BPI Challenge 2020: investigation for cost reduction and faster process management

Marinò Giosuè Cataldo

Business Information Systems
Prof. Paolo Ceravolo

June 28, 2022

Abstract

In this project I decided to carry out several investigations to try to reduce the company's costs by identifying critical issues through the analysis of the event log (eg travel declarations paid more than once). Subsequently, mining techniques are used to analyze the travel declaration processes, with the aim of understanding the various phases and looking for where it would be useful to automate to make everything more efficient.

Contents

1 Description of the case study

I chose the datasets made available in the BPI challenge 2020[1].

1.1 Description of the challenge

In many organizations, staff members travel for work. They travel to customers, to conferences or to project meetings and these travels are sometimes expensive. As an employee of an organization, you do not have to pay for your own travel expenses, but the company takes care of them. At Eindhoven University of Technology (TU/e), this is no different. The TU/e staff travels a lot to conferences or to other universities for project meetings and/or to meet up with colleagues in the field. And, as many companies, they have procedures in place for arranging the travels as well as for the reimbursement of costs. On a high level, we distinguish two types of trips, namely domestic and international. For domestic trips, no prior permission is needed, i.e. an employee can undertake these trips and ask for reimbursement of the costs afterwards. For international trips, permission is needed from the supervisor. This permission is obtained by filing a travel-permit and this travel permit should be approved before making any arrangements. To get the costs for a travel reimbursed, a claim is filed. This can be done as soon as costs are actually payed (for example for flights or conference registration fees), or within two months after the trip (for example hotel and food costs which are usually payed on the spot).

1.2 The Data

The data is split into travel permits and several request types, namely domestic declarations, international declarations, prepaid travel costs and requests for payment, where the latter refers to expenses which should not be related to trips (think of representation costs, hardware purchased for work, etc.). The available datasets are:

- Requests for Payment (should not be travel related): 6,886 cases, 36,796 events: RequestForPayment.xes;
- Domestic Declarations: 10,500 cases, 56,437 events: DomesticDeclarations.xes;
- Prepaid Travel Cost: 2,099 cases, 18,246 events: PrepaidTravelCost.xes;
- International Declarations: 6,449 cases, 72151 events: InternationalDeclarations.xes;
- Travel Permits (including all related events of relevant prepaid travel cost declarations and travel declarations): 7,065 cases, 86,581 events: PermitLog.xes.

[1] Available at https://www.tf-pm.org/competitions-awards/bpi-challenge/2020

2 Organisational goals

Going into detail, I initially perform a filtering on the date as indicated in the challenge since the process explained is valid starting from 2018. I then perform a series of descriptive analyzes to understand how the data are organized. Then I answer some of the questions asked in the challenge, focusing on those of temporal analysis on the various stages of the process and in the field of multiple payments. The questions of the challenge to which I answer are therefore:

- What is the throughput of a travel declaration from submission (or closing) to paying?
- Is there are difference in throughput between national and international trips?
- Where are the bottlenecks in the process of a travel declaration?
- Are there any double payments?
- How many travel declarations get rejected in the various processing steps and how many are never approved?

Next I focus on the most frequent variants:

1. I analyze the percentages of all the examples I cover;
2. through miners I generate the Petri nets associated with the various processes through a model selection that tests different parameters and finds the best model based on different metrics;
3. conformance checking of the various models with analysis of events that do not fit the model

Next I try to merge/split datalogs for a simpler process mining, after that I try to create a predictive model for the overpent flag; finally the conclusions with the reflections deduced from the results.

3 Knowledge Uplift Trail

	Input	Analytics/Model	Type	Output
Step 1	5 event logs	Temporal filtering	Prescriptive	Filtered event logs
Step 2	Step 1	Most significative fields And events (Disco)	Descriptive	Fields and events explanation
Step 3	Step 1 Step 2	Throughput for payed request	Descriptive	Statistics
Step 4	Step 1 Step 2 Step 3	Difference in throughput for payed request Between national and international Declaration	Descriptive	Statistics
Step 5	Step 1 Step 2 Step 3	Bottlenecks research	Descriptive	Statistics
Step 6	Step 1 Step 2	Checking double payments	Descriptive	Statistics
Step 7	Step 1 Step 2	Checking rejected declarations	Descriptive	Statistics
Step 8	Step 1	Filtering logs on top 5 variants	Prescriptive	Filtered event logs
Step 9	Step 8	Find best models for each log	Prescriptive	Petri net and Heuristic net for each log
Step 10	Step 9	Conformance Checking for each model	Descriptive	Anomalous cases for each log
Step 11	Step 10	Analysis of anomalous cases with Disco and PM4PY	Descriptive	Information of anomalous cases
Step 12	Step 8	Splitting InternationalDeclaration	Prescriptive	Analysis of splitted logs
Step 13	Step 8	Merge 2 events log for simpler process management	Prescriptive	Best combination Of merged logs
Step 14	Step 12 Step 9	Find best models for joined log (Domestic and Request)	Prescriptive	Petri net for best
Step 15	Step 14	Conformance Checking for model	Descriptive	Anomalous cases for joined log
Step 16	Step 8 Step 13	Comparing models for single Log with model for Merged log	Descriptive	Comparison of models
Step 17	Step 1	Predicting overspent declarations	Predictive	Predictive model

Log	Cases before	Cases after
Domestic Declarations	10500	8260
International Declarations	6449	4952
Permit Log	7065	5598
Prepaid Travel Cost	2099	1776
Request For Payment	6886	5778

Table 1: Number of traces before and after time filtering

4 Project Results

4.1 Temporal Filtering

As indicated on the BPI challenge 2020 page, the process described is represented by the event logs attached starting from 2018, as before then only two units were logging data as per description.

```
from pm4py.algo.filtering.log.timestamp import timestamp_filter
d_log = {k:timestamp_filter.filter_traces_contained(d_df[k], "2018-01-01 00:00:00", "2022-01-18 23:59:59",
                parameters={timestamp_filter.Parameters.TIMESTAMP_KEY: "time:timestamp"}) for k in d_df.keys()}
```

Figure 1: Snippet of temporal filtering.

I then filtered the five event logs using PM4PY[2], a piece of the procedure[3] used for filtering is shown in Figure 1.

d_df and *d_log* are two python dictionaries that have as their key a string that indicates shortly and the name of the log file while as value they contain respectively the DataFrame pandas and the event log associated with the key. Table 1 shows the number of cases before and after temporal filtering.

4.2 Fields and events analysis

In this section initially I want to understand in detail how the process is managed and through which events it is carried out. In this case I used Disco[4], an academically licensed software that allows you to easily apply filters, analyze different variants, look at different field statistics, etc.

Figure 2 shows a trace for the variants with multiple cases of the DomesticDeclarations dataset, it is not shown to avoid redundancy but most of the processes that come to an end (ie where employees receive the refund) start with a "Declaration SUBMITTED by event EMPLOYEE" (in some cases preceded by two events: "Start Trip" and "End Trip"; most successful refund processes end with "Request Payment" followed by "Payment Handled".

In order to check multiple payments, it is necessary to understand which field corresponds to the amount of money requested in a declaration or request for reimbursement. Also through Disco I noticed that the events in their respective datasets contain the field:

- Domestic Declarations → "RequestedAmount";
- International Declarations → "RequestedAmount";
- Permit Log → "TotalDeclared";
- Prepaid Travel → "RequestedAmount";
- Request For Payment → "RequestedAmount".

These fields are populated to 0 when the request is unsuccessful, as shown in Figure 3

[2] Documentation PM4PY
[3] PM4PY filtering
[4] Disco webpage

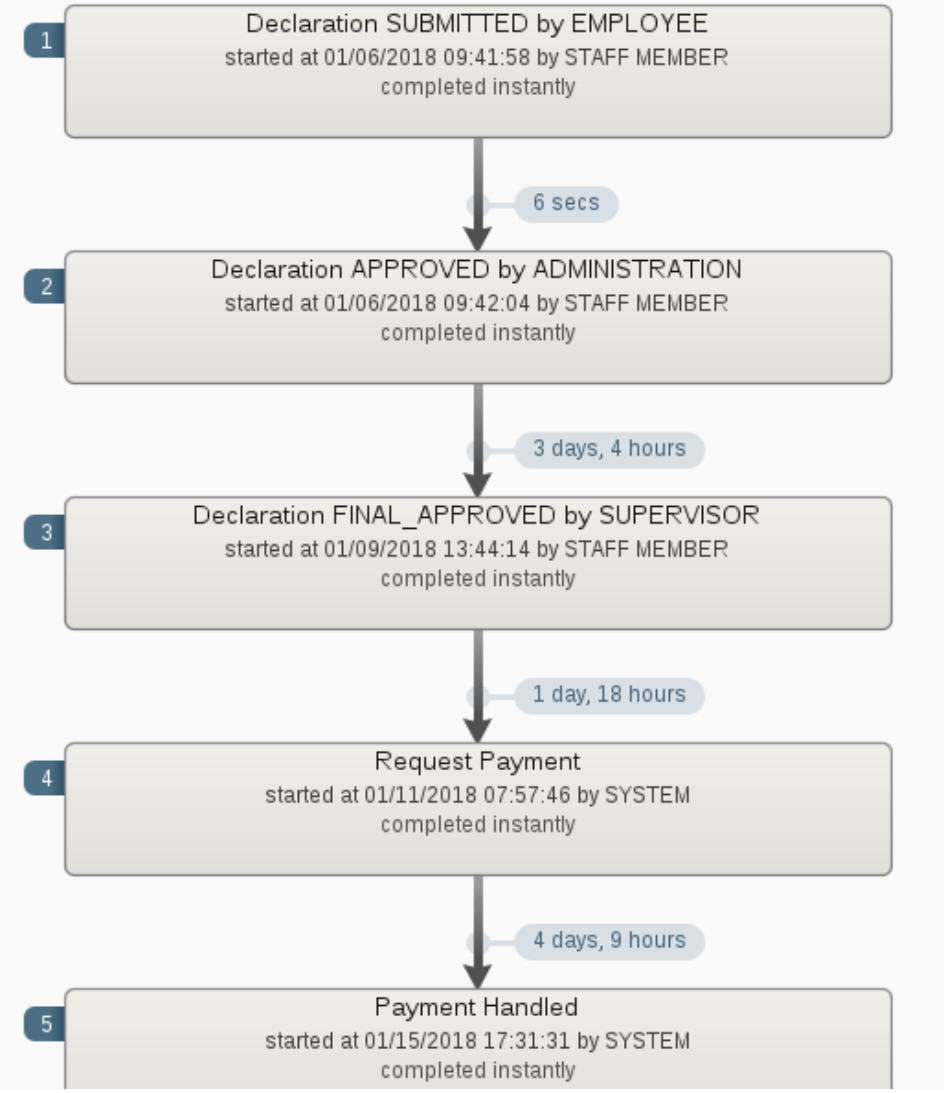

Figure 2: Events of one case of most popular variants in DomesticDeclarations.

	Activity	Resource	Date	Time	Permit travel permit number	DeclarationNumber	Amount	RequestedAmount
1	Permit SUBMITTED by EMPLOYEE	STAFF MEMBER	01/15/2018	21:20:47	travel permit number 61813	UNKNOWN	0.0	0.0
2	Permit APPROVED by ADMINISTRATION	STAFF MEMBER	01/15/2018	21:21:16	travel permit number 61813	UNKNOWN	0.0	0.0
3	Permit APPROVED by BUDGET OWNER	STAFF MEMBER	01/17/2018	13:12:54	travel permit number 61813	UNKNOWN	0.0	0.0
4	Permit FINAL_APPROVED by SUPERVISOR	STAFF MEMBER	01/25/2018	21:18:31	travel permit number 61813	UNKNOWN	0.0	0.0
5	Start trip	STAFF MEMBER	04/03/2018	00:00:00	travel permit number 61813	UNKNOWN	0.0	0.0
6	End trip	STAFF MEMBER	04/06/2018	00:00:00	travel permit number 61813	UNKNOWN	0.0	0.0
7	Declaration SUBMITTED by EMPLOYEE	STAFF MEMBER	04/18/2018	11:26:32	travel permit number 61813	UNKNOWN	0.0	0.0
8	Declaration REJECTED by ADMINISTRATION	STAFF MEMBER	04/18/2018	11:26:44	travel permit number 61813	UNKNOWN	0.0	0.0
9	Declaration REJECTED by EMPLOYEE	STAFF MEMBER	04/24/2018	16:34:53	travel permit number 61813	UNKNOWN	0.0	0.0

Figure 3: Events of one case of most popular variants in DomesticDeclarations.

4.3 Throughput for payed request

Now I want to measure the throughput for successful refund requests, to do this the snippet in Figure 4 shows the code to extract the time elapsed between the request made by the employee to the managed payment event. The results are shown in the Table 2. Now it is also possible to answer the question on the difference in throughtput between national (Domestic) and international declarations, observing the Table 2 we note that they have a very similar average time in days.

```
list_res = []
for trace in d_log["nat_dec"]:
    start = None
    end = None
    for event in trace:
        if event['concept:name'] == 'Declaration SUBMITTED by EMPLOYEE':
            start = event['time:timestamp']
        elif event['concept:name'] == 'Payment Handled':
            end = event['time:timestamp']
            break
    if start is not None and end is not None:
        list_res.append((start, end, trace.attributes['concept:name']))

pd.Series([(end-start).total_seconds()/60/60/24 for (start, end, _) in list_res]).describe()
```

Figure 4: Code to get the throughtput measured in days.

	Mean		
Log	SUBMITTED by EMPLOYEE to Payment Handled	Request Payment to Payment Handled	FINAL_APPROVED by SUPERVISOR to Request Payment
Domestic Declarations	10.35	3.45	2.84
International Declarations	12.38	3.41	2.86
Permit Log	12.18	3.45	2.90

Table 2: Average days to pass from one phase to another, indicated in the columns.

4.4 Bottlenecks

In order to speed up the process it is interesting to analyze in detail if there are any bottlenecks during the various stages of the process. The code snippet in Figure 5 shows the code to identify the 25th percentile, median time (50th percentile), 75th percentile and the maximum time, expressed in days.

The Table 3 shows the results. We can see that the generally slowest sequence is Request Payment to Payment Handled. Ignoring the outliers present in the maximums, we can see that the DomesticDeclarations have the same timing as InternationalDeclaration despite the latter we have a more complicated approval process that also requires the passage of Permit.

```
for k in d_log.keys():
    list_res = []
    for trace in d_log[k]:
        start = None
        end = None
        for event in trace:
            if event['concept:name'] == 'Declaration SUBMITTED by EMPLOYEE':
                start = event['time:timestamp']
            elif event['concept:name'] == 'Payment Handled':
                end = event['time:timestamp']
                break
        if start is not None and end is not None:
            list_res.append((start, end, trace.attributes['concept:name']))

    print(k,"\n", pd.Series([(end-start).total_seconds()/60/60/24 for (start, end, _) in list_res]).describe()[
    ["25%","50%", "75%", "max"]])
```

Figure 5: Code to analyze the bottlenecks in the sequence of events.

4.5 Checking double payments

Focusing on company costs reduction, I want to check for duplicate payments in the same trace. I started by writing a small script to verify the presence or absence of trace with this behavior, in Figure 6 I report the code with its output.

Since the PermitLog shows 1017 traces with multiple Payment Handled events, I specifically investigated this log to understand how much money it was. The suspicious traces are collected with the code of Figure 6, subsequently analyzed with the code in Figure 7, the first 10 rows are also shown in descending order with respect to the LOSS column. I then added the LOSS value for each line to get an amount equivalent to 4478912.

Log	Measure	Sequence 1	Sequence 2	Sequence 3	Sequence 4
Domestic Declarations	25th perc	2.32	0.30	<0.01	0.60
	Median	3.21	1.06	<0.01	1.70
	75th perc	4.29	3.67	<0.01	4.15
	Max	62.15	105.86	282.58	54.66
International Declarations	25th perc	2.32	0.26	<0.01	0.82
	Median	3.23	1.08	<0.01	2.63
	75th perc	4.29	3.75	0.02	6.31
	Max	29.06	266.10	422.00	57.79
Permit Log	25th perc	2.32	0.28	<0.01	0.81
	Median	3.24	1.07	<0.01	2.60
	75th perc	4.30	3.83	0.02	6.27
	Max	29.06	266.10	348.51	52.82

Table 3: Sequence 1: Request Payment to Payment Handled
Sequence 2: Declaration FINAL_APPROVED by SUPERVISOR to Request Payment
Sequence 3: Declaration SUBMITTED by EMPLOYEE to Declaration APPROVED by ADMINISTRATION
Sequence 4: Declaration APPROVED by ADMINISTRATION to Declaration FINAL_APPROVED by SUPERVISOR

```
def double_pay(log):
    dict_decl = set()
    res_double = []
    for trace in log:
        for event in trace:
            if event['concept:name'] == 'Payment Handled'
                dec = trace.attributes['concept:name']
                if dec in dict_decl:
                    res_double.append(dec)
                else:
                    dict_decl.add(dec)

    return len(res_double), set(res_double)
for k in d_log.keys():
    r = len(double_pay(d_log[k])[1])
    if r != 0:
        print(f"{k} --> {len(double_pay(d_log[k])[1])}")
```

✓ 0.2s

```
per_log --> 1017
```

Figure 6: Code to inspect double payments into log.

4.6 Checking rejected declarations

In this section, I answer the question about denied declarations from the BPI Challenge 2020. Figure 8 shows statistics for traces with denial events and traces never completed for the DomesticDeclarations and InternationalDeclarations log files.

4.7 Filtering logs on top k variants

In this section I have filtered the log files to get the top-k popular variants[5]. The Figure 9 shows the code to perform the filter and the relative coverage percentages with respect to the total traces and

[5] top-k variants filter

```
set_double = list(double_pay(d_log["per_log"])[1])
res_dict = {k:[] for k in set_double}
for trace in d_log["per_log"]:
    if (trace.attributes["concept:name"]) in set_double:
        for event in trace:
            if event['concept:name'] == 'Payment Handled':
                res_dict[trace.attributes["concept:name"]].append(trace.attributes["TotalDeclared"])
tmp = []
for k, v in res_dict.items():
    tmp.append([k, round(v[0]), len(v)])
df_res = pd.DataFrame(tmp, columns=["ID", "VALUE", "REPETITION"])
df_res["LOSS"] = df_res["VALUE"].to_numpy()*(df_res["REPETITION"].to_numpy()-1)
df_res.sort_values("LOSS", ascending=False).head(10)
```

	ID	VALUE	REPETITION	LOSS
622	travel permit 54518	1458579	2	1458579
481	travel permit 22776	12232	14	159016
276	travel permit 24054	9246	12	101706
789	travel permit 53343	4811	14	62543
120	travel permit 57181	4652	9	37216
904	travel permit 33319	6177	6	30885
676	travel permit 45672	6626	5	26504
943	travel permit 66595	5043	6	25215
813	travel permit 28756	4133	7	24798
602	travel permit 2501	3842	6	19210

Figure 7: Code to analys double payments into PermitLog.

```
DOMESTIC DECLARATION
Traces with rejected status=1072, Total traces=8260, %traces with rejected=12.98%
Traces never approved=357, Total traces=8260, %traces never approved=4.32%


INTERNATIONAL DECLARATION
Traces with rejected status=1458, Total traces=4952, %traces with rejected=29.44%
Traces never approved=211, Total traces=4952, %traces never approved=4.26%
```

Figure 8: Statistics of rejected/never approved traces in Domestic and International Declaration.

the number of traces kept.

```
import pm4py
d_log_var = {k:pm4py.filter_variants_top_k(d_log[k], 5) for k in d_log.keys()}

for k,v in d_log_var.items():
    print (f"{k} --> {round(100*(len(v)/len(d_log[k])))}% - {len(v)}")
```

✓ 1.1s

```
nat_dec --> 94% - 7786
int_dec --> 53% - 2632
per_log --> 38% - 2135
pre_tra_cost --> 68% - 1199
req_4_pay --> 94% - 5432
```

Figure 9: Code and result of top-k variants.

4.8 Best model for each log

We now want to find a good model that has a good trade-off between the metrics seen in class, namely: fitness, precision, generalization and simplicity. As models I used the three miners available in the PM4PY package[6]:

- Alpha Miner

[6] PM4PY Process Discovery

- Inductive Miner (also Inductive Miner infrequent and Inductive Miner directly-follows, trying differt values for NOISE parameter)
- Heuristic Miner (trying different combinations for the hyperparameters)

For NOISE of "Inductive Miner infrequent" I tried values $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$. For heuristic miner instead I tried to change the following parameters:

- DEPENDECY_TRESH $\in \{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$
- MIN_ACT_COUNT $\in \{1, 20, 100, 200, 300\}$
- MIN_DFG_OCCURRENCES $\in \{1, 20, 100, 200, 300\}$

I wrote some code to test the different combinations of parameters and patterns and then put the results in a DataFrame. I then filtered the results of the table obtained for each log with respect to the metrics indicated above to obtain the best model with the best parameters, finally I created a Petri net for each log. The best results obtained for each log file are shown in Figure 10.

```
r = tot_res[(tot_res["FITNESS"]>=92) & (tot_res["PRECISION"]>=.8)]\
    .sort_values(["SEMPLICITY","PRECISION", "GENERALIZATION"], ascending=False)\
        .groupby("LOG").head(1)
r.to_csv("best.csv")
r
```

✓ 0.9s

	MODEL	PARAMETERS	FITNESS	PRECISION	GENERALIZATION	SEMPLICITY	LOG
2	IMf	[0.0]	92.24	0.92	0.96	0.87	PrepayedTravelCost
14	HEU	[0.1, 1, 1]	100.00	0.99	0.98	0.85	DomesticDeclaration
14	HEU	[0.1, 1, 1]	100.00	0.99	0.97	0.85	RequestForPay
17	HEU	[0.1, 1, 200]	94.68	0.92	0.97	0.84	InternationalDeclaration
17	HEU	[0.1, 1, 200]	92.22	0.89	0.97	0.83	PermitLog

Figure 10: Best results.

To make the visualization easier, in the report I preferred to attach the heuristic nets, both the Petri nets and the heuristic nets of each log file are on Colab[7].

4.8.1 DomesticDeclaration

The heuristic net related to DomesticDeclarations is shown in Figure 11, in this case we can see that the model has learned the process well, this net covers the various types that can exist in this type of process:

- everything goes well, it starts with the declaration of the employee who receives the various approvals and arrives at the payment;
- after the declaration of the employee you get a reject from the administration followed by reject from the employee, here are two options: 1. the process ends, 2. the employee creates a new declaration and the process starts again.

In this process it is also evident that the approval of the budget owner is optional and for the cases analyzed, about two thirds of the cases do not need this approval.

4.8.2 InternationalDeclarations

Figure 12 shows heuristic net related to InternationalDeclarations. In this case the process is less precise, we immediately notice that reimbursement rejects are not covered because from the rejected declaration stage we then have only 2 options: re-execute the request by the employee which can be rejected by going in the loop or accepted and yes advance until payment. Unlike DomesticDeclaration, a permit procedure takes place before the various declarations. Permit also follows steps very similar to the declaration, i.e. submitted by employee to then receive the various approvals.

Also in this case we note that the approval of the budget owner is optional.

[7] Colab link

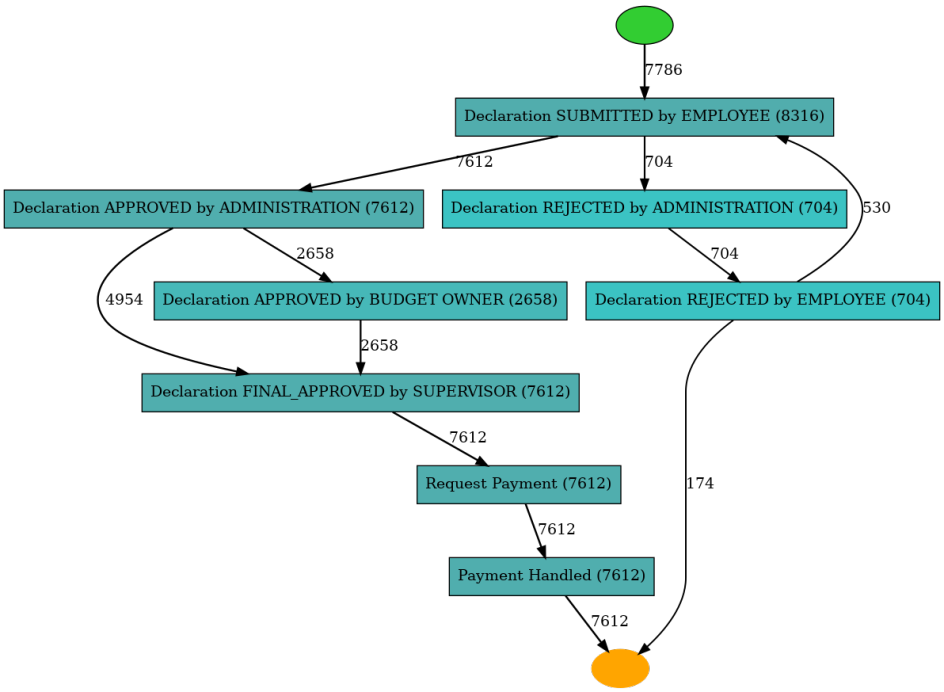

Figure 11: Heuristic Net of DomesticDeclarations.

4.8.3 PermitLog

Figure 13 shows heuristic net related to PermitLog. The net is a bit more complicated than the previous ones but still quite simple, very similar to the behavior in InternationalDeclarations with the addition of the Send Reminder event, which loops with itself and then ends. Here, too, the Budget Owner in the approvals remains optional.

4.8.4 RequestForPayment

Figure 14 shows heuristic net related to RequestForPayment. Process very similar to DomesticDeclaration, it seems to differ only in the prefix of the events that become "Request For Payment". Also here, the Budget Owner in the approvals remains optional.

4.9 Conformance Checking via alignments

So now I want to check the goodness of the various models just generated. To carry out this check I use PM4PY alignments, the script used is shown in Figure 15.

4.9.1 DomesticDeclarations and RequestForPayment

As shown in Figure 16 and Figure 17 for DomesticDeclarations and RequestForPayment there are no anomalous cases with respect to the two Petri nets found in the Best Model section.

4.9.2 InternationalDeclarations

For InternationalDeclarations I find 140 anomalous cases showed in Figure 18, analyzing these traces I realized that all non-aligned cases are those that start from "Start trips" and then follow the classic inter of submissions and approvals as shown in Figure 19, a case that perhaps shouldn't be possible?

4.9.3 PermitLog

For PermitLog I find 166 anomalous cases showed in Figure 20, analyzing these traces I realized that the cases not covered are those that have "Request For Payment SUBMITTED by EMPLOYEE" to

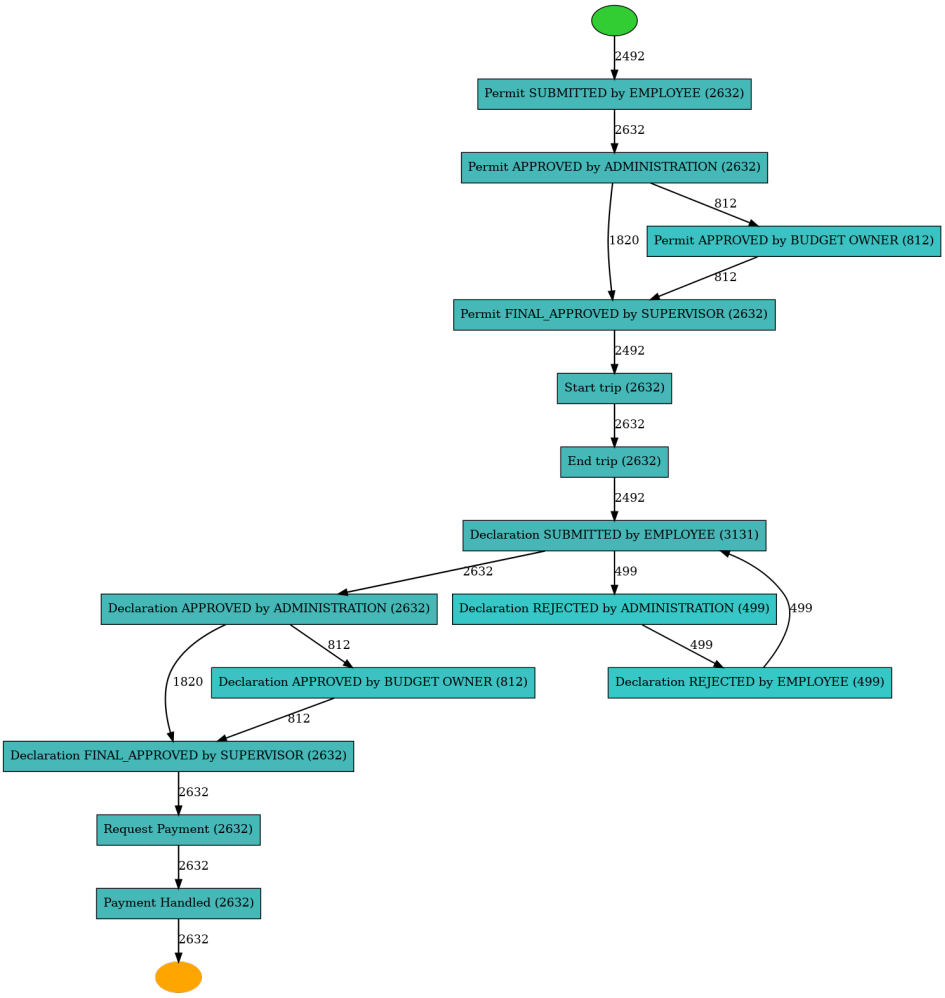

Figure 12: Heuristic Net of InternationalDeclarations.

"Request Payment" as shown in Figure 21. The question I ask myself is, "Is it correct that there are cases that come in demand for payment without the various approvals of higher-level employees?"

4.9.4 PrepaidTravelCost

For PermitLog I find 93 anomalous cases showed in Figure 22, analyzing these traces I realized that all non-aligned cases are those that start "Request For Payment SUBMITTED by EMPLOYEE" as shown in Figure 23, practically employees ask for money before submitting a permit? In this case I have also analyzed a particular case via Disco, shown in Figure 24, and actually it seems that the employees start asking for money from the company.

4.10 Splitting InternationalDeclaration

Given the results of the Section 4.8, I decided to focus on InternationalDeclarations and perform a division of the traces between those with successful and unsuccessful payments In Figure 25 the code used to make the split is shown, from this result I understand that actually the cases of unsuccessful payment are very few (only 0.03%) and it is therefore of little use to find a process model for so few traces.

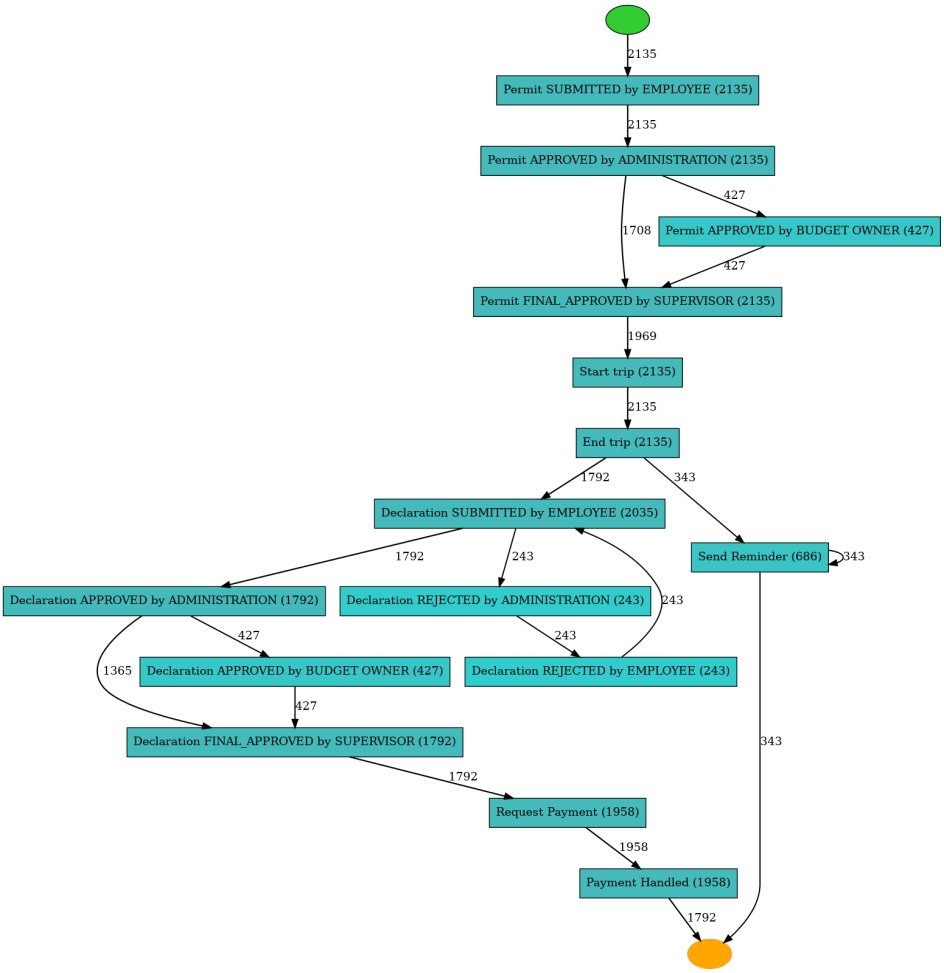

Figure 13: Heuristic Net of PermitLog.

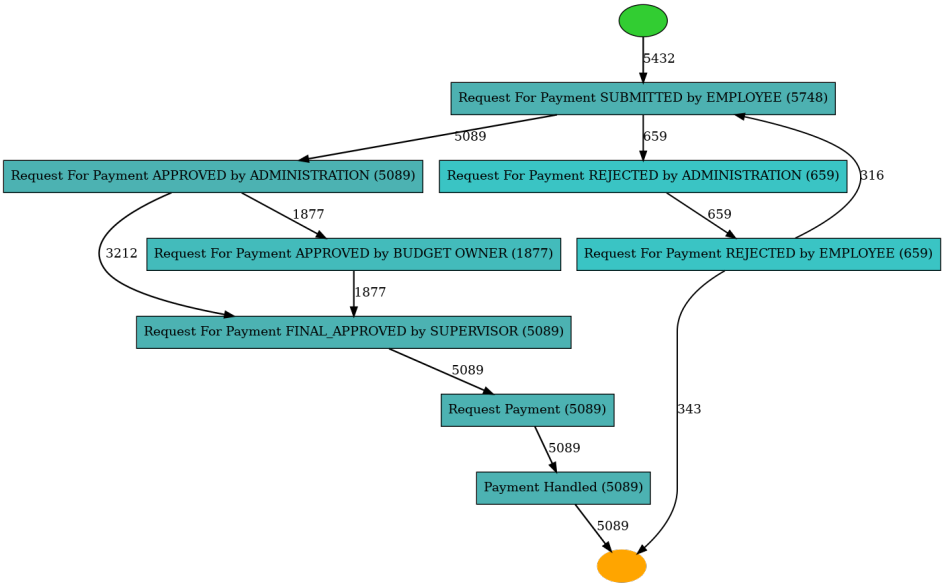

Figure 14: Heuristic Net of RequestForPayment.

```python
from pm4py.algo.conformance.alignments.petri_net import algorithm as alignments

def align(log, net, im, fm):
    aligned_traces = alignments.apply_log(log, net, im, fm)

    print("ALIGNMENTS")
    print("Number of traces", len(aligned_traces))
    regular_traces = []
    anomalous_traces = []
    case_ids = []
    i = 0
    while i < len(aligned_traces):
        if aligned_traces[i]["fitness"] != 1:
            anomalous_traces.append(aligned_traces[i]['alignment'])
            case_ids.append(log[i].attributes["concept:name"])
        else:
            regular_traces.append(aligned_traces[i]['alignment'])
        i += 1
    print("Number of anomalous traces ", len(anomalous_traces))
    print("Number of normal traces ", len(regular_traces))
    print("Percentage of anomalous traces", (len(anomalous_traces)/len(aligned_traces))*100 , '%')
    print("Odds of anomalous traces", round(len(anomalous_traces)/len(regular_traces),2))

    return anomalous_traces
```

Figure 15: Function for alignments.

```
ALIGNMENTS
Number of traces 7786
Number of anomalous traces  0
Number of normal traces  7786
Percentage of anomalous traces 0.0 %
Odds of anomalous traces 0.0
```

Figure 16: Alignments for DomesticDeclarations.

```
ALIGNMENTS
Number of traces 5432
Number of anomalous traces  0
Number of normal traces  5432
Percentage of anomalous traces 0.0 %
Odds of anomalous traces 0.0
```

Figure 17: Alignments for RequestForPayment.

```
ALIGNMENTS
Number of traces 2632
Number of anomalous traces  140
Number of normal traces  2492
Percentage of anomalous traces 5.319148936170213 %
Odds of anomalous traces 0.06
```

Figure 18: Alignments for InternationalDeclarations.

```python
from pm4py.algo.filtering.log.start_activities import start_activities_filter
log_start = start_activities_filter.get_start_activities(d_log_var["int_dec"])
filtered_log = start_activities_filter.apply(d_log_var["int_dec"], ["Start trip"])

len(filtered_log)
```

✓ 0.6s

```
140
```

Figure 19: Anomalous cases for InternationalDeclarations.

```
ALIGNMENTS
Number of traces 2135
Number of anomalous traces  166
Number of normal traces  1969
Percentage of anomalous traces 7.775175644028103 %
Odds of anomalous traces 0.08
```

Figure 20: Alignments for PermitLog.

```
filtered_log = pm4py.filter_between(d_log_var["per_log"],
        "Request For Payment SUBMITTED by EMPLOYEE", "Request Payment")
len(filtered_log)
```
✓ 0.5s

```
166
```

Figure 21: Anomalous cases for PermitLog.

```
ALIGNMENTS
Number of traces 1199
Number of anomalous traces  93
Number of normal traces  1106
Percentage of anomalous traces 7.756463719766472 %
Odds of anomalous traces 0.08
```

Figure 22: Alignments for PrepaidTravelCost.

```
start_activities_filter.get_start_activities(d_log_var["pre_tra_cost"])
```
✓ 0.4s

```
{'Permit SUBMITTED by EMPLOYEE': 1106,
 'Request For Payment SUBMITTED by EMPLOYEE': 93}
```

Figure 23: Anomalous cases for PrepaidTravelCost.

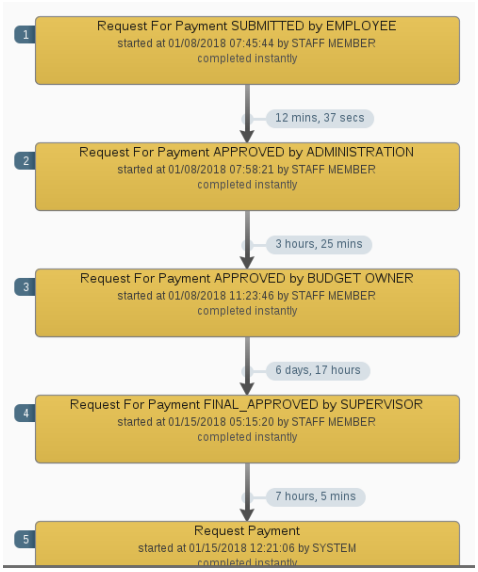

Figure 24: Anomalous cases for PrepaidTravelCost, request 185740.

4.11 Merge DomesticDeclarations and RequestForPayment

After the insight in the Section on the similarity between the heuristic net of DomesticDeclarations and that of RequestForPay I decide to remove the prefix to the events of the two log files and then go and find a good model for the joined log.

```
from pm4py.algo.filtering.log.end_activities import end_activities_filter

end_activities = end_activities_filter.get_end_activities(d_log["int_dec"])
unpayed = end_activities_filter.apply(d_log["int_dec"], ['Declaration REJECTED by EMPLOYEE', 'Declaration SAVED by EMPLOYEE',  'Send
Reminder',  'Request Payment', 'Declaration REJECTED by SUPERVISOR'])
print(f"Unpayed: {len(unpayed)}, Total: {len(d_log['int_dec'])}, % of unpayed wrt total: {round(len(unpayed)/len(d_log['int_dec']),2)}%")
```

✓ 0.5s Python

```
Unpayed: 167, Total: 4952, % of unpayed wrt total: 0.03%
```

Figure 25: Result of splitting InternationalDeclarations log.

```
req4 = log_to_df(d_log_var["req_4_pay"])
nat = log_to_df(d_log_var["nat_dec"])
prefix = "Request For Payment "
req4["concept:name"] = req4["concept:name"].map(lambda x: x[len(prefix):] if x[:len(prefix)]==prefix else x)
prefix = "Declaration "
nat["concept:name"] = nat["concept:name"].map(lambda x: x[len(prefix):] if x[:len(prefix)]==prefix else x)
all = df_to_log(pd.concat([nat, req4]))
```

Figure 26: Code for remove prefix and join.

4.11.1 Merge

To join I removed the prefixes which are respectively “Declaration ” and “Request For Payment ”, I get an event log with 13218 cases. The Figure 26 shows the code used by initially exploiting pandas and then converting it back to log.

4.11.2 Best Model

At this point I launch my procedure to find the best model, the resulting heuristic net is shown in Figure 27. As we can see it is practically the same as the two heuristic nets for DomesticDeclarations and RequestForPay shown in Figure 11 and Figure 14.

4.11.3 Conformance Checking

Also in this case we are going to check through alignments if there are cases in the log not covered by the model generated in the previous section. As the Figure 28 shows, there are no uncovered cases.

4.11.4 Comparing models for single Log with model for Merged log

As I had initially guessed, the process induced by the two logs is the same (ignoring the event prefix). To further confirm the situation you can see the result of conforming Checking in the previous section.

4.12 Predicting overspent declarations

In this section I initially try to carry out an analysis on the PermitLog and InternationalDeclarations columns in order to create a single table through a join, then extract the columns relating to the duration of the requests, the travel time and various available information relating to the budget required for the creation of a prediction model to try to predict whether or not the request will incur an overspent.

4.12.1 Columns analysis and computed columns

International declarations The useful columns are:

- “case: concept: name”: key of the table;
- “case: Permit RequestedBudget”: requested budget;
- “case: Permit ID”: key of the PermitLog table.

To these fields I have added a column that contains, for each declaration, the time required for approval.

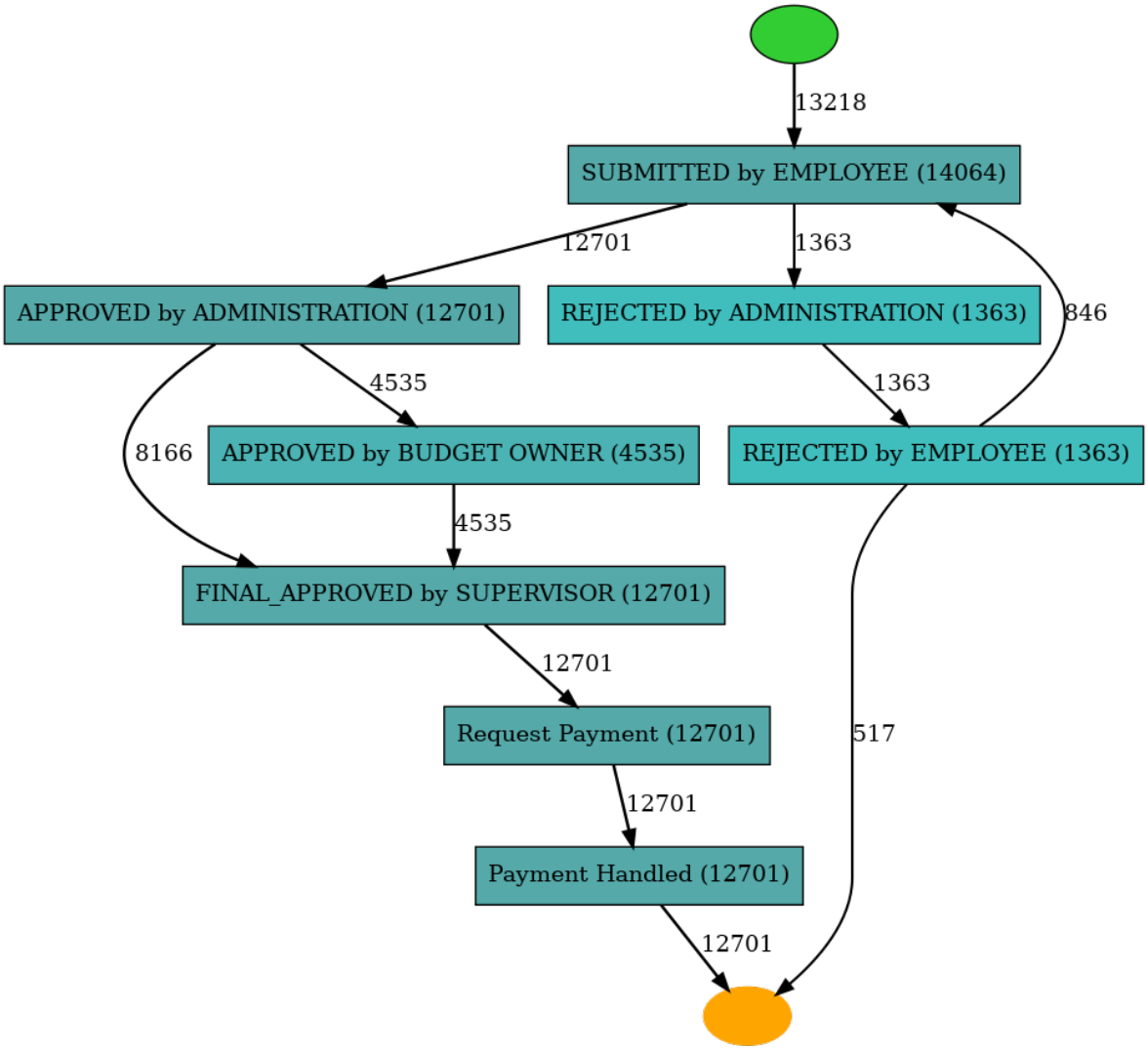

Figure 27: Heuristic net for DomesticDeclarations joined with RequestForPayment.

```
ALIGNMENTS
Number of traces 13218
Number of anomalous traces  0
Number of normal traces  13218
Percentage of anomalous traces 0.0 %
Odds of anomalous traces 0.0
```

Figure 28: Alignments for DomesticDeclarations joined with RequestForPayment.

PermitLog The useful columns are:

- "case: concept: name": key of the table;
- "case: RequestedBudget": requested budget;
- "case: Overspent": flag for budget overrun;
- "case: OverspentAmount": amount of additional money spent;
- "case: TotalDeclared": total of the declaration

Also in this case I added a column that contains, for each declaration, the time required for approval. I also added a column that contains the duration of the trip, i.e. the timestamp difference (in days) between Start Trip and End Trip.

4.12.2 Correlation

```
merged = per_single_row_time.merge(int_single_row_time, on="case:Permit ID")
(merged).corr()
```

✓ 0.1s Python

	case:Overspent	case:RequestedBudget	case:OverspentAmount	case:TotalDeclared	time:timestamp_x	time:timestamp_y	case:Amount	case:Permit RequestedBudget	case:AdjustedAmount	time:timestamp
case:Overspent	1.000000	-0.031160	0.494600	0.238889	-0.025087	-0.047438	0.122257	-0.031160	0.122240	0.016875
case:RequestedBudget	-0.031160	1.000000	-0.283150	0.735002	-0.023285	-0.045129	0.509022	1.000000	0.509057	-0.014122
case:OverspentAmount	0.494600	-0.283150	1.000000	0.189285	-0.093852	-0.060374	0.041477	-0.283150	0.041383	0.029687
case:TotalDeclared	0.238889	0.735002	0.189285	1.000000	-0.073989	-0.113228	0.708102	0.735002	0.708091	0.019212
time:timestamp_x	-0.025087	-0.023285	-0.093852	-0.073989	1.000000	0.160385	0.015067	-0.023285	0.015052	-0.038465
time:timestamp_y	-0.047438	-0.045129	-0.060374	-0.113228	0.160385	1.000000	0.023214	-0.045129	0.023212	0.052105
case:Amount	0.122257	0.509022	0.041477	0.708102	0.015067	0.023214	1.000000	0.509022	0.999998	0.003986
case:Permit RequestedBudget	-0.031160	1.000000	-0.283150	0.735002	-0.023285	-0.045129	0.509022	1.000000	0.509057	-0.014122
case:AdjustedAmount	0.122240	0.509057	0.041383	0.708091	0.015052	0.023212	0.999998	0.509057	1.000000	0.003970
time:timestamp	0.016875	-0.014122	0.029687	0.019212	-0.038465	0.052105	0.003986	-0.014122	0.003970	1.000000

Figure 29: Correlation on merged table.

Then I created the table through the pandas merge function and printed the correlations between the various attributes, shown in Figure 29. As expected, there is a strong correlation between the "case: Overspent" flag and the "case: OverspentAmount" attribute, for this reason I will not use OverspentAmount as the model input as it would result in a bias.

4.12.3 Columns used for train and test model and split train/test

The attributes used are therefore "case: RequestedBudget", "case: Permit RequestedBudget", "case: TotalDeclared" and three timestamps in days, that is: declaration approval time, permit approval time, time between departure and return. To create a train and a test set I used train_test_split from sklearn[8], it takes the feature columns and the label (flag overspent) as input and splits it using 75% of the data as a train and 25% as a test.

4.12.4 Model Selection

```
for est in [10,20,50,75,100,150,200,300]:
    rf = RandomForestClassifier(n_estimators=est, random_state=42, class_weight="balanced")
    rf.fit(x_train,y_train)
    print(est, accuracy_score(y_test, rf.predict(x_test)))
```

✓ 4.5s

```
10 0.7218430034129693
20 0.7389078498293515
50 0.7525597269624573
75 0.7474402730375427
100 0.7440273037542662
150 0.7440273037542662
200 0.7465870307167235
300 0.7559726962457338
```

Figure 30: RandomForest tuning on number of estimators.

I tried to use, again from skleran: KNeighborsClassifier[9], DecisionTreeClassifier[10], GaussianNB[11] and RandomForestClassifier[12]. Through a quick model selection RandomForest was the best in terms of accuracy on the test. On the latter I performed a tuning of the number of estimators used to find the right compromise between performance and size of the model. The Figure 30 shows the code and the results of RandomForest with different numbers of estimators.

[8] train_test_split documentation
[9] KNeighborsClassifier documentation
[10] DecisionTreeClassifier documentation
[11] GaussianNB documentation
[12] RandomForestClassifier documentation

I have therefore chosen 50 as the number of estimators, Figure 31 shows the test accuracy and the confusion matrix.

```
rf = RandomForestClassifier(n_estimators=50, random_state=42, class_weight="balanced")
rf.fit(x_train, y_train)
print("Test accuracy: ",accuracy_score(y_test, rf.predict(x_test)))
ConfusionMatrixDisplay(confusion_matrix(y_test, rf.predict(x_test))).plot()
```

✓ 0.4s

```
Test accuracy:  0.7525597269624573

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f295ea32830>
```

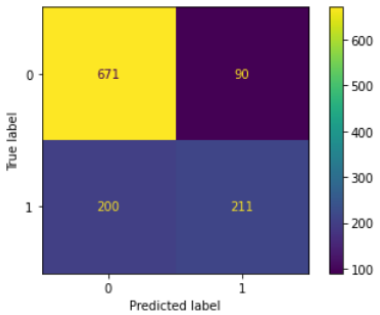

Figure 31: Test confusion matrix of RandomForest with 50 estimators.

The performances indicate that, on average, the model correctly predicts about three out of four cases of label overspent. By placing more emphasis on the True label (i.e. when you go over budget) performance drops to about seven out of ten correct predictions. Having more data available to train the model, we could certainly improve this performance by switching to a somewhat more complicated model than RandomForest, for example a MultiLayerPerceptron.

In the notebook I also tried to use PrincipalComponentAnalysis, I don't report the results because the performance are lower than those just shown.

5 Conclusions

In this report I have based myself on the idea of improving the process and avoiding / reducing any additional costs for the company. Now let's see in detail what I would do to improve the process.

5.1 Problems

The problems to be solved are mainly those concerning double payments, assuming a currency in $, the company has spent an extra amount of 4478912$. Subsequently I am concerned about the requests that have turned out to be anomalous, for example in the InternationalDeclarations log we find trips made without an approval process and subsequently reimbursed to the employee; another similar case can be found in the PermitLog log where payment requests are made without passing the approval process.

5.2 Better process management

Surely all the types of traces present in the various log files are not complicated to model, as can be seen in the dedicated section. I don't know how the process is currently managed but four small software could be implemented to manage the process more sensibly. So a small application for each log file (or type) by combining DomesticDeclarations with RequestForPayment, as shown in the dedicated section.

As you can see in the bottleneck section, it happens that requests remain in intermediate stages of the process in extreme cases even for hundreds of days, this would be avoidable by using a software, as indicated above, that sends reminders to the people concerned.

5.3 Use of the model for overspent and evolutions

In this case we could introduce the model shown before, possibly starting to use it only as a check of the performances shown in the relative section and then training it with a greater number of data. Once a more pre-existing model has been obtained, it could be used "in production", possibly creating a second model capable of predicting the budget necessary for a trip starting from the information of PermitLog and InternationalDeclarations.