DOCUMENTATION

1 Content

2 Introduction

2.1 What is A.R.C.?

A.R.C. (Automated Research for Cybersecurity) is an end-to-end framework that turns raw security data—especially smart-grid/ICS datasets—into reproducible, defensible experiments. It standardizes the full workflow from data loading and validation to splitting, preprocessing, model training/evaluation, comparison, and report generation. A.R.C. is delivered as a Streamlit application with clear presets and logging so that different users can obtain the same results from the same inputs.

2.2 Why does A.R.C. exists

Public cybersecurity datasets often suffer from inconsistent labels, drifting features, weak documentation, and class imbalance. These issues inflate reported performance and make results hard to trust or reproduce. A.R.C. solves this by enforcing consistent preparation, leakage-aware evaluation, and complete run provenance.

2.3 What A.R.C. does

- Ingests data: CSV/JSON and PCAP/PCAPNG-derived flows; timestamp detection/normalization; multi-file handling
- Validates datasets: schema conformance, types/coercion, label health (entropy, rarity, spelling/merge rules), temporal sanity checks
- Splits safely: random, stratified, and time-based splits with leakage controls and class-balance awareness
- Preprocesses consistently: imputation, encoding, scaling, feature selection; model-specific presets (RF/GBM/SVM/MLP/LR)
- Trains & evaluates: supervised baselines with a consistent metric suite (Accuracy, macro/micro F1, PR-AUC, ROC-AUC, inference time)
- Compares fairly: RAW vs. PRE (preprocessed) results across datasets and models
- Reports & logs: structured PDF report, JSON run metadata, and exportable figures/tables for papers or audits
- Containerized: Dockerized execution for consistent local and CI environments

2.4 Workflow at a glance

1. Load dataset(s) and detect timestamps.
2. Run schema and label validation; apply fixes if needed.
3. Choose split strategy (random/stratified/time) with leakage checks.
4. Select a preprocessing preset (or configure steps manually).
5. Train/evaluate one or more models; compare RAW vs. PRE.
6. Export artifacts: CSV/PKL/JSON, figures/tables, and a PDF report.

2.5 Who is A.R.C. for

- Researchers & students in cybersecurity (ICS/smart-grid focus) who need rigorous, repeatable baselines.
- Data scientists/engineers standardizing evaluation across heterogeneous datasets.
- Practitioners preparing audit-ready, shareable analyses with full provenance.

3 Architecture

3.1 High-level architecture

A.R.C. comprises eight cooperating modules arranged into six functional groups that together realize the end-to-end workflow, as seen in figure 3.8. Data ingestion is handled by the DataLoader, which accepts heterogeneous inputs (e.g., CSV, PCAP) and harmonizes them into A.R.C.'s internal tabular schema with designated timestamp and label fields. Validation consists of two complementary components the Schema Validator, which inspects structural and temporal properties (feature inventory, datatypes, duplicates, ordering, coverage), and the Label Validator, which audits label completeness and consistency and summarizes class distributions. Splitting is performed by the Splitter, which generates train/validation/test partitions (random, stratified, or time-based) with documented seeds and split manifests. Preprocessing is executed by the Preprocessor, a configurable pipeline that applies cleaning, encoding, feature filtering/selection, scaling or normalization, outlier handling, imbalance correction, and optional time-feature construction, all transformations are logged for replay. Modeling and evaluation comprises two modules. The Trainer & Evaluator, which fits baseline learners on the preprocessed splits and reports performance KPIs, efficiency KPIs and further graphs, like confusion matrices or ROC curves, under a common interface. And the Compare module, which trains the same model on not preprocessed ("RAW") data, with minimal preprocessing, and quantifies the delta to the preprocessed results produced by the Trainer & Evaluator. The reporting is provided by the Reporter, which compiles a PDF consolidating insights, parameter choices, figures, tables, and outcomes from each stage. All generated artifacts (datasets, split subsets, trained models, metadata) are available to download.

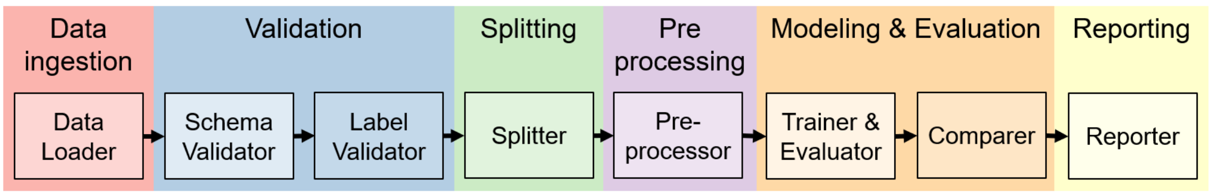

Figure 1 A.R.C. high-level architecture

All modules are orchestrated by a pipeline controller exposed through a Streamlit interface. This controller enables interactive configuration while preserving a deterministic execution order. Decoupled back-end components enable interactively use. The layered structure follows separation of concerns principles, improving maintainability and reliability across the backend.

3.2 Design principles

3.2.1 Modularity

Cybersecurity data preparation spans heterogeneous tasks, from missing value imputation and duplicate removal to temporal splitting and class rebalancing. Rather than entangling these steps in monolithic scripts, A.R.C. decomposes functionality into modules with stable, minimal interfaces. Components can be reordered, omitted or replaced without destabilizing the pipeline.

3.2.2 Extensibility

A.R.C. is designed to evolve. New methods can be registered behind existing interfaces without rewriting legacy code. An autoencoder based anomaly detector can be inserted into the modeling stage, or a novel resampling strategy introduced during preprocessing, without adverse effects on other modules.

3.2.3 Reproducibility

Each step records parameters, random seeds and outputs as machine- and human- readable metadata. Reports capture results together with preprocessing lineage, such as which features were imputed or dropped and how splits were generated, so that experiments can be reconstructed exactly. Reproducibility is an architectural property rather than an afterthought.

3.2.4 Transparency

All components build on peer-reviewed, open-source libraries (e.g. pandas, scikit- learn, imbalanced-learn) and expose decisions through visual reports (e.g. class distributions, drift plots, confusion matrices). The Streamlit interface renders these steps auditable, accessible and understandable, even to researchers who are not machine-learning specialists.

3.3 Dataflow

Figure 2 describes the end-to-end data path in A.R.C., from raw inputs to exportable artifacts. The Data Loader admits one or more RAW datasets (e.g. CSV or PCAP-derived tables) and normalizes them into A.R.C.'s internal schema with designated timestamp and label fields. The resulting table flows to the Schema Validator, which inspects structural and temporal properties and to the Label Validator, which audits completeness and consistency. Both components surface issues before modeling and record their findings for later reporting.

Validated data proceed to the Splitter, where train, validation and test subsets are produced under random, stratified, or time-based regimes with documented parameters. The Preprocessor then applies the configured transformations (cleaning, encoding, feature filtering/selection, scaling or normalization, outlier handling, imbalance correction, and optional time-feature construction) yielding the PP (preprocessed) datasets used for training. The Trainer & Evaluator consumes these PP splits, fits the selected baseline learner and reports both discrimination and efficiency metrics under a common interface.

A second branch in Figure 2, labeled Minimal preprocessing, routes the same splits through a lightweight RAW pipeline. The Compare module trains the identical model family on these minimally processed data and contrasts the resulting "RAW" metrics with the PP metrics produced by the Trainer & Evaluator, thereby quantifying the contribution of standardized preprocessing while holding model choice constant.

Throughout the pipeline, modules emit artifacts that can be downloaded directly from the interface. The data loader can export the harmonized dataset as CSV. The Splitter saves per-split CSVs. The preprocessor writes PP datasets together with a JSON metadata manifest that captures parameter choices and transformation lineage. The Trainer & Evaluator persists the trained model on PP data and its metric bundle, the Compare module does the same for the RAW branch. Finally, the Reporter assembles a PDF that consolidates insights, parameter settings, figures, and results from each stage and it also exposes the complete log to support exact reruns.

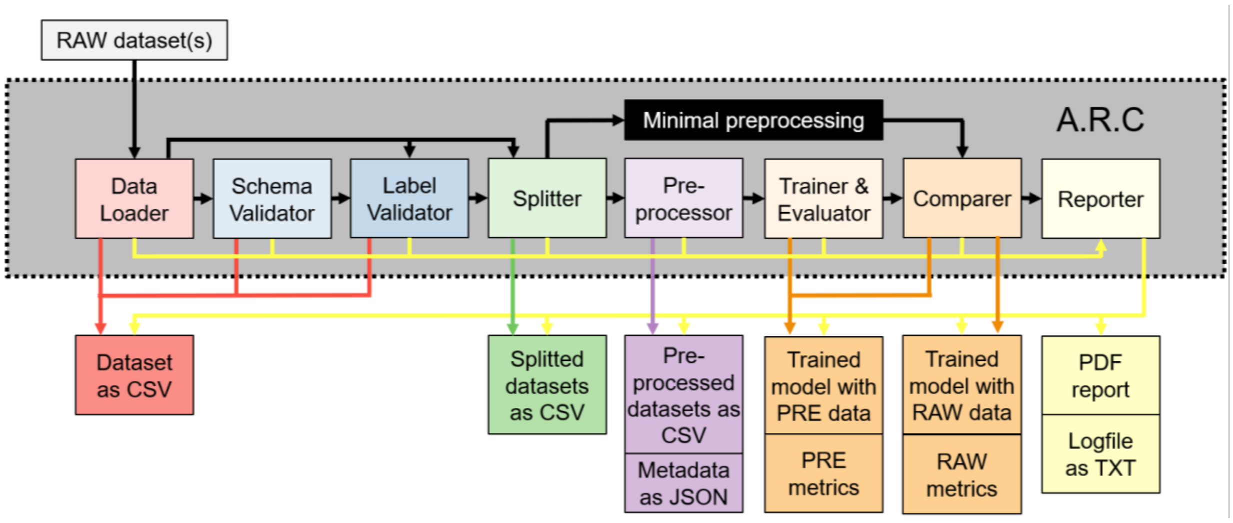

Figure 2 Dataflow through A.R.C.

3.4 Software stack

A.R.C. was build with Python (3.13.6). User interaction and orchestration are provided by Streamlit (1.48.0). Tabular processing relies pandas for joins, aggregations, input and output. NumPy is supplying numerical kernels. scikit-learn contributes preprocessing utilities (e.g. scalers, encoders, PCA), models and evaluation tooling, while imbalanced-learn provides methods for class-imbalance mitigation. Interactive visualizations are implemented with Plotly, and ReportLab handles PDF export to produce citable, shareable reports.

3.4.1 Python as a foundation

Python offers readable and maintainable code with access to high performance kernels through vectorization and extensions. Its flexibility supports rapid experimentation without sacrificing execution speed where it matters.

3.4.2 Streamlit for accessibility

The Streamlit front-end lowers the barrier to adoption and encourages good experimental hygiene. Inputs are explicit, outputs are visible and runs can be shared as self-contained dashboards.

3.4.3 Pandas and NumPy for data handling

Cybersecurity datasets are predominantly tabular, large, and occasionally messy. pandas NumPy support the core transformations (e.g. joins, filters, resampling) that determine preprocessing quality within A.R.C.

3.4.4 Scikit-learn and imbalanced-learn for modeling

These libraries provide stable APIs, deterministic behavior when seeded, and well tested implementations across preprocessing, model training, cross-validation, metrics, and resampling. While reducing engineering overhead and enhancing reliability.

3.4.5 Plotly and ReportLab for reporting

Exploratory analysis benefits from interactive visualizations and reproducibility benefits from fixed artefacts. Plotly covers the former, while ReportLab consolidates decisions, metrics, and figures into PDFs that do not depend on a live session.

3.5 Folder structure

The repository separates user interface and scientific logic. Streamlit pages (e.g. pages/01_Data_Loader.py) define UI elements and visuals. Computational logic for schema checks, preprocessing, and training resides under functions/ (e.g. function/f01_Dataloader.py), enabling import for testing or notebook use independent of the UI. This arrangement preserves a lightweight, replaceable UI while keeping scientific logic reusable and extensible within A.R.C.Structure:

- A_R_C.py: App bootstrap and navigation. Wires UI pages to core functions.
- .streamlit/config.toml: Settings for Streamlit
- pages/: Thin Streamlit pages. Each page owns a well-defined slice of st.session_state and calls functions from functions/
    - 0_Logger.py:
        - Show log
        - Add custom logs
    - 1_DataLoader.py:
        - File selection
        - Timestamp and label column selection
        - Dataset insights
    - 2_SchemaValidator.py:
        - Dataset preview
        - Schema insights
        - Schema modification controller
    - 3_LabelValidator.py:
        - Dataset preview
        - Label insights
        - Label modification controller
    - 4_Split.py:
        - Explanation of split methods
        - Split controller
        - Split quality insights
    - 5_Preprocess.py:
        - Explanation of models
        - Preprocessing explanations
        - Preprocessing controller (toggle switch)
    - 6_TrainEvaluate.py:
        - Button to start training
        - Display of metrics and graphs
    - 7_Compare.py
        - Automatic training on raw data
        - Display of metrics and graphs comparison
    - 8_ReportLogger.py:
        - Download buttons for all artifacs
        - PDF report configuration and generate button
- functions/: Pure Python logic (no UI). Deterministic given the same inputs and seeds. Reusable in notebooks
    - f00_Logger.py:
        - Functions for logging

- f00_Sidebar.py:
    - Functions for logging
- f00_VarInitialisator.py:
    - Initializing all session_states
- f01_DataLoader.py:
    - handles CSV/JSON/PCAP-derived flows
- f02_SchemaValidate.py:
    - Functions for schema validator checks
    - Functions for schema validator modification
- f03_LabelValidate.py:
    - Functions for label validator checks
    - Functions for label validator modification
- f04_Splitter.py:
    - Functions for splitting
    - Functions for split quality checks
- f05_Preprocessor.py:
    - Setting of presets
    - Functions for preprocessing
    - Preprocessing runner
- f06_TrainerEvaluator.py
    - Functions for model training
    - Functions for model evaluaton
- f09_Reporter.py:
    - Functions for generation of the PDF report

3.6 Integration strategy

Integration proceeds across three layers. The UI layer supplies controls, configuration, and visualization through Streamlit components. The processing layer implements validation, preprocessing, training, and evaluation as callable functions. The data layer manages shared datasets and parameters via streamlit's session_state. Because interfaces are stable, modules remain optional and can be bypassed or replaced without breaking the workflow that A.R.C. enforces.

3.7 Module interaction

Modules communicate primarily through session_state, which exposes shared data and configuration to all pages. A JSON log records key decisions and parameter values, producing an auditable trail from ingestion to reporting. Loose coupling allows skipping modules entirely or inserting domain-specific feature engineering without modifying downstream components. UI orchestration clarifies execution order while avoiding rigid, hard-wired dependencies.

3.8 State management

Long-running workflows require continuity of state. Datasets are stored under a key (e.g. _DF). Intermediate outputs such as schema statistics and preprocessing summaries persist as Python objects and user configurations follow the user across modules. Serialized snapshots of session state enable exact reconstruction of experiments, directly supporting the reproducibility goals of A.R.C..

3.9 Workflow orchestration

The default execution path is semi linear as seen in figure 1. datasets are loaded, schemas and labels are validated, splits are created and audited, preprocessing is applied, models are trained and evaluated, results are compared against a minimal “RAW” baseline, and reports and artifacts are generated. Individual steps can be re-run or skipped as required by the study design, while the controller maintains a coherent lineage for all outputs.

3.10 Export handling

Artifacts persist beyond the session. Datasets at each stage are exported as CSV, metadata and parameters as JSON, trained models as PKL and consolidated reports as PDF. The Streamlit interface provides download widgets for convenient access to all outputs produced by A.R.C..

4 Session state

The following table provides an overview of all Streamlit session state variables used in the A.R.C. framework. For each entry, it lists the session key, its data type, and a short description of its purpose in the pipeline. This overview serves as a reference for developers and users to understand how configuration choices, intermediate results, and user interactions are stored and reused across pages. It also supports debugging and reproducibility by making the internal state of the application transparent.

Name	Description	Type
_LogData	Stores the log	List
_DF	Stores the dataset	Dataframe
_HasTimeStamp	True if dataset has a timestamp	Boolean
_TimeStampCol	Stores name of the timestamp column	String
_HasLabel	True if dataset has a timestamp	Boolean
_LabelCol	Stores name of the label column	String
_DL_Start		Boolean
_DL_DataLoaded	Flag wether dataset was loaded	Boolean
_DL_Mode	Set the dataload mode „single“ or „multiple“ file	String
_DL_Filename	Name of dataset	String
_DL_UploadedFiles	Names of combined files	List
_SV_ReqCol	Listed required columns	String
_SV_RenameColM	Listed rename matrix	String
_SV_RenameColMMap	Mapped rename matrix	List
_SV_TDelta	Checkbox input to add a delta	Boolean
_SV_ColToDrop	Name oft he coumn to drop	String
_SV_NormCol	Flag wether coluns were normalized	Boolean
_SV_RenameCol	Flag wether columns were renamed	Boolean
_SV_RenameCols	List of renamed collumns	List
_SV_SortByT	Flag wether columns were sorted by timestamp	Boolean
_SV_DeltaT	Flag wether a time delta was added	Boolean
_SV_DropCol	Flag wether columns were droped	Boolean
_SV_DroppedCols	List of droped columns	List
_SV_DropDup	Flag wether duplicates were droped	Boolean
_SV_DropDupCount	Count of droped duplicates	Integer
_LV_InputRareClasses	Thershold for rare attacks	Double
_LV_InputDominantClasses	Threshold for dominant attacks	Double
_LV_TDNumBins	Number of bins for time drift analysis	Integer
_LV_TDTimeline	Flag wether the timedistribution should be shown as a timeline	Boolean
_LV_TDTime	Flag wether the timedistribution should be shown as a timebins across time	Boolean
_LV_TDRecords	Flag wether the timedistribution should be shown as timebins across record counts	Boolean
_LV_RenameLabel	Flag wether labels were renamed	Boolean
_LV_RenamedLabels	List of renamed labels	List
_LV_RenameM	Reneamematrix	String

_LV_RenameMMap	Mapping of renamematrix	List
_SP_SplitMethod	Stres Splitmethod	String
_SP_RandomState	Stores the random state for statified and random split	Interger
_SP_GapRatio	Gap for timebased split in percent 0-100	Interger
_SP_TestSize	Percentage of dataset that need tob e the test subset	Integer
_SP_ValSize	Percentage of dataset that needs tob e the validation subset	Integer
_SP_IsSplit	Falg wether dataset is split	Boolean
_SP_X_Train	Splitted subset	Dataframe
_SP_y_Train	Splitted subset	Dataframe
_SP_X_Validate	Splitted subset	Dataframe
_SP_y_Validate	Splitted subset	Dataframe
_SP_X_Test	Splitted subset	Dataframe
_SP_y_Test	Splitted subset	Dataframe
_PP_Is_PP	Falg wether data is preprocessed	Boolean
_PP_Model	Selected mashinelearning model for preselection	Sring
_PP_DD	Flag DopDuplicated	Boolean
_PP_MV	Falg impute missing values	Boolean
_PP_MV_N	Imputationmethod for missing values for numerical columns	String
_PP_MV_C	Imputation method for missing values for categorical columns	String
_PP_BR	Falg bucket rare categories	Boolean
_PP_BR_MF	Min frequency for bucket rare categories	Integer
_PP_EN_GA	Falg for generic/advanced encoding	Boolean
_PP_EN_GA_O	Options for generic advanced encoding	String
_PP_GMB	Flag for generic advanced encoding	Boolean
_PP_GMB_M	Mode for gmb	String
_PP_CE	Flag for count encoding	Boolean
_PP_TME	Flag for target mean encoding	Boolean
_PP_TME_NS	N splits for target mean encoding	Integer
_PP_TME_NOS	Noise std for target mean encoing	Double
_PP_TME_GS	Global smothing for target mean encoing	Double
_PP_DC	Flag frop constant	Boolean
_PP_DC_V	Variance thershold for drop constant	Double
_PP_DHC	Flaf for dtop highly corolated	Boolean
_PP_DHC_T	Thershold for drop highly corolated	Double
_PP_FS_RF	Falg for feature selection by random forest	Boolean
_PP_FS_RF_TN	Top n to keep for featureselection by random forest	Integer
_PP_FS_MI	Flag for feature selction by mutual information	Boolean
_PP_FS_MI_TK	Top k for featureselction by mutual information	Integer
_PP_FS_A	Falg for feature selection by Anova F	Boolean
_PP_FS_A_TK	Top k for feature selection by Anova f	Interger
_PP_PCA	Flag for PCA reduction	Boolean
_PP_PCA_N	N components for PCA reduction	Integer
_PP_SC	Falg for scaling	Boolean

_PP_SC_S	Scaler for for scaling	String
_PP_SC_S_MIN	Min a for min max scaler	Integer
_PP_SC_S_MAX	Max a for MinMax scaler	Integer
_PP_L2N	Falg for row-wise L2 normalisation	Boolean
_PP_OC	Flag for outlierclipping	Boolean
_PP_OC_M	Clipping method for outlier clipping	String
_PP_OC_WW	Wisker width for IQR outlierclipping	Double
_PP_OC_PL	Percentile low of percentile outlier clipping	Double
_PP_OC_PH	Percentile high for percentil outlier clipping	Double
_PP_LOG	Flag for log1p transformation	Boolean
_PP_CW	Flag for classweights	Boolean
_PP_CW_S	Schema for class balancing	String
_PP_RS	Falg for resampeling	Boolean
_PP_RS_M	Method for resampeling	String
_PP_RS_SK	K neighbour for Smote resameling	Integer
_PP_SVM	Flag SVM kernel approximation	Boolean
_PP_SVM_M	SVM kernel approximation method	String
_PP_SVM_NNC	Nönstöm n components for SVM kernel approximation	Integer
_PP_SVM_RFFNC	N features for random fourir features	Integer
_PP_X_Train	Preprocessed subset	Dataframe
_PP_y_Train	Preprocessed subset	Dataframe
_PP_X_Validate	Preprocessed subset	Dataframe
_PP_y_Validate	Preprocessed subset	Dataframe
_PP_X_Test	Preprocessed subset	Dataframe
_PP_y_Test	Preprocessed subset	Dataframe
_PP_ClassWeights	Class weights	List
_PP_SWTR	Sample weights Train	List
_PP_SWVA	Sample weights validate	List
_PP_SWTE	Sample weights test	List
_PP_META	Metadata	List
_PP_LE	Label encoding	List
_TE_Model	Trained model	
_TE_PTrained	Falg wether model is trained with preprocessed data	Boolean
_TE_PRes	Result from training on preprocessed data	
_C_NPTrained	Falg wether model is trained with raw data	Boolean
_C_NPRes	Result from training on raw data	

5 Code

5.1 Functions

5.1.1 Schema Validator

5.1.1.1 Sumary

analyze_sparsity	Computes missing-value statistics across the entire dataset.
detect_granularity	Determines whether data is flow-level, packet-level, or session-level.
analyze_feature_set	Categorizes all features by dtype (numerical, categorical, etc.).
analyze_time_span	Analyzes temporal coverage (start, end, duration, intervals).
check_column_presence	Compares dataframe columns against required schema.
parse_rename_matrix	Parses user-defined rename rules ("old = new") into a mapping.
normalize_column_names	Normalizes all feature names (lowercase, no spaces)
normalize_column_name	Normalizes a single feature name.
rename_column	Renames a single feature using a rename mapping
rename_columns	Batch-renames multiple features and logs all changes
sort_by_timestamp	Sorts dataframe by timestamp and optionally adds delta_seconds
drop_column	Drops a specified feature and logs the operation
drop_duplicates	Removes duplicate rows and logs the amount removed
schemaValidatorTotal	High-level summary: sparsity, granularity, types, timestamp quality

5.1.1.2 Detailed

5.1.1.2.1 Analyse_sparsity(df)

Inputs:

- df — pandas DataFrame

Outputs:

- Dictionary containing:
    - Total missing cells
    - Missing percentage
    - Per-column missing percentages

Description:

Computes sparsity statistics across the dataset. The function counts all missing entries, calculates the global percentage of missing values, and identifies columns with missing values. Columns without missing data are excluded from the per-column output.

5.1.1.2.2 detect_granularity(df)

Inputs:

- df — pandas DataFrame

Outputs:

- String: "Flow-level", "Packet-level", "Session-level", or "Unknown / Custom"

Description:

Determines the semantic granularity of the dataset by inspecting normalized column names. If flow identifiers exist, the dataset is treated as flow-level; if packet properties exist, as packet-level; and if session identifiers exist, as session-level.

5.1.1.2.3 analyze_feature_set(df)

Inputs:

- df — pandas DataFrame

Outputs:

- Dictionary containing lists of numerical, categorical, boolean, object, and datetime columns

Description:

Classifies all features of the dataframe by their pandas dtype. This helps understand the structure of the dataset and supports preprocessing modules that depend on data types.

5.1.1.2.4 analyze_time_span(df, timestamp_col)

Inputs:

- df — pandas DataFrame
- timestamp_col — Name of timestamp column

Outputs:

- Dictionary containing:
  - Start timestamp
  - End timestamp
  - Total duration
  - Most common interval
  - Average interval
- Exception (if analysis fails)

Description:

Analyzes the temporal characteristics of the dataset. Converts the column to datetime, identifies valid timestamps, and calculates time span and interval metrics. If parsing or analysis fails, an error message is returned.

5.1.1.2.5 check_column_presence(df, required_columns)

Inputs:

- df — pandas DataFrame
- required_columns — list of expected columns

Outputs:

- Dictionary with missing and extra column names

Description:

Compares the actual dataset structure with a predefined schema. Identifies missing required columns and unexpected extra columns.

5.1.1.2.6 parse_rename_matrix(text)

Inputs:

- text — multiline string containing rename rules (old = new)

Outputs:

- Dictionary mapping old column names to new names

Description:

Parses user-defined rename rules into a usable mapping. This function supports flexible and readable configuration of renaming logic.

5.1.1.2.7 normalize_column_names(df)

Inputs:

- df — pandas DataFrame

Outputs:

- DataFrame with normalized column names

Description:

Normalizes all feature names by lowercasing, trimming whitespace, and removing spaces. Logs the operation and sets internal status flags.

5.1.1.2.8 normalize_column_name(col)

Inputs:

- col — string

Outputs:

- Normalized string

Description:

Applies the same normalization logic as above to a single column name.

5.1.1.2.9 rename_column(col_name, rename_map)

Inputs:

- col_name — original column name
- rename_map — dict or text defining rename rules

Outputs:

- New column name

Description:

Renames a single column based on a provided mapping. If the mapping is provided as text, it is parsed using parse_rename_matrix.

5.1.1.2.10 rename_columns(df, rename_map)

Inputs:

- df — pandas DataFrame
- rename_map — dict or rename-matrix string

Outputs:

- DataFrame with renamed columns

Description:

Applies renaming rules to multiple columns at once. Logs every rename operation and stores metadata for reporting.

5.1.1.2.11 sort_by_timestamp(df, timestamp_col, add_delta)

Inputs:

- df — pandas DataFrame
- timestamp_col — column name
- add_delta — boolean

Outputs:

- Sorted DataFrame (optional delta_seconds added)

Description:

Sorts the dataframe chronologically. If requested, also computes per-record time differences. All actions are logged.

5.1.1.2.12 drop_column(df, col)

Inputs:

- df — pandas DataFrame
- col — column name

Outputs:

- DataFrame without the specified column

Description:

Drops a column and logs the action. Tracks dropped columns for report generation.

5.1.1.2.13 drop_duplicates(df)

Inputs:

- df — pandas DataFrame

Outputs:

- DataFrame without duplicates

Description:

Removes duplicate entries and logs the number of deleted rows.

5.1.1.2.14 schemaValidatorTotal(df, hastimestamp, timestamp_col)

Inputs:

- df — pandas DataFrame
- hastimestamp — Boolean
- timestamp_col — column name

Outputs:

- Dictionary containing the full schema analysis summary

Description:

Combines multiple validation checks into a single structured output: sparsity, granularity, dtype summary, mixed types, duplicates, timestamp ordering, and time-span metrics.

5.1.2 Label Validator

5.1.2.1 Sumary

Function	Short Description
get_rare_classes	Identifies classes that occur below a user-defined percentage threshold.
get_dominant_classes	Identifies classes that occur above a user-defined percentage threshold.
get_timebin	Creates time- or record-based bins and computes per-interval label distributions.
plot_pie_chart	Creates a label distribution pie chart.
plot_bar_chart	Generates a label distribution bar chart
plot_state_timeline	Builds a continuous timeline showing label transitions over time
plot_timebin	Plots stacked bar charts of label counts per time bin
format_time_bin_table	Formats time-bin data into a compact interval table
rename_row	Renames label values using a mapping; logs each applied rule
labelValidaorTotal	Full label integrity summary: missing values, inconsistent labels, entropy, and class distribution

5.1.2.2 Detailed

5.1.2.2.1 get_rare_classes(df, label_col, input_r_c)

Inputs:

- df: DataFrame
- label_col: label column name
- input_r_c: percentage threshold

Outputs:

- Series of class names with frequencies below the given threshold

Description:

Computes the relative frequencies of all labels, compares them against a user-defined threshold, and returns the set of classes considered “rare”. This supports imbalance detection and preprocessing decisions (e.g., grouping minor classes).

5.1.2.2.2 get_dominant_classes(df, label_col, input_d_c)

Inputs:

- df: DataFrame
- label_col: label column name
- input_d_c: percentage threshold

Outputs:

- Series of class names with frequencies above the threshold

Description:

Identifies classes that dominate the dataset by exceeding a user-specified percentage. Supports imbalance detection and entropy analysis.

5.1.2.2.3 get_timebin(df, label_col, timestamp_col, mode, n_bins)

Inputs:

- df: DataFrame
- label_col: label column name
- timestamp_col: timestamp column
- mode: "time" or "records"
- n_bins: number of bins

Outputs:

- Wide-format DataFrame containing, for each time bin:
    - Interval label
    - Per-class percentages
    - Per-class raw counts

Description:

Creates equally sized temporal or record-based intervals and computes a detailed per-interval label distribution. Ensures human-readable interval labels and synchronized class ordering. Used for drift detection and temporal imbalance analysis.

5.1.2.2.4 plot_pie_chart(labels, values, height)

Inputs:

- labels: class names
- values: class frequencies
- height: plot height

Outputs:

- Plotly pie chart

Description:

Creates a categorical distribution pie chart with inside percentage labels. Used for quick visualization of class imbalance.

5.1.2.2.5 plot_bar_chart(labels, values, height)

Inputs:

- labels: class names
- values: class frequencies

Outputs:

- Plotly bar chart

Description:

Creates a bar chart showing counts per class, sorted by label. Offers a more precise view than the pie chart.

5.1.2.2.6 plot_state_timeline(df, label_col, timestamp_col)

Inputs:

- df: DataFrame
- label_col: label column
- timestamp_col: timestamp column

Outputs:

- Plotly timeline figure

Description:

Builds a continuous timeline that highlights periods in which a label remains constant. Useful for understanding session-style behavior, state durations, anomalies, and rapid label switching.

5.1.2.2.7 plot_timebin(data, height)

Inputs:

- data: output of get_timebin
- height: plot height

Outputs:

- Plotly stacked bar chart

Description:

Visualizes label counts per time interval using stacked bars. Reveals changes in class frequencies over time and highlights drift or bursts of specific labels.

5.1.2.2.8 format_time_bin_table(data)

Inputs:

- data: output of get_timebin

Outputs:

- Formatted DataFrame with human-readable “count (percentage)” entries

Description:

Converts the wide time-bin data into a compact table suitable for reports, where each cell shows both count and percentage for a given interval.

5.1.2.2.9 rename_row(df, column, rename_map)

Inputs:

- df: DataFrame
- column: column whose values should be renamed
- rename_map: dict of old → new labels

Outputs:

- DataFrame with renamed label values

Description:

Applies value-level renaming within a label column. Logs each applied rule and records changes in session state. Only modifications with actual matches are logged, enabling transparent rename auditing.

5.1.2.2.10 labelValidaorTotal(df, label_col)

Inputs:

- df: DataFrame
- label_col: label column

Outputs:

- Dictionary containing:
    - Missing labels flag
    - Spelling inconsistencies (case variations)
    - Entropy value and interpretation
    - Full class counts and lists

Description:

Perform a full integrity assessment of label quality. Detects missing values, inconsistent spellings (case mismatches), class imbalance through entropy, and provides raw and ordered class distributions. This serves as the core analytical summary for the label validation step.

5.1.3 Splitter

5.1.3.1 Sumary

split_dataset	Splits the dataset into train/validation/test using random, stratified, or time-based logic.
getDistributionSet	Computes class distribution per split (counts and percentages within each set).
getDistributionLabel	Computes per-label distribution across splits (percentages per label summing to 100%).
getDistributionTotal	Computes global distribution across splits (percentages over the entire dataset).
Getunseen	Detects labels that appear in validation/test but not in the training set.
plot_distribution_bars	Plots stacked horizontal bars showing split-level percentages per label.
plot_distribution_label	Plots stacked vertical bars showing, per label, how records are distributed across splits.
make_distribution_table	Builds a human-readable table “count (percentage)” for train/val/test per label.

5.1.3.2 Detailed

5.1.3.2.1 split_dataset(df, label_col, time_col, method, test_size, val_size, gap_ratio, random_state)

Inputs:

- df: pandas DataFrame (full dataset)
- label_col: name of the target/label column
- time_col: name of the timestamp column (for time-based splits)
- method: "Random Split", "Stratified Split", or "Time-based Split"
- test_size: fraction of records allocated to the test set (0–1)
- val_size: fraction of records allocated to the validation set (0–1)
- gap_ratio: fraction of records used as a temporal gap between train and val+test (time-based only)
- random_state: integer seed for reproducibility

Outputs:

- X_train, y_train
- X_validate, y_validate
- X_test, y_test

Description:

Splits the dataset into training, validation, and test sets using one of three strategies:

- Random Split: purely random splitting using train_test_split.
- Stratified Split: preserves label distribution across splits using stratify=y.
- Time-based Split: sorts by a timestamp column, assigns earliest records to train, skips a configurable temporal gap, and then allocates the remaining records chronologically to validation and test (no shuffle, no stratify).

The function validates that the requested fractions are valid (val + test [+ gap] < 1), logs the chosen strategy and percentages, and sets st.session_state._SP_IsSplit = True on success.

5.1.3.2.2 getDistributionSet(y_train, y_validate, y_test)

Inputs:

- y_train: training labels (Series)
- y_validate: validation labels (Series)
- y_test: test labels (Series)

Outputs:

- dist_df: DataFrame indexed by label with columns:
    - train_count, val_count, test_count
    - train_%, val_%, test_% (percentages within each split)

Description:

Computes per-label counts and percentages in each split, normalizing percentages per set (train, validation, test). This shows how each split is composed in terms of labels and is used to check if the split preserved the original class distribution.

5.1.3.2.3 getDistributionLabel(y_train, y_validate, y_test)

Inputs

- The function signature includes y_train, y_validate, y_test, but internally it reads them from st.session_state._SP_y_Train, _SP_y_Validate, _SP_y_Test.

Outputs:

- dist_df: DataFrame indexed by label with:
    - train_count, val_count, test_count
    - train_%, val_%, test_% (percentages per label, summing to 100% across splits for each label)

Description:

Analyzes, for each label, how its records are distributed across train, validation, and test. Percentages are normalized per label, so train_% + val_% + test_% = 100% for each label. This reveals whether certain labels are mostly located in one split (e.g., only in test).

5.1.3.2.4 getDistributionTotal(y_train, y_validate, y_test)

Inputs:

- As above, labels are effectively taken from st.session_state._SP_y_Train, _SP_y_Validate, _SP_y_Test.

Outputs:

- dist_df: DataFrame indexed by label with:
    - train_count, val_count, test_count
    - train_%, val_%, test_% (percentages over the total dataset)

Description:

Computes how the entire dataset is distributed across train, validation, and test globally. Percentages are normalized by the total number of records (sum of all labels in all splits). This shows how much of the dataset mass each split represents for each label.

5.1.3.2.5 getunseen(y_train, y_validate, y_test)

Inputs:

- y_train: training labels
- y_validate: validation labels
- y_test: test labels

Outputs:

- Dictionary with:
    - unseen_test: sorted list of labels that appear in test but not in train
    - unseen_val: sorted list of labels that appear in validation but not in train

Description:

Detects labels that are unseen during training but appear in validation or test. This is critical to identify potential data leakage or unrealistic evaluation scenarios where the model is evaluated on classes it never saw during training.

5.1.3.2.6 plot_distribution_bars(dist_df, height)

Inputs:

- dist_df: distribution DataFrame (from one of the getDistribution* functions), containing percentage columns (train_%, val_%, test_%).
- height: plot height in pixels

Outputs:

- Plotly horizontal stacked bar chart

Description:

Creates a horizontal stacked bar chart where:

- The y-axis shows the splits (Train, Val, Test).
- The x-axis shows the percentage.
- Colors encode labels (classes).

This is a set-level view showing how each split is composed in terms of labels.

5.1.3.2.7 plot_distribution_label(dist_df, height)

Inputs:

- dist_df: distribution DataFrame with percentage columns
- height: plot height

Outputs:

- Plotly vertical stacked bar chart

Description:

Creates a vertical stacked bar chart where:

- The x-axis shows labels.
- The y-axis shows the percentage (0–100%).
- Colors encode the dataset split (Train, Val, Test).

This is a label-centric view, revealing for each label how its records are distributed over the three splits.

5.1.3.2.8 make_distribution_table(dist_df)

Inputs:

- dist_df: distribution DataFrame with *_count and *_% columns

Outputs:

- out: formatted DataFrame with columns: Label, Train, Validate, Test, each cell in the form "count (percentage%)"

Description:

Converts the numeric distribution DataFrame into a compact, human-readable table. For each label and split, it combines count and percentage into a single string. This format is well suited for inclusion in reports, PDFs, and thesis tables.

5.1.4 Preprocessor

5.1.4.1 Summary

modelPreset	Returns a configuration dictionary with recommended preprocessing flags and parameters for a chosen model type.
encode_labels	Encodes non-numeric labels into integer IDs based on the training labels only.
drop_duplicates	Removes duplicate rows in X and keeps y aligned for train/val/test, returning stats about dropped samples
impute_numeric_categorical	Imputes missing numeric and categorical values (fit on train; applied to val/test).
bucket_rare_categories	Groups infrequent categorical levels into a shared '__OTHER__' token based on a minimum frequency.
encode_one_hot	Applies one-hot encoding to categorical features (fit on train) and aligns val/test columns to train
encode_ordinal	Applies ordinal encoding to categorical features (fit on train) with a fixed unknown value for unseen categories
prepare_categoricals_gbm	Prepares categorical columns for GBM backends (CatBoost, LightGBM, XGBoost) using string/categorical types or indices
count_encoding	Adds count-encoded versions of categorical columns based on category frequencies observed in the training data
target_mean_encoding_kfold	Performs K-fold target mean encoding with out-of-fold estimates on train and smoothed means on val/test
drop_low_variance	Drops low-variance numeric features and optionally constant non-numeric features across splits
drop_high_corr_numeric	Removes highly correlated numeric features above a given correlation threshold
select_features_by_rf_importance	Selects top features based on Random Forest feature importances, keeping only the most important ones.
select_by_mutual_info	Selects top-k features using mutual information between features and target, optionally factorizing categoricals
select_by_anova_f	Selects top-k numeric features using ANOVA F-scores (with optional zero-variance filtering)
pca_reduction	Applies PCA to numeric features to reduce dimensionality while retaining a target variance fraction
zscore_scale_from_train	Standardizes numeric features (z-score) using train mean/std and applies the same scaling to val/test
robust_scale_from_train	Scales numeric features using median and IQR from train (robust scaling) and applies it to val/test
minmax_scale_from_train	Scales numeric features to a fixed [min, max] range using min/max from train
l2_normalize_rows	Normalizes each sample (row) to unit L2 norm across numeric features
clip_outliers_iqr	Clips numeric features using IQR-based lower/upper bounds computed from the training data
clip_by_percentile_from_train	Clips numeric features using percentile-based lower/upper bounds computed from train

log1p_transform	Applies log1p transform to skewed non-negative numeric columns
compute_class_weights	Computes class weights (balanced or uniform) and an optional positive-class weight for binary tasks
expand_sample_weights	Expands class weights into per-sample weights for a label vector
balance_classes_smote	Applies SMOTE oversampling on numeric features to balance classes in the training data
balance_classes_random_over	Performs random oversampling to balance classes in the training data
balance_classes_random_under	Performs random undersampling to balance classes in the training data
nystroem_rbf_from_train	Generates approximate RBF kernel features via Nyström on numeric columns, fit on train
rbf_sampler_from_train	Generates approximate RBF kernel features via Random Fourier Features (RFF) on numeric columns
add_time_features	Derives time-based features (hour, day-of-week, month, year, weekend flag) from a timestamp column
preprocessModel	Runs the full configurable preprocessing pipeline (all steps above) and returns transformed splits, weights, and metadata

5.1.4.2 Model Presets

5.1.4.2.1 modelPreset(choice, has_timestamp=False)

Returns a configuration dictionary with recommended preprocessing flags and hyperparameters for a given model type (RF, GBM, SVM, MLP, LR, LOF, or a default "no-op" preset). Each preset specifies cleaning, encoding, feature selection, scaling, outlier handling, class imbalance strategy, kernel approximations, and whether to add time features. This configuration is later consumed by preprocessModel to construct a consistent pipeline per model family.

5.1.4.3 Label Encoding

5.1.4.3.1 encode_labels(y_train, y_val=None, y_test=None)

Encodes non-numeric labels into integer IDs based solely on training labels. The function returns encoded y_train, y_val, y_test and a mapping label → int. Validation and test labels that are not in the training mapping become NaN, ensuring no leakage from validation/test into the encoding scheme.

5.1.4.4 Cleaning

5.1.4.4.1 drop_duplicates(X_train, y_train=None, X_val=None, y_val=None, X_test=None, y_test=None)

Removes duplicate feature rows in train/val/test separately while keeping labels aligned with the remaining rows. It returns cleaned splits and a statistics dict (total, kept, dropped per split) and logs how many duplicates were removed for each subset.

5.1.4.4.2 impute_numeric_categorical(X_train, X_val=None, X_test=None, num_strategy='median', cat_strategy='most_frequent')

Imputes missing values in numeric and categorical columns:

- Cleans ±inf to NaN
- Drops all-NaN numeric columns
- Fits SimpleImputer for numeric and categorical on the training data
- Applies the fitted imputers to validation and test.

Returns transformed splits and the numeric/categorical imputers plus a log entry describing strategies and dropped columns.

5.1.4.5 Encoding

5.1.4.5.1 bucket_rare_categories(X_train, X_val=None, X_test=None, min_freq=20, other_token='__OTHER__')

Identifies rare categories in each categorical column (frequency < min_freq) and replaces them with a shared other_token. The mapping per column is stored and applied consistently to train, val, and test. Used to stabilize encodings and avoid exploding cardinality.

5.1.4.5.2 encode_one_hot(X_train, X_val=None, X_test=None, handle_unknown='ignore')

Applies one-hot encoding to all categorical columns using OneHotEncoder fitted on the training data. Numeric columns are kept unchanged. Validation and test are transformed with the same encoder, and then reindexed to have exactly the same columns as the transformed training set (missing columns filled with 0). Returns encoded splits and the fitted encoder.

5.1.4.5.3 encode_ordinal(X_train, X_val=None, X_test=None, categories='auto')

Uses OrdinalEncoder to map categorical columns to integer codes. Unknown or new categories on val/test are assigned -1. Returns encoded splits and the fitted encoder and logs that categorical features were ordinal-encoded.

5.1.4.5.4 prepare_categoricals_gbm(X_train, X_val=None, X_test=None, backend="catboost")

Prepares categorical columns for specific GBM libraries:

- For "catboost", converts categoricals to strings and returns their indices.
- For "lightgbm" or "xgboost", converts to Categorical type with fixed categories from train and returns the categorical column names.

The function ensures that val/test use the same category set as train and logs which backend was prepared.

5.1.4.5.5 count_encoding(OG_X_train, X_train, X_val=None, X_test=None, default=0.0, suffix='_cnt')

Adds count-encoded versions of categorical columns, where each category is mapped to its frequency in the training data. It uses the current frame’s categoricals if available, otherwise falls back to the original training frame. New columns <col>_cnt are appended to train, val, and test, and the mapping dictionaries are returned.

5.1.4.5.6 target_mean_encoding_kfold(X_train, y_train, X_val=None, X_test=None, cols=None, n_splits=5, noise_std=0.0, global_smoothing=10.0, random_state=42)

Implements K-fold target mean encoding:

- For each categorical feature, computes out-of-fold target means on train (to reduce leakage).
- Optionally adds Gaussian noise for regularization.
- Fits a smoothed mapping on the full training data and applies it to val/test.

Creates new <col>_tgtmean features and returns updated splits and a dictionary of mappings per column.

5.1.4.6 Feature Pruning / Selection

5.1.4.6.1 drop_low_variance(X_train, X_val=None, X_test=None, threshold=0.0, drop_constant_non_numeric=True)

Drops numeric/bool columns with variance below threshold and, optionally, non-numeric columns that are constant. The same kept column set is enforced on val/test via reindexing. Returns the reduced splits and lists of kept/removed columns, and logs the removed ones.

5.1.4.6.2 drop_high_corr_numeric(X_train, X_val=None, X_test=None, threshold=0.98)

Computes the absolute correlation matrix of numeric features and drops columns that are highly correlated (≥ threshold) to any other column (using upper triangle logic). Returns splits restricted to the kept columns plus lists of kept and dropped features, and logs the dropped ones.

5.1.4.6.3 select_features_by_rf_importance(...)

Fits a RandomForestClassifier on numeric/bool (plus optionally converted datetime) columns of the training set and uses feature importances to select the top n_keep features. All splits are reindexed to these features; the function returns reduced splits, the kept feature list, a Series of importances, and removed features, plus a log entry.

5.1.4.6.4 select_by_mutual_info(...)

Computes mutual information between each feature and the target. Optionally factorizes categorical features and treats them as discrete. Selects the top k features based on MI scores, reindexes all splits to the selected set, and returns splits, the list of kept features, a Series of MI scores, and removed features.

5.1.4.6.5 select_by_anova_f(...)

Uses ANOVA F-scores (f_classif) on numeric/bool features to select the top k features. Zero-variance variables can be removed before scoring. Splits are reindexed to the selected columns; the function returns reduced splits, F-score series, and lists of kept/removed features.

5.1.4.6.6 pca_reduction(...)

Performs PCA on numeric/bool columns after cleaning NaN/inf and imputing means from train. It reduces to n_components (either a fixed number or variance ratio) and returns new DataFrames with columns PC1, PC2, ... for train/val/test plus the fitted PCA object. Logs how many components were used.

5.1.4.7 Scaling & Normalization

5.1.4.7.1 zscore_scale_from_train(X_train, X_val=None, X_test=None)

Computes the mean and standard deviation of numeric/bool features on the training set and applies z-score scaling to train, val, and test. Non-numeric columns remain unchanged. Returns scaled splits and (mean, std) and logs that z-score scaling was applied.

5.1.4.7.2 robust_scale_from_train(X_train, X_val=None, X_test=None)

Computes median and IQR on training numerics and scales all splits using (x - median) / IQR. This is robust to outliers. Returns scaled splits and (median, IQR) for later reuse and logs the operation.

5.1.4.7.3 minmax_scale_from_train(X_train, X_val=None, X_test=None, feature_range=(0.0, 1.0))

Scales numeric/bool features to the specified range [a, b] based on min/max from the training data. All splits are transformed with the same parameters. Returns scaled splits and (min, max, feature_range) and logs the configured range.

5.1.4.7.4 l2_normalize_rows(X_train, X_val=None, X_test=None)

Applies L2 normalization to rows (samples) over numeric/bool columns using Normalizer(norm='l2'). The result is that each row has unit L2 norm in numeric feature space. Returns normalized splits and the fitted normalizer object.

5.1.4.8 Outlier Handling & Transformations

5.1.4.8.1 clip_outliers_iqr(X_train, X_val=None, X_test=None, whisker=3.0)

Computes IQR-based bounds for numeric features from the training data and clips values outside [Q1 - whisker*IQR, Q3 + whisker*IQR]. The same bounds are applied to val/test. Returns transformed splits and (lo, hi) bounds.

5.1.4.8.2 clip_by_percentile_from_train(X_train, X_val=None, X_test=None, lo=0.5, hi=99.5)

Clips numeric features using percentile-based lower and upper bounds (e.g. 0.5–99.5th percentile). Bounds are computed on train and reused for val/test. Returns splits and (lows, highs).

5.1.4.8.3 log1p_transform(X_train, X_val=None, X_test=None)

Selects non-negative numeric columns with strong right skew (skew > 1.0) and applies log1p to those columns on train, val, and test. Returns transformed splits and the list of columns that were log-transformed.

5.1.4.9 Class Imbalance Handling

5.1.4.9.1 compute_class_weights(y_train, scheme='balanced')

Computes class weights from the class frequencies in y_train.

- 'balanced': inverse-frequency weighting.
- 'uniform': all classes weight 1.

For binary tasks, it also computes a scale_pos_weight (neg/pos ratio). Returns the class weight dict and scale_pos_weight.

5.1.4.9.2 expand_sample_weights(y, class_weights)

Maps each label in y to its corresponding weight from class_weights and returns a per-sample weight array. Used to pass sample weights into model training.

5.1.4.9.3 balance_classes_smote(X_train, y_train, random_state=42, k_neighbors=5)

Applies SMOTE on numeric/bool features to synthesize minority-class samples and balance the training label distribution. Non-numeric columns are repeated to match the resampled size. Returns the resampled X and y.

5.1.4.9.4 balance_classes_random_over(X_train, y_train, random_state=42)

Performs random oversampling using RandomOverSampler. Returns resampled training features and labels and logs that random oversampling was applied.

5.1.4.9.5 balance_classes_random_under(X_train, y_train, random_state=42)

Performs random undersampling using RandomUnderSampler, reducing majority classes. Returns the downsampled training features and labels and logs the undersampling.

5.1.4.10 SVM Kernel Approximations

5.1.4.10.1 nystroem_rbf_from_train(...)

Builds approximate RBF kernel features on numeric/bool columns using the Nyström method.

- Cleans numeric data
- Derives gamma based on data variance if set to 'scale' or 'auto'
- Fits a Nystroem transformer on train and transforms train/val/test
- Returns new DataFrames containing features RBF_NYS_1 ... RBF_NYS_n (plus non-numerics concatenated back) and the fitted transformer.

5.1.4.10.2 rbf_sampler_from_train(...)

Similar to the Nyström variant, but uses Random Fourier Features (RBFSampler) to approximate the RBF kernel. Returns transformed DataFrames with features RBF_RFF_1 ... and the fitted sampler.

5.1.4.11 Time Features

5.1.4.11.1 add_time_features(X_train, X_val=None, X_test=None, tz=None)

Reads a timestamp column from st.session_state._timeStampCol and generates derived time features: hour, day-of-week (dow), month, year, and a binary is_weekend. Optionally converts to a specified timezone. Returns updated train/val/test with these additional columns.

5.1.4.12 End-to-End Preprocessing Pipeline

5.1.4.12.1 preprocessModel(X_train, y_train, X_val, y_val, X_test, y_test, config)

Executes the full preprocessing pipeline driven by the config dictionary (e.g. from modelPreset):

1. Optional label encoding for non-numeric labels.
2. Optional time-feature generation.
3. Cleaning: drop duplicates, impute missing values.
4. Categorical handling: bucket rare categories, GBM-style or generic encodings.
5. Additional encodings: count encoding, target mean encoding.
6. Feature pruning/selection: low variance, high correlation, RF importance, ANOVA F, mutual information, PCA.
7. Outlier handling and transformations: IQR/percentile clipping, log1p.
8. Scaling and normalization: z-score, robust, min-max, L2 normalization.
9. Kernel approximations: Nyström or RFF for SVM-like methods.
10. Class imbalance: class weights, sample weights, and optional resampling (SMOTE/Random over/under).
11. Final alignment: ensures validation and test have exactly the same feature columns as the processed training set.

It returns a dictionary with processed X_train, X_val, X_test, labels, class/sample weights, the used encoding type, and a metadata structure that logs all applied steps and key parameters (for reproducibility and reporting).

5.1.5 Trainer Evaluator

5.1.5.1 Summary

checkGBM_engine	Checks whether LightGBM, XGBoost and CatBoost are installed and returns availability flags
get_datasets_from_session	Loads train/val/test splits (and optional sample weights/metadata) from Streamlit session state for either raw or preprocessed flow
_make_ohe_sparse	Creates a OneHotEncoder with sparse output and optional max_categories, compatible across sklearn versions
_preclean_df	Pre-cleans a DataFrame by converting datetimes to seconds and downcasting float64/int64 to float32/int32
_safety_dense_ct	Builds a dense ColumnTransformer that imputes and scales numeric features and one-hot encodes categoricals
to_float32	Utility to cast a NumPy array to float32
InfToNaN	Transformer that replaces ±inf values with NaN in numeric arrays
minimal_raw_pipeline	Builds a minimal sklearn Pipeline (preclean + ColumnTransformer) for raw data depending on model type
fmt_bytes	Converts a size in bytes to a human-readable string (KB, MB, GB, …)
summary_size_bytes	Estimates serialized model size in bytes using pickle
_predict_proba_safe	Safely calls predict_proba if available, otherwise returns None
metrics_all	Computes accuracy, macro-F1, macro-precision, macro-recall, ROC-AUC (if possible), confusion matrix and classification report
_pref	Helper to build parameter names for pipeline vs non-pipeline estimators (e.g. clf__sample_weight)
fit_and_evaluate	Fits an estimator, measures training time, and computes metrics (and inference time) for train/val/test splits.
build_estimator	Creates a configured sklearn / GBM classifier instance based on the chosen model and GBM engine.
train_eval_orchestrator	High-level runner that builds estimator, attaches minimal preprocessing if needed, and calls fit_and_evaluate
plot_confusion	Creates a Plotly confusion-matrix heatmap with counts as annotations and optional normalization for colors.
plot_roc	Builds a Plotly ROC curve (binary or multi-class) using model predict_proba output.
plot_pr	Builds a Plotly Precision–Recall curve (binary or multi-class) using model predict_proba output.
feature_importance_fig	Plots a horizontal bar chart of top-k feature importances for models exposing feature_importances_.
_pp_delta	Computes the difference between PRE and RAW metrics in percentage points.
_pct_delta	Computes the relative percentage change between PRE and RAW values.
render_compare_general	Renders Streamlit metrics comparing RAW vs PRE in train time, model size and number of features.
render_compare_metrics	Renders a styled Streamlit table comparing RAW vs PRE KPIs (accuracy, F1, precision, recall, ROC-AUC, inference time) with deltas.

5.1.5.2 GBM Detection & Dataset Loading

5.1.5.2.1 checkGBM_engine()

Detects which gradient boosting libraries are available at runtime. It tries to import LightGBM, XGBoost, and CatBoost and returns a dict with boolean flags (_HAS_LGB, _HAS_XGB, _HAS_CAT). This allows the UI to enable/disable GBM engines dynamically and avoid runtime errors if a library is missing.

5.1.5.2.2 get_datasets_from_session(use_preprocessed: bool)

Retrieves the currently active dataset splits from st.session_state.

- If use_preprocessed=True, it returns the preprocessed train/val/test features and labels, corresponding sample weights, unseen-label information, and preprocessing metadata.
- If False, it returns the raw train/val/test splits without weights or metadata.

This function centralizes dataset access for the training view.

5.1.5.3 Version-safe Encoders & Precleaning

5.1.5.3.1 _make_ohe_sparse()

Constructs a OneHotEncoder with arguments that are compatible across different scikit-learn versions. It checks whether sparse_output and max_categories are supported and configures them accordingly. The encoder always uses handle_unknown="ignore" and produces a sparse output, optionally limiting category explosion.

5.1.5.3.2 _preclean_df(df: pd.DataFrame) -> pd.DataFrame

Pre-cleaner used prior to building raw-data pipelines:

- Converts datetime columns to seconds since epoch (float32).
- Downcasts float64 to float32 and int64 to int32 to reduce memory footprint.
- Returns a new DataFrame with the same columns but more compact dtypes.

If the input is not a DataFrame, it is passed through unchanged.

5.1.5.3.3 _safety_dense_ct(X: pd.DataFrame)

Builds a dense ColumnTransformer for generic preprocessing:

- Numeric/bool columns: median imputation + standard scaling.
- Categorical columns: most-frequent imputation + dense one-hot encoding.

It returns a ColumnTransformer configured with sparse_threshold=0.0, which guarantees dense arrays – useful for models (like some MLPs) that prefer dense input.

5.1.5.3.4 to_float32(A)

Simple helper used in pipelines to cast an array to float32 dtype, primarily to save memory and improve speed on some models.

5.1.5.3.5 class InfToNaN(BaseEstimator, TransformerMixin)

Custom transformer that replaces +inf and -inf values with NaN in numeric arrays. It converts input to a float NumPy array, cleans infinities, and returns the safe array. It plays well inside sklearn pipelines.

5.1.5.4 Minimal Raw-data Pipeline

5.1.5.4.1 minimal_raw_pipeline(model: str, X: Any) -> Optional[Pipeline]

Builds a minimal preprocessing pipeline when the user chooses to train on raw data (i.e. without using the full Preprocessor module):

- Uses a FunctionTransformer preclean step to apply _preclean_df while preserving the DataFrame structure.
- Separates numeric/bool and categorical columns.
- For Random Forest: imputes numerics (median), replaces inf with NaN, and casts to float32; imputes categoricals and applies sparse OHE.
- For LR/SVM/MLP: imputes numerics, standard-scales them; imputes categoricals and applies sparse OHE.
- If no matching model is found, it returns a pipeline with only the preclean step.

If the input is not a DataFrame, it returns None (no pipeline), so the estimator is used directly.

5.1.5.5 Utility Functions for Size & Probabilities

5.1.5.5.1 fmt_bytes(n: int) -> str

Converts a byte count into a human-readable string with units (B, KB, MB, GB, TB, PB). Negative or invalid inputs return "n/a".

5.1.5.5.2 summary_size_bytes(obj) -> int

Approximates the serialized size of an object (typically a trained model) by pickling it with the highest protocol and returning the length of the byte string. Returns -1 on failure.

5.1.5.5.3 _predict_proba_safe(est, X)

Wrapper around est.predict_proba(X) that returns None if the estimator does not support probability predictions or if the call fails. This allows metric computation to gracefully skip ROC/PR curves when not available.

5.1.5.6 Metrics & Evaluation

5.1.5.6.1 metrics_all(y_true, y_pred, y_prob=None) -> Dict[str, Any]

Computes a standardized metric bundle for classification:

- Accuracy
- Macro-F1
- Macro-precision
- Macro-recall
- ROC-AUC (binary or multi-class, if y_prob is provided)
- Confusion matrix (as nested lists)
- Full sklearn classification_report as a dict

For ROC-AUC, it supports both 1-D (binary) and 2-D (multi-class) probability outputs and uses ovr for multi-class if possible.

5.1.5.6.2 _pref(is_pipeline: bool, name: str) -> str

Helper function that resolves parameter names depending on whether the estimator is a Pipeline. For pipelines, it prefixes with "clf__" (e.g. "clf__sample_weight"); for bare estimators, it leaves the name unchanged.

5.1.5.6.3 fit_and_evaluate(...)

Centralized training and evaluation function. Responsibilities:

1. Validates the task using type_of_target.
2. Detects whether the estimator is a Pipeline and configures fit_params accordingly.
3. Passes sample weights (if provided) to sample_weight at the correct level.
4. For CatBoost, optionally injects cat_features indices based on metadata or dtypes.
5. Measures training time with time.perf_counter().
6. Evaluates model on train/val/test via an inner eval_split:

- Predicts labels
- Measures inference time
- Optionally computes probabilities via _predict_proba_safe
- Calls metrics_all for each split

It returns a dict containing the trained model, metrics for each split, training time, model size in bytes, and the number of features used.

5.1.5.7 Estimator Factory & Orchestrator

5.1.5.7.1 build_estimator(model: str, gbm_engine: Optional[str])

Factory that builds the right base classifier:

- "Random Forest (RF)" → tuned RandomForestClassifier (max_depth, min_samples_leaf, max_features, max_samples).
- "Logistic Regression (LR)" → LogisticRegression with high max_iter and multi-core LBFGS.
- "Support Vector Machine (SVM)" → RBF-kernel SVC with probability=True.
- "Neural Network (MLP)" → MLP with two hidden layers (128, 64).
- "Gradient Boosting (...)" → chooses LGBM, XGBClassifier, or CatBoost depending on gbm_engine and availability flags.
- Fallback: a generic RandomForest if nothing matches.
-

5.1.5.7.2 train_eval_orchestrator(...) -> Dict[str, Any]

High-level runner that connects dataset, preprocessing, and estimator:

1. Builds a base estimator via build_estimator.
2. For preprocessed data and MLP, wraps the estimator into a dense safe pipeline if DataFrame still contains object/category columns.
3. For raw data (use_preprocessed=False), builds a minimal_raw_pipeline and wraps the estimator inside a Pipeline with "prep" + "clf".
4. Calls fit_and_evaluate with all splits, optional sample weights, selected GBM engine, and metadata.

The returned dict contains the trained model, metrics, training time, model size, feature count, and per-split inference times.

5.1.5.8 Visualizations (Confusion, ROC, PR, Feature Importance)

5.1.5.8.1 plot_confusion(cm: np.ndarray, class_names=None, normalize="true", decimals=2)

Builds a Plotly confusion-matrix heatmap:

- normalize controls how the color intensity is computed ("true" = per row, "pred" = per column, "all" = global, or None for raw counts).
- Colors encode normalized values, while annotations show integer counts.
- Hover text shows actual label, predicted label, and count.
- Layout is streamlined for embedding in Streamlit (no borders, fixed height).
-

5.1.5.8.2 plot_roc(model, X, y, class_names=None)

Plots ROC curves using predict_proba:

- Binary case: plots a single ROC curve and its AUC.
- Multi-class: plots one ROC curve per class in a one-vs-rest fashion.
- Adds a diagonal “chance” line for reference.
- Uses provided class_names or falls back to model.classes_.

Returns a Plotly Figure or None if probabilities/classes are not available.

5.1.5.8.3 plot_pr(model, X, y, class_names=None)

Plots Precision–Recall curves:

- Binary case: precision–recall curve for the positive class with AP (average precision).
- Multi-class: one curve per class with its AP.
- Uses predict_proba output and precision_recall_curve/average_precision_score.
- Layout is minimal (no title, compact margins) for use in Streamlit.
-

5.1.5.8.4 feature_importance_fig(model, feature_names=None, top_k=25)

Visualizes feature importances for tree-based models exposing feature_importances_:

- If the model is a Pipeline, it inspects the final clf step.
- Extracts importance scores, selects the top-k indices, and maps them to feature_names if provided.
- Renders a horizontal bar plot (most important at the top) using Plotly.

Returns None if the model has no feature_importances_ attribute.

5.1.5.9 RAW vs PRE Comparison Helpers

5.1.5.9.1 _pp_delta(pre, raw)

Computes the difference pre - raw in percentage points (assuming metrics are in [0,1]) and scales to [–100, 100]. Returns None if inputs are invalid.

5.1.5.9.2 _pct_delta(pre, raw)

Computes relative percentage change (pre - raw) / raw * 100 (e.g., –65% training time). Returns None if raw is zero or inputs are invalid.

5.1.5.9.3 render_compare_general(results_NP, results_P)

Streamlit UI helper that compares general efficiency KPIs between RAW and PRE runs:

- Training time (s)
- Model size (bytes, formatted via fmt_bytes)
- Number of features

For each metric, it displays RAW vs PRE side by side and shows percentage deltas where lower is better (inverse coloring for training time and size).

5.1.5.9.4 render_compare_metrics(results_P, results_NP)

Renders a detailed metric comparison table between RAW and PRE using Streamlit:

- Builds a long-form table with metrics for train/validation/test and tags RAW/PRE.
- Reshapes into a MultiIndex-wide table with columns for RAW, PRE, and deltas.
- Metrics include: Accuracy, F1 (macro), Precision (macro), Recall (macro), ROC-AUC, and inference time per split.
- Uses pandas styling to format values (percentages, seconds, percentage points) and to color improvements (green) / degradations (red) for each delta.

The result is a compact, at-a-glance impact analysis inside the UI.

5.2 Pages

5.2.1 Starting page

5.2.1.1 Purpose

A_R_C.py is the main entry point of the Streamlit application. It bootstraps the global session state, configures the UI layout, attaches the sidebar navigation, and presents the high-level introduction and user guide for the A.R.C. framework. From this page, users learn what A.R.C. does, who it is for, and how the end-to-end pipeline is structured before they move on to the individual modules (Data Loader, Schema Validator, etc.).

5.2.1.2 Initialization and session state

At the top, the script imports:

- streamlit as st for the UI.
- functions.f00_VarInitialisation as vi for initializing all st.session_state variables.
- functions.f00_Sidebar as sidebar for rendering the global sidebar navigation.

On the very first visit, it sets up a “first start” guard in st.session_state:

- If _First_Start is not present, it is created and set to True.
- If _First_Start is True, the script calls vi.init() to initialize all required session keys (flags for loaded data, split status, preprocessing options, model results, etc.). It also sets _First to False.

This mechanism guarantees that all subsequent pages can safely assume the presence of well-defined session variables without having to re-initialize them themselves. It also ensures that initialization happens only once per session.

5.2.1.3 Page configuration and layout

The script configures basic page properties via st.set_page_config:

- page_title="A.R.C." sets the browser tab title.
- layout="wide" uses Streamlit’s wide layout, giving more horizontal space for tables, plots, and multi-column layouts used in the rest of the app.

Immediately after that, the script sets the main page title
It then calls sidebar.sidebar(), which renders the persistent navigation panel (e.g., links to Data Loader, Schema Validator, Label Validator, Splitter, Preprocessor, Trainer & Evaluator, Comparer, Reporter, Logger, etc.). This makes the start page consistent with all other pages and gives users quick access to the pipeline steps.

5.2.1.4 Landing page content

The remainder of A_R_C.py is a large Markdown block that serves as the interactive landing page documentation visible to the user. It explains the purpose of A.R.C., the available functionality, target audience, and the recommended workflow across the modules.
The content is structured into several sections:
1. Welcome and high-level description
The first paragraph introduces A.R.C. as:

- A modular pipeline for preparing and assessing cybersecurity datasets.
- Capable of handling raw data ingestion, schema and label validation, splitting, preprocessing, training, evaluation, and report generation.
- Focused on reproducibility and transparency, enabling users to see and control each step of the data preparation and model training process.

This gives new users a clear mental model: A.R.C. is an end-to-end framework rather than a single monolithic training script.

2. “What you can do here”
This section enumerates the main capabilities of the app in bullet points, including:

- Data ingestion
    - Upload datasets in CSV/TXT (and other supported formats) or parse PCAP/PCAPNG into structured flows/sessions.
- Schema and label validation
    - Inspect column types, detect duplicates, check timestamp integrity.
    - Validate label consistency, class balance, and temporal drift.
- Splitting strategies
    - Create train/validation/test splits using Random, Stratified, or Time-based methods, with safeguards to avoid data leakage.
- Preprocessing
    - Configure cleaning, encoding, feature selection, scaling, outlier handling, class imbalance strategies, and time-feature extraction via explicit toggles.
- Training and evaluation
    - Train baseline models such as Random Forest, Gradient Boosting (XGBoost/LightGBM/CatBoost), SVM, MLP, and Logistic Regression.
    - Inspect standard metrics (Accuracy, macro-F1, macro-Precision/Recall, ROC-AUC) as well as efficiency metrics (training time, inference time, model size, feature count).
- Comparisons
    - Compare a minimal raw pipeline to a fully preprocessed pipeline to measure the real impact of preprocessing on performance and efficiency.
- Export and reporting
    - Download cleaned datasets, splits, JSON configuration and metadata, trained models (PKL), and a consolidated PDF report that aggregates tables, plots, and findings.

This section effectively describes the “feature set” of the entire A.R.C. UI.
3. “Who is this for?”
The next section states the target audience:

- Researchers and engineers who need trustworthy, auditable, and comparable machine learning experiments—especially in cybersecurity contexts.
- Users who face typical dataset issues such as inconsistent features, noisy labels, class imbalance, and time leakage in public datasets and want a systematic tool to handle them.

This clarifies that A.R.C. is designed not only for casual exploration but for serious, reproducible experimentation and publication-grade workflows.
4. “The A.R.C. flow at a glance”
The final section outlines the recommended pipeline steps, each linked conceptually to a separate Streamlit page:

1. Data Loader: Upload data (CSV/TXT/JSON or PCAP/PCAPNG), optionally parse network traffic into flows, and select label and timestamp columns. Users inspect a preview and basic summary of the dataset.
2. Schema Validator: Analyze the dataset structure: duplicates, missing values, mixed types, feature counts, and temporal properties (start/end, gaps, sort order). The page also offers safe schema manipulations (normalize column names, rename, sort by timestamp, drop duplicates). All changes and findings are logged.
3. Label Validator: Inspect label quality: missing labels, inconsistent spellings, class distribution, label entropy, rare vs. dominant classes, and temporal drift. Users can rename/merge labels before training to obtain stable taxonomies.

4. Splitter: Configure and execute dataset splits using:
    a. Random splits for quick baselines.
    b. Stratified splits for imbalanced datasets.
    c. Time-based splits for time-series and log data, ensuring the model only sees past data.

   Quality checks verify unseen labels and class distribution across splits. Random states and split sizes are part of the logged configuration for reproducibility.
5. Preprocessing: Set model-aware preprocessing options via toggles:
    a. Cleaning (drop duplicates, impute missing values).
    b. Encoding (One-Hot, ordinal, GBM-native, count encoding, target mean encoding).
    c. Feature filtering/selection (variance threshold, correlation pruning, RF importance, mutual information, ANOVA, PCA).
    d. Scaling and normalization.
    e. Outlier clipping and transformations (log1p).
    f. Class imbalance handling (class weights, SMOTE/over/under sampling).
    g. Time feature extraction from timestamps.

   All choices are captured in a metadata structure stored in session state and later exportable as JSON.
6. Training & Evaluation: Train selected baseline models on the preprocessed splits and compute:
    a. Core metrics: Accuracy, macro-F1, macro-Precision, macro-Recall, ROC-AUC.
    b. Efficiency measures: training time, inference time, model size, number of features.

   Graphical outputs include confusion matrices, ROC and Precision-Recall curves, and a ranked Top-25 feature importance chart.
7. Compare (Raw vs. Preprocessed)
    a. Run the same model on a minimal “raw” pipeline and on the full preprocessed pipeline, then compare:
    b. Predictive metrics (e.g., macro-F1).
    c. Efficiency metrics (train/inference time, model size, feature count).

   This quantifies the benefit of preprocessing in a transparent way.
8. Reporting & Export: Use the Reporter page to assemble everything into a reproducible bundle:
    a. CSVs for validated and preprocessed datasets and splits.
    b. JSON for preprocessing metadata and metrics.
    c. PKL for the final trained model.
    d. A multi-section PDF report including figures, tables, and configuration summaries from all selected modules.

   This “flow at a glance” section acts as an in-app user guide: it explains how the individual pages are meant to be used together, and what the recommended order of operations is.

5.2.2 Logger

5.2.2.1 Purpose

The Logger page provides a centralized interface for reviewing, adding, and exporting pipeline log messages. Throughout the system, many operations automatically create log entries (for example, dataset loading, schema changes, preprocessing steps). The Logger serves as the primary tool for monitoring these events, manually appending custom notes, and exporting the full log for later documentation or audit purposes. This page is particularly important for reproducibility, traceability, and debugging.

5.2.2.2 Initialization

The page begins by initializing the sidebar and setting the page title to "Logger." It loads all existing log entries from st.session_state._LogData, which stores each log entry as a tuple consisting of a timestamp and a message.

5.2.2.3 Adding Custom Log Messages

A form allows users to manually append new log messages. The form contains:

- A text area for entering a message.
- A submit button that appends the message to the log.

If the message is non-empty, it is added to the logger and the page reruns to display the updated log entries. This is useful for adding comments, documenting decisions, or annotating pipeline stages.

5.2.2.4 Clearing the Log

A button labeled "Clear log" removes all entries from the session log. After clearing, the page reruns so the empty log is immediately reflected in the display. This action is helpful when starting a new workflow or when prior logs are no longer relevant.

5.2.2.5 Downloading the Log

Below the message and clearing controls, the Logger provides a download button that allows the full log to be saved as a .txt file. The downloaded file includes:

- A timestamp for each entry
- The associated log message

Entries are separated by blank lines for readability. The download is disabled when the log is empty.

5.2.2.6 Viewing Log Entries

The page ends with a section titled "Log entries," which displays all collected log messages in reverse chronological order (latest first).

If no entries are present, the page shows an informational message.

This final view allows users to inspect the full system history and verify that all operations were executed as intended.

5.2.3 Data Loader

5.2.3.1 Purpose

The Data Loader page provides the entry point for importing datasets into the pipeline. It supports both single-file loading (including PCAP, PCAPNG, and CSV formats) and multi-file loading for combining multiple CSVs into a unified dataset. It also automatically detects timestamps, identifies label columns, and provides metadata about the loaded dataset.

This page is the first step of the pipeline because all subsequent components depend on a successfully loaded dataset.

5.2.3.2 Initialization

The sidebar is created and the title "Data Loader" is displayed. Two buttons allow the user to choose the upload mode:

1. Upload single file
2. Upload and connect multiple files

Clicking these buttons sets the session state _DL_Mode, which determines the active workflow.

5.2.3.3 Single File Upload Workflow

When the "single" upload mode is active, the user is prompted to upload a file. The page supports the following formats:

- .pcap
- .pcapng
- .csv
- .txt

After uploading, the file extension determines which configuration options appear.

PCAP/PCAPNG Files

If a packet capture is uploaded, the interface exposes extensive extraction options across network layers:

- Core metadata (timestamp, datetime, length)
- Layer 2 fields (Ethernet, VLAN, ARP)
- Layer 3 fields (IPv4, IPv6)
- Layer 4 fields (TCP, UDP, ICMP, ICMPv6)
- Layer 7 fields (DNS, HTTP, TLS, DHCP, NTP)
- Payload extraction options (payload length, preview of raw bytes)

These settings allow users to select exactly which protocol fields should be parsed into the dataset.

Non-PCAP Files

If the file is a CSV or TXT, all PCAP-related configuration fields are skipped automatically.

5.2.3.4 Loading the Dataset

Upon clicking "Load dataset," the system calls the DataLoader function load() with all chosen options. If successful:

- The loaded DataFrame is stored in session state (_DF).
- The filename is recorded.
- A success log entry is created.
- The system checks for timestamp-like columns and sets _HasTimeStamp and _TimeStampCol.
- The page reruns to show the preview.

5.2.3.5 Multiple File Upload Workflow

If the user selects “Upload and connect multiple files,” the page provides a multi-file uploader that accepts multiple CSVs.
Users can specify:

- Combined dataset name
- Whether to add the source filename as a new column
- Whether the files contain a header
- File encoding
- Delimiter (auto-detected or user-specified)
- Column handling strategy:
    - Union (retain all columns across files)
    - Intersection (retain only shared columns)
- Whether to drop duplicates after combining

After selecting files and clicking “Load dataset(s),” the DataLoader merges the CSV files and reports any schema inconsistencies or load errors.
If successful:

- The combined DataFrame is saved into session state.
- The filenames list is stored for later display.
- A log entry records the action.
- Timestamp detection is performed.

5.2.3.6 Dataset Preview and Metrics

Once a dataset is loaded, the page shows:

- A preview of the first five rows
- Basic dataset metrics:
    - Number of rows
    - Number of columns
    - Memory consumption

If multiple files were uploaded, an expandable section lists the source filenames.

5.2.3.7 Timestamp and Label Column Selection

Two parallel sections allow users to specify:
Timestamp Column

- Whether the dataset contains timestamps
- A list of detected timestamp-like columns
- A dropdown for selecting the correct timestamp column

Label Column

- Whether the dataset contains labels
- A dropdown listing all columns
- Selection of the label column for downstream modules

These selections are saved into session state for later use in schema validation, splitting, and preprocessing.

5.2.3.8 Downloading the Loaded Dataset

The page ends with a “Downloads” section that allows exporting the dataset as a CSV file using the current scrollable data as displayed.

5.2.4 Schema Validator

5.2.4.1 Purpose

The Schema Validator page provides an in-depth audit of the dataset structure before any preprocessing or modeling is performed. It evaluates the dataset's rows, columns, missingness, granularity, data types, timestamp behavior, and schema consistency. It also enables a range of schema modification tools including renaming, normalizing, sorting, and removing columns.
This page ensures that the dataset is structurally valid and consistent so that downstream modules (label validator, preprocessor, model trainer) operate on a clean and well-defined schema.

5.2.4.2 Initialization

The sidebar is displayed first, followed by the page title "Schema Validator." If no dataset has been loaded, the page shows an error and exits.
If a dataset is present, the top of the page displays a preview of the loaded DataFrame and proceeds with schema validation.

5.2.4.3 Schema Insights

The validator obtains a complete report from schemaValidatorTotal(). The page then presents several subsections.

5.2.4.4 Total Records and Duplicates

The page shows the total number of records. If duplicate rows are found, it displays a warning including:

- Number of duplicates
- Percentage relative to the dataset

Otherwise, it confirms that all records are unique.

5.2.4.5 Sparsity (Missing Values)

A collapsible section summarizes all missing values, including:

- Total missing cell count
- Overall percentage
- List of columns with missing values (sorted by severity)

If no missing values are found, the page displays a success message.

5.2.4.6 Granularity Detection

The system automatically classifies the dataset as:

- Flow-level
- Packet-level
- Session-level
- Unknown / Custom

This detection helps indicate whether a dataset originates from flow extraction, packet captures, or session summaries.

5.2.4.7 Feature Insights

The page reports:

- Total number of features
- Whether all feature names are unique
- Whether any features contain mixed data types
- Full breakdown of feature categories: numerical, categorical, boolean, object, and datetime features

The distribution of feature types is visualized in:

- A pie chart
- A detailed expandable list view

This provides users with a clear understanding of the dataset's structure.

5.2.4.8 Time Span Analysis

If a timestamp column is present, the page provides:

- Start and end time
- Total duration
- Most common time interval between rows
- Average interval

It also warns about:

- Duplicate timestamps
- Whether timestamps are sorted

If no timestamp is selected, the page reports that time analysis is unavailable.

5.2.4.9 Column Presence Check

Users may provide a list of required column names (comma-separated). The validator checks:

- Which required columns are missing
- Which extra columns are present that were not listed

Each category is listed in separate expandable containers.

5.2.4.10 Schema Modification Tools

The second major part of the page provides tools for modifying the dataset directly. Each tool is displayed with four aligned columns:

- The action button
- Input variables (if any)
- Description
- Purpose

This makes the interface self-explanatory and helps maintain documentation-style clarity.
The available tools are:

1. Normalize feature names
   a. Converts all feature names to lowercase without spaces.
   b. Purpose: eliminate name inconsistencies and simplify downstream processing.
2. Rename features
   a. Allows users to define rename rules using a simple old = new format.
   b. Purpose: harmonize schema across datasets, fix inconsistencies, match expected formats.
3. Sort by timestamp
   a. Sorts rows chronologically and optionally adds a delta_seconds column.
   b. Purpose: ensure temporal order and prepare for time-aware splitting and drift analysis.
4. Drop feature
   a. Removes a selected column from the dataset.
   b. Purpose: eliminate irrelevant, harmful, or redundant features.
5. Drop duplicates
   a. Removes all duplicate rows.
   b. Purpose: avoid double-counting and data leakage in modeling.

Every modification logs the action for full reproducibility.

5.2.4.11 Download Section

A final section allows the user to download the updated dataset as a CSV file reflecting all changes made through schema modification tools.

5.2.5 Label Validator

5.2.5.1 Purpose

The Label Validator page focuses on validating and analyzing the dataset's target variable (label). It helps ensure that label values are complete, consistent, meaningful, and stable over time. The page:

- Checks for missing labels and inconsistent spelling.
- Computes and interprets label entropy (a measure of balance).
- Visualizes class distribution using pie charts, bar charts, and tables.
- Identifies rare and dominant classes based on user-selected thresholds.
- Analyzes temporal drift if a timestamp column is available.
- Provides a tool for renaming or standardizing label values.
- Allows downloading the updated dataset.

This page is typically used after the Schema Validator to ensure the label column is ready for splitting, preprocessing, and modeling.

5.2.5.2 Initialization and Preconditions

The page initializes the sidebar and sets the title. It then checks two requirements:

1. A dataset must have been loaded.
2. A label column must be selected.

If either is missing, the page displays an error message and stops.
If both are satisfied, it shows a preview of the dataset and loads aggregate label statistics via the function labelValidatorTotal, which returns information about missing labels, inconsistent labels, entropy, class counts, and temporal distributions.

5.2.5.3 Label Consistency Checks

The section "Labeling consistency" reports two critical diagnostics:

1. Missing labels
    a. If the dataset contains missing or null label values, a warning is shown. Otherwise, the page reports that no missing labels are present.
2. Spelling inconsistencies
    a. The page detects cases where multiple labels differ only in capitalization or typos (for example, "DoS", "dos", "DOS"). If such groups exist, they are shown as a warning. If not, the label spelling is considered consistent.

These diagnostics help users detect issues that could impact class balance, model training, or reproducibility.

5.2.5.4 Label Entropy

The page computes the label entropy in bits, which summarizes how evenly distributed the labels are. It also provides a natural-language interpretation, such as:

- Perfectly balanced
- Mostly balanced
- Moderate imbalance
- High imbalance
- Only one label present

Entropy helps understand dataset diversity and imbalance at a glance.

5.2.5.5 Class Distribution

The class distribution section visualizes how often each label occurs. The UI provides three tabs:

1. Pie chart – visually highlights dominant and minor classes.
2. Table – shows numerical counts per class.
3. Bar chart – offers a clear comparison between class frequencies.

These views help users detect imbalance and better understand label distribution before modeling.

5.2.5.6 Rare Classes

Users can specify a threshold (between 0% and 15%) to define what constitutes a "rare" class.
The tool calculates the relative frequency of each class and identifies all classes that fall below the threshold.

- If no such classes exist, the page reports that none are below the threshold.
- If there are rare classes, a warning is shown listing them with their relative percentages.

Rare classes are important to detect because they often require oversampling, merging, special handling, or exclusion depending on the experimental design.

5.2.5.7 Dominant Classes

Similarly, users can specify a threshold (between 70% and 100%) to define a "dominant" class.
A dominant class is one that occupies a disproportionately large share of the dataset.

- If no classes exceed the threshold, the page reports this.
- If one or more are dominant, a warning lists them along with their proportions.

This detection helps identify severe class imbalance, which may degrade model performance and lead to misleading metrics.

5.2.5.8 Temporal Drift Analysis

If the dataset includes a timestamp column, the page unlocks a temporal drift section. Users can choose among three types of drift visualization:

1. Timeline view: Shows how the label changes over time as continuous segments. This helps reveal label bursts (for example, periods of intense attack activity).
2. Time-bin view: Divides the time range into equal-duration bins and shows class distribution within each bin. Useful for detecting changes in attack frequency, distribution shifts, or nonstationarity.
3. Record-bin view: Splits the dataset into bins with equal numbers of records rather than equal time duration. Useful when timestamps are irregular or unreliable.

If binning is enabled, the user can select the number of bins (from 1 to 100).
Each visualization includes both a chart and a formatted table.
This makes it easier to diagnose temporal drift — one of the most important challenges in cybersecurity and time-series datasets.

5.2.5.9 Label Modification (Renaming)

The “Label modification” section allows users to rename label values using a user-defined mapping.
The mapping is entered as lines of the form:
old_label = new_label
Examples:
DoS attack = DoS
scan = scanning
normal = benign
The entered mapping is parsed and previewed for correctness. Users can correct or expand the mapping before applying it.
Once the user clicks “Rename label”:

- The dataset’s label column is updated in place.
- A log entry is created for reproducibility.
- The page refreshes so that all plots and statistics reflect the updated labels.

The purpose of renaming is to:

- Fix typos
- Merge equivalent classes
- Standardize class names for modeling and comparison
- Ensure consistency in reports and evaluation metrics
-

5.2.5.10 Dataset Download

At the bottom of the page, users can download the dataset — now modified with cleaned labels — as a CSV file.
This allows exporting corrected or standardized versions of the dataset for use outside the system or for archival.

5.2.6 Splitter

5.2.6.1 Purpose

The Splitter page provides a structured interface for dividing the dataset into training, validation, and test subsets. Correct dataset splitting is essential for creating reproducible and unbiased machine-learning experiments. The Splitter module supports three complementary split strategies, Random, Stratified, and Time-based splitting, each designed for different use cases such as class-imbalanced datasets, time-dependent datasets, or quick baseline testing.
This page also includes a detailed quality-check section that evaluates the resulting splits, reports class distribution consistency, and detects unseen labels to ensure the reliability of downstream training.

5.2.6.2 Initialization

The page begins by creating the sidebar and setting the page title to "Splitter."
If no dataset has been loaded, an error message instructs the user to load data first.
If the dataset is loaded but no label column was selected, the page informs the user that splitting is not possible without a label.
Once both conditions are met, the full split configuration interface becomes available.

5.2.6.3 Split Method Explanation

Before choosing a split strategy, the page provides a clear side-by-side overview of:

- Split method name
- Advantages
- Use cases

This comparison helps users select the method appropriate for their data characteristics:

- Random Split
    - Advantages: Simple, fast, and evenly distributes instances when the dataset is balanced.
    - Use Case: Baseline experiments where time is irrelevant and classes are roughly balanced.
- Stratified Split
    - Advantages: Preserves class proportions across all splits.
    - Use Case: Imbalanced datasets or classification tasks where representation fairness is required.
- Time-based Split
    - Advantages: Enforces temporal integrity and avoids leakage from future data.
    - Use Case: Time series, log data, or cybersecurity event streams where time order must be preserved.

Each method is separated visually by horizontal rules for clarity.

5.2.6.4 Choosing the Split Method

The user selects the split strategy from a dropdown menu.
The system remembers the last selected method via session state, ensuring continuity across page refreshes.
Depending on the selected method, the interface dynamically displays relevant inputs:

- Random / Stratified Split Options: Random state (seed) for reproducibility
- Time-based Split Options: Gap ratio (percentage of samples removed between training and validation to avoid temporal leakage)
-

5.2.6.5 Configuring Split Sizes

The user adjusts split sizes using sliders:

- Test set size (%): typically between 10% and 40%
- Validation set size (%): typically between 10% and 40%

Both values are saved to session state.
Internally, the percentages are converted to fractions used by the splitting functions.

5.2.6.6 Performing the Split

When the user clicks “Split Dataset”, the page calls the split_dataset() function from the Backend Splitter module with:

- Selected method (random, stratified, time-based)
- Dataset (_DF)
- Label column
- Timestamp column (if applicable)
- Test/validation percentages
- Gap ratio (for time-based)
- Random seed (for random/stratified)

The split outputs:

- X_train, y_train
- X_validate, y_validate
- X_test, y_test

All results are stored in session state and the page reruns to display the quality checks.

5.2.6.7 Quality Checks After Splitting

If splitting was successful, the page displays a “Split” and a “Quality checks” section.

5.2.6.7.1 Split Size Summary

The page reports:

- Number of samples in each split
- Percentage corresponding to each split

This allows users to verify that:

- The split sizes match the intended proportions
- The dataset size is sufficient for training, validation, and testing

5.2.6.7.2 Unseen Label Detection

The system checks whether the validation or test set contains labels that do not appear in the training set.
This is crucial because unseen labels indicate:

- Potential leakage
- Stratification failure
- Temporal anomalies (for time-based splits)
- Uneven distribution of rare classes

The page displays warnings listing any unseen labels in validation or test sets.

5.2.6.7.3 Label Distribution Analysis

To ensure the split is fair and balanced, three distribution analyses are shown:

1. Distribution by split set
2. Distribution by label
3. Distribution normalized by total dataset frequency

Each distribution has both:

- A stacked bar chart (visual)
- A table view (numerical)

This helps users verify that split proportions are consistent and that all labels are adequately represented.

5.2.6.7.4 Distribution by Split

Shows each label's share within Training, Validation, and Test.

5.2.6.7.5 Distribution by Label

Shows how each label is distributed across the three splits proportionally.

5.2.6.7.6 Distribution Total

Shows each label's share of the entire dataset, with contributions from all three splits.
These checks help detect:

- Skewed splits
- Rare class isolation
- Imbalanced validation/test sets
- Incorrect gap configuration
-

5.2.6.8 Download Section

After split evaluation, the "Download Datasets" section provides a complete export interface for:

- X_train, y_train
- X_validate, y_validate
- X_test, y_test

Each dataset can be downloaded as a CSV file with a single click.
This is particularly useful for:

- Offline experimentation
- Sharing datasets across environments
- Saving pre-split versions in the project directory
- Logging reproducible pipeline steps

The data is encoded as UTF-8 CSV files suitable for pandas, Excel, Power BI, and external ML platforms.

5.2.7 Preprocessor

5.2.7.1 Purpose

The Preprocessor page configures and applies standardized preprocessing pipelines to the already split dataset. It connects high-level modeling choices (e.g., Random Forest vs. SVM) with a granular set of preprocessing operations such as cleaning, encoding, feature selection, scaling, outlier handling, class imbalance treatment, kernel approximations, and time-based features. The configuration is stored in Streamlit's session state and passed to a backend preprocessing function, which produces preprocessed training, validation, and test sets alongside metadata and class weights.

The page is designed so that non-expert users can start from recommended presets per model, while advanced users retain full control over each processing step.

5.2.7.2 Model Insights and Algorithm Selection

At the top, the page displays a section titled "Model insights." Here, expandable panels explain the trade-offs of each supported algorithm:

- Random Forest (RF)
- Gradient Boosting (XGBoost / LightGBM / CatBoost)
- Support Vector Machine (SVM)
- Neural Network (MLP)
- Logistic Regression (LR)

Each expander summarizes:

- Advantages / strengths
- Weaknesses / limitations
- Performance on imbalanced data
- Computational cost

This textual overview acts as a quick decision aid when choosing a model for a particular dataset (for example, RF as a robust default, Gradient Boosting for higher accuracy, SVM for high-dimensional small datasets, or LR as a fast and interpretable baseline).

Below the insights, an algorithm selection dropdown allows the user to choose one of the following:

- None
- Random Forest (RF)
- Gradient Boosting (XGBoost / LightGBM / CatBoost)
- Support Vector Machine (SVM)
- Neural Network (MLP)
- Logistic Regression (LR)

The current selection is stored in st.session_state._PP_Model.

A button "Apply suggested preset for " triggers the backend helper pp.modelPreset(), which returns a dictionary of recommended preprocessing settings for the chosen algorithm (e.g., imputation strategy, scaling choice, encoding options). These key–value pairs are then written back to the session state as _PP_<key>. This mechanism provides model-aware default configurations, ensuring that, for example, SVM is combined with scaling, tree-based methods with appropriate encoding, and so on.

5.2.7.3 Preprocessing Steps Overview

The “Preprocessing steps” section is divided into multiple expandable blocks, each corresponding to a logical group of operations:

- Cleaning
- Encoding
- Feature Filtering / Selection
- Scaling & Normalization
- Outliers & Transforms
- Class Imbalance
- SVM Kernel Approximations
- Time Features

Each expander follows a consistent structure:

- Left column: name of the preprocessing step
- Second column: toggle(s) and parameters
- Third column: short description
- Fourth column: purpose and rationale

This layout ensures that users immediately understand what each option does and why it matters.

5.2.7.4 Cleaning

The Cleaning expander contains basic data hygiene operations:

- Drop duplicates
  - Toggle: “Drop duplicates”
  - Description: Removes duplicate rows from the dataset.
  - Purpose: Prevents over-counting identical examples, which could bias learning—especially on small or imbalanced datasets.
- Impute missing values
  - Toggle: “Impute missing values”
  - Numeric setting: choice between “Median” and “Mode” for numerical features.
  - Categorical setting: “Most Frequent” for categorical features.
  - Description: Fills in missing entries for both numeric and categorical columns using robust defaults.
  - Purpose: Avoids dropping data and prevents models from failing on NaN values, preserving sample size and improving stability.

All choices are saved under _PP_MV, _PP_MV_N, and _PP_MV_C.

5.2.7.5 Encoding

The Encoding expander configures how categorical variables are transformed into numerical features:

- Bucket rare categories
  - Toggle: “Bucket rare categories”
  - Parameter: min_freq defines the minimal number of occurrences for a category to be kept; otherwise it is folded into an “Other” bucket.
  - Description: Combines infrequent categories into a single bucket.
  - Purpose: Reduces risk of overfitting to rare categories and stabilizes model training.
- Generic / Advanced Encoding
  - Toggle: “Generic/Advanced Encoding”
  - Mode: “One Hot Encoding” or “Ordinary” (ordinal encoding).
  - Description: Encodes categorical features either as one-hot vectors or as integer codes.
  - Purpose: Ensures compatibility with algorithms that require numeric inputs. One-hot is more expressive for unordered categories; ordinal is more compact but implies an ordering.
- GBM Native Categorical Prep
  - Toggle: “GBM Native Categorical Prep”
  - Mode: “CatBoost”, “Light GBM”, or “XG Boost”.
  - Description: Delegates categorical handling to the underlying gradient boosting library.
  - Purpose: Leverages each library’s native, optimized treatment of categorical data.
- Count Encoding
  - Toggle: “Count Encoding”
  - Description: Replaces categories with their frequency counts.
  - Purpose: Adds statistical signal for tree-based models and captures the prevalence of a category.
- Target Mean Encoding
  - Toggle: “Target mean encoding”
  - Parameters:
    - n_splits: number of cross-validation partitions for safer target encoding
    - noise_std: standard deviation for injected noise
    - global_smoothing: controls regularization between global mean and category mean
  - Description: Encodes categories by the average target value per category, with noise and smoothing options to reduce leakage.
  - Purpose: Provides a powerful encoding for high-cardinality features while mitigating overfitting when used carefully.

All encoding settings are collected under keys like _PP_BR, _PP_EN_GA, _PP_GBM, _PP_CE, _PP_TME, and their respective parameter fields.

5.2.7.6 Feature Filtering / Selection

The Feature Filtering / Selection expander defines how to reduce dimensionality and remove weak or redundant features:

- Drop constant / low variance features
    - Toggle: “Drop constant / low variance”
    - Parameter: variance threshold specifying the minimum variance allowed.
    - Description: Removes features with little or no variability.
    - Purpose: Eliminates useless predictors, reduces noise, and can speed up training.
- Drop highly correlated numerics
    - Toggle: “Drop highly correlated numerics”
    - Parameter: corr threshold (e.g., 0.95).
    - Description: Drops numeric features that are strongly correlated with each other.
    - Purpose: Addresses multicollinearity, reduces redundancy, and simplifies interpretation.
- Feature selection by Random Forest importance
    - Toggle: “Feature selection by RF importance”
    - Parameter: keep top-N most important features.
    - Description: Uses a Random Forest to rank features by importance and keeps only the top-N.
    - Purpose: Focuses the model on the most predictive features and reduces dimensionality.
- Feature selection by Mutual Information
    - Toggle: “Feature selection by Mutual Info”
    - Parameter: keep top-k features.
    - Description: Selects features based on mutual information with the target (captures linear and non-linear dependencies).
    - Purpose: Identifies features that share the most information with the label.
- Feature selection by ANOVA F
    - Toggle: “Feature selection by ANOVA F”
    - Parameter: keep top-k features.
    - Description: Uses ANOVA F-test to select features with high variance across classes.
    - Purpose: Effective for linear models, especially when numeric distributions fit ANOVA assumptions.
- PCA reduction
    - Toggle: “PCA reduction”
    - Parameter: n_components, interpreted either as fraction of variance (<=1) or number of components (int).
    - Description: Projects features into a lower-dimensional space using Principal Component Analysis.
    - Purpose: Removes redundancy, speeds up training, and can help models that are sensitive to collinearity.

5.2.7.7 Scaling & Normalization

The Scaling & Normalization expander defines how features are rescaled:

- Scaling
  - Toggle: “Scaling”
  - Scaler mode:
    - Z-Score (standard scaler)
    - Robust (median and IQR-based)
    - MinMax (maps into [a, b]) with custom bounds for min (a) and max (b)
  - Description: Rescales features to a standard range or distribution.
  - Purpose: Essential for models relying on distances or gradients (e.g., SVM, MLP), and beneficial for many optimization routines.
- Row-wise L2 normalization
  - Toggle: “Row-wise L2 normalization”
  - Description: Normalizes each sample (row) to unit L2 norm.
  - Purpose: Useful for similarity-based methods where magnitude is irrelevant and only direction matters, such as text feature vectors.

All options are stored in keys like _PP_SC, _PP_SC_S, _PP_SC_S_MIN, _PP_SC_S_MAX, and _PP_L2N.

5.2.7.8 Outliers & Transforms

This expander configures how to treat outliers and skewed distributions:

- Clipping
  - Toggle: “Clipping”
  - Method: “IQR” or “Percentile”
  - Parameters:
    - For IQR: whisker width
    - For Percentile: low % and high % bounds
  - Description: Caps extreme values using statistical thresholds.
  - Purpose: Prevents a few extreme points from distorting the scale and model behavior.
- log1p transform
  - Toggle: “log1p transform”
  - Description: Applies log(1 + x) to selected numeric features.
  - Purpose: Reduces skewness, stabilizes variance, and often improves performance of linear models and distance-based algorithms.

5.2.7.9 Class Imbalance

The Class Imbalance expander contains strategies for handling uneven class distributions:

- Class weights
  - Toggle: “Class weights”
  - Scheme: “balanced” or “uniform”
  - Description: Adjusts training loss so minority classes receive higher weight.
  - Purpose: Reduces bias toward majority classes and improves recall for minority labels without changing the dataset.
- Resampling
  - Toggle: “Resampling”
  - Method: “SMOTE”, “Random Over”, or “Random Under”
  - Parameter (for SMOTE): k_neighbors
  - Description: Changes dataset composition by adding or removing samples.
  - Purpose: Balances the dataset distribution by oversampling minority classes (SMOTE, Random Over) or undersampling majority ones (Random Under).

These settings are reflected in keys like _PP_CW, _PP_CW_S, _PP_RS, _PP_RS_M, and _PP_RS_SK.

5.2.7.10 SVM Kernel Approximations

This expander provides options to approximate non-linear SVM kernels with explicit feature mappings:

- Toggle: “SVM Kernel Approximation”
- Method: “Nyström” or “Random Fourier Features”
- Parameters: n_components for each method (controls dimensionality of the approximation)
- Description: Nyström approximates the kernel matrix via low-rank methods, while Random Fourier Features approximate RBF kernels using random basis expansions.
- Purpose: Allows SVM to scale to larger datasets by approximating kernel computations in an explicit feature space, reducing computational cost while retaining much of the non-linear decision boundary.
-

5.2.7.11 Time Features

The Time Features expander adds features derived from timestamp columns:

- Toggle: “Add time features” (only available if a timestamp column is present
- Description: Extracts calendar-based attributes such as day, month, weekday, hour, or seasonal indicators from timestamps.
- Purpose: Turns raw temporal information into structured predictors suitable for non-temporal models, capturing daily, weekly, or seasonal patterns.

If no timestamp column is available, the UI indicates that this option cannot be used.

5.2.7.12 Applying the Selected Preprocessing

Preprocessing can only be applied once the dataset has been split. If the data is not yet split, a warning informs the user that splitting is a prerequisite.

If splits exist, the button “Apply selected preprocessing” becomes active. When clicked:

- The current split datasets are read from session state: X_train, y_train, X_validate, y_validate, X_test, y_test
- A configuration dictionary is assembled from all _PP_* and _TF flags and parameters, representing the full preprocessing plan.
- The backend function pp.preprocessModel() is called with:
    - The train/validation/test splits
    - The configuration object
- The function returns:
    - Preprocessed X_train, X_val, X_test
    - Corresponding labels
    - Class weights
    - Sample weights per split
    - Metadata dictionary (e.g., label encoding, transformations applied)
- All returned artifacts are stored back into Streamlit session state under _PP_* keys (e.g., _PP_X_Train, _PP_SWTR, _PP_META, _PP_LE).
- A log entry “Dataset preprocessed” is written using the logger module, and a success message is displayed.

A boolean flag _PP_IsPP indicates that preprocessing has been successfully applied.

5.2.7.13 Results Preview

If preprocessing is completed, the page shows a “Results preview” section:

- For each split (Train, Validate, Test), the UI concatenates features and labels and displays the first 10 rows in separate tables.
- Class weights are printed as JSON so users can inspect how imbalance was handled internally.
- The full metadata dictionary is also shown as JSON, documenting encoding maps (e.g., label encoding), preprocessing decisions, and any other artifacts required for reproducibility.

This preview acts both as a sanity check and as lightweight documentation of the final feature space that the Trainer & Evaluator module will consume.

5.2.8 Trainer & Evaluator

5.2.8.1 Purpose

The "Train & Evaluate" page is the final core step of the ARC pipeline. It takes the preprocessed train/validation/test splits and trains a supervised machine learning model, computes common performance metrics, visualizes confusion matrices and ROC/PR curves, and exposes feature importance (for tree-based models). It also allows users to download the trained model and the metrics as artifacts for later reuse and reporting.

The page is tightly integrated with the Preprocessor: it assumes that the dataset has already been split and preprocessed, and it reads all required inputs from Streamlit's session state.

5.2.8.2 Initial checks and layout

At the top, the shared sidebar is rendered to expose global navigation. The page then sets the title "Train & Evaluate". Before any controls are shown, two safety checks are performed:

1. If no dataset has been loaded (_DL_DataLoaded == False), an error message instructs the user to load a dataset first.
2. If the dataset has not yet been preprocessed (_PP_IsPP == False), the page shows an informational message that preprocessing must be completed before training.

Only if both conditions are satisfied will the training UI be displayed.

5.2.8.3 Model selection

The main configuration starts with a model selection dropdown. The following model families are available:

- Random Forest (RF)
- Gradient Boosting (XGBoost / LightGBM / CatBoost)
- Support Vector Machine (SVM)
- Neural Network (MLP)
- Logistic Regression (LR)

The user selects one of these models via a select box labeled "Model:". The chosen option is assigned to a local model variable and drives how the underlying estimator is built in the backend (via te.build_estimator inside the trainer/evaluator module).

5.2.8.4 GBM engine selection

If the selected model is of type "Gradient Boosting (XGBoost / LightGBM / CatBoost)", the page checks which gradient boosting libraries are actually available in the environment by calling te.checkGBM_engine().

- For each available library (LightGBM, XGBoost, CatBoost), the name is added to a local list of engines.
- If none of the libraries are installed, the UI displays a warning ("No GBM library available. Falling back to Random Forest.") and silently switches the selected model to Random Forest (RF) as a fallback.
- If at least one engine is available, a second select box appears ("GBM Engine"), allowing the user to choose which implementation to use. The chosen engine name is stored in a local variable gbm_engine and later persisted in st.session_state._GBM_Engine when training runs.

5.2.8.5 Input datasets and sample weights

The page then retrieves the preprocessed datasets and their associated sample weights from session state:

- _PP_X_Train, _PP_y_Train
- _PP_X_Validate, _PP_y_Validate
- _PP_X_Test, _PP_y_Test
- _PP_SWTR (sample weights for training)
- _PP_SWVA (sample weights for validation)
- _PP_SWTE (sample weights for test)
- _PP_META (metadata, including label encoding and GBM categorical metadata)

These objects are assumed to have been produced by the Preprocessor page (pp.preprocessModel).

5.2.8.6 Running training

Training is triggered via the "Run training" button. When the user clicks this button:

1. The currently selected gbm_engine (if any) is saved in st.session_state._GBM_Engine.
2. A safety check ensures that training data (X_Train, y_Train) exists. If not, an error is displayed and execution stops.
3. The trainer-evaluator orchestrator te.train_eval_orchestrator is called with:
    a. model: the selected model family name
    b. use_preprocessed=True: indicates that inputs are already preprocessed
    c. gbm_engine: the chosen GBM backend (if applicable)
    d. Train/validation/test sets and their sample weights
    e. meta: metadata from preprocessing (e.g., label encoding, categorical indices for CatBoost)
4. The orchestrator function internally builds the estimator (and any required preprocessing pipeline for raw pipelines; in this UI it always uses the preprocessed flow), trains it, times training, evaluates on all available splits, and measures model size and feature count.
5. The resulting dictionary (containing the trained model and all metrics) is stored in session state as _TE_PRes. The model name is stored in _TE_Model, and a flag _TE_PTrained is set to True to indicate that training has been completed successfully. If a GBM model was used, _GBM_Engine is also saved.
6. A log entry "Model trained" is written via the logger module.
7. The page reruns to display the results section.

5.2.8.7 Results section — general metrics

If _TE_PTrained == True, the page renders the results in a structured way using the results dictionary stored in _TE_PRes.

First, a "Results of training with preprocessed data" header is shown, along with a small CSS snippet that centers the metrics. Under the "General" subheader, three key aggregate metrics are displayed using Streamlit metric widgets:

- Train time (seconds) — total training time measured in train_time_sec
- Model size — human-readable size based on model_size_bytes and formatted via te.fmt_bytes
- Number of features — feature dimensionality as inferred from the training data (n_features)

These provide a concise overview of efficiency: how long training took, how large the serialized model is, and how many features the final model uses.

5.2.8.8 Detailed metrics table

Under the “Metrics” subheader, the code builds a table of evaluation metrics for available splits (train, validation, test). For each split, if present, the following metrics are collected from results[split]["metrics"]:

- Accuracy
- F1 (macro)
- Precision (macro)
- Recall (macro)
- ROC-AUC (if probability estimates were available)
- Inference time [s] for that split (inference_time_sec)

The rows are assembled into a pandas DataFrame, indexed by split, and displayed directly. This gives a split-by-split view of predictive performance and inference latency.

5.2.8.9 Confusion matrices

Next, the page shows confusion matrices for each split under the “Confusion Matrix” subheader.
Class label names are determined in two steps:

1. If the trained model exposes a classes_ attribute, it is used as a base.
2. The stored label encoding mapping _PP_LE (a dictionary from label name to encoded integer) is loaded from session state. The final class_names list is constructed by sorting this mapping by the integer ID and taking the names in that order. This guarantees that the axis labels of the confusion matrix align with the label encoding used during training.

Three tabs are rendered:

- Train
- Validation
- Test

For each tab, if the corresponding split metrics are available, the confusion matrix is read from results[split]["metrics"]["confusion_matrix"], converted to a NumPy array, and plotted via te.plot_confusion. The Plotly-based heatmap shows counts in each cell, with color intensity reflecting normalized values, and provides tooltips for interactive inspection.

5.2.8.10 ROC curves

If the trained model exposes a predict_proba method, the page renders ROC curves under the “ROC” subheader. Again, three tabs are used (“Train”, “Validation”, “Test”). For each tab:

- If the split exists, the page calls te.plot_roc(results["model"], X_split, y_split, class_names).
- The helper computes ROC curves either in binary mode or one-vs-rest for multiclass cases.
- The curves are returned as Plotly figures and displayed if available.

The code defensively wraps plotting calls in try/except blocks so that any unexpected errors in probability computation or ROC plotting do not break the UI; instead, the curve simply remains absent.

5.2.8.11 Precision–Recall curves

Similarly, if predict_proba is available, the “Precision-Recall” subheader displays PR curves in three tabs (“Train”, “Validation”, “Test”) by calling te.plot_pr for each split.

- For binary problems, the positive class curve (and its average precision) is plotted.
- For multiclass problems, one curve per class is shown (one-vs-rest).

These plots are especially useful in imbalanced data scenarios, where PR curves often provide a more informative view than ROC curves.

5.2.8.12 Feature importance visualization

After the ROC/PR sections, the page tries to compute and display feature importances using te.feature_importance_fig.

- Feature names are pulled from X_Train.columns if available.
- The helper inspects the trained model (or the final classifier step of a pipeline) for a feature_importances_ attribute (present in many tree-based models such as Random Forest and Gradient Boosting).
- If importances are found, a horizontal bar chart with the top 25 features is rendered under the subheader "Feature Importances (top 25)".

This gives users insight into which features the model relied on most, aiding interpretability and explaining preprocessing decisions.

5.2.8.13 Model download

The page then attempts to provide a downloadable model artifact:

- A binary buffer is created and the trained model object is serialized with pickle.dump using the highest available protocol.
- A "Download model (.pkl)" button is shown, offering the serialized model as a .pkl file (model.pkl) with MIME type application/octet-stream.

If serialization fails for any reason, a short informational message is displayed explaining that model download is unavailable and including the exception message.

5.2.8.14 Metrics download

Finally, the tool offers a metrics JSON download:

- A shallow copy of the results dictionary is created. The model entry is replaced by a string containing the model's type instead of the full object (to keep the JSON serializable).
- The metrics dictionary is serialized to JSON using json.dumps (with default=float to handle NumPy numbers) and a readable indentation.
- A "Download metrics (.json)" button is shown, providing the JSON document as metrics.json with MIME type application/json.

If JSON export fails, the UI displays an informative message explaining that metrics download is unavailable and prints the underlying exception.

5.2.9 Comparer

5.2.9.1 Purpose

The “Comparison” page quantifies the impact of ARC’s standardized preprocessing pipeline. It trains the same model twice — once on the raw (only split) dataset and once on the fully preprocessed dataset — and then compares efficiency and predictive performance side by side. This gives a concrete, experiment-style view of whether preprocessing improves training time, model size, feature count, and metrics such as accuracy, F1, precision, recall, ROC-AUC, as well as confusion matrices, ROC/PR curves, and feature importances.
The page assumes that the user has already:

- Loaded a dataset via the Data Loader.
- Split the dataset into train/val/test via the Splitter page.
- Preprocessed the splits via the Preprocessor page.
- Trained at least one model on the preprocessed data via the Train & Evaluate page.
-

5.2.9.2 Initial checks and layout

As on all pages, the shared sidebar is rendered first using the sidebar helper. The page then displays the title “Comparison”. Before showing any controls, three prerequisites are checked in sequence:

1. If no dataset is loaded (_DL_DataLoaded == False), an error message instructs the user to load a dataset first.
2. If the dataset is not preprocessed (_PP_IsPP == False), an informational message tells the user to preprocess the data first.
3. If no model has been trained on preprocessed data (_TE_PTrained == False), an informational message indicates that a model needs to be trained in the “Train & Evaluate” page beforehand.

Only when all three conditions are satisfied does the comparison interface become active.

5.2.9.3 Preparing raw and preprocessed datasets

Once prerequisites are met, the page retrieves both versions of the dataset from session state.
Preprocessed version (used as “PRE” in the UI):

- _PP_X_Train, _PP_y_Train
- _PP_X_Validate, _PP_y_Validate
- _PP_X_Test, _PP_y_Test
- _PP_SWTR, _PP_SWVA, _PP_SWTE (sample weights for train/val/test)
- _PP_META (metadata, including label encoding and GBM-related metadata)

Raw version (used as “RAW” in the UI, but already split):

- _SP_X_Train, _SP_y_Train
- _SP_X_Validate, _SP_y_Validate
- _SP_X_Test, _SP_y_Test

For the raw model run, sample weights and metadata are explicitly set to None (NP_sw_tr, NP_sw_va, NP_sw_te, NP_meta = None), because the raw splits are used without the preprocessing pipeline’s rebalanced weights or metadata.

5.2.9.4 Training on raw data

The central control at the top of the comparison logic is the “Run training on raw data” button. When clicked, the page launches a new training run on the raw (non-preprocessed) splits using the same model type and GBM engine configuration that were previously used for the preprocessed model:

- model is taken from st.session_state._TE_Model, which was set in the Train & Evaluate page.
- gbm_engine is taken from st.session_state._GBM_Engine, if applicable.

The call to te.train_eval_orchestrator uses:

- use_preprocessed=False to signal that raw splits must be passed through the minimal raw pipeline (handled inside the trainer/evaluator module).
- X_train, y_train, X_val, y_val, X_test, y_test from the _SP_ (split-only) session state variables.
- No sample weights and no preprocessing metadata.

The resulting metrics dictionary, including the trained raw model, is stored in session state as _C_NPRes, with a flag _C_NPTrained set to True. A log entry “Raw model trained” is written via the logger module, and the page reruns to display the comparison.
If the raw model has not yet been trained (_C_NPTrained == False), no comparison plots or metrics are shown.

5.2.9.5 General comparison and split-wise metrics

Once _C_NPTrained == True, the page retrieves both result objects:

- results_NP — metrics and model for the raw pipeline (non-preprocessed).
- results_P — metrics and model for the preprocessed pipeline (from _TE_PRes).

The page enables a compact metric comparison at the top:

1. A short CSS snippet centers all metric widgets for better readability.
2. A caption clarifies the conventions: “RAW = unprocessed | PRE = preprocessed”.
3. te.render_compare_general(results_NP, results_P) is called, which renders three grouped metrics as Streamlit metrics:
    a. Training time (seconds) for RAW vs PRE, including percentage change (Δ%) with green/red highlighting (lower is better).
    b. Model size (bytes formatted via fmt_bytes) for RAW vs PRE, with Δ% highlighted (smaller is better).
    c. Number of features (n-features) for RAW vs PRE, with percentage change.
4. te.render_compare_metrics(results_NP, results_P) is called, which builds a multi-index comparison table across splits (train, validation, test) and metrics:
    a. Accuracy, F1 (macro), Precision (macro), Recall (macro), ROC-AUC.
    b. Inference time per split.

For each metric, the table includes RAW value, PRE value, and the difference in percentage points (Δ (pp)) or relative percent for inference time. Conditional styling (green for improvements, red for degradations) makes it easy to see where preprocessing helps or hurts.
This section serves as the high-level quantitative comparison, showing at a glance how preprocessing affects both efficiency and prediction quality.

5.2.9.6 Confusion matrix comparison

To compare error patterns, the page shows confusion matrices side by side for RAW and PRE models.
First, class label names are computed:

- For the preprocessed model, the label encoding mapping _PP_LE (from preprocessing metadata) is used. Sorting this mapping by encoded integer yields a stable class order class_names_P.
- For the raw model, class names are inferred from results_NP["model"].classes_, yielding class_names_NP.

Under the “Confusion Matrix” subheader, three tabs are presented:

- Train
- Validation
- Test

Within each tab, two columns are used for side-by-side comparison:

- Left column (labelled “RAW”) displays the confusion matrix for the raw model.
- Right column (labelled “PRE”) displays the confusion matrix for the preprocessed model.

For each split and pipeline, the confusion matrix is taken from results_*[split]["metrics"]["confusion_matrix"], converted to NumPy arrays, and plotted using te.plot_confusion. Using the same plotting function ensures a consistent color scale and annotation style across RAW and PRE.

5.2.9.7 ROC curve comparison

If the preprocessed model supports probability estimation (predict_proba), the page also compares ROC curves. This requirement is checked by hasattr(results_P["model"], "predict_proba") before rendering the ROC section.

The “ROC” subheader again provides three tabs (“Train”, “Validation”, “Test”), and inside each tab two columns (“RAW” and “PRE”) are shown:

- RAW side: calls te.plot_roc(results_NP["model"], X_split_raw, y_split_raw) for the selected split. Note that in the current implementation, a few calls pass X twice (e.g., NP_X_Train for both features and labels in one place). Conceptually, the intended behavior is to pass the correct X and y for each split to compute ROC curves.
- PRE side: calls te.plot_roc(results_P["model"], X_split_pre, y_split_pre, class_names_P) to generate ROC curves aligned with the known class names from the preprocessed label encoding.

Each ROC plotting call is wrapped in a try/except block so that any issues with probability prediction or plotting do not crash the app; instead, the corresponding curve is silently skipped.

5.2.9.8 Precision–Recall curve comparison

Similarly, if probability estimates are available, the “Precision-Recall” section displays PR curves side by side for RAW and PRE:

- Three tabs (“Train”, “Validation”, “Test”).
- In each tab, two columns compare RAW and PRE curves.

For each split:

- RAW: te.plot_pr(results_NP["model"], X_split_raw, y_split_raw) is called.
- PRE: te.plot_pr(results_P["model"], X_split_pre, y_split_pre, class_names_P) is called with known class names.

Again, each call is wrapped in try/except blocks for robustness. PR curves are especially useful to understand performance on minority classes and how preprocessing affects class imbalance handling.

5.2.9.9 Feature importance comparison

At the bottom, the page compares feature importances between the raw and preprocessed pipelines.

- For the raw model, feature names are taken from NP_X_Train.columns, if available, and te.feature_importance_fig is called with results_NP["model"] and these names.
- For the preprocessed model, feature names come from PP_X_Train.columns and the same helper is called with results_P["model"].

Both calls return Plotly bar charts (if the underlying estimator exposes feature_importances_, as is the case for tree-based models).

If at least one importance figure is available, the page shows a “Feature Importances (top 25)” subheader and displays two plots in columns:

- Left column (“RAW”) shows feature importances for the raw model.
- Right column (“PRE”) shows feature importances for the preprocessed model.

This visualization highlights how preprocessing changes the feature space (e.g., new engineered or encoded features, dropped columns) and how the model’s focus shifts across raw versus transformed features.

5.2.10 Reporter

5.2.10.1 Purpose

The “Reporter” page is the final export hub of the ARC UI. It lets users generate a consolidated PDF report of the entire pipeline run and download all important artifacts: validated datasets, split datasets, preprocessed datasets, logs, preprocessing metadata, and the trained model plus metrics. In other words, this page turns an interactive ARC session into a portable, auditable bundle that can be archived, shared, or attached to scientific experiments.

5.2.10.2 Overall layout

The page is structured into three vertical sections arranged as columns:

1. PDF report (left column) – controls what is included in the automatically generated PDF report and provides a download button.
2. Datasets (middle column) – provides CSV downloads for the validated dataset, split datasets, and preprocessed datasets.
3. Other (right column) – offers downloads for the pipeline log, preprocessing metadata, and the final trained (preprocessed) model plus its metrics.

At the top, the page displays the main title “Reporter”.

5.2.10.3 PDF report (left column)

The left column is headed with “PDF report” and the text “Include in the PDF report:”. This section configures which pipeline modules are added to the PDF produced by the Reporter module.
Before showing the checkboxes, the page enforces consistency by resetting flags if prerequisites are not met:

- If no dataset is loaded (_DL_DataLoaded == False), it forcibly disables the Data Loader and Schema Validator sections in the report (_R_DL = False, _R_SV = False), because these report sections would have no data.
- If the dataset has no label column (_HasLabel == False), the Label Validator section is disabled (_R_LV = False).
- If the dataset is not split (_SP_IsSplit == False), the Splitter section is disabled (_R_S = False).
- If preprocessing has not been completed (_PP_IsPP == False), the Preprocessor-related Trainer & Evaluator section for preprocessed runs is disabled (_R_PP = False).
- If no preprocessed model has been trained (_TE_PTrained == False), the Trainer & Evaluator section is disabled (_R_TE = False).
- If no raw-vs-preprocessed comparison has been run (_C_NPTrained == False), the Comparer section is disabled (_R_C = False).

Each module has a corresponding checkbox that the user can (de)select, but checkboxes are disabled when their prerequisites are not satisfied:

- Data Loader (_R_DL) – includes the Data Loader section in the PDF (dataset information, file sources, basic statistics). Disabled if no dataset is loaded.
- Schema Validator (_R_SV) – includes schema insights (feature types, sparsity, granularity, timestamps) in the PDF. Disabled if no dataset is loaded.
- Label Validator (_R_LV) – includes label consistency checks, entropy, class distribution, and temporal drift charts. Disabled if the dataset has no label.
- Splitter (_R_S) – includes split configuration and quality checks in the PDF. Disabled if splits have not been created.
- Trainer & Evaluator (_R_TE) – includes training metrics, confusion matrices, ROC/PR curves, and feature importances for the preprocessed pipeline. Disabled if no preprocessed model was trained.

- Comparer (_R_C) – includes comparison plots/metrics between raw and preprocessed pipelines. Disabled if the comparison run has not been performed.
- Log (_R_L) – includes the pipeline log section. This can always be enabled or disabled independently because the log is maintained across pages.

After the inclusion options are set, the page calls cPDF.create_pdf_story(), which generates the PDF content according to the selected flags and returns the report as bytes.

To name the PDF file, the code derives a base name from st.session_state._DataLoader_FileName. It takes only the stem (filename without extension); if that is missing, it falls back to "report". The download button then serves a file named <base>_report.pdf, with content type application/pdf.

This means the PDF report is always tied to the current ARC session's dataset name and the user's checkbox selections, and it can be downloaded directly from this column.

5.2.10.4 Datasets (middle column)

The middle column is titled "Datasets" and is subdivided into three logical parts:

1. Validated dataset
2. Splitted datasets
3. Preprocessed datasets

Each subsection uses conditional download buttons: if the corresponding data object is available in session state, a CSV download is offered; otherwise, a disabled placeholder button is shown so the user can see what would be available after running the previous steps.

5.2.10.4.1 Helper function

A small helper convert_df(df) converts any pandas DataFrame into UTF-8 encoded CSV bytes (df.to_csv(index=False).encode('utf-8')). All dataset downloads in this column rely on this helper to ensure consistent encoding and CSV formatting.

5.2.10.4.2 Validated dataset

- Uses st.session_state._DF, which holds the current validated and modified dataset after all schema and label operations.
- If _DF is not None, the user can download it as a CSV file. The filename is derived from the original dataset name st.session_state._DL_Filename, but with the extension normalized to ".csv" (<original_basename>.csv).
- If _DF is None, a disabled button is displayed, indicating that the validated dataset is not yet available (e.g., no dataset loaded).

5.2.10.4.3 Splitted datasets

This subsection offers downloads for each split from the Splitter page:

- X_train and y_train
- X_validate and y_validate
- X_test and y_test

These are read from:

- _SP_X_Train, _SP_y_Train
- _SP_X_Validate, _SP_y_Validate
- _SP_X_Test, _SP_y_Test

For each of these six objects:

- If the DataFrame (or Series) is not None, a download button is shown.
    - X_* splits are written directly via convert_df(X_*).

    - y_* splits are converted to a one-column DataFrame via y_*.to_frame() before conversion, then exported as CSV.
- If a given split is None, a disabled "as CSV" button is shown, clarifying that this artifact will become available once the Splitting step has been performed.

This gives users a complete set of split datasets they can export for external experiments or backup.

5.2.10.5 Preprocessed datasets

The final dataset subsection mirrors the split downloads but for the Preprocessor output:

- PP_X_train, PP_y_train
- PP_X_val, PP_y_val
- PP_X_test, PP_y_test

These are read from:

- _PP_X_Train, _PP_y_Train
- _PP_X_Validate, _PP_y_Validate
- _PP_X_Test, _PP_y_Test

Behavior is analogous to the split downloads:

- If a preprocessed split exists, the page offers a CSV download using convert_df, again converting label Series to DataFrames with to_frame().
- Otherwise, a disabled button signals that those preprocessed splits are not yet available (e.g., preprocessing has not been run).

This section thus exposes both pre-split (validated) and post-preprocessing splits in a reproducible CSV format, allowing external reuse of exactly the data that ARC used internally.

5.2.10.6 Other artifacts (right column)

The right column is titled "Other" and provides downloads for additional artifacts beyond the datasets:

1. Log – the pipeline log as a plain text file.
2. Preprocess meta data – the preprocessing metadata as JSON.
3. Preprocessed model – the final trained preprocessed model as a PKL file and its metrics as JSON.

5.2.10.6.1 Log export

The "Log" subsection takes the in-memory log stored in st.session_state._LogData and renders it into a single text blob:

- It iterates through _LogData, which contains log entries.
- If an entry is a two-element list or tuple (timestamp, message), it formats it as: <timestamp>\n<message>
- If the structure is different, it falls back to str(e) to ensure nothing breaks.
- All entries are joined with two newlines (\n\n) between them into a variable txt.

The "Download log as .txt" button then:

- Is enabled only if _LogData is non-empty (otherwise the button is disabled).
- Exports txt encoded as UTF-8, with file name log.txt and MIME type text/plain.

This gives users a complete textual trace of all operations performed and decisions made during the ARC run.

5.2.10.6.2 Preprocess meta data

The "Preprocess meta data" subsection exports the metadata stored in _PP_META as a JSON file. Because _PP_META often contains NumPy, pandas, sets, or arrays, two helper functions ensure the object is JSON-serializable:

- _json_ready(o) converts various types to JSON-friendly forms:
  - NumPy scalars (np.integer, np.floating, np.bool_) are converted to Python int, float (with NaN/inf mapped to None), or bool.
  - pandas.Series, DataFrame, and Index are converted to lists or dictionaries.
  - Sets and tuples are converted to lists.
  - Objects with .tolist() are converted via that method when possible.
  - Any remaining types fall back to str(o).
- to_json_bytes(obj) uses json.dumps with default=_json_ready, indentation, and UTF-8 encoding to produce clean JSON bytes.

If _PP_META is None, a disabled button labelled "Download PP metadata as JSON" is shown, indicating preprocessing hasn't produced metadata yet.
If _PP_META is available, a "Download metadata as JSON" button appears, allowing the user to download preprocess_metadata.json. This JSON captures label encodings, feature statistics, GBM settings, and any other metadata needed for reproducibility.

5.2.10.6.3 Preprocessed model and metrics

The final subsection, "Preprocessed model", handles export of the trained model and its evaluation metrics, but only if a model has been trained on preprocessed data (_TE_PTrained == True).
If _TE_PTrained is True:

- The preprocessed training results dictionary is loaded from _TE_PRes.
- The model object results["model"] is serialized with pickle.dump to an in-memory BytesIO buffer.
- The "Download PP model as PKL" button offers this buffer as a file named model.pkl with MIME type application/octet-stream.
  - If serialization fails, an informational message "Model download unavailable: " is shown instead of crashing the app.
- For the metrics, a shallow copy of the result dictionary is created and the model entry is replaced by a string describing its type (to keep the JSON serializable): copy["model"] = str(type(results["model"])).
- The "Download PP metrics as JSON" button exports this JSON as metrics.json with numeric values converted to native Python types by json.dumps (using default=float for non-standard numeric types). Again, if serialization fails, an informational message explains that the metrics download is unavailable.

If _TE_PTrained is False: Two disabled placeholder buttons are displayed ("Download PP model as PKL" and "Download PP metrics as JSON") to indicate what would become available after training a model.

6 Run the code

To run the project you can either use the provided Docker image or start Streamlit directly. If you use Docker, first load the prebuilt image from the Docker Image folder (for example docker load -i Docker\ Image/arc.tar), then start the app with docker run --rm -p 8501:8501 arc and open the browser at http://localhost:8501. If you want to build the image yourself, go to the ARC Code directory that contains the Dockerfile and run docker build -t arc ., then start it with the same docker run --rm -p 8501:8501 arc command. To run the code locally without Docker, create a Python environment, install the dependencies with pip install -r requirements.txt inside the ARC Code folder, and then start the app via streamlit run ARC/A_R_C.py.