Report of *Process Mining* project

Luca Allegro 1211142
Alberto Bezzon 1211016

Department of Science (Computer Science), University of Padua

This document aims to describe the procedure adopted to solve *Project 1: How much variable is my event log?* (probabilmente da rimuovere)

Introduction

This report has the aim of summarize the Process Mining project. In this project we aim at measuring how much variable are event logs, in terms of variety of behavior. 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 section.

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 present report contributes (1)

Project description

First approach

The first and easiest way to measure variability of an event log is counting the number of variants that it contains. Recall the definition of process variant, that is a sequence of process activities, the following fuctions does that we just described:

```
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 way is the simplest and intuitive way to compute log variability. However, this metrics does not consider the size of the log, i.e. big size logs are penalized w.r.t. small size log. Another reason why this way is unhelpful to our goal is that it does not consider how variants differ to one another.

Second approach

The second approach is to average the edit distance between each pair of traces in the event log. For this purpose, we decided to use the Levenshtein distance. Levenshtein edit distance is the most well-known string edit distance metric and 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: first we computed the sum of edit distance of all possible combination of two pair of traces and then we divided by the number of possible combination that is (*size_of_log * (size_of_log - 1)*).

```
def compute_edit_distance_variability(log: lg.EventLog) -> float:
"""Compute the average edit distance (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: the average edit distance between each pairs of traces
"""
# 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 edit distance and we multiply it for number of
# occurrences of the variants.
# In the end we sum all of them.
```

```
sum_of_distances = sum(
num_of_items_1 * num_of_items_2 * editdistance.eval(variant1, 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 float(sum_of_distances * 2) / (size_of_log * (size_of_log - 1))
```

This method has two main advantages: the first is that it takes into account the size of the log and the second is that it consider variants frequencies.

Some important caveats are:

- in the first rows of the function, we use variants instead of traces because we calculate the edit distance of two different traces multiplied by the n (number of repetition of a trace) instead of calculate edit distance between two traces n times.

- (line combination) method combination of the library itertools provide the combination of two different variants; the only case not covered is when two traces belong to the same variant. But, in this case, the edit distance is 0 so it does not affect the result.

- 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.

This method also has the following disadvantages: since it has a high complexity, it is very time-consuming for some large log and it does not consider the length of each traces and the variance. Indeed, logs with longest traces and high variance of length are penalized. (insert example of log).

We have tried another edit distance metric, that is Damerau-Levenshtein edit distance. This distance is similar to Levenshtein edit distance with the addition of another possibly operation (transposition of two adjacent characters) to convet a string into another. This metric performs better than Levenshtein distance only in the case where two traces differs only by order of events at distance one.

Third approach

To address the weakness of edit distance, we developed a third way that consist in normalization of edit distance. 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.

```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)))
```

This metric produces a result in the range [0, 1], that is immediatly interpretable because it represent the fraction of how traces are equals.

BPI Challenge 2011

At first we compute `compute_variant_variability` 981

- we have lots of differents variants, nothing else. Only computing the size of the log (1143) we view that most of traces are differt one another

Then we compute `compute_edit_distance_variability`

- 195.88194492325937
  It seems very high. You cannot undarstand if traces have something in common.

Compute my variability

- 0.14258102346211654: average 15% in common, not so much. It gives us more information than the others. We can say that this log has high variability....

Instructions

1. Extract 'Progetto.zip'
2. Open a terminal inside the folder
3. Install requirements usign 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`

up your figures in the final (post-peer-review) draft, do not include graphics in your source code, and do not cite figures in the text using LaTeX `\ref` commands. Instead, simply refer to the figure numbers in the text per *Science* style, and include the list of captions at the end of the document, coded as ordinary paragraphs as shown in the `scifile.tex` template file. Your actual figure files should be submitted separately.