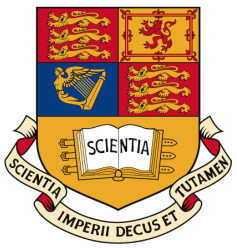
Understanding Revolution 2.0 - Mining the Arab Spring Social Network Data



George Eracleous

Department of Computing

Imperial College London

A thesis submitted for the degree of

MEng Information Systems Engineering

2012

I would like to dedicate this thesis to my loving parents.

Acknowledgements

And I would like to acknowledge ...

Abstract

TODO: WRITE ABSTRACT

Contents

Chapter 1

Introduction

In this chapter, we will explain the problem we aim to tackle in this project and what are its challenges. By breaking the problem down into smaller problems we will explain the steps we need to undertake in order to find the solution and the diffculties we expect to encounter in our endeavour.

1.1 Motivation

Social media have literally changed our lives in the recent years. Social media platforms such as Facebook, Twitter and many more not only offer their users a way to connect and communicate with their friends, colleagues and family but they have been used as a way to trigger and sustain revolutions and protests. Shirky in [20] argues that social media have the power to sustain political uprisings while others believe that the power of social media is overrated [14]. Nevertheless, there have been several examples of people using social media to oust dictators and protest against regimes. This is exactly what happened during the Iranian elections in 2009 and Egypt's uprising in the beginning of 2011. During these events people used social media and especially Twitter, to report news during protests and influence others to participate. The events that took place during the Egyptian uprising engendered an unprecedented flow of information and ideas on Twitter. People who were watching their Twitter timelines could literally see the events unfolding before their own eyes. One wonders what would

have happened if someone collected all this information and tried to make sense of them. What if we could be able to detect events and describe them as they are happening? These are the questions this project sets out to answer, not in a science fiction way but following a structured and scientific methodology. We have collected social media data from Egypt's uprising and we have tried to describe what happened. Using this case study as a starting point we developed a generic framework for detecting and describing events as they unfold. In the last few years a growing community of researchers started using Twitter to conduct research on social net- works and data mining. Recently, Lotan et al.[13] attempted to investigate the events that took place during the Egyptian revolution by collecting and analysing a large amount of tweets. They have identified different types of users (media organizations, bloggers, activists) in the dataset and they studied how each type was influencing the other by looking at the information flow between them. Another research project studied the information dissemination during the Iranian elections and offered important insight into the dynamics of information propa- gation that are special to Twitter [23]. Other studies have investigated event detection and summarisation during the Egyptian revolution as well as the different type of actors participating in the diffusion of news through Twitter. Mining and analysing Twitter data is not an easy task and research has revealed numerous challenges for researchers. The biggest challenge is the enormous amount of data flowing on Twitter every second requiring careful filtering in order to block unwanted tweets such as updates on someone's life or spam tweets. Additionally, an intrinsic problem of analysing on-line data is that they are not always related to real world occurrences. For example, Twitter specific memes such as #musicmonday(users tweet their music preferences) or #followfriday(someone suggesting to other users someone else to follow) produce a massive amount of tweets during a certain time interval but they do not correspond in real life events. Therefore, the motivation behind this project is firstly to overcome these diffculties and also contribute to the research community by providing robust and accurate tools for automatically describe and detect events from social media content.

1.2 Objectives

In this section we present our main objectives for this project by breaking down the main problems into smaller sub-problems. We believe that this breakdown will help us develop a robust framework for solving the problem by identifying the individual difficulties of each sub-problem.

1.2.1 Event detection

The main objective of this project is to develop a framework for event detection. We have collected over 100,000 tweets related to the revolution and our aim is to sift through them and identify the main events that took place. Below we present the formal definition of this problem along with the necessary definitions.

A **real-world event** identifies something non-trivial happening at a certain time and at a certain place **?**. We describe a real-world event to be a set of attributes describing that event such as keywords, geographic location, time of occurrence and the social actors (Twitter users) involved in it. It is possible to have incomplete or no information at all for an event therefore our system should take into consideration these possibilities.

A **tweet** is defined as a tuple (a, c, H, d, F_T) where $\boldsymbol{a}$ is the screen name of the author, $\boldsymbol{c}$ is the textual content, H is the set of hashtags associated with the tweet, $\boldsymbol{d}$ is the timestamp and F_T is the feature vector. The feature vector contains a set of attributes associated with the tweet such as the number of times this tweet has been retweeted, whether or not this tweet is a @-reply, the set of URLs and/or named entities if any.

A **tweet stream** is defined as $(t_1, t_2, t_3, ..., t_N)$ where N is the number of tweets in the stream, t_i is a tweet and $d_{t_i} < d_{t_{i+1}}$ (i.e. ti has occurred prior to t_{i+1}).

A **topic** is defined as $\{t_i | t_i \in T \wedge |A| \geq U \wedge F_P \wedge \forall t_i, t_j . t_i$ is similar to t_j with respect to this topic $\}$ where T is the tweet stream, A is the set of unique authors related to the topic and U is the minimum number of actors needed to be involved in the

topic. F_P is a feature vector containing attributes of the topic such as time, geographical location and keywords. The similarity of a tweet with respect to a topic is based on the tweet feature vector F_T as well as the textual content of the tweet.

Given a tweet stream T our aim is to partition it into a set of distinct topics $P = \{p_1, p_2, p_3, ..., p_{N_p}\}$ where N_p is the number of detected topics. Then based on the feature vector F_p we can associate each of the topics with one or more events.

1.2.2 Event summarisation

When an event has been detected it would be interesting to be able to describe what happened by extracting a description in the form explained below. Therefore, we wanted to develop the framework for generating automatic summaries for an event. The formal definition of the event summarisation problem is given below.

Given a topic or topics associated with an event as described above, we aim to extract an **event label** which serves as a description for this event.

An event label is defined as a set of descriptors $D = \{d_i | i \in \{1, 2, ..., N\} \wedge |D| = X\}$ where N is the number of tweets associated with this event and a descriptor d_i is a tuple (c_i, d_i, H_i, s_i) where c_i is the content of tweet t_i, di the timestamp of tweet t_i, H_i is the set of hashtags associated with tweet t_i and s_i is the score of tweet t_i calculated by the summarization algorithm. Only the X highest scored tweets will appear in the event label.

The score is calculated based on several criteria such as quality, relevance and usefulness of a tweet. Quality refers to the textual quality of the tweet i.e. low quality tweets contain slang and short hand notation. Relevance is how well a tweet reflects information related to the associated event and usefulness is how well the tweet informs a human about the event.

1.2.3 Actor classification

In order to further understand the structure of a detected event we wish to identify the different types of users that are involved in the information dissemination on Twitter. Varying from media organisations and activists to normal individuals, these people were the people who ignited, documented and sustained the revolution. We aim to develop a tool for automatic classification of the users into different categories.

An **actor** is defined as $\{a, fr, fl, T, F\}$ where $\boldsymbol{a}$ is the actor's username, fr and fl are the sets of actor's followers and followees respectively, T is the set of tweets which belong to this actor and F is the feature vector. The feature vector contains the activity features of the actor such as the total number of tweets, the fraction of retweets among all the tweets from a user, the fraction of @-replies directed to other users and the fraction of tweets containing a URL.

An **actor label** is defined as $l_a \in \{c_1, c_2, c_3, ..., c_N\}$ where a is an actor, N is the number of existing classes and c_i is an actor type such as blogger, activist and media organisation. Given a training set of actors and their corresponding actor labels $\{(a_1, l_{a_1}), (a_2, l_{a_2}), ..., (a_N, l_{a_N})\}$ our aim is to find a mapping $l_a = f(a)$ which can predict the associated actor label l_a for an actor $\boldsymbol{a}$. The prediction will be based on the feature vector F and the other attributes associated with an actor such as the sets of followers and followees.

1.3 Contributions

In the process of developing the solutions for the problems described above we have come across several challenges as well as some opportunities. We have tried to tackle all the difficulties and also exploit the opportunities we were given. Our efforts have led to several contributions and the following list summarises the main contributions of this project:

- The solution of the event detection and summarisation problems led to the development of a data mining toolset which consists of several sub-

components. These components vary from data acquisition tools to clustering algorithms and text processing tools. These individual components can act independently to solve smaller tasks but most importantly they can be combined to solve the problems posed by this project.

- In order to be able to develop our solution we have conducted an extensive literature survey about event detection, event summarisation and Twitter user classification. Our research was focused on these three areas in the context of social media and more specifically on Twitter content. The emergence of social networks has sparked the interest of the research community and numerous solutions have been proposed over the last few years. We have gathered the most prominent solutions in the literature and we have commented on their applicability and feasibility in our project.

- The core component of our solution for the event detection problem is text clustering. The challenging aspect of our work is that we had to deal with social media content which offers some advantages but at the same time it poses significant challenges. After implementing several clustering algorithms we have conducted a thorough evaluation study to assess their individual performance in several aspects that are unique to social media documents. Additionally, the event summarisation and user classification components have been evaluated as well and our results can be used as a guidance to future projects in this area.

- We have built a proof-of-concept web application which uses our algorithm to detect events and summarize them using Twitter data. The application provides a user friendly environment and data visualisations to allow the user to discover events in historic data.

- We have collected and analysed a large number of Twitter content related to the Arab Spring and we have used this dataset to conduct a case study on real data using our algorithm and web application.

1.4 Report Structure

TODO: Complete this section when the rest of the report is done.

Chapter 2

Background research

In this chapter we present the state of the art for the individual problems of this project. We discuss the most prominent solutions and methodologies and we comment on their applicability in our work.

2.1 Event detection

2.1.1 Traditional event detection vs social media event detection

Traditionally, event detection research has focused on detecting news events from continuous streams of news articles. The majority of the work done in this area does not account for any social effects but focuses only on the textual properties of a news article. Therefore, researchers have used natural language tools such as named entity extraction and part-of-speech tagging in order to perform event detection **?**. However, in this project our problem exhibits several differences compared to traditional event detection since we aim to use data from Twitter.

Twitter and more generally social media documents exhibits several advantages in relation to the event detection task but at the same time pose significant challenges for the researchers. More specifically, a large number of social media documents are irrelevant to real world events as they usually describe updates on

the user's life or information unconnected with an event. About 40% of all the tweets are pointless "babbles" **?**. Additionally, social media documents contain little textual content which is usually restricted to a few characters (140 in the case of Twitter) which renders traditional event detection methods undesirable. On the other hand, social media documents provide us with new prospects since they often come with additional context information such as tags, locations and network structures.

2.1.2 State of the art event detection in social media streams

One of the first research projects that tried to detect events in Twitter streams aimed to inform people about the occurrence of an earthquake **?**. The explicit assumption in their research is that a large number of Twitter users are tweeting when an earthquake happens and therefore it is possible to detect its existence. They have developed an earthquake reporting system which monitors Twitter streams and reports the occurrence of an earthquake promptly. In order to detect tweets discussing earthquake occurrences they have used a support vector machine to detect real occurrences. In order to train their classifier they have prepared a set of training data consisting of positive tweets (tweets which mention an earthquake occurrence) and negative tweets. For each tweet they used different types of features to be used by the classifier:

- statistical features such as the number of words in a tweet message and the position of the query word within a tweet.
- keyword features such as the words in a tweet.
- word context features such as the words before and after the query word. By treating users as sensors they have built a socio-temporal model which can predict the occurrence of an earthquake and estimate the location using Kalman filtering.

After evaluating their reporting system in Japan for a month, they reported that 96% of earthquakes larger than JMA seismic intensity scale 3 or more were successfully detected.

Researchers at University of Edinburgh **?** proposed a novel method for detecting news events in Twitter streams. The main objective in their work was the detection of a"first story" in a stream of tweets. The task of First Story Detection (FSD) was first coined in **?** and aims to identify the first story to discuss a particular event. The traditional method of First Story Detection (FSD) is to represent documents as vectors in term space. Then each document is compared to all the previous ones in the stream and a similarity value is produced. If this value is below a particular threshold it is declared to be a "first story". However, the challenge they faced was that Twitter provides a vast amount of data in real-time which renders the traditional method inefficient. Therefore, in order to alleviate this problem they implemented a novel algorithm based on locality-sensitive hashing. Their method uses the cosine between two documents as the similarity metric and a hashing scheme proposed by Charikar **?**. This scheme limits the number of documents to be compared and increases the performance of the algorithm. They reported that the hashing scheme alone yields poor results with a lot of variance. In order to solve the problem they used a variance reduction strategy. They have tried their algorithm on Twitter data collected over a period of six months and they have showed that their system is able to detect major events with reasonable precision.

A different approach is proposed by Sayyadi et al. **?** in their paper "Event Detection and Tracking in Social Streams". The main idea behind their proposed algorithm is that documents describing the same event will contain similar keywords, and the graph of keywords for a document collection will contain clusters which are effectively events. They extract keywords from the documents and they construct a graph whose nodes are the keywords and edges between the nodes are formed when those terms co-occur in a document. Their algorithm uses betweenness centrality to assign scores to the nodes and the nodes with high betweenness centrality score are removed. This way they manage to reduce the

graph into several unconnected sub-graphs (communities). Then they consider each community of keywords as the key document/event with the keywords being a bag of words summary of the event. Documents in the original corpus which are similar to this key document can be clustered, thus retrieving a cluster of topical documents. Again in their methodology they used cosine similarity to discover document clusters for key documents. An important feature of their system is that they also took their approach one step further by proposing a sliding window approach for subsequent event detection in social media streams as it is considered impossible to apply the event detection algorithm to each new document arriving in the stream without performance degradation.

One of the best methodologies for event detection is proposed by Becker et al. **?**. In this study they focus on on-line identification of real world event content. A common problem of event detection which arises in Twitter is the false identification of non-event content as event content. For example, specific conversation on topics or memes (e.g., using the hashtag #followfriday) which do not correspond to real world occurrences might be incorrectly identified as events. In order to deal with this problem they decided to introduce a two stage process where initially each event and its associated messages are grouped together with an online clustering algorithm. In the next stage they calculate several features associated with each cluster in order to train a classifier to distinguish between event and non-event clusters. The main advantage of this method is that the features used to classify the clusters exploit the Twitter-specific features thus making the process of event detection much more robust. They evaluated their system by collecting 2,600,000 tweets posted during February 2010. They have used a nave Bayesian classifier as the baseline method and they showed that their method outperformed the baseline.

2.2 Event summarization

The first attempt to summarize events on Twitter was attempted by Chakrabati et al. **?**. Their research paper discusses the problem of extracting the tweets which summarize long-running events, such as football games and they propose

a two-step solution. The first part uses a Hidden Markov Model in order to partition the event time-line in sub-events (e.g, tweets before a goal and tweets after the goal) based on the burstiness and the word distribution in tweets. The second stage selects the key tweets of each segment and eventually all key tweets are combined to give the summary. The system takes as its input an event, which is detected using one of the existing event detection algorithms and it outputs a few tweets that best describe the occurrences in that event. Their proposed algorithm is able to isolate at most one sub-event in each time segment, which can then be used by the summarizer to output summaries. They use a variant of the Hidden Markov Model which models each event as a sequence of states. The tweets are the observations generated by the states and the transition between states models the chain of sub-events. They propose two other methods for summarization which are based on simple TF/IDF scores and segmenting the event time-line in equally spaced time intervals. They compare their algorithms performance in comparison to these baseline techniques for tweets pertaining to football games and they showed that their algorithm was performing particularly well. However, the authors state that they have not tested their system for one-shot long-running events such as revolutions which can be a significant problem in our project.

Becker et al. **?** proposed a different methodology for selecting quality tweets from the set of tweets related to a specific event. The goal of their summarization process is to extract tweets which meet the criteria of quality, relevance and usefulness. The extracted tweets should be comprehensible by a human (i.e. they do not contain short-hand notation), they must reflect the information related to their associated event and they must be useful in describing the main occurrences event. In order to satisfy these criteria the authors propose several methods for extracting these tweets. The main idea behind their approaches is that the tweets which are closer to the centre of a cluster (event) are more likely to reflect the key aspects of the event than other tweets. Therefore, their approaches are based on the notion of centrality. The first method, the centroid similarity approach computes the cosine similarity of the TF-IDF representation of each message to its event cluster centroid. Then the tweets with the highest similarity score are

selected. The second method involves message similarity across all messages in an event cluster. This approach, constructs a graph where a cluster message is a node in the graph, and there is an edge connecting a pair of nodes whose cosine similarity exceeds a threshold. This method selects the nodes with the highest degree centrality which is defined as the degree of each node, weighted by the number of nodes in the graph. Finally they use a state of the art method, namely LexRank which defines centrality based on the idea that central nodes are connected to other central nodes. They have evaluated their approaches alongside with some other baseline approaches such as selecting the newest tweets in the cluster and selecting tweets from popular users. The results revealed that the centroid method outperformed the others in all three selection criteria: quality, relevance and usefulness.

2.3 User classification

Twitter users have diverse backgrounds and they tweet for a variety of topics. There are many different types of users such as celebrities, bloggers, journalists, media, and organizations. Several studies have tried to identify different type of users on Twitter and explore their distinct characteristics. In this section we discuss various techniques which have been used to identify and classify users.

In a recent study of the Arab Spring, Lotan et al. **?** investigated how information was propagating during Egypt's uprising. In order to identify information flows they have collected a large amount of tweets during that period and identified the active users in the social network. They selected 963 users, from their dataset, who either were first to tweet, or were retweeted or mentioned at least 15 times. Then they manually classified these users in different categories:

- Mainstream media organizations (e.g., @AJEnglish, @nytimes).
- Mainstream new media organizations (e.g., @HuffingtonPost).
- Non-media organizations (e.g., @Vodafone, @Wikileaks).

- Mainstream media employees (e.g., @AndersonCooper).
- Bloggers (e.g., @gr33ndata).
- Activists: (e.g., @Ghonim).
- Digerati: (e.g., @TimOReilly).
- Political actors: (e.g., @Diego_Arria, @JeanMarcAyrault).
- Celebrities (e.g., @Alyssa_Milano).
- Researchers: (e.g., @JRICole).
- Bots: (e.g., @toptweets).
- Other: users that do not fall under any of the other categories.

It is apparent that since we are interested in automatic classification their method is not well suited in our case. However, their work has revealed several challenges of the user classification problem in Twitter which we must overcome in our implementation. The main challenge was that a number of users were very difficult to classify due to the fact that these users could be classified in multiple classes. For example, some users are bloggers but they are also activists and this led to ambiguities during the classification process. Furthermore, a number of accounts were deleted or suspended during the data collection process which inevitably introduced missing nodes in the social network graph.

Leaders, lurkers, associates and spammers are some of the type of users active in Twitter and another research project aimed to automatically identify them **?**. They designed two different methods for user classification: a context-dependent method and a context-independent method. They classified the users in four types: leaders, lurkers, spammers and close associates. Their research is based on several assumptions on these types of users. More specifically, they assume that spammers follow many users but they are followed be a few people and they

make on average 1,000 tweets per day. On the other hand, leaders are identified by the high rate of tweeting and a large number of followers but almost no followees. Closes associates have strong connectivity to their followers and low rate of tweeting. The final class, lurkers, follow many people, but they rarely post any tweets. When there is no contextual information about a user, such as their followers and @-replies they use the context-independent method which examines the tweeting pattern, otherwise they exploit these contextual information by employing the alternative context-dependent method. The context-dependent method employs a fuzzy logic approach in the first stage to estimate inter-actor relationship strengths and then they remove the links with strong relationships because they naturally represent close associates. Then, in the second stage, they use a linear classifier, using the number of tweets and the followee-follower ratio as two features. The context-independent method uses traditional classifiers and generic actor tweet patterns. They collected the tweets of particular types of tweeters over a period of ten days and then they used these to train three classifiers: nave Bayesian, multi-layer perceptron (MLP) and a random forest (RF) classifier. In their evaluation the MLP classifier has been found to outperform the naive Bayesian and Random Forest classifiers on the more challenging problem of classifying actors with limited data. Their research gives strong indications that user classification with Twitter data is possible with high performance.

A similar approach was taken by Choudhury et al. **?** in their research which aimed to automatically classify Twitter users in three categories: organizations, journalists and ordinary individuals. Their work offers two significant advantages compared to other methods in the literature. They used Twitter specific features to classify the users which make their system much more robust. They use standard features from previous studies such as network features (followers and followees) and activity features such as the number of tweets for a user. However, they added some additional ones including interaction features (number of retweets, @-replies and tweets containing URLs) and named entities features which capture the presence/absence of named entities in the tweets of a user. Finally, another important feature used in the classifier is the topic distribution which associates the user to a set of 18 broad themes from the IPTC Media Topic

News Codes. They have used 8 different classifiers in order to evaluate their design and the results indicated that the best performing classifier was a k Nearest Neighbors classifier with $k = 10$. Another important contribution of their work is that they investigated the participation of different categories of users in different events. They have showed that different events gather different degrees of participation (number of users from each category) and attention (proportion of posts from each category).

Chapter 3

A data mining framework for automatic event detection and summarisation

In this chapter, we will define a theoretical framework for construction of a data mining system which can extract events from Twitter data. The system comprises of a number of individual components which when combined together construct a complete framework for event extraction. These components are explained in detail in the sections below and in Chapter **??** we discuss the concrete implementation of this framework as well as the development of a proof-of-concept application using this framework.
The sections of this chapter introduce the motivating work for the implementation of the system and we also take a closer look at the concepts and key ideas that are necessary to understand the process we employ.

3.1 Methodology overview

The main idea behind our methodology for event detction is the hypothesis that similar events will be described by tweets having similar content. Therefore, the main task in our methodology is to cluster the tweets in different groups. We further hypothesise that not all of the detected clusters will describe real events,

since clustering will detect topics such as discussions about celebrities or politicians which cannot be thought as events accortding to our definition of an event. Having idientified some events we can then further investigate these clusters extract useful information from them which will serve as the event summary. The theoretical foundations of this procedure and its sub-components are described in detail in the rest of this chapter.

3.2 Raw text processing

Mining and analysing text corpora are two well studied problems but in our case we face a slightly different problem. Social media content and especially tweets are very short (140 characters) and usually contain slang phrases, abbreviations and irrelevant information. Therefore, is of foremost importance to ensure that clever data preprocessing takes place in order to help us in the subsequent tasks. Below we describe the main steps we must follow to pre-process our raw text tweets.

3.2.1 Natural language processing

Natural language processing (NLP) is a field of computer science and linguistics which is concerned with the extraction of information from a human language input. NLP is used extensively throughout this project since we deal with human generated content and we use several of its applications. NLP in the context of raw data processing is used to clean and normalise our initial input which is a tweet containing raw text.
Since tweets usually contain URLs and HTML code one of the first priorities is to remove these elements. Then, the following NLP algorithms can be used on our data:

- **Sentence segmentation and tokenization:** Raw text is split into sentences and then each sentence is further subdivided into words using a tokenizer.
- **Stopword removal:** Some of the words occuring in the documents are

common English words such as 'the', 'and' and 'a'. These words convey almost no information but most importantly, they can reduce the clustering accuracy. Two documents can be erroneously considered related if they contain common English words like the ones we have already mentioned. Therefore, it is important to remove these stopwords. Usually, a dictionary of the common English stopwords is used to filter them out.

- **Stemming:** This is a mechanism for reducing English words to their stem form. For example, the stem form of the words 'connections' and 'connected' is 'connect'. This mechanism is particularly useful in the field of information retrieval and indexing. M.F Porter designed the most widely used stemming algorithm in 1980 [Insert ref here]. This algorithm works by applying a set of different rules, which after a number of iteration yield the final resut which is the stem of the word. Porter has developed 62 rules which may or may not apply to a given word.

In section **??** we explore more applications of NLP and how we can use them to extract automatic summaries, named entities or sentiment from documents.

3.2.2 Inverted index

An inverted index, also known as an inverted file, is a data structure central to text-based information retrieval. The name is derived from its purpose and design which is to map key-value pairs, where a key is a term in a document and the value is the list of documents that contain this term. For example if we have two documents:
Document1: The cat is on the tree.
Document2: The cat sat on the mat.
then the inverted index will look like:

Key	Value
the	{Document1, Document2}
cat	{Document1, Document2}
is	{Document1}
on	{Document1, Document2}
tree	{Document1}
sat	{Document2}
mat	{Document2}

The main reason for using an index is to increase the speed and efficiency of searches of the document collection. In our system the inverted index is vital component since it allows us to construct term-documment vectors easily and also filter terms and documents. For example, using our index we can find the words that appear either too often or less frequently and filter them out. This is used to reduce the dimensionality of our dataset by removing unneccesary words. Alternatively, we can remove documents/tweets which contain keywords that appear too often or less frequently.

3.3 Cluster Analysis

Cluster analysis is the process of grouping a set of data objects into multiple clusters where all the objects in a cluster have high similarity but they are very dissimilar to objects in other clusters. The set of clusters resulting from a cluster analysis can be referred to as a clustering. Clustering is an unsupervised learning method since deals with finding a structure in a collection of unlabeled data. Figure 3.3 depicts a simple clustering example where a clustering algorithm is applied on an unlabeled dataset and four different clusters are detected.
Cluster analysis is applied in a wide range of applications such as business intelligence, image pattern recognition, biology and security. In our project we are interested in cluster analysis and its application in document clustering. We would like to cluster tweets which are similar in terms of both textual content and describe the same topic.
In the rest of this section we describe the key ideas behind document clustering

and we explain the main clustering algorithms that will be used in our implementation.

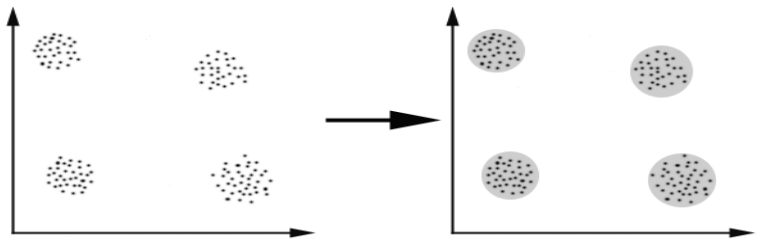


Figure 3.1: The result of appplying a clustering algorithm to unlabeled data. Four distinct clusters have been detected.

3.3.1 Vector space representation

Document need to be preprocessed in order to be in a form that will allow us to perform clustering. The most common approach is to use the vector space representation model. The vector space representation transforms the documents into vectors of term frequencies which can then be used as a means to assess the similarity between documents.

We can represent each document in a dataset by a vector of identifiers. Usually, these identifiers are the distinct words in the document and the resulting vector is called term-frequency vector. If we combine all the vectors for the documents in our dataset we will end up with an $\boldsymbol{m} \times \boldsymbol{n}$ matrix $\boldsymbol{A}$. Each of the m documents in the document collection are assigned a row in the matrix, while each of the n unique terms in each document are assigned a column in the matrix. A non-zero element a_{ij} in $\boldsymbol{A}$, indicates not only that term j occurs in document i, but also the number of times the term appears in that document. Since the number of terms in a given document is typically far less than the number of terms in the entire document collection, A is usually very sparse.

For example in Table 3.1 we see that *Document*1 contains three instances of the word team, while football occurs five times. We can also infer that the words ball and world are missing for the entire document, as indicated by the value of zero

Document	team	ball	football	countries	world	england
Document1	3	0	5	1	3	1
Document2	2	1	2	0	8	2
Document3	1	5	3	0	1	3
Document4	5	0	1	2	5	4

Table 3.1: Term-frequency vector representation of documents

at those entries of the matrix.
The main advantage of transforming the documents in the vector space is that we can define vector-space similarities between documents and therefore we can apply clustering algorithms on the document collection. In section we provide a more detailed discussion on similarity metrics and the vector space representation is used in clustering documents.

3.3.2 Assigning weights to terms with TF-IDF weighting

Once a document is transformed in its term-frequency vector we can assign weights to each term in the vector. So far the term-frequency vectors treat all the terms as equal but this may not be the case. For example, a word that appears more frequently than others in a document could be considered as an important word. At the same time words that appear frequently in the corpus, such as 'a', 'the', 'and', are not very useful and their importance must be discounted.
The most common method to solve this problem is the TF-IDF weight (TF-IDF stands for term frequency-inverse document frequency) which quantifies the importance of a term in a document of a document collection. More specifically, the more a word occurs in a document, and the less it occurs in the rest of the coprus, the higher its TF-IDF weighting will be. Mathematically, TF-IDF is expressed as:

$$tf - idf_{t,d} = tf_{t,d} \times idf_t \tag{3.1}$$

where $tf_{t,d}$ is the importance of term t in document d and idf_t is the importance of term t relative to the entire corpus. $tf_{t,d}$ is higher when the term occurs many times in the document and idf_t is higher when it occurs rarely in

the dataset. Therefore, the TF-IDF weighting for a term is very high if the term occurs frequently in a single document but very rarely in the entire corpus and it is low when the term either occurs rarely in a document or frequently in the entire corpus. TF-IDF is widely used to compare the similarity between documents and a common use case is for search queries where the similarity of a query q with a document d is calculated using TF-IDF, providing a sorted list of the most relevant documents.

3.3.3 Feature selection

TODO: Discusss feature selection methods used in our system.

3.3.4 Distance measures

In data mining applications including clustering we must decide whether two objects are similar or dissimilar. In document clustering we wish to assess how similar are two documents in comparison to one another. Based on the similarity between two objects we can make the decision whether to group them in the same cluster or not. In this section we present three commonly used similarity measures which will be used throughout our work.

3.3.4.1 Euclidean distance

Euclidean distance is probably the most popular distance measure and can be thought as the straight line connecting the two data points that we try to compare. Let the two data points be $i = (x_{i,1}, x_{i,2}, ..., x_{i,p})$ and $j = (x_{j,1}, x_{j,2}, ..., x_{j,p})$ where p is the number of the numeric attributes. The Euclidean distance between i and j can then be defined as:

$$d(i,j) = \sqrt{(x_{i,1} - x_{j,1})^2 + (x_{i,2} - x_{j,2})^2 + ... + (x_{i,p} - x_{j,p})^2} \tag{3.2}$$

Euclidean distance is the default distance measure used with the k-Means algorithm.

3.3.4.2 Cosine similarity

The similarity of two documents which are represented as term vectors, corresponds to the correlation between the vectors. This is calculated as the cosine of the angle between vectors, the so-called cosine similarity. Cosine similarity is widely used for text documents and especially in clustering documents. Let x and y be two vectors for comparison. Cosine similarity is defined as:

$$sim(x, y) = \frac{x \cdot y}{||x||\,||y||} \tag{3.3}$$

where $||a||$ is the Euclidean norm of vector a. This measure computes the cosine of the angle between the vectors x and y with a cosine value of 0 meaning that the two vectors are at 90 degrees to each otther and therefore are not a match. If the cosine value is 1 then the two vectors are identical and they are a perfect match.

3.3.4.3 Jaccard coefficient

The Jaccard coecient measures similarity as the intersection divided by the union of the objects. For text document, the Jaccard coecient compares the sum weight of shared terms to the sum weight of terms that are present in either of the two document but are not the shared terms.

$$sim(i, j) = \frac{q}{q + r + s} \tag{3.4}$$

where q is the number of variables that are positive for both objects, r is the number of variables that positive for i and negative for j and s is the number of variables that are positive for the j and negative for i.

3.3.5 Clustering methods

There are many clustering algorithms and usually the belong to one of the following categories, as they are described in [put reference to the book]:

- **Partitioning methods:** Partiotioning methods operate on a number of data points and form partitions of the data where each partition is a cluster. Usually, these methods have the restriction that each object must belong to a single cluster. They are only effective for a small to medium datasets.
- **Hierarchical methods:** A hierarchical method creates a hierarchical decomposition of the given set of data objects. These methods can be firther subdivided in agglomerative and divisive, based on how the decomposition is formed. Hierarchical methods suffer from the fact that erroneous merges or split cannot be undone.
- **Density-based methods:** These methods, unlike most other techniques, can find clusters of arbitrary shapes. They can do so by employing the notion of density. The main idea is to keep growing a cluster as long as the density of a 'neighborhood' exceeds some threshold. Therefore, clusters are dense regions of data points that are separated by low-density regions.
- **Grid-based methods:** Grid-based methods quantize the space into a number of cells which form a grid structure, All the clustering operations are performed on the grid structure. Since the provessing time is only dependent on the number of cells and not the number of data points these methods have fast processing time.

Below we outline the basic clustering methods that will be used in our system and we look into their key ideas that will guide our implementation.

3.3.5.1 k-Means algorithm

k-Means algorithm belongs to the partioning methods category. A partitioning method distributes the objects in D into k clusters, $C_1, ..., C_k$, such that $C_i \subset D$ and $C_i \cap C_j = \emptyset$ for $(1 \leq i, j \leq k)$. k-Means is a centroid-based technique whic means that a cluster, C_i, is represented by a centroid which can be the mean of all the points in the cluster. Practically, this representation is the centre point of the cluster.
More specifically, the k-Means algorithm starts by randomly selecting k of the

object in D, and each one represents a cluster centre. Then for each of the remaining objects in D, the algorithm calculates the distance between each of the k cluster centres. The closest cluster is selected and that point is assigned to this cluster. The mean of each cluster is re-calculated after a new point has been assigned to it. At this point the k-Means algorithm iteratively tries to improve the within cluster variation (improving an objective function) by reassigning all the objects according to the new cluster centres. The algorithm continues until it converges to the point where the clusters formed in the previous iteration are the same to the clusters formed in this iteration. Figure 3.4 (TODO:Put image ref) depicts a simple example of the k-Means algorithm with two clusters. In the first image, the two centroids, which are depicted as dark circles are initialised to a random position in the space. In the second image each of the data points is assigned to the nearest centroid and in this case A and B are assigned to the top centroid and C, D, and E are assigned to the bottom centroid. In the third image, each centroid's average is re-calculated and moved to its new position. When the assignments are calculated again, it turns out that C is now closer to the top centroid, while D and E remain closest to the bottom one. Thus, the final result is that A, B, and C belong in one cluster, and D and E in the other.

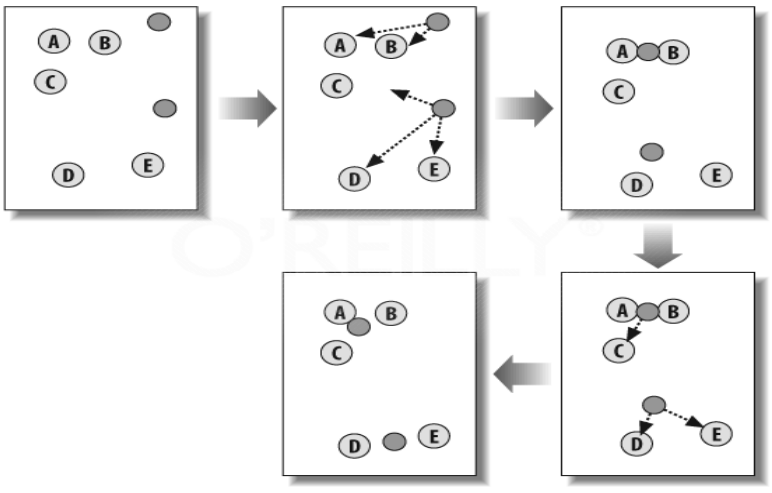


Figure 3.2: k-Means clustering with two clusters

The algorithm is not guaranteed to converge to a global optimum and the

results depend on the initial allocations of the k centres. The time complexity is $O(nkt)$ where n is the total number of of objects, k is the number of clusters and t is the number of iterations. Therefore, the k-Means algorith is relatively scalable.

3.3.5.2 DBSCAN algorithm

k-Means algorithm and hierarchical clustering algorithms are very good at clustering spherical-shaped clusters. However, they are not able to detect clusters of arbitrary shape such as the one below.

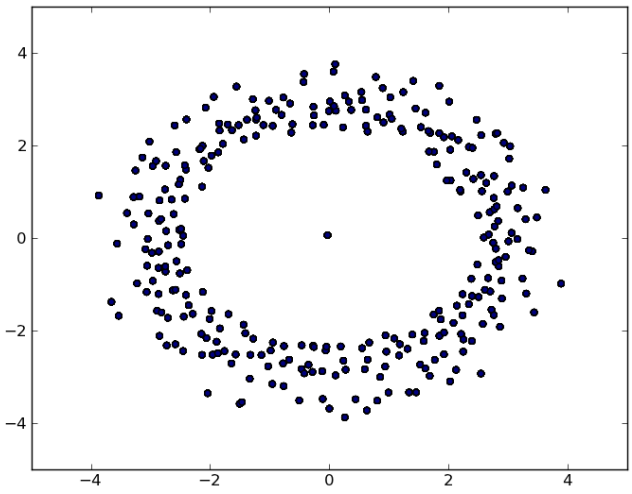


Figure 3.3: The DBSCAN algorithm belongs to the density-based methods which can find clusters of arbitrary shapes (not only spherical).

In order to alleviate the problem one have to look into density-based clustering algorithms. These algorithms model clusters as dense regions in the data space. One of the most popular algorithms of this kind is the Density-Based Clustering Based on Connected Regions with High Density (DBSCAN).
DBSCAN defines density of an object o as the number of objects close to o. DBSCAN finds core objects which are objects with dense neighborhoods. Then it connects those core objects and their neighborhoods to form clusters. In order

to assess density, DBSCAN uses a constant $\epsilon > 0$ which defines the radius of the neighborhood of an object o and then finds the number of objects in the neighborhood. If the number is above a threshold $MinPts$ then the neighborhood is considered dense. Given a set, D, of objects, we can identify the core objects using ϵ and $MinPts$. The problem of clustering then reduces to the task of finding dense regions using the core objects and their neighborhoods.

The complexity of the algorithm in our implementation is $O(n^2)$ but it can be reduced to $O(n \log n)$ if a spatial index is used. This means that this algorithm doesn't allow us to scale it up to large volume of data. The main disadvantage of the DBSCAN algorithm is that the user has to pass the ϵ and the $MinPts$ parameters which usually is a difficult task and some datasets are very sensitive to variations of these parameters.

3.3.5.3 Non-negative matrix factorisation algorithm

Non-negative matrix factorization (NMF) is not a clustering algorithm per se but it can be shown that it is equivalent to a relaxed form of k-Means algorithm. This algorithm takes as input the term-document matrix, constructed during clustering pre-processing and decomposes it into two other matrices. The first one is a feature-term matrix (or features matrix) and the other a document-feature matrix (or weight matrix). In the latent semantic space derived by the NMF, each axis captures the base topic of a particular document cluster, and each document is represented as an additive combination of the base topics. The cluster membership of each document can be easily determined by finding the base topic (the axis) with which the document has the largest projection value. The only constraint of the algorithm is that the two derived matrices should be non-negative.

In the context of document clustering we use matrix factorisation to reduce a large set of documents to a smaller set that captures their common topics (features) [put ref to the book Collective Intelligence]. We start with a term-frequency matrix and we wish to factorise this matrix to get the two matrices we mentioned above. The features matrix has a row for each feature and a column for each term

(word). An example of such a matrix is shown below. The values indicate how important a word is to a feature. Each feature represents a topic that is implied from a set of documents.

$$\begin{array}{c} \\ feature1 \\ feature2 \\ feature3 \end{array} \begin{array}{c} \begin{array}{cccc} football & ball & england & newspaper \end{array} \\ \begin{pmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & 3 & 1 \\ 0 & 2 & 2 & 1 \end{pmatrix} \end{array}$$

The weights matrix maps the features to the documents. It has a row for each document and a column for each feature. The values indicate how much each feature matches to each document. An example of the weights matrix is shown below.

$$\begin{array}{c} \\ Document1 \\ Document2 \\ Document3 \end{array} \begin{array}{c} \begin{array}{ccc} feature1 & feature2 & feature3 \end{array} \\ \begin{pmatrix} 1 & 5 & 3 \\ 3 & 1 & 1 \\ 0 & 2 & 3 \end{pmatrix} \end{array}$$

The original matrix can be reconstructed by multiplying the weights matrix by the features matrix, although the resulting matrix would not be an exact reconstruction of the original one but an approximation.

3.3.5.4 Online clustering algorithm

All the algorithms described above operate on the whole dataset, D, and sometimes this is not desirable or not even neccessary. Especially if the dataset is too large and we are concerned about performance and scalability. Therefore, a family of algorithms have been studied in order to modify common algorithms for scalability. These algorithms are called sequential or online clustering algorithms. By online in this context we mean an algorithm that does not keep all the data objects in memory at the same time, but processes them sequentially, keeping only a subset if them.
Suppose we are given a sequence of N tweets $T = t_1, ..., t_N$ where each t_i is a vec-

tor of a attribute values, and we want to split them into K clusters. Each cluster is described by a prototype vector $p_i = p_1, ..., p_f$ where $i = 1, ..., K$. Given that the task of finding good clusters is to minimize an objective distance function J the algorithm works as follows:

1. Pick the next example in the sequence T
2. Compute the distances from this example to all the cluster prototypes and pick the minimum one.
3. Update the prototype vector in order to come closer to the current example
4. Goto step 1

The implementation of our online algorithm is based on the work proposed in the paper "Improving the Robustness of Online Agglomerative Clustering Method Based on Kernel-Induce Distance Measures" by Zhang et al. They have used kernel-functions for distance measures and post-processing filtering is performed to remove noise from data. Extending their work, in our implementation we have tried to increase its scalability by allowing only a sliding window of n tweets to be in memory each time when the term-document vectors are calculated.

3.4 Automatic text summaries

Another integral component of our framework is the module repsonsible for generating automatic summaries of the events. In Chapter **??** we described the problem of event summarisation as the task that will extract the top tweets for an event that are most helpful for a human to understand the event. The term 'helpful' is vague and therefore we have defined three criteria, described in [TODO: put ref for the paper Selecting Quality Twitter Content for Events] that will evaluate qualitatively our summaries and guide our implementation choices. The three criteria are:

- **Quality:** It refers to the textual quality of the messages, which reflects how well they can be understood by a human.

- **Relevance:** It refers to how well a Twitter message reflects information related to its associated event.
- **Usefulness:** It refers to the potential value of a document for someone who is interested in learning more about an event.

3.4.1 Generating automatic document cluster summaries

In general, in automatic text summarisation we are interested in generating a shortened version of a large corpus or document but maintaining the integrity and meaning of the original text. There are to variations of the automatic text summarisation: the extractive summarisation method which produces summaries by choosing a subset of the sentences in the original document or documents and abstractive summarisation, where the information in the text is rephrased.

In our case the task is to reduce a large set of tweets in a smaller subset by ranking the individual tweets in a subset and selecting the top ranking ones. In our project we consider two methods extractive summarisation methods, the centroid-based summarisation and the LexRank algorithm.

3.4.1.1 Centroid-based summarisation

The centroid similarity approach computes the similarity of each message to its associated event cluster centroid. The cluster centroid is calculated by averaging the weight across all the document TF-IDF weigthed term-frequency vectors in a specific event. The method then selects the documents with the highest similarity value. The implicit assumption is that a clusters centroid emphasises important terms which are relevant to the event. For example a cluster describing the topic of some protesters giving flowers to policemen in Cairo would give higher weight to the terms "protesters", "flowers" and "policemen". Therefore documents with high similarity to these key terms are more likely to reveal key aspects of the event which is highly desired by the relevance and usefulness goals. Additionally, the quality criterion is satisfied since the centroid weights are based on frequency

across all messages, and therefore they contain no typos or spelling mistakes (which can greatly affect the quality of a document).

3.4.1.2 LexRank algorithm

The idea of the LexRank algorithm was proposed by [put ref to paper here] and it was successfully used for extractive summarisation tasks. Instead of the idea of a centroid LexRank uses the notion of centrality, which assumes that sentences which are similar to many other sentences in a cluster indicate "topicality". That is, the sentence that is the most similar to other sentences is the most topical. LexRank conceptualises sentences to be forming a graph which has the sentences as nodes and the edges indicate which sentences are similar (if the similarity is above a threshold we add an edge between two nodes). The sentence/node with the more edges has the higher centrality. Usually the cosine similarity measure is used for this purpose. Other methods such as degree centrality consider each edge as being an equal vote for centrality and therefore the most edges a node has the highest the centrality. However, LexRank takes a slightly less democratic approach by giving "prestigious" sentences more voting power. The higher the centrality of a node the more its vote counts. A way of formulating this idea is to distribute the centrality of a node to its neighbors. This can be formulated by:

$$L(u) = \sum_{v \in adj[u]}^{\infty} \frac{L(v)}{deg(v)} \tag{3.5}$$

where $L(u)$ is the centrality of node u, $adj[u]$ is the set of nodes adjacent to u, $deg(v)$ is the degree of node v.

In order to ensure that a "random walker" is not stuck during a walk in the graph we need to introduce a damping factor d which was first introduced by Page et al. (1998) in order to solve this problem. The equation for LexRank then becomes:

$$L(u) = \frac{d}{N} + (1 - d) \sum_{v \in adj[u]}^{\infty} \frac{L(v)}{deg(v)} \tag{3.6}$$

where N is the total number of nodes in the graph. The above equation is called the LexRank equation and is dened in a recursive manner, and can be computed via an iterative routine called the power method (the algorith is given in Chapter 4). This is essentially the same as the PageRank algorithm which is the backbone of the Google search engine with the only difference that the edges are undirected since the cosine similarity is a symmetric relation.

3.4.2 Detecting named entities and locations in documents

The summaries produced by the algorithms described above are a good representation of the event and can help a human to understand the event. However, sometimes some additional information is needed to comprehend an event. In Chapter **??** we described an event as a collection of attributes such as keywords, geographic location and entities involved. Therefore, we wish to be able to automatically extract these attributes for an event and included them in the summary of an event. The keywords describing the event are simply the words having the highests weight in an event cluster and can extracted very easily. For the named entities and locations we have to use a technique called named entity extraction which is part of the NLP algorithms. The method takes as input a piece of text and locates and classifies atomic elements in text into diffeent categories such as the names of persons, organisations and locations.

3.5 Twitter user classification

Different types of users interact and disseminate information daily on Twitter. Depending on the type of user we expect the content of a tweet to vary and consequently the quality of a tweet will vary too. For example renowned journalists are more likely to produce high quality tweets which easily be trusted as a reliable source of information. On the other hand tweets generated by a tweet bot might be considered unreliable since bots are known to release rumors. Therefore, it is crucial to be able to distinguish different types of users in order to ensure that we will extract high quality events that provide useful information and not just rumors. Additionally, it is interesting in its own right to investigate what is the

distribution of the users tweeting for an event. For example, we would like to find out whether activists can start and sustain an event discussion online or whether celebrities or political figures can spark the interest of other Twitter users. In order to do that we need to find a way to classify users.

Classification is the process of finding a model that describes and classifies data classes. The model is derived based on a set of training data (data objects which their class labels are known). Then the derived model is used to predict the class of an uknown instance which is presented to the learning system.

3.5.1 Decision trees

Decision Tree learning is a method for approximating discrete classiffication functions using a tree-based representation. We can make use of this method to inductively infer [put ref here] unknown relations within our dataset. A nice property of the decision trees is that they can also be represented as a set of if-then rules, therefore the task of interpreting them is an easy task for a human. A learned decision tree is able to classify a new instance by sorting it down the tree to the appropriate leaf node, then returning the classification associated with this leaf. Every node in the tree tests some attribute value of each new instance, and the branches of each node represent a possible value for that attribute.

Figure 3.6 illustrates a simple decision tree, which classifies a day's weather conditions according to wether they are suitable for playing tennis [put ref to mitchell's book]. If the decision tree is presented with an example of a day when the outlook is sunny, the temperature is hot, the humidity is high and the wind is strong then this example would be sorted down the leftmost branch and therefore would be classified as negative, i.e, this day is not suitable for playing tennis. Alternatively, the tree could be represented as if-then rules:

$$\begin{aligned}&(\text{Outlook} = \text{Sunny} \wedge \text{Humidity} = \text{Normal})\\ &\vee (\text{Outlook} = \text{Overcast})\\ &\vee (\text{Outlook} = \text{Rain} \wedge \text{Wind} = \text{Weak})\end{aligned}$$

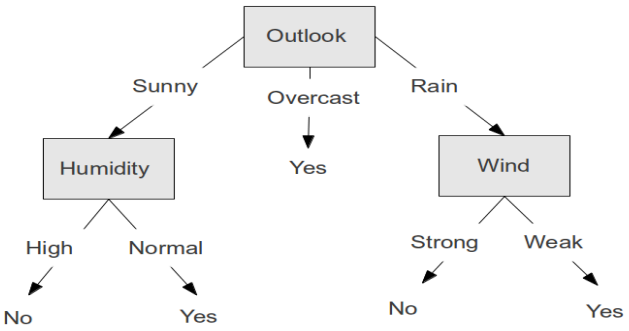


Figure 3.4: A simple decision tree.

3.5.2 Neural networks

[TODO explain Neural Nets]

3.6 Summary

Show a simple diagram with a lot of tweets becoming vectors and then those becoming clusters and then some of them events. This is the outout of the algorithm.

Chapter 4

Design and implementation

The aim of this chapter is to give a thorough description of our concrete implementation of the theoretical framework introduced in Chapter 3. We list and provide an explanation of the individual components and also explain our reasoning for certain design choices. In section 6.1 we present a proof-of-concept web application that we have developed in order to demonstrate how the different components can be integrated together to form a platform for event detection.

4.1 System Overview

Figure 4.1 shows an overview of the system architecture which is a pipeline of the individual components we described in Chapter 3. Each one of these components is depicted as an independent module in the figure. Initially, historical tweets from a service provider (Twitter API or another provider) are retrieved and stored in an appropriate format in the database. Subsequently, the system receives a stream of tweets from the database and processess and transforms them in a format that is appropriate for clustering. The next step in the pipeline is the actual clustering of the tweets in order to detect groups of tweets discussing the same topic. Then, the extracted clusters are processed in order to identify the events and generate their summaries. Finally, a visual representation of the results should be generated in order to aid understanding of the events.

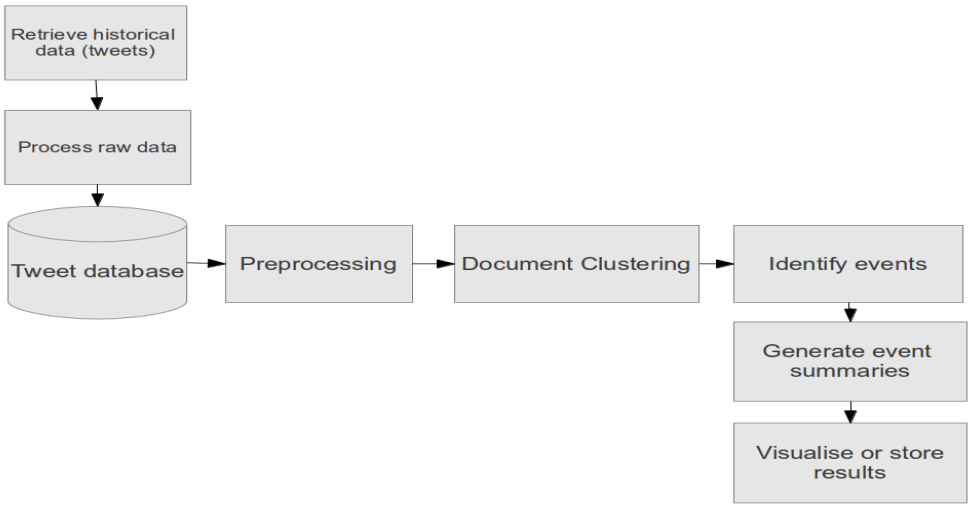


Figure 4.1: System overview - The event extraction system comprises of several independent components.

4.1.1 Tools

In the process of implementing the system we have mainly used our implementation of the algorithms but several third-party software libraries were used to implement the sub-components of the system. Here we describe the main tools we have used.

- **Python:**
- **Natural Language Toolkit (NLTK) :**
- **Lucene:**
- **Orange:**

4.2 Data Retrieval

A vital part of our system is the retrieval of a large amount of historical tweets. The first obvious choice is the Twitter API which provides tweets, user profiles and several metadata related to Twitter. They also provide a streaming API which is commonly used to collect tweets in real time. However, the main problem with the Twitter API is that it has a very restrictive limit policy (150 requests per hour) and it does not provide access to tweets posted more than a few days ago. This is a problem for our project since we require access to historical data. The solution is to use other archiving services and there are numerous possibilities. However, it is essential for us to have direct access to their database through an API and unfortunatelly, most of them do not provide an API. We have found that Topsy [1] provides an excellent API [2] and direct access to tweets from 2009 up to the present day. Additionally, Topsy API is free and the limit policy allows us to retrieve our data easily. Therefore, we have decided to use Topsy Otter API with its Python bindings.

4.3 Raw text processing

The raw tweets received from Topsy are not processed and therefore we must apply some pre-processing steps before storing them in the database. The reasons for pre-processing were outlined clearly in Chapter 3. Figure 4.2 depicts the sub-components of the raw text pre-processing module.

HTML and URL removal: Firstly, we need to clean the tweets from the URLs and HTML tags since they are useless for clustering. In order to do so we have used regular expressions which capture any possible format of URLs or HTML code.

Sentence segmentation and tokenisation: For the implementation of this module we have used the default sentence segmenter of NLTK and the Word-

[1] http://topsy.com/
[2] http://code.google.com/p/otterapi/

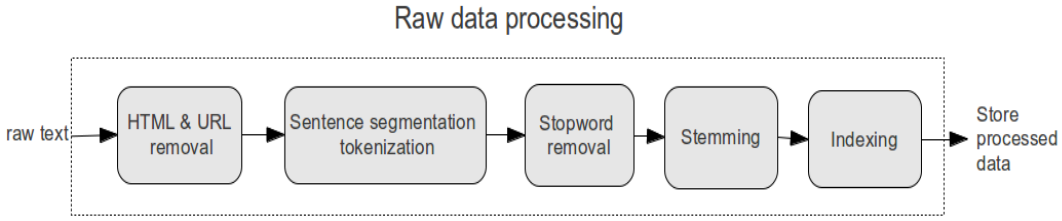


Figure 4.2: The raw data processing module - All the steps neccessary to convert raw documents to a format suitable for storage in a database.

PunctTokenizer to tokenise the resulting sentences. The reason we have used WordPunctTokenizer is due to the fact that it can handle alphabetic and non-alphabetic characters. Since it is common to use non-alphabetic characters in a tweet we could find an easy way to remove characters such as '.' and ','. Consider for example the following tweet:

> As #Egyptians prepare for #jan25 protests: #Mubarak has turned #Egypt into a #police state where torture & police brutality r systematic.

The output from this module will be a list of words containing the terms [**'as', 'egyptians', 'prepare', 'for' 'jan25', 'protests', 'mumbarak', 'has', 'turned', 'egypt', 'into', 'a', 'police', 'state', 'where', 'torture', 'police', 'brutality', 'r', 'systematic'**]. Note that characters '.', '#', '&' and ':' have been removed.

Stopword removal: The next step is to remove common English words that doesn't provide any information. NLTK provides a dictionary of the English stopwords and we have used it to filter out stopwords from the tweets. Using the list of words extracted for the example above the output of the stopword removal module will be: [**'egyptians', 'prepare', 'jan25', 'protests', 'mumbarak', 'turned', 'egypt', 'police', 'state', 'torture', 'police', 'brutality',**

'r', 'systematic']

Stemming: Once we have the list of our terms we can use a stemming algorithm to reduce the words to their root. Our implementation uses the widely used Porter stemmer which is also implemented in NLTK. The final list of words after the stemming becomes [**'egyptian', 'prepar', 'jan25', 'protest', 'mumbarak', 'turn', 'egypt', 'polic', 'state', 'tortur', 'polic', 'brutal', 'r', 'systemat'**]

Indexing: Just before storing the tweets in the database we take a last step which is to index the tweets. For each word occuring in our corpus we aggregate all the tweets that contain that term and the position of that word in the document. Effectively, we create a mapping from a word to a list of documents. In our implementation the inverted index is very important since it allows us to construct term-documment vectors easily and filter terms and documents. For example, using our index we can find the words that appear either too often or less frequently and filter them out. This is used to reduce the dimensionality of our dataset by removing unneccesary words. Alternatively, we can remove documents/tweets which contain keywords that appear too often or less frequently. In our implementation we have used PyLucene which is the Python equivalent of the Lucene indexing library. The library allow us to do what we have discussed above and in addition it provides helper functions such as the calculation of the TF-IDF weigtings for a dataset.

4.4 Clustering

So far we have managed to retrieve and store historical tweets in our database. The next module in our pipeline is to perform clustering on the tweets and identify clusters which discuss the same topic. The clustering module (Figure 4.3) is responsible to construct the term-frequency vectors by processing the dataset and subsequently to apply a clustering algorithm on the dataset.

There are several candidates algorithms for clustering documents and we have decided to implement four of them. The theoretical background of these algorithms is outlined in Chapter 3 and in Chapter 5 we present a thorough compar-

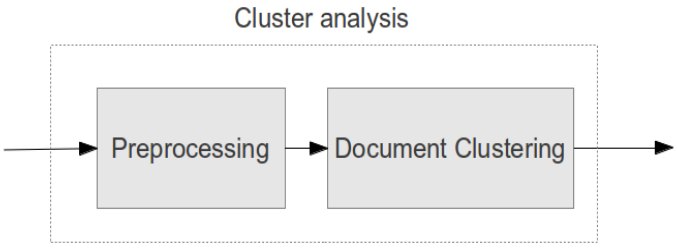


Figure 4.3: The cluster analysis module - The dataset is processed to construct the term-frequency vectors and then a clustering algorithm is used to identify the clusters.

ison of these algorithms with respect to their performance in clustering tweets. In the next section we provide our implementation of these algorithms and the components that are needed for clustering.

4.4.1 AbstractClusterer

The four different algorithms presented in this section are the k-Means, DBSCAN, Non-negative Matrix Factorisation (NMF) and online clusterers. Although these methods are fundamentally different they share some common functionality. For example the pre-processing steps, such as the construction of the term-frequency vectors is identical for all of them. Also, all of the algorithms output the clusters using the same format and therefore the methods for visualising the clusters will be identical. The architecture of the software components responsible for implementing the clustering functionality tries to incorporate this common functionality as well as accomodating the different core implementation of each algorithm. The UML diagram (Figure 4.4) below illustrates the different components of the clustering module. The AbstractClusterer class implements all the common functionalities such as add_document, construct_term_freq_matrix, plot_scatter and dump_clusters_to_file. Since each algorithm uses a different method to perform the actual clustering task, the AbstractClusterer class do not provide any implementation for the "run" function. This is the responsibility of each derived clusterer.

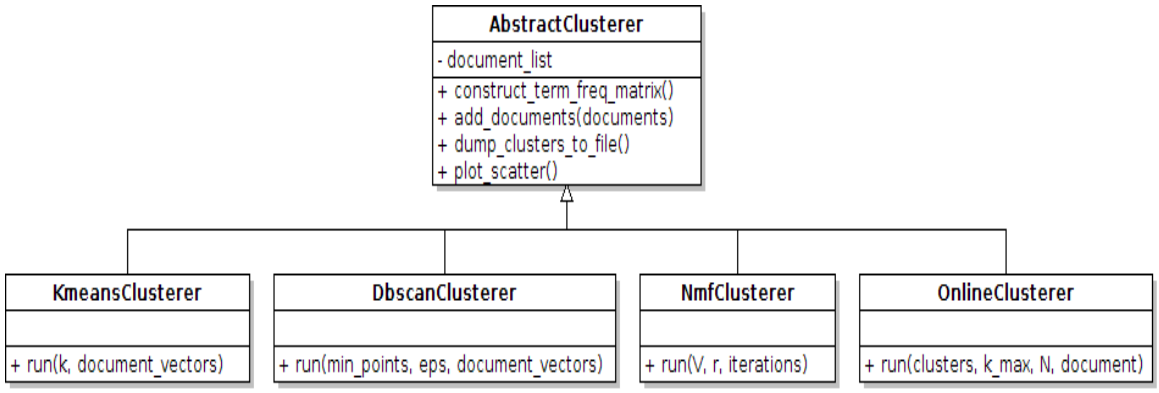


Figure 4.4: The four derived clusterers share common functionality which is implemented in the AbstractClusterer class.

The main functionality of the AbstractClusterer is to construct the term-frequency vector for each document and consequently the term-frequency matrix a which has a row for each document and a column for each distinct term in the dataset. This will convert the corpus in the vector space representation. Listing 4.1 shows the pseudocode for implementing the construct_term_freq_matrix function. We retrieve the dataset from the database and we find all the distinct terms that occur across all the documents. Then the term-frequency matrix is initialised and we iterate over the documents filling in the elements of the matrix with the TF-IDF weight of each term. The function returns the matrix a.

Listing 4.1: Pseudocode for constructing the term-frequency matrix for a dataset

```
def construct_term_freq_matrix(corpus):
  '''
  Inputs:
  corpus: The collections of documents we will cluster

  Outputs:
  a: A nxm matrix where n is the number of documents
     in the corpus and m the number of distinct terms
     in the corpus.
  '''
```

```
  terms = find_distinct_terms(corpus)
  a = initialise_term_frequency_matrix(rows=len(corpus),
                                       columns=len(terms))

  for i, document in enumerate(corpus):
    for term in document
      a[i][term] = tf_idf(term, document, corpus)

  return a
```

In the next pages we discuss the implementation of the concrete clusterers which implement the second sub-component of the cluster analysis module. We also show some preliminary results of the clustering process.

4.4.2 k-Means clusterer

Listing 4.2 shows the pseudocode for k-Means clustering algorithm. The algorithm works by randomly initialising k centroids and then finding out which document vectors are closest to the centroid. Then the centroid is recomputed based on the mean of the vectors belonging to this centroid and this mean value becomes the new centroid. An additional check is performed again to ensure that all the vectors are still belonging to that cluster after the recomputation of the centroid. This procedure is repeated until there are previous cluster membership has not changed. Note that the distance() function can be any of the three distance measures described in the previous chapter.

Listing 4.2: Pseudocode for k-Means algorithm

```
def run(k, document_vectors):
  '''
  Inputs:
  k: the number of clusters
  document_vectots: the set of n document
                    vectors I={i1,i2,..,in}
```

```
Outputs:
C: cluster centroids {c1, c2, ..., ck}
m: I --> C the cluster membership
'''
C = random_centroid_initialisation()

distances = []
for vector in document_vectors
  for c in C:
    dist = distance(c, vector)
    distances.append(dist)
  m[vector] = min(distances)

while has_changed(m):
  for c in C:
    c = recompute_centroid(c, vectors_of(c))

  for vector in document_vectors
    for c in C:
      dist = distance(c, vector)
      distances.append(dist)
    m[vector] = min(distances)
return C, m
```

4.4.3 DBSCAN clusterer

Listing 4.3 shows the pseudocode for the DBSCAN clustering algorithm. Initially, all documents are marked as unvistied and the cluster list is empty. Then the algorithm randomly selects a new document vector, marks it as visited and finds all its neighbors than are within a distance eps. The set of neighbors is called -neighborhood and we denote it here by N. In order to calculate the distance we use one of the three predefined distance measures. If N does not contain at

least min_pts then we mark this vector as a noise point. Otherwise, a new cluster is created which contains this vector and all the objects in N are added to a candidate set. DBSCAN iteratively adds to the new cluster those documents in the N that do not belong to any cluster. For any object in the -neighborhood DBSCSAN checks again its own -neighborhood N' and if it has at least min_pts those document vectors are added to the N. This continues, with DBSCAN adding new documents in the new cluster, until N is empty. Finally, to find the next cluster DBSCAN selects a new unvisited document vector and continues the same process until all vectors have been visited.

Listing 4.3: Pseudocode for DBSCAN algorithm

```
def run(min_points, eps, document_vectors):
  '''
  Inputs:
  eps: the radius parameter
  min_pts: the neighborhood density threshold.
  document_vectors: the set of n document vectors
                    I={i1,i2,..,in}

  Outputs:
  C: the clusters and the document vectors
     belonging to them
  '''
  C = [] #The cluster list
  mark_all_vectors_as_unvisited(document_vectors)

  while not all_is_visited(document_vectors):
    vector = pick_random(document_vectors)
    mark_as_visited(vector)
    neighbors = get_neighbors(document_vectors, vector)
    if len(neighbors) >= eps:
      new_cluster = create_cluster(vector)
      C.append(new_cluster)
```

```
      for n_vector in neighbors:
        if not is_visited(n_vector)
          mark_as_visited(n_vector)
          n_neighbors=get_neighbors(document_vectors,n_vector)
          if len(n_neighbors) >= eps:
            neighbors.append(n_neighbors)
        if not_in_cluster(n_vector):
          new_cluster.add(n_vector)
    else:
      mark_as_noise(vector)
  return C
```

4.4.4 Non-negative matrix factorisation (NMF) clusterer

Our implementation is based for NMF is based on the paper [put ref here for Learning the parts of objects by nonnegative matrix factorization]. The method accepts as input the term-frequency matrix V the number of basis vectors to generate r and the number of optimisation iterations. The algorithm starts from non-negative initialisations for W and H and then iteratively updates W and H until a factorisation $V \approx WH$ such that $|V - WH|$ is minimal. If the number of maximum iterations has not been reached and the error did not change since the last iteration then the algorithm halts and return W and H. Listing 4.4 shows the pseudocode for the NMF algorithm. The notation $A.T$ indicates that we shoud take the transpose of the matrix A.

Listing 4.4: Pseudocode for the NMF algorithm

```
def run(V, r, iterations):
  '''
  Inputs:
  V: the matrix to factorize
  r: number of basis vectors to generate
  iterations: number of optimisation
              iterations to perform
```

```
    Outputs:
    W: a set of r basis vectors
    H: represenations of the columns of V in
       the basis given by W
    '''

    C = size(V,1) #dimensionality of examples (# rows)
    N = size(V,2) #number of examples (columns)

    W = rand(N,r);
    H = rand(r,C);

    previous_error = 0
    error = 0
    for i in xrange(iterations):

      #Update W
      W2 = dot(dot(W, H), H.T) + 10**-9
      W *= dot(V[:,:], H.T)
      W /= W2

      #Update H
      H2 = dot(dot(W.T, W), H) + 10**-9
      H *= dot(W.T, V[:,:])
      H /= H2

      previous_error = error
      error = sqrt( sum((V[:,:] - dot(W, H))**2 ))

      if i > 1 and error:
        if previous_error == error:
          break
```

```
    return W, H
```

4.4.5 Online clusterer

The rest of the algorithms operate on the dataset as a whole. Since we are interested in clustering a vast amount of documents these approaches are expensive. However, we can use a category of clustering algorithms called sequential or online clustering methods. The main difference is that they cluster each individual example sequentially by updating the clusters once this specific example is presented to the system. Therefore, an online algorithm is scalable. Usually, this kind of algorithms are centroid-based and the mean values of centroids are updated using a moving average. This is the the main difference of an online clusterer and this why they are faster. However, in our case whenever we are presented with a new document the term-frequency vectors of all the previous documents must be updated. This operation is computationally expensive and therefore instead of considering all the previous document one can use sliding window approach. For example if we are presented with the *ith* document, denoted as x_i and our window size is N then only the documents $\{x_{i-1}, x_{i-2}, ..., x_{i-N}\}$ will be used for clustering. This can provide us with an extra performance boost.

Our implementation is illustrated in pseudocode in Listing 4.5 and it can be summarised in three main steps:

1. Find the closest centroid and move it closer to the new document vector.
2. Merge the two closest centroids, c1 and c2, which means that we are left with a redundant centroid c2.
3. Set the redundant centroid equal to the document vector.

These three steps satisfy three important criteria. Firstly, the within-cluster variance is minimised by Step 1 and the between-cluster distance is maximised by Step 2. Finally we capture the changes in the data distribution after a new document is clustered by Step 3 since we are treating each new document as an

indication to a potential new cluster.

In the algorithm there are two cases when we need to update the location and the size of an existing centroid. The first one is when a centroid is the closest to a new document and it must be moved closer to it. The location of the closest centroid, c, is updated according to the following update rule location:

$$c_{center} \leftarrow c_{center} + \frac{d_{center} - c_{center}}{c_{size} + 1} \tag{4.1}$$

where c_{center} is the centroid's location, c_{size} is the number of documents this centroid represents and d_{center} is the document that we are trying to get closer to. The size of a centroid is increased by one each time a new document is assigned to it. The second case for updating is when two clusters, c_1 and c_2 need to be merged. The update rules for the location and the size are then defined as:

$$c1_{center} \leftarrow \frac{c1_{center} \times c1_{size} + c2_{center} \times c2_{size}}{c1_{size} + c2_{size}} \tag{4.2}$$

$$\tag{4.3}$$

$$c1_{size} \leftarrow c1_{size} + c2_{size} \tag{4.4}$$

Listing 4.5: Pseudocode for the online clustering algorithm

```
def run(clusters, k_max, N, document):
    '''
    Inputs:
    N: the size of the document window
    k_max: the maximum number of clusters we can identify
    document: the new document to be clustered

    Outputs:
    clusters: returns the updated clusters
    '''
    #construct a new term-frequency vector based on the new
```

```
    #document and consider only the last N documents.
    vector = construct_term_freq_vector(self.document_list,
                                        document,
                                        N)

    distances = []
    for centroid in clusters:
      dist = distance(centroid, vector)
      distances.append(dist)

    #find the closest centroid (Step 1)
    closest = min(distances)
    closest.add(vector)

    #update the centroids center
    closest.size += 1
    closest.center += (vector-closest.center)/closest.size

    if len(clusters)>=k_max and len(clusters)>1:
      #merge the most similar clusters (Step 2)
      merged = merge_closest(clusters)


    #create a new cluster for this document (Step 3)
    newc = Cluster(vector)
    clusters.append(newc)
    return clusters

def merge_closest(clusters):
    c1, c2 = find_closest()
    c1.center=(c1.center*c1.size+
               c2.center*c2.size)/(c1.size+c2.size)
    c1.size+=c2.size
```

```
    c2.remove() #c2 is now merged with c1 so remove it
    return c1
```

4.4.6 Clustering results

[TODO: add screenshots from the clustering results]

4.5 Identifying events

[TODO: Complete this section]

4.6 Generating automatic summaries

So far we have managed to retrieve historical tweets from Twitter and identify clusters of topics that were discussed in them. After that we have filtered out the irrelevant clusters and kept the important ones which are considered as the events. Some of these events contain a lot of tweets and therefore it would be extremely tedious for a human observer to understand what happened during that event. Therefore, at this point we must implement our automatic summary generators. The main task here is to find the most important tweets, keywords and named entities. In the next sections we describe how we have implemented the functionality to extract this information. An overview of the summarisation module is shown in Figure 4.5.

4.6.1 Ranking tweets

The first sub-component of the summarisation module is to device a way to rank the tweets in order to extract the most *relevant*, *useful* and *high-quality* tweets. These are the three criteria that our summarises must meet. We described them in more detail in Chapter 3 and in the following sections we will look into how we implemented two algorithms to meet these requirements.

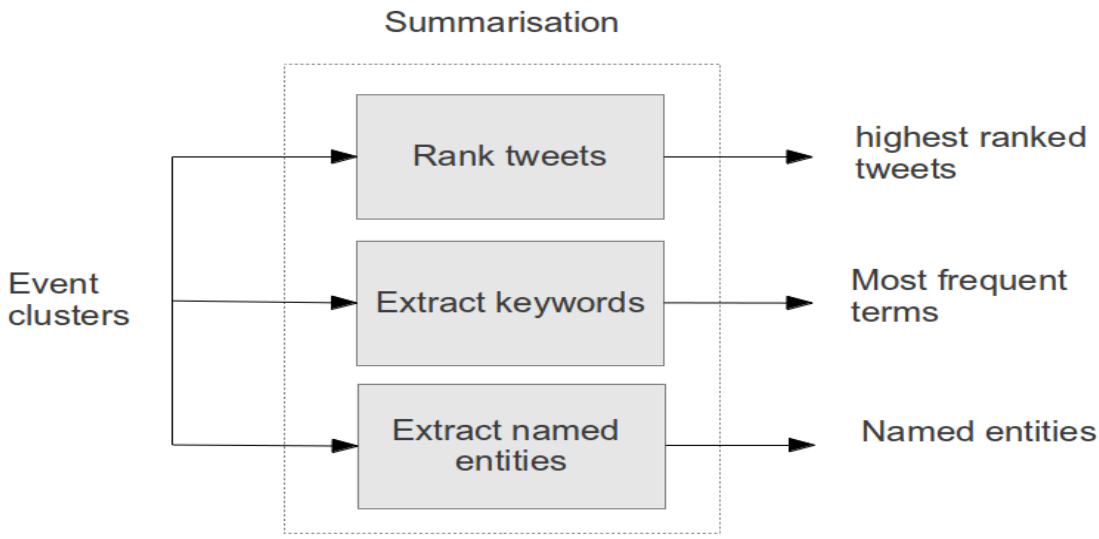


Figure 4.5: The summarisation module receives the event clusters and outputs the highest ranked tweetsm keywords and entities in each cluster.

4.6.1.1 Abstract summariser

Again, the two summarisers share some common functionality and like the clusterers we define an AbstractSummariser which is responsible to provide these common functions. The derived summarisers will inherit these functions and also provide their own concrete implementation of the run (the function which performs the actual ranking) function. Figure 4.6 illustrates the software architecture.

4.6.1.2 Centroid-based summariser

The main idea behind this summariser is that tweets closer to the event's cluster centroid are more likely to be useful and relevant to the event than others which are distant. Also, the third criterion is met since the centroid's weights are calculated based on the average of the tweets' vectors and therefore typos and spelling mistakes are more likely to be ruled out. The implementation of this algorithm is straightforward and is shown in Listing 4.7. First we retrieve the term-frequency vectors for all the documents in the cluster we want to rank.

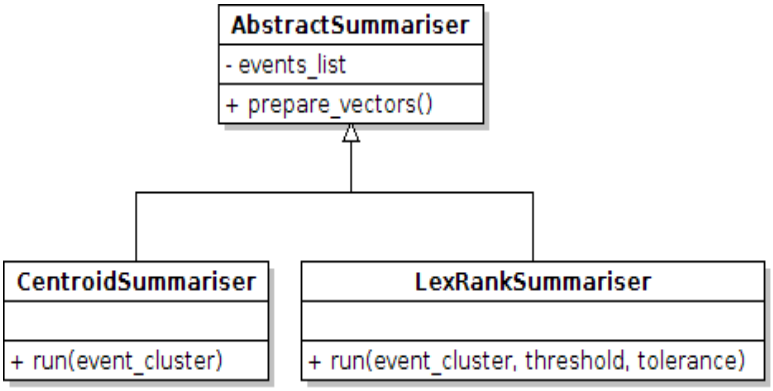


Figure 4.6: The two derived summarisers share common functionality which is implemented in the AbstractSummariser class.

Then we calculate the centroid by averaging the individual vectors. Finally, the cosine similarity of each document and the centroid is calculated and we return a sorted list of similarities. The similarity acts as a ranking score since the closer a document is to a centroid the highest its ranking.

Listing 4.6: Pseudocode for the centoid-based summariser.

```
def run(event_cluster):
    '''
    Inputs:
    event_clusters: The cluster for which we want to rank
                    the tweets.
    Outputs:
    ranked: A list of ranked tweets in descending order.
    '''
    vectors = event_cluster.get_document_vectors()

    #Calculates the centroid by averaging the individual
    #vectors of alll the documents in the cluster.
    centroid = calculate_centroid(vectors)

    similarities = []
```

```
  for vector in vectors:
    sim = cosine(vector, centroid)
    similarities.append(sim)

  return sorted(similarities)
```

4.6.1.3 LexRank summariser

As we have described in Chapter 3 LexRank is an algorithm where we dene a graph that has nodes for the sentences to be summarised the edges are placed between two sentences that are similar to each other. We can then rank all the sentences based on the expected probability of a random walker visiting each sentence using equation 3.6.

The main differentiation in our implementation to the usual LexRank algorithm is that we scale the probability of jumping to the next node based on the cosine similarity of two nodes. The reason we do this is that we want to favor jumping to a node that discuss a more similar topic. We can implement this alteration by scaling the summation in the equation by the cosine similarity of the nodes u and v. Since the similarity is in the range $[0, 1]$ an identical node it won't be discounted whereas a node with similarity equals 0 it will be ignored. The equation the becomes:

$$L(u) = \frac{d}{N} + (1 - d) * sim(u, v) \sum_{v \in adj[u]}^{\infty} \frac{L(v)}{deg(v)} \tag{4.5}$$

Listing 4.7: Pseudocode for the centoid-based summariser.

```
def run(event_cluster, threshold, tolerance):
  '''
  Inputs:
  event_clusters: The cluster for which we want to rank
                  the tweets.
  Outputs:
```

```
    ranked: A list of ranked tweets in descending order.
    '''

    vectors = event_cluster.get_document_vectors()
    n = len(vectors)
    adjacency_matrix,degree=calculate_similarities(vectors,
                                                   threshold)

    for i in xrange(n):
        for j in xrange(n):
          adjacency_matrix[i][j] = adjacency_matrix[i][j]
                                                 / degree[i]

    ranked = power_method(adjacency_matrix, tolerance)

    return ranked

def power_method(self, m, epsilon):
    n = len( m )
    p = [1.0 / n] * n
    while True:
        new_p = [0] * n
        for i in xrange( n ):
            for j in xrange( n ):
                new_p[i] += m[j][i] * p[j]
        total = 0
        for x in xrange( n ):
            total += ( new_p[i] - p[i] ) ** 2
        p = new_p
        if total < epsilon:
            break
    return p
```

4.6.2 Named entity and keyword extraction

The other two parts of the summarisation module is the named entity and keyword extraction. The keyword extraction task is straightforward since we mereley want to find the keywords in the event cluster that have the heighest TF-IDF weights. Therefore, it is just a matter of looking up the TF-IDF weighted term-frequnecy vectors and find the best keywords. For the name entity extraction task we decided to use the built-in capabilities of NLTK. More specifically, NLTK contains functions which can implement part of speech tagging and then extract named entities based on these tags. The pseudocode for this task is not included since it has not been implemented by us. For example if the input to the system is a tweet with the following content: "Most protesters in Cairo have gathered in front of the maspiro building, protest in Alex is also picking up #jan25" then the output will be "Cairo, GPE" meaning that Cairo is identified as a location. Named entity extraction is not always accurate and therefore we must be careful when we decide whicj entities to output. For example in the above tweet the system could erroneously identify the word "Alex" as a human name. However, in reality the author of the tweet mentioned the name of the city Alexandria in Egypt. Therefore, in order to minimise the possibility of a misinterpretation we decided to output only the named entities that appear most frequently in the cluster's tweets.

4.7 Classifying users

In our system user classification is important since it can allows us to understand the events even better. The user distribution in an event can give us insight into what has been discussed and ascribe quality to the events. For example, events that include a large amounts of tweets from journalists or media organisations are more likely to be of high quality with respect to reliability and descriptive quality.

Since we want to investigate the events that took place during the Arab Spring we know that the important types of users we need to consider are: Media organisations, Journalists, Activists, Celebrities and Common individuals. In order, to

classify a user to one of these categories we must know its profile. A user profile is an N-dimensional vector of attributes that are used to discriminate different user types. We have narrowed down these attributes to be:

- Retweet ratio
- Link ratio
- How ofthen this user gets retweeted?
- Ratio of replies
- Ratio of mentions
- Followers to followees ratio

The first attribute is a simple count of the number of retweets of a user divided by the overall number of their tweets. Retweets are most commonly used by common individuals to share a tweet they liked but they are rarely used by media organisations. Therefore, if the retweet ratio is very small we may suspect that this is a media organisation. The second attribute is also an important feature as we can very easily identify a media organisation by the number of tweets that contain links. The reason is because Twitter accounts for media organisation are used to share links of their articles. If a user gets retweeted too often this is usually an indication that they are popular or their tweets are considered interesting. Therefore, the number of times a user gets retweeted can be a measure of its popularity and therefore if we observe a user which gets retweeted frequently we may assume they are a celebrity, a journalist or even an activist. A user can reply to another user by using the "@" character followed by the username and the same holds for when a user wishes to mention another user in a tweet. The difference between a mention and a reply is that replies start with the "@username" pattern whereas in tweets that mention someone this pattern can appear anywhere. Replies are usually used by common people to start and maintain conversations but are very rarely used by celebrities and media organisations. Finally, one of the most discriminating features is the followers to followees ratio. The reason for this is because celebritiesand media organisations have very high followers to

followees ratio but common people have very low ratio. Activists and journalists are in the middle of the spectrum.

The task now becomes to collect a training dataset which contains labelled user profiles. This will be used to train our classifiers. The obvious way to collect the data is by manually looking up users on Twitter and constructing their profiles. However, this will be a tedious and time consuming process and thus we decided to automatically collect our data. We have done that by crawling a website called Twtrland [1] which provides Twitter user statistics.

4.7.1 Crawling user profiles

A web crawler is a type of software agent that browses websites in a methodical and automated way in order to retrieve up-to-date information. This technique is used in our case to retrieve user profiles from Twtrland. Our web crawlers are constructed using an open source crawling library for Python called Scrapy [2]. A crawler is given the URL of user profile on Twtrland and then it parses the HTML code of that page and automatically retrieves the information we need. A typical page on Twtrland is shown in Figure 4.7 and the rectangles indicate the information we collect.

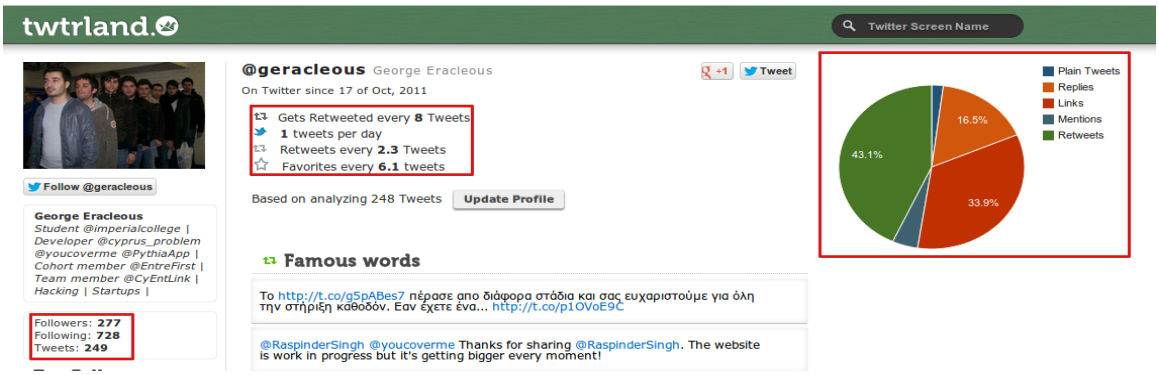


Figure 4.7: A typical Twtrland user profile. The information in the circles is retrieved by our crawler and used in the construction of the user feature vector.

[1] http://twtrland.com/
[2] http://scrapy.org/

4.7.2 Constructing the feature vectors

Based on the information we have collected using our crawlers we can very easily construct a feature vector for each user. The vector has one column for each of the six attributes we defined above. Some examples of feature vectors are shown in Table 4.1. The columns represent the retweet ratio, link ratio, how often this user gets retweeted, replies ratio, mentions ratio and Followers to Followees ratio (FF) respectively.

User	**Retweets**	**Links**	**Retweeted**	**Replies**	**Mentions**	**F/F**
nytimes	0.0540	0.9091	0.6700	0.0043	0.0033	7089.2700
aplusk	0.0970	0.4042	0.5000	0.1971	0.0880	14151.1944
bencnn	0.2702	0.0974	0.4500	0.0988	0.0093	186.1068
alaa	0.2500	0.0407	0.8700	0.5120	0.0733	134.4679

Table 4.1: The feature vectors of different types of users users

The first user is the Twitter account of New York times and it is representative of the feature vectors for media organisations. Low retweet rate and a large proportion of their tweets contain links as expected. Their Followers to Followees (FF) ratio is very high and one can easily infer by their low reply and mention ratii that they almost never engage in a conversation. The second example is Ashton Kutcher, a famous Hollywood actor. His FF ratio is very high as expected and another important observation is that his replies ratio is relatively high meaning that he usually chats with other users. The username bencnn belongs Ben Wedeman a famous American journalist working for CNN. An interesting observation is that his tweets are retweeted a lot. More specifically, he gets retweeted every 0.45 tweets which is the highest rate from all four examples. This is expected as he usually tweets content that people find interesting and worth sharing. Finally, the fourth example profile belongs to Alaa Abd El-Fattah, a well known Egyptian political activist. In his profile we can observe relatively low FF ratio but very high replies ratio which is expected as he usually engages in conversations with other users.

4.7.3 Software architecture

Figure 4.8 depicts the software architecture of the user classification module. Once again we wanted to make it as easy as possible to swap implementations of classifiers and therefore the AbstractClassifier class factors the common functionality of classifiers leaving the concrete implementations to the derived classes.

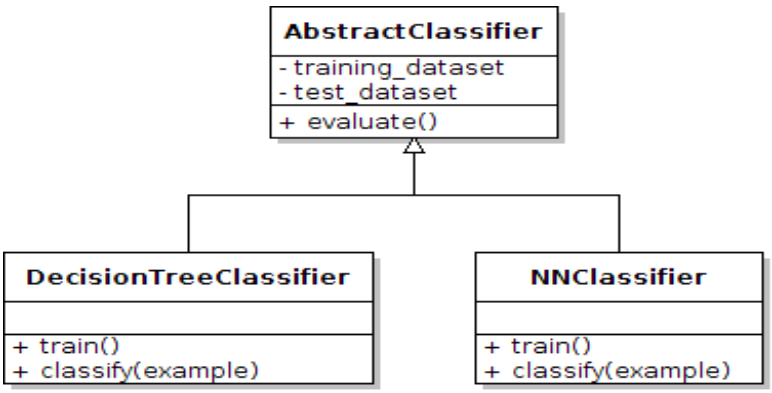


Figure 4.8: The two derived classifiers share common functionality which is implemented in the AbstractClassifier class.

4.7.3.1 IDE3Classifier

One of the most popular decision tree learning algorithms is the ID3 and there are a lot of open source implementations of it. It learns a decision trees by constructing it topdown iteratively and at each iteration it tries to find the best attribute from the feature vector to be tested, i.e. the attribute that helps the most in disciminating the examples). It selects this attribute using a statistical test to determine how well it alone classifies the training examples.

We have decided to use Orange's decision tree implementation which can easily extended to incorporate new functionality. In particular, in our implementation we have written wrapper functions for training and testing a classifier around Orange's functions. We train the classifier with the labelled user examples we have collected using our crawlers and then using the learned tree it can classify unkown examples. A part of the learned tree based on our training dataset is

shown in Figure 4.9. ID3 selected the root node to be the FF ratio and this is expected since this attribute is usually enough to get an indication of the type of the user. If the FF ratio for an example is lower than 143.068 then we move to the left branch otherwise to the right one. We keep moving downwards the tree until we hit a leaf node which gives the final classification for this example. For instance if an example has FF ratio higher than 143.068, it gets retwetted in less than 0.070 tweets and the proportion of links in the tweets is lower than 0.043 then the we hit the leaf node labelled with the number 0. This number indicates that the class of this example is "Celebrity".

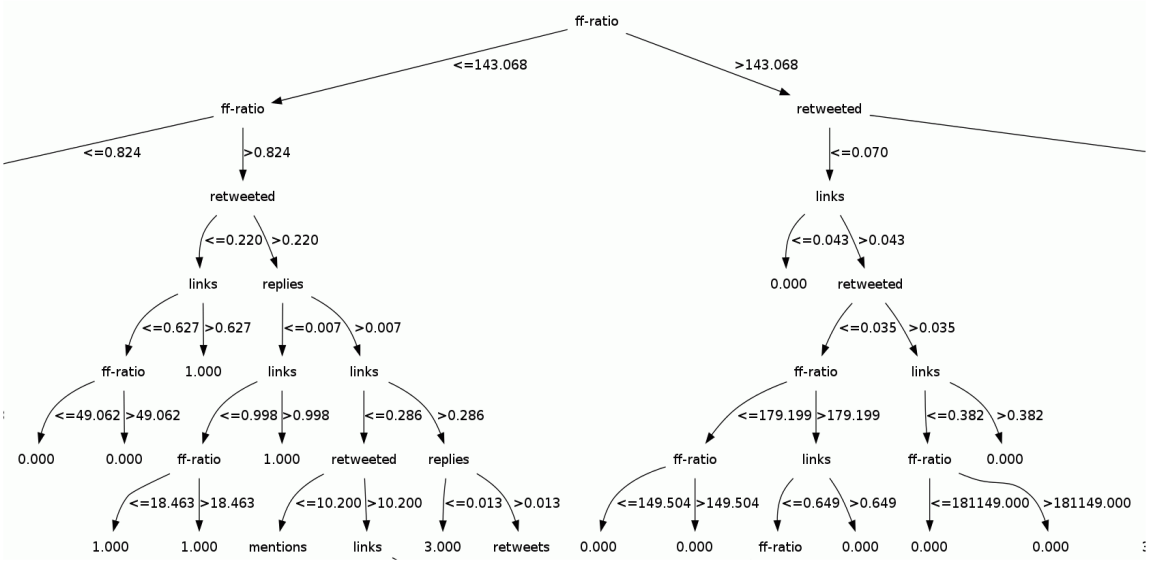


Figure 4.9: Part of the decision tree learned by our classifier.

4.7.3.2 NNClassifier

4.8 Summary

Chapter 5

Evaluation

5.1 Document clustering evaluation

5.1.1 Background

Explain the metrics and the method used.

5.1.2 Results and discussion

Present and discuss the results

5.2 Cluster labelling (summarization) evaluation

5.2.1 Background

Explain the metrics and the method used.

5.2.2 Results and discussion

Present and discuss the results

5.3 Twitter user classification evaluation

5.3.1 Background

Explain the metrics and the method used.

5.3.2 Results and discussion

Present and discuss the results

5.4 Optimisations

5.5 Summary

Chapter 6

Case study

6.1 Developing a proof-of-concept web application

6.2 Case study on a synthetic dataset

6.2.1 Results and discussion

6.3 Case study on a real dataset

6.3.1 Results and discussion

6.4 Summary

Chapter 7

Conclusions

Here I put my conclusions ...

Appdx A

and here I put a bit of postamble ...

Appdx B

and here I put some more postamble ...

References