Report of *Process Mining* Project

Luca Allegro 1211142
Alberto Bezzon 1211016

Department of Mathematics (Computer Science), University of Padua

The purpose of this report is to present the *Project 1: How much variable is my event log?* carried out for the Process Mining course at University of Padua.

Introduction

The goal of this report is to sum up the Process Mining project. In this project we aim at measuring how much variable are event logs, in terms of variety of behaviour. Computing the variability of a log can indeed be rather useful, for instance, to decide for (a procedural or a declarative) model discovery, or to decide which prediction technique to apply. The are a lot of different ways to measure the variability of an event log, three of them are described in the following sections.

A possible way to measure the variability of an event log is counting the number of variants that it contains. A second possibility is to average the edit distance between each pair of traces in the event log.

The report is organized as follows: in Sections First approach, Second approach and Third approach the three ways to compute variability of a log have been proposed. For each section there are two subsections that explain advantages and disadvantages of the metric described in the section. Section BPI Challenge 2011 contains the results performed by executing the functions in the BPIChallenge2011 log. The last section Project structure describes the project structure and provides instructions for executing the proposed code and related tests.

First approach

The first and easiest way to measure variability of an event log is counting the number of variants that it contains. The function in Snippet 1 computes this metric. We recall the definition of event, trace, process variant and process log: an event is an occurrence of an activity in a particular process instance, a trace is a sequence of events of the same such instance, a process variant is a sequence of process activities and a log is a multiset of such traces.

Code snippet 1: Function compute_variant_variability

```
def compute_variant_variability(log: lg.EventLog) -> int:
"""Compute the number of variants present in the event log

Args:
log (lg.EventLog): The log to examine
```

```

    Returns:
    int: The number of variants present in the event log
    """
    # For each case of the log we construct a tuple with the names of events.
    # Then we collect all of them in a set to remove duplicates.
    # Eventually we compute the length of the set.
    return len(set(tuple(event["concept:name"] for event in case) for case in log))
```

This is the simplest and intuitive way to compute log variability. However, this metric does not consider the size of the log, i.e. big size log are penalized with respect to small size log. There is also a second reason that this way is unhelpful to our goal: it does not consider how variants differ to one another.

Example: Consider the following two logs L_1=[<a,b,c>, <a,b,d>] and L_2=[<a,b,c>, <b,c>, <x,y,z>]. The function returns 2 in both cases, but L_1 has more similar variants rather than L_2 which variants are completely different. This example shows how poorly informative this metric is.

Second approach

In order to obtain a more informative metric to compute log variability, the second approach that we developed consists in averaging edit distance between each pair of traces in the event log. For this purpose, we decided to use the Levenshtein distance (*1*) that is the most well-known string edit distance metric and it is defined by the number of insertions, deletions and substitutions required to convert one string into another. The basic idea to compute average edit distance is, first compute the sum of edit distance of all possible combination of two pair of traces and then divide it by the number of possible combination that is `size_of_log · (size_of_log - 1)`.

Code snippet 2: Function compute_edit_distance_variability

```
def compute_edit_distance_variability(log: lg.EventLog) -> float:
"""Compute the average edit distance (Levenshtein distance) between each
pairs of traces.

```

```
5 This function uses function `eval` of module 'editdistance' because it's
6 implemented in C++ and it is faster than the corresponding implementation
7 in python.
8
9 Args:
10 log (EventLog): The log to examine
11
12 Returns:
13 float: the average edit distance between each pairs of traces
14 """
15 # Create a dictionary which contains variants as keys and its number of
16 # occurrences as values
17 variants_and_counts = Counter(
18 tuple(event["concept:name"] for event in case) for case in log
19 )
20 size_of_log = len(log)
21
22 # For each pair of distinct variants (obtained by 'combinations') we
23 # compute the edit distance and we multiply it for number of
24 # occurrences of the variants.
25 # In the end we sum all of them.
26 sum_of_distances = sum(
27 num_of_items_1 * num_of_items_2 * editdistance.eval(variant1, variant2)
28 for (variant1, num_of_items_1), (variant2, num_of_items_2)
29 in combinations(variants_and_counts.items(), 2)
30 )
31
32 # We multiply the sum of distances by 2 because to add the sum of distances
33 # of each inverted pairs of variants.
34 # In the end we divide it by the number of possible combination of pair
35 # of traces
36 return float(sum_of_distances * 2) / (size_of_log * (size_of_log - 1))
```

Some important caveats are:

- we use variants as an alternative to traces because, instead of calculating edit distance between two traces n times, we calculate the edit distance of two different traces multiplying by the number of repetitions of a trace (`line 26` of Snippet 2).
- `combination` method of the python library `itertools` provides all the combinations of length two of different variants (`line 29` of Snippet 2); the only case not covered

by combinations is when two traces have the same variant but the edit distance is 0, so it does not affect the result (because the sum of all distances does not change summing 0).

- To optimize the calculation of edit distance, we use a C++ library with API for python because the above function has high complexity and with very big log (e.g. *BPIchallenge2011.xes*) it takes very long time[1].

Advantages

The main advantage is that this metric solves the problems of the metric described in Section First approach. In fact:

- it takes into account the size of the log. Unlike the previous metric, this does not penalize big size logs.
- it considers how variants differ one another.

Disadvantages

This method has also the following disadvantages:

- it is very time-consuming for big size logs or long traces due to the fact that it has a high complexity.
- it does not consider the length of each trace, that's why logs with long traces are penalized).

Example: Consider the following logs: L_1=[<a,b>, <c,d>] and L_2=[<a,b,c,d,e>, <a,b,c,f,g>]. The function `compute_edit_distance_variability` returns 2 for both logs but L_1 has a higher variability compared to L_2 that has the first three events equal in both traces (the traces in L_2 are more similar).

[1] We know that computing edit distance between two traces has complexity $O(n \cdot m)$ where n and m are the lengths of the traces. The whole function has a complexity $O(l^2 \cdot n \cdot m)$ where l is the number of variants in the event log and n and m are the lengths of the longest traces.

Third approach

To address the weakness of the metrics based on edit distance, we developed a third way that consists in normalization of edit distance (we found this approach in Section *3.6.1 Edit Distance Between Traces* of article (*2*)). We normalise edit distance by the greatest possible distance between the traces to reflect that a distance of one operation on two very long strings should be considered less significant than on very short strings. In this way, we are able to compare the result with logs with a different average length.

Code snippet 3: Function compute_my_variability

```python
def compute_my_variability(log: lg.EventLog) -> float:
"""Compute the average of normalized edit distances (Levenshtein distance)
between each pairs of traces.

This function uses function `eval` of module 'editdistance' because it's
implemented in C++ and it is faster than the corresponding implementation
in python.

Args:
        log (EventLog): The log to examine

Returns:
        float: a number between 0 and 1 included. The higher the number the
        more similar the traces are
                - 0 : if all traces has nothing in common
                - 1 : all traces belongs to the same variant (are equals)
"""
# Create a dictionary which contains variants as keys and its number of
# occurrences as values
variants_and_counts = Counter(
                        tuple(event["concept:name"] for event in case) for case in log
                        )
size_of_log = len(log)

# For each pair of distinct variants (obtained by 'combinations') we
# compute the average edit distance normalized (divided by longest trace)
# and we multiply it for number of occurrences of the variants.
# In the end we sum all of them.
sum_of_distances = sum(
```

```
        float(num_of_items_1 * num_of_items_2 * editdistance.eval(variant1, variant2))
        / max(len(variant1), len(variant2))
        for (variant1, num_of_items_1), (variant2, num_of_items_2)
        in combinations(variants_and_counts.items(), 2)
)

# We multiply the sum of distances by 2 because to add the sum of distances
# of each inverted pairs of variants
# In the end we divide it by the number of possible combination of pair
# of traces
return 1 - (sum_of_distances * 2 / (size_of_log * (size_of_log - 1)))
```

Advantages

This metric produces a result in the range [0, 1], that is immediately interpretable because it represent the fraction of how much traces are equals. In fact, the higher the number is, the more similar the traces are. This number is 0 if all traces of the log have nothing in common. Conversely, it is 1 when all traces of the log have the same variant. In conclusion, this metric solve the second disadvantage of the previous metric.

Example: Consider once again the two log of the previous example (L_1=[<a,b>, <c,d>] and L_2=[<a,b,c,d,e>, <a,b,c,f,g>] reported for convenience). The function in Snippet 3 returns 0 for L_1 which traces are completely different and 0.6 for L_2 which traces are somehow similar.

Disadvantages

The main disadvantage of this metric is that it does not consider cycles in traces, i.e. traces with a repeated number of event are wrongly penalized. See the following example.

Example: Consider the following two logs: L_1=[<a,b,c,b,c,b,c,b,c,b,c,b,c>, <a,b,c,b,c>] and L_2=[<a,b,c,p,q,r>, <a,b,c,d,e,f>]. The function `compute_my_variability` returns 0.38 for L_1 and 0.5 for L_2; however L_1 is less variable w.r.t. L_2 but it si penalize because the first trace in L_2 is a very long traces.

BPI Challenge 2011

In this section, we report and discuss the results of the three different metrics realized. The following table summarize the results.

Functions	Results
`compute_variant_variability`	981
`compute_edit_distance_variability`	195.88194492325937
`compute_my_variability`	0.14258102346211654

Table 1: Table summarizing the results obtained

As you can see in Table 1, the result obtained applying `compute_variant_variability` is 981. This number indicates that BPIChallenge2011 log has 981 of variants. We verified this data importing the log on Disco and the two number match. The only conclusion, given the size of log (1143), is that most of traces are different one another. Thus, this metric is too simple too capture an interesting variability that involves how different are this variants.

The second result obtained applying `compute_edit_distance_variability` is $\approx$ *195.88*. This number is more informative w.r.t the previous result and it tell us that there is a big average edit distance between pair of traces, hence, the log has an high variability. But you cannot understand how much traces have in common.

The last result obtained by applying `compute_my_variability` is $\approx$ *0.14*. This number gives more information than the others, because it tells how much two traces, random picked, have in common. In this case, the result is 15% which means that traces are different and we can conclude that the log has high variability.

Project structure

The project is structured in the following manner:

- `_init.py`: contains the functions that implements the three different metrics
- `_test.py`: contains the test cases
- `main.py`: contains an example of using the three functions
- directory `resources` contains examples of logs used for testing functions

Instructions

The following steps are the instructions to run the code.

1. Extract *Progetto.zip*
2. Open a terminal inside the folder
3. Install requirements using the following command: `pip install requirements.txt`
4. Run the tests using: `pytest`
5. In file `main.py` you can find an use example and you can run it: `python main.py`

References and Notes

1. "Levenshtein distance." https://en.wikipedia.org/wiki/Levenshtein_distance. Accessed on 2020-02-11.
2. T. S. Christoffer Olling Back, Sren Debois, "Entropy as a Measure of Log Variability," *Data Semantics*, 2019.