University of Dublin, Trinity College
School of Computer Science and Statistics

IMPROVING REAL-TIME NEW EVENT DETECTION IN SOCIAL STREAMS

Shane Fitzpatrick

Dissertation

Presented to the
University of Dublin, Trinity College
in fulfilment
of the requirements
for the Degree of

Master in Computer Science

May 2014

Declaration

I, Shane Fitzpatrick, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Shane Fitzpatrick May 22^{nd}, 2014

Acknowledgments

I would like to thank my supervisor Prof. Vincent Wade for his advice during this dissertation. I would also like to thank my family and friends for their advice and support throughout this course.

Shane Fitzpatrick

Summary

Social streams have experienced a surge in popularity in recent years. There is an increasing number of electronic documents being sent through these streams. The noise and volume of documents make it infeasible for a human to sort through and extract documents which describe significant events. New event detection is the problem of identifying documents in these streams that report on real world events. This work aims to improve the real-time detection of new events with high-volue social streams. The objectives of the work are as follows. To perform a survey of the state of the art in the area of on-line NED and social streams. To create an on-line NED system that operates with social streams and is capable of detecting structured live events. To evaluate this system in comparison to a highly published baseline system.

The research approach undertaken for this work was to survey the state of the art in new event detection and social streams, to design and implement a system capable of detecting structured live events in real-time and to evaluate this system in comparison to a baseline system. A critical analysis of the state of the art is included and used to motivate the design and implementation of the system presented. The evaluation of this system uses the standard Precision, Recall, Acurracy and F_1 metrics, along with the $s - test$ and $S - test$ significance tests.

The findings of this work can be summarized as follows. The system presented throughout this report represents a significant improvement over a highly published baseline system. This improvment is dramatic when comparing the detection of low volume events. Furthermore, the evaluations of the system provide insights into crucial parameters of the various state of the art algorithms that were used.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Social streams have experienced a surge in popularity in recent years. There is an increasing number of electronic documents being sent through these streams. The noise and volume of documents make it infeasible for a human to sort through and extract documents which describe significant events. Sakai et al. found that real world events are often reported on social streams before official news sources [3]. With the proliferation of real time information available on social streams, it is crucial to have systems capable of detecting new events reported by users of these streams.

New event detection (NED) is the task of identifying news documents which describe previously unreported events. The objective is to classify each document as either *new* or *old*. This classification can occur in either a *retrospective* or *on-line* setting [4]. Retrospective solutions process a complete set of documents before making the individual classifications. On-line solutions watch a stream of documents and classify each document as it arrives in real-time. These streams are can be a reliable newswire or a social stream. A newswire represents a low volume, edited stream of news document produced by some trusted entity. Social streams are high volume, noisy streams of documents produced by laypeople. The quantity of documents and noisy nature of social streams has resulted in a paradigm shift in this field.

Social streams introduce a myriad of new challenges for NED. Users generate a large number of documents at any given time, e.g. the Twitter stream has 500 million daily tweets (documents)[1]. It is unrealistic to keep this volume of documents in the memory of a NED system. Furthermore, these streams do not have any central editing and users are free to post documents with any content. These documents often do not contain any news and are describing the users thoughts at that moment. Studies have shown that as little as 3.6% of the documents in these streams are

[1]https://about.twitter.com/company

news-related [5]. NED systems must thus manage the volume of documents and the noisy, spam-filled nature of social streams. Existing NED algorithms and methods need to be re-evaluated with these factors in mind.

1.2 Research Question & Objective

This work investigates how events can be detected in high volume social streams. The focus of the work is on recurring live events with a defined structure and set of possible sub-events. Sports match are examples of such events. The time-frame of the match and the set of possible events that can occur during it, e.g. a score, are known in advance.

The objectives of this work are as follows.

- To perform a survey of the state of the art in the area of on-line NED and social streams.
- To create an on-line NED system that operates with social streams and is capable of detecting structured live events.
- To evaluate this system in comparison to a highly published baseline system.

1.3 Research Approach

The research approach for this work is as follows. Firstly, to perform a survey on the state of the art to identify the key techniques used in this area. Secondly, to take a highly published baseline system with the same focus as this work and iterate on its design. The techniques identified in the survey are used to critique the baseline system and guide the iteration process. Thirdly, to implement the baseline system and the new system. Finally, to preform a comparative evaluation between both systems using a dataset with annotated real world events. Various elements of the new systems design will be evaluated during this stage also.

1.4 Structure of this Report

The remainder of this report is structured as follows. The second chapter gives a background into NED research and presents the state of the art in this area. This chapter also includes a discussion on social streams, in particular the Twitter stream. The third chapter presents the architecture of the system presented in this work. A chapter discussing the implementation of the system immediately follows. The penultimate chapter evaluates the baseline system and the new system with comparitive experiments. Finally, this report concludes with a discussion of the objectives achieved and areas for future work.

Chapter 2

Background & State of the Art

This chapter first provides a background into the new event detection (NED) task. The most common approaches to solve this task are then presented, i.e. clustering documents within vector space. A discussion of on-line new event detection problems immediately follows. This discussion includes a background into the locality sensitive hashing algorithm and its uses within NED. This chapter then presents the Twitter social stream and discusses its properties and challenges when in use with NED. Various NED system that use Twitter are then presented, and a baseline system for comparison with this work is discussed. This chapter concludes with a summary and critical analysis of this survey.

2.1 New Event Detection (NED)

From 1997 to 2004, DARPA funded a program, called Translingual Information Detection Extraction and Summarization (TIDES), to encourage research in the analysis of electronic news documents. NED is one of the five categories in the *Topic Detection and Tracking* (TDT)[1] subprogram in this domain. TDT research was focused on analyzing and organizing event-based news documents from a stream [6], and NED is the task of identifying the first document which describes an unreported event. There has been a recent resurgence in research in this area as a result of the popularity of social streams, and the domain experienced a paradigm shift wherein established solutions to the NED task wouldn't transfer to modern high volume social streams.

NED applications assign a confidence score to each document which depicts the likelihood of that document describing a new or old event. Applications fall into two categories: *Retrospective* or *On-line*. Retrospective applications process a complete corpus of documents before assigning the confidence scores. On-line applications process each document as it arrives and immediately assigns to it a score [7]. On-line applications make use of streaming algorithms [8], wherein documents arrive one at a time and must be processed before future documents are received.

[1]http://www.itl.nist.gov/iad/mig/tests/tdt/

Researchers believe that NED is the most difficult of the TDT tasks [9]. The state of the art in NED applications typically represent documents using the Vector Space Model (VSM) (see below). Clustering algorithms with the VSM determine a documents NEW/OLD classification.

2.1.1 Vector Space Model

Using the VSM is the standard approach for document classification in NED. This model represents documents as $|V|$-dimensional vectors where V is the vocabulary. The axes in the vector space are the terms in V. Figure 2.1 shows this concept.

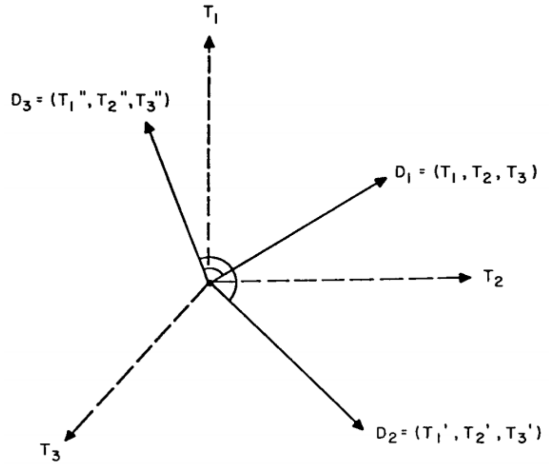


Figure 2.1: Three document representations in VSM with three terms.[1]

A pre-processing step is required to stem the words in the document and remove stopwords. Each term value is then alculated using the $tf \cdot idf$ scheme. This scheme is the standard approach in the information retrieval domain. The $tf \cdot idf$ scheme fulfills two key objectives. Firstly, frequent appearances of the same term in a document increases the likelihood of that term being important to the document's content. This follows the intutition that the higher the term frequency in a document, the more likely that term is relevant to the documents content. Secondly, terms which are rare across the entire document set N should be weighted highly when they appear in a document d. This follows another intution that rare terms across the entire corpus increase the relvance of those terms when they do appear. The tf and idf equations capture these objectives respectively. The term frequency $tf_{t,d}$ of term t in document d is the number of times t occurs in d. The log frequency weighting is used and is computed according to Equation 2.1.

$$w_{t,d} = \begin{cases} 1 + log_{10}(tf_{t,d}) & \text{if } tf_{t,d} \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The document frequency df_t of term t is defined as the number of documents that contain t. The inverse document freqency idf is computed according to Equation 2.2. It is important to note that there is one *idf* value per term in V. This necessitates the use of a training phase to determine the initial *idf* values.

$$idf_t = log_{10}(N/df_t) \tag{2.2}$$

Combining these equations gives the weighted value for each term in the document.

$$W_{t,d} = w_{t,d} \times idf_t \tag{2.3}$$

2.1.2 Document Similarity and Clustering

The standard approach to computing the similarity between documents is to compute the difference between their vectors. The *cosine similarity* equation 2.4 is an efficient means to compute this similarity between two vectors $\vec{u}$ and $\vec{v}$.

$$similarity = cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{||\vec{u}||||\vec{v}||} \tag{2.4}$$

The $cos(\theta)_{\vec{u},\vec{v}}$ score will range in values from 1 to 0, with 1 indicating the vectors are parallel and 0 being perpendicular. Negative values are not possible since the vector terms are computed using $tf \cdot idf$. The distance between two vectors is computed by $1 - cos(\theta)_{\vec{u},\vec{v}}$. Allen et al. reported that the cosine similarity is the best performing metric for the NED task [9].

A naive approach to NED is to assign a document as NEW/OLD depending on the distance between it and its nearest neighbour, as per algorithm 1. This algorithm has a quadratic running order of $O(n^2)$, where n is the number of documents in the corpus. This algorithm doesn't scale efficiently enough for use in a social stream setting.

Algorithm 1 A naive algorithm for the NED task

```
Require: $t \leftarrow input\ threshold$
1: for all d in document corpus do
2:     $dis_{min} \leftarrow min_{d'}\ \{distance(d, d')\}$
3:     if $dis_{min} > t$ then
4:         $d \leftarrow NEW$
5:     else
6:         $d \leftarrow OLD$
```

Clustering algorithms are more efficient approaches to the NED task. These algorithms create clusters of documents with the goal that the documents in a cluster all report on the same event. When classifying a new document, these algorithms compare the centroids of the clusters to the new document. If the distance is within a certain threshold then that document is added to that cluster. Otherwise it is classified as NEW and a new cluster is created. Algorithm 2 shows a basic clustering approach.

Algorithm 2 Clustering algorithm for the NED task

```
Require: $t \leftarrow input\ threshold$
1: for all d in documents do
2:     $dis_{min}, c \leftarrow min_c\ \{distance(d, clusters)\}$
3:     if $dis_{min} > t$ then
4:         $d \leftarrow NEW$
5:         $create\ new\ cluster(d)$
6:     else
7:         $d \leftarrow OLD$
8:         $c \leftarrow d$
```

The majority of NED applications under TDT use clustering algorithms. These algorithms use the $tf \cdot idf$ term weighting formula with the *cosine similarity* metric to determine cluster membership. A training phase is required to determine the threshold t and the initial *idf* values [6]. The state of the art in cluster algorithms differ in their use of the $tf \cdot idf$ weighting.

For example, Allan et al. used a modified version of $tf \cdot idf$ weighting that factored in document age [7]. Brants et al. used an incremental $tf \cdot idf$ model where they updated the *idf* values periodically [10]. This approach allows new terms to be added to the vocabulary as they appear in new documents. Various other researchers have made slight modifications to the $tf \cdot idf$ model. Schinas et al. boosted the weights of terms they deemed relevant to the events they were detecting [11]. They used a key-word lexicon based approach to detect this words. Makkonen et al. and Yang et al. proposed a new vector models where multiple vectors were constructed per document [12, 13]. These vectors represented separate semantic classes, and they computed similarity between documents by comparisons in their vector sets.

2.1.3 On-line NED with Social Streams

The volume of documents in social streams has necessitated the re-evaluation of NED algorithms. Google News has more than 4,500 sources [2], Yahoo! News has more than 5,000 sources [3] and there are more than 500 million documents produced on Twitter daily [4]. This volume of sources and documents has caused a resurgence in research in On-line NED systems [14, 15, 16, 17, 18, 19]. Luo et al. presented the first framework for a practical On-line NED system [17]. They apply heuristics on top of clustering NED approaches to make computation feasible. These heuristics include keeping only a limited set of documents in memory, limiting the vocabulary and employing parallel computing techniques. However, current state of the art in NED applications employ *locality sensitive hashing* (LSH) to reduce computation [20]. Constant processing time is possible using LSH, even in modern social streams.

[2]http://www.news.google.com
[3]http://www.news.yahoo.com
[4]http://www.about.twitter.com/company

2.1.3.1 Locality Sensitive Hashing (LSH)

Locality sensitive hashing (LSH) is a technique to probabilistically reduce the dimensions of data. The key approach is to hash the data into buckets such that similar points collide. LSH has various different forms and is used in a variety of domains. In the NED domain, LSH is used to solve the nearest neighbour problem, i.e. determining which point in the VSM is the closest to the query point. LSH works well for NED because of the large dimensionality of the vectors [21]. The distance between a document and its nearest neighbour determines whether it is assigned NEW/OLD. Linearly searching for the nearest neighbour for each document isn't feasible for the volume of documents in social streams. Rather than search for the nearest neighbour, recent research has focused on solving the problem of finding the *approximate* nearest neighbour. An approximate nearest neighbour is any point which lies between $(1 + \epsilon)r$ of the query point, with r being the distance to the actual nearest neighbour.

LSH works by hashing points into buckets with a high probability that close points will collide. A random projection hashing scheme works best for NED [22, 23]. This hashing scheme ensures the proportionality between the probability of collision and the *cosine similarity* of the vectors. The hash is computed by intersecting the vector space with k random hyperplanes, where k is the desired number of bits in the hash. Each bit k of the hash of vector v is either 1 or 0 depending on which side of the hyperplane v lies. Figure 2.2 shows the hash signature for a single vector.

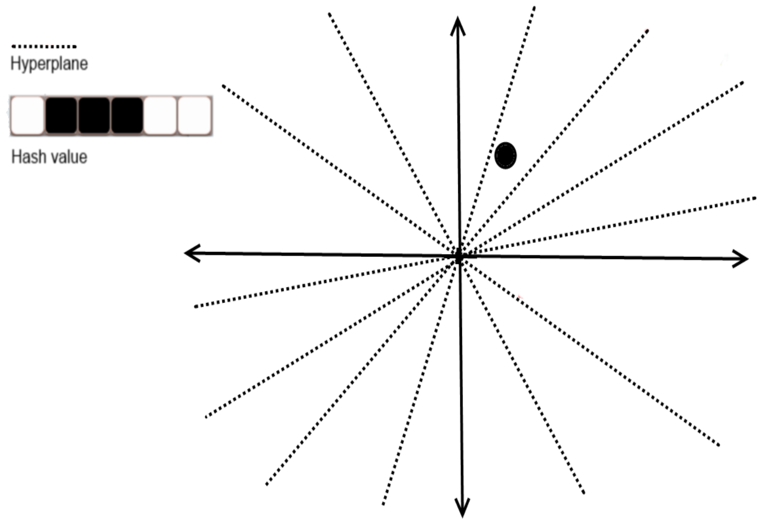


Figure 2.2: Hash signature computation using 6 hyperplanes.[2]

A LSH system contains L buckets each with a unique random projection hash function. A new document is hashed into each bucket and a set of colliding points is collected from each bucket.

This set of points is then linearly searched for the nearest neighbour. Petrovic et al. produced a LSH algorithm that is capable of operating in high volume modern social streams [24]. Their algorithm is shown in 3. This algorithm has a simple extension over the base LSH approach, corresponding to lines $9-11$. They show that simply applying LSH to a NED system with social streams produces poor results due to high variance. This is a byproduct of the failure of LSH to find a nearest neighbour if that neighbour is far from the query point. Petrovic et al. solve this problem by linearly searching a number of most recently seen documents. The algorithm attempts to find a recently seen document that is closer than that returned by LSH. This search is only performed if the incoming document is to be assigned NEW by the LSH system. Petrovic et al. also gave two important heuristics that should be used with social streams. Firstly, the number of documents allowed in a bucket should be bounded. They use a *first in first out* (FIFO) replacement scheme. Secondly, the number of colliding points that are searched for the nearest neighbour is also bounded. They give a figure of $3L$ comparisons where L is the number of buckets in the LSH system. These two heuristics are necessary to ensure a constant space and time running order for the NED system.

Algorithm 3 LSH algorithm with reduced variance[24]

```
Require: $t \leftarrow input\ threshold$
1:  for all d in documents do
2:      add d to LSH
3:      $S \leftarrow$ set of points that collide with d in LSH
4:      $dis_{min}(d) \leftarrow 1$
5:      for all d' in S do
6:          c = distance(d, d')
7:          if $c < dis_{min}(d)$ then
8:              $dis_{min}(d) \leftarrow c$
9:      if $dis_{min}(d) > t$ then
10:         compare d to a fixed number of most recent documents
11:         and update $dis_{min}$ if necessary
12:     assign score $dis_{min}(d)$ to $d$
13:     add $d$ to inverted index
```

2.1.4 Use of Semantics with NED

Various researchers report success with applying semantic analysis during the NED task. Zhang et al. give a metric to replace $tf \cdot idf$ [25]. This metric uses different term weights depending on the category of named entities the term belongs to. Makkonen et al. compute four vectors per document [12]. Each vector relates to one of four semantic classes: Locations, Proper Names, Temporal Expressions and Normal Terms. Similarity between documents is computed by comparing each set of vectors. Kumaran et al. give another modification to the vector model based on named entities [26]. They also produce category stop lists per named entity to improve topic-centered weighting. Finally, Petrovic et al. use paraphrases to increase term coverage in social streams [27].

2.2 The Twitter Stream

Having examining the NED task and associated algorithms, this chapter now discusses the Twitter social stream, its properties and the state of the art in NED tasks using this stream.

Twitter is a popular social stream where users post documents (*tweets*) up to 140 characters long. Tweets get posted to a users public timeline. They can contain regular text, *hashtags*, *mentions* or URLs. A *hashtag* is a Twitter notation used to tag your document for categorization. Structured events, such as sports games or political debates, usually advertise a specific hashtag for use. For example, the Six Nations rugby campaign encouraged the use of the hashtag #6nations. Mentions are Twitter usernames, preceded with the @ symbol. This allows users to interact with each other via tweets. Users also have the option of retweeting others tweets. Retweeting is the process of quoting another tweet and posting it on your timeline.

Twitter is growing in popularity. In 2007, the company reported 400 million tweets per quarter. Today, the Twitter stream has a daily volume of 500 million tweets. Twitter offers a public *Streaming API* [5] for access to a subset of these tweets. This API provides access to real-time tweets. Developers have the option of sampling all tweets currently being posted, or to provide some filtration parameters. The API currently returns up to 1% of total tweets occurring at that time. All tweets matching the filtration parameters will be returned unless that volume exceeds 1% of total tweets. Tweets are returned in JSON format and contain information such as the time the tweet was created and its text.

The Twitter stream is inherently noisy. Users engage in humor, project their personal thoughts, spread rumors and memes, and other frivolous activity that has no correlation with real world events [28, 29]. Pear Analytics report that 3.6% of tweets are news related [5]. The rest comprises of spam, conversational tweets, retweets, self-promotion and 'pointless babble'. Despite this noisy nature, Twitter is a useful social stream in a news-related context. Various researchers have studied Twitter user engagement during live events. Shamma et al. report Twitter user engagement during the 2008 US Presidential Debates [30] and Armstrong focused on live sports games [31]. Sakaki et al. concluded that Twitter users can be used as social sensors and are capable of reporting on events before official news sources [3].

[5] https://dev.twitter.com/docs/api/streaming

2.2.0.1 Twitter for NED

The volume of documents, public API and noisy nature of Twitter have made it a popular social stream for research. Sakaki et al. gave the first comprehensive study of using Twitter users as 'social sensors' for real world events [3]. Their work focused on the detection of earthquakes in the Japan area. They noticed that there is a temporal spike in activity during the immediate aftermath of an earthquake. This follows the intuition that a real world event causes a sudden increase in people talking about it. The system they produced detected these temporal spikes and used keyword extraction with geo-location information to recognize an earthquake. They concluded that events, such as earthquakes, are often reported on social streams before official news sources. The majority of later research has used this temporal property [16, 19, 32, 33, 34, 15, 30, 35, 36]. The noisy nature of Twitter has led researchers to use various techniques from the *natural language processing* domain. Spam detection, text stemming and stop word elimination improve the document quality for NED. Other researchers use sentiment analysis to improve their NED systems. Kwon et al. and Schinas et al. used sentiment in tweets to extract terms and weighted them differently for the similarity computations [33, 11].

2.3 NED systems that use Twitter

TwitInfo [35] allows users to input keywords describing events and presents them with the following features:

- Timeline: A graph showing tweet volume with peaks labelled as potential sub-events.
- Map: A map interface showing geo-locations of event tweets.
- Relevant Tweets: A list of tweets the system deems relevant to the event, coloured by sentiment (Red vs. Blue).
- Popular Links: A list of links that have been posted during the event.
- Sentiment Pie-chart: A pie-chart showing the proportion of positive to negative sentiment in event tweets

TwitInfo uses temporal peaks to detect sub-events during the user defined event. The system does not attempt to recognize these sub-events. The peaks are labelled as potential events and the user must recognize them. Furthermore, the system relies on user defined events at runtime and operates in a retrospective setting.

TwitterStand [37] reports on breaking news as reported by Twitter users. They make use of a list of pre-approved sources along with public Twitter users. They trained a Naive Baynes classifier to separate the reliable tweets from noise. The system uses a leader-follower clustering algorithm to detect events. Unlike this work, TwitterStand does not run on-line and relies heavily on pre-approved sources.

Other researchers have concentrated on sports game event detection and Twitter. Lanagan et al. [38] and Zhao et et al. [36] use temporal spikes during sports games to detect events. The system provided by Zhao et al. is the baseline system that was chosen to implement for this work.

2.3.0.2 A Baseline System

This dissertation is focues on the detection of structed live events using the Twitter stream. A baseline system has been identified which relies on the highly published temporal heuristice presented by Shakaki et al. [3]. This system serves as a comparison during the evaluations presented later in this work. The *SportsSense*, presented by Zhao et al., is the baseline system that was chosen [36]. This system uses prior knowledge of American Football sports games to build a NED system with Twitter. *SportsSense* is 'the first [system] to detect sports game events using Twitter.' The authors use the temporal peaks identified by Sakaki et al. [3] and lexicon-based keyword extraction to detect events. The authors built a document corpus by monitoring official NFL Twitter accounts and using the Twitter API to collect tweets with NFL-based *hashtags*. They found that the top 10 most frequent terms in this corpus were either team names or event terminology. They studied various filtration parameters for the data. They concluded that restricting tweets to those written in English and containing at least one team name provided the best results for NED. The *SportsSense* system has two stages: Event Detection and Event Recognition. Algorithm 4 outlines these stages. The Event Detection stage uses an adaptive sliding window algorithm to detect bursts of activity in the stream. The authors define bursts of activity as a period where the post rate of the window exceeds some threshold t. The post rate computation is the total tweets in the second half of the window divided by the total tweets in the first half. The algorithm starts with a 10 second window. The window size increases to 20, 30 and 60 seconds in turn until an activity burst is detected. The second stage, Event Recognition, is used when a burst of activity has been detected. This stage takes the tweets during this burst and attempts to correlate them with a pre-defined event. Stemming and stop word elimination are used as a pre-processing step. The post rates of pre-defined event keywords are then calculated. If a post rate for an event keyword exceeds a threshold t' then that event is reported.

Algorithm 4 *SportsSense* two-stage algorithm[36]

```
function EVENT_DETECTION($t$)
    loop
        Init window size to 10 seconds
        $post\_rate \leftarrow (post\_rate$ in first half$)/(post\_rate$ in second half$)$
        if $post\_rate < t$ then
            Increase window size
            go to 4
        else
            $Event\_Recognition()$
        $sleep(1\ second)$

function EVENT_RECOGNITION($t'$)
    pre-process tweet texts
    compute $post\_rate$ of event keywords in window
    if $post\_rate > t'$ then
        Report event
```

2.4 Summary

The survey of the state of the art presented in this chapter has lead to the following conclusions. A NED system with social streams must be able to cope with high volume and noise. Traditional NED clustering techniques are infeasible due to the volume. However, using a locality sensitive hashing algorithm to solve the nearest neighbour problem can make cluster feasible in an on-line setting. This requires using the $tf \cdot idf$ scheme to represent each document in vector space.

The state of the art NED systems that operate with the Twitter stream do not typically use these techniques. Instead, they overly rely on the temporal heuristic presented by Shakaki et al. [3]. This leads to the intuition that these systems will fail to detect events that do not cause a significant burst of activity on the stream. Therefore, this survey has motivated this work to incorporate clustering, LSH algorithms and the temporal heuristic to design and implement a NED system capable of running on-line with Twitter.

Chapter 3

New Event Detection in Social Streams (NEDISS) System Architecture

3.1 Introduction

This work presents *NEDISS*, a NED system that operates on-line with social streams. The NED algorithms used in *NEDISS* were designed using the state of the art in on-line NED and the temporal heuristic shown by Shakaki et al. [3]. This combination alleviates the reliance of current on-line NED systems on the temporal heuristic. The objective was to design a system capable of detecting all events, and not just those which caused a significant spike in document volume.

NEDISS was designed to operate with high volume social streams. These streams necessitate the use of various heuristics to ensure computation is feasible [17]. Therefore, the *NEDISS* architecture was modulated into three parallel processes: *Stream Processor, Cluster Manager* and *Event Recognizer.*

All the designs presented in this chapter and the implementation of them in chapter 4 were completly the work of the student.

The remainder of this chapter is structured as follows. First, the architecture of the *Stream Processor* process is outlined and each component design is discussed in detail. Next, the *Cluster Manager* process architecture is presented and each component design is again discussed in depth. Finally, the third process of the system, *Event Recognizer*, and its components are discussed.

3.2 Stream Processor

Stream Processor consumes one document[1] at a time from the social stream. This process was designed to represent each document in vector space and compute its nearest neighbour. The tuple of document and nearest neighbour is then sent to the *Cluster Manager* process. The components of this design can be seen in figure 3.1 and the algorithm is outlined in Algorithm 5.

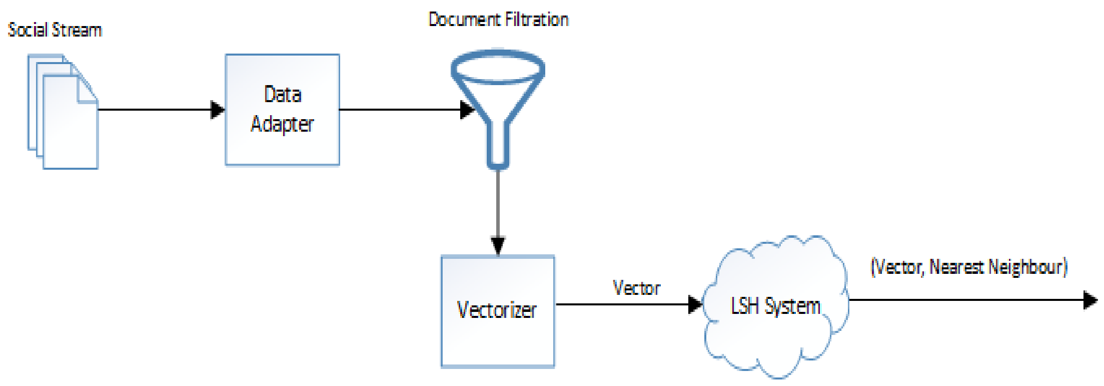


Figure 3.1: Components in the *Stream Processor* Architecture

Algorithm 5 Algorithm for Stream Processor

```
1: for all doc in data_adapter() do
2:     if $filter(doc)$ is True then
3:         $vect \leftarrow vectorize(doc)$
4:         $nearest\_neighbour \leftarrow locality\_sensitve\_hashing(vect)$
5:         output $(vect, nearest\_neighbour)$
```

3.2.1 Data Adapter

The *Data Adapter* component is the link between the social stream and *NEDISS*. This component was designed to retrieve one document at a time from the stream in chronological order. This document is then passed on to the *Document Filtration* component.

3.2.2 Document Filtration

The noisy nature of social streams necessitate a filtration layer. The *Document Filtration* component was designed to provide such a layer. Documents which aren't news-related are filtered out. This design eliminates further computation on irrelevant documents. The *Document Filtration* component receives the documents from the *Data Adapter* and passes them on to the *Vectorizer*

[1] A tweet is analogous to a document

if they pass the filter test. The implementation of the filter test will vary depending on the social stream.

3.2.3 Vectorizer

The *Vectorizer* component is responsible for representing each document in vector space. The following design decisions were made to ensure feasible computation with a high volume social stream.

Firstly, the vocabulary was fixed. This decision was necessary because of the use of $tf \cdot idf$ to calculate the term values of the vector. There can only be one *idf* value per term in the vocabulary. Having an unbounded vocabulary would result in unpredictable memory limitations and *idf* value updates/additions would increase computation dramatically. Luo et al. showed the benefits of a fixed vocabulary with $tf \cdot idf$ [17]. Locality sensitive hashing (used in a future component) also requires a fixed vocabulary.

Secondly, the vectorizer object was built using a training phase. This was mandatory to learn the initial *idf* values. Having a pre-built vectorizer alleviates the need for an on-line learning phase wherein events may be missed. The use of a fixed, learned vectorizer in the design gives predictable performance and replicative results.

The vector representation of each document is passed onto the *Locality Sensitive Hashing* component.

3.2.4 Locality Sensitive Hashing

The *Locality Sensitive Hashing* component was designed to receive a document vector as input and output the nearest neighbour to that document. The component design is built with L number of *Buckets* and a *Bucket Manager*.

Each *Bucket* contains a hash-table and unique hash function. These hash functions were designed using random projection. A random projection hash function creates k random vectors in term space. The k-bit hash for an incoming vector is computed by calculating the dot product between it and each k random projection. Each *Bucket* will have different random projections and thus each hash function will be different. A list of documents is kept for each entry in the hash-table. For an incoming vector, the *Bucket* will return the list of documents that it collides with and add the vector to the hash-table.

The *Bucket Manager* was designed as the entry point for the *Locality Sensitive Hashing* component. It manages the L different buckets. For an incoming vector, the manager gets the set of colliding documents from each Bucket. This set of colliding documents is collected and a similarity metric is computed between the incoming vector and all collisions. The *cosine distance* metric is used, and the document with the lowest score is returned as the nearest neighbour. This process is outlined in algorithm 6.

Algorithm 6 Algorithm for Bucket Manager

```
Require: $vect \leftarrow$ input vector
 1: $set\_of\_collisions \leftarrow []$
 2: for all b in buckets do
 3:     $collisions \leftarrow b.get\_collisions(vect)$
 4:     $set\_of\_collisions.append(collisions)$
 5:
 6: for all v' in set_of_collisions do
 7:     $dist \leftarrow cosine\_distance(vect, v')$
 8:     if $dist < dist_{min}$ then
 9:         $dist_{min} \leftarrow dist$
10:         $nearest\_neighbour \leftarrow v'$
11:
12: output nearest_neighbour
```

Petrovic et al. reported that additional steps are required in the LSH algorithm to reduce variance [24]. The *Locality Sensitive Hashing* component design doesn't include these steps. These steps require keeping a number of previously seen documents in memory and linearly searching them for a closer document than the nearest neighbour. This requires additional memory and running time that this work doesn't deem necessary. Instead, a subsequent component (*Cluster Manager*) was designed to reduce the variance that LSH introduces. This results in a quicker and more memory efficient algorithm than that presented by Petrovic et al.

Petrovic et al. gave two heuristics to follow if deploying a LSH system in an on-line environment [24]. Firstly, to limit the number of documents allowed in each hash-table entry. This gives an upper bound for the memory usage of an LSH system. Each Bucket uses a *First-In-First-Out* replacement scheme for the hash-table entries. Secondly, to limit the size of the set of collisions that is linearly searched for the nearest neighbour. They gave a figure of $3L$ where L is the number of Buckets in the LSH system. This gives an upper bound for the running time of an LSH system. Both of these heuristics were included in the design.

Once the *Locality Sensitive Hashing* component has found a nearest neighbour, it is sent to the *Cluster Manager* process along with the original document and their similarity score.

3.3 Cluster Manager

Cluster Manger maintains and monitors clusters of similar documents. This process receives each new document and its nearest neighbour from the *Stream Processor* process. The decision is then made whether to assign the document to an existing cluster or create a new one. This descision is made based on the similarity between the document and its nearest neighbour. Events are detected by monitoring the growth rate of the clusters. Once an event has been detected, the corresponding cluster is sent to the *Event Recognizer* process. The *Cluster Manager* process is designed to perform two functions: to maintain clusters and to monitor clusters. The central component to this process is the *Cluster*.

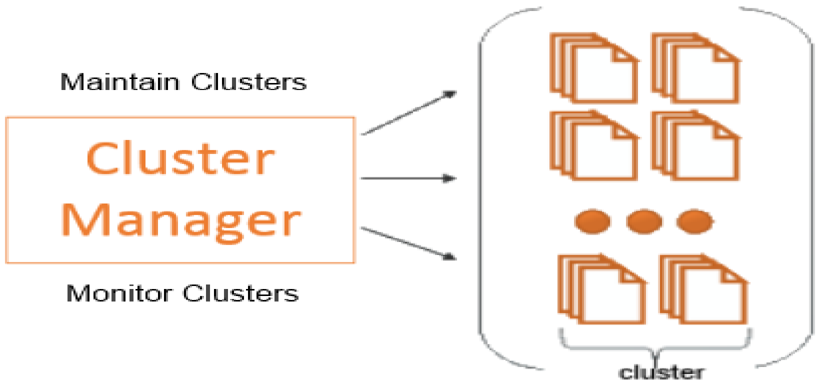


Figure 3.2: Cluster Manager process

3.3.1 Cluster

A cluster is a collection of similar tweet documents. The objective of clustering is to cluster all documents which describe the same event together. The *NEDISS* cluster architecture was designed to reduce memory and provide quick access to the information needed by the *Cluster Manager* to maintain and monitor the set of clusters. The design places an upper bound on the number of documents that a cluster can hold. Once this bound is exceeded, only document IDs are held by a cluster. These IDs are needed to determine document membership in the clusters, but the full document is discarded. The logic behind this decision is that a cluster contains documents describing the same event, therefore only a limited number of documents are required to recognize such an event. In the case of Twitter, a document ID is 8 bytes and the full document is typically greater than 6100 bytes. Therefore, this design reduces memory usage considerably.

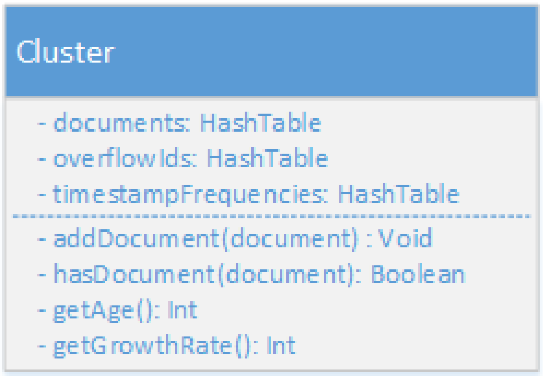

Cluster
- documents: HashTable
- overflowIds: HashTable
- timestampFrequencies: HashTable
- addDocument(document) : Void
- hasDocument(document): Boolean
- getAge(): Int
- getGrowthRate(): Int

Figure 3.3: UML Class diagram for a Cluster

The most frequent operations performed on a cluster are document addition, a document membership test, getting the age of the cluster and getting the growth rate of the cluster. The latter two functions require the design to include additional information other than the raw documents. The age and growth rate of a cluster are determined using the creation times of each document in the cluster. Therefore, an additional data-structure (along with the one holding the documents) was added to the design to hold this information. Thus, the cluster object was designed with the following data-structures.

Firstly, a hash-table mapping document IDs to raw documents. There is an upper bound placed on the size of this hash-table, as mentioned previously.

Secondly, a hash-table mapping document IDs to a NULL entry. This data-structure is used as the overflow for the first hash-table when the size bound is met. The entries are NULL because only a key membership test is required.

Finally, a third hash-table object is used to track the timestamps of the documents in the cluster. This hash-table maps timestamps to number of documents contained in the cluster with that timestamp. Hash-tables were chosen for these data-structures because they provide a $O(1)$ lookup time. The growth rate algorithm is outlined in Algorithm 7.

Algorithm 7 Cluster growth rate calculation

```
1: $sortedFreqs \leftarrow sort(timestampFrequencies.keys())$
2: $startTime \leftarrow sortedFreqs(0)$
3: $endTime \leftarrow sortedFreqs(-1)$
4: $midTime \leftarrow endTime - StartTime$
5:
6: $firstHalf \leftarrow$ Number of documents in cluster with timestamp $<= midTime$
7: $secondHalf \leftarrow$ Number of documents in cluster with timestamp $> midTime$
8: **output** secondHalf / firstHalf
```

3.3.2 Maintaining Clusters

The first responsibility of the *Cluster Manager* process is to maintain the clusters. This involves the creation of new clusters and the addition of new documents to the appropriate cluster. A pairing of a new document and its nearest neighbour, and their distance score, is received as input to this process. A decision must first be made whether this new document should be added to a cluster or a new cluster should be created for it. This is determined using a threshold t. If the distance between the new document and its nearest neighbour is less than t, then the new document is added to an existing cluster. Otherwise a new cluster is created for the document. The existing cluster that is selected for the document to be added to is the cluster which contains the nearest neighbour. The algorithm for adding/creating clusters is shown in Algorithm 8. The design does not place an upper bound on the number of active clusters. Instead, the *Cluster Manager* process monitors the existing clusters and deletes those which are deemed unlikely to be classified as an event.

Algorithm 8 Cluster Maintenance

```
**Require:** $vect \leftarrow$ input document
**Require:** $nearest\_neighbour \leftarrow$ nearest neighbour of vect
**Require:** $t \leftarrow$ threshold
1: $dist \leftarrow cosine\_distance(vect, nearest\_neighbour)$
2: **if** $dist < t$ **then**
3:     add $vect$ to same cluster as $nearest\_neighbour$
4: **else**
5:     create new cluster with $vect$
```

3.3.3 Monitoring Clusters

The second responsibility of the *Cluster Manager* process is to monitor the clusters. This entails monitoring the age of the clusters and their growth rate. The age of a cluster is defined as the difference in time between the oldest and newest document in the cluster. The goal of *NEDISS* is to detect realtime events in social streams. Therefore, if the age of a cluster is above some threshold, and it hasn't been classified as an event, then that cluster should be deleted as it is unlikely to correspond to a real world event. The *Cluster Manager* was designed to aggressively delete such clusters to save memory. This design decision was made in lieu of placing an upper bound on the possible number of clusters.

Cluster growth rate determines if a cluster is classified as an event. This captures the temporal burst of activity heuristic presented by Shakaki et al. [3]. Clusters which have a growth rate greater than some threshold should correspond to real world events. The *Cluster Manager* process was designed to linearly search each cluster and get their growth rate. If any of these values exceed the threshold, then that cluster is classified as a potential event and sent to the *Event Recognizer* process.

Monitoring clusters relies on two thresholds: maximum age allowed and minimum growth rate to be classified an event. The maximum age allowed determines how aggressively clusters are deleted. A low value will result in clusters getting deleted quickly. This is the safest in terms of memory usage, but could result in missed events where a cluster was deleted that would have been classified as an event at a later time. The minimum growth rate determines how fast a cluster must grow for it to be classified as an event. A low value will result in more clusters being classified as events. This increases the likelihood of false positives. A high value will result in fewer clusters getting classified as events. This would result in fewer, more accurate classifications but with more events being undetected. This threshold value defines the granularity of events that are detected by the system. The implementation and evaluation in this work explores a number values for these thresholds.

Once a cluster has been classified as a potential event by the *Cluster Manager* process, it is sent to the *Event Recognizer* process.

3.4 Event Recognizer

The final process in the *NEDISS* system is called *Event Recognizer*. This process receives clusters from the *Cluster Manager* process that are classified as new events. The role of this process is to recognize these events given knowledge of the structured events the system is trying to detect. This process was designed to include knowledge of the events possible during the structured events. Using this knowledge, the system attempts to categorize each cluster into a possible event. If successful, this event is output from the system. Otherwise, the cluster is rejected. This design helps alleviate some false positives from the previous processes. This process is split into three components: *Event Classifier*, *Named Entity Extractor* and *Data Output*.

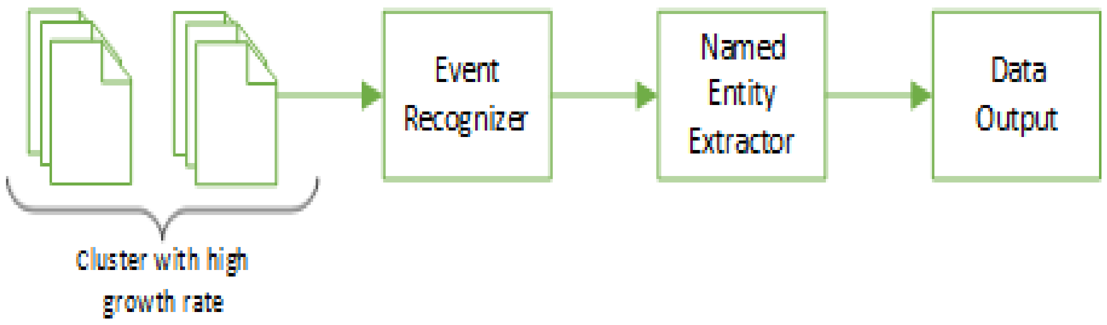


Figure 3.4: Event Recognizer process

3.4.1 Event Classifier

The *Event Classifier* component was designed to attempt to classify an input cluster as a predefined event. A keyword lexicon-based approach was taken. The design assumes a list of predefined events is available, and a list of keywords or phrases for each event. For example, a possible event may be ‘goal’ and the list of keywords associated with that event could be [‘score’, ‘scores’, ‘scored’, ‘goal’, ‘goals’]. The algorithm for classifying an event is shown in Algorithm 9.

Algorithm 9 Event Classification

```
1: for all doc in cluster do
2:     for all event in predefined_events do
3:         $lexicons \leftarrow get\_lexicons(event)$
4:         if any lexicon in doc then
5:             $votes[event]++$
6: $event \leftarrow max(votes)$
7: if $event.count > (cluster.count/2)$ then
8:     output event
9: else
10:    output unclassified
```

The classification algorithm was designed as a voting system. Each document in the cluster increments the vote for the predefined events that it contains the keywords of. The predefined event that the cluster is classified as is that which has the most amount of votes after this algorithm. A final check is applied to ensure that over 50% of the documents in the cluster voted for that event. This clause alleviates some false positives when a cluster contains documents describing various different events. Once the cluster has been classified, it is sent to the *Named Entity Extractor* component.

3.4.2 Named Entity Extractor

The *Named Entity Extractor* component was designed to improve the output of *NEDISS*. Intuitively, a real world event will have a list of associated named entities. This component was designed to collect this list of named entities. As well as improving the quality of output from *NEDISS*, named entity extraction can be used to improve accuracy. With the list of predefined events, one could associate a minimum number of named entities expected with such an event. For example, a substitution event in a soccer game should have three named entities involved: the two players and the team. If a cluster was classified as a substitution, the *Named Entity Extractor* could reject it if there wasn't at least three named entities extracting from the documents. This design reduces some false positives from the system.

The *Named Entity Extractor* component is optional. The task of named entity extraction is a slow process. An on-line NED system such as *NEDISS* must evaluate the speed cost in relation to the accuracy improvements such a component would provide. This evaluation can been found in chapter 5. The final component of *NEDISS* is *Data Output*.

3.4.3 Data Output

The *Data Output* component was designed to output the events detected by *NEDISS*. This component receives the final event detection information that was computed using the previous components in the system. The inclusion of this component allows the design of *NEDISS* to output structured information for use by other applications. This information includes the event detected, an estimated timestamp for the event and the named entities involved in the event (where applicable). The implementation of this component is flexible and determines the format of the final output of *NEDISS*. For example, an implementation may output an XML file describing each event. The *Data Output* component is the final component in the *NEDISS* design.

Chapter 4

NEDISS Implementation

4.1 Introduction

This chapter presents the implementation of the *NEDISS* design given in chapter 3. The primary technology used was the Python programming language. Python is a multi-paradigm interpreted language that has a comprehensive set of libraries available in the standard implementation. The implementation presented in this work focuses on the Twitter social stream and detected pre-defined events during sports games. Three separate threads form the basis of this implementation, encapsulating the design of the *Stream Processor, Cluster Manager* and *Event Recognizer.*

4.2 Stream Processor

4.2.1 Data Adapter

The *Data Adapter* connects to the Twitter public Streaming API. This API provides access to real-time documents on the Twitter stream. *NEDISS* was implemented to focus on live sports games. Therefore, the *Data Adapter* component provides a set of filtration parameters to the Twitter API to improve the relevance of the documents returned. These parameters consist of a language restriction and a set of keywords. To reduce computation, only documents written in the English language are acceptable to *NEDISS.* The set of keywords limits the possible documents to those which contain at least one of the keywords. Zhao et al. showed that 60% of sports game related tweets have the team name [36]. They concluded that a filtration set of team names and game terminology was effective. *NEDISS* extends this by also including the nicknames and common abbreviations of the teams, and official Twitter hashtags (where applicable). The full set of keywords is available in Appendix A.

This component was implemented using a Twitter package for Python, called *twitter*. This package provides objects for communication with the Twitter API. These objects accept standard

OAuth[1] tokens for authentication and allow the inclusion of filtration parameters. The OAuth tokens were retrieved from the Twitter dev center [2]. The implementation provides access to the filtered stream through a Python generator object. Documents are returned in JSON format and *Data Adapter* converts them to Python dictionary objects. Each new dictionary (document) is then passed on to the *Document Filtration* component.

4.2.2 Document Filtration

The *Document Filtration* component was implemented to remove spam, retweets and conversational tweets from the Twitter stream. Grier eg al. demonstrated how spam tweets must contain URLs [39]. The Tweet objects returned by the Twitter API include fields which can be checked to classify the tweet as spam, retweet, conversational or acceptable. A tweet is deemed spam if it contains a URL. The 'urls' field of the 'entities' field in the tweet object contains this information. The tweet contains a 'retweeted_status' field that shows if the tweet is a retweet of another tweet. Conversational tweets are marked by the 'in_reply_to_user_id' field which gives the user id of the person the tweet is conversing with. The implementation classifies a tweet as acceptable if these three fields are 'None'. This was implemented using a filter with a lambda function (shown in listing 1) on the generator returned by the *Data Adapter*. The text of the acceptable tweet is then preprocessed and sent to the *Vectorizer* component. This preprocessing step removes mentions and grammar from the tweet.

Listing 4.1: Lambda filter functions applied to Twitter Stream

```
not_a_reply = lambda x: not x.get('in_reply_to_user_id')
not_a_retweet = lambda x: not x.get('retweeted_status')
no_urls = lambda x: not x.get('entities').get('urls')
acceptable_tweet = lambda x: not_a_reply(x) and not_a_retweet(x)
                             and no_urls(x)
```

4.2.3 Vectorizer

The vectorizer object was implemented using the *scikit-learn* Python package [3]. This is the de facto machine learning and data analysis package for Python. Two vectorizer objects were implemented, each with different vocabularies. The first vocabulary consisted of sports game terms and team names (the same list used for the filtration parameters). This vocabulary is available in Appendix A. The second vocabulary was learned from the training set. The 0.5% most frequent words in the training set were used. This percentage was chosen empirically by investigating the terms used throughout the training set. The evaluation of these two vectorizer objects is included in chapter 5.

[1]http://oauth.net/
[2]http://www.dev.twitter.com
[3]http://www.scikit-learn.org

Each vectorizer was implemented to use lowercase terms in the computation and ignore stopwords. The stopword list chosen was the one included in the *nltk* Python package. The two vectorizer objects differ in the n-grams they consider. The learned vocabulary vectorizer only considers 1-gram terms. The other vectorizer considers l-gram terms, where l is the largest number of terms in a vocabulary lexicon. Each vectorizer was saved to a file after it was trained with the training set.

For each tweet the *Vectorizer* component received, it transforms that tweet to a sparse matrix using the trained vectorizer. Tweets are inherently short documents consisting of upto 140 characters. This results in few terms relative to the vocabulary. Therefore, a sparse matrix provides a more memory efficient representation for the vectors. Each sparse matrix is then passed along to the *Locality Sensitive Hashing* component.

4.2.4 Locality Sensitive Hashing

The *Locality Sensitive Hashing* component was implemented in Python. Two classes were created: *Bucket* and *BucketManager*.

Listing 4.2: Bucket Class for the LSH System

```
class Bucket:
  def __init__(self, k, num_features, max_values):
    self.hash_table = defaultdict(list)
    self.randv = numpy.random.randn(k, num_features)
    self.max_values = max_values

  def get_collisions_and_add_value(self, tweetVect):
    sig = get_signature(tweetVect.vect, self.randv)

    pointer = self.hash_table[sig]
    res = list(pointer)

    if len(res) >= self.max_values:
      pointer.pop(0)

    pointer.append(tweetVect)

    return res
```

The *Bucket* class was designed with internal variables for the hash-table, k hyperplanes and an upper bound for the number of documents allowed in each hash-table entry. The hash-table was implemented using Python's *defaultdict* collection. This collection provides a dictionary object that returns a default value when there are no entries for the key. The LSH hash-table is designed such that an empty list of documents is returned when a key isn't present in the hash-table. The k hyperplanes are generated using a helper function in the *numpy* Python package, *randn*. This function returns an array of random values from the standard normal distribution. By default these values will be from $N(1, 0)$.

The *Bucket* class implementation has one function which handles all the necessary logic: *get_collisions_and_add_value*. This function takes a *TweetVect* tuple as input and returns the list of *TweetVect* objects that the input collides with. The function also has the side effect of adding the input to the hash-table. A *TweetVect* tuple is a helper object that holds the original tweet document and its vector representation. The *get_collisions_and_add_value* function uses the signature of the input to hash into its hash-table and return the list at that entry. If the list length is equal to the upper bound, then the first entry added to the list is removed to make room for the new item. The signature is retrieved via a helper function *get_signature*, shown in listing 3.

Listing 4.3: Function to calculate the signature

```
def get_signature(user_vector, rand_projs):
  res = 0
  for p in rand_projs:
    res = res << 1
    val = numpy.dot(p, user_vector)
    if val >= 0:
      res |= 1
  return res
```

The *BucketManger* class manages the L buckets in the LSH system. The logic for this class was implemented in a single function: *get_nearest_neighbour*. This function first iterates through each Bucket and collects the collisions with the input document vector. Next, the cosine distance between the input document and up to the first $3L$ collisions is computed. This distance value was implemented using a helper function available in the *scipy.spatial.distance* Python package. Finally, the function returns a tuple of the input vector and the nearest neighbour, along with the distance between them.

Listing 4.4: BucketManager Class for the LSH System

```
class BucketManager:
  def __init__(self, l, k, num_features, max_values):
    self.buckets = tuple(Bucket(k, num_features, max_values)
                  for _ in xrange(l))
    self.max_comparisons = 3 * l

  def get_nearest_neighbour(self, tweetVect):
    set_of_collisions = chain(*[bucket.get_collisions_and_add_value(
                            tweetVect)
                            for bucket in self.buckets])
    max_comparisons = islice(set_of_collisions,
                             0,
                             self.max_comparisons)

    dist_and_tweetVecs = [(cosine(tweetVect.vect, collision.vect),
                          collision)
                          for collision in max_comparisons]

    if len(dist_and_tweetVecs):
      return min(dist_and_tweetVecs, key=lambda x: x[0])
```

The *Locality Sensitive Hashing* component was designed to accept a value for k and L. The values 13 and 20 were chosen respectively. A value of 13 for k was presented by Petrovic et al. as a good "balance between time spent computing the distance and the time spent computing the hash" [24]. A value of 20 for L was chosen empirically using the method described by Datar et al. [21]. They suggested to first guess the value of L and use a binary search to improve it. A value of 20 was found to provide a good balance between memory requirements and probability of finding a nearest neighbour.

Once the nearest neighbour and distance score is computed by this component, they are sent to the *Cluster Manager* process.

4.3 Cluster Manager

The *Cluster Manager* process was implemented in Python. Two classes were created: *Cluster* and *ClusterManager*.

The *Cluster* class implementation has three data-structures which hold the information required by the *Cluster Manager* process. Firstly, the *frequencies* default dictionary is used do store the timestamps of the documents in the cluster. Each key refers to a unique timestamp and the entry at that key is the number of documents in the cluster that have that timestamp. This was implemented as a default dictionary of integer values. This type of collection returns a default value (zero) if the key isn't found in the dictionary. This logic is desirable for this implementation because timestamps that don't appear in the dictionary should automatically be added with a value of zero.

Secondly, the *tweet_vects* dictionary stores the TweetVect objects that are in the cluster, i.e. the documents. Each key in the dictionary is a document id and the value at that key is the TweetVect object with that key. The number of entries in this dictionary has an upper bound of *max_values*.

Finally, the *overflow_tweets* dictionary is used to store the ids of the documents that would be *tweet_vects* if it weren't full. These ids must still be saved in the cluster because of potential membership tests. Each key is a document id and the entry at that key is simply set to 1. The entries are not important in this dictionary, the implementation only uses it to check if a document is in the cluster. These data-structures were designed as hash-tables and are therefore implemented with Python dictionaries.

The *Cluster* class implementation has six functions that are used by other objects. The *add_tweet_vects* function first increments the appropriate entry in *frequencies*. Next, the entire TweetVect is saved in *tweet_vects* if the upper bound hasn't been reached, otherwise its ID is added to *overflow_tweets*. The *has_tweet_vect* function searches the *tweet_vects* and *overflow_tweets* dictionaries for the input tweet. This function returns a boolean value depicting whether or not the input document is in the cluster. The *get_age_seconds* function returns the age of the cluster. This was implemented as the difference in seconds between the oldest and newest document in the cluster. The *get_timestr* function returns a string of the estimated time the event this cluster describes occurred. This was implemented as the median timestamp of the documents in the cluster. The *get_texts* function returns a list representing the texts of the documents in the cluster. Finally, the *get_growth_rate* function returns the growth rate of the cluster. This was implemented using Algorithm 7.

Listing 4.5: Cluster class implementation

```python
class Cluster:
  def __init__(self, max_values):
    self.frequencies = defaultdict(int)
    self.tweet_vects = {}
    self.overflow_tweets = {}
    self.max_values = max_values

  def add_tweet_vect(self, tweetVect):
    timestamp = tweetVect.tweet.get('created_at')

    self.frequencies[timestamp] += 1

    tweet_id = tweetVect.tweet.get('id')

    if len(self.tweet_vects) < self.max_values:
      self.tweet_vects[tweet_id] = tweetVect
    else:
      self.overflow_tweets[tweet_id] = 1

  def has_tweet_vect(self, tweet_vect):
    tweet_id = tweet_vect.tweet.get('id')
    return tweet_id in self.tweet_vects or
           tweet_id in self.overflow_tweets

  def get_age_seconds(self):
    sorted_times = sorted([tweet.datetime_from_tweet_created_at(s)
                          for s in self.frequencies.keys()])

    return (sorted_times[-1] - sorted_times[0]).total_seconds()

  def get_timestr(self):
    sorted_freqs = sorted([s for s in self.frequencies.keys()])
    return sorted_freqs[len(sorted_freqs) /2]

  def get_texts(self):
    tweets = [v.tweet for k,v in self.tweet_vects.iteritems()]
    return [t.get('text').encode('utf-8') for t in tweets]

  def get_growth_rate(self):
    sorted_freqs = sorted([s for s in self.frequencies.keys()])

    mid_time = sorted_freqs[-1] - (sorted_freqs[-1] - sorted_freqs[0])

    first_half = sum([freq for time, freq in sorted_freqs
                      if time <= mid_time])

    second_half = sum([freq for time, freq in sorted_freqs
                      if time > mid_time])

    return float(second_half / first_half)
```

The *ClusterManager* maintains a list of *Cluster* objects. The logic for this process was implemented using a constantly running thread. The *Stream Processor* process adds items into a work queue for the *ClusterManager*. This implementation avoids deadlock and starvation between the running processes. The *ClusterManager* thread performs three functions every second. Firstly, it clears its work queue. This queue is a list of tuples consisting of new documents and their nearest neighbours. Algorithm 8 was implemented to determine if a cluster should be created or the document added to an existing cluster. The queue is emptied after each entry has been processed. The second function of *ClusterManager* is to monitor the age of the clusters and delete appropriate ones. This was implemented by iterating over each cluster, getting its age and comparing it to a given threshold. If the age of a cluster exceeds the threshold then it is deleted. Finally, the *ClusterManager* thread monitors the growth rate of the clusters. This was implemented by iterating over each cluster and getting its growth rate. If this value exceeded a given threshold, then that cluster was classified as an event. The cluster is then sent to the *Event Recognizer* process and deleted from the *ClusterManager's* cluster list.

Listing 4.6: ClusterManager class implementation

```
class ClusterManager(threading.Thread):
  def __init__(self, max_values_per_cluster,
               old_cluster_secs, post_rate, recognizer):
    super(ClusterManager, self).__init__()
    self.clusters = []
    self.max_values_per_cluster = max_values_per_cluster
    self.old_cluster_secs = old_cluster_secs
    self.add_queue = []
    self.post_rate, self.recognizer = post_rate, recognizer

  def run(self):
    one_second = timedelta(seconds=1)

    while True:
      wake_time = datetime.now() + one_second

      self.clear_add_queue()
      self.delete_old_clusters()
      self.detect_events()

      sleep_seconds = (wake_time - datetime.now()).total_seconds()
      if sleep_seconds > 0:
        sleep(sleep_seconds)

  def add_to_work_queue(self, tweet_vect, nearest_neighbour):
    self.add_queue.append((tweet_vect, nearest_neighbour))

  def clear_add_queue(self):
    for tweet_vect, nearest_neighbour in self.add_queue:
      create_new = True

      for c in self.clusters:
        if c.has_tweet_vect(nearest_neighbour):
          c.add_tweet_vect(tweet_vect)
          create_new = False
          break

        if create_new:
          new_cluster = self.create_new_cluster(self.max_values_per_cluster,
                                                tweet_vect,
                                                nearest_neighbour)
          self.clusters.append(new_cluster)

    del self.add_queue[:]

  def create_new_cluster(self, max_values, *tweet_vects):
    new_cluster = Cluster(max_values)

    for tv in tweet_vects:
      new_cluster.add_tweet_vect(tv)

    return new_cluster

  def delete_old_clusters(self):
    for c in self.clusters:
      if c.get_age_seconds() > self.old_cluster_secs:
        self.clusters.remove(c)

  def detect_events(self):
    for c in self.clusters:
      if c.get_growth_rate() > self.post_rate:
      self.recognizer.recognize_event(c)
      self.clusters.remove(c)
```

The cluster monitoring logic was designed to be controlled by two thresholds: maximum age and minimum growth rate. The maximum age parameter was set as sixty seconds. This value was chosen empirically by investigating the tweet volumes of events during live sports games. Events during a sports game are only discussed around the time of the event. Subsequent documents usually refer to subsequent events that have occurred. Intuitively, if a cluster has documents more than sixty seconds apart, the probability of that cluster being classified as an event is low. The cluster is therefore deleted and associated memory is freed. This work includes the evaluation of several values for the minimum growth rate parameter. Zhao et al. used a parameter of 1.7 when implementing a similar growth rate heuristic [36]. The list of parameters used and their effects on the results of *NEDISS* can be found in chapter 5.

4.4 Event Recognizer

Listing 4.7: Event Recognizer class implementation

```
class EventRecognizer(threading.Thread):
  def __init__(self):
    super(EventRecognizer, self).__init__()
    self.cluster_queue = []
    self.cv = threading.Condition()
    self.data_output = DataOutput()

  def run(self):
    while True:
      self.cv.acquire()
      while not self.cluster_queue:
        self.cv.wait()

      for cluster in self.cluster_queue:
        texts = cluster.get_texts()
        event = classify_list(texts)

        if event:
          named_entities = ner.get_named_entities(texts)
          timestamp = cluster.get_timestr()
          self.data_output.output_event(event, named_entities, timestamp)

        self.cluster_queue.remove(cluster)

      self.cv.release()

  def recognize_cluster(self, cluster):
    self.cv.acquire()
    self.cluster_queue.append(cluster)
    self.cv.notify()
    self.cv.release()

  def classify_list(self, texts):
    votes = defaultdict(int)

    for text in texts:
      for event in predefined_events:
        if any([term in text for term in event.lexicons]):
          votes[event] += 1

    if len(votes):
      winner = max(votes.iteritems(), key=lambda (k,v):v)
      if winner[1] > (len(texts) / 2):
        return winner[0]
```

The *Event Recognizer* process was implemented as a thread in Python. The thread sleeps until the *Cluster Manager* process detects an event. Once this occurs, the cluster is placed on a work queue for the thread and it awakes. This is controlled with a condition variable *cv*. The thread waits on this condition variable if its work queue is empty. Clusters are placed into the queue by the *ClusterManager* object via a call to the function *recognize_cluster*.

Algorithm 9 was implemented in the function *classify_list* to classify the cluster to a pre-defined event. This function uses a *predefined_events* data structure to classify the cluster. A vote is assigned to each possible event by the texts that are classified as refering to it. This classification is performed using lexicon-based keyword analysis. The cluster is classified as the event with the highest votes, if $> 50\%$ of texts voted for it. Otherwise, the cluster remains unclassified. Unclassified clusters are deleted from the *Event Recognizer*. This implementation alleviates some of the false positives from the *Cluster Manager* process.

Once an event has been classified by the *EventRecognizer* thread, it is sent to a *Named Entity Extractor* object.

4.4.1 Named Entity Extractor

The Stanford NER[4] implementation was used for the *Named Entity Extractor*. Various NER implementations were tested and the Stanford one offered the most desirable results for tagging entities that may be involved in events. In particular, tagging locations, people and organizations. This was determined empirically through trial and error of various implementations. The Stanford NER also provides various models through which to train the NER. The *english.all.3class.caseless.distsim.crf.ser* model was chosen for the following reasons. Firstly, it is written in the English language which this work is focused on. Secondly, it only attempts to recognize three classes of named entities: Location, Person or Organization. These classes represent the only named entities in an event that this work wishes to extract. Finally, it ignores capitalization. This is important because of the absence of a guarantee that Twitter users will properly capitalize the named entities in their tweets. The *get_named_entities* fucntion returns the list of successfully tagged entities from the text, shown in listing 4.8.

Listing 4.8: NER function

```python
st = NERTagger('english.all.3class.caseless.distsim.crf.ser.gz',
               'stanford-ner.jar')

def get_named_entities(text):
  tagged = st.tag(text.split())
  return [t for t in tagged if t[1] is not 'O']
```

[4]http://www.nlp.stanford.edu/software/CRF-NER

The list of named entities are returned to the *EventRecognizer* thread. The number of named entities that were extracted could be compared against an expected figure for the event that was detected. For example, a *substitution* event should have at least 2 named entities. If the retured list of named entities has less than 2 elements, then this event can be rejected. This implementation would improve the quality of the structured output. However, correctly detected events could be rejected if the cluster texts aren't detailed enough to extract the desired number of named entities. The implementation in this work does not include such a check for this reason.

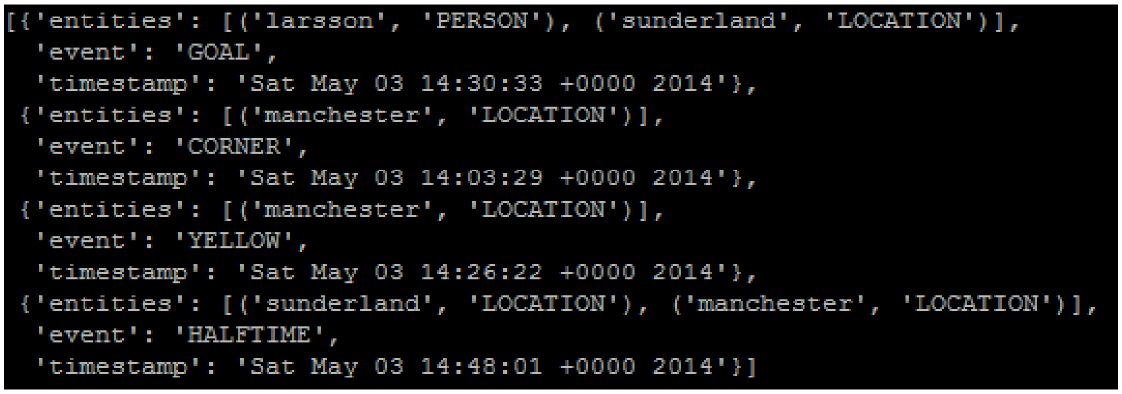
4.4.2 Data Output

The final task in the *NEDISS* system is to output the information in a structured format. The *Data Output* component was designed for this task. This work implemented this component to output a JSON file describing the event. The event name, time of event and the named entities recognized are output. This function is shown in listing 4.9.

Listing 4.9: Data Output function

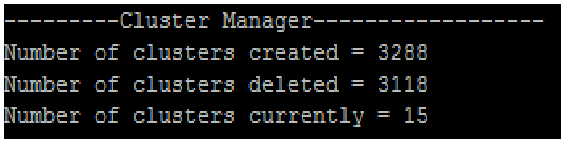
```
def output_event(event, named_entites, timestamp):
    json.dumps(dict(event=event,entities=named_entities,timestamp=timestamp))
```

4.4.2.1 Sample Output



```
[{'entities': [('larsson', 'PERSON'), ('sunderland', 'LOCATION')],
  'event': 'GOAL',
  'timestamp': 'Sat May 03 14:30:33 +0000 2014'},
 {'entities': [('manchester', 'LOCATION')],
  'event': 'CORNER',
  'timestamp': 'Sat May 03 14:03:29 +0000 2014'},
 {'entities': [('manchester', 'LOCATION')],
  'event': 'YELLOW',
  'timestamp': 'Sat May 03 14:26:22 +0000 2014'},
 {'entities': [('sunderland', 'LOCATION'), ('manchester', 'LOCATION')],
  'event': 'HALFTIME',
  'timestamp': 'Sat May 03 14:48:01 +0000 2014'}]
```

Figure 4.1: Sample output during the Manchester United Vs. Sunderland match



```
---------Cluster Manager------------------
Number of clusters created = 3288
Number of clusters deleted = 3118
Number of clusters currently = 15
```

Figure 4.2: Sample Cluster Manager stats after the Manchester United Vs. Sunderland match

4.5 Baseline System

As previously discussed, the *SportsSense* system presented by Zhao et al. [36] was implemented as a baseline system. This implementation was done in Python and as described by the authors in their paper.

Chapter 5

Evaluation of NEDISS

5.1 Introduction & Objectives

This chapter presents the evaluation of the *NEDISS* system. The objective is to evaluate the performance of the *NEDISS* system. To accomplish this, the following experiments are presented. The first and second experiment present a comparison between the results of *NEDISS*, a baseline system and the real world events during a number of soccer matches. The first experiment was performed on streams of documents from a single match, whereas the second experiment used a stream consisting of multiple matches. These experiments provide a comparison between the values for Recall, Precision, Accuracy and F_1 for each of the systems. The third experiment investigates the effects of the *growth_rate* parameter of *NEDISS*. Four different values are used for the *growth_rate* and the Recall, Precision, Accuracy and F_1 for each are presented. This experiment also gives a comparison of the number of clusters classified as potential events by the *Cluster Manager* process of *NEDISS*. The fourth experiment provides a comparison between the use of a learned vocabulary and one which was provided explicitly. Again, the Recall, Precision, Accuracy and F_1 metrics are presented for each. The final experiment investigates the effect of the *Named Entity Extractor* component of *NEDISS*. Along with the metrics already discussed, this experiment presents some running time scores of *NEDISS* with and without the *Named Entity Extractor* component. These experiments provide a thorough investigation into the performance of *NEDISS*. The comparison between *NEDISS*, a baseline system and the expected results provides a clear view of the accuracy and comparative accuracy of the system. Also, the comparison between the various *NEDISS* configurations gives an indication of the effect the various elements have on the system.

The remainder of this chapter is presented as follows. First, the baseline system is discussed. Next, the experimental setup is presented. This section provides the details of the data and metrics used, and the different configurations of *NEDISS* that are evaluated. Following that, each of the experiments discussed previously are presented. Finally, this chapter concludes with a summary of the evaluations.

5.2 Baseline System

The evaluations presented in this chapter include comparisons between *NEDISS* and a baseline system. The *SportsSense* system published by Zhao et al [36] was chosen. This system represents the best existing system that focuses on event detection during sports games with the Twitter stream. The reliance of this system on the highly published temporal heuristic, first presented by Shakaki et al. [3], motivated the design of *NEDISS*. The authors also present values for Precision and Recall, although in a different context to this work. The implementation of this system was discussed in chapter 4.

5.3 Experimental Setup

This section outlines the experimental setup used during this evaluation. The training dataset, experimental dataset, various systems and their configurations under evaluation, and the metrics used are presented.

A training dataset was required to train the *Vectorizer* component of *NEDISS*. This dataset was used to learn the initial $tf \cdot idf$ values and provide a learned vocabulary instance of the *Vectorizer*. This dataset contains 1,094,502 raw tweets returned from the Twitter API. These tweets were collected using the *Streaming* API with filtration parameters for matches during the 2013/2014 Barclay's Premiership season. The data was collected between 1/3/2014 and 8/3/2014 (inclusive) and thirteen different matches were captured.

The experimental dataset is used during the evaluation experiments. This dataset was captured with the objective of it being a representative sample of volume and events possible during soccer games. Tweets during one hundred and seventeen matches between 1/4/2014 and 11/5/2014 (inclusive) were captured. For the proposes of the evaluations presented in this work, human annotation is required to determine the actual events during the matches. As such, it isn't feasible to use an experimental dataset which covers a large number of soccer matches. Therefore, the experimental dataset used consisted of twenty soccer matches. These twenty matches were chosen because of the distribution of events that occurred and the different volume rates. This dataset contains 5,063,668 tweets.

The evaluations presented in this chapter include 8 different system, 7 of which are variations of the *NEDISS* system. The first system is the baseline system discussed previously. The other systems are variations of *NEDISS*, with different values for the *growth_rate* parameter, vocabulary used to train the *Vectorizer* and the inclusion/exclusion of the *Named Entity Extractor* (NER). Table 5.1 gives an overview of these different configurations and a label for each.

Label	**growth_rate**	**Vocabulary used**	**NER?**
NEDISS	1.7	Given	No
NEDISS11	1.1	Given	No
NEDISS25	2.5	Given	No
NEDISS34	3.4	Given	No
NEDISS40	4.0	Given	No
NEDISSL	1.7	Learned	No
NEDISSNER	1.7	Given	Yes

Table 5.1: Different configurations of the *NEDISS* system under evaluation

The accuracy metrics used in these evaluations rely on *True Positives (TP), False Positives (FP)* and *False Negative (FN)*. In the context of this work, a *TP* is a correctly detected event, a *FP* is a detected event that didn't occur in the real world and a *FN* is a failure to detect an event. Using these values, one can calculate the Precision, Recall, Accuracy and F_1 metrics using the following equations.

$$Precision = \frac{|TP|}{|TP| + |FP|} \quad Recall = \frac{|TP|}{|expected\ TP|}$$

$$Accuracy = \frac{|TP|}{|TP| + |FN| + |FP|} \quad F_1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

These scores are computed over each event category individually and on a global scale, i.e. *macro-averaging* and *micro-averaging*. *Macro-averaging* scores are influenced heavily by rare events and *micro-averaging* scores are heavily weighted on the performance with common events. Both methods are used to provide additional information and not be subject to the flaws of either. The events considered for each method are outlined in table 5.2. These events were chosen because they represent the major events during a soccer game and exhibit different volume spikes.

Micro-averaging	**Macro-averaging**
GOAL	GOAL
PENALTY	YELLOW CARD
YELLOW CARD	SUBSTITUTION
HALF-TIME	
FULL-TIME	
SUBSTITUTION	

Table 5.2: Events under consideration for accuracy metrics

In addition to the metrics above, this work presents two significance tests for comparing systems. Specifically, the *Micro sign test (s-test)* and *Macro sign test (S-test)* presented by Yang et al. are used [40]. These significance tests compare two systems, A and B, using values for n and k. For the *s-test*, n is the number of times the two systems differ in their classifications, and k is the number of times A categorizes correctly when B doesn't. For the *S-test*, n is the number of times the F_1 score per category differs between the system, and k is the number of times A's per category F_1 score is higher than B's. The *null hypothesis* for both tests is $k = 0.5n$, or k has a binomial distribution of $Bin(n, p)$ where $p = 0.5$. The *alternative hypothesis* states k has a binomial distribution of $Bin(n, p)$ where $p > 0.5$. In other words, the *null hypothesis* states that A isn't significantly better than B, and the *alternative hypothesis* states that A is significantly better. The 1-sided P-value for each test is calculated using the following equations.

if $n \leqslant 12$ and $k \geqslant 0.5n$:

$$P(Z \geqslant k) = \sum_{i=k}^{n} \binom{n}{i} \times 0.5^n$$

if $n \leqslant 12$ and $k \leq 0.5n$:

$$P(Z \leqslant k) = \sum_{i=0}^{k} \binom{n}{i} \times 0.5^n$$

Otherwise, the P-value can be computed using the standard normal distribution:

$$Z = \frac{k - 0.5n}{0.5\sqrt{n}}$$

The *s-test* and *S-test* are widely used throughout the text classification domain.

5.4 Experiments

5.4.1 Comparison between NEDISS, Baseline and Actual Events in a Single-match Stream

5.4.1.1 Objectives

The objectives of this experiment are as follows:

1. Provide accuracy metrics for the *NEDISS* system.
2. Compare the accuracy of *NEDISS* to a baseline system.
3. Determine if *NEDISS* performs statistically better than the baseline system.

5.4.1.2 Experimental Method

This experiment compares the *NEDISS17* configuration and the baseline system to annotated real world events during soccer matches. The experimental dataset previously discussed was used and both systems operated on single match streams. Recall, Precision, Accuracy and F_1 score metrics were computed on both a *macro-averaging* and *mirco-averaging* scale for these two systems. The events outlined in table 5.2 were used to calculate these metrics. Finally, an *s-test* and *S-test* was computed between the baseline and *NEDISS17*.

5.4.1.3 Experiment Results

Table 5.3 displays the Precision, Recall, Accuracy and F_1 score metrics for the baseline system and *NEDISS17*. Figure 5.1 shows these values in a bar chart. These metrics were computed over the experimental dataset by means of *micro-averaging*.

	Baseline	**NEDISS17**
Precision	0.92	0.85
Recall	0.27	0.53
Accuracy	0.26	0.49
F_1	0.42	0.65

Table 5.3: Single-stream micro-averaged metrics for Baseline and *NEDISS17*

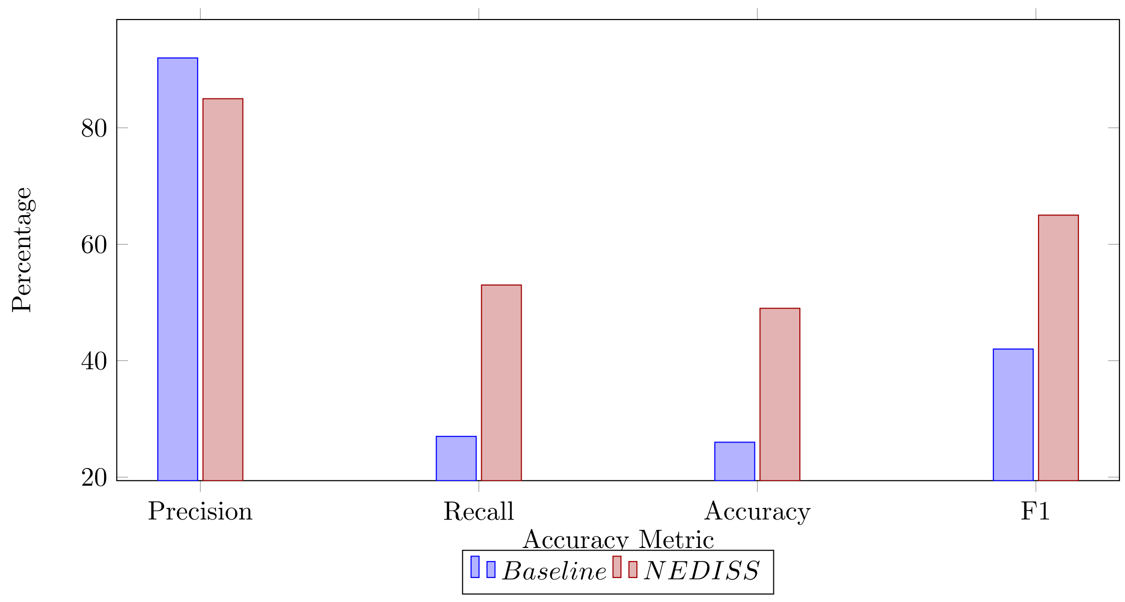


Figure 5.1: Bar chart of micro-averaged metrics for baseline and *NEDISS17*

Table 5.4 displays the same metrics as shown previously between the two systems. However, they are split into GOAL, YELLOW CARD (Y) and SUBSTITUTION (SUB) *macro-averaged* values. Figures 5.2, 5.3 and 5.4 show each category graphically.

	GOAL		**Y**		**SUB**	
	B	***N***	**B**	***N***	**B**	***N***
Precision	0.96	0.86	0	0.86	0	1.00
Recall	0.88	0.99	0	0.75	0	0.06
Accuracy	0.85	0.86	0	0.66	0	0.06
F_1	0.91	0.92	0	0.80	0	0.11

Table 5.4: Single-stream macro-averaged metrics for Baseline and *NEDISS17*

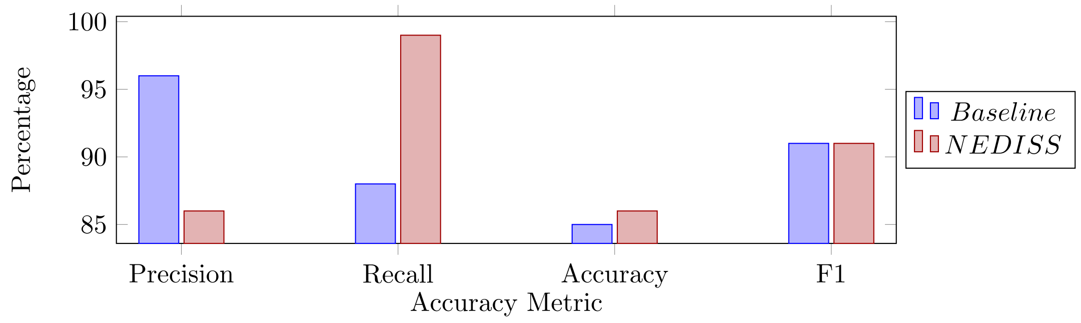


Figure 5.2: Bar chart of GOAL macro-averaged metrics for baseline and *NEDISS17*

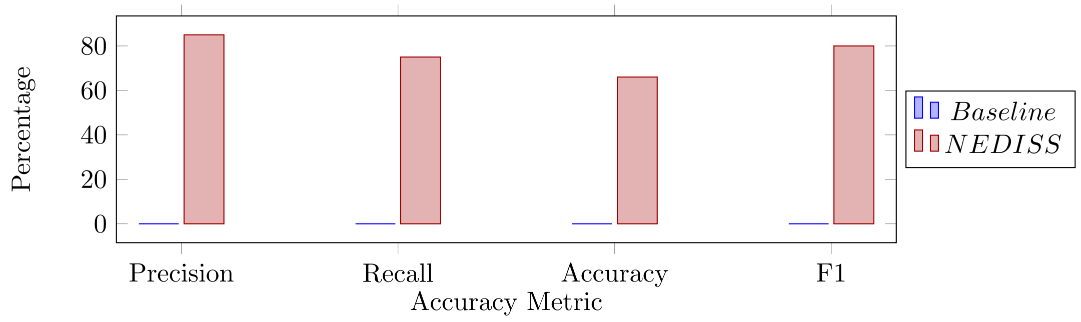


Figure 5.3: Bar chart of YELLOW CARD macro-averaged metrics for baseline and *NEDISS17*

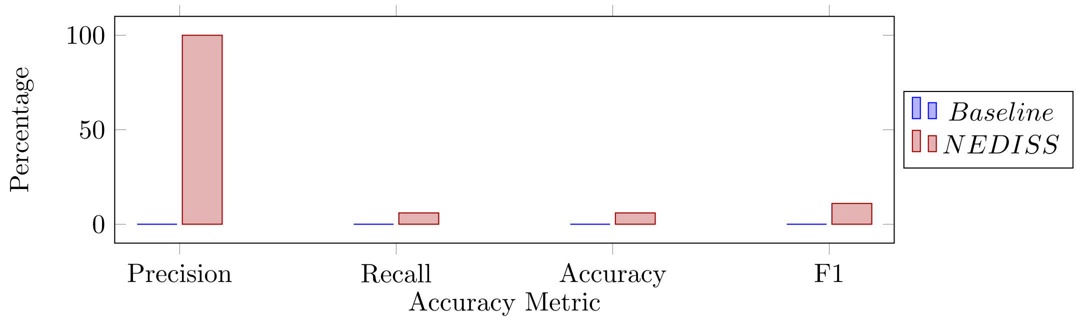


Figure 5.4: Bar chart of SUBSTITUTION macro-averaged metrics for baseline and *NEDISS17*

Finally, a value of $p \leq 0.01$ was computed by both the *s-test* and *S-test* statistical significance tests.

5.4.1.4 Experiment Findings

The findings of this experiment can be summarized as follows:

NEDISS:

- has a lower Precision than the baseline.
- has a higher Recall, Accuracy and F_1 score than the baseline.
- detects lower volume events significantly better than the baseline system.
- is a significant improvement over the baseline system.

The accuracy metrics for *NEDISS* are as expected when comparing to the baseline system. Recall and Accuracy exhibit an $\approx 2\times$ increase over the baseline system. However, the Precision value is lower than the baseline. This can be attributed to the more sensitive event detection design of *NEDISS*. Overall, the baseline system has a very low *false positive* count and therefore shows a high Precision value. The increased coverage of events detected by *NEDISS* naturally leads to more *false positive* potentials and explains why the *Precision* value is lower than the baseline. This increased coverage is evident by the improvements in each of the other accuracy metrics.

The *macro-averaged* accuracy metrics for the two system provide some interesting analysis. Both systems preform equally well for the high volume event, i.e. GOAL. However, there is a drastic difference between the two systems the lower volume events, i.e. YELLOW CARD and SUBSTITUTION. The baseline system failed to correctly detect a single one of these events throughout the *experimental* dataset. This is as expected, as the baseline system relies on document volume to detect events, and these events do not cause a significant rise in document volume. On the other hand, *NEDISS* performs well for these events. The accuracy scores for YELLOW CARD events are significantly higher than those for the SUBSTITUTION events. This can be explained by the real world importance of these events. Yellow cards are simply more discussed than substitutions during soccer matches. Also, there is a limited vocabulary used when a yellow card occurs and is easy to classify. However, people dramatically differ in the vocabulary they use when discussing substitutions. These two factors explain why there is a large difference between these accuracy scores.

Finally, *NEDISS* is a significant improvement over the baseline system. A value of $p \leq 0.01$ was computed by both *s-test* and *S-test*. Both *null hypotheses* can be rejected and therefore *NEDISS* is a significant improvement over the baseline system on both a *macro* and *micro* scale.

5.4.2 Comparison between NEDISS, Baseline and Actual Events in a Multi-match Stream

5.4.2.1 Objectives

The objectives of this experiment are the same as before:

1. Provide accuracy metrics for the *NEDISS* system.
2. Compare the accuracy of *NEDISS* to a baseline system.
3. Determine if *NEDISS* performs statistically better than the baseline system.

5.4.2.2 Experimental Method

This experiment uses the same methods as before, with one variation. Both systems are run with a stream of documents from multiple soccer matches at once. This is in direct contrast to the previous experiment where the systems were detecting events during a single match stream.

5.4.2.3 Experiment Results

Table 5.5 displays the Precision, Recall, Accuracy and F_1 score metrics for the baseline system and *NEDISS17*. Figure 5.5 shows these values in a bar chart. These metrics were computed over the experimental dataset by means of *micro-averaging*.

	Baseline	**NEDISS17**
Precision	1.00	0.92
Recall	0.05	0.38
Accuracy	0.05	0.37
F_1	0.01	0.54

Table 5.5: Multi-stream micro-averaged metrics for Baseline and *NEDISS17*

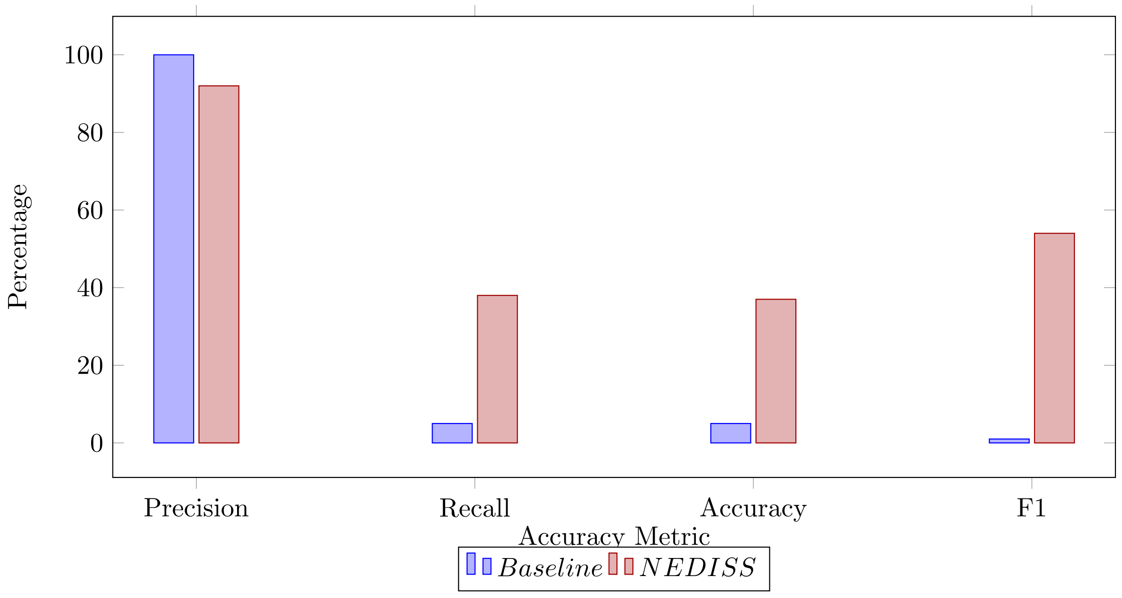


Figure 5.5: Bar chart of micro-averaged metrics for baseline and *NEDISS17*

Table 5.6 displays the same metrics as shown previously between the two systems. However, they are split into GOAL, YELLOW CARD (Y) and SUBSTITUTION (SUB) *macro-averaged* values. Figures 5.6, 5.7 and 5.8 show each category graphically.

	GOAL		**Y**		**SUB**	
	B	***N***	**B**	***N***	**B**	***N***
Precision	1.00	0.83	0	1.00	0	1.00
Recall	0.23	0.77	0	0.50	0	0.04
Accuracy	0.23	0.67	0	0.50	0	0.04
F_1	0.37	0.80	0	0.67	0	0.08

Table 5.6: Multi-stream macro-averaged metrics for Baseline and *NEDISS17*

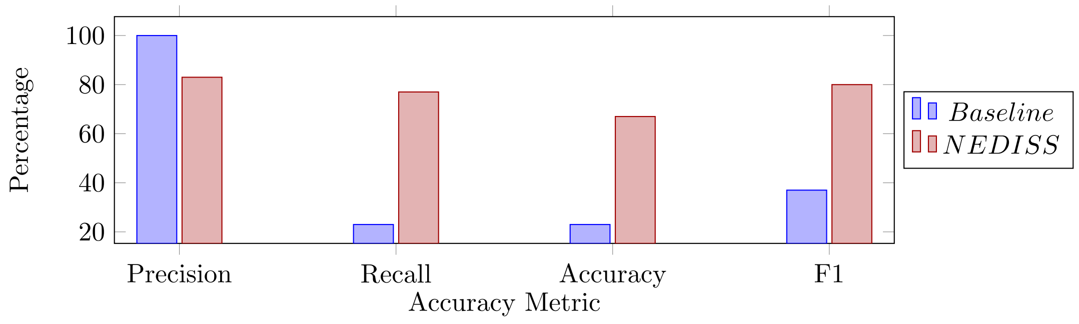


Figure 5.6: Bar chart of GOAL macro-averaged metrics for baseline and *NEDISS17*

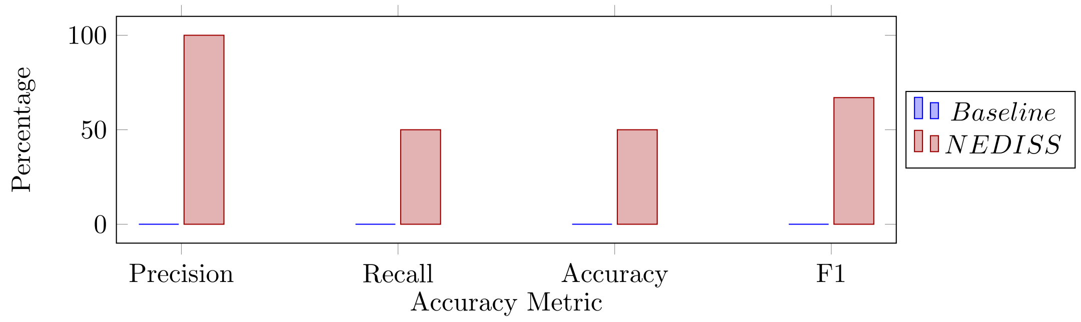


Figure 5.7: Bar chart of YELLOW CARD macro-averaged metrics for baseline and *NEDISS17*

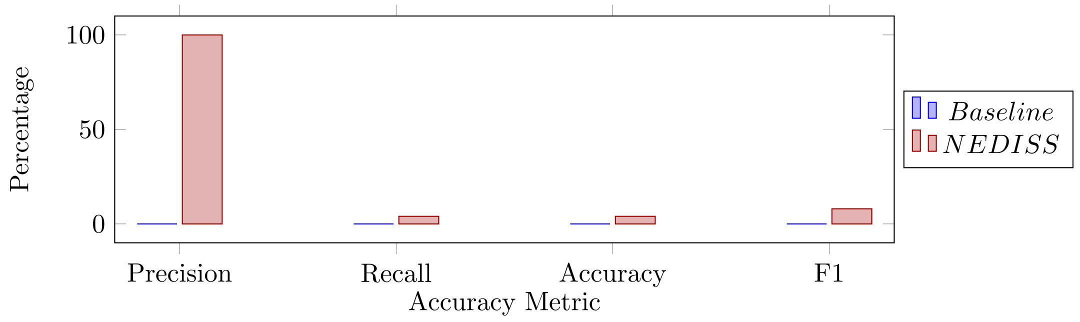


Figure 5.8: Bar chart of SUBSTITUTION macro-averaged metrics for baseline and *NEDISS17*

Finally, a value of $p \leq 0.01$ was computed by both the *s-test* and *S-test* statistical significance tests.

5.4.2.4 Experiment Findings

The findings of this experiment can be summarized as follows:

- Both systems experience a dramatic decrease in performance Vs. a single stream context.
- Precision is perfect for lower volume events.
- *NEDISS* performs better than the baseline in this context.

As expected, there is a dramatic increase in each accuracy metric in comparison to the systems performance in the single stream context. This can be attributed to the increased noise in the stream and the difficulty of document separation into different matches. It is clear that *NEDISS* performs better at this. The use of document vectors leads to separate clusters for documents that describe different match-event pairs. On the other hand, the baseline system perform poorly in each accuracy metric. This can be attributed to the reliance on the temporal bursts of activity in the stream. With a multi-stream context, these bursts are harder to detect and multiple event occurrences at the same time cause *false negatives*.

The *NEDISS* results indicate a high Precision metric for lower volume events, which directly implies a low number of *false positives*. The increased stream volume with the multiple matches results in an increase of clusters which do not get classified as events. The increased number of clusters being created directly decreased the ability of the system to detect lower volume events, as their clusters are smaller and deleted more aggressively. This results in lower *true positive* values for these events and thus decreased *false negative*. In brief, if a lower volume event manages to get detected during this multiple match stream, it is highly likely to be an accurate detection. Thus the high and low Recall values for these events.

Finally, *NEDISS* is again a significant improvement over the baseline system. A value of $p \leq 0.01$ was computed by both *s-test* and *S-test*. Both *null hypotheses* can be rejected and therefore *NEDISS* is a significant improvement over the baseline system on both a *macro* and *micro* scale. This is intuitive from the results shown, as the baseline system performed badly during this experiment.

5.4.3 Comparing different growth_rate parameters for NEDISS

5.4.3.1 Objectives

The objectives of this experiment are as follows:

1. Investigate the effects of the *growth_rate* parameter on the accuracy of *NEDISS*.
2. Investigate the effects of the *growth_rate* parameter on the number of clusters that are classified as potential events.

5.4.3.2 Experimental Method

This experiment compares different values for the *growth_rate* parameter. From table 5.1, the *NEDISS11, NEDISS17, NEDISS25, NEDISS34* and *NEDISS40* systems are evaluated in this experiment. These systems have *growth_rate* values of 1.1, 1.7, 2.5, 3.4 and 4.0 respectively. The experimental dataset previously discussed was used and each system operated on single match streams. Recall, Precision, Accuracy and F_1 score metrics were computed on both a *macro-averaging* and *mirco-averaging* scale for each. The events outlined in table 5.2 were used to calculate these metrics. Finally, an *s-test* and *S-test* was computed for each system pair.

5.4.3.3 Experiment Results

Table 5.4.2.4 displays the *Precision, Recall, Accuracy* and F_1 score metrics for each system. Figure 5.9 shows these values in a bar graph. These metrics were computed over the *experimental* dataset by means of *micro-averaging*.

	NEDISS11	***NEDISS17***	**NEDISS25**	**NEDISS34**	**NEDISS40**
Precision	0.83	0.85	0.83	0.89	0.94
Recall	0.49	0.53	0.32	0.25	0.16
Accuracy	0.45	0.49	0.30	0.24	0.16
F_1	0.62	0.65	0.46	0.39	0.28

Table 5.7: Micro-averaged metrics for various *NEDISS* configurations

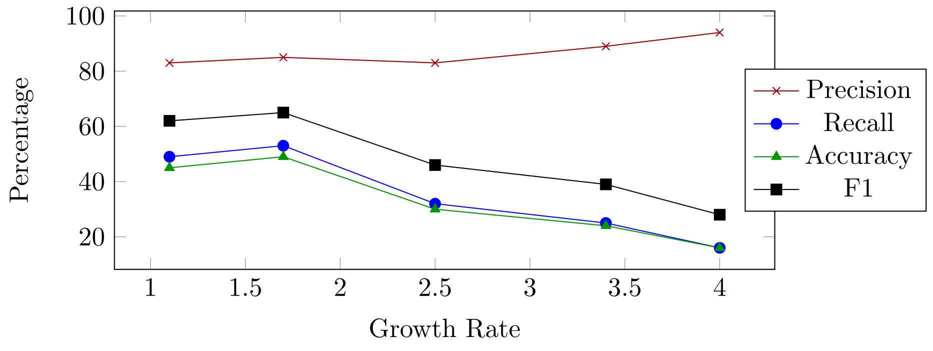


Figure 5.9: Growth rate and Accuracy metric bar chart

Table 5.8 displays the accuracy metrics for each *NEDISS* configuration, for each of the *macro-averaged* events. Figures 5.10, 5.11 and 5.12 show these values graphically.

	NEDISS11	**NEDISS17**	**NEDISS25**	**NEDISS34**	**NEDISS40**
Precision	0.76	0.86	0.78	0.89	0.92
Recall	1.00	1.00	0.84	0.64	0.48
Accuracy	0.76	0.86	0.68	0.59	0.46
F_1	0.86	0.93	0.80	0.74	0.63

(a) GOAL

	NEDISS11	**NEDISS17**	**NEDISS25**	**NEDISS34**	**NEDISS40**
Precision	0.83	0.86	0.67	0.88	0.75
Recall	0.63	0.75	0.25	0.44	0.19
Accuracy	0.56	0.66	0.22	0.41	0.18
F_1	0.71	0.80	0.36	0.58	0.30

(b) YELLOW

	NEDISS11	**NEDISS17**	**NEDISS25**	**NEDISS34**	**NEDISS40**
Precision	1.00	1.00	0	0	0
Recall	0.05	0.06	0	0	0
Accuracy	0.05	0.06	0	0	0
F_1	0.10	0.11	0	0	0

(c) SUBSTITUTION

Table 5.8: Macro-averged metrics for various *NEDISS* growth rates

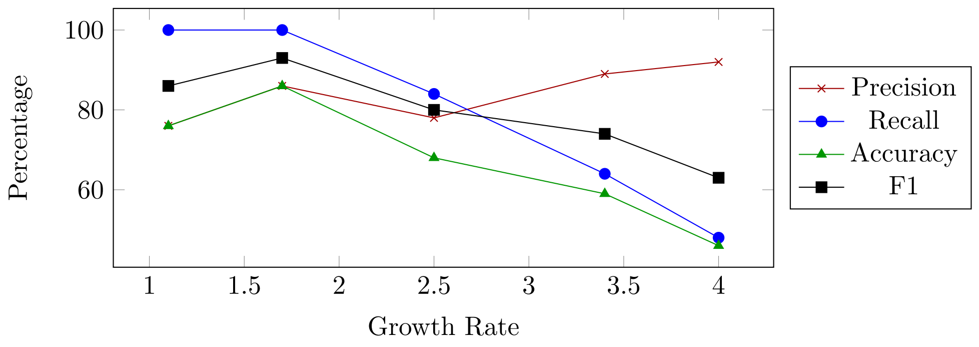


Figure 5.10: GOAL event: Growth rate and Accuracy metric bar graph

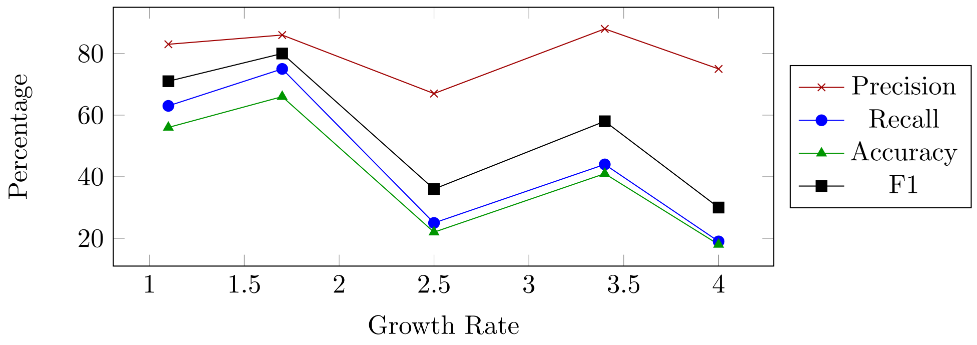


Figure 5.11: YELLOW event: Growth rate and Accuracy metric bar graph

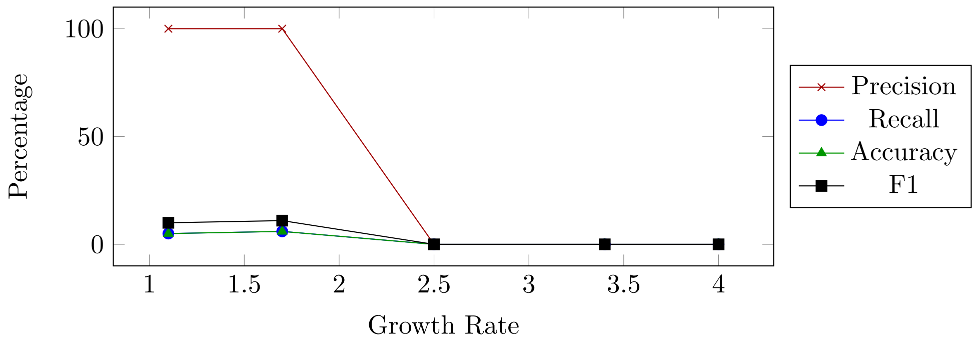


Figure 5.12: SUBSTITUTION event: Growth rate and Accuracy metric bar graph

Figure 5.13 shows the number of clusters classified as potential events for each of the systems during a selection of the experimental matches.

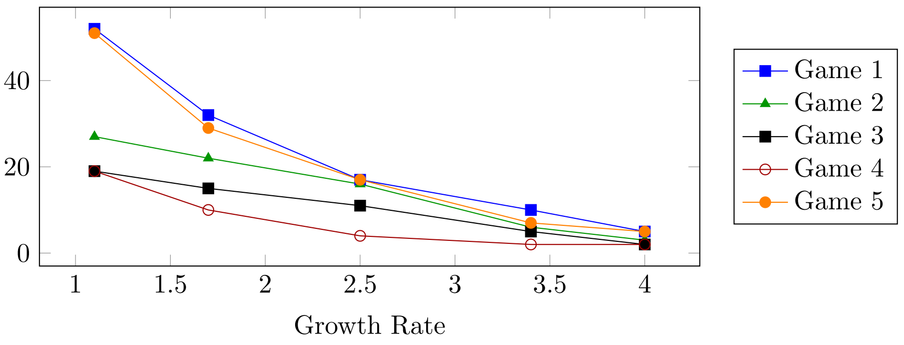


Figure 5.13: Number of clusters classified as events for various *NEDISS* configurations

Finally, table 5.9 outlines the results of the significance tests: *s-test* and *S-test*.

	NEDISS11	**NEDISS17**	**NEDISS25**	**NEDISS34**	**NEDISS40**
NEDISS11	■	≥ 0.1	≤ 0.01	≤ 0.01	≤ 0.01
NEDISS17	≥ 0.1	■	≤ 0.01	≤ 0.01	≤ 0.01
NEDISS25	≥ 0.1	≥ 0.1	■	$0.01 \leq p \leq 0.05$	≤ 0.01
NEDISS34	≥ 0.1	≥ 0.1	$0.05 \leq p \leq 0.1$	■	$0.01 \leq p \leq 0.05$
NEDISS40	≥ 0.1	≥ 0.1	≥ 0.1	$0.01 \leq p \leq 0.05$	■

(a) s-test

	NEDISS11	**NEDISS17**	**NEDISS25**	**NEDISS34**	**NEDISS40**
NEDISS11	■	≥ 0.1	≤ 0.01	≤ 0.01	≤ 0.01
NEDISS17	≥ 0.1	■	≤ 0.01	≤ 0.01	≤ 0.01
NEDISS25	≥ 0.1	≥ 0.1	■	≥ 0.1	≤ 0.01
NEDISS34	≥ 0.1	≥ 0.1	≥ 0.1	■	p ≥ 0.1
NEDISS40	≥ 0.1	≥ 0.1	≥ 0.1	$0.01 \leq p \leq 0.05$	■

(b) S-test

Table 5.9: P-values of various combinations of *NEDISS* growth paramater systems

5.4.3.4 Experiment Findings

The findings of this experiment can be summarized as follows:

- The *growth_rate* parameter has a dramatic effect on the performance of *NEDISS*
- Precision is proportional to the *growth_rate*
- Recall, Accuracy and F_1 metrics are inversely proportional to the *growth_rate*
- There is an exponential distribution between number of clusters classified as potential events and the *growth_rate*
- A value of 1.7 performs the best of the different values tested

The *growth_rate* parameter has a dramatic effect on the performance of *NEDISS*. A value that is too low results in high values for the accuracy metrics, but a large number of clusters are classified as potential events. This results in a large number of duplicate events being detected and extra computational time necessary to determine the duplicity.

The results indicate that the Precision metric is linearly proportional to the *growth_rate*. This is as expected since the higher the *growth_rate* rate, the less clusters will be classified as events. Therefore, less *false positivies* will be produced by the system and the Precision metric will rise accordingly.

The other accuracy metrics are inversely proportional to the *growth_rate*. Again, this is as expected. A large value for the *growth_rate* will result in few clusters being classified as potential events. This leads to increased missed events, or *false negatives*. An increased number of *false negatives* directly results in lower values for Recall, Accuracy and F_1.

As per figure 5.13, there is an exponential distribution between the number of clusters classified as potential events and the *growth_rate*. Again, this is as expected since the *growth_rate* parameter directly controls whether or not a cluster is marked as an event. A high value will result in no clusters getting classified, while a zero value results in every cluster becoming classified.

The P-values in table 5.9 show the results of the significance tests between each possible pair of systems. Using these values to test the *null hypothesis* that there isn't a significant difference between the performance of each system, one can infer the following regarding the *growth_rate*:

- A value of 1.1 is better than 2.5, 3.4 and 4.0 on a macro and micro level.
- There is no significant difference between 1.1 and 1.7.
- A value of 1.7 is better than 2.5, 3.4 and 4.0 on a macro and micro level.
- A value of 2.5 is better than 3.4 on a macro level, and 4.0 on a macro and micro level.
- There is a strong presumption that 2.5 is also better than 3.4 on a micro level.
- Values of 3.4 and 4.0 aren't significantly different.

With the findings above, it is clear that a value of 1.7 is the best of the values evaluated. Although 1.1 preformed equally well, a higher value should always be preferred because of its affect on the number of clusters being classified as events and therefore computation running time. The findings of this experiment highlight the importance of experimentation to determine the optimal value for the *growth_rate* based on the desired accuracy and performance of the system.

5.4.4 Comparing a learned and given vocabulary for NEDISS

5.4.4.1 Objectives

The objective of this experiment is to determine if a learned vocabulary or a provided vocabulary is better for the *Vectorizer* component of *NEDISS*.

5.4.4.2 Experimental Method

This experiment compares two different *NEDISS* configurations: *NEDISS17* and *NEDISSL* as per table 5.1. The provided vocabulary was discussed previously in chapter 4 and can been found in Appendix A. The learned vocabulary was determined as the 0.05% of the most frequent words used throughout the training dataset. The experimental dataset previously discussed was used and both systems operated on single match streams. Recall, Precision, Accuracy and F_1 score metrics were computed on both a *macro-averaging* and *mirco-averaging* scale for these two systems. The events outlined in table 5.2 were used to calculate these metrics. Finally, an *s-test* and *S-test* was computed between each system.

5.4.4.3 Experiment Results

Table 5.14 displays the *Precision, Recall, Accuracy* and F_1 score metrics for the *NEDISS17* and *NEDISSL* systems. Figure 5.15 shows these values in a bar chart. These metrics were computed over the *experimental* dataset by means of *micro-averaging*.

	NEDISSL	**NEDISS17**
Precision	0.84	0.85
Recall	0.39	0.53
Accuracy	0.36	0.49
F_1	0.53	0.65

Figure 5.14: Micro-averaged metrics for *NEDISSL* and *NEDISS17*

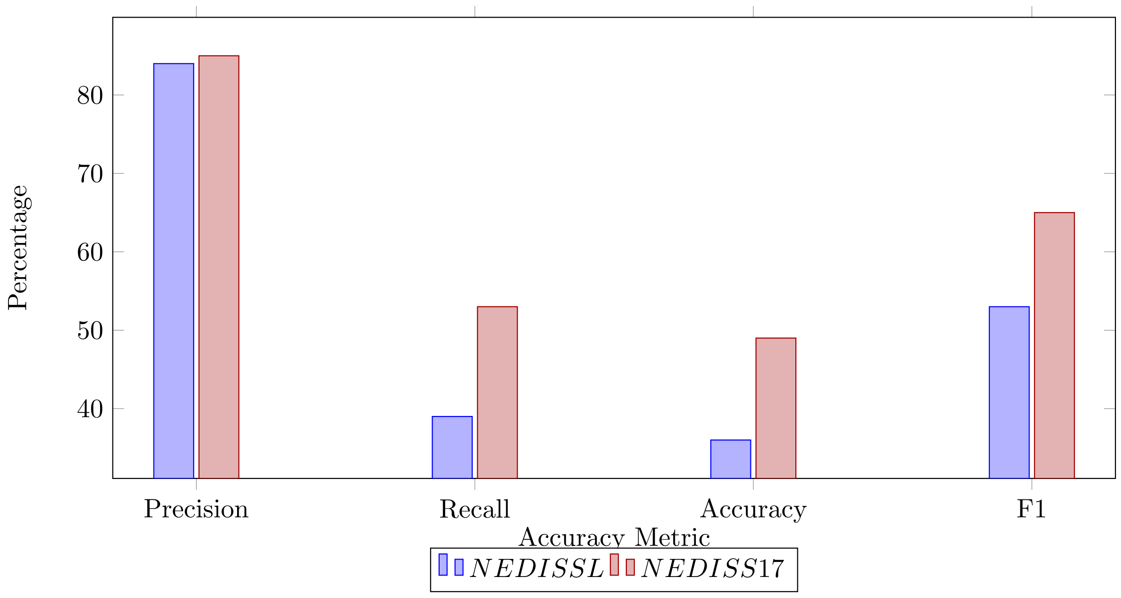


Figure 5.15: Bar chart of micro-averaged metrics for *NEDISSL* and *NEDISS17*

Table 5.10 displays the same metrics as shown previously between the two systems. However, they are split into GOAL, YELLOW CARD (Y) and SUBSTITUTION (SUB) *macro-averaged* values. Figures 5.16, 5.17 and 5.18 show each category graphically.

	GOAL		**Y**		**SUB**	
	NEDISSL	**NEDISS17**	**NEDISSL**	**NEDISS17**	**NEDISSL**	**NEDISS17**
Precision	0.80	0.86	1.00	0.86	1.00	1.00
Recall	0.80	0.99	0.44	0.75	0.03	0.06
Accuracy	0.67	0.86	0.44	0.66	0.03	0.06
F_1	0.80	0.92	0.61	0.80	0.03	0.11

Table 5.10: Macro-averaged metrics for *NEDISSL* and *NEDISS17*

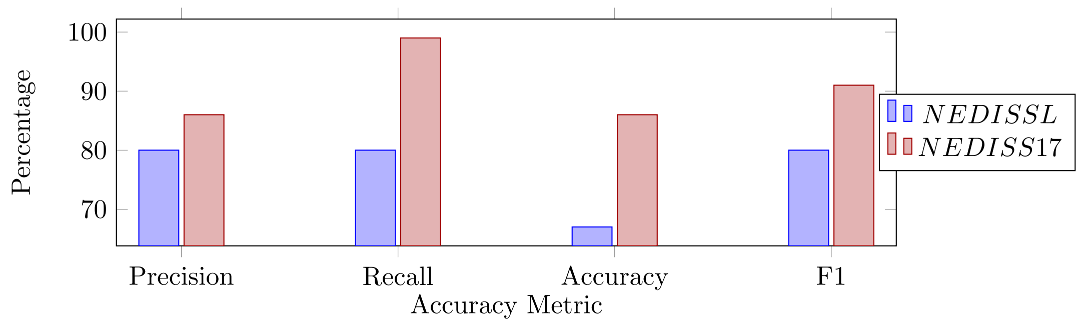


Figure 5.16: Bar chart of GOAL macro-averaged metrics for *NEDISSL* and *NEDISS17*

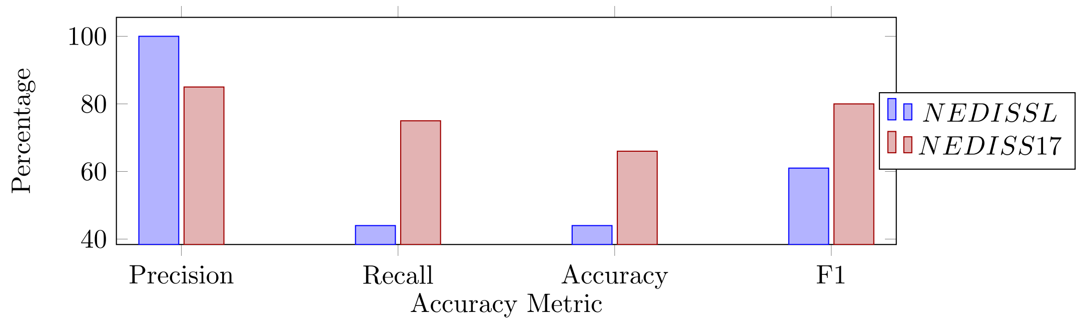


Figure 5.17: Bar chart of YELLOW CARD macro-averaged metrics for *NEDISSL* and *NEDISS17*

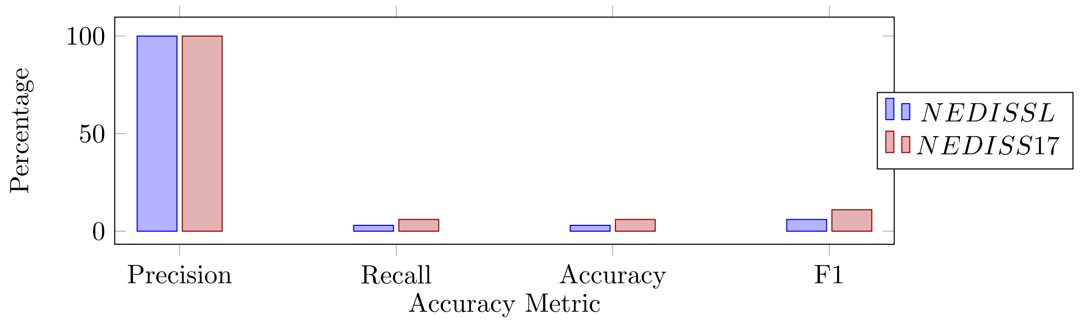


Figure 5.18: Bar chart of SUBSTITUTION macro-averaged metrics for *NEDISSL* and *NEDISS17*

Finally, a value of $p \leq 0.01$ was computed by both the *s-test* and *S-test* statistical significance tests.

5.4.4.4 Experiment Findings

This experiment found that a provided vocabulary performs better than a learned one. This conclusion can be drawn from the improved values for each accuracy metric and the indication of the significances tests. This is as expected, since the provided vocabulary was designed specifically to describe documents that would correlate to events during these matches. Irrelevant documents score lowly using a vectorizer object trained on this vocabulary. On the other hand, the learned vocabulary was learned via a noisy Twitter dataset. The increase in irrelevant terms in the vector space increase the difficulty of clustering similar documents. This is most evident with the performance of *NEDISSL* with the lower volume events. There is a significant decrease in its performance in comparison to the provided vocabulary. An explanation for this would be the intuition that the terms used during these low volume events may not occur in the learned vocabulary due to their low frequency in relation to the entire term set.

5.4.5 Investigating the runtime cost of Named Entity extraction

5.4.5.1 Experiment Objectives

The objective of this experiment is to investigate the runtime cost incurred when named entity extraction is integrated into *NEDISS*.

5.4.5.2 Experimental Method

This experiment compares the running time of two different *NEDISS* configurations: *NEDISS17* and *NEDISSNER* as per table 5.1. Each system was run with the experimental dataset in a single match stream context. The UNIX time utility was used to capture *real, user* and *sys* values. Due to the varying sizes of the match streams, the relative difference between the two systems was calculated and averaged across all matches.

5.4.5.3 Experiment Results

The results of this experiment are displayed below in table 5.11.

real	**user**	**sys**
2.38×	3.81×	33×

Table 5.11: Speedup difference for *NEDISS17* compared to *NEDISSNER*

5.4.5.4 Experiment Findings

This experiment has found that there is a significant running cost when integrating named entity extracting into *NEDISS*. The *user* and *sys* speedups are most relevant. The 3.81× increase for the *user* running time indicates a significant increase in the amount of time executing code within the process that isn't making kernel calls. The 33× increase for the *sys* is a dramatic result. This shows the large number of kernel calls required when using a state of the art named entity recognizer. These increases indicate that a system should carefully consider whether named entity extraction is necessary. In the case of *NEDISS*, it is simply used to improve the output by adding named entities that were involved in the events. This is not crucial to the overall goal of the system which is to detect events. Given that *NEDISS* operates with high volume social streams, it is critical that the system has an efficient running time. Therefore, this experiment has shown that a named entity extractor should not be integrated with the system.

5.5 Summary of experiments

This chapter presented 5 experiments which evaluate the performance of the *NEDISS* system.

The first experiment used accuracy metrics and significance tests to compare *NEDISS* to a baseline system, when operating in single match streams. The experiment concluded that *NEDISS* represents a significant improvement over the baseline system, particularly in the detection of low volume events.

The second experiment used accuracy metrics and significance tests to compare *NEDISS* to a baseline system, when operating in multiple match streams. Again, the experiment concluded that *NEDISS* represents a significant improvement over the baseline system.

The third experiment presented an evaluation of the *growth_rate* parameter of *NEDISS*. Five different values were used: 1.1, 1.7, 2.5, 3.4 and 4.0. This experiment showed the relationship between each accuracy metric and the *growth_rate*, and showed the exponential distribution between it and the number of clusters classified as potential events. The experiment concluded that 1.7 was the best value tested and gave evidence of the necessary experiments needed to determine a *growth_rate* parameter when building a similar system.

The penultimate experiment compared the performance of *NEDISS17* and *NEDISSL*. These systems differ only in the vocabulary used to train the *Vectorizer* component. The experiment concluded that the system with the provided vocabulary performed significantly better than that with the learned.

The final experiment analyzed the running cost involved with named entity extraction. The *real, user* and *sys* running time metrics were captured for *NEDISS17* and *NEDISSNER*. The difference in running time was dramatic and the experiment concluded that named entity extraction is too costly in this context of this work.

As a final note, Appendix A.3 contains some confusion matrices with values obtained from the various systems during individual matches.

Chapter 6

Conclusion

6.1 Introduction

This chapter concludes this report. Each of the objectives outlined in the introduction is revisited and the outcome is discussed. Next, the key contributions of this report are summarized. Finally, indications for future work are given.

6.2 Objectives Revisited

To perform a survey of the state of the art in the area of on-line NED and social streams.

This objective was achieved with chapter 2. The survey presented various key techniques in use throughout NED. Firstly, a background of the information retrieval techniques used in the NED task, such as the *vector space model* and $tf \cdot idf$, was presented. The methods through which these techniques are used to produce clusters of documents was then discussed. The state of the art in clustering algorithms to solve the NED task was then presented. The chapter then proceeded to survey the NED task in an on-line social stream setting. The key technique utilized here is the temporal property introduced by Shakaki et al. [3]. The survey continued on to discuss how various authors used *locality sensitive hashing* to solve the nearest neighbour problem and thus making clustering algorithms feasible in the on-line social stream context. The NED section of the survey concluded with a discussion of the use of semantics to improve system performance. The Twitter stream was then surveyed. Its properties and the challenges it brings to the NED task were discussed. Various state of the art systems that operate with Twitter were then presented. Finally, chapter 2 concludes with a discussion of the baseline system and a critical analysis of the survey.

To create an on-line NED system that operates with social streams and is capable of detecting structured live events.

This objective was fully achieved. *NEDISS* represents a NED system that is capable of detecting structured live events in an on-line social stream setting. The system was designed and implemented using the state of the art survey. The focus of this work was to detect events during live soccer matches using the Twitter stream, and *NEDISS* was implemented and evaluated within this area. The system architecture and implementation was entirely the work of the student.

To evaluate this system in comparison to a highly published baseline system.

Chapter 5 presents the evaluations that fulfill this objective. The *SportsSense* [36] system was chosen as the baseline system because of its reliance on the highly published temporal heuristic, first presented by Shakaki et al. [3]. The student implemented this system entirely as described by the authors. The experiments presented in chapter 5 represent a comprehensive evaluation of *NEDISS*. A comparative study was preformed between *NEDISS* and the baseline system, using an experimental dataset with annotated real world events. Furthermore, various aspects of the *NEDISS* design were evaluated. Firstly, a comparison between various *growth_rate* parameters was performed. Secondly, the use of a learned vocabulary versus a provided one was investigated. Finally, the runtime effects of integrating named entity extraction into the system were presented. The experiments included a discussion of common accuracy metrics and significance tests were preformed where appropriate. The findings and conclusions drawn from this evaluation are significant and should aid future researchers with their similar NED systems.

6.3 Contributions

The contributions of this work can be summarized as follows. Firstly, NED systems should not use the temporal heuristic presented by Shakaki et al. [3] on a global scale. Rather, this heuristic should be used on a much more granular level, ideally per event. This conclusion can be drawn from the evaluations between *NEDISS* and the baseline system. Secondly, this work presents a comprehensive study on the effects of cluster growth rate within a social stream setting. The significant differences between the performance of the system with varying values for the growth rate should motivate further researchers to carefully select their growth rate values. Again, this difference is evident with the evaluations presented in this work. Finally, this work presents a real implementation of an on-line NED system with Twitter using locality sensitive hashing. The techniques surveyed in the state of the art were used and a comprehensive evaluation was performed. Future researchers can use the values and evaluations in chapter 5 to compare their work to *NEDISS*.

6.4 Future Work

This report concludes with a discussion of potential future work. Firstly, various authors have reported success applying semantic analysis not utilized in this work. For instance, Petrovic et al. [27] used paraphrases to improved NED with Twitter. They also use *locality sensitive hashing* and the Twitter stream. It is likely that the use of paraphrases would improve the detection of low volume events which don't have a concrete vocabulary associated with them. An example from this work would be SUBSTITUTION events, as evaluated in chapter 5. Secondly, other authors have used Out-Of-Vocabulary (OOV) processing to improve the quality of Twitter documents. Again, it is likely this will improve the output of a system like *NEDISS* due to the poor grammar and vocabulary used by Twitter users during live events.

Another area of future work would be in machine learning to create a topic-conditioned classifier. Yang et al. [13] presented the idea of first separating documents into broad event categories before performing event detection. They report promising results and this approach lends itself nicely to structured live events. The idea would be to have the documents separated to each possible event (GOAL, PENALTY, etc.) before entering the NED system. There is then the potential to use different NED systems per event, offering finer granularity on the algorithm designs.

Appendix A

Sports Game Keywords

A.1 Soccer Team Keywords

arsenal	afc	gunners	gooners	coyg
aston villa	villa	villans	avfc	cardiff city
bluebirds	cardiffcity	cardiff	chelsea	blues
cfc	crystal palace	palace	eagles	cpfc
everton	toffees	efc	coyb	fulham
cottagers	ffc	londonsoriginal	hull city	hull
tigers	utt	liverpool	reds	lfc
pool	ynwa	manchester city	manchester	citizens
mcfc	manchester united	red devils	manunited	united
manu	man u	mufc	man utd	manutd
newcastle united	newcastle	magpies	nufc	norwich city
norwich	canaries	ncfc	southampton	saints
saintsfc	stoke city	stoke	potters	scfc
sunderland	black cats	safc	redandwhitearmy	swansea city
swansea	swans	tottenham hotspur	spurs	lilywhites
tottenham	hotspur	coys	west bromwich albion	west brom
baggies	wba	west ham united	west ham	hammers
whufc				

Table A.1: Soccer Team names, abbreviations, nicknames and common hashtags

A.2 Soccer Terminology

goal	half time	penalty	score	goals
bpl	scored	scores	offside	full time
fulltime	halftime	yellow card	red card	sub
substitution	replaced by	sent off	change for	corner
premierleague	premier league	offside	off side	freekick
free kick	handball	hand ball		

Table A.2: Soccer game terminology

A.3 Confusion Matrices for Various Soccer Mathes

		Actual Results							
		Goal	**Penalty**	**Yellow**	**Red**	**HT**	**FT**	**Sub**	**Null**
Test Output	**Goal**	1							
	Penalty								
	Yellow			2					
	Red								
	HT								
	FT								
	Sub								
	Missed					1	1	4	

Table A.3: NEDISSL: Swansea Vs. Southampton

		Actual Results							
		Goal	**Penalty**	**Yellow**	**Red**	**HT**	**FT**	**Sub**	**Null**
Test Output	**Goal**	6							3
	Penalty								1
	Yellow			3					
	Red								
	HT					1			
	FT								
	Sub								
	Missed			2			1	5	

Table A.4: NEDISSL: Liverpool Vs. Crystal Palace

		Actual Results							
		Goal	**Penalty**	**Yellow**	**Red**	**HT**	**FT**	**Sub**	**Null**
Test Output	**Goal**	1							
	Penalty								1
	Yellow								
	Red								
	HT								
	FT								
	Sub								
	Missed			2		1	1	4	

(a) Basline System

		Actual Results							
		Goal	**Penalty**	**Yellow**	**Red**	**HT**	**FT**	**Sub**	**Null**
Test Output	**Goal**	1							1
	Penalty								1
	Yellow			2					
	Red								1
	HT					1			
	FT								
	Sub								
	Missed						1	4	

(b) *NEDISS* System

Table A.5: NESISS Vs. Baseline: Swansea Vs. Southampton

Test Output	Actual Results							
	Goal	**Penalty**	**Yellow**	**Red**	**HT**	**FT**	**Sub**	**Null**
Goal	1							
Penalty								
Yellow								
Red								
HT					1			
FT								
Sub								
Missed			3			1	6	

(a) Basline System

Test Output	Actual Results							
	Goal	**Penalty**	**Yellow**	**Red**	**HT**	**FT**	**Sub**	**Null**
Goal	1							1
Penalty								
Yellow			3					
Red								
HT					1			
FT						1		
Sub							1	
Missed							5	

(b) *NEDISS* System

Table A.6: NESISS Vs. Baseline: Manchester United Vs. Sunderland

Bibliography

[1] Salton, G., A. Wong, and C.-S. Yang (1975) "A vector space model for automatic indexing," *Communications of the ACM*, **18**(11), pp. 613–620.

[2] Van Durme, B. and A. Lall (2010) "Online generation of locality sensitive hash signatures," in *Proceedings of the ACL 2010 Conference Short Papers*, Association for Computational Linguistics, pp. 231–235.

[3] Sakaki, T., M. Okazaki, and Y. Matsuo (2010) "Earthquake shakes Twitter users: real-time event detection by social sensors," in *Proceedings of the 19th international conference on World wide web*, ACM, pp. 851–860.

[4] Yang, Y., T. Pierce, and J. Carbonell (1998) "A study of retrospective and on-line event detection," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 28–36.

[5] Analytics, P. (2009) "Twitter Study–August 2009," *San Antonio, TX: Pear Analytics. Available at: www. pearanalytics. com/blog/wp-content/uploads/2010/05/Twitter-Study-August-2009. pdf.*

[6] Allan, J. (2002) "Topic detection and tracking: event-based information organization," .

[7] Allan, J., R. Papka, and V. Lavrenko (1998) "On-line new event detection and tracking," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 37–45.

[8] Muthukrishnan, S. (2005) "Data streams: Algorithms and applications," .

[9] Allan, J., V. Lavrenko, and H. Jin (2000) "First story detection in TDT is hard," in *Proceedings of the ninth international conference on Information and knowledge management*, ACM, pp. 374–381.

[10] Brants, T., F. Chen, and A. Farahat (2003) "A system for new event detection," in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, ACM, pp. 330–337.

[11] Schinas, E., S. Papadopoulos, S. Diplaris, Y. Kompatsiaris, Y. Mass, J. Herzig, and L. Boudakidis (2013) "Eventsense: Capturing the pulse of large-scale events by mining social media streams," in *Proceedings of the 17th Panhellenic Conference on Informatics*, ACM, pp. 17–24.

[12] Makkonen, J., H. Ahonen-Myka, and M. Salmenkivi (2002) "Applying semantic classes in event detection and tracking," in *Proceedings of International Conference on Natural Language Processing (ICON 2002)*, pp. 175–183.

[13] Yang, Y., J. Zhang, J. Carbonell, and C. Jin (2002) "Topic-conditioned novelty detection," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, pp. 688–693.

[14] Weiler, A., S. Mansmann, and M. H. Scholl (2012) "Towards an advanced system for real-time event detection in high-volume data streams," in *Proceedings of the 5th Ph. D. workshop on Information and knowledge*, ACM, pp. 87–90.

[15] Zhao, W. X., B. Shu, J. Jiang, Y. Song, H. Yan, and X. Li (2012) "Identifying event-related bursts via social media activities," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Association for Computational Linguistics, pp. 1466–1477.

[16] Aggarwal, C. C. and K. Subbian (2012) "Event Detection in Social Streams." in *SDM*, vol. 12, pp. 624–635.

[17] Luo, G., C. Tang, and P. S. Yu (2007) "Resource-adaptive real-time new event detection," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ACM, pp. 497–508.

[18] Papka, R. and J. Allan (1998) "On-line new event detection using single pass clustering," *University of Massachusetts, Amherst*.

[19] Sayyadi, H., M. Hurst, and A. Maykov (2009) "Event Detection and Tracking in Social Streams." *ICWSM*.

[20] Indyk, P. and R. Motwani (1998) "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ACM, pp. 604–613.

[21] Datar, M., N. Immorlica, P. Indyk, and V. S. Mirrokni (2004) "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, ACM, pp. 253–262.

[22] Charikar, M. S. (2002) "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, ACM, pp. 380–388.

[23] Andoni, A. and P. Indyk (2006) "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, IEEE, pp. 459–468.

[24] Petrović, S., M. Osborne, and V. Lavrenko (2010) "Streaming first story detection with application to twitter," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, pp. 181–189.

[25] Zhang, K., J. Zi, and L. G. Wu (2007) "New event detection based on indexing-tree and named entity," in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 215–222.

[26] Kumaran, G. and J. Allan (2004) "Text classification and named entities for new event detection," in *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 297–304.

[27] Petrović, S., M. Osborne, and V. Lavrenko (2012) "Using paraphrases for improving first story detection in news and Twitter," in *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Association for Computational Linguistics, pp. 338–346.

[28] Java, A., X. Song, T. Finin, and B. Tseng (2007) "Why we twitter: understanding microblogging usage and communities," in *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis*, ACM, pp. 56–65.

[29] Krishnamurthy, B., P. Gill, and M. Arlitt (2008) "A few chirps about twitter," in *Proceedings of the first workshop on Online social networks*, ACM, pp. 19–24.

[30] Shamma, D. A., L. Kennedy, and E. F. Churchill (2009) "Tweet the debates: understanding community annotation of uncollected sources," in *Proceedings of the first SIGMM workshop on Social media*, ACM, pp. 3–10.

[31] Armstrong, S. "Understanding Twitter Activity During Live Sporting Events," .

[32] Weng, J. and B.-S. Lee (2011) "Event Detection in Twitter." *ICWSM*.

[33] Tsolmon, B., A.-R. Kwon, and K.-S. Lee (2012) "Extracting social events based on timeline and sentiment analysis in twitter corpus," *Natural Language Processing and Information Systems*, pp. 265–270.

[34] Becker, H., M. Naaman, and L. Gravano (2011) "Beyond Trending Topics: Real-World Event Identification on Twitter." *ICWSM*, **11**, pp. 438–441.

[35] Marcus, A., M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller (2011) "Twitinfo: aggregating and visualizing microblogs for event exploration," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, pp. 227–236.

[36] Zhao, S., L. Zhong, J. Wickramasuriya, and V. Vasudevan (2011) "Human as real-time sensors of social and physical events: A case study of twitter and sports games," *arXiv preprint arXiv:1106.4300*.

[37] Sankaranarayanan, J., H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling (2009) "Twitterstand: news in tweets," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ACM, pp. 42–51.

[38] Lanagan, J. and A. F. Smeaton (2011) "Using twitter to detect and tag important events in live sports," *Artificial Intelligence*, pp. 542–545.

[39] Grier, C., K. Thomas, V. Paxson, and M. Zhang (2010) "@Spam: The Underground on 140 Characters or Less," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, ACM, New York, NY, USA, pp. 27–37.
URL `http://doi.acm.org/10.1145/1866307.1866311`

[40] Yang, Y. and X. Liu (1999) "A re-examination of text categorization methods," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 42–49.