Project 3: Predictions with intercase feature encoding

by Nicola Mores, Marco Roccon

presented for the course
Process Mining
at
University of Trento

- Nicola Mores, nicola.mores@studenti.unitn.it, 248490
- Marco Roccon, marco.roccon@studenti.unitn.it, 248491

Introduction:

The project focuses on Predictive Process Mining, specifically on the encoding of a provided event log using a well-known technique called *simple index encoding*. This approach is further enhanced by incorporating intercase information to enrich the feature set. The encoding obtained in this process is then used to train various different decision tree models enabling a comparative analysis of the impact of different intercase features. The ultimate goal is to identify the combination of features that achieves the highest accuracy on a predefined test set.

Dataset:

The dataset provided for this project consists of two .xes logs of different sizes. The larger log, *Production_avg_dur_testing_0-80.xes*, is used as the training set, while the smaller log, Production_avg_dur_testing_80-100.xes, serves as the test set. These datasets contain real-world data from a manufacturing process, where each trace is classified based on its execution time. Traces are labelled as "*fast*" if their cycle time is less than the average cycle time and as "*slow*" otherwise. This classification is encoded in the dataset as a trace-level boolean variable called *label*, where *True* indicates a fast trace and *False* indicates a slow trace.

Code:

To implement the functionalities required for this project, the code has been organized into multiple separate files to ensure clarity and maintainability:

- ***encoding.py***: This file contains all the functions necessary for encoding the event logs and extracting the intercase features specified in the project description. These include functions for simple index encoding, counting concurrent cases, calculating average durations of concurrent cases, and other intercase feature computations.
- ***prediction_model.py***: This file is dedicated to the predictive aspect of the project. It includes the implementation of the training, testing, and optimization processes for the decision tree models used to make predictions based on the encoded data.
- ***evaluate_results.py***: This file facilitates the evaluation of the predictive model's performance. It automatically scans the .log file generated by *prediction_model.py* and computes average metrics (e.g., Accuracy, Precision) from the results, providing a concise summary of the model's effectiveness.

Below is a brief description of every function implemented in the two files, specifying its purpose and how it works:

encoding.py

- **import_xes(file_path)**: This function takes as input the file path of a .xes dataset and imports it using PM4Py, a Python library used for process mining. It converts the log data frame into a structured event log format compatible with further processing.
- **get_label_encoder(log)**: This function creates a label encoder for mapping activity names, iterating through all events in the cases of the event log, to unique numeric values. The resulting mapping is returned as a dictionary, with activity names as keys and their respective numeric values as values.
- **print_all_log(log)**: This utility function provides a printout of the entire log, showing for each case its ID, name, and the timestamp of the first and last events. It then iterates through the events within each case, printing the event ID and its activity name.
- **encode_case_simple_index(case, prefix_length, label_encoder)**: This function encodes a trace using the simple index encoding method. It takes as input a trace, a desired prefix length and a precomputed label encoder mapping activity names to numeric values. Each event in the trace is converted to its numeric representation using the label encoder. If a trace is shorter than the prefix_lenght, zero-padding is applied. The resulting trace is truncated to the prefix_lenght, and the trace's label is appended.
- **decode_case_simple_index(encoded_case, prefix_length, label_encoder):** This function decodes a trace encoded using the simple index encoding method. It takes as input an encoded trace, the prefix length and the label encoder used during the encoding phase. Each event in the trace is converted from its numeric representation to its original name, using the decoder obtained reversing the encoder taken as input. Eventual padding is removed according to the prefix length.
- **count_concurrent_cases(case, log)**: This function takes as input a case and the complete event log and calculates the number of cases in the event log that were

executed concurrently with the specified trace. It identifies the start and end timestamps of the input trace and checks how many other traces overlap within this time range. This process leverages PM4Py's function '*pm4py.filter_time_rang*' with the 'traces_intersecting' mode for retrieving time intervals and analyzing concurrency. It returns the number of traces in the filtered log.

- **count_avg_duration(case, log)**: This function takes as input a case and the complete event log and computes the average duration (in seconds) of the cases in the event log that were executed concurrently with the specified trace. As in the previous function, it uses PM4Py's filtering function to extract concurrent cases. Iterating over these cases, the total duration of their completion times is calculated, and the average is returned.
- **count_avg_resources_concurrent_cases(case, log)**: This function takes as input a case and the complete event log and calculates the average number of unique resources involved in the cases that overlap with the specified trace. As in the previous function, it uses PM4Py's filtering function to extract concurrent cases. For each overlapping trace, the function iterates through its events to extract unique resources, using a counter. The average number of unique resources is computed dividing the counter by the total number of traces in the log, and the result is rounded to two decimal places before being returned.
- **count_avg_overlapping_duration_concurrent_cases(case, log)**: This function takes as input a case and the complete event log and computes the average overlapping duration (in seconds) between the input trace and all other traces that are executed concurrently. As in the previous function, it uses PM4Py's filtering function to extract concurrent cases. For each overlapping trace, the function calculates the eventual overlap duration. If overlaps exist, their durations are averaged, rounded to two decimal places, and returned. If no overlaps are found, the function returns 0.
- **count_avg_concurrent_cases_per_event(case, log)**: This function takes as input a case and the complete event log and computes the average number of concurrent cases for each event within the input trace. For each event, it determines the event's duration based on its start timestamp and activity_duration. Using this time range, it retrieves all traces that overlap using PM4Py's '*pm4py.filter_time_range*' function. The number of concurrent traces is recorded for each event and stored in a list. The function computes the average of these counts, rounds it to two decimal places, and returns the result. If no concurrent cases are found for any event, the function returns 0.
- **simple_index_encode(log, prefix_length, label_encoder, conc_cases=False, avg_dur=False, my_int1=False, my_int2=False, my_int3=False)**: This function encodes an entire log using the Simple Index Encoding method, with optional intercase features. It takes as input the event log to be encoded, a fixed length for padding or truncating each trace, and several optional flags that specify which intercase features to include in the encoding. The function processes each case in the event log by:
    1. Encoding it using the Simple Index Encoding method.
    2. Appending the enabled intercase features to the encoding.
    3. Adding the classification label

  The output is a pandas DataFrame containing the encoded traces, allowing for easy creation of various combinations of intercase features.

prediction_model.py:

- **train_dt(model, encoded_data: pd.DataFrame)**: This function trains a decision tree model using the provided encoded dataset. It takes as input the model instance and the encoded data (in DataFrame format), where the feature columns are used as predictors, and the last column contains the target labels. The function fits the model on this data and returns the trained model.
- **predict(model:tree.DecisionTreeClassifier, encoded_data:pd.DataFrame)**: This function generates predictions for a dataset using a given trained decision tree model. It takes as input the trained model and an encoded dataset. The function generates boolean predictions and returns them as an array.
- **get_metrics(predictions, gold_standard)**: This function calculates and returns evaluation metrics for a set of given predictions compared to the ground truth (gold standard). It computes and returns accuracy, precision, recall, and F1-score using scikit-learn's metrics functions.
- **model_optimization(encoded_data:pd.DataFrame, max_evals=1000)**: This function performs hyperparameter optimization for a decision tree classifier using the hyperopt library. The search space includes parameters such as *max_depth*, *max_features*, *min_samples_split*, *min_samples_leaf*, *max_leaf_nodes*, and the splitting criterion (*gini*, *entropy*, or *log_loss*). The optimization process evaluates up to max_evals trials, identifying the parameter combination that minimizes the loss function. The function returns the best-found parameters.

Results and Observations:

The model was tested through many iterations to ensure the reliability and consistency of its results. For each iteration, key metrics such as Accuracy and Precision were calculated to provide an evaluation of the performance. During the testing phase, the model was assessed using all possible combinations of intercase features to determine their impact on its predictive capabilities.

The obtained results consistently demonstrated that enhancing the simple index encoding with intercase features had a significant effect on the model's performance. Below is a table summarizing the average metric values obtained across 25 iterations:

The table is organized into two parts:

- Left side: Displays the combinations of intercase features. Each column is labeled with the name of a specific feature, while the rows indicate which features are active for each combination using True/False values.
- Right side: Contains the performance metrics (e.g., Accuracy, Precision) for each combination. Improvements and declines in the metrics are visually highlighted using a color scale from red to green.

Concurrent Cases	Average Duration	Average Resources	Overlapping Duration	Concurrent Cases per Event		Accuracy	Precision	Recall	F1-Score
F	F	F	F	F		0,6546	0,5929	0,6122	0,5940
F	F	F	F	T		0,7769	0,7368	0,6590	0,6692
F	F	F	T	F		0,9070	0,8778	0,8778	0,8778
F	F	F	T	T		0,8984	0,8638	0,8831	0,8708
F	F	T	F	F		0,9070	0,8778	0,8778	0,8778
F	F	T	F	T		0,7821	0,7710	0,8536	0,7651
F	F	T	T	F		0,8837	0,8545	0,8324	0,8425
F	F	T	T	T		0,8984	0,8638	0,8831	0,8708
F	T	F	F	F		0,7648	0,6897	0,6840	0,6863
F	T	F	F	T		0,6762	0,4606	0,4863	0,4639
F	T	F	T	F		0,9509	0,9493	0,9217	0,9339
F	T	F	T	T		0,9328	0,9144	0,9096	0,9114
F	T	T	F	F		0,9044	0,8684	0,8894	0,8780
F	T	T	F	T		0,5564	0,6413	0,6677	0,5517
F	T	T	T	F		0,9535	0,9487	0,9301	0,9381
F	T	T	T	T		0,9285	0,9049	0,9144	0,9081
T	F	F	F	F		0,7442	0,3721	0,5000	0,4267
T	F	F	F	T		0,7674	0,8810	0,5455	0,5158
T	F	F	T	F		0,8458	0,8131	0,7539	0,7704
T	F	F	T	T		0,8570	0,8203	0,7957	0,8001
T	F	T	F	F		0,8837	0,8804	0,8026	0,8311
T	F	T	F	T		0,6279	0,7037	0,7500	0,6228
T	F	T	T	F		0,8217	0,7384	0,7046	0,7086
T	F	T	T	T		0,8346	0,7707	0,7662	0,7608
T	T	F	F	F		0,7622	0,7494	0,5354	0,4955
T	T	F	F	T		0,7674	0,8810	0,5455	0,5158
T	T	F	T	F		0,8880	0,8937	0,8043	0,8303
T	T	F	T	T		0,8889	0,8840	0,8270	0,8400
T	T	T	F	F		0,8794	0,8624	0,8096	0,8302
T	T	T	F	T		0,6408	0,7080	0,7587	0,6347
T	T	T	T	F		0,8527	0,8282	0,7375	0,7541
T	T	T	T	T		0,8803	0,8219	0,8047	0,8058

The table shows how in many cases, the model's performance improved with specific combinations of intercase features, such as {Average Duration, Overlapping Duration, Concurrent Cases per Event} or {Average Duration, Average Resources, Overlapping Duration, Concurrent Cases per Event}. However, in a minority of cases, the addition of these features introduced noise, resulting in a slight decline in performance metrics. It is also worth noting that including all intercase features did not consistently yield the best results. This variability in performance across different feature combinations highlights the importance of feature selection in predictive modeling.
In conclusion, the use of intercase features significantly enhanced the model's overall performance compared to the baseline configuration, with particular improvements observed with feature combinations such as {Average Duration, Overlapping Duration} and {Average Duration, Average Resources, Overlapping Duration}, demonstrating their value in improving predictive accuracy.

Installation and Execution Instructions:

To run our code and replicate the results, follow the steps below. The implementation is available in our GitHub Repository.

Step 1: Install Required Libraries

Before running the code, install all the necessary libraries using the provided *requirements.txt* file.

```
Unset
pip install -r requirements.txt
```

Step 2: Run the Prediction Model

To train and test the prediction model using the dataset, execute the following command:

```
Unset
python ./prediction_model.py
```

This will generate the results for a single train-test iteration, which will be logged in a results file *results.log*. Note that this process will take approximately 3 minutes to complete, depending on your system configuration.

Step 3: Evaluate the Metrics

After generating the prediction results, you can evaluate the overall metrics by running:

```
Unset
python ./evaluate_results.py
```

This script reads the *results.log* file created during the previous step, processes the data, and outputs the averaged metrics in a concise format for easier interpretation.