PROCESS MINING

Lab Session 6

Hands-on Session on Pm4py

Content

Laboratory Session: Introduction to PM4PY

This laboratory session is intended to let students familiarize with the python library *Pm4py* and the logs in *xes format*.

Installation of PM4Py

PM4Py[1] is a process mining package for Python that implements the latest, most useful, and extensively tested methods of process mining.

In order to install PM4Py and its dependencies, the command in *Figure 1* should be executed.

```
pip install pm4py
```

Figure 1. Install PM4py library

For all information on installing PM4Py on different operating systems, see on the official page[2].

Step 1: Loading of XES Files with PM4Py and Visualization of Events and Traces

For this lab, we use the real-life event log ***purchasingExample.xes*** that refers to executions of instances of a purchase-order process, namely, to deal with orders of goods to providers. This is the same event log that was used in the first laboratory session with Disco, but with a small difference: the events related to the completion of the activities (i.e. those with value *Complete* for attribute *lifecycle:transition*) also contains the timestamp of the corresponding "start event"), stored in attribute *StartTimestamp*. This is a requirement for Step 3 below to compute the case arrival rate.

Firstly it is necessary to import the library ***import pm4py***, then the function ***xes_importer.apply()*** imports the log in XES format. At the end of the loading bar (*Figure 2*), you can observe the total number of cases for this log, which is equal to 608, see the red circle.

In this first step, we start with importing the log and exploring his attributes.

```
import pm4py
from pm4py.objects.log.importer.xes import importer as xes_importer
log = xes_importer.apply('purchasingExample.xes')
```

```
parsing log, completed traces :: 100% 608/608 [00:00<00:00, 857.02it/s]
```

Figure 2. Import the event log

[1] The PM4Py documentation is available on https://pm4py.fit.fraunhofer.de/docs
[2] Installation guide https://pm4py.fit.fraunhofer.de/install-page#item-1-2

It is possible to access every trace and event by the index. Event logs are stored as an extension of the Python list data structure. To access a trace in the log, it is enough to provide its index in the event log as shown in Figure 3.

```
log[0] #first trace in the event log
```

```
{'attributes': {'concept:name': '1'}, 'events': [{'Case': 1, 'concept:name': 'Create Purchase Requisition', 'org:r
esource': 'Kim Passa', 'lifecycle:transition': 'start', 'time:timestamp': datetime.datetime(2011, 1, 1, 0, 0, tzin
fo=datetime.timezone.utc), 'StartTimestamp': 'NaT', '@@index': 0}, '..', {'Case': 1, 'concept:name': 'Pay Invoic
e', 'org:resource': 'Pedro Alvares', 'lifecycle:transition': 'complete', 'time:timestamp': datetime.datetime(2011,
1, 4, 15, 31, tzinfo=datetime.timezone.utc), 'StartTimestamp': datetime.datetime(2011, 1, 4, 15, 22, tzinfo=dateti
me.timezone.utc), '@@index': 33}]}
```

Figure 3. Exploring a trace

Figure 4 shows how to calculate the total number of events within **trace 2** and attributes in the event log.

```
#number of event in trace 1
print(len(log[0]))
```

```
34
```

Figure 4. Get the number of events in a trace

As mentioned, an event log is a list of traces, each of which is a list of event. Therefore, to access the **events inside a trace** we have to use a second index; for instance, `log[0][0]` returns the first event of the first trace, `log[0][1]` returns the second of the first trace, or `log[0][2]` would return the third. Events are extension of the standard Python dictionary, namely each contains pairs attribute-value. For instance, if we visualized the third event of the first trace, we would see the attributes of that events and the associated values (see *Figure 5*)


```
log[0][0] #first event of the first trace in the event log
```

```
{'Case': 1, 'concept:name': 'Create Purchase Requisition', 'org:resource': 'Kim Passa', 'lifecycle:transition': 's
tart', 'time:timestamp': datetime.datetime(2011, 1, 1, 0, 0, tzinfo=datetime.timezone.utc), 'StartTimestamp': 'Na
T', '@@index': 0}
```

Figure 5. Exploring an event

Let's check the event attributes using `pm4py.get_event_attributes(log)` as given in Figure 6.

```
#to see the event attributes:
pm4py.get_event_attributes(log)
```

```
['StartTimestamp',
 'Case',
 'concept:name',
 'time:timestamp',
 '@@index',
 'org:resource']
```

Figure 6. Get event attributes

You can also call attributes for any events in the traces, as shown in Figure 7.

```
log[0][0]['concept:name'] #Activiy name of third event of the first trace in the event log
```

```
'Create Purchase Requisition'
```

Figure 7. Exploring an attribute

In order to apply the different steps of this laboratory session, we first need to filter out the start timestamp, as some the techniques below might consider the start and completion events as two different occurrences of activity executions. As a consequence, this may yield erroneous computations. Figure 8 shows how to quickly filter out the start events using the build-in function ***pm4py.filter_event_attribute_values***. If we aim to verify the successful application of filtering, we can, e.g., compare the first event of the first trace for the log before filtering (see *Figure 5*) and the first event of the first trace of the log after filtering. Before filtering, it referred to a start event; after filtering, it is a completion event (cf. values of attribute *lifecycle:transition)* Also, note the presence of the special *StartTimestamp* attribute.

```
#Filter start events
log = pm4py.filter_event_attribute_values(log, "lifecycle:transition", ["complete"], level="event")
log[0][0]
```

```
{'Case': 1, 'concept:name': 'Create Purchase Requisition', 'org:resource': 'Kim Passa', 'lifecycle:transition': 'c
omplete', 'time:timestamp': datetime.datetime(2011, 1, 1, 0, 37, tzinfo=datetime.timezone.utc), 'StartTimestamp':
datetime.datetime(2011, 1, 1, 0, 0, tzinfo=datetime.timezone.utc), '@@index': 1}
```

Figure 8. Exploring an attribute

The function in Figure 9 takes as input a trace's index and prints the attributes for each event inside the trace. The structure of an event is a dictionary, so it is sufficient to use the corresponding key to get its attributes.

```
def attribute_trace(trace_id):
    row = 0
    for event in log[trace_id]:
        row = row + 1
        print("Row_",row, " ", event['Case'], " ", event['concept:name'], " ", event['lifecycle:transition'], event['StartTimestamp'],
              " ", event['time:timestamp'], " ", event['org:resource'])
```

```
attribute_trace(0) # check Row_1 and with filtered log[0][0]
```

```
Row_ 1   1   Create Purchase Requisition   complete 2011-01-01 00:00:00+00:00   2011-01-01 00:37:00+00:00   Kim Passa
Row_ 2   1   Create Request for Quotation   complete 2011-01-01 05:37:00+00:00   2011-01-01 05:45:00+00:00   Kim Passa
Row_ 3   1   Analyze Request for Quotation   complete 2011-01-01 06:41:00+00:00   2011-01-01 06:55:00+00:00   Karel de Groot
Row_ 4   1   Send Request for Quotation to Supplier   complete 2011-01-01 11:43:00+00:00   2011-01-01 12:09:00+00:00   Karel de Groot
Row_ 5   1   Create Quotation comparison Map   complete 2011-01-01 12:32:00+00:00   2011-01-01 16:03:00+00:00   Magdalena Predutta
Row_ 6   1   Analyze Quotation Comparison Map   complete 2011-01-01 22:44:00+00:00   2011-01-01 23:13:00+00:00   Immanuel Karagianni
Row_ 7   1   Choose best option   complete 2011-01-01 23:13:00+00:00   2011-01-01 23:13:00+00:00   Tesca Lobes
Row_ 8   1   Settle Conditions With Supplier   complete 2011-01-02 01:22:00+00:00   2011-01-02 09:20:00+00:00   Francois de Perrier
Row_ 9   1   Create Purchase Order   complete 2011-01-02 09:58:00+00:00   2011-01-02 10:10:00+00:00   Karel de Groot
Row_ 10   1   Confirm Purchase Order   complete 2011-01-02 14:09:00+00:00   2011-01-02 14:43:00+00:00   Sean Manney
Row_ 11   1   Deliver Goods Services   complete 2011-01-02 20:49:00+00:00   2011-01-03 03:37:00+00:00   Sean Manney
Row_ 12   1   Release Purchase Order   complete 2011-01-03 11:20:00+00:00   2011-01-03 11:21:00+00:00   Elvira Lores
Row_ 13   1   Approve Purchase Order for payment   complete 2011-01-03 19:09:00+00:00   2011-01-03 19:10:00+00:00   Karel de Groot
Row_ 14   1   Send Invoice   complete 2011-01-04 00:54:00+00:00   2011-01-04 00:54:00+00:00   Kiu Kan
Row_ 15   1   Release Supplier's Invoice   complete 2011-01-04 15:08:00+00:00   2011-01-04 15:13:00+00:00   Karalda Nimwada
Row_ 16   1   Authorize Supplier's Invoice payment   complete 2011-01-04 15:13:00+00:00   2011-01-04 15:13:00+00:00   Karalda Nimwada
Row_ 17   1   Pay Invoice   complete 2011-01-04 15:22:00+00:00   2011-01-04 15:31:00+00:00   Pedro Alvares
```

Figure 9. Finding the attributes of the events

Step 2: Inspection of the Process' Start and End Activities

It is also possible to compute the start and end activities of the traces in the event log with the functions `get_start_activities()` and `get_end_activities()` respectively, as shown in Figure 7. We can observe that each trace starts with the activity *Create Purchase Requisition* while the process can terminate with three different activities: *Pay Invoice, Analyze Request for Quotation and Analyse Purchase Requisition.* The number after the activity name indicates the frequency of occurrence (e.g. 413 ends with *Pay Invoice*).

```
start_act = pm4py.get_start_activities(log)
print(start_act)
```

```
{'Create Purchase Requisition': 608}
```

```
end_act = pm4py.get_end_activities(log)
print(end_act)
```

```
{'Pay Invoice': 413, 'Analyze Request for Quotation': 131, 'Analyze Purchase Requisition': 64}
```

```
#to write plural or singular expressions considering the number of activities
if len(start_act)==1 and len(end_act)==1:
    print("Start activity: {}\nEnd activity: {}".format(start_act, end_act))
elif len(start_act)==1 and len(end_act)>1:
    print("Start activity: {}\nEnd activities: {}".format(start_act, end_act))
elif len(start_act)>1 and len(end_act)>1:
    print("Start activities: {}\nEnd activities: {}".format(start_act, end_act))
```

```
Start activity: {'Create Purchase Requisition': 608}
End activities: {'Pay Invoice': 413, 'Analyze Request for Quotation': 131, 'Analyze Purchase Requisition': 64}
```

Figure 10. Finding the start and end activities of the traces

Step 3: Discovery Information about the Time Perspective

PM4Py enables retrieving information about the time perspective that was not available in ProM. For instance, given an event log, it is possible to retrieve the case arrival ratio, that is the average distance between the arrival of two consecutive cases in the log. Since this log has the *Start Timestamp* and the *time:timestamp*, we have to specify what we want to use to compute the arrival rate. Since we have the start timestamp, we can better compute the arrival rate[3], so we set the key to case arrival time parameters as *Start Timestamp* (Figure 11). Then we apply the function `case_arrival.get_case_arrival_avg()` on the log which returns the case arrival in seconds. This log shows that the arrival rate is 40604 seconds, namely every 11 hours.

```
case_arrival_ratio = pm4py.get_case_arrival_average(log)
case_arrival_ratio
```

```
40604.97528830313
```

Figure 11. Case arrival rate

[3] Note that, if we only had the complete timestamp, the arrival rate would only be approximated unless the first activity had a constant duration. Indeed, the arrival rate is the rate of the conclusion of the first process activity, which would coincide with the actual arrival rate only if the first process activity has the same duration in all process executions.

The computation above assumes that cases arrive at any time of the day. Given the nature of this process, this is likely to be the case. If the assumption does not hold, the last case of one day and the first case of the day after are largely spaced out in time. This introduces several outliers, and the arrival rate estimate will therefore not be accurate. In order to obtain a better estimate, i.e. not considering the time between two arrivals on different days, we define a function to compute the daily case arrival rate as in Figure 12. With this other approach the arrival rate is 17210 seconds, namely every 4 hours.

```
#Daily case arrivals
import numpy as np

case_arrival = []
for i in range(1, len(log)):
    arrive1 = log[i-1][0]['StartTimestamp']
    arrive2 = log[i][0]['StartTimestamp']
    if arrive1.date() == arrive2.date():
        case_arrival.append(arrive2-arrive1)
np.mean(case_arrival)
```

```
datetime.timedelta(seconds=17210, microseconds=852018)
```

Figure 12. Daily case arrival rate

PM4py allows us to easy compute the activity duration. Figure 13 shows how to use the PM4Py function ***soj_time_get.apply()*** to compute, the average duration for each *activity a*. This is obtained by computing the average value of the difference between the completion and start timestamp for each execution of *activity a*. This function requires that every event refers to the completion of the activity and the start timestamp is also provided as an additional information of the completion. The invocation of this function requires to provide which attribute corresponds the completion timestamp (shown with ***TIMESTAMP_KEY*** parameter), which is the usual *time:timestamp*, and which attribute stores the information about the start timestamp (shown with ***START_TIMESTAMP_KEY*** parameter).


```
#Case sensitive!!!
#apply function requires two inputs: i)log name ii)parameters refering start and end time for activity duration
#Parameters can be ACTIVITY_KEY, START_TIMESTAMP_KEY, TIMESTAMP_KEY, AGGREGATION_MEASURE (default: "mean"),
# BUSINESS_HOURS (default: False), WORKTIMING (default: [7, 17]), WEEKENDS (default: [6, 7])

import pm4py.statistics.sojourn_time.log.get as soj_time_get
soj_time = soj_time_get.apply(log, parameters={soj_time_get.Parameters.TIMESTAMP_KEY: "time:timestamp",
                                               soj_time_get.Parameters.START_TIMESTAMP_KEY: "StartTimestamp"})
print(soj_time)
```

```
{'Create Purchase Requisition': 1843.0263157894738, 'Create Request for Quotation': 321.8382352941176, 'Analyze Re
quest for Quotation': 1382.0054200542006, 'Send Request for Quotation to Supplier': 1415.7384987893463, 'Create Qu
otation comparison Map': 12155.01210653753, 'Analyze Quotation Comparison Map': 1209.2978208232446, 'Choose best o
ption': 0.0, 'Settle Conditions With Supplier': 32389.10411622276, 'Create Purchase Order': 573.1234866828087, 'Co
nfirm Purchase Order': 1188.5230024213074, 'Deliver Goods Services': 91075.49636803874, 'Release Purchase Order':
60.0, 'Approve Purchase Order for payment': 60.0, 'Send Invoice': 0.0, "Release Supplier's Invoice": 269.032258064
51616, "Authorize Supplier's Invoice payment": 0.0, 'Pay Invoice': 561.7917675544794, 'Settle Dispute With Supplie
r': 9221.32075471698, 'Analyze Purchase Requisition': 394.869109947644, 'Amend Request for Quotation': 639.4316163
410302, 'Amend Purchase Requisition': 1641.8181818181818}
```

Figure 13. Computing the average processing time of tasks

Step 4: Discover the Roles of Resource (Resource Perspective)

PM4Py implements an elaborated algorithm to discover roles that is more advanced than the algorithm that was introduced during the lectures. The algorithm introduced during the lecture assumes that one resource belongs to one single role. In reality, a resource can be playing different roles and, thus, perform activities of various nature. The algorithm implemented in PM4Py allows a resource to be part of multiple roles.

The role discovery in PM4Py is implemented via function ***roles_discovery.apply()*** (see Figure 14). This function returns a set of pairs **(x,y)**, each indicating that the resources in **y** belong to the same role, namely that enabling performing the activities in **x**. Each resource is also associated with the number of executions of activities in **x**. As an example, the first role in output in Figure 14 is that to perform *Amend Purchase Requisition*, which includes eight resources (in these case process executors): *Penn Osterwalder, Nico Ojenbeer, Clement Duchot, Kim Passa, Miu Hanwan, Elvira Lores, Immanuel Karagianni, and Christian Francois*. Also, *Kim Passa, Miu Hanwan* and *Immanuel Karagianni* performed *Amend Purchase Requision* twice each, while the other resources only once.

Note that PM4Py implements a role-discovery algorithm that is more elaborated of that seen in the lecture: it indeed allows the same resource to belong to more than one role.

```
#Alternative way:
#from pm4py.algo.organizational_mining.roles import algorithm as role_discovery
#roles = role_discovery.apply(log)

roles = pm4py.discover_organizational_roles(log)
roles
```

```
[[['Amend Purchase Requisition'],
  {'Kim Passa': 2,
   'Penn Osterwalder': 1,
   'Christian Francois': 1,
   'Nico Ojenbeer': 1,
   'Miu Hanwan': 2,
   'Elvira Lores': 1,
   'Immanuel Karagianni': 2,
   'Clement Duchot': 1}],
 [['Amend Request for Quotation',
   'Analyze Quotation Comparison Map',
   'Choose best option',
   'Create Purchase Requisition',
   'Release Purchase Order'],
  {'Kim Passa': 176,
   'Immanuel Karagianni': 175,
   'Anne Olwada': 181,
   'Heinz Gutschmidt': 19,
   'Miu Hanwan': 175,
   'Fjodor Kowalski': 177,
   'Christian Francois': 170,
   'Esmana Liubiata': 160,
   'Clement Duchot': 162,
```

Figure 14. Discovery of Resource' Roles

Step 5: Computation of working days and hours

PM4Py allows one to compute the working calendars of resources, namely the days and hours in which each resource works.

In order to discover the timetable for the resources, firstly it is necessary to detect the workdays i.e on which days of the week each resource works. Figure 15 illustrates a function that plots the working days for a given resource. It is possible to observe how many times the resource works for each day. In fact, the function `get_timetable_day()` builds a dictionary *week* by iterating the log. For each event in the log performed by the given resource the function retrieves the day of the week and increments the corresponding entry in the dictionary by one. Finally, the function creates a bar plot with the library *matplotlib*, with the values of the dictionary in the y-axis and the days of week in the x-axis.

```
import time
import matplotlib.pyplot as plt

def get_timetable_day(log,resource):
    ## define dictionary with days of week
    week={'Monday': 0, 'Tuesday': 0, 'Wednesday': 0, 'Thursday': 0, 'Friday': 0, 'Saturday': 0, 'Sunday': 0}
    for trace in log: ## iterate for trace
        for event in trace: ## iterate for event
            if event['org:resource']==resource: ## check the resource
                date=event['time:timestamp']
                date=time.strftime("%A", time.strptime(str(date.date()), "%Y-%m-%d")) ### find the day of the week
                week[date]=week[date]+1 ## add item to the corresponding histogram

    fig = plt.figure(figsize = (15, 8)) ## define the size of bar plot
    ## create a bar plot, the first value is for the x-axis and the second for the y-axis
    ## the x-axis contains the days of the week while the y-axis contains the number of times that the resource worked
    plt.bar(week.keys(), week.values())
    plt.xlabel("Days") ## label for the x-axis
    plt.title("Timetable day") ## title of bar plot
    plt.show() ## show the bar plot
```

Figure 15. Computing the day timetable for a given resource

The resulting plot in Figure 16 shows the working days for the resource *Elvira Lores*, in particular she works every day of week.

```
resource='Elvira Lores'
get_timetable_day(log,resource)
```

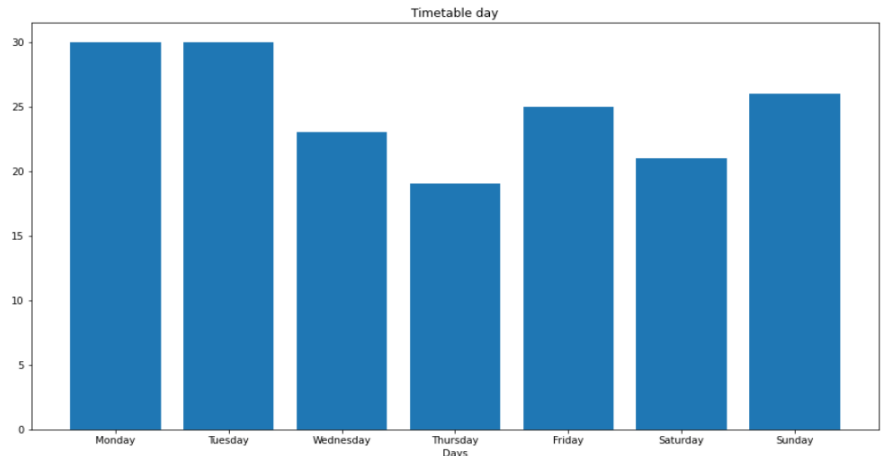

Figure 16. Day timetable for the resource Elvira Lores

Once the working days are established, we want to discover the working hours for the resources. In the same way it is possible to define a function that extracts the hour when the resource performs an activity. Figure 17 describes the function ***get_timetable_hour()*** that given a resource it returns his hour timetable.

```
import pytz
import pandas as pd
from collections import defaultdict
import matplotlib.pyplot as plt
utc=pytz.UTC

def get_timetable_hour(log,resource):
    ## define dictionary for the time of day
    hours=dict()
    for i in range(0,24):
        hours[i]=0
    for trace in log: ## iterate for trace
        for event in trace: ## iterate for event in trace
            if event['org:resource']==resource: ## check the resource
                date=event['time:timestamp']
                date=date.replace(tzinfo=utc) ## to transform string into datetime ("2011-01-01 00:00:00+00:00")
                time=date.time().hour ## extract hour to datatime format
                hours[time]=hours[time]+1 ## add item to the corresponding histogram

    fig = plt.figure(figsize = (20, 8)) ## define the size of bar plot
    ## create a bar plot, the first value is for the x-axis and the second for the y-axis
    ## the x-axis contains the time of day while the y-axis contains the number of times that the resource worked
    plt.bar(hours.keys(), hours.values())
    plt.xlabel("Time of day") ## label for the x-axis
    plt.title("Hour timetable") ## title of bar plot
    plt.show() ## show the bar plot
```

Figure 17. Computing the hour timetable for a given resource

```
resource = 'Elvira Lores'
get_timetable_hour(log, resource)
```

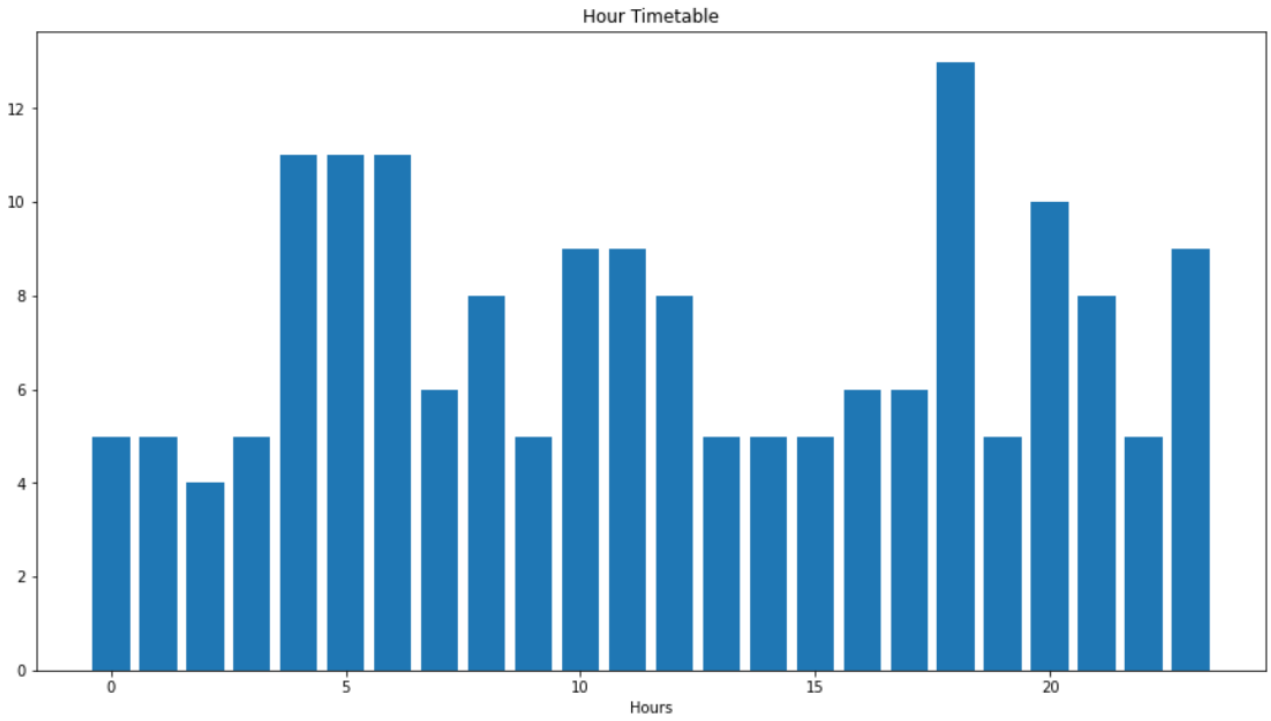

Figure 18. Hour timetable for the resource Francois

Final Exercises

Exercise 1 – Find all the activities inside the log

Iterate the log structure to identify activities within the log and to calculate, for each activity, how many times it occurs in the process. For each activity, print out its name and the number of occurrences.

Exercise 2 – Find all the resources inside the log

Iterate the log structure to identify the resources within the log and to calculate, for each resource, how many times the resource is involved in executions of process activities. For each resource, print out its name and the number of occurrences.

Exercise 3 – Use the Footprints function

Footprints are used in discovery (cf. Alpha Miner) and in conformance checking (recall the comparison between the footprints of logs and models). This can include: directly-follows relationships and parallel behavior. Use the function ***footprints_discovery.apply()*** (see below how to use it) to detect what activities are in parallel. Then, using the same function to find the most frequent directly-follows relationship between two activities within the log.

```
from pm4py.algo.discovery.footprints import algorithm as footprints_discovery

fp_log = footprints_discovery.apply(log, variant=footprints_discovery.Variants.ENTIRE_EVENT_LOG)
```


Exercise 4 – Discovery the hour timetable for roles

Step 5: Computation of working days and hours above calculates the working calendars for each resource. However, resources belong to roles, and one can compute the working calendars for each role, assuming that resources playing the same role have similar calendars. Modify the functions defined in Figure 14 to calculate the day and the hour timetable for the third role that was discovered through function ***roles_discovery.apply()*** :

Role 3: (['Analyze Purchase Requisition'],{'Heinz Gutschmidt': 124, 'Maris Freeman': 133, 'Francis Odell': 125})

Finally, plot the day and the hour timetables.

Exercise 5 – Find the distribution of the events over the days of week

Through the function ***view_events_distribution_graph()*** check if the company works on weekends as we estimated through the resources. Change the parameters ***distr_type*** in *days_month/hours/months* and observe what happens. In which month did the company work the hardest?

```
from IPython.display import SVG,display
pm4py.view_events_distribution_graph(log, distr_type="days_week", format="jpg")
```

Exercise 6 – Discovery a process model (optional)

Mine the Petri net for the event log used in this laboratory session, using the Inductive Miner. Try with different parameters of the noise threshold (see below).

The invocation of the Inductive Miner is through the function ***inductive_miner.apply()*** where the parameter ***Parameters.NOISE_THRESHOLD*** indicates the minimum frequency for a certain behaviour to be incorporated into the model:

```
from pm4py.algo.discovery.inductive import algorithm as inductive_miner
from pm4py.algo.discovery.inductive.variants.im_clean.algorithm import Parameters

net, im,fm = inductive_miner.apply(log, parameters={Parameters.NOISE_THRESHOLD: 0.5})
```

The function returns the Petri net *net*, the initial marking *im* and the final marking *fm*.

The visualisation of the Petri net can be performed as follows:

```
from pm4py.visualization.petri_net import visualizer as pn_visualizer
gviz = pn_visualizer.apply(net, im, fm)
pn_visualizer.view(gviz)
```

The generated Petri net can be imported or exported. For instance, to export the Petri net just discovered, you can do as follows:


```
from pm4py.objects.petri_net.exporter import exporter as pnml_exporter
pnml_exporter.apply(net, im, "petri.pnml")
```