Heriot-Watt University | Edinburgh | Edinburgh Campus, Edinburgh, EH14 4AS.

AutoMedTS Project: Technical Report

Baobing Zhang and Mustafa Suphi Erden

Edinburgh, May 18, 2025

Abstract

AutoMedTS is an open-source automated machine learning framework designed for medical time-series data, originally developed to deliver real-time action recognition and feedback in laparoscopic surgery training. It employs sliding-window segmentation on surgical suture coordinate sequences and integrates multiple class-imbalance handling strategies (including power transform, temperature-scaled softmax, and SMOTE). *AutoMedTS* automatically searches for optimal models and hyperparameters, providing an end-to-end pipeline for training and inference. The framework is easily extensible to other medical time-series tasks and offers comprehensive evaluation metrics (Macro-F1, Balanced Accuracy, MCC) along with visualization tools for performance analysis and feedback. This document serves as a comprehensive technical report, detailing the key workflows, core functions, and the associated mathematical formulations and pseudocode of *AutoMedTS*, offering a reusable and efficient solution for intelligent medical time-series analysis.

Contents

1 Introduction

1.1 Background & Motivation

In clinical settings, many critical events—such as rare surgical maneuvers or abnormal physiological signals—are represented by only a handful of data samples, while common events occur in abundance. This severe class imbalance not only causes traditional classifiers to overfit the majority classes but also risks missing minority events, thereby undermining the reliability and safety of downstream decision-making. Hence, devising effective resampling strategies tailored for medical time-series data is a pivotal step toward improving model robustness and real-world clinical performance.

1.2 Project Overview

AutoMedTS is an end-to-end automated ML platform tailored for medical time-series data, focusing on fine-grained analysis and classification of continuous actions such as surgical maneuvers and physiological signals. Its core features include:

1. **Seamless Pipeline:** One-click workflow from raw time series → sliding-window segmentation → feature flattening → model search → evaluation and deployment.

2. **Multi-Strategy Balancing:** Built-in modules for frame-level jitter augmentation, power-law mapping, temperature-scaled softmax, SMOTE, etc., which can be flexibly combined based on data characteristics.

3. **Intelligent Auto-Tuning:** Integrates Bayesian optimization and meta-learning to automatically discover optimal model architectures, hyperparameters, and preprocessing pipelines, significantly reducing experiment time.

4. **High Customizability:** Modular plugin architecture supports user-defined features, balancing strategies, model structures, and evaluation metrics.

5. **Clinical-Grade Deployment:** Supports real-time online inference and visual report generation, balancing performance and interpretability to meet medical safety and compliance requirements.

2 Overall System Architecture

The *AutoMedTS* framework is organized into five cohesive stages. First, *Data Loading & Preprocessing* ingests raw 3D instrument-tip trajectories, validates and encodes labels, handles missing values, and—if necessary—applies dataset compression to fit memory constraints. Second, *Sliding-Window Segmentation & Balancing* partitions the continuous trajectory into overlapping fixed-length windows and applies distribution-mapping strategies (e.g., temperature-scaled softmax) coupled with over- and under-sampling to mitigate class imbalance. Third, *Configuration-Space Construction* dynamically builds a unified hyperparameter search space—spanning data

preprocessing, feature transformation, and a suite of candidate learning algorithms—tailored to the task (classification vs. regression) and data sparsity. Fourth, a lightweight *Baseline Prediction* is executed to verify the end-to-end pipeline and establish a reference loss. Finally, *Bayesian SMBO & Uncertainty-Aware Ensembling* drives the core optimization loop: Sequential Model-Based Optimization (SMBO) iteratively proposes and evaluates model configurations, while an ensemble manager incrementally assembles a weighted committee of top candidates, penalizing high-variance learners to ensure robust, real-time surgical action recognition. The overall workflow as shown in Figure 1.

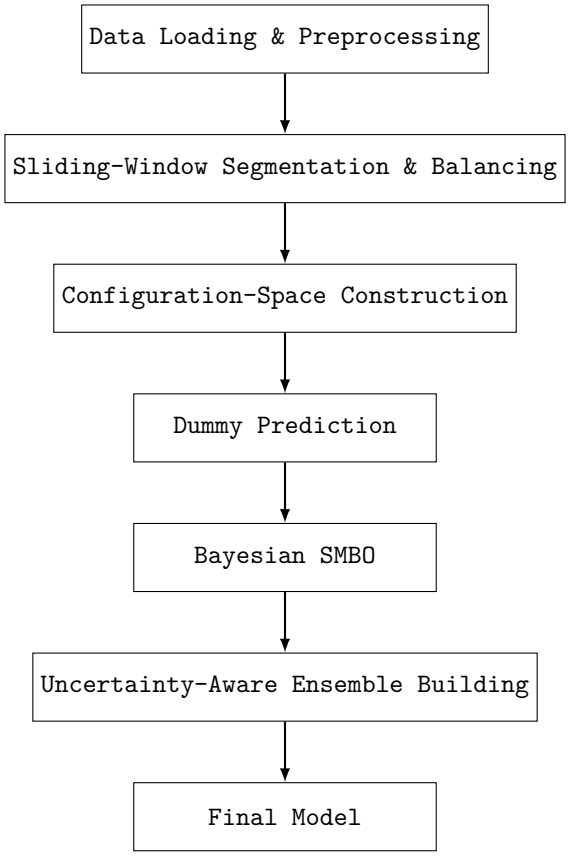

Abb. 1: *AutoMedTS* overall workflow

3 Core Modules and Functions

3.1 Data Processing and Class Balancing

Description. This module performs all preprocessing on the raw frame-level trajectories—first applying sliding-window segmentation and jitter augmentation to minority frames; next remapping class proportions (e.g. via temperature-scaled softmax); and finally over- or under-sampling each class to achieve a balanced distribution.

```
def window_and_balance(
X, y, window_size, step_size, strategy="softmax"
):
# 1) Frame-level jitter augmentation
X_aug, y_aug = augment_jitter(X, y, sigma=sigma, random_state=seed)
# 2) Sliding-window segmentation
X_win, y_win = create_windows(X_aug, y_aug, window_size, step_size)
```

```
# 3) Compute & map class proportions
new_props = _map_softmax(props, T)  # or _map_power, _map_linear
# 4) Determine per-class targets and resample
for cls in classes:
if orig_count >= target_count:
undersample(...)
else:
oversample(...)  # simple copy+jitter or SMOTE
return X_final, y_final
```

3.2 Input Validation and Transformation

Description. All inputs are run through an `InputValidator` that checks data types, encodes categorical features, applies one-hot or ordinal encoding, and enforces consistency between training and test sets.

```
# inside fit():
self.InputValidator = InputValidator(
is_classification=is_classification,
feat_type=feat_type,
logger_port=self._logger_port,
)
self.InputValidator.fit(
X_train=X, y_train=y, X_test=X_test, y_test=y_test
)
X, y = self.InputValidator.transform(X, y)
if X_test is not None:
X_test, y_test = self.InputValidator.transform(X_test, y_test)
```

3.3 Data Management

Description. The `XYDataManager` encapsulates training and test data, along with metadata such as task type and feature types, and serializes everything via the backend for downstream modules.

```
# inside fit():
datamanager = XYDataManager(
X, y,
X_test=X_test, y_test=y_test,
task=self._task,
feat_type=self._feat_type,
dataset_name=dataset_name,
)
self._backend._make_internals_directory()
self._backend.save_datamanager(datamanager)
```

3.4 Configuration-Space Construction

Description. Constructs the joint hyperparameter search space for preprocessing steps and base learners. It delegates to `get_configuration_space`, which in turn invokes either `SimpleClassificationPipeline` to enumerate all choices and forbidden combinations.

```
# inside fit():
with self._stopwatch.time("Create Search space"):
self.configuration_space, configspace_path = \
self._create_search_space(
self._backend.temporary_directory,
self._backend,
datamanager,
include=self._include,
exclude=self._exclude,
```

```
        )
```

`_get_classification_configuration_space`

Description. Builds the search space for classification tasks (binary, multi-class, multi-label), handling data sparsity via `SimpleClassificationPipeline`.

```
        def _get_classification_configuration_space(
        datamanager, include, exclude, random_state
        ):
        sparse     = datamanager.info["is_sparse"] == 1
        multiclass = datamanager.info["task"] == MULTICLASS_CLASSIFICATION
        multilabel = datamanager.info["task"] == MULTILABEL_CLASSIFICATION
        dataset_props = {
                "sparse": sparse,
                "multiclass": multiclass,
                "multilabel": multilabel
        }

        cs = SimpleClassificationPipeline(
        feat_type=datamanager.feat_type,
        dataset_properties=dataset_props,
        include=include,
        exclude=exclude,
        random_state=random_state
        ).get_hyperparameter_search_space(feat_type=datamanager.feat_type)
        return cs
```

`SimpleClassificationPipeline._get_hyperparameter_search_space`

Description. Delegates to the base search-space builder and then adds forbidden clauses to avoid incompatible choices (e.g. sparse data + densifier + incompatible classifier).

```
        class SimpleClassificationPipeline(BasePipeline, ClassifierMixin):
        def _get_hyperparameter_search_space(
        self, feat_type=None, include=None, exclude=None, dataset_properties=None
        ):
        cs = self._get_base_search_space(
        cs, feat_type, dataset_properties,
        include=include, exclude=exclude,
        pipeline=self.steps
        )
        # For sparse data + densifier, forbid incompatible classifiers
        for cls in cs.get_hyperparameter("classifier:__choice__").choices:
        if SPARSE in available_components[cls].get_properties()["input"]:
        cs.add_forbidden_clause(
        ForbiddenAndConjunction(
        ForbiddenEqualsClause(
        cs.get_hyperparameter("classifier:__choice__"), cls
        ),
        ForbiddenEqualsClause(
        cs.get_hyperparameter("feature_preprocessor:__choice__"),
        "densifier"
        )
        )
        )
        return cs
```

`SimpleClassificationPipeline._get_pipeline_steps`

Description. Defines the ordered list of pipeline stages and their choice containers.

```
def _get_pipeline_steps(self, dataset_properties, feat_type=None):
return [
["data_preprocessor",    DataPreprocessorChoice(...)],
["balancing",            Balancing(...)         ],
["feature_preprocessor", FeaturePreprocessorChoice(...)],
["classifier",           ClassifierChoice(...)  ],
]
```

3.5 Baseline (Dummy) Prediction

Description. Runs a single "dummy" configuration to establish a performance lower bound. This prediction is included among ensemble candidates.

```
# inside fit():
with self._stopwatch.time("Dummy predictions"):
self.num_run += 1
self._do_dummy_prediction()

# key lines in _do_dummy_prediction():
status, cost, runtime, info = ta.run(
config=1,
cutoff=self._time_for_task,
)
```

3.5.1 Dummy Prediction

Before the main SMBO loop, *AutoMedTS* performs a single "dummy" run (run number 1) to validate the evaluation pipeline and obtain a baseline loss. The method `_do_dummy_prediction`:

1. Skips execution if a partial-CV strategy is active.
2. Initializes a `Stats` object to track wall-clock time.
3. Calls `ExecuteTaFuncWithQueue` with `initial_num_run=1`.
4. Captures `(status, cost, runtime, info)`; logs success or raises `ValueError` with diagnostic information on failure.

```
def _do_dummy_prediction(self) -> None:
# Skip for partial-CV strategies
if self._resampling_strategy in [
"partial-cv", "partial-cv-iterative-fit"
]:
return

if self._metrics is None:
raise ValueError("No metrics set for dummy run.")

dummy_run = 1
scenario_mock = unittest.mock.Mock()
scenario_mock.wallclock_limit = self._time_for_task

# Track timing
stats = Stats(scenario_mock)
stats.start_timing()

# Execute exactly one target-algorithm run
ta = ExecuteTaFuncWithQueue(
backend=self._backend,
initial_num_run=dummy_run,
stats=stats,
```

```
            metrics=self._metrics,
            memory_limit=int(self._memory_limit or 0),
            disable_file_output=self._disable_evaluator_output,
            abort_on_first_run_crash=False,
            cost_for_crash=get_cost_of_crash(self._metrics),
            port=self._logger_port,
            pynisher_context=self._multiprocessing_context,
            **self._resampling_strategy_arguments
            )

            status, cost, runtime, info = ta.run(
            config=dummy_run,
            cutoff=self._time_for_task
            )
            if status != StatusType.SUCCESS:
            raise ValueError(f"Dummy prediction failed: {info}")
```

3.6 Ensemble Construction

Description. An `EnsembleBuilderManager` is spawned to incrementally collect new runs, evaluate their validation losses, and select a subset via an uncertainty-aware greedy forward-selection algorithm (`EnsembleSelection`) that penalizes high-variance predictors.

```
            # inside fit():
            proc_ensemble = EnsembleBuilderManager(
            start_time=time.time(),
            time_left_for_ensembles=time_left,
            backend=copy.deepcopy(self._backend),
            dataset_name=dataset_name,
            task=self._task,
            metrics=self._metrics,
            ensemble_class=self._ensemble_class,
            ...
            )

            # in EnsembleSelection.fit():
            def fit(self, base_models_predictions, true_targets, runs, ...):
            self._fast(predictions=base_models_predictions, labels=true_targets)
            self._calculate_weights()
            return self
```

`EnsembleBuilderManager` (SMAC Callback Manager)

- `__call__(self, smbo, run_info, result, time_left)`: Invoked by SMAC after each configuration evaluation; decides whether to trigger or collect an ensemble-building job.
- `build_ensemble(self, dask_client)`: Checks time and iteration limits, and—if appropriate— submits an asynchronous Dask task that calls the static method `fit_and_return_ensemble`.
- `fit_and_return_ensemble(...)`: Static method that creates an `EnsembleBuilder` instance, invokes its `run` method, and returns the ensemble performance history and updated `nbest`.

`EnsembleBuilder` (Runtime Ensemble Construction)

- `run(self, iteration, end_at, ...)`: Main loop that continues until timeout or max iterations; returns `(ensemble_history, nbest)`.

- `main(self, time_left, iteration)`: Core routine that
  1. loads and merges all `Run` objects (including dummy runs),
  2. updates missing losses,
  3. prunes via `requires_deletion` and `candidate_selection`,
  4. calls `fit_ensemble` to train the ensemble,
  5. records performance stamps and deletes discarded models.
- `requires_deletion(self, runs, max_models, memory_limit)`: Enforces an upper bound on model count or disk usage by discarding the worst-performing runs.
- `candidate_selection(self, runs, dummies, better_than_dummy, nbest)`: Selects a final set of candidate runs, first filtering by whether they beat a dummy baseline, then applying an *n*-best round-robin across metrics.
- `fit_ensemble(self, candidates, runs, targets, ...)`:
  Instantiates the chosen `ensemble_class` (default `EnsembleSelection`) and calls its `fit` method on the selected candidates.

`EnsembleSelection` (Uncertainty-Aware Greedy Ensemble)

- `__init__(..., uncertainty_penalty=0.1)`: Adds an uncertainty penalty coefficient to the standard greedy algorithm.
- `fit(self, base_models_predictions, true_targets, ...)`: Entry point that validates inputs, delegates to `_bagging` or `_fit`, and finally computes weights via `_calculate_weights()`.
- `_fast(self, predictions, labels, X_data)`: Greedy forward selection; at each round, picks the model minimizing

$$\text{loss}(\hat{y}) \;+\; \text{uncertainty\_penalty} \times \text{Var}(\text{prediction}).$$

- `_slow(self, predictions, labels, X_data)`: Exhaustive search version, similarly penalized by variance.
- `_calculate_weights(self)`: Counts how often each model was selected and normalizes these counts into final ensemble weights.
- `predict(self, base_models_predictions)`: Produces the weighted average of the base-model predictions using the learned weights.

3.7 Bayesian SMBO Optimization

Description. Invokes SMAC to perform Sequential Model-Based Optimization (SMBO) over the constructed search space. Optionally warm-starts with meta-learned configurations and registers the ensemble callback so that each new model is considered for ensembling.

```
# inside fit():
with self._stopwatch.time("Run SMAC"):
_proc_smac = AutoMLSMBO(
config_space=self.configuration_space,
dataset_name=self._dataset_name,
backend=self._backend,
total_walltime_limit=time_left,
func_eval_time_limit=per_run_time_limit,
metrics=self._metrics,
ensemble_callback=proc_ensemble,
...
)
self.runhistory_, self.trajectory_, self._budget_type = \
_proc_smac.run_smbo()
```

Bayesian SMBO Optimization: Call Flow and Key Components

3.7.1 Entry: Triggering SMAC in `AutoML.fit`

```
# == RUN SMAC
with self._stopwatch.time("Run SMAC"):
elapsed = self._stopwatch.time_since(self._dataset_name, "start")
time_left = self._time_for_task - elapsed

# 1.1 Compute per_run_time_limit (capped by time\_left)
# 1.2 Compute how many models can be generated:
#     num\_models = time\_left // per\_run\_time\_limit
# 1.3 Initialize the SMBO driver
smbo = AutoMLSMBO(
config_space=\dots,
dataset_name=self.\_dataset\_name,
backend=self.\_backend,
total\_walltime\_limit=time\_left,
func\_eval\_time\_limit=per\_run\_time\_limit,
\dots,
ensemble\_callback=proc\_ensemble,
trials\_callback=self.\_get\_trials\_callback,
)
# 1.4 Launch the SMBO loop and retrieve results
self.runhistory\_, self.trajectory\_, self.\_budget\_type = smbo.run\_smbo()
```

- **Responsibilities:**
  1. Compute remaining global budget and per-evaluation time limit.
  2. Instantiate `AutoMLSMBO` with resources, callbacks, configuration.
  3. Invoke `run_smbo()` to execute the Bayesian optimization loop.

3.7.2 `AutoMLSMBO.run_smbo()` Main Flow

```
def run_smbo(self):
self.stopwatch.start("SMBO")

# 2.1 Reload DataManager and set self.task
self.reset_data_manager()

# 2.2 Meta-learning: compute initial configurations
meta_cfgs = self.get_metalearning_suggestions()

# 2.3 Build the SMAC Scenario
scenario_dict = {
        "cs": self.config_space,
        "cutoff_time": self.func_eval_time_limit,
```

```
                "wallclock_limit": total_walltime_limit,
                "memory_limit": self.memory_limit,
                "instances": instances,
                "output-dir": self.backend.get_smac_output_directory(),
                "run_obj": "quality",
                "cost_for_crash": self.worst_possible_result,
                %\dots%
            }

            # 2.4 Create SMAC4AC (possibly via callback)
            smac = (self.get_smac_object_callback or get_smac_object)(
            scenario_dict=scenario_dict,
            seed=self.seed,
            ta=ExecuteTaFuncWithQueue,
            ta_kwargs=ta_kwargs,
            metalearning_configurations=meta_cfgs,
            n_jobs=self.n_jobs,
            dask_client=self.dask_client,
            %\dots%
            )

            # 2.5 Register callbacks
            if self.ensemble_callback:
            smac.register_callback(self.ensemble_callback)
            if self.trials_callback:
            smac.register_callback(self.trials_callback)

            # 2.6 Run optimization
            smac.optimize()

            # 2.7 Collect results
            self.runhistory   = smac.solver.runhistory
            self.trajectory   = smac.solver.intensifier.traj_logger.trajectory
            self._budget_type = smac.solver.tae_runner.budget_type

            self.stopwatch.stop("SMBO")
            return self.runhistory, self.trajectory, self._budget_type
```

- **2.1** `reset_data_manager()`: reloads the `XYDataManager`, sets `self.task`.
- **2.2** Meta-learning seed configurations via metafeature computation and `suggest_via_metalearning`.
- **2.3** Builds a `Scenario` defining CS, per-eval cutoff, global wall-clock budget, memory limit, instances, I/O.
- **2.4** Creates `SMAC4AC`, injecting scenario, target algorithm, meta-learning configs, `Dask` client, multi-objective settings.
- **2.5** Registers:
  - `EnsembleBuilderManager` for on-the-fly ensembles.
  - Trials callback (`IncorporateRunResultCallback`) for trajectory logging.
- **2.6** `smac.optimize()`: runs Bayesian optimization loop with surrogate fitting, acquisition optimization, parallel evaluations, and callback invocation.
- **2.7** Gathers:
  - `runhistory`: all evaluated configurations and losses.
  - `trajectory`: best-so-far performance curve.
  - `budget_type`: time- or iteration-based budget.

3.7.3 SMAC4AC and Core Components

Component	Role
`Scenario`	Defines the optimization problem: CS, resource limits, instances, I/O.
`SMAC4AC`	Main Bayesian optimizer: surrogate model, acquisition function, intensification.
`RunHistory2EPM4LogCost`	Converts run history to training data for surrogate (RF/GP).
`Intensifier/ SimpleIntensifier`	Budget allocation strategy: serial vs. parallel, multi-arm scheduling.
`ExecuteTaFuncWithQueue`	Wrapped target algorithm executor: trains model, evaluates performance, queues result.
`DaskParallelRunner/ SerialRunner`	Distributes evaluation jobs across Dask or local processes.

Tab. 1: Core SMAC4AC components and their responsibilities.

3.8 Model Loading

Description. After search and ensembling finish, the chosen base-learners and the final ensemble are loaded into memory for prediction.

```
# inside fit():
if load_models:
self._logger.info("Loading models...")
self._load_models()
self._logger.info("Finished loading models...")
```

3.9 Evaluation & Visualization

- **Classification Report**: precision, recall, F1, support per class.
- **Aggregate Metrics**: Accuracy, Macro-F1, Balanced Accuracy, Matthews Correlation Coefficient (MCC), PR–AUC.
- **Plots**:
  - Bar charts for testing result and visualization.

4 End-to-End Workflow

Figure 2 summarizes the complete data flow from raw time series to model predictions.

5 Usage Example

5.1 Training & Saving

```
from automedts.classification import automedtsClassifier
```

```
import joblib
import pandas as pd

# 1. load data
X_train, y_train = pd.read_csv('X_train.csv'), pd.read_csv('y_train.csv').values.rave

# 2. train
clf = automedtsClassifier(time_left_for_this_task=3600,
per_run_time_limit=100,
ensemble_kwargs={'ensemble_size': 100})
clf.fit(X_train.values, y_train)

# 3. save model
joblib.dump(clf, 'surgery_stage_model.pkl')
```

5.2 Loading & Predicting

```
import joblib
import pandas as pd

# load model
clf = joblib.load('surgery_stage_model.pkl')

# load test data
X_test, y_test = pd.read_csv('X_test.csv'), pd.read_csv('y_test.csv').values.ravel()

# predict
y_pred, _ = clf.predict(X_test.values, y=y_test)
```

6 Ablation Study & Results

Table 2 shows comparison of different resampling strategies. Subsequent experiments use the best strategy (Random Oversampling [1]).

Tab. 2: Comparison of different resampling strategies

Method	Accuracy	Macro-F1	Macro-Recall	Balanced Acc	MCC
Baseline (No Resampling)	0.8921	0.8791	0.8534	0.8534	0.8596
Random Oversampling	**0.9136**	**0.9094**	**0.9030**	**0.9030**	**0.8887**
SMOTE Resampling	0.8508	0.8282	0.8033	0.8033	0.8053
SMOTE + Jitter Augmentation	0.8231	0.8001	0.8196	0.8196	0.7759

7 Extensions & Hyperparameters

To facilitate future extensions and fine-tuning, *AutoMedTS* exposes a variety of parameters and plugin interfaces at multiple stages:

- **Resampling Strategy Extensions**
  - By adding new mapping functions (e.g. `_map_power`, `_map_linear`, or custom methods) inside `window_and_balance`, one can introduce novel class–proportion remapping strategies.
  - The boolean flags `USE_SMOTE_OVERSAMPLING` and `USE_JITTER_OVERSAMPLING` switch between simple copy+`jitter` augmentation and SMOTE+`jitter` for minority-class oversampling.
- **Windowing & Jitter Augmentation**
  - `window_size` (length of sliding window) and `step_size` (stride) control the spatial–temporal resolution of trajectory segmentation.
  - `sigma` is the standard deviation of Gaussian noise in frame-level jitter; increasing it can improve robustness to measurement noise.
- **Mapping & Resampling Hyperparameters**
  - *Temperature-scaled Softmax*: parameter `T` controls the smoothness of the remapped distribution (`T>1` yields a flatter distribution; `T<1` yields a sharper one).
  - *Power-law Mapping*: exponent `gamma` amplifies or attenuates original class proportions.
  - *Linear Interpolation*: mixing coefficient `alpha` trades off between the original distribution (`alpha=0`) and uniform (`alpha=1`).
- **SMBO & Meta-Learning Warm-Start**
  - `total_walltime_limit` and `func_eval_time_limit` define the overall budget and per-configuration evaluation time for SMAC.
  - `num_metalearning_cfgs` sets the number of configurations obtained via meta-learning for cold start—too many or too few can slow initial convergence.
  - `n_jobs` and `dask_client` govern parallelism in SMAC's evaluation phase.
- **Ensemble Builder Hyperparameters**
  - `ensemble_size`: number of base models to include in the ensemble.
  - `uncertainty_penalty`: weight of variance penalty during uncertainty-aware greedy selection.
  - `ensemble_nbest` and `max_models_on_disc`: control fraction/number of candidate models and on-disk model retention limit, respectively.
- **Pipeline Search-Space Customization**
  - Through the `include`/`exclude` arguments of `get_configuration_space`, one can whitelist or blacklist specific preprocessors, feature transformations, and classifiers.

– The method `SimpleClassificationPipeline._get_pipeline_steps` can be extended to insert new data-preprocessing, feature-processing, or estimator stages.

8 Conclusions and Future Work

We have presented *AutoMedTS*, an open-source automated machine learning framework tailored for class-imbalanced medical time-series data. By integrating a `sliding-window` segmentation pipeline with multiple resampling strategies (e.g. temperature-scaled Softmax, power-law mapping), *AutoMedTS* significantly enhances minority class detection and overall model generalization. Employing a Bayesian Sequential Model-Based Optimization (`SMBO`) search enhanced by meta-learning warm-starts, our system efficiently converges to high-quality model configurations under strict time and resource constraints. We further reinforce prediction stability and robustness via an uncertainty-aware greedy ensemble selection algorithm.

Extensive experiments on laparoscopic suturing trajectory data demonstrate that *AutoMedTS* outperforms both deep learning baselines and traditional statistical approaches in terms of accuracy, F1-score, and balanced performance across underrepresented action classes. Thanks to its modular design, the framework readily accommodates custom preprocessing steps, feature transformations, and new resampling strategies through user-friendly APIs. Tunable hyperparameters—such as window size (`w`), step size (`s`), exponent (γ), temperature (`T`), and jitter standard deviation (σ)—allow practitioners to balance performance and real-time requirements for diverse clinical scenarios.

Looking forward, we plan to extend *AutoMedTS* with multimodal sensor fusion (e.g. video, force feedback, EMG) to capture richer surgical action representations, and to incorporate online dynamic rebalancing and incremental learning techniques for real-time adaptation to distributional shifts. We also aim to validate the framework in large-scale clinical and simulated environments, and to develop interpretable reports and interactive visualizations to facilitate deployment in intelligent surgical training and assessment systems.

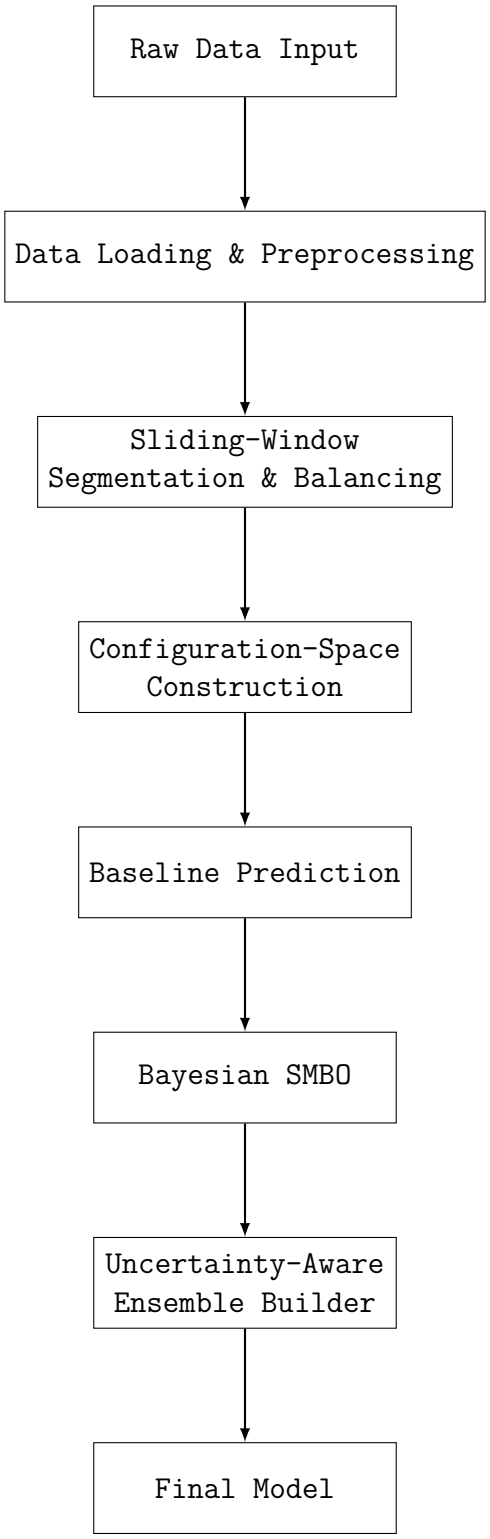

Abb. 2: End-to-end workflow of AutoMedTS

References

[1] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 20–29, 2004.