University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Clustering Conversations in Social Networks

Scientific Adviser:
Ș.l.dr.ing. Costin Chiru

Author:
Victor Andrei Oprea

Bucharest, 2015

Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

LUCRARE DE DIPLOMĂ

Clustering Conversations in Social Networks

Conducător Științific:
Șl.dr.ing. Costin Chiru

Autor:
Victor Andrei Oprea

București, 2015

Abstract

We present Streamer a search application running over streams of Twitter message. As opposed to most services that just do simple text search over conversations Streamer aims to cluster messages together in order to simplify going through large number of messages involving similar topics. Unlike most software solutions that use a fixed corpus when clustering, Streamer works with streaming data, Twitter messages are continuously retrieved and the clusters update as more data comes in. We try to combine a variety of methods to improve our clustering estimates such as TF-IDF and part of speech tagging. This paper describes the implementation of the system and explores similar projects that tackle problems similar to ours.

Contents

List of Figures

Chapter 1

Introduction

Social media platforms are no longer an emerging field, they have become well established and millions of messages are exchanged daily by people globally. A variety of application clients and services try to keep up with this surge of information by offering users information about popular topics or highlights about the events that are happening around them. They are promoting popular content, either by number of clicks, views, favorites or other metrics, and although this is effective for controversial topics, it does little to highlight other subjects of conversation.

In this paper, we present a study on the clustering of messages from the Twitter platform, also known as *tweets*, inspired by the website's *Trends*category which presents the most popular subjects either worldwide or in a certain geographical region. The aim of our paper is to explore more than just popular subjects by just providing a query by which to extract conversations and cluster the resulting discussions based on keywords, frequencies, weights and part of speech tagging. The purpose of the clustering is to offer an in depth view of the different conversation taking place on the same topic either trending or not and to be able to gain a high level overview of the conversations taking place around a certain subject.

It is no longer possible to track individual messages when hundreds of them per second are coming in. By clustering similar conversations together and offering an interface through which to explore the information, it is possible to view multiple opinions on the same subject in a structured way and get more information than just having everything in one place.

The process through which the clustering is achieved is separated into three parts: data collection via the Twitter public API, message annotation with part-of-speech tags and message clustering. The messages provided by the website's API already provide a filtering option, you can specify keywords that you want to be part of the messages you get back. This might provide some indication of the conversation topic but getting an overview of the different conversations on the same subject is not a trivial task because messages have no obvious order. Another concern related to online conversations is the amount of syntactic errors due to the fast paced way they usually take place or due to the restrictions imposed by Twitter (the character limit inside of messages).

We will provide more information about the problems encountered and the solutions we implemented in the following sections.

1.1 Project Description

This first sections of the paper will discuss project scope and what objectives we have set to accomplish. Afterwards we will discuss related work in the field, implemented solutions that deal with Twitter conversations or papers written on this subject. The popularity of the platform, the large quantity of messages and the fact that it is relatively easy to have access to large corpuses of text from different websites have made Twitter conversations a popular topic of research. We then continue by giving a detailed explanation of the clustering process the algorithm chosen, how does K-Means work, and the weight function, cosine similarity which is part of the algorithm. The Implementation details section provides a look into the system architecture, some of the trade-offs made and the benefits, and continues with parsing and message aggregation. The next section will go into details regarding testing and the results obtained and the last part will offer suggestions for further improvements and research.

1.1.1 Project Scope

We will first offer a broader explanation of the problem we wish to solve.
Twitter[1] is an online micro blogging platform and social networking website. Users communicate through short 140 character messages called ẗweets.̈ To ease communication people use m̈entions,̈ the @̈c̈haracter followed by a persons name. This is a way to involve another account into the conversation. Another feature of the conversation is the ḧashtag,̈ the #̈s̈ign followed by a word this is used to highlight key parts in the message, either a feeling or subject. Popular such hashtags are included in the Trending Topics. These are popular subjects automatically generated from conversations taking place worldwide or in a certain region. This is why people often times include hashtags in order to associate their message with a popular topic. Users can also favorite a tweet, and "retweet" it, meaning they share it with the people that follow them while still attributing the message to the original author. These two metrics contribute to the overall popularity of a tweet.

Twitter is an interesting platform for research because of the large number of messages exchanged daily. There are 500 million tweets sent daily by its 302 million monthly active users[2] with a record of 143,199 tweets per second[3]. People usually turn to Twitter during major natural events, sporting events, award ceremonies and so on. With such a high amount of information coming in every second it is almost impossible to keep track of everything that is discussed.

The "Trends" category highlights all popular topics and hashtags but exploring any one topic reveals a very large number of messages with just as more coming in every second (for popular subjects). This makes it very hard to see different points of view, different opinions on the matter. The aim of Streamer is to offer a high level overview of the conversation, where different opinions on the same topic are grouped into different clusters. This would make it easy to identify what kind of messages you are likely to find in a cluster just by reading some of them and would allow quick understand all aspects of a developing story.

Consider the following scenario: during a global sporting event such as the football world cup which takes place over the course of several week, you want to keep track of the public opinion for your country's national team. If they are playing a game it's most likely that they are trending and a hashtag has already become popular between the people exchanging messages about the game. But how do you track the information? What if the hashtag includes

[1] https://support.twitter.com/articles/49309-using-hashtags-on-twitter
[2] https://about.twitter.com/company
[3] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

both team (for example #GERvsBRA used to track Germany - Brazil game) what if you want to see information about the first goal, or a certain player in the game. Clustering all available messages would easily reveal the ones referring to one of the teams or an individual player. This is where Streamer will in and change the way information is being discovered.

Twitter has a powerful web platform and that is why we want to take advantage of this by allowing our project to interact with the web portal. As we present the clusters and associated messages it should be accessible to navigate between our interface and Twitter. This would allow access to the profile of the users who sent out the messages. It would give direct link to the message and therefor the context of the message would be available.

The scope of the project **Streamer** is to provide close to realtime clustering of conversations that take place in the online medium. My choice for a social network is Twitter. Twitter has around 302 million active users (May 2015) [1] who send 500 million tweets each day mostly from their mobile phones. A tweet is a 140 character long message and because of this conversations are hard to keep track of and provide little to no context on their subject. This makes them an excellent candidate for a clustering application like **Streamer** which aims to provide an overview for conversations spanning over all the topics the user of the application provided.

1.1.2 Project Objectives

With **Streamer** we want to provide close to realtime clustering of conversations that take place on the Twitter social social platform. As a result of its popularity the platform is generating massive amounts of messages that make it increasingly difficult to go through and read the content. We are trying to answer several questions: is it possible to manage a large number of online conversations on a particular subject, can you easily identify the different points of view expressed, can you identify the different opinions expressed by the users of the platform.

Having answers to these questions we can easily monitor news reports, popular social events and track people's feelings about current events. It is also a test for the current technology stack available, Twitter employs hundreds of engineers to handle its infrastructure and services, we want to see if it is possible to build a system capable of scaling and handling a fraction of the traffic generated by the platform just by using open source libraries for our solution.

We want to achieve a solution that is easy to deploy and set up, that is able to connect and retrieve real time messages and that can group messages together with certain accuracy based on their topic.
To achieve this objective there are several components that go into the architecture:

- Data collection — This step involves connecting and retrieving public messages from the Twitter platform matching a particular query of interest. This is achieved via Twitter's streaming API which serves a percentage amount of all conversations. The system requires that all requests be done by authenticated clients. After registering with the service it automatically starts sending realtime messages that match your search terms. Due to the difference between the rate of processing and the rate at which messages are coming in a temporary data storage is required.
- Message parsing — The tweets that have been acquired need to be parsed. By parsing we are referring to an initial processing step in which meta information is added to messages before they can enter the clustering algorithm. Due to the short nature of the messages, 10-14 words on average, the information is filled with jargon or syntactic errors in an attempt to increase the amount of information in such a small payload. In this step

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

each word is augmented with its corresponding part of speech tag and its contribution to the overall document is computed. The importance of terms is the result of TF-IDF, a computational step in which word frequency is used to attribute a weight to each of them.

- Message clustering — The result of this step will be a mapping of message ids to clusters. By cluster we are referring to a group of messages that all concern the same topic. The new messages obtained as a result of the previous step are clustered using K-Means algorithm. The algorithm requires a weight function, for this we have chosen the cosine similarity function. The algorithm works in steps, each step is an attempt to improve the overall score of a cluster (how precise is the grouping we have created). This can be quantified using the weight function we previously mentioned. This works by normalizing all the messages, a bucket is created which contains the sum of all words in all documents and each sentence is transformed into a vector who's elements specify if a certain word form the bucket is present or not.
- Cluster API — The mapping from the previous step is transformed into actual clusters of messages now that we now how to assign them together. The backend exposes the clusters of messages via an HTTP enpoint. Any number of clients can connect and have access to the information. The endpoint serves the messages in JSON format, a popular solution of web applications. This makes it easy and accessible for any number of applications to connect and consume the available data without the server having any prior knowledge of the clients it serves.
- Frontend — An application that runs in the browser and connects to the endpoint described in the previous step. Using the canvas API it draws messages that belong to the same clusters closer to each other making it easy to identify them. It also displays the full message and user and allows you to explore the different clusters. This is also an easy way to verify some of our assumptions: we can expand the clusters and assert that the messages have been grouped with certain precision. The interface also links back to the author and original message making it easy to get more context or explore a certain users timeline.

This was meant to provide a high level overview of the system. We will go into further details in the Design and Implementation categories.

1.1.3 Related Work

There are many services and applications that are in some way dependent of Twitter data, whether they just simply archive information so that they may sell large corpuses of data, or actually parse and analyze public user streams and send out alerts or reports regarding search terms or mentions.

At the same time a lot of academic papers are concerned with the content of Twitter conversations. Topics such as summarization of Twitter trends, efficient searching of content, tracking user sentiment and clustering are still open for discussion.

We will delve into further details in the Related Work section and comment on some of these applications as well as the academic papers.

Chapter 2

State of the Art

This chapter will focus on providing insight in the current state of both existing client applications and academic papers regarding the subject of Twitter conversations. Due to the popularity of the platform applications come in a wide variety ranging from simple proof of concept/demo applications to businesses.

2.1 Background

The Internet has become an immense platform of communication and information with an estimate of 3 billion people online[1] and an exponential growth, it is offering a voice to millions of people that have access to it. Creating content which is accessible by everyone in the network is possible with little or no barriers. Anyone can report on events, share their thoughts and ideas and because there are no limits to this process the results are massive amounts of information. Facebook[2] and Twitter[3] are two examples of Internet platforms that have made it increasingly easier for people to generate content in a multitude of different formats: from text and pictures to rich media such as audio and video.

People send on average 50 million tweets with a peek record of 6939 TPS record. Facebook has even bigger numbers, in an average of 20 minutes, 1 million links are shared, 10 million comments are posted and 1,6 million wall posts are made[4]. These are just two examples of popular social media websites and the rate at which content is being generated.

At the same time Facebook offers no way for a user to search through those comments and posts.

People communicate in short gists with the help of special annotations made possible on the platform. Mentions are a way of including another Twitter user into the conversation, they are formed by prepending the character "@" to the string that represents the user. Hashtags are a method of creating channels of communication, they are a way to distill the idea or felling of your tweet to a single word, and by doing so you ensure the inclusion of your message to a certain ongoing conversation. Hashtags are created by prepending the character "#" in front of words. Users can also reply to tweets, their message is grouped with the one they are replying to and context is preserved this way. Due to its short message format of 140 characters per

[1] http://www.internetlivestats.com/internet-users/

[2] www.facebook.com

[3] www.twitter.com

[4] http://highscalability.com/blog/2010/12/31/facebook-in-20-minutes-27m-photos-102m-comments-46m-messages.html

message, Twitter has become a popular micro blogging platform for reporting on news and events as they occur. As a user you can always view the 10 most popular hashtags in different regions around the world or worldwide and participate in the conversation. You can also search for a certain query and messages that match gets returned either if it contains the string as a hashtag or in the message body.

The massive amount of information being generated especially on popular topics make it difficult to keep track of conversations as they happen. Unlike Facebook where the people you interact with are mostly people you know and that number can be within reasonable limits, on Twitter there are no barriers in communication and you have access to all the messages produced by every user of the platform. Even though hashtags help filter conversations they are too inclusive, there are no constraints over how to use them or how many to use so messages are included to any conversation as long as they have the hashtag.

There are a number of services that use Twitter data and attempt to solve some of these problems. We will be presenting some of them in the following section.

2.2 Existing Solutions

2.2.1 Analytics

There are a number of analytics services that provide information regarding Twitter data. Most of them are businesses which offer information about the engagement of followers with the content created. Their goal is to help increase the visibility of tweets for businesses and therefor the metrics are related to the followers and focus less on exploring content.

This is also the solution offered by the Twitter analytics [1] some of the information it provides is the number of user that views your messages, how many new users are now following your account.

One such example is SproutSocial [2] which allow you to publish content from their application to Twitter, monitor your content for engagement and offer analytics on the users which interacted with your content.

Another example that tries to solve a similar problem to Streamer is TweetArchivist [3]. You are able to query specific time frames and see top users and words related to certain search terms, as well as the most shared URLs and the most influential users that have send messages. Influential users are users with a large following, their messages reach a large number of people, get viewed, retweeted and shared by thousands of other users.

TweetMotif[4] retrieves tweets using the Twitter API based off of a user provided query. N-grams are a continuous sequence of items from a given document, they can be syllables, letters or words and n denotes the length of the group. They use n-grams to extracts a certain number of topics and groups messages behind those topics, therefor giving an overview of what people are saying.

2.2.2 3rd Party Clients

There are a number of 3rd party clients. They allow for filtering of content based on a particular hashtag and popularity (this is rated by number of retweets and favorites). This is a good alternative for finding popular opinions, you can judge it by how popular that certain tweet is

[1] https://analytics.twitter.com
[2] http://sproutsocial.com
[3] https://www.tweetarchivist.com
[4] http://tweetmotif.com/about

but it conveys either the voice of popular users which get lots of favorites and retweets or some tweets which happen to gain popularity by accident.

2.2.3 Trends

Twitter website offers access to world-wide trends and also custom trends. First off world trends represents a list of key words present in tweets in a certain region. This allows you to browse all the tweets with those key words in real time. You do not have any other type of control over the data. The data is not grouped by any other means so exploring it means going through each tweet and reading it and taking into account the volume of tweets some trends may produce (as presented in the introduction of this chapter) this task may be impossible.

2.3 Related Work

This section is concerned with similar work in the field of data mining and all related to Twitter conversations and topic trends. They are pushing the limits for the information and insights we can gather from analyzing online conversation and how to achieve this most efficiently.

Politics, Twitter, and information discovery by **Moritz Sudhof**. [1].
The aim of the paper is to cluster Twitter users into groups based on the opinions they expressed regarding a political controversy. The corpus is fixed and contains tweets from the time the events occurred, they have been selected due to using the same hashtag specific to the event.
Several different attempts are made at clustering the users using different methods. *Tf-idf* and *odds weighting* are used to extract relevant key words from messages. Multiple keywords shared between tweets are an indication of how similar they are and thus link the users together. *Mentions*, referencing one or more users in your tweet, are also used. Mentioning someone in your tweet means that they are relevant to your opinion or somehow involved.
Finally *hashtags* are taken into consideration the idea behind it being that users who send out messages using the same hashtags share similar opinions, again the more hashtags users share the similar they must be.

Topical Clustering of Tweets by **Kevin Dela Rosa, Rushin Shah**. [2]
The scope of this paper is to classify Twitter messages into different categories. The authors consider hashtags an approximate indication of the message topic and use it to improve results. The topics categories in which the messages are sorted are predefined, and the corpus is fixed, composed of selected tweets that cover the predefined topics.
Before being able to cluster the messages they undergo an intermediate processing step. The processing step includes normalization in which several variations are experimented: tokenization, removal of rare terms, conversion to lowercase each in different combinations and results are tracked.
Both unsupervised and supervised methods are used. K-Means is used in combination with TF-IDF as a weight function for the unsupervised clustering. Rocchio classifier is used for the supervised clustering. Results are compared and it is noted that the supervised method has better results.

TweetMotif: Exploratory Search and Topic Summarization for Twitter by **Brendan O'Connor, Michel Krieger, David Ahn** [3]

[1] http://web.stanford.edu/group/journal/cgi-bin/wordpress/wp-content/uploads/2012/09/Sudhof_Eng_-2012.pdf

[2] http://www.cs.cmu.edu/ kdelaros/sigir-swsm-2011.pdf

[3] http://brenocon.com/oconnor_krieger_ahn.icwsm2010.tweetmotif.pdf

The paper presents the implementation details of TweetMotif. A platform that allows fetching tweets from Twitter Search API, generates 2-3 words topics from the newly formed corpus and associates messages to these topics.
The interface allows for a recursive drilldown into topics the goal being to offer a concise summary of the topics generated. Topic generation is achieved using n-grams with certain heuristics such as disregarding unigrams that are function words, or bigrams, trigrams that cross syntactic boundaries. Topics are merged and their sets of messages are combined. The user is presented with a limited number of topics to preserve cognitive load.

A survey of text clustering algorithms by **Charu C. Aggarwal** and **ChengXiang Zhai** [1]
The paper discusses the problem of text clustering in a broader context, not only related to social networks. They present a very common problem that of misspellings or typographical errors in documents, a usual mistake in most online conversations. They advice on the removal of common terms that can skew results either using a list of stop words or using TF-IDF.
The paper is also concerned with the clustering of text streams. One of the methods they explore involves a weight function that rates newer documents better than older ones. So as time progresses old centroids might not prove relevant anymore and fade out. At the same time new centroids form up but not all might, this is why initially all new centroids start up as being treated as outliers.
In order to optimize and improve the quality of the clustering the paper introduces the method of *semantic smoothing* meant to reduce the errors caused by semantic ambiguity. The method works by extracting phrases instead of single words from sentences therefor reducing the probability of a word meaning multiple things. For example the word *star* might have different meanings but in the phrase *fixed star* it most certainly refers to a celestial body.

A Web-based Kernel Function for Measuring the Similarity of Short Text Snippets by **Mehran Sahami** and **Timothy D. Heilman** [2]

The paper handles the issue of measuring similarity for really short sentences such as search queries which might not have any terms in common so techniques such as cosine similarity do nothing to help. The proposed solution in the paper is to use each term in the queries we are trying to match up as web search terms. Each term is used to query the web for documents and therefor provide context on it, these are called *context vectors.* Therefor even though the terms *AI* and *Artificial Intelligence* would yield a cosine of 0, and therefor no similarity gathering a number of documents on the terms would help improve the context and compute a much better similarity for the two.
The main objective of the technique and the objective of the paper is finding ways to improve search suggestions to Google users.

A Latent Source Model for Nonparametric Time Series Classification[3] by **George H. Chen**, **Stanislav Nikolov** and Devavrat Shah

The aim of the paper is to provide an accurate estimation of topics that will become trending on the Twitter social platform, before they are declared trending by the website itself. This allows to turn data mining on Twitter messages into valuable insight into different social phenomenons.
The method proposed classifies topics into trending or not trending and does this by using a nearest neighbor classifier. They employ the classifier over time series weighted using weight majority voting.
The paper also provides theoretical justification of the efficiency of the chosen nearest neighbor classification for time series.

[1] http://www.charuaggarwal.net/text-cluster.pdf

[2] http://wwwconference.org/www2006/programme/files/pdf/3069.pdf

[3] https://cdn2.hubspot.net/hubfs/489432/docs/A_Latent_Source_Model_for_Nonparametric_Time_Series_Classification.pdf

The results of their experiments yielded a 79% success rate in detecting trends faster than Twitter and with a true positive rate of 95% and a false positive rate of 4%.

Chapter 3

Design

Streamer is designed as a data processing pipeline. It takes in messages from Twitter that match a certain provided query and at each step of the pipeline it modifies the message in some way either by removing parts of it that are not relevant to our process or annotating the content to provide further information for the clustering algorithm, resulting in the end with a classification of all messages into different categories all related to the initial query. At a high level overview Streamer is composed of two parts:

1. A backend consisting of several services that communicates with the Twitter API, retrieve tweets that match the query provided by the user, and perform parsing and clustering. The result is a list of messages annotated with the cluster id they belong to. This information is made available with the help of an HTTP server that exposes an endpoint. We have chosen JSON as the format to export the data via the API.
2. A frontend which is responsible for taking the JSON and rendering clusters of tweets as well as providing an interface for the user to explore the conversations by seeing all related messages, view the original message to get context and replies and ability to visit the user profile.

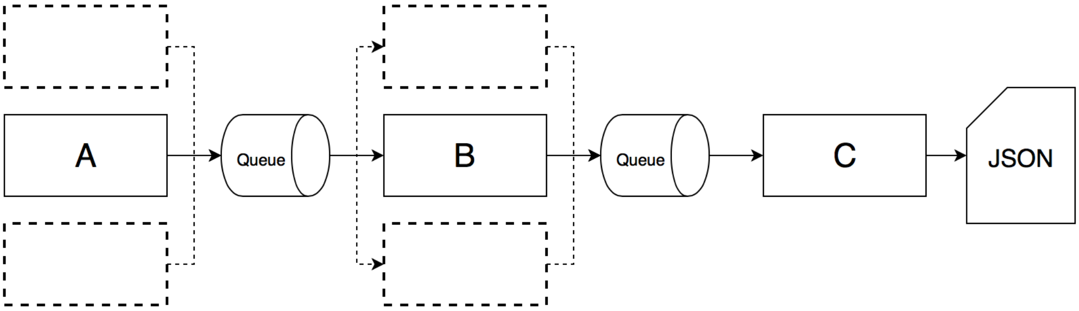


Figure 3.1: Design for the data processing pipeline

The data processing pipeline is composed of three main parts. Data acquisition (marked with A in the design) is responsible for fetching data that will later be processed, the component is agnostic of the data that passes through but in our case it is Twitter messages (tweets). It uses a library [1] to communicate with Twitter API, it retrieves the tweets and stores them in a

[1] http://twitter4j.org/en/index.html

queue. The queue library is provided by Apache Kafka [1]. As shown in the design, the reason for using queues between intermediary steps is that they allow for the producer and consumer to operate and different frequencies. The data acquisition segment could launch several clients regardless of what is happening further down the pipeline.

The second section (B), data processing, parses the raw tweets and converts them into shorter messages with key terms. Messages are read from the queue that is being filled by the data acquisition layer (A). The processed messages are written to a new queue, thus allowing this layer to scale just as the previous one. Tweets are parsed and using StanfordNLP library [2] each word is categorized with its own part-of-speech tag. Tags such as personal pronouns, possessive pronouns, prepositions, conjunctions are removed because they are too common. We could easily spin up several clients that consume messages because reading and writing is performed using two queues and thus the layer is decoupled from the other components of the pipeline. One issue related to parsing conversations especially ones from social media is the jargon used and possible spelling errors. This issue is exacerbated by the fact that Twitter conversations have such hard limits on the number of characters, on overage a message does not have more that 10-12 words.

The last part that processes data is responsible for clustering the messages based on the keywords generated in the previous step. The clustering algorithm uses K-Means with cosine similarity as a distance function, which I will go into more detail in the following section.

The visualization (Streamer-Frontend) is rendered in the browser. This offers the advantage of being able to explore the profile of Twitter users and see the messages and their context. It works by polling the webserver that is serving a JSON file through it API. When new data becomes available it renders the clusters and the corresponding adjacent nodes. The polling process will continue in the background. From the user interface you are able to see the clusters and quickly identify large clusters. You are able to see all the clusters that belong to it either by hovering over the nodes or clicking the cluster and getting an expanded view with all messages.

3.1 Implementation

In the following paragraphs I will go into further details on how the system is implemented.
The implementation is done in Scala[3]. The reason behind this choice is the fact that Scala enables us to use a functional programming paradigm and at the same time provides a type system that makes implementation easier. Many of the operations in the application include transformations of lists of messages, something that functional programming is very good at. At the same time the Scala source code is intended to be compiled into Java bytecode and run on the JVM. This allows us to include any Java library into the project as Scala was designed with Java interoperability in mind.
Another benefit of Scala is their implementation of parallel processing into the standard library. The aim of the language designers was achieving a high level abstraction that is easy to use thus achieving efficient parallel computations over collections in a transparent manner.

```
// Example of using parallel collections in Scala

list.map(_ + 42) // regular, sequential map over a collection
list.par.map(_ + 42) // parallel processing of the collection
```

Listing 3.1: Example of parallel collection usage in Scala

[1] http://kafka.apache.org
[2] http://nlp.stanford.edu/software/index.shtml
[3] http://www.scala-lang.org

Using a similar approach we can speed up message parsing and also the clustering step. This decision has had beneficial results for the overall processing time of the Twitter messages. We will go into further details about the running time and speed benefits of parallel processing in the pipeline in the Testing and Evaluation section.

Accessing the Twitter API requires a developer account and an application created on their website dev.twitter.com [1]. The application gives you access to public, user or site streams. We will be using the public streams which returns data flowing through Twitter in real time given a certain array of keywords. This is the most useful endpoint for our data mining usecase.Using the provided API authentication tokens you can configure twitter4j library to retrieve tweets that match a specific keyword (or multiple keywords). The library comes with OAuth support and handles authentication with the endpoint. All messages received are passed to queue.

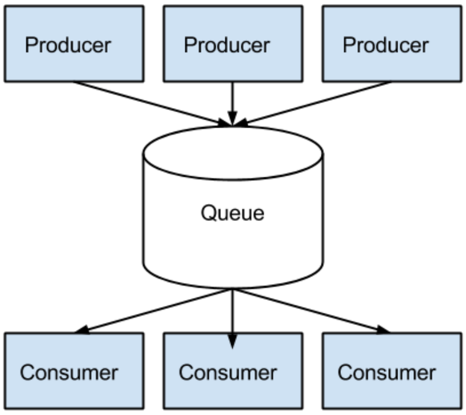


Figure 3.2: Design for queue processing in the pipeline

The messages retrieved are stored in a queue provided by Apache Kafka[2]. The queue, has a configurable storage period for the messages which allows us to use it for temporary storage. Communication between producers, consumers and the queue is done via a binary protocol over TCP so message delivery is always assured. The protocol does not require a handshake step in order to get or put messages in the queue, after a socket connection is established the client simply writes the messages it requires as request and then reads them. Kafka also guarantees message order and load balancing over a group of consumers but these features are not of interest for our project. Using a queue provides an advantage over the fact that messages arrive and are consumed at different frequencies. Depending on the popularity of the keywords specified

[1] https://dev.twitter.com/streaming/overview

[2] http://kafka.apache.org/documentation.html#introduction

tweets can come in at different rates. Based on a series of tests the rate of messages has been up to 150 per minute. At the same time the StanfordNLP has a parsing speed of one tweet per second. Using a queue also means that it is possible to start several consumers and producers at the same time. A consumer is concerned with getting the messages out of the Apache Kafka queue and placing them in a a new queue where they will eventually be parsed. The Apache Kafka documentation lists "Stream Processing" as an ideal use case for the library, and the processing of the Twitter API feed is exactly this sort of use case, thus confirming our design decision.

Before applying the clustering algorithm the tweets are first parsed. Parsing the tweets means removing all non alphanumerical characters or punctuation: such as unicode characters. One of the reasons for removing non-alphanumerical characters is that the StanfordNLP library cannot parse emojis [1].

The new messages are annotated using StanfordNLP part-of-speech tagger library. This library reads the text and assigns a part of speech or other token to each word. The set of part of speech tags follows the Penn Treebank Project [2]. The resulting output is a list of tuples containing of the word, its POS tag and its level in the dependency graph. The dependency tree is constructed by drawing an edge between a token and the all others that it determines. The tagger has an accuracy of 97.24

The next step is to filter out words based on the part-of-speech tagging. Tags such as personal pronouns, possessive pronouns, prepositions, conjunctions are removed because they are too common and might introduce false positives for the clustering algorithm.

These parsed messages are pushed to a new queue. The reason for this is to completely decouple the 3 different stages of the pipeline:

1. Retrieving messages in realtime from Twitter using its API.
2. Tagging the messages with their part-of-speech tag and using this to filter out common terms.
3. Running the clustering algorithm using the parsed messages as input.

The final stage of the pipeline takes all the parsed messages from the queue and applies the clustering algorithm.

A clustering algorithm will group or set of documents (tweets) into subsets. These subsets are called clusters. The goal of the algorithm is to generate clusters that have sufficiently similar documents in one place, being internally coherent, while at the same time given any two clusters they should be different enough from each other.

Clustering falls in the category of unsupervised learning, meaning that there is no human involvement in classifying any documents beforehand and everything is decided depending on your document set. Usually in clustering algorithm the key metric is distance, having to cluster points in a two dimensional space and therefor the Euclidian distance is used. For document similarity we will employ a different function, the cosine similarity, the following paragraphs will provide more details.

The clusters generated have no relationship between them, this is called *flat clustering* and we suggest this by having the clusters at equal distance from each other. It is also a *hard clustering* assignment meaning that any one message will belong to just one cluster, the opposite being soft clustering where each message belongs to a cluster with a certain degree of certainty similar to fuzzy logic.

A hierarchical clustering would output information in a more structured form as opposed to flat clustering, it also has the added benefit that it does not require a predefined number of clusters beforehand. The reason we did not go forward with choosing it over flat clustering is that all these benefits come with a performance cost. Hierarchical clustering having at least quadratic

[1] An emoji is a pictogram, similar to ASCII emoticons, which have been incorporated into Unicode meaning a wide adoption in online communication. Emoji symbols are two-byte sequences and support for them in browsers and mobile devices varies.

[2] https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

time was not a good fit for clustering real time streams of messages.
To measure the quality of our clustering algorithm we used an evaluation method called *purity* further described in the Testing and Evaluation section.
The algorithm used for this step is K-Means clustering and the distance function is the cosine similarity. We chose K-Means because of its simplicity and efficiency. The K-Means clustering algorithm works by starting the iterations with a random seed consisting of randomly selected points (in our case messages), we run the clustering using different starting seeds and choose the highest value out of all the runs as our best clustering option. This in turn will affect the overall running time of the application as we will see. Depending on our goals either speed or precision we can choose to just rely on the first random seed provided. The name K-Means, first introduced in **Some methods for classification and analysis of multivariate observations** by J. MacQueen. K-Means is an important clustering algorithm; its objective is to minimize the average distance between a point and its cluster center where a cluster center is the mean of all points in that cluster.

3.1.1 Clustering Algorithm

The first step of the algorithm is to compute the TF (term frequency) and IDF (inverse document frequency) for each of the tweets. TF-IDF is a numerical statistic intended to provide information about the importance of a word in a document. TF is a direct measure for the number of times a word appears in a document while IDF helps scale down that measure for words that tend to appear often and scale up for words that are rare.

$$\text{TF(D, t)} = \frac{\sum \text{occurrences of t in D}}{\sum \text{number of terms in D}} \tag{3.1}$$

The TF of a term t in a document D is the total number of occurrences of that term divided by the total number of terms in the document. A document in this case would constitute a tweet. And the terms we are computing TF for are all the terms that remain after the filtering in the previous stage.

$$\text{IDF(D, t)} = \log\left(\frac{\sum \text{total number of terms in D}}{\sum \text{total number of occurrences of t}}\right) \tag{3.2}$$

The IDF of a term t in a document D is the total number of terms in D divided by the total number of occurrences of t in D. The document in this case constitutes all of the available tweets that we are attempting to cluster. The number of occurrences is also computed taking into account all the available messages.
Using TF and IDF we transform each tweet into a vector with weights for each of the terms. This way we can use the cosine similarity as a distance function in the clustering algorithm. Cosine similarity is a way of computing the similarity of two vectors by measuring the cosine of the angle between them. The cosine function output is in the range [-1, 1] where -1 means that the vectors have opposite directions and 1 means that they have the same direction.

$$\text{cosine similarity(A, B)} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{3.3}$$

The A and B vectors in our equation constitute two vectors i.e. two tweets that have been turned into vectors after computing their TF and IDF. And the result of the *cosine similarity* function tells us how similar the two vectors are, in return how similar two tweets are.

The clustering algorithm uses the K-Means clustering algorithm. To define some terms that are used in the implementation of the algorithm, a cluster center is the mean of all points in that cluster, or more formally

$$\mu\ (\mathrm{w}) = \frac{1}{|w|} \sum_{x \in w} \mathrm{x} \tag{3.4}$$

Where w is the cluster. Ideally the clusters that are formed do not overlap. To get a sense of how well a centroid, the mean point of a cluster, represents all of its members, we define RSS *residual sum of squares*

$$\mathrm{RSS}\ = \sum_{x \in w_k} |\ \mathrm{x}\ \text{-}\ \mu(\mathrm{w})|^2 \tag{3.5}$$

The objective of the algorithm, and therefor our objective is minimizing this RSS function. By reducing this value we reduce the average squared distance which is a measure of how well points in a cluster are represented by their centroid.

The algorithm starts with an initial set of clusters chosen at random from all the tweets. It then uses the distance function (the cosine similarity described above) to assign the rest of the tweets to those clusters. For each cluster now formed it computes a weighted average. These steps are repeated only now we use the weighted average centroids instead of the random points until the centroids remain the same between 2 consecutive steps or until an upper bound of steps is reached. This upper bound is added in to ensure the algorithm halts if it fails to reach convergence in a fixed number of steps.
There are a number of termination conditions we can apply:

1. Terminate when RSS falls below a certain ϵ
2. Centroids do not change between iterations
3. Assignments of points to clusters do not change over iterations
4. A fixed number of iterations has completed. This will always ensure the termination of our algorithm but it might also affect the quality of our results.

Several issues we might run into while running the algorithm are data outliers. An outlier is a data point on a graph or in a set of results that is very much bigger or smaller than the next nearest data point. Therefor such points do not fit well in any cluster being to far away from the rest of the data. One such example would be having a cluster with only one message.

Figure 3.3: Example of cluster rendering with outlier

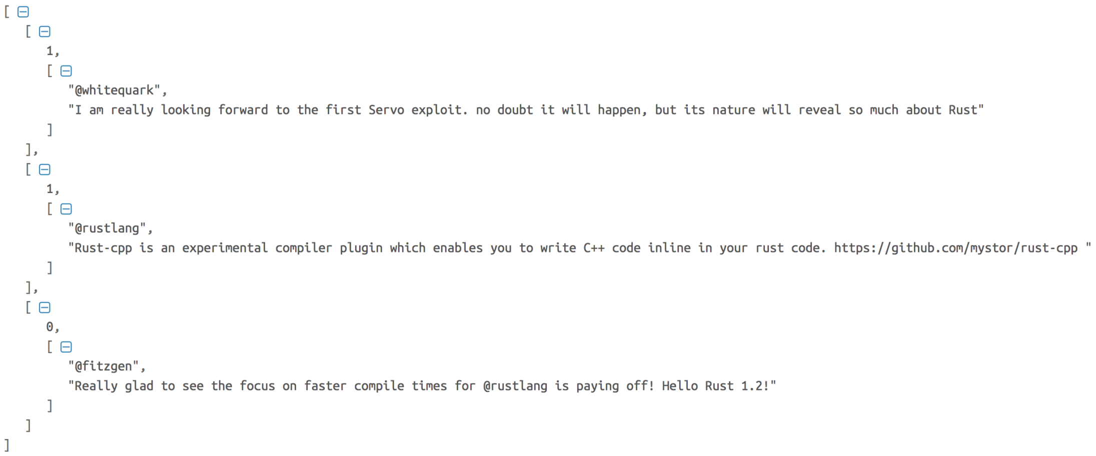
Another such example of clustering done sub optimally is having a cluster with no actual messages in it. This is due to the methods used to select the initial seeds for the clustering algorithm. Selecting a "bad" seed, by which we mean an outlier will always lead to sub optimal results. Heuristics used to improve results are using the RSS function to choose the seeds. By using this method we would go through different seed candidates and chose one. Another possible method is simply removing outliers from our sample data.
Different assignments yield different clusters which in turn have different weights. A higher weight means better similarity between the node and the cluster they belong to and therefore a better result. This similarity is computed with the help of the *cosine similarity* function after the K-Means algorithm has finished its assignment. For better results several iterations of the algorithm are used and the best score is chosen, at the cost of time spent clustering.
Another problem when using K-Means is choosing the correct K value, number of clusters. Most papers suggest having domain knowledge over the data that is being clustered but in our case it is not entirely possible due to the fact that data is coming in real time. Using RSS function and attempting to select a K value that minimizes it is an naive approach because RSS will reach its minimum for K = N meaning one cluster for every tweet in the dataset.
A different approach[1] for choosing K is to initially start with $K = 1$ and then increment that at each step until the optimum K is found.

$$\mathrm{K} \ = \min[\ RSS_{\min}(\mathrm{K}) + \lambda\mathrm{K}\ ] \tag{3.6}$$

The first input of the equation measures *distortion* or in other words how much the documents in the current clusters have deviated from the centroid they are associated with. This gives a quantifiable measure on the quality of the clusters. The second term is set to monitor introduction of new clusters into the system, the λ term is a weight factor and it introduces a penalty for the solution when new clusters are added. This is a generalized approach to the previous solution, as we can see that setting λ to equal 0 will yield the best solution for $K = N$. The article suggests applying a λ that yielded good results for similar data types in the past and adjusting that based on the results. They also go into further details on how to compute the optimum value for λ.
While this method does improve results, it removes all cluster outliers the ones having zero, one or a few tweets associated with them, the computational cost is increased. The performance hit is not worth the benefits and just by using the frontend application it is easy to identify and disregard such clusters.
The result of the clustering algorithm as well as the tweet message and tweet author are combined and converted to a JSON data structure. This is in turn written to disk to be consumed by the endpoint accessing the server.. Using *Finagle*[2] an open source library from Twitter that provides an HTTP server implementation we can create the API endpoint.

In this example output we can see how the JSON response looks like. An array of tuples consisting of the cluster id in the first position and the tweet on the second position. We can just as easily append additional information to the output such as tweet statistics: number of retweets, number of favorites and so on.
Having the data accessible via HTTP it can easily be consumed by any application regardless of the programming language used, also the reason we have chosen JSON as a format is because it is lightweight and most languages have an implementation of a JSON parser. The connection the the API endpoint is not kept alive, all clients are expected to poll the endpoint and react to changes. As an optimization the server could reply with status code 304 Not Modified [3] this would not include a message body and would let the client know that no new data is currently available. This would reduce the payload that has to be transmitted. On the client side this

[1] http://nlp.stanford.edu/IR-book/html/htmledition/cluster-cardinality-in-k-means-1.html
[2] https://twitter.github.io/finagle/
[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html



```
[ ⊟
   [ ⊟
      1,
      [ ⊟
         "@whitequark",
         "I am really looking forward to the first Servo exploit. no doubt it will happen, but its nature will reveal so much about Rust"
      ]
   ],
   [ ⊟
      1,
      [ ⊟
         "@rustlang",
         "Rust-cpp is an experimental compiler plugin which enables you to write C++ code inline in your rust code. https://github.com/mystor/rust-cpp "
      ]
   ],
   [ ⊟
      0,
      [ ⊟
         "@fitzgen",
         "Really glad to see the focus on faster compile times for @rustlang is paying off! Hello Rust 1.2!"
      ]
   ]
]
```

Figure 3.4: Example of JSON output, cluster id, user name and tweet content

can be improved by using an exponential backoff approach for long polling. The polling period would always double when a 304 status code is presented and it would reset to a lower default value when the endpoint has new data available.

3.1.2 Data Visualization

Visualizing the data is made possible in the browser using JavaScript. We chose the web as a platform because it made more sense for a number of reasons:

1. It is easily accessible from a number of different devices such as laptops, phones and tablets much like the content we are clustering.

2. There are a multitude of libraries that implement visualizations, graphs, plots for JavaScript. Transforming the JSON file exposed by the API in a graphical representation is faster this way.

3. Using URLs the tweets can easily be traced back, the user profile and be viewed and content can be explored without having the implement it.

The visualization for Streamer named **Streamer-Frontend**[1] was build using two libraries.

1. React[2] an Open Source JavaScript library from Facebook that takes care of updating the DOM (Document Object Model) every time new data comes in.

2. D3.js[3] which stands for Data Driven Documents, this is an Open Source library used for plotting, graphs and other types of data transformations.

Both libraries work really well together because they offer a similar declarative API for describing the data flow and how the overall interface should look like. In D3.js you describe how a cluster element looks like and the function that should be used to position the elements. When the data input is populated with messages it will use the provided information to draw the clusters.
React allows you to break down your interface into smaller unit components. The idea behind this concept is that each component should be a working, functioning part of the big application. In the same way D3.js allows describing how the graph should look programatically,

[1] add link to Github
[2] https://facebook.github.io/react/
[3] http://d3js.org

React advocates for small reusable components that encapsulate their internal logic. Using this pattern it is easy to achieve a fairly complex data visualization interface to display our clusters.

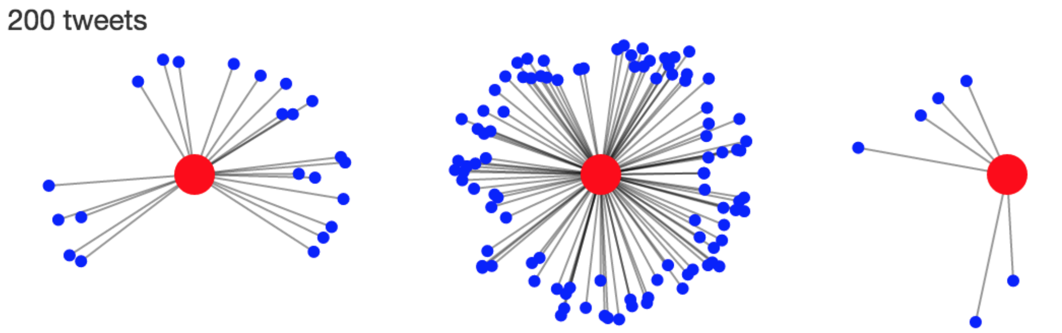


Figure 3.5: Example of cluster rendering

Streamer-Frontend polls the server API at a fixed time intervals and gets the latest version of the tweets as well as their respective cluster. Using an appropriate data structure we group the tweets together based on cluster id and draw the clusters to the webpage. Drawing is done using an SVG element with the help of the D3 library. The library provides an easy to use API to draw graphics and also provides methods for event listening. We attach events for click, hover and so on, allowing us to provide extended interaction with content. We can therefor provide the tweet message when a tweet point is being hovered on, or expand the cluster when clicking the centroid and providing a full list of tweets.
Because the frontend application continuously polls the backend to fetch the latest version of the data, we would get an unpleasant refresh effect of the page as the clusters are redrawn every time, even when the data has not changed (in the case when two consecutive polls return the same information). To prevent this from happening we first check to see if the new data available has a different length from the previous payload. We expect to always get more tweets to cluster in step n than we previously had in step n - *1*. If that condition is not true than either the request returned the same payload or there are no new tweets to cluster, in both cases we do not redraw the canvas.
The interface presents the user a more meaningful representation of the data: tweets that belong to the same cluster are shown drawn together, also it is possible to explore the cluster and see all the messages that compose it.
The clusters will automatically update when new data comes in without the need to refresh the webpage. The application polls the API and when new messages become available it will redraw the clusters. A small optimization to the polling requests has been added: the polling interval doubles every time the request does not return with new data. This prevents the server from receiving too many requests if multiple Streamer-Frontend clients are running and also works similarly to the data processing step which will always take longer as more tweets are fed into the pipeline.
An explanation of Figure 3.3: We have tweets (blue dots) and centroids (the red dots). The centroid is only used as a visual cue, making it obvious which tweets belong together. Hovering over any of the blue dots to view the content of the tweet in an overlay that will appear or clicking on the red dot reveals a side menu on the right side with all the tweets that make up the cluster as well as a total. There is no relationship or hierarchy between neighboring clusters just between a centroid and the tweets associated with it. We tried to emphasis this by drawing lines between the tweet and its centroid and by placing centroids equally distanced from each other.

3.1.3 Deployment

For automated deployment of the project I used Docker[1], it is an Open Source project that runs processes inside of isolated software containers. A container is similar to a virtual machine but avoids the overhead of it by sharing the same kernel as the host. Dockerfiles are configuration files for Docker images, they are scripts that describe the steps necessary to create the environment for the application. Docker allows to generated any number of such images on the host machine with different configurations and requirements, each one created receives a unique hash used for identification when it is required to run it. A Dockerfile for Streameris also available as a gist[2]. Some of the requirements of the project that the image contains and will automatically generate a container with:

1. Scala 2.10
2. sbt 0.13.1 (Scala built tool) - Allows for task automation, running tasks and access to the Maven Central Repository for installing and updating packages
3. Kafka 2.9.1
4. Java 8
5. These are are all running on a machine with Ubuntu 12.04

The script installs the required software with appropriate versions, updates all packages and it also created all the necessary configurations. An image created using the Docker syntax generates a read only template, from which you start the Docker containers. The main advantage of a docker container is that once configured and fully functioning it is a completely reproducible environment on any machine and no further adjustments are needed. An advantage over virtual machines which in turn have the same benefit of reproducibility is the file size, a VM can easily exceed 1GB in size because it contains the whole environment, OS and other pieces of software while a Dockerfile is just a file. When building the Docker image it is then the software that pulls in all the dependencies in the order specified in the file. This allows us to easily share the environment and deploy it with little requirements for the host machine (just a Docker installation).

```
docker run -itp 8001:8080 ee7c18b2a566 /bin/bash
```

Listing 3.2: Starting a Docker container

This is an example of starting a Docker container. We specify a hash which points to an existing Docker image from which our container will be created. We also specify -p which is the *expose* option, which makes port 8001 from the container accessible outside the container through port 8080 of the host machine. Through this method we are able to expose the API that our server has set up inside of the container to the outside world.

Using this setup benchmarking was greatly simplified and we were able to try out different parameters for the clustering algorithm at the same time and observe how that affected the clustering performance and quality. Much of the testing involved in the development of the application was done on DigitalOcean. They are a PaaS[3] that offer virtual private servers, and one of the most useful feature they provide is the ability to start up several machines at the same time capable of running the project, each with custom performance capabilities. The servers used were equipped with 4 CPUs Intel(R) Xeon(R) CPUs and 8GB of RAM. Using virtual machines was an ideal setup for benchmarking because even though we used several different instances we were able to use the same snapshot across all machines thus ensuring a

[1] https://www.docker.com

[2] https://gist.github.com/piatra/53c0b6d185d10eeeaf8b

[3] Platform as a Service

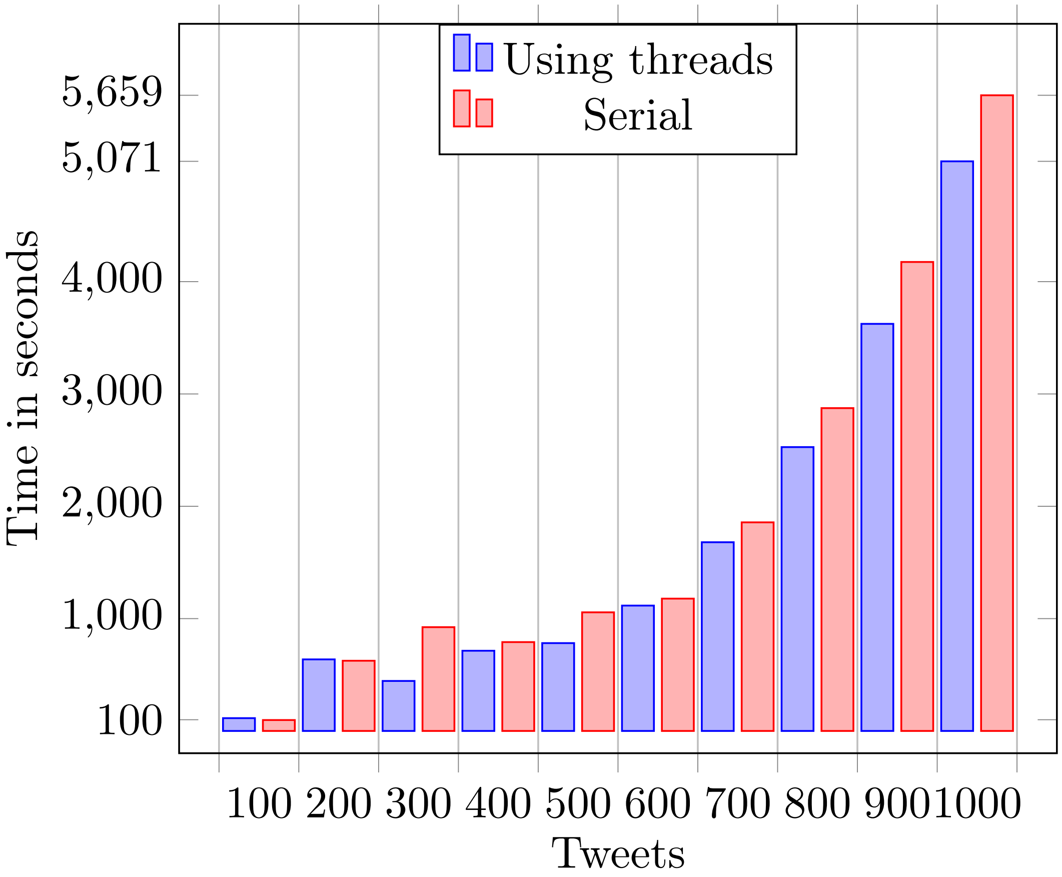
uniform testing environment. Once a container is started it will automatically begin fetching new data from Twitter and after an initial waiting period (in which the queue fills up with messages) the processing pipeline starts as well.

The versioning software used was Git[1]. Git is a distributed revision control system and one of the most popular and widely adopted in the industry. We chose to use it because we were most familiar with it and due to its popularity it is easy to get support. It is also the preferred way of delivering software updates to the containers. The images we have built contain an outdated version of the project (the most recent one at time of creation), but through the initialization script any container performs a *git pull*[2] before starting Streamer to ensure latest version is running. It was also ideal during development for refactoring and introducing new features.

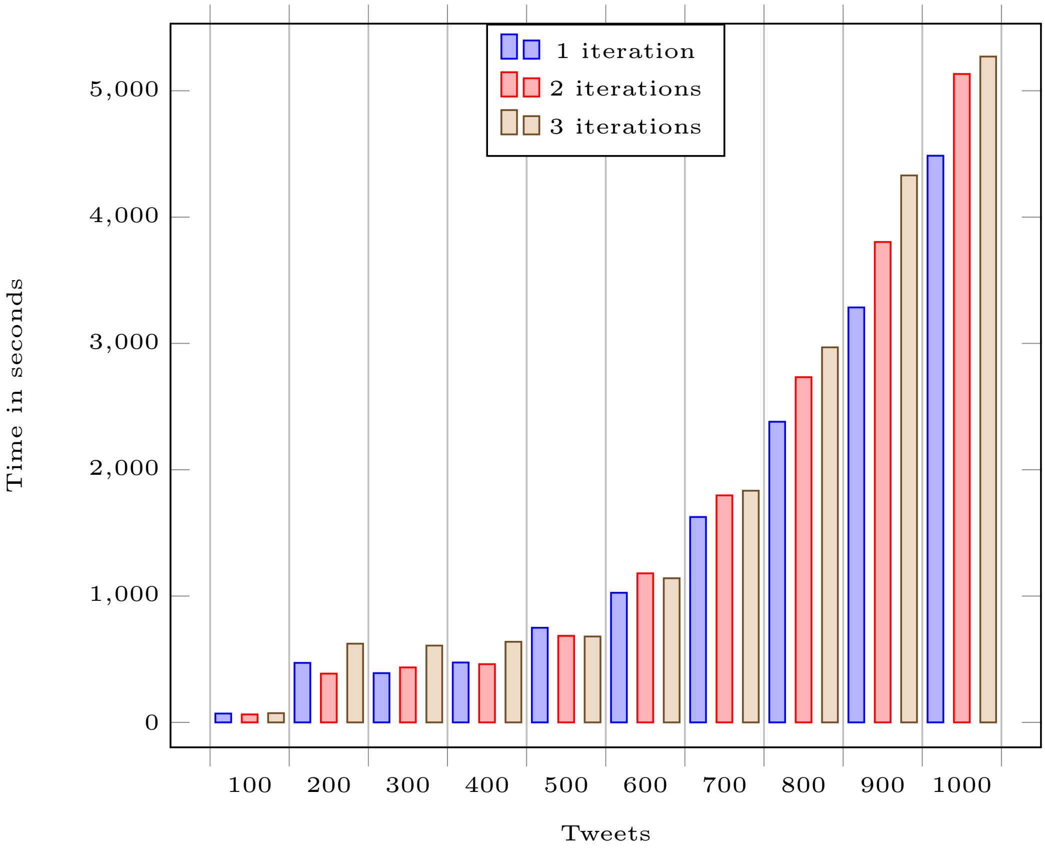
3.2 Testing and Evaluation

One of the advantages provided by the Scala programming language is its ease of manipulating collections using functions familiar to most functional programming languages such as map, reduce, filter, fold. This was an advantage that made it a perfect choice for manipulating large collections of messages. At the same time Scala provides *parallel collections*[3]. These are special collections included in the standard library that offer a high level API that facilitates parallelization. Calling the **.par** method on a regular collection such as **List**s or **Vector**s transforms it into a parallel collection and one can continue using that as a regular collection but now the transformations are done in parallel.
Below is a chart that outlines the performance improvements of using parallel collections. For the final set of data containing 1000 messages the performance improvement is of 588s (9,8 minutes).

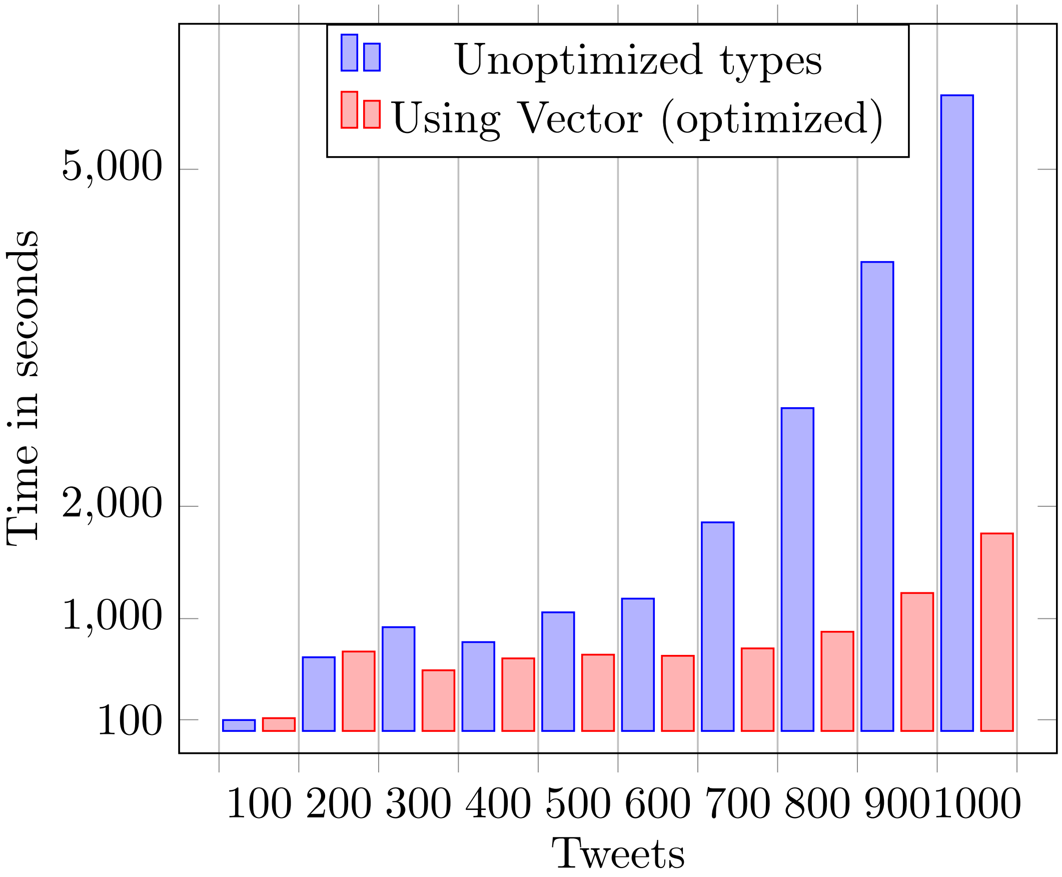
[1] https://git-scm.com
[2] This command incorporates the latest changes from a remote repository into the current working branch.
[3] http://docs.scala-lang.org/overviews/parallel-collections/overview.html



As mentioned in the **Implementation** section, each cluster generated by the K-Means algorithm has a different weight and running the algorithm multiple times can yield better results, at the cost of running time.



Above we can see how running 1, 2 or 3 iterations impacts the running time.
An advantage of using queues means that the consumer and producer processes that interact with the queue are completely decoupled. Therefore we are able to spin up several producers and consumers at the same time. Producers could either be Twitter API producers that provide the tweets, this would speed up the cold start of the application, currently it waits for 6 minutes for the queue to fill up with messages. Another producer in the pipeline is the parser that handles part-of-speech tagging and filtering of messages, this consumes raw tweets and produces tokenized messages that are stored in another queue.
Types in Scala are another optimization that yielded improved performance. Parallel collections offer improved running time by delegating the work to several threads but using incorrect types can yield unfavorable results. **List**s in Scala are collections optimized for LIFO (last-in-first-out) type of access. Although random access is possible it comes with a performance loss. This disadvantage is also present when converting to a parallel collection *ParVector*, the overhead comes from the fact that all sequential collections such as lists have to be copied over to each thread. Using a different data structure improves performance especially when the quantity of messages increases.



From the chart above we can notice a 3x speed improvement when choosing the appropriate data structures.

In order to measure the quality of our clustering we used an evaluation method called *purity*. To compute *purity* we first compute the valid messages in each cluster. For each of our clusters we compare the messages they contain with the cluster's centroid. We use the weights generated by TF-IDF to be able to compare the sentences. All sentences that do not fall under a certain ϵ are considered to be incorrectly clustered. The number of correctly clustered messages is divided by the total number of tweets in all clusters and summed. The output of this computation is the *purity*.

$$\text{purity(C)} = \frac{1}{N}\sum_{k} max_i \; (w_k \cap c_i) \tag{3.7}$$

High purity is easy to achieve when we have a high number of clusters so this must not be the only measure of cluster quality. Another indicator is *Normalized mutual information* which determines how similar clusters are in relation to each other.

The benchmarking and results we achieved offered positive results suggesting that such an application can cope with the large volume of data and can produce relevant results.
Language specific optimization step further improved the running time of the project. We can see a clear trade-off between speed and precision: attempting to experiment with several different initial clusters from which the iterations advance in the K-Means algorithm leands to a better clustering while on the other hand if our main objective is yielding fast results we have to cut down on some of the precision. Further improvements can be made to the precision of

the topics by reducing semantic ambiguity with ngrams, this idea for the project is mentioned in the Further Work section.
Having gather enough information regarding the running time and performance of the application, the next section is concerned with drawing conclusions regarding the objectives set and what are results have managed to prove.

Chapter 4

Conclusion

The Internet allows anyone to create content, publish and spread information. Social platforms are lowering the barrier of entry and are making it especially easy for everyone to have a voice on the Internet and as a result millions of messages and content of all forms is generated daily. Making sense of everything, and keeping track is becoming difficult and therefore it is time for tools that help understand the content to evolve and adapt to these mediums and make content exploration as easy as it is to post a message.

Dynamic applications such as social networks generate large amounts of text data and having an application capable of clustering and aggregating all the content is important for keeping track of events as news are more likely to be first reported in the online medium before others catch up.

The aim of this project was to build an application capable of clustering in real time streams of text generated by public messages from Twitter. Such a tool allows us to easily track news, trending topics or social media events as they occur and we can easily detect popular messages or common clusters of opinion.

Streamer attempts to solve this problem of content discovery and exploration. Streamer tracks the topics being published and presents it to the user in a way that is accessible and easy to use. It creates clusters of messages by interpreting content and presents it to the user in a web interface that allows for him to browse through a large number of messages efficiently.

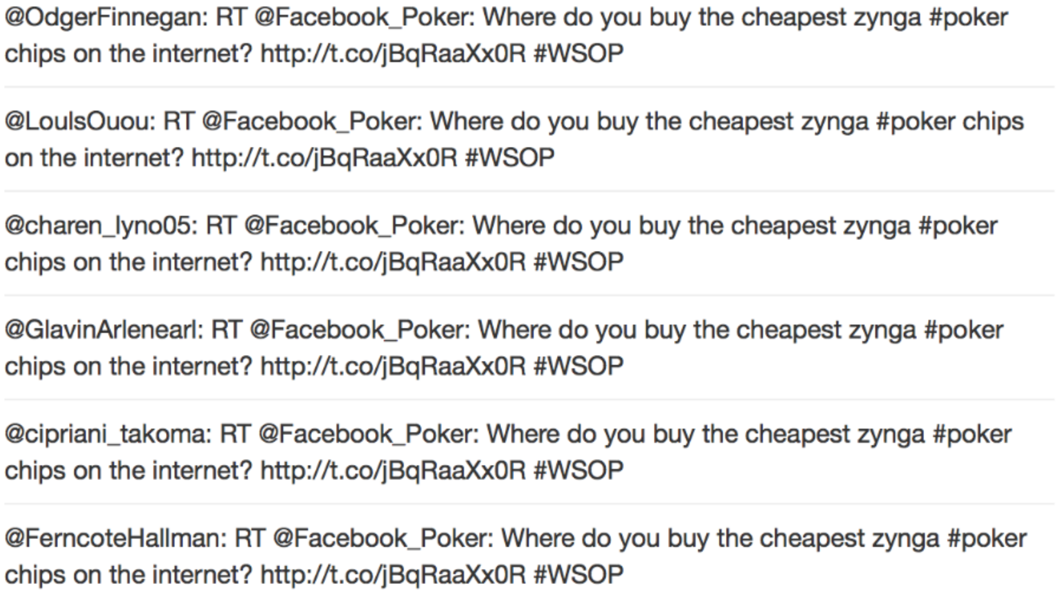
The feature that makes Streamer relevant for the fast passed rate of tweets is its ability to parse the messages in real time. The data is not based on an archived corpus of documents but on streaming tweets as they happen. This way popular events, news and messages get reported in the interface and the user is able to keep in touch with what is happening as events are taking place.

The goals we have set out for our project and managed to achieve:

- Based on an input query, get real time messages from Twitter using its API.
- Parse tweets as they arrive: tokenize the sentences and using part-of-speech tagging to make annotations, help filter messages and extract important information.
- Cluster tweets efficiently. For this we chose K-Means algorithm which employs the meta information gathered in the previous step.
- Set up a decoupled system that can easily scale through the use of queues which allow for different rates of consumption and multiple consumers that can process the workload in parallel.
- Present the information through an accessible medium: the web browser, with an easy to

use interface that allows the information to be explored.

Evaluating the results was done through calculating the purity coefficient for each cluster and thus computing internal coherence and by using the frontend Streamer-Frontend. We expanded the clusters and started exploring the different topics making sure the results made sense. One unexpected result that kept appearing in our experiments was discovering topic influencers. We define influencers as being people with large following in social media that also are really engaged with their followers through comments, retweets and favorites. The way it comes up in Streamer is that a lot of people will end up retweeting a certain tweet written by one of these accounts, and it will show up as a cluster composed of the same tweet. There is also an exception, meaning that cluster can be composed of a single tweet of an account with few followers if their message was in turn retweeted by an influencer. In this case the cluster is formed entirely out of one popular tweet.



@OdgerFinnegan: RT @Facebook_Poker: Where do you buy the cheapest zynga #poker chips on the internet? http://t.co/jBqRaaXx0R #WSOP

@LoulsOuou: RT @Facebook_Poker: Where do you buy the cheapest zynga #poker chips on the internet? http://t.co/jBqRaaXx0R #WSOP

@charen_lyno05: RT @Facebook_Poker: Where do you buy the cheapest zynga #poker chips on the internet? http://t.co/jBqRaaXx0R #WSOP

@GlavinArlenearl: RT @Facebook_Poker: Where do you buy the cheapest zynga #poker chips on the internet? http://t.co/jBqRaaXx0R #WSOP

@cipriani_takoma: RT @Facebook_Poker: Where do you buy the cheapest zynga #poker chips on the internet? http://t.co/jBqRaaXx0R #WSOP

@FerncoteHallman: RT @Facebook_Poker: Where do you buy the cheapest zynga #poker chips on the internet? http://t.co/jBqRaaXx0R #WSOP

Figure 4.1: Automated messages sent by bots

Due to the way the Twitter stream API works: it sends a percentage of all tweets currently being exchanged that match your query, we were able to make another interesting observation. A large number of Twitter topic clusters are formed from messages from bots. Twitter bots produce automatic messages usually with spam or promotional links. For the topics we tested on, mostly programming language topics, we noticed a large number of clusters related to job advertisements which turned out to be automated messages.

In the figure we extracted an example of bot messages that are using a popular hashtag (World Series of Poker) to advertise a game. This also provides some indication that relying completely on hashtags in order to generate topics and cluster messages is not always very efficient. Bots might use trending topics to their advantage: most Twitter clients offer limited discovery features and usually just show tweets that match a certain hashtag, by using the same one, spam such as this can end up in your message list.

The application proved successful in clustering conversations, some of the topics we have experimented with were movies, receiving a wide variety of reviews from social media. Another such topic was programming languages where clusters usually grouped into ones containing tutorial links or job advertisements.

Having completed this initial first step, there are now potentially more areas where we can work on improving our project. Improvements both in terms of clustering and the information we highlight and present to the user. The following section **Future work** explores some of these ideas, what it would take to implement them and the benefits they would bring to the project.

Chapter 5

Further work

We have presented a solution for clustering Twitter message streams that is able to process and scale for the large amount of data delivered by its API. The project lays the foundation for several lines of work to improve overall speed and precision. The system is perfectly usable and but requires that each user deploys its own version of Streamer. Although the use of containers for automatic deployment via Docker reduces this task to running a script, the overhead involved with managing your own machine or environment where this can be done is a drawback. Wanting to improve on this issue but also increase the overall performance of the project, we have identified some areas where it can be improved:

- As [3] suggests the clustering can be further improved by using n grams instead of matching documents just by using single terms. Single terms might skew results due to semantic ambiguity while having two or more reduces the probability of different semantical meanings. Using the StanfordNLP library which we already use in the project we are able to detect not only the part of speech corresponding to each term but also which term determines others. This information can be leveraged into producing more useful ngrams in the processing step. The new vectors associated with each document would still use term frequency to weigh the importance of the ngrams but reducing some of the semantic ambiguity should yield better results. This would also help cluster shorter phrases and messages since not all tweets use up the maximum amount of characters at their disposal and usually end up oddly clustered.
- Stream clustering improvements could be achieved as presented in [3]. Currently TF-IDF weight function is used to convey the importance of different terms. This method could be augmented to include the age of the documents. Current clustering issues include outliers based on new data coming into the system and at the same time old clusters becoming obsolete. To work around this problem the paper suggests taking document age into account, scaling down older documents and clusters that might not be relevant anymore and offering a higher weight to newer documents in the system. This way outliers can be promoted to clusters and slowly older clusters can be discarded.
- Further improving the clustering algorithm both in terms of speed and precision. Currently a bottleneck of the project is the part-of-speech library which has an average parsing speed of 1 message per second. This could be replaced with other methods for determining the relevant keywords in a message. One example is using G-test log likelihood to remove statistically insignificant terms and improve clustering precision. This solution could run on multiple threads and improve the overall performance, but tests are required to ensure the project does not suffer a loss of precision.

- Using a distributed solution over multiple containers or even multiple machines. The container used in deployment contains both Streamer and the queues that hold the data. Extracting the queues would mean they can be shared between multiple instances and would be oblivious to any restarts to the server and would not lose data.
- Expanding the pipeline to add multiple consumers for the data, using multiple part-of-speech-taggers, and having more than one producer that retrieves data from the Twitter API. This would allow the system to retrieve more data as well as improve the time it takes to process it.
- Use websockets to notify Streamer-Frontend of new data that has been clustered. It would improve the user experience: right now when the API has new data the whole interface is redrawn including the clusters meaning that clusters may change position on the page making it harder to identify them. With a websocket implementation Streamer-Frontend can use event listeners and simply append new information to the page.
- Adding a storage layer or a caching layer in order to speed up the results. The storage could be shared between all running instances of Streamer and provided that the same query is used clustered results could be returned instantly. The storage layer could also hold raw or parsed tweets but basic check to ensure that the data is relatively new are required. Caching would make more sense due to the fact that real time messages are expected.

We hope to explore some of these improvements ideas in the future.

Bibliography

[1] **George H. Chen**, **Stanislav Nikolov** and **Devavrat Shah**, *A Latent Source Model for Nonparametric Time Series Classification* [1]

[2] **Mehran Sahami** and **Timothy D. Heilman**, *A Web-based Kernel Function for Measuring the Similarity of Short Text Snippets* [2]

[3] **Charu C. Aggarwal** and **ChengXiang Zhai**, *A survey of text clustering algorithms* [3]

[4] **Brendan O'Connor, Michel Krieger, David Ahn**, *TweetMotif: Exploratory Search and Topic Summarization for Twitter* [4]

[5] **Kevin Dela Rosa, Rushin Shah**, *Topical Clustering of Tweets* [5]

[6] **Moritz Sudhof**, *Politics, Twitter, and information discovery* [6]

[7] **Manning, Christopher D., Surdeanu, Mihai, Bauer, John, Finkel, Jenny, Bethard, Steven J., and McClosky, David**. 2014. *The Stanford CoreNLP Natural Language Processing Toolkit* [7]

[8] **Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze**, *Introduction to Information Retrieval* [8]

[1] https://cdn2.hubspot.net/hubfs/489432/docs/A_Latent_Source_Model_for_Nonparametric_Time_Series_Classification.pdf

[2] http://wwwconference.org/www2006/programme/files/pdf/3069.pdf

[3] http://www.charuaggarwal.net/text-cluster.pdf

[4] http://brenocon.com/oconnor_krieger_ahn.icwsm2010.tweetmotif.pdf

[5] http://www.cs.cmu.edu/ kdelaros/sigir-swsm-2011.pdf

[6] http://web.stanford.edu/group/journal/cgi-bin/wordpress/wp-content/uploads/2012/09/Sudhof_Eng_-2012.pdf

[7] http://nlp.stanford.edu/pubs/StanfordCoreNlp2014.pdf

[8] http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html