(f"{p.name}: voltage length {T} != x length {x_stored.
                    shape[0]}")
406         if p_primary.shape[0] != T:
407             raise ValueError(f"{p.name}: target length mismatch (T={T}, target={
                    p_primary.shape[0]})")
408
409         w, w_key = _load_confidence(d, T)
410
411         source_file = _npz_get_scalar_str(d, "source_file")
412         used_recompute = False
413         recompute_error = None
414
415         V_mV = V_mV_stored
416         x = x_stored
417
418         if self.prefer_recompute_from_source and source_file:
419             try:
420                 V_mV_r, x_r = _recompute_VmV_x_from_source_file(source_file)
421                 if V_mV_r.size > 0 and x_r.size > 0:
422                     if V_mV_r.shape[0] != T or x_r.shape[0] != T:
423 # fall back to stored to preserve alignment with labels.
424                         raise ValueError(f"recompute length mismatch: recompute {
                                V_mV_r.size} vs labels {T}")
425                     V_mV = V_mV_r
426                     x = x_r
427                     used_recompute = True
428             except Exception as e:
429                 recompute_error = f"{type(e).__name__}: {e}"
430                 used_recompute = False
431
432         segments = split_into_four_segments(V_mV, x)
433         dir_feat = _direction_feature_from_segments(segments)
434         is_full = _full_sweep_mask_from_segments(segments)
435
436         if dir_feat.size != T:
437             dir_feat = np.resize(dir_feat, (T,)).astype(np.float32)
438         if is_full.size != T:
439             is_full = np.resize(is_full, (T,)).astype(np.float32)
440
441         v_norm = (V_mV / float(self.v_scale_mv)).astype(np.float32)
442
443         x_mean = float(np.mean(x))
444         x_std = float(np.std(x))
445         x_std = x_std if x_std > self.eps else 1.0
```

```
        xz = ((x - x_mean) / x_std).astype(np.float32)

        dx = np.gradient(xz).astype(np.float32)
        dx = np.clip(dx, -20.0, 20.0)

        X = np.stack([v_norm, xz, dx, dir_feat, is_full], axis=1).astype(np.
            float32)
        X = np.nan_to_num(X, nan=0.0, posinf=0.0, neginf=0.0)

        used_gmm = False
        gmm_error = None
        gmm_info: Dict[str, float] = {}

        p_teacher = p_primary.copy()
        if USE_GMM_1D_TEACHER:
            try:
                p_gmm, gmm_info = _gmm_1d_teacher_from_x(x)
                alpha = _conf_to_alpha(w)
                p_teacher = alpha * p_primary + (1.0 - alpha) * p_gmm
                used_gmm = True
            except Exception as e:
                gmm_error = f"{type(e).__name__}: {e}"
                p_teacher = p_primary.copy()
                used_gmm = False

        p_teacher = np.clip(p_teacher, 0.0, 1.0).astype(np.float32)

        meta = {
            "path": str(p),
            "voltage_key": v_key,
            "target_key": target_key,
            "weight_key": w_key,
            "source_file": source_file,
            "used_recompute_from_source": used_recompute,
            "recompute_error": recompute_error,
            "used_gmm_teacher": used_gmm,
            "gmm_error": gmm_error,
            **gmm_info,
        }

        return Sample(
            x=torch.from_numpy(X),
            p_teacher=torch.from_numpy(p_teacher),
            w=torch.from_numpy(w),
            mask=torch.ones((T,), dtype=torch.bool),
            meta=meta,
        )

def collate_pad(batch: List[Sample]) -> Dict[str, torch.Tensor]:
    T_max = max(s.x.shape[0] for s in batch)
    C = batch[0].x.shape[1]

```

```
    X = torch.zeros((len(batch), T_max, C), dtype=torch.float32)
    P = torch.zeros((len(batch), T_max), dtype=torch.float32)
    W = torch.zeros((len(batch), T_max), dtype=torch.float32)
    M = torch.zeros((len(batch), T_max), dtype=torch.bool)

    for i, s in enumerate(batch):
        T = s.x.shape[0]
        X[i, :T] = s.x
        P[i, :T] = s.p_teacher
        W[i, :T] = s.w
        M[i, :T] = s.mask

    return {"X": X, "P": P, "W": W, "M": M}

# model: simple tcn

class TCNBlock(nn.Module):
    def __init__(self, channels: int, kernel_size: int, dilation: int, dropout: float):
        super().__init__()
        pad = (kernel_size - 1) * dilation // 2
        self.conv1 = nn.Conv1d(channels, channels, kernel_size, padding=pad, dilation=dilation)
        self.conv2 = nn.Conv1d(channels, channels, kernel_size, padding=pad, dilation=dilation)
        self.dropout = nn.Dropout(dropout)
        self.norm1 = nn.GroupNorm(1, channels)
        self.norm2 = nn.GroupNorm(1, channels)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        y = self.conv1(x)
        y = self.norm1(y)
        y = F.relu(y)
        y = self.dropout(y)

        y = self.conv2(y)
        y = self.norm2(y)
        y = F.relu(y)
        y = self.dropout(y)

        return x + y

class TCNPointwise(nn.Module):
    def __init__(self, in_ch: int = 5, width: int = 64, kernel_size: int = 5, dropout: float = 0.1, n_blocks: int = 6):
        super().__init__()
        self.in_proj = nn.Conv1d(in_ch, width, kernel_size=1)
        dilations = [2 ** i for i in range(n_blocks)]
        self.blocks = nn.ModuleList([
            TCNBlock(width, kernel_size=kernel_size, dilation=d, dropout=dropout)
            for d in dilations
        ])
```

```
        self.out_proj = nn.Conv1d(width, 1, kernel_size=1)

    def forward(self, X: torch.Tensor) -> torch.Tensor:
        x = X.transpose(1, 2)
        x = self.in_proj(x)
        for blk in self.blocks:
            x = blk(x)
        logits = self.out_proj(x).squeeze(1)
        return logits

# loss

def weighted_bce_with_logits(
    logits: torch.Tensor,
    targets: torch.Tensor,
    weights: torch.Tensor,
    mask: torch.Tensor,
) -> torch.Tensor:
    bce = F.binary_cross_entropy_with_logits(logits, targets, reduction="none")
    w = weights * mask.float()
    denom = torch.clamp(w.sum(), min=1.0)
    return (bce * w).sum() / denom

# splits

def write_list(path: Path, items: List[Path]) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)
    with open(path, "w", encoding="utf-8") as f:
        for p in items:
            f.write(str(p) + "\n")

def main():
    device = "cuda" if torch.cuda.is_available() else "cpu"
    print(f"Using device: {device}")

    roots = [Path(p) for p in DATA_DIRS]
    ds = MultiRootIVPointDataset(
        roots,
        use_soft_targets=USE_SOFT_TARGETS,
        v_scale_mv=V_SCALE_MV,
        eps=EPS,
        prefer_recompute_from_source=PREFER_RECOMPUTE_FROM_SOURCE,
    )

    total = len(ds)
    print(f"Total .npz files: {total}")
    print(f"PREFER_RECOMPUTE_FROM_SOURCE={PREFER_RECOMPUTE_FROM_SOURCE} | ZERO_V_CUTOFF_MV={ZERO_V_CUTOFF_MV}")
    print(f"USE_GMM_1D_TEACHER={USE_GMM_1D_TEACHER} | alpha=[{ALPHA_MIN},{ALPHA_MAX}] | GMM_MIN_POINTS={GMM_MIN_POINTS}")

    SAVE_PATH.parent.mkdir(parents=True, exist_ok=True)
```

```
595
596     g = torch.Generator().manual_seed(int(SPLIT_SEED))
597     perm = torch.randperm(total, generator=g).tolist()
598
599     n_test = int(round(TEST_FRACTION * total))
600     test_idx = perm[:n_test]
601     remain_idx = perm[n_test:]
602
603     n_val = int(round(VAL_FRACTION_OF_REMAIN * len(remain_idx)))
604     val_idx = remain_idx[:n_val]
605     train_idx = remain_idx[n_val:]
606
607     train_ds = Subset(ds, train_idx)
608     val_ds = Subset(ds, val_idx)
609     test_ds = Subset(ds, test_idx)
610
611     train_files = [ds.files[i] for i in train_idx]
612     val_files = [ds.files[i] for i in val_idx]
613     test_files = [ds.files[i] for i in test_idx]
614
615     write_list(SAVE_PATH.with_suffix(".train_files.txt"), train_files)
616     write_list(SAVE_PATH.with_suffix(".val_files.txt"), val_files)
617     write_list(SAVE_PATH.with_suffix(".test_files.txt"), test_files)
618
619     print(f"Split: train={len(train_ds)} val={len(val_ds)} test={len(test_ds)}")
620     print(f"Wrote: {SAVE_PATH.with_suffix('.test_files.txt')}")
621
622     train_loader = DataLoader(
623         train_ds,
624         batch_size=BATCH_SIZE,
625         shuffle=True,
626         collate_fn=collate_pad,
627         num_workers=NUM_WORKERS,
628         pin_memory=PIN_MEMORY,
629     )
630     val_loader = DataLoader(
631         val_ds,
632         batch_size=BATCH_SIZE,
633         shuffle=False,
634         collate_fn=collate_pad,
635         num_workers=NUM_WORKERS,
636         pin_memory=PIN_MEMORY,
637     )
638
639     model = TCNPointwise(
640         in_ch=5,
641         width=MODEL_WIDTH,
642         kernel_size=KERNEL_SIZE,
643         dropout=DROPOUT,
644         n_blocks=N_BLOCKS,
645     ).to(device)
646
```

```
    opt = torch.optim.AdamW(model.parameters(), lr=LR, weight_decay=1e-4)

    best_val = float("inf")

    for epoch in range(1, EPOCHS + 1):
        model.train()
        tr_loss = 0.0
        for batch in train_loader:
            X = batch["X"].to(device)
            P = batch["P"].to(device)
            W = batch["W"].to(device)
            M = batch["M"].to(device)

            logits = model(X)
            loss = weighted_bce_with_logits(logits, P, W, M)

            opt.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            opt.step()

            tr_loss += loss.item()
        tr_loss /= max(1, len(train_loader))

        model.eval()
        va_loss = 0.0
        with torch.no_grad():
            for batch in val_loader:
                X = batch["X"].to(device)
                P = batch["P"].to(device)
                W = batch["W"].to(device)
                M = batch["M"].to(device)

                logits = model(X)
                loss = weighted_bce_with_logits(logits, P, W, M)
                va_loss += loss.item()
        va_loss /= max(1, len(val_loader))

        print(f"Epoch {epoch:03d} | train {tr_loss:.5f} | val {va_loss:.5f}")

        if va_loss < best_val:
            best_val = va_loss
            torch.save(
                {
                    "model_state": model.state_dict(),
                    "args": {
                        "in_ch": 5,
                        "width": MODEL_WIDTH,
                        "kernel": KERNEL_SIZE,
                        "dropout": DROPOUT,
                        "blocks": N_BLOCKS,
                        "v_scale_mv": V_SCALE_MV,
```

```
                        "eps": EPS,
                        "prefer_recompute_from_source":
                            PREFER_RECOMPUTE_FROM_SOURCE,
                        "zero_v_cutoff_mv": ZERO_V_CUTOFF_MV,
                        "eps_g": EPS_G,
                        "features": ["v_norm", "x_z", "dx", "dir(+1/-1)", "is_full
                            (0/1)"],
                        "teacher": {
                            "use_soft_targets": USE_SOFT_TARGETS,
                            "primary_preference": "p_state1 if present else z/
                                z_slr_hmm",
                            "use_gmm_1d": USE_GMM_1D_TEACHER,
                            "alpha_min": ALPHA_MIN,
                            "alpha_max": ALPHA_MAX,
                            "gmm_min_points": GMM_MIN_POINTS,
                            "gmm": {
                                "n_components": GMM_N_COMPONENTS,
                                "n_init": GMM_N_INIT,
                                "reg_covar": GMM_REG_COVAR,
                                "max_iter": GMM_MAX_ITER,
                                "random_state": GMM_RANDOM_STATE,
                                "feature": "conductance_clustering = log10(G)*10 (
                                    implemented as x*10)",
                                "state1_definition": "higher-mean component",
                            },
                        },
                    },
                },
                SAVE_PATH,
            )
            print(f"Saved best model âĘŠ {SAVE_PATH}")

    print("Training complete.")
    print("Test set is held out. Use a compare/view script to evaluate on test.")
    _ = test_ds  # held out; intentionally unused here

if __name__ == "__main__":
    main()
```

Listing A.3: TCN training script

A.3.2 Classification

```python
#!/usr/bin/env python3
from __future__ import annotations

import os
import re
import math
import json
from pathlib import Path
from typing import Dict, List, Tuple, Optional

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

try:
    from tqdm import tqdm
except Exception as e:
    raise RuntimeError("tqdm is required. Install with: pip install tqdm") from e

# settings

NN_CLUSTERING_ROOT = Path(r"C:\Users\maxdo\nn_clustering")

RAW_ROOT = NN_CLUSTERING_ROOT / "data" / "raw"

# model checkpoint path
MODEL_PATH = NN_CLUSTERING_ROOT / "TCN_Model_Outputs_Old" / "tcn_pointwise.pt"

DATA_DIRS: List[Path] = [
    RAW_ROOT / "10_MMA1944-10D",
    RAW_ROOT / "id4_MSH0028",
    ]

IV_GLOB = "IV*.dat"

V_CUTOFF_MV = 30.0
V_SCALE_MV = 1000.0
EPS = 1e-8
EPS_G = 1e-20

OUT_DIRNAME = "Finalized Labels"

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

# iv import
try:
    from mechanical_switches.utils import IV  # type: ignore
except Exception:
    from iv import IV  # type: ignore
```

```
51
52 # model
53
54 class TCNBlock(nn.Module):
55     def __init__(self, channels: int, kernel_size: int, dilation: int, dropout:
       float):
56         super().__init__()
57         pad = (kernel_size - 1) * dilation // 2
58         self.conv1 = nn.Conv1d(channels, channels, kernel_size, padding=pad,
           dilation=dilation)
59         self.conv2 = nn.Conv1d(channels, channels, kernel_size, padding=pad,
           dilation=dilation)
60         self.dropout = nn.Dropout(dropout)
61         self.norm1 = nn.GroupNorm(1, channels)
62         self.norm2 = nn.GroupNorm(1, channels)
63
64     def forward(self, x: torch.Tensor) -> torch.Tensor:
65         y = self.conv1(x)
66         y = self.norm1(y)
67         y = F.relu(y)
68         y = self.dropout(y)
69
70         y = self.conv2(y)
71         y = self.norm2(y)
72         y = F.relu(y)
73         y = self.dropout(y)
74         return x + y
75
76 class TCNPointwise(nn.Module):
77     def __init__(self, in_ch: int = 5, width: int = 64, kernel_size: int = 5,
       dropout: float = 0.1, n_blocks: int = 6):
78         super().__init__()
79         self.in_proj = nn.Conv1d(in_ch, width, kernel_size=1)
80         dilations = [2 ** i for i in range(n_blocks)]
81         self.blocks = nn.ModuleList([
82             TCNBlock(width, kernel_size=kernel_size, dilation=d, dropout=dropout)
83             for d in dilations
84         ])
85         self.out_proj = nn.Conv1d(width, 1, kernel_size=1)
86
87     def forward(self, X: torch.Tensor) -> torch.Tensor:
88         x = X.transpose(1, 2)
89         x = self.in_proj(x)
90         for blk in self.blocks:
91             x = blk(x)
92         logits = self.out_proj(x).squeeze(1)
93         return logits
94
95 # sweep helpers
96
97 def _split_by_direction_changes(Vv: np.ndarray) -> List[Tuple[int, int]]:
98     Vv = np.asarray(Vv, float)
```

```
    n = Vv.size
    if n < 4:
        return [(0, n)] if n > 0 else []

    d = np.diff(Vv)
    s = np.sign(d)

    last = 0.0
    for i in range(s.size):
        if s[i] == 0:
            s[i] = last if last != 0 else 1.0
        else:
            last = s[i]

    change = np.where(np.diff(s) != 0)[0] + 1
    cuts = np.concatenate(([0], change, [n]))

    segs: List[Tuple[int, int]] = []
    for i in range(len(cuts) - 1):
        a = int(cuts[i])
        b = int(cuts[i + 1])
        if b - a >= 2:
            segs.append((a, b))
    return segs

def split_into_four_segments(Vv: np.ndarray, y: np.ndarray) -> List[Dict[str,
    object]]:
    Vv = np.asarray(Vv, float)
    y = np.asarray(y, float)
    segs = _split_by_direction_changes(Vv)

    def seg_len(ab: Tuple[int, int]) -> int:
        return ab[1] - ab[0]

    while len(segs) > 4:
        lengths = [seg_len(s) for s in segs]
        k = int(np.argmin(lengths))

        if k == 0:
            segs[1] = (segs[0][0], segs[1][1])
            segs.pop(0)
        elif k == len(segs) - 1:
            segs[-2] = (segs[-2][0], segs[-1][1])
            segs.pop(-1)
        else:
            a0, a1 = segs[k]
            left0, left1 = segs[k - 1]
            right0, right1 = segs[k + 1]
            jump_left = abs(Vv[a0] - Vv[left1 - 1])
            jump_right = abs(Vv[right0] - Vv[a1 - 1])
            if jump_left <= jump_right:
                segs[k - 1] = (left0, a1)
```

```
                segs.pop(k)
            else:
                segs[k + 1] = (a0, right1)
                segs.pop(k)

    out: List[Dict[str, object]] = []
    for (a, b) in segs:
        Vseg = Vv[a:b]
        direction = "inc" if (Vseg[-1] > Vseg[0]) else "dec"
        out.append({"i0": a, "i1": b, "direction": direction})
    return out

def _direction_feature_from_segments(segments: List[Dict[str, object]], T: int) ->
    np.ndarray:
    dfeat = np.zeros((T,), dtype=np.float32)
    for s in segments:
        a = int(s["i0"])
        b = int(s["i1"])
        dfeat[a:b] = 1.0 if (s["direction"] == "inc") else -1.0
    return dfeat

def _full_sweep_mask_from_segments(segments: List[Dict[str, object]], T: int) ->
    np.ndarray:
    is_full = np.zeros((T,), dtype=np.float32)
    if not segments:
        return is_full
    if len(segments) == 4:
        full = [segments[1], segments[2]]
    else:
        full = sorted(segments, key=lambda s: int(s["i1"]) - int(s["i0"]), reverse
            =True)[:2]
    for s in full:
        a = int(s["i0"])
        b = int(s["i1"])
        is_full[a:b] = 1.0
    return is_full

# read iv dat

def read_iv_dat_two_col(path: Path) -> Tuple[np.ndarray, np.ndarray]:
    try:
        arr = np.loadtxt(path, dtype=float)
        if arr.ndim == 1:
            arr = arr.reshape(-1, 1)
        if arr.shape[1] < 2:
            raise ValueError("Need >=2 columns")
        V = np.asarray(arr[:, 0], float)
        I = np.asarray(arr[:, 1], float)
        return V, I
    except Exception:
        V, I = [], []
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
```

```
            for line in f:
                s = line.strip()
                if (not s) or s.startswith("#"):
                    continue
                parts = re.split(r"[,\s;]+", s)
                if len(parts) < 2:
                    continue
                try:
                    v = float(parts[0])
                    i = float(parts[1])
                except ValueError:
                    continue
                if math.isfinite(v) and math.isfinite(i):
                    V.append(v)
                    I.append(i)
        if not V:
            raise ValueError(f"No numeric V,I rows in: {path}")
        return np.asarray(V, float), np.asarray(I, float)

PAT = re.compile(r"^IV.*_(\d+)_(\d{4})\.dat$", re.IGNORECASE)

def parse_ids_and_date_from_name(name: str) -> Tuple[int, int, str]:
    trace_id = 0
    iv_id = 0
    m = PAT.match(name)
    if m:
        trace_id = int(m.group(1))
        iv_id = int(m.group(2))
    date_match = re.search(r"\d{2}-[A-Za-z]{3}-\d{2}", name)
    date = date_match.group() if date_match else ""
    return trace_id, iv_id, date

# feature builder (must match training)

def compute_VmV_xG_from_dat(dat_path: Path) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    V_mV_raw, I = read_iv_dat_two_col(dat_path)
    V_V_raw = V_mV_raw * 1e-3

    trace_id, iv_id, date = parse_ids_and_date_from_name(dat_path.name)

    iv = IV(
        file=dat_path.name,
        date=date,
        trace_id=trace_id,
        id=iv_id,
        vbias=V_V_raw,
        current=I,
        motor_position=0,
        config_id=None,
        magnetic_field=0,
        settings={},
```

```
    )

    zero_thr_V = float(V_CUTOFF_MV) * 1e-3
    G, Vbias = iv.calculate_conductance(zero_threshold=zero_thr_V, offsetted=True)

    G = np.asarray(G, dtype=np.float32)
    Vbias = np.asarray(Vbias, dtype=np.float32)
    if G.size == 0 or Vbias.size == 0:
        return np.array([], np.float32), np.array([], np.float32), np.array([], np
            .float32)

    n = min(G.size, Vbias.size)
    G = G[:n]
    Vbias = Vbias[:n]

    V_mV = (Vbias * 1e3).astype(np.float32)

    m = np.isfinite(V_mV) & np.isfinite(G)
    V_mV = V_mV[m]
    G = G[m]

    x = np.log10(np.maximum(np.abs(G), float(EPS_G))).astype(np.float32)
    return V_mV, x, G

def build_features_5(V_mV: np.ndarray, x: np.ndarray) -> np.ndarray:
    V_mV = np.asarray(V_mV, np.float32)
    x = np.asarray(x, np.float32)
    T = int(V_mV.size)

    v_norm = V_mV / float(V_SCALE_MV)

    x_mean = float(np.mean(x))
    x_std = float(np.std(x))
    x_std = x_std if x_std > EPS else 1.0
    xz = (x - x_mean) / x_std

    dx = np.gradient(xz).astype(np.float32)
    dx = np.clip(dx, -20.0, 20.0)

    segs = split_into_four_segments(V_mV, x)
    dir_feat = _direction_feature_from_segments(segs, T)
    is_full = _full_sweep_mask_from_segments(segs, T)

    X = np.stack([v_norm, xz, dx, dir_feat, is_full], axis=1).astype(np.float32)
    X = np.nan_to_num(X, nan=0.0, posinf=0.0, neginf=0.0)
    return X

@torch.no_grad()
def predict(model: nn.Module, X: np.ndarray, device: str) -> Tuple[np.ndarray, np.
    ndarray]:
    Xt = torch.from_numpy(X[None, ...]).to(device)  # (1,T,5)
    logits = model(Xt)
```

```
    p_hat = torch.sigmoid(logits).squeeze(0).detach().cpu().numpy().astype(np.
        float32)
    z_hat = (p_hat >= 0.5).astype(np.int8)
    return p_hat, z_hat

# main loop

def discover_molecule_dirs() -> List[Path]:
    if DATA_DIRS:
        return [Path(d) for d in DATA_DIRS]
    if not RAW_ROOT.exists():
        raise FileNotFoundError(f"RAW_ROOT does not exist: {RAW_ROOT}")
    out = []
    for p in sorted(RAW_ROOT.iterdir()):
        if p.is_dir():
            out.append(p)
    return out

def list_all_iv_files(molecule_dir: Path) -> List[Path]:
    return sorted(molecule_dir.rglob(IV_GLOB))

def save_labels_npz(out_path: Path, *, dat_file: str, V_mV: np.ndarray, x: np.
    ndarray, G: np.ndarray, p_hat: np.ndarray, z_hat: np.ndarray, meta: Dict):
    out_path.parent.mkdir(parents=True, exist_ok=True)
    V_V = (V_mV.astype(np.float32) * 1e-3).astype(np.float32)
    np.savez_compressed(
        out_path,
        iv_file=dat_file,
        V_mV=V_mV.astype(np.float32),
        V_V=V_V,
        x=x.astype(np.float32),
        G=G.astype(np.float32),
        p_hat=p_hat.astype(np.float32),
        z_hat=z_hat.astype(np.int8),
        meta=json.dumps(meta, indent=2),
    )

def main():
    print(f"Device: {DEVICE}")

    if not MODEL_PATH.exists():
        raise FileNotFoundError(f"MODEL_PATH not found: {MODEL_PATH}")

    ckpt = torch.load(MODEL_PATH, map_location=DEVICE)
    args = ckpt.get("args", {}) if isinstance(ckpt, dict) else {}

    in_ch = int(args.get("in_ch", 5))
    if in_ch != 5:
        print(f"WARNING: checkpoint in_ch={in_ch}, but this script builds 5
            features.")

    model = TCNPointwise(
```

```
        in_ch=in_ch,
        width=int(args.get("width", 64)),
        kernel_size=int(args.get("kernel", 5)),
        dropout=float(args.get("dropout", 0.1)),
        n_blocks=int(args.get("blocks", 6)),
    ).to(DEVICE)
    model.load_state_dict(ckpt["model_state"])
    model.eval()

    molecule_dirs = discover_molecule_dirs()
    if not molecule_dirs:
        raise RuntimeError("No molecule folders found to process.")

    errors_global: List[str] = []

    for mol_dir in molecule_dirs:
        mol_name = mol_dir.name
        iv_files = list_all_iv_files(mol_dir)

        if not iv_files:
            print(f"[{mol_name}] No files found matching {IV_GLOB}")
            continue

        out_dir = mol_dir / OUT_DIRNAME
        out_dir.mkdir(parents=True, exist_ok=True)

        err_log = out_dir / "errors.txt"
        if err_log.exists():
            err_log.unlink()

        print(f"\n[{mol_name}] Found {len(iv_files)} IV files. Output -> {out_dir}
            ")

        bar = tqdm(iv_files, desc=f"{mol_name}", unit="IV", dynamic_ncols=True)
        for dat_path in bar:
            try:
                V_mV, x, G = compute_VmV_xG_from_dat(dat_path)
                if V_mV.size == 0:
                    raise ValueError("Empty after conductance computation/cutoff")

                X = build_features_5(V_mV, x)
                p_hat, z_hat = predict(model, X, DEVICE)

                stem = dat_path.stem
                out_path = out_dir / f"{stem}.npz"

                meta = {
                    "model_path": str(MODEL_PATH),
                    "v_cutoff_mV": float(V_CUTOFF_MV),
                    "features": ["v_norm", "x_z", "dx", "dir(+1/-1)", "is_full
                        (0/1)"],
                    "source_dat": str(dat_path),
```

```
                }

                save_labels_npz(
                    out_path,
                    dat_file=dat_path.name,
                    V_mV=V_mV,
                    x=x,
                    G=G,
                    p_hat=p_hat,
                    z_hat=z_hat,
                    meta=meta,
                )

            except Exception as e:
                msg = f"{mol_name} | {dat_path} | {type(e).__name__}: {e}"
                errors_global.append(msg)
                with open(err_log, "a", encoding="utf-8") as f:
                    f.write(msg + "\n")

        bar.close()
        if err_log.exists():
            print(f"[{mol_name}] Completed with some errors. See: {err_log}")
        else:
            print(f"[{mol_name}] Completed without errors.")

    if errors_global:
        print(f"\nDone. Total errors: {len(errors_global)} (see per-molecule
            errors.txt).")
    else:
        print("\nDone. No errors.")

if __name__ == "__main__":
    main()
```

Listing A.4: TCN classifier script

A.4 Hysteretic Filter

```python
#!/usr/bin/env python3
from __future__ import annotations

import csv
import time
from dataclasses import dataclass
from pathlib import Path
from typing import Dict, List, Optional, Tuple

import numpy as np

try:
    from tqdm import tqdm
except Exception:  # pragma: no cover
    def tqdm(iterable, **kwargs):
        return iterable

import tkinter as tk
from tkinter import ttk, messagebox

import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

# settings

NN_CLUSTERING_ROOT = Path(r"C:\Users\maxdo\nn_clustering")
RAW_ROOT = NN_CLUSTERING_ROOT / "data" / "raw"

MOLECULE_DIRS: List[Path] = [
    RAW_ROOT / "1_MSH0443",
    RAW_ROOT / "11_MSH0477",
    RAW_ROOT / "12_MSH0501",
    RAW_ROOT / "10_MMA1944-10D",
    RAW_ROOT / "id4_MSH0028",
]

FINALIZED_LABELS_DIRNAME = "Finalized Labels"
NPZ_GLOB = "*.npz"

# cutoff
V_CUTOFF_MV = 30.0

# rule 1
LOW_BIAS_MV = 100.0
LOW_BIAS_DOMINANCE_FRAC = 0.20

# rule 2
HIGH_BIAS_MV = 100.0
```

```
51
52  # spike filter
53  MIN_RUN_AFTER_SWITCH = 1
54  ENABLE_SPIKE_FILTER_RULE2 = True
55
56  # rule 3
57  ENABLE_SPIKE_FILTER_RULE3 = True
58  RULE3_BIAS_THRESHOLD_MODE = "max_frac"  # "max_frac" | "fixed" | "none"
59  RULE3_BIAS_THRESHOLD_MAX_FRAC = 0.10
60  RULE3_BIAS_THRESHOLD_FIXED_MV = 100.0
61
62  ELEMENTWISE_RATIO_THRESHOLD = 1.4
63  ELEMENTWISE_RATIO_REQUIRED_FRAC = 0.30
64
65  # voltage window
66  ELEMENTWISE_RATIO_VMIN_MV = V_CUTOFF_MV
67  ELEMENTWISE_RATIO_VMAX_MV = 1100.0
68
69  # binning
70  ELEMENTWISE_RATIO_N_BINS = 20
71  ELEMENTWISE_RATIO_MIN_POINTS_PER_BIN_PER_STATE = 3
72  ELEMENTWISE_RATIO_MIN_VALID_BINS = 6
73
74  OUTPUT_DIR = NN_CLUSTERING_ROOT / "hysteretic_results_all_molecules"
75  OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
76
77  SIMPLE_LABELS_CSV = OUTPUT_DIR / "hysteretic_labels_all_molecules.csv"
78
79  PLOT_DOWNSAMPLE_MAX_POINTS = 4000
80  LINE_WIDTH = 0.8
81  LINE_ALPHA = 0.9
82  DOT_SIZE = 8
83  DOT_ALPHA = 0.9
84
85  # data classes
86
87  @dataclass
88  class SweepSegment:
89      idx: np.ndarray
90      V_mV: np.ndarray
91      x_logG: np.ndarray
92      z: np.ndarray
93
94  @dataclass
95  class HysteresisDecision:
96      is_hysteretic: bool
97      rule1_pass: bool
98      rule2_pass: bool
99      rule3_pass: bool
100     rule4_pass: bool
101
102 # rule 1 diagnostics
```

```
    low_bias_dom_frac_sweep2: float
    low_bias_dom_frac_sweep3: float
    low_bias_dom_label_sweep2: Optional[int]
    low_bias_dom_label_sweep3: Optional[int]

# rule 2 diagnostics (full sweeps only)
    true_switches_sweep2: int
    true_switches_sweep3: int
    true_switches_pos_sweep2: int
    true_switches_neg_sweep2: int
    true_switches_pos_sweep3: int
    true_switches_neg_sweep3: int

# rule 3 diagnostics (riccardo-style direction counting)
    switches_to_right: int
    switches_to_left: int
    rule3_pos_to_right: int
    rule3_pos_to_left: int
    rule3_neg_to_right: int
    rule3_neg_to_left: int
    rule3_bias_threshold_mv: float

# rule 4 diagnostics
    elem_ratio_valid_bins: int
    elem_ratio_pass_bins: int
    elem_ratio_pass_frac: float
    elem_ratio_threshold: float
    elem_ratio_required_frac: float
    elem_ratio_fail_reason: str

    reject_reasons: List[str]

# plotting markers (untrue switches when spike-filtering is enabled)
    untrue_switch_points_fullsweeps: List[int]

@dataclass
class IVRecord:
    molecule: str
    npz_path: Path
    base_name: str
    V_mV: np.ndarray
    x_logG: np.ndarray
    z: np.ndarray
    segs: List[SweepSegment]
    decision: HysteresisDecision

# progress window

class ProgressWindow:
    def __init__(self, title: str = "Processing IVs"):
        self.root = tk.Tk()
        self.root.title(title)
```

```
155         self.root.geometry("520x170")
156         self.root.resizable(False, False)
157 
158         self.label_stage = ttk.Label(self.root, text="Starting...", font=("Segoe
                UI", 10))
159         self.label_stage.pack(pady=(14, 6))
160 
161         self.pb = ttk.Progressbar(self.root, orient="horizontal", length=470, mode
                ="determinate")
162         self.pb.pack(pady=(0, 6))
163 
164         self.label_counts = ttk.Label(self.root, text="0 / 0", font=("Segoe UI",
                10))
165         self.label_counts.pack()
166 
167         self.label_file = ttk.Label(self.root, text="", font=("Segoe UI", 9))
168         self.label_file.pack(pady=(10, 0))
169 
170         self._last_update = 0.0
171         self.root.update_idletasks()
172         self.root.update()
173 
174     def set_total(self, total: int):
175         self.pb["maximum"] = max(1, total)
176         self.pb["value"] = 0
177         self.label_counts.config(text=f"0 / {total}")
178         self.root.update_idletasks()
179         self.root.update()
180 
181     def update(self, i: int, total: int, stage: str, filename: str):
182         now = time.time()
183         if now - self._last_update < 0.02 and i < total:
184             return
185         self._last_update = now
186         self.label_stage.config(text=stage)
187         self.pb["value"] = i
188         self.label_counts.config(text=f"{i} / {total}")
189         self.label_file.config(text=filename)
190         self.root.update_idletasks()
191         self.root.update()
192 
193     def close(self):
194         try:
195             self.root.destroy()
196         except Exception:
197             pass
198 
199 # load npz
200 
201 def load_npz_arrays(npz_path: Path) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
202     data = np.load(npz_path, allow_pickle=True)
203     keys = set(data.files)
```

```
204
205     def pick(*candidates: str) -> Optional[str]:
206         for k in candidates:
207             if k in keys:
208                 return k
209         return None
210
211     kV = pick("V_mV", "V", "voltage_mV", "voltage")
212     kz = pick("z", "z_hat", "hard_label", "labels", "y_hat")
213     kx = pick("x_logG", "logG", "x", "x_t", "log10G")
214
215     if kV is None or kz is None:
216         raise KeyError(f"{npz_path.name}: missing required keys. Found keys: {
            sorted(keys)}")
217
218     V = np.asarray(data[kV]).astype(float).reshape(-1)
219     z = np.asarray(data[kz]).astype(int).reshape(-1)
220
221     if kx is not None:
222         x = np.asarray(data[kx]).astype(float).reshape(-1)
223     else:
224         kG = pick("G", "conductance", "G_S", "G_uS")
225         if kG is None:
226             raise KeyError(f"{npz_path.name}: missing x/logG and missing G. Found
                keys: {sorted(keys)}")
227         G = np.asarray(data[kG]).astype(float).reshape(-1)
228         eps = 1e-30
229         x = np.log10(np.maximum(np.abs(G), eps))
230
231     n = min(len(V), len(x), len(z))
232     return V[:n], x[:n], z[:n]
233
234 # sweep split
235
236 def _find_turning_points(V: np.ndarray) -> List[int]:
237     if len(V) < 10:
238         return [0, len(V) - 1]
239
240     dV = np.diff(V)
241     s = np.sign(dV)
242     for i in range(1, len(s)):
243         if s[i] == 0:
244             s[i] = s[i - 1]
245     for i in range(len(s) - 2, -1, -1):
246         if s[i] == 0:
247             s[i] = s[i + 1] if i + 1 < len(s) else 0
248
249     tp = [0]
250     for i in range(1, len(s)):
251         if s[i] == 0 or s[i - 1] == 0:
252             continue
253         if s[i] != s[i - 1]:
```

```
            tp.append(i)
    tp.append(len(V) - 1)

    if len(tp) <= 5:
        return tp

    interior = tp[1:-1]
    scored = sorted(interior, key=lambda idx: abs(V[idx]), reverse=True)
    chosen = sorted(scored[:3])
    return [0] + chosen + [len(V) - 1]

def split_sweep_segments(V_mV: np.ndarray, x_logG: np.ndarray, z: np.ndarray) ->
    List[SweepSegment]:
    n = len(V_mV)
    idx_all = np.arange(n, dtype=int)

    boundaries = _find_turning_points(V_mV)

    if len(boundaries) < 5:
        boundaries = [0, n // 4, n // 2, (3 * n) // 4, n - 1]
    elif len(boundaries) > 5:
        boundaries = boundaries[:4] + [n - 1]

    segs: List[SweepSegment] = []
    for b0, b1 in zip(boundaries[:-1], boundaries[1:]):
        if b1 <= b0:
            continue
        seg_idx = idx_all[b0:b1 + 1]
        segs.append(SweepSegment(idx=seg_idx, V_mV=V_mV[seg_idx], x_logG=x_logG[
            seg_idx], z=z[seg_idx]))

# ensure 4 segments
    if len(segs) != 4:
        if len(segs) > 4:
            merged = segs[:3]
            rest_idx = np.concatenate([s.idx for s in segs[3:]])
            merged.append(SweepSegment(idx=rest_idx, V_mV=V_mV[rest_idx], x_logG=
                x_logG[rest_idx], z=z[rest_idx]))
            segs = merged
        else:
            while len(segs) < 4:
                segs.append(SweepSegment(
                    idx=np.array([], dtype=int),
                    V_mV=np.array([], dtype=float),
                    x_logG=np.array([], dtype=float),
                    z=np.array([], dtype=int),
                ))
    return segs

# switch logic

def _switch_events(z: np.ndarray) -> List[int]:
```

```
303     # Switch event indices in the *new-state* convention: indices
304     if len(z) < 2:
305         return []
306     return (np.where(z[1:] != z[:-1])[0] + 1).astype(int).tolist()
307
308 def _switch_edges(z: np.ndarray) -> List[int]:
309     # Switch edge indices in Riccardo convention: indices i (0..n-
310     if len(z) < 2:
311         return []
312     return (np.where(z[:-1] != z[1:])[0]).astype(int).tolist()
313
314 def _apply_spike_filter(z: np.ndarray, switch_indices: List[int],
    min_run_after_switch: int) -> Tuple[List[int], List[int]]:
315     # Spike-filter for *new-state* indices (i where z[i]!=z[i-1])
316     if not switch_indices:
317         return [], []
318
319     true_sw: List[int] = []
320     untrue_sw: List[int] = []
321
322     for idx in switch_indices:
323         new_state = z[idx]
324         end = min(len(z), idx + 1 + min_run_after_switch)
325         window = z[idx:end]
326         if len(window) < 1:
327             untrue_sw.append(idx)
328             continue
329         if np.all(window == new_state):
330             true_sw.append(idx)
331         else:
332             untrue_sw.append(idx)
333
334     sw_sorted = sorted(switch_indices)
335     consecutive = set()
336     for a, b in zip(sw_sorted[:-1], sw_sorted[1:]):
337         if b == a + 1:
338             consecutive.add(a)
339             consecutive.add(b)
340
341     true_final: List[int] = []
342     for idx in true_sw:
343         if idx in consecutive:
344             untrue_sw.append(idx)
345         else:
346             true_final.append(idx)
347
348     return sorted(true_final), sorted(set(untrue_sw))
349
350 def _apply_spike_filter_edges(z: np.ndarray, edge_indices: List[int],
    min_run_after_switch: int) -> Tuple[List[int], List[int]]:
351     # Spike-filter for *edge* indices i where transition is betwee
352     if not edge_indices:
```

```
353         return [], []
354
355     true_sw: List[int] = []
356     untrue_sw: List[int] = []
357
358     for i in edge_indices:
359         new_state = z[i + 1]
360         end = min(len(z), (i + 1) + min_run_after_switch)
361         window = z[i + 1:end]
362         if window.size < 1:
363             untrue_sw.append(i)
364             continue
365         if np.all(window == new_state):
366             true_sw.append(i)
367         else:
368             untrue_sw.append(i)
369
370     sw_sorted = sorted(edge_indices)
371     consecutive = set()
372     for a, b in zip(sw_sorted[:-1], sw_sorted[1:]):
373         if b == a + 1:
374             consecutive.add(a)
375             consecutive.add(b)
376
377     true_final: List[int] = []
378     for i in true_sw:
379         if i in consecutive:
380             untrue_sw.append(i)
381         else:
382             true_final.append(i)
383
384     return sorted(true_final), sorted(set(untrue_sw))
385
386 # rules
387
388 def low_bias_dominance(seg: SweepSegment, low_bias_mv: float) -> Tuple[Optional[
    int], float]:
389     if seg.idx.size == 0:
390         return None, 0.0
391     mask = np.abs(seg.V_mV) <= low_bias_mv
392     if not np.any(mask):
393         return None, 0.0
394     zz = seg.z[mask]
395     p1 = float(np.mean(zz == 1))
396     p0 = 1.0 - p1
397     return (1, p1) if p1 >= p0 else (0, p0)
398
399 def count_switches_by_sign_in_segment(
400     seg: SweepSegment,
401     high_bias_mv: float,
402     enable_spike_filter: bool,
403     min_run_after_switch: int,
```

```
) -> Tuple[int, int, int, List[int]]:
    # Rule 2: count switches (new-state convention) and check sign
    if seg.idx.size < 2:
        return 0, 0, 0, []

    sw = _switch_events(seg.z)
    if enable_spike_filter:
        true_sw, untrue_sw = _apply_spike_filter(seg.z, sw, min_run_after_switch)
    else:
        true_sw, untrue_sw = sw, []

    n_true = len(true_sw)
    n_pos = 0
    n_neg = 0
    for i_local in true_sw:
        v = float(seg.V_mV[i_local])
        if v > high_bias_mv:
            n_pos += 1
        if v < -high_bias_mv:
            n_neg += 1

    untrue_original = [int(seg.idx[i]) for i in untrue_sw]
    return n_true, n_pos, n_neg, untrue_original

def _rule3_bias_threshold_mv(V_mV: np.ndarray) -> float:
    if RULE3_BIAS_THRESHOLD_MODE == "none":
        return 0.0
    if RULE3_BIAS_THRESHOLD_MODE == "fixed":
        return float(RULE3_BIAS_THRESHOLD_FIXED_MV)
    vmax = float(np.nanmax(np.abs(V_mV))) if V_mV.size else 0.0
    return float(vmax * float(RULE3_BIAS_THRESHOLD_MAX_FRAC))

def count_switches_rule3_riccardo_style(
    V_mV: np.ndarray,
    z: np.ndarray,
    enable_spike_filter: bool,
    min_run_after_switch: int,
) -> Tuple[int, int, int, int, int, int, float]:
    # Returns:
    if V_mV.size < 2:
        return 0, 0, 0, 0, 0, 0, 0.0

    bias_th = _rule3_bias_threshold_mv(V_mV)

    edges = _switch_edges(z)
    if enable_spike_filter:
        edges_true, _edges_untrue = _apply_spike_filter_edges(z, edges,
            min_run_after_switch)
    else:
        edges_true = edges

    ps_r = ps_l = ns_r = ns_l = 0
```

```
455
456     for i in edges_true:
457         if abs(float(V_mV[i])) < bias_th:
458             continue
459
460         if float(V_mV[i]) > 0:
461             if float(V_mV[i + 1]) > float(V_mV[i]):
462                 ps_r += 1
463             else:
464                 ps_l += 1
465         elif float(V_mV[i]) < 0:
466             if float(V_mV[i + 1]) > float(V_mV[i]):
467                 ns_r += 1
468             else:
469                 ns_l += 1
470
471     to_right = ps_r + ns_r
472     to_left = ps_l + ns_l
473     return to_right, to_left, ps_r, ps_l, ns_r, ns_l, bias_th
474
475 def rule4_elementwise_ratio(
476     V_mV: np.ndarray,
477     x_logG: np.ndarray,
478     z: np.ndarray,
479 ) -> Tuple[bool, int, int, float, str]:
480     # Rule 4: element-wise ratio by binning in signed V and compar
481     if V_mV.size == 0:
482         return False, 0, 0, 0.0, "Rule 4 failed: empty IV after cutoff."
483
484     mask = (np.abs(V_mV) >= float(ELEMENTWISE_RATIO_VMIN_MV)) & (np.abs(V_mV) <=
        float(ELEMENTWISE_RATIO_VMAX_MV))
485     if not np.any(mask):
486         return False, 0, 0, 0.0, "Rule 4 failed: no points in chosen V window."
487
488     Vw = V_mV[mask]
489     xw = x_logG[mask]
490     zw = z[mask]
491
492     vmin = float(np.min(Vw))
493     vmax = float(np.max(Vw))
494     if not np.isfinite(vmin) or not np.isfinite(vmax) or vmin == vmax:
495         return False, 0, 0, 0.0, "Rule 4 failed: degenerate voltage range in V
            window."
496
497     edges = np.linspace(vmin, vmax, int(ELEMENTWISE_RATIO_N_BINS) + 1)
498     bin_id = np.digitize(Vw, edges) - 1
499     bin_id = np.clip(bin_id, 0, len(edges) - 2)
500
501     valid_bins = 0
502     pass_bins = 0
503
504     for b in range(len(edges) - 1):
```

```
        inb = (bin_id == b)
        if not np.any(inb):
            continue

        xb = xw[inb]
        zb = zw[inb]

        x0 = xb[zb == 0]
        x1 = xb[zb == 1]

        if x0.size < int(ELEMENTWISE_RATIO_MIN_POINTS_PER_BIN_PER_STATE):
            continue
        if x1.size < int(ELEMENTWISE_RATIO_MIN_POINTS_PER_BIN_PER_STATE):
            continue

        m0 = float(np.median(x0))
        m1 = float(np.median(x1))
        ratio = 10.0 ** abs(m1 - m0)

        valid_bins += 1
        if ratio >= float(ELEMENTWISE_RATIO_THRESHOLD):
            pass_bins += 1

    if valid_bins < int(ELEMENTWISE_RATIO_MIN_VALID_BINS):
        return (
            False,
            valid_bins,
            pass_bins,
            0.0 if valid_bins == 0 else pass_bins / valid_bins,
            f"Rule 4 failed: too few valid bins (valid={valid_bins}, need>={
                ELEMENTWISE_RATIO_MIN_VALID_BINS}).",
        )

    frac = pass_bins / valid_bins
    if frac >= float(ELEMENTWISE_RATIO_REQUIRED_FRAC):
        return True, valid_bins, pass_bins, frac, ""
    return (
        False,
        valid_bins,
        pass_bins,
        frac,
        f"Rule 4 failed: only {frac:.3f} of bins pass (need>={
            ELEMENTWISE_RATIO_REQUIRED_FRAC:.3f}) "
        f"for ratio>={ELEMENTWISE_RATIO_THRESHOLD}.",
    )

def decide_hysteretic(
    segs: List[SweepSegment],
    V_mV: np.ndarray,
    x_logG: np.ndarray,
    z: np.ndarray,
) -> HysteresisDecision:
```

```
    reject: List[str] = []

    sweep2 = segs[1]
    sweep3 = segs[2]

    dom2, frac2 = low_bias_dominance(sweep2, LOW_BIAS_MV)
    dom3, frac3 = low_bias_dominance(sweep3, LOW_BIAS_MV)

    rule1_pass = True
    if dom2 is None or dom3 is None:
        rule1_pass = False
        reject.append("Rule 1 failed: no low-bias points in one/both full sweeps.")
    else:
        if frac2 < LOW_BIAS_DOMINANCE_FRAC or frac3 < LOW_BIAS_DOMINANCE_FRAC:
            rule1_pass = False
            reject.append(
                f"Rule 1 failed: low-bias dominance below threshold "
                f"(sweep2={frac2:.3f}, sweep3={frac3:.3f}, thresh={LOW_BIAS_DOMINANCE_FRAC:.3f})."
            )
        if dom2 == dom3:
            rule1_pass = False
            reject.append(
                f"Rule 1 failed: dominant low-bias labels are the same (sweep2={dom2}, sweep3={dom3})."
            )

    n2, n2pos, n2neg, untrue2 = count_switches_by_sign_in_segment(
        sweep2, HIGH_BIAS_MV, ENABLE_SPIKE_FILTER_RULE2, MIN_RUN_AFTER_SWITCH
    )
    n3, n3pos, n3neg, untrue3 = count_switches_by_sign_in_segment(
        sweep3, HIGH_BIAS_MV, ENABLE_SPIKE_FILTER_RULE2, MIN_RUN_AFTER_SWITCH
    )

    rule2_pass = ((n2pos >= 1 and n2neg >= 1) or (n3pos >= 1 and n3neg >= 1))
    if not rule2_pass:
        reject.append(
            f"Rule 2 failed: no full sweep has both +HV and -HV switches "
            f"(sweep2 +{n2pos}/-{n2neg}, sweep3 +{n3pos}/-{n3neg}, HV={HIGH_BIAS_MV}mV)."
        )

    to_r, to_l, ps_r, ps_l, ns_r, ns_l, bias_th =
        count_switches_rule3_riccardo_style(
        V_mV, z, ENABLE_SPIKE_FILTER_RULE3, MIN_RUN_AFTER_SWITCH
    )
    rule3_pass = (to_r >= 1 and to_l >= 1)
    if not rule3_pass:
        reject.append(
            f"Rule 3 failed (Lucienne rule 5): need >=1 switch to right AND to left "
```

```
            f"after bias-threshold |V|>={bias_th:.2f}mV (to_right={to_r}, to_left
                ={to_l})."
        )

    r4_pass, r4_valid, r4_passbins, r4_frac, r4_reason = rule4_elementwise_ratio(
        V_mV, x_logG, z)
    rule4_pass = bool(r4_pass)
    if not rule4_pass:
        reject.append(r4_reason)

    is_hyst = rule1_pass and rule2_pass and rule3_pass and rule4_pass
    untrue_all = sorted(set(untrue2 + untrue3))

    return HysteresisDecision(
        is_hysteretic=is_hyst,
        rule1_pass=rule1_pass,
        rule2_pass=rule2_pass,
        rule3_pass=rule3_pass,
        rule4_pass=rule4_pass,

        low_bias_dom_frac_sweep2=frac2,
        low_bias_dom_frac_sweep3=frac3,
        low_bias_dom_label_sweep2=dom2,
        low_bias_dom_label_sweep3=dom3,

        true_switches_sweep2=n2,
        true_switches_sweep3=n3,
        true_switches_pos_sweep2=n2pos,
        true_switches_neg_sweep2=n2neg,
        true_switches_pos_sweep3=n3pos,
        true_switches_neg_sweep3=n3neg,

        switches_to_right=to_r,
        switches_to_left=to_l,
        rule3_pos_to_right=ps_r,
        rule3_pos_to_left=ps_l,
        rule3_neg_to_right=ns_r,
        rule3_neg_to_left=ns_l,
        rule3_bias_threshold_mv=bias_th,

        elem_ratio_valid_bins=int(r4_valid),
        elem_ratio_pass_bins=int(r4_passbins),
        elem_ratio_pass_frac=float(r4_frac),
        elem_ratio_threshold=float(ELEMENTWISE_RATIO_THRESHOLD),
        elem_ratio_required_frac=float(ELEMENTWISE_RATIO_REQUIRED_FRAC),
        elem_ratio_fail_reason=str(r4_reason),

        reject_reasons=reject,
        untrue_switch_points_fullsweeps=untrue_all,
    )

# summary stats
```

```
651
652 def compute_rule_stats(records: List[IVRecord]) -> str:
653     n = len(records)
654     if n == 0:
655         return "No IVs loaded."
656
657     def pct(k: int) -> float:
658         return 100.0 * k / n
659
660     r1 = sum(1 for r in records if r.decision.rule1_pass)
661     r2 = sum(1 for r in records if r.decision.rule2_pass)
662     r3 = sum(1 for r in records if r.decision.rule3_pass)
663     r4 = sum(1 for r in records if r.decision.rule4_pass)
664     hyst = sum(1 for r in records if r.decision.is_hysteretic)
665
666     lines = [
667         f"Total IVs: {n}",
668         f"Hysteretic (all rules pass): {hyst}/{n} ({pct(hyst):.2f}%)",
669         f"Rule 1 pass/fail: {r1}/{n} ({pct(r1):.2f}% pass, {100 - pct(r1):.2f}%
                fail)",
670         f"Rule 2 pass/fail: {r2}/{n} ({pct(r2):.2f}% pass, {100 - pct(r2):.2f}%
                fail)",
671         f"Rule 3 pass/fail: {r3}/{n} ({pct(r3):.2f}% pass, {100 - pct(r3):.2f}%
                fail)",
672         f"Rule 4 pass/fail: {r4}/{n} ({pct(r4):.2f}% pass, {100 - pct(r4):.2f}%
                fail)",
673         f"Rule 3 settings: bias_threshold_mode={RULE3_BIAS_THRESHOLD_MODE} (
                max_frac={RULE3_BIAS_THRESHOLD_MAX_FRAC}, fixed={
                RULE3_BIAS_THRESHOLD_FIXED_MV}mV), "
674         f"spike_filter={ENABLE_SPIKE_FILTER_RULE3} (MIN_RUN_AFTER_SWITCH={
                MIN_RUN_AFTER_SWITCH})",
675         f"Rule 4 settings: ratio>={ELEMENTWISE_RATIO_THRESHOLD} for >= {
                ELEMENTWISE_RATIO_REQUIRED_FRAC:.2f} of valid bins "
676         f"(V window: {ELEMENTWISE_RATIO_VMIN_MV}..{ELEMENTWISE_RATIO_VMAX_MV} mV,
                bins={ELEMENTWISE_RATIO_N_BINS})",
677         f"Spike-filter Rule 2: {ENABLE_SPIKE_FILTER_RULE2} (MIN_RUN_AFTER_SWITCH={
                MIN_RUN_AFTER_SWITCH})",
678     ]
679     return "\n".join(lines)
680
681 # csv output
682
683 def write_csv(path: Path, rows: List[Dict[str, object]]):
684     path.parent.mkdir(parents=True, exist_ok=True)
685     if not rows:
686         return
687     cols = list(rows[0].keys())
688     with path.open("w", newline="", encoding="utf-8") as f:
689         w = csv.DictWriter(f, fieldnames=cols)
690         w.writeheader()
691         for r in rows:
692             w.writerow(r)
```

```
693
694 # viewer
695
696 def segment_direction(seg: SweepSegment) -> Optional[str]:
697     if seg.idx.size < 2:
698         return None
699     dv = np.diff(seg.V_mV)
700     m = float(np.nanmean(dv))
701     if m > 0:
702         return "right"
703     if m < 0:
704         return "left"
705     return None
706
707 class ViewerApp:
708     def __init__(self, records: List[IVRecord], summary_text: str):
709         self.records_all = records
710         self.filtered_records: List[IVRecord] = []
711         self.selected_index = 0
712         self.summary_text = summary_text
713
714         self.root = tk.Tk()
715         self.root.title("Hysteretic Filter Viewer (Rules 1–4)")
716         self.root.geometry("1320x920")
717
718         left = ttk.Frame(self.root)
719         left.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)
720
721         ttk.Label(left, text="Summary", font=("Segoe UI", 11, "bold")).pack(anchor
                ="w")
722         self.text_summary = tk.Text(left, width=56, height=10, wrap="word")
723         self.text_summary.pack(fill=tk.X)
724         self.text_summary.insert("1.0", self.summary_text)
725         self.text_summary.config(state="disabled")
726
727         ttk.Label(left, text="").pack()
728
729         ttk.Label(left, text="Hysteretic filter", font=("Segoe UI", 11, "bold")).
                pack(anchor="w")
730         self.var_show_hyst = tk.BooleanVar(value=False)
731         self.var_show_non = tk.BooleanVar(value=False)
732         ttk.Checkbutton(left, text="Show only hysteretic", variable=self.
                var_show_hyst, command=self.apply_filter).pack(anchor="w")
733         ttk.Checkbutton(left, text="Show only non-hysteretic", variable=self.
                var_show_non, command=self.apply_filter).pack(anchor="w")
734
735         ttk.Label(left, text="").pack()
736
737         ttk.Label(left, text="Rule filters (Any/Pass/Fail)", font=("Segoe UI", 11,
                "bold")).pack(anchor="w")
738         self.rule_filter_values = ["Any", "Pass", "Fail"]
739         self.var_r1 = tk.StringVar(value="Any")
```

```
740             self.var_r2 = tk.StringVar(value="Any")
741             self.var_r3 = tk.StringVar(value="Any")
742             self.var_r4 = tk.StringVar(value="Any")
743             self._add_rule_filter_row(left, "Rule 1", self.var_r1)
744             self._add_rule_filter_row(left, "Rule 2", self.var_r2)
745             self._add_rule_filter_row(left, "Rule 3", self.var_r3)
746             self._add_rule_filter_row(left, "Rule 4", self.var_r4)
747             ttk.Button(left, text="Apply rule filters", command=self.apply_filter).
                    pack(anchor="w", pady=(6, 0))
748
749             ttk.Label(left, text="").pack()
750
751             ttk.Label(left, text="Files", font=("Segoe UI", 11, "bold")).pack(anchor="
                    w")
752             self.listbox = tk.Listbox(left, width=56, height=20)
753             self.listbox.pack(fill=tk.Y)
754             self.listbox.bind("<<ListboxSelect>>", self.on_select)
755
756             ttk.Label(left, text="").pack()
757             ttk.Label(left, text="Diagnostics (why failed)", font=("Segoe UI", 11, "
                    bold")).pack(anchor="w")
758             self.text_diag = tk.Text(left, width=56, height=22, wrap="word")
759             self.text_diag.pack(fill=tk.BOTH, expand=True)
760
761             right = ttk.Frame(self.root)
762             right.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True, padx=10, pady=10)
763
764             self.fig = plt.Figure(figsize=(8.8, 7.2), dpi=110)
765             self.ax = self.fig.add_subplot(111)
766             self.ax.set_xlabel("V (mV)")
767             self.ax.set_ylabel("log10(|G|)")
768
769             self.canvas = FigureCanvasTkAgg(self.fig, master=right)
770             self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
771
772             self.apply_filter(initial=True)
773
774         def _add_rule_filter_row(self, parent, label: str, var: tk.StringVar):
775             row = ttk.Frame(parent)
776             row.pack(fill=tk.X, pady=2)
777             ttk.Label(row, text=label, width=8).pack(side=tk.LEFT)
778             cb = ttk.Combobox(row, textvariable=var, values=self.rule_filter_values,
                    state="readonly", width=8)
779             cb.pack(side=tk.LEFT)
780
781         def _rule_match(self, desired: str, actual_pass: bool) -> bool:
782             if desired == "Any":
783                 return True
784             if desired == "Pass":
785                 return actual_pass
786             if desired == "Fail":
787                 return not actual_pass
```

```
788         return True
789
790     def apply_filter(self, initial: bool = False):
791         show_h = self.var_show_hyst.get()
792         show_n = self.var_show_non.get()
793
794         r1f = self.var_r1.get()
795         r2f = self.var_r2.get()
796         r3f = self.var_r3.get()
797         r4f = self.var_r4.get()
798
799         recs = self.records_all
800
801         if show_h and not show_n:
802             recs = [r for r in recs if r.decision.is_hysteretic]
803         elif show_n and not show_h:
804             recs = [r for r in recs if not r.decision.is_hysteretic]
805
806         recs2: List[IVRecord] = []
807         for r in recs:
808             d = r.decision
809             if not self._rule_match(r1f, d.rule1_pass):
810                 continue
811             if not self._rule_match(r2f, d.rule2_pass):
812                 continue
813             if not self._rule_match(r3f, d.rule3_pass):
814                 continue
815             if not self._rule_match(r4f, d.rule4_pass):
816                 continue
817             recs2.append(r)
818
819         self.filtered_records = recs2
820
821         self.listbox.delete(0, tk.END)
822         for r in self.filtered_records:
823             tag = "HYST" if r.decision.is_hysteretic else "NO"
824             self.listbox.insert(tk.END, f"[{tag}] {r.molecule} | {r.base_name}")
825
826         if self.filtered_records:
827             self.listbox.selection_clear(0, tk.END)
828             self.listbox.selection_set(0)
829             self.listbox.activate(0)
830             self.selected_index = 0
831             self.render_selected()
832         else:
833             self.ax.clear()
834             self.ax.set_xlabel("V (mV)")
835             self.ax.set_ylabel("log10(|G|)")
836             self.ax.set_title("No files match current filters")
837             self.canvas.draw()
838             self.text_diag.delete("1.0", tk.END)
```

```
839             self.text_diag.insert(tk.END, "No files match the current filter
                    selection.\n")
840             self.text_diag.insert(tk.END, "Set rule filters back to Any to widen.\
                    n")
841
842         if not initial:
843             self.root.update_idletasks()
844
845     def on_select(self, _evt=None):
846         try:
847             idx = int(self.listbox.curselection()[0])
848         except Exception:
849             return
850         self.selected_index = idx
851         self.render_selected()
852
853     def render_selected(self):
854         if not self.filtered_records:
855             return
856         rec = self.filtered_records[self.selected_index]
857         d = rec.decision
858
859         V = rec.V_mV
860         x = rec.x_logG
861         z = rec.z
862
863         self.ax.clear()
864         self.ax.set_xlabel("V (mV)")
865         self.ax.set_ylabel("log10(|G|)")
866         self.ax.set_title(f"{rec.molecule} | {rec.base_name} | hysteretic={d.
                is_hysteretic}")
867
868 # lines per segment (blue right, red left)
869         for seg in rec.segs:
870             if seg.idx.size < 2:
871                 continue
872             direction = segment_direction(seg)
873             c = "blue" if direction == "right" else ("red" if direction == "left"
                    else "gray")
874             self.ax.plot(seg.V_mV, seg.x_logG, linewidth=LINE_WIDTH, alpha=
                    LINE_ALPHA, color=c)
875
876 # dots by state
877         n = len(V)
878         if n > PLOT_DOWNSAMPLE_MAX_POINTS:
879             step = max(1, n // PLOT_DOWNSAMPLE_MAX_POINTS)
880             Vp, xp, zp = V[::step], x[::step], z[::step]
881         else:
882             Vp, xp, zp = V, x, z
883
884         m0 = (zp == 0)
885         m1 = (zp == 1)
```

```
886         self.ax.scatter(Vp[m0], xp[m0], s=DOT_SIZE, alpha=DOT_ALPHA, c="green",
                marker="o", label="state 0")
887         self.ax.scatter(Vp[m1], xp[m1], s=DOT_SIZE, alpha=DOT_ALPHA, c="purple",
                marker="o", label="state 1")
888
889 # mark untrue switches (from full sweeps) if spike-filter enabled for rule2
890         if ENABLE_SPIKE_FILTER_RULE2 and d.untrue_switch_points_fullsweeps:
891             idxs = d.untrue_switch_points_fullsweeps
892             self.ax.scatter(V[idxs], x[idxs], s=40, c="black", marker="^", label="
                    filtered spike-switch")
893
894         self.ax.legend(loc="best")
895         self.canvas.draw()
896
897 # diagnostics
898         self.text_diag.delete("1.0", tk.END)
899         self.text_diag.insert(tk.END, f"NPZ: {rec.npz_path}\n\n")
900
901         self.text_diag.insert(tk.END, f"Rule 1 pass: {d.rule1_pass}\n")
902         self.text_diag.insert(tk.END, f"  sweep2 dom label/frac: {d.
                low_bias_dom_label_sweep2} / {d.low_bias_dom_frac_sweep2:.3f}\n")
903         self.text_diag.insert(tk.END, f"  sweep3 dom label/frac: {d.
                low_bias_dom_label_sweep3} / {d.low_bias_dom_frac_sweep3:.3f}\n\n")
904
905         self.text_diag.insert(tk.END, f"Rule 2 pass: {d.rule2_pass}\n")
906         self.text_diag.insert(tk.END, f"  sweep2 true switches: total={d.
                true_switches_sweep2}, +HV={d.true_switches_pos_sweep2}, -HV={d.
                true_switches_neg_sweep2}\n")
907         self.text_diag.insert(tk.END, f"  sweep3 true switches: total={d.
                true_switches_sweep3}, +HV={d.true_switches_pos_sweep3}, -HV={d.
                true_switches_neg_sweep3}\n\n")
908
909         self.text_diag.insert(tk.END, f"Rule 3 pass: {d.rule3_pass}  (Lucienne
                rule 5, Riccardo-style)\n")
910         self.text_diag.insert(tk.END, f"  bias threshold: |V| >= {d.
                rule3_bias_threshold_mv:.2f} mV\n")
911         self.text_diag.insert(tk.END, f"  to_right switches: {d.switches_to_right}
                 (pos {d.rule3_pos_to_right}, neg {d.rule3_neg_to_right})\n")
912         self.text_diag.insert(tk.END, f"  to_left switches: {d.switches_to_left} (
                pos {d.rule3_pos_to_left}, neg {d.rule3_neg_to_left})\n")
913         self.text_diag.insert(tk.END, f"  spike filter: {ENABLE_SPIKE_FILTER_RULE3
                } (MIN_RUN_AFTER_SWITCH={MIN_RUN_AFTER_SWITCH})\n\n")
914
915         self.text_diag.insert(tk.END, f"Rule 4 pass: {d.rule4_pass}  (element-wise
                 ratio)\n")
916         self.text_diag.insert(tk.END, f"  ratio threshold: {d.elem_ratio_threshold
                }\n")
917         self.text_diag.insert(tk.END, f"  required fraction: {d.
                elem_ratio_required_frac:.3f}\n")
918         self.text_diag.insert(tk.END, f"  valid bins: {d.elem_ratio_valid_bins}\n"
                )
919         self.text_diag.insert(tk.END, f"  pass bins: {d.elem_ratio_pass_bins}\n")
```

```
920             self.text_diag.insert(tk.END, f"  pass fraction: {d.elem_ratio_pass_frac
                    :.3f}\n")
921             self.text_diag.insert(tk.END, f"  V window: {ELEMENTWISE_RATIO_VMIN_MV}..{
                    ELEMENTWISE_RATIO_VMAX_MV} mV\n\n")
922 
923             self.text_diag.insert(tk.END, f"Spike-filter Rule 2: {
                    ENABLE_SPIKE_FILTER_RULE2} (MIN_RUN_AFTER_SWITCH={MIN_RUN_AFTER_SWITCH
                    })\n\n")
924 
925             if d.is_hysteretic:
926                 self.text_diag.insert(tk.END, "FINAL: ACCEPTED as hysteretic.\n")
927             else:
928                 self.text_diag.insert(tk.END, "FINAL: REJECTED.\n\nFailed by:\n")
929                 for rr in d.reject_reasons:
930                     self.text_diag.insert(tk.END, f"- {rr}\n")
931 
932     def run(self):
933         self.root.mainloop()
934 
935 # pipeline
936 
937 def gather_npz_files(molecule_dir: Path) -> List[Path]:
938     labels_dir = molecule_dir / FINALIZED_LABELS_DIRNAME
939     if not labels_dir.exists():
940         return []
941     return sorted([p for p in labels_dir.glob(NPZ_GLOB) if p.is_file()])
942 
943 def enforce_v_cutoff(V_mV: np.ndarray, x: np.ndarray, z: np.ndarray, cutoff_mv:
        float) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
944     mask = np.abs(V_mV) >= cutoff_mv
945     return V_mV[mask], x[mask], z[mask]
946 
947 def build_records_with_progress(progress: ProgressWindow) -> List[IVRecord]:
948     all_npz: List[Tuple[str, Path]] = []
949     for mol_dir in MOLECULE_DIRS:
950         mol_name = mol_dir.name
951         for f in gather_npz_files(mol_dir):
952             all_npz.append((mol_name, f))
953 
954     total = len(all_npz)
955     progress.set_total(total)
956 
957     records: List[IVRecord] = []
958     for i, (mol_name, npz_path) in enumerate(tqdm(all_npz, desc="Processing NPZ
            files", unit="file"), start=1):
959         progress.update(i, total, stage="Loading + filtering + deciding hysteresis
                ", filename=npz_path.name)
960 
961         V_raw, x_raw, z_raw = load_npz_arrays(npz_path)
962         V, x, z = enforce_v_cutoff(V_raw, x_raw, z_raw, V_CUTOFF_MV)
963 
964         segs = split_sweep_segments(V, x, z)
```

```
965         decision = decide_hysteretic(segs, V, x, z)
966
967         records.append(
968             IVRecord(
969                 molecule=mol_name,
970                 npz_path=npz_path,
971                 base_name=npz_path.stem,
972                 V_mV=V,
973                 x_logG=x,
974                 z=z,
975                 segs=segs,
976                 decision=decision,
977             )
978         )
979
980     return records
981
982 def write_summaries(records: List[IVRecord]) -> str:
983     rows_global: List[Dict[str, object]] = []
984     per_mol: Dict[str, List[Dict[str, object]]] = {}
985     simple_rows: List[Dict[str, object]] = []
986
987     for r in records:
988         d = r.decision
989
990         row = {
991             "molecule": r.molecule,
992             "file": r.npz_path.name,
993             "is_hysteretic": int(d.is_hysteretic),
994
995             "rule1_pass": int(d.rule1_pass),
996             "rule2_pass": int(d.rule2_pass),
997             "rule3_pass": int(d.rule3_pass),
998             "rule4_pass": int(d.rule4_pass),
999
1000            "low_bias_dom_label_sweep2": d.low_bias_dom_label_sweep2,
1001            "low_bias_dom_frac_sweep2": round(d.low_bias_dom_frac_sweep2, 6),
1002            "low_bias_dom_label_sweep3": d.low_bias_dom_label_sweep3,
1003            "low_bias_dom_frac_sweep3": round(d.low_bias_dom_frac_sweep3, 6),
1004
1005            "true_switches_sweep2": d.true_switches_sweep2,
1006            "true_switches_pos_sweep2": d.true_switches_pos_sweep2,
1007            "true_switches_neg_sweep2": d.true_switches_neg_sweep2,
1008            "true_switches_sweep3": d.true_switches_sweep3,
1009            "true_switches_pos_sweep3": d.true_switches_pos_sweep3,
1010            "true_switches_neg_sweep3": d.true_switches_neg_sweep3,
1011
1012 # rule 3
1013            "rule3_bias_threshold_mv": round(d.rule3_bias_threshold_mv, 6),
1014            "switches_to_right": d.switches_to_right,
1015            "switches_to_left": d.switches_to_left,
1016            "rule3_pos_to_right": d.rule3_pos_to_right,
```

```
            "rule3_pos_to_left": d.rule3_pos_to_left,
            "rule3_neg_to_right": d.rule3_neg_to_right,
            "rule3_neg_to_left": d.rule3_neg_to_left,

# rule 4
            "elem_ratio_valid_bins": d.elem_ratio_valid_bins,
            "elem_ratio_pass_bins": d.elem_ratio_pass_bins,
            "elem_ratio_pass_frac": round(d.elem_ratio_pass_frac, 6),
            "elem_ratio_threshold": d.elem_ratio_threshold,
            "elem_ratio_required_frac": d.elem_ratio_required_frac,

            "reject_reasons": " | ".join(d.reject_reasons),
        }

        rows_global.append(row)
        per_mol.setdefault(r.molecule, []).append(row)

        simple_rows.append(
            {
                "molecule": r.molecule,
                "file": r.npz_path.name,
                "hysteretic": int(d.is_hysteretic),
            }
        )

    write_csv(OUTPUT_DIR / "hysteretic_summary_all.csv", rows_global)
    for mol, rows in per_mol.items():
        write_csv(OUTPUT_DIR / f"hysteretic_summary_{mol}.csv", rows)
    write_csv(SIMPLE_LABELS_CSV, simple_rows)

    n = len(records)
    nh = sum(1 for r in records if r.decision.is_hysteretic)
    pct = 0.0 if n == 0 else 100.0 * nh / n
    return (
        f"Overall: {nh}/{n} hysteretic ({pct:.2f}%).\n"
        f"Detailed CSV: {OUTPUT_DIR / 'hysteretic_summary_all.csv'}\n"
        f"Simple labels CSV: {SIMPLE_LABELS_CSV}"
    )

def main():
    missing = [p for p in MOLECULE_DIRS if not p.exists()]
    if missing:
        msg = "Missing molecule directories:\n" + "\n".join(str(p) for p in
            missing) + "\n\nFix MOLECULE_DIRS at top of file."
        messagebox.showerror("Path error", msg)
        return

    progress = ProgressWindow()
    try:
        records = build_records_with_progress(progress)
    finally:
        progress.close()
```

```
1068
1069        summary_text = compute_rule_stats(records)
1070        print("\n" + summary_text + "\n")
1071
1072        print(write_summaries(records))
1073
1074        app = ViewerApp(records, summary_text=summary_text)
1075        app.run()
1076
1077    if __name__ == "__main__":
1078        main()
```

Listing A.5: Hysteretic filter script