Projekt Systementwicklung: Process Data Mining

Authors: Max Bohling, Johann Ehlers
Supervisor: Prof. Dr. Oliver Skroch

Table of contents

Jupyter Notebook 1: data conversion from csv to xes

Part one of our task was to convert a dataset, provided by Prof. Skroch, formatted as csv, to the XML- based XES ‘eXtensible Event Stream’ format.
During the conversion, the dataset is enriched by additional information, provided in separate csv files.
This is realized within the Jupyter Notebook named *pse_csv_to_xes.ipynb*, using two additional external Python modules (*src/csv_filehandlingmodule.py* and *src/csv_processclasses.py*).
The module *csv_filehandlingmodule.py* provides general file opening and closing operations for better Notebook readability, as well as the functions responsible for reading csv data into the Notebook’s data structures. The main event data is immediately enriched while being read.
The second module, *csv_processclasses.py*, provides a data structure for events, and a helper function to convert a single event into an XES-compliant XML structure.

The Notebook itself requires no user input except running the cells in top-down order. This gives it a mere illustrating functionality, since the exact data structure of an XES file heavily depends on the input data and available fields. It is therefore sadly impossible (or at least would be unreasonably time-consuming) to write a csv to XES converter which works in all cases. However, it gives a basic template for how csv to XES conversions work, and can be manually adapted to work with different data. To achieve this, changes are necessary in the Notebook as well as the Python modules. To this extent we commented extensively. A

consultation of the XES standard definition at XES – Standard Definition is necessary to grasp the underlying XES structure and required meta-information.

After executing the *pse_csv_to_xes.ipynb* Notebook, the converted data can be found in *(Notebook directory)/data/xes/events.xes*, unless specified otherwise. This file is also used as the default import file for the second notebook. Therefore this Notebook needs to be run before the second Notebook for it to work with default values.

Jupyter Notebook 2: dynamic data mining pipeline

1. Overview

Part two of our task was to implement a Process Mining meta-pipeline, meaning a Jupyter Notebook offering enhanced functionality for importing and exporting XES data, saving and restoring data checkpoints for comfortably reversing data operations as well as the ability to define and apply user-written functions. These features were implemented using several Python modules and IPython Widgets.
Since all functionality resides within a Notebook, the cells containing these functions must be run by the user before becoming available. To this end, we created a template Notebook (*pse_dynamic_pipeline_template.ipynb*) containing all setup function calls within the first code cell and detailed instructions within markdown cells. Further, it is also important for the functionality described in the following sections that the Notebook is trusted by the user. If this is not the case there is a risk of malfunctions.

2. Initialization, widget execution and data usage

To access all extended functionality of the Notebook as intended, several steps need to be taken by the user. Firstly, unless the first cell is executed, the Notebook will not offer any additional features (“Initialization”). Secondly, to access IPython Widget functionality, the appropriate *widget_handler* functions must be included in the cells where the widget is desired. Thirdly, all manual data operations should be applied to the DataFrame “data”, since all functionality (e.g. exporting the data) is always implicitly applied to it.

3. XES Import/Export

The Notebook will import a default XES file on execution of the first code cell. This default is changeable by modifying the *pickle_dir* (directory in which all pickle files are stored), *pickle_unaltered_fil*e (name for the file in which the unaltered import data is stored), *xes_dir* (directory in which all XES files are stored) and *xes_import_file* (name of the XES file which will be imported at initialization) variables within the first code cell.
However, the Notebook will automatically prefer to load the data from a Pickle file, should a file with the *pickle_unaltered_file* name exist. To force an import from XES, you must delete this Pickle file.

The imported XES data will automatically be loaded into a Pandas DataFrame named “data”. If the XES data contains multiple traces, the DataFrame will be Multi-Indexed with all events indexed by their trace attributes.

The export functionality is realised by using a widget. Whenever the *widget_handler.display_data_widgets(...)* function is executed, three widgets will be appended to the current cell. The third, consisting of the label “File name:”, a textbox and a button labeled “Export as .xes file”, realises the export functionality. After entering the desired filename and selecting the button, the Notebook will automatically export the current DataFrame (“data”) into a properly formatted XES file.
However, due to the restrictive nature of XES, and the complicated nature of transforming a two-dimensional array into a hierarchical XML structure, some restrictions apply: The DataFrame must either be not Multi-Indexed, in which case the XES file will contain one “trace” node and as many “event” nodes as there are DataFrame rows, or if it is Multi-Indexed, the Index has to contain the row “concept:instance”, which is the trace ID by which the export function groups “event” nodes into “trace” nodes. All further indexes within the Multi-Index will be saved as attributes of the appropriate “trace” node. Please be aware that this operation can take several minutes.

4. Saving/Loading of intermediate DataFrames (“checkpoints”)

On execution of the *widget_handler.display_data_widgets(...)* function, the two upper widgets, consisting of a “File name:” label, a textbox and a button labeled “Save current state” and a “File name:” label, a dropdown select and a button labeled “Load state” respectively, provide functionality to save the current state of the DataFrame (“data”) or replace it with a previously saved state. The saved states are persistent over Notebook restarts, as long as the *pickle_dir* remains the same. This also allows to save a current state for presentation purposes or taking a break, and resume later where you left off without having to keep the IPython kernel running.
To save the current state, simply enter a desired name into the appropriate textbox and click the button. This operation is fairly quick.
After executing the *widget_handler.display_data_widgets(...)* function again, the saved state will appear in the dropdown select, and can be loaded by clicking the “Load state” button. This operation is also fairly quick.
The dropdown select is populated with all available files in the *pickle_dir* directory upon execution of the first cell. It is assumed that all files within this directory are Pickle binary files. Therefore, please do not put any other files into this directory.

5. Adding user-defined functions

To allow the user to easily write and apply his own functions to the DataFrame “data”, we offer a widget for writing, managing and applying these functions. This widget will appear whenever the *widget_handler.display_function_widgets(...)* function is executed. It firstly consists of a textbox with label “Command:”, in which the function name should be entered, a large textbox with label “Function:”, in which the function

commands should be entered, and a “Create function” button, which saves the defined function. Secondly, it consists of a dropdown select labeled “Command”, which contains all currently available functions, as well as buttons to run, show or delete the currently selected function.
To create a function, enter the function name (e.g. “*orderbytrace*”) and the function commands, ***without*** the usual “*def orderbytrace(self):*”. It should look like this:
self.data.set_index(['concept:instance'], inplace=True)
self.data.sort_index(inplace=True)
Please note that any references to “data” need to be prepended by “self”, and DataFrame functions should run “inplace”. To finish creating the function, simply click the “Create function” button.
After rerunning the *widget_handler.display_function_widgets(...)* function, the newly created function will appear in the dropdown select, and be able to be run, shown or deleted.
Functions created this way will be persisted in the *(Notebook directory)/data/functions/ directory*, and will be available after a kernel restart.

6. Dynamic pipeline steps

Once done with the actions above, in the actual step of the pipeline, using the “Next step” button will prepare the pipeline for its next execution. The button is linked to the module *cell_generator.py*, which will generate two new cells at the end of the Notebook. On executing those cells, they generate all of the widgets described above. The first created cell contains all widgets needed for the creation of user-defined functions. The second generated cell is taking care of the file export and the management of intermediate Dataframes. For easier understanding, the Notebook contains a description and guide on what happens after the use of the “Next step” button. Please note that the widgets will only appear after executing the cell containing their calling function.
The dynamic creation of cells is done using Javascript functions in the Notebook, allowing the modification of the Notebook structure and its nodes content. Go to jupyter/notebook: Jupyter Interactive Notebook for further reading about the JavaScript functions and better understanding how they act in the cell_generator.py module.