UNIVERSITAT DE BARCELONA

Bachelor's Thesis

COMPUTER SCIENCE DEGREE

Facultat de Matemàtiques
Universitat de Barcelona

News Similarity with Natural Language Processing

Author: Álvaro Parafita Martínez

Advisor: Dr. Jordi Vitrià Marca
Affiliation: Departament de Matemàtica Aplicada i Anàlisi, UB

Barcelona, January 28, 2016

Abstract. News articles are pieces of Natural Language that comply with the model of 5W1H, meaning, they should answer to the following six questions: What, Who, Where, When, Why and How. This project takes advantage of that assumption to create an algorithm capable of building a representation of a news article and a distance between such representations for any pair of politics news. With that knowledge, a global distance between entries based on similarity of content is built. That algorithm is assessed in comparison with the topic modeling algorithm Latent Dirichlet Allocation (LDA). Applications of the system with their corresponding visualisations are presented too.

Keywords: Natural language processing, Artificial intelligence

Contents

Introduction

Motivation

Our world is packed with information. That is surely a great opportunity for many subjects, but it can be overwhelming for the common user.

Considering the news world, nowadays it is possible to access news from every country quite easily, and given the right keywords, it should be possible to retrieve information from almost every story. But for users, when it is not desired to search for a specific news story, but find one that fulfills the user taste, that information overload makes it difficult to find the desired article. That is a huge problem that can lead to information blindness, being subject to the narrow visions of the feeds the user agrees to follow and missing the vast landscape that is available outside.

There are many examples of problems driven by this situation, problems that raise some questions.

- Is there a way to see just in a glance all the major stories that happened in a day?
- Is it possible to get the most accurate depiction of a specific story available on the web?
- Is there a way to see how a given story still has interest days after happening?

All these questions are not easily answered, even for a human with the right dataset and the needed amount of time. But it is clear that they have some interest.

The main motivation of this work is to extract some kind of meaning from news articles in order to connect them, getting to know, in a programmatically way, the subject they are talking about and how closer two articles are just by the words they are using. More specifically, the objective of this work is to define an algorithm capable of knowing how similar two news articles are. That way, information can be extracted based on the network that those similarities build. The proposed algorithm is written entirely in Python, except for the visualisations, that are made using the Pandas wrapping of Matplotlib and the famous Javascript Library D3 (Data Driven Documents).

In the following sections, that algorithm is described and assessed comparing it to an stablished topic modeling algorithm, Latent Dirichlet Allocation, known as LDA.

Natural Language Processing

By "Natural Language" it is meant any language used for everyday communications, such as English, Spanish, Chinese, etc. Natural Language Processing is a discipline that tries to process any of those languages. Its main difficulty is that Natural Language is an unstructured source of information, in contrast to Formal Languages such as Programming Languages or Mathematical Notations.

Natural Languages deal with lots of ambiguity, interpretations driven by context and are constantly evolving, making it more difficult to learn and process. All these characteristics make it a tough discipline but it is nonetheless an interesting problem to solve.

Natural Language Processing (NLP) is becoming widespread in nowadays society, due to the improvements in hardware -making it possible to execute the complex and demanding algorithms it deals with- and due to the evolution of society to a more than ever connected world. Technologies such as predictive text, handwriting technologies, machine translation like Google Translate or Information Retrieval systems all use NLP techniques. NLP covers many use cases, and more will arise as long as the field continues to spread.

This work uses NLP in its most crucial step: translating unprocessed text to a machine readable structure (hashmaps), that will be used to compare two news articles. It is also used when comparing those hashmaps, creating a pseudo-distance using Wikipedia Articles, a great source of written information that makes it possible to compare Named Entities, such as Barack Obama, Hillary Clinton or Vladimir Putin.

Since the techniques used in the algorithm described in this work are all encapsulated in a single library, *spaCy*, it could seem that NLP has no importance in the process whatsoever; that is absolutely incorrect. In order to reflect that crucial part and to make it possible to understand how those methods work, there is a summary of the NLP techniques needed for this project in the following chapter.

Why Python?

Since this project is not production-oriented, problems like optimisation are not a priority, even when some processes have taken hours to work. Python is an excellent language in terms of data analysis and exploration, which is the most important step in this work. It has a huge community and a vast landscape of libraries for all kinds of disciplines; NLP is one of them.

Flexibility, code clarity and code simplicity are essential in any data exploration and Python is the shining star in this aspect. The flexibility of the language makes it possible to write really complex methods almost impossible in other languages, and that is an essential advantage for this project, because the algorithm has suffered major changes during its creation.

Organisation

This work will be organized in the following way:

- Chapter 1 introduces the basic concepts of Natural Language Processing needed throughout this project.
- Chapter 2 explains the two main approaches considered for this problem, one closer to First Order Logic and the other more centred on the characteristics of news articles. Both are discussed and the latter will be chosen, with some considerations, given the limitations that have been faced during the process.
- Chapter 3 discusses the implementation of the algorithm in depth, detailing all technologies and Python libraries used in the process. It also depicts all decisions made for each of its steps.
- Chapter 4 is split in two halves. The first part describes the results of the system, comparing it, as said before, with LDA, which is considered the "ground truth" to base the results on. The second one shows two applications of the system, discussed along with its visualisations.
- Chapter 5 details the conclusions of this project and proposes other approaches and further work for the problem.

Chapter 1

Fundaments of NLP

This chapter explains the basic concepts of Natural Language Processing needed throughout this work. For any of the following considered approaches, there are several NLP challenges to face.

Each of the following NLP-processes will be accompanied by an example using the *spacy* library, a NLP toolkit optimised both for accuracy and performance, with a simple and easy-to-use Python API. Its documentation can be found in [1].

Many different approaches and optimisations are used with those algorithms and processes, but it is not the point of this work to enumerate them. For further details, [2] discusses each NLP problem in depth.

For any of the following code snippets, it is assumed that the following lines have been executed previously:

```
>>> from spacy.en import English
>>> nlp = English()
>>> text = 'By "Natural Language" it is meant any language used for everyday
    communications, such as English, Spanish, Chinese, etc. Natural Language
    Processing is a discipline that tries to process any of those languages. '
>>> doc = nlp(text)
```

1.1 Word and sentence segmentation

Word segmentation tries to split a text string in a list of strings, each of them being a separate word of the text. That way words can be considered as separate objects to work with.

Sentence segmentation splits a text and returns a list of strings, each of them being a separate sentence in the text. This process is generally combined with word segmentation, then returning lists of lists of words (each list is a sentence and each sublist is the list of words of that sentence).

These processes are solved thanks to supervised learning, with a proper dataset for each of them.

An example of word segmentation is:

```
>>> for i, token in enumerate(doc):
...     print(token)
...     if i == 10: break

By
"
Natural
Language
"
it
is
meant
any
language
used
```

doc is an iterable object; its iterator returns a list of *spacy.Token* instances, each of them containing a single word. That Token has a *__str__* method that returns the word itself; that is why by printing token, its word appears.

Sentence segmentation can be achieved by a similar manner. The following code is more complex to get a more compact representation, but the spacy API is nevertheless quite simple: just by calling doc.sents, a sentence iterator is returned.

```
>>> print([[str(token) for token in sent] for sent in doc.sents])

[['By ', '"', 'Natural ', 'Language', '" ', 'it ', 'is ', 'meant ', 'any ', '
    language ', 'used ', 'for ', 'everyday ', 'communications', ', ', 'such ', 'as
     ', 'English', ', ', 'Spanish', ', ', 'Chinese', ', ', 'etc', '. '], ['Natural
     ', 'Language ', 'Processing ', 'is ', 'a ', 'discipline ', 'that ', 'tries ',
     'to ', 'process ', 'any ', 'of ', 'those ', 'languages', '. ']]
```

It is worth mentioning that the Token object has some attributes for an easy use of the token, like "is_punct", "is_lower" or "is_title". "is_punct", for example, can be useful if it is desired to avoid punctuation signs as tokens.

1.2 POS tagging

Given those segmented words, the next step is to POS-tag them.

Part-Of-Speech refers to the grammatical category assigned to any given word, such as N (noun), V (verb), ADJ (adjective) and so on. Each NLP library and dataset uses its own terminology, though there are some standards motivated by the most used datasets, like Penn Treebank POS Tagset or the Brown Corpus Tagset.

This process is solved by sequentially categorising each token (word) in every sentence joined with the N previous words, making tuples of words known as N-Grams (unigrams, bigrams, trigrams...). Those N-Grams are categorised using a tagged dataset and a classifier algorithm.

POS-tagging in spaCy is quite simple; by retrieving each token in a text, that Token instance has a *pos* and *pos_* attributes that store the POS category of a word. *pos* is an

integer identifier of that POS category and *pos_* a string representation of that category. The following example returns tuples of each word in the first sentence with its POS tag and its name.

```
>>> print(*[
...     (str(token), token.pos, token.pos_)
...     for token in list(doc.sents)[0]
... ], sep='\n')

('By ', 2, 'ADP')
('"', 13, 'PUNCT')
('Natural ', 8, 'NOUN')
('Language', 8, 'NOUN')
('" ', 13, 'PUNCT')
('it ', 8, 'NOUN')
('is ', 16, 'VERB')
('meant ', 16, 'VERB')
('any ', 1, 'ADJ')
('language ', 8, 'NOUN')
('used ', 16, 'VERB')
('for ', 2, 'ADP')
('everyday ', 1, 'ADJ')
('communications', 8, 'NOUN')
(', ', 13, 'PUNCT')
('such ', 1, 'ADJ')
('as ', 2, 'ADP')
('English', 8, 'NOUN')
(', ', 13, 'PUNCT')
('Spanish', 8, 'NOUN')
(', ', 13, 'PUNCT')
('Chinese', 8, 'NOUN')
(', ', 13, 'PUNCT')
('etc', 17, 'X')
('. ', 13, 'PUNCT')
```

1.3 Chunking

The previous word-POStag pairs are joined at the sentence level to form segments and labels of multitoken sequences, on what is called "chunks", such as Noun Phrases (NP), Verb Phrases (VP) or Prepositional Phrases (PP), among others. Those chunks can be part of other chunks, thus being part of a tree that has its root at a node called S (Sentence). This is where traditional grammar would take place, decomposing complex sentences in a tree structure.

Chunking is made with a classifier that uses N-Grams of word-POStag pairs and the tree structure is built on those chunks with the help of Context-Free Grammars (CFG), that state some rules to join chunks in super-chunks, thus forming the tree with those chunks as nodes. Given the CFG, the tree can be built using several algorithms, such as Recursive Descent Parsing, a top-bottom approach, or Shift-Reduce Parsing, a bottom-up parser.

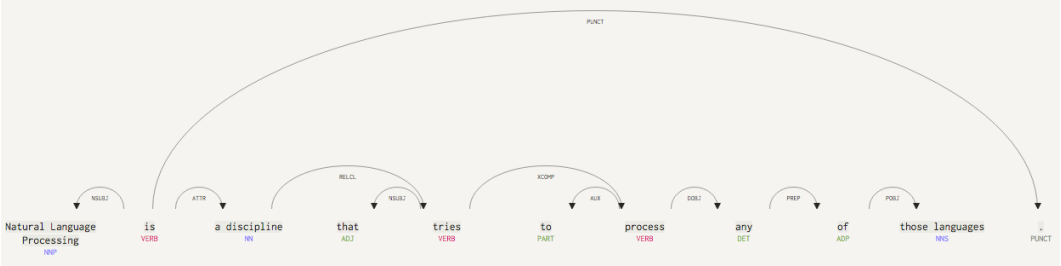

Figure 1.1: Dependency graph of a sentence

Based on those chunks, dependency graphs can be built, such as figure 1.1. This graph was built with displaCy (https://api.spacy.io/displacy/index.html), an online tool based on spaCy.

1.4 Named Entities

Named Entity is a concept related to Proper Nouns. It takes the chunks from the previous step and selects all the Noun Phrases, classifying them as Named Entities of several types using a tagged dataset and a proper classifier.

Obtaining all Named Entities and their types with spaCy is quite simple:

```
>>> text2 = "The Washington Monument is the most prominent structure in Washington
    , D.C. and one of the city's early attractions. It was built in honor of
    George Washington, who led the country to independence and then became its
    first President."
>>> doc2 = nlp(text2)
>>> print([
...     (ent.text, ent.label, ent.label_)
...     for ent in doc2.ents
... ])

[('The Washington Monument', 202115, 'ORG'), ('Washington', 85248, 'GPE'), ('D.C.'
    , 85248, 'GPE'), ('George Washington', 28061, 'PERSON'), ('first', 1499632, '
    ORDINAL')]
```

Each of those labels will be used later to filter Named Entities, and those Named Entities will be used to represent a news article.

Chapter 2

Approaches

In order to compare news articles and get to know which of them are more similar than the others, the obvious step would be to build a distance between articles.

This is the objective of the algorithm: to create a function that takes two entries and, given its text, creates two vectors that will be compared with a distance function. This being the case, this algorithm faces two problems: how to represent an entry as a vector and which function to define that takes two of those vectors as input and outputs a real non-negative number that reflects how close those two entries are.

During the development of this project, two different approaches were considered, one more theoretical and one more practical, being the latter the selected one. The two following sections describe those approaches and the last one explains how the second approach has been simplified for the scope of this project.

2.1 First-Order Logic

This approach tries to process a text sentence by sentence, extracting information from each of those sentences and storing it for later comparison.

Given the tree structure of any sentence, the use of another Context-Free Grammar to translate the tree into First-Order Logic (FOL) propositions allows, together with the propositions of all the other sentences in the text, to form a representation of the knowledge contained in it. This, together with the previous processes explained in Fundaments of NLP, are the basic steps towards the goal of representing the knowledge of any given text.

The First-Order Logic approach proposed for the problem of defining a distance between entries consists of a FOL representation of an entry and the definition of a distance between such representations with the help of the rules of FOL. For example, if a given action or verb (a relationship between elements in FOL) applied to the same subject and object of the sentence (elements affected by the relationship) appears in both texts, that would mean that those texts were similar in the sense that both share that knowledge. Doing that process with any pair of entries and qualifying those correspondences with a Data-Driven approach (such as using an optimisation algorithm with a tagged dataset)

could very well define the desired distance function.

Using the FOL rules, that could be refined. Given this example:

$A \rightarrow B$
A

By the rules of FOL, these propositions imply B. In case an entry had sentences that meant these propositions and another had a sentence that meant B, those two entries should consider B as a shared knowledge, thus affecting the distance between the two.

With this approach, given the right dataset and a properly-tuned system that could represent an entry perfectly with FOL it should be possible to fulfill the objective.

But that would be a huge assumption. For starters, none of the NLP steps defined before have 100% accuracy, thus affecting the last step of translating text to FOL. That problem, though, affects all NLP-based approaches, so it should not be the only reason for discarding this approach.

The real problem arises with the ambiguity, duplicity and omissions of Natural Language.

Barack Obama has stated that...
The White House has stated that...
The U.S. Government has stated that...

These three sentences should have the same FOL representation except for the subject (or the first element of the relationship), but the three of them mean almost the same, at least in most cases. The FOL approach, alone, has no way of knowing that the three should be interchangeable, at least without a predefined knowledge base that made it possible to imply (State, Barack Obama, sth) $\rightarrow$ (State, White House, sth). Defining such a knowledge base is not an option, since its creation would entail to explicitly state all truths of human knowledge in a non-ambiguous and consistent representation of FOL, a feat virtually impossible.

More so, those relationships cannot be absolute. If the first sentence said "Barack Obama has stated that the U.S. Government is not him", by that replacement, in FOL it would be Obama != Obama, a contradictory statement even in Natural Language. Without the replacement, however, it makes perfect sense. Being so, a simple FOL knowledge base would not be enough to extract conclusions from news articles, since Natural Language could get into contradictions with itself following a set of supposedly absolute truths. For this to work, some sort of common sense is needed to extract the true meaning of every sentence.

Another problem comes with the ambiguity of Natural Language. Firstly, most words have multiple meanings, and that would mean that in order to translate words into FOL statements, some sort of word-sense disambiguation is needed, a non-trivial step. Secondly, when dealing with syntactic ambiguity, meaning, when a sentence can have two meanings because of the structure of the sentence, that word-FOL translation can lead to wrong results too. An example for this would be "He ate the cookies on the couch"; it

could mean that he was seated on the couch and ate the cookies or that the cookies were on the couch and he ate them.

Finally, about Natural Language omissions:

The White House has promised a better public healthcare, even with all the criticism.

To know who is criticising it is not that simple: it can be the opposition, the public opinion, or both. Given the right context, it could be known who is the critic, or maybe whether it is even important to begin with.

The main problem with the FOL approach is that the reader needs to fill the blanks in a text with its world knowledge, giving it a proper meaning, solving ambiguities and omissions, and assigning the importance of selected parts of the text, all done with processes not driven by pure logic, but by a mix of inference on a prior knowledge, experience and intuition.

These problems should not make this approach unfeasible, but its results may be poor and its complexity too high for a project like this one. That is why the selected approach will be another, more practical and specific than this one.

2.2 5W1H

This second approach tries to take advantage of the structure of a news item. 5W1H, or the Six Ws, is a technique employed in journalism to gather all information about a story, to turn it into a news article. It consists of six questions: *What*, *Who*, *Where*, *When*, *Why* and *How*. An article is not considered complete until all of these six questions are answered.

Since the problem is to connect pairs of news articles that deal with the same subject, it makes sense to take advantage of the fact that any entry should answer those questions, or that those answers should represent the story completely. One way to establish a distance between entries would be to extract each of these answers and compare them with a defined distance function. The combination of the six distances with appropriate weights (trained with a suitable optimisation algorithm) would make for a global distance between any two entries.

There is some previous work about this idea. [3] tries to extract the 5W of every sentence of a text, with the help of a tagged dataset and a classifier that takes as input all types of features defined in the previous section (POS tagging, chunks...) and morphological features of the words of the sentence too. However, [3] shows two problems: even the human taggers did not agree on what to consider *What* or *Why*, compared to *Who*, *Where* and *When*, and the average F-score of the system only reached a 62%.

Nevertheless, this is a fine idea subject to some considerations. It seems that little agreement could be reached on what to consider part of *What* or *Why*, or even how to represent *How*, but *Where*, *Who* and *When* are more concrete and could give better results. In fact, in a news story, *Who* does the action, *Where* it has been done and *When* has been done could make all the difference. The following sections focus on this idea.

2.3 Named Entities *Who-Where*, with LDA *What*

The 5W1H approach looks promising and has been the centre of the algorithm, with some considerations:

1. Extracting the 5Ws in every sentence is more complicated and could have less value in assessing if two articles talk about the same. For that reason, the algorithm tries to extract the 5Ws of the entry as a whole.

2. Since not all Ws are quite objective, and not all have the same value for the purpose of the project, the algorithm only takes into account the *What*, *Who*, *Where* and *When* of the news item, considering the *What* as the topic of the entry, in a specific way that will be discussed later. *When* will not be computed, since it will be considered the date of publication of the entry, and only entries with similar dates will be compared.

3. *Where* and *Who*, for simplification, will be represented as Named Entities (Proper Nouns) that appear in the text. Also, those Named Entities (NE) will be weighted according to the importance of the sentence they appear in, based on the following.

   Inspired by [4], the algorithm creates a score of every sentence in an article by creating a score of each of its words, which is the TF-IDF score of each word within the text. The IDF part of the score is retrieved from the spaCy vocabulary corpus. Then, the score of a sentence is the mean of the scores of the words it contains. That way, a given Named Entity in the text will be assigned the sum of all the sentence scores of the sentences it appears in. Thus, for *Who* and *Where*, the 10 most scored Named Entities are selected and a vector with those pairs NE-score is formed.

4. *What* will be represented by the output vector of the LDA algorithm, that models the topic of a text by a vector of K non-negative real numbers that sum 1. That way, *What* becomes a representation of the topic of the entry.

5. Only politics entries will be considered. Since the algorithm uses Named Entities on *Who* and *Where*, it is clear that only entries with a relevant number of Named Entities would work well. More so, those Named Entities should be public figures famous enough for them to have a relevant Wikipedia page, which is used for the comparison between the *Who*/*Where* vectors.

   Selecting only politics entries narrows the scope of the algorithm, but this consideration was done to assess the approach in a scope where it would shine, to check if the method is on the right track or should be discarded.

A similar approach to this one has been tried before in [5]. There, the goal is to cluster documents of two different languages by its content. It has been made only using Named Entity recognition and matching those NEs between the two languages with the help of Levenshtein distance. The selected NEs are filtered too, using the type of NE, that can be retrieved also with NLP techniques. Examples of such types are PERSON, FACILITY or GPE (Geopolitical Entities). In [5], only PERSON, ORG (organization), LOC (location) and MISCELLANY are considered. It should be said that those types can be related to *Who* and *Where*, which justifies the choice made for the algorithm.

The Named Entities picked in the algorithm, however, are the following.

- *Who*: PERSON and ORG.
- *Where*: FAC (facility), GPE and LOC.

Finally, with those NE-vectors, a distance between them should be defined; that is done with the help of the Wikipedia API, that allows to extract the 10 most significant articles given any phrase (in this case, the considered NE), and a comparison between the 10 articles of any pair of NEs is done to establish a distance between the two.

All these points are explained in full detail in the following chapter, detailing all the implementation considerations that had to be made too.

Chapter 3

Implementation

In this chapter all the steps that form the system are discussed in depth. The first section describes the four libraries created for that purpose, and the following go through each of the steps taken on the project.

3.1 System description

This project has been organised in four Python libraries, detailed here:

- *newsparser*: defines classes Feed and Entry to extract entries from a RSS feed, saving all the necessary metadata and the article text in the central database.
- *newsfilter*: defines classes and methods to filter those entries that should not be considered in the system, such as entries with a broken link, that had no title, that had no meaningful content, etc.
- *newstagger*: defines a Flask HTTP server and its pages to allow an easy creation of a tagged dataset for the creation of the system.
- *newsbreaker*: defines functions and classes that inherit from the Entry class in *newsparser* and allow for an easy access to its content, its counters of words, its *What/Who/Where* vectors and some methods to compute the distance between two of them.

Figure 3.1 displays these four libraries and all their main external dependencies.

All these libraries work saving the results into a central MongoDB database that runs on local, for simplicity, except for some parts of *newstagger*, that due to limitations on the RSS entry retrieval part needed a more rudimentary approach. The *pymongo* library is used to access that MongoDB database. Its documentation can be found in [6].

In the following sections, each step in the process is detailed, and its code will be fit in one of those libraries, or on separate iPython Notebooks where some exploratory work has been done (stored in the Github repository along with the libraries code).

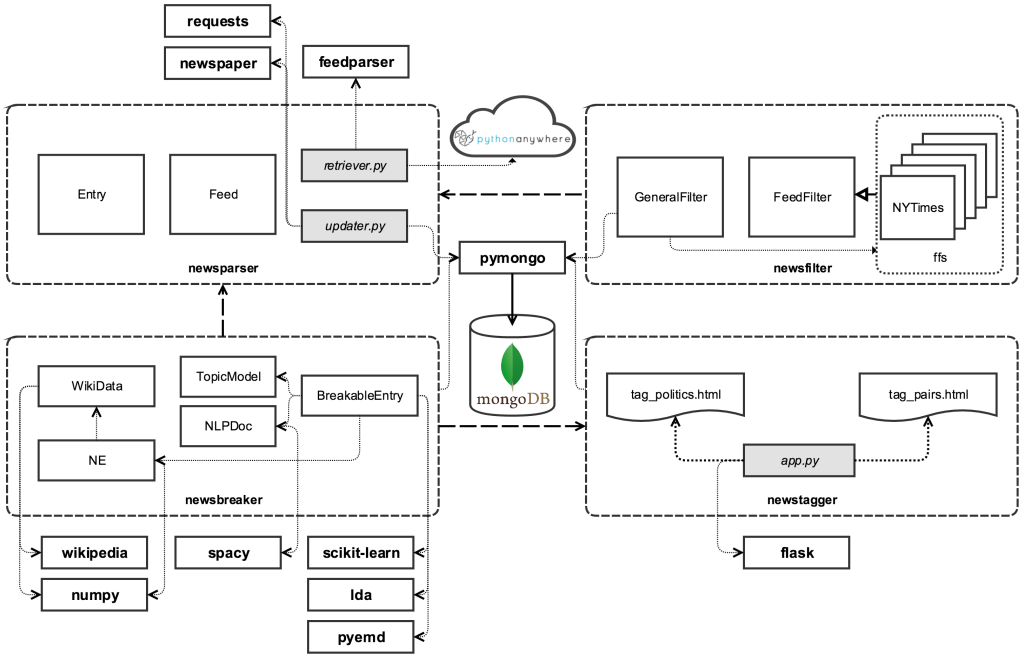

Figure 3.1: Structure of the system, with all libraries and external dependencies

3.2 Dataset creation

In order to conduct experiments with the proposed approach and assess its value, a dataset is needed. Such a dataset should contain entries from several feeds of similar nature. For this work, 10 of the most popular digital newspapers in the United States have been selected (based on [7]) and its entries were extracted from the RSS Feed they publish.

Those RSS Feeds only return metadata from that entry; to access the entry content, the code must access the link where the article is located, process the HTML code and extract the entry text, without any HTML code whenever possible.

Not all the entries fit the purpose of this work; then, they must be filtered before work is built on them.

3.2.1 RSS Feed Retrieval

It is a common practise nowadays for a popular blog or digital newspaper to have a RSS Feed of its entries, since it allows RSS Readers to use those feeds in their system, thus enhancing the experience of its users.

Those RSS Feeds can be read by a well designed code, but since most feeds do not follow all the RSS standards, it can be troublesome to process them manually. That is the purpose of *feedparser*; it is a Python library that loads a given RSS URL and processes its XML code to extract all feed metadata and its entries metadata, offering a simple API to

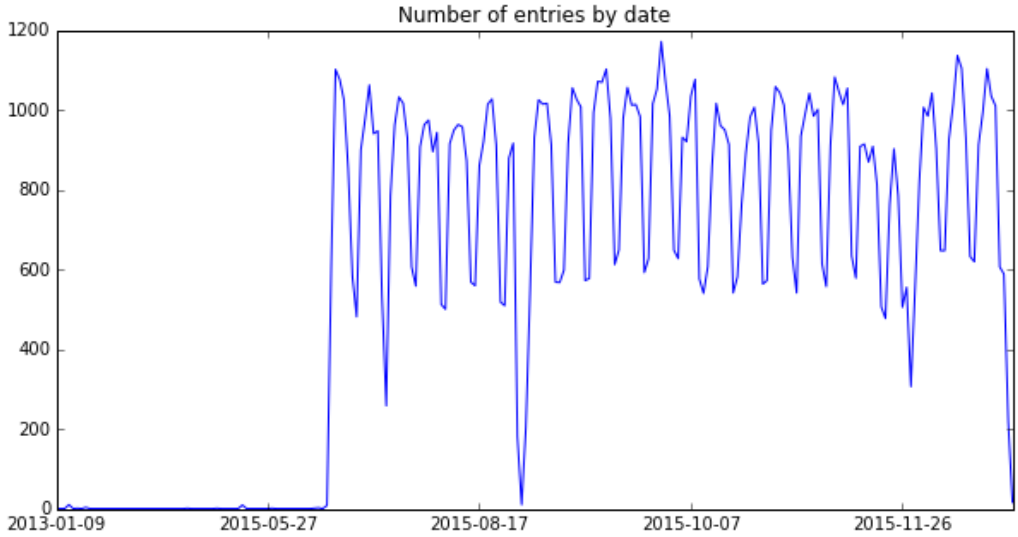

Figure 3.2: Number of entries by date

access those attributes. Its documentation can be found in [8].

The code in the *newsparser* library uses *feedparser* to access the news of the Feeds it keeps track on and stores all the entry and feed metadata in an Entry and Feed instance, respectively. When trying to download the RSS after some time (in this system, every hour), it checks which entries have not been downloaded yet and stores them, giving them an index to access it easily.

This code runs in a Python Server on the cloud, in a service called PythonAnywhere (https://www.pythonanywhere.com/). This server was chosen because it has a full Python environment already installed and it is quite easy to work with. Since the objective of this project was to deal with a NLP problem, and not with server configuration or setup, this server was chosen.

This server has no manual configuration. Thus, some utilities like MongoDB are not easily connected with it. For that reason, to save an entry metadata, pure JSON is used, files that are stored in specific folders depending on their feeds. That structure is done just to overpass the limitations of the server. When working with the local libraries, all other information is stored in the MongoDB database.

The server serves an endpoint that can be fetched to get all the entries metadata from a selected feed and a selected index interval. A copy of the *newsparser* library runs on a local desktop and calls this endpoint, in order to retrieve the entries from the RSS Feed and then, work with them on local. The purpose of this setup was to make the RSS Feed parsing an automatic task, running it hourly, each day.

Figure 3.2 shows the volumes of entries retrieved each day, from 2015-07-13 to 2015-12-22, the last day considered for the implementation.

Some previous days appear in the visualisation; those are entries that are edited days (or even years or months) after its publication and the RSS Feed returns them again. It can

also be observed that there has been a major drop-off near 2015-08-17, where there was a bug with the feed retrieval algorithm that wasn't solved until a couple of days later, thus making it hard to retrieve the lost entries.

Finally, it is worth mentioning that for the purpose of storing intermediate variables needed for the project along with the basic Entry metadata, Entry has a special variable named *data*, which is a dictionary that should be JSON serializable in order to save it later along with the rest of the metadata.

3.2.2 Entry content retrieval

Each Entry instance has a "link" attribute that can be used to access the URL where the entry content should be located. Those links are for browsers, and therefore, their content is pure HTML code that needs to be processed in order to access the entry content in plain text. That is no trivial task; for that purpose, the system uses another Python library, *newspaper*, specifically designed to extract plain text from HTML code using XML parsing and NLP techniques. *newspaper* documentation can be found in [9].

Some of the entries link to a faulty URL and thus the algorithm cannot retrieve anything. Those entries are marked as not downloaded in the entry metadata. There are some errors too, where some of those faulty links return HTML code in the *newspaper* library. Also, all entry multimedia content is replaced by a specific phrase describing it, detailed in the *newspaper* code.

All those imperfections need to be filtered to avoid problems in further code. That is the next step in the algorithm.

3.3 News filtering

Not every news article should continue in the pipeline. As said before, some entries do not have content, or a publishing date, or useful content at all. Some entries like obituaries, weather forecasts or day summaries should be discarded too. That's why *newsfilter* is necessary. In this section, the library is detailed, first talking about the more general aspects of entry filtering and, later, discussing the two more complex aspects of entry duplicity and news agency articles.

newsfilter is structured as two classes. FeedFilter tries to take advantage of the specific characteristics of every feed to filter some of their useless entries. GeneralFilter takes all entries in the system, calls the corresponding FeedFilter on each entry depending on its feed and finally applies the last two steps in filtering, much more complex than the other ones: duplicate_filter and news_agency_filter.

After all this process, every entry processed will be assigned a key-value pair in an attribute inside the *data* variable named "newsbreaker"; it ought to be a boolean telling if the entry passes all filters or not.

3.3.1 FeedFilter

The important concept about FeedFilter is that it is a class with two class attributes, precontent_filters and content_filters, which are lists of bound methods that should be applied to every entry in order. Each of those functions is a filter to apply to each entry, and the main difference between precontent and content is that for any content filter to run, it is needed that its content has been loaded and stored in the "content" variable in the Entry instance.

Each of those filter functions return a boolean with whether the entry is filtered or not in that step. If an entry is discarded somewhere, the rest of the filters do not run. The results of those filters are stored with the Entry metadata in the *data* variable, in a subdictionary: "data.filter.step_flags". Each of those filters are discussed next.

First, FeedFilter has to check that the entry content could be downloaded and that the publishing date makes sense. After that, FeedFilter tries to discard some of the entries based on its title and its tags (because some entries that should be filtered had titles or tags with certain patterns).

For example, NYTimes has some periodic entries titled "Your Monday Briefing", LATimes some like "For the record" or BostonHerald has entries with tags like "Obituaries", "Horoscop" or "JobFind".

Since every Feed has its own rules at this point, title_filter and tags_filter do not hardcode those filters, but access a class variable of FeedFilter. That variable stores patterns that should match the title or tag according to specific rules: maybe a perfect match, maybe a lowered match (all the letters would be lowercased) or maybe a match with a regular expression. Even, if needed, each of those elements can be passed a function that returns if the entry should be discarded or not.

Those structures are filled by inheritance; every Feed in the system has its own FeedFilter-inherited class in the subpackage *ffs* and when GeneralFilter calls the corresponding FeedFilter of an entry according to its Feed, it accesses that subclass, thus using only the patterns specified for that feed.

The last two filters are content filters (it is needed for the entry to have its content loaded and stored in the *content* attribute). The first one, politics_filter, is not a filter by itself, since all entries that enter the function will always pass the filter. Its purpose is to fill a variable in the *data* dictionary that tells us if that entry is about politics or not. How this can be done is explained in the following section.

Finally, the last content filter is paragraphs_filter, that filters some of the paragraphs of an entry if they match some patterns. This filter is used to delete the multimedia replacement made by the *newspaper* library to clean the entry content. The new content is saved into the MongoDB database under the key "filtered_content", even when nothing has been filtered (to allow a consistent content retrieval). At last, paragraphs_filters discards those entries that after paragraph filtering have less than two paragraphs, since such an entry is assumed to be a paid entry that had its content cut off, and having less than two paragraphs would not give any meaningful information about the entry.

When all of the filters are run or one of them discards the entry, the result of this process is saved in data['filter'] with a key named "discarded". This is the end of the process for FeedFilter.

3.3.2 Politics filtering

As said before, this project tries to solve the proposed problem only with politics entries; thus, a politics filtering was needed.

For that purpose, since the Entry metadata did not offer any information about the topic of the entry (at least, for the majority of the feeds) a classifier had to be made in order to establish if an entry talked about politics or not.

To achieve this it was necessary a tagged dataset that went through some entries and tagged whether it was a political entry or not. How to create such a dataset will be discussed later. For now, it is assumed that there is a dataset with 1000 tagged entries in MongoDB.

Given that dataset, the next step is to pick a classifier and define what will be the entry features. Those features usually are counters of words in a text, being them of the following types:

- **Bag Of Words (BOW)**: a dictionary where keys are words in the text and their values are booleans telling if the word is in the text or not.
- **Counter**: a dictionary where words are keys and their values, the number of times they appear in the text.
- **TF-IDF Counter**: words are keys and their values are the TF-IDF score of those words.

Those three alternatives are all tried with appropriate classifiers to assess which one yields a better score.

Those features are created by using word segmentation and lowercasing each of those words. Word segmentation is quite easily done with the use of the *spacy* library.

All the classifiers and the scoring algorithm come from the famous *scikit-learn* Python library, which offers a consistent and easy-to-use API for lots of Machine Learning algorithms. *scikit-learn* comes with an extensive documentation explaining all its algorithms in detail; it can be found in [10]. It should also be said that many of the Machine Learning concepts used throughout this work are covered in [11].

In particular, for BOW a BernoulliNB classifier was used (a Naive Bayes classifier that expects boolean features) and for the other two, a MultinomialNB classifier (a Naive Bayes classifier that expects counts of items as features). Some Random Forest classifiers with different max_depth and n_estimators parameters were also used.

By assessing their scores using K-Fold with multiple values for K, to avoid overfitting, it was observed that all those classifiers gave almost the same scores, around 0.92. Counter + MultinomialNB and Counter + RandomForests(n_estimators=40, max_depth=20) gave the best accuracies. Finally, to decide on a classifier, their performance was checked: MultinomialNB was almost three times faster. For that reason, it was the selected classifier.

All this analysis and its results can be read from the corresponding iPython Notebook in the project Github page.

The politics model is then pickled for further use. Returning to the entries filtering, the politics_filter uses that model to establish if a given entry is about politics or not and stores that result in the data variable. As said before, non-politics entries were not filtered,

since it was desired to keep all entries for further research after the politics trial. However, checking if an entry is about politics becomes as easy as executing "entry.data.get('politics', False)".

After all FeedFilter-related filters are executed, those entries that were not discarded must pass two more filters: duplicate_filter and news_agency_filter.

3.3.3 Duplicate filter

Some feeds correct their entries or update them with more information. When doing so, some of them update the entry without altering the link or the entryid, but some do, and that produces a duplicity in the entries the system downloads, thus producing some duplicate articles that ought to be discarded.

For doing so the idea is quite simple: produce a Counter of words (this time without lowercasing them, since it is desired to find exact matches between two entries) and divide that counter by the sum of all values. That way, the counter becomes a histogram of the distribution of words in that entry. The "histogram similarity" is defined by the sum of the minimum value of each word for all words in both entries. That similarity is a real number between 0 and 1 and is higher when both entries are more similar.

An example of this process is the following:

```
>>> from collections import Counter
>>> from spacy.en import English
>>> nlp = English()
>>> text = 'By "Natural Language" it is meant any language used for everyday
    communications, such as English, Spanish, Chinese, etc. Natural Language
    Processing is a discipline that tries to process any of those languages. '
>>> doc = nlp(text)
>>> sent1, sent2 = list(doc.sents)
>>> c1, c2 = [Counter(word.text for word in sent) for sent in [sent1, sent2]]
>>> for counter in [c1, c2]:
...     s = sum(counter.values())
...     for k in counter:
...         counter[k] /= s
>>> print(min(
...     1, sum(
...         min(c1[k], c2[k])
...         for k in set(c1).intersection(set(c2))
...     )
... ))
0.2
```

Note that *word.text* is not lowered, as said before, and that the sumatorium is not enough. Due to floating-point calculation imprecision, that sum can overpass 1. That is why the distance must use the minimum value between 1 and that sum.

To check what would be an appropriate value for a threshold, the histogram of pairs of entries compared with this method was plotted. In case there was a meaningful gap between non-related entries and almost equal entries that would be the desired threshold.

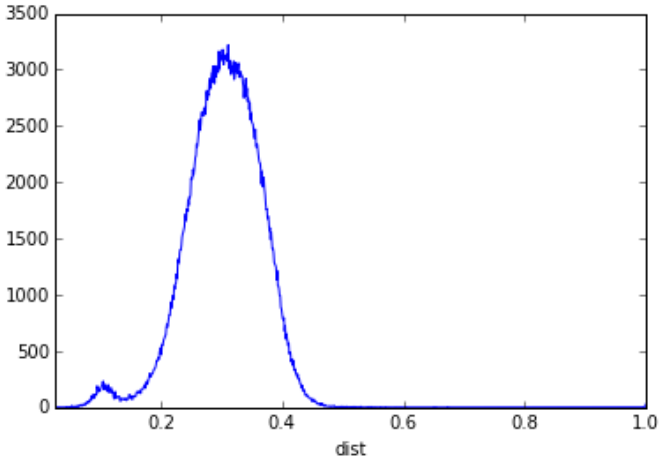

Figure 3.3: Duplicate-similarities between entries 0-1

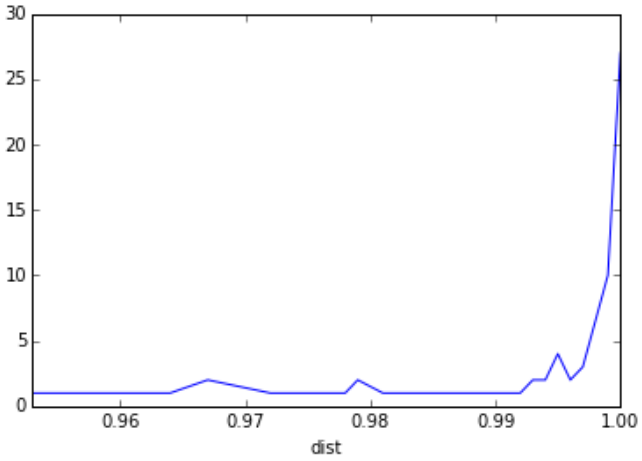

Figure 3.4: Duplicate-similarities between entries 0.99-1

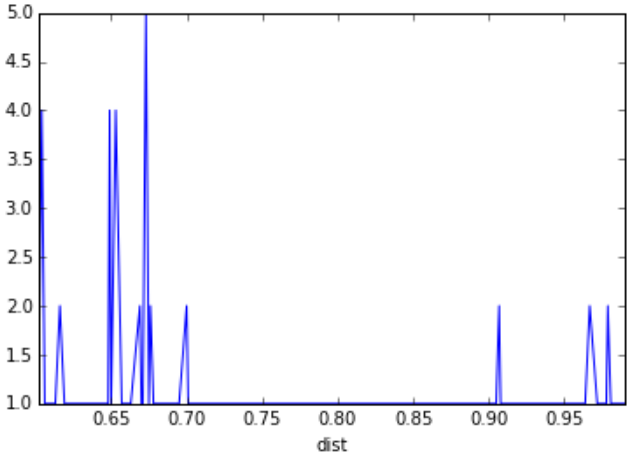

Figure 3.5: Duplicate-similarities between entries 0.6-0.99

Based on figure 3.3, it is obvious that this distribution tends to a normal and those pairs of entries near the bell should not be related. Checking on the entries near 1, in figure 3.4, it can be seen that those numbers increase again. Lastly, figure 3.5 checks on the values between 0.6 and 1, away from the bell, and excluding 1 since it increases quite heavily the scale of the plot. It can be seen that there is a meaningful gap between 0.7 and 0.9. That is why, without further inquiry, the threshold for duplicate entries is set at 0.75.

With that value, the objective is to discard all duplicate entries except one, since it is not wanted to exclude an entry when there is another similar to it. The algorithm for doing so goes as follows:

- For each day, for each feed, for each entry:
  - Create a set structure in its instance. It is called the "duplicate_set".
  - Check the histogram similarity between this entry and all the non-processed ones.
    - If that similarity exceeds the threshold, add that other entry to the base entry *duplicate_set* and set the *duplicate_set* of the other entry as the one from the base entry (the same reference).
- After all entries in a day-feed have been processed, take all the different duplicate sets, pick one entry from them to mark it as non-duplicate and mark as duplicate all the other ones.

After this process, duplicate entries have been filtered.

3.3.4 News Agency Filter

Some articles in a newspaper come from news agencies around the world, that write the news items and sell it to those newspapers. For that reason, different feeds have entries with almost identical content; in fact, those newspapers only change the first two paragraphs, leaving the rest identical.

Obviously, any algorithm designed to check for similarity between news would join those entries repeated across feeds, but it would not mean that such an algorithm worked well. To avoid that nuisance, it is better to filter those entries too, just as done previously with the duplicate entries in a feed.

To do that filtering process, the algorithm is exactly the same but with one exception: instead of working with each feed independently, obviously, it ought to consider all entries in a day no matter what feed they may come from.

With this process, all news agency entries detected are marked as "news_agency"" in the data dictionary and only one passes the filter; the others are discarded.

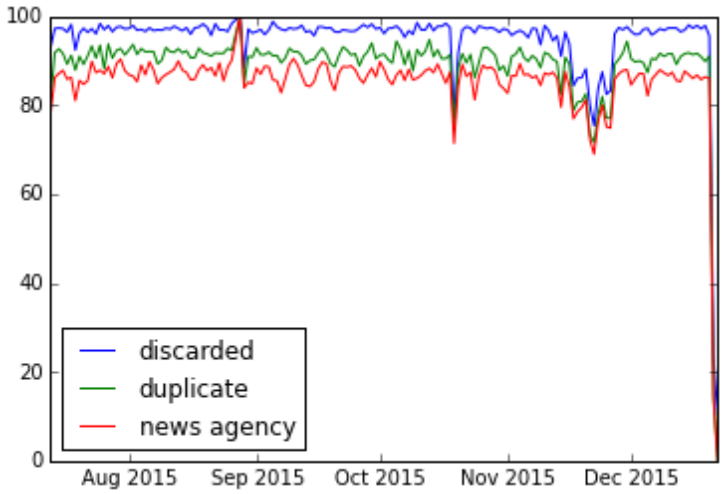

Figure 3.6: Filtering ratios

3.3.5 Filtering results

After all this filtering process, the ratio of filtered entries goes as follows. Each line in figure 3.6 represents the filtering ratio applied only until that filtering step. Discarded refers to all filters applied by FeedFilters and, as news_agency_filter is the last step, its line is the final filtering ratio. It can be seen easily that the filtering ratio is somewhat stable, around 85%.

3.4 News tagging

As said before, to classify entries as politics, a tagged dataset is needed. For an easy and secure tagging, a specific library was created; that is the part that *newstagger* plays in this project.

newstagger uses the *Flask* framework for building an HTTP server that is run on localhost, so it displays a web GUI for easy tagging and stores all results retrieved from that GUI in the MongoDB database. A *Flask* quickstart guide and its documentation can be found in [12].

Without much detail, the GUI is built with the help of JQuery and the server is built with the Flask framework and Jinja2 as the template engine. *newstagger* offers two different ways of tagging.

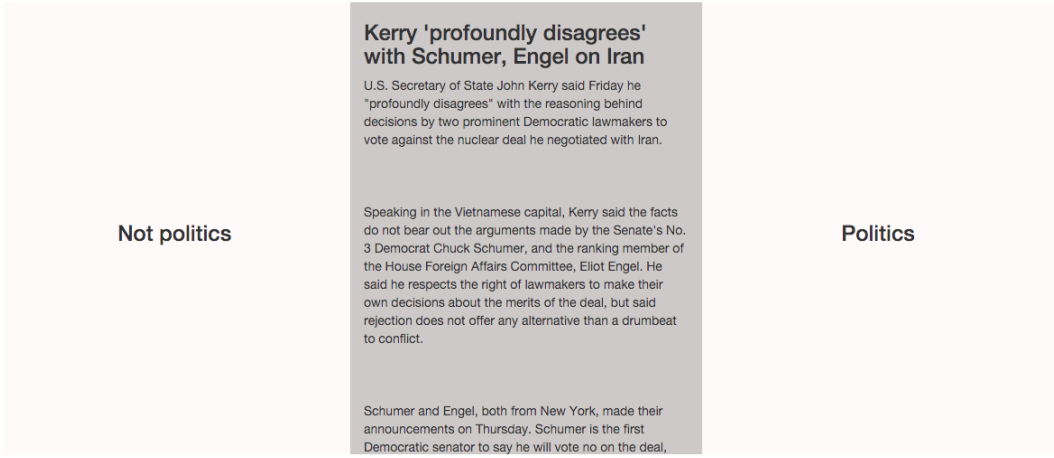

Figure 3.7: Politics tagging GUI

3.4.1 Politics tagging

One of the two needed tagged datasets consists of entries that can or cannot be tagged as politics. The GUI displays an entry and lets the user choose politics or no-politics by tapping right or left, respectively. Figure 3.7 shows this GUI.

3.4.2 Pairs tagging

The other needed dataset consists of a list of tuples of three entries, named for this example as A, B and C. The objective is to tell the system whether A and B are more similar than A and C or the contrary. That is used later for building the global distance between two entries and assessing its accuracy.

For that purpose, the GUI displays the base entry at the centre and the alternatives on both sides of it. When tapping left or right, the selected entry is highlighted (and thus, the user is saying that the base entry and the selected one are more similar than the alternative). When pressing up or down, the entries on both sides are changed but the base entry remains the same. When the user finishes with that base entry and presses Enter, the results are sent to the server for it to store them in MongoDB.

If a test is not completed (no left or right entry is selected) that will not be added to the system. That is so because when trying to fill this dataset, there were many occasions when none of the two alternatives had anything to do with the base entry.

In fact, that has been a huge problem when tagging. For that reason, since lots of times entries had nothing to do with one another, the proposed tests are selected according to the topic-distance between them (retrieved with the LDA algorithm). Lower distance meant appearing first in the tests, and only the N most closer tests were selected. That way, the number of untaggable tests dropped a little.

Even so, many times the base entry was a news article that did not relate to any other news entry of that day. To avoid this, before starting tagging with a random base entry,

Figure 3.8: Pairs tagging GUI

the user can select which entry to use as a base, so that those entries where it is clear that not many entries will relate can be avoided to appear as bases.

Even with those two adjustments, tagging for this dataset was really difficult and tiresome. For almost 6 hours of work, not even 150 tests were completed, since it is rare for a test to be taggable because the base has nothing to do with the alternatives in most cases. More so, as the tagging progressed, the types of news that became tagged started talking about the same issues over and over; that is due to the fact that the selected feeds were focused on the U.S. for a relatively short period of time -5 months- and almost all stories talked about the same issues: the U.S.-Iran Deal, the Clinton emails, the GOP debates and specially the Donald Trump controversies. Apart from that, only three more issues outside the U.S. were highlighted: the Greece-UE Deal, the Migratory Crisis and the Paris Terrorist Attacks.

For that reason, the tagging was stopped at almost 150 tests. It was clear at that point that the feeds should have been more varied and it should have started extracting news earlier (thought it started extracting 3 months before the actual date where the project should have begun). It is clear that by doing those tricks when tagging, the dataset can get easily biased, but because of the limited amount of time and work that could be spent on this part of the project, this could not be avoided.

For that reason, it must be considered that the results of this project are not meant to be taken for granted, but as a guide for future work.

3.5 Newsbreaker functions and model

With the processed dataset, it is time to work with the data available. For simplicity, the *newsbreaker* library includes some wrappers on *newsparser* that allow it to access the entry content automatically, just by calling the content attribute, without loading on creation, to avoid unnecessary callings. It loads the NLP model (used for any NLP task) just when needed, to avoid an excessive delay when importing the package. And it loads the NLP-processed content of an entry, the named Doc (terminology from the *spacy* library) in a simple way, saving execution time and lots of boilerplate code.

All these points will be discussed in detail in the following sections.

3.5.1 BreakableEntry

BreakableEntry is a class defined in *newsbreaker* that inherits from newsparser.Entry and has some Python hacks to allow an easy use of the content and the spacy.Doc of that content.

One major problem when working with *newsparser* was that the attribute content was None as long as its method load_content had not run. This was a major inconvenience since it caused a lot boilerplate code when writing some algorithms over the entry content, but at the same time it was needed for newsparser since its main focus was to offer functions for an easy RSS Feed tracking. Furthermore, *newsbreaker* does not want to access the original content of an entry, but the processed content produced by the paragraphs_filter in *newsfilter*.

Since the *newsbreaker* package tries to work on the entry content, avoiding that nuisance becomes a major convenience. That is why "content" becomes a Python property of the class; meaning, accessing the variable "content" calls the bound method "content", which checks if the variable "_content" is defined and, if not, tries to recover the content of the entry (the processed content, not the original one) from the database, finally saving it in the Entry instance and returning it.

With that minor tweak, loading the content becomes as easy as typing "entry.content".

3.5.2 LazyInit

When working with the *spacy* library for NLP methods, the first step is to load the English model by executing:

```
>>> from spacy.en import English
>>> nlp = English()
```

The creation of that nlp variable is quite expensive; it could take as much as 30 seconds on a laptop. Since this nlp object is the base to performing any NLP tasks in the code, it was decided to set it as a global variable for the package. In fact, it can be accessed simply like this:

```
>>> from newsbreaker import nlp
```

30 seconds of loading on the first time the library is imported would not seem much, but when working on the library, adding functions or solving bugs, every time it is needed to restart and reimport the code, it loads the model and, therefore, takes a long time. More so, when installing the package in a Python distribution, the setup script would load the model too.

That was a big problem, and the fact that this variable had to be global made it difficult to delay its initialization until needed, because that would mean a lot of boilerplate code that checked that the variable was already initialized before its use. This could be solved in many ways, but it was decided to use a Python tweak to avoid it.

lazyinit is a function on *newsbreaker.utils.lazyinit* that can act too as a class decorator. Its purpose is to delay the creation of a variable until the moment itself or any of its attributes or methods are required. At that moment, it creates itself and acts like it was created beforehand. That way, nlp need not be created on import, but only when needed or initialized explicitly (calling *__doinit__*).

How *lazyinit* works is somehow simple. Using the Python magic methods, whenever an attribute (that includes user-defined functions) is called, that call is interrupted by the *__doinit__* method that actually loads the instance and then returns the expected attribute. When calling one of the Python magic methods (like *__add__*, *__lt__*, *__repr__* and so on), it does exactly the same. From the point *__doinit__* is called, the instance becomes an instance of the original class and all its methods and attributes return to normal.

How to use *lazyinit*? When defining a class, that class can be decorated with lazyinit directly, like this:

```
>>> @lazyinit
>>> class A:
>>>     pass
```

Or with a predefined class, like *spacy.en.English*, it is as simple as:

```
>>> English = lazyinit(English)
```

Then, when executing "nlp = English()" nothing actually happens, not until nlp is really used for something.

The following snippet is an example of usage of *lazyinit* to avoid the initialization of some heavy objects until the moment the object is needed.

```
>>> from newsbreaker.utils.lazyinit import lazyinit
>>> import time
>>> from datetime import datetime
>>> # define class A with a heavy initialization
>>> class A:
...     def __init__(self, x):
...         self.x = x
...         print(datetime.now())
...         time.sleep(1)
...         print(datetime.now())
...
...     def __str__(self):
...         return str(self.x)
>>> a = A(1)
2016-01-16 12:37:31.875630
2016-01-16 12:37:32.880804
>>> print(a)
1
>>> # a has taken a second to initialize
>>> A = lazyinit(A) # decorate class
>>> a = A() # nothing is printed; __init__ has not run yet
>>> print(a) # see how __init__ will run now that the object is used
2016-01-16 12:40:24.051691
2016-01-16 12:40:25.051935
2
```

3.5.3 NLPDoc

spacy uses the forementioned nlp variable to process text, and therefore, to run NLP tasks. The usual workflow goes as follows: given the entry.content, the code calls *nlp* with the entry content and that returns a *spacy.Doc* instance. That is an object with all the information and methods needed to process the text. The problem is that when calling nlp with a text, it takes quite some time to process it, because it does the complete analysis on the text, when it is not always needed. That method offers three keyword arguments: *tag*, *parse* and *entity*. *tag* applies the POS tagger, *parse*, the syntactic dependency parser (that will not be needed in this project) and *entity*, the Named Entity recogniser.

In previous steps, when it was only wanted to apply word and sentence segmentation, none of those keywords were applied, since it was not necessary to tag the POS of every word, nor detect NEs. However, in the following sections, the Named Entity recogniser will be needed, and for that to work, the POS tagger must run too. For that reason, two different Docs should be created, depending on the use case. Obviously, the second Doc, more complete, would be enough for the first use case but it would require much more time.

NLPDoc is a class that allows for an easy use of the *spacy.Doc*. Every BreakableEntry is assigned an instance of this NLPDoc as a variable named "doc", passing it the entry on the constructor. Then, when the following is executed:

```
>>> entry.doc(tag=True, parse=False, entity=True)
```

what it is doing is that the *entry.doc* instance is parsing the *entry.content* with the options mentioned and storing that doc instance inside the object. From that moment on, whenever any of the variables of NLPDoc is accessed, what it is returning are the variables of the loaded doc. If the previous statement is executed again, with another choice of parameters, it loads another doc whenever necessary and stores it too. If a previous option of variables is enough for the current order, NLPDoc won't load another *spacy.Doc*, but use that other doc that fulfilled the conditions.

So, when using the Doc of a BreakableEntry, the workflow would be:

- Always call entry.doc on start with the needed parameters. It will only create a new Doc when necessary.
- Access the Doc attributes directly from entry.doc.

This has been a major improvement in the library, since it deletes some boilerplate code in the next algorithms.

3.6 Named Entity Processing

The proposed approach tries to define an entry by its *What*, *Who* and *Where* vectors. *Who* and *Where* will be defined by Named Entities in the text and their importance score, a value that will be defined later. However, to compute the distance between two *Who* or *Where* vectors of different entries, there should be, at least, a distance between Named Entities.

The objective of this section is to define a notion of distance between vectors of Named Entities, vectors defined as dictionaries with keys as Named Entities and values as their importance score. For that purpose, the algorithm uses the Wikipedia API and works with Wikipedia article contents.

Before that, it is worth reminding that to extract the Named Entities in a text, and their categories, the spaCy library offers a simple API. The attribute "ents" of a processed Doc is a list of Named Entity objects that offers its text, its category and its position within the list of words of the text. This object is essential in the following algorithms.

3.6.1 WikiData

WikiData is a class in *newsbreaker* that extends the Wikipedia API Python wrapping, a library named *wikipedia*, providing it with some convenience functions. *wikipedia* documentation can be found in [13].

In the first place, any *wikipedia* method call that raises a HTTP-related Exception will be decorated to retry in case of such Exceptions at most three times, in order to avoid the unnecessary boilerplate code that tries to prevent such errors.

WikiData offers four methods:

- *article*: given an Article title or an Article id, calls the wikipedia page endpoint (if it has not been done before) to retrieve the Article content and save it in the MongoDB database.

- *ne*: given a Named Entity as parameter, it uses the Wikipedia search endpoint when necessary to look for the (at most 10) top articles similar to the given Named Entity. Those articles are then retrieved (if necessary) with the *article* method, and the relationship of that NE with those articles is stored in the database too for the following steps.

  An example of this method would be the following:

```
>>> wiki = WikiData()
>>> wiki.ne('Barack Obama') # returns a list of article ids related to Obama,
      in order of importance
{'_id': 'barack obama',
 'articles': [534366,
  20329140,
  20767983,
  25692130,
  15767746,
  37307408,
  41828619,
  10027799]}
```

- *article_vector*: given an Article id, retrieves the content of that article from the database and creates a Counter of the Named Entities it contains, filtering to the NE categories PERSON, ORG, FAC, GPE and LOC (the ones used for *Who* and *Where*). That counter is normalised so that its values sum 1.

```
>>> sorted(wiki.article_vector(534366).items(), key=lambda pair: pair[1],
    reverse=True)[:10]
[('jimmy carter', 0.05970149253731343),
 ('ford', 0.05970149253731343),
 ('carter', 0.05970149253731343),
 ('ronald reagan', 0.04477611940298507),
 ('the united states', 0.04477611940298507),
 ('herbert hoover', 0.04477611940298507),
 ('theodore roosevelt', 0.04477611940298507),
 ('united states', 0.04477611940298507),
 ('george h. w. bush', 0.04477611940298507),
```

```
('reagan', 0.04477611940298507)]
```

- *article_similarity*: given two Article ids, computes the histogram distance between the two articles, using the Counter retrieved by *article_vector*.

The last three methods use the *lru_cache* decorator from the standard Python library *functools* that creates a cache for the method it decorates to avoid unnecessary execution. That is done to avoid multiple readings to the database, thus creating a significant overhead in I/O tasks.

This class is essential in the following section, where it is explained how to use those NE articles.

3.6.2 NE

NE is another *newsbreaker* class that stores a Named Entity and has a *similarity* method that returns a [0, 1] real number with the similarity of two Named Entities. How this similarity is computed is quite complex.

Considering the top articles of each NE in order of importance, all the pairs of articles, picking one from each NE, are considered. An integer, named "level", is assigned to the pair. The level of such a pair is the addition of the position in the sorted list of articles based on importance, being 0 the most important.

For example, each of the following articles has a position assigned by *enumerate*. *level* would be the addition of those two positions, the third number in each line, and that *level* is assigned to the combination of *article1* and *article2*.

```
>>> for i, article1 in enumerate(wiki.ne('Barack Obama')['articles']):
...     for j, article2 in enumerate(wiki.ne('George Bush')['articles']):
...         print([i, j, i+j, article1, article2])
[0, 0, 0, 534366, 706778]
[0, 1, 1, 534366, 1864257]
[0, 2, 2, 534366, 3414021]
[0, 3, 3, 534366, 21203789]
[0, 4, 4, 534366, 149577]
[0, 5, 5, 534366, 11955]
[1, 0, 1, 20329140, 706778]
[1, 1, 2, 20329140, 1864257]
[1, 2, 3, 20329140, 3414021]
[1, 3, 4, 20329140, 21203789]
[1, 4, 5, 20329140, 149577]
[1, 5, 6, 20329140, 11955]
...
```

All the similarities between those pairs of articles are computed in order of level. Each of those similarities is normalised; the values needed for that normalisation were recovered by computing almost a million similarities between pairs of Wikipedia Articles and computing their mean and standard deviation: 0.02314 and 0.0316 respectively.

	Barack Obama	Courtney Cox	George Bush	Jennifer Aniston	Kremlin	Michael Jordan	Oprah Winfrey	Susan Boyle	Tiger Woods	Vladimir Putin
Barack Obama	0.000000	0.469090	0.409683	0.567886	1.000000	0.576376	0.387271	0.466265	0.472915	0.367056
Courtney Cox	0.469090	0.000000	0.579669	0.437244	1.000000	0.718366	0.687073	0.491046	0.453967	1.000000
George Bush	0.409683	0.579669	0.000000	0.542471	1.000000	0.719144	0.587893	0.571215	0.552032	0.534565
Jennifer Aniston	0.567886	0.437244	0.542471	0.000000	1.000000	0.805817	0.449633	0.877403	0.482891	0.666461
Kremlin	1.000000	1.000000	1.000000	1.000000	0.000000	1.000000	1.000000	1.000000	1.000000	0.146276
Michael Jordan	0.576376	0.718366	0.719144	0.805817	1.000000	0.000000	0.458775	0.504332	0.717637	0.729235
Oprah Winfrey	0.387271	0.687073	0.587893	0.449633	1.000000	0.458775	0.000000	0.473584	0.478402	0.603655
Susan Boyle	0.466265	0.491046	0.571215	0.877403	1.000000	0.504332	0.473584	0.000000	0.496056	0.593294
Tiger Woods	0.472915	0.453967	0.552032	0.482891	1.000000	0.717637	0.478402	0.496056	0.000000	0.856105
Vladimir Putin	0.367056	1.000000	0.534565	0.666461	0.146276	0.729235	0.603655	0.593294	0.856105	0.000000

Figure 3.9: NEs matrix example

If the maximum value in a level is over a defined threshold, that pair is considered related. When a level has a related pair, the maximum similarity in that level is selected and multiplied by 0.9 to the power of "level". That factor is applied to reflect how unimportant were the two similar articles to their respective Named Entities. That way, NEs related by their fifth respective articles (that would mean that no more related articles matched) would get a similarity not higher than 0.43.

That threshold mentioned in the last paragraph was defined by using the similarities of several Wikipedia Articles again. The threshold is set so that half of the pairs of Articles are similar. That way, better results were achieved with the algorithm.

Finally, the last important method of class NE is *matrix*, a static method (or a class method in other languages) that gets a list of Named Entities and returns a numpy array of arrays (in fact, a matrix) of the **distances** between those NEs. It is important to notice that those are not similarities; distance is considered 1 minus similarity, since similarities are contained in [0, 1]. That matrix is returned along with the list of those Named Entities, to know which column relates to which NE.

That matrix is the base of the *Who/Where* distance algorithms, explained later.

An example of such a Matrix is shown in figure 3.9. Given more time and a tagged dataset a better distance could be defined for these values. Even so, in most cases it gives appropriate orders. For example: Vladimir Putin with Kremlin, or George Bush with Barack Obama.

As before, to avoid a significant overhead, the method *similarity* is cached.

3.7 Entry Ws

The next step in the algorithm is to create the vectors *What*, *Who* and *Where*. How those are created and how its distance is defined will be detailed in the following sections.

3.7.1 *What*

What refers to the topic of the entry. Since all entries are about politics, that topic must be quite specific to compare with the others.

The approach used is based on LDA. LDA, as said many times now, is a well established topic modeling algorithm. This project uses the *lda* Python library, a wrapper to a C-implementation of LDA. Its documentation can be found in [14]. Considering N common words for the texts to process, it takes a matrix of counts of those words in the set of documents to process, and takes as parameter the number K of topics to define. Then, the model tries to define a vector space where unobserved groups are defined based on why some parts of the data are similar. In other words, it transforms a counter-vector of those N words to a vector of K reals that sums to 1, where each of its coeficients is the proximity to a certain topic for that entry.

In that case, the *What* vector of an entry is the LDA transformed vector, given an LDA model. That model was trained with the whole set of entries considered, and serialized with the help of the *scikit-learn* and the *pickle* libraries.

The *What* distance is defined as the histogram distance, since the LDA transformed vectors all sum to 1.

3.7.2 *Who/Where*

Who and *Where* get the same treatment. First, all Named Entities in the text of the selected categories are detected, its sentence identified and the sentence score assigned to each NE appearance. The NE score becomes the result of the sum of each of the appearance scores. Finally, taking all the NE scores in a vector, it is normalised so that all scores sum to 1. The *Who/Where* vector is defined as the top 10 NEs with their respective scores.

The score of a sentence is the mean of the scores of the words it contains, and the score of a word is the TF-IDF score of that word in the text. The IDF part of that score is retrieved from the *spacy* vocabulary model.

An example for such a vector is shown in figure 3.10.

The top 10 NEs for *Who* and *Where* are the following:

```
>>> sorted(entry.who.items(), key=lambda pair: pair[1], reverse=True)[:10]
[('senate', 0.09259259259259266),
 ('congress', 0.07407407407407413),
 ('ground zero', 0.07407407407407413),
 ('the world trade center health program', 0.055555555555555594),
 ('house', 0.055555555555555594),
 ('maloney', 0.03703703703703706),
 ('feal', 0.03703703703703706),
 ('the daily show', 0.01851851851851853),
 ('bob menendez', 0.01851851851851853),
```

9/11 bill to aid Ground Zero workers rapidly gaining support in Congress
USAToday | 6268

Recovery and cleanup work continues at New York's World Trade Center in this Feb. 13, 2002, file photo. (Photo: David Karp, AP)

WASHINGTON - Eleven people who worked in rescue and recovery efforts at Ground Zero after the Sept. 11, 2001, terrorist attacks have died in the six weeks since the most recent anniversary of the 2001 terrorist attacks.

Another Ground Zero worker, Roy McLaughlin, died Sept. 10, one day before the anniversary. McLaughlin, 38, was a Yonkers police officer when the World Trade Center was attacked. Later promoted to a lieutenant, the married father of four young children was diagnosed with brain cancer five years ago.

The 12 recent deaths add a sense of urgency to efforts to renew legislation that provided medical care to rescue workers and other first responders who became ill as a result of their work at Ground Zero, advocates say. That legislation, the James Zadroga 9/11 Health and Compensation Act, expired Sept. 30.

All told, more than 1,700 people have died from 9/11-related illnesses, according to John Feal, a former demolition supervisor from eastern Long Island who has lobbied lawmakers to reauthorize the Ground Zero health legislation.

One of the law's key components, the World Trade Center Health Program, has enough money to continue operating until March or April. Without action by Congress, the program will begin notifying patients in January that they will lose services, Feal said.

Feal, who lost part of his foot during cleanup at Ground Zero, has led ailing firefighters, police officers and construction workers in making personal appeals to members of Congress.

His group was joined recently by members of Iraq and Afghanistan Veterans of America who say the Ground Zero first responders deserve lifelong medical care for their injuries in the same way military personnel deserve care for war-related injuries.

Figure 3.10: Extract of a news article of the digital newspaper USAToday, on 2015-10-25

```
 ('orrin hatch', 0.01851851851851853)]

>>> sorted(entry.where.items(), key=lambda pair: pair[1], reverse=True)[:10]
[('new york', 0.21052631578947367),
 ('afghanistan', 0.10526315789473684),
 ('new jersey', 0.10526315789473684),
 ('iraq', 0.10526315789473684),
 ('arizona', 0.05263157894736842),
 ('the united states', 0.05263157894736842),
 ("new york's", 0.05263157894736842),
 ('utah', 0.05263157894736842),
 ('south carolina', 0.05263157894736842),
 ('manhattan', 0.05263157894736842)]
```

The *Who/Where* distance between two entries is defined as the Earth Mover's Distance (EMD) between the two vectors of Named Entities. The EMD takes two vectors and a matrix of costs to move from one vector position to the other. In other words, taking the NEs in both *Who/Where* vectors and computing the matrix of distances between those NEs, the algorithm uses the EMD to define a value of the cost of passing from one vector to the other, considering the distance between two NEs as the cost of translation between them.

The EMD distance is used with the help of the *pyemd* Python library, that provides a wrapping of the C implementation of the algorithm. Its documentation can be found in [15].

Finally, since some of the Named Entities of the *Who/Where* vectors were discarded, their scores were ignored, and those should have some weight on the final distance. Since those scores are in some way a level of uncertainty of the distance measure between the two entries, the EMD distance (which is another real inside [0, 1]) is multiplied by 1 - penalty, where penalty is the maximum level of uncertainty in both entries. Then, the addition of that number with the penalty gives the final distance between the *Who/Where* vectors: another number between [0, 1].

3.8 Entry distance

Once with the three intermediate distances of an Entry, what remains is to define a global distance based on those values. That final step is done with help of the Stochastic Gradient Descent (SGD) algorithm.

3.8.1 SGD

Stochastic Gradient Descent is an optimisation algorithm that is capable of finding a local minimum of a function, given its derivative function, a starting point and a maximum number of iterations or a minimum precission.

The main idea behind this algorithm is to use the gradient of the function at any given point to locate the direction of maximum increment of the function value, so that follow-

ing the opposite direction approaches a local minimum. That idea, iterated a maximum number of times or until convergence ensures to find a local minimum if such exists.

This simple idea can be used for an optimisation algorithm. Given a cost function with at least one minimum and its derivative, by applying the SGD algorithm a local minimum is found, which could be a global minimum. In other words, a point where the cost function could be at the lowest. That cost function can have as input the coefficients of the global distance between entries; that way, if it was optimised on a proper dataset, the best choice of parameters (or at least a local minimum) would be the result of the optimisation.

The algorithm is implemented in a separate iPython Notebook. Based on the Pairs dataset retrieved with *newstagger*, all the W-distances between those pairs of entries in that dataset were computed and saved in a separate file. Then, with the cost function detailed in the following section, a choice of parameters was made.

3.8.2 Cost function

Assuming that the global distance function is linear with the intermediate distances, and since a distance is non-negative, the global distance would be defined as:

$$d(e_1, e_2) = c_{\text{what}} \cdot d_{\text{what}}(e_1, e_2) + c_{\text{who}} \cdot d_{\text{who}}(e_1, e_2) + c_{\text{where}} \cdot d_{\text{where}}(e_1, e_2)$$

where the coefficients c_{what}, c_{who} and c_{where} are non-negative and sum to 1 (to make the distance a value in [0, 1]).

The Pairs dataset offers tests formed from three entries and a value: a base entry, two different entries and the result of the following question: is base more similar to A or B? That value is an integer, 0 or 1, depending on which is more similar.

The cost function takes the intermediate distances, multiplies them with their respective coefficients and subtracts the base-A distance with the base-B distance. That value is positive or negative depending on which is more similar to base. Applying a sigmoid function to that value, it results in a real number closer to 0 or 1 depending on its sign and magnitude.

Finally, the cost function is defined as the sum of squared residuals between the dataset comparison value and the computed sigmoid value. And, for optimisation, that operation, instead of defined as a sum, is computed as a matrix operation to take advantage of the *numpy* library C-code optimisations.

Its derivative need not be computed by hand; the *autograd* Python library takes a function and returns another one that acts as its derivative. This operation is not an approximation of the derivative; the *autograd* library uses the chain rule and the definition of most *numpy* mathematical functions to calculate the appropriate derivative function.

For example, given any *numpy* mathematical function, autograd is capable of returning the derivative of that function as another Python function, as follows:

```
>>> # numpy must be used from autograd, since some functions are decorated to work
      with grad
>>> from autograd import grad, numpy as np
>>> from math import pi
>>> np.rad2deg(pi)
180.0
>>> rad2deg_derivative = grad(np.rad2deg) # this is the derivative
>>> rad2deg_derivative(pi), rad2deg_derivative(pi/2)
(57.29577951308232, 57.29577951308232)
>>> rad2deg_derivative(pi) == 360 / 2 / pi #Â derivative is constant: 360/2pi
True
```

With the established cost function, its autograd derivative and the SGD algorithm, with a precision of 0.001 and almost 3000 iterations the following coefficients are found:

$$c_{\text{what}} = 6.22461219$$
$$c_{\text{who}} = 5.57323577$$
$$c_{\text{where}} = 5.22079321$$

Those coefficients are then normalised so that they add to 1, in order to contain the global distance in [0, 1]. Then, it results in the following values:

$$c_{\text{what}} = 0.3657526$$
$$c_{\text{who}} = 0.32747831$$
$$c_{\text{where}} = 0.3067691$$

It can be seen how those values are almost identical, but *What* has the priority. It is not surprising, considering that it gives a good model for topics, but it is worth mentioning that the algorithm still uses *Who* and *Where* as important values for the distance.

In the following chapter, the results of the algorithm will be detailed and assessed in comparison with the LDA algorithm.

Chapter 4

Results, conclusions and applications

In order to check if the algorithm is suitable for the problem, it has to be evaluated with a training set. With the Pairs Dataset and using Cross Validation, the algorithm can be assessed and compared with the LDA algorithm.

This analysis is made with an iPython Notebook, but this section covers the main insights and discusses the results.

4.1 Algorithm assessment

As said before, the algorithm is checked with the use of Cross Validation and the Pairs dataset. The SGD algorithm is encapsulated in a Python class that implements the basic *scikit-learn* methods, so that the "sklearn.cross_validation" module can be used with this structure.

Figure 4.1 shows the average accuracy of each fold for several values of K, being K the number of folds to compute. The X-axis shows the proportion of the whole Pairs dataset that the K-Fold considers as training set. The Y-axis represents the average accuracy of that experiment. The number of folds goes from 2 to 20.

The Y-axis is rescaled to 0.84-0.91 since the lines are too close and displaying [0, 1] would not make the visualisation useful. Each of these lines is a different algorithm, explained below.

- **sgd, what with NE**: the basic algorithm explained throughout this work. Uses *What*, *Who*, *Where* normally.
- **sgd, what without NE**: the basic algorithm, but *What* (LDA) does not get any Named Entities as input.
- **what with NE**: basic LDA algorithm, without *Who* or *Where*. NE are used as LDA input.

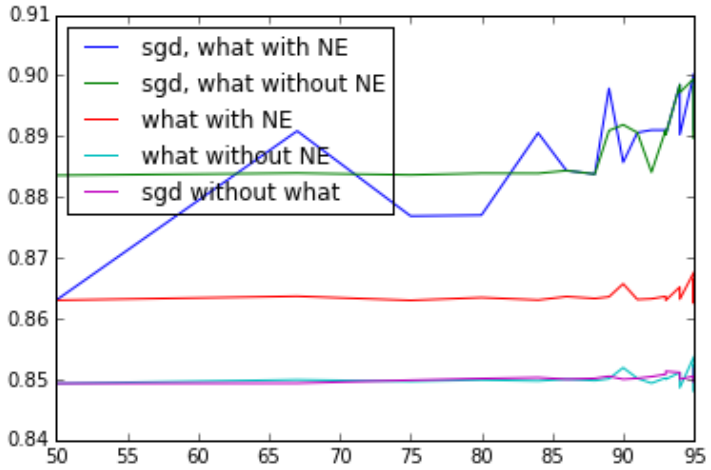

Figure 4.1: Algorithm assessment 0.84-0.91

- **what without NE**: basic LDA algorithm, without *Who* or *Where*. NE are not used as LDA input.
- **sgd without what**: basic algorithm without using the *What* (LDA) part.

The reason why all these variants were analised will be explained later.

Firstly, comparing "sgd, what with NE" (proposed algorithm) with "what with NE" (basic LDA) shows that the proposed algorithm improves accuracy on a 2% approximately with respect to LDA, achieving a total accuracy of 88.5%. That means that, according to the Pairs dataset, the algorithm creates a distance that is capable of sorting closeness between politics entries on a 88.5% of cases.

However, this figure is not very reliable. As stated before, the Pairs dataset was really difficult to build and has got only 146 tests, a small amount considering what is usual for Machine Learning algorithms. This scarcity of tests to base results on makes that 88.5% uncertain, and in reality could result in a different figure. So, the fact that the algorithm improves in a 2% of accuracy should not be considered relevant. More so, it is worth remembering that the algorithm is still using LDA as an intermediate step; for that reason, it is not surprising that it improves accuracy.

So the fact that the accuracy improved a 2% should not be considered. The analysis goes on another direction.

LDA is being used as a established algorithm for comparison, and it is also a part of the proposed algorithm. But LDA uses counts of words in its input, and the Named Entities might be taking part in that process. Since both parts of the method use NE, what may be happening is that the accuracy got by LDA might be influenced by those NEs that form the whole basis of the *Who*/*Where* parts.

So, "what without NE" uses LDA without NEs, and the result is that its accuracy decreases on almost the same amount that was gained with SGD. This ensures that LDA is in fact using NEs for this problem, so eliminating it should have an effect on the results.

But with "sgd, what without NE", those results are intact. Even if LDA loses the information gained by NEs, *Who* and *Where* use it and, it seems, use it better, because the main accuracy is not affected but becomes stable across the several values of K for the K-Fold.

Finally, the last alternative, "sgd without what", uses the *Who/Where* parts of the distance but ignores the LDA part. Its results are almost the same as "what without NE", and that is a sign of the true power of those vectors. The *Who/Where* vectors, along with their defined distance, using Wikipedia data in the process, is capable of mimicking the LDA algorithm if it did not use Named Entities as its input.

The conclusion to extract here is that the proposed algorithm is at least as suitable as the LDA algorithm in the problem of defining a distance between politics entries. But it should not be forgotten that this approach is quite intensive in terms of space and time; the LDA algorithm is much faster and has no need for all the Wikipedia data and the intermediate computations needed for the proposed approach.

Conclusions about these results will be detailed in the following chapter.

4.2 Applications

This section tries to answer the questions asked in the introduction, giving some applications to the algorithm.

For the following two visualisations a threshold value to distinguish between related and not related entries is needed. The default value for such a threshold is 0.6, but it is actually computed with a tagged dataset (created with a script from *newsbreaker.scripts*) and the *scikit-learn* implementation of Linear Discriminant Analysis, used to get the decision boundary of the classifier, which would be the threshold.

This computed threshold is used in the second visualisation, that compares a base entry with all entries in the next 6 days. However, the first visualisations do not use it, since the objective there is to define a value with which isolated groups of nodes can be defined. The specific values are stored in the data folders for each of the visualisations.

All the visualisations from this section are built with the D3 visualisation library. A basic introduction to D3 can be found in [16].

- Is there a way to see just in a glance all the major stories that happened in a day? Is it possible to get the most accurate depiction of a specific story available on the web?

Given that the algorithm constructs distances between entries, by establishing a threshold of what are considered similar articles and what are not, a network can be built with those distances. If a pair has a distance lower than the threshold, it is considered connected by an edge with value as that distance.

Given that network, clusters can be computed, both programmatically or visually. The following charts show visualisations of those networks, computed with a Force Layout of the D3 Javascript library. Each node color represents a different feed and its size represents the addition of all edge values where that node is connected. That is done to highlight the

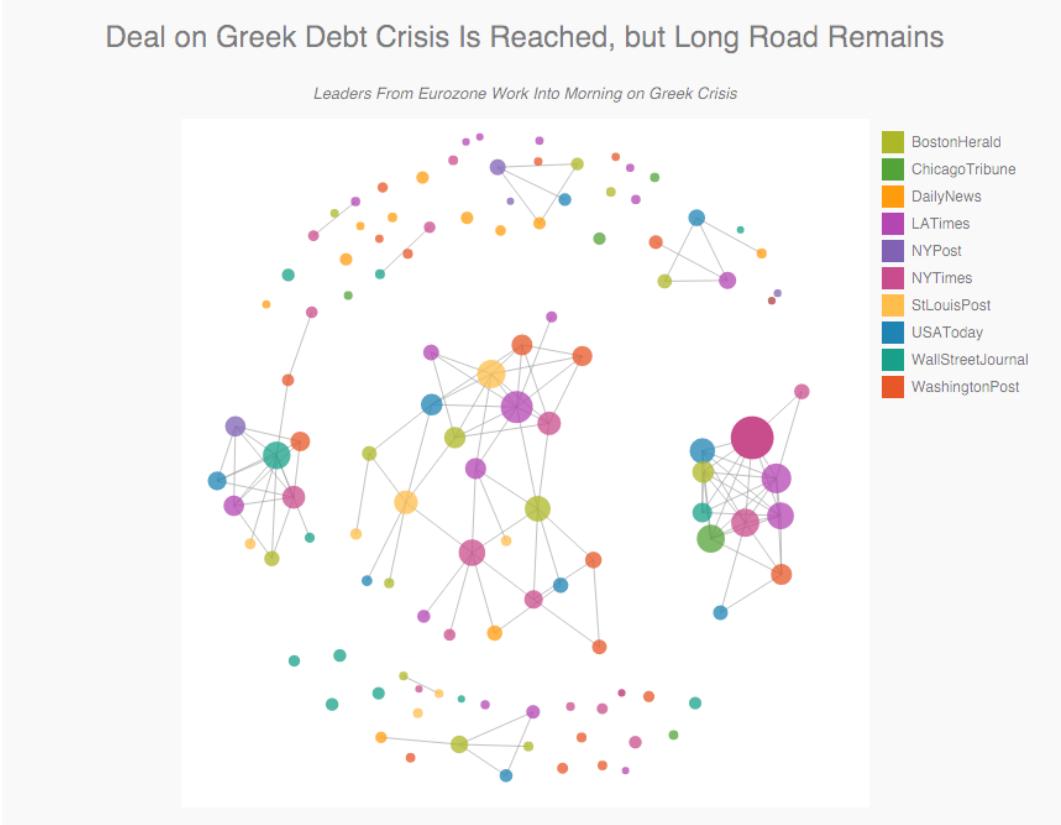

Figure 4.2: 2015-07-13, Greece bailout deal

most connected news in each cluster. Also, each node repels with a charge proportional to that connectivity value (its size).

Charge and threshold values can be set with query string values in the URL: threshold and charge, respectively.

Figure 4.2 displays the day after the Greece bailout deal. There are three main groups at the centre of the image. From left to right, they talk about the Iran Nuclear Deal, the United States presidential campaign (the nodes on top talk about the Democratic party, and the lower nodes, about Scott Walker entering the race as a Republican Party candidate) and, on the right, the Greece bailout deal.

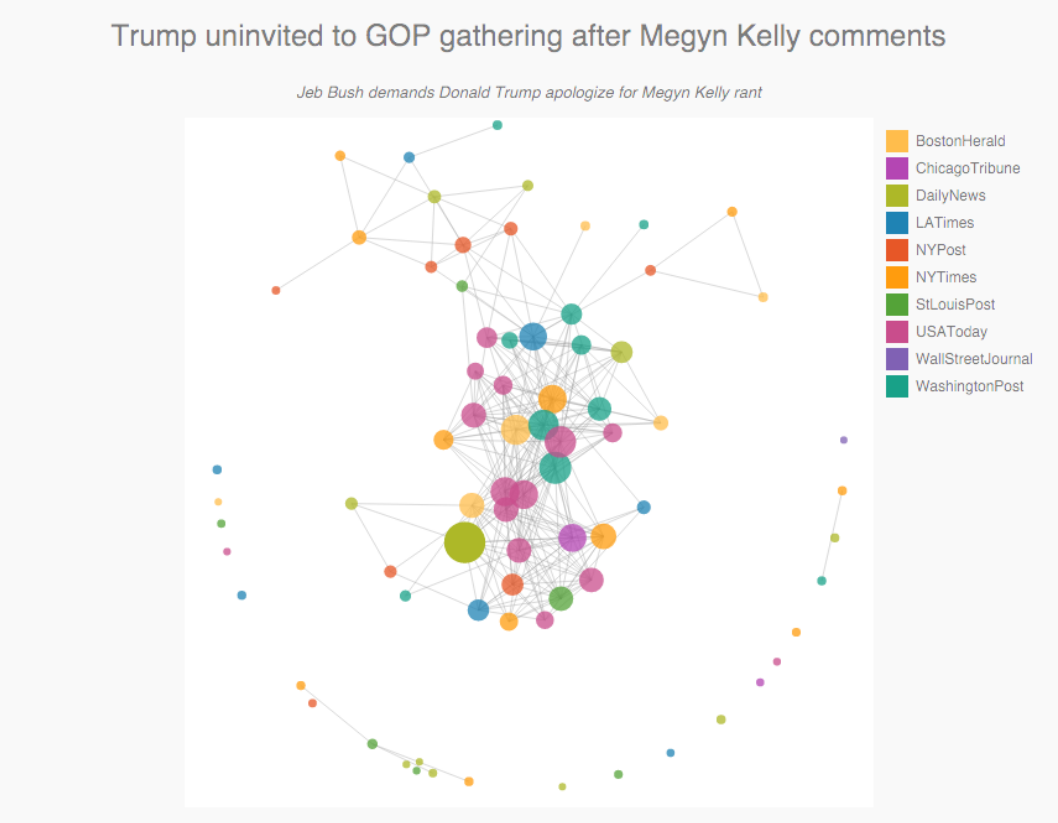

Figure 4.3: 2015-08-08, Donald Trump makes outrageous remarks about Megyn Kelly

Figure 4.3 shows a clear domination of the top story of the day: the GOP debate on the previous night, with a subset of the nodes talking about the Donald Trump remarks, that caused a lot of controversy.

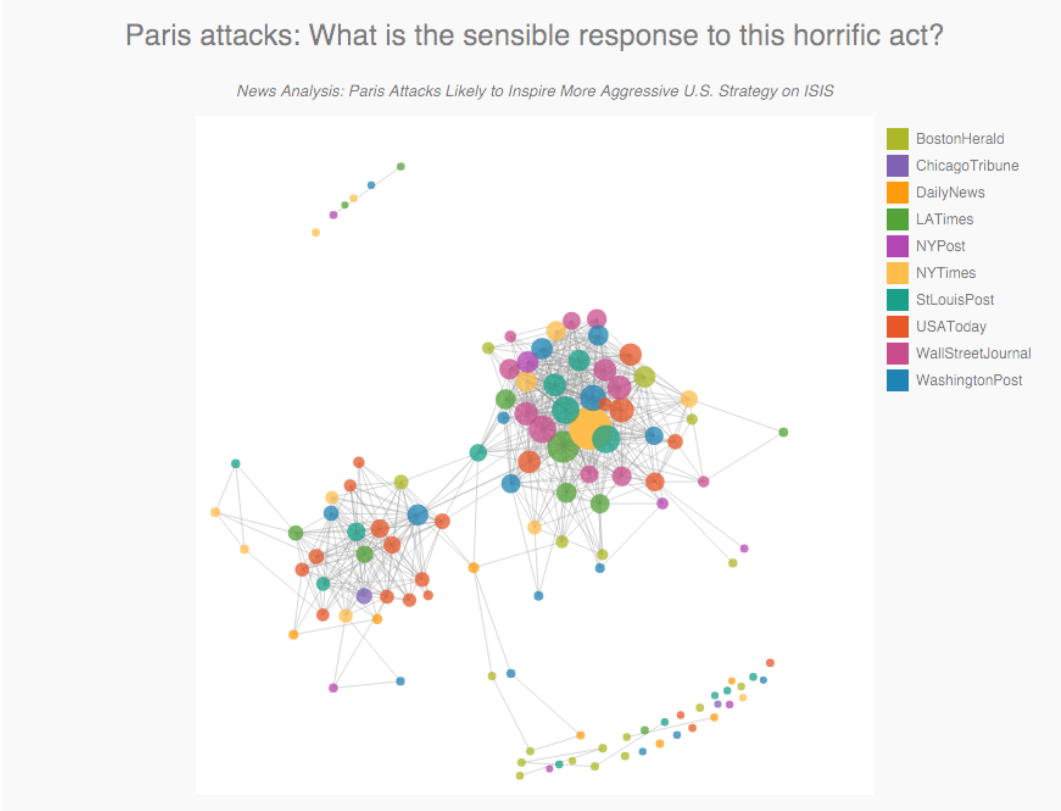

Figure 4.4: 2015-11-14, Paris terrorist attacks

Finally, figure 4.4 talks about the Paris terrorist attacks. There are two main stories on this day, but both groups are connected by some news. The bigger group talks about the Paris attacks; the other talks about the presidential race and how the Paris attacks can affect U.S. politics. That is the reason why they are connected.

With this kind of visualisation (better checked on the browser, since nodes are interactive and display the title of each entry) a quick glance at the stories of the day can be made. With the help of a clustering algorithm and a Word Cloud visualisation of every cluster found, that visualisation can be improved.

On the question of the most accurate depiction of a story, several approaches could be tried, but one is to find the most central node in the cluster of that story based on the distance returned by the algorithm. Since this distance tries to establish similarity between nodes based on their content, the entry that has more similarity to the rest of the articles that talk about the same story should have parts in common with all the rest, thus making for the most accurate article.

Apart from that, the applications of the system showed the importance of building a content-based distance between entries.

The distance value itself is not what is important in this algorithm; in fact, the distance function was trained to improve ordering, not to accomplish certain definite values. By that ordering, clusters can be found between entries on the same day and then further analysis on the topics of those clusters and on the network they build lead to an understanding of the news landscape for that particular day.

In other words, work on this problem is in fact a way to build a representation of the daily state of the world, in terms of news events.

- Is there a way to see how a given story still has interest days after happening?

Given the established threshold to check if two entries talk about the same matter, taking an entry as a base and computing the distance of that base to the rest of the articles on that day and on the following six days could give an answer to that question. The following chart plots each entry as a dot, whose size is proportional to the similarity (1 - distance) to the base entry, and its x coordinate corresponds to the day it was published. By giving an alpha value to each of those dots, overlapping of entries can be seen too.

The number of entries overpassing the assigned threshold marks how that entry is still being discussed days after happening. Each chart takes an entry as base, which is the title of the chart. By putting the cursor over any of the feeds in the legend box, that feed is highlighted and the interest in that story for that newspaper can be seen.

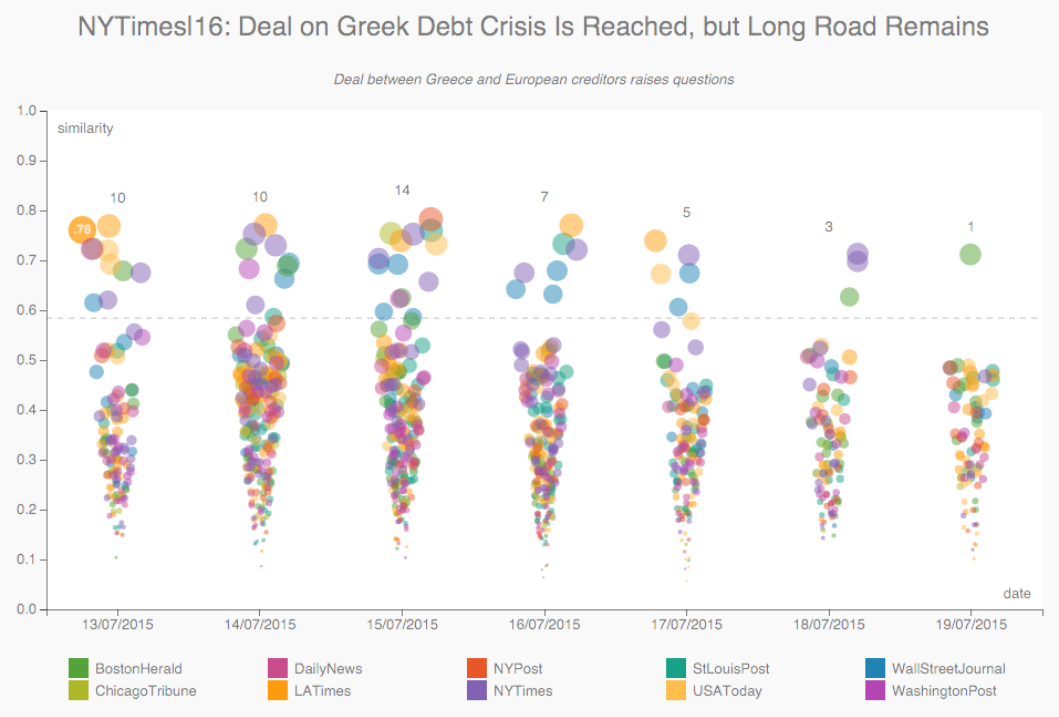

Figure 4.5: Leaders from Eurozone work into morning on Greek crisis

Figure 4.5 shows an interest on that story the first three days after happening, first with the news about the deal itself and in the following days, with the statements of several public figures. After 2015-07-15 that interest starts decreasing.

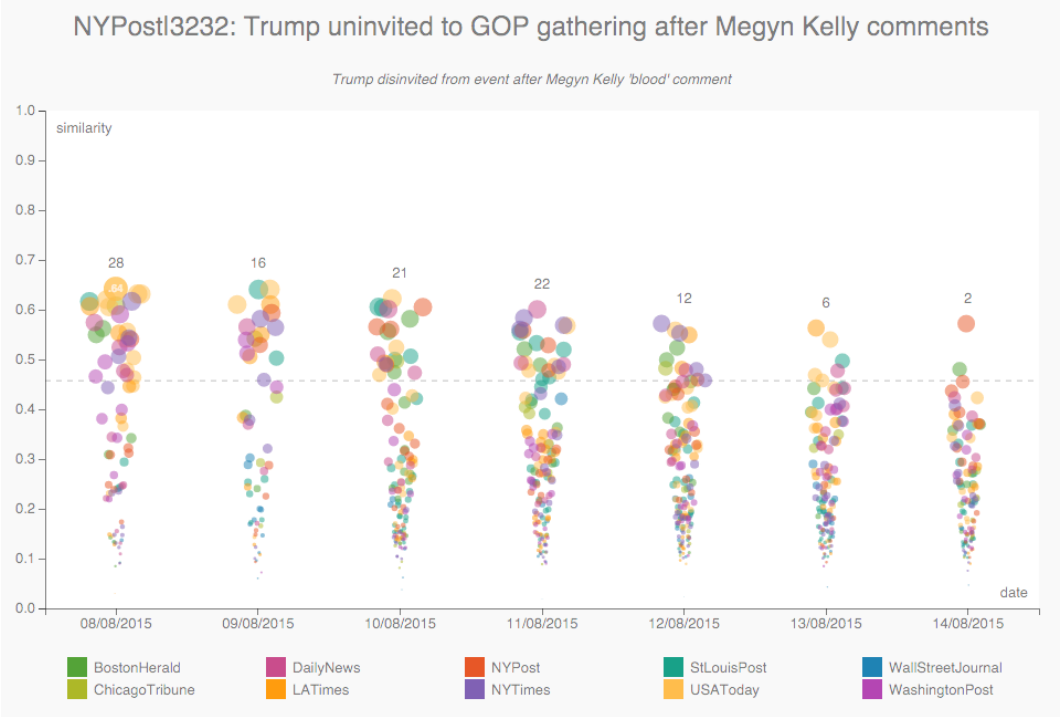

Figure 4.6: Hand-Wringing in G.O.P. after Donald Trump's remarks on Megyn Kelly

Figure 4.6 talks about the Donald Trump remarks. Since this is a U.S. affair with controversy, interest lasts more days. It is worth noticing that Wall Street Journal takes no interest in this story, since by hightlighting it's nodes, it doesn't appear over the line in any day. This is not surprising considering it is supposed to be an economic newspaper.

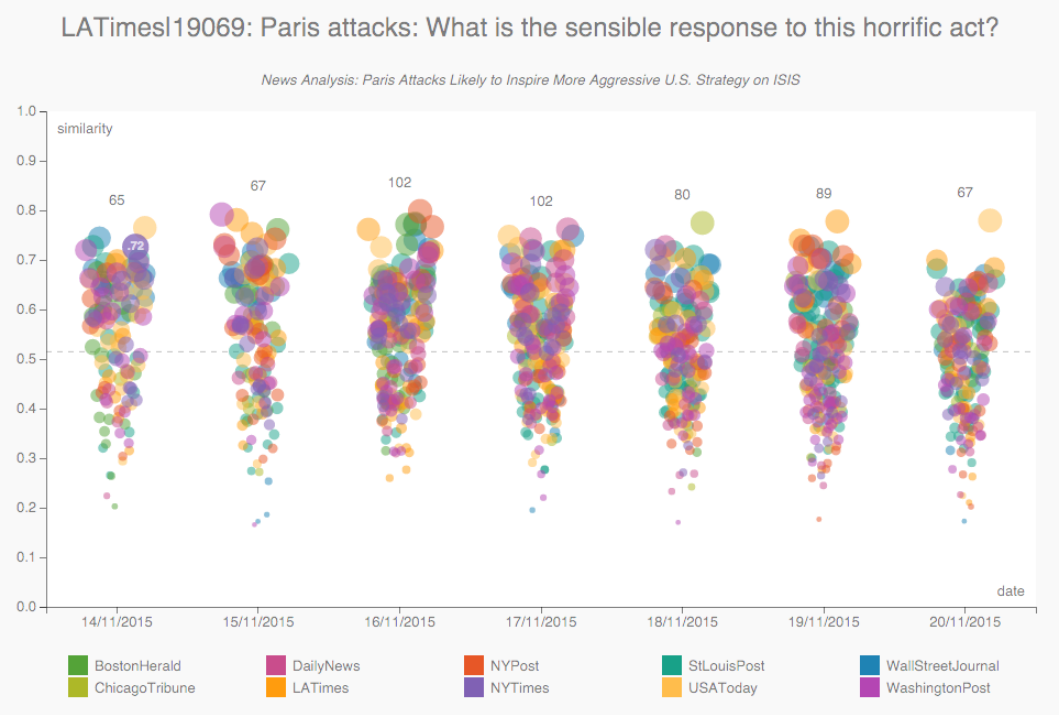

Figure 4.7: Paris attacks: What is the sensible response to this horrific act?

Lastly, figure 4.7 shows the impact of the Paris terrorist attacks. A conclusion driven with this chart is that Wall Street Journal takes an interest in this story: every day an average of 9 news articles talk about this matter, or up to 12 in the fourth day. This proves that those attacks had economic interest too.

As a final note, the amounts of data in every database of MongoDB used for this project are the following:

distances	0.078GB
entries	1.953GB
newstagger	0.078GB
wiki	3.952GB

Chapter 5

Conclusions and further work

This last chapter details the conclusions of this work and how this project could continue, both improving the current approach or taking a different path.

5.1 Conclusions

Based on the results of the previous chapter, one conclusion is clear: LDA is an excellent algorithm to face this kind of problem. The transformation it performs on the word-frequency vector of an entry becomes a representation of the topic of that entry. That vector is easily comparable to another one by means of a distance function, which is just the histogram distance. That and the LDA transform method are really inexpensive, both in terms of time and space.

LDA accuracy is quite high for a general purpose algorithm: it should be reminded that LDA uses no special knowledge about the documents it transforms other than the frequency of the words they contain. The proposed algorithm did use more information: the 5W1H model establishes a framework to properly consider each category of information (each W question) and the Named Entities system based on Wikipedia data provides more knowledge about the words of an entry. Even with that, LDA was almost equivalent to that system and, as said before, its performance was the best.

For these reasons, LDA should become the *de facto* algorithm for this kind of problem, both as a basic system and as a ground truth to assess other approaches. More so, the fact that it takes no advantage of the characteristics of news articles (the compliance with the 5W1H model) implies that it could still be improved by combinating it with other algorithms.

This analysis also proves that the proposed algorithm is a path to explore. The *Who*/*Where* vectors, along with its distance, do a better job at working with the Named Entities in the text than the LDA algorithm, as proved by the comparison of the whole algorithm with the What-without-NEs option. With a better dataset (quite limited in this project due to limitations of time and data) and more work on those Named Entities, better and more stable results could appear.

Even more, the 5W1H approach could lead to better results when working not only

with Named Entities. The fact that *Who* and *Where* were based on NEs restricted the scope of this project to politics or other kinds of articles where public figures usually appear. News like natural disasters, crime or economics (when not talking about public figures) may not act properly with this system. However, if *Who* and *Where* are not restricted to NEs, those topics could be processed with this approach.

The 5W1H model should then be used as a framework for different vector representations of an entry. With each of those vectors, different techniques could be used, restricted to the corresponding W category; those alternatives could then improve the accuracy of the system. Examples of such algorithms are the recent *sent2text* algorithm, that transforms a sentence into a numerical vector, or other Neural Network approaches.

In summary, the 5W1H approach, along with proper distances for each of the questions, is a good way to build the representation of a news article, as the analysis indicates. Further work with this approach, solving the weak spots of the current algorithm and expanding its scope to other techniques other than NE vectors could very well lead to better accuracy on the problem.

5.2 Further work

This last section suggests some improvements to the current algorithm, points that could not be worked on this project due to limitations of time, data and human tagging.

- Follow more RSS feeds, from several countries: by following only ten feeds, all from the United States, the dataset became too biased to that kind of articles. Also, the entries downloaded go from July to December, and in those months only a couple of stories were highlighted in United States Newspapers. With bigger time spans and more diversity, the algorithm would learn on other kinds of stories and further analysis could be made with the distance function.

- Download the Wikipedia dumps and work on local: the Wikipedia API is really slow and that affected as the major bottleneck of the whole system. By having all that information on local already stored and processed, the algorithm would run much faster. However, processing the Wikipedia dump (50GB) was not the objective of this work and was discarded for lack of time and complexity.

- Built a NE distance dataset: just like the Pairs dataset was built, a dataset for pairs of NEs could be used for the distance between NEs. With such a dataset, the proposed distance between Named Entities could be replaced by another continuous function whose parameters may be acquired by an optimisation algorithm, just like it was done with SGD and the final entry distance. That way, the NE matrices could improve their values and, as a result, have an impact on how *Who* and *Where* work.

- More diverse and bigger Pairs dataset: the Pairs dataset is essential in building the final distance between entries, and the fact that it was so small made impossible to consider other kinds of distances or assess the true accuracy of the system. This dataset should also be built by more than one person, since its results can get biased because of the influence that a single tagger has.

- Improve performance of the system by optimising several steps of the process.
  - Index MongoDB properly to reduce fetching times. MongoDB was used as a storage system and no optimisation techniques were tried on it, since it was not the focus of this project and there was no time to do so.
  - Distance between Named Entities was not stored persistently because the algorithm was constantly changing during development, and that would create the need to erase that data every time the algorithm changed. Such a dataset would also increase in terms of space quite heavily and fetching those values would become an issue if the database was not properly indexed.
  - Store the representation of an entry in terms of *What*, *Who* and *Where* vectors, since those vectors were deleted when the entry object was destroyed. The real bottleneck of the algorithm was in the Wikipedia API, but this step also takes some time and is easily solved.
  - Use some modern approaches like Hadoop or Spark and Cloud Computing to work through the more extensive steps of this process, since they are capable of processing huge amounts of data in short periods of time, and thus could shorten quite heavily the execution time of the algorithm.

This work, however, does not consider the proposed approach as the only way. The 5W1H approach is valid and could lead to better results, even in more topics other than politics. Also, it is worth trying the use of a Neural Network for this problem, since it is a field that is giving good results on all kinds of problems, but it would entail the need of a much bigger dataset, something this project could not afford to try.

Bibliography

[1] Docs | spaCy.io. (v0.15.1). [Documentation]. Retrieved from https://spacy.io/docs

[2] Bird, S.; Klein, E.; Loper, E. (2009, Jun.). *Natural Language Processing with Python*. Sebastopol: O'Reilly Media Inc.

[3] Das, A., Ghosh, A, Bandyopadhyay, S. (2010, Aug.). Semantic role labeling for Bengali using 5Ws. *2010 International Conference on Natural Language Processing and Knowledge Engineering (NLP-KE)*. 1-8, 21-23.

[4] Luhn, H. P. (1958, Apr.). The Automatic Creation of Literature Abstracts. *IBM Journal of Research and Development*. Vol.2, No.2, 159-165.

[5] Montalvo, S., Martínez, R., Casillas, A., Fresno, V. (2007). Bilingual News Clustering Using Named Entities and Fuzzy Similari ty. *Text, Speech and Dialogue; 10th International Conference*. Vol. 4629, 107-114.

[6] PyMongo 3.2 Documentation — PyMongo 3.2 documentation. (v3.2). [Documentation]. Retrieved from http://api.mongodb.org/python/current/index.html

[7] (n.d.). *Top 50 United States Newspapers*. Retrieved July 2, 2015, from http://www.onlinenewspapers.com/Top50/Top50-CurrentUS.htm

[8] Documentation — feedparser 5.2.0 documentation. (v5.2.0). [Documentation]. Retrieved from https://pythonhosted.org/feedparser/

[9] Newspaper: Article scraping & curation — newspaper 0.0.2 documentation. (v0.0.2). [Documentation]. Retrieved from http://newspaper.readthedocs.org/en/latest/

[10] Documentation scikit-learn: machine learning in Python — scikit-learn 0.17 documentation. (v0.17). [Documentation]. Retrieved from http://scikit-learn.org/stable/documentation.html

[11] Richert, W.; Coelho, L. P. (2013, Jul.). *Building Machine Learning Systems with Python*. Birmingham: Packt Publishing Ltd.

[12] Welcome to Flask — Flask Documentation (0.10). (v0.10). [Documentation]. Retrieved from
http://flask.pocoo.org/docs/0.10/

[13] Wikipedia — wikipedia 0.9 documentation. (v0.9). [Documentation]. Retrieved from https://wikipedia.readthedocs.org/en/latest/

[14] lda: Topic modeling with latent Dirichlet Allocation — lda 1.0.3 documentation (v1.0.3). [Documentation]. Retrieved from
http://pythonhosted.org/lda/

[15] wmayner/pyemd: A Python wrapper for Pele and Werman's implementation of the Earth Mover's Distance metric. (v0.2.0). [README with Usage]. Retrieved from
https://github.com/wmayner/pyemd

[16] Murray, S. (2013, Apr.). *Interactive Data Visualization for the Web* (2nd ed.). Sebastopol: O'Reilly Media Inc.