CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

On-Line New Event Detection Using Minimal New Sets

Master's thesis in Computer Science

Örn Guðjónsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Abstract

On-line New Event Detection can be used to monitor multiple news streams and determine when a new newsworthy event that might warrant further attention occurs, thus helping the user not to miss important information while helping him sift through the vast amount of information available.

In this thesis we explore a novel approach for On-line New Event Detection based on computing minimal new sets of small sizes from new stream documents with regard to previously seen documents. We present an algorithm that outputs sets of words likely to be indicative of new events which users easily can browse through and detail several different approaches to select the sets most likely to be informative of new events. In addition we examine the practicality of our approaches and compare them to the most prominent current techniques.

Acknowledgements

I'd like to thank my supervisor, prof. Peter Damaschke, for the idea for this project and assistance and insights throughout the course of this project.

I'd also like to thank my family for supporting and encouraging me throughout the course of this project. I would especially like to thank my wife, Elfa Björk, for her patience and support without which this thesis would never have been completed.

Contents

1 Introduction **4**

2 Background **6**
- 2.1 What are events? . . . 6
- 2.2 Related works . . . 7

3 Method **10**
- 3.1 The base idea . . . 10
- 3.2 Enumeration . . . 11
- 3.3 Pre-processing . . . 12
- 3.4 Extensions . . . 12
  - 3.4.1 Filtering common or unimportant words . . . 12
  - 3.4.2 Prioritizing words that occur in many minimal new pairs 14
  - 3.4.3 Prioritizing words that appear in close proximity . . . 14
  - 3.4.4 Splitting articles into smaller parts . . . 14
  - 3.4.5 Combining methods . . . 16

4 Experiments **17**
- 4.1 Implementation . . . 17
  - 4.1.1 Pre-processing . . . 17
  - 4.1.2 Enumeration . . . 18
- 4.2 Experiments . . . 18
  - 4.2.1 Data . . . 19
  - 4.2.2 Evaluation . . . 20

5 Results **21**
- 5.1 Benchmarks . . . 24

6 Discussion **28**
6.1 Comparisons with other approaches 29
6.1.1 Future Work 30

7 Conclusion **32**

A Source Code **35**
A.1 main.py 35
A.2 bag.py 39
A.3 filters.py 43
A.4 preprocessing.py 47
A.5 helpers.py 48

Chapter 1

Introduction

In today's society the amount of information available is vast. The internet, one of the largest sources of information available today, makes finding information easy. Although large parts of the internet consist of images and videos of cats doing silly things there are still numerous sources with structured text information streams, such as news. However, the number of available news streams is so vast that no man is capable of digesting them all. Moreover, the various news sources tend to re-iterate upon events previously reported by themselves or other sources. It can therefore become tedious to try and stay on top of the news. Within several domains, such as stock trading, it can be of vital importance to be able to identify important events as soon as they occur and thus the availability of fast and reliable computer systems that aid in the detection of such events can be of great benefit.

On-line New Event Detection (ONED) is the process of monitoring text information streams and detecting stories that report about new events. For instance, a story reporting a large oil leakage in the Pacific Ocean should be identified as a new story the first time that the event is reported, while consecutive stories further iterating upon the event or stories discussing the environmental effects of the spill should be identified as not containing new events.

In this thesis we present a new approach to ONED, built upon the concept of identifying minimal new sets of words from new articles and using those as basis for determining whether or not the text refers to a new event. This concept has previously been proposed by Damaschke [4] and Wurzer et al. [11]. The intuition is that when a new event occurs, like the death of a celebrity, we are likely to find words together within the same article that

have never occurred before. Moreover we are likely to find small sets of words that have never previously occurred together. For instance, the first report of the recent death of the artist Prince would likely have been the first time the words “Prince” and “dies” appeared together. By finding and filtering these minimal new sets to only include informative word combinations based on quantitative criteria we are able to present likely events succinctly which allows a user to browse through them.

Unfortunately, finding cases where the aforementioned example fails is trivial. Consider the sentence: “On Mondays I listen to Prince. On Tuesdays I play Candy Crush until the battery in my phone dies”. Here the words “dies” and “Prince” occur together without any relation to each other and the pair of them does in this case not indicate any new event. Moreover, if this co-occurrence of these two words would appear before any reports of Prince’s death then this would prevent them from being recognized as a new pair in those later reports. Irrelevant co-occurrences could thus yield both false positives and false negatives. In this thesis we explore a few methods which aim to prevent this scenario, such as considering the distance between the words and filtering words by importance.

We will start this thesis off by discussing the background of ONED and related works on the topic in chapter 2. We then go through the goals and details of our method in chapter 3. In chapter 4 we explain the details of our implementation used for the testing and the methods we employ for testing. Chapter 5 presents the results of the tests. Finally, in chapter 6 we discuss our findings and present suggestions for future work.

Chapter 2

Background

The task of detecting new events within a stream of documents can be separated into two camps, *Retrospective Event Detection* and *On-line New Event Detection*. Retrospective Event Detection focuses on discovering previously unidentified events from a finite collection of articles [12]. On-line New Event Detection (ONED), which is sometimes known as either simply *New Event Detection* (NED) or *First Story Detection* (FSD), instead tries to identify new events in real-time as soon as they arrive from live news feeds.

On-line New Event Detection has seen much development in the past 15 years as it has been one of the topics covered by the *Topic Detection and Tracking* (TDT) research program [10]. In this chapter we will present a brief overview of the techniques used for ONED by other researchers within the field.

2.1 What are events?

The concept of events in daily speech can be somewhat ambiguous [6] and in order for us to reason about events it is important that we specify what we mean. "The United States invade Vietnam" could be considered an event but at the same time the whole Vietnam War could be considered an event. We use TDT's definition of events: An event is "a particular thing that happens at a specific time and place, along with all necessary preconditions and unavoidable consequences" [10]. A new news-story does therefore not necessarily report a new event. For instance a news story reporting about a natural disaster would contain a new event, whereas consecutive articles de-

tailing the extent of the damage caused and the particulars of the rebuilding efforts that follow, would not. In ONED we want to be able to detect only those stories which contain events, and in particular, only new events that have not been reported before.

2.2 Related works

A common approach to ONED is to represent documents as term vectors where each term within a document is weighted using some metric, typically TF-IDF (term frequency-inverse document frequency), which are then compared in some way to the vector representations of previously seen documents. Since comparing incoming stories to all previous stories is slow, organizing stories into clusters and comparing incoming stories to the clusters is a popular step employed by many. Often, the stories within the best fitting clusters would then be compared to the incoming story.

Papka, Allan and Lavrenko [7] use a single-pass clustering technique where feature extraction and selection techniques are used to build query representations of all stories. They then compare any incoming document to the previously seen queries and flag the document as new if its comparison score exceeds a threshold which is calculated for each document.

Yang, Pierce and Carbonell [12] also use a single-pass clustering algorithm and incremental IDF for TF-IDF weighting, IDF is first trained on a dataset an then updated for each document. In order to increase efficiency and due to the temporal nature of events they use a sliding time-window to only compare incoming documents to stories within the time window. Additionally they also examine decay weighting where documents further away in time are marked as less important.

Brants, Chen and Farahat [3] use a source-specific TF-IDF model where the news stream is comprised of multiple sources and certain words are more common depending on the source. Each incoming document is weighted and compared to previously seen documents using either Hellinger- or cosine-distance but normalized by subtracting the average distance of the current document to all other documents.

Another popular approach is to represent documents using named entities, the idea being that events can be summarized by “what”, “where”, “when” and “how” as well as other similar properties.

Yang et al. [13] use a supervised learning algorithm to classify documents

in an on-line document stream into pre-defined topic categories. Weighting is done using named entities. They then perform topic specific stopword removal and topic sensitive feature weighting.

Kumaran and Allan [5] use named entities in addition to term vectors with incremental TF-IDF weighting as well as topic terms and compute the cosine similarities of each of those features to previously seen documents. A Support Vector Machine (SVM) classifier is then trained on each of those three features.

Zhang, Li and We [14] create a tree of clustered stories from the stories they process which allows them to quickly find the stories which most closely resemble the incoming story by comparison. In addition they present two new approaches to term weighting. The first approach takes into account the frequency of words in the previously identified clusters, a word that is very common in one cluster but not common in others is deemed important for that cluster while a word that is common in all clusters is determined to be of less importance. The second weighing approach classifies various named entities into different classes and tries to determine what type of words are important for different story topics. For instance, in their testing they found that locations were of utmost importance for stories relating to natural disasters.

Petrovic, Osborne and Lavrenko [8] paraphrase incoming articles and use locality-sensitive hashing for comparisons. By paraphrasing they can for instance determine that "die" and "kick the bucket" hold the same meaning and achieve good results.

Wurzer, Lavrenko and Osborne [11] present an algorithm for ONED which operates in constant time/space with regards to the number of processed articles. Their approach is based on the same intuition as ours and revolves around identifying new small subsets of words from the words that make up an article. While our method focuses on the subsets themselves, their approach considers the amount of such new subsets identified. Particularly, they perform ONED by generating all subsets of up to k terms for incoming documents and estimating novelty as the fraction of such sets that have not appeared before. For this purpose they store all these subsets using a Bloom filter with a 32-bit hashing function. For the Bloom filter they use a fixed-length bit array. This allows their system to perform lookups and updates in constant time as well as constant space. To limit the amount of false positives reported by Bloom-filter lookups they zero out a random bit each time the proportion of non-zero bits exceeds a given threshold. This of course means

that the system forgets some of the sets of words that it has seen.

Chapter 3

Method

On-line new event detection is hard [1]. In this chapter we present a new method which aims to perform ONED, based around the concept of *minimal new pairs*. We explain the rationale behind the idea, give a detailed explanation of the base algorithm and discuss some of the approaches we have examined for expanding upon the base idea.

3.1 The base idea

Damaschke [4] observed that for new articles, the existence of previously unseen small sets of words could be good indicators for new events and presented an efficient algorithm for finding such sets. This is similar to the approach used by Wurzer et al. [11] except they use the proportion of new such sets to quantify the novelty of incoming articles while our approach focuses on the new sets themselves and the terms within them. However, the underlying principle is the same: the first report of Prince's death is likely to be the first document within the stream containing the words "Prince" and "dead" or similar.

In more formal terms, Damaschke [4] presents the definition of *minimal new sets*.

Definition 1. *Let $B_0, B_1, B_2 ... B_{m-1}$ be a sequence of sets that we call bags. For another bag $B := B_m$ we call a subset $X \subseteq B$ new at m, if X was not already a subset of an earlier bag : $\forall i < m : X \setminus B_i \neq \emptyset$. Otherwise X is said to be old at m. We call $X \subseteq B$ minimal new at m if X is new and also minimal (with respect to inclusion) with this property.*

Thus, to continue with our previous example, if we tokenize incoming news articles into bags of words, then $\{prince, dead\}$ would likely be a minimal new set for the first article reporting the tragic event of Prince's death.

A first, naive approach could be to simply base the detection of new events on the existence of minimal sets below a given set size k. For example, if we choose $k = 3$ we would say that if an article contains new words or new pairs of old words it contains a new event, otherwise it does not. This naive approach does, however, not work well in practice since articles covering the same event are quite likely to contain new words or new pairs and thus this method is likely to falsely flag articles as containing new events.

Instead of blindly flagging articles based on the existence of minimal sets we propose a method which automatically highlights and presents important informative word combinations by certain quantitative criteria that allow the user to quickly browse through them and see which of them raise interest. This introduces the problem of having to prioritize the minimal sets identified for any incoming article.

3.2 Enumeration

Central to our approach is the process of enumerating minimal new sets from incoming articles. Luckily, Damaschke [4] presents an effective enumeration algorithm:

Let's say that at time m we have stored the bags of previous articles sequentially: $B_0, B_1, B_2...B_{m-1}$. In order to find all minimal new sets of bag B_m we generate candidate sets $X \subset B_m$ of increasing sizes until all candidates are supersets of already discovered minimal new sets at m. First we find all, if any, words that are new at m, then from any words that are not new at m we find all possible sets of size 2 (pairs) and check for each of them if they are new at m. If there are any pairs that were not new at m we find sets of size 3 from those, etc.

The key to making the algorithm efficient lies in how we determine whether a set X is new at m. We create a function f, so that $f(X) := min\{i \mid X \subseteq B_i\}$ and let $f(X)$ be undefined if X is not a subset of any bag B_i. Now X is new at $f(X) = i$ if such a value exists, and old at any subsequent index $j > i$.

When enumerating minimal new sets of bag B_i we store X along with $f(X) = i$ for each of the candidate sets that we determine to be minimal

new at i. For any set of size 1 we can easily determine if it is a minimal new set simply by checking whether a value for that set has been stored in the table. Any time that a word makes its first appearance any larger subset of words from the same bag containing that word will also be new at that time, but not minimal. For any larger candidate set $X \subseteq B_m$, $|X| > 1$ we therefore might have to take further steps to determine whether X is new at m. Again, if $f(X)$ is stored we know that X is old. If $f(X)$ is not stored then for each $Y \subset X$ we check if $f(Y)$ is stored. If $f(Y) = j$ is stored then we check if $X \subset B_j$. If such a j is found then X was new at j but is old at m. If no such m is found we conclude that X is new at m. The minimal new sets of bag B_j can be enumerated in $\mathrm{O}\big(|B_n|2^k/k!\big)$ time.

3.3 Pre-processing

In order to enumerate minimal sets from an article, some preprocessing is required. Incoming articles are tokenized, i.e. split into lists of words, stopwords, the most common words of the input language, are removed along with any punctuation and the remaining words are stemmed using a Porter2 [9] stemmer and converted to lowercase.

3.4 Extensions

As explained in section 3.1 we might want to improve the performance of our results by limiting the outputs of the enumerations. Two simple ways of doing this are to simply limit the input the enumeration algorithm receives or filtering its output. In this section we will discuss the approaches we examined in order to try to accomplish this which mostly are based on filtering common or unimportant words, prioritizing words that occur in many minimal new pairs and prioritizing pairs of words that appear with close proximity.

3.4.1 Filtering common or unimportant words

Words that describe a new event are likely to be relatively unique to that event. Reversely, words which are common over multiple different articles are unlikely to be descriptive for new specific events. Similarly, words which are common within an article are likely to be important for that article. We

could filter out words that are common within the previously seen articles, or words that can be determined not to be important within the incoming article. We will examine filters based on three different metrics

- *collection frequency*: how often a given word has appeared within the accumulated corpus.
- *document frequency*: the number of documents in which a given word has previously appeared.
- *TF-IDF* score: a score indicating the importance of a word within the given article with relation to how common it is within the corpus. TF-IDF is calculated as $tfidf = tf * idf$ where tf is the term frequency of the given word within the given article and $idf = \log \frac{N}{df}$ for N as the number of processed articles and df the document frequency of the given word.

This type of filtering can be done in several different ways. We could define thresholds for our filters and focus on words which score above or below that threshold, this is similar to the approach used by Brants et al. [3] where any word with a document frequency lower than 2 was discarded. However, determining the thresholds might prove difficult and we most certainly do not want to discard all words which have not appeared in previous bags. Another approach would be to select the n words with the highest or lowest score and focus on them. In either case we also have the option to either completely ignore all words but the ones we have selected or to choose only sets containing our chosen words. The distinction between these two approaches is subtle but important. Given that the two words "Prince" and "purple" are part of the same bag but only "Prince" satisfies the condition of our filter. If the pair ("Prince", "purple") is minimal new in the unfiltered bag the first approach would not output it since it contains "purple" which is to be removed, while the second approach would output it since the pair contains "Prince" which is to be included.

Another important aspect to consider is when filtering is applied. If filtering is done as part of the preprocessing step, then the pair in the above example will not be found. Consequently, any subsets containing the filtered words will not be stored in the table of $f(X)$ values. In that case only the filtered bags should be stored and filtered words should be free to be identified as new in later bags.

3.4.2 Prioritizing words that occur in many minimal new pairs

It is not hard to imagine that an article reporting a new event is likely to contain many new pairs of words that previously have not appeared together. Commonly used words are however likely to have already appeared together. Thus, minimal new sets of size > 1 are likely to contain at least one word that is more indicative of the events described within the given article. If a particular word can be found within a majority of the minimal new sets, then it is likely to be an old word within a new context and likely to be a key property of the article. For instance, the words "purple", "rain" and "Prince" are likely to have appeared together before in Prince related articles. An article that then yields the minimal new sets $\{Prince, dead\}, \{purple, dead\}, \{rain, dead\}$ gives us a common denominator in "dead" within the minimal set, indicating that "dead" is likely to be an important indicator of a possible new event. Based on this idea we can identify important words by counting the number of minimum new sets they appear in. In particular we focus on prioritizing words that occur in many minimal new pairs.

3.4.3 Prioritizing words that appear in close proximity

Words that are indicative of new events are likely to be found in close proximity. For instance an article reporting about a volcanic eruption at Yellowstone is much more likely to contain something along the lines of "Volcanic eruption at Yellowstone" than mentioning "Yellowstone" in one paragraph and "eruption" several paragraphs later.

For any minimal new sets of words from an article we can calculate the minimum amount of words between the elements within the sets and find which sets contain words which occur close to each other within the article. Using this we can prioritize sets of words which appear more closely together within the article in the hope that they are good indicators of potential events.

3.4.4 Splitting articles into smaller parts

Another approach for increasing the importance of words that appear together is to split an article into several smaller, sub-articles. Some natural

examples of such sub-articles would be paragraphs or sentences. One can easily imagine that important event-related words appear within the same paragraph or even the same sentence. This requires an extra pre-processing step: splitting up incoming articles into the wanted bits. In addition this requires some trivial changes to the base algorithm. For each article we have to feed each sub-bag to our algorithm but take care not to add any words to the list of previously seen bags until after we have processed all the sub-bags of the incoming article so as to avoid identifying words that first appear within the given article as old in sub-bags which are subsequent to the sub-bag in which they first appear.

Let the sub-bags $P_k := [S_0, S_1, ..., S_q]$ be a list of the sub-bags generated from article k so that for the bag of all the words in the article B_k it holds that $\bigcup_{i=0}^{q} S_i = B_k$. We call P_k the parent bag for article k. We assume that we have stored all the previous parent bags $S_0, ..., S_{k-1}$ as well as mappings from all previously minimal bags to sub-bags within parent-bags. The enumeration of B_k is now the union of all the enumerations from the sub-bags $S_i \mid i \in [0, q]$ where the minimal news sets of each S_i are enumerated using the approach described in section 3.2. If a minimal new set X is found while processing sub-bag $S_i \in P_k$ we store the minimal new set along with both the numbers k and i. If the same set X is found in another sub-bag $S_j \in P_k$ where $j > i$ we need to make sure to store j as well. We let $f'(X)$ be a function which returns the number of the parent-bag in which X was minimal new as the sub-bags of that bag which contained X. In other words we store $f'(X) = (k, [i, j])$, to indicate that X was minimal new at k and found in sub-bags i and j.

Just like in the original enumeration algorithm, we don't have to rely on naive exhaustive search to calculate $f'(X)$ for any $|X| > 1$. For each $Y \subset X$ we can lookup $f'(Y)$ and if a pair $(k, [l_0...l_q])$ is found we check $X \subseteq S_{k,l_i} \mid S_{k,l_i} \in P_k, i \in [0, q]$. If such a S_{k,l_i} is found we know that X is old. Furthermore we can conclude that X was new at the smallest $k \mid f'(Y) = (k, [l_0, ..., l_q])$.

Theorem 1. *Given that we have previously processed bags $B_1...B_{n-1}$ the minimal-new sets of maximum size h can be enumerated from sub-bag $S_{n,j} \in B_n$ in $\mathrm{O}\big(p|S_j|2^h/h!\big)$ time, where p is the amount of sub-bags in any bag $B_i \mid i \in \mathbb{N}, i \in [1, n-1]$.*

Proof. For any given candidate set $X \subseteq S_{n,j}$ of size r containing only words which we have previously seen we know that if $f'(X)$ is undefined we have to look up $f'(Y)$ for each $Y \subset X$. For $f'(Y) = (k, l_0, .., l_q)$ we then check

if $Y \subseteq S_{k,t} \mid t \in f'(Y)$. We will have to lookup at most $2^r - 2$ candidate sets and check at most p possible subsets $S_{k,t}$. So we can check if a set X of size r is new in $\mathrm{O}\big(p2^r\big)$ time. The number of candidate sets we have to consider is $\binom{|S_{n,j}|}{r} < \frac{|B_n|}{r!}$. It follows that finding all minimal new sets of size h is time-bounded by $\mathrm{O}\big(p|S_j|2^h/h!\big)$. □

Joining sub-bags

Using the same idea we can create a "sliding window" of k sub-bags to increase the sizes of the sub-bags and thus increasing the odds of finding new combinations of words. A simple example would be to always use three consecutive sentences as sub-bags when available. Thus the first three sentences of an article would make up a sub-bag, then the second to the fourth sentence would make up the next sub-bag etc. For each consecutive sentence the first sentence of the previous bag would be removed and the new sentence added, like sliding a three-sentence wide window over the article. This approach allows us to explore larger sub-bags while preventing us from missing important word combinations which we could miss if we were to simply choose the first three sentences followed by the next three etc. Using a similar approach as previously explained for sub-bags we store the number of the parent bag along with the numbers of the sub-bags. The only difference now is that the sub-bags are slightly larger and fewer.

3.4.5 Combining methods

We can of course also combine several of the methods described above. For instance we could generate sub-bags from incoming articles but filter the output of the enumeration to only include subsets with words that scored above a given TF-IDF threshold.

Chapter 4

Experiments

We implemented the concepts discussed in chapter 3 using the Python programming language. We then used these implementations to test the effectiveness of the various methods on several different datasets. In this chapter we discuss our implementation in some detail as well as how the tests were carried out and what data was used for testing.

4.1 Implementation

For our implementation we chose Python because it supports many high-level concepts such as lambda functions, list comprehensions etc. as well as for its extensive standard library and excellent availability of good third-party libraries such as the Natural Language Tool Kit (NLTK) [2]. The most important files are included in appendix A but the full source used in this project can be found on GitHub[1].

4.1.1 Pre-processing

The NLTK provided us with most of the parts needed for preprocessing: we used the built-in word-tokenizer for splitting whole-string articles into lists of words, stopwords from the English corpus for stopword-removal and the porter2 stemmer for fast and simple stemming of words. Python's string module provided us with a list of punctuation characters which we used to remove punctuation. Our pre-processing step first removed any punctuation

[1] `https://github.com/orng/ONED-Thesis`

character, then tokenized the input-string, removed all the stopwords and then stemmed each of the tokens.

4.1.2 Enumeration

We implemented the algorithm described in section 3.2. Our implementation used sets of strings (or Python's frozensets in cases where the sets had to be hashed) to represent the bags of words to enumerate, a list of bags (list of sets of strings) to store previously seen bags and a dictionary (hashmap) with frozensets as keys and integers as values for mapping previously seen minimal new sets to the number of the bag in which they first appeared.

As discussed by Damaschke [4] the existence of small minimal-new sets is likely to prove more informative than large sets. In order to simplify results we therefore chose to limit the enumerations to only consider subsets of size two or smaller.

4.2 Experiments

We conducted a series of tests on various data sources using the base-approach to enumeration as well as the different extensions described in chapter 3. We created several configurations, each using one or more of the aforementioned approaches. Each configuration was tested on four different sets of articles and the result of each processed article was stored into a text file. Additionally, for any word found in a minimal new set of size two we printed the amount of such minimal new sets that the word occurred in. The configurations we used in our experiments were as follows:

Configuration-1: The base algorithm without any further analysis or filtering of the input and output sets beyond the basic pre-processing described in section 3.3. Additionally, we conducted experiments with sorting new pairs in descending order based on the minimum distance between the words in the pairs or the combined number of minimal-new sets that the words appeared in. In the latter case we also outputted the number of minimal-new sets the words appeared in.

Configuration-2: Output filtering where any minimal-set containing at least one of the k highest scoring words was included in the output, for positive integer numbers of k.

Configuration-3: Filtering using the same technique as configuration-2, except filtering was applied prior to enumeration.

Configuration-4: Processing articles by splitting them into sub-bags where each sub-bag was a sentence.

4.2.1 Data

All of the data that we used for testing was real data obtained by crawling various sources from the internet. They contain articles of various types and lengths since one of our goals was to identify to some extent for what type of input the approaches were successful. The crawled articles were stored along with their dates of publication so that they could later be consumed in the same order that they would have arrived had they been processed by an on-line monitoring system. We used the following four different datasets for testing:

D1: A collection of very short summary articles and headlines collected from articles regarding the 2016 presidential election in the USA, gathered from PBS Newshour[2], Reuters[3] and CBS News[4]. The data was gathered on 04.04.2016 containing 300 stories published on dates ranging from 03.03.2016 to 04.04.2016. This dataset served to test our approach on short stories within a narrow topic. Our hope was that short articles within a shared topic would provide good results, since high word density within a shared topic combined with the low overall word count of each article would decrease the amount of new words found and instead yield interesting new pairs.

D2: Timelines of president Barack Obama's years of presidency, '09, '10 and '11, taken from Wikipedia[5]. Each point of the timeline can be regarded as a very short article and in this case is very likely to refer to an event. The data was gathered on 26.04.2016 and contained 419 stories in total.

D3: A chronological account of the events of the Second World War taken from world-war-2.info[6] where we treated each paragraph as a separate article. The point of this dataset was to test our approaches on slightly longer and less densely focused articles within a single topic. The data was gathered on

[2] `http://www.pbs.org/newshour/tag/vote-2016`

[3] `http://www.reuters.com/politics/election2016`

[4] `http://www.cbsnews.com/election-2016/`

[5] `https://en.wikipedia.org/wiki/Timeline_of_the_presidency_of_Barack_Obama`

[6] `http://world-war-2.info/history/index.php`

21.03.2016 and contained 495 stories in total.

D4: Real life full-length news articles taken from Reuters[7]. The data was gathered on 03.08.2016 and includes 5227 articles dating from 04.11.2016 to 03.08.2016. The purpose of this dataset was to simulate real-life usage of an ONED system on a large set of data.

4.2.2 Evaluation

Evaluation of the results was done through manual evaluation and comparison of the results outputted by the experiments. In particular, we were interested in finding methods that prioritized minimal new sets that captured the main points of the input articles.

In chapter 5 we present the results of our tests and in chapter 6 we discuss and analyse those results.

[7] `http://www.reuters.com/news`

Chapter 5

Results

In this chapter we will look at the results of the experiments discussed in chapter 4.

Tables 5.1 to 5.4 show the ratio of articles which resulted in no new words, no new pairs or both. In the cases where filtering was applied, a shorthand for the filter type (cf=collection frequency, tf=term frequency) is included as well as the threshold. We noted that filtering did in fact increase the number of empty output-sets of either size and in the cases of the collection frequency and term frequency filters along with any pre-enumeration filtering (Configuration-3) the change was quite drastic.

	No new words	No new Pairs	Both
Configuration-1	0.04	0.03	0.01
Configuration-2 (cf: 5)	0.00	1.00	0.93
Configuration-2 (tf: 5)	0.58	0.42	0.41
Configuration-2 (tfidf: 5)	0.05	0.35	0.01
Configuration-3 (cf: 5)	0.19	0.92	0.85
Configuration-3 (tf: 5)	0.01	0.99	0.96
Configuration-3 (tfidf: 5)	0.00	0.74	0.01
Configuration-4	0.04	0.03	0.01
Configuration-4 (tfidf: 5)	0.04	0.52	0.02

Table 5.1: The outcomes from running different configurations on dataset D1

	No new words	No new Pairs	Both
Configuration-1	0.10	0.01	0.00
Configuration-2 (tfidf: 5)	0.12	0.25	0.00
Configuration-3 (tfidf: 5)	0.01	0.69	0.00
Configuration-4	0.10	0.01	0.00
Configuration-4 (tfidf: 5)	0.11	0.36	0.01

Table 5.2: The outcomes from running different configurations on dataset D2

	No new words	No new Pairs	Both
Configuration-1	0.16	0.10	0.01
Configuration-2 (tfidf: 5)	0.16	0.30	0.01
Configuration-3 (tfidf: 5)	0.11	0.55	0.01
Configuration-4	0.16	0.10	0.01
Configuration-4 (tfidf: 5)	0.16	0.32	0.01

Table 5.3: The outcomes from running different configurations on dataset D3

	No new words	No new Pairs	Both
Configuration-1	0.13	0.74	0.13
Configuration-2 (tfidf: 5)	0.23	0.74	0.65
Configuration-3 (tfidf: 5)	0.18	0.65	0.33
Configuration-4	0.18	0.64	0.08
Configuration-4 (tfidf: 5)	0.32	0.64	0.55

Table 5.4: The outcomes from running different configurations on dataset D4

One of the events touched upon in the D2 dataset is the BP-oil spill in the Gulf of Mexico[1] and president Obama's actions revolving that incident.

[1] `https://en.wikipedia.org/wiki/Deepwater_Horizon_oil_spill`

Listings 5.1 to 5.5 show the enumerations of an article in which the event was mentioned for the second time, enumerated using different configurations. In this case the event would be that president Obama answered questions regarding the oil spill.

As is to be expected, the enumerations using Configuration-2 and Configuration-4 are very similar since most of the articles in the dataset only contain one sentence and therefore only one sub-bag. Apart from the enumeration generated by Configuration-3 which contains no useful insights about the contents of the article, the generated enumerations appear to capture the main gist of the article. For configurations 1,3 and 4 the minimal new sets have been sorted based on TF-IDF score, in descending order.

Listing 5.1: Original text

```
May 27 - President Obama holds a news conference in the East Room to
    answer questions about the BP Deepwater Horizon Gulf of Mexico oil
    spill .
```

Listing 5.2: Configuration-1

```
New Words: {deepwat, horizon}
New Pairs: {(question, bp), (spill, question), (gulf, bp), (spill, answer)
    , (bp, answer), (gulf, question), (gulf, spill), (question, news), (bp
    , news), (spill, news), (gulf, answer), (question, oil), (mexico,
    question), (mexico, bp), (mexico, spill), (spill, 27), (question, 27),
     (bp, 27), (answer, news), (gulf, news), (question, east), (bp, east),
     (spill, east), (bp, confer), (spill, confer), (question, confer), (
    question, room), (bp, room), (spill, room), (question, may), (spill,
    hold), (bp, hold), (question, hold), (oil, answer), (mexico, answer),
    (gulf, 27), (27, answer), (oil, news), (east, answer), (confer, answer
    ), (gulf, confer), (room, answer), (gulf, room), (mexico, news), (gulf
    , may), (may, answer), (gulf, hold), (27, news), (east, news), (room,
    news), (may, news), (oil, 27), (confer, oil), (mexico, 27), (room, oil
    ), (oil, hold), (mexico, confer), (east, 27), (mexico, room), (confer,
     27), (mexico, hold), (room, 27), (27, may), (east, may)}
```

Listing 5.3: Configuration-2

```
New Words: {deepwat, horizon}
New Pairs: {(question, bp), (spill, question), (gulf, bp), (spill, answer)
    , (bp, answer), (gulf, question), (gulf, spill), (question, news), (bp
    , news), (spill, news), (question, oil), (mexico, question), (mexico,
    bp), (mexico, spill), (spill, 27), (question, 27), (bp, 27), (question
    , east), (bp, east), (spill, east), (bp, confer), (spill, confer), (
```

```
    question, confer), (question, room), (bp, room), (spill, room), (
    question, may), (spill, hold), (bp, hold), (question, hold)}
```

Listing 5.4: Configuration-3

```
New Words: {horizon, deepwat, question}
  New Pairs: {}
```

Listing 5.5: Configuration-4

```
New Words: {deepwat, horizon}
New Pairs: {(question, bp), (spill, question), (gulf, bp), (spill, answer)
    , (bp, answer), (gulf, question), (gulf, spill), (question, news), (bp
    , news), (spill, news), (question, oil), (mexico, question), (mexico,
    bp), (mexico, spill), (spill, 27), (question, 27), (bp, 27), (question
    , east), (bp, east), (spill, east), (bp, confer), (spill, confer), (
    question, confer), (question, room), (bp, room), (spill, room), (
    question, may), (spill, hold), (bp, hold), (question, hold), (question
    , -)}
```

Figures 5.1 to 5.4 show how the amount of minimal new sets of size less than 3 found in enumeration change over time using different configurations. In all cases the amount of new words found decreases over time while the amount of new pairs found tends to increase over time.

5.1 Benchmarks

For each configuration we measured the time taken to process each of the datasets. Although each run produced the same output we performed each test 20 times to achieve more accurate benchmarks. Table 5.5 shows the average time per article for different configurations. Table 5.6 shows the combined average time per article for the same configurations for configurations using sub-bags and vice versa.

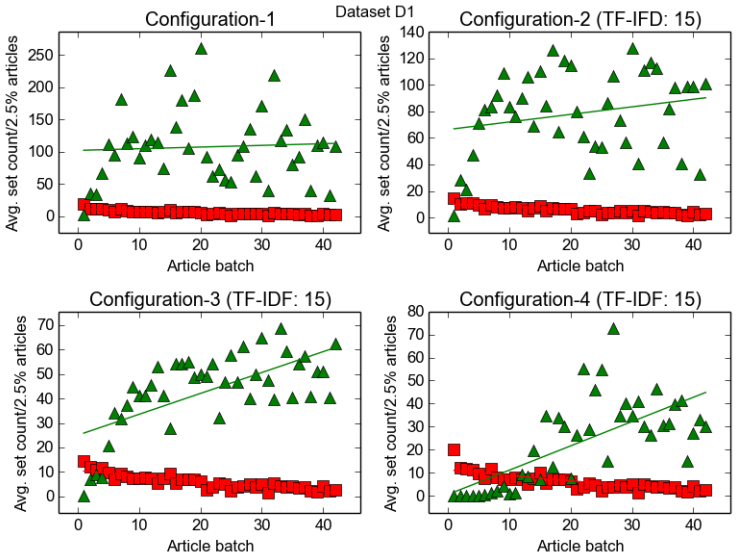

Figure 5.1: Average number of new words and new pairs per batch of 2.5% of all articles in dataset D1. Green triangles are new pairs and red boxes are new words.

Configuration	time/article (s)
Configuration-1	0.3909
Configuration-2 (tfidf: 15)	0.2939
Configuration-4	0.0533
Configuration-4 (tfidf: 15)	0.0488

Table 5.5: Average runtimes per article using different configurations

Configuration	time/article (s)
Configurations without sub-bags	0.3424
Configurations with sub-bags	0.0510

Table 5.6: Average runtimes per article using sub-bags or not

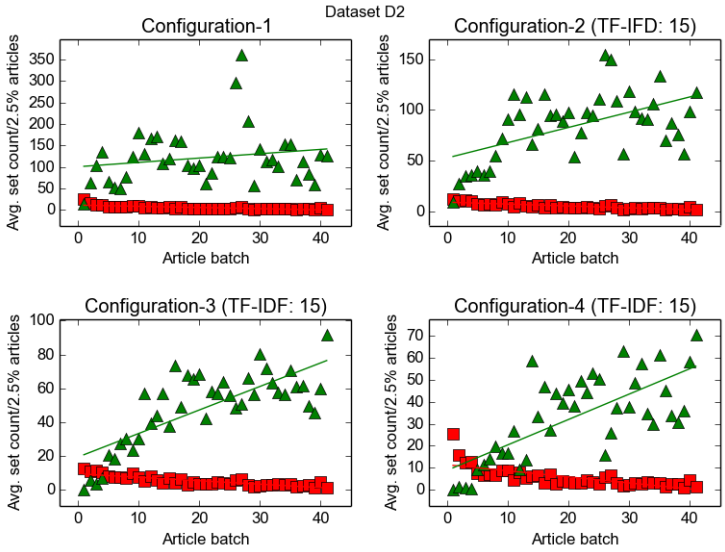

Figure 5.2: Average number of new words and new pairs per batch of 2.5% of all articles in dataset D2. Green triangles are new pairs and red boxes are new words.

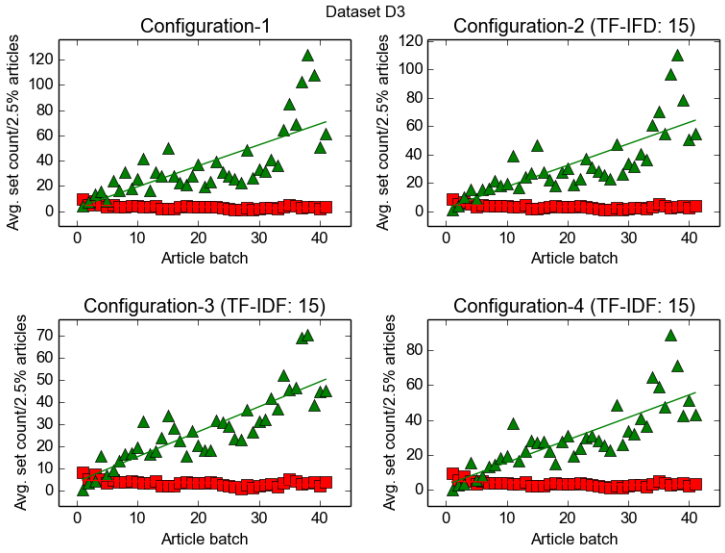

Figure 5.3: Average number of new words and new pairs per batch of 2.5% of all articles in dataset D3. Green triangles are new pairs and red boxes are new words.

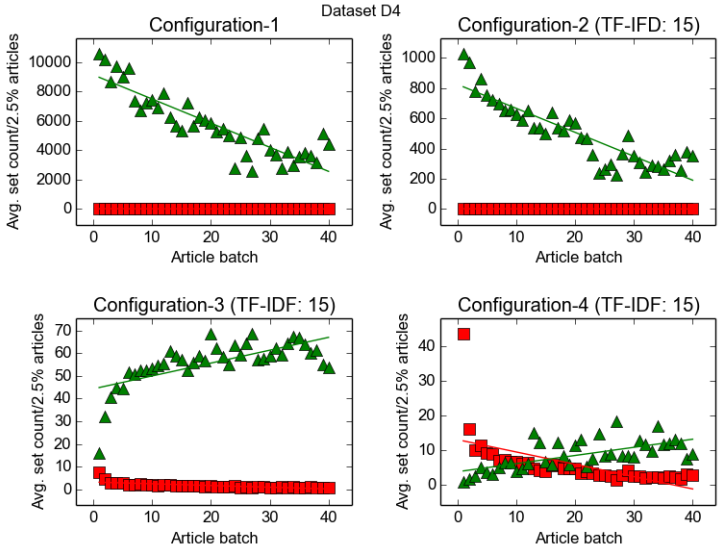

Figure 5.4: Average number of new words and new pairs per batch of 2.5% of all articles in dataset D4. Green triangles are new pairs and red boxes are new words.

Chapter 6

Discussion

Figures 5.1 to 5.4 show that over time, at least to begin with, the amount of new words found decreases while at the same time the amount of new pairs appears to increase. If we are more interested in new pairs than new words then it might therefore be a good idea to pre-train the system on some set of articles so as to skip past the period in which we mostly find new words.

The results of our experiments indicate that TF-IDF is a good filter to use if we want to decrease the output size, but that filtering should be done on the output rather than the input.

Listings 5.2, 5.4 and 5.5 all have `(question, bp)`, `(spill, question)`, `(spill , answer)` etc. among the first pairs of output, which capture the essence of the original article. Overall, the results presented in the previous chapter indicate that minimal-new sets can indeed be good indicators of new events or at least novelty within articles. However, simply checking for the existence of minimal new sets containing only one or two elements is not sufficient for determining if an article contains a new event or not and further work has to be done to develop a method to quantify the enumerations to calculate the probability of them relating to a new event.

One thing that is worth highlighting is the good performance of the sub-bag approach which, as indicated by tables 5.5 and 5.6, achieves good results with a significant increase in performance. It should be noted for these benchmarks that each result was written to disk after being calculated. This might be partially responsible for the unfiltered configurations performing worse than the filtered ones. In any case, combining filtered and unfiltered configurations and grouping only on whether or not configurations use sub-bags as in table 5.6 still points to sub-bags being roughly 6.7 times faster

than using entire articles as bags. This is perhaps unsurprising since for articles with very different sub-bags the number of candidate sets to examine is smaller and checking if candidate sets are subsets of sub-bags is also slightly faster due to sub-bags being smaller. An additional potential benefit of the sub-bags approach is that each sub-bag could be processed in parallel and therefore the runtime could be improved even further, especially in cases when long articles are processed.

Although it could be argued that our system could be useful for humans to perform tool-assisted New Event Detection where users scan through a small number of minimal new sets for incoming articles and are as such able to process the incoming article faster than they would by reading through them, the system still requires manual labour. In, fact since it is not yet able to perform automatic Event Detection it is currently not a full-blown ONED-system.

6.1 Comparisons with other approaches

Since we have not yet developed a true ONED system, comparisons with other methods becomes troublesome. Had we developed such a system we could have tested our approach on any of the TDT datasets, such as TDT5 and compared the results with the results of other approaches. If we assume that the outputs of our approach could be quantified effectively we can compare the time complexity of our approach to other existing approaches.

As discussed in chapter 2 Wurzer, Lavrenko and Osborne [11] present an ONED system based on the proportion of subsets up to size k, for some small integer k, which have not been previously stored. Through the use of a Bloom-filter they are able to operate in constant time per input set in relation to the number of seen articles as well as in constant memory. In their tests they used $k = 3$ and achieved impressive results. However, since they do not only consider minimal new sets they always have to find all possible subsets of size k of the incoming bag B_n whereas we might achieve faster times, especially for larger values of k. The base approach is time-bounded by $\mathrm{O}\big(|B_n|2^k/k!\big)$ while their approach is time-bounded by $\mathrm{O}\big(|B_n|\big)$ and since $\lim_{x\to\infty} 2^k/k! = 0$ our approach is better suited for larger values of k.

Like Wurzer, Lavrenko and Osborne [11] and unlike all the other approaches discussed in chapter 2 ours is not time-bound by the number of processed articles and therefore does not slow down over time. However, un-

like the method of Wurzer et al. the memory consumption of our approach grows with the number of processed articles as our approach requires storing all processed bags.

One weakness of our current enumeration algorithm is events which are repeated or events which occur as history repeats itself, e.g. someone is shot in Times Square today and then someone else is shot during similar conditions some time later. This later event is a separate and new event, distinct from the first, but although it would likely include new information such as other names etc., it is also possible that it would not result in any minimal new sets since they would have been introduced by the first shooting.

Since our enumeration algorithm builds up a collection of minimal new sets, one can not easily remove old enumerations without sacrificing accuracy, since the elements which were minimal-new in the set that is to be removed might have reappeared in a later set and the removal of the first set would therefore require the later set to be updated. Because we do not know what sets require updates, we would have to traverse all the stored bags which arrived after the set we are removing until we either run out of bags to check or minimal new sets found in the removed enumeration.

Others have developed such systems which forget seen articles over time. Yang, Pierce and Carbonell [12], for instance, use a sliding window which specifies which previously seen stories they compare incoming stories to. Furthermore they explore weight decay where older stories within the window are given less weight.

6.1.1 Future Work

We believe that the results indicate that an effective fully automatic ONED system could be built using our approach of enumerating minimal new sets but different ways for quantifying the enumeration have to be studied. Possible quantification methods might include combining our approach with named entity extraction and using the number of minimal new sets containing named entities as indicators of new events. Another possible approach would be to incorporate the previously discussed technique used by Wurzer, Lavrenko and Osborne [11] and use the ratio of the amount of minimal new sets of size less than some integer k over the amount of possible subsets of size less than k. We might be able to achieve better accuracy since we never discard any found minimal-new sets while they flip random bits in the Bloom-filter in order to decrease the probability of false positives.

It might also be possible to improve upon the filtering methods presented in this article through smarter filtering. Instead of generating all candidate sets it might be worthwhile to examine only certain words, for instance named entities or words with high TF-IDF scores and only include those as candidate sets when enumerating.

As discussed earlier, not being able to forget stories over time might be a drawback. Possible future work would be to modify the enumeration algorithm in order to allow for that and examine if it improves results at all.

Lastly, we would also like to examine the feasibility of using minimal new sets to detect trending topics. If a word or group of words appear in many minimal new sets from articles in close succession this might be an indication that they belong to a trending topic.

Chapter 7

Conclusion

We have presented the basis for a novel approach to ONED based on finding minimal new subsets from incoming news article. Through experiments we have examined the feasibility of such an approach being used for successful ONED. Furthermore we have presented and tested some ideas for further improving the data outputted by the base approach. We believe that the results of our experiments support the idea that minimal new sets could be good indicators of new events within news articles and we have shown that our current system could potentially be used for tool assisted event detection which would allow users to quickly scan through enumerations of articles and detect new events. However, further work is required in order to develop the system into a fully automatic ONED system.

Bibliography

[1] James Allan, Victor Lavrenko, and Hubert Jin. "First story detection in TDT is hard". In: *Proceedings of the ninth international conference on Information and knowledge management*. ACM. 2000, pp. 374–381.

[2] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python*. " O'Reilly Media, Inc.", 2009.

[3] Thorsten Brants, Francine Chen, and Ayman Farahat. "A system for new event detection". In: *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*. ACM. 2003, pp. 330–337.

[4] Peter Damaschke. "Pairs Covered by a Sequence of Sets". In: *20th International Symposium on Fundamentals of Computation Theory FCT 2015, Lecture Notes in Computer Science*. Springer. 2015, pp. 214–226.

[5] Giridhar Kumaran and James Allan. "Using names and topics for new event detection". In: *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. 2005, pp. 121–128.

[6] Ron Papka. *On-line new event detection, clustering, and tracking*. Tech. rep. DTIC Document, 1999.

[7] Ron Papka, James Allan, et al. "On-line new event detection using single pass clustering". In: *University of Massachusetts, Amherst* (1998), pp. 37–45.

[8] Saša Petrović, Miles Osborne, and Victor Lavrenko. "Using paraphrases for improving first story detection in news and Twitter". In: *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics. 2012, pp. 338–346.

[9] Martin F Porter, Richard Boulton, and Andrew Macfarlane. *The english (porter2) stemming algorithm*. 2002. URL: `https://tartarus.org/martin/PorterStemmer` (visited on 04/04/2016).

[10] Tdt TDT. "Annotation Manual Version 1.2". In: *From knowledge accumulation to accommodation: cycles of collective cognition in work groups* (2004).

[11] Dominik Wurzer, Victor Lavrenko, and Miles Osborne. "Twitter-scale New Event Detection via K-term Hashing". In: *Empirical Methods in Natural Language Processing* (2015), pp. 2584–2589.

[12] Yiming Yang, Tom Pierce, and Jaime Carbonell. "A study of retrospective and on-line event detection". In: *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 1998, pp. 28–36.

[13] Yiming Yang et al. "Topic-conditioned novelty detection". In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2002, pp. 688–693.

[14] Kuo Zhang, Juan Zi, and Li Gang Wu. "New event detection based on indexing-tree and named entity". In: *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2007, pp. 215–222.

Appendix A

Source Code

A.1 main.py

The driver for our implementation.

```
#!/usr/bin/env python

import sys
import argparse
import time
from pprint import pprint
from collections import defaultdict
import pdb
import regex as re
from Queue import Queue
from threading import Thread

import bag
import filters
import preprocessing as pre
from helpers import *

RESULTFILE = 'result.txt'

def loadWordbanks(wordbanks):
    """
        Input:
            wordbanks: [filepath]
        Output:
            texts: [{'text': string, 'url': string', 'date': datetime}]
            sorted in descending order by date
    """
    texts = list()
    for item in wordbanks:
        texts = loadJson(item, texts)
    return sorted(texts, key=lambda d: d['date'])
```

```
def textWithTitle(textItem):
    """
    Combines the title and text of textItem if a title exists
    """
    if 'title' in textItem:
        return u"{title}\n{text}".format(title=textItem['title'], text=textItem['text'])
    else:
        return textItem['text']

def enumerateTexts(
        texts,
        threshold,
        filterType,
        resultFile,
        useSubBags=False,
        printToJson=False,
        preFilter=False,
        useNeighbours=False):
    """
    Enumerate the texts and print the results to resultFile.
    """
    words = []
    enumeratedBags = []
    bagDict = {}
    old = []
    i = 0
    nodes = set([])
    edges = set([])
    wordFrequency = defaultdict(int)
    enumerations = []

    resultFileObject = open(resultFile, 'a')

    startTime = time.time()
    for text in texts:
        i = i+1
        sys.stdout.write("Processing: {0}/{1}".format(i, len(texts)))
        sys.stdout.flush()
        sys.stdout.write("\r")

        if useSubBags:
            subBagList = pre.to_wordlist_multi(textWithTitle(text))
            flatWords = [y for x in subBagList for y in x]
            wordsToFilter, wordFrequency, tfidfList = filters.filterWordList(flatWords,
            ↪  wordFrequency, filterType, threshold, i)
            words = [set(x) for x in subBagList]
        else:
            words = pre.preprocess(textWithTitle(text))
            wordsToFilter, wordFrequency, tfidfList = filters.filterWordList(words,
            ↪  wordFrequency, filterType, threshold, i)
            words = set(words)

        if useSubBags:
            if useNeighbours:
                enumeration, bagDict = bag.enumerateMultiBagWithNeighbours(words,
                ↪  enumeratedBags, bagDict)
            else:
```

```
                enumeration, bagDict = bag.enumerateMultiBag(words, enumeratedBags, bagDict)
            else:
                if preFilter:
                    enumeration, bagDict = bag.enumerateBag(wordsToFilter, enumeratedBags,
                     ↪  bagDict)
                else:
                    enumeration, bagDict = bag.enumerateBag(words, enumeratedBags, bagDict)

            filteredEnumeration = filters.filter_enumeration(enumeration, wordsToFilter,
             ↪  tfidfList)
            textCopy = dict(text)
            textCopy['enumeration'] = filteredEnumeration

            if printToJson:
                enumerations.append(textCopy)

            printEnumerationToFileObject(
                text['url'],
                textWithTitle(text),
                filteredEnumeration,
                resultFileObject,
                tfidfList
            )

            if useSubBags:
                enumeratedBags.append([bag.getSubsets(x, 1) for x in words])
            else:
                enumeratedBags.append(bag.getSubsets(words, 1))

            if filteredEnumeration == set([]):
                old.append(text['url'])

        endTime = time.time()
        sys.stdout.write("Processing: {0}/{1}\n".format(i, len(texts)))
        sys.stdout.write("Done!\n")
        sys.stdout.write("Completed in {time}s.".format(time=endTime-startTime))
        resultFileObject.write("Completed in {time}s.".format(time=endTime-startTime))

        resultFileObject.close()

        return enumerations, old


def main(threshold,
         filterType,
         wordbanks,
         resultFile=RESULTFILE,
         useSubBags=False,
         printJson=False,
         useNeighbours=False,
         preFilter=False):
    """
    Driver of the main stuff:
    load the wordbanks, enumerate the bags, print the results
    """
    texts = loadWordbanks(wordbanks)
```

```
    #replace result-file contents with header
    with open(resultFile, 'w') as f:
        s = "Initializing run\nInputFiles: {input}\nFilter: {filter}\nThreshold:
        ↪  {threshold}\nPre-filter: {preFilter}\nSubBags: {useSubBags}\nNeighbours:
        ↪  {useNeighbours}\n========================================\n"
        f.write(
            s.format(
                input=wordbanks,
                filter=filterType,
                threshold=threshold,
                preFilter = preFilter,
                useSubBags = useSubBags,
                useNeighbours = useNeighbours,
            )
        )


    enumerations, old = enumerateTexts(
            texts,
            threshold,
            filterType,
            resultFile,
            useSubBags,
            printJson,
            preFilter,
            useNeighbours)

    #print as json
    if printJson:
        printEnumerationJson(enumerations, resultFile)


def massRun():
    thresholds = [x*0.1+0.05 for x in range(0,10)]
    filters = ['cf', 'df', 'tfidf', 'none']
    articles = [
                '../data/ww2.jl'
            ]
    useSubBags = [True, False]
    filenameForm = "results/ww2-{filter}-{threshold}.txt"
    for f in filters:
        for t in thresholds:
            #for s in useSubBags:
                #subBagStr = '-sub' if s else ''
                filename = filenameForm.format(filter=f, threshold=t)
                main(t, f, articles, filename, False)
                if f=='none':
                    break


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Process the provided data")
    parser.add_argument('-m', '--massRun', action='store_true')
    parser.add_argument('-s', '--subBags', action='store_true')
    parser.add_argument('-n', '--neighbours', action='store_true')
    parser.add_argument('-t', '--threshold', type=float, default=10)
    parser.add_argument('-f', '--filter', default='none')
```

```
    parser.add_argument('-p', '--preFilter', action='store_true')
    parser.add_argument('-j', '--printToJson',action='store_true')
    parser.add_argument('-o', '--output', default=RESULTFILE)
    parser.add_argument('articles', nargs='*')
    args = parser.parse_args()

    if args.massRun:
        massRun()
    else:
        main(
                args.threshold,
                args.filter,
                args.articles,
                resultFile=args.output,
                useSubBags=args.subBags,
                useNeighbours=args.neighbours,
                printJson=args.printToJson,
                preFilter=args.preFilter
            )
```

A.2 bag.py

The main code for enumeration calculations and bag handling.

```
#!/usr/bin/env python

"""Functions for enumerating and working with bags"""

__author__ = "Orn Gudjonsson"

from collections import defaultdict

inf = float("+inf")

def getSubsets(x, n):
    """
    Given a set or a list returns all possible subsets of size n
    """
    if n == 1:
        return frozenset([frozenset([i]) for i in x])

    subsets = []
    for item in x:
        for y in getSubsets(x, n-1):
            subset = y | frozenset([item])
            if len(subset) == n:
                subsets.append(subset)
    return frozenset(subsets)


def f(x, bags):
    i = 1
    for bag in bags:
        if isSubset(x, bag):
```

```
            return i
        i = i+1

def isNewAtM(x, bags, bagDict, m):
    if bagDict.get(x, inf) < m:
        return False

    retval = True

    if len(x) > 1:
        yvalMin = inf
        for y in x:
            yval = bagDict.get(y, inf)
            if yval < inf and isSubset(x, bags[yval-1]):
                retval =  False
                yvalMin = min(yvalMin, yval)
    if retval:
        bagDict[x] = m
    else:
        bagDict[x] = yvalMin
    return retval

def isNewAtMMulti(x, bags, bagDict, (m, s)):
    """
    bagDict now stores (k, [i]) where k is parent bag number
    and [i] is a list of all subbags of k containing i
    """
    lookupTuple = bagDict.get(x, (inf, []))
    if lookupTuple[0] < m:
        return False
    elif lookupTuple[0] == m:
        bagDict[x] = (m, lookupTuple[1]+[s])
        return True

    retval = True

    if len(x) > 1:
        yvalMin = inf
        ySubsMin = []
        for y in x:
            (yval, ySubs) = bagDict.get(y, (inf, []))
            foundSubBags = []
            isFound = False
            for subBag in ySubs:
                if yval < inf and isSubset(x, list(bags[yval-1])[subBag]):
                    retval = False
                    isFound = True
                    foundSubBags.append(subBag)
            if isFound:
                if yval < yvalMin:
                    yvalMin = yval
                    ySubsMin = foundSubBags

    if retval:
        #completely new, store it
        bagDict[x] = (m, [s])
    else:
```

```
            bagDict[x] = (yvalMin, ySubsMin)
        return retval

def isSubset(a,b):
    return a-b == set([])


def enumerateBagHelper(newBag, bags, bagDict, n, i, isNewAtMFunc=isNewAtM):
    newSets = []
    subsets = getSubsets(newBag, n)
    for subset in subsets:
        if isNewAtMFunc(subset, bags, bagDict, i):
            newSets.append(subset)
    return set(newSets)

def enumerateBag(newBag, bags, bagDict):
    """
    Performs enumeration using the algorithm described in section 2.2

    Args:
        newBag: a list of words to enumerate
            :: [string]
        bags:  a list of previously seen bags. Each bag is a set of frozensets
            :: [set([frozenset([string])]]
        bagDict: a dictionary mapping previously seen sets to the bag nr
                (1-indexed) where they were first seen.
            :: dict(frozenset([string])) | dict(frozenset([frozenset([string]))))
    """
    enumeration = set([])
    for n in range(1, 3):
        enumeration = enumeration | enumerateBagHelper(newBag, bags, bagDict, n, len(bags) +
        ↪  1)
        newBag = getSubsets(newBag, n) - enumeration
        if newBag == set([]):
            break
    return (enumeration, bagDict)


def enumerateMultiBag(newBags, bags, bagDict):
    enumeration = set([])
    bagNr = len(bags) +1
    subsetNr = 0
    for subBag in newBags:
        for n in range(1,3):
            enumeration = enumeration | enumerateBagHelper(subBag, bags, bagDict, n, (bagNr,
            ↪  subsetNr), isNewAtMMulti)
            subBag = getSubsets(subBag, n) - enumeration
            if subBag == set([]):
                break

        subsetNr += 1
    return (enumeration, bagDict)

def enumerateMultiBagWithNeighbours(newBags, bags, bagDict):
    """
    Enumerate using the sub bag approach, but use a sliding window of 3 subbags as bag
    """
```

```
    enumeration = set([])
    index = 0
    compositeBags = []
    while index + 2 < len(newBags):
        first  = newBags[index]
        second = newBags[index+1]
        third = newBags[index+2]
        compositeBag = set(first | second | third)
        compositeBags.append(compositeBag)
        enumeration = enumeration | enumerateBagHelper(compositeBag, bags, bagDict, 1,
        ↪  len(bags) + 1)
        index += 1

    for bag in compositeBags:
        pairBag = getSubsets(bag, 1) - enumeration
        enumeration = enumeration | enumerateBagHelper(pairBag, bags, bagDict, 2, len(bags)
        ↪  + 1)
    return (enumeration, bagDict)

def enumerate(bags):
    """
    Enumerates a list of 'bags'
    """
    enumeratedBags = []
    bagDict = {}
    i = 1
    for bag in bags:
        newEnumeration, bagDict = enumerateBag(bag, enumeratedBags, bagDict, i)
        enumeratedBags.append(newEnumeration)
        i = i+1
    return enumeratedBags, bagDict


def enumerationToGraph(pairs):
    """Given an enumeration (a set of frozensets with one or two elements)
    returns the set of nodes (words involved in pairs) and the set of edges
    (pairs)
    """
    #nodes are the words that are involved in pairs
    nodes = set([])
    for pair in pairs:
        for elem in pair:
            nodes.add(elem)
    return nodes, set(pairs)


def nodeDegrees(edges):
    """
    Given a set of eges, return a list of tuples where
    the first element is a node and the second is the degree of that node,
    that is the number of edges it is included in

    Input:
        edges: a set of edges on the form
            frozenset([frozenset([a]), frozenset([b])])

    Ouput:
```

```
        a list of tuples [(word, degree)]
    """
    vertices = defaultdict(int)
    for edge in edges:
        for vertex in edge:
            vertices[list(vertex)[0]] += 1

    return sorted(zip(vertices.keys(), vertices.values()), key=lambda x: x[1], reverse=True)
```

A.3 filters.py

Contains filter related calculations.

```
#!/usr/bin/env python

"""Filters and filter helper functions"""

__author__ = "Orn Gudjonsson"

from math import log
from collections import defaultdict

def filterWordList(words, wordFrequency, filterType, threshold, docuCount):
    """
        Input:
            words: the list of words to filter
            wordFrequency: a dictionary counting the document frequency of words
            filterType: the type of filtering to use
            threshold: the threshold to use for filtering
            docuCount: the number of documents previously seen
        Output:
            wordsToFilter: words that should be filtered based on filterType and
                           threshold
            wordFrequency: updated dictionary with document frequencies
    """
    wordsToFilter = []
    tfidfList = []
    if filterType == 'cf':
        wordFrequency  = collection_frequency(words, wordFrequency)
        wordsToFilter, tfidfList = filter_common(words, wordFrequency, threshold)
    elif filterType == 'df':
        wordFrequency = document_frequency(words, wordFrequency)
        wordsToFilter, tfidfList = filter_documentFrequency(words, wordFrequency, threshold)
    elif filterType == 'tfidf':
        wordsToFilter, tfidfList = filter_tfidf(words, wordFrequency, threshold, docuCount)
        wordFrequency = document_frequency(words, wordFrequency)
    elif filterType == 'none':
        wordsToFilter, tfidfList = filter_tfidf(words, wordFrequency, len(words), docuCount)
        wordFrequency = document_frequency(words, wordFrequency)
    else:
        raise Exception("Invalid filter type" + filterType)

    return wordsToFilter, wordFrequency, tfidfList
```

```
def filter_enumeration(enumeration, whitelist, tfidfList):
    words = [x for x in enumeration if len(x) == 1]
    pairs = [x for x in enumeration if len(x) > 1]
    retWords = removeWords(whitelist, words)
    retPairs = removePairs(whitelist, pairs)
    retPairs = filter_pairs_tfidf(retPairs, tfidfList)
    return retWords + retPairs

def pair_tfidf(pair, tfidfDict):
    pairList = list(pair)
    first = list(pairList[0])[0]
    second = list(pairList[1])[0]
    tfidf = tfidfDict[first] + tfidfDict[second]
    return tfidf

def filter_pairs_tfidf(pairs, tfidfList):
    return pairs #remove this to filter top tfidf scoring words
    tfidfDict = {x: y for (x,y) in tfidfList}
    retPairs = []
    for pair in pairs:
        tfidf = pair_tfidf(pair, tfidfDict)
        if(tfidf >= 0.5): #TODO: magic number!
            retPairs.append(pair)
    return retPairs

def document_frequency(wordList, docFreqDict):
    """
    Add 1 for each unique item in wordList to docFreqDict
    """
    for word in set(wordList):
        docFreqDict[word] += 1
    return docFreqDict

def collection_frequency(wordList, freqDict):
    """
    Add the number of a currances of each word to freqDict
    """
    for word in wordList:
        freqDict[word] += 1
    return freqDict

def term_frequency(wordList):
    """
    Given a list of words, returns a dict mapping words
    to how often they appear in the list in proportion to the length of the
    wordlist.
    """
    freqDict = defaultdict(float)
    freqDict = collection_frequency(wordList, freqDict)
    l = float(len(wordList))
    for key, value in freqDict.iteritems():
        freqDict[key] /= l
    return freqDict

def filter_common(wordList, frequencyDict, threshold):
    """
    Filter the word list based on the top threshold most common items
```

```
    in the frequencyDict
    """
    freqTuples = sorted(frequencyDict.items(), key = lambda x: x[1], reverse=True)
    wordsToInclude = [x[0] for x in freqTuples[:int(threshold)]]
    return wordsToInclude, freqTuples

    totalFreq = 0
    total = sum(frequencyDict.values())
    mostFrequent = []
    if total == 0:
        return []
    for i in range(int(threshold*len(frequencyDict))):
    #while totalFreq/float(total) < threshold and len(freqTuples) > 1:
        nextTuple = freqTuples[i]

        frequency = nextTuple[1]
        mostFrequent.append(nextTuple[0])
        totalFreq += frequency
    return mostFrequent

def filter_documentFrequency(wordList, frequencyDict, threshold):
    freqTuples = sorted(frequencyDict.items(), key = lambda x: x[1])
    wordsToInclude = [x[0] for x in freqTuples[:int(threshold)]]
    return wordsToInclude, freqTuples


def filter_overThreshold(wordList, frequencyDict, threshold, seenDocuments):
    """
    Filter the word list base on the frequencyDict
    anything above the given threshold is removed
    """
    wordsToFilter = []
    if seenDocuments == 0:
        return wordsToFilter

    n = float(seenDocuments)
    for word in set(wordList):
        if frequencyDict[word]/n > threshold:
            wordsToFilter.append(word)
    return wordsToFilter


def tfidf(word, termFrequencies, docFrequencies, docNumber):
    """
    Input:
        word: "someword"
        termFrequencies: {"someword": 0.3, "someotherword": 0.7}
        docFrequencies: {"someword": 40, "otherword": 10, "foo": 1}
        docNumber: 50
    Ouput:
        the tf-idf of word, i.e. 0.4
    """
    df = docFrequencies[word]
    tf = termFrequencies[word]
    idf = log(docNumber+1/(float(df+1)))
    return tf*idf
```

```
def filter_tfidf(wordList, dfDict, threshold, n):
    """
    Filter the given wordlist using td-idf.
    Using the document frequency dict dfDict
    Any item who's td-idf score is below threshold is ignored
    n is the number of documents
    """
    tfDict = term_frequency(wordList)
    tfidfTuples = []
    td_idf = 0
    for word in set(wordList):
        tf_idf = tfidf(word, tfDict, dfDict, n)
        tfidfTuples.append((word, tf_idf))
    sortedItems = sorted(tfidfTuples, key=lambda x: x[1], reverse=True)
    wordsToFilter = [x[0] for x in sortedItems[:int(threshold)]]
    return wordsToFilter, tfidfTuples


def removeListFromList(filterList, wordList):
    return filter(lambda x: x not in filterList, wordList)

def removeFilterWords(filterList, wordList):
    """
    input:
        filterList: ['foo', 'baz']
        wordList: [frozenset(['foo']), frozenset(['bar'])]
    output: ['bar']
    """
    words = [y for x in wordList for y in x]
    return removeListFromList(filterList, words)

def removeWords(whitelist, wordList):
    """
    input:
        filterList: ['foo', 'baz']
        wordList: [frozenset(['foo']), frozenset(['bar'])]
    output: [frozenset(['bar'])]
    """
    filterSetList = [frozenset([x]) for x in whitelist]
    return filter(lambda x: x in filterSetList, wordList)

def removePairs(whitelist, pairList):
    return filter(lambda s: isSetItemInList(whitelist, s), pairList)

def flattenPairSet(pairSet):
    """
    input: frozenset([frozenset(['foo']), frozenset(['bar'])])
    output: frozenset(['foo', 'bar'])
    """
    return frozenset([y for x in pairSet for y in x])

def isSetItemInList(filterList, pairSet):
    """
    input:
        pairSet: frozenset([frozenset(['foo']), frozenset(['bar'])])
        filterList = ['foo', 'baz']
```

```
    output:
        true if an item in filterlist is in any subset of pairset
        false otherwise
    """
    flatPairs = flattenPairSet(pairSet)
    for item in flatPairs:
        if item in filterList:
            return True
    return False
```

A.4 preprocessing.py

Contains code that's used for preprocessing.

```
#!/usr/bin/env python

"""Module containing preprocessing stuff"""

__author__ = "Orn Gudjonsson"

from stemming.porter2 import stem
import nltk
from nltk.corpus import stopwords
import regex as re
import string


def stem_words(wordlist):
    """
    Given a string returns a list of all the words, stemmed.
    """
    return [stem(x) for x in wordlist]

def remove_stopwords(words):
    """
    Given a list of english words returns the list with all stopwords removed.
    """
    swords = set(stopwords.words('english'))
    return [x for x in words if x not in swords]

def remove_duplicates(words):
    return list(set(words))

def remove_punctuation(text):
    """
    Input:
        unicode text
    Ouput:
        unicode text with removed punctuation
    """
    table = {ord(c): u' ' for c in string.punctuation}
    return text.translate(table)

def remove_numbers(words):
```

```
    return filter(lambda x: not x.isdigit(), words)

def tokenize(text):
    """
    Given a string outputs a list of words
    """
    return nltk.word_tokenize(text)


def to_wordlist(text):
    return stem_words(
        remove_stopwords(
            tokenize(
                remove_punctuation(
                    text.lower()
                )
            )
        )
    )

def get_sentences(text):
    return nltk.sent_tokenize(text)

def to_wordlist_multi(text):
    sentences = get_sentences(text)
    return [tuple(to_wordlist(sentence)) for sentence in sentences]


def preprocess(string):
    return to_wordlist(string)
```

A.5 helpers.py

Utility functions for printing out results etc.

```
#!/usr/bin/env python

import json
import sys

import preprocessing as pre
import bag

def loadJson(filename, texts):
    """
    load the given json lines file and return an array of whatever the lines
    contain
    """
    with open(filename, 'r') as f:
        for line in f:
            texts.append(json.loads(line))
    return texts

def stringify(s):
```

```
    """Converts a set to string"""
    if type(s) not in map(type, [set([]), frozenset([])]):
        return unicode(s)
    l = list(s)
    isMultiSet = len(l) > 1
    if isMultiSet:
        retStr = "("
        for item in l[:-1]:
            retStr += stringify(item) + ', '
        retStr += stringify(l[-1])
        retStr += ")"
    else:
        retStr = ""
        retStr += stringify(l[0])
    return retStr

def outputToString(output):
    """Converts a list of sets to string"""
    if output == []:
        return "{}"
    retStr = ''
    for item in output[:-1]:
        retStr += stringify(item) + ', '
    retStr += stringify(output[-1])
    return '{' + retStr + '}'

def distanceDictToString(distanceDict):
    fmtStr = u'{pair}: {distance}'
    output = ""
    if distanceDict == {}:
        return ""
    valueList = sorted(distanceDict.items(), key=lambda x: x[1])
    for (k,v) in valueList[:-1]:
        output += fmtStr.format(pair=stringify(k), distance=v)
        output += ", "
    output += fmtStr.format(pair=stringify(valueList[-1][0]), distance=valueList[-1][1])
    return output

def frozenPairToTuple(pair):
    pairList = list(pair)
    first = list(pairList[0])[0]
    second = list(pairList[1])[0]
    return (first, second)


def tfidfPairsToString(pairs, tfidfDict):
    if len(pairs) == 0:
        return "{}"
    retString = ""
    fmtString = u"{pair}: {tfidf}, "
    for pair in sorted(pairs, key=lambda x: list(x)[1]):
        p = frozenPairToTuple(pair)
        tfidf = tfidfDict[p[0]] + tfidfDict[p[1]]
        retString += fmtString.format(pair=stringify(pair), tfidf=tfidf)
    return retString
```

```
def nodeDegreesToString(nodeDegrees):
    res = ""
    formatString = u"{node}: {degree}, "
    for nodeTuple in nodeDegrees:
        res += formatString.format(node=nodeTuple[0], degree=nodeTuple[1])
    return res

def find(lst, item):
    """
    returns list of indices where item is within list
    empty list if not precent in list
    """
    return [i for i, x in enumerate(lst) if x==item]

def distanceDictFromPairs(pairList, textList):
    distDict = {}
    for pair in pairList:
        p = list(pair)
        distDict[pair] = calculateWordDistance(p[0], p[1], textList)
    return distDict

def extractString(s):
    """
    input: frozenset([frozenset([...frozenset(['somestring'])...)
    output: somestring
    """
    if type(s) in map(type, [u'', '']):
        return s
    if len(s) == 0:
        return ""
    return extractString(list(s)[0])

def calculateWordDistance(word1, word2, textList):
    word1 = extractString(word1)
    word2 = extractString(word2)
    indexes1 = find(textList, word1)
    indexes2 = find(textList, word2)
    minDist = sys.maxint
    for i1 in indexes1:
        for i2 in indexes2:
            dist = abs(i1-i2)
            minDist = min(dist, minDist)
    return minDist

def enumerationToJsonable(enumeration):
    """
    Turns an enumeration set into datatypes that can be serialized
    Input:
        enumeration: set([frozenset['newWord'], frozenset([frozenset(['new']),
↪    frozenset(['pair'])])])
    Output:
        [(newWord,), ('new', 'pair')]
    """
    retval = []
    for item in enumeration:
        if len(item) > 1:
            newItem = [tuple(x) for x in item]
```

```
        else:
            newItem = tuple(item)
        retval.append(newItem)
    return retval


def printEnumerationJson(enumList, filename):
    """
    Input:
        enumList: [{k:v, },]
        filename: /some/file.txt
    Output: None, prints each item as jsonline to filename
    """
    with open(filename, 'w') as f:
        for enumeration in enumList:
            enum = list(enumeration['enumeration'])
            enum = enumerationToJsonable(enum)
            enumeration['enumeration'] = enum
            jsonString = json.dumps(enumeration)
            f.write(jsonString+'\n')

def printEnumeration(url, words, enumeration, filename):
    enumerationString = enumerationToString(url, words, enumeration)
    with open(filename, 'a') as f:
        f.write(enumerationString)

def printEnumerationToFileObject(url, words, enumeration, fileobject, tfidfList):
    enumerationString = enumerationToString(url, words, enumeration, tfidfList)
    fileobject.write(enumerationString)


def enumerationToString(url, words, enumeration, tfidfList):
    lineString = u'Url: {url}\nWords: {words}\nNew Words: {newWords}\nNew Pairs:
    ↪  {pairs}\nNodes: {nodes}\n\n\n'
    newWords = [x for x in enumeration if len(x) < 2]
    pairs = [x for x in enumeration if len(x) >= 2]
    tfidfDict = {x: y for (x,y) in tfidfList}
    newWords = sorted(newWords, key=lambda frozenWord: -tfidfDict[list(frozenWord)[0]])
    pairs = sorted(pairs, key=lambda frozenPair: -sum(tfidfDict[x] for x in
    ↪  frozenPairToTuple(frozenPair)))
    newWordStr = outputToString(newWords)
    textList = pre.to_wordlist(words)
    pairStr = outputToString(pairs)
    nodes, edges = bag.enumerationToGraph(pairs)
    nodeDegrees = bag.nodeDegrees(edges)
    nodeDegreeStr = nodeDegreesToString(nodeDegrees)
    lineString = lineString.format(
            url=url,
            words=words,
            newWords=newWordStr,
            pairs=pairStr,
            nodes=nodeDegreeStr,
        )
    return lineString.encode('UTF-8')
```