UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DEVELOPMENT OF AN EVENT DETECTOR IN TWITTER STREAMS BASED ON MENTION-ANOMALY DETECTION FOR THE CITY OF MADRID

LUIS CRISTÓBAL LÓPEZ GARCÍA
JUNIO 2019

TRABAJO DE FIN DE GRADO

Título:	Desarrollo de un Detector de Eventos en capturas en directo de Twitter basado en la detección de Menciones para la ciudad de Madrid
Título (inglés):	Development of an Event Detector in Twitter Streams based on Mention-Anomaly Detection for the City of Madrid
Autor:	Luis Cristóbal López García
Tutor:	Carlos A. Iglesias Fernández
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO FIN DE GRADO

DEVELOPMENT OF AN EVENT DETECTOR IN TWITTER STREAMS BASED ON MENTION-ANOMALY DETECTION FOR THE CITY OF MADRID

Luis Cristóbal López García

Junio 2019

Resumen

La detección de eventos ha sido un campo de investigación desde mucho antes que las redes sociales alcanzaran el gran impacto que tienen hoy en día. Estos eventos eran monitoreados desde los sitios web tradicionales de noticias, blogs u otros canales de información. Sin embargo cuando el *nanoblogueo* apareció como una forma de comunicación social todo este panorama cambió.

En este proyecto hemos desarrollado un sistema capaz de detectar los eventos más importantes ocurridos en una ciudad a base de analizar datos publicados en redes sociales. Para ello hemos adaptado un modelo de clustering ya existente el cual se basa en el número de interacciones entre usuarios para medir el grado de impacto. Nuestras principales contribuciones a este modelo han sido mejorar la precisión del propio algoritmo de impacto y proporcionar una nueva definición de redundancia que ha significado un mejor comportamiento frente a eventos duplicados.

La red social que utiliza nuestro detector es *Twitter*, considerada una preciada fuente de lo que se conoce como *datos sociales*. La información es proporcionada por los propios usuarios a través de unos documentos de pequeño tamaño llamados *tweets*.

De forma adicional hemos desarrollado una arquitectura que junto al cluster convierte este proyecto en un sistema. Los datos serán recolectados mediante una captura en directo que después proporcionaremos a nuestro detector. Sin embargo estos necesitan pasar primero por un módulo de preprocesado que se encarga de filtrar el spam y lematizar el texto para así conseguir un mejor comportamiento. Una vez ha finalizado la tarea de detección los resultados serán almacenados en el subsistema de persistencia y posteriormente visualizados en un cuadro de mandos que interactúa con el usuario y facilita la fase cognitiva del análisis realizado. Todo este flujo de datos está supervisado por un orquestador encargado de asegurar la correcta interacción entre módulos.

Este proceso se repite periódicamente cada media hora mostrando los tres eventos con mayor impacto que han tenido lugar en la ciudad de Madrid en las últimas 24 horas.

Palabras clave: evento, detección, detector, cluster, clustering, menciones, redundancia, filtro, visualización, Machine Learning, Python, Twitter

Abstract

Event detection has been a field of research long before social networks reached the high impact they have nowadays. Events were tracked from traditional news web sites, blogs or other information channels. However when microblogging as a form of social media emerged all this landscape changed.

In this project we have developed a system capable of detecting the most important events occurred in a city by analyzing data published on social networks. For this, we have adapted and improved an already existing clustering approach named MABED, which relies on the number of interactions between users to measure the impact. Our main contributions to this model has been to improve that impact algorithm accuracy and to provide a new definition of redundancy leading to a better performance on duplicated events.

The social network our detector reads is *Twitter*, considered a valuable source of what is known as *Social Data*. Information is provided by short length documents posted by users, called *tweets*. These publications are collected from our Streamer, gathering posts that have just been published in the city of Madrid.

In addition to the cluster we have also developed an architecture that turns our project into a system. Streamer is in charge of collecting the data that we feed to our detector. However it first needs to pass through a preprocessing module which filters spam out and lemmatizes the text in order to achieve a better performance. Once the detection task is finished results are saved in a persistence subsystem. These results are finally visualized in a dashboard which interacts with the user and facilitates the cognitive process of the performed analysis. All this data flow is supervised by an orchestrator which assures the correct interaction between modules.

The process we have just explained is repeated periodically every half an hour showing top three events with the higher impact that took place in the city of Madrid in the last 24 hours.

Keywords: event, detection, detector, cluster, clustering, mentions, redundancy, filter, visualization, Machine Learning, Python, Twitter

Agradecimientos

Me gustaría dedicar esta sección para dar las gracias a todas las personas que me han ayudado a lo largo de estos años.

A mi familia y en especial a mis padres, José Luis y María del Carmen, por haberme apoyado en todo momento, haberme ofrecido la mejor educación posible y haberme enseñado a través del ejemplo lo que es realmente importante en esta vida.

A mis amigos, los de siempre y los que he descubierto aquí en Madrid, porque soy quien soy hoy en parte gracias a ellos.

Y cómo no a mi tutor Carlos Ángel Iglesias y todos mis compañeros del Grupo de Sistemas Inteligentes que me han ayudado y orientado estos meses durante la realización de este proyecto.

Muchas gracias a todos.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Context

Microblogging as a form of social media has fast emerged in recent years. One of the most representative examples is *Twitter*, which allows users to publish short tweets (messages within a 140-character limit) about "what's happening". This social network has become an important information channel for users to receive and to exchange opinions. This information ranges from thoughts to private moments and livings which they want to share with their environment. In 2014, Twitter daily volume of tweets was estimated at nearly 500 million [1], being created and redistributed by millions of active users.

Companies are increasingly using Twitter to advertise and recommend products, brands, and services; to build and maintain reputations; to analyze users' sentiment regarding their products (or those of their competitors); to respond to customers' complaints; and to improve decision making and business intelligence. Twitter has also emerged as a fast communication channel for predicting election results, sharing political events and conversations or for gathering and spreading breaking news. The field on which we will focus our study is precisely this last one.

Twitter has several unique advantages that distinguish it from news web sites, blogs or other information channels:

- With the brevity guaranteed by its 140-character limit and the existence of Twitter's mobile applications, users tweet and retweet instantly. There are reported cases were tweets related to a shooting were detected 10 minutes after shots fired, while the first news report appeared approximately 3 hours later [2].
- Tweets covers nearly every aspect of daily life, from national breaking news, local events to personal feelings. This makes possible that, since millions of accounts are constantly publishing tweets, every user can report news that is happening around him or her.
- Tweets are not isolated as they are associated with rich information. For example, for each tweet, we can find an explicit time stamp, the name of the user or even the GPS coordinates if the document was created with a GPS-enabled mobile device.

With these features Twitter is in nature a good resource for detecting and analyzing events.

In general, events can be defined as real-world occurrences that unfold over space and time. An important characteristic of Twitter is its real-time nature; users tend to post about any situation happening in their day-to-day life with a margin of minutes. Monitoring and analyzing this rich and continuous user-generated content can yield valuable information, enabling users and organizations to acquire actionable knowledge [3].

However, Twitter streams contain large amounts of meaningless documents and polluted content which negatively affect the usable ones. In contrast with the well written news releases, Twitter messages are restricted in length and written by anyone. They include large amounts of irregular and abbreviated words, in addition of pointless messages and rumors which hinder the performance of the detection algorithms.

For this reason, and although an immense number of different detection approaches exist, all of them have the common purpose of efficiently discerning between these two types of data, being able to extract useful information from the entire stream and analyze it in order to detect when a possible incident occurs.

In the following section we will introduce the different objectives we set at the beginning of the project in order to develop our own event detector. However, as it progressed new necessities arose and were added to the final achieved goals.

1.2 Project goals

The main objective of this project is the development of an event detection system which tracks the most recent occurrences in and around the city of Madrid. This system will need to be able to: (I) detect new events, (II) analyze the spatial and temporal pattern of an event, and (III) identify importance of events. Our detector will be based on the definition of event as a number of keywords showing a *burst* in appearance count.

As argued in Sect. 1.1 the information used for this process will be collected from Twitter streams. This study will focus only on Spanish-written tweets as it is the language which will return a higher number of coincidences given the considered city.

In order to achieve that goal we will need to modify and improve the already existent model so that it fits our project requirements.

Detection results will be finally displayed in a visualization module allowing the viewer to identify each event characteristics as well as its temporal and spatial patterns through different widgets.

1.3 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The structure is as follows:

Chapter 1 is the introduction of the project. A brief context explanation is first provided, followed by the goals it aims to achieve.

Chapter 2 enumerates the different technologies which have been employed along the project and explains what they will be used for.

Chapter 3 introduces the detection model. First the approach our project is based on is explained. Then a description of the limitations found in it is listed. Finally we explain each different change proposal we have raised, resulting into our final model.

Chapter 4 describes our event detection monitoring system. We first provide an overall view of its architecture and then each module is deeply explained.

Chapter 5 shows the use cases our model has been tested on demonstrating that our change proposals are actually improvements.

Chapter 6 discusses the conclusions, the achieved goals and future work.

CHAPTER 2

Enabling Technologies

In this chapter we summarise the different technologies that have been used to develop this project. Although the main detection task is performed in Python we have used several programming languages such as HTML, CSS and Javascript, among others.

2.1 Python libraries

As mentioned above the detector is Python-written, what means that it will make use of several modules required in order to achieve a proper functioning. In the following subsections we are going to describe those of them considered essential in our field of study and declare what will they be used for.

2.1.1 NumPy

NumPy [4] is the fundamental package for scientific computing in Python. We have made use of it in our project since it provides an efficient multi-dimensional container of generic data and a large collection of tools for working with them. NumPy's arrays provides a better performance than traditional Python lists in terms of size, speed and functionality,

so there will be occasions when managing large amounts of data or numerical values where we will use them.

Besides the quoted above use, this package also contains many other interesting functionalities such as:

- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

2.1.2 SciPy

SciPy [5] is a Python-based ecosystem of Open-Source software for mathematics, science, and engineering. It is built on NumPy array object and it is part of the NumPy stack; however SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

Our main interaction with this library has been the usage of its *sparse matrices* classes and methods since it is a very efficient way of relating two variables along their different possible values over time.

2.1.3 Scikit-learn

Scikit-learn [6] is Python machine learning library whose features range from machine learning tasks such as classification, regression and clustering algorithms to support vector machines, random forests, gradient boosting, k-means and DBSCAN. It is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

In our project we have used some of the features quoted in the previous paragraph, in concrete *tfidfvectorizer* and *cosine similarity*. With this two packages we will build our spam filter, explained in Sect. 4.3.1.

2.1.4 NetworkX

NetworkX [7] is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. These networks are composed by *nodes*, which can be text, images or XML records (among other things) and contain any kind of information you want to add in order to provide the nodes with uniqueness. At the same

time these nodes can be linked between them by *edges*, which can also contain information describing them. There are divers types of Graphs, being *Directed Graph* and *Undirected Graph* the ones we have used.

This package has a huge variety of uses and it is incredibly customizable to the point where it can be used in any project where it exists any relationship between their components with interest of being studied. In our case it has been used with as objective finding similar events detected by our algorithm which can be considered as redundant. This is described deeply in Sect. 3.3.3.

2.1.5 Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms [8].

Used in conjunction with NetworkX allows us to see the representation of our different Graphs and the relations established between elements. Some of the figures shown in this memory are generated using this tool.

2.2 NLP Cube

NLP Cube [9] is an opensource Natural Language Processing Pipeline developed by Adobe[1] which allows us to perform text analysis and transformations to the desired form. Although it is a relatively new project (it has just released version 1.1) it has been tested and proved to have a high accuracy performance working with Spanish sentences.

Some of the tasks this framework can develop are sentence Splitting, Tokenization, Lemmatization, Part-of-speech Tagging and Dependency Parsing. As our project has its particular Tokenization model we will make use only of Tagging and Lemmatization NLP Cube's models.

DyNet [10] is also used in order to maximize the efficiency of the running CPU. With a proper initial configuration we can set DyNet so that NLP Cube can take advantage of all our CPU cores and perform a faster and more efficient analysis, leading to a faster whole script running time.

[1] http://opensource.adobe.com/NLP-Cube/index.html

2.3 Twitter API

Twitter is a social network which generates loads of information about their users which comes in small documents called *tweets*. Not only the content of the document is considered as valuable information, but also its metada as the author ID, creation date or coordinates. Twitter provides developers an API [11] which they can use in order to facilitate this task. The platform[2] allows us to access, read, write and collect Twitter data specifying any search terms to filter the results. These terms can be a hashtag or a specific word or location, for example.

If we follow the link in the footnote we will find that there are several APIs being offered, going from strict tweet collecting to personalized Direct Messages ones. Due to the project objective we have considered using only two of them:

- **Search API:** as it is described on its main page this API can be used to find *historical tweets* that has already been posted. Twitter sets a limit of an elapsed week to the tweets we can retrieve using this platform.
- **Streaming API:** opposite to the previous service, Streaming API allows us to collect tweets that match our ruleset within seconds of being published on the platform. However, although time limits are not established in this API the volume of received tweets is affected instead, returning only 1-2% of all tweets posted while the streamer is working.

The decision of which one we have decided to use is detailed on Sect. 4.2.

2.3.1 Tweepy

The number of people using APIs like Twitter has increased in recent years and this is one of the reasons why the developer community has implemented numerous wrappers to be able to use them through programming languages.

With the aim of easing the communication with the Twitter API we have used Tweepy [12], a Python library developed for accessing it. Tweepy comes with classes of both API and *OAuthHandler* so that our only preoccupation must be creating instances of those classes, provide them with our consumer and access keys/secret and run the appropriates methods to initiate the quest. This library will handle by itself the authentication, connection, disconnection and errors that may occur during the transaction.

[2] https://developer.twitter.com/en/docs

2.4 Elasticsearch

ElasticSearch is a document-oriented database designed to store, retrieve, and manage document-oriented or semi-structured data. When using ES we store data in JSON document form and then we query them for retrieval. It is schema-less, using some defaults to index the data unless we provide mapping as per our needs. ElasticSearch uses Lucene StandardAnalyzer for indexing for automatic type guessing and for high precision [13].

Every feature of ElasticSearch is exposed as a REST API:

- **Index API:** used to document the index.
- **Get API:** used to retrieve the document.
- **Search API:** used to submit your query and get a result.
- **Put Mapping API:** used to override default choices and define the mapping.

ES works as a system of JSON pairs query/response. The basic concepts which compose it are:

- **Cluster:** a cluster is a collection of one or more servers that together hold entire data and give federated indexing and search capabilities across all servers.
- **Node:** a single server that holds some data and participates on the cluster's indexing and querying. A node can be configured to join a specific cluster by the particular cluster name.
- **Index:** the index is a collection of documents that have similar characteristics. For example, we can have an index for customer data and another one for a product information.
- **Document:** it is a collection of fields in a specific way defined in JSON format with a unique identifier called UID.
- **Shards:** we can define a shard as a subset of documents of an index. An index can be divided into many shards.

We have used ES as the persistence layer where we will store our detection results that will be visualized later.

2.5 Sefarad

Sefarad[3] is an Open-Source framework for browsing Linked Data developed by the GSI group at ETSIT-UPM. It also provides visual configuration and allows both faceted and text search. Moreover, it is HTML5-based with all the customization possibilities this involves.

Sefarad environment is divided in modules, each one focusing on one concrete task:

- **Visualization:** the main function of this module is to represent data which were processed. The representation is divided in *Dashboards* where each one allows graphic visualization of interesting data. These dashboards are divided in other components (Polymer Web Components) that globally compound the dashboard itself.
- **ElasticSearch:** represents the persistence layer of the project and stores all the amount of data needed for the visualization.

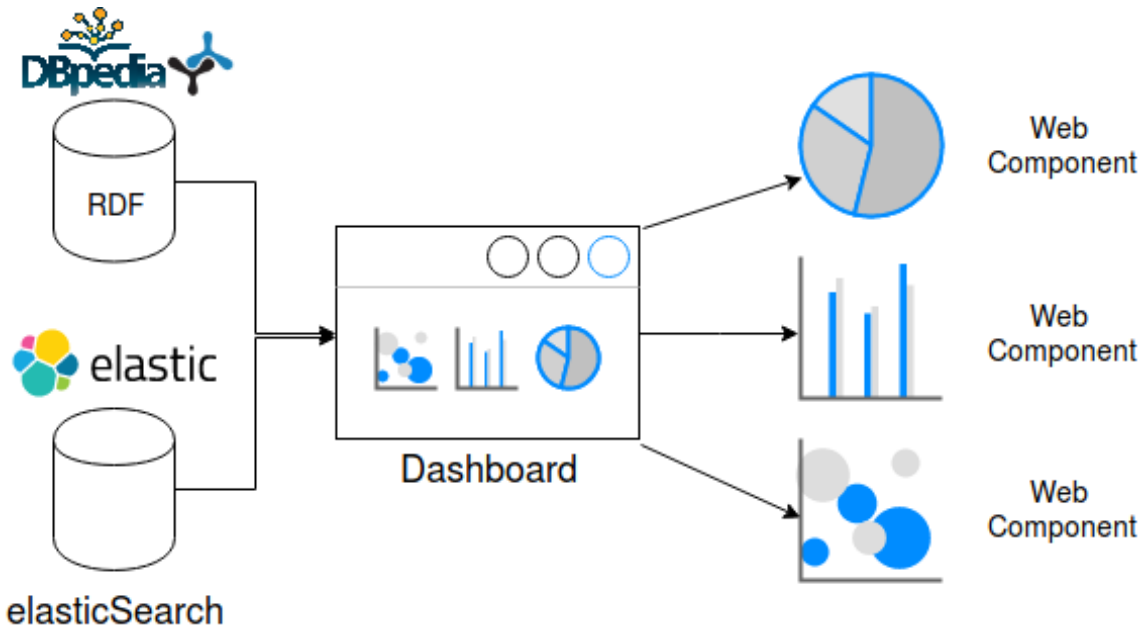


Figure 2.1: Sefarad's Arquitecture

As shown in the architecture, Serafad is also capable to retrieve semantic data from external sources, such as Fuseki or DBPedia.

[3]http://sefarad.readthedocs.io

2.6 Luigi

Luigi [14] is a Python module that helps us build complex pipelines of batch jobs. It handles dependency resolution, workflow management and visualization among other features.

It is developed by Spotify and it is used to run thousands of tasks, organized in complex dependency graphs. Luigi provides an infrastructure that powers several Spotify features including recommendations, top lists, A/B test analysis, external reports, internal dashboards and many more.

Conceptually, Luigi is similar to GNU Make where you have certain tasks and these tasks in turn may have dependencies on other tasks. This way, a task will not be able to be run until the task on which it depends have finished its performance.

It is no coincidence that we left this section for the end, as it is the one in charge of orchestrating all the process. We have used Luigi in order to achieve a correct functioning between modules and ensure that there are no failures when sequencing tasks.

CHAPTER 3

MABSED Model

In this chapter the event detection model used in this project is described. First we will introduce MABED, the approach our detector is based on. Then we will quote what limitations we have found while using this model, providing some examples where its performance was limited. Finally, we present our proposals for solving those problems. The resulting model is named MABSED (Mention-Anomaly Based Streaming Event Detection) as it has been designed for short term datasets with a duration of approximately twenty-four hours.

3.1 MABED

MABED is the event detection approach our detector is based on. Although some of the most important concepts needed for understanding this project are explained in this chapter the lecture of the original article [15] is strongly recommended. The link[1] to MABED GitHub repository is also provided at the footnote of this page.

Mention-Anomaly-Based Event Detection (MABED) is a novel statistical method that relies solely on tweets and leverages the creation frequency of dynamic links (i.e., mentions) that users insert in tweets to detect significant events and estimate the magnitude of their

[1] https://github.com/AdrienGuille/pyMABED

impact over the crowd. MABED also differs from the literature in that it dynamically estimates the period of time during which each event is discussed, rather than assuming a predefined fixed duration for all events.

This method relies solely on statistical measures computed from tweets and produces a list of events, each event being described by (I) a main word and a set of weighted related words, (II) a period of time, and (III) the magnitude of its impact over the crowd.

MABED is compound mainly by two different classes: *Corpus* and *MABED*. In the following subsections we provide a slight explanation of what each structure is used for and its main components.

3.1.1 Corpus class

The first step on MABED work flow is the processing of the data. It receives as an input a dataset of tweets in *CSV* format which at least needs to have two required fields: *date* and *text*. From these data it will create an object called *Corpus* containing important attributes such as start and end date, number of tweets, vocabulary and matrices containing mention and word frequency values which evolve over time.

We will introduce here a very important concept for the detection task which is the *time slice*. A time slice represents each part our Corpus will be split on. As its name suggest it is strongly related with time since it will have a duration given in minutes. This duration is stored in our Corpus as a parameter received by the detector named *time slice length*. This way, given a start date (d_a) and an end date (d_z) of our Corpus and a time slice length (tsl) we can easily discretize it into n time slices of equal length applying the following equation:

$$n = \lceil \frac{d_a - d_z}{tsl} \rceil \tag{3.1}$$

Similarly to Eq. 3.1, we can retrieve the time slice (ts) a tweet (t) with a given date (d_i) is located at:

$$ts_t = \lfloor \frac{d_i - d_a}{tsl} \rfloor \tag{3.2}$$

In Eq. 3.1 and Eq. 3.2, dates subtractions must be given in minutes in order to return a dimensionless unity when dividing by *time slice length*.

Corpus also contains some other very important features for the detection as the tweets vocabulary, which contains its most used words sorted by appearance frequency, and sparse matrices, which are 2-dimensional arrays used to store a value evolution over each Corpus

time slices and words. We talk in detail about how to generate and work with these matrices in Sect. 3.3.2.

3.1.2 MABED class

MABED is the class in charge of performing the detection. It receives a Corpus object as an argument and stores it in its attributes in order to have access to the processed data it contains. However this object will not only be a data storage, as MABED will also make use of some of its methods.

This class has a two-phase flow. It relies on three components: (I) the detection of events based on mention anomaly, (II) the selection of words that best describe each event, and (III) the generation of the list of the k most impactful events, where k must be provided as a parameter. The overall flow is illustrated in Fig. 3.1.

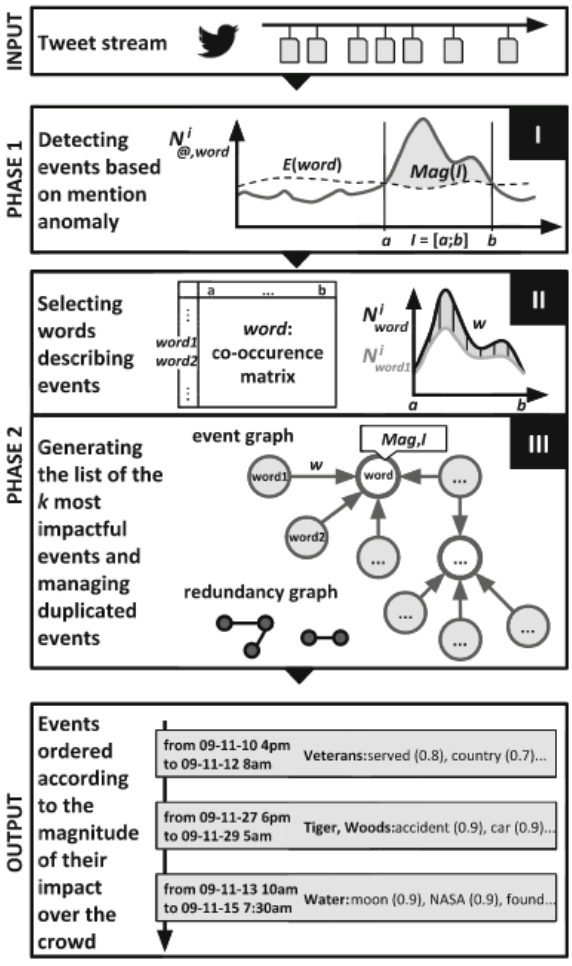


Figure 3.1: MABED's overall flow

In *Phase 1* the mention creation frequency related to each word is analyzed with the first component. The result is a list of partially defined events, in which are missing the set of related words. This list is ordered according to the impact of the events.

Then it is time for *Phase 2* which encompasses components two and three. In component two, the list is iterated through starting from the most impactful event; for each event, the second component selects a set of words that best describe it called *related words*. The selection relies on measures based on the co-occurrence and the temporal dynamics of words tweeted during an interval I. Each event processed by this component is then passed to the third component, which is responsible for storing event descriptions and managing duplicated events. Eventually, when k distinct events have been processed, this component merges duplicated events and returns the list containing the top k events.

Once these phases are over the event detection is finished and each event description is printed on console showing significant information as the event duration, main words and related words (each one coupled by its weight).

MABED also allows the user to save the computed results to a file so that their main attributes can be visualized in the browser.

3.2 Current model limitations

3.2.1 Spam importance

MABSED works analyzing tweets coming from Twitter streams [16]. These data are not previously processed so we can not assure that they are useful information. Social networks can generate a huge impact with a single publication reaching million of potential clients or viewers. Twitter is not an exception, and for this reason it is used by thousands of companies or users with the intention of gaining followers or being seen by as many users as possible.

In this context the concept of spam is born. Quoting [17], Twitter users can be exposed to spam mainly in three different ways:

- **User Timeline:** users receive content in the form of a timeline that includes tweets broadcast by each of a user's friends as well as posts that mention a timeline's owner.
- **Trending Topics:** spammers can post tweets that contain popular keywords from trending topics. Users that explore these trends will receive a feed of legitimate tweets interspersed with spam.

- **Search:** Twitter provides a tool for searching public tweets beyond popular topics. Spammers can embed popular search terms into their tweets, similar to hijacking trends and search engine optimization.

At an early stage we did not filter any content and our events were formed mainly by terms contained in spam tweets. This situation allowed us to analyze what kind of spam most conditioned the detection task:

1. Repetitive spam is one of the most common types of spam we had to deal with. It consists on posting the same content mentioning different users each time. These users are usually celebrities or official entities accounts which people tend to search.
2. There are specific accounts which dedicate to publishing information about services they offer. Publications are posted automatically by a software program or interactions with an API, so although they can not be catalogued as the previous type of spam they usually have a predefined structure in which only a few words change. Examples of this type of spam are accounts which inform about trending topics changes or even government accounts talking about traffic jams or accidents on cities.
3. The third type of spam detected is similar to the first one but it does not post the same content all along. It is a larger document split into pieces using the same final hashtags or mentions.

Last case is solved by the implementation described in Sect. 3.3.2 as it is mainly posted by a single user. However, types *1* and *2* needed a special treatment in order to be detected and processed. In Sect. 4.3.1 we propose a solution to this limitation developed when trying to use the current model.

3.2.2 Events underestimation

Original MABED impact measure algorithm bases only on mentions to perform this estimation. This is useful because mentions are considered dynamic links which represents the frequency at which users interacts. However Twitter is not a reliable source of data since users will not generate clean and ready to use information. We are talking about repetitive tweets that, although they can not be tagged as spam, they tend to originate events without actually being so. Some examples of these types of documents can be:

- **Single source of tweets:** this type covers cases where a single user conforms an event by talking about one same topic mentioning other users with the intention of reaching a larger audience.
- **Conversations between two users:** as we know Twitter is a social network where

users can publish their opinion about current topics and reply to others opinion, being this interaction between users produced through mentions. If the conversation stretches too far it could be considered as an event with interlocutors usernames as main or related words.

These are just a couple of examples of how tweets commonly found on Twitter timeline can affect our impact measure.

Moreover, there are cases where public interest events do not require the usage of mentions as they do not involve any Twitter user to be quoted in the document. We mean situations related to occurrences, accidents, disasters and broadly speaking any happening where quotable people are not engaged.

In order to illustrate this problem we have used a dataset containing 12,026 tweets collected on April 15th 2019, when the Cathedral of Notre Dame fire happened caused by an accident during a restoration work; this fire left damages valued at hundreds of millions of euros. Everyone would expect that given the dimensions and repercussion of this event it would appear the first one in our detector. In Table 3.1 top three events detected that day by original MABED approach are represented. To make it fairer, we filtered away spam from dataset before the detection task.

$e_{\#}$	**Time interval**	**Topic**
1	From: 2019-04-15 20:30 To: 2019-04-15 21:30	**díamundialdelarte, 15deabril:** enfocat(0.94), meningitis(0.94), construyendolarepública(0.94), 14deabril(0.68), ley(0.66), españa(0.62)
2	From: 2019-04-15 09:00 To: 2019-04-15 17:00	**gracias:** semana(0.61), muchísimas(0.64)
3	From: 2019-04-15 19:30 To: 2019-04-16 01:00	**notre:** dame(0.98), catedral(0.92), incendio(0.76), parís(0.91), historia(0.73), humanidad(0.77), llamas(0.74), paris(0.64), tristeza(0.65)

Table 3.1: April 15th 2019 top 3 events

Surprisingly, Notre Dame event was considered as the third event with highest impact rate. This is one of the cases we mentioned above: #1 event was entirely formed by a single user talking about a same topic by himself. Each tweet this account posted contained mentions to other Twitter users trying to reach a larger audience. Main and related words are most hashtags he used in all his posts.

Moreover and as pointed before, Notre Dame event does not involves anything which can be mentioned since it is a monument. This situation implies that its real impact can not be correctly measured using this approach.

For these reasons we conclude that it is not appropriate to base our impact measures only on the number of mentions caused by an event. In Sect. 3.3.2 we propose an alternative for this issue.

3.2.3 Unmerged events

Quoting Sect. 3.1.2, MABED *Phase 2* process is probably the most important one, but between its components we would like to emphasize on the third one. This component function is merging events considered to be actually the same one. This is a very important task as if events are not correctly tagged as redundant a same event will be returned several times maybe causing other relevant ones not appearing on the list.

Arguably MABED only covers a very small redundancy spectrum as it only checks what we have defined as *directly redundant*. We will illustrate this explanation with a code extract and some graphs.

Listing 3.1: Directly redundant definition

```
def directly_redundant(self, event):
    main_word = event[2]

    if self.event_graph.has_node(main_word):
        for related_word, weight in event[3]:
            if self.event_graph.has_edge(main_word, related_word):
                interval_0 = self.event_graph.node[related_word]['interval']
                interval_1 = event[1]
                if st.overlap_coefficient(interval_0, interval_1) > self.
                    sigma:
                    self.redundancy_graph.add_node(main_word, description=
                        event)
                    self.redundancy_graph.add_edge(main_word, related_word)

    return False
```

According to the previous code, the definition of directly redundant only covers situations in which an event has as its main word one of the related words of another event already present in the event graph. If this happens we will check if the candidate event

has among its related words the main word of that event already present in the graph. In case this word is contained we will get both events interval duration and check their `overlap_coefficient`, defined in the following extract:

Listing 3.2: Overlap coefficient

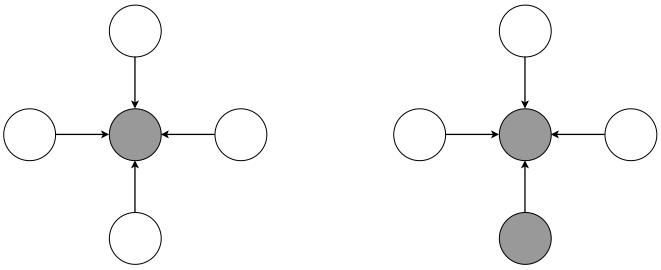
```
def overlap_coefficient(interval_0, interval_1):
    intersection_cardinality = float(min(interval_0[1], interval_1[1]) - max
        (interval_0[0], interval_1[0]))
    smallest_interval_cardinality = float(min(interval_0[1] - interval_0[0],
         interval_1[1] - interval_1[0]))

    if smallest_interval_cardinality == 0.0:
        smallest_interval_cardinality = 1.0

    return float(intersection_cardinality / smallest_interval_cardinality)
```

This method will return a coefficient representing how much both intervals coincide over time. We have added a security precaution on `smallest_interval_cardinalty` as if one event starts and finishes in the same time slice a *ZeroDivisionError* will be raised. Finally, in case overlap coefficient is greater than a threshold set to 0.6 (see [15] for more information), event will be considered as redundant and added to our redundancy graph.

The process we just explained is illustrated in Fig. 3.2, showing the content of the event graph. White nodes represents related words while grey ones are main words.



(a) MABED - Original graph

(b) MABED - Redundancy detected

Figure 3.2: MABED Directly redundant process

The above redundancy algorithm has a limited functionality since it just consider an ideal scenario where events contains the necessary information to detect relations between them. However as the number of related words is limited to *p=10* with a weight threshold of $\theta = 0.6$ [15] the scenario is far from being the desired one. High impact events can

surpass the related word limit having more than just 10 words with a higher weight than the threshold. This may cause that the specific word searched by `directly_redundant` method is missing on the list.

In order to justify this limitation we have used a dataset of 19,850 tweets collected on April 28th 2019, day on which Spain's general elections were held. This is a perfect situation in which testing how the redundancy algorithm performs since elections are a national interest event and it will be formed by very different terms due to the amount of people talking about it. To show redundancy algorithm limitations we will detect top *k=3* events in two different models:

- **MABED original approach:** using the collected dataset with spam already filtered away.
- **Final MABSED model:** applying spam filter and lemmatizer modules and impact measure improvement. However, we have used MABED redundancy algorithm in order to prove that this limitation is not solved by any other modification applied to original model apart of the one we will propose later.

Tables 3.2 and 3.3 shows detection results for each model used.

In both tables events detected are quite concise having the biggest one two main words. This means that only two events were considered to be redundant and were merged.

Having a look at Table 3.2 third event with the highest impact has nothing to do with elections, representing something related to a marathon and Adidas. By checking that

$e_{\#}$	**Time interval**	**Topic**
1	From: 2019-04-28 21:30 To: 2019-04-29 02:30	**sanchezcastejon, psoe:** conriverano(0.95), rivera(0.94), 28a(0.84), gobierno(0.81), eleccionesgenerales28a(0.78), albert_rivera(0.77), ciudadanos(0.76), pp(0.74), escaños(0.70), voto(0.67)
2	From: 2019-04-28 07:30 To: 2019-04-28 17:30	**votar:** voto(0.68), madrid(0.67), eleccionesgenerales28a(0.63)
3	From: 2019-04-28 07:30 To: 2019-04-28 17:31	**gracias:** rnrmadmaraton(0.80), ad_mapoma(0.76), adidas_es(0.75), adidasrunning(0.75), crack(0.72), madrid(0.61), abrazo(0.61)

Table 3.2: April 28th 2019 top 3 events, MABED approach

$e_{\#}$	Time interval	Topic
1	From: 2019-04-28 07:30 To: 2019-04-28 18:30	**votar:** vox(0.72), españa(0.71), gente(0.68), 28a(0.67), madrid(0.67), colegio(0.65), eleccionesgenerales28a(0.65), salir(0.62), derecho(0.61)
2	From: 2019-04-28 21:00 To: 2019-04-29 02:30	**psoe, pp:** pactar(0.88), vox(0.87), sanchezcastejon(0.77), ciudadano(0.76), escaño(0.70), 28a(0.70), voto(0.69), ganar(0.67), resultado(0.65), derecha(0.63)
3	From: 2019-04-28 19:30 To: 2019-04-29 01:30	**eleccionesl6:** vox(0.77), españa(0.74), eleccionesrtve(0.73), derecha(0.71), 28a(0.69), eleccionesgenerales(0.67), pp(0.66), noche(0.66), eleccionesgenerales28a(0.62))

Table 3.3: April 28th 2019 top 3 events, MABSED using MABED redundancy algorithm

event's tweets we found out that it was formed by almost a single account as exposed in 3.2.2. However it does not appear in Table 3.3 as MABSED model implements the improvement exposed in 3.3.2.

Moreover, third event of Table 3.3 is clearly part of the same event represented by e_2. They were not merged because the high repercussion of the event generated a list of related words long enough so that *eleccionesl6* did not appear between its top *k=10* words. For this reason `directly redundant` algorithm did not evaluate them as the same event.

Considering what has been stated in this section we conclude that the redundancy algorithm is not as accurate as it should be. High impact events as the one we exposed are not analyzed correctly by this approach generating not concise events. In Sect. 3.3.3 we propose an additional algorithm to check weather two events are redundant or not.

3.3 Implemented improvements

This section describes what changes have been applied to the model in order to achieve a better performance under our project requirements.

3.3.1 Corpus class

Corpus class changes are indeed minimum and can be detailed in a few lines since both models receives the same information and process it the same way.

The first change we will talk about is the **input** as it is the first step our Corpus will take. MABED is designed to read all the input data from a single dataset. This makes sense since the model works with large datasets already preprocessed; the set of tweets MABED was tested on contained more than a million of documents. However, and attending to the way our project is raised, this type of input does not make any sense in our case as our Streamer will be generating files on an ongoing basis. This makes it necessary that our model can read data from a changing input with the least memory waste possible. That way our Corpus input is defined as a list of files passed as a parameter by the orchestrator.

Another change we have performed is the **tokenization** method. By observing the vocabulary generated by the Corpus we realized that some words appearance were not being considered because writing errors were not handled properly. We are talking about cases where spaces between words were missing and two words were written together with a semicolon, a parenthesis or any other punctuation sign between them. Our solution was to detect the punctuation signs prone to separate phrases and be accompanied by a missing space (as they could be colons, dots, parentheses or quotes) and replace them by a space instead of just clearing them. The result was just as expected since the frequency of most of the words was increased and the number of non-existing words generated by the concatenation of two of them with a punctuation point was drastically reduced.

Finally, the last change effected was to add a new **sparse matrix** to the Corpus. This change is closely related with Sect. 3.3.2 and the matrix is used by MABSED class, so although we are not going to explain it here it has been quoted with the intention of letting the reader know where it is generated.

3.3.2 Impact measure

In this section we will talk about one of the main and more significant changes applied to the model. It will directly affect the impact estimated for each event and, therefore, the order of importance events will be sorted in.

One of the main magnitudes we can base an event impact on is the **number of users** talking about it; after all, a significant occurrence does not necessarily have to involve mentions but it objectively has to be discussed by a large number of people.

In Sect. 3.3.1 we mentioned the existence of a new sparse matrix generated in Corpus class. This matrix measures the number of users talking about each word contained in the Corpus vocabulary at any given time and is represented at Eq. 3.3.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ u_{21} & u_{22} & u_{13} & \cdots & u_{2n} \\ u_{31} & u_{32} & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & u_{n3} & \cdots & u_{nn} \end{bmatrix} \tag{3.3}$$

Each value contained in the matrix represents *number of users involved*. Each column stands for a word of the vocabulary sorted with respect to their frequency, being the one in the left the most used one. On the other hand matrix rows represent time slices, where row one is the time slice elapsing from Corpus start date to a *time slice length* later. With this in mind, u_{11} represents the number of users talking about the Corpus most used word during its first time slice. In conclusion, this matrix contains the *users involved* evolution of each word over time.

As we have said before each word w of the matrix is indeed a vector containing number of users. Our objective now is to convert this vector to a new one containing *number of users* anomalies instead of absolutes values:

$$a_w = \begin{bmatrix} a_{w1} & a_{w2} & a_{w3} & \dots & a_{wi} & \dots & a_{wn} \end{bmatrix} \tag{3.4}$$

where a_{wi} stands for the anomaly computed during the ith-time slice for its corresponding word. The anomaly estimation method is shown in the following Python fragment.

Listing 3.3: User Anomaly estimation

```
def user_anomaly(self, time_slice, observation, total_user_freq):
    expectation = float(self.corpus.tweet_count[time_slice]) * (float(
        total_user_freq)/(float(self.corpus.size)))

    return observation - expectation
```

Anomalies are defined as simple subtractions between an expected value and the observed one at a given time slice i.

$$a_{wi} = obs_{wi} - exp_{wi} \tag{3.5}$$

Expression obs_{wi} corresponds to the value contained in the matrix defined at Eq. 3.3 for the ith-time slice as they are actually previously computed values.

$$obs_{wi} = u_{wi} \tag{3.6}$$

In order to define exp_{wi} we first need to introduce the concept of a word total user frequency (tuf_w) defined as the sum of each value of the sparse matrix (u) for a given word (w) along each time slice (i):

$$tuf_w = \sum_{i=1}^{n} u_{wi} \tag{3.7}$$

This represents the total number of users who have used that word during the whole time spectrum. Thereupon, we compute the expectation (exp_{wi}) multiplying this value by the percentage of tweets contained in the time slice (i) we are calculating at with respect to the Corpus (C) total of tweets. This last sentence is described by Eq. 3.8.

$$exp_{wi} = tuf_w \cdot \frac{|T_i|}{|T_C|} \tag{3.8}$$

where T refers to a tweets set and $|T|$ represents the set cardinality.

Applying this computations on each time slice we would finally get the vector defined at Eq. 3.4, containing the anomaly estimated with respect to the number of users using that word.

Finally, total anomaly is now estimated as the sum of both *users* and *mention* vectors, this last one calculated following the same steps we have just explained.

At a given moment we raised the possibility of adding a third vector to the equation that would measure the anomaly with respect to the appearance frequency of each word. However, an increase in the use of a certain word is directly related to the number of users using that word as these are the ones generating it. We can define the frequency of a word in a given time slice as:

$$f(u) = u + k \tag{3.9}$$

As we can see the frequency $f(u)$ is given according to a number of users that have used it (u) plus a constant (k) which represents the number of extra tweets using that same word posted by those users.

If we added that third vector we would be giving a greater weight to the user-frequency ratio and leaving the impact generated by the number of mentions in the background. Moreover, word frequency does not handle the case where it is a single user the one talking about a topic.

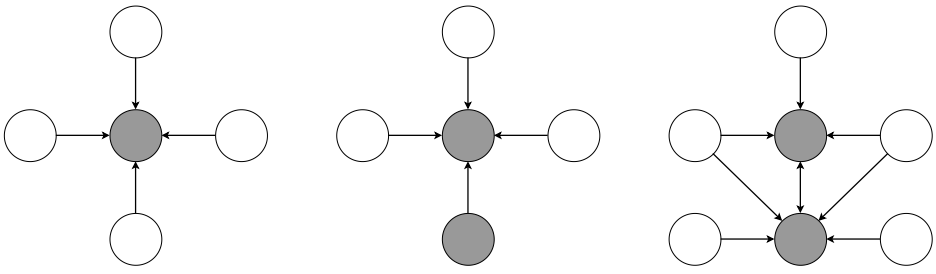
3.3.3 Redundancy algorithm

Redundancy algorithm is another quite significant change applied to the model. In this section we will detail how redundancy detection task is handled on MABED and what modifications has been applied to it [18].

As said in Sect. 2.1.4, to detect events redundancy we will use the Python library NetworkX. Before explaining the algorithm we will introduce the scenario. Events are defined by a main word, a set of related words, an impact and a duration interval. In our project, nodes are words, both main and related ones. This nodes are linked by edges containing the weight each related word has with its main word. In order to store all the process nodes and edges we will use two types of graphs:

- **Event graph:** which is actually a directed graph, what means that edges direction matters. Nodes will be connected only from a source node to a destiny one and not the other way around.
- **Redundancy graph:** where we will store events considered as redundant. It is an undirected graph so edges direction does not matter, we only want to establish relations between two nodes regardless of the direction.

As a first change we edited MABED directly redundant definition introduced in Sect. 3.2.3. We went a little further and added one last step where we append to the event graph all the related words of the new redundant event. This way we will have a more extensive description of our events in order to detect further redundancies. Fig. 3.3 shows how event graph is affected by this change.



(a) MABED - Original graph

(b) MABED - Redundancy detected

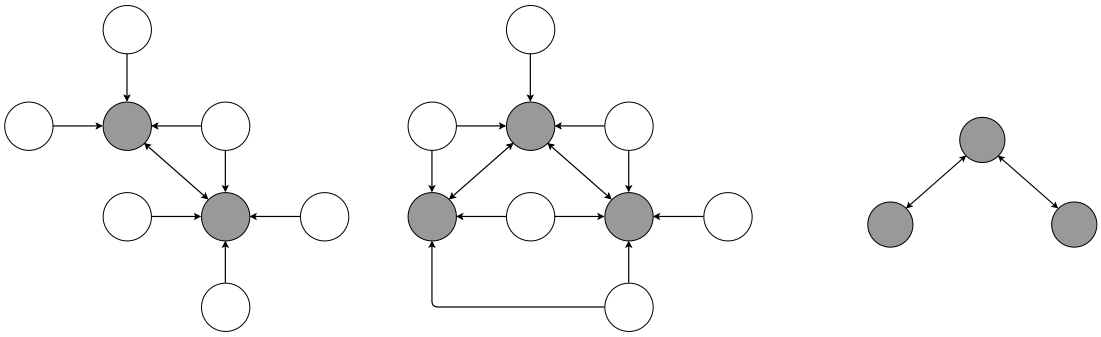
(c) MABSED - Redundancy detected

Figure 3.3: Directly redundant process comparison

Additional algorithm we propose has been named as *indirectly redundant*. This method overall reasoning is: if two events have a wide set of related words in common and they happen in the same time interval, they are likely to represent the same event.

To be able to translate that idea into code we had to make use of the redundancy graph. We will introduce the concept of ***component*** of a graph as a set of nodes linked by an edge. The key idea for this method is: each component of our redundancy graph will be seen as a single node. This way, when we receive a new event we will check the number of related words this event has in common with any of the components stored in the redundancy graph. If the number of coincidences surpass an absolutely set threshold, we will check then both events overlap coefficient. The rest of the process is the same as described in *directly redundant* method, updating both graphs in case it gets considered as redundant.

This reasoning will become clearer after seeing it graphically in Fig. 3.4, showing the process over a indirectly redundancy detection.



(a) Original event graph

(b) Event graph - Redundancy detected

(c) Updated redundancy graph

Figure 3.4: Indirectly redundant process

In our project we have set the number of coincidences required to three. As it can be seen in the process figures, the redundant event had two related words in common with one of the events already stored in the graph and another one with the other event. However, as nodes of a same component are seen as a unit, the event was considered redundant and the graphs were updated.

3.3.4 Output

The last change applied to the model is how to handle the detection output. By default MABED printed the detection results on console showing significant information as the event duration, main words and related words (each one coupled by its weight). As we want this results to be portrayed at the visualization module we need them to be saved on a file with the format our persistence system expects.

We save data in two different files, one for each index in ElasticSearch. In the following lines we will describe the content of each one.

The first file we are saving is the events description. As we have said this information is later represented in our Dashboard so we have tried to store any useful information the user would like to know. Each event is described by the following fields:

- **ID:** sorts the events according to their importance.
- **Impact:** calculated as described in Sect. 3.3.2. Contains three different values: total, user and mention impact.
- **Main words:** set of words which describes the event. If more than one word appears it means that originally they were different events considered as redundant.
- **Related words:** those which most co-occurs with the main words.
- **Duration:** lapse of time while the event was discussed.
- **Anomaly:** list containing in turn three other lists, each of which represents the total, user and mention anomalies computed as introduced in Sect. 3.3.2 for each time slice.

The second files stores all tweets related with each event. We do not save only the document body, as tweet metadata contains also interesting information we want to represent:

- **Event ID:** identifier of the event which each tweet represents.
- **Tweet ID:** in case the user wants to see the original tweet in Twitter.
- **Text:** content of the document.
- **Coordinates:** in case they are public. They will be represented in a heat map.

CHAPTER 4

Event Detection Monitoring Service

In this chapter, we cover the design phase of this project, as well as implementation details involving its architecture. Firstly, we present an overview of the project, divided into several modules; this is intended to offer the reader a general view of this project architecture. After that, we present each module separately and in much more depth.

4.1 Architecture

This section will introduce the global architecture with the intention of providing the reader with an overall vision of the project. Fig. 4.1 shows this data work flow.

Luigi, as described in Sect. 2.6, will be in charge of orchestrating the full process. By using Luigi we can assure that a task will not be able to be run until the task on which it depends have finished its performance.

In the following sections we will describe deeply each of the modules involved in the system.

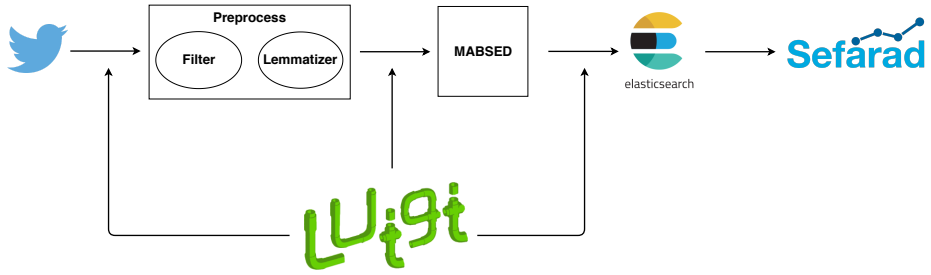


Figure 4.1: Architecture of the monitoring system

4.2 Capture subsystem

This subsystem is responsible for collecting all tweets posted on the social network located in Madrid and surroundings. In Sect. 2.3 we proposed two different API options in order to perform this capture. After doing some approaches with both of them, the one returning better results was the **Streaming API** so finally we decided to use it despite its tweet volume limit. Remaining received tweets are a percentage of the total and will still represents the social network topicality, so they will allow us to detect the most impacting events by analyzing them.

Our streamer has been developed using a `StreamListener` class implementation which will save tweets in half-hour bands. This means that a tweet received at '17:28' will be stored in a file named '17:00', and therefore other tweet received at '17:31' will be saved in a '17:30' file. This standard has been implemented because Luigi can not depend on when we start the script. Normalizing file names was the easiest and more efficient way of achieving a properly functioning pipeline which received the correct files.

Streaming API expects different filters in order to know what tweets it needs to retrieve. In our particular case we have not set any term filter because events can be composed by any word or even by usernames or hashtags. However we have used two other filters:

- **Language:** we are interested only in Spanish-written tweets. Considering tweets written in other languages would just mud our detection results.
- **Location:** as we have said before, our event detector is focused on the city of Madrid so we will need to specify the area whose tweets we want to receive. For getting the suitable coordinates we used the online tool BoundingBox[1] and selected the desired area, setting the coordinates to CSV Raw format.

Finally, Streaming API returns quite a lot data and metadata about each tweet. Saving

[1]https://boundingbox.klokantech.com/

all this information would be a waste of memory as we do not need all of it for our project. In particular we are saving each tweet author ID, date and hour of posting, coordinates and the tweet body. This data is stored in a CSV (Comma Separated Values) file as it is the most compact file extension for storing data.

4.3 Preprocess subsystem

Next module is the preprocess subsystem. Data we are going to analyze comes from twitter streams, what means it is raw data not subjected to any previous selection. The function of this module is to prepare these data in a way they can be properly examined and ease the detection task to MABSED.

Data preprocessing is an integral step in Machine Learning as the quality of data and the useful information that can be derived from it directly affects our model results; therefore, it is extremely important that we preprocess our data before feeding it into our model.

In the following subsection we will describe the two modules which composes in turn the preprocess subsystem: the spam filter and the lemmatizer.

4.3.1 Filter

As introduced in Sect. 3.2.1 filtering away spam is a quite important task for our project. We finally decided to separate this process from the detection one and include it in a previous module since it will allow us to gradually preprocess data as it comes instead of doing it all at once.

Spam we want to detect is characterized by having almost the same document body, so for this reason we decided to base the filter decision on a text similarity basis. In the following lines all the needed concepts required to understand our approach [19, 20] are described.

In order to compute similarity between texts we need a *word2vec* process, which consists on modeling the tweet text as a vector. For this part we decided to use a TF-IDF vectorizer [21] as it will take into consideration not only if a word appears on a document but also how many times it does it.

TF-IDF stands for "Term Frequency — Inverse Document Frequency". First, we will learn what this term means mathematically.

Term Frequency (tf) gives us the frequency of the word in each document in the corpus. It is the ratio of number of times the word appears in a document compared to the total number of words in that document. It increases as the number of occurrences of that word within the document increases. Each document has its own tf. Its formula is represented in Eq. 4.1

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}} \tag{4.1}$$

Inverse Document Frequency (idf) is used to calculate the weight of rare words across all documents in the corpus. The words that occur rarely in the corpus have a high idf score. It is given by Eq. 4.2.

$$idf(w) = \log\left(\frac{N}{df_t}\right) \tag{4.2}$$

Combining these two we come up with the **TF-IDF score** (w) for a word in a document in the corpus. In Eq. 4.3 we can see that it is the product of Eq. 4.1 and Eq. 4.2:

$$w_{i,j} = tf_{i,j} \cdot \log\left(\frac{N}{df_i}\right) \tag{4.3}$$

Now we will introduce the concept of **cosine similarity** [22, 23]. It is used to measure how similar documents are irrespective of their size. Mathematically, it measures the cosine of the angle between two vectors projected in a multi-dimensional space. The cosine similarity is advantageous because even if the two similar documents are far apart by the Euclidean distance (due to the size of the document), there are chances that they may still be oriented close together. The smaller the angle, higher the cosine similarity. Eq. 4.4 shows what has just been explained in a mathematically way.

$$similarity = cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4.4}$$

where A_i and B_i are components of vector A and B respectively and $\|.\|$ is the vector norm.

As a similarity metric, cosine similarity differs from the number of common words because plotted on a multi-dimensional space where each dimension corresponds to a word in the document, the cosine similarity captures the orientation (angle) of the documents and not the magnitude. This last sentence means that two documents can be considered to be similar despite of having different lengths.

Now that the concepts used by the spam filter has been explained we can describe its functioning. As a vectorizer we will use `TfidfVectorizer` provided by Scikit-Learn as

mentioned in Sect. 2.1.3. Before we can use it we have to fit it providing it with a list of sentences containing the words it will later vectorize; in our case we train it using the same tweets we want to filter. Once it is trained next step will be vectorize each document; we have created a buffer where we will store tweets considered as not spam. Now using Scikit-Learn `CosineSimilarity` we will compare the similarity of that new vector with each one of the vectors stored in our buffer returning a list of similarities with the same length as the number of vectors in the buffer. As we are only interested in the greatest similarity we will use `.max()` method to get it. Then, if the top similarity value exceeds a set threshold we will consider that tweet as spam and ignore it. Otherwise we will add it to the buffer in case of a similar tweet appears in the future and will be passed as an input to our lemmatizer. Filter expects as input a list of tweets and generates as output a new list of cleaned tweets.

Buffer size is a problem as our project will be receiving tweets continuously and at some point the operation of checking the similarity between the received tweet and the ones in the buffer would become unmanageable. For this reason we decided to clean the buffer and fit the vectorizer each time a thirty minutes tweet file is received. This way we would miss at most a copy of each type of spam per file without affecting our detection results as they are based on spontaneous rises of frequency word usage.

4.3.2 Lemmatizer

The lemmatizer is the second and last module which forms the preprocess subsystem. As said before, preprocessing the data will increase drastically our model results as leading to a better detection and a reduction on the number of mistakes made [24]. Thereupon we will describe some of most common text manipulation methods employed on Machine Learning:

- **Tokenization:** which consists on converting sentences to words. As we said in 3.3.1 our model implements its own tokenization method which is able to mend certain typing errors as missing spaces between words.
- **Stemming:** it is a text normalization method. In linguistics, stemming is the process of eliminating affixes (suffixed, prefixes, infixes, circumfixes) from a word in order to obtain a word stem. Stems may be a root or morphologically complex.
- **Lemmatization:** also a normalization method and related to stemming, lemmatization is able to capture canonical forms based on a word's lemma. Plainly put, it could be said that it provides us with the 'dictionary form' of a word. This method not only takes into account the word, but also its semantic sense in the phrase.

Of the last two methods it should be easy to see why the implementation of a stemmer would be the less difficult feat. However using stems as our events descriptions would lead to incomprehensible words that actually would not provide the reader any useful information. Moreover, to retrieve the original word from a stem is difficult as stemming is a lossy process having many retrieval possibilities. These were the reasons why we decided to implement a lemmatizer instead of a stemmer for our text normalization method.

As indicated in Sect. 2.2 for this purpose we have used NLP Cube, a Natural Language Processing Pipeline. We do not only use its lemmatization functionality but also its tagger, as words with different tags will result in different lemmas. Moreover we set manually DyNet configuration in order to exploit the four cores of the computer this project was developed on. This framework reports to have a 98% success rate on lemmatizing Spanish sentences.

Lemmatization module receives as input the list of tweets provided by the spam filter and add to each tweet a new field with its lemmatized text. The list generated is then handled by Luigi which is in charge of saving them to a new CSV file, which will be in turn the input sent to MABSED.

4.4 Persistence subsystem

Persistence subsystem is responsible for the storage and subsequent search of the output generated by MABSED. As said in Sect. 2.4 the technology that we have chosen to perform this function is Elasticsearch. We are storing two different files, one containing events information and another one storing each event tweets; for this reason we need to manage two different indices in ES, one for each of these files. So that data can be correctly interpreted by ES files contain information stored in JSON format with one index entry per line.

Moreover ElasticSearch allows us to create **aggregations** which can be composed in order to build complex summaries of the data. An aggregation can be seen as a unit-of-work that builds analytic information over a set of documents. The context of the execution defines what this document set is. There are many different types of aggregations, each with its own purpose and output. In our project we have use them as a way of easily creating groups of documents, i.e. count how many tweets compose each event.

Detection results are uploaded to each index by a Luigi task which implements the class `CopyToIndex`. This task receives in its `requires` method the data to be stored and the

index in which doing it as a parameter. However, as the detection task returns a list of two files we had to create an intermediate task called `Mediator` which receives as a parameter which file is needed and returns it as an output.

4.5 Visualization subsystem

The last module of our system and the one users will interact with is the visualization subsystem. Quoting Sect. 2.5, our dashboard is implemented from a Sefarad image and gets the data to represent from the persistence subsystem we just explained. A dashboard is composed of a series of widgets developed through the JavaScript Polymer library, each covering a different functionality. We will later explain each one deeply.

Our dashboard must be able to perform API REST requests to ES so that the desired content is returned; specifically these requests should be made to Search API. As we store data in two different indices we need to perform two requests, one per index. Responses are used by widgets to represent the data they contain.

Using Elascticsearch's ability to concatenate aggregations, widgets allow the user to filter and view only the content that meets certain characteristics specified by him. For example, our dashboard will be able to show only tweets related with one of the events or filter them by a specific term. These possibilities make our visualization subsystem a powerful tool when searching for specific data.

Widgets employed to create this dashboard will be defined in the following subsections. We will introduce them according to where they are placed in the page. An overall view of the full dashboard can bee seen at Fig. 4.7.

4.5.1 Material search

The first component users finds when entering the dashboard is the *material search* bar. This widget allows the user to introduce any term query he wants to search for. The other widgets will detect the set query and will change their representation according to it. Fig. 4.2 shows its visualization.

Figure 4.2: Material search visualization

4.5.2 Podium chart

In order to display our events we needed a widget which allowed us to represent elements in a way which made it clear that some were more important than others. Moreover, each event is described by a set of fields so we also wanted this widget to show more information according to the user's interests. As we could not find any component which meet these requirements we proceeded to develop it ourselves.

This widget has been developed by us from scratch using different technologies as HTML, CSS, JavaScript and Polymer. For the graphic part we were inspired by a podium as it would allow us to place events in different steps according to their impact. On top of each step there are trophies each with a different colour according to the position obtained in a competition. Events will be distinguished by their main words, which will be represented above the trophy pretending to be its engraving. Fig. 4.3 shows what we have just explained.

Figure 4.3: Podium Chart view

In case a user wants to know more about one event he can simply click on the corresponding trophy and a modal window will spawn showing that event information. This window has several sections as the event impact (total, mention and user one), the time interval in which the event occurred and the list of main and related words describing it. Also an icon is displayed on the top remembering which event was clicked on. When the user wants to close the modal he can just use the 'Close' button or click anywhere outside it. This functionality can be seen in Fig. 4.4.

This widget is in part customizable through its input parameters. These are:

- **data:** object containing the data received from ElasticSearch which will be represented in the widget.
- **height:** podium container height in pixels. Steps and trophies will rescale according to it.
- **stylebg:** podium background color. Changing this will affect steps, modal header

Figure 4.4: Podium Chart - Modal window

and button colours.

- **nwords:** maximum number of words showing on each trophy engraving.

4.5.3 Number chart

This simple component allows the user to quickly know the number of elements a group is formed by. As said in Sect. 4.4 we used ElasticSearch aggregations in order to get the count of how many tweets compose each event. If the user clicks on one of them the rest of the widgets will show only data related with the event which was selected. Fig. 4.5 shows how we have placed each widget, setting its background color and image according to the event they represent.

4.5.4 Tweet chart

By using this component we developed three different widgets showing each event tweets. They are placed just below its corresponding number chart, which as a whole let the user quickly know how many tweets each event is composed by and let him scroll and navigate on those tweets, as represented in Fig. 4.5.

We implemented some changes to this component. Each widget has an aggregation input to let it know what group of data it has to display. Moreover we added a new functionality:

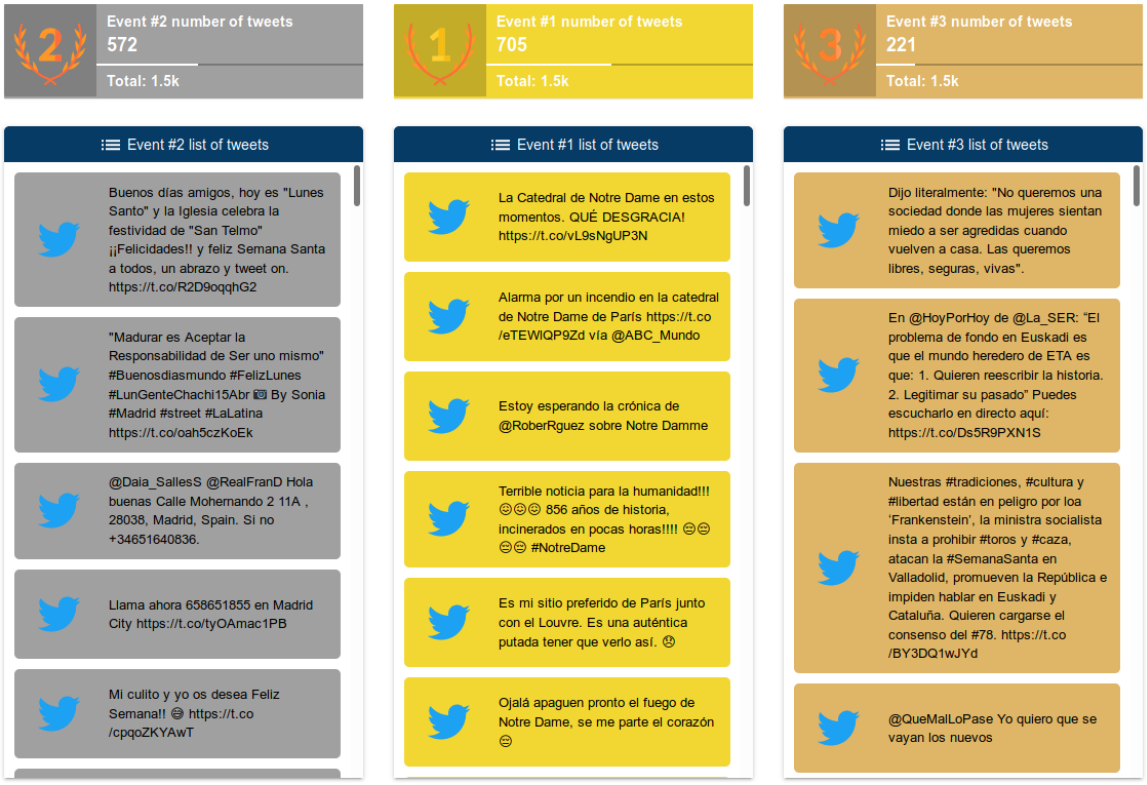


Figure 4.5: Tweet informing widgets

if the user clicks on any of the tweets a new browser window will open showing it in Twitter. To achieve this we made use of the tweet ID previously collected by the Streamer.

4.5.5 Google chart

Google Charts is a pure JavaScript based charting library meant to enhance web applications by adding interactive capabilities. Google Charts provides a wide variety of charts going from line charts or spline charts to area charts or pie charts. Google chart component is a Polymer adaptation of the plain JavaScript version which will allow us to represent the information stored in our ES container.

This widget is used for displaying the evolution of events over time, represented on a Google Line Chart. We selected this type of graphic because data is a linear succession, where each value must be seen with respect to the previous and the next one. Data to be represented come from the anomalies vector introduced in Sect. 3.3.2. Although *curve lines* option is available for this types of graph we decided not to use it since our chart's peaks allow us to know the maximums and minimums exact value.

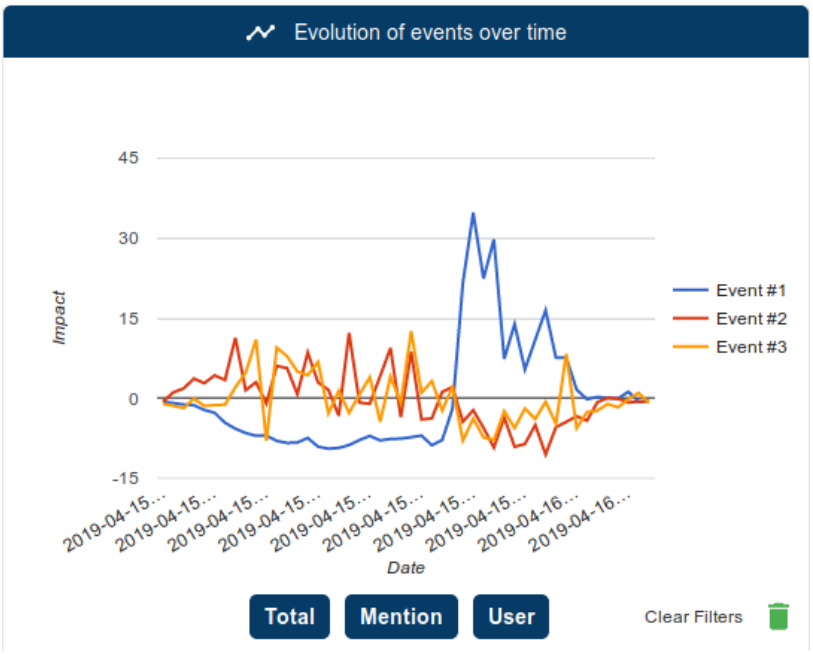


Figure 4.6: Google chart widget

Chart's horizontal axis represents dates, shown in intervals of thirty minutes, while vertical one is for the impact. By seeing Fig. 4.6, users can get a better idea of when an event arises and how long it has lasted. Moreover we have implemented three different buttons, each one allowing the user to see each event impact according to the number of mentions, number of users talking about it or the total impact.

4.5.6 Happymap

The last widget of our visualization subsystem has been built using a component named Happymap. It is a heat map which shows the location from which tweets were posted, marked as a spot in the map. It can be of different colours depending on the number of nearby posted tweets in that area, going from navy blue to isolated tweets to a green colour for locations where a large number of tweets were posted.

In order to represent a point this widget needs to be provided with its coordinates, given in latitude-longitude format. A heat map example appears in Fig. 4.7

Figure 4.7: Overall dashboard view

CHAPTER 5

Case study

In this chapter we are going to provide use cases which demonstrate why changes introduced in Sect. 3.3 are actually improvements. We will first approach each change by separate, and finally provide an overall comparison between the original model and ours in Sect. 5.4.

5.1 Filter behaviour

This first section evaluates the similarity threshold between two documents established so that they get considered as spam. We are talking about the concept introduced in Sect. 4.3.1.

For this evaluation we will need to have in mind the different types of spam we want to filter numerated in Sect. 3.2.1. Initially we set this threshold to *0.7* in order to start with the analysis as it is at first sight a prudent value. We were able to check that although spam type *1* (full document's repetition) was completely filtered, type *2* (structures repetitions) were not detected correctly. We needed to find a value which allowed us filtering both types of spam without affecting tweets which actually were not spam.

After doing some more approaches we finally tried using *0.5* as a value. This was a more conflictive option as we were not sure if it will introduce too many classifying errors. To

settle any doubts we used our larger 24-hour dataset, formed by more than 19,000 tweets. From it and using our filter we obtained a CSV file which contained tweets classified as spam. We repeated this process with both 0.5 and 0.7 values.

In order to check its correct functioning we manually searched and label in that dataset a total of 100 tweets which we know they were actually spam. Those tweets were selected impartially: 50 of them were type *1* plain repetitions (named as T_1); 30 were structures moderately easy to detect (named as T_2) and the other 20 were structures considered as difficult to detect by a simple text similarity filter (named as T_3). Our last step was looking for labeled tweets in each filter output file. Results of that search are shown in Table 5.1.

Threshold	**Total**	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$
0.7	56	50	6	0
0.5	82	50	27	5

Table 5.1: Filtered spam obtained for each threshold

In the light of the results displayed by this table we can conclude that the best value for this threshold should be **0.5** as it fits better to the objectives we will use our filter for. Moreover we checked its spam file looking for possible classifying errors. Only questionable decisions were those on which small size tweets were considered as spam. We are talking about posts saying 'Good morning', 'Good night' or saying 'Thank you' to their timeline. However we consider that filtering this type of content does not deprive our detector of any relevant information, so we concluded that setting our threshold to the agreed value will not affect negatively the filtering task.

5.2 Impact analysis

In this section results of the new impact measure algorithm exposed in Sect. 3.3.2 are shown. As point of reference we will use Table 3.1 regarding to Notre Dame dataset used in Sect. 3.2.2.

We analyzed same dataset but this time using our MABSED approach. Detection results are shown in Table 5.2.

As can be seen, Notre Dame event has now been detected as the most impacting one of that day. It has received an impact measure of 182 points of which 153 were due to the number of users talking about it. By contrast if we only had taken into account number

$e_{\#}$	Time interval	Topic
1	From: 2019-04-15 19:30 To: 2019-04-16 03:30	**historia, notredame, catedral, notre, dame, incendio, parís:** llama(0.82), siglo(0.79), paris(0.79), terrible(0.78), humanidad(0.77), arte(0.76), triste(0.73), arder(0.72), tristeza(0.66), año(0.66)
2	From: 2019-04-15 05:30 To: 2019-04-15 20:00	**semana, madrid:** comunidad(0.70), santo(0.69), feliz(0.68), of(0.67), lunes(0.66), community(0.64), empezar(0.64), spain(0.63)
3	From: 2019-04-15 08:30 To: 2019-04-15 19:00	**querer:** derecho(0.71), madrid(0.65), españa(0.63)

Table 5.2: April 15th 2019 top 3 events, MABSED approach

of mentions it would have just received a punctuation of 29 points. Its number of main terms has raised to seven, each one being a single event considered to be a duplicate by our redundancy algorithm. Another sample of this improvement is showed in Sect. 5.3.

Moreover, events formed by a single user talking about a topic as e_1 from Table 3.1 has now disappeared since other events with a higher impact have emerged.

From this comparison we can conclude that, when calculating an event impact, considering the number of different users talking about it not only help us to perform a more accurate estimation but also to filter away undesired events which actually are not supposed to be catalogued as it.

5.3 Redundancies detection

We will next expose the experimental results obtained from testing the new redundancy algorithm introduced in Sect. 3.3.3. Tables 3.2 and 3.3 shows the results obtained using original MABED approach and our final MABSED model using *directly redundant* algorithm implemented by MABED, respectively. We recall that the dataset analyzed corresponds to the Spain's general elections. Table 5.3 shows these results.

First and most striking thing is the fact that e_2 main words has increased considerably; in turn that event has been formed by another fifteen ones considered as redundant. This differs significantly with results obtained using both other mentioned approaches where the largest event had only two main words. e_1 has also increased its number of main terms.

$e_{\#}$	Time interval	Topic
1	From: 2019-04-28 07:30 To: 2019-04-28 18:30	**votar, madrid, colegio:** españa(0.73), vox(0.72), apoderado(0.70), gente(0.68), 28a(0.67), in(0.65), eleccionesgenerales28a(0.65), of(0.62), salir(0.62), derecho(0.61)
2	From: 2019-04-28 21:00 To: 2019-04-29 02:30	**rivera, noche, españa, derecha, ganar, sánchez, izquierda, pp, escaño, eleccionesl6, resultado, vox, psoe, sanchezcastejon:** ferraz(0.97), gritar(0.96), conriverano(0.95), pedro(0.94), pactar(0.91), 28a(0.84), extremo(0.79), gobierno(0.78), eleccionesgenerales28a(0.78), albert_rivera(0.77)
3	From: 2019-04-28 13:30 To: 2019-04-28 20:30	**participación:** 2016(0.91), 60(0.82), punto(0.82), subir(0.75), alto(0.71), hora(0.67), elección(0.65)

Table 5.3: April 28th 2019 top 3 events, MABSED algorithm

Another remarkable thing is that although those three events represents the same occurrence, MABSED has managed to differentiate three segments. We will relate them in chronological order:

1. $\mathbf{e_1}$, from 7am to 18pm. This event is formed by tweets encouraging people to go out to streets and exercise their right to vote. It can be tell by its time interval and by words as 'votar', 'colegio', 'salir' and 'derecho'.
2. $\mathbf{e_3}$, from 13pm to 20pm. This other event relates that elections participation has been much higher than the last ones. Words indicating this are 'participación', 'subir', 'alto' and '2016'.
3. $\mathbf{e_2}$, from April 28th 21pm to April 29th 2am. This last event, as its time interval suggests, represents elections results. This is the reason why it is the one on which most redundancies has been found. It is formed by most political groups and politicians names.

According to evidences provided in this section we can say that our resultant model has proven to be more accurate in deciding whether two events should be considered redundant or not. Moreover we have handled conflictive cases of events with a great media coverage which will present a large list of related words.

5.4 Model evaluation

In this last section we evaluate the global performance of the proposed model by comparing it with the original MABED approach.

Evaluating an unsupervised clustering performance is a quite difficult task. In this case we have used the same evaluation method employed on a similar project [25], which consists on judging the proposed model by analyzing real case studies.

For this final survey we used our collected April 15th 2019 dataset. We manually looked up what was talked about on social media, labeling tweets for most important events happened that day. We also looked for news in digital newspapers that confirmed our criteria. Results are shown in Table 5.4.

$e_{\#}$	**Description**
1	Cathedral of Notre Dame fire during restorations work[1].
2	Beginning of 'Semana Santa' vacances in Madrid[2].
3	World Art Day[3].
4	Max Pradera, an Spanish journalist, posted an unfortunate comment about Almudena Cathedral and Notre Dame[4].
5	Premiere of the last season of Game of Thrones[5].

Table 5.4: April 15th 2019 most important events

Considering this we run both models retrieving the top k=*5* events; providing a higher value of k would not make sense as we have considered this dataset to only have 5 significant occurrences. We also performed both detections on filtered and unfiltered dataset. Table 5.5 shows all predictions performed.

We have followed a simple standard, labeling predictions as **event** or **non-event** depending on whether they appear in Table 5.4 or not.

[1]https://www.nytimes.com/es/2019/04/15/notre-dame-incendio/

[2]https://www.abc.es/espana/madrid/abci-semana-santa-2019-procesiones-madrid-recorrido-horario-y-programa-201904110105_noticia.html

[3]http://www.aimdigital.com.ar/15-de-abril-dia-mundial-del-arte/

[4]https://www.abc.es/internacional/abci-pradera-pudiendose-haber-quemado-almudena-y-quema-notre-dame-201904160310_noticia.html

[5]https://www.elmundo.es/television/2019/04/09/5cac665ffc6c8393168b46eb.html

		Dataset size	Detected events	Accuracy
MABED	**Unfiltered**	12,026	e_1	0,2
	Filtered	11,263	e_1	0,2
MABSED	**Unfiltered**	12,026	e_1, e_2	0,4
	Filtered	11,263	e_1, e_2, e_4	0,6

Table 5.5: April 15th 2019, models comparison

As can be seen accuracy increments as we go down the table. MABED reports an accuracy of $0,2$ for both filtered and unfiltered datasets. Analyzing each event tweets we retrieved how they were detected:

- **Unfiltered:** detected two events formed by spam, one created by a single user, an unmerged redundancy of this last one and one event from Table 5.4.
- **Filtered:** it got rid of spam events, but detected the same event created by a single user and three unmerged redundancies of that event. It also detected only one right event.

However, MABSED managed to retrieve one more event from the unfiltered dataset reaching a $0,4$ accuracy, and even found one more when we feed it with the filtered one:

- **Unfiltered:** two right events, two formed mainly by spam and one last undetermined event.
- **Filtered:** three events contained in Table 5.4 and two undetermined ones.

The maximum accuracy we have managed to obtain has been $0,6$ by using MABSED model with the filtered dataset, meaning an improvement of $0,4$ points with respect to the original model. These results demonstrate the importance of spam presence when detecting bursts in words appearance count and the model improvements that have been applied along this project.

CHAPTER 6

Conclusions and future work

In this chapter we will first describe in Sect. 6.1 the extracted conclusions. Next, in Sect. 6.2 we will enumerate goals achieved along this project. Finally, in Sect. 6.3 we will propose possible future lines of work that we could follow for new investigations.

6.1 Conclusions

In the following lines we compile the conclusions reached by developing this project. Although some of them has already been raised over the course of the document, we are reformulating them again in its corresponding section.

First conclusion is that when analyzing Twitter Streams we have to pay special attention to the existence of spam. Collecting tweets containing a particular term may help to filter it partially; however and as it was in our case, collecting every single tweet discriminating only by location and language is not helpful as it will only reduce the obtained spam to a particular area. In order to perform a better data analysis we need to filter away those undesired documents by differentiating them from the actually important ones in some way. We raised a particular solution in Sect. 4.3.1, however it is not the only one which can be implemented.

A second conclusion reached by working with both models is that we can not base the event's impact measure only on the number of mentions involved in it. There are cases where public interest events do not require the usage of mentions as they do not involve any Twitter user to be quoted in the document. We mean situations related to occurrences, accidents, disasters and broadly speaking any happening where quotable people are not engaged. Delegating the impact measure exclusively to mentions would lead to real impact underestimations and we would be exposed to detecting events formed by a single user. Compute that impact attending to the number of users has demonstrated to be a better option, using also the number of mentions to reach a more precise detection. Quoting Sect. 5.2, *'considering the number of different users talking about an event not only help us to perform a more accurate estimation but also to filter away undesired events which actually are not supposed to be catalogued as it'*.

Our last conclusion is the importance of a good redundancy algorithm for clustering projects [26, 27]. If two clusters meant to be classified as one are actually separated in two different groups we are introducing redundancies that can affect later calculations. Moreover, in projects as ours where the number of clusters asked to form is small this task is crucial in order to perform an objectively correct detection.

6.2 Achieved goals

Then we will list the objectives achieved by carrying out this project.

1. **Development of a Streamer for the city of Madrid.** Through interactions with Twitter Streaming API we were able to develop a refined Streamer saving Madrid's citizens posts. They are saved in CSV files divided into half-hour strips for a better processing.
2. **Design of a spam filter.** In order to filter away spam from our Streamer output we built a filter which detected and managed undesired documents. For this purpose we used concepts completely new for us as the *TF-IDF vectorizer* or *Cosine Similarity* between two vectors.
3. **Implementation of a lemmatizer.** Machine Learning models have been proved to show a better performance when feeding them with preprocessed data as normalized text. For this component we searched and tested different frameworks until we found and implemented in our project the one we considered to gave us better results.
4. **Adaptation and improvement of the detection model.** Starting from an already raised model, MABED, we adapted it to our project's requirements. Moreover, we

noticed that some improvements needed to be applied in order to fit the purpose we wanted to use it on. Finally a new model named MABSED was proposed facing some previous limitations as impact measure and redundancy detection.

5. **Development of a monitoring system for detected events.** In order to ease the user the visualization of detected events we developed a dashboard on which he could understand better the analysis performed by our model. It also shows data not generated by our detector as tweets composing each event or a heat map showing its location.
6. **Design of the pipeline.** Using Luigi as orchestrator we finally made our system autonomous allowing data to flow over different modules without synchronization problems. This process will repeat constantly while it is deployed showing new events each thirty minutes.

6.3 Future work

We will finally propose some possible work that could be done in the future in order to improve this project.

- **Filter improvement:** as we have said, filtering away spam is crucial in order to assure the quality of the results. Twitter spam can appears in many ways, and as we have seen even repetitive posts which can not be considered as spam can affect our detector. A quite important improvement is adjusting and training our filter so that it can detect and avoid undesired tweets. An example of these documents can be Instagram's posts published in Twitter. By default they follow a structure where user uploaded photo caption is followed by his location and the link to Instagram's picture. This may cause that the usage frequency of the city in which the post was realized increases and conforms an undesired event or related word. However as caption can be anything the user writes and it is quite difficult that two posts have a similar caption this type of repetitive document is not detected as spam by our implemented filter.
- **Event tracking:** our project has centered on monitoring the three events with higher impact in the last 24 hours. However, having a register of previous events would be interesting showing for example all the last month or last week events and representing them on a table sorted by impact. Implementing this will require a quite good algorithm since as we run our detection task each half an hour most events will be duplicates or same events varying in just a few related words. Maybe this algorithm could use our redundancy detection one in order to define duplicates, but it would

need to be tested and modified to fit the new requirements.

- **Event prediction:** our detector is designed to be working 24 hours, 7 days a week analyzing what people talks about. Accidents and catastrophes are spontaneous events which can not be predicted. However there may be other types of events which happen periodically without us noticing. If we changed the way our model learns it could be possible to predict certain events by detecting social trends which preceded similar previous events. This model application may be quite different from the one we are using right now, but with the appropriate changes and setting existent algorithms we could study if it can be used for such a task.

APPENDIX A

Impact of this project

This appendix reflects, quantitatively and qualitatively, on the possible impact that this thesis could cause. We will consider social impact (Sect. A.1), economic (Sect. A.2), environmental (Sect. A.3) as well as its ethical implications (Sect. A.4).

A.1 Social impact

As introduced in Sect. 1.1, social networks as Twitter provides a new dimension to events reports due to its immediacy nature and the fact that the only needed condition to be able to post and contribute is having an account. This makes possible that people can report any incident they witness or they have heard of with a margin of minutes.

Our project has a big social impact since it allows society to be aware of any incident occurred in the city where they live long before they would be informed by traditional news media. In fact it will not only report locals events, but also national and international interest ones.

Through the use of this platform citizens can get to know occurrences truly important to society as they are based in people worries reflected by its social networks posts.

A.2 Economic impact

In this section we will raise the possible economic impact which companies using the monitoring subsystem developed in this project may experience, as it is the part of the project intended to interact with the user.

From the point of view of a company, the immediacy of our project analysis would be profitable for those firms where response time is critical. By using our system they might react faster to occurrences, allowing them to be the first to take appropriate action. This could be exploited by for example the marketing department in order to start a certain campaign, with the consequent economic benefits it could bring.

A.3 Environmental impact

We will next discuss the environmental impact that the development of a system as ours would involve. Since the only need this projects involves is a infrastructure which hosts and deploys our architecture, the main environmental consequences would be caused by this.

On the one hand we have the electricity usage. Our system is designed to run autonomously 24 hours a day, 7 days a week. Generating the electricity needed to assure its functioning would contribute with the greenhouse gas emission, with consequent implications for the ozone layer.

On the other hand, cooling and keep running smoothly the whole architecture would also require an energetic and other resources usage which would leave an imprint on the environment.

A.4 Ethical implications

Lastly, we will evaluate the ethical implications of our project.

First implication would be, as in almost every Machine Learning project, the destruction of some existing jobs. As introduced in Sect. A.1 this approach can develop a better and more efficient performance than most of the traditional news sources. If a faster and more concise source of news appears and replace the current ones many jobs could disappear or hopefully evolve with the new needs.

The last ethical implication is related to the source where analyzed data comes from,

Twitter. Their privacy police states that users consent to collect, transfer and storage data published by them is public since they have the option to change their account's privacy settings and convert it into a private one. This project exclusively employs publicly tagged content for its performed analysis as non private data on Twitter is considered to be a public data source. Moreover, collected tweets and its metadata are not stored in any archive with commercial intentions.

APPENDIX B

Economic budget

This appendix details an adequate economic budget to bring about this project. Physical resources needed are detailed in Sect. B.1, human resources in Sect. B.2, and licenses and taxes in Sect. B.3 and B.4, respectively.

B.1 Physical resources

Budgets derived from physical resources are detailed in this section. They are exclusively derived by the personal computer costs the project has been developed in.

This technical characteristics are not exclusives and the development could have been accomplish in many other environments, however we are going to enumerate the ones it has been proved to work in:

1. **CPU:** Intel Core i5-4210U 1.70GHz x 4
2. **RAM:** 8 GB
3. **Disk:** 500 GB SSD

This computer is an older version which is currently out of the market. However a similar features computer can be easily found and bought for an approximate price of 700€.

B.2 Human resources

In this section we will board the costs arising from the workers needed for the realization of this project.

For this estimation we will consider that the draft has been carried out by a single person. For the number of hours employed we have used the number of credits assigned by ETSIT to its final degree project, which is 12. According to a convention also approved by this entity, each ECTS credit represents a total of 30 hours of work, reaching a total of 360 hours. Considering a part-time schedule (4 hours per working day) and that a moth has 22 working days, this will lead to a total of 88 working hours per month. As a result, this project is meant to have a duration of 4 months. With this in mind and considering that the standard salary of an internship for an undergraduate student is 450 €per month, the human resources costs would amount to a total of 1,800 €.

B.3 Licenses

Costs corresponding to the licences of the software tools necessary for the development of the system carried out in this project are contained in this section.

However, all the software used in this project is open-source and free to use even with commercial purposes, so there are not any licenses cost involved in the realization of this draft.

B.4 Taxes

One of the possible actions we could take once we have fully developed and tested the viability of this project is selling it to another company. In that case, some taxes derived from that sale should be considered.

This sale is subject to a tax of 15% of the price of the product, as defined in Statute 4/2008 of Spanish law. Besides, if we wanted to sell the software to a foreign company more taxes should be considered.

Bibliography

[1] John Krumm and Eric Horvitz. Eyewitness: Identifying local events via space-time signals in twitter feeds. In *Proceedings of the 23rd sigspatial international conference on advances in geographic information systems*, page 20. ACM, 2015.

[2] Rui Li, Kin Hou Lei, Ravi Khadiwala, and Kevin Chen-Chuan Chang. Tedas: A twitter-based event detection and analysis system. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1273–1276. IEEE, 2012.

[3] Farzindar Atefeh and Wael Khreich. A survey of techniques for event detection in twitter. *Computational Intelligence*, 31(1):132–164, 2015.

[4] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[5] Francisco J Blanco-Silva. *Learning SciPy for Numerical and Scientific Computing*. Packt Publishing Ltd, 2013.

[6] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[7] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[8] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.

[9] Tiberiu Boroș, Stefan Daniel Dumitrescu, and Ruxandra Burtica. NLP-cube: End-to-end raw text processing with neural networks. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 171–179, Brussels, Belgium, October 2018. Association for Computational Linguistics.

[10] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[11] Kevin Makice. *Twitter API: Up and running: Learn how to build applications with the Twitter API.* " O'Reilly Media, Inc.", 2009.

[12] Joshua Roesslein. tweepy documentation. *Online] http://tweepy. readthedocs. io/en/v3*, 5, 2009.

[13] Clinton Gormley and Zachary Tong. *Elasticsearch: The definitive guide: A distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[14] Erik Bernhardsson, Elias Freider, and Arash Rouhani. spotify/luigi-github.

[15] Adrien Guille and Cécile Favre. Event detection, tracking, and visualization in twitter: a mention-anomaly-based approach. *Social Network Analysis and Mining*, 5(1):18, 2015.

[16] Gabriel Pui Cheong Fung, Jeffrey Xu Yu, Philip S Yu, and Hongjun Lu. Parameter free bursty events detection in text streams. In *Proceedings of the 31st international conference on Very large data bases*, pages 181–192. VLDB Endowment, 2005.

[17] Kurt Thomas, Chris Grier, Dawn Song, and Vern Paxson. Suspended accounts in retrospect: an analysis of twitter spam. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 243–258. ACM, 2011.

[18] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[19] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schutze. Introduction to information retrieval? cambridge university press 2008. *Ch*, 20:405–416.

[20] Wael H Gomaa and Aly A Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications*, 68(13):13–18, 2013.

[21] Djoerd Hiemstra. A probabilistic justification for using tf× idf term weighting in information retrieval. *International Journal on Digital Libraries*, 3(2):131–139, 2000.

[22] Baoli Li and Liping Han. Distance weighted cosine similarity measure for text classification. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 611–618. Springer, 2013.

[23] Charles Romesburg. *Cluster analysis for researchers*. Lulu. com, 2004.

[24] Michal Toman, Roman Tesar, and Karel Jezek. Influence of word normalization on text classification. *Proceedings of InSciT*, 4:354–358, 2006.

[25] Mario Cataldi, Luigi Di Caro, and Claudio Schifanella. Emerging topic detection on twitter based on temporal and social terms evaluation. In *Proceedings of the tenth international workshop on multimedia data mining*, page 4. ACM, 2010.

[26] Yong-Hyuk Kim, Sehoon Seo, Yong-Ho Ha, Seongwon Lim, and Yourim Yoon. Two applications of clustering techniques to twitter: Community detection and issue extraction. *Discrete dynamics in nature and society*, 2013, 2013.

[27] Jon Kleinberg. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, 7(4):373–397, 2003.