COBRAS: A tool for analyzing the collaborative writing process including semantic analysis

Montero Lucia
lucia.monterosanchis@epfl.ch

Shengzao Lei
shengzao.lei@epfl.ch

January 16, 2019

Abstract

This report covers the work done during the semester projects of L. Montero and S. Lei under the supervision of S. Håklev at EPFL's CHILI lab.

This work is based on the tool developed by L. Baligand and A. Pace. The aim of the project is to improve the previous analysis tool by including semantic analysis on the text and by introducing new heuristics that allow for a more relevant analysis of collaboratively written documents.

Keywords Collaborative writing, FROG, online editors

1 Introduction

As stated by V. Southavilay et al., Collaborative Writing has received attention since computers have been used for word processing [1]. The many tools available for collaborative writing make the collaboration process easier and have also modified the way in which collaborative text is produced. Particularly, in Education it has been noted that, when using computers, students prefer to make revisions while producing instead of after producing the text [2].

The process of writing is crucial in determining the final text, hence the interest in analyzing this process in Education. Learning about the process of collaboratively writing a text can help us understand different aspects about the roles taken by collaborators (big text additions, small edits...) or how the general structure of the document changes in time (list-like paragraphs, regular paragraphs with a similar length, big semantic differences between paragraphs...), among others.

This information can be used to identify the practices that can potentially lead to a good quality text - and the practices that may decrease the quality of a text. Defining the quality of a text is not in the scope of this project, as we focus on defining the heuristics that allow to correctly capture the relevant information about the writing process.

It would also be useful to be able to do some predictions before the task is finished, using the information gathered about the writing process up until that moment, as a tool for teachers to know how well students are collaborating in time for correcting potential mistakes.

COBRAS (COllaBorative pRocess Analysis with Semantics) allows to learn about collaboratively written texts by looking at the process of writing. The 'S' in the name has been added because one of the characteristics of the tool is the use of Semantic Analysis.

The tool and the data used for the current report have been previously used to obtain the results presented in the submission to the International Conference on Computer Sup-

ported Collaborative Learning (CSCL 2019) [3].

2 Data

The data we use to test our tool consists of a group of pads written by first year Economics students. They took part in an approximately 45 minute long writing session and were asked to work in pairs to complete a task.

The Pads obtained from these sessions include some documents produced during a shorter *warm-up* writing session and there are some Pads that were written by a single author. These Pads have not been considered when carrying out some parts of the analysis.

3 Architecture

The tool presented in this report is written in Python3 and the architecture (Figure 2) is based on the one implemented by L. Baligand and A. Pace [4, 5] (Figure 1).

`ElementaryOperations`, `Operations` and `Paragraphs` have been kept the same as in the previous version of the tool:

- An `ElementaryOperation` represents a writing event and is defined by its position, timestamp, author and text to add or length to delete. It can be only be of type `add` or `del`.

- `Operations` consist of groupings of `ElementaryOperations`. They are classified in five different categories: write, delete, edit, paste or jump. `Operations` also have *context information* that give more information about it. This context information is described in more detail in Section 4.2.

- `Paragraphs` keep track of the lines in the text. Each `Paragraph` may represent either one line of text or one new line character. They are obtained by iterating over the `Pad`'s `ElementaryOperations` and identifying the line that they are modifying. The general idea is that inserting a new line character in the middle of an existing `Paragraph` results in three `Paragraphs`: the first part of the original one, a new line `Paragraph` and a third one representing the second part of the original `Paragraph`. This is explained in more detail in L. Balingand and A. Pace's report.

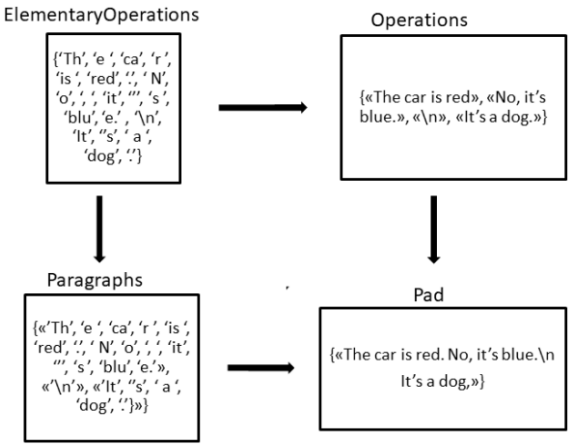

Figure 1: Architecture of the previous system

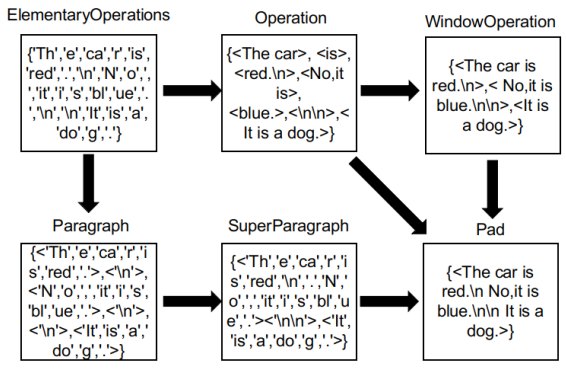

Figure 2: New architecture after adding `WindowOperation` and `SuperParagraph`

After reading the data, it is parsed to create ElementaryOperations. ElementaryOperations are grouped into Operations, which make up a Pad. ElementaryOperations are then used to deduce the Paragraphs of the document. Then, the heuristics are computed based on the Pads' paragraphs, operations and operations' context information.

However, we realized that the majority of students use two new line characters (i.e. two new line characters written one after the other) to separate *paragraphs*. In other cases students use bullet points lists to synthesize information. Although a list's elements are separated by new lines, they should be considered part of one same structure - whereas different paragraphs should not. These observations lead us to introduce the concept of SuperParagraphs, which group several Paragraphs, keeping track of the fact that some Paragraphs are more related to some Paragraphs than to others. They are explained in Section 3.2.

We also introduce the concept of Paragraph indices and identifiers, which are explained in Section 3.1.

Although we can obtain complete words from Operations, they are too fine-grained to contain a complete meaning and cannot be used to carry out the Semantic Analysis. Instead, we group the Operations with the same author and that take place within a specific time interval into a WindowOperation. A WindowOperations is defined by its group number (the group it belongs to), its author, the list of Operations it is composed of, its time interval and its starting and ending timestamp. WindowOperation are explained in Section 3.3.

3.1 Paragraph indices and identifiers

Paragraph indices track the updated position of each text Paragraph in the Pad, and are useful to know the absolute position of the line or lines that have been modified by each Operation.

This information can be used to determine if consecutive Operations modify the same lines of the text, but a Paragraph's index does not necessarily identify the same Paragraph at different points in time. It is interesting to keep track of how Paragraphs are split or merged, how the meaning of a Paragraph changes or where are the new Paragraphs introduced - for this, we introduce *Paragraph identifiers (IDs)*.

3.1.1 Paragraph indices

Paragraph indices are assigned to ElementaryOperations to keep track of the modified *lines*' position in the text. With *line* we refer to a text Paragraphs, as some Paragraphs only contain a new line and are not taken into account when computing paragraph indices.

The position of a line is computed as the position of the text Paragraph in the Pad minus the number of newlines from the beginning until the position of the text Paragraph. Therefore, the paragraph index i_p of a line that corresponds to the text Paragraph in position p in the Pad would be computed as:

$$i_p = p - \mathrm{nl}(0, p-1) \tag{1}$$

where $\mathrm{nl}(p, q)$ is the number of Paragraphs in the Pad between positions p and q (both included) that are new line Paragraphs.

Paragraph indices assigned to ElementaryOperations can give us an insight on whether the changes of an Operation are being done in the same part of the document as the previous or next Operations, or in a different part.

Some questions that indices can give an answer to are: *Is an author modifying one line of text or several ones? Are these lines close to each other? Are several authors working in the same text area?*

Paragraph indices are assigned with the object function

ElementaryOperation.assign_para, which is called by Pad.create_paragraphs_from_ops while Paragraphs are created.

When one or more Paragraphs are added or deleted entirely, the indices of other Paragraphs need to be updated. Ideally, when verifying if two consecutive ElementaryOperations EOperation1 and EOperation2 happen in the same part of the text we would compare EOperation1's indices (after updating the indices with EOperation1's changes) and EOperation2's indices (before updating the indices with EOperation2's changes). Therefore we need to distinguish between the indices before updating the Paragraph indices and the indices after updating the Paragraph indices.

Moreover, some ElementaryOperations may modify several consecutive Paragraphs, so we define an ElementaryOperation's paragraphs indices as two arrays of paragraphs indices (the first one for the paragraph indices **before** taking into account the ElementaryOperation's changes, and the second one for the paragraph indices **after** taking into account the ElementaryOperation's changes).

For example, inserting a new text Paragraph in the beginning of the document shifts the index of all Paragraphs and index 0 would start being used by the newly inserted Paragraph. The indices assigned to the ElementaryOperation that adds a new paragraph in the beginning of the text would be:

- **Before** changes: [0]
  We will insert the text line in the beginning, that is, next to line 0 - if we insert it between two existing lines, this array will contain both lines' indices.

- **After** changes: [0]
  We just inserted paragraph 0.

The rules applied to assign a Paragraph index to an Elementary Operation are shown in Table 1. The definition of i_p is the one from Equation 1. N represents the position of the last Paragraph in the Paragraph array, which may be a new line or text depending on the case. p is the position where we insert the new Paragraph.

Table 1: Paragraph indices assigned to an ElementaryOperation - which always affects exactly one paragraph.

Elem. Operation	i before	i after
Add text to existing text paragraph at *p*		
$(0 \leq p \leq N)$	$[i_p]$	$[i_p]$
Add a new text paragraph at *p*		
Beginning $(p = i_p = 0)$	[0]	[0]
End $(p = N + 1)$	$[i_N]$	$[i_{N+1}]$
Middle $(0 < p \leq N)$	$[i_p, i_p{+}1]$	$[i_p{+}1]$
Add a new new-line paragraph at *p*		
Beginning $(p = 0)$	[0]	[0]
End $(p = N + 1)$	$[i_N]$	$[i_N]$
Middle $(0 < p \leq N)$	$[i_p, i_p{+}1]$	$[i_p, i_p{+}1]$
Add new-line in existing text paragraph at *p*		
$(0 \leq p \leq N)$	$[i_p]$	$[i_p, i_p{+}1]$
Delete paragraph at *p*		
Beginning $(p = i_p{=}0)$	[0]	[0]
End $(i_p = i_{-1})$	$[i_{-1}]$	$[i_{-1}{-}1]$
Between $i_p{-}1$ and $i_p{+}1$	$[i_p]$	$[i_p{-}1, i_p]$

Since we are interested in knowing the paragraph indices for Operations, we compute them based on the paragraph indices that have been assigned to the Operation's ElementaryOperations. The computation is carried out by the object function Operation.get_assigned_para, which receives two arguments: whether we are interested in the paragraphs before or after applying the changes and whether we want the first or the last paragraph involved.

3.1.2 Paragraph IDs

Although it is not being used at the moment, it is interesting to keep track of how Paragraphs are modified, split or merged or where new Paragraphs are introduced. For this purpose we introduce *Paragraph identifiers* (*Paragraph IDs*).

Paragraph *IDs* are computed by looking at the `Pad`'s array of all `Paragraphs` - which contains the `Paragraphs` that have been deleted as well as the ones that currently exist. This way we make sure that IDs are unique.

This functionality is implemented in the class function `Operation.compute_para_id`. This function receives as parameters: one or two existing IDs that will be used to compute the new ID (or IDs, in some cases) and the *relation* between the new and the reference paragraphs.

The possible values of `relation` are initial, merge, split, insert_before, insert_after and insert_between. The value of the new ID or IDs based on the parameter `relation` are:

- `initial`: The first ID is always 0.

- `insert_before (id)`: Paragraph is inserted in the beginning of the text or when inserting a paragraph between two paragraphs if the second one has a shorter ID than the first one.
  Examples:
  {0 → -1}; {-1 → -2}; {2 → 1};
  {1 → 0}; {0.A.B_1_3 → -1}

- `insert_after (id)`: Paragraph is inserted in the end of the text or when inserting a paragraph between two paragraphs if the first one has a shorter ID than the second one. Examples:
  {-1 → -1_0}; {0 → 0_0}; {0_2 → 0_3}

- `merge (id1, id2)`: Lastly, we can obtain a new ID for a paragraph that results from merging two previously existing paragraphs. Examples:
  {0, 1 → (0+1)}; {0.A, 0.B → 0.A};
  {0.C, 0.D → 0.C};
  {0.A, 0.C → (0.A+0.C)};
  {0.A.B, 0.A.C → 0.A.B};
  {0.A, 0.B.C → (0.A+0.B.C)}

- `insert_between (id1, id2)`: Paragraph is inserted between two paragraphs. It compares the lengths of both IDs and inserts the new one either before id2 or after id1. For example, the paragraph inserted between 0 and 1 is 0_0.

- `split (id1)`: If an existing paragraph is split into two, three IDs are generated from the original ID. For example, splitting paragraph 0 results in
  0.A (text), 0.B (new line) and 0.C (text).

These IDs are called **Paragraph History**. However, for when we want to see if two users have contributed for the same paragraph or not, we can use the variable **Paragraph Original**. Paragraph Original is computed based on Paragraph History, but it considers paragraphs 0.A, 0.B, 0.C all as paragraph 0. Also, it would consider paragraph (0+1) as paragraph 0. We can increase the complexity, for example, paragraph history ((0.C+1).C+4) would become paragraph original number 0.

One of the problems that we identified with this implementation of Paragraph IDs is that in some cases IDs become very long. For this reason, when the paragraph ID is longer than the maximum length specified in `config.py`, we generate a completely new ID - sort of *restarting* the ID to a short length again. This solution is not very convenient, as the analysis becomes more difficult.

A possible solution for this would be to come up with a way to track how many parts a paragraph has been split into. With the current implementation, if we decide to merge paragraphs 0.A and 0.B we cannot be sure if the result should be 0 (which would be the case if 0=(0.A+0.B), i.e. there are no paragraphs 0.C, etc), or if paragraph 0 was split into more than 2 parts and therefore (0.A+0.B) would only be a part of paragraph 0.

Using SuperParagraph IDs (the concept of a SuperParagraph is introduced in the next section) instead of Paragraph IDs would also reduce the length of the IDs used, but the implementation is a bit less straightforward.

However, a simplified version has been implemented.

3.2 SuperParagraphs and Paragraphs

In this version of the tool we introduce the concept of SuperParagraph that groups several Paragraphs if certain conditions are met. SuperParagraphs allows to keep track of the likely relationship that exists between lines of text that are only separated by one new line character.

SuperParagraphs contain a boolean that states whether the SuperParagraph consists of only new lines or not, which could be understood (in a simplied way) as the result of doing the logical AND operation of the boolean that states whether each of the SuperParagraph's Paragraphs are new lines. Hence, this boolean defined for SuperParagraphs determines the possible Paragraphs that it can contain.

- New line SuperParagraph:
  It may contain any number $N \geq 2$ of new line Paragraphs.
  There cannot be two new line SuperParagraphs one after the other, as in that case they should be merged. Therefore, a new line SuperParagraph can only be surrounded by text SuperParagraphs.

- Non-new line (*text*) SuperParagraph:
  It may contain any number of Paragraphs, either text or new line, as long as there are not two (or more) new line Paragraphs one after the other. Analogously to the previous case, they can only be surrounded by new line SuperParagraphs.

Object function Pad.create_paragraphs_from_ops(), which is the function that updates Paragraphs, is the one that updates SuperParagraphs as well.

Table 2: Rules for updating SuperParagraphs after inserting or deleting a Paragraph. **NL** stands for *New Line* and **T** stands for *Text*.

Para	SuperPara	Action
Any	None	Insert in new T SuperPara
T	NL	Insert; Run Listing 1
T	T	Insert
NL	NL	Insert
NL	T	Insert; Run Listing 2

The rules that determine how to update SuperParagraphs after inserting or deleting a Paragraph are shown in Table 2.

After inserting a text Paragraph in a new line SuperParagraph we also need to check which are the changes that we will need to do in the SuperParagraph, as described in Listing 1.

Listing 1: After inserting text Paragraph in new line Superparagraph.

```
sp = modified_superparagraph
p = paragraph_to_insert
sp1, sp2 = sp.split_at(p.position)
if len(sp1) >= 2 or len(sp2) >= 2:
  remove(sp)
  if len(sp1) >= 2:
    insert(sp1)
  insert(new_sp(text, content=p))
  if len(sp2) >= 2:
    insert(sp2)
else:
  sp.insert(p); sp.is_newline=False
```

In a similar way, after inserting a new line Paragraph in a text SuperParagraph we need to check whether we will need to split the SuperParagraph or merge it with other SuperParagraphs, as described in Listing 2.

Listing 2: After inserting new line Paragraph in text Superparagraph.

```
sp = modified_superparagraph
p = paragraph_to_insert
if (len(sp.paras) == 1 and
  sp.ends_by_newline):
    sp.insert(p); sp.is_newline=True
```

```
elif ((p.at_start_of(sp) and
  sp.starts_by_newline) or
  (p.at_end_of(sp) and
  sp.ends_by_newline)):
  insert(new_sp(newline, len=2))
elif p.at_middle_of(sp):
  sp1, sp2 = sp.split_at(p.position)
  if sp1.not_empty and sp2.not_empty:
    remove(sp)
    insert(sp1)
    insert(new_sp(newline, len=2))
    insert(sp2)
  else:
    insert(new_sp(newline, len=2))
else:
  sp.insert(p)
```

3.2.1 Discussion about SuperParagraphs: Issues and possible solutions

Although using SuperParagraphs seems to work for the majority of the pads analyzed, there is still an inconsistency in the number of new line characters used to separate paragraphs. In some cases, both the separation between paragraphs and the separation between list elements is one new line character. This makes it difficult to apply the concept of SuperParagraphs - at least with the current implementation. These cases can be identified by a small number of SuperParagraphs and a very large average number of Paragraphs per SuperParagraph. A possible improvement for these cases would be to assume SuperParagraphs to be the same as the Paragraphs, unless there seems to be a list (which can be identified by having several short Paragraphs that often start with one of the characters that are typically used for lists).

A correct paragraph separation was crucial at this point for carrying out a relevant semantic analysis comparison across paragraphs. For this reason, we have implemented the object function Pad.get_paragraphs_text() that first obtains the text split by double new lines and then compares the average paragraph length obtained with a configuration parameter that we considered appropriate for the texts we are analyzing (1000 characters, defined in the configuration file config.py). If the average length is longer than the one defined in the configuration file, we assume that the authors used single new line characters to separate their paragraphs and split the text by single new lines instead. In both cases we delete whitespace lines.

Another issue that we came up with when dealing with SuperParagraphs is the fact that lines consisting only of whitespace are considered as text Paragraphs, and therefore we may end up grouping in a single SuperParagraphs lines of text that the authors intended to separate in different paragraphs. The solution to this is in the way Paragraphs are treated: whitespace Paragraphs behave like text Paragraphs when being updated, but should be considered like new line Paragraphs when grouped into SuperParagraphs. This is not completely straightforward, as a SuperParagraph containing a whitespace Paragraph may need to be split and recombined if non-whitespace characters are inserted in its whitespace Paragraph.

Another of the issues we identified happens as a result of the behaviour of many text editors that automatically insert a new bullet point after inserting a new line character in a line with a bullet point. In many cases, authors leave this new bullet point entry empty - using it as if it were an empty line - but our program considers it as a line containing text.

3.3 WindowOperation

A WindowOperation is a group of Operations within a time interval that have the same author. We introduced WindowOperations because we would like to apply semantic analysis on what users' wrote and sometimes we can only obtain one or two words from

Operations, not enough to constitute meaning.

3.3.1 Build WindowOperation

Function Pad.BuildWindowOperation creates the WindowOperation. It takes time_interval (default 100 second) as a parameter, which needs to satisfy some conditions that we will explain later. The function traverses all Operations inside the pad and computes their group number, which is then used to assign each Operation to its corresponding group. Within each group, Operations are sorted and split by author, resulting in WindowOperations. The group number (num) can be computed as:

$$\text{num} = \lceil (\text{end\_time} - \text{start\_time})/t \rceil \qquad (2)$$

where t is the time interval, end_time is time when the Operation ends and start_time is the starting time of the Operation's Pad.

3.3.2 Select time interval

Function BuildWindowOperation uses parameter time_interval, which has a great impact on applying semantic analysis. Too short time interval may lead to not having enough words that have a meaning, whereas too long time interval may result in not having enough points in time for analyzing of collaborative writing behavior in time.

We first tried to choose the best time interval by counting the number of Operations or string length inside one WindowOperation, but we found that there are some Operations which only add a new line or punctuation. Since we would like to do semantic analysis to the WindowOperation's text, we are interested in the WindowOperation's meaningful text and these Operations should not be considered. Therefore, it makes sense to evaluate the best time interval by counting the number of words inside the WindowOperation. We consider a WindowOperation to be valid if it has enough words. A Pad is considered valid if the Pad has enough valid WindowOperations. Function SelectTimeInterval labels whether the Pad is valid or not for a given list of time intervals to be considered. It also takes the threshold number of words needed for a valid WindowOperation and the threshold of the group number in a Pad as parameters. It returns, for each Pad, a list containing the 0 and 1 (valid) of Pad for each time interval.

3.3.3 Fitting similarity distribution

Users' behavior is quite complex when they are writing together. However, we can simplify collaborative behavior by using some metrics to measure it and try to discover potential common patterns. After choosing a time interval and building the corresponding WindowOperations, we can recover the text (see Section 5.4 for more details) and apply a pre-trained model (Section 5.1) to compute the similarity (see Section 5.5) between different authors' WindowOperations in the same group. From here on we will refer to the different authors' WindowOperations in the same group call as WindowOperation pair. After obtaining the sequence of similarity values for all Pads and for each authors pair, we can plot the similarity distribution and fit the similarity distribution, to find the pattern or the *law of similarity*. We used linear fitting to fit similarity distribution, so that we can classify the collaborative behaviours in:

- Writing converges towards the same topic gradually.
- Writing topic diverges gradually.
- No change in the writing topic similarity.

Function PlotSimilarityDistribution takes care of fitting similarity distribution.

4 General Heuristics Computation

There have been many different approaches for analyzing Collaborative Writing (CW). Although most of the research has been traditionally done on the final result, there have been several approaches that focus their analysis on the process of writing. In particular, the WriteProc framework presented by V. Southavilay et al. in uses both process mining and semantic analysis [1].

Lowry et al. present in [6] a common nomenclature and taxonomy for CW works. They define the common iterative activities of CW, which are: brainstorming, outlining, drafting, reviewing, revising and editing. Several participants' roles are introduced by Lowry et al. in [6] as well, including writer, editor, scribe or facilitator. Depending on the task that students are working on, some of these activities and roles may not be relevant. In our case, it is very unlikely to find a student taking the role of scribe, as all interaction was done through the computer. Nevertheless, we would expect students to often take the roles of writer and editor. For a similar reason, we would not be able to find the reviewing activity either.

In this section we will define different heuristics and measures that may help us identify different roles taken by students or different collaborative activities, that we might be able to relate to the ones proposed by Lowry et al. For example, as explored in [7], an increase in the number of lines together with a decrease in the length of such lines could suggest an outlining activity.

There are many other specific questions that we may want to answer, such as which areas of text have been modified the most, if students have worked at the same time, or whether several students have contributed to the same parts of the document. The heuristics that were proposed in the previous version of our tool [5] contained some information that could be relevant for some of these questions, but we can see that other researchers have been interested in this topic as well [8].

4.1 Pad heuristics

In the previous version of the tool [5] the heuristics or metrics that are summarized in Table 3 were defined in order to assess the quality of a `Pad`. From here on, we refer to this kind of heuristics as *Pad heuristics* because they are computed taking into account the `Pad`'s `Operations`, as opposed to *Operation heuristics* that will be introduced in the next section.

Table 3: Pad heuristics. Column *Window* indicates whether the scores can be computed ignoring Operations before a given timestamp.

Heuristic name	**Window**
Alternating	No
Day break	Yes
Short break	Yes
Overall type	Yes
Proportion	No
Synchronous	Yes
User type	Yes
User proportion per paragraph	No

We can compare the scores of a `Pad`'s heuristics at different points in time by getting versions of the `Pad` at the timestamps of interest, using the object function `Pad.pad_at_timestamp`. When calling function `run_analytics.py` with argument `--generate_csv_summary` a tsv file is printed containing one line per `Pad` with the values of the previous and new *Pad heuristics*. These Pad heuristics are computed by calling the object function `Pad.compute_metrics`.

The metrics implemented in the previous version of the tool take into account all `Operations` within a `Pad`, from the first timestamp until the last one. However, these scores *dilute* the impact of each `Operation` because,

as the number of `Operations` in the `Pad` increases, the new `Operations` have a smaller impact on the scores.

To solve this issue, we can specify a starting timestamp so that `Operations` that take place before the specified time are not taken into account when computing the scores. This, together with the previously explained strategy of getting versions of the `Pad` that end at different timestamps, allows us to define time windows of interest. This can be applied to the scores that are computed from the `Pads`' `Operations`, but not to the ones computed from `Paragraphs` information. Table 3 summarizes which scores can be computed for specific time windows.

We also defined some new heuristics that capture `Paragraphs` information in a different way. These new pad heuristics are summarized in Table 4.

Table 4: New Pad heuristics (not normalized). Column *Wnd* (Window) indicates whether they can be computed ignoring Operations before a given timestamp.

Heuristic name	**Wnd**	**Details**
Length	No	Contribution
Length all	Yes	Contribution
Length all write	Yes	Student effort and contribution
Length all paste	Yes	Contribution
Added chars	Yes	Contribution
Deleted chars	Yes	Contribution
Para avg length	No	Text structure
Superpara avg length	No	Text structure
Avg paras per superpara	No	Text structure

4.2 Operation heuristics

As mentioned previously, `Operation` heuristics only take into account the context of the `Operation`. They are shown in Table 5, together with other relevant information.

Table 5: Operation information (heuristics not normalized).

Information	**Details**
Author	Name of the Operation author.
Position Start	First text position edited.
Position End	Last text position edited.
Time Start	Operation starting timestamp.
Time End	Operation ending timestamp.
Atomic Op Count	Number of atomic ops.
Type	Operation type.
Text Added	Number of chars. added.
Deletion Length	Number of chars. deleted.
Paragraph	Paragraph index.
Paragraph History	Paragraph ID, see 3.1.2)
Paragraph Original	Simplified Para. ID, see 3.1.2)
Superparagraph ID	Computed as in 3.1.2.
Coauthor Number	Number of other authors in SuperParagraph.
Proportion Pad	Ratio of Operation chars to Total Pad chars.
Proportion Paragraph	Ratio of Operation chars to Paragraph chars.

5 Semantic Analysis

When analyzing students' collaborative behaviour, the meaning of what they are writing can give us very useful insights. In order to apply machine learning algorithms to obtain these insights into the semantic information of the texts or in order to project the text into a latent d-dimensional space we need to use semantic analysis.

5.1 Pre-trained models

Due to the relatively small size of our data, as well as computational constraints and time limitations, it would be impossible for us to build a good performance model. However, there are many pre-trained models available, which have already been trained on large datasets and make semantic analysis faster and less expensive (data and computation-wise).

There are several pre-trained models available online that have been shared by different Machine Learning research groups. We chose three different pre-trained models and compare their performance on our dataset.

5.1.1 doc2vec

Le and Mikolov proposed doc2vec as a extension to word2vec to learn document-level embeddings [9]. Then, Jey Han and Timoth verified doc2vec model's performance by comparing it to two baselines and two state-of-the-art document embedding methodologies and shared their pre-trained models [10]. We downloaded two pre-trained doc2vec (DBOW method) models that have been trained on the English Wikipedia and Associated Press News websites respectively.

5.1.2 sent2vec

A novel, computationally efficient, unsupervised, C-BOW-inspired method was introduced by Matteo and approach for training and inferring sentence embedding[11]. We use two of their pre-trained sent2vec models, which were trained on the English Wikipedia and Toronto books respectively.

5.1.3 Spacy

Spacy is an open-source python library for industrial-strength natural language processing. It offers several pre-trained models such as *en_core_web_lg*, which is an English multi-task CNN trained on OntoNotes, with GloVe vectors trained on Commom Crawl.

5.2 Test Dataset

In order to compare the performance of each pre-trained model on our dataset, we manually select some data as the test dataset and split it into three groups of text pairs depending on whether we considered them to be similar, medium similar or non-similar to each other. The text pairs of similar data were created by translating English text into Chinese and then translating Chinese to English. As for non-similar data and medium similar data, the text pairs are selected from each document of our dataset.

5.3 Data Preprocessing

There are many characters and groups of characters that are useless for semantic analysis such as punctuation, whitespace, very common words, numbers, URLs and so on in the text. Function `cleanedText` removes common words (Stoplist in Spacy), punctuation characters, numbers and URLs, then changes everything to lowercase and stores the prototype of each word. In addition, we defined different text processes that were applied to the text after our first processing, that were necessary for using the pre-trained models afterwards because of each pre-trained model's special requirements.

5.4 Text Recovery

Recovering text based on `WindowOperation`, `SuperParagraph` or `Pad` is the most essential step before we do semantic analysis and we will use recovery text in `WindowOperation` as expample in following introduction. Recovering text is quite difficult in `WindowOperation` since we separated Operations by authors when creating `WindowOperations` from `Pad`, and collaborators' interaction information between cannot be retained. For example, when user1 wrote 'year' in one `Operation` and user2 found this word should be plural form and then user2 add a `Operation` which added 's' at the end of 'year'. This interaction is ignored when we only record one author's `Operations` in one `WindowOperation`. Fortunately, this kind of situation only happened by chance and it does not have a great impact on sentence embedding when we have enough words in one `WindowOperation`. In the beginning we tried to recovery text by `ElementaryOperation`, in which we needed to real-time trace the position of each `ElementaryOperation`. And tracing `ElementaryOperations`' position is not easy since you do not have completed position change information. For example, the recovered text will be wrong in use1's `WindowOperation` when use2 modi-

fied(add or delete) a letter in the medium of text, since the ElementaryOperations' position is based on total Pad and user1 do not have user2's ElementaryOperations' information when two users ElementaryOperation are in different WindowOpeartions. Finally we tried to recover text base on Operation since we can obtain completed words from each Operation. We created a function getOpText to recover text and then sorted the Operations by start time and recover WindowOperation's text by Operation's text and its position. As for Superparagraph, function pad_at_timestamp creates a copy of a pad until a specific time and we can then retrieve the Pad's text and separate it to obtain the superparagraphs' texts. Function PreprocessOperationByAuthor groups the Operations by authors and recovers their text, to see the similarity between different users in the total Pad.

5.5 Similarity of WindowOperation and SuperParagraph

After retrieving the text to be analyzed and choosing the pre-trained model we can carry out the text analysis. We use cosine similarity between two vectors of texts, which is one of the most commonly used metrics for measuring similarity. When there is only one valid text in the pair, which means that there is only one non-zero text vector, we set the similarity value to 0. We apply the similarity metric on WindowOperation in order to find whether collaborators' writing topics tend to be gradually more similar. We will also look at the comparisons between SuperParagraphs.

6 Results

We will select the best time interval based on the requirements for the Semantic Analysis and then we will keep the consistency by using the chosen value for the entire analysis.

6.1 Time interval selection

Before selecting the best time interval we need to set two parameters: the threshold number of words in one WindowOperation and the threshold number of valid WindowOperation groups in one Pad (see Section 3.3.2). We set the minimum number of words to 10 according to the average length datasets STS2014-Forum (10.12 words), SICK2014 (9.67 words) [11]. Also, in order to fit the similarity distribution we need enough similarity values in one Pad, so we set 5 as the minimum value of valid WindowOperation groups in a valid Pad. As we can see in Figure 3, the candidate time intervals are from 60s to 380s. We chose 180s as it is the time interval with the most valid Pads.

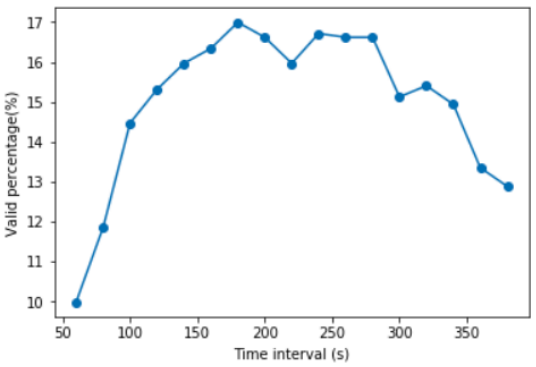

Figure 3: Valid Pad percentage for different time interval durations

6.2 Heuristics Analysis

We first look at the effect of considering only Operations after a specific start timestamp for computing Pad heuristics and compare it to the score that considers Operations from the start. Then we see how Paragraph number and Superparagraph length evolve in time and we use time series analysis to see whether we could use these metrics to predict how much longer the students will take to finish the task.

6.2.1 Comparison between full Pad and window scores

We presented the need for modifying the scores that we were computing in the previous version of the tool so that the changes introduced by new `Operations` have a more noticeable impact. However, we need to keep in mind that some `Operations` may not be considered when using the window-contained Operations approach, particularly if some `Operations` are not fully contained within a single time window.

Figures 4 and 5 show the overall scores for writing operations, both using the defined time interval of 3 minutes. These Figures show in gray the boxplots of the score values per window for all pads, and in blue the average value with the standard deviation error bars.

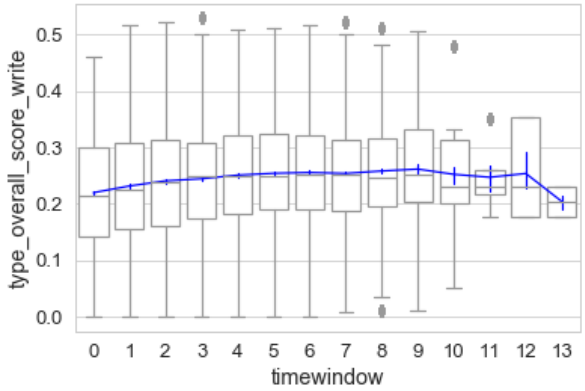

Figure 4: Overall score for write type using 3 min time windows and considering Operations from the beginning of the Pad.

In Figure 4 the line is smoother, as we are considering all the Operations from the beginning of the Pad until the end of the current window and therefore at every window we are averaging any possible changes with the rest of the Pad. Figure 5 shows the score changes from one window to another more clearly, as it is only considering the Operations within the time window.

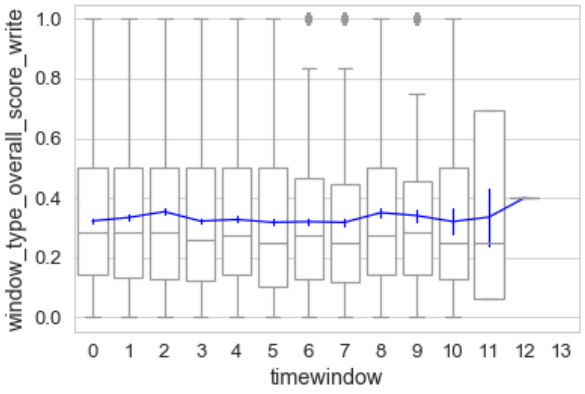

Figure 5: Overall score for write type using 3 min time windows and considering Operations from the beginning of the window (new implementation).

6.2.2 Paragraph and Superparagraph length in time

In this section we decided to look at the number of paragraphs per superparagraph and at the average paragraph length and how these metrics change in time. We decided to try a simple model to see how well it could fit our data, and the results are shown in Figures 6 and 7. In both cases we decided to fit the values observed that correspond to windows 0 to 13, as windows 14 to 17 have less observations (because most paragraphs last 45 minutes or less, i.e. they last only 15 timewindows). However, we plot the results for windows 15 to 17 as well.

From the two metrics obtained here we could obtain the value of the average Superparagraph length, as it would be the product of both.

We see that the tendency does not seem to converge, so from these results we would not be able to say whether we could predict when the task will be finished. However, we need to keep in mind that the average for first windows has been computed over more pads than for the last windows. As we can see, up until timewindow 11 (which would correspond to minute 33 of the task), it seems like the val-

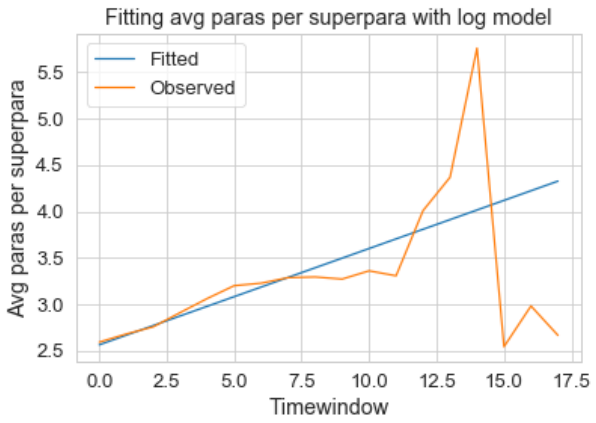

Figure 6: Average number of Paragraphs per Superparagraph. The linear model fitted has slope 0.1 and offset 2.55.

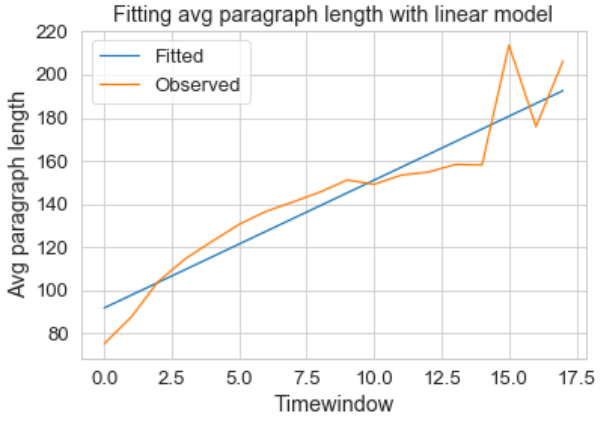

Figure 7: Average length in characters per Paragraph. The linear model fitted has slope 5.9 and offset 91.8.

ues would converge. For this reason, it may be interesting to repeat this analysis after implementing a different WindowOperation split that takes into account the total duration of the pad.

6.3 Semantic Analysis

Figure 8 shows the similarity value of three datasets. As we can see from those similarity values of each models for different similarity paragraph texts. `doc2vec` models can clearly recognize different similarity paragraph texts, but it has lower values even the the paragraph looks very similar and took the longest time to compute similarity. As for the fastest `spacy` model, it performed bad since the values of them are always very high even the two texts are totally different. And `sent2vec` model can correctly recognize them . Additionally, we can discover the different performances of those models when applying to same paragraph pair. `Spacy` model always have the highest values and `doc2vec` always have the lowest values. Overall, for each two paragraph pairs, the similarity difference(increase or decrease) between these two pairs are almost the same between different models. We finally chose `sent2vec` as our pre-trained model since it took less time and performed well in different similarity datasets.

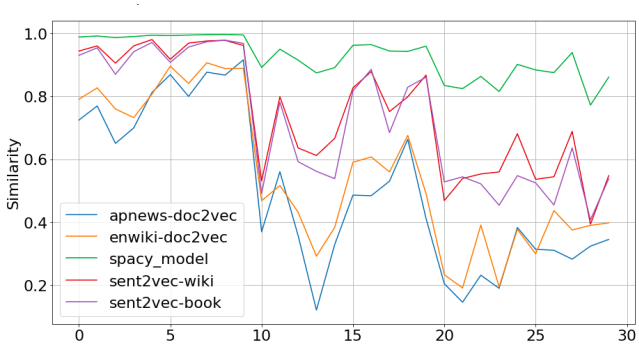

Figure 8: The evaluation similarity value of datasets

6.4 Fitting similarity distribution

We could roughly separate our dataset into three categories depending on the trend of the semantic similarity distribution (increasing, stable and decreasing). However, at this point it is difficult to tell whether this classification would give us useful insights on the collaborative behavior of the students.

Figure 9 shows the similarity distribution of one valid `Pad`. We fitted a linear model to the observed data. From the Figure, we can easily see an increasing trend in topic similarity between this `Pad`'s authors, reflected in the positive slope of the fitted model.

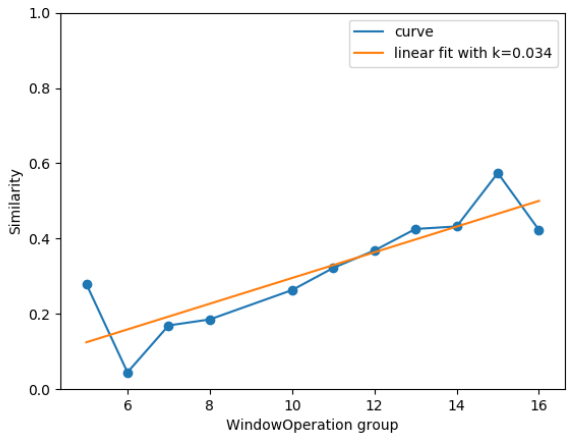

Figure 9: Fitting similarity distribution

6.5 SuperParagraph similarity visualization

When applying semantic analysis on `SuperParagraphs`, we first plot the heatmap for similarity values between all `SuperParagraphs` to get an overview of the document semantics. The heatmap in Figure 10 shows the inter `SuperParagraphs`' similarity in one document from our dataset.

We can conclude that `SuperParagraphs` 0 and 13 are the most different pair and `SuperParagraphs` 1 and 12 are the most similar pair. We include below the texts of those 4 `SuperParagraphs`, as it can be used to verify if we agree with the results and to get some intuition on how the results are computed:

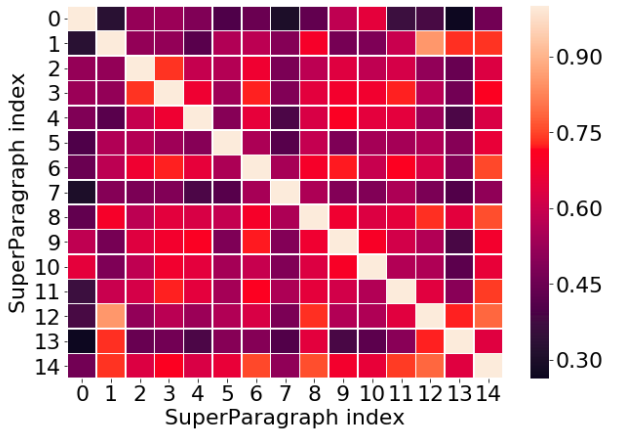

Figure 10: Inter `SuperParagraph` similarity heatmap

`SuperParagraph 0`:

Your task is to act as an advisor to an official within the science ministry. You are advising an official on the issues below. The official is not an expert in the area, but you can assume they are a generally informed reader.
They are interested in the best supported claims in the documents. Produce a summary of the best supported claims you find and explain why you think they are.
Note you are not being asked to "create your own argument" or "summarise everything you find" but rather, make a judgement about which claims have the strongest support.

`SuperParagraph 13`:

CONCLUSION:
Currently, glyphosate dominates crop weed control in soybean, maize, canola and cotton in North and South America. Consequently, throughout large areas, glyphosate reliance without diversity in weed control practices is a strong selection pressure favoring the evolution and eventual domination of glyphosate-resistant weed populations.

SuperParagraph 1:
INTRODUCTION :
Glyphosate is the active ingredient in Roundup agricultural herbicides and other herbicide formulations that are widely used for agricultural, forestry, and residential weed control.
It is one of the most widely-used weedkillers in the world, used by farmers, local government and gardeners, as well as sprayed
extensively on some genetically modified crops imported into Europe for use as animal feed.
Glyphosate, N-(phosphonomethyl)glycine, is the most extensively used herbicide in the history of agriculture. Weed management programs in glyphosate resistant (GR) field crops have provided highly effective weed control, simplified management decisions, and given cleaner harvested products

SuperParagraph 12:
6.Glyphosate effects on diseases of plants
Glyphosate, N-(phosphonomethyl)glycine, is the most extensively used herbicide in the history of agriculture. Weed management programs in glyphosate resistant (GR) field crops have provided highly effective weed control, simplified management decisions, and given cleaner harvested products. However, this relatively simple, broad-spectrum, systemic herbicide can have extensive unintended effects on nutrient efficiency and disease severity, thereby threatening its agricultural sustainability. Given that recommended doses of glyphosate are often many times higher than needed to control weeds, we believe the most prudent method to reduce the detrimental effects of glyphosate on GR crops will be to use this herbicide in as small a dose as practically needed. Such a frugal approach will not only curtail disease predisposition of GR crops, but will also benefit the grower and the environment.

We also plotted the time evaluation of inter Superparagraph similarity. In Figure 11, each picture shows the evolution in time of the similarity between one SuperParagraph and all SuperParagraphs. The zero values at the beginning indicate that this SuperParagraph was not created yet.

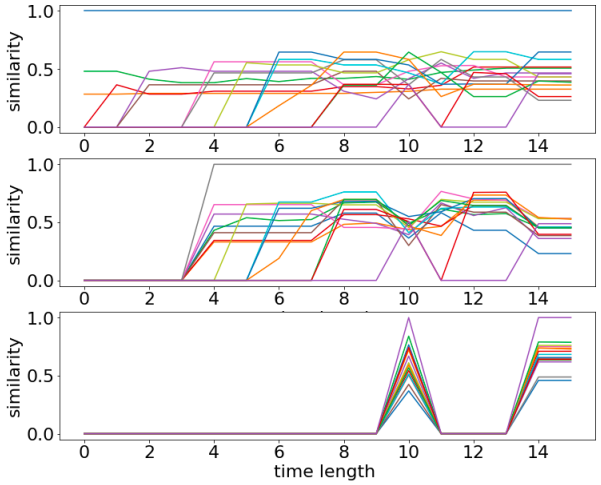

Figure 11: Inter SuperParagraph Similarity through time

7 Conclusion

Compared with previous tools we were based on, we created WindowOperation in order to apply semantic analysis on our tool, and a better time interval will help us to analyze users' data easier. In addition, we tested several pre-trained models and found that sent2vec model has the best performance in our dataset. We can use it to analyze the similarity between SuperParagraph pairs to see if they are related, or apply it to different authors' WindowOperations. We can then obtain the slope of linear model that fits the similarity of authors' WindowOperations for hints on the collaborators' writing behavior.

Regarding the heuristics implemented for the pad we could see an improvement by redefining the scores that were introduced previously so that they considered only the operations in the last window, as the latter operations are no longer averaged over the total pad and we can see their effect more clearly.

We defined some new measurements regarding the lengths of the Paragraphs and

`SuperParagraphs` (that usually correspond with text lines and text paragraphs, respectively), but with the current timewindow implementation we were not able to reach any conclusion. However, we saw that these measures could potentially give us some interesting insights on how the pad's structure changes in time, suggesting that it may be worth it to explore them further.

8 Future work

There are several constrains in our study, mainly due to the limitation of valid data. Besides, the functions defined in our system could be improved by adjusting the parameters with more precision. There are several aspects of the tool in which further work can be carried out, the main ones being:

- **WindowOperation**: The code could be refactored so that the current implementation of WindowOperations is reused for computing pad heuristics in a defined time window. We could also consider reaching a balance between constant-length windows (the approach that we used) and windows that have a length proportional to each Pad's total length as, for instance, in some cases some students may work (or type) faster than others and finish the task earlier.

- **Pad similarity slope**: it could be interesting to discover whether the similarity slope contributes to users' collaborative behavior and try to correlate it with document equality evaluation. Based on those results, we could evaluate whether it would be useful to apply machine learning method to predict the collaborative behavior based on these values.

- **Collaborative behavior pattern**: Analyzing users' collaborative behavior is our main target, as we would like to be able to identify common patterns in order to identify successful collaboration strategies or even predict how much longer will the students take to finish the task. How can we find a way to represent the behavior based on all the metrics we compute is a complex task and could be the main research direction for the future work.

- **Paragraphs and SuperParagraphs tracking**: We implemented paragraph IDs because we were interested in tracking how specific paragraphs could be shifted down by inserting new paragraphs on top, or on how paragraphs were merged and split. However, the paragraph IDs computed seem overly complex. To be able to track specific paragraphs or superparagraphs in the future, the way how we compute this IDs should be improved so that the analysis is more accurate.

References

[1] Vilaythong Southavilay, Kalina Yacef, and Rafael A Calvo. Writeproc: A framework for exploring collaborative writing processes. *ADCS 2009*, page 129, 2009.

[2] Amie Goldberg, Michael Russell, and Abigail Cook. The effect of computers on student writing: A meta-analysis of studies from 1992 to 2002. *The Journal of Technology, Learning and Assessment*, 2(1), 2003.

[3] Stian Håklev, Jennifer K Olsen, Kshitij Sharma, Lucia Montero Sanchis, Lei Shengzhao, and Simon Knight. Detecting synchronous collaborative writing strategies through bottom-up analysis. In *CSCL 2019 Full Papers, Short*, 2018.

[4] Adrian Pace, Louis Baligand, Stian Håklev, Jennifer K Olsen, Nore de Grez, and Bram De Wever. Quantifying collaboration in synchronous document edit-

ing. In *Proceedings of the 3rd Swiss Text Analytics Conference (Swiss-Text 2018)*, 2018.

[5] CHILI Lab EPFL. FROG-Analytics. https://github.com/chili-epfl/FROG-analytics, 2018.

[6] Paul Benjamin Lowry, Aaron Curtis, and Michelle René Lowry. Building a taxonomy and nomenclature of collaborative writing to improve interdisciplinary research and practice. *The Journal of Business Communication (1973)*, 41(1):66–99, 2004.

[7] Vilaythong Southavilay, Kalina Yacef, and Rafael A Callvo. Process mining to support students' collaborative writing. In *Educational Data Mining 2010*, 2010.

[8] Vilaythong Southavilay, Kalina Yacef, Peter Reimann, and Rafael A Calvo. Analysis of collaborative writing processes using revision maps and probabilistic topic models. In *Proceedings of the third international conference on learning analytics and knowledge*, pages 38–47. ACM, 2013.

[9] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196, 2014.

[10] Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*, 2016.

[11] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi. Unsupervised Learning of Sentence Embeddings using Compositional n-Gram Features. In *NAACL 2018 - Conference of the North American Chapter of the Association for Computational Linguistics*, 2018.