NTNU
Norwegian University of
Science and Technology

Continuous Queries on Streaming Data

Sara Phrida K. Norrhall

Master of Science in Computer Science
Submission date: June 2018
Supervisor: Herindrasana Ramampiaro, IDI
Co-supervisor: Kjetil Nørvåg, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

We are all living in a world that is becoming more digital for every day. While people are connecting to the Internet, there has been an explosion of applications that are being used on a daily basis. All of these applications have especially one thing in common - they are all generating data that are valuable for stakeholders to understand their users. Despite that, few know how to take advantage of this information. To unlock the information hidden in these applications, the concept of visualizations of both stored and streaming data has been explored. Furthermore, this project has investigated how a system can be implemented that makes it possible to visualize relevant information based on the interests of users. To filter out relevant content, the system performs continuous queries defined by the user. The filtering allows the system to avoid unnecessary storage and indexing of irrelevant data, in addition to making the visualization both intuitive and straightforward to interpret as all information that would not provide the user with valuable insights are excluded. The project has also explored how Machine Learning can be used to unlock valuable information about the data, and thus a sentiment analysis was performed. This project has utilized available information provided by Twitter as proof of concept. The system has been implemented to be modular, allowing other sources of streaming data to be used. The proposed solution should also consider the various aspects of big, streaming data and tolerate high enough throughputs of events. Our experiments show the practicality and feasibility of the proposed approach.

Sammanfattning

Vi lever alla i en värld som för var dag blir allt mer digitaliserad. Samtidigt som människor kopplat upp sig mot internet så har det skett en explosion av applikationer som många av oss använder dagligen. Dessa applikationer har särskilt en sak gemensam – de genererar alla data värdefull för intressenter som vill förstå sina användare. Trots det, så vet få hur de skall dra nytta av denna information. För att utvinna informationen som dessa applikationer döljer så har en modell för att synliggöra både lagrad och strömmad data undersökts. Dessutom har detta projekt undersökt hur ett system kan implementeras som gör det möjligt att synliggöra relevant information baserad på användarnas intressen. För att filtrera ut relevant innehåll utförde systemet kontinuerliga frågor definierade av användarna. I tillägg tillät filtreringen systemet att undvika onödig lagring och indexering av irrelevant data, något som gjorde synliggörandet både intuitivt och okomplicerat, genom att all information som inte förser användarna med värdefull information har uteslutits. Projektet har också undersökt hur "Machine Learning" kan användas för att ge värdefull kunskap om dessa data och därför utfördes en sentimentanalys. Detta projekt har använt tillgänglig data från "Twitter" för att bevisa modellen. Systemet har implementerats för att vara modulärt och på så sätt kunna användas även för andra källor där strömmad data används. Den föreslagna lösningen tar också i beaktande olika aspekter av stora volymer strömmad data och kan tolerera händelser av tillräckligt höga nivåer av genomströmmande data. Experiment visar genomförbarheten av det föreslagna tillvägagångssättet.

Preface

This thesis is written by Sara Phrida Kristina Norrhall as part of a five years study program in Computer Science at the Norwegian University of Science and Technology (NTNU) in Trondheim. The thesis is the final requirement for a degree in Master of Science (MSc) with specialization in Databases and Search. The work was conducted during the spring semester of 2018.

Acknowledgements

The motivation for the project was formed together with associate professor Heri Ramampiaro at the Department of Computer Science (IDI). Together with professor Kjetil Nørvåg, they have been my supervisors during the last year, and I would like to express my sincere gratitude for their guidance and support. They have allowed me to work independently and explore things on my own. Although, they have always been there when guidance was needed. I would also like to thank my father for giving me valuable input throughout the year and my mother for giving me input on this report. You both are an inspiration to me. Finally, I would like to thank my boyfriend for his emotional support whenever needed.

Sara Phrida Kristina Norrhall 13.6.2018

Contents

List of Figures

List of Listings

List of Tables

Acronyms

BAD Big Active Data. 30

BDMS Big Data Management Systems. 30

BI Business Intelligence. 16

CCR Cumulative Citation Recommendation. 38

DAG Directed Acyclic Graph. 32

DBMS Database Management Systems. 30

DSMS Data Stream Management Systems. 30

ETL Extract, Transform, Load. 13

IBM International Business Machines Corporation. 36

IBMWA IBM Watson Analytics. 36

JSON JavaScript Object Notation. 45

KB Knowledge Base. 38

ML Machine Learning. 14

NLP Natural Language Processing. 14

SA Sentiment Analysis. 14

SQL Structured Query Language. 30

UDF User-Defined Function. 33

VCA Visual Content Analysis. 35

Chapter 1

Introduction

This chapter gives an introduction to the project by providing a background for the problem and research area. It will describe the motivation along with the primary goal, associated research questions and chosen research strategy. The scope and limitations will be explained in addition to the contributions of the project. Finally, the chapter will present an outline of the following sections.

1.1 Background

The thesis is concerned with streaming data and the use of visualization tools to gain insights into the incoming data. Furthermore, the thesis explores how the incoming data can be classified to avoid unnecessary storage of irrelevant information. The work presented in this thesis is a continuation of the results of the specialization project made as part of the course TDT4501 [1]. The specialization project showed how data from Twitter could be visualized in near real-time. However, since there were no constraints to what data should be sent into the system, all types of filtering had to happen in the visualization tool. To avoid unnecessary indexing and storage, we want to filter out spam and irrelevant information and only push data that are relevant to the user. In that way, the user will get a visualization of the relevant information that is easy to interpret. Such a system can be of use for companies and other stakeholders who wish to turn the information generated in applications into knowledge, allowing them to understand their users and get useful business insights.

Therefore, the main contribution of this project is to develop a system that can filter out relevant information from Twitter. By combining historical and real-time data with sentiment analysis, stakeholders can analyze events and opinions to understand their users and customers better.

1.2 Motivation

As an increasing number of people from all over the world connects to the Internet, a huge portion of those people finds their way to social media applications, such as Twitter[1], Facebook[2] and YouTube[3]. Twitter is a platform where people tend to express their feelings freely, making it an ideal source for a vast amount of opinions about a wide spectrum of topics [2]. By the end of 2017, the micro-blog application had 330 million monthly active users that post approximately 500 million status updates every day, all over the world[4]. These status updates, hereafter called *Tweets*, are short, concise and straight to the point and thus making the threshold for posting updates very low. The simple expression form might explain the popularity and enormous volume of Tweets. In addition to opinions, Twitter users also tend to comment on real-world events, both locally and globally, whenever something captures their attention. The Twitter users are functioning as first-hand witnesses, meaning that there is a chance that they will report about an event before traditional media does.

The amount of data that are being stored and produced every day is enormous and keeps growing. It is clear that this data is valuable and offers various possibilities. Despite that, the potential hidden in this data is still in many ways unexplored and not fully utilized. Due to the informal nature of Tweets and the way people choose to express themselves with humor and sarcasm, computers have difficulties with interpreting and understanding the semantics of the Tweets. Another challenge with Twitter data is due to its volume. As Twitter users are continuously posting statuses about events and opinions, Twitter contains lots of interesting information that could help companies and other stakeholders to facilitate decision making. However, as it is almost impossible to interpret all incoming Tweets by the eye, there is a need to summarize them. One way to solve this is to use visualization. The goal with visualization tools is to allow users to quickly and intuitively understand the content, context, and the organization of the incoming data stream. By overcoming the grammatical challenges with Twitter and by visualizing the relevant content, we will be able to explore

[1]Twitter: `https://twitter.com`

[2]Facebook: `https://www.facebook.com`

[3]YouTube: `https://www.youtube.com`

[4]`https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/`

the information hidden in social media.

1.3 Research Goals

This section will present the goal of the project and the associated research questions.

The specialization project presented a solution that utilized a suitable visualization tool together with a Big Data platform to visualize streaming data. This thesis will continue the development of the proposed system with a focus on making the output as relevant as possible for the user. The goal is defined as:

To examine how a system can be implemented, which can visualize relevant information from Twitter based on user-defined queries and provide valuable insights for the user.

Various aspects of the system need to be considered to fulfill the described project goal. The primary motivation of this project is to explore the information hidden in social media, by extracting relevant Tweets from Twitter. Hence, there is a need to examine how Tweets can be detected and classified as relevant in real-time. The project should also look into how relevance should be defined with respect to the stated project goal.

The research question for the project can be defined as follows:

RQ 1: How can Tweets be detected in real-time based on a user-defined continuous query?

> This research question is one of the most important questions because it will answer how the main parts of the system should be implemented. The different parts include how the data should be extracted from Twitter, how it can be filtered and classified as relevant, and lastly how it can be presented to the user.

RQ 2: How can a Tweet be classified as relevant relative to a query?

> This question should be considered as a supplement for **RQ 1** as it will focus on examining how the data can be filtered and classified as relevant. To answer this question, the definition of relevance for this specific case has to be considered.

RQ 3: How can Tweets be ranked based on relevance?

> Once the Tweets has been filtered, there is a need to rank them based on how relevant they are to a specific query. It is important that the system can update the ranking list continuously whenever new matches arrive. Since the motivation behind this project is to allow stakeholders interpret Twitter data and draw conclusions about their users, it is vital that the ranking will be based on both new, streaming data and historical, stored data.

Furthermore, it is important that the system is implemented with respect to the characteristics of large volumes of streaming data and different aspects regarding performance should be examined.

RQ 4: How does the implemented system perform with increasing amount of queries?

> The system will be filtering out Tweets based on queries from users. This research question will answer how performance is affected when the number of queries is increasing.

RQ 5: How does the implemented system perform with increasing amount of data?

> Two characteristics of Twitter data are velocity and volume. This question will investigate how the implemented system manages to handle a significant amount of data arriving at high speed.

1.4 Research Strategy

This section will present the research process and methods used in the different stages for approaching the stated research goal. The overall research process will first be presented before the phases of literature review, development and experiments are explained in further details.

1.4.1 Overall Research Process

An overview of the research process can be found in Figure 1.1. The figure was retrieved from the book *Researching Information Systems and Computing* [3], and the elements that were found most relevant for this project are marked in red. The adapted figure can be found in Figure 1.2.

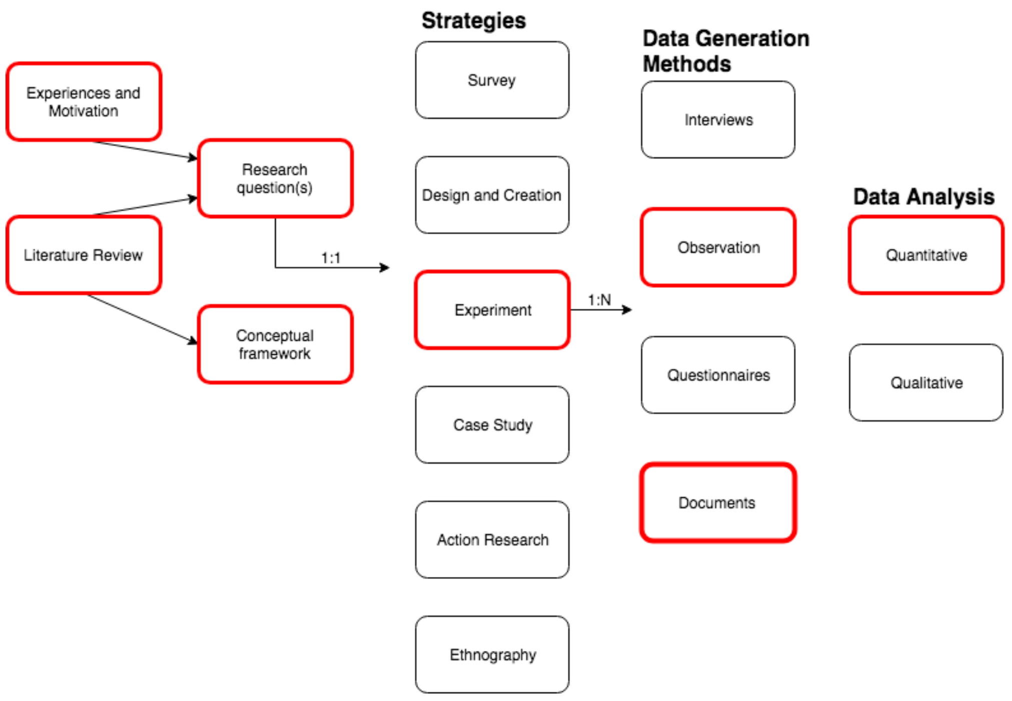


Figure 1.1: Research Process found in [3]

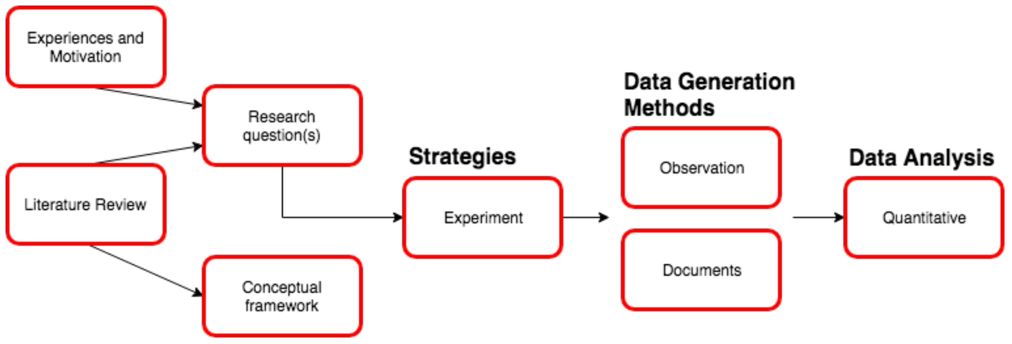


Figure 1.2: Adapted Research Process

The steps of personal experiences, motivation, and literature review were necessary for determining the project goal and associated research questions. The approach used for determining the goal and the research questions was based on the Goal/Question/Metrics paradigm defined by Basili [4]. The paradigm is used as a mechanism for defining and interpreting software measurement along with templates for defining goals and generating research questions. The approach starts by defining the project goal by including *purpose*, *perspective*, and *environmental* characteristics. Furthermore, a set of research questions are defined to supplement the project goal. The literature review did also provide a conceptual framework for the project to be done and specified the research methodology including strategy and data generation methods.

The chosen research strategy for this project is *Experiment.* The reason for this strategy is mainly due to the stated project goal which is to examine how a system can visualize relevant information from Twitter based on user queries. According to [3], the Experiment strategy includes testing hypotheses and seeking to prove or disprove a causal link between a factor and an observed outcome. When using this strategy, it is crucial that all factors that might affect the results be carefully excluded from the study. The strategy also focuses on investigating cause and effect of relationships. The project will utilize a **method triangulation** by combining Observation and Documents as methods used for Data Generation. The use of triangulation will allow us to look at the results in different perspectives, where observations include observing how the system behaves based on different variables, and documents include the visuals to be produced in this project.

The last step of the research process is *Data analysis*, where a Quantitative approach will be used. Quantitative data analysis uses mathematical approaches such as statistics to examine and interpret the data. The data analysis will mainly be based on evaluating the system to see if the proposed functions behave as intended and meet the requirements. However, some *Qualitative* analysis will also be performed, as the goal of the project is to create visualizations that also will be evaluated.

1.4.2 Literature Review

A preliminary study started with a literature review of existing systems and research of Big Data and streaming data. Since the scope of the project is rather broad, a research method called *Snowballing Sampling* or *Chain-referral Sampling* has been used [5]. The research method is mentioned by Lewis-Beck et al. in the SAGE Encyclopedia of Social Science Research Methods. However, it is possible to apply the technique to other research fields as well. The Snowballing method is a process where one starts with a few numbers of papers, and by looking at the relevant work mentioned in those papers, it is possible to extend the reading list. As one can imagine, the method tends to generate large numbers of possibly relevant articles, meaning that the researcher must be able to determine which paper is worth using. For this specific project, materials that could help reach the defined goal were chosen. The search was executed using Google Scholar[5]

[5]Google Scholar: `https://scholar.google.no/`

and the NTNU University Library[6]. The chosen articles were prioritized based on the number of citations, as well as known publishers and authors. The publication date was also a priority as this is a research field that is moving quickly and, therefore, recently introduced research provides the latest discoveries.

For the specialization project, the chosen method resulted in relevant sources regarding Big Data Management Systems and streaming data. As the project proceeded, a need for performing Sentiment Analysis with some Machine Learning technique emerged, and articles regarding those topics were reviewed as well. For this thesis, the focus shifted to examine literature about classification of relevant information. Various ranking and filtering techniques were also examined.

1.4.3 Development

The development of the final system has been done using an *Incremental and Iterative Development* methodology [6]. The methodology is based on the statement that incremental development is distinctly different from iterative development and that they should be treated differently but at the same time be used together. The definition of incremental development says that it is a staging and scheduling strategy where the work is divided into smaller parts which are then developed at different times and integrated as they are completed. Iterative development is a rework scheduling strategy where the system is revised and improved repeatedly. By taking advantage of the incremental approach, one will get the opportunity to refine the development process and adjust the requirements as the project proceeds. The iterative approach, on the other hand, will enhance the product quality, even if it probably will imply rework.

A total of three iterations were used to implement the final system. By using an agile methodology, each iteration was treated like a sprint. The first sprint was focusing on the requirements of the system and building a simple prototype including the main parts of the system. The second sprint was an initial implementation of the prototype and was focusing on the task of filtering the incoming Tweets. Lastly, the ranking function was defined, and the final system was implemented after evaluating the result of the second sprint.

To adapt the incremental methodology, the work to be conducted were divided into smaller tasks. A type of agile methodology similar to the Kanban method was used to keep track of the tasks [7]. The various tasks

[6]NTNU University Library: `https://www.ntnu.edu/ub`

were tracked by using a digital post-it board. The categories on the board were "to do", "doing", "done". The tasks were formulated based on the goal and research questions defined in section 1.3.

1.4.4 Experiments

Some experiments had to be performed to evaluate the proposed solution. A total of three experiments were conducted in addition to one final execution of the whole system. The experiments were defined concerning the stated project goal and research questions and the goal was to test various aspects of performance in addition to ranking based on relevance.

1.5 Contributions

The thesis introduces a system for visualization of relevant content in the Twitter stream based on a user-defined continuous query. The implemented system is an attempt to test the various theories regarding the subject of the project. It aims to help companies and other stakeholders to understand the Twitter users which will lead to useful business insights that can help them make decisions. The user of the system will be able to subscribe for themes of interest and get updates whenever new information is available. However, the system will be implemented as modular as possible which will make it usable for other applications and data sources as well.

To provide exciting content for the visualization tool, the thesis will also explore sentiment analysis and ranking techniques.

1.6 Scope and Limitations

The parts in this Master's thesis shall be developed, implemented and written during the spring semester of 2018. The official time limit for the project is 21 weeks, and the submission deadline is the 13th of June. All work regarding this thesis will be done by one person, Phrida Norrhall. It should be emphasized that the final solution should not be considered as a system ready for production but regarded as an attempt to prove the underlying theories. Furthermore, the project has not been focusing on the accuracy of either the sentiment analysis or the proposed filter and ranking functions, and the output is only used as a *proof of concept.*

The data that will be used is retrieved from Twitter through the Twitter Streaming API[7], which returns a sample of public statuses that match one or more filter predicates [8]. The API has recently changed the name to Filter real-time Tweets[8], but for simplicity, it will still be referred as the Twitter Streaming API throughout this thesis. Due to resource limitations, retrieving streaming data and the visualization of it will be performed on a single computer.

1.7 Thesis Structure

The remainder of the thesis is structured as follows:

Chapter 2 - Background Theory: Provides the relevant background theory needed for understanding the work to be conducted.

Chapter 3 - Related Work: An overview of related work and research regarding topics that concerns the subject of the project.

Chapter 4 - Ranking of Results from Continuous Queries: A presentation of the implemented solution and decisions made during the process.

Chapter 5 - Experiments and Results: Presents the conducted experiments and its associated results.

Chapter 6 - Discussion: Gives an evaluation and discussion of the results and the project as a whole.

Chapter 7 - Conclusion: Provides a conclusion and an introduction to future work.

[7]`https://developer.twitter.com`

[8]`https://developer.twitter.com/en/docs/tweets/filter-realtime/overview`

Chapter 2

Background Theory

This chapter describes the background theory needed for understanding the foundation of the work to be conducted in this project. The topics here are not explained in detail, and only the parts relevant to this project will be covered. The chapter starts by describing the concepts of Big Data, Streaming Data, and Social Media Analysis. Some metrics for measuring relevance will be presented, before describing the chosen technologies used.

2.1 Big Data

Big Data is a term for data sets that are so large and complex that original Database Management Systems (DBMS) cannot handle them. There exist many definitions of Big Data where Gartner defines one of the most acknowledged definitions [9]. The data management challenges are defined as three dimensions: **Volume**, **Velocity** and **Variety**. These dimensions concern the amount of data that is being produced and stored, the speed at which it is generated in addition to its variated structure.

In the world of Big Data there exists a distinction between *Batch Processing* and *Stream Processing*. In a batch processing model, the data is being collected in batches before it is being sent to the system for processing. The stream processing model, on the other hand, feeds data into the system piece-by-piece and the processing is usually done in real-time.

As data sizes have outpaced the capabilities of single machines, users have needed new systems to scale out computations to multiple nodes. As a result, there has been an explosion of different systems that are targeting diverse computing workloads. Unfortunately, most big data applications need to combine many different processing types. Specialized engines may, therefore, cause both complexity and inefficiency when users must create their own engine by combining disparate systems [10]. Some developers have tried to avoid this problem by building unified engines, and one example of such engine will be presented later in this chapter.

2.2 Streaming Data

Big Data that are being generated at a steady high-speed rate can be defined as *Streaming Data*. Streaming data differs from traditional structured, stored data. Unstructured, or semi-structured data, such as Twitter messages are being generated in volumes that grows every day. A stream of data can be described as a time-stamped, temporally ordered collection of unstructured or semi-structured data. The data elements in this stream arrive online at high speed, and the system has no control over the arrival order, either within a data stream or across streams. Once an element from a data stream has been processed, it is discarded or archived which makes it hard to be retrieved again, meaning that the arriving data must be treated immediately or it will be lost [11]. For that reason, streaming data requires new models and algorithms than the ones that have been used for structured data.

Along with the explosion of applications involving streaming data, there has been a shift of focus in the industry. The most important thing is no longer how big your data is. It is far more important how fast you can analyze it and gain insights. One central aspect of streaming data is therefore **time**. The incoming data must somehow be partitioned to have any value. A standard way to do this is to divide the stream into fragments called *windows*. There exist various types of windows, and the type used for this project is called *sliding windows*. Sliding windows have a fixed length and are separated by a time interval t. The stream processor can then keep the recent n records of the stream that has arrived within the last t time units. The records are being held in working space, and those that are too old will be dropped.

There are two main categories of queries for streaming systems: *continuous queries* and *ad-hoc queries* [12]. The first represents queries that are asked for every incoming record in the system, and it must be defined a priori. Examples of continuous queries are "What is the maximum value seen so far?" or "Send a notification when a Tweet about Basketball is posted". The other category represents queries that are asked once and run to completion over the current state of the stream. An ad-hoc query is created to obtain information as the need arises and one example might be "How many users have tweeted about Basketball the last week?". This type of query requires that the stream processor remember the state of previously seen records. So for the stated example, it needs to keep track of the last week's tweets.

2.2.1 The Streaming ETL Process

Data ingestion is the process of getting data from its source to its final destination as efficiently and correctly as possible. The data ingestion process has often been discussed under the name of Extract, Transform and Load (ETL). The ETL concept first became popular in the 1970s and is a process that has typically been used in data warehousing. As a computational need for streaming data has emerged, the concept of ETL has evolved with it. However, the main idea behind the three steps in the process is yet still the same. *Data Extraction* is the process where data is extracted from different data sources. These sources can both be streaming and static. *Data Transformation* is where the retrieved information is transformed into the proper format needed for storing it. Lastly, *Data Loading* takes the processed data and loads it into the final data storage system.

Traditionally, ETL was constructed as a pipeline of batch processes, where each process took its input from one file and wrote its output to another. The next process in the pipeline could then consume the output. Older applications, such as data warehouses, were not dependent on receiving the most current data nor were they sensitive to the latency introduced by the process of reading and writing of files. Newer applications, on the other hand, such as real-time analytics applications need to provide a more accurate model of the real world to accommodate real-time decision making. Hence, it is dependent on receiving the most current data while at the same time avoid latencies. With the evolution of modern applications, the ETL process can now be seen as a streaming problem [13].

2.3 Social Media Analysis

People express opinions and feelings through messages and statuses in social media, and analysis of this can be done for many reasons. One of the reasons is to classify conversation topics to find trending topics that are being discussed. Another reason is to analyze the sentiment of people to a subject. Sentiment can be defined as "A thought, view, or attitude, especially one based mainly on emotion instead of reason"[2]. By performing this analysis, stakeholders such as product companies can see ratings of their products or services from the perspective of their users.

2.3.1 Sentiment Analysis

Sentiment Analysis (SA) refers to the use of Natural Language Processing (NLP and Text Analysis to extract, identify and characterize the sentiment content of a text unit [14]. SA can be used for a great variety of occasions and applications, for example, to predict the success of a political campaign or a product launch, or to decide whether to invest in a particular company or not. Other use cases are targeted advertising and reviews of products and services. Twitter is a platform that contains a lot of sentiment information, and it is often used when performing SA.

2.3.2 Machine Learning

One way to perform sentiment analysis is to use Machine Learning (ML) [15]. ML is the study and construction of algorithms that can learn from data and make data-driven predictions. ML can be *supervised* or *unsupervised*. The supervised approach means that the training data consist of a set of training examples that have been labeled. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new items. Examples of problems that are solved with supervised learning are classification and regression. The unsupervised learning approach, on the other hand, does not require any human intervention and is instead used to describe hidden structures in unlabeled data. Examples of problems that are solved with unsupervised learning are clustering and association problems.

2.3.3 Naive Bayes

Due to the nature of microblogs, with its limited message size and the wide range of topics that are being discussed, sentiment extraction is indeed a challenging process. However, the most common way to solve it has been to use long-known machine learning algorithms, which turns sentiment extraction into a classification problem [16]. Naive Bayes is a multiclass classification algorithm that can be used for supervised learning [17]. The algorithm uses datasets containing classified Tweets to train the classifier. Naive Bayes is not a single algorithm, but a family of probabilistic classifiers of supervised learning algorithms. The algorithms are based on Bayes' theorem and the "naive" assumption that there is independence between every pair of features. Naive Bayes can be trained very efficiently. With only one single pass to the training data, it can compute the conditional probability distribution of each feature. The algorithm then applies Bayes' theorem to compute the conditional probability distribution of a label, and use it for prediction. Bayes' theorem in 2.1 describes the probability of an event A, based on conditions that might be related to the event. Both A and B are events. $P(A|B)$ is a conditional probability and the likelihood of event A occurring given that B is true. $P(B|A)$ is similar but is instead the likelihood of event B occurring given that A is true. $P(A)$ and $P(B)$ are the probabilities of observing A and B independently of each other.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

There are different versions of Naive Bayes, multinomial naive Bayes and Bernoulli naive Bayes. Both of these versions can be used for document classification. Within the context of this project, each observation will be a Tweet, and each feature represents a term whose value is the frequency of the term (in multinomial naive Bayes) or a 0 or 1 indicating whether the term was present in the document or not (in Bernoulli naive Bayes).

2.4 Visualization

Due to the evolvement of real-time applications, stakeholders need to get insights into data streams immediately to facilitate decision making. One way to solve this is to visualize these applications. Data visualization is a visual depiction of information, meaning that it is used to present or communicate information in an imagery way often in the form of dashboards. However, visualization of real-time data streams presents many interesting challenges due to the data being dynamic, transient, high-volume and temporal. The visualizations need to be able to handle abstract and dynamic data behavior and to present the data in ways that make sense to and are usable by humans [18]. Compared to static data, dynamic data has some unique characteristics that make it more challenging to analyze and visualize. For example, with a continuous data stream, the total amount of data that are being analyzed can be massive in size. In cases where the data rate is exceptionally high, stakeholders may struggle in examining the data and interpret it in real time. Visualization tools should, therefore, be able to allow users to quickly and intuitively understand the content, context, and organization of the incoming data stream. It is vital that these tools connect to the user's mental model of the problem in addition to the data and its changing behavior. Before the development of today's existing Business Intelligence (BI) tools, understanding raw data was a multi-step process. Executives had to present their questions to a database expert who then could execute a query against the database or data warehouse. The result was then given to another person who could write the necessary code to represent it as a dashboard or a readable document. At this point, the data presented was out of date and not in real-time any more, which often made it irrelevant. With today's BI tools, companies can bypass many unnecessary steps and avoid bottlenecks [19].

2.5 Metrics for Measuring Relevance

An important aspect of information retrieval and full-text search is *relevance* and more precisely, the ability to rank results by how relevant they are to a given query. This section will present characteristics and measurements that should be considered when estimating relevance. Furthermore, some known scoring functions will be presented.

2.5.1 Precision and Recall

Two measurements is important when discussing relevance, *Precision* and *Recall*. The calculation on these depends on four different values called *true-positives* (*tp*), *false-positives* (fp), *true-negatives* (*tn*), and *false-negatives* (fn). Where *tp* and *tn* are the numbers of records that has been correctly classified as positive or negative, while fp and fn are the numbers of records that has been falsely classified as positive or negative. The relationship between these values can be seen in the confusion matrix shown in Table 2.1.

		Predicted	
		Positive	Negative
Actual	Positive	*tp*	fn
	Negative	fp	*tn*

Table 2.1: Confusion matrix for possible prediction outcomes

Precision is the number of relevant documents retrieved divided by the total number of retrieved documents and can be seen as a measure of exactness or quality:

$$\textbf{Precision} = \frac{\text{relevant documents retrieved}}{\text{retrieved documents}} \tag{2.2}$$

By using the values presented in Table 2.1, precision can be calculated as follows:

$$\textbf{Precision} = \frac{tp}{tp + fp} \tag{2.3}$$

Recall, on the other hand, is the number of relevant documents retrieved divided by the total number of all relevant documents, and can be seen as a measure of completeness or quantity:

$$\textbf{Recall} = \frac{\text{relevant documents retrieved}}{\text{total relevant documents}} \tag{2.4}$$

Or by using the possible prediction outcomes:

$$\textbf{Recall} = \frac{tp}{tp + fn} \tag{2.5}$$

In other words, high precision means that a search returned substantially more relevant results than irrelevant ones, thus high quality, while high recall means that a search returned most of the relevant results, thus high quantity. Optimal recall and precision are achieved when the search result returns all and only the relevant documents in the corpus.

2.5.2 Scoring Functions

Term Frequency - Inverse Document Frequency

The Term Frequency - Inverse Document Frequency (TF-IDF) is a numerical statistic that provides information about word importance [20]. Given a collection of documents that make up a corpus, TF-IDF assigns scores to words in each document based on how important the word is in describing the document.

The score is a product of two distinct features:

Term frequency (TF): The number of times a term appears in one field. The more often it appears, the more likely it is to be relevant.

Inverse document frequency (IDF): The number of times each term appears in all documents. The more often it appears, the less likely it is to be relevant. This means that terms that appear in many documents will have a lower weight than terms that are more unique.

Following equations shows how the TF-IDF weight of a term within a document is calculated, where t is a term appearing in document d:

$$TF - IDF(t,d) = TF(t,d) \cdot IDF(t) \tag{2.6}$$

$$IDF(t) = log\frac{N}{DF(t)} \tag{2.7}$$

By applying this scoring function, a word that often appears in a small number of documents will get a high weight and a word that occurs either few times within a document or many times across all documents in the corpus will get a lower weight. Common words such as 'the', 'is', and 'it' will, therefore, get low weights.

Vector Space Model

The Vector Space Model (VSM) is an algebraic model for representing text documents as vectors of terms. VSM is often used for comparing a multiterm query against documents to rank the documents based on their similarity to the query. The output of the model is a single score that represents how well the document matches the given query. The score is calculated by comparing the deviation of angles between each document vector and the original query vector [21].

To calculate the similarity between documents and the query, the *cosine similarity* measure can be used. Given two vectors of attributes, A and B, the cosine similarity, $\cos(\theta)$, is defined as follows:

$$similarity = cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.8}$$

2.6 Technologies

This section will present the technologies used in this project. Every choice of technology has been made with regards to the project goal and its fulfillment. Hence, technologies and frameworks that can handle transformation and indexing of streaming data, machine learning, and visualization of data streams has been chosen.

2.6.1 Apache Spark

Apache Spark is a unified analytics engine for large-scale processing of data. It has a programming model that is similar to MapReduce [22] but has extended it with a data-sharing abstraction called *Resilient Distributed Datasets*, or RDDs [10]. RDDs are fault-tolerant collections of objects distributed across a cluster that can be manipulated in parallel. To create RDDs, users have to apply operations that are called *transformations* to their data. Examples of these are `map, filter` and `groupBy`. Spark exposes RDDs through Application Programming Interfaces (APIs) in Java, Scala, R and Python, which makes it possible for users to pass local functions to run on the cluster. Because of this unified API, applications are easy to develop.

Spark lazily evaluates RDDs, meaning that the user has to call for another operation type called *actions* for Spark to start. This behavior allows Spark to find an efficient plan for the user's computation. Examples of *actions* are `count, collect` and `save`. When an action is called, Spark looks at the whole graph of transformations that the user has called to create an execution plan.

By implementing the extension of RDDs, Spark can take care of different kinds of processing workloads that previously needed separate engines. Examples of these processing types are SQL, streaming, machine learning, and graph processing. So where prior systems required writing data to storage and then passing it to another engine, Spark can run diverse functions over the same data, often in memory. However, one should notice that Spark is strictly a processing engine, meaning that it does not persist data for use outside of the current execution. As soon as the engine has performed the whole execution graph, one must send it to a separate system for offline storage.

Spark Streaming

Spark Streaming is one of the most popular streaming processing engines. It allows processing of a stream of data on a Spark cluster. It builds on the general Spark execution engine, but instead of having RDDs as its central abstraction it has *discretized streams* called DStreams. The main idea behind DStreams is to treat computation of a data stream as "a series of deterministic batch computations on small time intervals" [23]. Hence, a DStream is defined by a time interval, which is used to pre-group the incoming stream elements into discrete chunks. These chunks forms RDDs which can be processed by the Spark execution engine, as can be seen in Figure 2.1. This way of handling streaming data is defined as micro-batching.

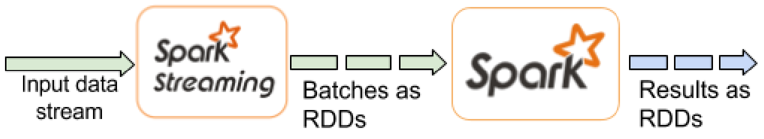


Figure 2.1: Spark Streaming

The motivation behind Spark Streaming was to have a system that provides consistency, efficient fault recovery, and robust integration with batch systems. Thus by using DStreams, Spark Streaming can treat the incoming stream as a series of short batch jobs which will bring down the latency. Consequently, Spark Streaming is a streaming processing engine that has many of the benefits of batch processing models in addition to the ones for stream processing. So even if Spark Streaming is not defined as "true" streaming where events flow into the system at a given rate, it is still possible to implement many streaming operations on top of this architecture. A DStream group together a series of RDDs and lets the user manipulate them through both *stateless* and *stateful* operators. One example of the former is `map` which act independently on each time interval, while an example of the latter is aggregation over a sliding window, which operates on multiple time intervals and may produce new RDDs as intermediate states.

Spark Machine Learning Library

Spark has its own library for machine learning functions called MLlib[1]. MLlib is a distributed machine learning framework that runs on top of the Spark core. The library contains a variety of learning algorithms and is accessible from all Spark's programming languages. It consists of standard learning algorithms and features, which includes classification, regression, clustering, and collaborative filtering, where classification will be in focus for this project.

[1]MLlib: `https://spark.apache.org/mllib/`

2.6.2 Apache Kafka

Apache Kafka[2] is an open-source stream-processing software platform that was initially developed at LinkedIn[3]. The project started as an answer to the problems regarding building real-time streaming applications and the piping of data between systems. The development at LinkedIn was at first very ad hoc, meaning that they built piping between systems and applications as it was needed. Over time this became increasingly more complex, with data pipelines between all kinds of systems. However, the problem was not only about data transportation. The developers also wanted to be able to transform the data. With the rise of processing platforms such as Hadoop, there was a need for getting the data from the users in an asynchronous way with low-latency. Consequently, LinkedIn started on building a system that would focus on modeling streams of data [24]. The project is now part of the Apache Software Foundation and aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka allows users to publish and subscribe to streams of records, similar to a message queue.

Kafka has some capabilities that should be explained to understand how it works. Firstly, Kafka is run as a cluster on one or more servers that can be distributed to multiple data centers. The streams of records are stored in the Kafka cluster in categories called *topics*, where each record consists of a key, a value, and a timestamp. Topics are multi-subscriber, meaning that a topic can have zero, one, or many consumers that are subscribing to the data that are being written to it. For each topic, the Kafka cluster maintains a partitioned *log* as can be seen in Figure 2.2.

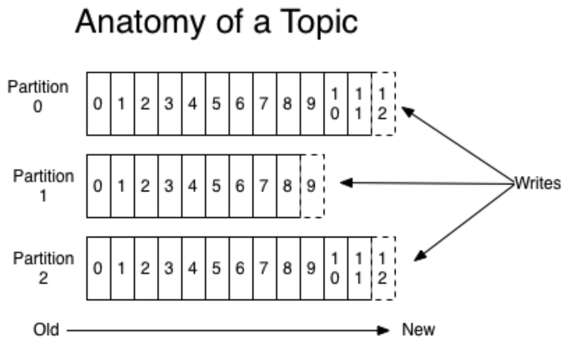


Figure 2.2: Anatomy of a topic in Kafka [25]

Each *partition* is an ordered, immutable sequence of records that is ap-

[2]Apache Kafka: `https://kafka.apache.org/`
[3]LinkedIn: `https://www.linkedin.com/`

pended to the log continuously. The records in the partitions have a sequential id number that is called the *offset* which is used to identify each record within a partition uniquely. Furthermore, Kafka has four core APIs which allows applications to integrate with the system. The API used for this project is called the `Producer API`, and it allows an application to publish a stream of records to one or more Kafka topics [25].

A traditional enterprise messaging system typically has two models - *queuing* and *publish-subscribe*. Where in a queue, a pool of consumers can read from a server and each record are sent to one of them, and in publish-subscribe, the record is broadcasted to all consumers. Each of the two models has a strength and a weakness. Publish-subscribe allows the user to broadcast data to multiple processes, but it is not able to scale processing since every message goes to every subscriber. Queuing, on the other hand, allows the user to scale the processing by dividing processing of data over multiple consumer instances. However, queues are not able to handle multiple subscribers, once one process reads the data it will be gone. Luckily, Kafka generalizes these two concepts and allows the users to both divide processing over a collection of processes while also allowing broadcasting of messages to multiple consumer groups. As every topic in Kafka has these properties, there is no need to choose one or the other.

Kafka gives some guarantees that are useful when retrieving data streams. Messages sent by a producer to a specific topic will be appended in the order they are sent. At the same time, a consumer instance will see records in the order as they are stored in the log. Another advantage is that all published records will be persisted in the Kafka cluster, even if they have been consumed or not. Kafka is using a configurable retention period, meaning that a retention policy can be set to an optional amount of time. A record will then be available for consumption during the retention period before it is being discarded to free up space. Also, as Kafka's performance is effectively constant with respect to data size, there is no problem to store data for a longer time.

2.6.3 The Elastic Stack

The following section will present parts of the Elastic stack that has been used in this project. The components in the stack can reliably and securely take data from any source, in any format and allow full-text searches, analysis, and visualizations of it in near real time.

Elasticsearch

To analyze and process information from real-time applications that contain semistructured or unstructured data there is a need for an analytics engine that can handle voluminous data and full-text searches. Elasticsearch[4] is a highly scalable open-source full-text search and analytics engine that is built upon Apache Lucene[5], which is an open-source full-text search engine written in Java. Elasticsearch is often used as the underlying engine for applications that have complex search features and requirements and allows users to get near real-time data insights. Near real-time means that there is a slight latency from the time that the data is being indexed until the time it becomes searchable. However, the latency usually is just a second and Elasticsearch is, therefore, suitable for analytics applications. An example of a typical use case is applications where it is vital to quickly investigate, analyze, visualize, and ask ad-hoc questions on millions or billions of records. For these cases, it is common to use Elasticsearch to store the data and then use another part of the Elastic stack, called Kibana, to build custom dashboards that visualize different aspects of the data that are important for the user [26].

A few concepts should be presented to understand the core functionality of Elasticsearch. To utilize Elasticsearch one has to define a *cluster*. A cluster is a collection of one or more *nodes*, where each node is a single server, which together holds the entire data set. The cluster provides indexing and search capabilities across all nodes. An *index* is a collection of *documents* that have similar characteristics. A document is a basic unit of information that is being indexed, expressed in JavaScript Object Notation (JSON). An index is identified by its name and has to be set if the visualization tool presented in the next section is going to discover it. The name is also used when performing search, update, and delete operations against the documents in the index.

Since Elasticsearch is going to store a significant amount of data that possibly can exceed the hardware limits of a single node, it provides the ability to divide the defined index into smaller pieces called *shards*. The main advantages with sharding are that it allows the content volume to be split horizontally and it also allows operations to be distributed and parallelized across shards which will increase performance. As for all environments where failures can be expected at any time, it is essential to have a failover mechanism in case a shard or node goes offline or disappears. The mechanism in Elasticsearch allows the user to make copies of the index's shards into something called *replicas*. There are two primary reasons why repli-

[4]Elasticsearch: `https://www.elastic.co/products/elasticsearch`
[5]Apache Lucene: `https://lucene.apache.org`

cation is important. First of all, it provides high availability if something goes wrong with the shards or nodes. However, this is depending on that a replica is never allocated on the same node as the original shard. Secondly, replication allows the system to scale out the search volume and throughput since searches can be executed in parallel on all replicas [27].

Kibana

Kibana[6] is an analytics and visualization platform that is designed to work with Elasticsearch. After storing the data in Elasticsearch indices, Kibana allows the user to search, view and interact with it. Kibana has a simple, browser-based interface which will enable users to quickly create dynamic dashboards that will display changes to Elasticsearch in near real-time. An example of a valuable visualization in the context of Twitter could be exploring trending hashtags on Twitter right now, or for a specific time interval. Hence, Kibana allows users to perform advanced data analysis and visualizations in near real-time [28].

Figure 2.3 shows an overview of some of the different visualization types provided by Kibana.

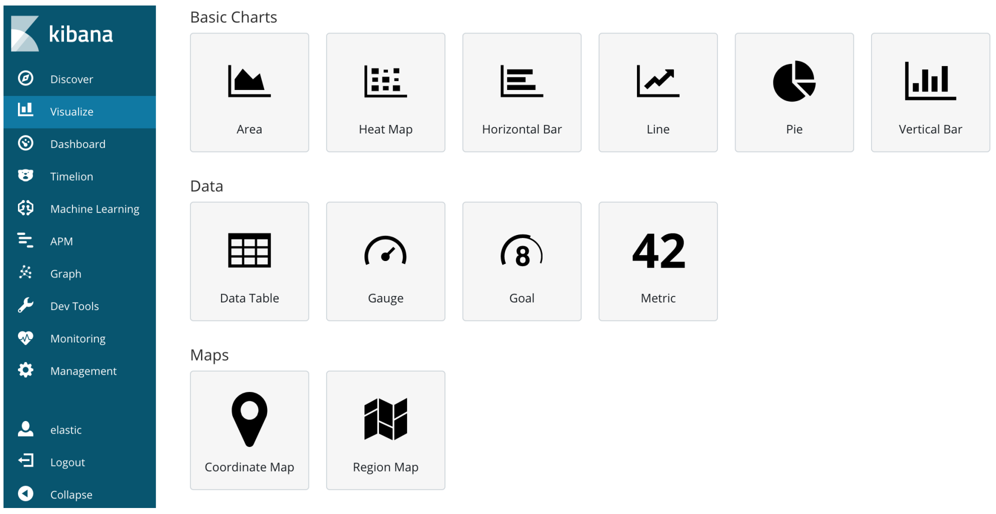


Figure 2.3: An overview of visualization options in Kibana

[6]Kibana: `https://www.elastic.co/products/kibana`

The Practical Scoring Function

As Elasticsearch is built on top of Lucene, it can take advantage of its full-text search capabilities. By default, Lucene is returning results in descending order of relevance where relevance is represented as a positive floating-point number called `_score`. This score is calculated based on various types of query clauses where different clauses are used for different purposes. Examples of query clauses are calculating how similar the spelling of a found word is to the original search term or the percentage of terms that were found. The most common example, however, is to calculate how similar the contents of a full-text field are to a full-text query string [29].

The standard similarity algorithm used in Lucene and Elasticsearch is TF-IDF. However, in addition to *term frequency* and *inverse document frequency*, Lucene has added another feature called *field-length normalization.* This factor is based on how long every field is. If an index over books contain the fields `title` and `content`, a term appearing in the short `title` field will most likely give more weight than the same term appearing in the longer `content` field.

These three factors are calculated and stored at index time. To calculate the `_score`, Lucene and Elasticsearch uses a formula called the *practical scoring function* together with the *Boolean model.* The Boolean model uses the `AND`, `OR`, and `NOT` conditions to find all the documents that match the query. An example of a boolean query can be seen in Listing 2.1. The query will return documents that include both President and America, and either Trump or Obama. The goal of this process is to exclude documents that cannot match the defined query before calculating the relevance score.

Listing 2.1: Example of a query for the Boolean model

```
President AND America AND (Trump OR Obama)
```

The practical scoring function borrows concepts from TF-IDF and the Vector Space Model but can also be combined with other features such as boosting a specific property that is important for the query clause [30]. To calculate the score, both the documents and the query are represented as *vectors* where each number in the vector is the weight of a term calculated by TF-IDF. It should be mentioned that TF-IDF is the default way of calculating the term weights. However, other similarity algorithms such as the Okapi-BM25 can also be used in Elasticsearch [31].

Boosting of Relevance Score

Even if TF-IDF and Okapi-BM25 are both great algorithms for finding documents that are textually similar to a submitted query, there are occasions where the *practical scoring function* needs to be augmented with other scoring heuristics. An example of an occasion where text similarity is not the most important factor is geo search. Imagine that a user is interested in finding a gas station. Ranking the relevance score of gas stations based on their textual similarity to the query would not be useful to the user. Instead, the ranking score should be based on how geographically close they are to the user. Examples of other factors that might be used to boost the relevance score are popularity and temporality [32].

Elasticsearch provides various features which allow the user to implement a customized ranking function based on the occasion. Two convenient features are called `script_score` and `function_score`. Imagine that we want to rank music videos on a website using a combination of textual relevance relative to a query and the popularity of the videos on the website. A total of three factors should be taken into considerations:

- `_score`: textual similarity of the video's metadata based on TF-IDF
- `likes`: The number of likes a given video has received
- `views`: The number of views a given video has received

A formula can be constructed to take all factors into consideration:

$$\texttt{\_score} * log(\texttt{likes} + \texttt{views} + 1)$$

The formula treats the sum of `likes` and `views` as a coefficient. The number 1 is added in case both of them are 0. The logarithm of the coefficient is then used to not let the most popular videos dominate all results. Listing 2.2 shows an example of how `function_score` and `script_score` has been utilized to implement the proposed formula and rank music videos based on the query `Justin Bieber`.

Listing 2.2: Query for ranking music videos

```
{
    "query": {
        "function_score": {
            "query": {
                "match": {
                    "message": "Justin Bieber"
                }
            },
            "script_score": {
                "script": "_score * log(doc['likes'].value + doc['views'].value + 1)"
            }
        }
    }
}
```

Another use case where functional scoring techniques are useful is for showing relevance based on trends. Such relevance score cannot be based on simple metrics such as `likes` or `views` but must continuously be updated based on the current time. A video that receives 1000 views in 1 hour should be considered more trendy and thus relevant than a video that receives 10000 views in 24 hours. Elasticsearch provides various decay functions that allow the user to weight more recent documents higher. The function can be based on three different curves, linear, Gaussian, and exponential distribution. Each document is then mapped to a point on the chosen curve to determine its score. An illustration of the different curves can be seen in Figure 2.4. The linear distribution has a decay where all points in time decay evenly. The Gaussian distribution decays slowly for values near the origin before falling distinctly and finally tapering off for low enough values. Lastly, the exponential distribution has the shape of a parabola, the farther away from the origin, the faster the values are decayed.

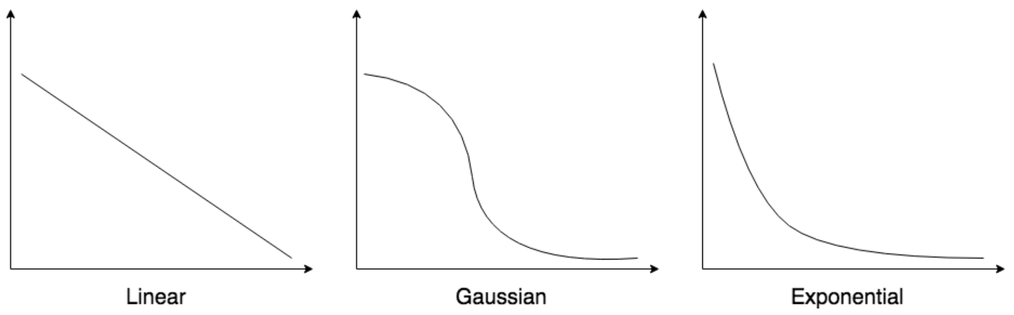


Figure 2.4: Linear, Gaussian and Exponential curves

Chapter 3

Related Work

This chapter presents a selection of research related to this project. The first parts are focused on related frameworks, research related to stream processing and visualization of data streams. Finally, some research related to classification and ranking of microblogs will be presented.

3.1 Related Frameworks to Handle Big Data

As mentioned earlier, Big Data can be defined by the three V's - Volume, Velocity, and Variety. Research has advanced regarding the *volume* of Big Data, while research about *velocity* of data is still an issue that is becoming more urgent to solve. Data arrives at high speed, and analytics of this information is often time sensitive. In 2013 Twitter realized that the analysis platform **Hadoop**[1] did not meet their latency requirements as they had hoped [33]. Twitter had tried to build a *real-time related query suggestion and spelling correction service* with strict latency requirements. This service was supposed to suggest trending queries to users in real-time, and Twitter wanted it to do so within 10 minutes after a major news event breaks. The first implementation was built on a typical Hadoop-based analytics stack. However, since Hadoop is designed to handle petabyte-scale datasets through large batch jobs, it was not suitable for the strict latency requirements that Twitter had for the application. This discovery points out that there is a need for data analytics platforms that can handle "big" as well as "fast" data.

To handle high velocity, it must have been inherited into the system design from the early beginning, and this is not the case in many of today's systems

[1]Hadoop: `https://hadoop.apache.org/`

[34]. Especially traditional Database Management Systems (DBMSs) are not able to support microblogs as they expect all data to be managed as persistent data sets and not data with high arrival rate. Because of this significant limitation in DBMS, several Data Stream Management Systems (DSMSs) has been introduced. A challenge when working with queries for microblog data is that they often require indexing of both real-time and historical data. So even if DSMS is better at handling data streams with high arrival rates, the systems are limited to support the concept of *continuous queries*. Continuous queries are queries that are issued once and then logically run continuously over the data stream [12]. This gives incremental answers for queries that are registered a priori. However, this is somehow different from the needs of microblog queries where the users are mostly asking about data that has already arrived at the system. The functionality of indexing and querying historical data is something that most DSMS lack.

To handle indexing of historical data, high arrival rates, and semistructured data, the development of Big Data Management Systems (BDMSs) has become a trend. The developers behind **AsterixDB**[2] has defined some design decisions as they believed were important when designing a BDMS [35]. They wanted the system to have a flexible, semistructured data model and a full query language with at least the power of Structured Query Language (SQL), and the ability to support a variety of data types and query types. The system should be able to do automatic indexing, continuous data ingestion, and scale gracefully to manage and query large volumes of data.

Different approaches have been proposed to solve the problem of combining real-time and historical data. One example is the Big Active Data (BAD) system from the University of California [36]. The BAD system is an extension of AsterixDB, and the goal is to create a scalable system that will act as a platform for what they call *active* data. The platform is designed to deliver data of interest to a large number of users while still supporting analyses of historical information, without compromising the relationships between data items. One can think of BAD as a combination of BDMS and DSMS. It will leverage from BDMS because of AsterixDB and its functionality, but it also includes concepts from DSMS for the real-time data analysis. BAD strives for a *Publish/Subscribe*-pattern where it collects data from a large number of sources and then publish it to an even higher number of subscribers that are interested in both incoming data as well as the changing of state for historical data. Figure 3.1 summarizes how the BAD system fits into the overall active systems platform space.

[2]AsterixDB: `https://asterixdb.apache.org`

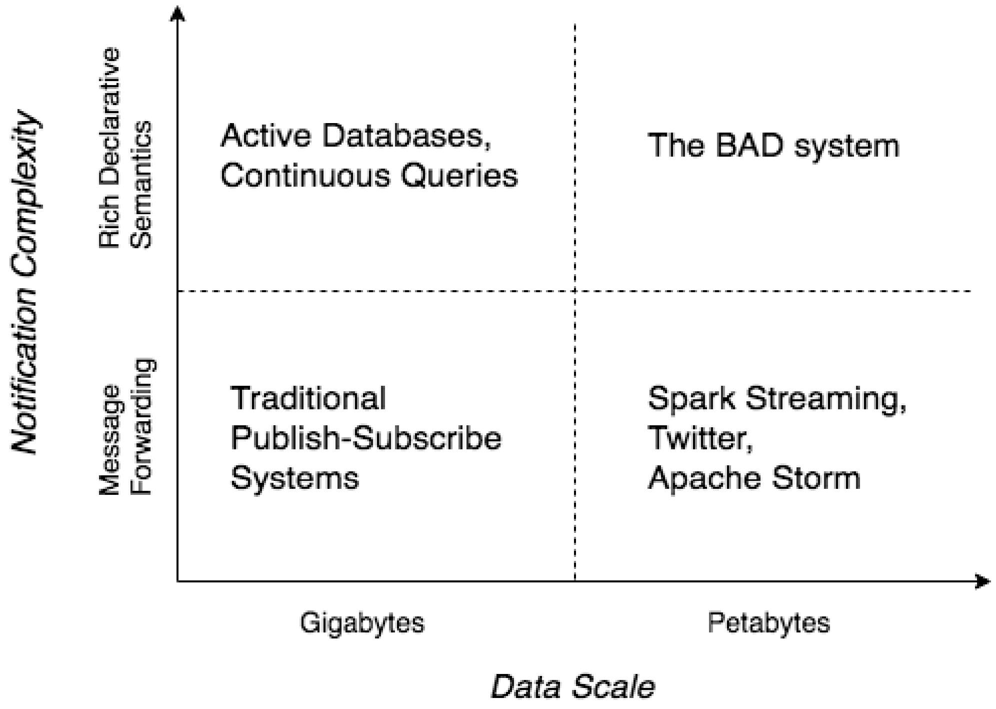

Notification Complexity
Rich Declarative Semantics
Message Forwarding
Active Databases, Continuous Queries
The BAD system
Traditional Publish-Subscribe Systems
Spark Streaming, Twitter, Apache Storm
Gigabytes
Petabytes
Data Scale

Figure 3.1: The BAD system in context of other systems [37]

As previously mentioned, one of the leading challenges with today's existing BDMSs is that they often are specialized in different processing types. This means that users must create their own engine by combining different systems to achieve the desired result. There have been many interesting attempts to benefit from both DSMSs and BDMSs. One example of a system that has been "glued" together are **Apache Storm**[3], a popular streaming engine, and **MongoDB**[4], a successful persistence store. In an article written by Grover and Carey, this combination is evaluated against AsterixDB [38], which is considered to be a unified engine. The authors argue that the glued solution requires that the end user understand the layout of the cluster and can include specific information in the source code. The authors conclude that AsterixDB outperforms the glued solution when it comes to both performance and user experience.

[3]Apache Storm: `http://storm.apache.org`
[4]MongoDB: `https://www.mongodb.com`

3.2 Related Methods to Handle Streaming Data

Many existing systems handling streaming data origins from the *MapReduce* programming model. The model was designed as an abstraction that allows users to express simple computations without having to take count of the details of parallelization, fault tolerance, data distribution and load balancing [22]. It was inspired by the *map* and *reduce* primitives that are present in many functional programming languages. The map function takes key-value pairs as input and produces a set of intermediate key-value pairs. The MapReduce model then groups together all pairs associated with the same key and passes them to the reduce function, which will produce the final output key-value pairs. Systems that are based on this programming model typically operates on static data by scheduling batch jobs. However, streaming data is not static, and instead of treating streams as streams, they are often batched into static data sets and then processed in a time-agnostic fashion. By looking at the definition of streaming data, presented in the last chapter, it is clear that it has emerged a need of having a different architecture than the one used for batch processing.

One system that has tried to solve this new need is **Apache Flink**[5]. Flink is an open-source system for processing streaming and batch data [39]. Flink is based on the philosophy that many applications can be expressed and executed as pipelined fault-tolerant data flows. This philosophy concerns real-time analytics, batch processing of historical data and iterative algorithms, like ML and graph analysis. A Flink runtime program is a *Directed Acyclic Graph* (DAG) of stateful operators that are connected with data streams. To handle both streams and batch processing, Flink has two APIs called DataSet API, for processing finite data sets, and the DataStream API, for processing potentially unbounded data streams. The APIs in addition to a highly flexible windowing mechanism, allows Flink programs to compute both early approximate results as well as delayed accurate results in the same operation. Flink is based on the *Actors* model, a model for concurrent computations in distributed systems, which means that actors are the universal primitive which messages are exchanged through [40]. The model defines some general rules for how the system's components should behave and interact with each other. In this case, the components are the FlinkClient, the Job Manager, and multiple Task Managers. The actors can only communicate with each other through messages that are being sent asynchronously. When an actor is receiving a message, the actor itself can create more actors, send messages to other

[5]Apache Flink: `https://flink.apache.org`

actors or designate what to do with the next message it will receive.

The previously mentioned system AsterixDB does also support ingestion of data streams. The mechanism used is called *Data Feeds* and allows having continuous data streams arriving into AsterixDB from external sources [38]. The functionality of establishing a connection with a data source and receiving, parsing and translating its data into the right storage objects is contained in a *feed adapter*. The feed adapter may operate in *push* or *pull* mode. Push mode involves one initial request by the adapter to the data source to set up the connection. Once the connection is authorized, the data source can push data to the adapter without any further requests. However, when operating in pull mode, the adapter has to make separate requests every time it wants to receive data. When defining a feed, there is also an option to include the specification of a user-defined function (UDF) that will be applied to each incoming feed record before persistence. The UDFs allows users to perform sentiment analysis, feature extraction, filtering of records, and other useful operations. A feasibility analysis of stream-based processing of semi-structured data has been conducted by Päkkkönen [41]. The goal of the analysis was to compare AsterixDB with a composite system made by **Spark Streaming** and the **Cassandra**[6] NoSQL database. Where Spark Streaming was used for processing streams and Cassandra for persistence. The study focused on finding out how the two systems performed when a content analysis and a sentiment analysis was executed on a stream of Tweets. The results of the study indicated that the two systems scaled roughly similarly. However, the author concludes that AsterixDB performed relatively better in the processing of semi-structured Twitter data, by achieving higher throughput and lower latency than Spark Streaming + Cassandra.

Apache Storm[7] is another system for distributed real-time computations. Storm uses some key abstractions called *spouts* and *bolts*. The spouts work as the sources of streams in a computation while the bolts will process input streams and produce output streams. Similar to other systems, the bolts can run functions, filter the data, aggregate or join, or talk to databases. A Storm application is designed as a *topology* with the shape of a DAG, where the spouts and bolts work as the graph nodes and the edges of the graph are streams going from one node to another. Altogether, the DAG works as a data transformation pipeline. A Storm cluster contains two types of nodes, a master node called *Nimbus* node and worker nodes called *Supervisor* nodes. The Nimbus node is responsible for distributing code around the cluster, assigning tasks to the worker nodes and monitoring for failures. The Supervisor nodes are listening for work assigned to its

[6]Cassandra: `https://cassandra.apache.org/`

[7]Apache Storm: `http://storm.apache.org`

machine and starts and stop work processes based on what Nimbus wants. The coordination between Nimbus and the Supervisors is done through a **ZooKeeper**[8] cluster. Both Nimbus and the Supervisors are fail-fast and stateless. Hence, all state is kept in Zookeeper. Meaning that in case of a system failure, both Nimbus and the Supervisors can be restarted like nothing happened, which makes Storm a fault-tolerant system.

Storm has been used in many popular applications. For a long time Storm served as the main platform for real-time analytics on Twitter. However, when the amount of data that was being processed in real-time started to increase, many limitations of Storm became apparent. Twitter needed a system that could scale gracefully, had better performance, and was easier to manage while also working on a shared cluster infrastructure. Twitter argues that the Storm topology can be challenging to handle in production because of difficulties in debugging [42]. When a topology misbehaves, it is important to find the root-cause as soon as possible to avoid long system downtimes. Since Storm is built on multiple components in one topology that is bounded into one operating system process, debugging can be very challenging. Hence, Twitter needed a system with a clearer mapping from the logical computation units to each physical process. However, since Twitter has a large number of existing applications relying on Storm, it was essential that the new system was compatible with Storm. The resulting system is called **Heron**[9] and is API-compatible with Storm, which allows Storm users to migrate to Heron easily. Heron is, as Storm, designed as a topology with spouts and bolts forming a DAG. As for Storm, the spouts generates the input, and the bolts do the computation. The Heron topology is equivalent to a logical query plan for a database system, meaning that this logical plan is translated into a physical plan before the actual execution. Each topology is run as an Aurora[10] job, which consists of multiple *containers*. One container is called the *Topology Master* and the remaining containers each run a *Stream Manager*, a *Metrics Manager* and a number of processes called *Heron Instances*. The Stream Manager is responsible for routing the incoming tuples in one container, and the Metrics Manager collects and exports metrics from all the components in the system. The Heron instances are the spouts and bolts that run user logic code. Heron allows for many containers to run in parallel and it is the Topology Master's job to control these. For coordination of resource schedulers, Heron relies on ZooKeeper.

Previous work has been done to investigate how Storm, Flink and Spark Streaming perform compared to each other. The motivation behind the

[8]ZooKeeper: `https://zookeeper.apache.org`
[9]Heron: `https://twitter.github.io/heron/`
[10]Aurora: `http://aurora.apache.org`

work was to provide benchmarks for three representative computation engines to simplify the process when choosing the most appropriate platform for a given need. To test the different platforms, a full data pipeline was constructed using Kafka and Redis[11] to mimic the real-world production scenarios. The authors argue that Storm and Flink behave like true streaming processing systems with lower latencies, while Spark Streaming can handle higher throughput but at the cost of somewhat higher latencies. Based on the results, the authors conclude that there is no clear winner and that all platforms have their advantages and disadvantages [43].

3.3 Related Methods to Visualize Streaming Data

Real-time visualization of data streams has become one of the most important research topics in the visualization domain. The need for finding suitable tools that can extract the content, context and the organization of the incoming data stream has increased with the explosion of real-time data applications. The Visual Content Analysis of Real-Time Data Streams project (the VCA project) is a project at the Pacific Northwest National Laboratory. Their goal was to allow users to quickly grasp dynamic data in forms that are intuitive and natural without requiring intensive training in the use of specific visualization or analysis tools and methods [18]. In their research, they have tried to develop dynamic visualization tools that are suitable for different fields and disciplines. Examples of areas are cybersecurity, computer networks, counterintelligence, supercomputing, biological sciences and homeland security.

During the last couple of years, several visualization software products have been developed that do not require any coding experience. Hence, the products are more or less *plug-and-play* and can be used by companies that want to get insights into their raw data without having to do any coding. One example of a graphical system that can perform ad-hoc exploration and visualization of customer data sets is **Tableau**[12]. The system allows companies to prepare interactive visualizations through a desktop application that can either connect to an online data source or work offline on a downloaded source of data [44]. It supports a wide variety of charts, graphs, maps and other graphics and the created charts can easily be integrated into websites and other applications. According to the developers behind Tableau, the system is a convenient alternative for companies and

[11]Redis: `https://redis.io/`

[12]Tableau: `https://www.tableau.com`

customers who do not own a dedicated analytic database server by themselves. Another analytics tool is Microsoft's **Power BI**[13], which allows users to connect to different data sources, such as Excel, MySQL, Google Analytics and Salesforce to mention a few. The tool provides cloud-based BI services, known as *Power BI Services*, together with *Power BI Desktop* which is a desktop-based interface. Power BI offers interactive visualizations, allowing end users to create custom dashboards and reports without having to depend on experts on databases.

In 2015, International Business Machines Corporation (IBM) released the **IBM Watson Analytics** (IBMWA) software [45], which is a data analytics software that automates descriptive, predictive, and visual analytics. In contrast to the Watson system that won *Jeopardy* in 2011, which is based on cognitive computing, IBMWA is based on advanced statistics. To present the process of IBMWA, one can explain it in four stages - Refine, Explore, Predict and Assemble. The first step is used for data exploration and manipulation and the second step allows the user to use their own words to discover the data. The third step is used for predictive analytics, where the user selects a target attribute and the program creates a diagram showing the factors that are most likely to influence business outcomes. The final step is used to display the findings as visualizations on dashboards. The program has been described as very user-friendly. However, it requires data preprocessing, statistical conceptual understanding as well as domain expertise.

With regards to the domain of this project, some research related to visualization of Twitter data has been conducted. **Taghreed** is implemented to be a complete system for efficient and scalable querying, analyzing, and visualizing geo-tagged microblogs. The system is built on four main components to manage and query billions of microblogs records. First, it uses an indexer that handles both main-memory and disk-resident indexes. The main-memory indexes would digest the incoming records in real-time with high arrival rates. When the memory becomes full, a flushing manager is responsible for transferring main-memory contents to the disk indexes. The second component is a query engine that provides query optimization and query processing on top of the system indexes. The third component is a recovery manager that restores the system in case of failure, and lastly, an interactive visualization is used which allows the end user to interact with the system. An overview of the visualization can be seen in Figure 3.2. The authors do also discuss other alternatives such as using AsterixDB to support interactive queries on big semi-structured data. However, they argue that AsterixDB is not suitable to handle microblogs as it, up to that date, did not support digesting and indexing of fast streaming data in

[13]Power BI: `https://powerbi.microsoft.com/en-us/`

real-time or had mechanisms to manage data flushing for memory-resident data to disk [46]. It should be emphasized that the Data Feed functionality described in section 3.2, was introduced after this article was published.

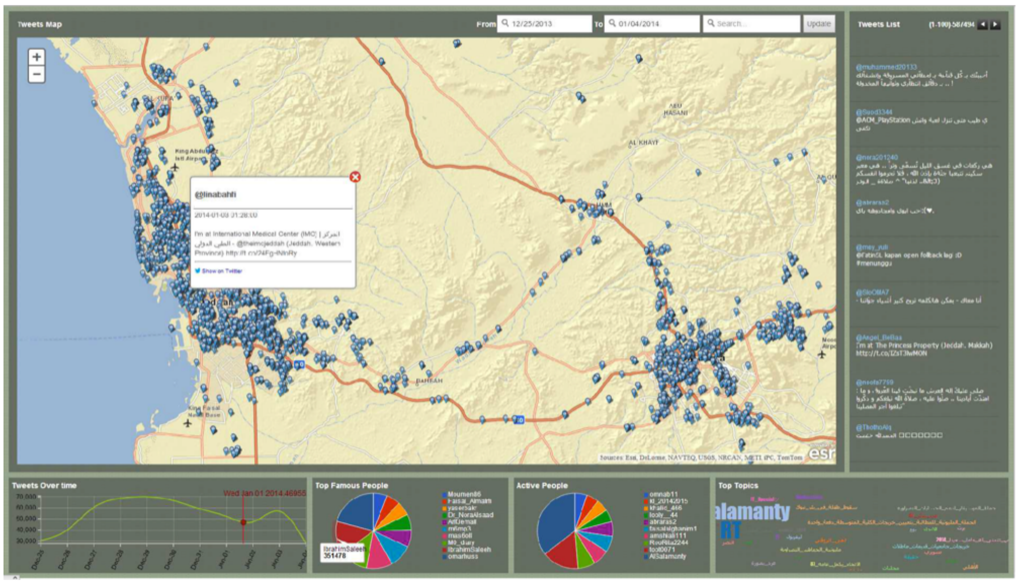


Figure 3.2: Overview of the Taghreed interface [46]

Another system that has been focusing on visualizing Twitter data is **Cloudberry**. Cloudberry is a system that allows users to interactively query, analyze, and visualize a significant amount of data with temporal, spatial, and textual dimensions. These attributes are commonly available in social media and the reason why Twitter was used as the subject for demonstration. Despite the critiques made by the creators of Taghreed, Cloudberry is built with AsterixDB running as backend and has utilized the Data Feed functionality. The authors argue that it allow scalability, as it can utilize a computer cluster to store, index, and query large amounts of information. The idea behind the system was to create a platform for analytical purposes, where the user can use the interface to zoom into more specific details on both spatial and temporal dimensions. On top of AsterixDB is the Cloudberry middleware which is responsible for processing front-end requests and transforming them into efficient AsterixDB queries. Lastly, a Web interface called TwitterMap is developed which displays the spatial and temporal distribution of Tweets. The user can interact with the interface by sending filter and aggregation requests to the Cloudberry middleware which will update the visualization [47]. Figure 3.3 gives an overview of the interface.

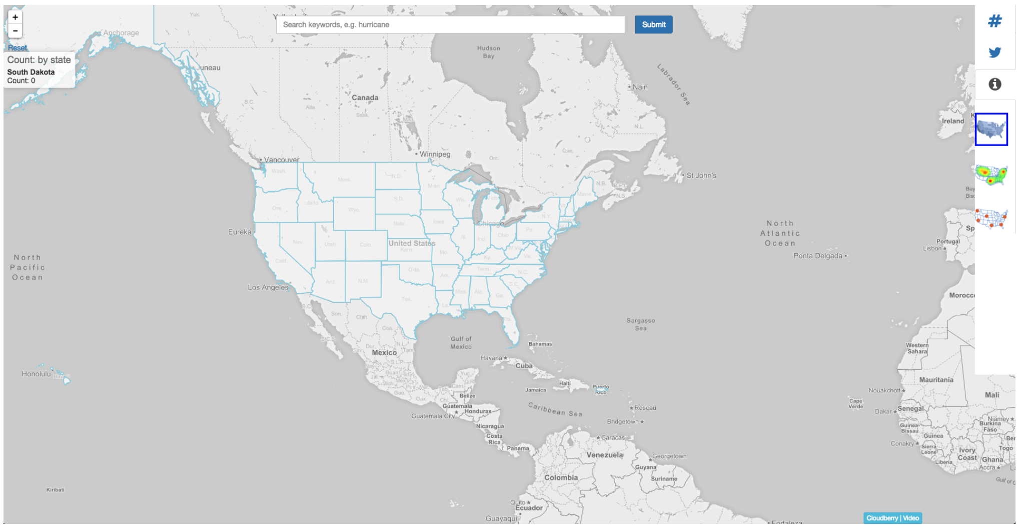


Figure 3.3: Overview of the TwitterMap interface [48]

3.4 Classification and Ranking of Microblogs

The task of filtering a time-ordered corpus of documents that are highly relevant to a predefined set of entities was introduced at the TREC Knowledge Base Acceleration[14] track in 2012. Two main families of approaches occurred, namely *Classification* and *Ranking*. An experiment made by Balog and Ramampiaro, compared classification and ranking for cumulative citation recommendation (CCR) [49]. The goal with CCR is to automatically filter vitally relevant documents from a textual stream consisting of news and social media content and evaluate their citation-worthiness to target Knowledge Base (KB) entities. An example of a Knowledge Base is Wikipedia. In other words, the goal with CCR is to propose documents that a human would want to cite in the Wikipedia article of the target entity. Common to both classification methods and ranking methods is an initial step that provides a filter to identify whether a document contains a mention of the target entity. The filtering step is based on strict string matching and uses known name variants of the entity extracted from DBpedia[15] to find mentions. The goal with identifying entity mentions was to maintain high recall and keep the number of false positives at a low rate. An earlier experiment conducted by the same authors had investigated how recall was affected when expanding the filter with known name variants [50]. The experiment achieved a recall of 86.2% and 84.2% on the training and testing periods, respectively, by just using the name of the URL. By adding

[14]TREC Knowledge Base Acceleration: `http://trec-kba.org/`
[15]DBpedia: `http://www.dbpedia-spotlight.org/`

known name variants with DBpedia, the recall were pushed to 97.4%. The authors concludes that using DBpedia variants with strict matches provides a balanced setting for identifying entity mentions. After identifying entity mentions, the experiment continues to investigate whether classification or ranking performs best. The study concludes that ranking methods outperform classification methods with respect to their ability to identify central documents. Even if the CCR task does not concern microblogs explicitly, some parallels can be drawn and the work conducted in the study are in many ways applicable to the work to be done in this project.

After the arising of microblogs, research regarding the ranking of relevant content has emerged. Before presenting part of this research, Twitter's ranking algorithm will be explained. At the beginning of the Twitter history, a user's timeline composition was easy to describe. It would show all the Tweets from the people you follow since your last visit, in reverse-chronological order. However, this strategy is not able to avoid spam nor present the most relevant content for the user. During the past years, Twitter has made subtle additions to their application to boost user engagement. One of those additions is their ranking algorithm. The motivation behind the algorithm is that recency does not always equal relevance, and Twitter wants to present content that is even more relevant to the user [51]. Historically, a Tweet's relevance has been determined by the user's interests and rationale. To solve this, each Tweet is scored by a relevance model at publication time. To calculate the score, the model uses a variety of features such as the Tweet itself, the author of the Tweet and the user. The goal with the score is to predict how exciting and engaging a Tweet would be for a specific user. A set of the highest-scoring Tweets is shown at the top of the timeline while the remainder is shown below in reverse-chronological order.

Through the history, the majority of all microblogging search engines has relied on keyword-based retrieval strategies. However, as the need for returning even more relevant answers has increased, research that explores other strategies has taken place. A paper written by Tao et al. investigates if there exist additional micropost characteristics that are more predictive of a post's relevance and interestingness than its keyword-based similarity with the query [52]. The authors explore sixteen features along two dimensions: topic-dependent and topic-independent. Examples of topic-dependent features are the retrieval score derived from retrieval strategies based on statistics or the semantic overlap score which determines the overlap between the semantic meaning of a search topic and a micropost. Topic-independent features, on the other hand, can be divided into *Syntactical features* (presence of URLs or hashtags), *Semantic features* (the diversity of semantic concepts mentioned in the post) and *Social context features* (the authority and popularity of the user who posted the micropost). The analysis concludes that an understanding of the semantic meaning of the Tweets plays a vital role in determining the relevance of a Tweet based on a query. The authors also point out that the length of a Tweet and the social context of the user posting it have little impact on the prediction of relevancy.

The feature set is a widely discussed theme in research regarding the ranking of microblogs. Duan et al. proposes a feature set based on *Content relevance features*, *Twitter specific features*, and *Account authority features*. Where the first refers to those features which describe the content relevance between queries and Tweets and the second to those features which represent the particular characteristics of Tweets, such as the number of retweets and likes. The last refers to features which represent the influence of the authors [53]. The paper proposes a Tweet ranking strategy by applying learning to rank algorithms to determine the best set of features. However, the paper concludes that the system performs best by using a set of five different features, where whether a Tweet contains a URL or not is the most effective feature. Regarding account authority, the authors conclude that the number of times other users have listed an account performs better than the number of followers which was used earlier on Twitter.

Chapter 4

Ranking of Results from Continuous Queries

This chapter will provide the solution for the stated project goal and associated research questions. The chapter will provide an insight into the development phase and how the solution was implemented. The task and domain for the project will be introduced, before the conducted approach is presented by explaining the process through each iteration introduced in section 1.4.3. The solution and different decisions through each sprint will be described before the practical implementation will be presented.

4.1 Domain

This section outlines the domain for the project.

4.1.1 Twitter

The domain for this project will be Twitter. Twitter is considered as a microblog where users can follow other users and post messages that their followers can read. These messages are called *Tweets* and have a restricted message length, which makes them concise and straight to the point. Twitter constitutes a wide spreading instant messaging platform, and people use it to get informed about world news, reviews and critiques, product launches and events for a specific geographical location, to mention a few. Since tweets naturally contain rich sentiment information, they are often used for sentiment analysis. Twitter provides samples of its data available as data streams for consumers and developers which makes it ideal for analysis and research.

There exist some sentiment labels in the Twitter-domain [2]. Consider the following Twitter example message: `RT @louisa has won a #gold medal`. The tweet shows three of these sentiment labels. The symbol `#` is called a hashtag, and it is used to denote a topic, subject or category of a tweet. `RT` is used at the beginning of a tweet to indicate that the message is a so-called "retweet", a reposting of a previous tweet. The character `@`, as in, `@louisa` is called a mention and is used to reply to other users by indicating their usernames. Another sentiment label is emoticons, which refers to a digital icon or a sequence of keyboard symbols that serves to represent a facial expression. All of these labels are suitable for Social Media analysis, and Twitter is considered as a suitable data source to prove the underlying theories for this project.

4.2 Task

This section presents the tasks to conduct the project within.

4.2.1 Filtering on Relevant Keywords

The primary task for this project is filtering the Twitter stream on words inserted by the user. Throughout this report, the inserted search words will be referred to as **keywords**. As mentioned in section 1.1, the system from the specialization project sent all retrieved Twitter data straight to Elasticsearch and Kibana, including non-relevant information as spam [1]. By filtering the Twitter stream directly in Spark, noisy information with no value can be excluded and make the visualizations as relevant as possible for the user. The user will, therefore, insert a query based on what he or she wants to visualize. The system then needs to find a list of related keywords based on the query which will be used by the Spark engine to filter the Twitter data. The reason why it is important to find related keywords is mainly due to the nature of tweets. The maximum length of a Tweet was initially set to 140 characters, but since November 7, 2017, the length was extended to 280 characters for all languages except Japanese, Korean, and Chinese [54]. However, regardless of the extension, a Tweet is still very concise and straight to the point and is often too short to contain any detailed information. This means that a Tweet itself might not give any useful insights. However, all Tweets generated daily will collectively provide valuable information about public opinions and current trends. As the fixed message length prohibits detailed descriptions of the messages, filtering on only one keyword will probably be too narrow, and a lot of useful and essential Tweets will be discarded. To increase the vocabulary and improve the analytics of a specific theme, related keywords that means the same as the keyword inserted by the user needs to be found. One example might be the keyword `President`, where other related keywords are `Trump` and `Obama`.

4.2.2 Sentiment Analysis

The other task for this project is sentiment analysis. More precisely, the task of classifying whether a tweet is positive or negative. However, the focus of this project is not the implementation or execution of the sentiment analysis. But it is used to show how Spark and its libraries can be utilized, and also how it can be visualized in Elasticsearch and Kibana. The goal of the project is to create a visualization that allows the user to understand the data in an intuitive way. Thus by classifying the sentiment on every incoming Tweet, the user will be able to interpret the overall feelings and opinions about a topic.

4.3 Theoretical Solution

The problem that was going to be solved had its background in streaming data and the use of visualization tools to gain insights into the incoming data. In recent years, several attempts have been made to solve this problem. Two examples were mentioned in section 3.3, where Cloudberry is perhaps the most comparable to what is trying to be developed in this project. One thing, however, distinguishes Cloudberry from the chosen solution as it retrieves all data from Twitter and then performs filtration after it has been stored. In this project, there is a wish to avoid unnecessary storage of irrelevant information, and thus the filtering should take place earlier in the pipeline.

Following is a review of the work done in each of the three sprints conducted in the development phase. Each section will provide an explanation of decisions made and any challenge that has arisen.

4.3.1 First Sprint

The main focus of this iteration was to implement a pipeline with the chosen technologies presented in section 2.6. After the pipeline was realized, a classification model used for sentiment analysis was implemented.

Implementation of Pipeline

The first step of the sprint was to become familiar with existing technologies to find possibilities and limitations with each of them. This was to a large extent done during the specialization project. However, for this project, the streaming ETL process mentioned in section 2.2.1 was going to be taken into account, and thus let the Extraction part be separated from the Transformation and Loading parts. By considering the ETL process, the goal was to create a pipeline that could extract data from Twitter, make appropriate transformations, before loading it to the final data storage system. The pipeline should be fault-tolerant and executed in real-time. The Transformation part of the system needed to be an analytics engine for large-scale data processing that was able to handle streaming data and support machine learning for the sentiment analysis task. Based on the motivation that Spark is a unified engine that targets various processing workloads, such as stream processing and machine learning, it was used as the underlying technology responsible for the Transformation part. Even if Spark Streaming is fully capable of receiving Twitter messages directly from the Twitter Streaming API, it is mainly a stream processing engine and should also be treated like that. To distribute the workload, and allow the Spark engine to focus on the Transformation part, another technology was going to be used to extract Tweets from the Twitter Streaming API. Kafka provides publish-subscribe messaging and can be thought of as a distributed, partitioned, replicated commit log service. One of the main use cases of Kafka is to build real-time streaming data pipelines that reliably get data between systems or applications. As Kafka provides integration with Spark, it was seen as a natural choice for the task. An advantage with Kafka is that it keeps the extracted data throughout the retention period. All messages written to Kafka will be written to disk and replicated for fault-tolerance. Consequently, this means that if the Spark application would go down, the data would not be lost but stored in Kafka. Similarly to Spark Streaming, Kafka is also able to retrieve Tweets from the Twitter Streaming API. Hence, Kafka was chosen as the technology responsible for the Extraction part. The rest of the pipeline was implemented as for the initial project, where Tweets were sent to Elasticsearch for offline storage. Elasticsearch allows the incoming data to be stored in a distributed manner while at the same time visualize it in near real-time through Kibana. The seamless integration between Elasticsearch and Kibana was the main reason for using the Elastic stack in the proposed solution. Figure 4.1 and Figure 4.2 shows the evolvement of the pipeline.

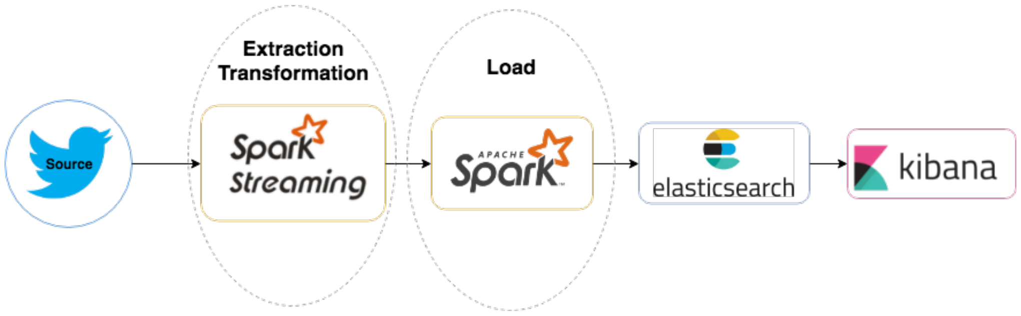


Figure 4.1: Setup for initial project

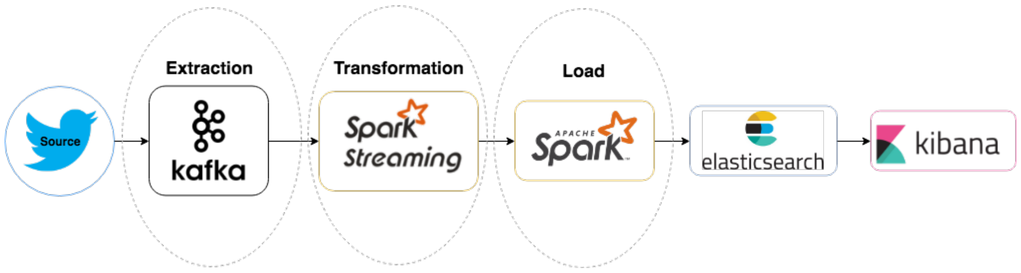


Figure 4.2: Extended setup for current project

A challenge that arose during the implementation of the pipeline was parsing the stream to appropriate format. The Twitter Streaming API returns Tweets that are encoded using JavaScript Object Notation (JSON). JSON is based on key-value pairs with named attributes and associated values. These attributes and their state are used to describe objects. Two core objects that are represented as JSON in Twitter are *Tweets* and *Users*. Where each Tweet has an author, a message, a unique ID, a timestamp of when it was posted and sometimes other metadata shared by the user. Each User has a username, an ID, a number of followers, and often an account biography [55]. An example of a Tweet in JSON format can be found in Appendix B in Listing B.1. However, to take advantage of the Twitter4j library and extract the information needed for the visualization, the incoming JSON objects must be parsed to Twitter4j Status objects[1]. After an extensive investigation of how Kafka process and converts an object into a stream of bytes before transmitting it, the solution was to use a

[1]http://twitter4j.org/javadoc/twitter4j/Status.html

StringSerializer and send Tweets in raw JSON-format. Spark is then able to utilize the Twitter4j library to parse the raw JSON string into a Twitter4j Status object. Example of a Tweet that has been parsed to a Status object can be seen in Listing B.2 in Appendix B.

Sentiment Analysis

The next part of the first sprint was focusing on implementing a machine learning algorithm that could perform sentiment analysis. Even if the visualization in Kibana could provide exciting findings just by showing standard information about the received Tweets, there is a need to demonstrate how the Machine Learning library in Spark can be utilized to provide valuable information. By including sentiment analysis in the visualization, the relation between positive and negative Tweets related to a specific topic will be shown, and stakeholders can use it to get business insights about for example customers and users of a product or service. To perform sentiment analysis, a classification model was implemented in Spark. Spark's Machine Learning library MLlib supports both Bernoulli Naive Bayes and multinomial Naive Bayes, where the latter was used for the model. The advantage of using Naive Bayes as classification algorithm is that the data only needs one pass against the training set to classify it. Hence, considering this as a proof of concept it was decided to be suitable for the task.

Figure 4.3 shows an overview of the Machine Learning mechanism to be implemented. The first step is to create a model using Naive Bayes classifier. Given a training dataset with annotated data, the `Data Preparation` step is responsible for preparing the raw data for the `Classifier`. During the `Data Preparation` step, predefined features are extracted for each record in the dataset. Each record is transformed into a vector of tokens, where each token is an occurring term along with its frequency. As the Tweets may contain frequently occurring words that might skew the sentiment, a list containing *stop words* is uploaded and broadcast. Removing stop words will hopefully remove noise and make the model as accurate as possible. After the preferred features are extracted, and all stop words in the Tweets are removed, it is sent to the `Classifier`. The `Classifier` takes the desired classification algorithm as a parameter and builds and stores the model. In Figure 4.3, `f` is the extracted feature and `c` is the sentiment classification - Positive, Negative or Neutral.

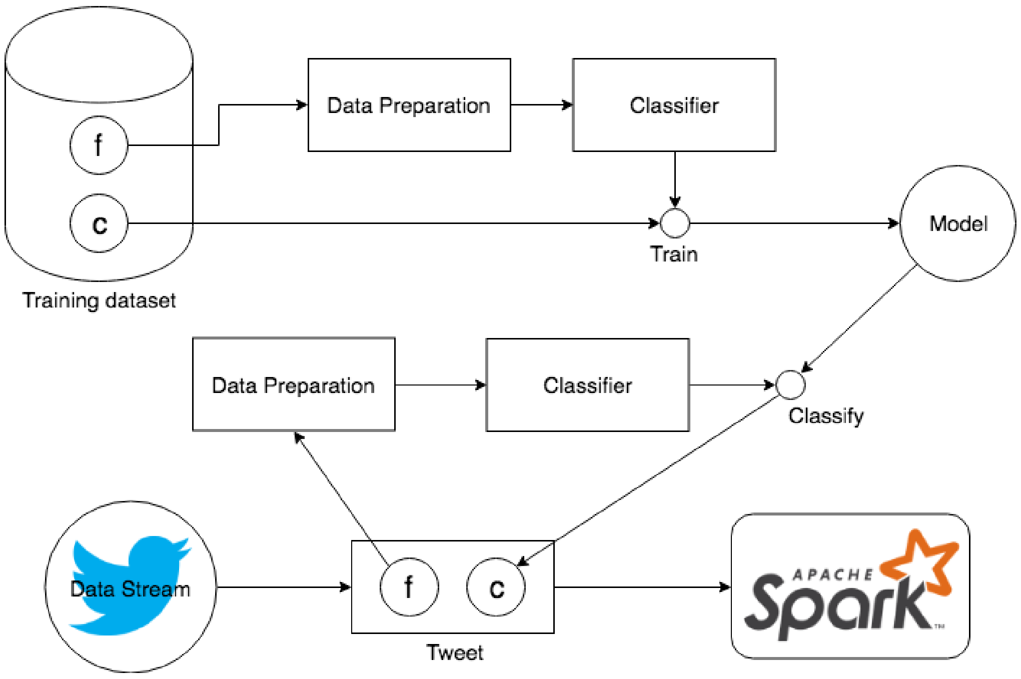


Figure 4.3: A high-level overview of the Machine Learning mechanism

4.3.2 Second Sprint

After the pipeline had been implemented and records were transferred smoothly between the different parts of the system, the focus was on answering **RQ1** and **RQ2**. The sprint started by exploring related research concerning filtering and classification of microblogs. Based on previous results, a method for filtering the incoming stream was implemented. However, as stated in both **RQ1** and **RQ2**, each Tweet should be classified and filtered based on queries determined by the user. Hence, related research regarding continuous queries and various possibilities of generating user input were examined. Finally, a solution that combined them both were implemented.

Filter Function

As stated in section 3.4, research regarding classification of microblogs has emerged with its increasing popularity. Especially the feature set used for determining relevance has been a popular topic. As stated in **RQ2**, the definition of relevance has to be considered for this specific use case, and thus which feature set to use. For a system that is focusing on *event detection*, a proper feature could be locations in terms of latitude-longitude pairs.

However, the use case for this specific system is that a user should be able to filter the incoming stream based on interests. In this context, relevance will, therefore, be defined as the textual similarity between a user's interest and the content of a Tweet. Hence, a keyword-based retrieval strategy was chosen as feature set for the query. Since the chances are small that a Tweet will contain exactly the word that the user has searched for, there is a need to expand the filter and find other related words that can be used to filter the incoming stream. However, the system should be user-friendly and require as little as possible from the user, and thus the expanding of the vocabulary should be done automatically. The CCR task mentioned in section 3.4, performs an initial filtering step to identify whether a document contains a mention of the target entity. The filtering was based on strict string matching and used known name variants retrieved from DBpedia to expand the filter. As the approach was used for identifying entity mentions in Wikipedia articles and had shown to provide high recall, it would be interesting to implement a similar method for filtering on Tweets which has a completely different structure regarding length and content. A method for retrieving related words was, therefore, implemented. However, instead of using DBpedia, another existing Knowledge Base was utilized, more precisely the Google Knowledge Graph. Google provides a read-only API called Knowledge Graph Search API[2], which lets you find entities in the Knowledge Graph by returning a ranked list of the most notable entities that match certain criteria.

Before continuing with examining user-defined queries, the implemented filter function was tested with static, predefined words and by monitoring the output, it was found to be successful.

User-Defined Continuous Query

To answer **RQ1**, the concept of continuous queries and how it can be executed on streaming data had to be explored. As stated in the previous chapter, a challenge when working with microblog data is that they often require indexing of both real-time and historical data since users are mostly asking about data that has already arrived at the system. This is indeed a requirement for this project as well and has been solved by using Elasticsearch. However, for the proposed solution we are also going to ask for information before it has arrived at the system. A system that has tried to implement the same idea is the BAD system presented in section 3.1. The BAD system is designed to deliver data of interest for users while still supporting analysis of historical information based on AsterixDB and its

[2]Google Knowledge Graph: `https://developers.google.com/knowledge-graph/`

streaming functionalities. By allowing the users to subscribe for themes of interests, the system will collect data from various sources and provide it for the user. The BAD system is comparable to the proposed solution for this project, but instead of using AsterixDB, Spark and Spark Streaming is utilized. Furthermore, this project will investigate how the same idea can be solved by allowing users to submit a continuous query instead of using subscriptions. The definition of a continuous query is that it is asked for every incoming record in the system, and must be defined a priori. Hence, it needs to be retrieved by Spark. Figure 4.4 shows how the vision behind this solution fits into the overall active systems platform space.

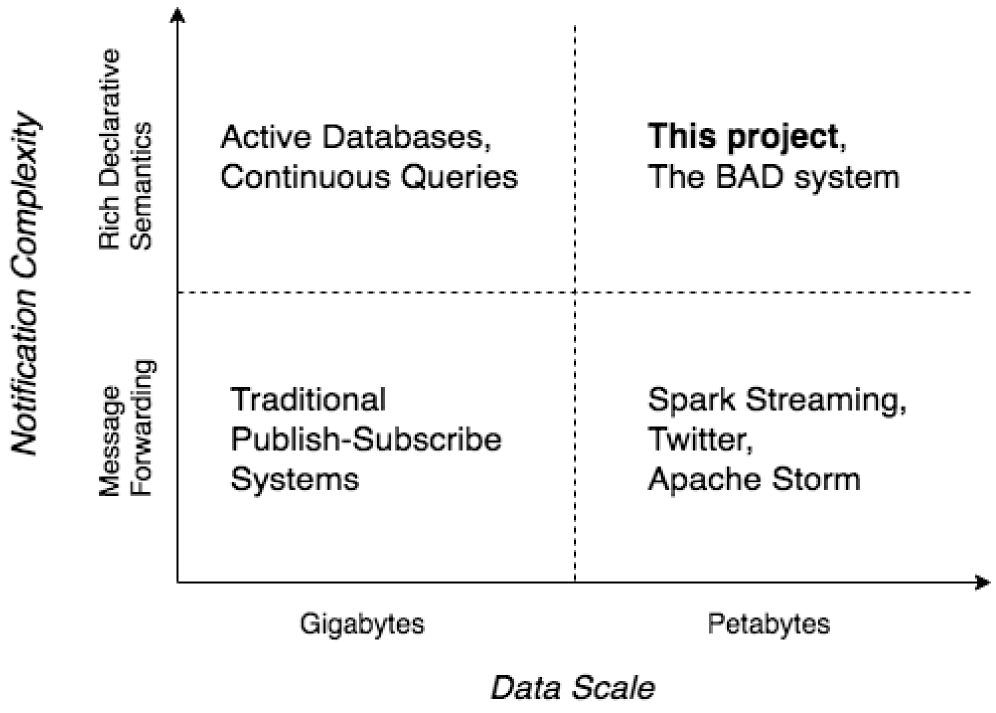

Notification Complexity
Rich Declarative Semantics
Message Forwarding
Active Databases, Continuous Queries
This project, The BAD system
Traditional Publish-Subscribe Systems
Spark Streaming, Twitter, Apache Storm
Gigabytes
Petabytes
Data Scale

Figure 4.4: The proposed solution in context of other systems

However, the query should not only be of a specific type, but it should also be defined by the user. Therefore, functionality for generating user input was examined. There are various possibilities for retrieving user input, and options such as using a socket and implementing a form in HTML and PHP were tested. As different options were reviewed, the need for storing the input became evident. The same approach used for retrieving Tweets with Kafka was tested and proved to be suitable for the task. Kafka provides a *Kafka Console Producer* where the user can insert words and send it to a defined topic. The Spark application is then able to create a new DStream by subscribing to that topic. The same way Tweets are being stored, so will the user-defined query be.

Filter Based on Query

The last step of the second sprint was to combine the user-defined query with the implemented filter function. One specific challenge arose during this step. As Spark is treating the user-defined query provided by Kafka as a stream, it contains mutable objects. The implemented filter function, however, can not filter on mutable objects as they are constantly changing. To overcome this problem, some interim storage was needed. The user input should be written to a distributed repository and different options such as MemCached[3], HBase[4], and a regular database were examined. However, due to the time constraints and limited resources, it was concluded to use a regular file as proof of concept. Consequently, the words inserted by the user will, together with the related words retrieved from the Google Knowledge Graph Search API, be written to a separate file. The filter function will then read from the file and check if the Tweets contains any of the words. The function will create a new DStream by selecting only the records on which the given function returns true.

Since the proposed solution may potentially lead to many words being read from the file, it is essential that the function is tested. An experiment should, therefore, be conducted to investigate how the execution time behaves when the number of words increases.

4.3.3 Third Sprint

The focus of the third and last sprint was to answer **RQ3**. In the context of the project, relevance was defined as the textual similarity between a Tweet and the query. Hence, the sprint started with an examination of known statistical scoring functions that can be used to rank the incoming Tweets based on their relevance to a query. Furthermore, a set of requirements were formulated to answer the research question properly. Finally, various possibilities for implementing a ranking function was investigated before the final function was realized.

Requirements

To provide a basis for what the ranking function should achieve, a set of requirements were formulated as followed:

[3]MemCached: `https://memcached.org/`
[4]HBase: `https://hbase.apache.org/`

1. The function should return a ranked list based on both historical, stored data and streaming, real-time data
2. Recent posted Tweets should be considered as more relevant than old ones
3. The function should rank Tweets based on keywords determined by the user

Choice of Technology

With the defined requirements in mind, an investigation of how the ranking function could be implemented begun. The function should be implemented somewhere in the proposed pipeline. To avoid unnecessary computations, the ranking function needs to be applied after the Tweets have been filtered in Spark, and thus Kafka was not appropriate for the task. The choice whether to use Spark or Elasticsearch was determined based on their capabilities of implementing known scoring functions and whether they fulfilled the stated requirements.

MLlib has the possibilities of implementing various feature extraction algorithms such as TF-IDF[5], which makes Spark a potential candidate for the task. As explained in section 2.5.2, TF-IDF uses a collection of documents that make up a corpus, to assign scores to words in each document which represents how important the word is. In this context, the corpus would be the collection of all retrieved Tweets. However, as Spark is strictly an in-memory processing engine, it does not persist data for use outside of the current execution. Once a Tweet has been processed, it will be gone. To use Spark for this task, there is a need to retain a vocabulary of words seen in all Tweets which will function as the corpus for TF-IDF. The second requirement can quite easily be met by adding a temporal aspect to the ranking function. The third requirement is also achievable since Spark is already retrieving user-defined keywords for the filter function. However, the first requirement makes it more complicated. To provide a ranking list based on both historical and real-time data, Spark must keep a list of the top-k most relevant Tweets, where a naive approach would be to justify for every incoming record, whether or not it should be added to the list.

Elasticsearch, on the other hand, is built on top of Lucene and is by that able to take advantage of its full-text search capabilities. By utilizing the *practical scoring function* provided by Lucene, Elasticsearch can implement a ranking function based on known scoring functions that will be applied on every incoming record. Each record will then be assigned a score based

[5]https://spark.apache.org/docs/2.2.0/mllib-feature-extraction.html

on the defined query, and the score will continuously be updated as new records arrive at the system. Consequently, the first and third requirements can be considered as fulfilled. As Elasticsearch also has its ways of boosting the relevance score, the second requirement can be achieved by applying a decay function.

After both possibilities were considered, it was concluded to use Elasticsearch as the underlying technology for the ranking function.

4.4 Implementation

This section will provide an overview of the practical implementation. The section begins with a description of the prerequisites needed. Furthermore, the most central parts of the solution will be described based on the steps of the ETL process.

4.4.1 Prerequisites

The project has been developed using IntelliJ IDEA, which is a Java integrated development environment developed by JetBrains[6] that also offers support for Scala applications. The Spark program was implemented using Spark version 2.2.1. Spark itself has been developed in Scala but provides high-level APIs in Java, Scala, Python, and R. The process of choosing which language to use, was depending on a few requirements as performance, the complexity of the language, integration with existing libraries and the best utilization of Spark's core capabilities. Since Java is more verbose compared to the other which unnecessarily increases the lines of code, the choice was mainly between Scala and Python. Comparing Scala and Python, the most significant difference is that Scala is up to 10 times faster than Python. The syntax for Python might be easier to learn, but since Spark is developed in Scala, knowing Scala will make the source code more understandable if something does not function as expected. Another reason for choosing Scala is that new features in Spark will be available in Scala before other languages. Hence, the Spark program has been developed in Scala and more specifically Scala version 2.11 since later releases are not compatible with Spark. Kafka version 1.0.1 was used, and the Kafka Producer has been developed in Java version 1.8.0.

All parts of the system are available through Maven Central Repository[7].

[6]IntelliJ: `https://intellij-support.jetbrains.com/hc/en-us`
[7]Maven Central Repository: `https://mvnrepository.com`

To write a Spark Streaming program, one will have to add the proper dependencies to an SBT or Maven project, where the former has been used for the Spark application and the latter for the Kafka Producer. To ingest data from the Twitter source, one must also add the corresponding artifact to the dependencies. The Spark application and the Kafka Producer is available at Github, and links to them can be found in Appendix C.

Elasticsearch version 6.2.4 and corresponding Kibana version 6.2.4 has been used and added to the SBT project in the Spark application. Before any data could be stored in Elasticsearch, the nodes and cluster had to be configured. The system is implemented to run on one node with an index called `visualization`. As the implementation only will run on one machine, there is no point for replication, and thus the number of replicas is set to 0.

4.4.2 Extraction

The Twitter Streaming API

The Twitter data are delivered through an open, streaming API connection. A single connection is opened between the application and the API. The link allows the user to receive the data through repeated requests by the client application, rather than as batches as might be expected from a REST API [56]. Instead, new results will be sent through the link whenever there are new matches, which leads to a low-latency delivery mechanism that can support high throughput. Figure 4.5 shows the overall interaction between the client application and the Twitter Streaming API.

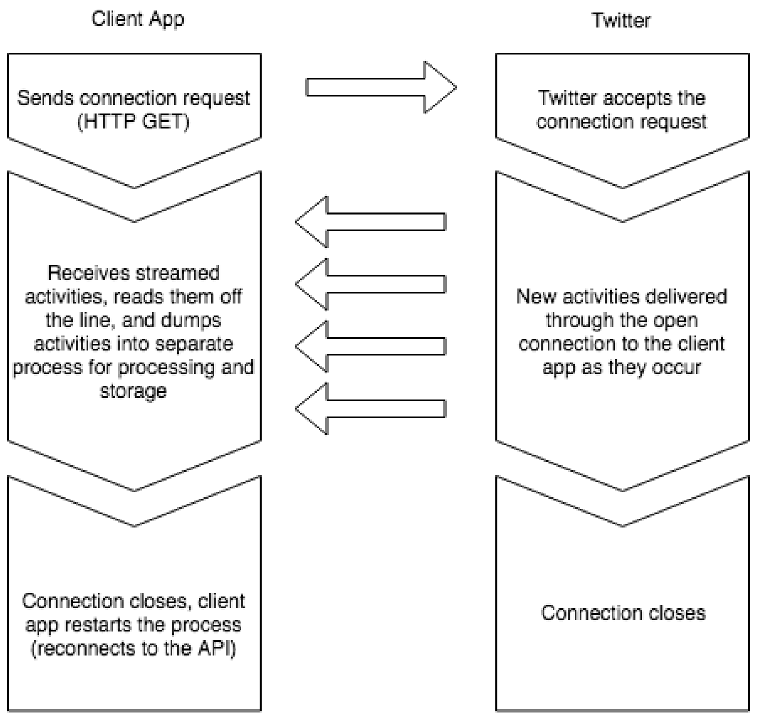

Client App
Twitter
Sends connection request (HTTP GET)
Twitter accepts the connection request
Receives streamed activities, reads them off the line, and dumps activities into separate process for processing and storage
New activities delivered through the open connection to the client app as they occur
Connection closes, client app restarts the process (reconnects to the API)
Connection closes

Figure 4.5: Twitter Streaming API [56]

To retrieve data from the Twitter API, one must first create an application on the Twitter API website[8]. The creation is necessary to get the keys needed for accessing the API. For this project, the *twitter4j* library has been used. Twitter4j is an open-source, unofficial Java library, that provides a Java-based module that can easily access the Twitter Streaming API.

Sending Tweets to a Topic in Kafka

Some prerequisites must be in place before Tweets can be sent to Kafka. By referring to Listing 4.1, the most important parts will be explained. Line 1-11 sets the properties for the Kafka producer which are then being initialized on Line 13. Line 15-21 initializes a new ConfigurationBuilder which sets the authentication keys retrieved from the Twitter API website. The ConfigurationBuilder are then used on Line 23-24 to create a TwitterStream. Line 26-31 creates a new StatusListener which will listen on new Status updates. For each new Status, the TwitterObjectFactory is used to parse it to the right format, before the Kafka producer sends it to

[8]Twitter: `https://apps.twitter.com/`

the given topic. Line 34 starts the stream by calling the `sample`-function. The argument ”en” means that the stream is filtered on Tweets written in English.

Listing 4.1: Producer of Tweets to topic in Kafka

```
 1  Properties props = new Properties();
 2  props.put("bootstrap.servers", "localhost:9092");
 3  props.put("acks", "all");
 4  props.put("retries", 0);
 5  props.put("batch.size", 16384);
 6  props.put("linger.ms", 1);
 7  props.put("buffer.memory", 33554432);
 8  props.put("key.serializer",
 9      "org.apache.kafka.common.serialization.StringSerializer");
10  props.put("value.serializer",
11      "org.apache.kafka.common.serialization.StringSerializer");
12
13  Producer<String, String> producer = new KafkaProducer<>(props);
14
15  ConfigurationBuilder cb = new ConfigurationBuilder();
16  cb.setDebugEnabled(true)
17      .setOAuthConsumerKey(consumerKey)
18      .setOAuthConsumerSecret(consumerSecret)
19      .setOAuthAccessToken(accessToken)
20      .setOAuthAccessTokenSecret(accessTokenSecret)
21      .setJSONStoreEnabled(true)
22
23  TwitterStreamFactory tf = new TwitterStreamFactory(cb.build());
24  TwitterStream twitterStream = tf.getInstance();
25
26  StatusListener listener = new StatusListener() {
27      public void onStatus(Status status) {
28          String rawJson = TwitterObjectFactory.getRawJSON(status);
29          producer.send(new ProducerRecord<String, String>(topicName, rawJson));
30      }
31  };
32
33  twitterStream.addListener(listener);
34  twitterStream.sample("en");
```

Subscribing to a Topic in Kafka

After Tweets have been sent to the right topic by Kafka, Spark Streaming can subscribe to that topic and receive the stream. As mentioned earlier, Spark Streaming discretizes the incoming stream into tiny, sub-second micro-batches called DStreams. Spark Streaming's *Receivers* accept data from the Kafka Producer and buffer it in the memory of Spark's workers' nodes. The Spark engine is then able to run short tasks to process the DStreams and later output the results to other systems. The Spark tasks are assigned dynamically to the workers based on the locality of the data and available resources to achieve better load balancing and faster fault recovery [57].

To receive streams with Spark Streaming, a new StreamingContext must be created as shown in Listing 4.2, where `tweets` will retrieve the input DStream.

Listing 4.2: DStream subscribing to the topic "twitterdata"

```
val kafkaParams = Map[String, Object](
  "bootstrap.servers" -> "localhost:9092",
  "key.deserializer" -> classOf[StringDeserializer],
  "value.deserializer" -> classOf[StringDeserializer],
  "group.id" -> "my-group"
)
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = KafkaUtils.createDirectStream[String, String](ssc,
                                                  PreferConsistent,
                                                  Subscribe[String, String](topicName,
                                                                            kafkaParams))
                      .map(record => record.value)
```

To create a DStream with Kafka one has to utilize the KafkaUtils library which allows us to create an input stream that pulls messages from Kafka brokers. The function takes three arguments where `ssc` is the Streaming-Context, `PreferConsistent` is the location strategy which will consistently distribute partitions across all executors, `Subscribe` subscribes to the right topic and applies the appropriate settings. In this case, the `topicName` is set to "twitterdata". Lastly, the stream is represented in key-value pairs and a map function is performed to retrieve only the values.

Figure 4.6 shows how the incoming data stream is being divided into small batches which will form the DStream, stored in `tweets`. The DStream will be stored in memory as RDDs.

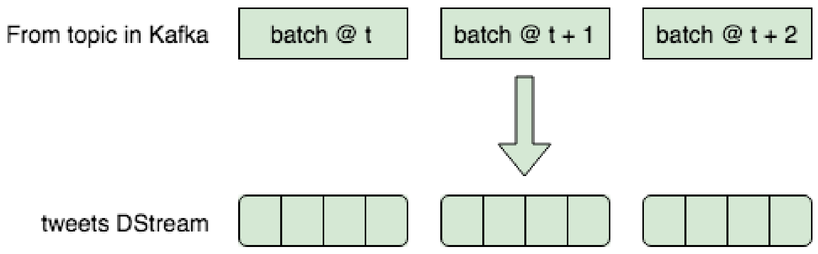


Figure 4.6: Creation of DStream

4.4.3 Transformation

Filter Function

As explained in the previous section, the filter function is based on a continuous query defined by the user. To receive the input, a new topic called "keyworddata" was set up. By using the built-in *Kafka Console Producer*, the user can insert keywords and send it to the topic. The Spark application is then able to use the initialized StreamingContext to set up a new DStream that are listening to the topic. Listing 4.3 subscribes to a new topic, where `topicName` is set to "keyworddata".

Listing 4.3: DStream subscribing to the topic "keyworddata"

```
val filterLines = KafkaUtils.createDirectStream[String, String](ssc,
                                                PreferConsistent,
                                                Subscribe[String, String](topicName,
                                                                          kafkaParams))
                          .map(record => record.value)
```

When the keywords have been retrieved in Spark, the next step is to utilize the Knowledge Graph Search API to retrieve related words that can expand the query. The following example will search for the word "Trump" and return the top 10 highest ranked results shown in Listing 4.4:

`https://kgsearch.googleapis.com/v1/entities:search?query=Trump&key=API_KEY&limit=10&indent=True`

Listing 4.4: Returned results from the Knowledge Graph Search API

```
Donald Trump
Presidency of Donald Trump
Protests against Donald Trump
Donald Trump 2017 presidential inauguration
Presidential transition of Donald Trump
US Presidential Election 2016
The Trump Organization
Trump International Hotel Las Vegas
Melania Trump
Trump International Hotel and Tower
```

The keywords and its associated words will be written to a separate file. The filter function will then read from the file and check if the Tweets contains any of the words. The function will create a new DStream containing the Tweets for which the function returns true. An example of the function can be seen in Listing 4.5. Throughout this report, the filter function will be referred to as the `filter`-function.

Listing 4.5: Example of the `filter`-function

```
val filteredTweets = tweets.filter(status => readWords().exists(status.contains(_)))
```

Sentiment Analysis

The goal of the sentiment analysis is to classify whether a Tweet is Positive, Negative or Neutral. The model created with Naive Bayes is applied in real-time to the retrieved Tweets to determine the sentiment of each of them.

Listing 4.6 shows how the method for generating a classification model using the multinomial Naive Bayes algorithm has been implemented. Line 14 illustrates how MLlib in Spark has been utilized to train the model based on the training set. Where `labeledRDD` is the training data with the extracted features “polarity” and “status”, `lambda` is the *additive smoothing* parameter and `modelType` defines whether the method should us multinomial or Bernoulli Naive Bayes.

Listing 4.6: Creation of Naive Bayes Classifier”

```
1  def createAndSaveNBModel(sc: SparkContext,
2                          stopWordsList: Broadcast[List[String]]): Unit = {
3     val tweetsDF: DataFrame = loadTrainingFile(sc,
4                                               PropertiesLoader.trainingSetFilePath)
5     val labeledRDD = tweetsDF.select("polarity", "status").rdd.map {
6        case Row(polarity: Int, tweet: String) =>
7        val tweetInWords: Seq[String] = getStrippedTweetText(tweet,
8                                                             stopWordsList.value)
9        LabeledPoint(polarity, transformFeatures(tweetInWords))
10    }
11
12    labeledRDD.cache()
13
14    val naiveBayesModel: NaiveBayesModel = NaiveBayes.train(labeledRDD,
15                                                            lambda = 1.0,
16                                                            modelType = "multinomial")
17    naiveBayesModel.save(sc, PropertiesLoader.naiveBayesModelPath)
18 }
```

Once the model has been trained with the training dataset, it can be used to classify sentiment for the incoming data stream. Each record is processed in the same way as the training dataset was, where the `Data Preparation` step extracts the features and removes words that are occurring in the stop-words list, before passing it forward to the `Classifier`. The `Classifier` uses the trained model to classify the sentiment for each Tweet and returns the result to the Spark system. Line 5 in Listing 4.7 shows how MLlib is used for prediction. The predictions are later translated into “POSITIVE”, “NEGATIVE”, or “NEUTRAL” to visualize it more intuitively in Kibana. It is also worth noting that the model is only working on Tweets written in English and all non-English Tweets will be classified as neutral.

Listing 4.7: Computation of Sentiment

```scala
def computeSentiment(text: String,
                     stopWordsList: Broadcast[List[String]],
                     model: NaiveBayesModel): Int = {
    val tweetInWords: Seq[String] = getStrippedTweetText(text, stopWordsList.value)
    val polarity = model.predict(transformFeatures(tweetInWords))
    normalizeMLlibSentiment(polarity)
}

def normalizeMLlibSentiment(sentiment: Double) = {
    sentiment match {
        case x if x == 0.0 => "NEGATIVE"
        case x if x == 2.0 => "NEUTRAL"
        case x if x == 4.0 => "POSITIVE"
        case _ => "NEUTRAL" //neutral if model can't figure out sentiment
    }
}
```

4.4.4 Load

Extracting Appropriate Fields

A visualization should allow the user to quickly and intuitively understand the content, context, and the organization of the incoming data stream. As can be seen in Listing B.1 in Appendix B, every incoming Tweet contains various fields where some of them might not be of interest for our visualization. To extract the fields needed, a HashMap is used as shown in Listing 4.8. As the Tweets have been parsed to Twitter4j Status objects, it is simple to extract the appropriate fields.

Listing 4.8: Mapping of data types

```scala
val tweetMap = tweets.map(status => {
     val status = TwitterObjectFactory.createStatus(json)
     val hashtags = status.getHashtagEntities().map(_.getText())
     val formatter = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss")
     val sentiment = predictSentiment(status)

   HashMap(
       "userID" -> status.getUser.getId(),
       "userScreenName" -> status.getUser.getScreenName(),
       "userName" -> status.getUser.getName(),
       "userDescription" -> status.getUser.getDescription(),
       "message" -> status.getText(),
       "messageLength" -> status.getText.length(),
       "hashtags" -> hashtags.mkString(" "),
       "createdAt" -> formatter.format(status.getCreatedAt.getTime()),
       "friendsCount" -> status.getUser.getFriendsCount(),
       "followersCount" -> status.getUser.getFollowersCount(),
       "coordinates" -> Option(status.getGeoLocation).map(geo =>
          {s"${geo.getLatitude},${geo.getLongitude}"}),
       "placeCountry" -> Option(status.getPlace).map(place =>
          {s"${place.getCountry}"}),
       "userLanguage" -> status.getUser.getLang,
       "statusLanguage" -> status.getLang,
       "sentiment" -> sentiment,
       "deviceType" -> status.getSource,
       "retweetCount" -> status.getRetweetCount
       )
   })
   tweetMap.foreachRDD{ tweet => EsSpark.saveToEs(tweet, "visualization/tweets")
}
```

Elasticsearch allows for automatic detection of types, meaning that there is no need for explicitly defining the types in the application. When the Spark application is executed, the index, type and data fields will automatically be created. This feature is called *Dynamic Mapping* [58]. By referring to Listing 4.8, the `userID` field will automatically be mapped as type `number` and `createdAt` as type `date`. However, some mappings can be hard for Elasticsearch to detect. As Twitter is a worldwide used application, it is of great interest to see where a Tweet was tweeted by extracting a user's geo-location. Elasticsearch has its own data type for geo-location called *geo-point*[9]. A field of this type will accept latitude-longitude pairs, which can be extracted from a Tweet shown as `coordinates` in Listing 4.8. This field would dynamically be mapped as a `string` and not a `geo-point` and hence the mapping has to be done manually in Elasticsearch. Fortunately, Elasticsearch provides a full Query Domain-Specific Language based on JSON to define queries, and the manual mapping can be seen in Listing 4.9. However, it is important to point out that the manual mapping has to be done before the Tweets are being stored in Elasticsearch as types cannot be changed in retrospect.

[9]Geo-point: `https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-point.html`

Listing 4.9: Mapping of type geo-point

```
PUT visualization
{
    "mappings": {
        "tweets": {
            "properties": {
                "coordinates": {
                    "type": "geo_point"
                }
            }
        }
    }
}
```

4.4.5 Ranking Function

Listing 4.10 contains the implemented ranking function. It is based on Elasticsearch as the underlying technology and is implemented by using its Query DSL in JSON. The following example contains the query needed for listing the most relevant Tweets based on the terms ”Trump”, ”America”, and ”President”. The query is built on two components, `aggs` and `query`. The first component, `aggs`, is an abbreviation for aggregation. The aggregation framework in Elasticsearch allows the user to aggregate data based on a search query. There are different types of aggregations, but for this specific case, the *Bucketing* aggregation was used. The *Bucketing* aggregation builds buckets, where each bucket is associated with a *key* and a document criterion. The *key* in this case will be the `message`-field in each Tweet and the document criterion will be the max `_score`. The next component is `query` which is responsible for both the query and any defined `function_score`. For this case the `function_score` is defined as a *Gaussian* function with four parameters. The parameters are set to give a higher score to Tweets created during the last 24 hours and removing any penalties for all Tweets created within the last hour. The function will be performed at the `createdAt`-field. For a Tweet should be considered a match, at least one of the keywords needs to be found in the `message`-field.

The ranking function will be applied to all incoming Tweets at indexing time and will continuously be updated according to the decay function.

Listing 4.10: Query for finding most relevant Tweets based on the terms `Trump`, `America`, and `President`

```
{
  "aggs": {
    "top_k": {
      "aggs": {
        "max_score": {
          "max": {
            "script": "_score"
          }
        }
      },
      "terms": {
        "field": "message.keyword",
        "order": {
          "max_score": "desc"
        }
      }
    }
  },
  "query": {
    "function_score": {
      "functions": [
        {
          "gauss": {
            "createdAt": {
              "origin": "now",
              "scale": "24h",
              "offset": "1h",
              "decay": 0.5
            }
          }
        }
      ],
      "query": {
        "bool": {
          "minimum_should_match": 1,
          "should": [
            {
              "match_phrase": {
                "message": "America"
              }
            },
            {
              "match_phrase": {
                "message": "President"
              }
            },
            {
              "match_phrase": {
                "message": "Trump"
              }
            }
          ]
        }
      }
    }
  }
}
```

4.4.6 System Overview

Figure 4.7 provides an overview of the implemented system and the multiple steps presented in this chapter. The system starts by receiving the Tweets and user input. Furthermore, the Tweets are being processed and filtered in real-time, and the sentiment is predicted. Finally, the Tweets are made available by storing and visualizing them.

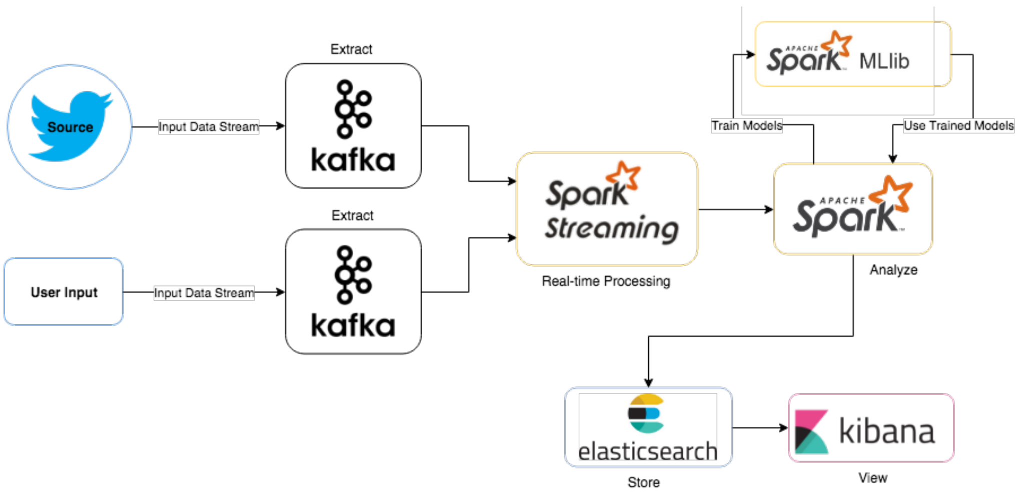


Figure 4.7: An overview of the implemented system

Chapter 5

Experiments and Results

This chapter will present the planning and setup used to perform the experiments. It will present the goal, along with the dataset and metrics used. A total of three experiments in addition to a final execution will be explained before presenting the results.

5.1 Goals with Experiments

The overall project goal presented in section 1.3 is to implement a solution that can visualize relevant information from Twitter based on user queries. It is essential that the implemented system has high performance and can handle a significant amount of data to facilitate near real-time decision making. The experiments should, therefore, be designed to measure how the system performs under increasing load. The system should also allow the user to insert queries based on interests. Hence, another part of the experiment should measure how performance is affected by increasing number of terms to filter the stream on. Lastly, it is of interest to retrieve a list of the most relevant Tweets based on the query. An investigation of how the ranking function described in section 4.4.5 performs will, therefore, be conducted.

As mentioned in section 4.2, the tasks for this project is to filter the incoming stream on relevant keywords and to classify the sentiment of each Tweet. These tasks have been implemented in the system and will be part of the experiments. However, the accuracy of neither the sentiment analysis nor the filtering is not a primary focus of this experiment. Hence, the output from both tasks will be used as a proof of concept, showing how streaming data can be transformed and analyzed and how the results can be used to give insights to the user.

5.2 Evaluation Methodology

5.2.1 Dataset

Training data: The classification model was trained with data retrieved from Sentiment140[1]. The dataset was a result of a research project at Stanford University and contains 1.600.000 Tweets together with its annotated sentiment, where there are equally many Tweets considered as positive and negative. The persons behind the project argue that it would be complicated to manually collect enough data to train a classification model for Tweets due to its wide range of topics. Instead, they used *distant supervision* which automatically annotates a large amount of data, without any human intervention, using emoticons as noisy labels. Meaning that a Tweet would be considered as positive if it contains a positive emoticon or negative if it contains a negative emoticon [59]. The training data from Sentiment140 contains the columns "polarity", "id", "date", "query", "user" and "status". Since the goal is to get the sentiment of the tweets, we are only interested in "polarity" and "status". These columns will be retained while discarding the rest.

Test data: As stated in section 5.1, the experiment should measure how the system handles increasing load of incoming data. A dataset with a size of 36GB and approximately 6.500.000 Tweets were collected using the Twitter Streaming API from April 15, 2018, to April 23, 2018. This dataset was used to control the input rate during the first two experiments. Furthermore, there is also a need to show how the sentiment of all incoming Tweets can be classified and visualized in Kibana with real-time data. Therefore, Tweets was received as explained in section 4.4.2 and nearly 835.000 Tweets were filtered by Spark and retrieved by Elasticsearch. The execution started May 16, 2018, and ended May 22, 2018. This dataset was also used for the third experiment.

Data Generator: To control the throughput of Tweets per second and simulate a Twitter stream, a `Data Generator` was used. The `Data Generator` is developed in Node.js[2] and allows the user to read the collected test data and output the Tweets to a socket in a configurable rate of Tweets per second. The `Data Generator` was initially implemented as part of an earlier Master Thesis made by Thor Martin Abrahamsen [60], and has been used

[1]Sentiment140: `http://help.sentiment140.com/home`
[2]`https://nodejs.org/en/`

with permission and some modifications. Link to the `Data Generator` can be found in Appendix C.

5.2.2 Evaluation Metrics

The number of processed Tweets per second and the processing time will be measured to evaluate the performance of the system. The experiment will be repeated with increasing load until the system is too congested to successfully process and store the incoming Tweets in a stable manner. Furthermore, the performance of the filtering is of interest as the number of search words is increasing. The execution time of the `filter`-function will, therefore, be measured. The last experiment will investigate how the proposed ranking function performs. The evaluation of the function will be based on the three requirements presented in section 4.3.3.

For the two initial experiments, a `Statistics` component has been implemented. The component will report the rate at which records are processed as well as execution time.

5.3 Experiments

The experiments will be divided into different parts to evaluate various aspects of the system. During the first part of the experiments, a load performance test will be executed to investigate how the implemented system handles increasing velocity of the incoming data. The next part will investigate how the system handles increasing number of terms to filter the stream on. The last experiment will explore how the ranking algorithm performs and thus rank the incoming Tweets based on their relevance. Lastly, the results will be used to perform the final execution of the system where the visualization will be created.

5.3.1 Test Setup

All experiments were executed in a small-scale environment running in local mode. The experiments were performed on a Dell OptiPlex 9020 with the following hardware:

- Intel® Core™ i7-4770 CPU @ 3.40GHz x 8
- 16 GB DDR3 1600 MHz RAM

5.3.2 Experiment 1: Test Load Performance

In the initial experiments, the velocity of the incoming data is of interest to see how the implemented system performs under increasing pressure. The purpose of the investigation is to examine how many Tweets/second the system can handle while still having stable performance.

To vary the incoming data rates, the `Data Generator` was used and connected to the Kafka Producer instead of the Twitter Streaming API. For a duration of ten minutes, a given rate of Tweets per second was ingested by the `Data Generator`. The various ingestion rates were set to 1000, 5000, 10000, 15000, and 20000 Tweets per second. Both the input rate from the Kafka Producer and the processing time after classifying and transforming the Tweets were reported to the `Statistics` component, to discover potential bottlenecks. The experiment was tested with a batch interval of 1 second. However, before anything was reported, each test started with a five minutes warm-up round.

An overview of the communication timeline can be seen in Figure 5.1.

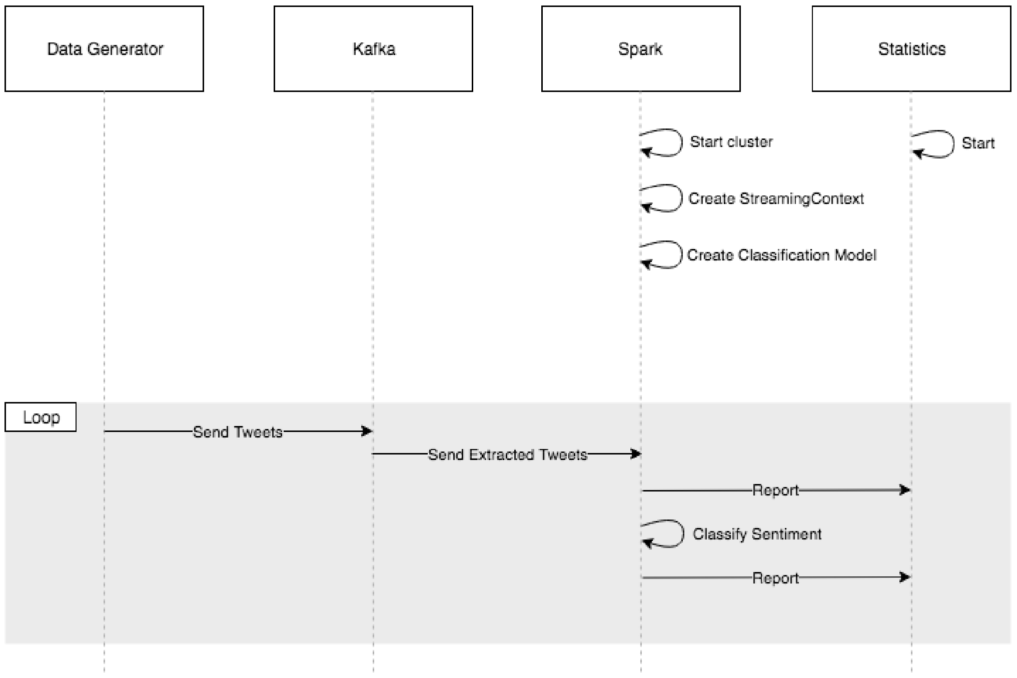


Figure 5.1: Communication timeline for testing load performance

5.3.3 Experiment 2: Increase Number of Keywords

As explained in the last chapter, the incoming Twitter stream will be filtered on a list of words. For each word the user is inserting, nine related words will be retrieved and written to a separate file in addition to the queried word. For every incoming micro-batch, Spark will read from the file and filter out those Tweets that contain any of the words in the file. To find out how the system performs with respect to the size of the list, an experiment was designed to measure the execution time of the `filter`-function as new words are inserted into the list.

To measure the execution time of the `filter`-function, a method called `time` was created as can be seen in Listing 5.1. The method is called as shown in Listing 5.2 and sends the execution time to the `Statistics` component before returning the result from the `filter`-function. Figure 5.2 shows the communication timeline for the conducted experiment. As can be seen in the Figure, the `Data Generator` is sending Tweets directly to Spark without using the Kafka Producer. The reason for that is simply that the only function of interest for this test is the `filter`-function. The Kafka Producer is saving all data written to a topic during the retention period, which will increase the pressure of the computer's resources. Hence, by not using the Kafka Producer, there will be less shared resources.

Listing 5.1: Method for measuring execution time

```scala
def time[R](block: => R): R = {
    val t0 = System.nanoTime()
    val result = block // call-by-name
    val t1 = System.nanoTime()
    val time = (t1 - t0) / 1000000.0
    println("Elapsed time: " + time + " ms")
    classification.setTime(time)
    result
  }
```

Listing 5.2: Method call for measuring execution time on `filter`-function

```scala
val filteredTweets = tweets.filter(status =>
                                   time { readFromFile().exists(status.contains(_)) })
```

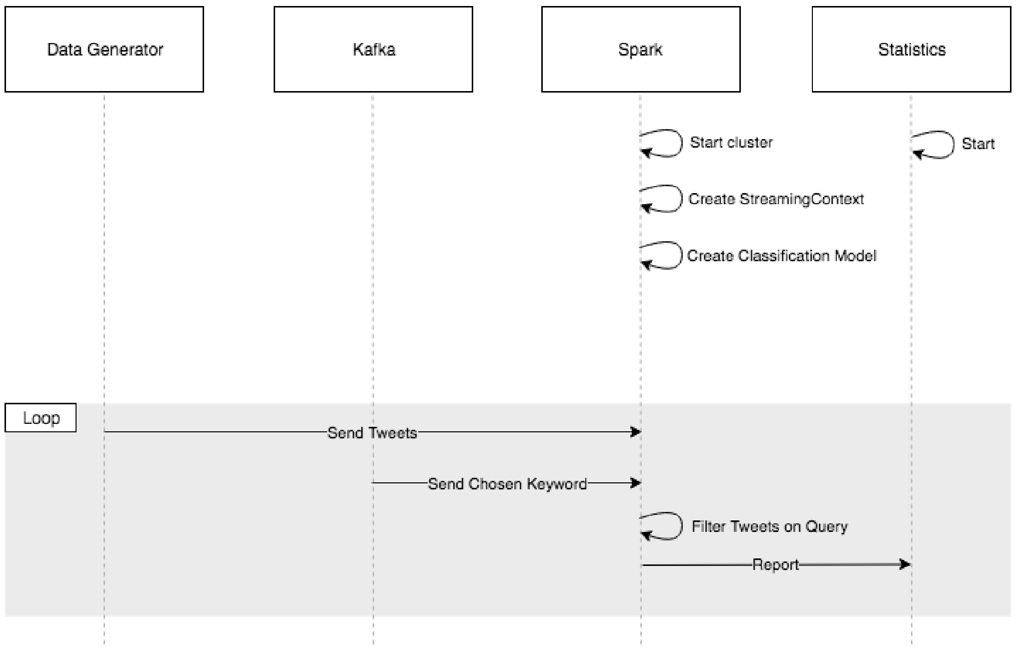


Figure 5.2: Communication timeline for measuring execution time of the `filter`-function

The test was executed eight times with different keywords: `Zlatan`, `Sweden`, `Basketball`, `Netflix`, `America`, `Obama`, and `Trump`. The words were chosen based on their various probabilities of generating matches. Where some words were expected to have few matches, and some were expected to have many matches. Each test started with a five minutes long warm-up round where no keywords were inserted. Then for every five minutes, the same chosen keyword was inserted, and the list of keywords was increased by ten words. The reason for inserting the same keyword for every round and thus increasing the list with the same ten words, was mainly due to keeping the number of matches as constant as possible. The goal of the experiment is to see how the execution time behaves when the size of the list increases. Unless any of the first ten words gives a match, neither will the same words do when they are repeated. Consequently, the function will read through the entire list. By executing each round for five minutes and calculate the average execution time, the result will compensate for any matches in the Twitter stream and reveal how the function behaves.

Lastly, another test was executed with the same time constraints. However, instead of using only one keyword, different keywords were inserted every five minutes. A total of 20 European countries were used as keywords. The list of words can be found in Listing B.3 in Appendix B.

A total of 6.300.000 Tweets was generated for each test by the `Data Generator`.

The average execution time, the size of the list and number of matches was reported.

5.3.4 Experiment 3: Rank Tweets based on Relevance

The filtering function in Spark can primarily be seen as a filter for removing spam and irrelevant information, in order to not store unnecessary data. However, there is also a need for finding the most relevant information. The goal of the proposed ranking function in section 4.4.5 was to create a list of the most relevant Tweets based on the user's interests which could be part of the final visualization. An experiment was conducted to test the result of the ranking function. The experiment applied the ranking function on every incoming Tweet in Elasticsearch, and it was evaluated both in regards to calculation of the relevance score and the resulting visualization.

Figure 5.3 shows an overview of the testing environment for the experiment.

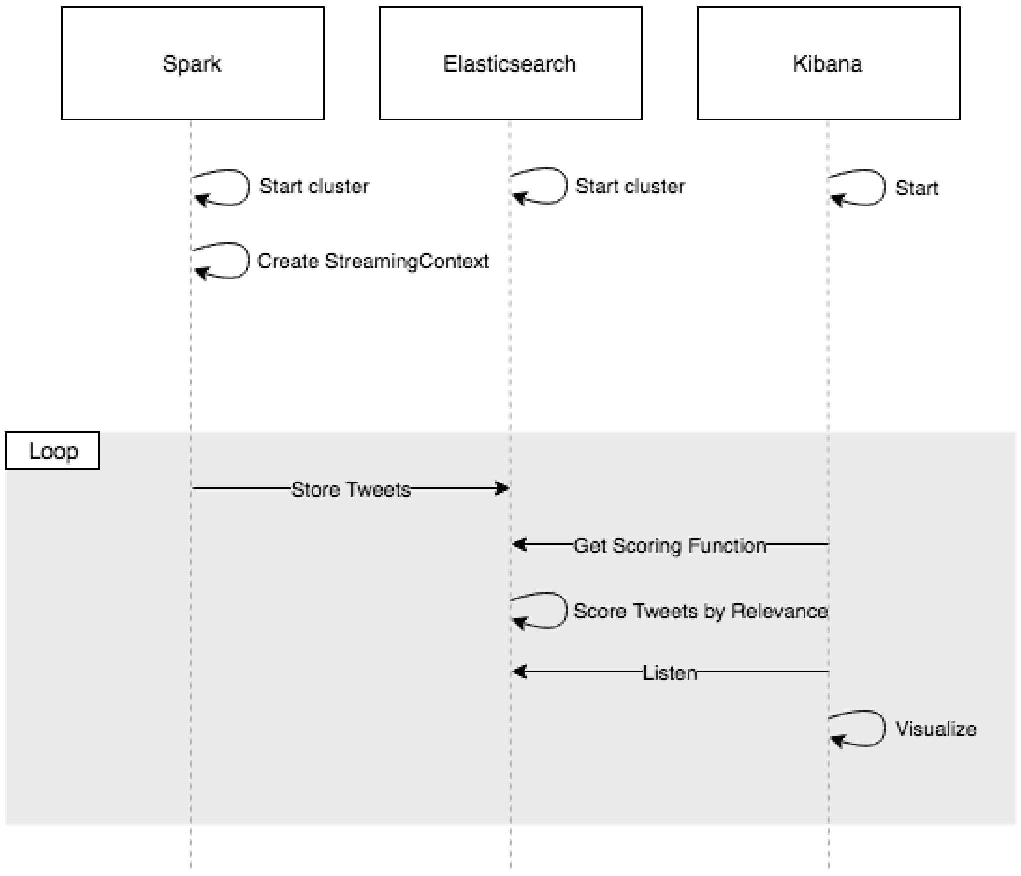


Figure 5.3: Testing environment for ranking Tweets based on relevance

5.3.5 Final Execution: Visualize Stream

To show how the implemented system explained in Chapter 4 works, a final execution was conducted. The execution started with a five minute warm-up round to allow the Kafka Producer and the Spark application to be started properly. After five minutes, three chosen keywords was inserted: `Trump`, `America` and `President`. The words were chosen mainly because of their high probability of generating matches.

It should be emphasized that during this execution, the Kafka Producer is retrieving Tweets from the Twitter Streaming API and not the `Data Generator`. On an average day, about 6000 Tweets are posted on Twitter per second. The Twitter Streaming API allows users to access a small portion (<=1%) of those. Furthermore, the stream received from the Twitter Streaming API has been filtered on Tweets written in English, which means that an even smaller portion of all Tweets will be retrieved. Consequently, the rate at which the system will be retrieving Tweets is a lot less compared to the first experiment. However, the goal of the final execution is to test the whole implemented pipeline. Which means that all parts described in Chapter 4 should be part of the execution, including retrieving Tweets from the Twitter Streaming API that has been generated in real-time.

An overview of the setup for the final execution can be seen in Figure 5.4.

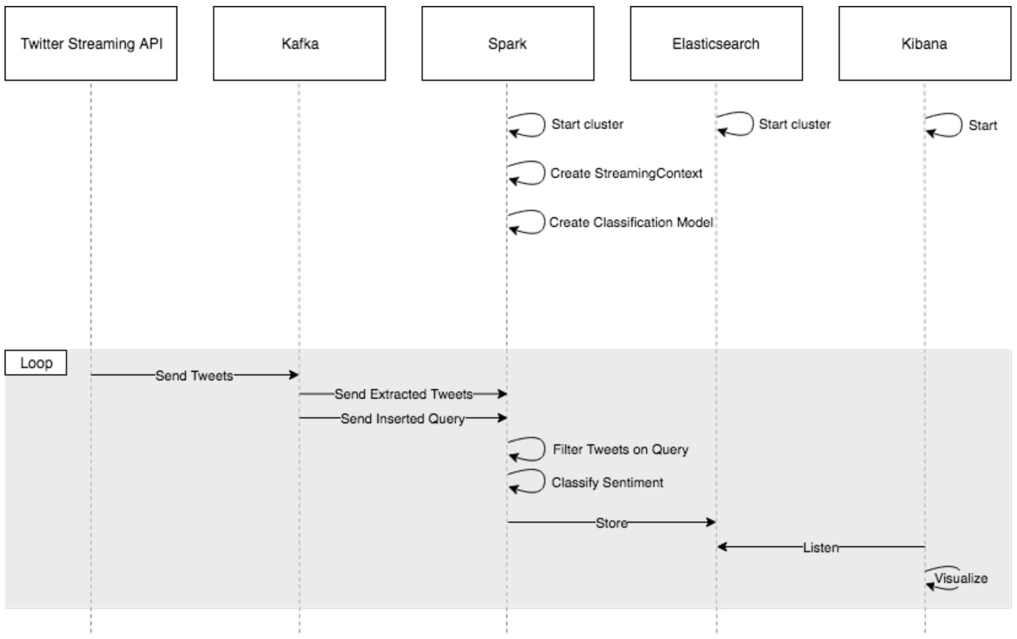


Figure 5.4: Setup for final execution

5.4 Results

This section presents the result of the described experiments.

5.4.1 Experiment 1: Test Load Performance

The following section will present each of the five tests. The input rate and processing time will be presented, along with some associated metrics.

Table 5.1: Metrics for 1000 Tweets/second

Average Input Rate:	1000 Tweets/second
Total Input Size:	600 000 Tweets
Average Processing Time:	201 ms

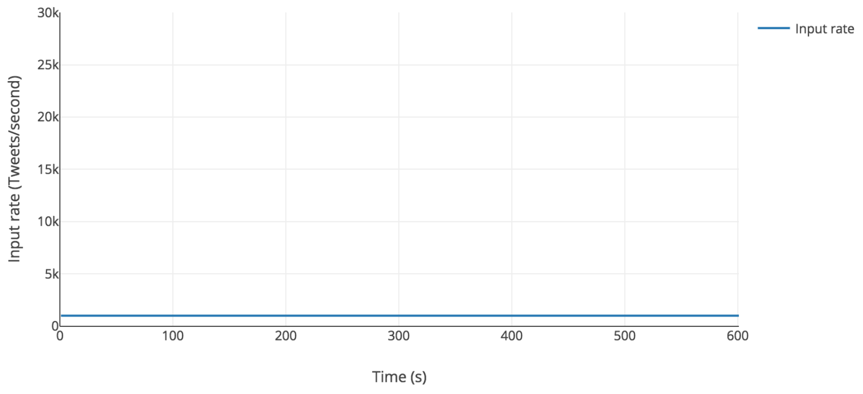


Figure 5.5: Classification of 1000 Tweets/second

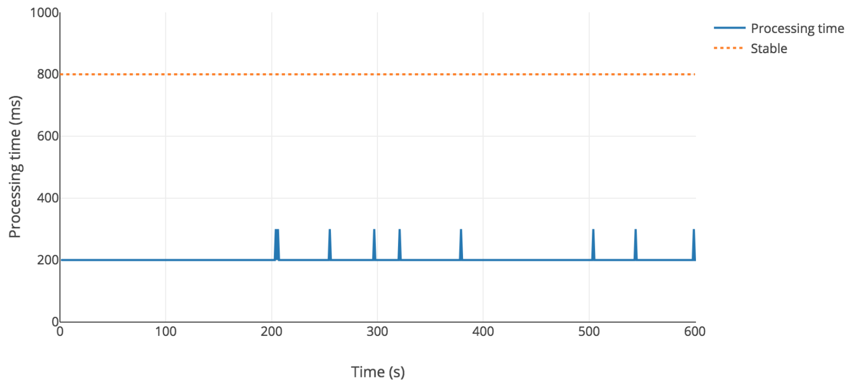


Figure 5.6: Processing time for classifying 1000 Tweets/second

Table 5.2: Metrics for 5000 Tweets/second

Average Input Rate:	4995 Tweets/second
Total Input Size:	3 000 000 Tweets
Average Processing Time:	207 ms

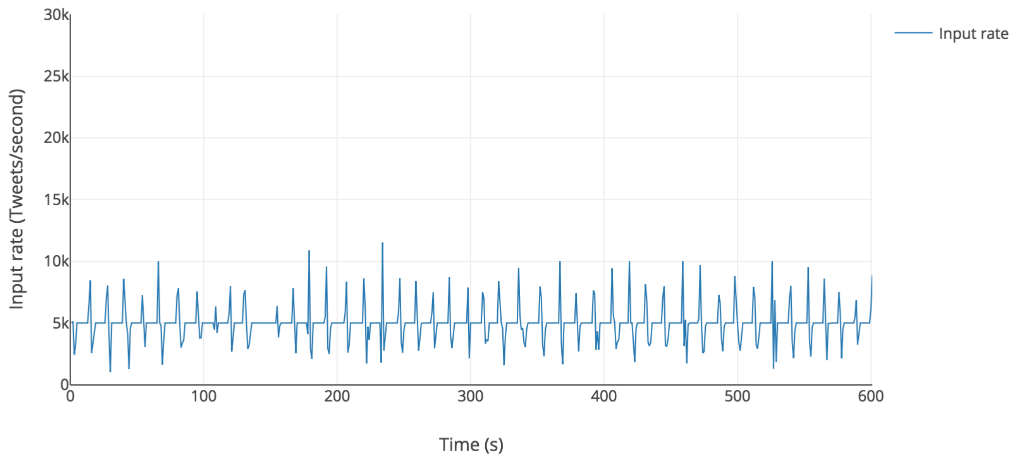


Figure 5.7: Classification of 5000 Tweets/second

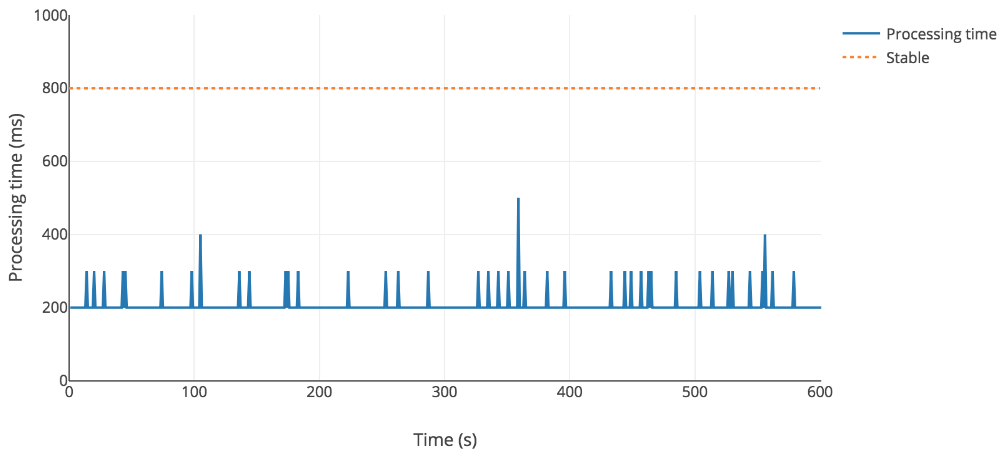


Figure 5.8: Processing time for classifying 5000 Tweets/second

Table 5.3: Metrics for 10000 Tweets/second

Average Input Rate:	9998 Tweets/second
Total Input Size:	6 000 000 Tweets
Average Processing Time:	226 ms

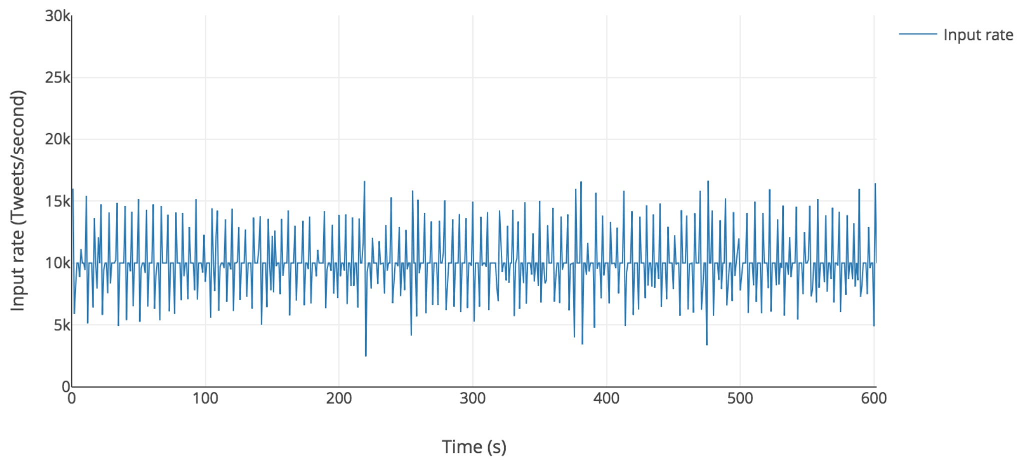


Figure 5.9: Classification of 10000 Tweets/second

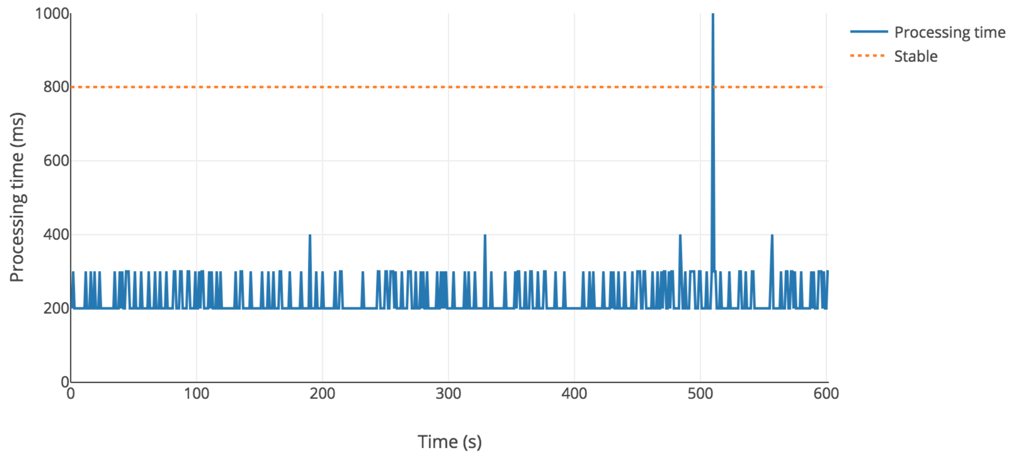


Figure 5.10: Processing time for classifying 10000 Tweets/second

Table 5.4: Metrics for 15000 Tweets/second

Average Input Rate:	14479 Tweets/second
Total Input Size:	9 000 000 Tweets
Average Processing Time:	231 ms

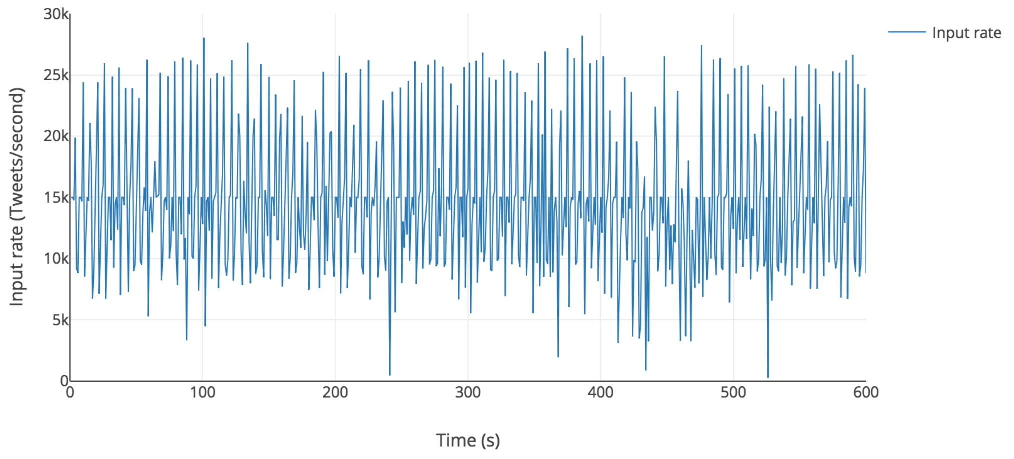


Figure 5.11: Classification of 15000 Tweets/second

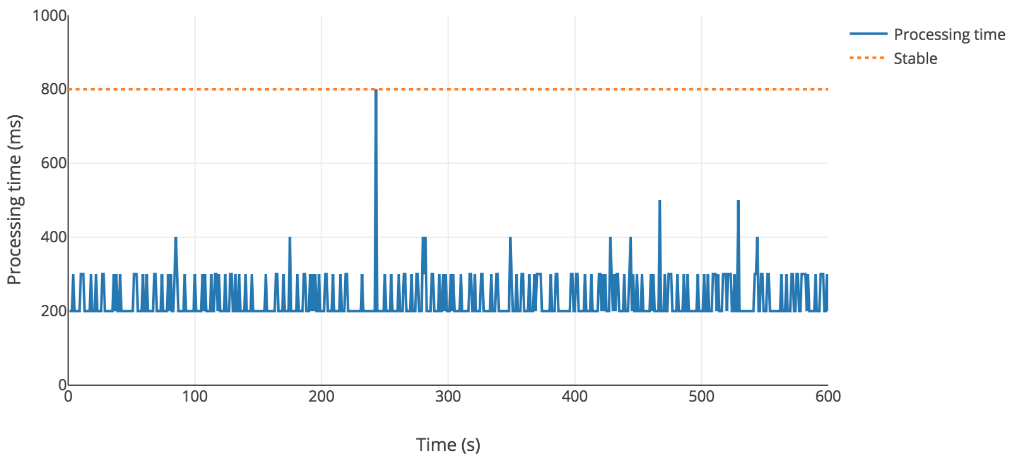


Figure 5.12: Processing time for classifying 15000 Tweets/second

Table 5.5: Metrics for 20000 Tweets/second

Average Input Rate:	14617 Tweets/second
Total Input Size:	12 000 000 Tweets
Average Processing Time:	240 ms

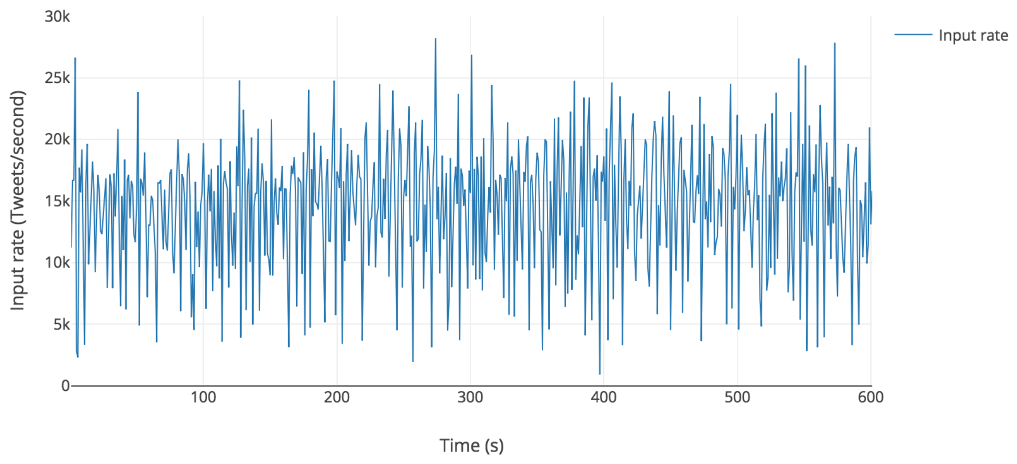


Figure 5.13: Classification of 20000 Tweets/second

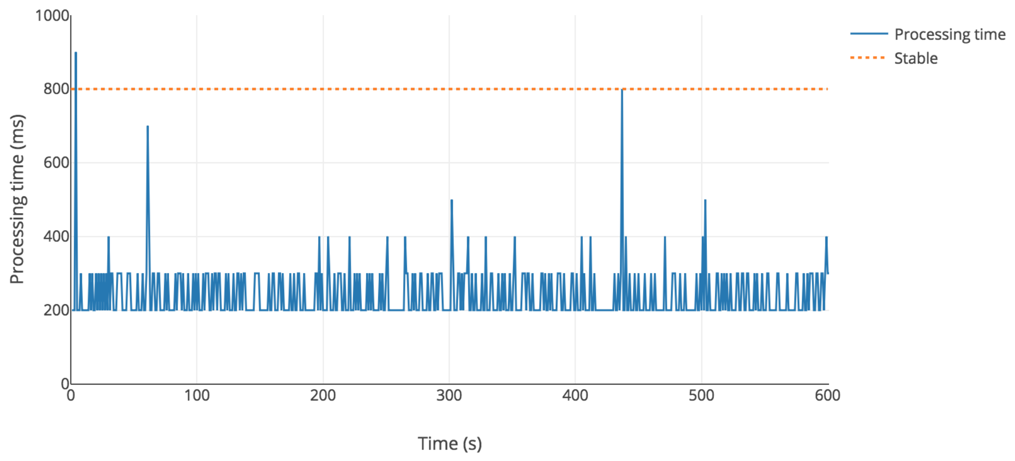


Figure 5.14: Processing time for classifying 20000 Tweets/second

5.4.2 Experiment 2: Increase Number of Keywords

Each of the tested keywords will be presented with regards to the execution time when the number of words is increasing. The number of matches is also presented along with the percentage of total matches.

Table 5.6: Metrics for keyword `Zlatan`

Number of matches:	589
Percentage of total:	0,000093%

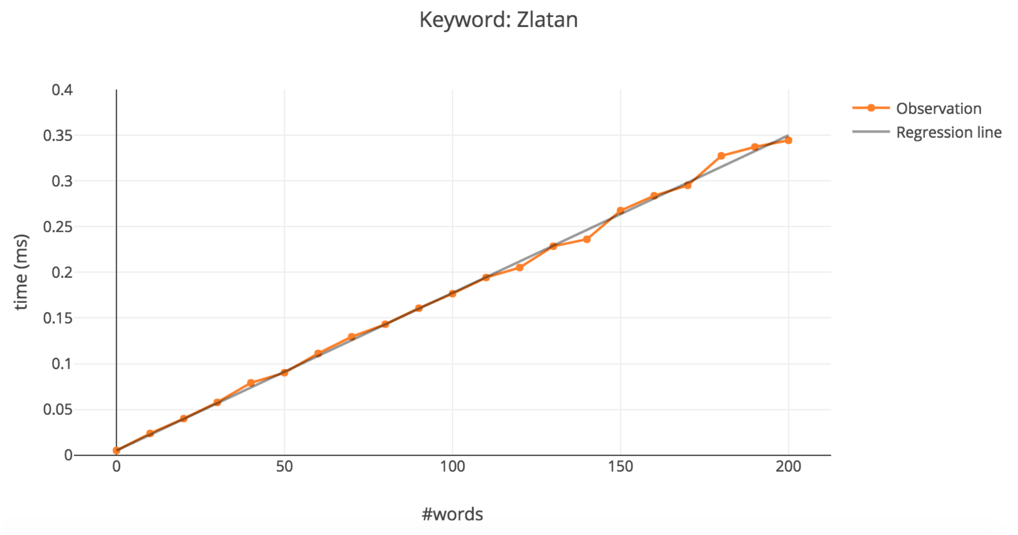


Figure 5.15: Keyword: `Zlatan`

Table 5.7: Metrics for keyword `Sweden`

Number of matches:	14918
Percentage of total:	0,0024%

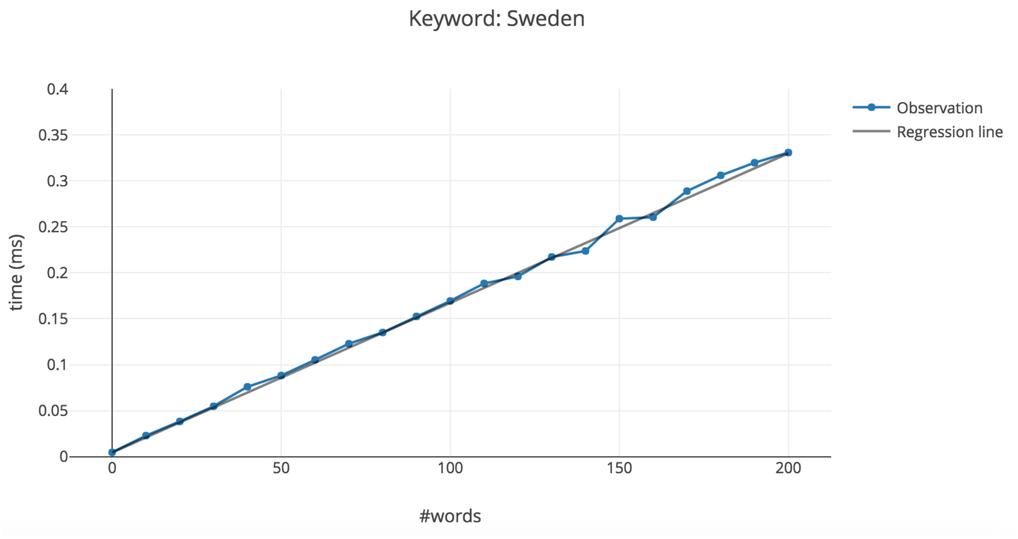


Figure 5.16: Keyword: `Sweden`

Table 5.8: Metrics for keyword `Basketball`

Number of matches:	14207
Percentage of total:	0,0023%

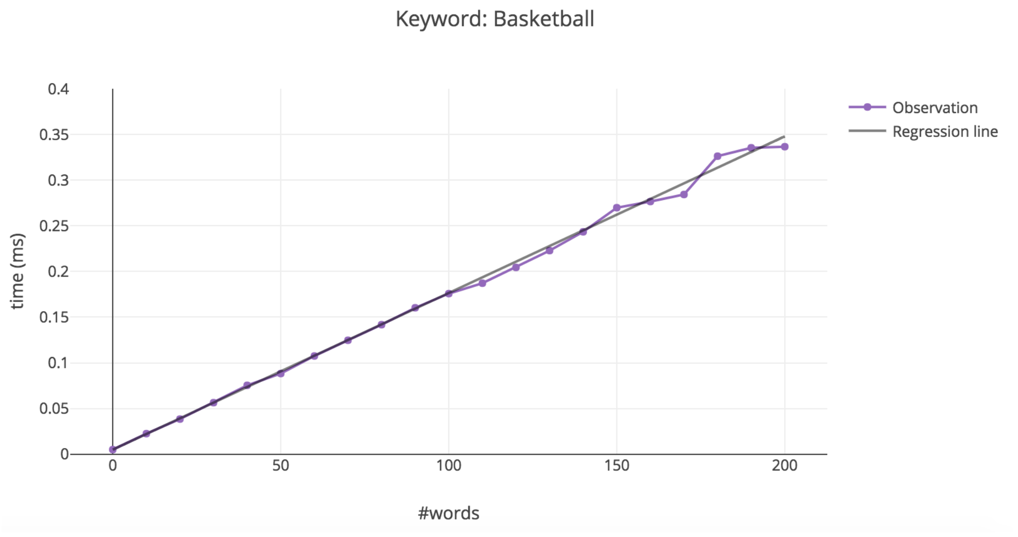


Figure 5.17: Keyword: `Basketball`

Table 5.9: Metrics for keyword `Obama`

Number of matches:	32513
Percentage of total:	0,0052%

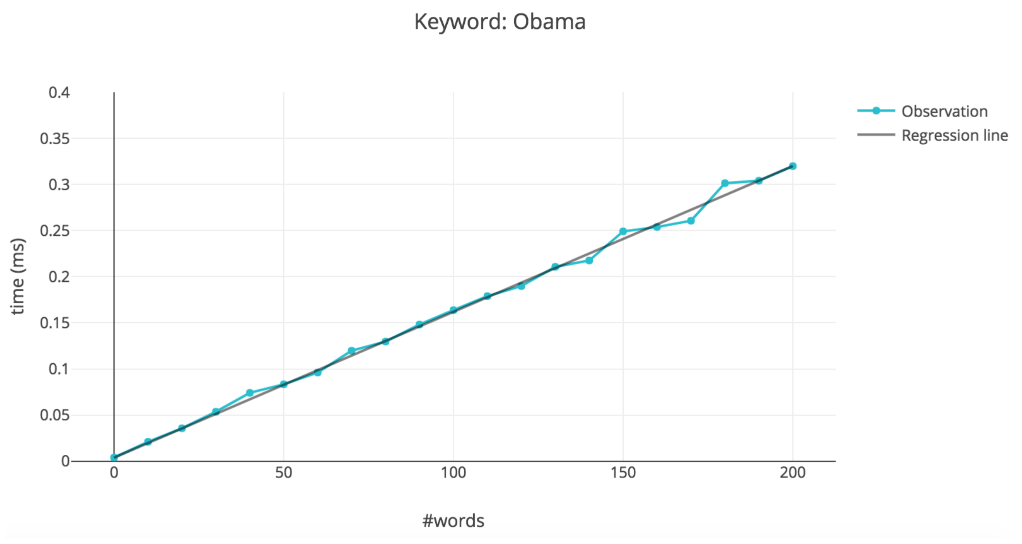


Figure 5.18: Keyword: `Obama`

Table 5.10: Metrics for keyword `Trump`

Number of matches:	264263
Percentage of total:	0,042%

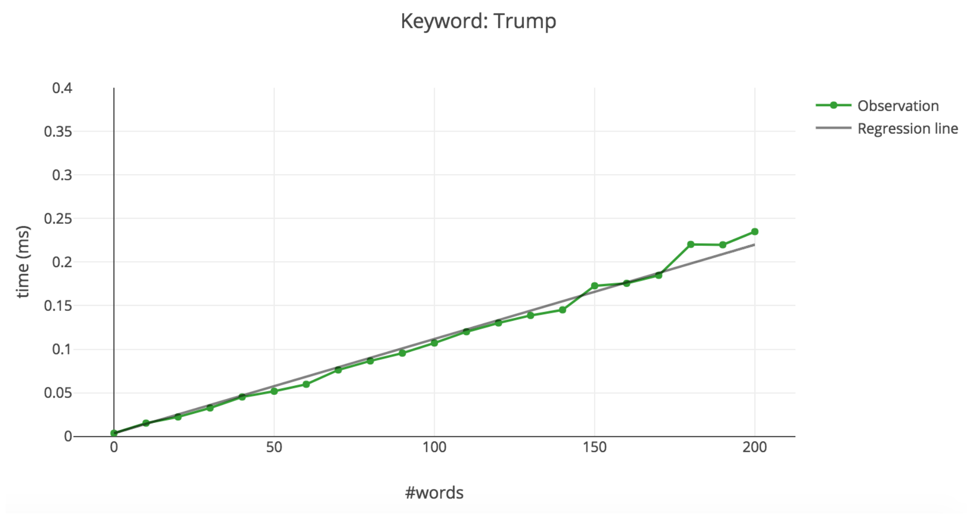


Figure 5.19: Keyword: `Trump`

Table 5.11: Metrics for keyword `Netflix`

Number of matches:	338164
Percentage of total:	0,054%

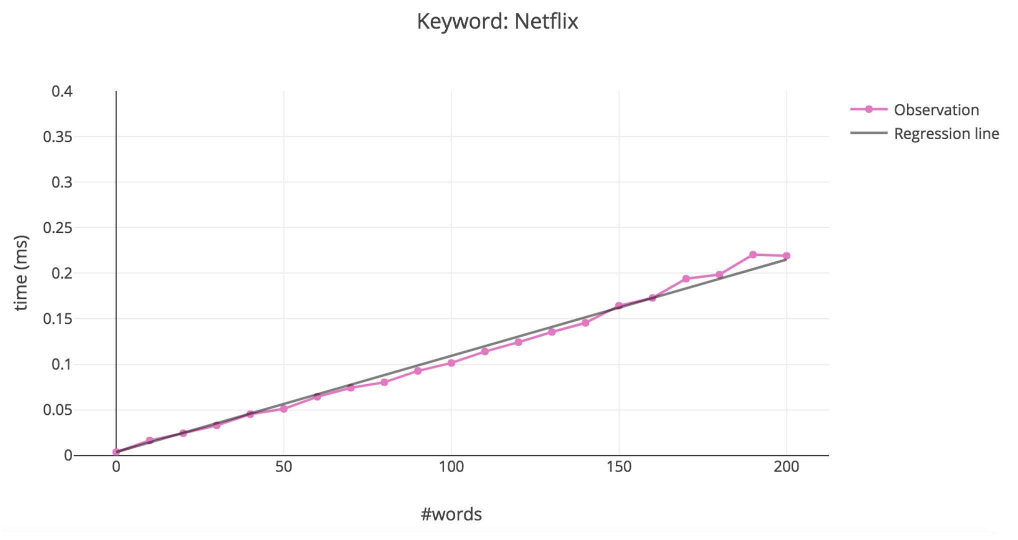


Figure 5.20: Keyword: `Netflix`

Table 5.12: Metrics for keyword `America`

Number of matches:	541177
Percentage of total:	0,086%

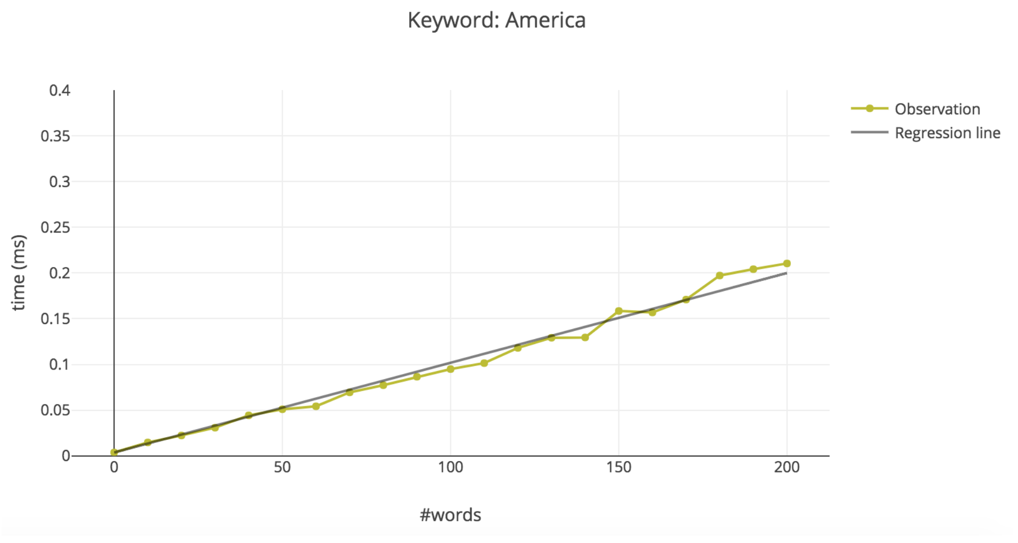


Figure 5.21: Keyword: `America`

Table 5.13: Metrics for keywords `European countries`

Number of matches:	284270
Percentage of total:	0,045%

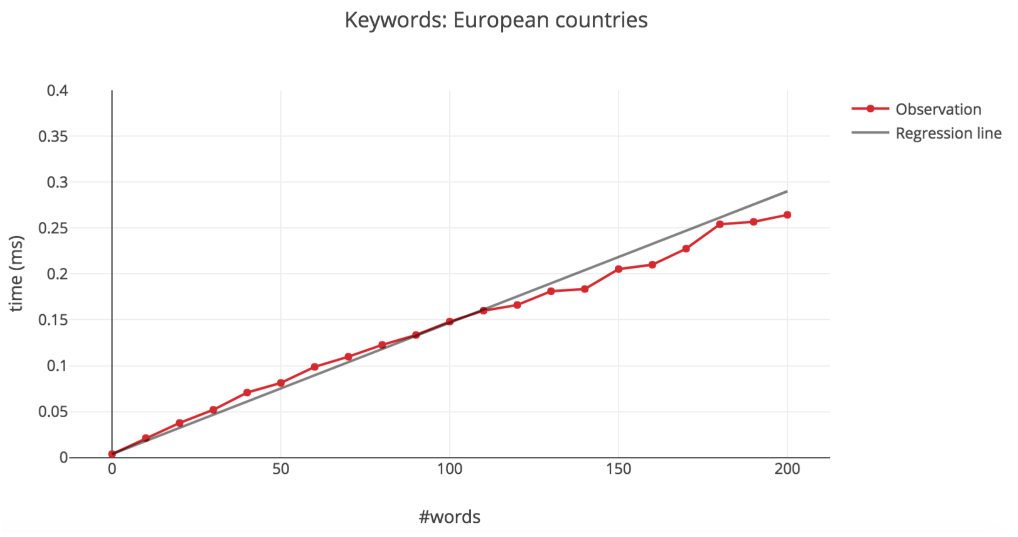


Figure 5.22: Keywords: `European countries`

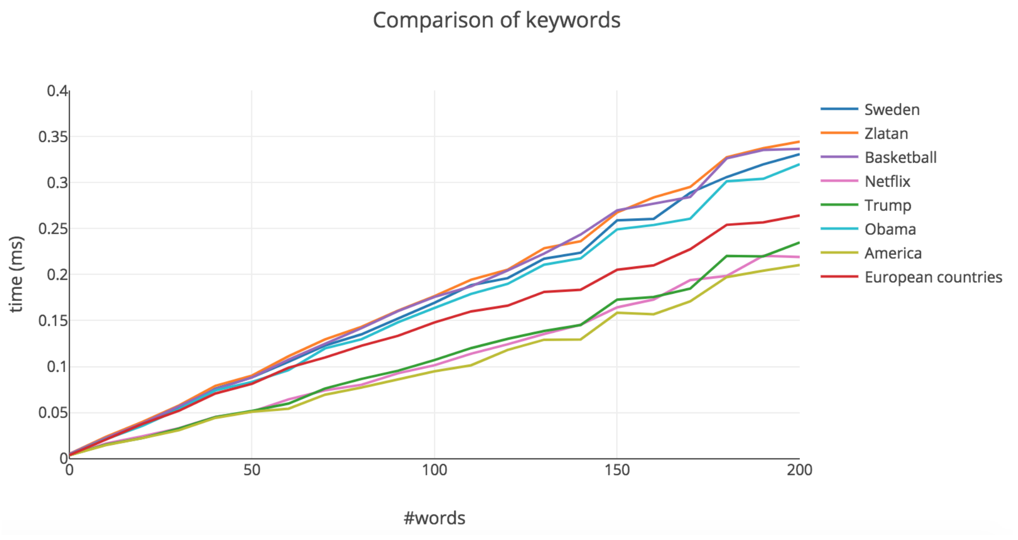


Figure 5.23: Comparison of all keywords

5.4.3 Experiment 3: Rank Tweets based on Relevance

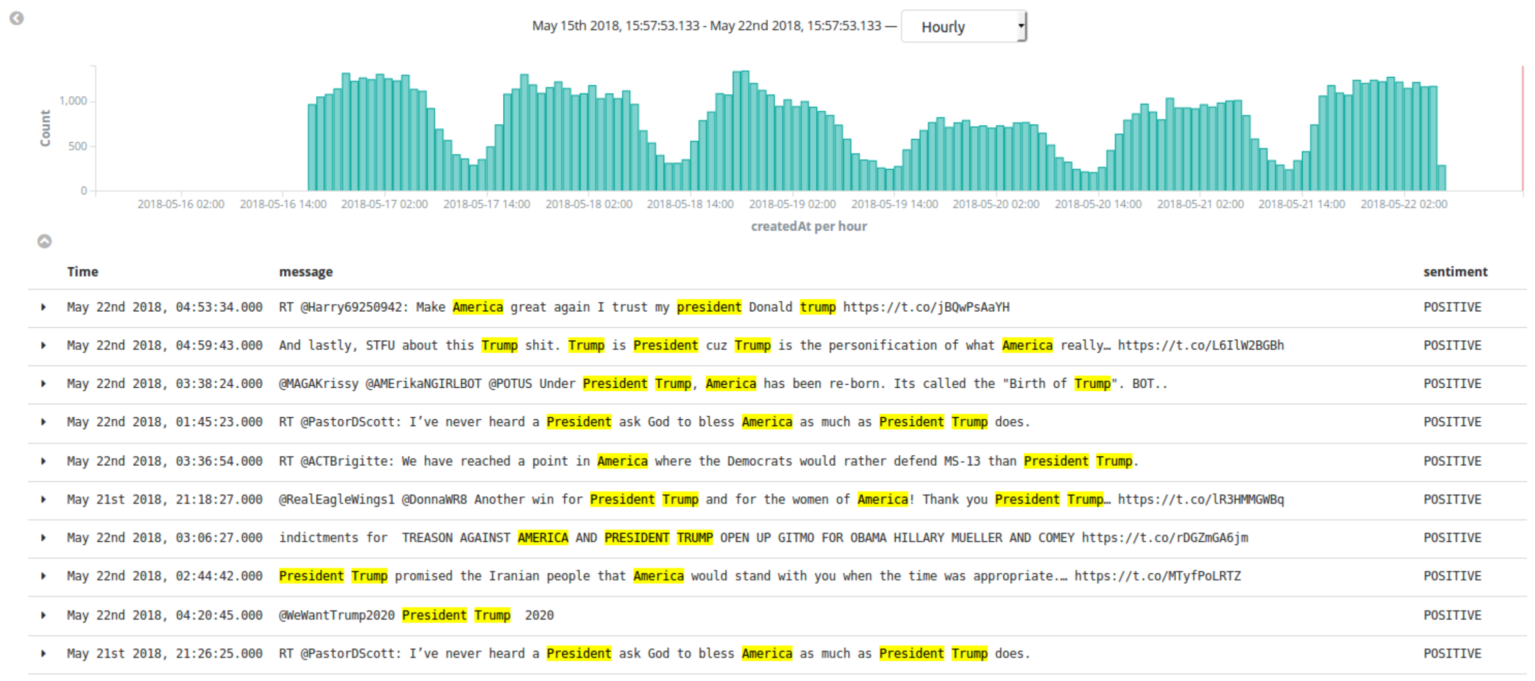
The query defined in Listing 4.10, will return the most relevant Tweets based on the terms `Trump`, `America`, and `President` by taking into account both TF-IDF and a decay function. An example of ten Tweets that have been ranked as the most relevant at a given time is shown in Listing 5.3.

Listing 5.3: Top-k results from the ranking function

```
{
  "top_k": {
       "doc_count_error_upper_bound": -1,
       "sum_other_doc_count": 111475,
       "buckets": [
         {
           "key": "RT @Harry69250942: Make America great again I trust my
           president Donald trump https://t.co/jBQwPsAaYH",
           "doc_count": 1,
           "max_score": {
             "value": 9.097686767578125
           }
         },
         {
           "key": "And lastly, STFU about this Trump shit. Trump is President cuz
           Trump is the personification of what America really...
           https://t.co/L6I1W2BGBh",
           "doc_count": 1,
           "max_score": {
             "value": 9.008153915405273
           }
         },
         {
           "key": "@MAGAKrissy @AMErikaNGIRLBOT @POTUS Under President Trump,
           America has been re-born. Its called the "Birth of Trump". ",
           "doc_count": 1,
           "max_score": {
             "value": 9.002861022949219
           }
         },
         {
           "key": "RT @PastorDScott: I've never heard a President ask God to bless
           America as much as President Trump does.",
           "doc_count": 4,
           "max_score": {
             "value": 8.791154861450195
           }
         },
         {
           "key": "RT @ACTBrigitte: We have reached a point in America where the
           Democrats would rather defend MS-13 than President Trump.",
           "doc_count": 3,
           "max_score": {
             "value": 7.863028526306152
           }
         },
         {
           "key": "@RealEagleWings1 @DonnaWR8 Another win for President Trump
           and for the women of America! Thank you President Trump...
           https://t.co/lR3HMMGWBq",
           "doc_count": 1,
            "max_score": {
              "value": 7.844945907592773
            }
          },
          {
            "key": "indictments for TREASON AGAINST AMERICA AND PRESIDENT
            TRUMP OPEN UP GITMO FOR OBAMA HILLARY MUELLER AND COMEY
            https://t.co/rDGZmGA6jm",
            "doc_count": 1,
            "max_score": {
              "value": 7.7633562088012695
            }
          },
          {
            "key": "President Trump promised the Iranian people that
            America would stand
            with you when the time was appropriate. https://t.co/MTyfPoLRTZ",
            "doc_count": 1,
            "max_score": {
              "value": 7.6242146492004395
            }
          },
          {
            "key": "@WeWantTrump2020 President Trump 2020",
            "doc_count": 1,
            "max_score": {
              "value": 7.538751602172852
            }
          },
          {
            "key": "@realDonaldTrump Thank you, President Trump!",
            "doc_count": 1,
            "max_score": {
              "value": 7.2856059074401855
            }
          }
        ]
      }
    }
  }
}
```

Listing B.4 in Appendix B explains how the relevance `_score` has been calculated for a given Tweet. The `_score` is the sum of the TF-IDF for every term in the query, where the term frequency has been normalized. Lastly, the `_score` is multiplied by the decay function which is shown at the end of the listing.

To fulfill the stated research goal, there is also a need to visualize the returned list of ranked items. The same Tweets shown in Listing 5.3 has been visualized in Figure 5.24. However, there is one difference. Listing 5.3 has aggregated the Tweets into buckets based on the `key`. Which means that retweeted messages go into the same bucket. For the visualization in Figure 5.24 no aggregation is done and the Tweets with the highest `_score` is shown. Hence, Tweet number four and Tweet number ten are two retweeted messages with the same content. For readability, the visualization can also be seen in Table 5.14.



Time	message	sentiment
May 22nd 2018, 04:53:34.000	RT @Harry69250942: Make America great again I trust my president Donald trump https://t.co/jBQwPsAaYH	POSITIVE
May 22nd 2018, 04:59:43.000	And lastly, STFU about this Trump shit. Trump is President cuz Trump is the personification of what America really… https://t.co/L6IlW2BGBh	POSITIVE
May 22nd 2018, 03:38:24.000	@MAGAKrissy @AMErikaNGIRLBOT @POTUS Under President Trump, America has been re-born. Its called the "Birth of Trump". BOT..	POSITIVE
May 22nd 2018, 01:45:23.000	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE
May 22nd 2018, 03:36:54.000	RT @ACTBrigitte: We have reached a point in America where the Democrats would rather defend MS-13 than President Trump.	POSITIVE
May 21st 2018, 21:18:27.000	@RealEagleWings1 @DonnaWR8 Another win for President Trump and for the women of America! Thank you President Trump… https://t.co/lR3HMMGWBq	POSITIVE
May 22nd 2018, 03:06:27.000	indictments for TREASON AGAINST AMERICA AND PRESIDENT TRUMP OPEN UP GITMO FOR OBAMA HILLARY MUELLER AND COMEY https://t.co/rDGZmGA6jm	POSITIVE
May 22nd 2018, 02:44:42.000	President Trump promised the Iranian people that America would stand with you when the time was appropriate.… https://t.co/MTyfPoLRTZ	POSITIVE
May 22nd 2018, 04:20:45.000	@WeWantTrump2020 President Trump 2020	POSITIVE
May 21st 2018, 21:26:25.000	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE

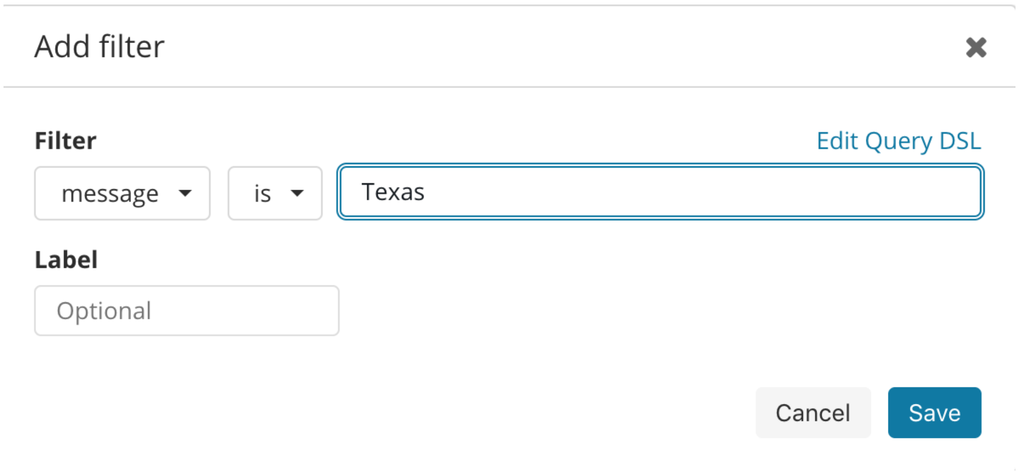
Figure 5.24: Ranked list of most relevant Tweets

Table 5.14: Ranked list of most relevant Tweets

Time	**Message**	**Sentiment**
May 22nd 2018, 04:53:34.000	RT @Harry69250942: Make America great again I trust my president Donald trump https://t.co/jBQwPsAaYH	POSITIVE
May 22nd 2018, 04:59:43.000	And lastly, STFU about this Trump shit. Trump is President cuz Trump is the personification of what America really... https://t.co/L6IlW2BGBh	POSITIVE
May 22nd 2018, 03:38:24.000	@MAGAKrissy @AMErikaNGIRLBOT @POTUS Under President Trump, America has been re-born. Its called the "Birth of Trump". BOT..	POSITIVE
May 22nd 2018, 01:45:23.000	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE
May 22nd 2018, 03:36:54.000	RT @ACTBrigitte: We have reached a point in America where the Democrats would rather defend MS-13 than President Trump.	POSITIVE
May 21st 2018, 21:18:27.000	@RealEagleWings1 @DonnaWR8 Another win for President Trump and for the women of America! Thank you President Trump... https://t.co/lR3HMMGWBq	POSITIVE
May 22nd 2018, 03:06:27.000	indictments for TREASON AGAINST AMERICA AND PRESIDENT TRUMP OPEN UP GITMO FOR OBAMA HILLARY MUELLER AND COMEY https://t.co/rDGZmGA6jm	POSITIVE
May 22nd 2018, 02:44:42.000	President Trump promised the Iranian people that America would stand with you when the time was appropriate. https://t.co/MTyfPoLRTZ	POSITIVE
May 22nd 2018, 04:20:45.000	@WeWantTrump2020 President Trump 2020	POSITIVE
May 21st 2018, 21:26.25.000	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE

The ranking function has been based on the same keyword that is inserted in the filtering step in Spark. However, it is still possible to add other keywords that might be of interest to the user. While this experiment was performed, a school shooting was conducted at a high school in Santa Fe, Texas. By simply adding the keyword `Texas` in the Kibana user interface as shown in Figure 5.25, the filter and thus the ranked list is updated.

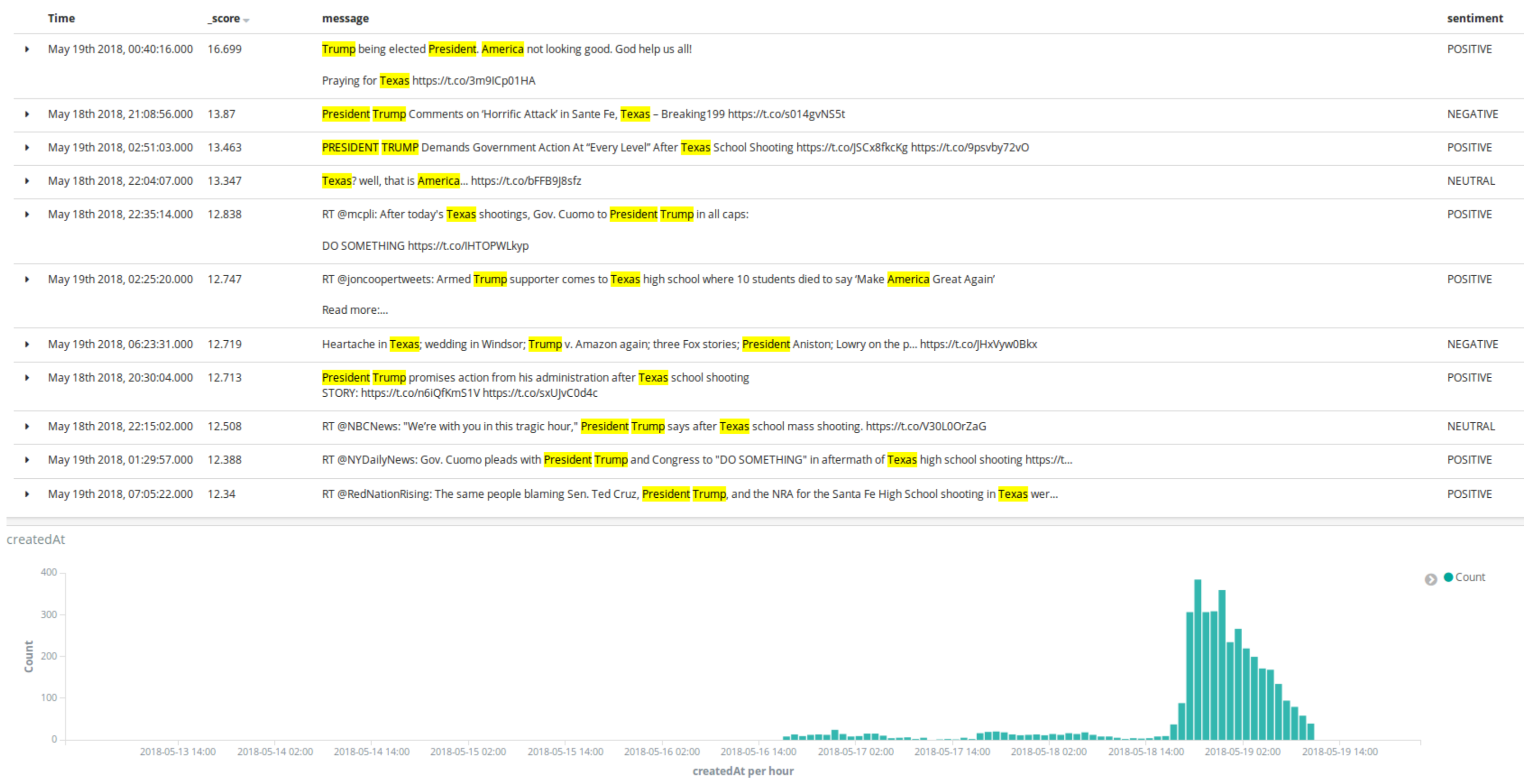
The updated list can be seen in Figure 5.26, where the number of created Tweets related to the keywords indicates that a breaking news event has occurred. For readability, the list can also be seen in Table 5.15


Add filter

Filter: message is Texas

Label: Optional

Figure 5.25: Adding filter on keyword `Texas`



	Time	_score	message	sentiment
▸	May 19th 2018, 00:40:16.000	16.699	Trump being elected President. America not looking good. God help us all! Praying for Texas https://t.co/3m9ICp01HA	POSITIVE
▸	May 18th 2018, 21:08:56.000	13.87	President Trump Comments on 'Horrific Attack' in Sante Fe, Texas – Breaking199 https://t.co/s014gvNS5t	NEGATIVE
▸	May 19th 2018, 02:51:03.000	13.463	PRESIDENT TRUMP Demands Government Action At "Every Level" After Texas School Shooting https://t.co/JSCx8fkcKg https://t.co/9psvby72vO	POSITIVE
▸	May 18th 2018, 22:04:07.000	13.347	Texas? well, that is America... https://t.co/bFFB9J8sfz	NEUTRAL
▸	May 18th 2018, 22:35:14.000	12.838	RT @mcpli: After today's Texas shootings, Gov. Cuomo to President Trump in all caps: DO SOMETHING https://t.co/lHTOPWLkyp	POSITIVE
▸	May 19th 2018, 02:25:20.000	12.747	RT @joncoopertweets: Armed Trump supporter comes to Texas high school where 10 students died to say 'Make America Great Again' Read more:...	POSITIVE
▸	May 19th 2018, 06:23:31.000	12.719	Heartache in Texas; wedding in Windsor; Trump v. Amazon again; three Fox stories; President Aniston; Lowry on the p... https://t.co/JHxVyw0Bkx	NEGATIVE
▸	May 18th 2018, 20:30:04.000	12.713	President Trump promises action from his administration after Texas school shooting STORY: https://t.co/n6iQfKmS1V https://t.co/sxUJvC0d4c	POSITIVE
▸	May 18th 2018, 22:15:02.000	12.508	RT @NBCNews: "We're with you in this tragic hour," President Trump says after Texas school mass shooting. https://t.co/V30L0OrZaG	NEUTRAL
▸	May 19th 2018, 01:29:57.000	12.388	RT @NYDailyNews: Gov. Cuomo pleads with President Trump and Congress to "DO SOMETHING" in aftermath of Texas high school shooting https://t...	POSITIVE
▸	May 19th 2018, 07:05:22.000	12.34	RT @RedNationRising: The same people blaming Sen. Ted Cruz, President Trump, and the NRA for the Santa Fe High School shooting in Texas wer...	POSITIVE

Figure 5.26: Ranked list with Tweets based on the keywords Trump, America, President, and Texas

Table 5.15: Ranked list with Tweets based on the keywords `Trump`, `America`, `President`, and `Texas`

Time	Message	Sentiment
May 19th 2018, 00:40:16.000	Trump being elected President. America not looking good. God help us all! Praying for Texas https://t.co/3m9lCp01HA	POSITIVE
May 18th 2018, 21:08:56.000	President Trump Comments on 'Horrific Attack' in Santa Fe, Texas - Breaking199 https://t.co/s014gvNS5t	NEGATIVE
May 19th 2018, 02:51:03.000	PRESIDENT TRUMP Demands Government Action At "Every Level" After Texas School Shooting https://t.co/JSCx8fkcKg https://t.co/9psvby72vO	POSITIVE
May 18th 2018, 22:04:07.000	Texas? well, that is America... https://t.co/bFFB9J8sfz	NEUTRAL
May 18th 2018, 22:35:14.000	RT @mcpli: After today's Texas shootings, Gov. Cuomo to President Trump in all caps: DO SOMETHING https://t.co/IHTOPWLkyp	POSITIVE
May 19th 2018, 02:25:20.000	RT @joncoopertweets: Armed Trump supporter comes to Texas high school where 10 students died to say 'Make America Great Again' Read more: ...	POSITIVE
May 19th 2018, 06:23:31.000	Heartache in Texas; wedding in Windsor; Trump v. Amazon again; three Fox stories; President Aniston; Lowry on the p... https://t.co/JHxVyw0Bkx	NEGATIVE
May 18th 2018, 20:30:04.000	President Trump promises action from his administration after Texas school shooting STORY: https://t.co/n6iQfKmS1V https://t.co/sxUJvC0d4c	POSITIVE
May 18th 2018, 22:15:02.000	RT @NBCNews: "We're with you in this tragic hour," President Trump says after Texas school mass shooting. https://t.co/V30L0OrZaG	NEUTRAL
May 19th 2018, 01:29:57.000	RT @NYDailyNews: Gov. Cuomo pleads with President Trump and Congress to "DO SOMETHING" in aftermath of Texas high school shooting https://t...	POSITIVE
May 19th 2018, 07:05:22.000	RT @RedNationRising: The same people blaming Sen. Ted Cruz, President Trump, and the NRA for Santa Fe High School shooting in Texas wer...	POSITIVE

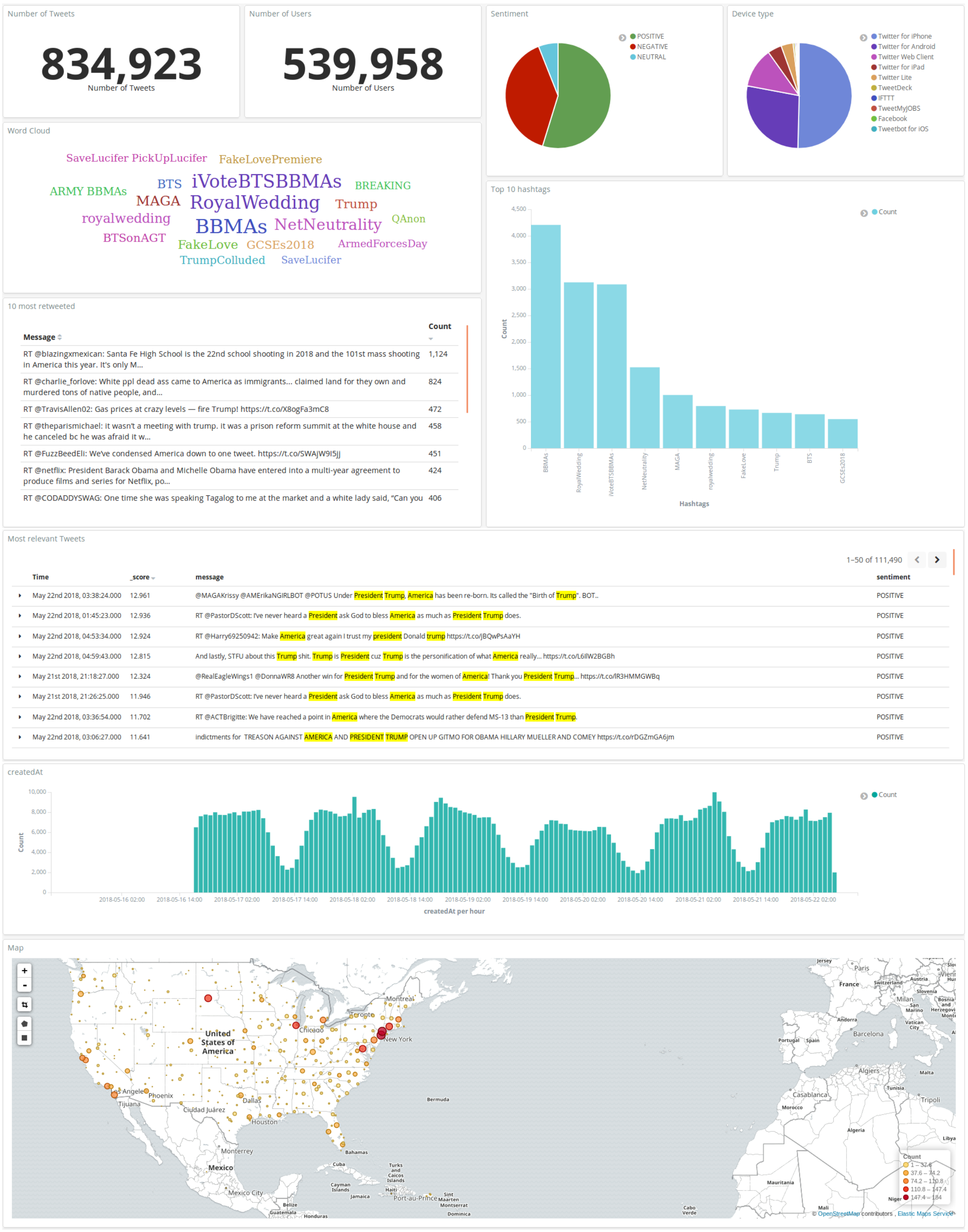
5.4.4 Final Execution: Visualize Stream

Figure 5.27 shows the dashboard created after running the system for almost a week. The dashboard includes the following visualizations:

Visualization type	Description
Metric	Number of retrieved Tweets
Metric	Number of distinct users
Pie	Predicted sentiment
Pie	Distribution of used device type
Tag Cloud	Cloud showing the most trending hashtags
Data Table	Table showing the Tweets that are most re-tweeted
Vertical Bar	Top 10 most popular hashtags
Saved Search	Ranked list of the most relevant Tweets
Vertical Bar	Timeline of when Tweets was created
Coordinate Map	Global map showing lat-long pairs

Table 5.16: Visualization in Kibana

A close-up of the visualizations can be found in Appendix A.



10 most retweeted

Message	Count
RT @blazingxmexican: Santa Fe High School is the 22nd school shooting in 2018 and the 101st mass shooting in America this year. It's only M...	1,124
RT @charlie_forlove: White ppl dead ass came to America as immigrants... claimed land for they own and murdered tons of native people, and...	824
RT @TravisAllen02: Gas prices at crazy levels — fire Trump! https://t.co/X8ogFa3mC8	472
RT @theparismichael: it wasn't a meeting with trump. it was a prison reform summit at the white house and he canceled bc he was afraid it w...	458
RT @FuzzBeedEli: We've condensed America down to one tweet. https://t.co/SWAjW9I5jj	451
RT @netflix: President Barack Obama and Michelle Obama have entered into a multi-year agreement to produce films and series for Netflix, po...	424
RT @CODADDYSWAG: One time she was speaking Tagalog to me at the market and a white lady said, "Can you	406

Most relevant Tweets

1–50 of 111,490

Time	_score	message	sentiment
May 22nd 2018, 03:38:24.000	12.961	@MAGAKrissy @AMErikaNGIRLBOT @POTUS Under President Trump, America has been re-born. Its called the "Birth of Trump". BOT..	POSITIVE
May 22nd 2018, 01:45:23.000	12.936	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE
May 22nd 2018, 04:53:34.000	12.924	RT @Harry69250942: Make America great again I trust my president Donald trump https://t.co/jBQwPsAaYH	POSITIVE
May 22nd 2018, 04:59:43.000	12.815	And lastly, STFU about this Trump shit. Trump is President cuz Trump is the personification of what America really... https://t.co/L6IlW2BGBh	POSITIVE
May 21st 2018, 21:18:27.000	12.324	@RealEagleWings1 @DonnaWR8 Another win for President Trump and for the women of America! Thank you President Trump... https://t.co/lR3HMMGWBq	POSITIVE
May 21st 2018, 21:26:25.000	11.946	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE
May 22nd 2018, 03:36:54.000	11.702	RT @ACTBrigitte: We have reached a point in America where the Democrats would rather defend MS-13 than President Trump.	POSITIVE
May 22nd 2018, 03:06:27.000	11.641	indictments for TREASON AGAINST AMERICA AND PRESIDENT TRUMP OPEN UP GITMO FOR OBAMA HILLARY MUELLER AND COMEY https://t.co/rDGZmGA6jm	POSITIVE

Figure 5.27: Dashboard created in Kibana

Chapter 6

Evaluation and Discussion

This chapter will begin with an evaluation and discussion of the experiments and results presented in Chapter 5. Furthermore, a discussion of the system and the project as a whole will be given with reflections upon the findings.

6.1 Evaluation of Results

This section will evaluate the results presented in the previous chapter. Each experiment will be evaluated individually.

6.1.1 Experiment 1: Test Load Performance

Section 5.4.1 shows the results of the first experiment where the goal was to investigate how the system behaves under increasing pressure. The experiment was repeated five times with various input rates. According to Databricks[1], the company founded by the creators of Spark, each batch should be processed within 80% of the batch interval [61]. If the average processing time is closer or greater than the batch interval, the application will start queuing up the incoming records in a backlog and add latencies. For the conducted experiment, the batch interval was set to 1 second, meaning that each batch should be processed within 800 ms. Figure 5.6, 5.8, 5.10, 5.12, and 5.14 shows the processing time for the various input rates. The average processing time is slightly increasing from 201 ms to 240 ms for the first test compared to the last. However, all results are by far within the boundaries of what is considered as stable, and every record

[1]Databricks: `https://databricks.com/`

can be processed in near real-time. In addition to stable processing time, none of the tests introduced any scheduling delay.

Except for the result in Figure 5.5, the classification rates in Figure 5.7, 5.9, 5.11, and 5.13 might appear rather unstable. A possible reason for the varying input rate may be due to a bug in the Kafka Spark integration. The `group.id` that was set in Listing 4.2 is a string that uniquely identifies the group of consumer processes in which this consumer belong. If all the consumer instances have the same consumer group, which is the case here, then the records will effectively be load balanced over the consumer instances. If they have different consumer groups, then each record will be broadcast to all the consumer processes. The Kafka Consumer API used in Spark pre-fetches records into buffers. To keep performance high, Spark wants to keep cached consumers on the executors instead of recreating them for each batch, which is the preferable procedure. According to Spark the cache is keyed by `topicpartition` and `group.id`. However, due to a bug described in the issue SPARK-19185[2], this does not seem to work as intended. It is possible to disable caching for Kafka consumers. However, when disabling the cache, a workaround is needed to avoid the bug. According to Spark, the workaround solution is to use a separate `group.id` for each call to `createDirectStream` and thus go against the purpose of the `group.id`. Once the bug is resolved, caching can be used as it was supposed to and the input rate will probably flow more evenly. However, by looking at the average input rate, the first four tests show that they are almost consistent with what was generated by the `Data Generator`. Although the last test was set to 20000 Tweets/second, it did only achieve 14617 Tweets/second on average which indicates that the maximum rate that the implemented system can handle is $\sim$15000 Tweets/second.

The average throughput can be compared to other related work such as the feasibility analysis of AsterixDB and Spark Streaming + Cassandra [41]. Like this project, the study was performing sentiment analysis on a stream of Tweets. The result of the study showed that Spark Streaming by itself was able to process $\sim$7000 Tweets/second and when integrated with Cassandra the throughput was $\sim$1000 Tweets/second [41]. However, the classification task was performed by reading the records from Cassandra, whereas for this experiment, the records were classified at arrival time. This approach is suggested as further work by the author. Another difference between the study and this experiment is the used version of Spark. Whereas this project has used version 2.2.1, the feasibility analysis used version 1.3.1. Consequently, there has probably been made improvements to Spark Streaming between the versions. Other related work is the benchmarking investigation, where a pipeline consisting of Kafka and Spark

[2]Spark bug: `https://issues.apache.org/jira/browse/SPARK-19185`

Streaming were tested [43]. The pipeline was constructed to simulate an advertisement analytics pipeline, and some operations such as filtering of irrelevant events, projection and join were performed. The test was performed with throughput varied from 50000 to 170000 events/second and showed that Spark Streaming was able to achieve high throughput with some tuning of the batch interval. However, this result was performed with ten instances of Kafka producers and between 25 to 30 nodes. The authors point out that the reason for using ten instances of Kafka is that individual producers begin to fall behind at around 17000 events/second, which is in good agreement with the results for this project.

Considering the domain for this specific project, the goal is to visualize data generated from Twitter. As stated earlier, the average number of generated Tweets is 6000 Tweets/second. Hence, a throughput of 15000 Tweets/second should be regarded as approved.

6.1.2 Experiment 2: Increase Number of Keywords

Section 5.4.2 shows the result for the second experiment. The goal of the experiment was to investigate how the execution time of the `filter`-function behaves as the list of words increases. As seen in all results, the time it takes to filter increases as new words are inserted, which is expected behavior. However, what is interesting is the rate at which the execution time increases. For all observations, a regression line has been added. Although none of the observations are perfectly straight, they are all following the regression line to some extent and should be considered as linear. Hence, the time complexity for the `filter`-function is $\mathcal{O}(n)$.

However, even if all observations are linear, they are increasing at different rates. Figure 5.23 shows a comparison between the different keywords. As can be seen in the figure, the keywords `Sweden`, `Zlatan`, `Basketball`, and `Obama` performed similarly to each other. This as opposed to the keywords `Trump`, `Netflix`, and `America` who have a less steep increase in execution time. The observation with 20 different European countries is intermediate between the other two groups. The most logical reason for the various steepness in execution time is depending on the number of matches for each of the keywords. The way the `filter`-function works is that it returns a boolean depending on whether a Tweet contains any of the words in the list. If there is a match, the function will stop reading the list and return `true`. The group of observations that had a steeper gradient had between 0,000093% and 0,0052% matches, whereas the other group had between 0,042% and 0,086% matches. This indicates that the function had to read through the whole list of words more often for the first group of observations compared to the other group which will affect the average execution time.

The observation in Figure 5.22 was somewhat different from the other observations. Whereas the others were tested with the same keyword, this observation was tested with 20 keywords consisting of different European countries. The overall execution time could be considered as increasing linearly compared to the regression line. However, between 100 and 200 inserted words, the average execution time seems to be decreasing. This behavior makes sense if compared to the hypothesis that the execution time will increase slower if the list of words has a high probability of generating matches.

6.1.3 Experiment 3: Rank Tweets based on Relevance

The results of the experiment present how a customized scoring function has been implemented by taking advantage of the built-in features in Elasticsearch. Listing 5.3 shows the returned list with the ten most relevant Tweets corresponding to the executed query presented in Listing 4.10.

The function was implemented with respect to the predefined requirements in section 4.3.3. First of all, the ranked list should be based on both historical and real-time data. This requirement has been satisfied by using Elasticsearch as the fundamental technology. As all Tweets are indexed and stored in the established cluster, the ranking function can update the relevance score for each Tweet and combine it with the score calculated for newly arrived Tweets. As the function is based on both posting time and TF-IDF, the ranking list will continuously be updated to provide the most relevant Tweets at any time. The second requirement said that recent posted Tweets should be considered as more relevant than old ones. This requirement has been fulfilled by defining a decay function in the query. The decay function was set to give higher score to Tweets created during the last 24 hours and removing any penalties for all Tweets created within the last hour. An example of how the decay function has been applied can be seen in Figure 5.24 or Table 5.14. Tweet number four and ten are both retweeted messages with the exactly same content. Hence, their TF-IDF score will be the same. However, the time of creation is what is of interest here. One Tweet was created May 22nd 2018, 01:45:23, and the other May 21st 2018, 21:26:25. By applying the decay function, the newest Tweet will be placed higher up in the ranking list. The last requirement concerns the query, and that the ranking function should be based on keywords determined by the user. The implemented function is based on the boolean model, where at least one of the stated keywords needs to be matched. The three keywords used for this experiment, is the same keywords that were used for filtering the dataset in Spark during the final execution. Hence, the keywords used has been determined by the user. Yet this solution is not considered as optimal. A query can easily be created in Kibana by utilizing the user interface as shown in Figure 5.25. This will allow the user to extend the query at any time and is useful for trend detection as shown in the Texas example. However, to create the query defined in Listing 4.10, some coding knowledge is required. The current solution requires that the keywords are inserted in both Kafka and Elasticsearch. Optimally, the query should be created automatically by reading the list of words that has been inserted by the user in Kafka to increase usability. Another disadvantage with the query is that it only checks matches on the `message`-field. This is presumably the most valuable field, but there is possibilities that more important information is to be found in other fields such as the `userDescription`-field.

6.1.4 Final Execution: Visualize Stream

The resulting dashboard in Figure 5.27 can be compared with the result from the specialization project shown in Figure A.11 in Appendix A. For the former project, no filtering was done prior to the Tweets arrival in Elasticsearch. This resulted in a dashboard containing Tweets written from all over the world in 92 different languages. For the current project, the Tweets were filtered on keywords, and Kafka retrieved only those written in English from the Twitter Streaming API. Not surprisingly, this resulted in less number of retrieved Tweets and a great majority of those were from the United States. Except for the number of retrieved Tweets, the biggest difference between the two dashboards is the level of relevance. By looking at the new dashboard, it is clear that the data has been filtered, where both hashtags and messages are related to the inserted keywords.

As mentioned in section 3.3, some related projects exists within the domain of visualizing Twitter data. The Taghreed system was focusing on geo-tagged microblogs whereas Cloudberry was built as a more generic solution that can utilize Big Data with various attributes. The interface of both systems is mainly built as a map with additional features such as explicit Twitter messages and timelines. The appearance of the proposed system is somewhat different compared to the others. The dashboard does also include a heat map. However, that is not the central object of the dashboard. The goal of the proposed dashboard has been to show both general information such as the number of users and a timeline, but also more explicit information such as the ranked list of relevant Tweets and the most popular hashtags. However, the imagination of the user is the limit when creating the dashboard, and the result in Figure 5.27 is just an example of how it can be done. If the user would like more information about any feature, he or she can in an intuitive and user-friendly way add queries directly in the interface of Kibana. As the proposed system is built on an existing search engine, Elasticsearch is responsible for returning the relevant information and update the dashboard. Hence, compared to Taghreed and Cloudberry whom both have implemented their own query engines to update the visualizations upon requests from the user, the proposed solution is depending on the Elastic stack to deal with requested queries. By looking at Figure 5.27, it is clear that what earlier needed both database experts and someone that could interpret the results, can now be performed by anyone. Filters in Kibana can both be defined as a full-text search as shown in Figure 5.25, or by clicking on any of the created visualizations. It is also possible to combine different filters to gain more insights into the data. This is similar to Cloudberry which allows their users to drill down to more fine-grained information or type in keywords to focus on a particular topic. It is hard to tell which of the systems that perform best with regards to the underlying query engines, as this project has not been focusing on the performance of Elasticsearch. However, there exists multiple use cases of how the Elastic stack has been used for application search[3].

Another aspect that had to be considered was the task of choosing which keywords to use. The keywords should have a high probability of generating matches, but at the same time exclude spam and irrelevant information. Some parallels can be drawn to the measurements, *precision* and *recall*, presented in section 2.5.1. The purpose of the system is to get as many relevant tweets as possible, that is, a high quantity of Tweets with high quality. Three keywords were used for the final execution. The keywords `Trump` and `President` were chosen based on what is trending on Twitter right now and `America` was chosen based on its high probability of getting matches. However, this is indeed a balancing act, and it might be that the choice of using `America` was too general and gave much information that was not of interest. The keyword `America` also generated some other general words in its list of related words, such as `United States`. Which means that every Twitter user that has declared the United States as his or her home country will be a match.

[3]Use cases: `https://www.elastic.co/use-cases?usecase=application-search`

6.2 Discussion

This section will give a final discussion of the system and reflections upon the findings.

The proposed solution can be compared with the BAD system presented in section 3.1. The BAD system is designed to deliver data of interest to users from a variety of data sources. By combining real-time and historical data and allowing the users to subscribe for specific themes, the system can collect data and provide it for the users in a seamless way. When formulating the motivation behind the proposed solution, the BAD system was used as an inspiration. However, instead of using the Publish/Subscribe-pattern, this project looked into how to solve the same idea with continuous queries. The BAD system was built by extending AsterixDB, whereas the proposed system used Spark as the underlying analytics engine. The advantages with AsterixDB compared to Spark is that it allows fast data ingestion, scalable query executions in addition to distributed storage, whereas Spark is strictly a processing engine and does not persist data outside of the current execution. However, this has not been considered as a problem. To index and store the retrieved Tweets, parts of the Elastic stack was used. The Elastic stack does not only allow indexing and storing of data, but it also provides various search capabilities and the possibility of visualizing it. These features have been of great value for this project and the composed system of Kafka + Spark + Elasticsearch + Kibana has shown to provide the needed functionality to fulfill the project goal. As mentioned, the BAD system uses a Publish/Subscribe-pattern which means that users can subscribe to a topic, and whenever new information regarding that topic is found, the user is notified. This is somehow different from how the proposed system works. The solution uses continuous queries to filter the incoming stream based on the interests of the user. This means that all records that match the query will be stored in Elasticsearch and be made available for the user. Hence, the user will not be explicitly notified when new records arrive, but the visualizations will be updated automatically. This approach is considered to be appropriate for the proposed system as the visualization is supposed to give a high-level representation of all information regarding the defined query, whereas the BAD system will return more specific information such as explicit Tweets regarding a topic. For example, if the user is interested in getting information about news-related issues that may affect travel from the airport in Los Angeles (LAX), he or she might subscribe for themes regarding ”bomb” and ”LAX”. The BAD system would then notify the user whenever new Tweets have been written with those words, whereas the proposed system would give an overview of all Tweets containing those words. Hence, the proposed system is more appropriate for use cases where the goal is detection of trends, and the BAD system for detection of explicit information. However, if the user would like more detailed information such as specific Tweets, it is possible to include that in the visualization as shown in Figure 5.27.

Another aspect of Spark that had to be considered was the use of batch intervals. As Spark is using micro-batches to perform transformations on streams of data, a batch interval had to be defined. As the goal of the project was to visualize Twitter data in near real-time, the batch interval was set to 1 second. It should be noticed that Spark Streaming's smallest batching window is 500 ms, which means that if a particular use case requires results to be generated at even higher resolution, it will not be able to deliver. An example of such use case is when stream processing is used for monitoring a manufacturing process that must be able to react within milliseconds on change of conditions such as humidity, pressure, and temperature. In such cases, too much latency can be a significant threat to the entire monitoring infrastructure. However, the latency requirements for this application are not that strict, and Spark Streaming was considered as a suitable streaming processing engine for the task.

Chapter 7

Conclusion and Future Work

This chapter gives a conclusion to the work done in this thesis. Furthermore, the chapter will include pointers to interesting directions for future research and improvements to the system.

7.1 Conclusion

This section will present the fulfillment of the stated research questions as well as a conclusion of the project goal.

7.1.1 Research Questions

To conclude whether this project has fulfilled its goal, each research question should be answered individually.

RQ 1: How can Tweets be detected in real-time based on a user-defined continuous query?

This was considered to be the primary research question for this project as it would try to answer how the main parts of the system should be implemented. The proposed solution was based on the ETL process and consisted of a pipeline of Kafka, Spark, and parts of the Elastic stack. There are many advantages to let Kafka handle the data extraction. Since Kafka is reliable, scalable, and efficient, it is an appropriate tool for building data pipelines. As mentioned in section 2.6.2, Kafka can continuously receive Tweets and keep them stored for a given amount of time. Consequently, the implemented system will be more fault-tolerant as downtime in the Spark application will not lead to any data loss as every record is persisted in

Kafka. Spark was used for the data transformation as it is a unified engine that can handle both streaming data and machine learning. The last part of the pipeline are parts of the Elastic stack consisting of Elasticsearch and Kibana. Elasticsearch should mainly be considered as a search and analytics engine. However, for this project, it has both been used as a search engine, but also as a storage medium where all retrieved Tweets are being indexed and stored. As all components of the pipeline can scale horizontally to hundreds of nodes, its potential has not been fully utilized during this project as it has been used in a small-scale environment running in local mode.

To perform continuous queries, it was decided to use Kafka to receive user input. The chosen method did fulfill the requirements for this project as it should be considered as a proof of concept. However, this part of the system has a lot of potentials. For example, a user interface should be implemented to make the query insertion more intuitive for the user. The user should also be able to change the query by adding and deleting keywords as the need arises. Another improvement that should be considered is the use of a file as intermediate storage of the inserted query. If the system is being scaled to multiple nodes, other options that are more suitable for distributed systems should be investigated.

The proposed system shows how Tweets can be detected in real-time and how relevant Tweets can be filtered based on a user-defined continuous query. Hence, the research question is considered to be fulfilled.

RQ 2: How can a Tweet be classified as relevant relative to a query?

To answer this question, the definition of relevance for this specific project had to be defined. As the domain for this project was Twitter, relevance was defined to be the textual similarity between a user's interest and the content of a Tweet, and thus a keyword-based retrieval strategy was chosen as the feature set for the query. Based on the CCR task mentioned in section 3.4, it was decided to use strict string matching as filtration approach and add some known name variants to extend the query. The Google Knowledge Search API was used to extend the search vocabulary. By referring to Figure 4.4, it is clear that the API returns some phrases that might not be optimal to use as keywords and the probability that any Tweet will match phrases like `Donald Trump 2017 presidential inauguration` is considered as low. The idea of extending the vocabulary with known name variants has many potentials. However, regarding the nature and structure of Tweets, the name variant should probably be of a shorter length to increase the probability of generating matches in the Twitter stream. As mentioned in section 5.1, the goal of the filtering task was never to investigate how accurate the filter function is with regards to precision and recall. Instead, it was mainly used as a proof of concept, showing how a Tweet can be classified as relevant based on a continuous query.

RQ 3: How can Tweets be ranked based on relevance?

Based on the definition of relevance, a ranking function was implemented as shown in section 4.10 by utilizing the built-in functions of Elasticsearch. An experiment was conducted to investigate the results of the function, both in regards to calculation of the relevance score and the resulting visualization. The result showed that the relevance score was calculated by taking the sum of the TF-IDF for every term in the query multiplied by the defined decay function. The function was found successful as two Tweets with the same content were given a different score of relevance based on the time the Tweets were posted. However, a drawback of the implemented function is that it requires some coding knowledge from the user and that it is not able to read from the list of words that have been inserted by the user in Kafka. The proposed ranking function is also quite naive as it is calculating and updating the score for every Tweet continuously. The question that should be considered for future work is whether all Tweets needs to be ranked or if it is enough to maintain a small list with only the top-k most relevant Tweets.

Even if the proposed ranking function has room for improvement, it is based on both new, streaming data and historical, stored data. It is also updating the ranking list continuously whenever new matches arrive and thus the research question is considered to be answered.

RQ 4: How does the implemented system perform with increasing amount of queries?

To answer this question, an experiment was conducted to test how the execution time of the `filter`-function behaved as the number of search words increases. The result from the experiment showed that the time complexity for the function is $\mathcal{O}(n)$ which was expected behavior.

RQ 5: How does the implemented system perform with increasing amount of data?

Another experiment was conducted to answer this question. The first experiment investigated how the system behaves under increasing pressure. By experimenting with various input rates, it was shown that the processing time should be considered as stable for all tests. However, the maximum throughput that the system can handle is $\sim$15000 Tweets/second.

7.1.2 Project Goal

The project has been an attempt to investigate the intersection between information retrieval and processing of streaming data by implementing a solution that could visualize relevant content from Twitter based on user-defined queries. The goal of the project was stated as:

To examine how a system can be implemented, which can visualize relevant information from Twitter based on user-defined queries and provide valuable insights for the user.

The final execution presented in section 5.4.4 shows how Twitter data can be visualized in near real-time. By using continuous queries determined by the user, relevant content has been filtered from the Twitter stream, and the dashboard consists of various visualizations that in different ways provides the user with valuable insights. The final execution was performed with `Trump`, `President`, and `America` as keywords. However, the system is not limited to political opinions among the users of Twitter. The system has been built as modular as possible so that every building block is possible to replace. By using Kafka, it is possible to connect other data sources than the Twitter Streaming API.

All technologies used for this project are examples of state-of-the-art within different areas of Big Data computations that are frequently used in the industry. This project has allowed me to learn these technologies and explore topics that will be of great value for my future career.

As stated in section 1.6, the implemented system should only be seen as a prototype used to prove the underlying theories. The proposed solution shows how new information can be revealed from already existing data by executing a continuous query on streaming data, and thus the project goal is considered to be fulfilled.

7.2 Future Work

This section provides a summary of improvements that should be considered in future work.

Filter Function

The proposed solution uses known name variants to increase the search vocabulary. As discussed in the previous section, the name variants retrieved from the Google Knowledge Search API may not always be appropriate for filtering of Tweets. To improve the filtering and increase the number of relevant matches, it is important that the known name variants be suitable for the structure of the records. In the case of Twitter data which contains Tweets that are short, concise and straight to the point, the known name variants should probably not be longer than one word to increase the probability of generating matches. Another interesting aspect that should be tested is the accuracy of the filter function regarding precision and recall. By using a dataset that contains documents that are marked as relevant or not relative to a query, it would be possible to investigate this aspect. Lastly, the intermediate storage medium that is used for storing the inserted keywords should be improved. Examples of options that might be suitable for a distributed environment are MemCached[1], HBase[2], and Redis[3].

Ranking Function

Some improvements regarding the ranking function have already been mentioned. First and foremost, the function should be able to read from the same file as the filter function does and create the ranking query automatically. This will make the system more usable as the user does not need to insert the keywords more than once nor have any coding experience. The proposed ranking function should also be considered as quite naive. A more sophisticated ranking function could, therefore, be to maintain a shorter list of relevant records and thus not need to calculate and update the relevance score for every record in the system. Algorithms such as rank-join should be investigated to see how it can be implemented on both stored and streaming data.

Scale to Test in a Realistic Environment

During this project, the proposed system has only been tested in a small-scale local environment. To fully utilize the potential of the used technologies, the experiment should be conducted in a larger environment consisting of multiple nodes to simulate a more realistic use case. It would also be interesting to connect other types of data sources to Kafka and thus test to visualize other content types than Tweets.

[1]MemCached: `https://memcached.org/`
[2]HBase: `https://hbase.apache.org/`
[3]Redis: `https://redis.io/`

Bibliography

[1] P. Norrhall, "Visualization of Trends on Twitter." https://www.dropbox.com/s/8sfjvmo41ls8982/fordypningsprosjekt-final.pdf?dl=0, 2017.

[2] A. Kanavos, N. Nodarakis, S. Sioutas, A. Tsakalidis, D. Tsolis, and G. Tzimas, "Large Scale Implementations for Twitter Sentiment Classification," *Algorithms*, vol. 10, no. 1, p. 33, 2017.

[3] B. J. Oates in *Researching Information Systems and Computing*, p. 137, SAGE, 2005.

[4] V. R. Basili, "Software Modeling and Measurement: the Goal/Question/Metric Paradigm," tech. rep., 1992.

[5] M. Lewis-Beck, A. Bryman, and T. Futing Liao, "The SAGE Encyclopedia of Social Science Research Methods." http://methods.sagepub.com/reference/the-sage-encyclopedia-of-social-science-research-methods, 2004. Retrieved 2017-11-27.

[6] A. Cockburn, "Using both Incremental and Iterative Development," *STSC CrossTalk (USAF Software Technology Support Center)*, vol. 21, no. 5, pp. 27–30, 2008.

[7] D. Radigan, "Kanban - A brief Introduction." https://www.atlassian.com/agile/kanban, 2018. Retrieved 2018-04-30.

[8] Twitter, "Connecting to a Streaming Endpoint - Twitter Developers." https://developer.twitter.com/en/docs/tweets/filter-realtime/guides/connecting, 2017. Retrieved 2017-11-27.

[9] D. Laney, "3D Data Management: Controlling Data Volume, Velocity and Variety," *META Group Research Note*, vol. 6, p. 70, 2001.

[10] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache Spark: A Unified Engine for Big Data Processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[11] C. Rohrdantz, D. Oelke, M. Krstajic, and F. Fischer, "Real-time Visualization of Streaming Text Data: Tasks and Challenges," in *VIS-Week*, 2011.

[12] S. Babu and J. Widom, "Continuous Queries over Data Streams," *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001.

[13] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du, "Data Ingestion for the Connected World," in *CIDR*, 2017.

[14] B. Pang, L. Lee, *et al.*, "Opinion Mining and Sentiment Analysis," *Foundations and Trends® in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008.

[15] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, *Machine Learning: An Artificial Intelligence Approach*. Springer Science & Business Media, 2013.

[16] A. Baltas, A. Kanavos, and A. K. Tsakalidis, "An Apache Spark Implementation for Sentiment Analysis on Twitter Data," in *International Workshop of Algorithmic Aspects of Cloud Computing*, pp. 15–25, Springer, 2016.

[17] "MLlib: Naive Bayes." `http://spark.apache.org/docs/2.2.0/mllib-naive-bayes.html`, 2018. Retrieved 2018-05-23.

[18] G. Chin Jr, M. Singhal, G. Nakamura, V. Gurumoorthi, and N. Freeman-Cadoret, "Visual Analysis of Dynamic Data Streams," *Information Visualization*, vol. 8, no. 3, pp. 212–229, 2009.

[19] O. Rist, "The Best Data Visualization Tools of 2017." `http://uk.pcmag.com/cloud-services/83744/guide/the-best-data-visualization-tools-of-2017`, 2017. Retrieved 2017-12-01.

[20] K. Sparck Jones, "A Statistical Interpretation of Term Specificity and its Application in Retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[21] G. Salton, A. Wong, and C.-S. Yang, "A Vector Space Model for Automatic Indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[22] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[23] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters.," *HotCloud*, vol. 12, pp. 10–10, 2012.

[24] J. Kreps, "Putting Apache Kafka To Use: A Practical Guide To Building a Streaming Platform." `https://www.confluent.io/blog/stream-data-platform-1/`, 2018. Retrieved 2018-06-02.

[25] "Apache Kafka - Introduction." `https://kafka.apache.org/intro.html`, 2017. Retrieved 2018-04-02.

[26] "Getting Started - Elasticsearch Reference [6.2]." `https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html`, 2018. Retrieved 2018-04-22.

[27] "Basic Concepts - Elasticsearch Reference [6.2]." `https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html`, 2018. Retrieved 2018-04-22.

[28] "Introduction - Kibana User Guide [6.2]." `https://www.elastic.co/guide/en/kibana/current/introduction.html`, 2018. Retrieved 2018-04-22.

[29] C. Gormley and Z. Tong, "What is Relevance? - Elasticsearch: The Definitive Guide [2.x]." `https://www.elastic.co/guide/en/elasticsearch/guide/current/relevance-intro.html`, 2015. Retrieved 2018-05-11.

[30] C. Gormley and Z. Tong, "Theory Behind Relevance Scoring - Elasticsearch: The Definitive Guide [2.x]." `https://www.elastic.co/guide/en/elasticsearch/guide/current/scoring-theory.html`, 2015. Retrieved 2018-05-11.

[31] S. Robertson, H. Zaragoza, *et al.*, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[32] A. Cholakian, "A Gentle Intro to Function Scoring." `https://www.elastic.co/blog/found-function-scoring`, 2014. Retrieved 2018-05-12.

[33] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin, "Fast Data in the Era of Big Data: Twitter's Real-time Related Query Suggestion Architecture," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1147–1158, ACM, 2013.

[34] A. Magdy, "Scalable Microblogs Data Management," in *Proceedings of the 2016 on SIGMOD'16 PhD Symposium*, pp. 32–36, ACM, 2016.

[35] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, *et al.*, "AsterixDB: A Scalable, Open Source BDMS," *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1905–1916, 2014.

[36] S. Jacobs, M. Y. S. Uddin, M. Carey, V. Hristidis, V. J. Tsotras, and N. Venkatasubramanian, "A BAD Demonstration: Towards Big Active Data," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1941–1944, 2017.

[37] M. J. Carey, S. Jacobs, and V. J. Tsotras, "Breaking BAD: A Data Serving Vision for Big Active Data," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 181–186, ACM, 2016.

[38] R. Grover and M. J. Carey, "Data Ingestion in AsterixDB," in *EDBT*, pp. 605–616, 2015.

[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[40] G. A. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems.," tech. rep., Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[41] P. Pääkkönen, "Feasibility Analysis of AsterixDB and Spark Streaming with Cassandra for Stream-based Processing," *Journal of Big Data*, vol. 3, no. 1, p. 6, 2016.

[42] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 239–250, ACM, 2015.

[43] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, *et al.*,

"Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pp. 1789–1792, IEEE, 2016.

[44] R. Wesley, M. Eldridge, and P. T. Terlecki, "An Analytic Data Engine for Visualization in Tableau," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 1185–1194, ACM, 2011.

[45] R. E. Hoyt, D. Snider, C. Thompson, and S. Mantravadi, "IBM Watson Analytics: Automating Visualization, Descriptive, and Predictive Statistics," *JMIR public health and surveillance*, vol. 2, no. 2, 2016.

[46] A. Magdy, L. Alarabi, S. Al-Harthi, M. Musleh, T. M. Ghanem, S. Ghani, and M. F. Mokbel, "Taghreed: A System for Querying, Analyzing, and Visualizing Geotagged Microblogs," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 163–172, ACM, 2014.

[47] J. Jia, C. Li, X. Zhang, C. Li, M. J. Carey, *et al.*, "Towards Interactive Analytics and Visualization on One Billion Tweets," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, p. 85, ACM, 2016.

[48] Cloudberry, "Cloudberry Demo - TwitterMap." `http://cloudberry.ics.uci.edu/apps/twittermap`, 2016. Retrieved 2018-05-31.

[49] K. Balog and H. Ramampiaro, "Cumulative Citation Recommendation: Classification vs. Ranking," in *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pp. 941–944, ACM, 2013.

[50] K. Balog, H. Ramampiaro, N. Takhirov, and K. Nørvåg, "Multi-step Classification approaches to Cumulative Citation Recommendation," in *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval*, pp. 121–128, Le centre de hautes etudes internationales d'informatique documentaire, 2013.

[51] "Using Deep Learning at Scale in Twitter's Timelines." `https://blog.twitter.com/engineering/en_us/topics/insights/2017/using-deep-learning-at-scale-in-twitters-timelines.html`, 2017. Retrieved 2018-02-21.

[52] K. Tao, F. Abel, C. Hauff, and G.-J. Houben, "What Makes a Tweet Relevant for a Topic," *Making Sense of Microposts (# MSM2012)*, pp. 49–56, 2012.

[53] Y. Duan, L. Jiang, T. Qin, M. Zhou, and H.-Y. Shum, "An Empirical Study on Learning to Rank of Tweets," in *Proceedings of the 23rd International Conference on Computational Linguistics*, pp. 295–303, Association for Computational Linguistics, 2010.

[54] A. Rosen, "Tweeting Made Easier." `https://blog.twitter.com/official/en_us/topics/product/2017/tweetingmadeeasier.html`, 2017. Retrieved 2018-04-05.

[55] "Introduction to Tweet JSON - Twitter Developers." `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json.html`, 2018. Retrieved 2018-04-13.

[56] Twitter, "Consuming Streaming Data - Twitter Developers." `https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data`, 2017. Retrieved 2018-06-02.

[57] T. Das, M. Zaharia, and P. Wendell, "Diving into Apache Spark Streaming's Execution Model." `https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html`, 2016. Retrieved 2017-12-05.

[58] C. Gormley and Z. Tong, "Dynamic Mapping - Elasticsearch: The Definitive Guide [2.x]." `https://www.elastic.co/guide/en/elasticsearch/guide/current/relevance-intro.html`, 2015. Retrieved 2018-05-11.

[59] A. Go, R. Bhayani, and L. Huang, "Twitter Sentiment Classification using Distant Supervision," *CS224N Project Report, Stanford*, vol. 1, no. 2009, p. 12, 2009.

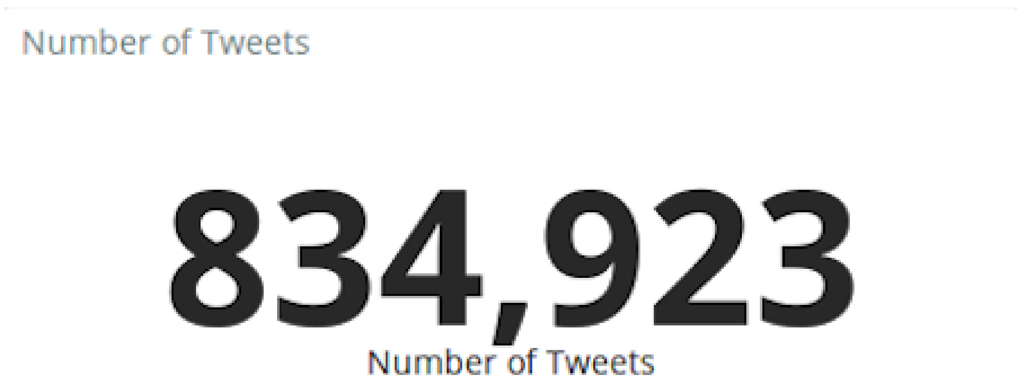
[60] T. M. Abrahamsen, "Scaling Machine Learning Methods to Big Data Systems," Master's thesis, NTNU, 2017.

[61] Databricks, "Debugging Spark Streaming Application." `https://docs.databricks.com/spark/latest/rdd-streaming/debugging-streaming-applications.html`, 2018. Retrieved 2018-06-05.

Appendices

Appendix A

Figures

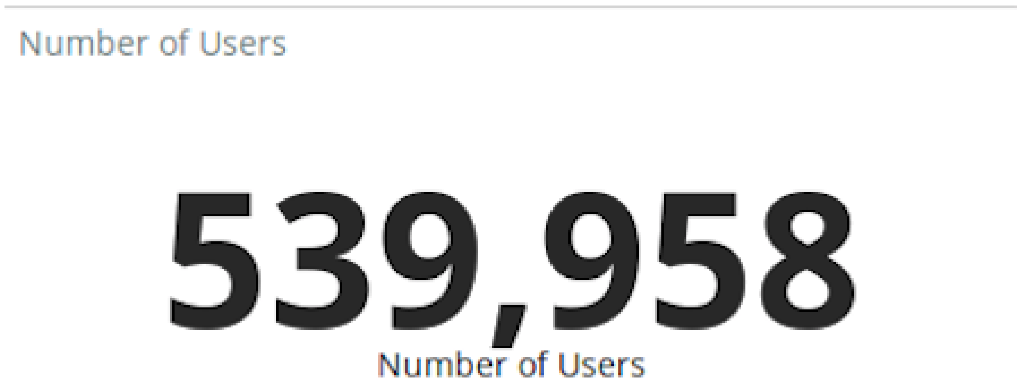


Number of Tweets

834,923

Number of Tweets

Figure A.1: Number of retrieved Tweets



Number of Users

539,958

Number of Users

Figure A.2: Number of distinct users

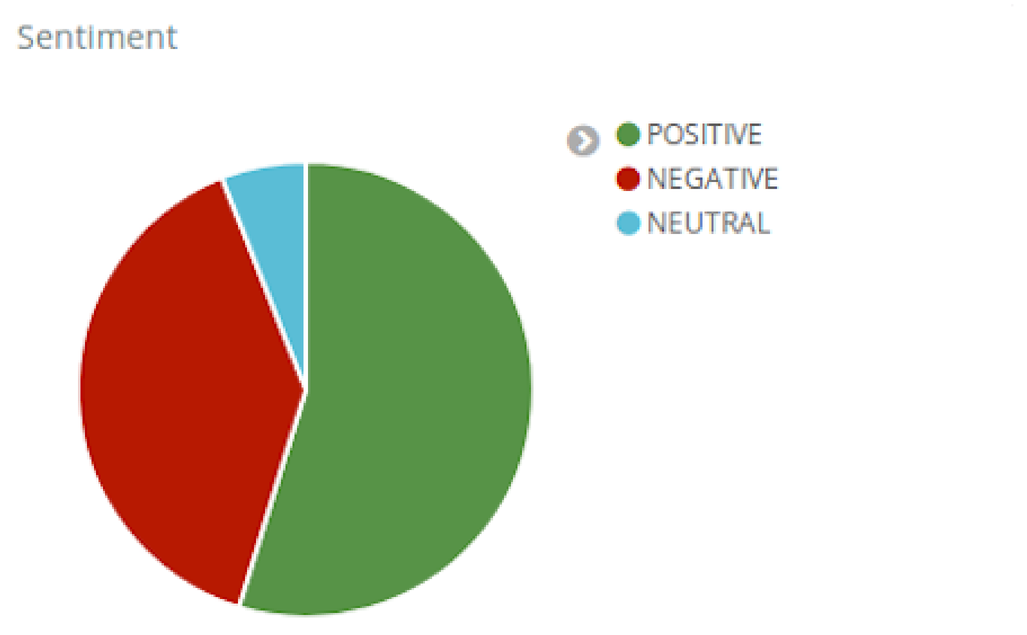


Figure A.3: Predicted sentiment

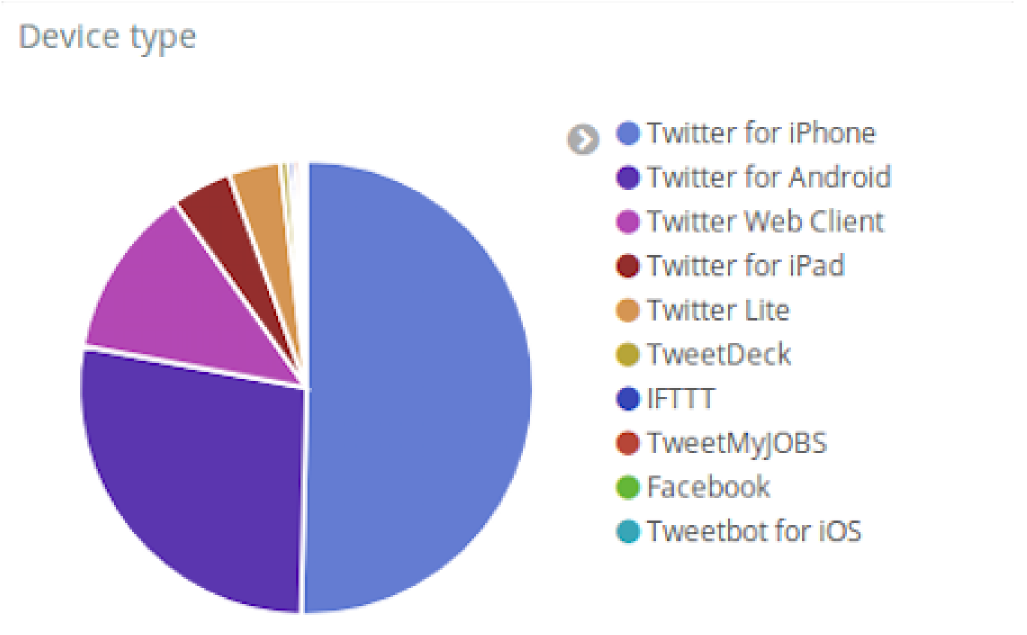


Figure A.4: Distribution of used device type

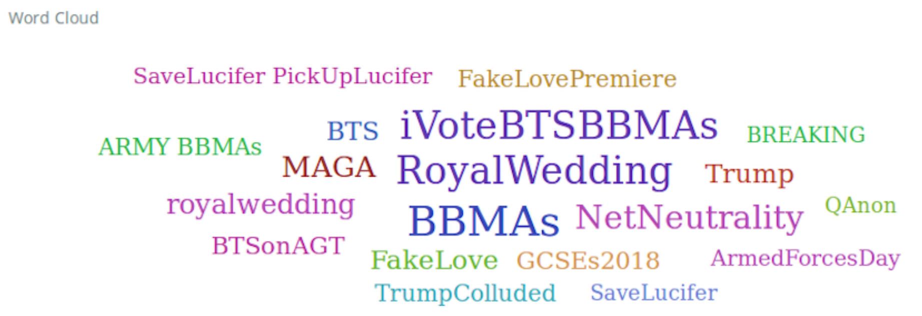
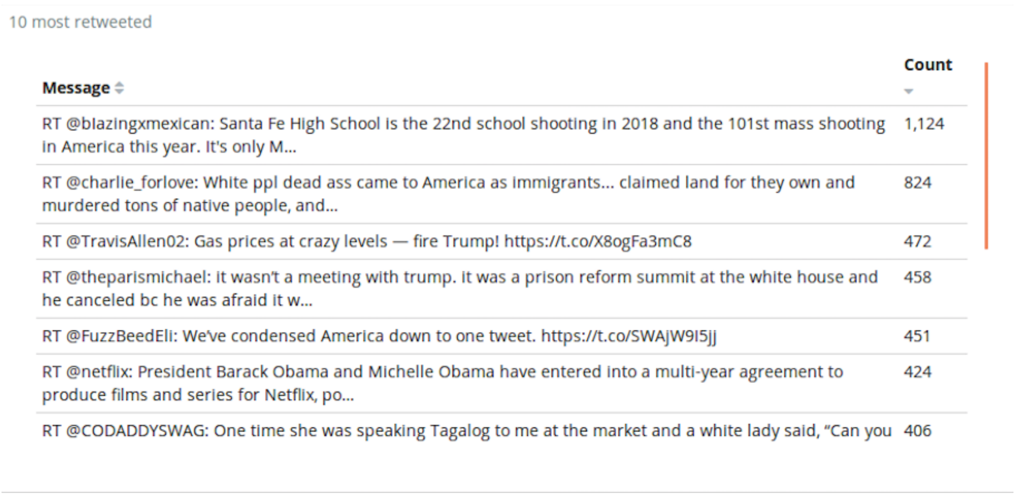


Figure A.5: Cloud showing the most trending hashtags



10 most retweeted

Message	Count
RT @blazingxmexican: Santa Fe High School is the 22nd school shooting in 2018 and the 101st mass shooting in America this year. It's only M...	1,124
RT @charlie_forlove: White ppl dead ass came to America as immigrants... claimed land for they own and murdered tons of native people, and...	824
RT @TravisAllen02: Gas prices at crazy levels — fire Trump! https://t.co/X8ogFa3mC8	472
RT @theparismichael: it wasn't a meeting with trump. it was a prison reform summit at the white house and he canceled bc he was afraid it w...	458
RT @FuzzBeedEli: We've condensed America down to one tweet. https://t.co/SWAjW9I5jJ	451
RT @netflix: President Barack Obama and Michelle Obama have entered into a multi-year agreement to produce films and series for Netflix, po...	424
RT @CODADDYSWAG: One time she was speaking Tagalog to me at the market and a white lady said, "Can you	406

Figure A.6: Table showing the Tweets that are most re-tweeted

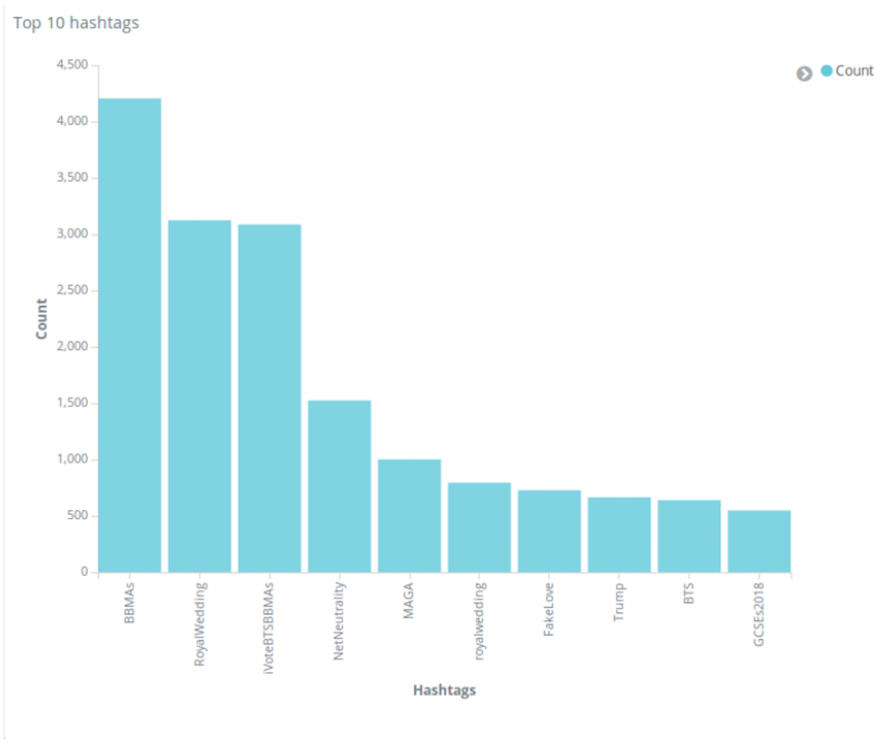
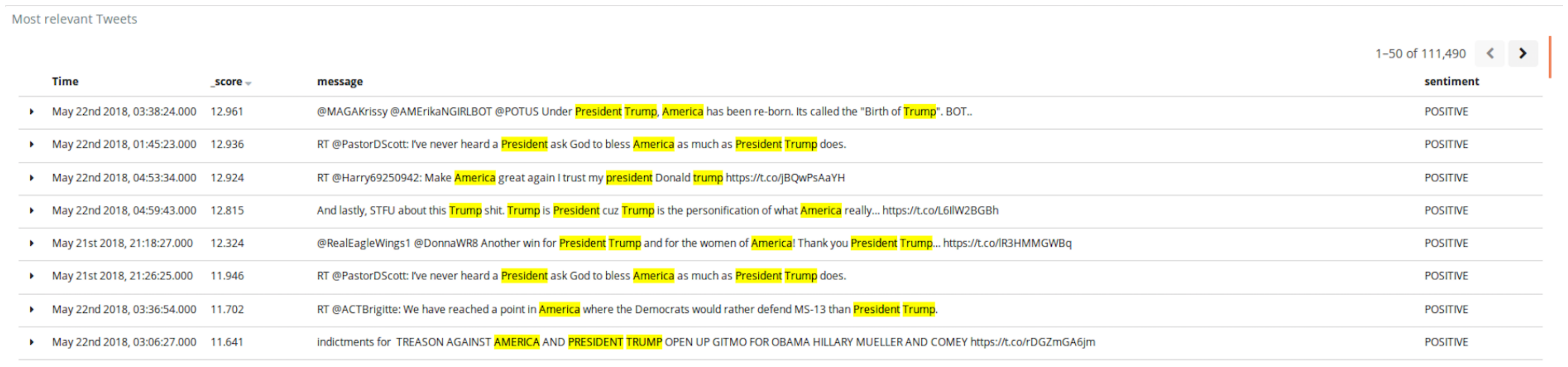


Figure A.7: Top 10 most popular hashtags



Most relevant Tweets

1–50 of 111,490

Time	_score	message	sentiment
May 22nd 2018, 03:38:24.000	12.961	@MAGAKrissy @AMErikaNGIRLBOT @POTUS Under President Trump, America has been re-born. Its called the "Birth of Trump". BOT..	POSITIVE
May 22nd 2018, 01:45:23.000	12.936	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE
May 22nd 2018, 04:53:34.000	12.924	RT @Harry69250942: Make America great again I trust my president Donald trump https://t.co/jBQwPsAaYH	POSITIVE
May 22nd 2018, 04:59:43.000	12.815	And lastly, STFU about this Trump shit. Trump is President cuz Trump is the personification of what America really... https://t.co/L6llW2BGBh	POSITIVE
May 21st 2018, 21:18:27.000	12.324	@RealEagleWings1 @DonnaWR8 Another win for President Trump and for the women of America! Thank you President Trump... https://t.co/lR3HMMGWBq	POSITIVE
May 21st 2018, 21:26:25.000	11.946	RT @PastorDScott: I've never heard a President ask God to bless America as much as President Trump does.	POSITIVE
May 22nd 2018, 03:36:54.000	11.702	RT @ACTBrigitte: We have reached a point in America where the Democrats would rather defend MS-13 than President Trump.	POSITIVE
May 22nd 2018, 03:06:27.000	11.641	indictments for TREASON AGAINST AMERICA AND PRESIDENT TRUMP OPEN UP GITMO FOR OBAMA HILLARY MUELLER AND COMEY https://t.co/rDGZmGA6jm	POSITIVE

Figure A.8: Ranked list of the most relevant Tweets

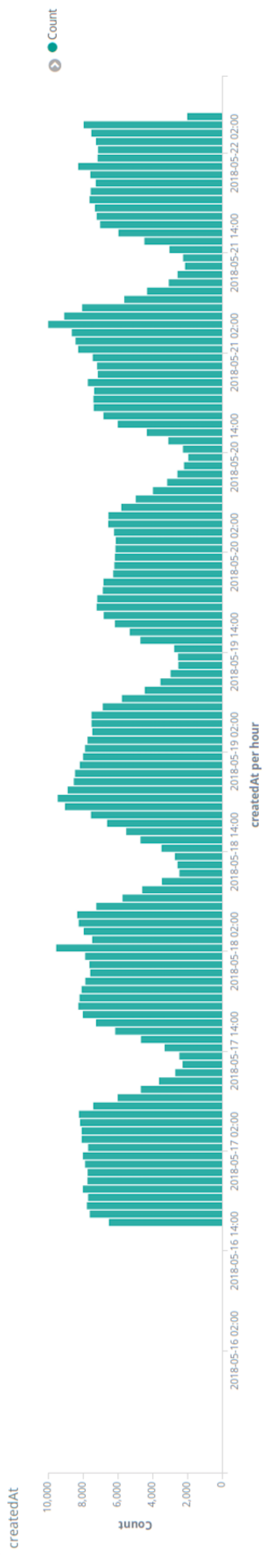


Figure A.9: Timeline of when Tweets was created

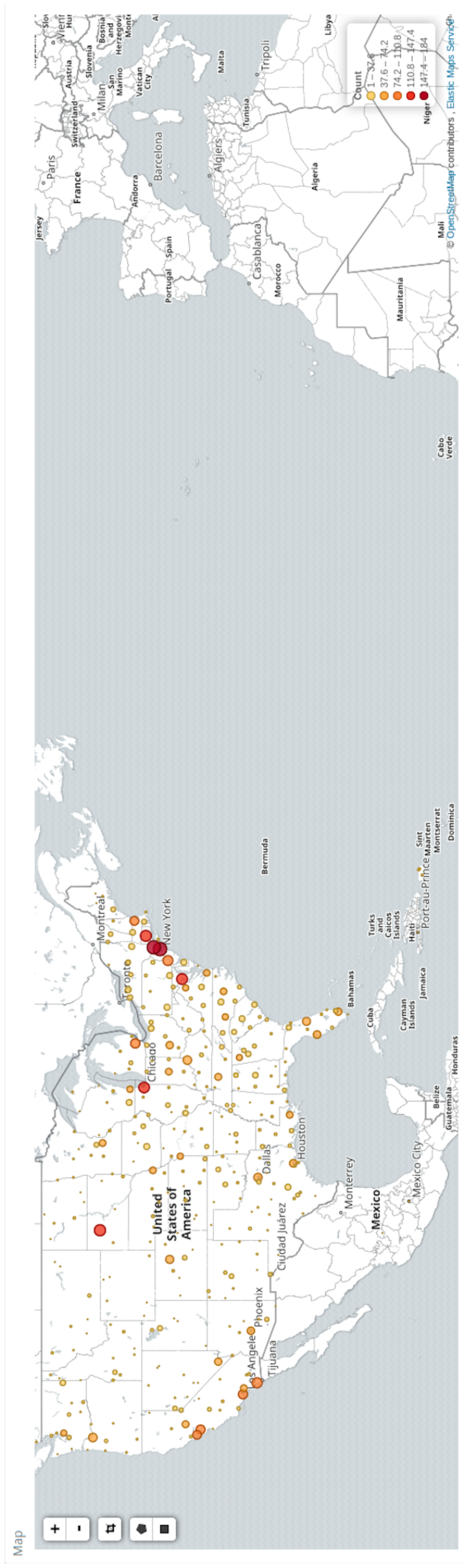
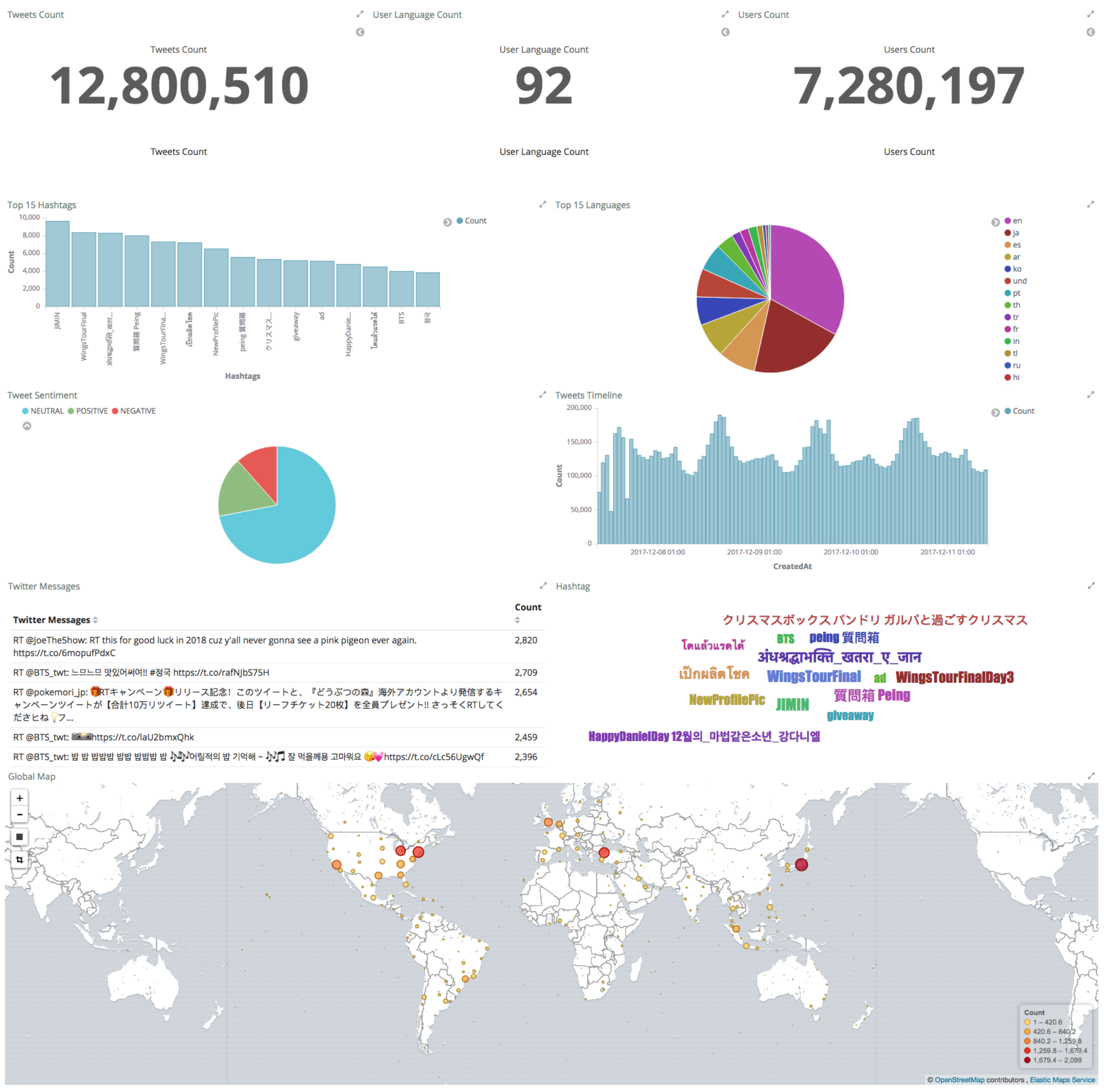


Figure A.10: Global map showing lat-long pairs



Twitter Messages

Twitter Messages	Count
RT @JoeThe5how: RT this for good luck in 2018 cuz y'all never gonna see a pink pigeon ever again. https://t.co/6mopufPdxC	2,820
RT @BTS_twt: 느므느므 맛있어써여!! #정국 https://t.co/rafNJbS75H	2,709
RT @pokemori_jp: 🎁RTキャンペーン🎁リリース記念！このツイートと、『どうぶつの森』海外アカウントより発信するキャンペーンツイートが【合計10万リツイート】達成で、後日【リーフチケット20枚】を全員プレゼント!! さっそくRTしてくださいね💡フ…	2,654
RT @BTS_twt: 📷📸https://t.co/laU2bmxQhk	2,459
RT @BTS_twt: 밥 밥 밥밥밥 밥밥 밥밥밥 밥 🎶🎶어릴적의 밥 기억해 ~ 🎶🎵 잘 먹을께용 고마워요 😘💓https://t.co/cLc56UgwQf	2,396

Figure A.11: Dashboard created as part of the specialization project

Appendix B

Listings

Listing B.1: Example of Tweet in JSON format

```json
{
  "tweet": {
    "created_at": "Thu Apr 06 15:24:15 +0000 2017",
    "id_str": "850006245121695744",
    "text": "1\/ Today we\u2019re sharing our vision for the future
    of the Twitter API platform!\nhttps:\/\/t.co\/XweGngmxlP",
    "user": {
      "id": 2244994945,
      "name": "Twitter Dev",
      "screen_name": "TwitterDev",
      "location": "Internet",
      "url": "https:\/\/dev.twitter.com\/",
      "description": "Your official source for Twitter Platform news,
      updates & events. Need technical help? Visit https:\/\/twittercommunity.com\/
      \u2328\ufe0f #TapIntoTwitter"
    },
    "place": {

    },
    "entities": {
      "hashtags": [

      ],
      "urls": [
        {
          "url": "https:\/\/t.co\/XweGngmxlP",
          "unwound": {
            "url": "https:\/\/cards.twitter.com\/cards\/18ce53wgo4h\/3xo1c",
            "title": "Building the Future of the Twitter API Platform"
          }
        }
      ],
      "user_mentions": [

      ]
    }
  }
}
```

Listing B.2: Example of Tweet in Twitter4j Status format

```
StatusJSONImpl{
  createdAt=Thu Apr 06 15:24:15 CEST 2017,
  id=850006245121695744,
  text='1\/ Today we\u2019re sharing our vision for the future
  of the Twitter API platform!\nhttps:\/\/t.co\/XweGngmxlP',
  urlEntities=[URLEntityJSONImpl{
    url='https:\/\/t.co\/XweGngmxlP',
    expandedURL='https:\/\/cards.twitter.com\/cards\/18ce53wgo4h\/3xo1c'}],
  user=UserJSONImpl{
    id=2244994945,
    name='Twitter Dev',
    screenName='TwitterDev',
    location='Internet',
    description='Your official source for Twitter Platform news,
    updates & events. Need technical help? Visit https:\/\/twittercommunity.com\/
    \u2328\ufe0f #TapIntoTwitter'}
  }
```

Listing B.3: Words to test `filter`-function on

```
Sweden
Norway
Denmark
Finland
Netherlands
Spain
France
Italy
Portugal
Germany
Greece
Switzerland
Poland
Austria
Belgium
Ukraine
Croatia
Iceland
Cyprus
Romania
```

Listing B.4: Explanation of how the ranking score is calculated for a Tweet

```
{
  "hits": {
    "total": 111490,
    "max_score": 9.304917,
    "hits": [
      {
        "_shard": "[visualization][2]",
        "_node": "meWimDjoRXavG-MQk3-l9A",
        "_index": "visualization",
        "_type": "tweets",
        "_id": "ET1WhWMBtZQDwiDZS8Hf",
        "_score": 9.304917,
        "_source": {
          "coordinates": null,
          "userDescription": "Poet, Patriot, Sovereign Human, Being.
          I follow back Patriots!
          "friendsCount": 2344,
          "userLanguage": "en",
          "sentiment": "POSITIVE",
        "placeCountry": null,
        "deviceType": "Twitter for Android",
        "userScreenName": "Lady_Greenstone",
        "statusLanguage": "en",
        "createdAt": "2018-05-22T02:53:34",
        "message": "RT @Harry69250942: Make America great again I trust my
        president Donald trump https://t.co/jBQwPsAaYH",
        "retweetCount": 0,
        "userID": 1167806174,
        "messageLength": 101,
        "userName": "Annie Oakley",
        "followersCount": 2279,
        "hashtags": ""
      },
      "_explanation": {
        "value": 9.304917,
        "description": "function score, product of:",
        "details": [
          {
            "value": 10.268894,
            "description": "sum of:",
            "details": [
              {
                "value": 3.953562,
                "description": "weight(message:america in 1749)
                        [PerFieldSimilarity], result of:",
                "details": [
                  {
                    "value": 3.953562,
                    "description": "score(doc=1749,freq=1.0 = termFreq=1.0\n),
                            product of:",
                    "details": [
                      {
                        "value": 3.6883576,
                        "description": "idf, computed as
                        log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",
                        "details": [
                          {
                            "value": 4168,
                            "description": "docFreq"
                          },
                          {
                            "value": 166652,
                            "description": "docCount"
                          }
                        ]
                      },
                      {
                        "value": 1.0719031,
                        "description": "tfNorm, computed as
    (freq * (k1 + 1)) / (freq + k1 * (1 - b + b * fieldLength / avgFieldLength))
                                from:",
                        "details": [
                          {
                            "value": 1,
                            "description": "termFreq=1.0"
                          },
                          {
                            "value": 1.2,
                            "description": "parameter k1"
                          },
                          {
                            "value": 0.75,
                            "description": "parameter b"
                          },
                          {
                            "value": 17.942,
                            "description": "avgFieldLength"
                          },
                          {
                            "value": 15,
                            "description": "fieldLength"
                          }
                        ]
                      }
                    ]
                  }
                ]
              },
              {
                "value": 3.740428,
                "description": "weight(message:president in 1749)
                        [PerFieldSimilarity],
                result of:",
                "details": [
                  {
                    "value": 3.740428,
                    "description": "score(doc=1749,freq=1.0 = termFreq=1.0\n),
                           product of:",
                    "details": [
                      {
                        "value": 3.4895205,
                        "description": "idf, computed as
        log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",
                        "details": [
                          {
                            "value": 5085,
                            "description": "docFreq"
                          },
                          {
                            "value": 166652,
                            "description": "docCount"
                          }
                        ]
                      },
                      {
                        "value": 1.0719031,
                        "description": "tfNorm, computed as
    (freq * (k1 + 1)) / (freq + k1 * (1 - b + b * fieldLength / avgFieldLength))
                        from:",
                        "details": [
                          {
                            "value": 1,
                            "description": "termFreq=1.0"
                          },
                          {
                            "value": 1.2,
                            "description": "parameter k1"
                          },
                          {
                            "value": 0.75,
                            "description": "parameter b"
                          },
                          {
                            "value": 17.942,
                            "description": "avgFieldLength"
                          },
                          {
                            "value": 15,
                            "description": "fieldLength"
                        }
                      ]
                    }
                  ]
                }
              ]
            },
            {
              "value": 2.574904,
              "description": "weight(message:trump in 1749)
                      [PerFieldSimilarity], result of:",
              "details": [
                {
                  "value": 2.574904,
                  "description": "score(doc=1749,freq=1.0 = termFreq=1.0\n),
                          product of:",
                  "details": [
                    {
                      "value": 2.4021797,
                      "description": "idf, computed as
                      log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",
                      "details": [
                        {
                          "value": 15085,
                          "description": "docFreq"
                        },
                        {
                          "value": 166652,
                          "description": "docCount"
                        }
                      ]
                    },
                    {
                      "value": 1.0719031,
                      "description": "tfNorm, computed as
(freq * (k1 + 1)) / (freq + k1 * (1 - b + b * fieldLength / avgFieldLength))
                      from:",
                      "details": [
                        {
                          "value": 1,
                          "description": "termFreq=1.0"
                        },
                        {
                          "value": 1.2,
                          "description": "parameter k1"
                        },
                        {
                          "value": 0.75,
                          "description": "parameter b"
                        },
                        {
                          "value": 17.942,
                          "description": "avgFieldLength"
                        },
                        {
                          "value": 15,
                          "description": "fieldLength"
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
          },
          {
            "value": 0.90612656,
            "description": "min of:",
            "details": [
              {
                "value": 0.90612656,
                "description": "Function for field createdAt:",
                "details": [
                  {
                    "value": 0.90612656,
                    "description":
                    "exp(-0.5*pow(MIN[Math.max(Math.abs(1.526957614E12(=doc value)
                     - 1.52699379671E12(=origin)))
                     - 3600000.0(=offset), 0)],2.0)/5.384830386217238E15)"
                  }
                ]
              },
              {
                "value": 3.4028235e+38,
                "description": "maxBoost"
              }
            ]
          }
        ]
      }
    }
  ]
  }
}
```

Listing B.5: Words to filter Tweets on in final execution

```
Trump
Donald Trump
Presidency of Donald Trump
Protests against Donald Trump
Donald Trump 2017 presidential inauguration
The Trump Organization
United States presidential election 2016
Presidential transition of Donald Trump
Trump International Hotel Las Vegas
Melania Trump
President
Barack Obama
Donald Trump
Bill Clinton
George W. Bush
Assassination of John F. Kennedy
Ronald Reagan
Franklin D. Roosevelt
John F. Kennedy
Abraham Lincoln
America
United States
North America
Latin America
Americas
South America
Bank of America
Captain America
Indigenous peoples of the Americas
Club America
```

Appendix C

Project Code

C.1 GitHub Projects

Spark Application: `https://github.com/phrida/MasterProject`

Kafka Producer: `https://github.com/phrida/KafkaProducer`

C.2 Data Generator

Data Generator: `https://github.com/phrida/DataGenerator`

Data Set: `https://www.dropbox.com/s/flut7z4zmxxljo0/rawTweetsFinal.json?dl=0`

C.3 Specialization Project

Specialization Project: `https://www.dropbox.com/s/8sfjvmo41ls8982/fordypningsprosjekt-final.pdf?dl=0`