Process Mining and Management Project

Davide Donà - 264467 Andrea Blushi - 264468

December 2025

Contents

1 Introduction **1**

2 Implementation **1**
2.1 Encoding the Event Log 1
2.2 Training the Decision Tree Classifier 2
2.3 Optimizing the Classifier hyperparameters 2
2.4 Generating Predictions 2
2.5 Extracting Recommendations 2
2.6 Evaluation of Recommendations 4

3 Evaluation **5**
3.1 Hyperparameters 5
3.2 Decision Tree Structure 5
3.2.1 Evaluation Metrics 5
3.3 Recommendation Analysis 6
3.3.1 Recommendations Quality 6

1 Introduction

In modern business process, predicting outcomes and providing recommendations based on historical data is crucial for improving efficiency and decision-making. In this project, using process monitoring and machine learning techniques, we analyze a real-life *Production* log to predict whether a process instance will lead to a positive (fast trace, cycle-time < avg-cycle-time) or negative outcome (slow trace). When a negative outcome is predicted, we generate recommendations that suggest changes in the sequence of activities to steer the process towards a positive result. Then, we evaluate the effectiveness of our recommendation system by comparing the predicted outcomes with the actual outcomes in a test dataset.

2 Implementation

We implemented a recommendation system using a decision tree classifier to analyze production event logs. The implementation was achieved using Python, leveraging libraries such as *PM4Py* for process mining tasks and *scikit-learn* for machine learning functionalities. The execution flow of our implementation resides within the *notebook.ipynb* file, which calls all the designed utility functions.

2.1 Encoding the Event Log

We began by importing the event log file, which is in *XES* format, using the PM4Py library. We then converted this file into a PM4Py event log object suitable for subsequent process mining tasks.

A pruned copy of the event log was created in order to take into consideration only the prefixes of each trace. To do this, each trace in the log was truncated to *prefix_length* events. Since in a real-world scenario the partial ongoing trace is incomplete, we need to train the classifier only on the prefixes of the traces.

Next, from the complete event log, we extracted the unique activity names to serve as columns (features) for the encoding step. To these columns, we also added the trace ID and the ground truth label.

To use this data for training a decision tree classifier, we encoded the pruned event using a *Boolean encoding* scheme. This encoding involved creating binary features for each unique activity name. For each trace in the event log, if an activity was present within the trace, the corresponding feature in that row was set to true (1); otherwise, it was set to false (0). This approach allows to build a feature set with a fixed number of columns, regardless of the varying lengths of the traces. This implies that we don't need to worry about padding the traces to a uniform length. Upon completion, we obtained a DataFrame where each row

represented a trace and the feature columns indicated the presence or absence of specific activities within that trace.

2.2 Training the Decision Tree Classifier

After encoding the pruned event log, a decision tree classifier was trained using the *scikit-learn* library. We split the encoded log into features (by removing the trace identifier and label columns from the original encoding) and labels, which were the ground truth extracted directly from the log. The *XES* file was already partitioned into separate training and testing sets; we used these to train and evaluate the classifier, respectively.

After evaluating the classifier on the test set, we visualized the learned decision rules by plotting the decision tree using built-in *scikit-learn* functionality.

2.3 Optimizing the Classifier hyperparameters

To enhance the performance of our decision tree classifier, we employed the *Hyperopt* library to perform hyperparameter optimization. This tool systematically iterates through various combinations of hyperparameters, such as maximum tree depth, maximum features, and criterion (e.g., Gini or entropy), to identify the configuration that yields the optimal performance. At every iteration, the model was trained and evaluated using the F_1-score as the primary performance metric. Only the configuration that resulted in the highest F_1-score was retained for the final model. The Hyperopt library also allows us to define a search space and to specify the maximum number of iterations to perform.

2.4 Generating Predictions

After training the classifier, we used it to generate predictions on the test set. The predicted labels were obtained by applying the trained model to the feature set derived from the encoded test log.

2.5 Extracting Recommendations

Given the predictions obtained from the decision tree classifier, using the transparent structure of the tree, we can provide recommendations for traces predicted to have a negative outcome. To do this, we need to:

1. Extract the positive paths from the decision tree, which are the paths leading to leaf nodes with a positive outcome.
2. For each prefix trace predicted as negative:
    - Filter the positive paths to find the ones that are compliant with the current trace (no condition along the path is violated by the current trace);

- From the compliant paths, take the one with the highest confidence score;
- The recommended activities are the ones that need to be added to the current trace to follow the selected positive path.

Extracting Positive Paths from the Decision Tree: To extract the positive paths from the decision tree, we traversed the tree structure starting from the root node in a *depth-first* manner. At each node in the path, we record the feature and the boolean condition (true or false) that leads to the next node. When we reach a leaf node, we check if it corresponds to a positive outcome. If it does, we store the entire path taken to reach that leaf, with the associated confidence score, as a positive path.

Algorithm 1 Get Positive Paths

```
function GetPositivePaths(node, current_path, positive_paths)
    if node is leaf then
        if node.label == positive then
            Add (current_path, node.confidence) to positive_paths
        end if
    else
        Append (node.feature, False) to current_path
        Call GetPositivePaths(node.left, current_path, positive_paths)
        Remove last element from current_path
        Append (node.feature, True) to current_path
        Call GetPositivePaths(node.right, current_path, positive_paths)
        Remove last element from current_path
    end if
end function
```

Filter compliant paths: After extracting all positive paths from the decision tree, for each prefix trace predicted as negative, we filtered only the *compliant* paths. A path is considered *compliant* with a trace if none of the conditions along the path are violated by the current prefix trace. Among all the compliant paths, we selected the one with the highest confidence score. If two or more paths had the same confidence score, we selected the shortest one.

Generate Recommendations: Once we identified the most suitable positive compliant path, we generated recommendations by determining the activities that needed to be added to the current trace to follow the selected path. For each prefix seen in the test set, its recommendation is defined as follows:

- $\emptyset$ if the prefix was predicted as positive or if no compliant positive path was found;
- The set of activities corresponding to the conditions in the selected positive path that were not satisfied by the current prefix trace otherwise.

2.6 Evaluation of Recommendations

To evaluate the quality of the recommendations generated, we compared them against the actual traces in the test set, where the encoded test set served as the ground truth for comparison.
For each complete trace in the test set, we first checked if the trace's prefix matched the specific conditions (the features present) leading to a particular recommendation as defined by the decision tree classifier. Only if a prefix match occurred, we then proceeded to check if the recommended activity was actually performed in the subsequent step of the actual trace following that prefix. This recommendation outcome was then compared with the ground truth outcome of the trace to determine the predictive accuracy of the recommendation.
With those checks we can create an approximated confusion matrix, where:

- True Positives: The recommended activity was followed in the actual trace, and the ground truth outcome is positive.
- False Positives: The recommended activity was not followed in the actual trace, but the ground truth outcome is positive.
- True Negatives: The recommended activity was not followed in the actual trace, and the ground truth outcome is negative.
- False Negatives: The recommended activity was followed in the actual trace, but the ground truth outcome is negative.

Starting from this confusion matrix, we computed standard evaluation metrics such as accuracy, precision, recall, and F1-score to assess the effectiveness of the recommendations provided by our system.

3 Evaluation

3.1 Hyperparameters

3.2 Decision Tree Structure

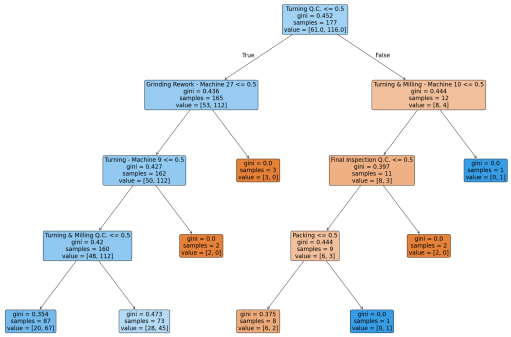

Figure 1: Visualization of the trained decision tree classifier. Each node represents a decision based on the presence or absence of specific activities, leading to recommendations at the leaf nodes.

3.2.1 Evaluation Metrics

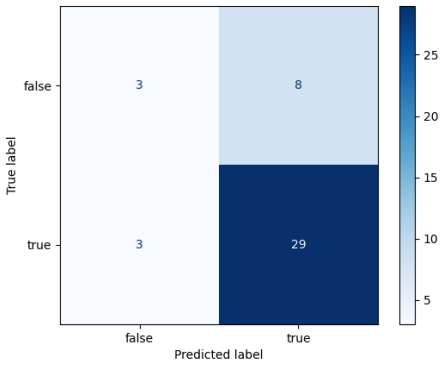

Figure 2: Confusion matrix for the decision tree classifier's performance on the test set.

Metric	Value
Accuracy	85%
Precision	82%
Recall	83%
F1-Score	85%

Table 1: Evaluation metrics for the decision tree classifier on the test set.

3.3 Recommendation Analysis

3.3.1 Recommendations Quality

Metric	Value
Accuracy	80%
Precision	78%
Recall	79%
F1-Score	80%

Table 2: Evaluation metrics for the recommendation system