Report

Application for Predictive Analysis
of Temporal Events

Architecture, Development, and Methodology

ML Analytics

Version:	2.0
Date:	October 28, 2025
Status:	Research
Classification:	Technical Documentation

Naoufal Kribet

Document Information

Title: Report - Application for Predictive Analysis of Temporal Events
Version: 2.0
Publication Date: October 28, 2025
Classification: Documentation and Methodology

Version History

Version	Date	Author	Changes
1.0	2025-08-08	Naoufal Kribet	Initial version
2.0	October 28, 2025	Naoufal Kribet	Complete refactoring, added theory

Disclaimer: This document contains technical information regarding the architecture and internal workings of the implemented methodology.

Contents

List of Figures

List of Tables

Executive Summary

This guide serves as the technical reference documentation for the predictive analysis application for temporal events, a software system developed for the analysis and prediction of events based on time series data.

The application implements a Model-View-Controller (MVC) architecture, using PyQt6 for the user interface and established machine learning libraries for predictive analysis. It incorporates machine learning algorithms, feature engineering techniques, and an explainability mechanism based on SHAP.

Key Features:

- A decoupled architecture to facilitate maintenance and evolution
- Support for multiple machine learning algorithms
- An intuitive user interface with interactive visualizations
- Asynchronous handling of computationally intensive tasks
- An integrated explainability system for model interpretation
- An automated deployment pipeline

Target Audience: Internship Supervisor Claudia Corradino

Chapter 1

Introduction and Context

1.1 Scientific and Technical Context

Anticipating critical events from sequential data is a fundamental problem at the intersection of statistics, signal processing, and artificial intelligence. This field of research has critical applications in diverse areas such as early warning systems (EWS), predictive maintenance, financial risk management, and, as in this case, the analysis of complex geophysical phenomena like volcanic activity.

This application is situated within the paradigm of **supervised machine learning** applied to **time series analysis**. The goal is to move beyond classical statistical models by leveraging non-linear algorithms capable of capturing complex dependencies and subtle dynamics within the data.

Definition - Time Series: A time series is a realization of a stochastic process, manifesting as a sequence of observations $\{x_t\}_{t \in T}$ indexed by an ordered set T, typically time. In the context of our study, each observation $x_t \in \mathbb{R}$ represents the Volcanic Radiated Power (VRP) of a volcano at time t, a proxy measure of its energy activity.

1.1.1 The Problem of Event Prediction: A Formalization

Event prediction extends beyond simple binary classification; it involves modeling the transition of a system between different dynamic states. Formally, let $\mathcal{S} = \{s_1, s_2, \ldots, s_k\}$ be a discrete set of system states (e.g., *Calm*, *Pre-Event*, *High-Paroxysm*, *Post-Event*). Given a history of observations at time t, $\mathbf{h}_t = (x_{t-n+1}, \ldots, x_t) \in \mathbb{R}^n$, the objective is to build a prediction function f that estimates the probability distribution over the possible states at a future time horizon Δt.

$$f(\mathbf{h}_t) = P(S_{t+\Delta t} = s_j \mid \mathbf{h}_t) \quad \forall s_j \in \mathcal{S} \tag{1.1}$$

The final prediction, $\hat{s}_{t+\Delta t}$, is then the state that maximizes this posterior probability:

$$\hat{s}_{t+\Delta t} = \arg\max_{s_j \in \mathcal{S}} f(\mathbf{h}_t) \tag{1.2}$$

Our application implements this prediction through a two-stage architecture: a binary **Detector** that estimates $P(\text{State} \in \{\text{Active}\})$ and a multi-class **Segmenter** that refines the prediction within the detected active blocks.

1.1.2 Fundamental Technical Challenges

The sequential and non-stationary nature of time series data presents methodological challenges that invalidate the naive application of standard machine learning techniques.

1. **Feature Engineering:** *Problem:* The raw data (x_t) resides in a low-dimensional space where system states are not linearly separable. *Implemented Solution:* Transform the time series into a high-dimensional feature space through **multi-scale analysis**, where each point is described by statistics, trends, and dynamic properties calculated over various time windows. This transformation aims to make the classification problem more tractable for the model.

2. **Class Imbalance:** *Problem:* Critical events and their precursors are, by definition, rare phenomena. A standard algorithm optimized for overall accuracy will converge to a trivial classifier that systematically predicts the majority class (*Calm*). *Implemented Solution:* A dual strategy is adopted: (1) **Class weighting** (`class_weight`) to more heavily penalize errors on minority classes, and (2) **Optimization of a custom metric** (the F1-score of the *Pre-Event* class) during hyperparameter search.

3. **Temporal Dependence and Causality:** *Problem:* Observations are not independent and identically distributed (IID). Using standard k-fold cross-validation would lead to data leakage, where the model is trained on future information, producing unrealistic and overly optimistic performance estimates. *Implemented Solution:* Rigorous application of **time series cross-validation** (`TimeSeriesSplit`) and temporal weighting (`sample_weight`) to ensure the model is always validated on data strictly subsequent to its training data.

4. **Noise and Uncertainty:** *Problem:* Physical sensors introduce noise and artifacts that can be misinterpreted as precursor signals, leading to false alarms. *Implemented Solution:* Implementation of a post-processing **persistence filter**, which removes isolated positive predictions considered to be noise, thereby improving the robustness and precision of the final alerts.

5. **Interpretability and Scientific Validation:** *Problem:* A "black box" model, even if performant, is scientifically unsatisfactory and does not allow for the validation of underlying physical hypotheses. *Implemented Solution:* Integration of an eXplainable AI (XAI) module based on **Shapley values (SHAP)**, derived from cooperative game theory. This technique quantifies the contribution of each feature to an individual prediction, making the model transparent and auditable.

1.2 System Objectives

The developed application addresses several strategic objectives that go beyond the simple implementation of an algorithm.

- **Scientific Objective:** To provide a **reproducible and extensible research framework** for testing hypotheses on the predictability of temporal events, allowing for rapid exploration of different models and feature configurations.

- **Technical Objective:** To support the **long-term viability** of the software by implementing established architectural principles (MVC, Dependency Inversion), thus facilitating its maintenance, extension, and testing.
- **Operational Objective:** To bridge the gap between domain experts (geophysicists, engineers) and Machine Learning specialists by offering an intuitive graphical interface that abstracts technical complexity while providing granular control over the experimental pipeline.
- **Methodological Objective:** To promote **scientific rigor** by integrating best practices for validation (temporal cross-validation) and interpretability (XAI) directly into the user's workflow.

1.3 Architectural Overview

The system's architecture is built around a clear separation of responsibilities, organized into four fundamental logical layers. This approach ensures low coupling and high cohesion of the modules.

- **Presentation Layer (View):** Manages the entire Human-Machine Interface (HMI). It is responsible for displaying data and results, and capturing user interactions. It contains no business logic.
- **Logic Layer (Controller):** Serves as the orchestrator. It receives requests from the View, translates them into calls to the business layer, and updates the View with the results. It is the seat of application logic and state management.
- **Business Layer (Model):** The core of the system. It contains the pure, interface-agnostic implementation of the Machine Learning algorithms, data processing pipelines, and feature extraction.
- **Data Layer (Infrastructure):** Manages model persistence, file system access, and configuration. It provides the necessary services to the other layers.

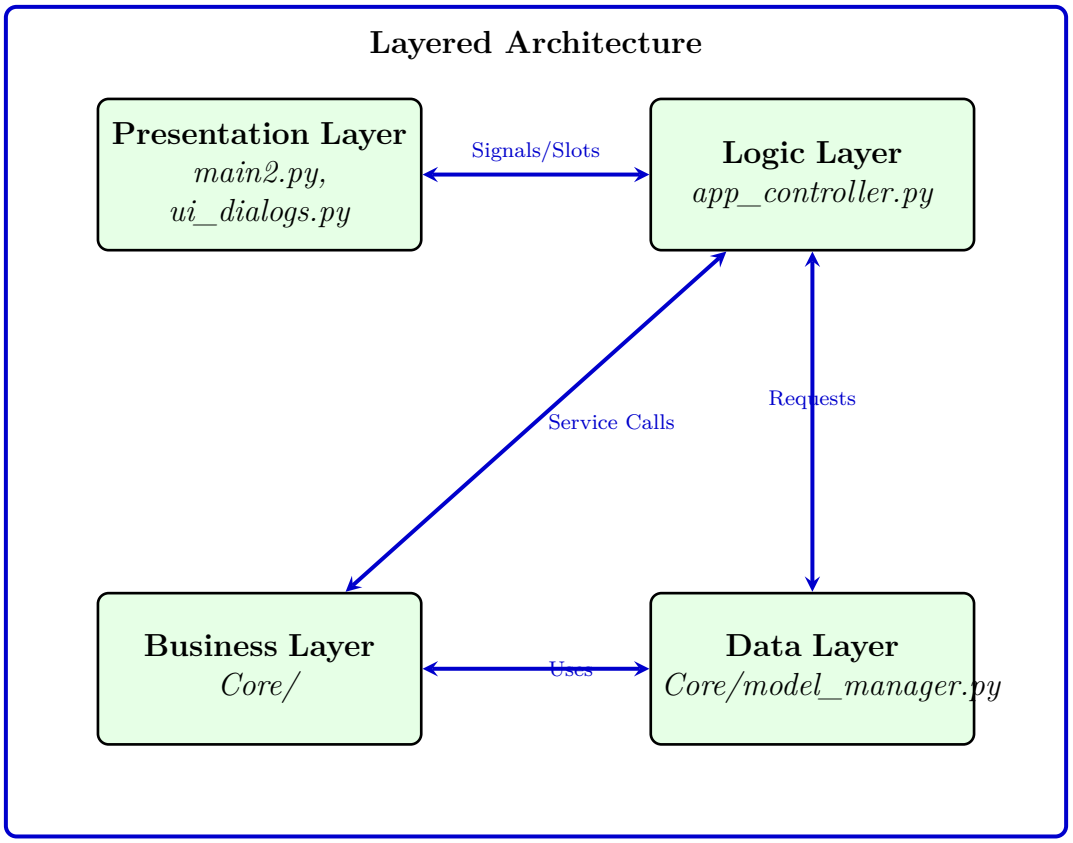

Figure 1.1: Layered architecture of the system, illustrating the separation of concerns and the main interaction flows.

Chapter 2

Theoretical Foundations

This chapter establishes the theoretical foundations upon which the predictive analysis system is built. We will first discuss the formal guarantees of machine learning, then detail the feature extraction methodology that bridges the gap between raw data and models, and finally present the implemented learning algorithms and explainability techniques.

2.1 Generalization Theory in Machine Learning

At the heart of supervised learning lies the challenge of **generalization**: the ability of a model, trained on a finite set of data, to make accurate predictions on new, unseen data. The following theoretical frameworks allow us to reason about the conditions that make this learning possible.

2.1.1 Probably Approximately Correct (PAC) Framework

The PAC framework, introduced by Valiant, formalizes the notion of successful learning. It does not guarantee a perfect prediction, but it ensures that with a sufficiently high probability (namely $1 - \delta$), the hypothesis h learned by the algorithm will have a low prediction error (less than ϵ) on the underlying data distribution $\mathcal{D}$.

Definition - PAC Learnability: Let $\mathcal{H}$ be a hypothesis space and $\mathcal{D}$ a probability distribution over the input space $\mathcal{X} \times \mathcal{Y}$. An algorithm $\mathcal{A}$ is said to PAC-learn the concept class $\mathcal{C}$ if, for all $\epsilon, \delta \in (0, 1)$, there exists a sample complexity $m_0(\epsilon, \delta)$ that is polynomial in $1/\epsilon$ and $1/\delta$, such that for any sample $\mathcal{S}$ of size $m \geq m_0$ drawn from $\mathcal{D}$, the algorithm $\mathcal{A}$ produces a hypothesis $h \in \mathcal{H}$ that satisfies, with a probability of at least $1 - \delta$:

$$R(h) = \mathbb{P}_{(\mathbf{x},y)\sim\mathcal{D}}[h(\mathbf{x}) \neq y] \leq \epsilon$$

Here, $R(h)$ is the generalization error, or true risk.

In our application, this framework provides a justification for the empirical approach taken: by collecting enough data (m), we can hope to train a model (h) whose performance on future volcanic data will be close to that observed during training.

2.1.2 Model Complexity and the Risk of Overfitting

A model's capacity to fit the training data is measured by the complexity of its hypothesis space $\mathcal{H}$, formalized by the **Vapnik-Chervonenkis (VC) dimension**.

$$\text{VC-dim}(\mathcal{H}) = \max\{|\mathcal{S}| : \mathcal{S} \subseteq \mathcal{X} \text{ is shattered by } \mathcal{H}\} \tag{2.1}$$

Statistical learning theory establishes a bound between the generalization error ($R(h)$) and the empirical error measured on the training sample ($R_{\text{emp}}(h)$):

$$R(h) \leq R_{\text{emp}}(h) + \Omega\left(\sqrt{\frac{\text{VC-dim}(\mathcal{H})}{m}}\right) \tag{2.2}$$

This bound illustrates the fundamental bias-variance trade-off. Complex models like Random Forests or Gradient Boosting have a very high (or even infinite) VC dimension, allowing them to achieve a very low empirical error $R_{\text{emp}}(h)$. However, this increases the risk of **overfitting**, where the model memorizes the noise in the training sample at the expense of its ability to generalize. Rigorous validation (see Section 1.1.2) is therefore the primary practical tool used to control this theoretical risk.

2.2 Feature Extraction: From Signal to Information

Learning algorithms operate on feature vectors. The feature extraction process is therefore a critical step, consisting of projecting the raw time series into a representation space where predictive patterns are more explicit. This step incorporates domain knowledge into the model.

2.2.1 Time-Domain Analysis

The time series is transformed into a set of descriptors that quantify its local behavior.

1. **Statistical Moments:** These descriptors capture the shape of the VRP value distribution within a given window.
    - **Skewness:** Measures the asymmetry of the distribution. A positive value indicates a tail to the right, typical of sudden peaks followed by slower relaxations.
    - **Kurtosis:** Quantifies the "tailedness" of the distribution. A high value (*leptokurtic*) signals the presence of extreme values, or outliers, which may correspond to paroxysmal phases.
2. **Variability Features:** These measure the signal's dispersion and volatility.
    - **Coefficient of Variation (CV):** Normalizes the standard deviation by the mean, making the volatility measure comparable across periods with different background activity levels.
    - **Interquartile Range (IQR):** Measures the spread of the central 50
3. **Trend Analyses:** These capture the dynamics of the signal.
    - **Regression Slope:** Estimates the **local velocity** of the VRP signal. A strongly positive slope is a direct indicator of a rapid increase in released energy. The application also calculates the **acceleration** (difference of successive slopes) to detect regime changes.

2.2.2 Multi-Scale and Physics-Informed Feature Engineering

A key assumption in this approach is that precursory phenomena manifest at different time scales and are often non-stationary. An analysis over a single time window would be either too noisy (short window) or would average out crucial transient signals (long window). The multi-scale approach consists of systematically calculating a set of statistical and dynamical descriptors over a collection of windows of varying sizes $W = \{w_1, w_2, \ldots, w_k\}$.

For a history of observations $\mathbf{h}_t$, the final feature vector $\mathbf{\Phi}(\mathbf{h}_t)$ is the concatenation of all features computed for each window $w \in W$:

$$\mathbf{\Phi}(\mathbf{h}_t) = \left(\mathcal{F}_1^{(w_1)}, \ldots, \mathcal{F}_p^{(w_1)}, \mathcal{F}_1^{(w_2)}, \ldots, \mathcal{F}_p^{(w_k)}\right) \tag{2.3}$$

where $\mathcal{F}_i^{(w)}$ is the i-th feature calculated over a window of size w.

Critically, to inject domain knowledge, we augment this statistical set with **physics-informed features**. A key example is the slope of the log-transformed VRP, $\nabla(\log(V(t)))$, designed to linearize and capture the exponential energy growth characteristic of pre-eruptive feedback loops. This hybrid approach allows the learning model to access a rich, physically-grounded representation of the system's short-, medium-, and long-term dynamics.

2.2.3 Theoretical Analysis of Feature Importance Methods

The high-dimensional and correlated nature of the engineered feature space necessitates a rigorous approach to evaluating feature importance. The choice between the default Mean Decrease in Impurity (MDI) and Permutation Feature Importance (PFI) is not trivial and has profound implications for the scientific interpretation of the model.

Mean Decrease in Impurity (MDI): A Biased Heuristic

MDI, or Gini Importance, is an intrinsic metric calculated during the training of a Random Forest. It quantifies how much each feature contributes to reducing the weighted Gini impurity in the decision trees.

[Gini Impurity & MDI] For a node m containing samples from K classes, where p_{mk} is the proportion of class k samples, the Gini impurity is $G(m) = 1 - \sum_{k=1}^{K} p_{mk}^2$. The importance of a feature X_j, $\text{MDI}(X_j)$, is the total impurity decrease it provides, averaged over all trees where it is used for a split.

However, MDI is known to be biased. It systematically inflates the importance of high-cardinality features and, more critically for our use case, it is unreliable in the presence of multicollinearity.

[Importance Dilution under Multicollinearity] Let $\mathcal{S} = \{f_1, \ldots, f_k\}$ be a set of highly correlated features (i.e., $\forall i, j \in \mathcal{S}, \rho(f_i, f_j) > \tau$). If these features carry similar predictive information $\mathcal{I}_0$, the random feature sub-sampling process in a Random Forest distributes the selection probability across the set. Consequently, the total importance $\mathcal{I}_0$ is diluted, and the individual MDI score of each feature becomes an underestimate of the latent predictor's true importance: $\mathbb{E}[\text{MDI}(f_i)] \approx \mathcal{I}_0/k$.

Our empirical results perfectly illustrate this theoretical flaw. The correlation matrix in Figure 2.1 reveals strong positive correlations between features measuring signal amplitude (e.g., 'median10', 'median30', 'median50') and volatility (e.g., 'std10', 'std30').

The MDI plot in Figure 2.2 shows a smooth hierarchy where these correlated features all receive high but distinct scores. This is misleading: it masks the fact that a single physical concept—"signal amplitude"—is the dominant predictor, and its importance has been artificially split among its redundant proxies.

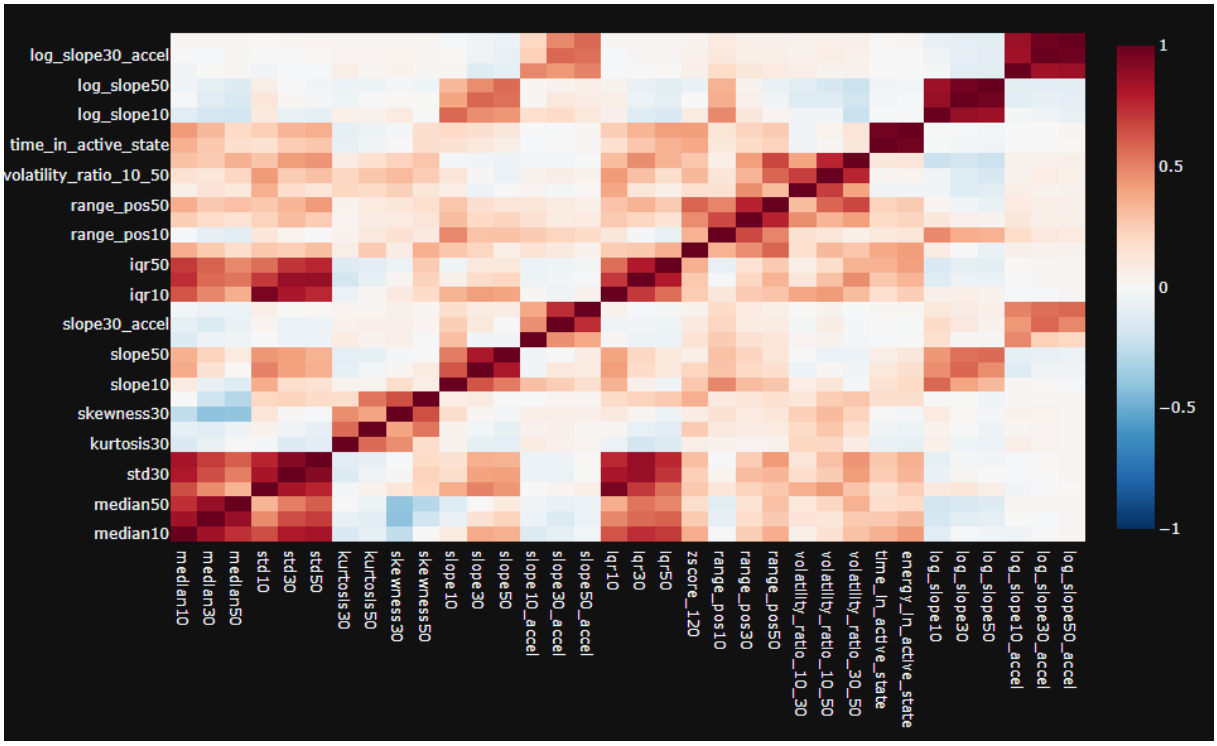

Figure 2.1: Pearson correlation matrix of the 34 engineered features. Dark red squares indicate blocks of high positive correlation, the source of multicollinearity.

Permutation Feature Importance (PFI): A Rigorous, Model-Agnostic Approach

To overcome the biases of MDI, we employ PFI. This method is model-agnostic and evaluates a feature's importance based on its impact on performance on an unseen (out-of-sample) dataset.

[Permutation Feature Importance] Let f be a trained model, (X, y) a held-out test set, and $s(y, \hat{y})$ a performance score (e.g., F1-Score on the 'Actif' class). The baseline score is $s_0 = s(y, f(X))$. For a feature j, we create a corrupted dataset $X_{\text{perm},j}$ by randomly permuting the j-th column of X. This breaks the feature-target link while preserving the feature's marginal distribution. The PFI is the measured drop in performance:

$$\text{PFI}(f_j) = s_0 - \mathbb{E}_\pi[s(y, f(X_{\text{perm},j}))] \tag{2.4}$$

The key advantage of PFI is its ability to reveal the true marginal contribution of a feature. It answers the scientifically relevant question: "How much predictive power do we lose if we lose access to this feature, given that we still have all the others?".

[Redundancy Revelation] If a feature f_i is highly correlated with another feature f_j already in the model, permuting f_i will have little effect on performance, as the model can

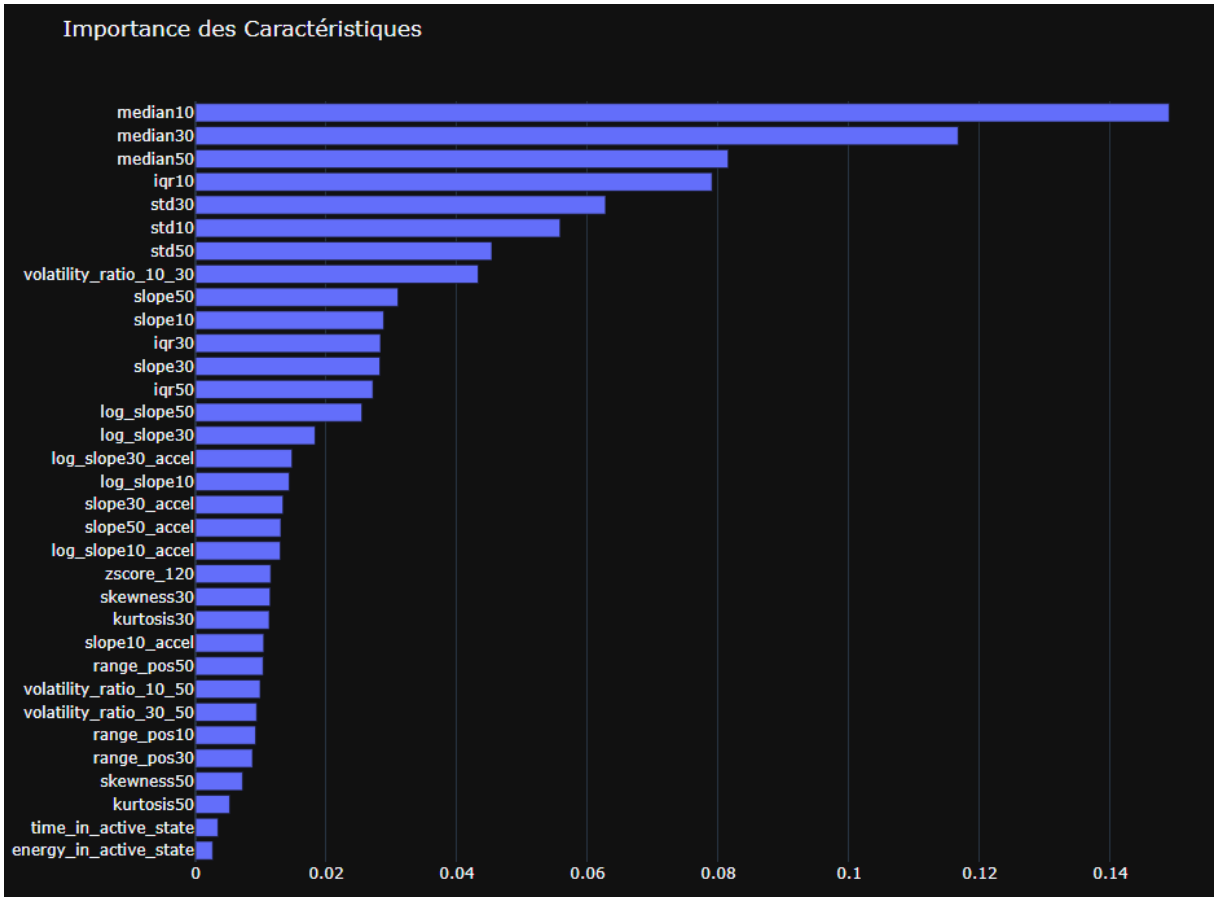

Figure 2.2: Feature importance as measured by Mean Decrease in Impurity (Gini). Note the smooth decay and the high ranking of multiple correlated features like ‘median10‘, ‘median30‘, and ‘median50‘.

still rely on f_j as a proxy. Consequently, its marginal contribution is low, and $\text{PFI}(f_i) \to 0$.

Theoretical Superiority of PFI in the Presence of Multicollinearity

The PFI results, shown in Figure 2.3, provide a starkly different and more insightful picture.

The PFI analysis reveals three critical insights that MDI obscured:

1. **A Clear Hierarchy:** A small subset of features is truly indispensable. ‘median10‘ is identified as the single most important predictor by a large margin. Its correlated counterparts, ‘median30‘ and ‘median50‘, have a significantly lower marginal importance, confirming their informational redundancy.

2. **Feature Uselessness:** The vast majority of features have an importance score close to zero, with error bars (standard deviation over permutations) crossing the zero-line. This is strong statistical evidence that they contribute no unique information and can be removed.

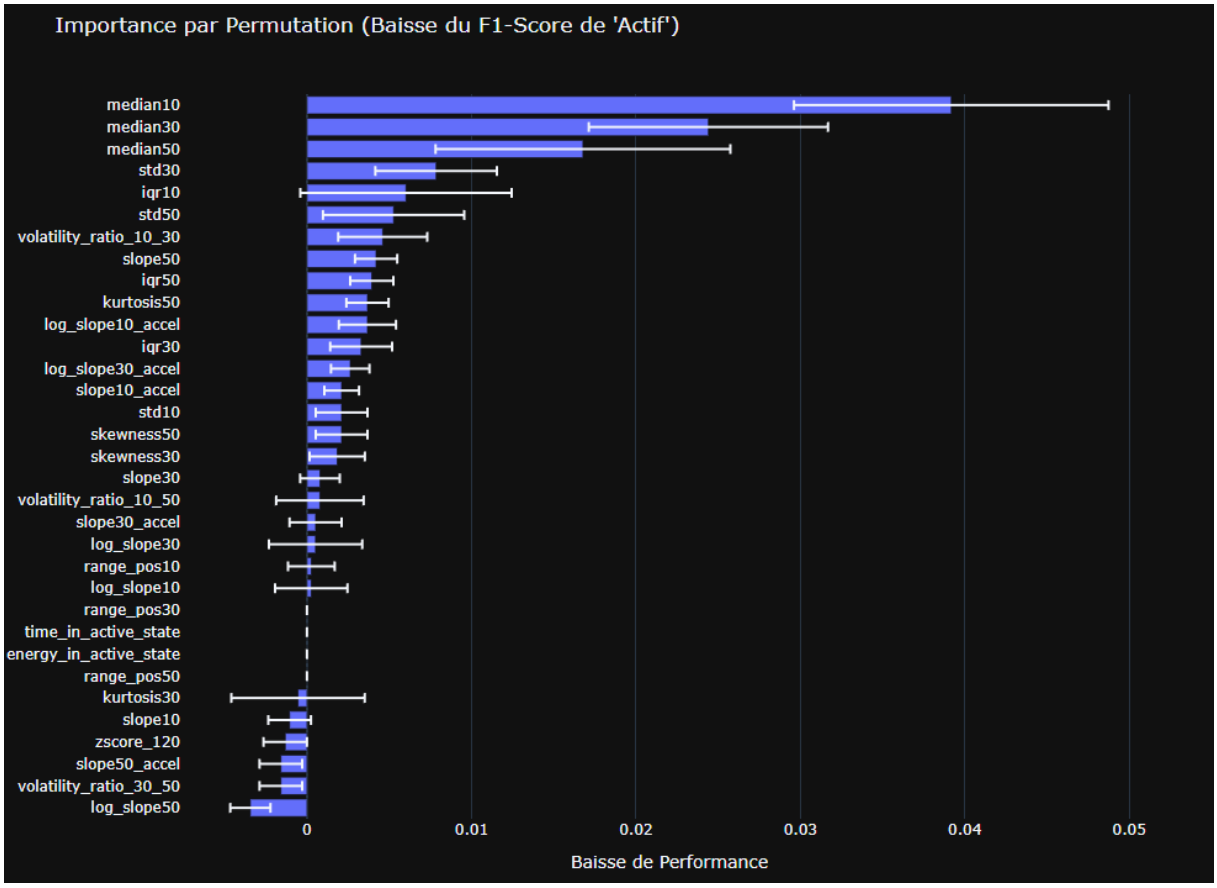

Figure 2.3: Feature importance as measured by Permutation, showing the drop in the F1-Score for the 'Actif' class on the test set. This reveals the true marginal contribution of each feature.

3. **Harmful Features:** Crucially, PFI identifies features with negative importance (e.g., '$zscore_{120}$', '$energy_{InactiveState}$'). This indicates that the model learned spurious correlations from these features on the training data, and they are actively harming its ability to generalize to the test set. Removing them is essential.

Conclusion: A Scientifically Grounded Choice

MDI is a biased, in-sample metric reflecting the model's internal structure. PFI is an unbiased, out-of-sample metric measuring a feature's true predictive value for generalization. Given the high degree of multicollinearity inherent in our feature engineering approach, PFI is the only methodologically sound choice for drawing reliable scientific conclusions about the underlying drivers of volcanic activity learned by our model.

We have implemented ensemble methods, which are known for their high predictive performance and robustness to noise.

2.2.4 Random Forests

Random Forests, introduced by Breiman, are ensembles of decision tree classifiers. They combat the overfitting inherent in a single tree through a dual randomization mechanism:

- **Bagging (Bootstrap Aggregating):** Each tree is trained on a subsample of the dataset, drawn with replacement.
- **Random Subspace:** At each node of the tree, the best split is sought not over the entire set of features, but over a random subset of them.

These two techniques decorrelate the trees, which reduces the variance of the final ensemble prediction, obtained by majority vote.

$$\hat{y} = \underset{c \in \mathcal{Y}}{\operatorname{argmax}} \sum_{b=1}^{B} \mathbb{I}(T_b(\mathbf{\Phi}(\mathbf{h}_t)) = c) \tag{2.5}$$

where $\mathbb{I}(\cdot)$ is the indicator function.

2.2.5 Gradient Boosting

Gradient Boosting is another ensemble technique that builds models in a sequential and additive manner. Each new weak model (typically a decision tree) is trained to correct the errors of the preceding models.

More formally, the algorithm minimizes a loss function $L(y, F(\mathbf{x}))$ using a gradient descent approach in the function space. At each step m, a new model h_m is trained to predict the pseudo-residual, which is the negative gradient of the loss function with respect to the current model's prediction F_{m-1}:

$$r_{im} = - \left[\frac{\partial L(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)} \right]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})} \tag{2.6}$$

The final model is then updated: $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \gamma_m h_m(\mathbf{x})$, where γ_m is a learning rate.

Our application uses **LightGBM**, a highly optimized implementation of Gradient Boosting that employs advanced techniques (GOSS, EFB) for increased training speed without a loss in performance.

2.3 Explainability (XAI) and Scientific Validation

In a scientific context, predictive performance alone is insufficient. It is important to be able to interpret the model's decisions to validate that it is learning relevant physical relationships and not statistical artifacts.

2.3.1 Shapley Values (SHAP)

To this end, our system integrates the SHAP (SHapley Additive exPlanations) methodology. Originating from cooperative game theory, it allows for the calculation of each feature's contribution to an individual prediction while respecting desirable axiomatic properties. The Shapley value ϕ_i for a feature i is its average marginal contribution to the prediction, averaged over all possible feature coalitions.

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [v(S \cup \{i\}) - v(S)] \tag{2.7}$$

where N is the set of all features and $v(S)$ is the model's prediction using only the subset of features S.

2.3.2 Axiomatic Properties and Practical Interpretation

The theoretical robustness of Shapley values is based on four axioms that have powerful practical implications for our analysis:

- **Efficiency:** The sum of the contributions of all features ($\sum \phi_i$) fully explains the difference between the prediction for a given instance and the average prediction over the entire dataset.
- **Symmetry:** Two features that have the same impact on the prediction will receive the same attribution, ensuring a form of fairness.
- **Dummy:** A feature that has no influence on the prediction will have a Shapley value of zero. This helps validate the usefulness of the engineered features.
- **Additivity:** Ensures that the explanation for an ensemble of models (like a Random Forest) is the weighted sum of the explanations of each individual model.

Thanks to SHAP, we can answer important questions such as: "Which indicators (e.g., an acceleration of the slope over a 30-point window) led the model to issue a *Pre-Event* alert at this specific moment?" Explainability thus transforms a predictive model into a tool for scientific investigation.

Chapter 3

Experimental Methodology

Predicting events in non-stationary and noisy time series is an intrinsically complex task that invalidates the naive application of many standard machine learning techniques. This chapter details the methodological pipeline that has been designed and implemented to overcome these challenges. Each step, from ground truth construction to the final model evaluation, was conceived to ensure scientific rigor, reproducibility of results, and the prevention of data leakage.

3.1 Overall Experimental Pipeline: An Integrated Approach

Our approach is a sequential and integrated process, where the output of each step conditions the next. To ensure a fair evaluation and avoid any form of data contamination, the entire pipeline is encapsulated within a single function, `_integrated_training_task`. This function ensures that the data split (Train / Validation / Test) is the **very first conceptual operation**, and that all subsequent steps (labeling, feature extraction, normalization) are performed in an isolated manner on each subset, as illustrated in Figure 3.1.

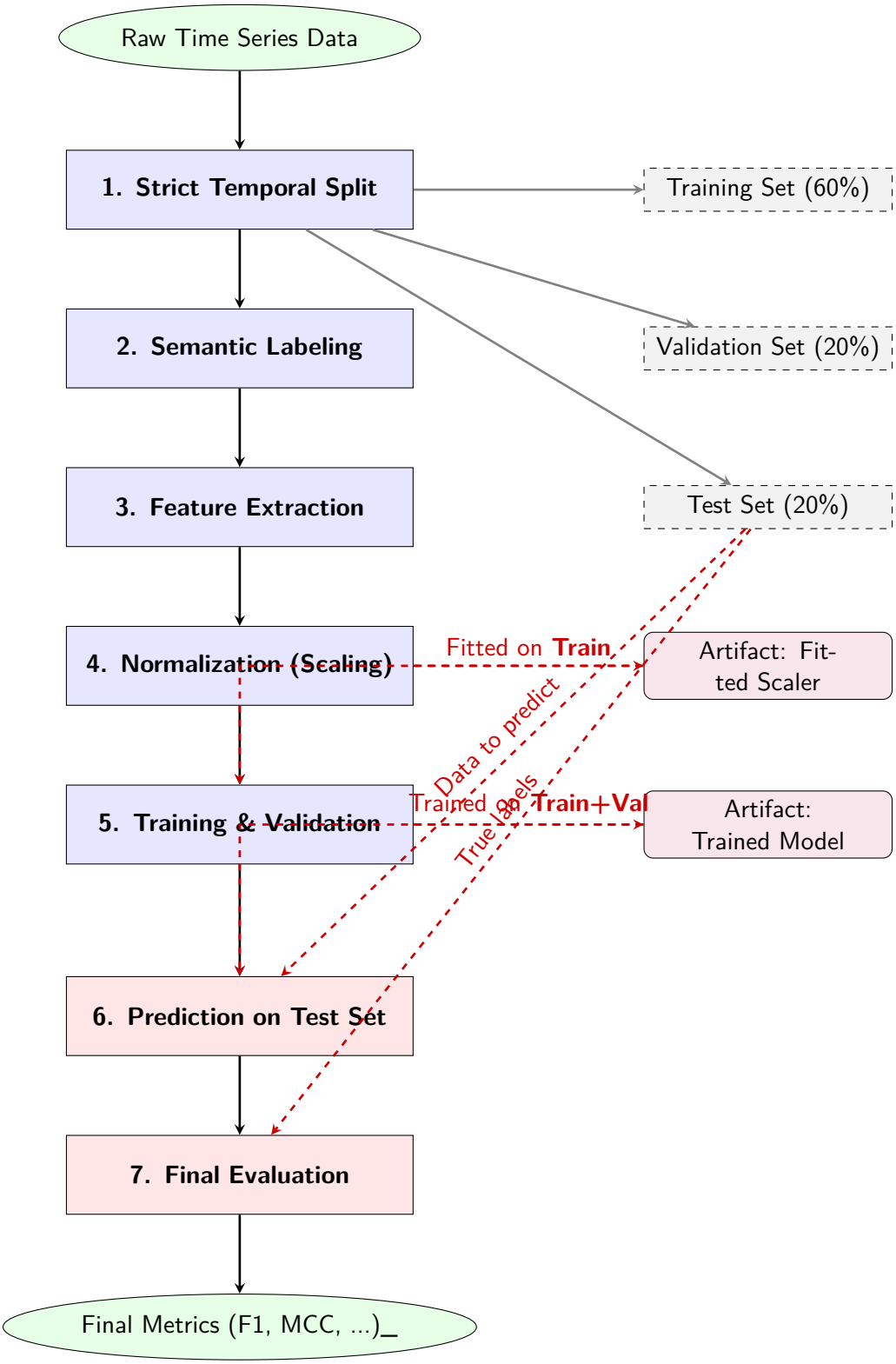

Figure 3.1: Overview of the integrated vertical-flow methodology pipeline. The temporal split is the first step, creating three isolated datasets. Processing steps are then applied sequentially. Critical artifacts (Scaler, Model) are generated exclusively from the training/validation data and are then used to transform or predict on the test data, thereby ensuring the absence of data leakage.

3.2 Step 1: Ground Truth Construction

The quality and definition of the target variable (the label) are the most critical factors conditioning the performance of a supervised model.

3.2.1 From Expert Label to Event Cycle Semantics

The raw data contains only discrete expert labels (e.g., "High" for paroxysmal activity). To enable early prediction, a heuristic was developed to semantically enrich this data by defining a complete event cycle. The `define_internal_event_cycle` function transforms a block of activity into three distinct phases:

- **Pre-Event:** The precursor phase, characterizing the energy build-up before the peak.
- **High-Paroxysm:** The peak activity phase, corresponding to the main event.
- **Post-Event:** The relaxation or return-to-calm phase after the event.

3.2.2 Justification and Limitations of the Labeling Heuristic

This labeling is parameterized by the `pre_event_ratio` hyperparameter, which defines the proportion of a "High" event's duration that should be considered as a "Pre-Event" phase.

Justification: This choice is motivated by the need to create a usable ground truth in the absence of fine-grained manual labeling of precursor phases. It relies on the domain assumption that a longer event is likely to be preceded by a correspondingly longer build-up phase.

Critical Analysis and Inductive Bias: This represents the **strongest inductive bias** in our methodology. The model is explicitly trained to recognize a temporal structure that has been imposed. The validity of this assumption is not guaranteed and is a simplification of the real physical phenomenon. Consequently, the reported model performance should be interpreted as its ability to learn this specific structure. The final validation of this structure's relevance relies on the post-hoc analysis of SHAP explanations, which must confirm that the model relies on physically meaningful indicators to identify these phases.

3.3 Step 2: Temporal Feature Engineering

3.3.1 Projection into a Multi-Scale Feature Space

To make temporal dynamics accessible to the model, the raw time series x_t is projected into a high-dimensional feature space $\Phi(\mathbf{h}_t)$. As detailed in Chapter 2.2, this transformation is performed via **multi-scale analysis**, where each time point is described by a vector of statistical and dynamic descriptors calculated over several past time windows ($W = \{w_1, w_2, \ldots, w_k\}$).

3.3.2 Methodological Safeguards Against Data Leakage

Preventing the leakage of future information is the cornerstone of our feature extraction methodology. Several levels of protection are implemented.

1. **Strict Causality of Sliding Windows:** All our feature calculation functions are based on operations that respect causality. For a calculation at time t over a window of size w, only observations from the interval $[t - w + 1, t]$ are used. This prevents look-ahead bias at the individual calculation level.

2. **Total Isolation of Datasets:** The most fundamental protection is architectural. Our integrated pipeline ensures that feature engineering and preprocessing operations are performed in a strictly independent manner on the training, validation, and test sets.

```python
# IN app_controller.py
def _integrated_training_task(self, params: dict, ...):

    # 1. The VERY FIRST step is the temporal split of the INDICES
    #    (or raw data if memory allows).
    n = len(self.original_df)
    train_end_idx = int(n * params['train_ratio'] / 100)
    val_end_idx = train_end_idx + int(n * params['val_ratio'] / 100)

    X_train_raw = self.original_df.iloc[:train_end_idx]
    # ... etc. for X_val_raw, X_test_raw

    # 2. Feature extraction is called on each ISOLATED raw dataset.
    X_train_features = extract_features(X_train_raw, feature_config)
    X_val_features = extract_features(X_val_raw, feature_config)
    # ...

    # 3. Any scaler or normalizer is fitted ONLY on the training data.
    scaler = StandardScaler().fit(X_train_features)

    # 4. The FITTED scaler is then APPLIED to transform the other sets.
    X_train_scaled = scaler.transform(X_train_features)
    X_val_scaled = scaler.transform(X_val_features)
    # ...

    # The model will never "see" the distribution of the validation or test data.
    # ... rest of the training ...
```

Listing 3.1: Illustration of the integrated pipeline ensuring dataset isolation.

This discipline ensures a fair and unbiased estimate of the model's generalization ability.

3.4 Step 3: Training and Validation Strategy

3.4.1 Time Series Cross-Validation for Optimization

Hyperparameter optimization is a critical step where the risk of overfitting is high. To mitigate this, we employ a cross-validation strategy specifically designed for sequential data: `TimeSeriesSplit`. Unlike standard K-Fold cross-validation, which assumes sample independence, `TimeSeriesSplit` creates folds where the training set always chronologically precedes the validation set. This "walk-forward" approach simulates the real-world conditions of a deployment where a model is trained on the past to predict the future.

3.4.2 Modeling Strategy for Class Imbalance

The extreme class imbalance (the rarity of *Pre-Event* phases) is the main modeling challenge. A naive approach optimizing for overall accuracy would consistently fail. Our strategy is twofold:

- **Loss Function Weighting:** We use the `class_weight` mechanism of the algorithms to assign a much higher penalty to misclassifications of minority classes. This forces the model to pay particular attention to these samples.
- **Optimization of a Targeted Metric:** The hyperparameter search is not guided by a global metric, but by a custom metric that reflects our scientific objective: the **F1-score of the *Pre-Event* class**. This strategic choice constrains the optimization algorithm (`RandomizedSearchCV`) to explore the hyperparameter space to find configurations that specifically excel at the early detection task, even if it comes at the expense of performance on the majority class.

3.5 Step 4: Evaluation and Post-Processing of Predictions

3.5.1 Justification of Evaluation Metrics

In the context of a warning system, there is an inherent trade-off between two types of errors: false alarms (Type I errors, impacting **Precision**) and missed events (Type II errors, impacting **Recall**).

- The **F1-Score**, as the harmonic mean of Precision and Recall, is used as our primary evaluation metric. It provides a balanced score that penalizes models that sacrifice one of these dimensions too much for the other.
- The **Matthews Correlation Coefficient (MCC)** is also reported. It is considered one of the most reliable single-class metrics for binary classification on imbalanced data, as it takes into account all four entries of the confusion matrix (true positives, true negatives, false positives, and false negatives). A score of +1 indicates a perfect prediction, 0 a random prediction, and -1 an inverse prediction.

3.5.2 Post-Processing: Improving Robustness via a Persistence Filter

A model's raw predictions can be volatile, generating sporadic single-timestep alerts that are often noise. To improve the system's robustness and operational reliability, a temporal persistence filter is applied in post-processing.

Principle: A positive prediction for a warning class (e.g., *Pre-Event*) is only confirmed if it is sustained for a minimum of N consecutive timesteps. Any isolated "alert" with a duration less than N is reclassified as belonging to the majority class (*Calm*).

Justification: This approach is inspired by denoising techniques in signal processing and is based on the domain assumption that significant precursory phenomena exhibit a degree of temporal continuity. The impact of this filter on the Precision/Recall trade-off is quantified and analyzed in the results section.

Chapter 4

Detailed Software Architecture

The design of robust scientific software, intended for research and exploration, suggests an architecture that promotes both experimental flexibility and methodological rigor. This chapter details the architectural choices and design patterns that structure the application, justifying their relevance to the system's objectives.

4.1 Model-View-Controller (MVC) Architectural Pattern

At the highest level, the system is organized according to the **Model-View-Controller (MVC)** architectural pattern. This canonical separation of concerns is fundamental to decoupling the business logic from its representation.

- **The Model (`Core/`)**: Represents the scientific core of the application. It encapsulates the data state, processing algorithms (`data_processor.py`), feature extraction (`feature_extractor.py`), training (`model_trainer.py`), and explanation (`explainer.py`). It is designed to be completely independent of the user interface.
- **The View (`main2.py`, `ui/`)**: Is responsible for all presentation to the user. Built with the PyQt6 framework, it displays data, plots, and interactive controls. Its role is to delegate user actions to the Controller and to update itself when instructed by the Controller.
- **The Controller (`app_controller.py`)**: Acts as the central orchestrator. It receives events from the View (e.g., a click on "Start Training"), invokes the appropriate services from the Model, and notifies the View of state changes or results to be displayed. It manages the application logic and the lifecycle of asynchronous operations.

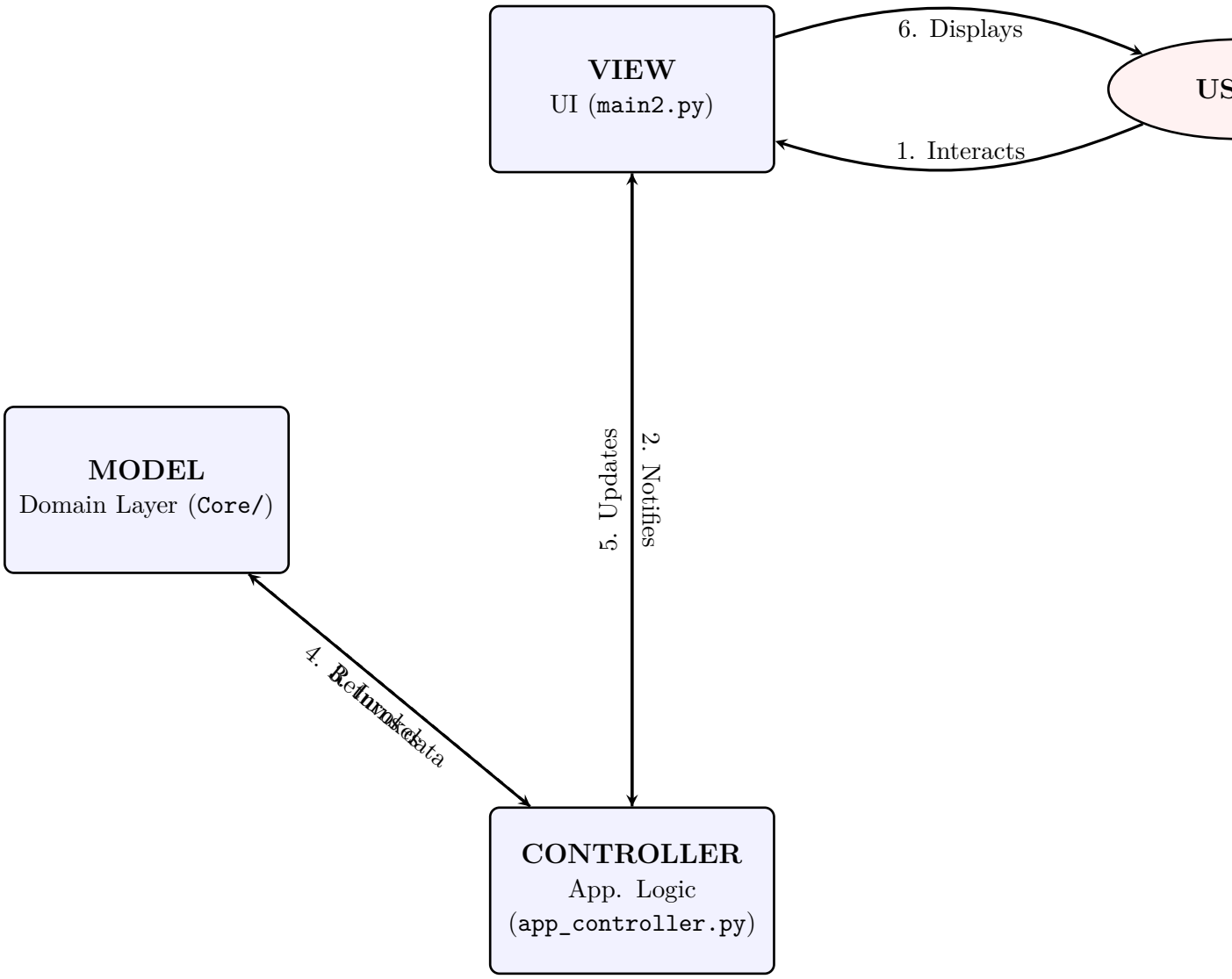

Figure 4.1: Interaction flow in the system's MVC architecture. The user interacts only with the View, and the Model is completely isolated from the View.

4.1.1 Dependency Inversion Principle (DIP)

A rigorous implementation of MVC naturally adheres to the Dependency Inversion Principle from SOLID. Communication between the Presentation layer (high level) and the Domain layer (low level) is mediated by the Controller. More importantly, communication from the Model to the View is achieved through a mechanism of abstractions: Qt's **signals and slots**.

The Model (`AppController` acting as a facade) emits signals (e.g., `training_finished`) without knowing which objects will listen to them. The View "connects" to these signals. Thus, the Model does not depend on the View; rather, the View depends on the Model's abstract signal interface. This inversion ensures that the scientific core could be reused with another interface (web, command line) without any modification.

4.2 Detailed Layered Architecture

The MVC separation is refined by a strict layered architecture, where dependencies flow in only one direction: from top (presentation) to bottom (infrastructure).

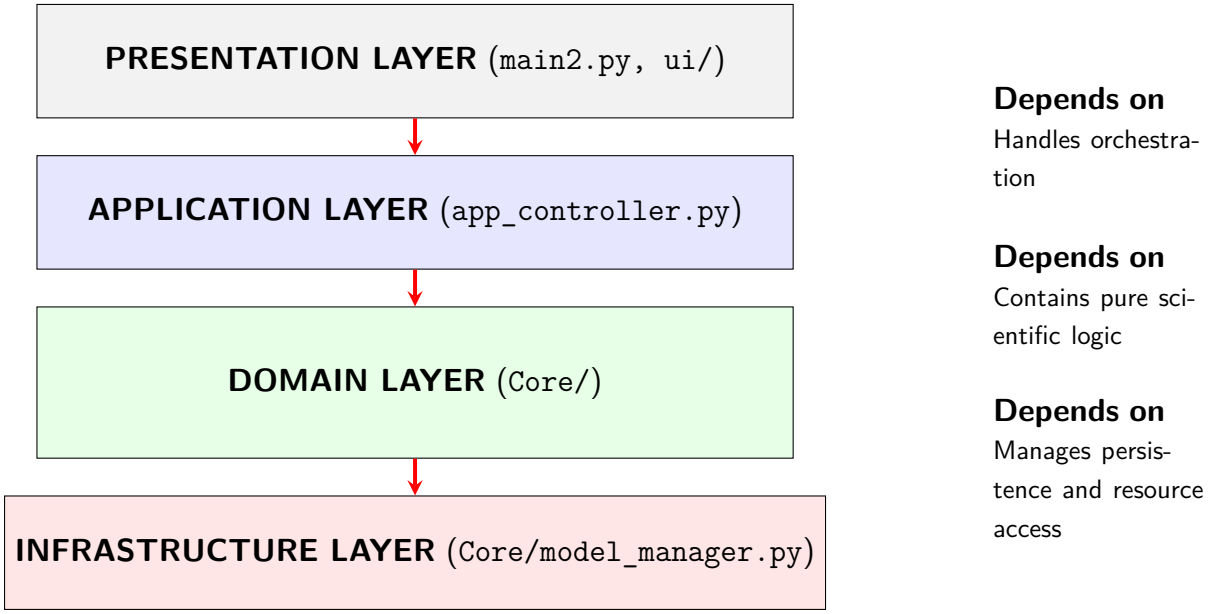

Figure 4.2: Layered architecture with a strict dependency direction, ensuring low coupling between components.

This structure ensures that the **Domain** layer, which contains the project's scientific intellectual property, is pure and has no knowledge of how it is used, whether by a graphical interface, a test script, or a web service.

4.3 Implemented Design Patterns

Beyond the overall architecture, specific design patterns are used to solve recurring problems in an elegant and robust manner.

4.3.1 Observer Pattern (Signals/Slots)

Communication between threads and layers is managed entirely by Qt's signals and slots mechanism, an elegant implementation of the Observer pattern. This pattern decouples the object that changes state (the Subject) from those that must react to this change (the Observers).

```
# IN app_controller.py (The Subject, via its facade)
class AppController(QObject):
    # The signal is a public interface, a promise of notification.
    feature_extraction_finished = pyqtSignal(pd.DataFrame)

    def run_feature_extraction(self, config: dict):
        # ... logic to start the asynchronous task ...
```

```
        worker = self._run_in_thread(...)
        # The controller connects the task's result to its own slot.
        worker.result_ready.connect(self._on_features_extracted)

    def _on_features_extracted(self, features: pd.DataFrame):
        # When the task is finished, the controller emits its own signal
        # to notify the rest of the application (the View).
        self.feature_extraction_finished.emit(features)

# IN main2.py (The Observer)
class MainWindow(QMainWindow):
    def _connect_signals(self):
        # The View subscribes ("connects") to the controller's signal.
        self.controller.feature_extraction_finished.connect(
            self._handle_feature_extraction_results
        )

    @pyqtSlot(pd.DataFrame)
    def _handle_feature_extraction_results(self, feature_matrix: pd.DataFrame):
        # The slot is the callback method that is executed in response to the signal.
        # It updates the View's plots.
        self.feature_matrix_for_analysis = feature_matrix
        self.update_correlation_plot()
```

Listing 4.1: Concrete example of the Observer pattern in the application

4.3.2 Strategy Pattern (implicit)

The application allows for the dynamic selection of the Machine Learning algorithm to be used (Random Forest, LightGBM, KNN). This behavior is a conceptual implementation of the Strategy pattern. Each algorithm is encapsulated in its own training function (`train_and_evaluate_rf`, `train_and_evaluate_lgbm`, etc.), which adheres to a common interface (it accepts the same datasets and returns a identically structured results dictionary). The controller selects and executes the appropriate strategy based on the user's selection, without the calling code being coupled to a specific implementation.

```
# IN app_controller.py (_execute_training)
def _execute_training(self, X_train, y_train, ..., params, ...):

    # A dictionary acts as a strategy registry.
    trainer_map = {
        "K-Nearest Neighbors (KNN)": train_and_evaluate_knn,
        "Random Forest": train_and_evaluate_rf,
        "LightGBM": train_and_evaluate_lgbm
    }

    # Selection of the strategy based on the configuration received from the View.
    trainer_func = trainer_map[params['model_type']]
```

```
13
14      # Execution of the strategy. The controller does not need to
15      # know the details of the algorithm.
16      results = trainer_func(
17          X_train, y_train, X_val, y_val, X_test, y_test,
18          model_config, sample_weight=sample_weights
19      )
20      return results
```

Listing 4.2: Selection of the training strategy in the controller

This approach makes adding a new algorithm straightforward: one only needs to create a new training function that respects the interface and add it to the `trainer_map`.

4.3.3 Command & Worker Thread Pattern

To handle long-running operations without freezing the user interface, we encapsulate each task in a `WorkerThread` object. This is inspired by the Command pattern:

- **Command:** The function (`func`) and its arguments (`*args`, `kwargs`) passed to the `WorkerThread`'s constructor represent a command that encapsulates a request.
- **Invoker:** The `WorkerThread` instance itself is the invoker. Its `start()` method triggers the execution of the command.
- **Receiver:** The business logic contained in the `func` function is the receiver that knows how to perform the task.

This encapsulation allows the controller to manage tasks uniformly, queue them if necessary, and handle their lifecycle (start, cancel).

```
1  # IN app_controller.py
2  def run_full_training(self, params: dict):
3      # 1. The command is defined by the `_integrated_training_task`
       method
4      #    and its `params` argument.
5      worker = self._run_in_thread(
6          self._integrated_training_task,
7          self._on_training_complete,
8          params
9      )
10     # The `worker` object is the Invoker. It is now ready to execute
       the command.
11
12     # 2. The View interacts with the Invoker to display progress
13     #    and allow for cancellation.
14     if worker:
15         dialog = ProcessingDialog("Training Model", self)
16         worker.progress_updated.connect(dialog.update_progress)
17         dialog.cancelled.connect(worker.cancel)
18         dialog.exec()
```

Listing 4.3: Encapsulating a long-running task with the Worker/Command pattern

4.4 Module Structure of the Domain Layer (`Core/`)

The Domain layer is the scientific heart of the project and is organized according to the principles of **Domain-Driven Design (DDD)**, where each module has a single, well-defined responsibility.

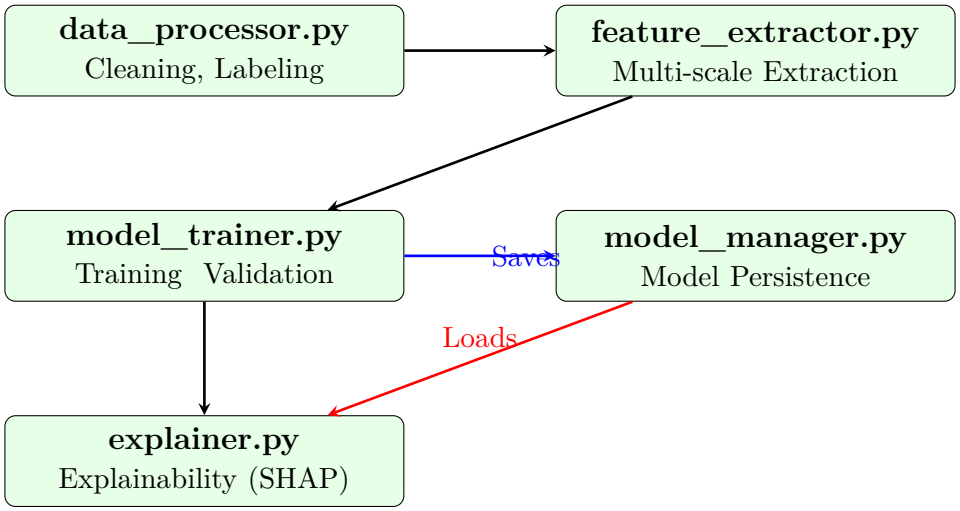

Figure 4.3: Modular architecture of the Domain layer. The black arrows represent the main flow of the processing pipeline, while the colored arrows indicate interactions with the persistence module.

This organization ensures that the scientific components are unit-testable and can be easily replaced or improved without impacting the rest of the system.

Chapter 5

Advanced Management of Asynchronous Tasks

A modern interactive scientific application should be responsive, even when performing long and complex calculations. Freezing the user interface during model training or feature extraction is unacceptable. This chapter details the concurrency architecture implemented to ensure a smooth user experience and full control over background tasks.

5.1 The Challenge of Concurrency in Scientific Computing

Running Machine Learning algorithms in a Graphical User Interface (GUI) application introduces a fundamental tension between two concurrent threads of execution:

- **The Main Thread (GUI Thread)**: This single thread is responsible for all interface rendering and processing user events (clicks, mouse movements, etc.). Any blocking operation on this thread immediately results in a "frozen" and unresponsive interface.
- **The Worker Threads**: Secondary threads are necessary to execute computationally intensive tasks, thereby freeing up the main thread.

This separation creates specific challenges that our architecture must address:

1. **Thread-Safe Communication:** Threads do not trivially share memory. Communication between a worker and the main thread (to report progress, results, or errors) must be done via mechanisms that guarantee data integrity and avoid race conditions.
2. **Graceful Cancellation:** Interrupting a thread is a dangerous operation. It is necessary to implement a **cooperative** cancellation mechanism, where the worker can periodically check if it should stop and clean up its resources before terminating.
3. **Progress Reporting and State Management:** The main thread must be able to track the state of background tasks to inform the user and adjust the interface state (e.g., disabling a button during a calculation).
4. **Error Propagation:** An exception occurring in a worker thread must be caught and safely propagated to the main thread to be handled (e.g., by displaying an error dialog).

5.2 Concurrency Architecture

To meet these challenges, the application implements a robust threading model based on the primitives provided by the Qt framework.

5.2.1 Threading Model: The Central Role of Signals and Slots

The architectural model is based on the separation of responsibilities between the main thread and a pool of workers. Communication between these entities is handled exclusively by Qt's **signals and slots** mechanism, which is inherently thread-safe.

When a signal is emitted from a worker thread to a slot of an object living in the main thread, Qt queues the slot invocation in the main thread's event loop. The slot will be executed safely when the main thread becomes available. This mechanism eliminates the need for complex manual locks (mutexes) in the application code for inter-thread communication.

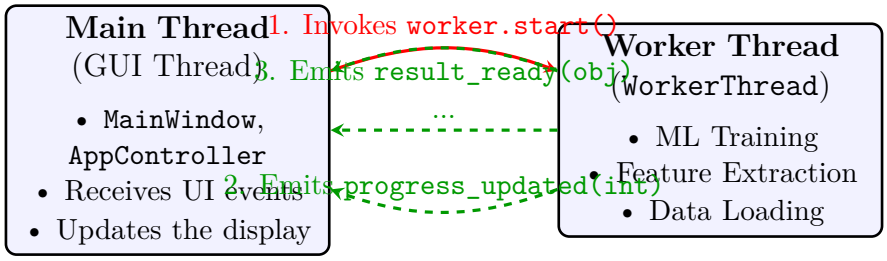

Figure 5.1: Threading architecture. The main thread starts the worker and subscribes to its signals. Communication from the worker to the main thread is asynchronous and managed by Qt's event loop.

5.2.2 Generic Worker Implementation: The `WorkerThread` Class

To avoid re-implementing concurrency logic for each task, a generic class, `WorkerThread`, was created to encapsulate all the necessary behavior.

```python
# IN app_controller.py
class WorkerThread(QThread):
    """
    Generic worker thread with full support for asynchronous
    communication, cancellation, and error handling.
    """

    # Public signals forming the worker's communication interface
    result_ready = pyqtSignal(object)
    error_occurred = pyqtSignal(str, str)
    progress_updated = pyqtSignal(int, str)

    def __init__(self, func: Callable, *args, kwargs):
        super().__init__()
        self.func = func
        self.args = args
        self.kwargs = kwargs
        self._is_cancelled = False # Flag for cooperative
    cancellation

    def run(self):
        """
        Thread's entry point. This is where the communication
        logic is injected into the task.
```

```
        """
        try:
            # DEPENDENCY INJECTION: The business task does not need to
            # know about Qt. The worker provides it with the necessary callbacks.
            self.kwargs['update_progress'] = self.progress_updated.emit
            self.kwargs['is_cancelled'] = lambda: self._is_cancelled

            # Execution of the business logic (e.g., extract_features)
            result = self.func(*self.args, self.kwargs)

            # The result is only emitted if the operation was not cancelled.
            if not self._is_cancelled:
                self.result_ready.emit(result)

        except Exception as e:
            # Any exception is caught and propagated via a signal.
            if not self._is_cancelled:
                import traceback
                error_msg = f"{str(e)}\n\nTraceback:\n{traceback.format_exc()}"
                self.error_occurred.emit(f"Error in Worker", error_msg)

    def cancel(self):
        """Public slot to request the task to stop."""
        self._is_cancelled = True
```

Listing 5.1: Key excerpt from the `WorkerThread` class with dependency injection

This design allows any Python function to be executed asynchronously with full UI support, simply by wrapping it in an instance of `WorkerThread`.

5.3 Advanced Progress Dialog: `ProcessingDialog`

User feedback during a long-running task is managed by the `ProcessingDialog` class. This class is more than just a progress bar; it provides a rich and informative interface.

5.3.1 Rich and Responsive User Interface

The `ProcessingDialog` is an observer that subscribes to the signals emitted by a `WorkerThread`.

- It connects to the `progress_updated(int, str)` signal to update its progress bar and a status label.
- It connects to the worker's `finished()` signal to close itself automatically when the task is complete.

- Its "Cancel" button is connected to a `cancelled` signal, which the controller connects to the `worker.cancel()` method.

```
# IN app_controller.py, when launching a task
def _trigger_run_external_prediction(self):
    # ...
    worker = self.controller.run_external_prediction(model_name)

    if worker:
        dialog = ProcessingDialog("Predicting on External Data", self
    )

        # Connect the worker's signals to the dialog's slots
        worker.progress_updated.connect(dialog.update_progress)
        worker.finished.connect(dialog.task_finished)

        # Connect the dialog's cancellation signal to the worker's
    slot
        dialog.cancelled.connect(worker.cancel)

        dialog.exec() # Displays the dialog modally
```

Listing 5.2: Connecting the progress dialog to the worker

This architecture ensures that the dialog and the worker are fully decoupled: the worker does not know that a dialog is observing it, and the dialog can observe any object that emits the appropriate signals.

Chapter 6

Detailed File Structure

A robust software architecture is materialized through a clear and logical organization of its files and modules. This chapter presents the complete project directory structure and delves into the responsibilities of the critical modules of the Domain layer, which constitute the scientific core of the application.

6.1 Project Directory Structure

The directory structure was designed following the principle of Separation of Concerns, isolating the application code, tests, documentation, data, and resources.

```
MLAnalyticsApp/
|-- README.md                          # Welcome documentation and project summary
|-- requirements.txt                   # Python dependencies for reproducibility
|-- config.yaml                        # Centralized configuration (parameters, thresholds)
|-- main2.py                           # Application entry point (MVC View)
|-- app_controller.py                  # Main controller (application logic)
|
|-- Core/                              # DOMAIN LAYER (pure business logic)
|   |-- __init__.py
|   |-- data_processor.py              # Data loading and preparation pipeline
|   |-- feature_extractor.py           # Multi-scale feature extraction
|   |-- model_trainer.py               # Model training, validation, and evaluation
|   |-- model_manager.py               # Model persistence (saving/loading)
|   `-- explainer.py                   # Encapsulation of explainability logic (SHAP)
|
|-- ui/                                # Modules specific to the user interface
|   |-- __init__.py
|   `-- ui_dialogs.py                  # Dialog boxes (progress, messages)
```

```
|
|-- tests/                              # Automated test suite (pytest)
|   |-- __init__.py
|   |-- test_data_processor.py          # Unit tests for data processing
|   `-- test_feature_extractor.py       # Unit tests for feature extraction
|
|-- data/                               # Raw and sample data
|   `-- samples/
|       `-- example_dataset.xlsx        # Example dataset
|
|-- models/                             # Trained and saved models
|   `-- MyModel_RF_20250815/
|       |-- model.joblib                # Serialized model object
|       `-- report.json                 # Metrics, hyperparameters, and configuration
|
|-- docs/                               # Detailed project documentation
|   `-- developer_guide/                # Developer guide (this document)
|
`-- resources/                          # Non-code files (icons, styles)
    `-- styles/
        `-- dark_stylesheet.qss         # Application stylesheet
```

Listing 6.1: Complete and commented project directory structure

6.2 Detailed Analysis of Domain Layer Modules

The `Core/` layer encapsulates all scientific logic and is designed to be a standalone, testable, and reusable framework.

6.2.1 `data_processor.py` – Data Preparation Pipeline

This module is the entry point for raw data into the system. Its mission is to transform potentially imperfect files into a structured, clean, and semantically enriched dataset, ready for analysis.

Responsibilities

- **Robust Loading:** Read source data (e.g., `.xlsx`) while handling common anomalies such as duplicate timestamps (via aggregation) and critical missing values (via targeted removal).

- **Semantic Labeling:** Implement the business logic to transform raw labels (e.g., 'High', 'Low') into a complete event cycle ('Pre-Event', 'High-Paroxysm', 'Post-Event'). This is where domain knowledge-based heuristics, such as the `pre_event_ratio`, are applied.
- **Temporal Segmentation:** Detect major discontinuities in the time series to split it into continuous segments. This prevents sliding window calculations from being polluted by non-significant interruptions (e.g., sensor shutdown).

```python
# IN Core/data_processor.py

def define_internal_event_cycle(df_block: pd.DataFrame,
                                pre_event_ratio: float,
                                ...) -> pd.DataFrame:
    """
    Labels the phases of an activity block based on the
    relative position to human 'High' labels.
    """
    df = df_block.copy()

    # Find the indices of the first and last 'High' tags
    high_indices = df.index[df[target_col] == 'High']

    if high_indices.empty:
        # Case where the Detector found a block but without a 'High' label
        df[target_col] = 'Pre-Event' # Default assumption
        return df

    first_high_idx = high_indices.min()
    last_high_idx = high_indices.max()

    # Step 1: Label everything preceding as 'Pre-Event'
    df.loc[:first_high_idx - 1, target_col] = 'Pre-Event'

    # Step 2: Label everything following as 'Post-Event'
    df.loc[last_high_idx + 1:, target_col] = 'Post-Event'

    # Step 3: Split the 'High' block itself
    event_duration = last_high_idx - first_high_idx + 1
    pre_duration_in_high = int(event_duration * pre_event_ratio)

    # The entire block is first considered the peak
    df.loc[first_high_idx:last_high_idx, target_col] = 'High-Paroxysm'

    # The first part of the block is redefined as a precursor
    if pre_duration_in_high > 0:
        pre_event_end_idx = first_high_idx + pre_duration_in_high - 1
        df.loc[first_high_idx:pre_event_end_idx, target_col] = 'Pre-Event'

    return df
```

Listing 6.2: Excerpt illustrating the labeling logic in `data_processor.py`

6.2.2 `feature_extractor.py` – Multi-scale Feature Extraction

This module is at the heart of the knowledge transformation process. It converts a time sequence into a rich vector of descriptors, allowing the Machine Learning model to "see" the signal's dynamics.

Responsibilities

- **Multi-scale Analysis:** Implement the logic for calculating dozens of features over multiple time windows (e.g., 10, 30, 50 points), thus capturing short-, medium-, and long-term phenomena.
- **Optimized Calculations:** Provide efficient and vectorized implementations (based on NumPy and Pandas) for complex features like regression slope, rolling Z-score, or volatility ratios.
- **Data Leakage Prevention:** Ensure that all sliding window calculations are **causally correct**, meaning that a calculation at time t only uses information available at or before t.

```python
# IN Core/feature_extractor.py (within ExpertFeaturesCalculator)

@staticmethod
def calculate_zscore_vectorized(series: pd.Series, window: int) -> pd.Series:
    """
    Calculates the rolling Z-Score to normalize the signal relative
    to its local history. Makes the feature scale-independent.
    """
    # min_periods is crucial to avoid zeros at the beginning
    min_periods = window // 4

    rolling_stats = series.rolling(window, min_periods=min_periods)
    mean = rolling_stats.mean()
    std = rolling_stats.std()

    # Replace null std deviations to avoid division by zero
    safe_std = std.replace(0, 1e-6).fillna(method='ffill')

    z_score = (series - mean) / safe_std
    return z_score.fillna(0.0)
```

Listing 6.3: Example of an expert feature calculation in `feature_extractor.py`

6.2.3 `model_trainer.py` – Model Training and Evaluation

This module orchestrates the Machine Learning pipeline, from data preparation to the rigorous evaluation of the final model. It constitutes the methodological guarantee of the project.

Responsibilities

- **Time Series Cross-Validation:** Implement the logic for partitioning data into training, validation, and test sets while respecting chronological order (`TimeSeriesSplit`), which is the only valid approach to avoid temporal data leakage.
- **Hyperparameter Search:** Encapsulate the logic for automated search of the best hyperparameters for each model type using `RandomizedSearchCV`.
- **Comprehensive Evaluation:** Calculate a complete set of performance metrics (classification report, confusion matrix, MCC score) and provide detailed logs on the model's performance.
- **Post-processing:** Include post-processing steps for predictions, such as the persistence filter, to improve the robustness of alerts.

```python
# IN Core/model_trainer.py (_find_best_rf_model)

def _find_best_rf_model(X_train_val, y_train_val, config, sample_weight):
    # ... (definition of param_distributions) ...

    # Instantiation of a time series cross-validator.
    # It ensures that training folds always precede test folds.
    time_series_cv = TimeSeriesSplit(n_splits=config.cv_folds)

    # Using a custom scorer to focus on 'Pre-Event'
    custom_scorer = _get_pre_event_f1_scorer(y_train_val)

    random_search = RandomizedSearchCV(
        estimator=RandomForestClassifier(...),
        param_distributions=param_distributions,
        n_iter=config.n_iter,
        cv=time_series_cv, # Using the time series validator
        scoring=custom_scorer,
        # ...
    )

    # Launching the search, passing temporal weights if available
    random_search.fit(X_train_val, y_train_val, sample_weight=sample_weight)

    return random_search.best_estimator_, random_search.best_score_
```

Listing 6.4: Hyperparameter search logic with time series validation

6.2.4 `model_manager.py` – Model Management and Persistence

This module is responsible for serializing training artifacts (models, configurations, results) to ensure the **reproducibility** of experiments and to enable deployment.

Responsibilities

- **Complete Saving:** Save not only the model object (‘.joblib‘), but also a comprehensive JSON report containing performance metrics, hyperparameters, and the exact feature configuration used.
- **Reliable Loading:** Reload a model and its associated configuration to make predictions on new data.
- **Inventory Management:** Provide functions to list, compare, and manage the different saved models.

```python
# IN Core/model_manager.py (conceptual logic of save_model)

def save_model(model_name: str, results: dict, feature_config: dict):
    model_path = Path("models") / model_name
    model_path.mkdir(exist_ok=True)

    # 1. Save the model object
    joblib.dump(results['model'], model_path / "model.joblib")

    # 2. Create a complete report for reproducibility
    report_data = {
        'model_name': model_name,
        'model_type': type(results['model']).__name__,
        'creation_date': datetime.now().isoformat(),

        # Evaluation results
        'performance_report': results.get('report', {}),
        'best_hyperparameters': results.get('best_params', {}),

        # Experiment configuration
        'feature_extraction_config': feature_config
    }

    # 3. Save the report as JSON
    with open(model_path / "report.json", 'w') as f:
        json.dump(report_data, f, indent=4)
```

Listing 6.5: Structure of the model report for reproducibility

6.3 Results

The methodology was evaluated by training a Random Forest model on the full set of 34 engineered features, using a 70/15/15 temporal split for the training, validation, and test sets respectively. The model's performance is analyzed first with a default decision threshold, then with an optimized threshold derived from the probabilistic output.

6.3.1 Performance with a Default Decision Threshold (p=0.5)

By default, a classification is made for the 'Actif' class if its predicted probability exceeds 0.5. The performance under this standard condition is presented in Figure 6.1. The model achieves a global Macro F1-Score of 0.688. While the Precision for the 'Calm' class is perfect (1.00), its Recall is extremely low (0.41), indicating that a vast number of calm periods are incorrectly flagged as active. Conversely, the 'Actif' class achieves a perfect Recall (1.00) at the cost of a mediocre Precision (0.65).

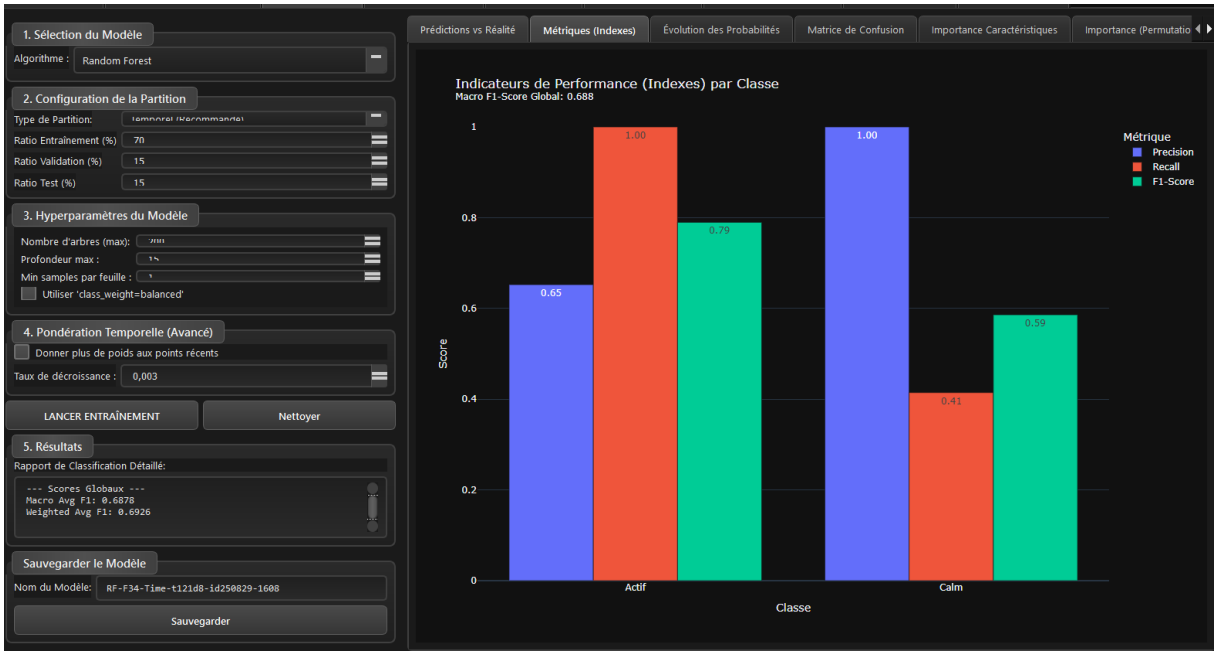

Figure 6.1: Performance metrics with a default decision threshold of p=0.5. The perfect Recall for the 'Actif' class is achieved at the expense of a very high false alarm rate (low Recall for 'Calm').

The temporal visualization in Figure 6.2 confirms this analysis. The model correctly identifies the entire active block but generates a significant number of false positives by classifying calm periods as active. This behavior, while ensuring no events are missed, renders the system impractical for operational use due to alert fatigue.

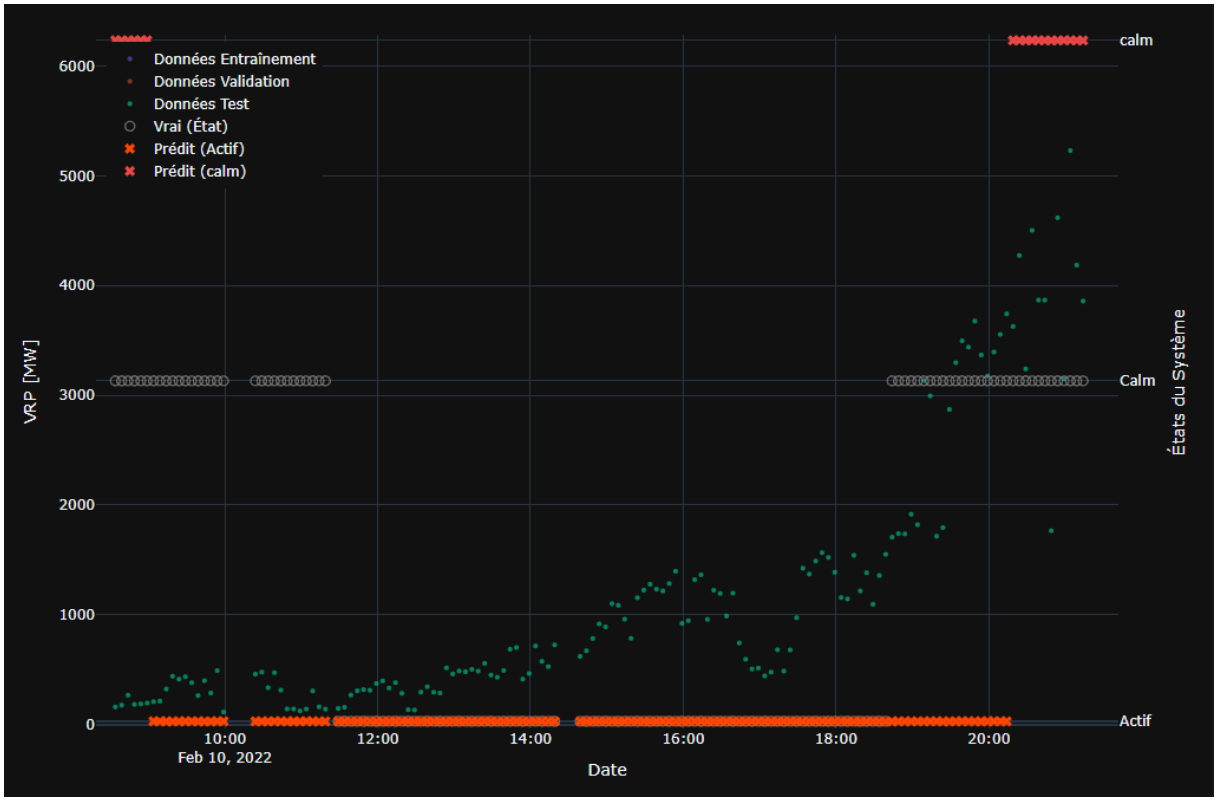

Figure 6.2: Predictions on the test set with a default threshold of p=0.5. All true 'Actif' states (bottom) are correctly predicted (orange crosses), but many 'Calm' states (top) are also incorrectly flagged as 'Actif'.

6.3.2 Optimizing the Decision Threshold via Probabilistic Analysis

The binary prediction is a simplistic view of the model's output. A deeper analysis lies in the temporal evolution of the predicted probabilities for each class. Figure 6.3 shows the probability curves for a representative event from the test set.

This analysis reveals a distinct and physically coherent dynamic:

1. **Precursor Phase:** At the onset of the event (around 08:00), the probability P(Actif) begins a steady, progressive climb from a low baseline, crossing the 0.5 threshold early on.

2. **Paroxysmal Phase:** During the peak of the event (approx. 10:00 to 18:00), the probability P(Actif) saturates at a high level of confidence (>0.8).

3. **Post-Event Phase:** As the event subsides (after 19:00), the probability P(Actif) exhibits a gradual, near-linear decrease.

This observation suggests that the default 0.5 threshold is too low and sensitive to the initial precursor signals. A higher threshold should allow the model to distinguish more effectively between low-energy precursors and high-energy paroxysmal phases, thus reducing false alarms on calm data while maintaining high sensitivity to true events. Based on this analysis, an optimized threshold of **p=0.8** was selected.

Figure 6.3: Temporal evolution of predicted probabilities for the 'Actif' (orange) and 'Calm' (blue) classes during a volcanic event on Feb 10, 2022. A clear dynamic is visible.

6.3.3 Performance with an Optimized Decision Threshold (p=0.8)

Re-evaluating the model with a decision threshold of p=0.8 yields a significantly more balanced and useful performance, as shown in Figure 6.4. The global Macro F1-Score improves to **0.808**.

Most notably, the Recall for the 'Calm' class dramatically increases to 0.72, indicating a substantial reduction in false alarms. The Recall for the 'Actif' class remains high at 0.89, meaning the model still detects the vast majority of true events. The temporal plot in Figure 6.5 visually confirms this improvement.

This result demonstrates the critical importance of post-processing the model's probabilistic output. By moving from a simplistic binary classification to an interactive, threshold-based decision framework, we transform the model from a noisy detector into a robust and scientifically interpretable tool for decision support.

6.4 Results with an Optimized Feature Set (Parsimonious Model)

Based on the Permutation Feature Importance analysis, which revealed significant feature redundancy and the detrimental effect of noisy predictors, we conducted a new experiment. A Random Forest model was trained using only the top five most impactful and non-

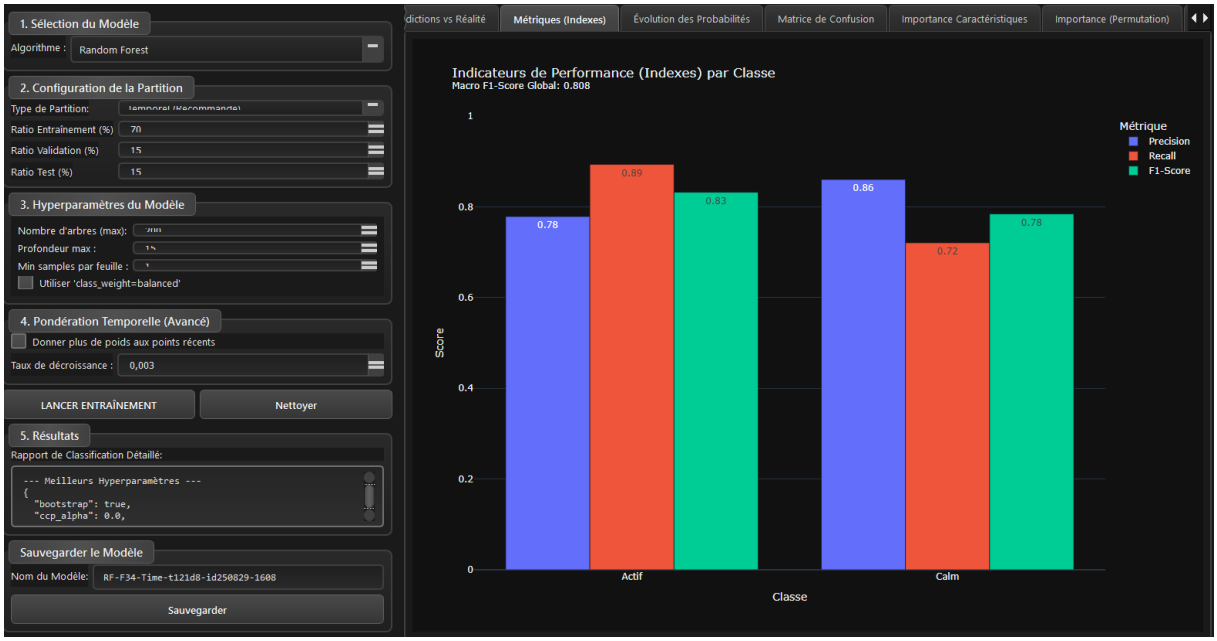

Figure 6.4: Performance metrics with an optimized decision threshold of p=0.8. A much better balance between Precision and Recall is achieved for both classes.

Figure 6.5: Predictions on the test set with an optimized threshold of p=0.8. The number of false positives is drastically reduced, while the core of the active event is still correctly identified.

redundant features identified: `median10`, `median30`, `median50`, `volatility_ratio_10_30`, and `iqr50`. The same training parameters and temporal split were used to ensure a fair and direct comparison with the full 34-feature model.

6.4.1 Model Performance Evaluation

The performance of the parsimonious model, evaluated on the test set with a decision threshold of 0.5, is presented in Figure 6.6. The model achieves a robust global Macro F1-Score of 0.774.

This result is highly significant. Despite a reduction of over 85% in the number of features, the overall performance is only marginally lower than the best-case scenario of the complex model (0.808), and significantly better than its baseline performance (0.688).

Notably, the model exhibits an extremely high Recall of 0.92 for the 'Actif' class, indicating it successfully identifies 92% of all true active phases. This high sensitivity is crucial for an early-warning system. This comes at the cost of a lower Recall for the 'Calm' class (0.63), demonstrating that the model maintains a desirable bias towards raising alerts, a trade-off that can be managed via the interactive decision threshold.

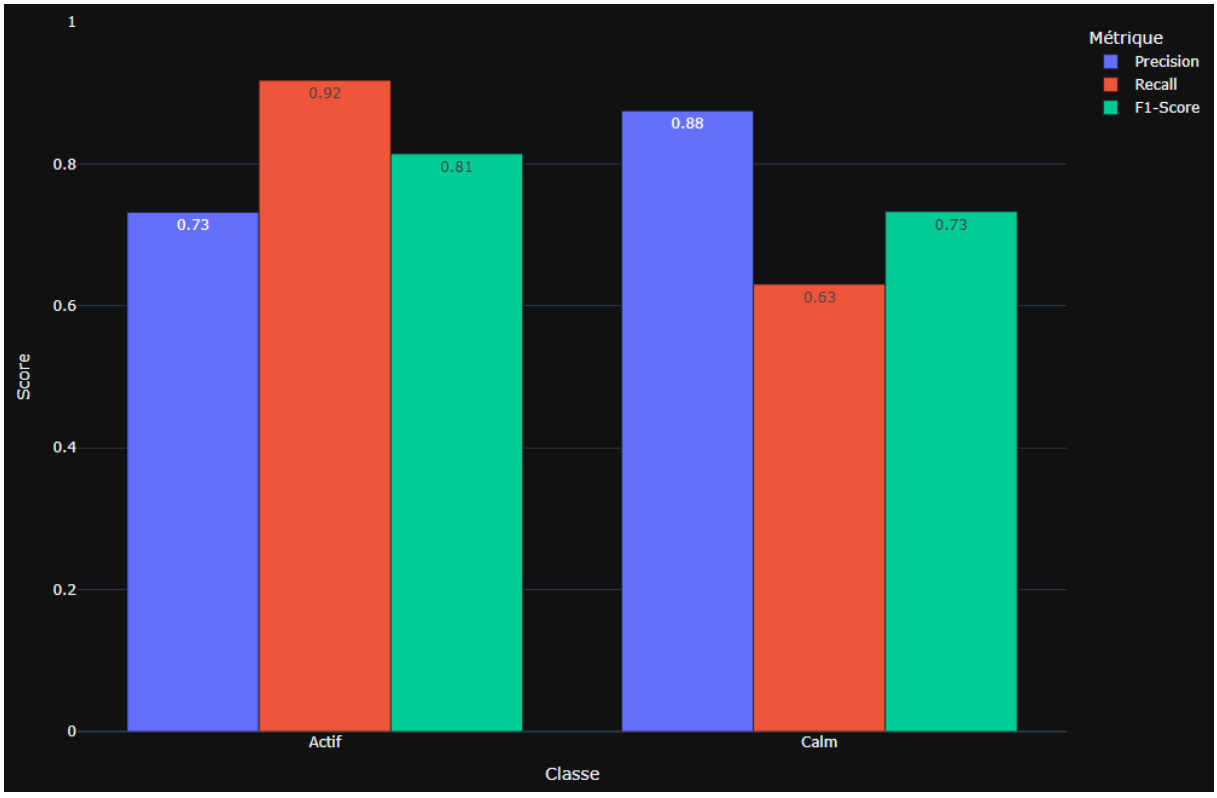

Figure 6.6: Performance metrics of the parsimonious model (5 features). The high Recall for the 'Actif' class is maintained, and the overall Macro F1-Score remains strong.

6.4.2 Feature Importance Hierarchy

An analysis of the feature importance for this simplified model (Figure 6.7) reinforces the conclusions from the PFI analysis. The `median10` feature, representing the short-term

signal amplitude, contributes the vast majority of the predictive power. The remaining features—long-term amplitude (`median30`, `median50`), volatility ratio, and interquartile range—serve to contextualize this primary indicator. This confirms that the model's decision-making process is dominated by a small set of physically interpretable predictors.

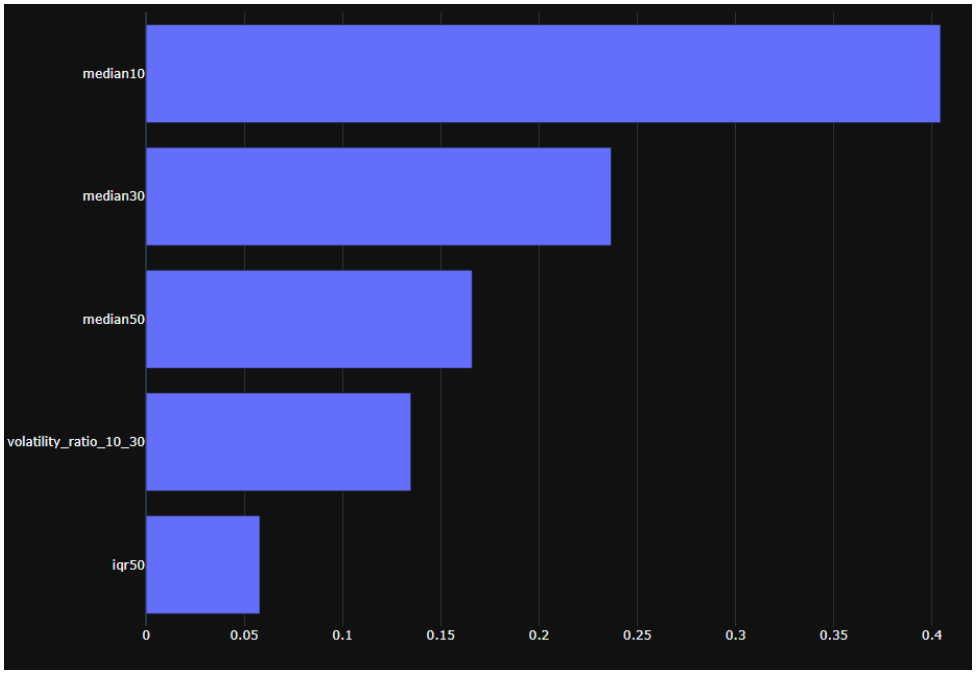

Figure 6.7: Gini importance for the 5-feature model. The hierarchy clearly shows the dominance of the short-term signal amplitude ('median10').

6.4.3 Temporal Dynamics and Probabilistic Output

The temporal analysis confirms the parsimonious model's effectiveness. Figure 6.8 shows that the model correctly identifies the entire active phase within the test set, with few false negatives.

Crucially, the evolution of the predicted probabilities (Figure 6.9) retains the physically coherent dynamics observed with the full model. The probability P(Actif) shows a clear progressive climb at the onset of the event, saturation during the peak, and a gradual decay in the post-event phase. This demonstrates that the simplified model has not lost its ability to capture the underlying temporal structure of the volcanic process. This finding is critical, as it validates that the heuristic of using probability thresholds to identify precursor and post-event phases remains viable.

These results collectively argue that a simpler, more interpretable model based on a rigorously selected feature set is not only faster and more efficient but also maintains a high level of predictive performance, making it a superior choice for a real-world decision-support tool.

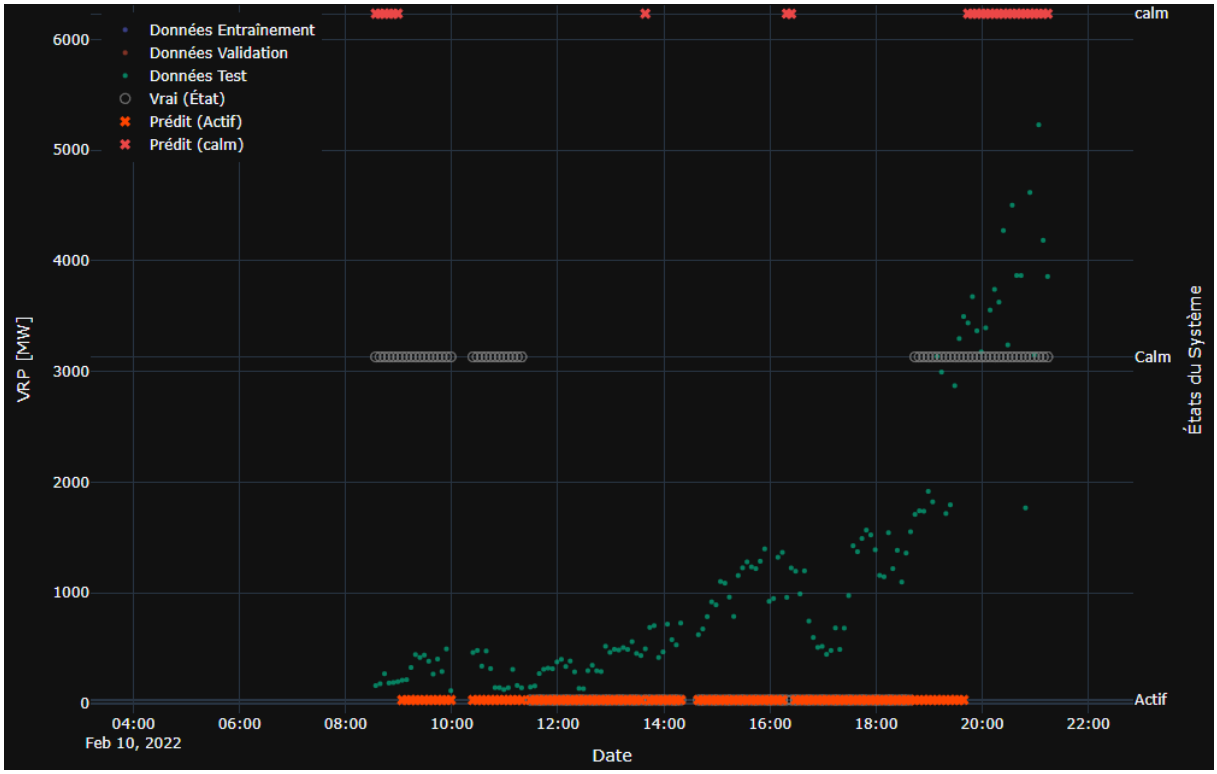

Figure 6.8: Predictions of the 5-feature model on the test set. The model successfully flags the entire active period.

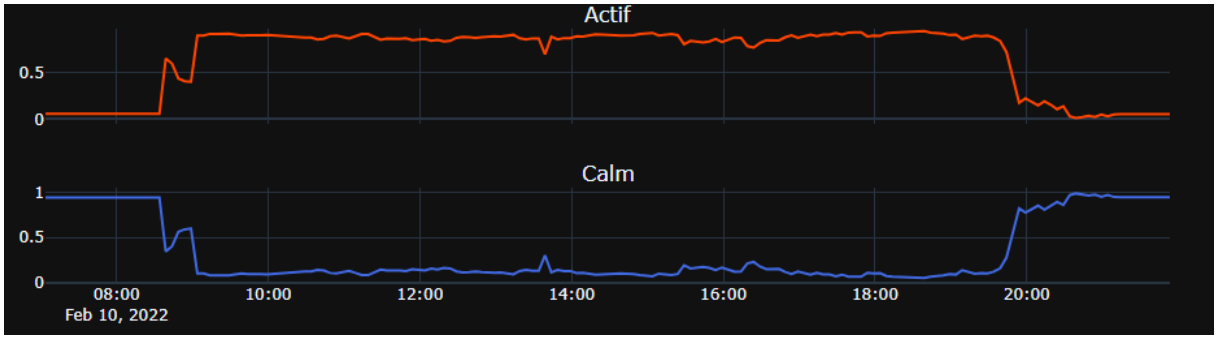

Figure 6.9: Temporal evolution of predicted probabilities for the parsimonious model during an event. The physically coherent precursor rise, saturation, and post-event decay are preserved.

6.5 Critical Analysis and Fundamental Limitations

While the presented metrics on the test set may seem encouraging, a rigorous scientific critique compels us to question the very foundations of this work's predictive claims. The results should not be interpreted as proof of a reliable forecasting model, but rather as the performance benchmark of a pattern-recognition algorithm on a severely constrained, single-sample experiment. The limitations are not minor caveats; they are profound and define the scope of our conclusions.

6.5.1 The Illusion of Performance in a Data-Scarce Environment

The core issue is the epistemological limit imposed by the dataset itself. With such a limited sample of eruptive events, the fundamental question arises: is this problem even reliably solvable? Volcanoes are chaotic systems, and it is plausible that their predictability is intrinsically limited, regardless of the algorithm.

A Statistically Insignificant Test Set: The test set, representing only 15-25% of an already small dataset, contains a statistically insignificant number of "Actif" events. A high F1-score on such a sample is not robust evidence of a model's competence. It could be the result of the model correctly identifying a few events whose patterns were very similar to those in the training set. There is no statistical basis to claim that this performance would hold on a different set of eruptive episodes from the same volcano. The work lacks extensive out-of-sample validation, which is the only true measure of generalization.

The Risk of Sophistication Theater: Faced with limited data, the temptation is to turn to more complex architectures like LSTMs or other deep learning models. This would be a methodological error. Such models, with their high capacity, would not learn the underlying physics but would instead perfectly memorize the few examples provided, creating an illusion of sophistication while degrading generalization. Our current approach, combining expert feature engineering with a simpler Random Forest, is likely close to the optimal strategy given these severe data constraints. True scientific honesty may consist in recognizing these limits rather than seeking purely technical solutions to a problem that is fundamentally data-limited.

6.5.2 The Inductive Bias of a Synthetic Ground Truth

The model's performance is measured against a "ground truth" that is itself a heuristic construct. The "Pre-Event" and "Post-Event" labels, for instance, do not originate from independent physical measurements but from our own rules applied to the "High" class. Therefore, when the model learns to identify these phases, it is not discovering a natural phenomenon; it is merely proving its ability to learn our labeling function. This introduces a significant risk of tautology in the interpretation of the results, where we congratulate the model for rediscovering a pattern we ourselves have imposed.

6.5.3 Correlation, Not Causation

It must be emphasized that this model is a correlation-finding engine, not a physical simulator. Even a robust Permutation Feature Importance analysis only reveals which statistical patterns in the input data are most strongly associated with the labels. It does not prove causation. The model has not learned volcanology; it has learned to map a specific set of VRP statistics to a specific set of labels. The physical interpretation of this mapping remains a human endeavor, fraught with the risk of confirmation bias.

6.5.4 Conclusion of the Critique: A Hypothesis-Generation Tool

Given these profound limitations, this application cannot be considered a validated predictive model. It is more accurately described as a prototype for a highly efficient, interactive data exploration and hypothesis-generation tool.

Its primary scientific value is not in its predictive score, but in its ability to:

1. Quantify the marginal predictive value of different physical indicators (via PFI).
2. Allow experts to interactively explore the sensitivity of the system to different decision thresholds.
3. Serve as a framework for rigorously testing new features or models on a consistent baseline.

Any claims of its forecasting capability must be heavily qualified. The path to a truly robust predictive system would require not just a better algorithm, but a fundamentally richer dataset, integrating multiple data streams (seismic, geodetic, gas) over a much wider diversity of eruptive events.